Week 9 - Friday

COMP 2100

Last time

- What did we talk about last time?
- Graph representations
    - Adjacency matrix
    - Adjacency lists
- Depth-first search
- Breadth-first search

Questions?

Project 3

Assignment 4

Cycle Detection

Trees

- We have spent a huge amount of time on trees in this class
- Trees have many useful properties
- What is the important difference between a tree and a graph?
- **Cycles**
  - Well, technically a tree is also connected

When a tree falls in the woods…

- Is a graph a tree?
- It might be hard to tell
- We need to come up with an algorithm for detecting any cycles within the possible tree
- What can we use?
- **Depth First Search!**

Cycle detect pseudocode

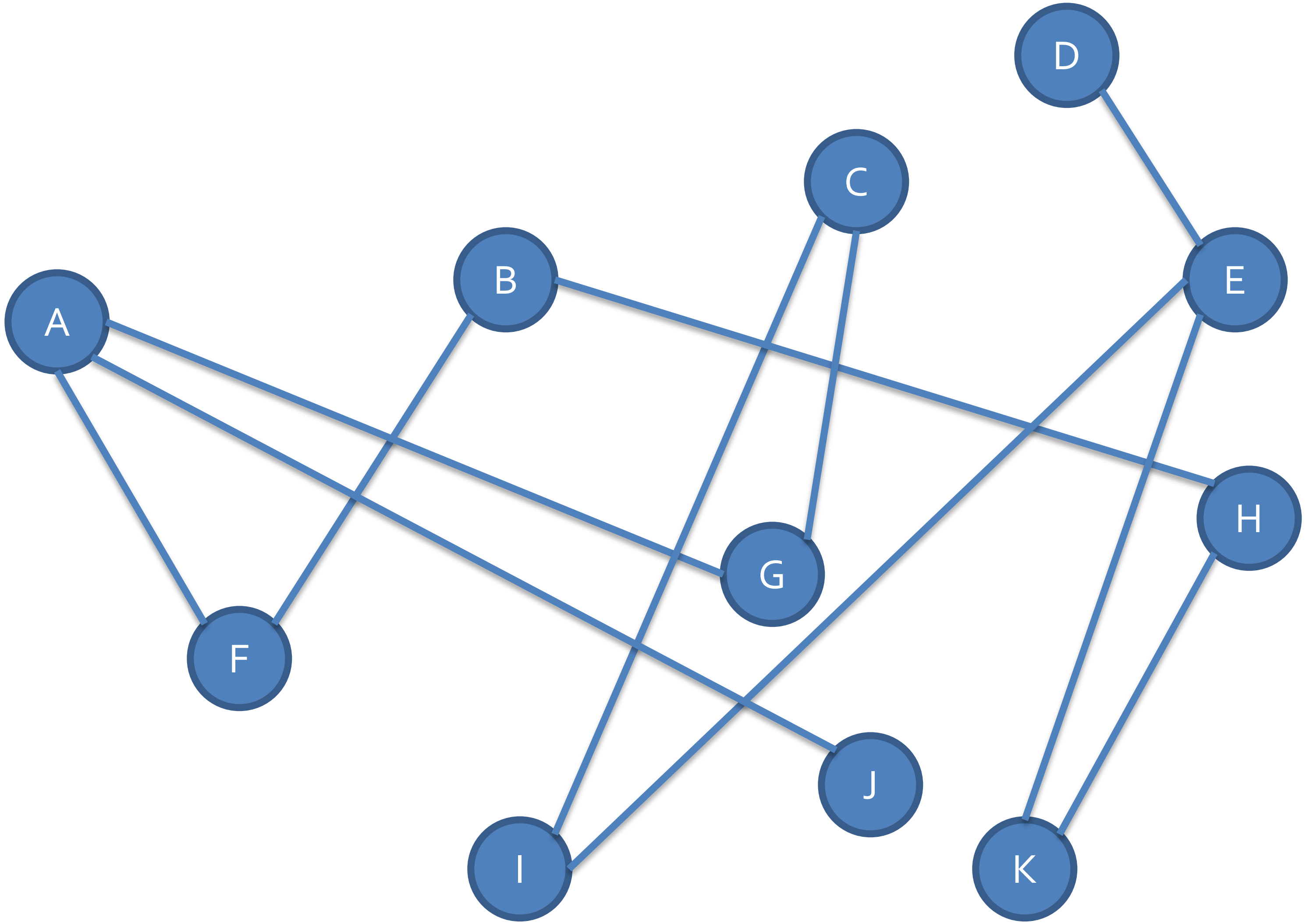
- Nodes all need some extra information, call it **number**
- Startup
  1. Set the number of all nodes to 0
  2. Pick an arbitrary node ***u*** and run Detect(***u***, 1)
- Detect(node ***v*,** int ***i***)
  1. Set number(***v***) = ***i***++
  2. For each node ***u*** adjacent to ***v***

     ```
     If number(u) is 0
         Detect( u, i )
     Else
         Print “Cycle found!”
     ```

Full cycle detection

- Even graphs with unconnected components can have cycles
- To be sure that there are no cycles, we need to run the algorithm on every starting node that hasn't been visited yet

Is there a cycle?



