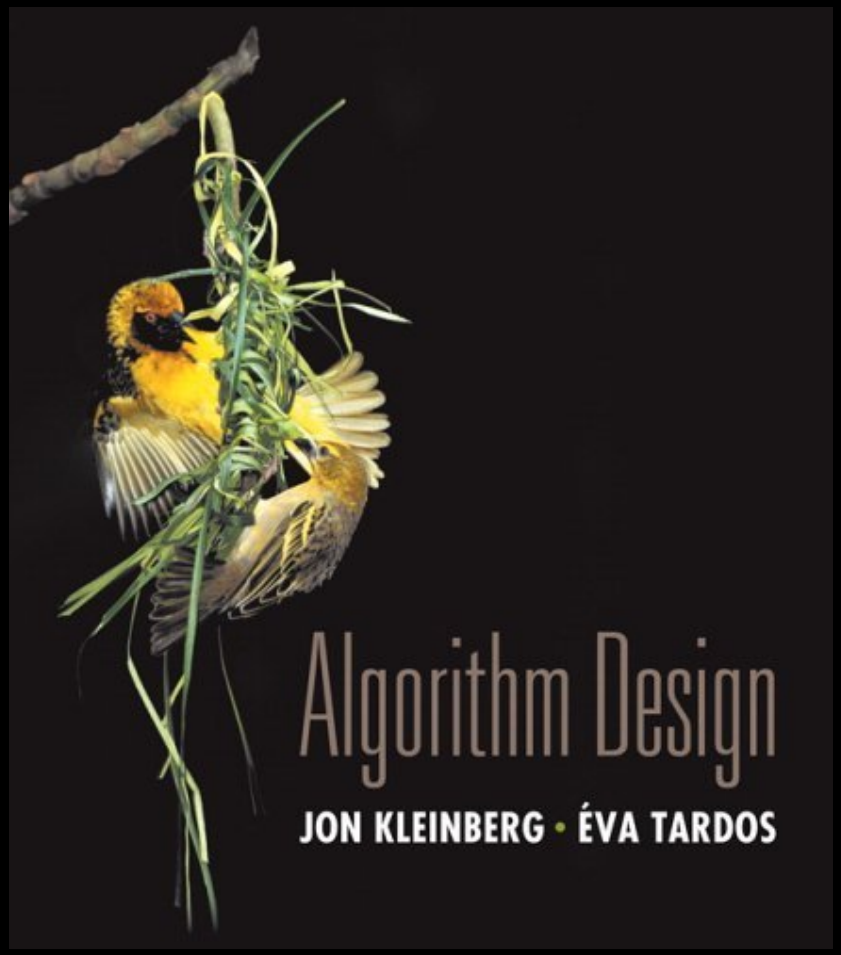


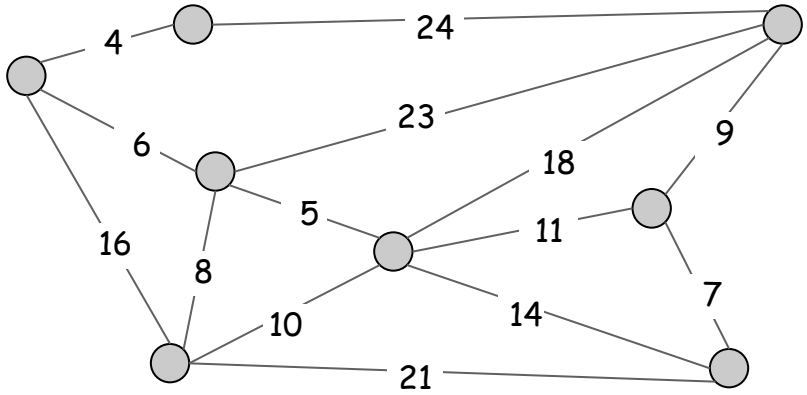
Chapter 4

Greedy Algorithms

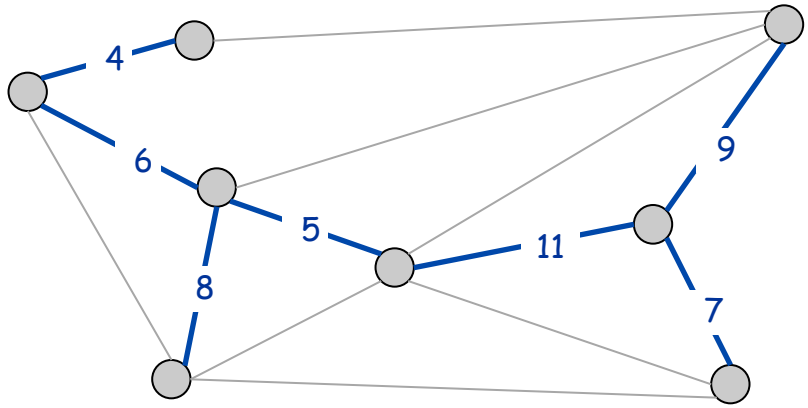
4.5 Minimum Spanning Tree

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 $T = \phi$. 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.

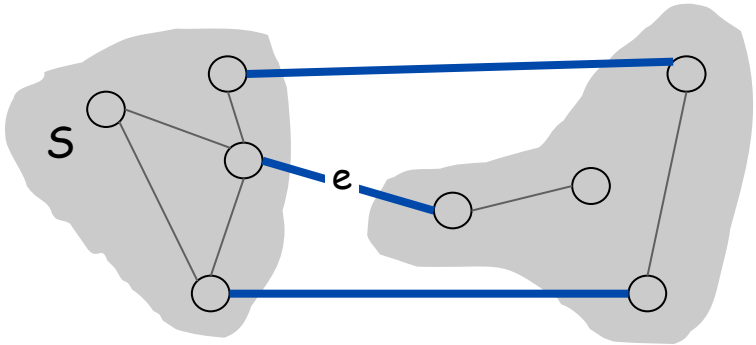
Remark. All three algorithms produce an MST.

Greedy Algorithms

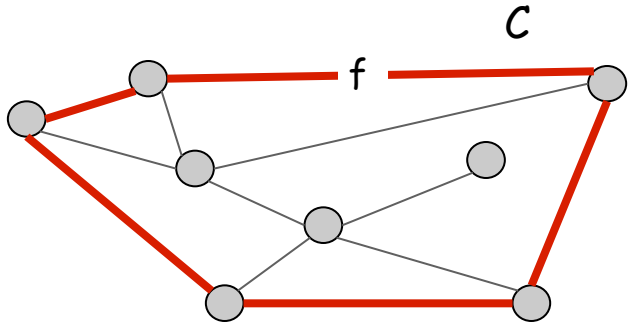
Simplifying assumption. All edge costs c_e are distinct.
Also, the graph is connected.

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.



