CSCI 136
Data Structures & Advanced Programming

Lecture 32

Fall 2018

Instructors: Bills

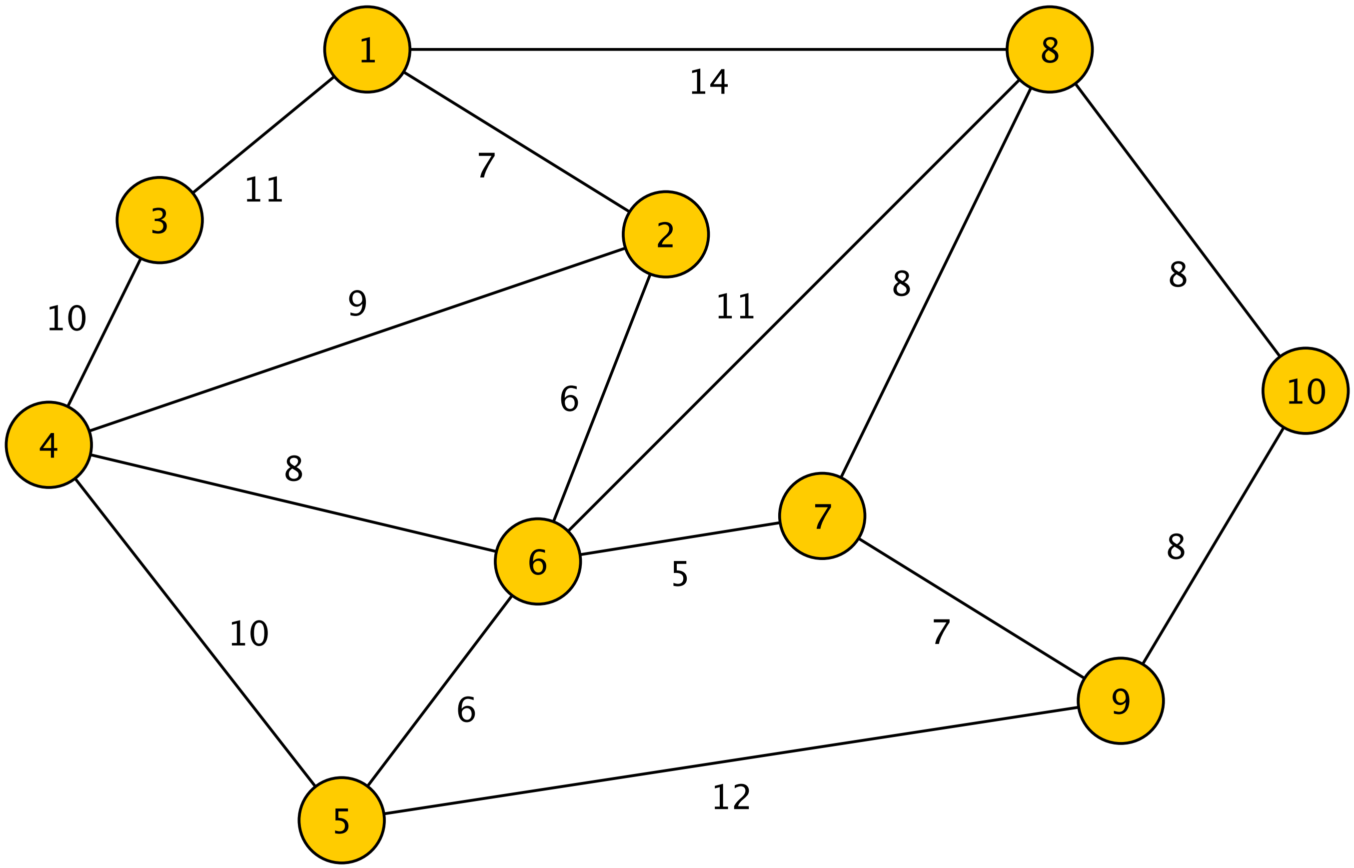
Last Time

- Adjacency List Implementation Details
  - Featuring many Iterators!
- More Fundamental Graph Properties
- An Important Algorithm: Minimum-cost spanning subgraph

