CSCI 136
Data Structures &
Advanced Programming

Lecture 32

Fall 2017

Instructors: Bills

Last Time

- Adjacency List Implementation Details
- Time/space complexity
  - *see corrected table in slides posted online

Today's Outline

- System.out.println(GraphListDirected)?
- Fundamental Graph Properties
- Minimum-cost spanning subgraph
  - Prim's Algorithm

Printing A GraphList

- What happens when we execute the following code:

```
Graph<String, Integer> g =
  new GraphListUndirected<String, Integer>();
g.add(“CSCI 136”);
g.add(“PSYC 101”);
…
g.addEdge(“CSCI 136”, “MATH 200”, 1);
g.addEdge(“CSCI 136”, “HIST 101”, 1);
…
System.out.println(g);
```

System.out.println(g);

```
09wkj-lab11/ -> java Schedule small.txt
  <GraphListUndirected: <Hashtable: size=7
  capacity=997 key=HIST 301,
  value=<GraphListVertex: HIST 301> key=CSCI
  136, value=<GraphListVertex: CSCI 136>
  key=ENGL 201, value=<GraphListVertex: ENGL
  201> key=PHIL 101, value=<GraphListVertex:
  PHIL 101> key=MATH 251,
  value=<GraphListVertex: MATH 251> key=SOCI
  201, value=<GraphListVertex: SOCI 201>
  key=PSYC 212, value=<GraphListVertex: PSYC
  212>>>
```

????

<u>System</u>.out.println(g);

- `public final class` **<u>`System`</u>** `extends` <u>`Object`</u>

  The System class contains several useful class fields and methods. *It cannot be instantiated*.

  Among the facilities provided by the System class are *<u>standard input</u>*, *<u>standard output</u>*, and *<u>error output</u>* streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

System.out.println(g);

The `System` class has 3 static fields useful for communicating with the outside world (aka the terminal)

Fields	
Modifier and Type	**Field and Description**
static PrintStream	err The "standard" error output stream.
static InputStream	in The "standard" input stream.
static PrintStream	out The "standard" output stream.

System.**out**.println(g);

- `public class PrintStream extends FilterOutputStream implements Appendable, Closeable`

  A `PrintStream` adds functionality to another output stream, namely the ability to *print representations of various data values* conveniently. Two other features are …

System.**out**.println(g);

The **`PrintStream`** println() method is overloaded – the method executed depends on the argument type.

void	**println**() Terminates the current line by writing the line separator string.
void	**println**(boolean x) Prints a boolean and then terminate the line.
void	**println**(char x) Prints a character and then terminate the line.
void	**println**(char[] x) Prints an array of characters and then terminate the line.
void	**println**(double x) Prints a double and then terminate the line.
void	**println**(float x) Prints a float and then terminate the line.
void	**println**(int x) Prints an integer and then terminate the line.
void	**println**(long x) Prints a long and then terminate the line.
void	**println**(**String** x) Prints a String and then terminate the line.
void	**println**(**Object** x) Prints an Object and then terminate the line.

Primitive Types

String

Object

System.out.println(g);

g is a Graph, not a primitive or a String. println(g) will call the version of println that takes an Object–everything (including GraphListDirected) inherits from Object.

```
public void println(Object x)
```

Prints an Object and then terminate the line. This method calls at first String.valueOf(x) to get the printed object's string value, then behaves as though it invokes print(String) and then println().

Parameters: x - The Object to be printed.

String.valueOf(obj)

The **`PrintStream`** class' `println(Object x)` method calls `String.valueOf(x)` to convert `x` into a `String` for printing.

`public static String valueOf(Object obj)`

Parameters: `obj` - an `Object`.

Returns: if the argument is `null`, then a string equal to `"null"`; otherwise, the value of `obj.toString()` is returned.