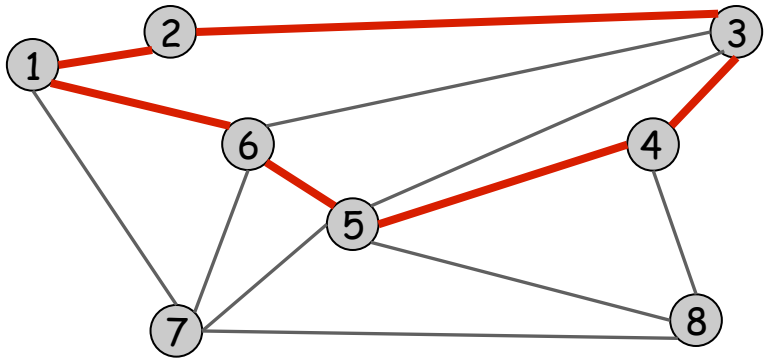
e is in the MST



f is not in the MST

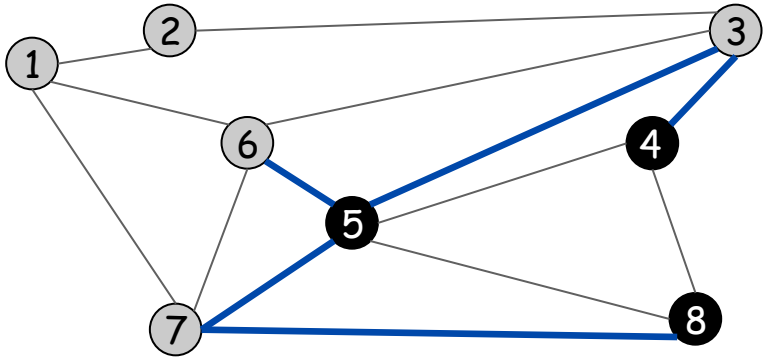
Cycles and Cuts

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

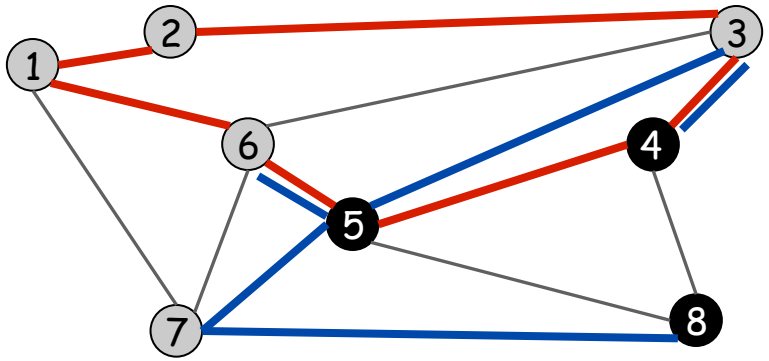
Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



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

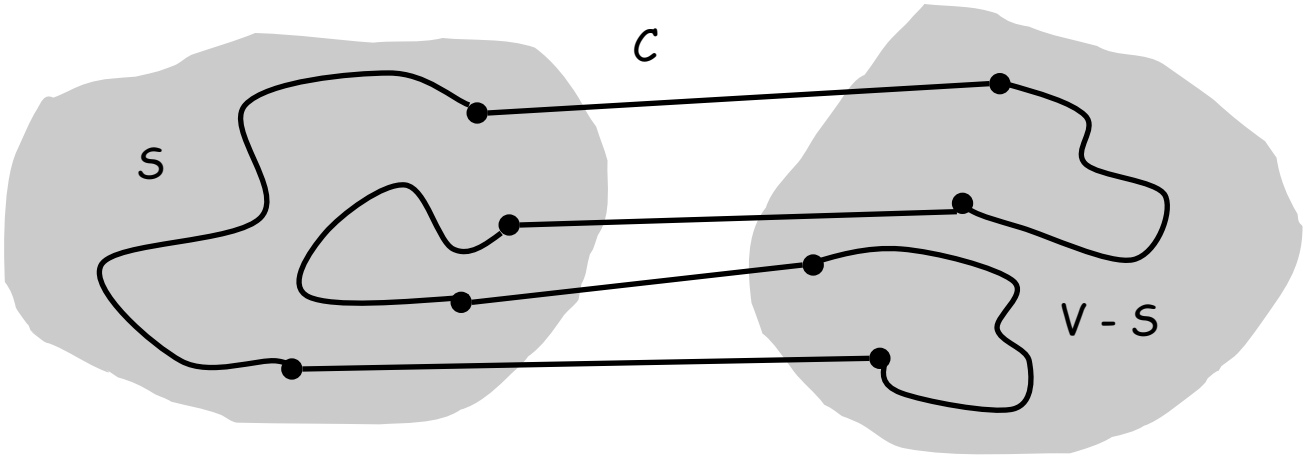
Cycle-Cut Intersection

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)

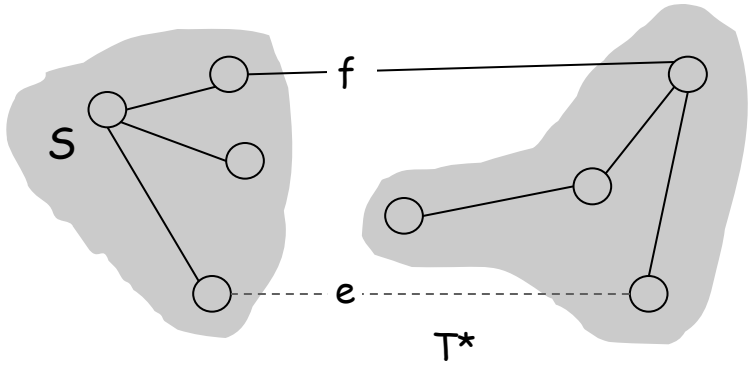


Simplifying assumption. All edge costs c_e are distinct.

