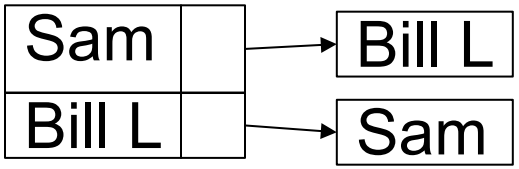
CSCI 136
Data Structures & Advanced Programming

Lecture 31

Fall 2019

Instructors:



Last Time

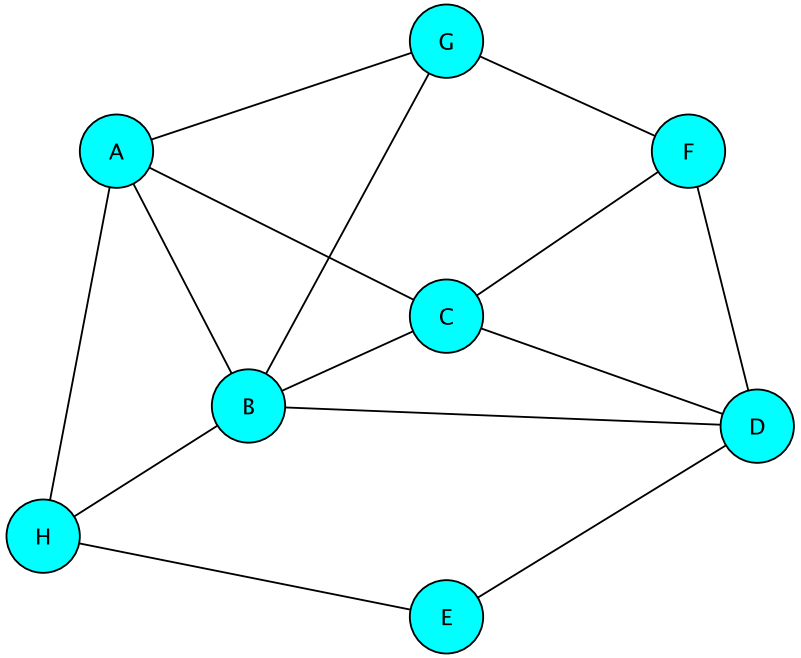
- Graph Data Structures: Implementation
  - Graph Interface
  - Graph Matrix

Today's Outline

- Graph Data Structures: Implementation
  - Adjacency List Implementation
    - Featuring many Iterators!
- Minimum Spanning Tree
  - Prim's algorithm

Adjacency Array: Undirected Graph

	A	B	C	D	E	F	G	H
A	0	I	I	0	0	0	I	I
B	I	0	I	I	0	0	I	I
C	I	I	0	I	0	I	0	0
D	0	I	I	0	I	I	0	0
E	0	0	0	I	0	0	0	I
F	0	0	I	I	0	0	I	0
G	I	I	0	0	0	I	0	0
H	I	I	0	0	I	0	0	0

