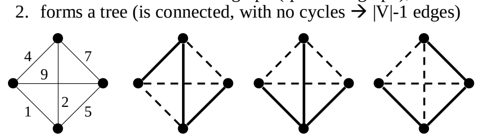
Lecture 25: Kruskal and beyond…

- <u>What we will munch on today:</u>
  - Minimum Spanning Trees
    - Prim's Algorithm
    - Kruskal's Algorithm
  - Those Puzzles from 4^{th} grade!
  - Euler Circuits and Tours
- Covered in Chapter 9 in the textbook

Recall from Last Time: Spanning Trees

- *Spanning tree*: subset of edges from a connected graph G = (*V*,*E*) that:
  1. touches all vertices in the graph (*spans* the graph), and
  2. forms a tree (is connected, with no cycles → |V|-1 edges)



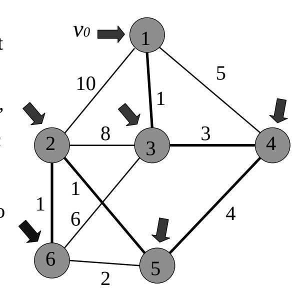
- *Minimum spanning tree (MST)*: spanning tree with the least total edge cost

Why greed works for finding MSTs…

- For any spanning tree T, inserting an edge e not in T creates a cycle → Removing any edge gives back a spanning tree
  - If e had a lower cost than removed edge, we get a lower cost spanning tree
- <u>Idea</u>: Create a spanning tree as follows:
  1. Add an edge of minimum cost that doesn't create a cycle
  2. Repeat Step 1 for |*V*|-1 edges
- This spanning tree has minimum cost because:
  - if you can replace an edge with another edge of lower cost without creating a cycle, our algorithm would have picked it
- <u>Two MST algorithms</u>: Prim (1957) and Kruskal, Jr. (1956)
  - Differ in how an edge of minimum cost is picked

Prim's Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize: *V'* = {*v0*} and *E'* = {}
2. Choose a vertex *v* not in *V'* such that *edge weight from v to a vertex in V' is minimal* (get greedy!)
3. Add *v* to *V'* and the edge to *E'* if no cycle is created
4. Repeat until all vertices have been added



Prim's Algorithm: Implemented and Analyzed

- Implementation details:
  1. Initialize cost of each node to ∞ and mark it unknown
  2. Initialize cost of one selected node S to 0, with Prev[S] = 0
  3. While there are unknown nodes left in the graph
     1. Select the unknown node *N* with the *lowest cost*
     2. Mark *N* as known
     3. For each unknown node *A* adjacent to *N*
        If cost of (*N*, *A*) < *A*'s cost
          A's cost = cost of (*N*, *A*)
          Prev[*A*] = *N* *//store preceding node*
- This is almost identical to Dijkstra's algorithm!
- Run time is O($|V|^2$) without heaps and O($|V| \log |V| + |E| \log |V|$) using binary heaps

Kruskal finds another greedy way to MST

- In 1956, J. B. Kruskal, Jr. found another way to find MSTs
- Main Idea: Select edges in order of increasing cost and accept an edge only if it does not cause a cycle
- Pseudocode for Kruskal's MST algorithm:
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue with key = edge cost
  - Repeat until |V|-1 edges have been accepted
    - *Extract cheapest edge*
    - If it forms a cycle, ignore it
      else accept the edge – it will join two existing trees and yield a larger tree
  - Return the accepted edges (they form the spanning tree)

Kruskal's Algorithm in C

```
Forest Kruskal_MST( Graph g, int n, double **costs )
{
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = BuildHeap( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = DeleteMin( q );
       } while ( Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

n = $|V|$

Initial Forest: single vertex trees

Priority Q of edges

Kruskal's Algorithm in C

```
Forest Kruskal_MST( Graph g, int n, double **costs )
{
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = BuildHeap( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = DeleteMin( q );
       } while ( Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

Need n-1 edges to fully connect (span) n vertices

Kruskal’s Algorithm in C

```
Forest Kruskal_MST( Graph g, int n, double **costs )
{
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = BuildHeap( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = DeleteMin( q );
       } while ( Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

Try the cheapest edge

Until we find one that doesn’t form a cycle

... and add it to the forest!

Kruskal’s Algorithm in C

```
Forest Kruskal_MST( Graph g, int n, double **costs )
{
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = BuildHeap( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = DeleteMin( q );
       } while ( Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

But how do we detect a cycle?

Hints for Detecting Cycles in Kruskal’s Method

- Initially, you have n different elements (single vertex trees)
- After you have added some edges, you have fewer elements – several disconnected trees, each with a <u>subset</u> of vertices
- When do you get a cycle? If you add an edge (u,v) where both u and v are already in the same tree T_i, you get a cycle
  - Therefore, to check for cycles, you only need to <u>find</u> out if u and v are in the same tree
  - If not, then the edge can be added and we <u>union</u> vertices in u’s tree with vertices in v’s tree
- What is your favorite data structure for such operations?