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. ▪



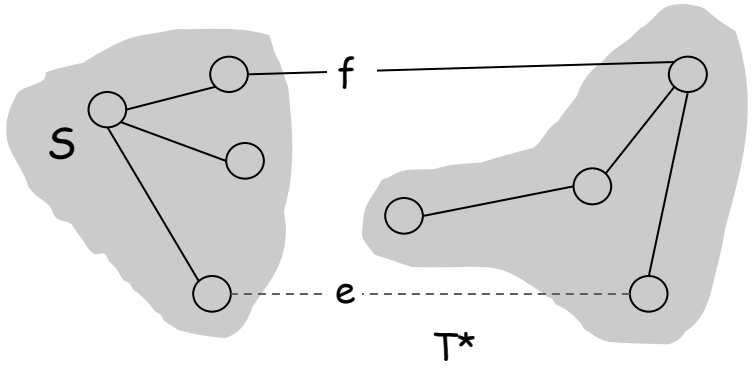
Greedy Algorithms

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



Greedy Algorithms

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.

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

Prim's Algorithm

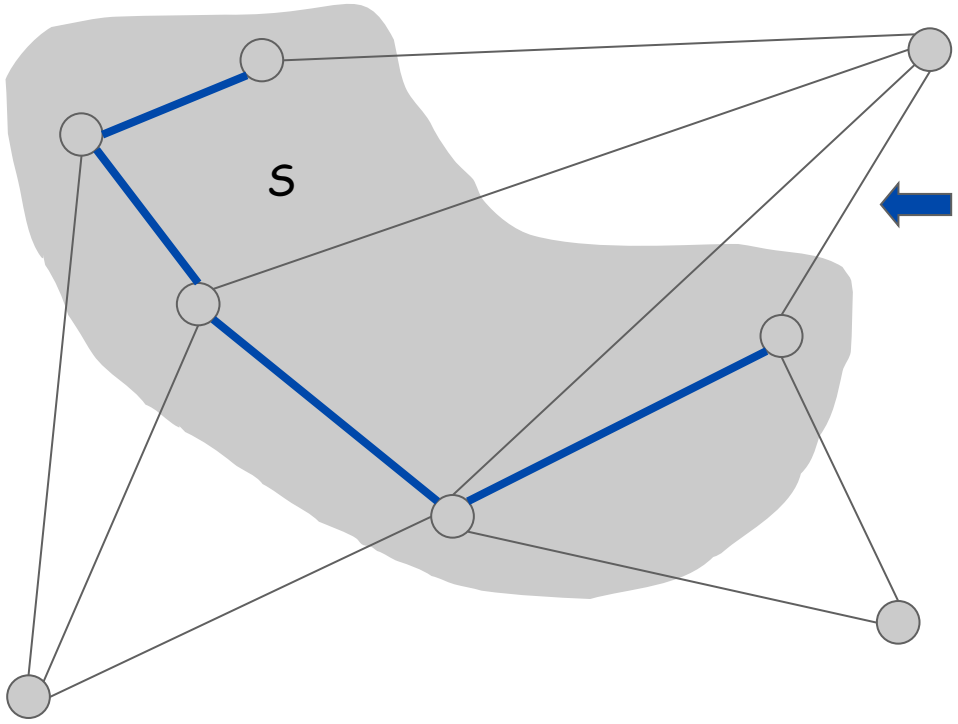
Initialize X = {s} [vertex s can be chosen arbitrarily]

T (edges in the tree) initialized to empty

While not all vertices spanned

- Let e = (u,v) be the cheapest edge of G such that u in X and v in V-X
- Add e to T
- Add v to X

At each step, increasing the number of spanned vertices in cheapest way possible.



Correctness of Prim

Theorem: Prim's algorithm always computes an MST

Proof:

1. Every edge in T* is in MST by the cut property so
   -- no cycles are created
   -- edges are a subset of the MST.

2. But the set of edges spans the graph: can't get stuck until X = V. Otherwise the cut (X, V-X) has no edges crossing it and the graph is disconnected.