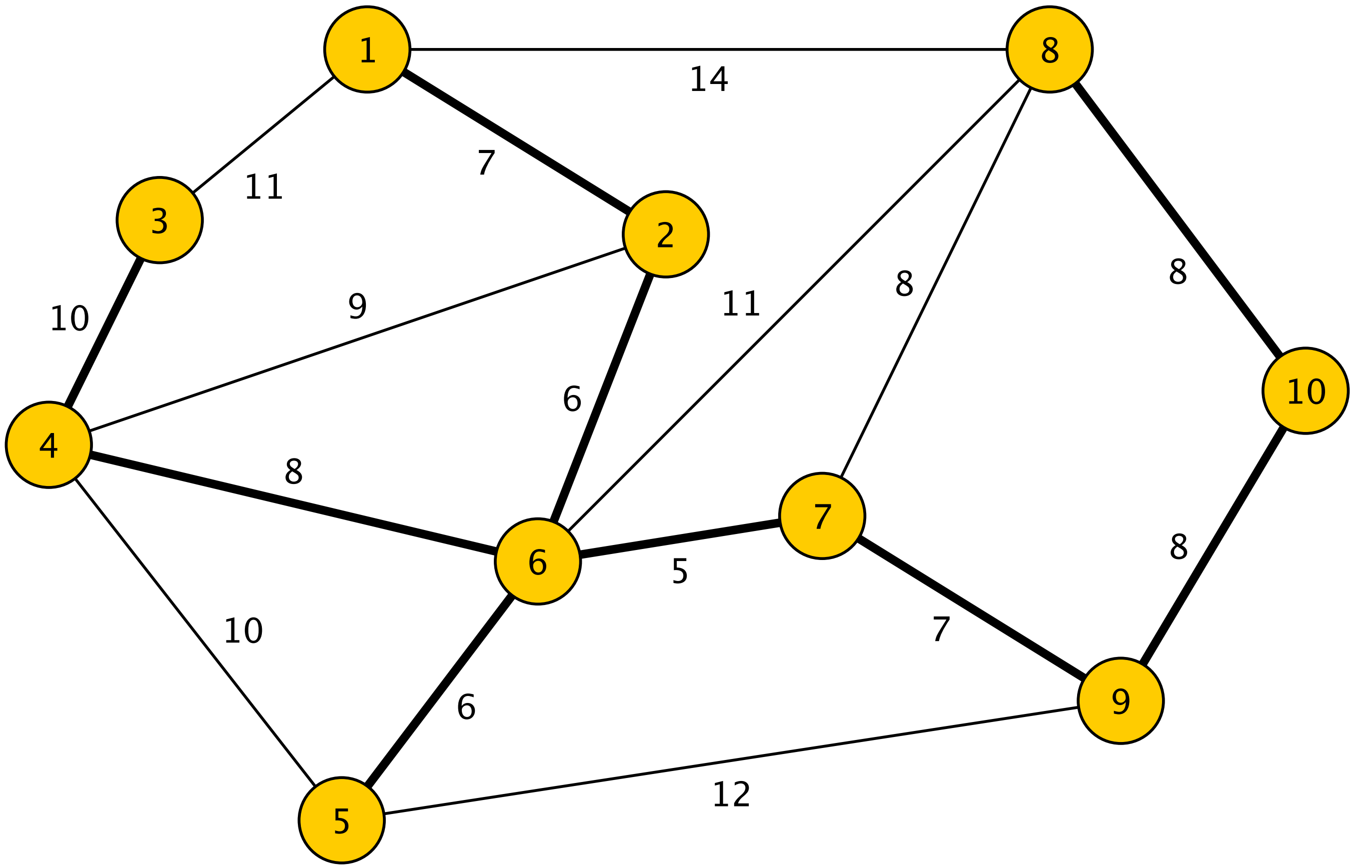
Today's Outline

- More on Prim's Algorithm
- More Core Algorithms: Directed Graphs
  - Dijkstra's Algorithm

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



Recall: Finding a MCST

Suppose we just wanted to find a PCST (pretty cheap spanning tree), here's one idea:

Grow It Greedily!

