Objectives

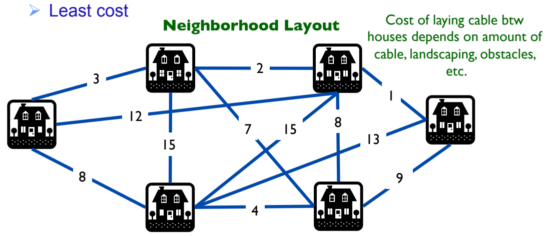
- Minimum Spanning Tree
- Union-Find data structure
- Clustering

Announcements, Discussion

- Wiki readings
  - Low risk, high reward assignments
  - Helpful feedback
  - Process: follow book closely
  - Wed: Chap 3.6, 4, 4.1, 4.2, 4.4,
- Jeopardy! Challenge
  - Today– Wednesday
  - 7:30 on CBS
  - Answer questions on Sakai forum for 5 pts towards your problem set grade

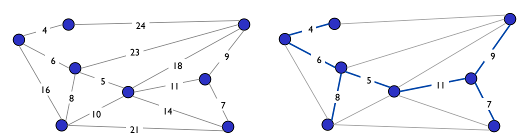
Review: Laying Cable

- Comcast knows how to make money and how to save money
- They want to lay cable in a neighborhood
  - Reach all houses
  - Least cost



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*



G = (V, E) T, $\Sigma_{e \in T} c_e = 50$

Review: Greedy Algorithms

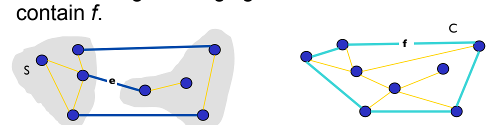
All three algorithms produce a MST

- 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?

Review: Important Properties

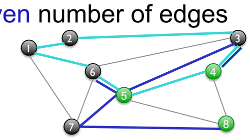
- Simplifying assumption: All edge costs c_e are distinct
  - ➜ MST is unique
- Cut property. Let *S* be any subset of nodes, and let *e* be the min cost edge with exactly one endpoint in *S*. Then MST contains *e*.
- Cycle property. Let *C* be any cycle, and let *f* be the max cost edge belonging to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



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

Review: Cycle-Cut Intersection

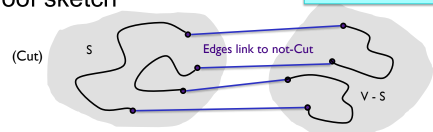
- Claim. A *cycle* and a *cutset* intersect in an even number of edges



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.?

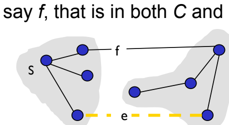
Proving Cut Property: OK to Include Edge

- 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

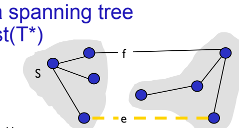
- Cut property. Let *S* be any subset of nodes, and let *e* be the min cost edge with exactly one endpoint in *S*. Then the MST T* contains *e*.
- Pf. (exchange argument)
  - Suppose there is an MST T* that does not contain *e*
  - Adding *e* to T* creates a cycle *C* in T*
  - Edge *e* is in cycle *C*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



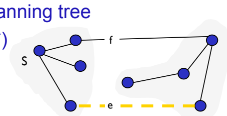
Proving Cycle Property: OK to Remove Edge

- 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?

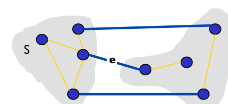
Cycle Property: OK to Remove Edge

- 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. ▪

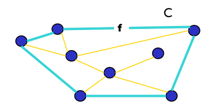


Summary of What Just Proved

- 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

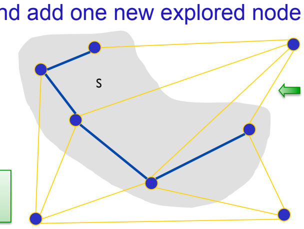
[Jarník 1930, Dijkstra 1957, Prim 1959]

- 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 (ce < a[v]))
           decrease priority a[v] to ce
```


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)
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 (ce < a[v]))
           decrease priority a[v] to ce         O(log n)
```

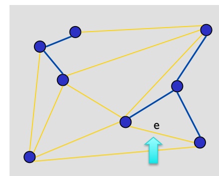
Kruskal's Algorithm [1956]

- Start with T = ϕ
- Consider edges in *ascending order of cost*
- Insert edge *e* in T *unless doing so would create a cycle*
  - Add edge as long as "compatible"

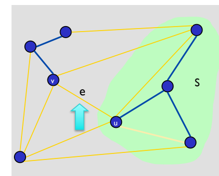
How can we prove algorithm's correctness?

Kruskal's Algorithm: Proof of Correctness

- Consider edges in ascending order of weight
- Case 1: If adding *e* to T creates a cycle, discard *e* according to ***cycle property*** (*e* must be max weight)
- 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

Implementing Kruskal's Algorithm

What is tricky about implementing Kruskal's algorithm?

How do we know when adding an edge will create a cycle?
- What are the properties of a graph/its nodes when adding an edge will create a cycle?

UNION-FIND DATA STRUCTURE

Union-Find Data Structure

- Keeps track of a graph as edges are added
  - Cannot handle when edges are deleted
- Maintains disjoint sets
  - E.g., graph's connected components
- Operations:
  - `Find(u)`: returns name of set containing u
    - How utilized to see if two nodes are in the same set?
    - Goal implementation: **O(log n)**
  - `Union(A, B)`: merge sets A and B into one set
    - Goal implementation: **O(log n)**

Best darn U-F Data Structure

Implementing Kruskal's Algorithm

- Using the **union-find** data structure
  - Build set T of edges in the MST
  - Maintain set for each connected component

Costs?

```
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?

merge two components

Implementing Kruskal's Algorithm

- Using best implementation of union-find
  - Sorting: O(m log n) ← $m \le n^2 \Rightarrow \log m$ is O(log n)
  - Union-find: $O(m\ \underbrace{\alpha\ (m, n)}_{\text{essentially a constant}})$
  - O(m log n)

```
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?

merge two components

Limitations to Applying MST?

- Motivating Example: Comcast laying cable

Neighborhood Layout