Fast Implementation of Prim's Algorithm

Initialize X = {s} [vertex s can be chosen arbitrarily]

T initialized to empty

While not all vertices spanned

- Let e = (u,v) be the cheapest edge of G such that u in X and v in V-X
- Add e to T
- Add v to X

Straightforward implementation:

- O(n) iterations [n = # of vertices]
- O(m) time per iteration [m = # edges]

- Overall O(mn)

- Can get it down to O(m log n) using priority queue

Prim's algorithm with priority queue (similar to Dijkstra)

Invariant # 1: Elements in priority queue = vertices of V-X

Invariant # 2: For v in V-X, set key[v] = cheapest edge (u,v) connecting v to X. (infinity if there is no edge)

Can initialize the PQ with $O(m + n \log n) = O(m \log n)$ preprocessing time

- $O(m)$ to figure out the initial key values
- $O(n-1)$ inserts.

Repeatedly use Delete-Min to get the vertex outside X with cheapest connection cost.

Work to maintain invariant #2?

Prim's algorithm with priority queue (similar to Dijkstra)

Invariant # 2: For v in V-X, set key[v] = cheapest edge (u,v) connecting v to X.

To maintain this invariant after DeleteMin, may need to recompute some keys.

When v added to X:

- For each edge (v,w) in E
  - If w in V-X then
    Delete w from PQ
    Recompute key[w] := min{ key[w], c_{vw}}
    Re-insert w into PQ

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 key[v] = cost of cheapest edge v to a node in S.
- O(m log n) with a binary heap implementation.