- Pick a vertex and find its cheapest incident edge. Now we have a (small) tree
- Repeatedly add the cheapest edge to the tree that keeps it a tree (connected, no cycles)
- How close might this get us to the MCST?

Recall: An Amazing Fact

Thm: (Prim 1957) The greedy tree-growing algorithm always finds a minimum-cost spanning tree for any connected graph.

Contrast this with the greedy exam scheduling algorithm, which does *not* always find a minimum coloring

Recall: The Key

Lemma: Let $G=(V,E)$ be a connected graph and let V_1 and V_2 be a partition of V.

Then *every* MCST of G contains a cheapest edge between V_1 and V_2

Note: If all edge costs are *distinct* there is *only one* cheapest edge between V_1 and V_2

Using The Key to Prove Prim

We'll assume all edge costs are distinct

Otherwise proof is slightly less elegant

Let T be the tree produced by the greedy algorithm and suppose T* is a MCST for G

Claim: T = T*

Idea of Proof: Show that every edge added to the tree T by the greedy algorithm is in T*

Clearly the first edge added to T is in T*

Why? Use the key!

Using The Key

Now use induction!

- Suppose, for some k **≥ 1, that the** first k edges added to T are in T*. These form a tree T_k
- Let V_1 be the vertices of T_k and let $V_2 = V-V_1$
- Now, the greedy algorithm will add to T the cheapest edge e between V_1 and V_2
- But any MCST contains the (only!) cheapest edge between V_1 and V_2, so e is in T*
- Thus the first k+1 edges of T are in T*

Prim's Algorithm

prim(G) // finds a MCST of connected $G=(V,E)$

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V - \{v\}$

while($|V_1| < |V|$)

  let $e \leftarrow$ *cheapest edge between* V_1 *and* V_2

  add e to MCST

  let $u \leftarrow$ *the vertex of e in* V_2

  move u from V_2 *to* V_1;

Prim's Algorithm

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V - \{v\}$

let A be the set of all edges between V_1 *and* V_2

while$(|V_1|<|V|)$

  let e$\leftarrow$*cheapest edge in A between* V_1 *and* V_2

  add e to MCST

  let u$\leftarrow$*the vertex of e in* V_2

  remove from A any edges from V_1 *to u*

  move u from V_2 *to* V_1*;*

  add to A all edges incident to u

Prim's Algorithm (Variant)

- Note: If G is not connected, A will eventually be empty even though $|V_1| < |V|$
- We fix this by
  - Replacing *while($|V_1| < |V|$)* with
    - *while($|V_1| < |V|$) && $A \neq \emptyset$)*
  - Replacing
    - *until e is an edge between V_1 and V_2*
  - with
    - *until $A \neq \emptyset$ or e is an edge between V_1 and V_2*
- Then Prim will find the MCST for the component containing v

Prim's Algorithm (Variant)

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V - \{v\}$

let A be the set of all edges between V_1 *and* V_2

while $|V_1| < |V|$ && $|A| > 0$

  repeat

    remove cheapest edge e from A

  until A is empty || e is an edge between V_1 *and* V_2

  if e is an edge between V_1 *and* V_2

    let $v \leftarrow$ *the vertex of e in* V_2

    move v from V_2 *to* V_1*;*

    add to A all edges incident to v

Implementing Prim's Algorithm

- We'll "build" the MCST by marking its edges as "visited" in G
- We'll "build" V_1 by marking its vertices visited
- How should we represent A?
  - What operations are important to A?
    - Add edges
    - Remove cheapest edge
  - A priority queue!
- When we remove an edge from A, check to ensure it has one end in each of V_1 and V_2