Topological Sort

Directed acyclic graph

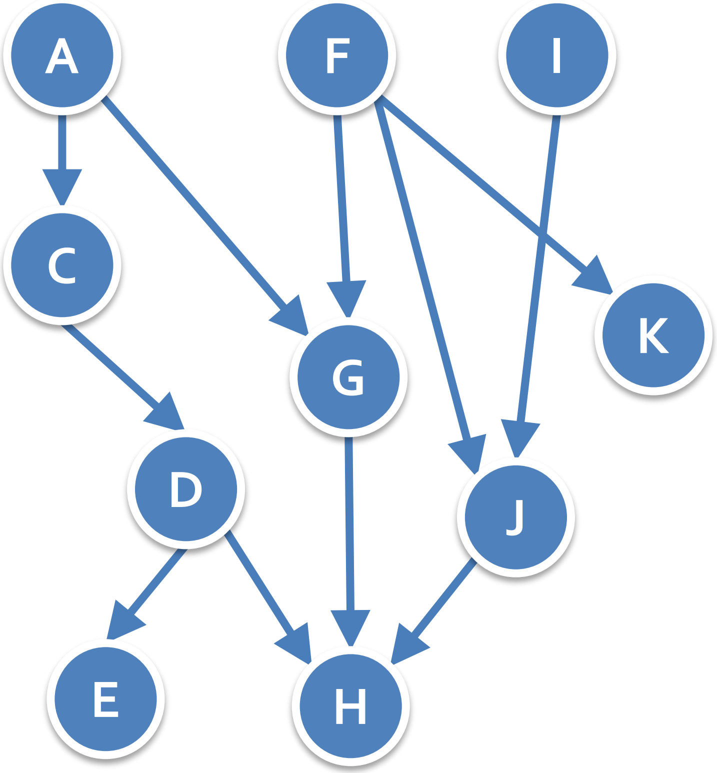
- A directed acyclic graph (DAG) is a directed graph without cycles in it
  - Well, obviously.
- These can be used to represent dependencies between tasks
- An edge flows from the task that must be completed first to a task that must come after
- This is a good model for course sequencing
  - Especially during advising
- A cycle in such a graph would mean there was a circular dependency
- By running topological sort, we discover if a directed graph has a cycle, as a side benefit

Topological sort

- A **topological sort** gives an ordering of the tasks such that all tasks are completed in dependency ordering
- In other words, no task is attempted before its prerequisite tasks have been done
- There are usually multiple legal topological sorts for a given DAG

Topological sort

- Give a topological sort for the following DAG:



- A F I C G K D J E H

Topological sort algorithm

- Create list ***L***
- Add all nodes with no incoming edges into set ***S***
- While ***S*** is not empty
  - Remove a node ***u*** from ***S***
  - Add ***u*** to ***L***
  - For each node ***v*** with an edge ***e*** from ***u*** to ***v***
    - Remove edge ***e*** from the graph
    - If ***v*** has no other incoming edges, add ***v*** to ***S***
- If the graph still has edges
  - Print "Error! Graph has a cycle"
- Otherwise
  - Return ***L***

Connectivity

Connected graph?

- **Connected for an undirected graph:**
  There is a path from every node to every other node
- How can we determine if a graph is connected?