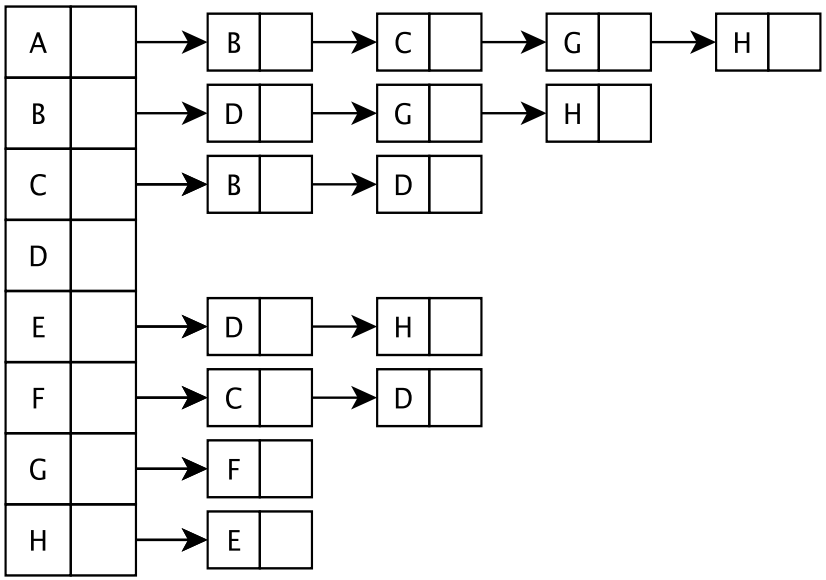


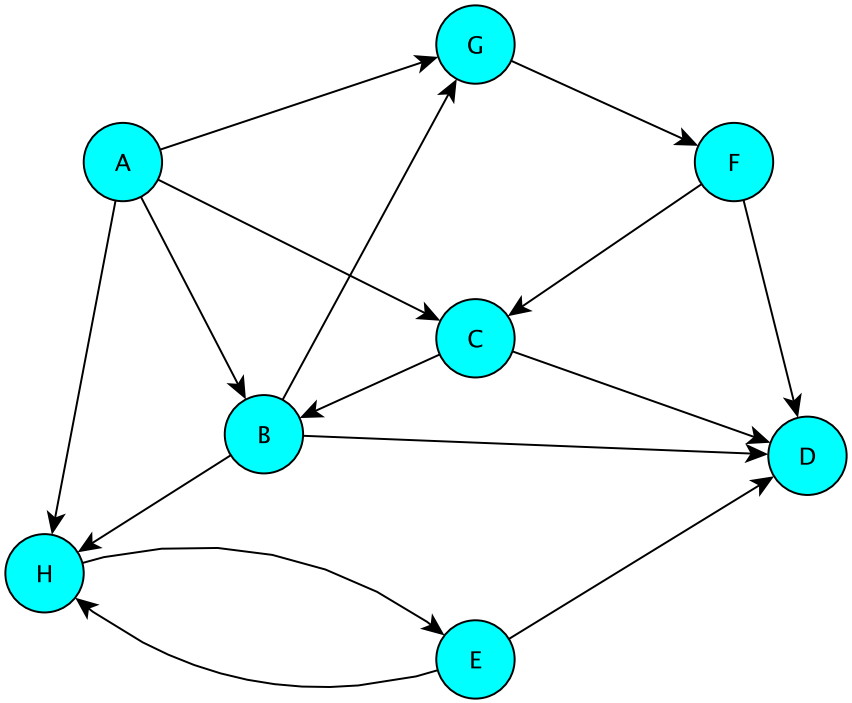
Entry (i,j) store 1 if there is an edge between i and j; else 0
E.G.: edges(B,C) = 1 = edges(C,B)

Efficiency : Assuming Fast Map Ops

	GraphMatrix
add	O(1)
addEdge	O(1)
getEdge	O(1)
removeEdge	O(1)
remove	O(|V|)
space	O($\|V\|^2$)

