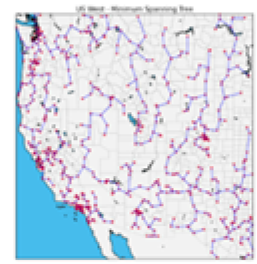
Lecture13



CSE 417
Algorithms and Complexity

Autumn 2023
Lecture 13
Minimum Spanning Trees

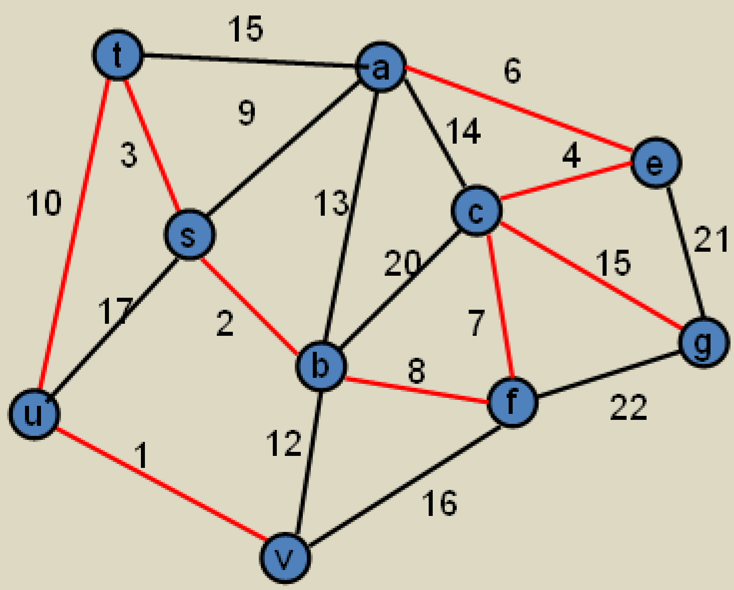
10/24/2023 CSE 417 1

Announcements

- Midterm, Monday, October 30

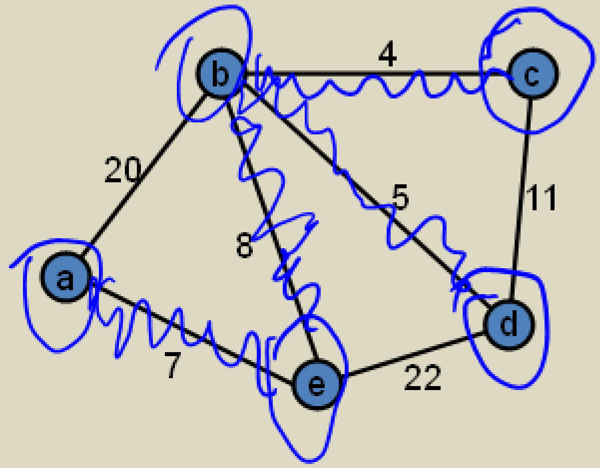
- Topics: Material Presented in Lecture
  - Stable Matching
  - Graphs and simple graph algorithms
    - Breadth First Search
    - Topological Sort
  - Greedy Algorithms
    - Interval Scheduling Problems
    - Graph Coloring
  - Shortest Paths Algorithms
  - Minimum Spanning Tree Algorithms

Minimum Spanning Tree



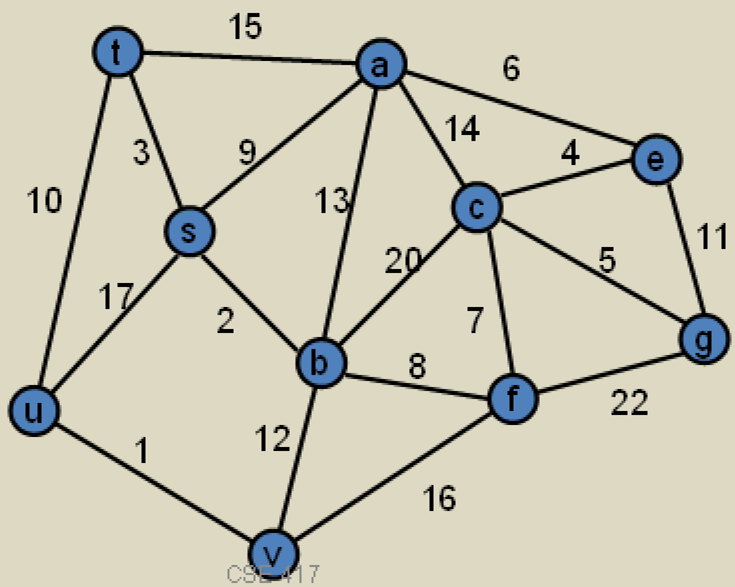
Greedy Algorithms for Minimum Spanning Tree