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

    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 < key[v]))
                decrease priority key[v] to c_e
}
```

Greedy Algorithms

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.

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

Kruskal's Algorithm (and proof of correctness)

Kruskal's algorithm. [Kruskal, 1956]

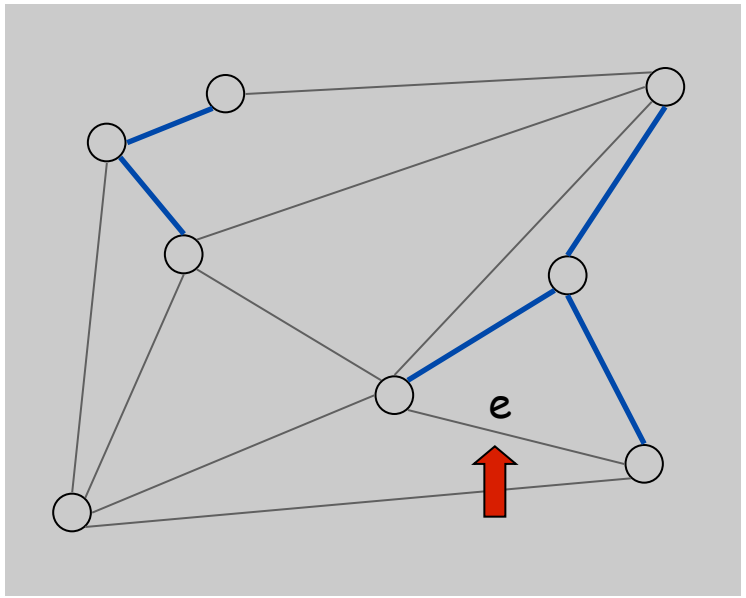
Consider edges in ascending order of weight. (i.e. sort first)

Case 1: If adding e to T creates a cycle, discard e.

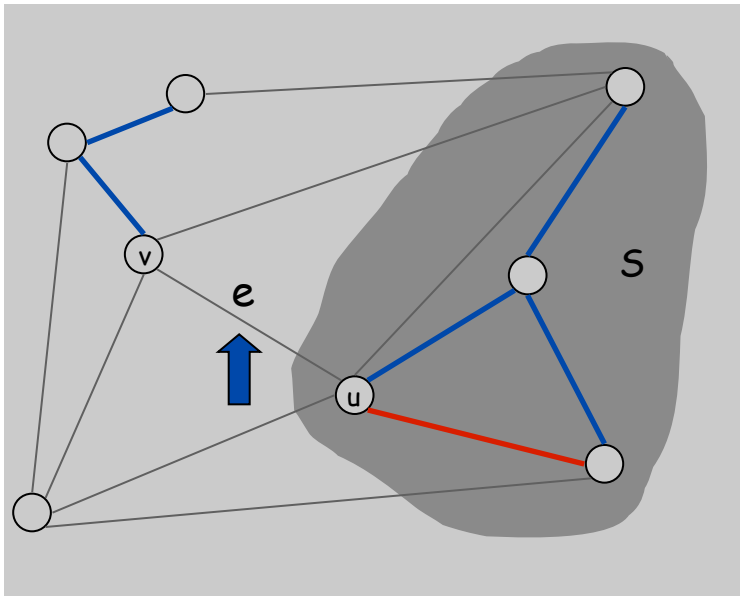
cycle property

Case 2: Otherwise, insert e = (u, v) into T.

cut property where S is set of nodes in u's component.



Case 1



Case 2

Kruskal's Algorithm: Naive Implementation

Kruskal's algorithm. [Kruskal, 1956]

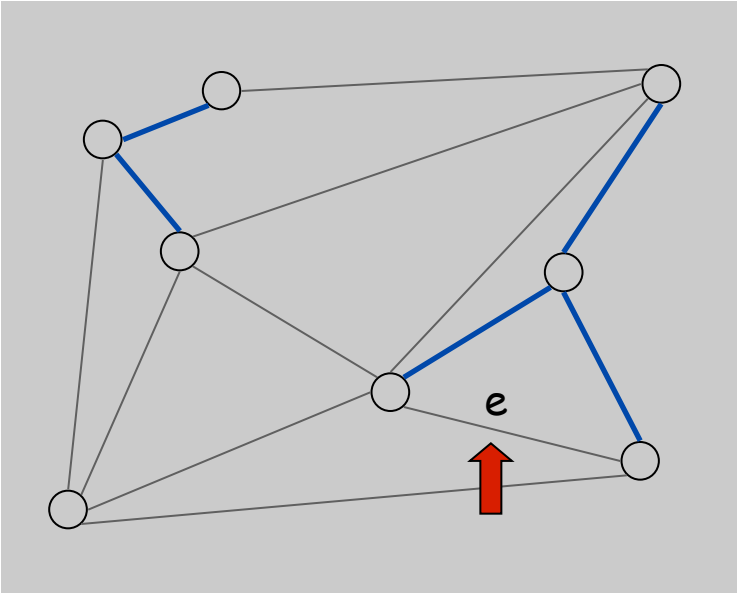
Consider edges in ascending order of weight. O(m log n)

Case 1: If adding e to T creates a cycle, discard e.

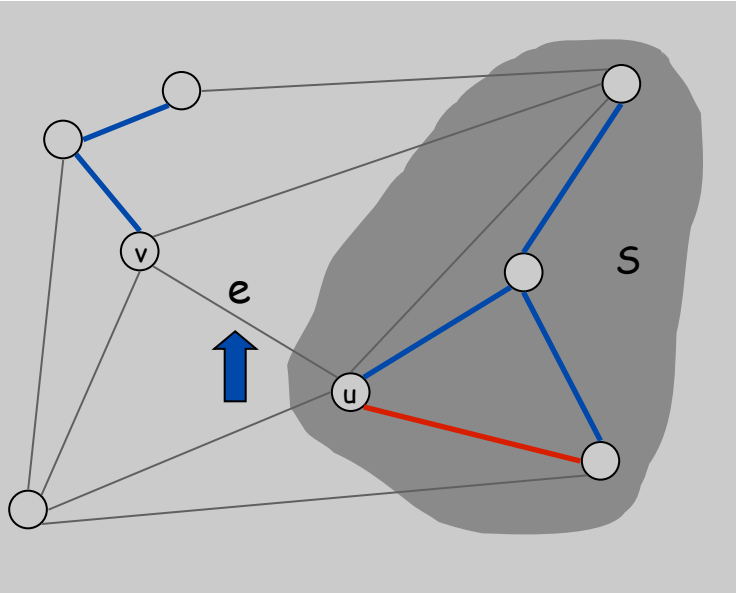
O(n) (e.g. DFS or BFS)

Case 2: Otherwise, insert e = (u, v) into T.

O(1)



Case 1



Case 2

Kruskal's Algorithm: Better implementation

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight. **O(m log n)**
  - Case 1: If adding e to T creates a cycle, discard e.
  - Case 2: Otherwise, insert e = (u, v) into T.

Use Union-Find data structure.

Maintains partition of a set of objects:

Find(X): returns name of the group that X belongs to

Union (C_i, C_j): Fuses groups C_i and C_j into a single group