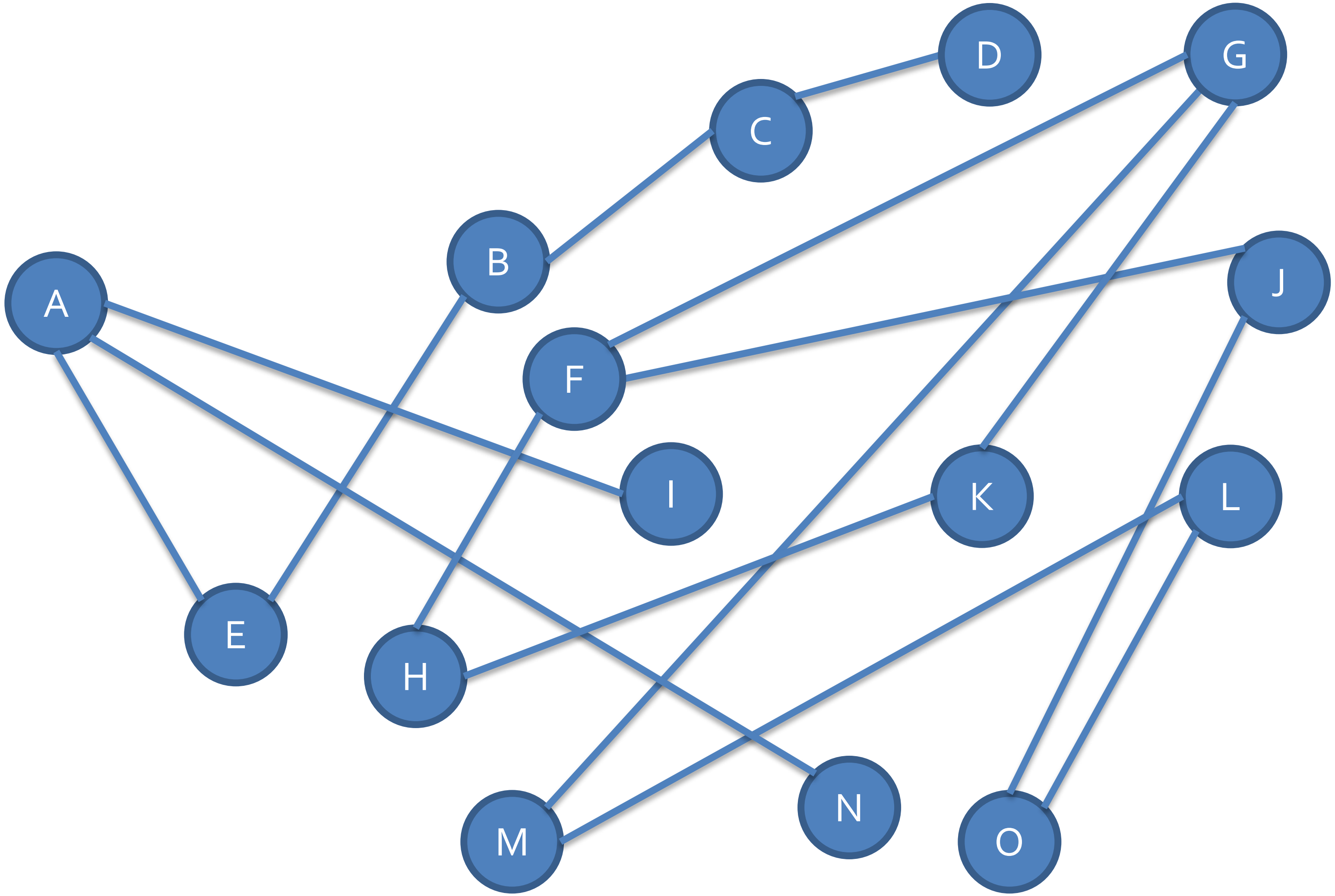
DFS to the rescue again!

- Startup
  1. Set the number of all nodes to 0
  2. Pick an arbitrary node ***u*** and run DFS(***u***, 1)
- DFS(node ***v***, int ***i***)
  1. Set number(***v***) = ***i***++
  2. For each node ***u*** adjacent to ***v***

     If number(***u***) is 0

       DFS(***u***, ***i***)
- If any node has a number of 0, the graph is not connected

Connected?



Connected components

- Connected components are the parts of the graph that are connected to each other
- In a connected graph, the whole graph forms a connected component
- In a graph that is not entirely connected, how do we find connected components?
- **DFS again!**
  - We run DFS on every unmarked node and mark all nodes with a number count
  - Each time DFS completes, we increment count and start DFS on the next unmarked node
  - All nodes with the same value are in a connected component

Directed connectivity

- **Weakly connected directed graph:**
  If the graph is connected when you make all the edges undirected
- **Strongly connected directed graph:**
  If for every pair of nodes, there is a path between them in both directions

Short of strong connectivity

- Components of a directed graph can be **strongly connected**
- A strongly connected component is a subgraph such that all its nodes are strongly connected
- To find strongly connected components, we can use a special DFS
- It includes the notion of a predecessor node, which is the lowest numbered node in the DFS that can reach a particular node
- There's an algorithm for it, but it's more complicated than we want to get into

Minimum Spanning Tree

What if…

- An airline has to stop running direct flights between some cities
- But, it still wants to be able to reach all the cities that it can now
- What’s the set of flights with the lowest total cost that will keep all cities connected?
- Essentially, what’s the lowest cost tree that keeps the graph connected?