- Prim's Algorithm: Extend a tree by including the cheapest out going edge
- Kruskal's Algorithm: Add the cheapest edge that joins disjoint components



Greedy Algorithm 1
Prim's Algorithm

- Extend a tree by including the cheapest out going edge

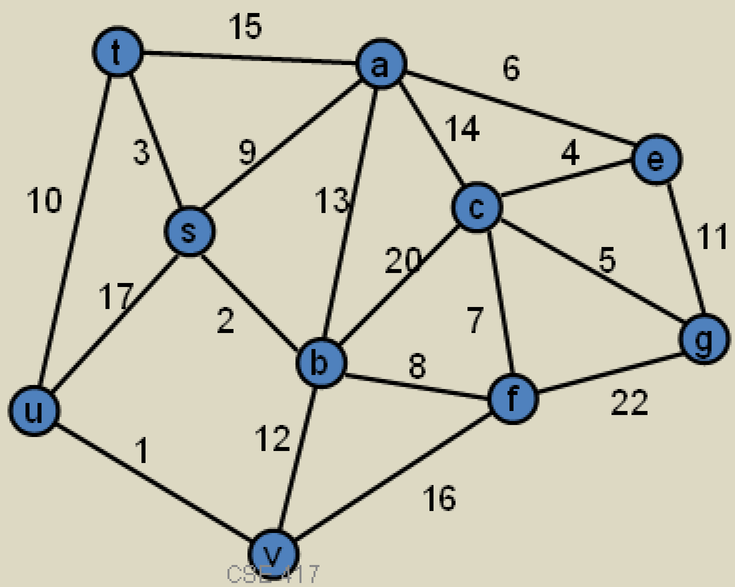


Construct the MST with Prim's algorithm starting from vertex a

Label the edges in order of insertion

Greedy Algorithm 2
Kruskal's Algorithm

- Add the cheapest edge that joins disjoint components



Construct the MST with Kruskal's algorithm

Label the edges in order of insertion

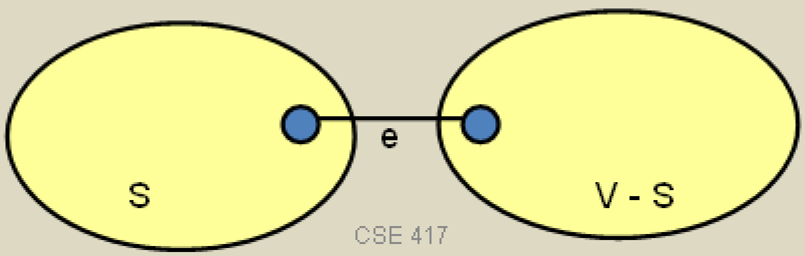
Why do the greedy algorithms work?

- For simplicity, assume all edge costs are distinct

Show smallest edge e=(u,v) in graph is in every MST

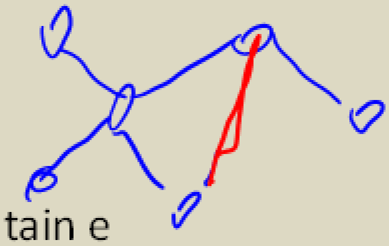
Edge inclusion lemma

- Let S be a subset of V, and suppose e = (u, v) is the minimum cost edge of E, with u in S and v in V-S
- e is in every minimum spanning tree of G
  - Or equivalently, if e is not in T, then T is not a minimum spanning tree

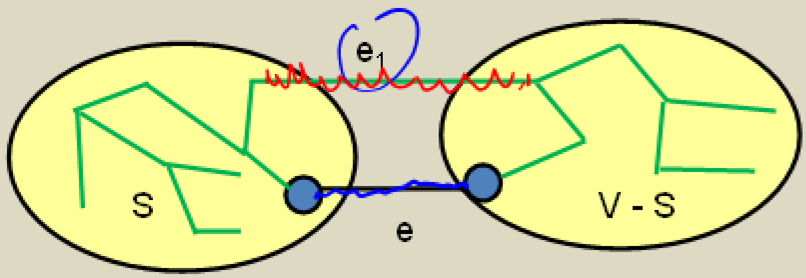


e is the minimum cost edge between S and V-S

Proof



- Suppose T is a spanning tree that does not contain e
- Add e to T, this creates a cycle
- The cycle must have some edge $e_1 = (u_1, v_1)$ with u_1 in S and v_1 in V-S



- $T_1 = T - \{e_1\} + \{e\}$ is a spanning tree with lower cost
- Hence, T is not a minimum spanning tree

Optimality Proofs

- Prim's Algorithm computes a MST
- Kruskal's Algorithm computes a MST

- Show that when an edge is added to the MST by Prim or Kruskal, the edge is the minimum cost edge between S and V-S for some set S.