A Chain of `toString()`s

```
System.out.println(g);
    ↳ String.valueof(g);
        ↳ g.toString();
```

GraphListDirected.java

```
public String toString() {
   return "<GraphListDirected:" +
               dict.toString() + ">";
}
```

A Chain of `toString()`s

Hashtable.java

```
public String toString()    {
    StringBuffer s = new StringBuffer();
    int i;
    s.append("<Hashtable: size=" + size() +
             " capacity=" + data.size());
    Iterator<Association<K,V>> hi =
           new HashtableIterator<K,V>(data);
    while (hi.hasNext()) {
        Association<K,V> a = hi.next();
        s.append(" key=" + a.getKey()+
                 "value=" + a.getValue());
    }
    s.append(">");
    return s.toString();
}
```

A Chain of `toString()`s

GraphListVertex.java

```
public String toString() {
    return "<GraphListVertex: "+label()+">";
}
```

The `GraphListVertex` class stores all of the adjacent edges but its `toString()` only prints the label. How do we debug?

Printing a GraphList…

- Why must write our own method?
  - We can't modify structure5.GraphListVertex
    - Plus the class is private --- the Graph interface hides it
  - We can't modify structure5.Graph
- Where should our function go?
- What should its arguments be?
- What should its return type be?

Task: implement

```
public static void printGraph(Graph<String, Integer> graph);
```

```java
// Graph, AbstractIterator implement Iterable interface
// This lets us use the for-each loop syntax!
public static <V,E> void printGraph(Graph<V, E> graph) {
   for (V vertex : graph) {
      System.out.print(vertex + " ->");
      AbstractIterator<V> neighbors =
         (AbstractIterator<V>) graph.neighbors(vertex);
      for (V neighbor : neighbors) {
         System.out.print(" " + neighbor);
      }
      System.out.println();
   }
}
```

printGraph(g);

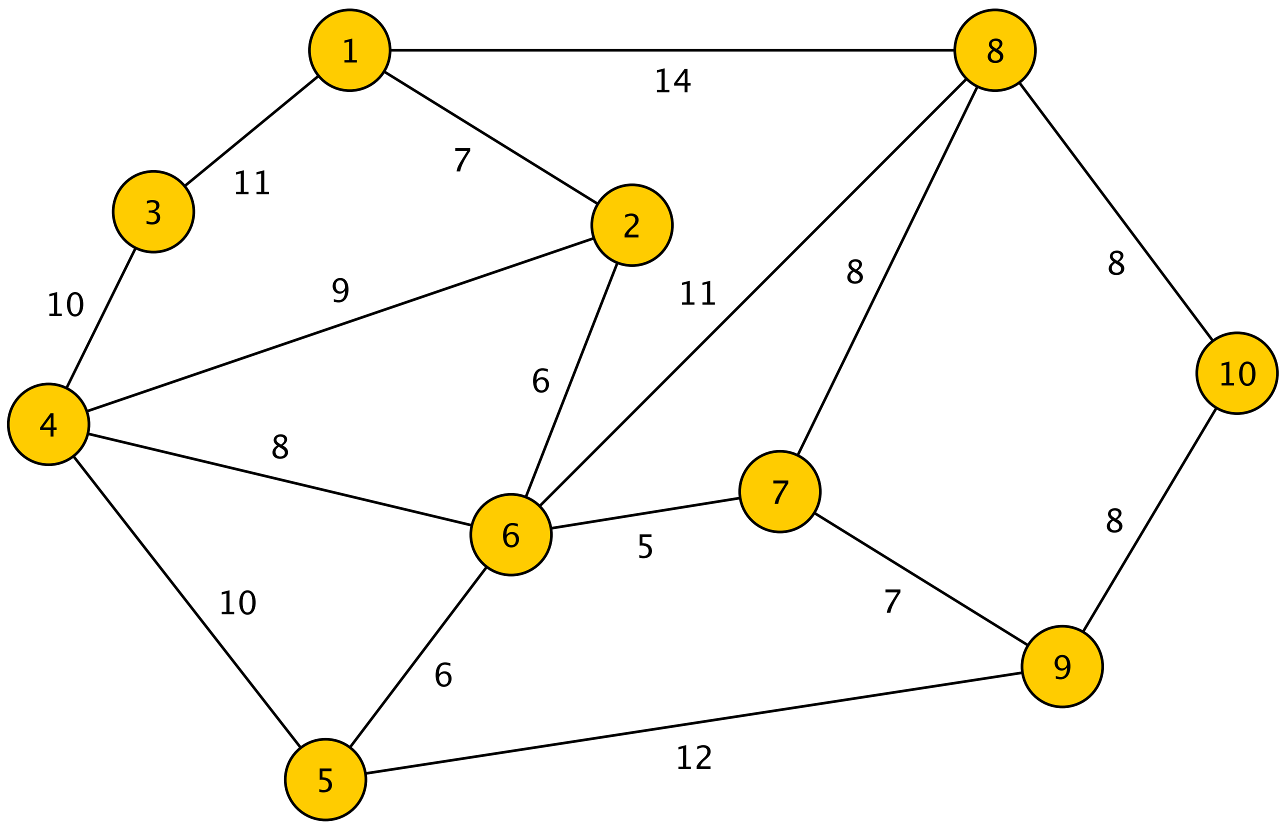
```
09wkj-lab11/ -> java Schedule small.txt

  HIST 301 -> PSYC 212 ENGL 201 CSCI 136
  CSCI 136 -> MATH 251 ENGL 201 PHIL 101 PSYC 212 HIST
301 SOCI 201
  ENGL 201 -> CSCI 136 MATH 251 PHIL 101 PSYC 212 HIST
301
  PHIL 101 -> CSCI 136 MATH 251 ENGL 201
  MATH 251 -> CSCI 136 ENGL 201 PHIL 101 SOCI 201 PSYC
212
  SOCI 201 -> CSCI 136 MATH 251 PSYC 212
  PSYC 212 -> ENGL 201 HIST 301 CSCI 136 SOCI 201 MATH
251
```