Disjoint Set ADT in Kruskal’s Algorithm

- Here’s how the disjoint set ADT makes an appearance:
  - In Kruskal’s algorithm, connected vertices form equivalence classes (they are in the same tree)
    - “Being connected” is the equivalence relation
- Initially, each vertex is in a class by itself
- As edges are added, more vertices become related and the equivalence classes grow
- Until finally all the vertices are in a single equivalence class

Union/Find in Kruskal’s Algorithm

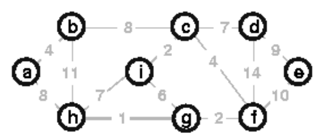
- Representatives
  - One vertex in each class can be the representative of that class
  - Vertices can be stored in up-tree data structures with roots = class representatives
    - This is what we used for Union-Find
- Detecting cycles is easy!
  - For each edge (u,v) that you’re going to add
    - If Find(u) == Find(v), then u and v are in the same class (same tree) and therefore the edge will form a cycle
    - Otherwise, we accept the edge and do Union(u,v)

Kruskal’s Algorithm in C

```
Forest Kruskal_MST( Graph g, int n, double **costs ) {
   Forest T;
   Queue q;
   Edge e;
   DisjSet S = InitializeSet( g );
   T = ConsForest( g );
   q = BuildHeap( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = DeleteMin( q ); // e = (u,v)
       } while ( (Find(u,S) == Find(v,S)) );
       AddEdge( T, e );
       Union(S, u, v);
   }
   return T;  }
```

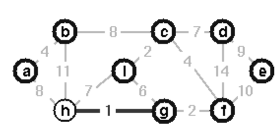

Kruskal in action

All the vertices are in single element trees



Each vertex is its own representative

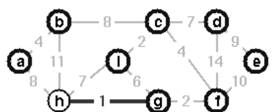
Kruskal in action



The cheapest edge is h-g

Add it to the forest, joining h and g into a 2-element tree

Kruskal in action



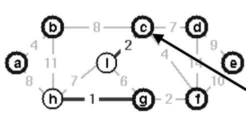
Add it to the forest, joining **h** and **g** into a 2-element tree

Choose **g** as its representative

Kruskal in action

The next cheapest edge is **c-i**

Add it to the forest, joining **c** and **i** into a 2-element tree



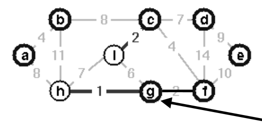
Choose **c** as its representative

Our forest now has 2 two-element trees and 5 single vertex ones

Kruskal in action

The next cheapest edge is **g-f**

Add it to the forest, joining **g** and **f** into a 3-element tree



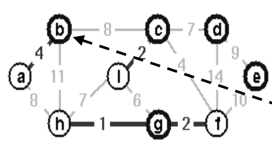
Choose **g** as its representative

Our forest now has 1 three-element tree, 1 two-element tree, and 4 single vertex ones

Kruskal in action

The next cheapest edge is **a-b**

Add it to the forest, joining **a** and **b** into a 2-element tree



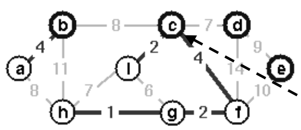
Choose **b** as its representative

Our forest now has only 2 single vertex trees

Kruskal in action

The next cheapest edge is **c-f**

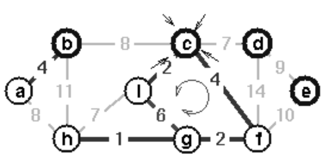
Add it to the forest: ***Merge*** **two 2-element trees (Union the 2 sets)**



Choose the rep of one as its representative

Kruskal in action

The next cheapest edge is **g-i**



Find(g) is c

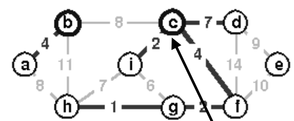
Find(i) is also c

g-i forms a cycle!

Ignore this edge

Kruskal in action

The next cheapest edge is **c-d**



Find(c) is c

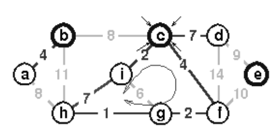
Find(d) is d

c-d joins two trees, so we add d

.. and keep c as the representative

Kruskal in action

The next cheapest edge is **h-i**



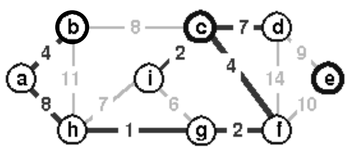
Find(h) is c

Find(i) is c

h-i forms a cycle, so ignore it!

Kruskal in action

The next cheapest edge is **a-h**



Find(**a**) is **b**

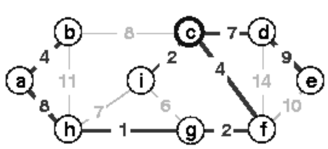
Find(**h**) is **c**

a-h joins two trees, and we add it

Kruskal done!