Prim's Algorithm

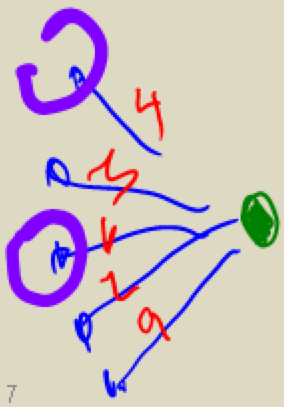
S = { a }; T = { };

while S != V

choose the minimum cost edge
e = (u,v), with u in S, and v in V-S

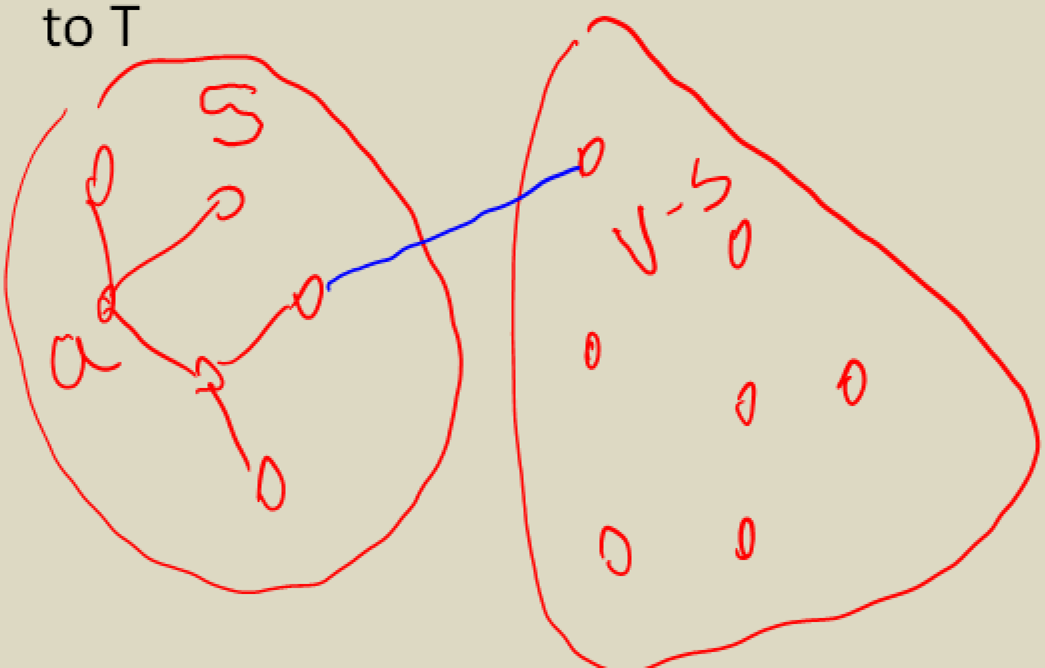
add e to T

add v to S

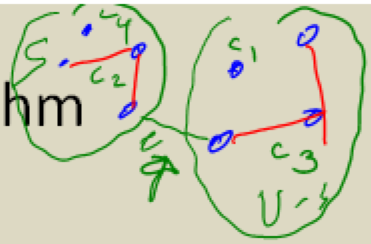


Prove Prim's algorithm computes an MST

- Show an edge e is in the MST when it is added to T



Kruskal's Algorithm



$m \leq n^2$

Let C = {{v_1}, {v_2}, . . ., {v_n}}; T = { }

while |C| > 1

Let e = (u, v) with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C

Replace C_i and C_j by C_i U C_j

Add e to T

sorting edges $e_1, e_2, \ldots, e_m$ in increasing cost.