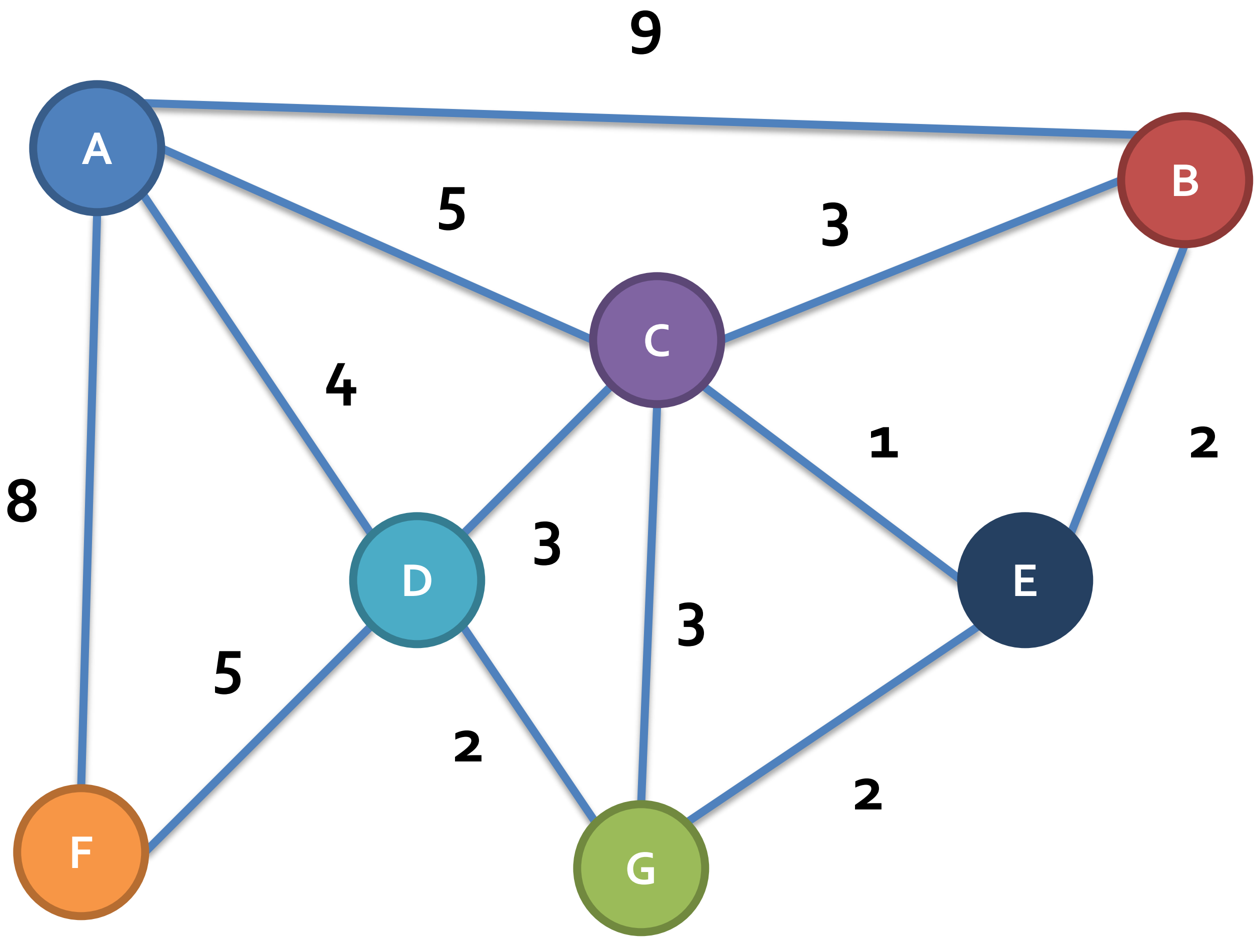
Minimum spanning tree

- This tree is called the **minimum spanning tree** or MST
- It has countless applications in graph problems
- How do we find such a thing?