To use in Kruskal's algorithm:

Objects = vertices

Groups = connected components of edges in T

Adding new edge (u,v) to T == fusing connected components of u,v.

Union- Find Basics

Union-Find data structure: maintains partition of a set of objects

Find(X): returns name of the group that X belongs to

Union (C_i, C_j): Fuses groups C_i and C_j into a single group

To use in Kruskal's algorithm:

Objects = vertices

Groups = connected components of edges in T

Adding new edge (u,v) to T == fusing connected components of u,v

Maintain one linked structure per component (group) of (V, T).

Each component has arbitrary leader vertex

Invariant: Each vertex points to the leader of its component.

To check if u and v are in same component: check if Find(u) = Find(v).

$O(1)$ time.

Implementation using Union-Find

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight. **O(m log n)**
  - Case 1: If adding e to T creates a cycle, discard e.
  - Case 2: Otherwise, insert e = (u, v) into T.

Maintain one linked structure per component (group) of (V, T).

Each component has arbitrary leader vertex

Invariant: Each vertex points to the leader of its component.

When new edge (u,v) added to T, connected components of u and v merge.

How many leader pointer updates needed to maintain invariant in worst case?

Theta (n) (e.g., when merging two components of size n/2 each)

Implementation using Union-Find

Maintain one linked structure per component (group) of (V, T).