ComparableEdge Class

- Values in a PriorityQueue need to implement Comparable
- We wrap edges of the PQ in a class called ComparableEdge
  - It requires the label used by graph edges to be of a Comparable type

Prim's Algorithm (Variant)

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V - \{v\}$

let A be the set of all edges between V_1 *and* V_2

while $|V_1| < |V|$ && $|A| > 0$

    repeat

        remove cheapest edge e from A

    until A is empty || e is an edge between V_1 *and* V_2

    if e is an edge between V_1 *and* V_2

        let $v \leftarrow$ *the vertex of e in* V_2

        move v from V_2 *to* V_1*;*

        add to A all edges incident to v

MCST: The Code

```
PriorityQueue<ComparableEdge<String,Integer>> q =
  new SkewHeap<ComparableEdge<String,Integer>>();

String v = null;           // current vertex
Edge<String,Integer> e; // current edge
boolean searching;         // still building tree
g.reset();                 // clear visited flags

// select a node from the graph, if any
Iterator<String> vi = g.iterator();
if (!vi.hasNext()) return;
v = vi.next();
```

MCST: The Code

```
do {
    // visit the vertex and add all outgoing edges
    g.visit(v);
    Iterator<String> ai = g.neighbors(v);
    while (ai.hasNext()) {
        // turn it into outgoing edge
        e = g.getEdge(v,ai.next());
        // add the edge to the queue
        q.add(new
          ComparableEdge<String,Integer>(e));
    }
    ...
```

MCST: The Code

```
      searching = true;
      while (searching && !q.isEmpty()) {
            // grab next shortest edge
            e = q.remove();
            // Is e between V1 and V2 (subtle code!!)
            v = e.there();
            if (g.isVisited(v)) v = e.here();
            if (!g.isVisited(v)) {
                  searching = false;
                  g.visitEdge(g.getEdge(e.here(),
                        e.there()));
            }
      }
} while (!searching);
```

Prim : Space Complexity

- Graph: O(|V| + |E|)
  - Each vertex and edge uses a constant amount of space
- Priority Queue O(|E|)
  - Each edge takes up constant amount of space
- Every other object (including the neighbor iterator) uses a constant amount of space
- Result: O(|V| + |E|)
  - Optimal in Big-O sense!

Prim : Time Complexity

Assume Map ops are O(1) time (not quite true!)

For each iteration of do ... while loop

- Add neighbors to queue: O(deg(v) log |E|)
  - Iterator operations are O(1) [Why?]
  - Adding an edge to the queue is O(log |E|)
- Find next edge: O(# edges checked * log |E|)
  - Removing an edge from queue is O(log |E|) time
  - All other operations are O(1) time

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step 1: Add neighbors to queue:

- For each vertex, it's O(deg(v) log |E|) time
- Adding over all vertices gives

$$\sum_{v \in V} \deg(v) \log |E| = \log |E| \sum_{v \in V} \deg(v) = \log |E| * 2 |E|$$

- which is O(|E| log |E|) = O(|E| log |V|)
  - $|E| \leq |V|^2$,so $\log |E| \leq \log |V|^2 = 2 \log |V|$ = O(log |V|)

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step 2: Find next edge: O(# edges checked * log |E|)

- Each edge is checked at most once
- Adding over all edges gives O(|E| log |E|) again

Thus, overall time complexity (worst case) of Prim's Algorithm is O(|E| log |V|)

- Typically written as O(m log n)
  - Where m= |E| and n = |V|

Single Source Shortest Paths

The Problem: Given a graph G and a starting vertex v, find, for *each* vertex u ≠ v reachable from v, a shortest path from v to u.

- The Single Source Shortest Paths Problem
- Arises in many contexts, including network communications
- Uses edge weights (but we'll call them "lengths"): assume they are non-negative numbers
- Could be a directed or undirected graph

Single Source Shortest Paths

- We'll look at directed graphs
  - So the paths must be directed paths
- Let's think....
- Suppose we have a set shortest paths $\{P_u : u \neq v\}$, where P_u is a shortest path from v to u
- Let H be the subgraph of G consisting of each vertex of G along with all of the edges in each P_u
- What can we say about H?