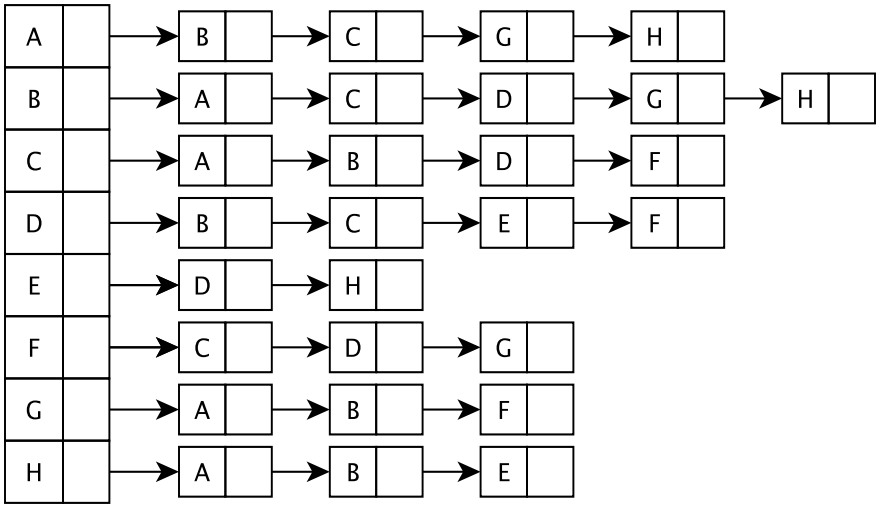
Adjacency List : Directed Graph

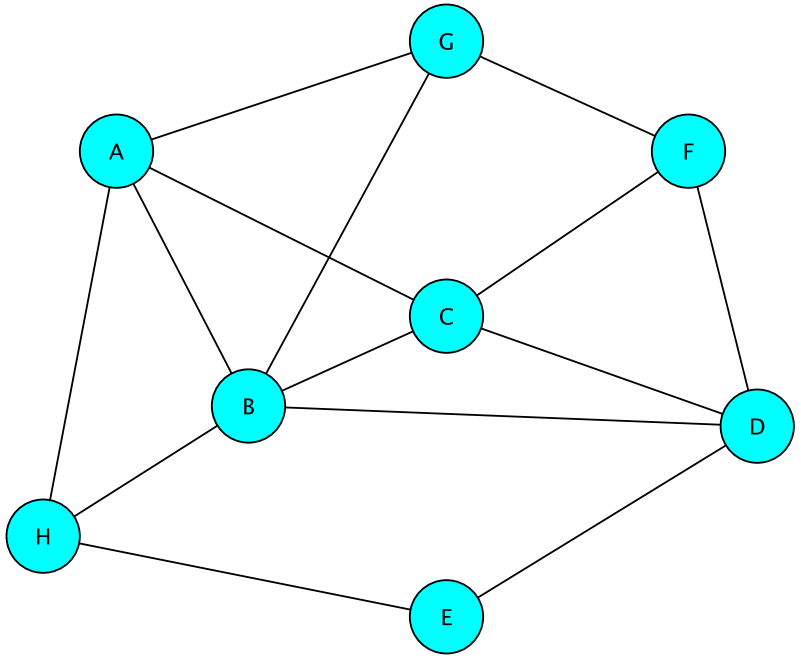




The vertices are stored in an array V[]
V[] contains a linked list of edges having a given source

Adjacency List : Undirected Graph





The vertices are stored in an array V[]
V[] contains a linked list of edges incident to a given vertex

GraphList

- Rather than keep an adjacency matrix, maintain an *adjacency list of edges* at each vertex (only keep outgoing edges for directed graphs)
- Support both directed and undirected graphs (GraphListDirected, GraphListUndirected)

Vertex and GraphListVertex

- We use the same Edge class for list-based graphs
- We extend Vertex to include an Edge list
- GraphListVertex class adds to Vertex class
  - A Structure to store edges adjacent to the vertex

    protected Structure<Edge<V,E>> adjacencies; // adjacent edges
    - adjacencies is created as a SinglyLinkedList of edges
  - Several methods

```
public void addEdge(Edge<V,E> e)
public boolean containsEdge(Edge<V,E> e)
public Edge<V,E> removeEdge(Edge<V,E> e)
public Edge<V,E> getEdge(Edge<V,E> e)
public int degree()
// and methods to produce Iterators...
```

GraphListVertex

```
public GraphListVertex(V key){
        super(key); // init Vertex fields
        adjacencies = new SinglyLinkedList<Edge<V,E>>();
}

public void addEdge(Edge<V,E> e){
        if (!containsEdge(e)) adjacencies.add(e);
}

public boolean containsEdge(Edge<V,E> e){
        return adjacencies.contains(e);
}

public Edge<V,E> removeEdge(Edge<V,E> e) {
        return adjacencies.remove(e);
}
```

GraphListVertex Iterators

```
// Iterator for incident edges
public Iterator<Edge<V,E>> adjacentEdges() {
        return adjacencies.iterator();
}

// Iterator for adjacent vertices
public Iterator<V> adjacentVertices() {
        return new GraphListAIterator<V,E>
            (adjacentEdges(), label());
}
```