Prim's Algorithm

1. Start with two sets, ***S*** and ***V***:
    - ***S*** has the starting node in it
    - ***V*** has everything else
2. Find the node ***u*** in ***V*** that is closest to any node in ***S***
3. Put the edge to ***u*** into the MST
4. Move ***u*** from ***V*** to ***S***
5. If ***V*** is not empty, go back to Step 2

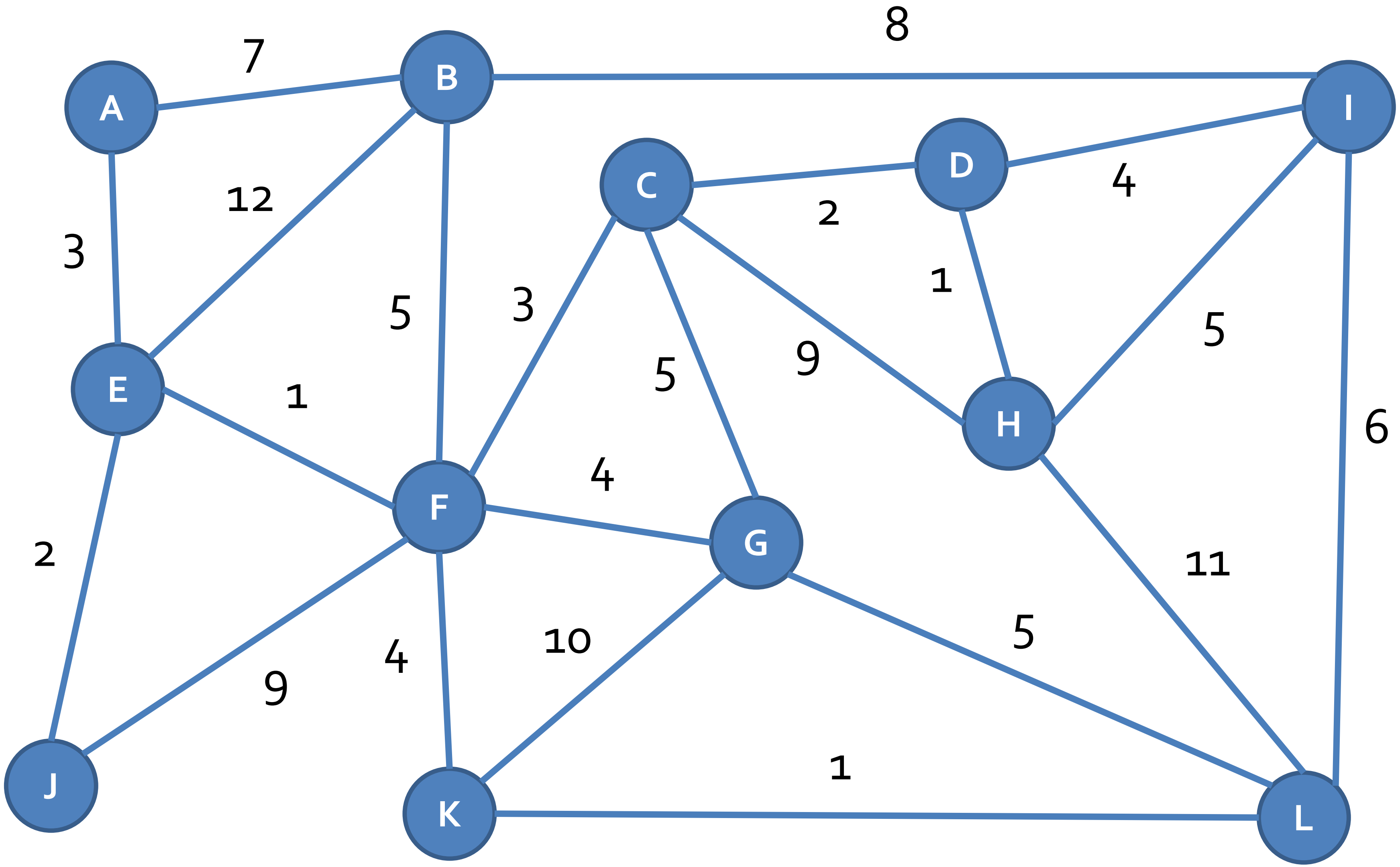
MST Example



Prim's algorithm running time

- Naïve implementation with an adjacency matrix
  - O($|V|^2$)
- Adjacency lists with binary heap
  - O($|E| \log |V|$)
- Adjacency lists with Fibonacci heap
  - O($|E| + |V| \log |V|$)

MST practice



Upcoming

Next time…

- Dijkstra's algorithm
- Matching
- Stable marriage
- Euler paths and tours

Reminders

- Because of the AI Task Force, I won't have my normal 1:45-2:45 office hours today
  - **But! I will available from 1-1:45 instead**
- Keep working on Project 3
- Finish Assignment 4
  - **Due tonight!**
- Read sections 6.2 and 6.4