Hooray!

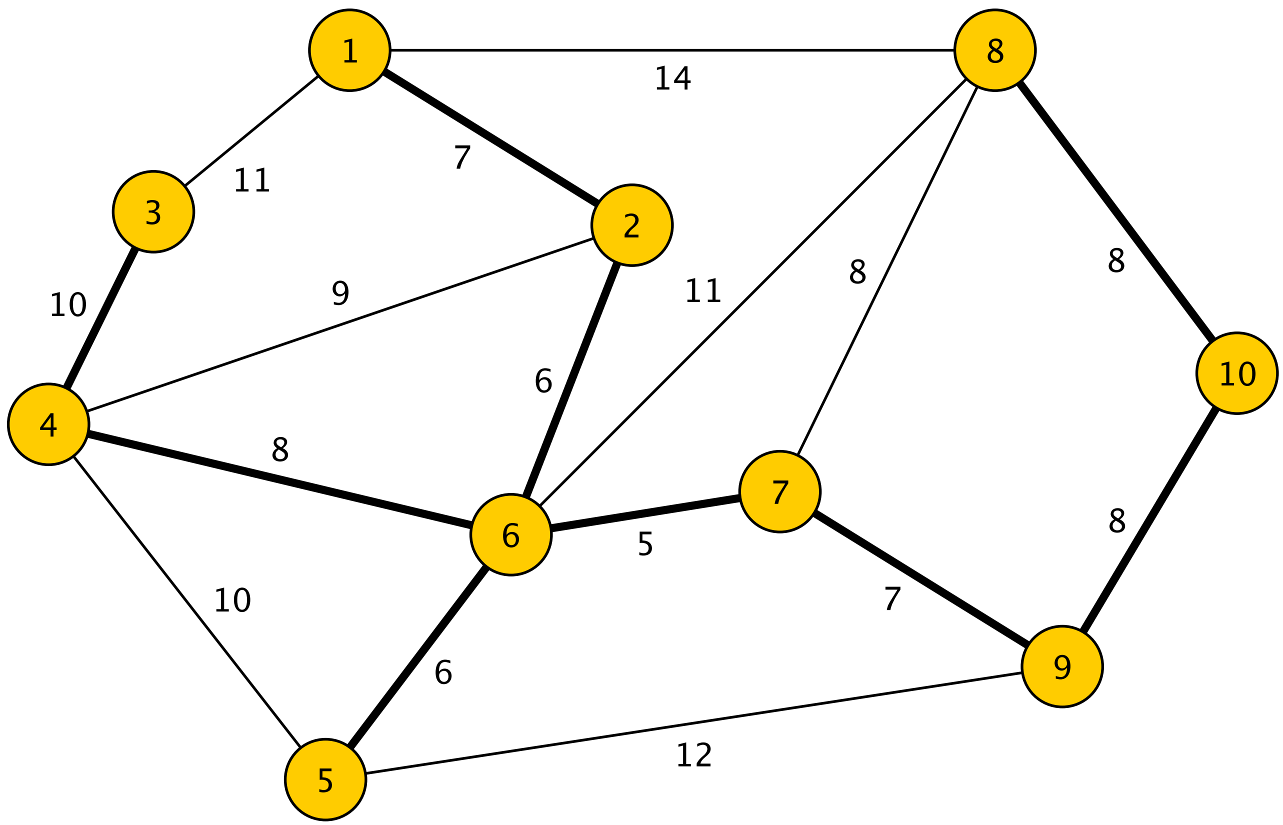
Graph Applications

Minimum-Cost Spanning Trees



Input: Undirected, edge-weighted graph

Minimum-Cost Spanning Trees



Input: Undirected, edge-weighted graph
Output: A subgraph that includes all vertices, is