GraphListAIterator creates an Iterator over *vertices* based on The Iterator over *edges* produced by `adjacentEdges()`

GraphListAIterator

GraphListAIterator uses two instance variables

```
    protected AbstractIterator<Edge<V,E>> edges;
    protected V vertex;

public GraphListAIterator(Iterator<Edge<V,E>> i, V v) {
        edges = (AbstractIterator<Edge<V,E>>)i;
        vertex = v;
}

public V next() {
        Edge<V,E> e = edges.next();
        if (vertex.equals(e.here()))
            return e.there();
        else { // could be an undirected edge!
            return e.here();
}
```

GraphListEIterator

GraphListEIterator uses one instance variable

```
protected AbstractIterator<Edge<V,E>> edges;
```

GraphListEIterator

- Takes the Map storing the vertices
- Uses it to build a linked list of all edges
- Gets an iterator for this linked list and stores it, using it in its own methods

GraphList

- To implement GraphList, we use the GraphListVertex (GLV) class
- GraphListVertex class
  - Maintain linked list of edges at each vertex
  - Instance vars: label, visited flag, linked list of edges
- GraphList abstract class
  - Instance vars:
    - Map<V,GraphListVertex<V,E>> dict; // label -> vertex
    - boolean directed; // is graph directed?
- How do we implement key GL methods?
  - GraphList(), add(), getEdge(), …

```
protected GraphList(boolean dir){
      dict = new Hashtable<V,GraphListVertex<V,E>>();
      directed = dir;
}

public void add(V label) {
      if (dict.containsKey(label)) return;
      GraphListVertex<V,E> v = new
            GraphListVertex<V,E>(label);
      dict.put(label,v);
}

public Edge<V,E> getEdge(V label1, V label2) {
      Edge<V,E> e = new Edge<V,E> (get(label1),
      get(label2), null, directed);
      return dict.get(label1).getEdge(e);
}
```

GraphListDirected

- GraphListDirected (GraphListUndirected) implements the methods requiring different treatment due to (un)directedness of edges
  - addEdge, remove, removeEdge, …

```
// addEdge in GraphListDirected.java
// first vertex is source, second is destination
 public void addEdge(V vLabel1, V vLabel2, E label) {
   // first get the vertices
   GraphListVertex<V,E> v1 = dict.get(vLabel1);
   GraphListVertex<V,E> v2 = dict.get(vLabel2);
   // create the new edge
   Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), label, true);
   // add edge only to source vertex linked list (aka adjacency list)
   v1.addEdge(e);
}
```

```
public V remove(V label) {
   //Get vertex out of map/dictionary
   GraphListVertex<V,E> v = dict.get(label);

   //Iterate over all vertex labels (called the map "keyset")
   Iterator<V> vi = iterator();
   while (vi.hasNext()) {
        //Get next vertex label in iterator
        V v2 = vi.next();

        //Skip over the vertex label we're removing
        //(Nodes don't have edges to themselves...)
        if (!label.equals(v2)) {
            //Remove all edges to "label"
            //If edge does not exist, removeEdge returns null
            removeEdge(v2,label);
        }
    }
    //Remove vertex from map
    dict.remove(label);
    return v.label();
}
```

```
public E removeEdge(V vLabel1, V vLabel2) {
   //Get vertices out of map
   GraphListVertex<V,E> v1 = dict.get(vLabel1);
   GraphListVertex<V,E> v2 = dict.get(vLabel2);

   //Create a "temporary" edge connecting two vertices
   Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), null, true);

   //Remove edge from source vertex linked list
   e = v1.removeEdge(e);
   if (e == null) return null;
   else return e.label();
}
```