Each component has arbitrary leader vertex

Invariant: Each vertex points to the leader of its component.

When new edge (u,v) added to T, connected components of u and v merge.

Idea #2: when two components merge, have smaller one inherit the leader of the larger one. (keep track of size of component to implement).

How many leader pointer updates needed to maintain invariant in worst case?

Theta (n) (e.g., when merging two components of size n/2 each)

Implementation using Union-Find

Maintain one linked structure per component (group) of (V, T).

Each component has arbitrary leader vertex

Invariant: Each vertex points to the leader of its component.

Idea #2: when two components merge, have smaller one inherit the leader of the larger one. (keep track of size of component to implement).

How many times does a single vertex v have its leader pointer updated over the course of the algorithm?

Theta (log n)

Reason: Every time v's leader pointer gets updated, size of its component at least doubles. Can only happen log n times.

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 $O(m \log n)$ for union-find.

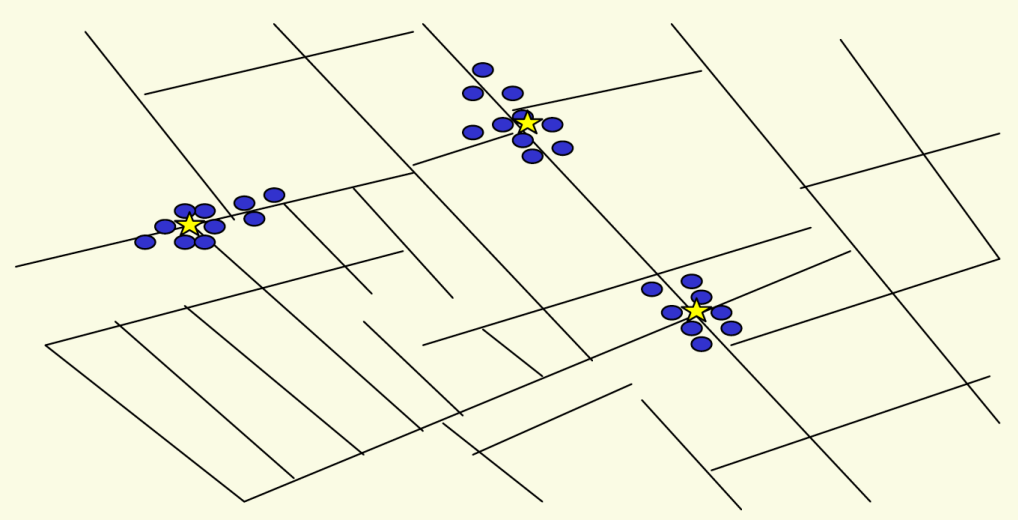
$m \le n^2 \Rightarrow \log m$ is $O(\log n)$

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← φ

   foreach (u ∈ 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 ← T ∪ {e_i}
         merge the sets containing u and v
      }                                ↖
   return T                            merge two components
}
```