Single Source Shortest Paths

Observations

- If some vertex u has in-degree greater than 1, we can drop one of the incoming edges: Why?
  - Only the last edge of the shortest path from v-u is needed as an in-edge to u [Why?]
  - So we assume H has in-deg(u)=1 for all u≠**v**
    - We need *no* in-edges for v [Why?]
- H can't have any directed cycles
  - Well, v can't be on any cycles (in-deg(v) = 0)
  - If there were a cycle, some vertex on it would have in-degree > 1 [Why?]

Single Source Shortest Paths

Observations

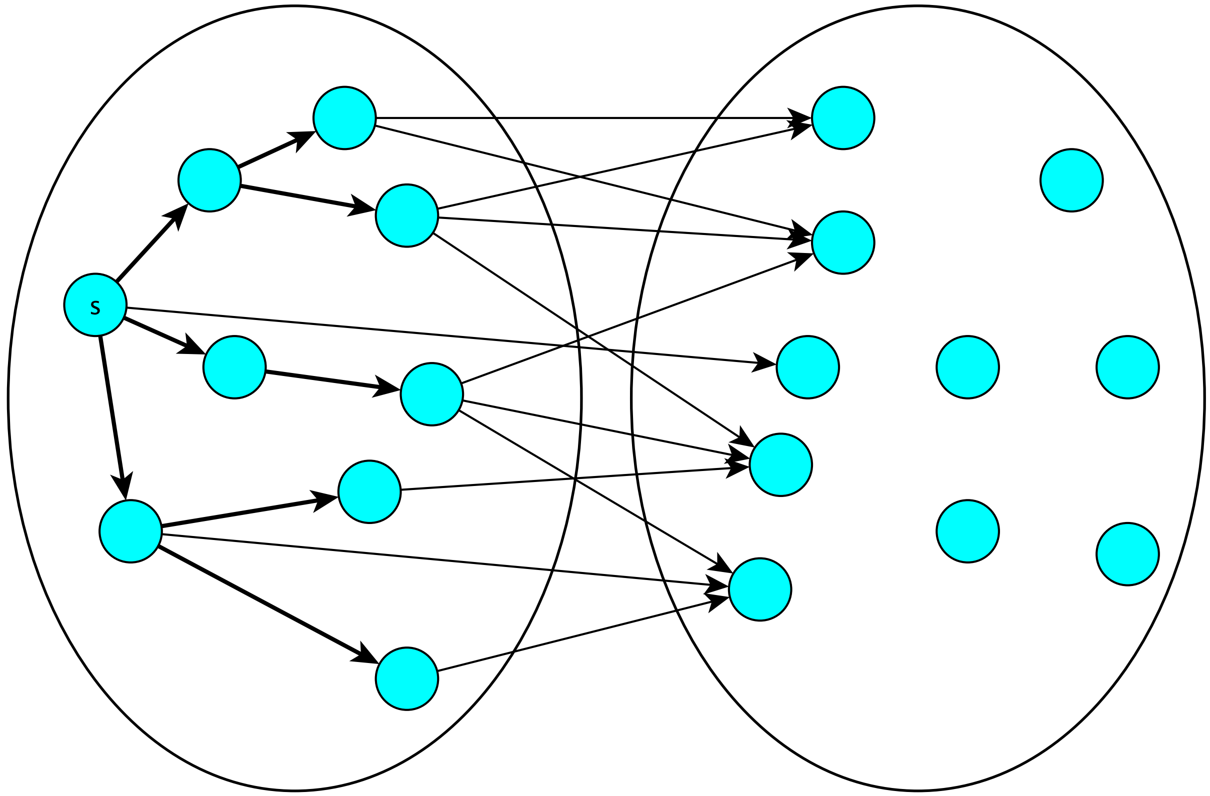
- In fact, even disregarding edge directions, there would be no cycles
  - Some vertex would have in-degree at least 2
    - Or else there's a directed cycle (Why?)
- So, we can assume that there is some set of shortest paths that forms a (directed) tree
- This suggests that we try again to

  Greedily grow a tree
- The question is: How?