fully-connected, and contains no cycles. The sum of all edge weights is minimal.

Basic Graph Properties

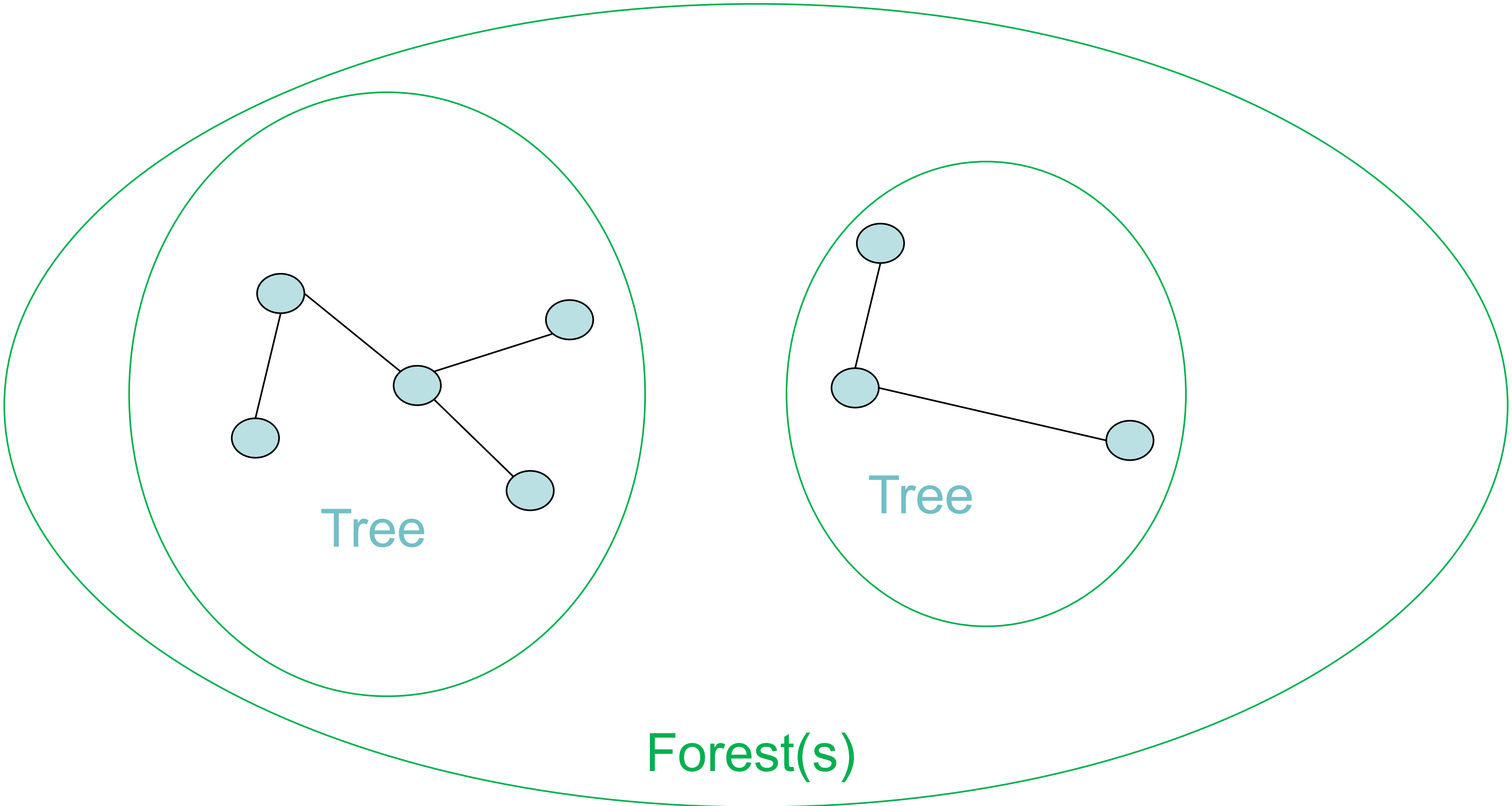
- A *subgraph* of a graph G=(V, E) is a graph G'=(V',E') where
  - $V' \subseteq V$
  - $E' \subseteq E$, and