The next cheapest edge is **b-c**

But **b-c** forms a cycle



So add **d-e** instead

That's n-1 edges added – we now have a spanning tree!

Kruskal's Algorithm: Analysis

```
Forest Kruskal_MST( Graph g, int n, double **costs ) {
   Forest T;
   Queue q;
   Edge e;
   DisjSet S = InitializeSet( g );      → O(|V|)
   T = ConsForest( g );                 → O(|V|)
   q = BuildHeap( g, costs );           → O(|E|)
   for(i=0;i<(n-1);i++) {
      do {
         e = DeleteMin( q ); // e = (u,v)
      } while ( (Find(u,S) == Find(v,S)) );   → ≈ O(1) amortized
      AddEdge( T, e ); O(1)
      Union(S, u, v);                   → ≈ O(1) amortized
   }
   return T;  }
```

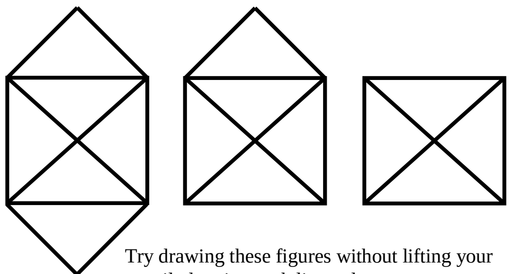
Worst case: DeleteMin $|E|$ edges, each $O(\log |E|)$

Total time = $O(|E| \log |E|)$

Kruskal versus Prim

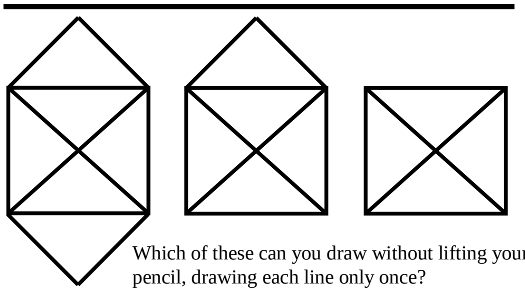
- Worst case running time
  - Prim: $O(|V| \log |V| + |E| \log |V|)$
  - Kruskal: $O(|E| \log |E|) = O(|E| \log |V|)$ since $|E| = O(|V|^2)$
- Kruskal usually runs much faster than $O(|E| \log |V|)$ in practice
  - Not all edges need to be DeleteMin-ed typically
  - The required $|V|$-1 edges are usually found quickly
  - So, Kruskal tends to be faster than Prim

It’s Puzzle Time!



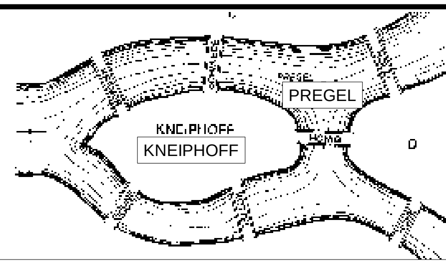
Try drawing these figures without lifting your pencil, drawing each line only once…
(begin: memories of 4^{th} grade days…)

It’s Puzzle Time!



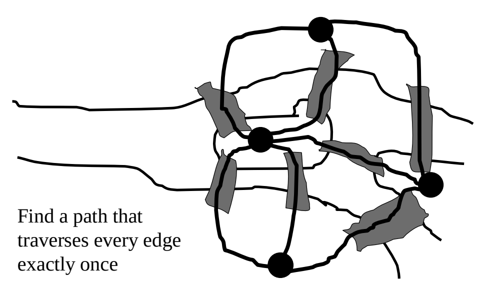
Which of these can you draw without lifting your pencil, drawing each line only once?
Can you start and end at the same point?
(end: memories of 4^{th} grade days…)

Historical Puzzle: Seven Bridges of Königsberg



Want to cross all bridges but…
Can cross each bridge only once (High toll to cross twice?!)

A “Multigraph” for the Bridges of Königsberg



Find a path that traverses every edge exactly once

Euler Circuits and Tours

- Euler tour: a path through a graph that visits each edge exactly once
- Euler circuit: an Euler tour that starts and ends at the same vertex
- Named after Leonhard Euler (1707-1783), who cracked this problem and founded graph theory in 1736
- Some observations for undirected graphs:
  - An Euler circuit is only possible if the graph is connected and each vertex has even degree (= # of edges on the vertex) [Why?]
  - An Euler tour is only possible if the graph is connected and either all vertices have even degree or exactly two have odd degree [Why?]

Euler Circuit Problem

- Problem: Given an undirected graph G = (V,E), find an Euler circuit in G
- Note: Can check if one exists in linear time (how?)
- Given that an Euler circuit exists, how do we *construct* an Euler circuit for G?
- Hint: Think deep! We've discussed the answer in depth before…

Next Class:

Constructing Euler circuits

The vast gulf between Euler and Hamiltonian circuits

The dreaded world of NP hardness

To Do:

Programming Assignment #2 (Due in 6 days!!)

Finish reading chapter 9 (and have a great weekend!)