The Right Kind of Greed

- Build a MCST?
  - No: It won't always give shortest paths
- A start: take shortest edge from start vertex s
  - That must be a shortest path!
  - And now we have a small tree of shortest paths
- What next?
  - Design an algorithm thinking inductively
  - Suppose we have found a tree T_k that has shortest paths from s to the k-1 vertices "closest" to s
  - What vertex would we want to add next?

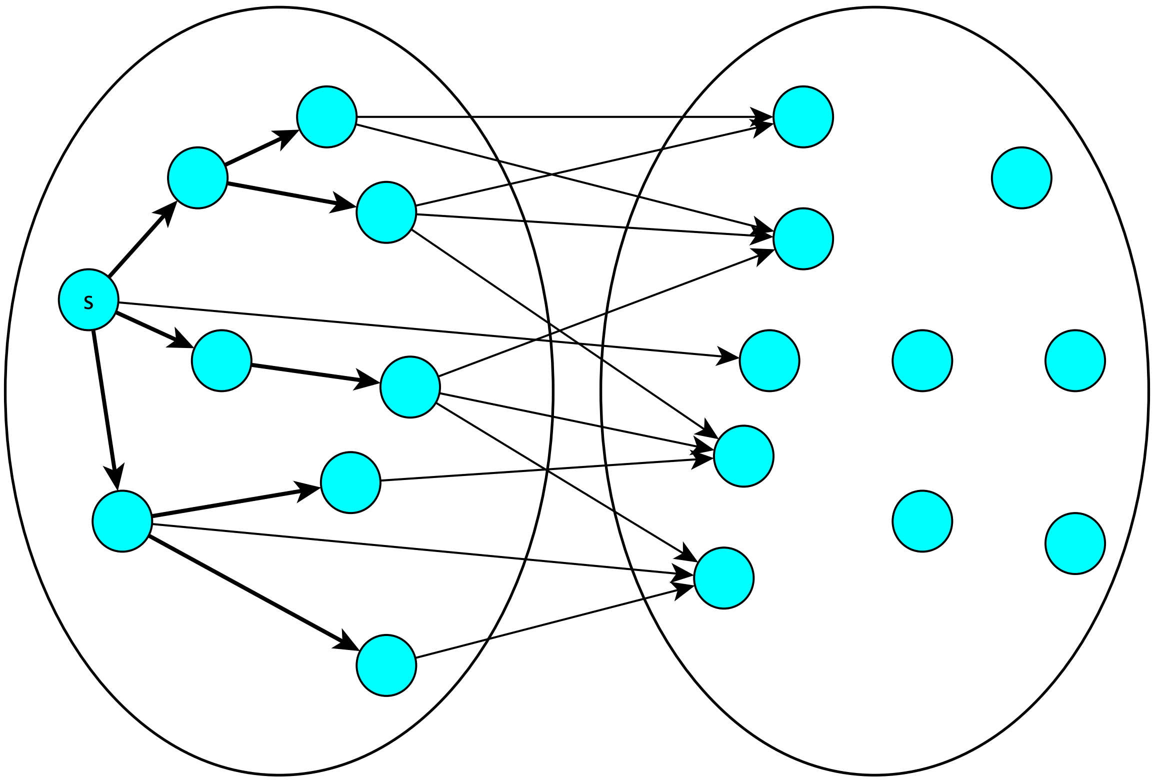
Finding the Best Vertex to Add to T_k



Not all edges are displayed

Question: Can we find the next closest vertex to s?

What's a Good Greedy Choice?



Idea: Pick edge e from u in T_k to v in G-T_k that minimizes the length of the tree path from s up to–and through–e

Now add v and e to T_k to get tree T_{k+1}

Now T_{k+1} is a tree consisting of shortest paths from s to the k vertices closest to s! [Proof?] Repeat until k = |V|