4.7 Clustering



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

Clustering

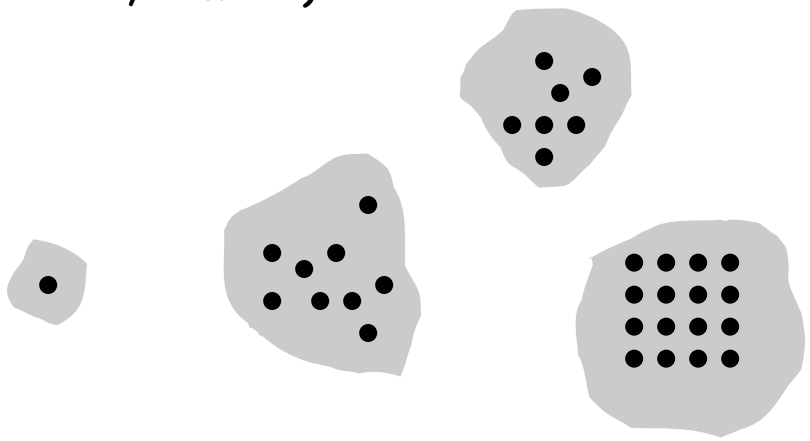
Clustering [aka "unsupervised learning"] . Given a set U of n objects labeled p_1, …, p_n, classify into coherent groups.

photos, documents. micro-organisms

Goal: objects in same cluster similar; objects in different clusters dissimilar.

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

Fundamental problem. Divide into clusters so that points in different clusters are far apart (i.e., clusters themselves represent points that are close, similar)



Clustering of Maximum Spacing

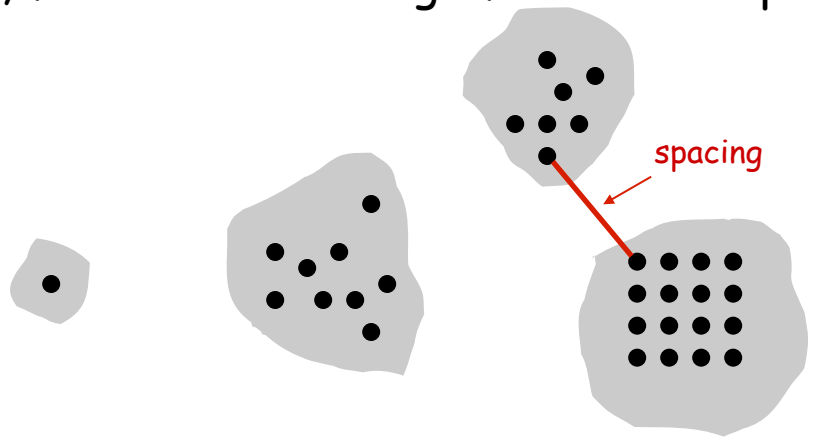
k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$
- $d(p_i, p_j) \geq 0$ (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, and distance function, 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. Single-link clustering finds the max-spacing k-clustering.

Proof: Let $C^*_1, \ldots, C^*_k$ be the greedy clustering; suppose that the spacing is S. Let $C_1, \ldots, C_k$ be some other (different) clustering. Need to show that spacing of $C_1, \ldots, C_k$ at most S.

If not the same, then there is a pair of points p_i, p_j in the same greedy cluster (say C^*_r), but in different clusters in $C_1, \ldots, C_k$, say C_s and C_t.

- Since they are in the same cluster in C^*, there is a path in the (partial) MST between them in C^* and every edge on this path has length at most S (since edges are added in order of distance).
- Some edge (p, q) on p_i-p_j path in C^*_r crosses two different clusters in C.
- Then the spacing between these is ≤ S,
- So max spacing of C is ≤ S ▪