$O(m \log m) = O(m \log n)$

for i = 1 to m
if e_i joins distinct components
merge components joined by e

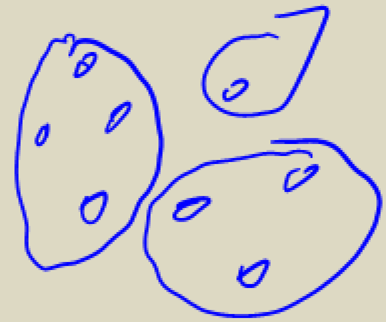
Prove Kruskal's algorithm computes an MST

- Show an edge e is in the MST when it is added to T

MST Implementation and runtime

- Prim's Algorithm
  - Implementation, runtime: just like Dijkstra's algorithm
  - Use a heap, runtime O(m log n)
- Kruskal's Algorithm
  - Sorting edges by cost: O(m log n)
  - Managing connected components uses the Union-Find data structure
    - Amazing, pointer based data structure
    - Very interesting mathematical result

Disjoint Set ADT



- Data: set of pairwise **disjoint sets**.
- Required operations
  - **Union** – merge two sets to create their union
  - **Find** – determine which set an item appears in
- Check Find(v) ≠ Find(w) to determine if (v,w) joins separate components
- Do Union(v,w) to merge sets

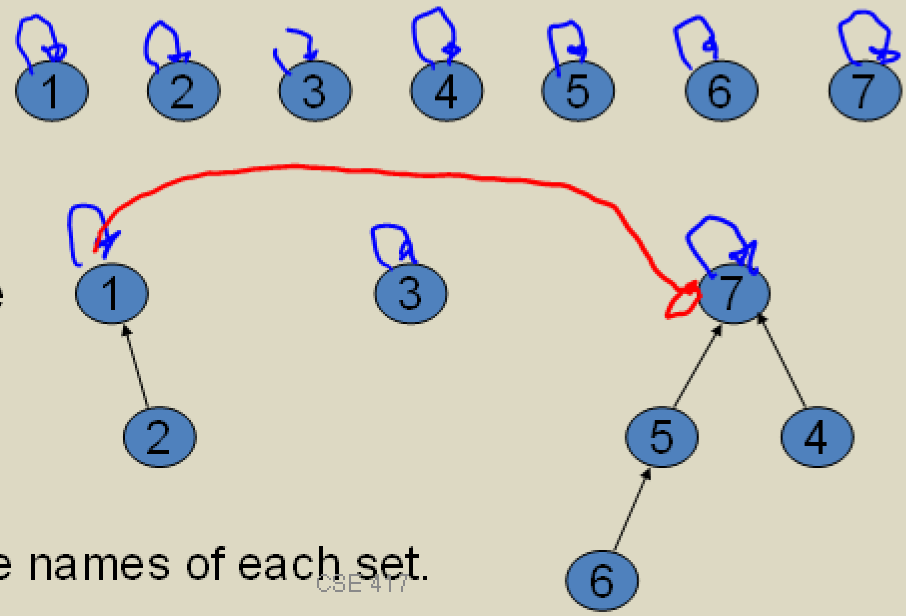
Up-Tree for DS Union/Find

Observation: we will only traverse these trees upward from any given node to find the root.

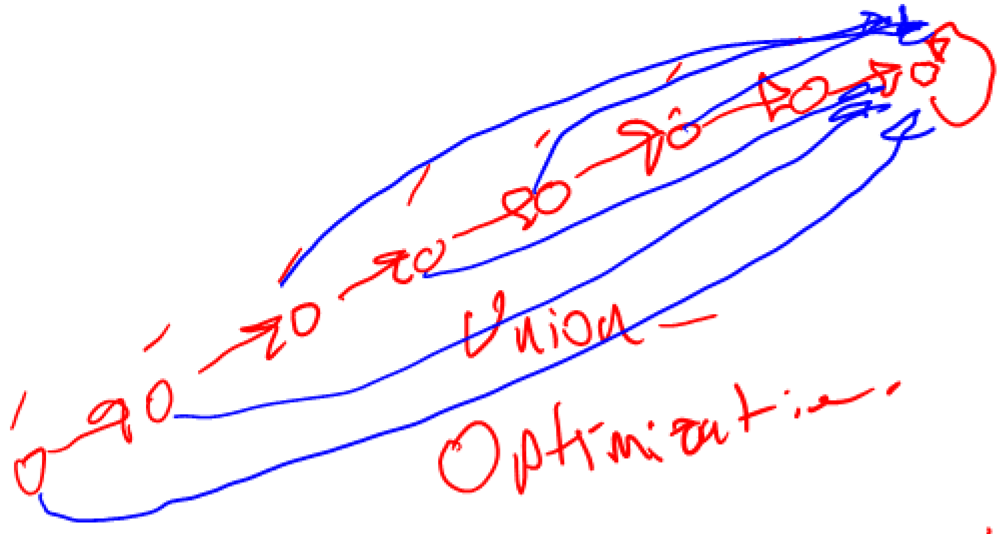
Idea: *reverse* the pointers (make them point up from child to parent). The result is an **up-tree**.

Initial state

Intermediate state



Roots are the names of each set.



a. Union by size $O(\log n)$

path compression $O(\log n)$

size + compression $O(\alpha(n))$