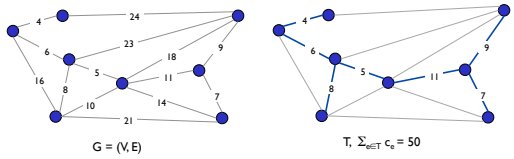
Objectives

- Wrap Up Minimum Spanning Tree
- Union-Find data structure
- Clustering

Review: Minimum Spanning Tree

- Spanning tree: spans all nodes in graph
- Given a connected graph G = (V, E) with positive 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*



What were the three algorithms we proposed?

Review

- What is a minimum spanning tree?
- What are three greedy solutions to finding the minimal spanning tree?

Review: Greedy Algorithms

All three algorithms produce a MST

- 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*.
  - Similar to Dijkstra's (but simpler)
- 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*.

What do these algorithms have/do/check in common?

What Do These Algorithms Have in Common?

- When is it safe to include an edge in the minimum spanning tree?

Cut Property

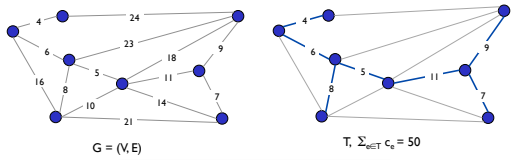
- When is it safe to eliminate an edge from the minimum spanning tree?

Cycle Property

More on this in a bit…

Review: Minimum Spanning Tree

- Spanning tree: spans all nodes in graph
- Given a connected graph G = (V, E) with positive 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*



Why must the solution be a tree?

Minimum Spanning Tree

- Assume have a minimal solution that is not a tree, i.e., it has a cycle
- What could we do?
  - What do we know about the edges?
  - How does that change the cost of the solution?

Minimum Spanning Tree

- Proof by Contradiction.
- Assume have a minimal solution V that is not a tree, i.e., it has a cycle
  - Contains edges to all nodes because, to be a solution, solution must be connected (spanning)
- Remove an edge from the cycle
  - Can still reach all nodes (could go "long way around")
  - **But** at lower total cost
  - Contradiction to our minimal solution

What Do These Algorithms Have in Common?

- When is it safe to include an edge in the minimum spanning tree?

  Cut Property

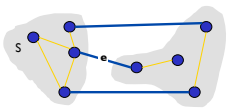
- When is it safe to eliminate an edge from the minimum spanning tree?

  Cycle Property

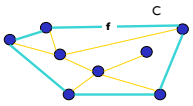
More on this in a bit…

Cut and Cycle Properties

- Simplifying assumption: All edge costs c_e are distinct
  - ➜ MST is unique
- Cut property. Let *S* be any subset of nodes, and let *e* be the min cost edge with exactly one endpoint in *S*. Then MST contains *e*.
- Cycle property. Let *C* be any cycle, and let *f* be the max cost edge belonging to *C*. Then MST does *not* contain *f*.



Cut Property: e is in MST

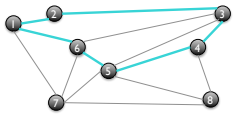


Cycle Property: f is **not** in MST

Let's try to prove these …

Cycles and Cuts

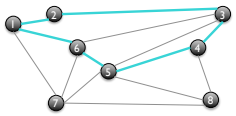
- Cycle. Set of edges in the form a-b, b-c, c-d, …, y-z, z-a



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

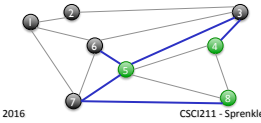
Cycles and Cuts

- Cycle. Set of edges in 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

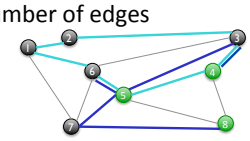
- Cutset. A *cut* is a subset of nodes *S*. The corresponding *cutset D* is the subset of edges with *exactly one* endpoint in *S*.



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

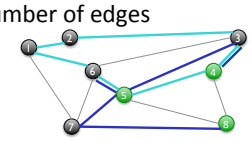


Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cut S = { 4, 5, 8 }
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

What are the possibilities for the cycle?

Cycle-Cut Intersection

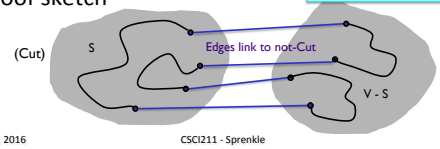
- Claim. A *cycle* and a *cutset* intersect in an even number of edges



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cut S = { 4, 5, 8 }
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

1. Cycle all in S
2. Cycle not in S
3. Cycle has to go from S→V-S *and* back

- Proof sketch



Proving Cut Property: OK to Include Edge

- 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.?