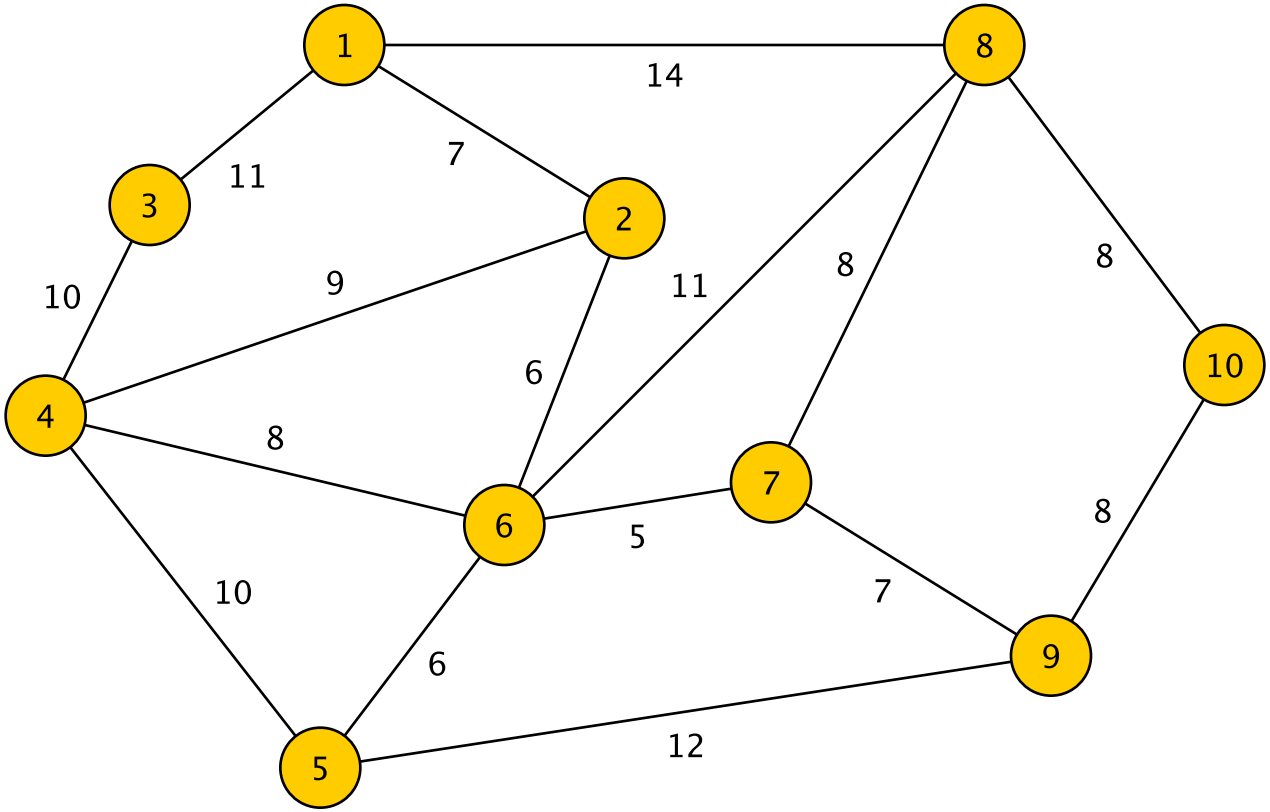
Efficiency Revisited

- Assume Map operations are O(1) (for now)
  - |E| = number of edges
  - |V| = number of vertices
- Runtime of add, addEdge, getEdge, removeEdge, remove?
- Space usage?
- Conclusions
  - Matrix is better for dense graphs
  - List is better for sparse graphs
  - For graphs "in the middle" there is no clear winner