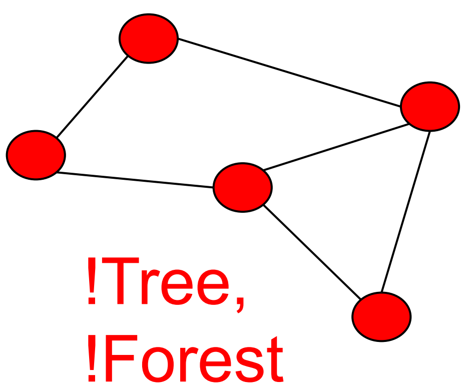
  (the vertices and edges in G' are subsets of the vertices and edges in G)
  - If $e \in E'$ where $e = \{u,v\}$, then $u, v \in V'$

  (every edge in G' has both of its ends in G')
- If E' contains *every* edge of E that has both ends in V', then G' is called the subgraph of G *induced by* V'
- If V' = V, then G' is called a *spanning subgraph of G*

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!





(All three "units" are connected Components)

Facts About Graphs

Thm: If G=(V,E) is a forest with |E| > 0, then G has at least one vertex v of degree 1 (a *leaf*)

- Let's prove this: Consider a longest simple path in G…

Thm: If G=(V,E) is a tree then |E| = |V| - 1.

- Hint: Induction on v: delete a leaf

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'`
- We will focus on edge-weights that are non-negative, so if `G'` is a subgraph of `G`, then `w(G') ≤ w(G)`

A Famous Problem: MCST

Given a connected, undirected graph G=(V,E) with non-negative edge weights, find a *minimum-weight*, *connected*, *spanning subgraph* of G.