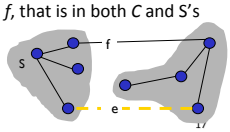
Proving Cut Property: OK to Include Edge

- 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 there is an MST T* that does not contain *e*
    - What do we know about T, by defn?
    - What do we know about the nodes *e* connects?

Proving Cut Property: OK to Include Edge

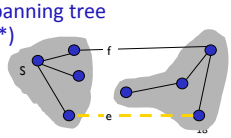
- Cut property. Let *S* be any subset of nodes, and let *e* be the min cost edge with exactly one endpoint in *S*. Then the MST T* contains *e*.
- Pf. (exchange argument)
  - Suppose there is an MST T* that does not contain *e*
  - Adding *e* to T* creates a cycle *C* in T*
  - Edge *e* is in cycle *C* and in cutset corresponding to *S*
    ⇒there exists another edge, say *f*, that is in both *C* and S's cutset

Which means?



Proving Cut Property: OK to Include Edge

- 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 there is an MST T* that does not contain *e*
  - Adding *e* to T* creates a cycle *C* in T*
  - Edge *e* is in cycle *C* and in cutset corresponding to *S*
    ⇒there exists another edge, say *f*, that is in both *C* and S's cutset
  - T' = T* ∪ { e } - { f } is also a spanning tree
  - Since $c_e < c_f$, cost(T') < cost(T*)
  - This is a contradiction. ▪



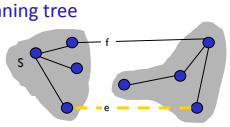
Proving Cycle Property: OK to Remove Edge

- 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*.

Ideas about approach?

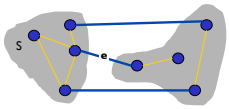
Cycle Property: OK to Remove Edge

- 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*
  - Deleting *f* from T* creates a cut *S* in T*
  - Edge *f* is both in the cycle *C* and in the cutset *S*
    - ⇒ there exists another edge, say *e*, that is in both *C* and S
  - T' = T* ∪ { *e* } - { *f* } is also a spanning tree
  - Since $c_e < c_f$, cost(T') < cost(T*)
  - This is a contradiction. ▪

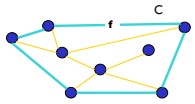


Summary of What Just Proved

- Simplifying assumption: All edge costs c_e are distinct
  - ➜ MST is unique
- Cut property. Let *S* be any subset of nodes, and let *e* be the min cost edge with exactly one endpoint in *S*. Then MST contains *e*.
- Cycle property. Let *C* be any cycle, and let *f* be the max cost edge belonging to *C*. Then MST does not contain *f*.



Cut Property: e is in MST



Cycle Property: f is **not** in MST

Prim's Algorithm

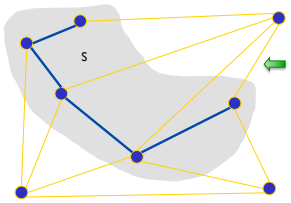
[Jarník 1930, Dijkstra 1957, Prim 1959]

- 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.

How can we prove its correctness?

Prim's Algorithm: Proof of Correctness

- Initialize S to be any node
- Apply ***cut property*** to S
  - Add min cost edge (v, u) in *cutset* corresponding to S, and add one new explored node u to S



Ideas about implementation?

Implementation: Prim's Algorithm

Similar to Dijkstra's algorithm

- 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*

Running Time?

```
foreach (v ∈ V) a[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 < a[v]))
         decrease priority a[v] to c_e
```

Implementation: Prim's Algorithm

Similar to Dijkstra's algorithm

- 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(m log n) with a heap

```
foreach (v ∈ V) a[v] = ∞                          O(n)
Initialize an empty priority queue Q
foreach (v ∈ V) insert v onto Q                   O(n logn)
Initialize set of explored nodes S = ϕ
while (Q is not empty)                            O(n)
   u = delete min element from Q                  O(log n)
   S = S ∪ { u }
   foreach (edge e = (u, v) incident to u)        O(deg(u))
      if ((v ∉ S) and (c_e < a[v]))
         decrease priority a[v] to c_e            O(log n)
```


Looking ahead

- PS 5 due Friday