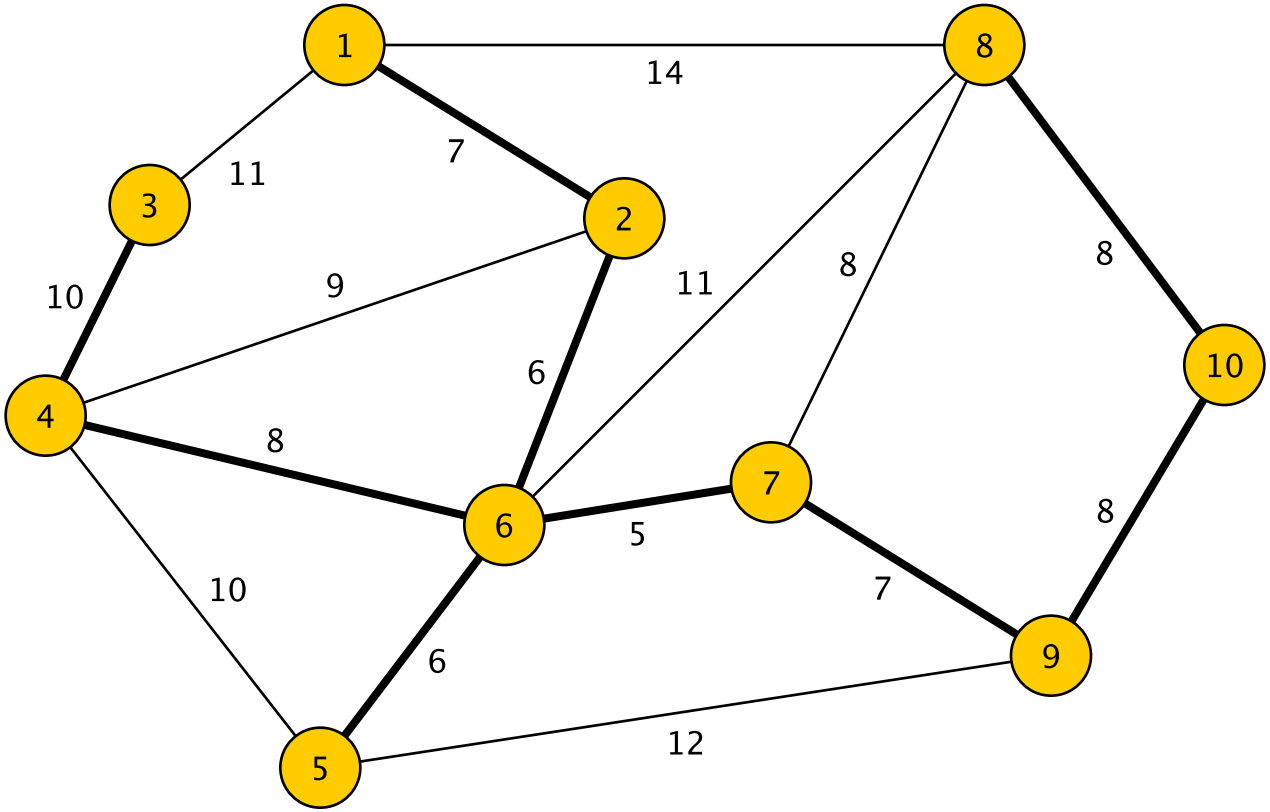
Efficiency : Assuming Fast Map

	Matrix	GraphList
add	O(1)	O(1)
addEdge	O(1)	O(1)
getEdge	O(1)	O(|V|)
removeEdge	O(1)	O(|V|)
remove	O(|V|)	O(|E|)
space	$O(\|V\|^2)$	O(|V|+|E|)

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



MST: Applications

- Cable/phone/ethernet network
- Circuit design
- Subroutine for other algorithms
  - Travelling salesman approximation
- Financial markets

Basic Graph Properties

- Recall: An undirected graph G=(V,E) is *connected* if for every pair u,v in V, there is a path from u to v (and so from v to u)
- The maximal sized connected subgraphs of G are called its *connected components*
  - Note: They are induced subgraphs of G
- An undirected graph without cycles is a *forest*
- A connected forest is called a *tree*.
  - Not to be confused with the data structure!