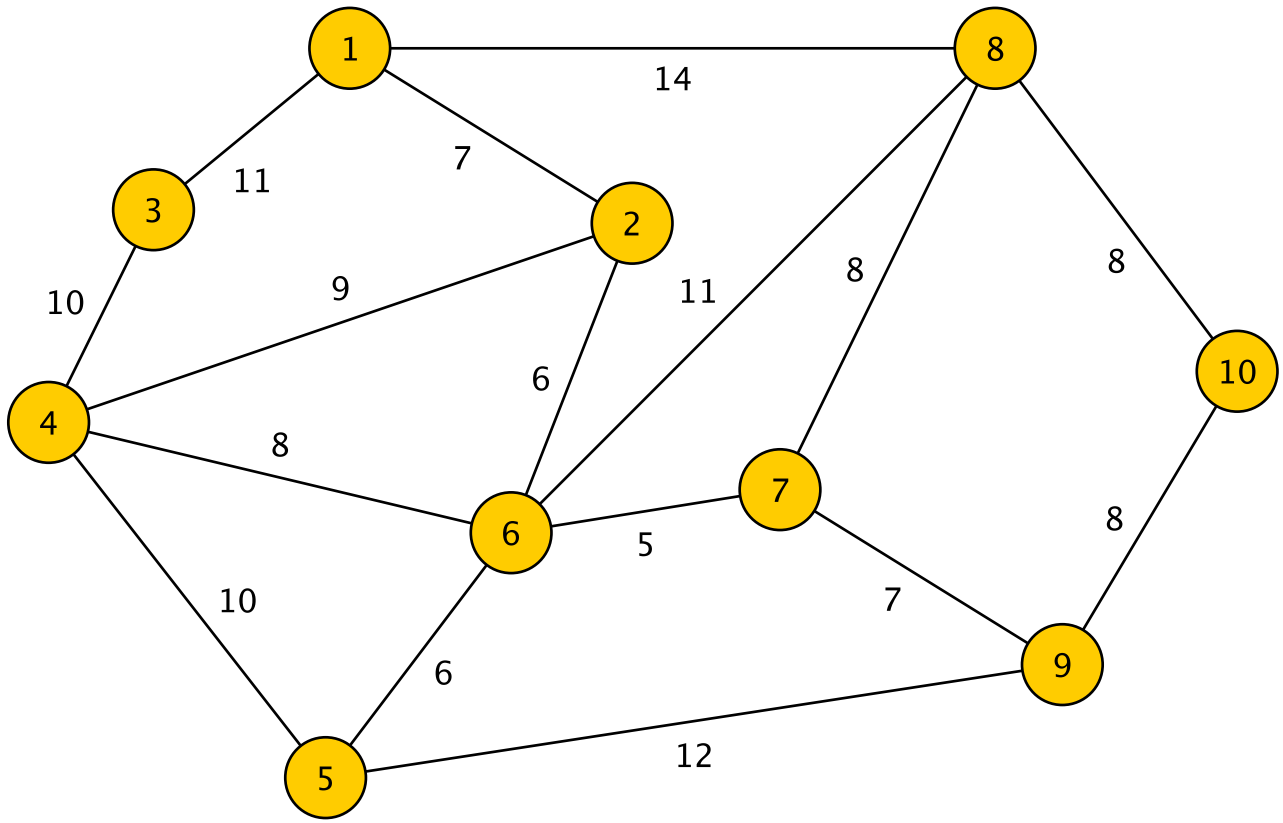
Note: Such a subgraph must be a spanning tree!