Facts About Graphs

Thm: Every connected graph G=(V,E) contains a spanning subgraph G'=(V,E') that is a tree

- That is, a *spanning tree*

Proof idea:

- If G is not a tree, then it contains a cycle C
- Removing an edge from C leaves G connected (why)
- Repeat until no more cycles remain

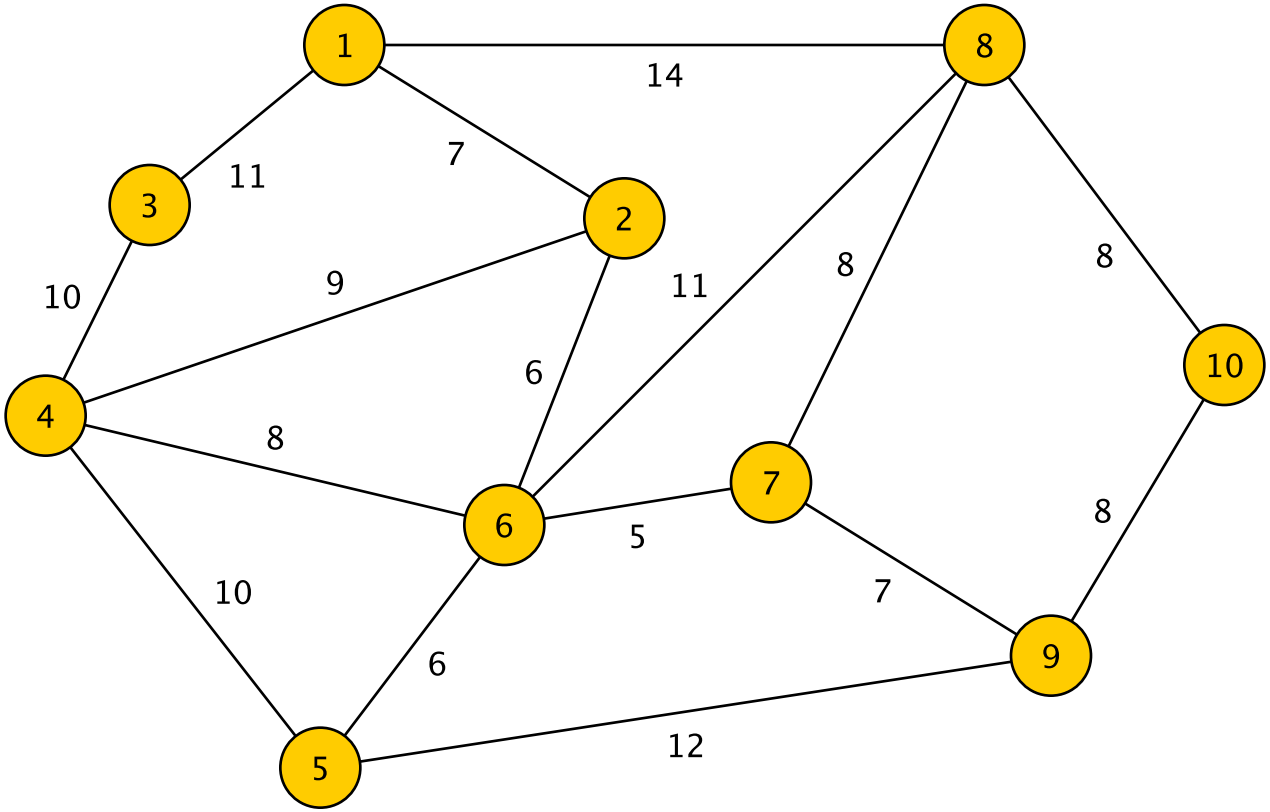
Edge-Weighted Graphs

- An *edge-weighting* of a graph G=(V,E) is an assignment of a number (weight) to each edge of G
  - We write the weight of e as w(e) or w_e
- The weight w(G') of any subgraph G' of G is the sum of the weights of the edges in G'