- If it weren't, we could remove an edge and reduce `w(G')`

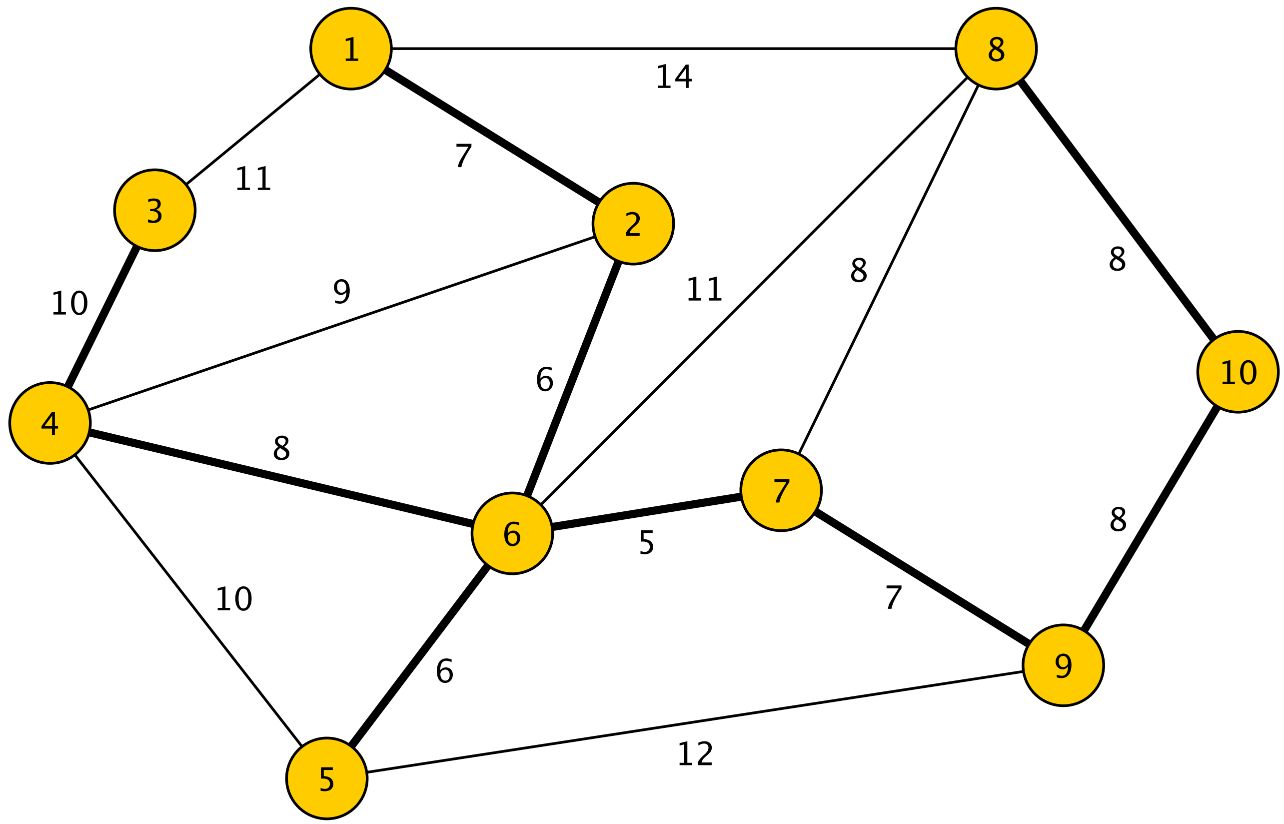
Frequently, we refer to the edge weights as *costs* and so this problem becomes:

> Given an undirected graph G with edge costs, compute a minimum-cost spanning tree of G.

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



MCST:

- fully connected (path from every vertex to ever other vertex)
- spanning subgraph (V' = V),
- tree (no cycles),
- the sum of the edge costs is minimal.

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 coloring

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* $V_2 \leftarrow V_1 - \{v\}$ // V_1 is v and V_2 is everything else

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

while$(|V_1| < |V|)$ // there are still vertices not in V_1

> *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
>
> *move u from* V_2 *to* V_1*;*
>
> *add to A all edges incident to u*
>
> *// note: A now may have edges with both ends in* V_1

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($|V_1| < |V|$) && $A \neq \varnothing$)
  - Replacing until e is an edge between V_1 and V_2 with
    - until $A \neq \varnothing$ 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 V1←{v} and V2←V1 - {v}
let A ←∅        // A will contain ALL edges between V1 and V2
while |V1|<|V| && |A| > 0
        add to A all edges incident to v
        repeat
                remove cheapest edge e from A
        until A is empty || e is an edge between V1 and V2
        if e is an edge between V1 and V2
                let v ← the vertex of e in V2
                move v from V2 to V1;
```

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

let $A \leftarrow \varnothing$ *// A will contain ALL edges between* V_1 *and* V_2

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

  add to A all edges incident to v

  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*;*

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|