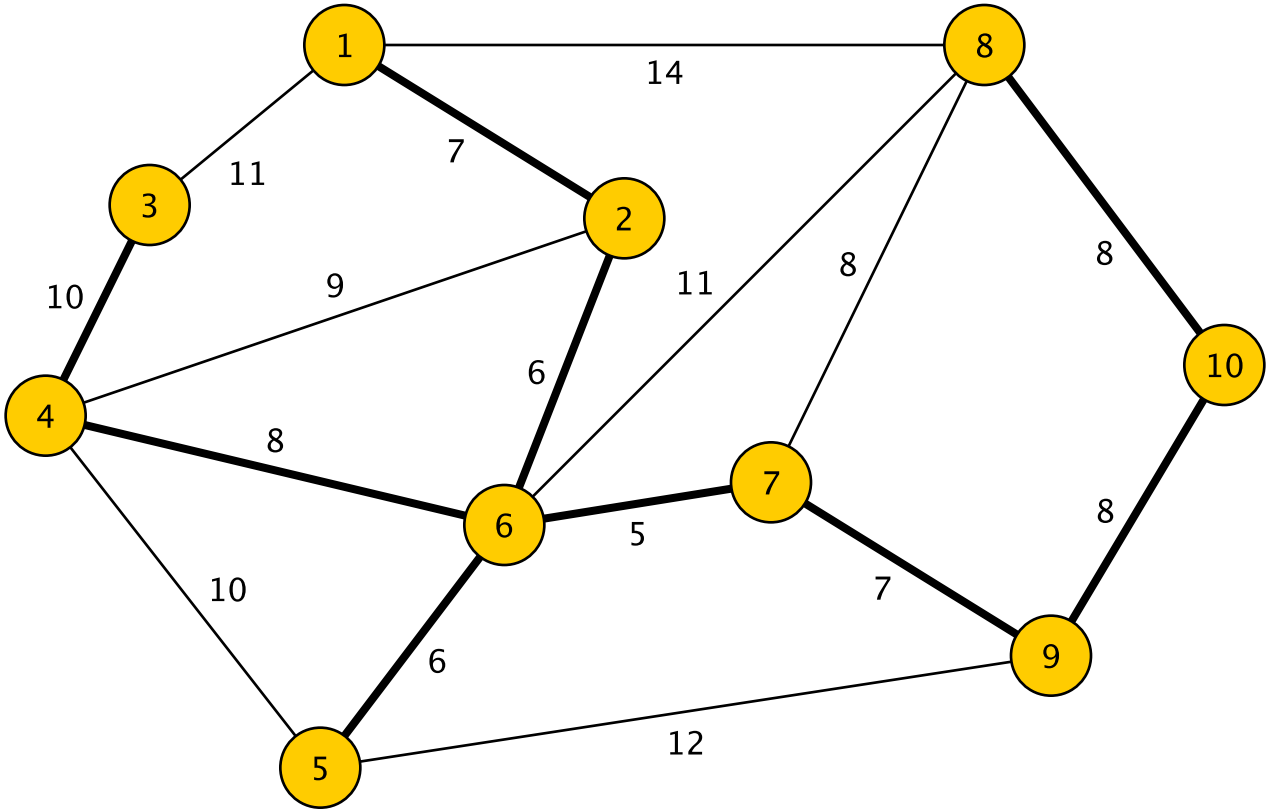
A Famous Problem

- Given a connected, undirected graph G=(V,E) with non-negative edge weights, find a minimum-weight, connected, spanning subgraph of G.
- Frequently, we refer to the edge weights as *costs* and so this problem becomes:
- Given an undirected graph G with edge costs, compute a minimum-cost spanning tree of G.

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



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)
- This method is called *Prim's Algorithm*
- How close might this get us to the MCST?

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 schedule (coloring)

Why does this work?

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 $V_2 \leftarrow V_1 - \{v\}$

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

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

    let e←cheapest edge in A between V_1 and V_2

    add e to MCST

    let u←the vertex of e in V_2

    move u from V_2 to V_1;

    add to A all edges incident to u

    // note: A may have edges with both ends in V_1

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

MCST: The Code

```
PriorityQueue<ComparableEdge<String,Integer>> q =
  new VectorHeap<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
      to the priority queue
      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(); // does e connect V1 to V2?
                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(log |E|) time each
- Find next edge: O(log |E|) time each

- Each edge is added to the queue at most once
  - O(E log |E|) time