Week 9 - Monday

COMP 2100

Last time

- What did we talk about last time?
- (Chaining) hash table implementation
- Maps and sets in the JCF

Questions?

Project 3

Assignment 4

Maps in the Java Collections Framework

JCF Map

- The Java interface for maps is, unsurprisingly, **`Map<K,V>`**
  - **`K`** is the type of the key
  - **`V`** is the type of the value
  - Yes, it's a container with **two** generic types
- Any Java class that implements this interface can do the important things that you need for a map
  - **`get(Object key)`**
  - **`containsKey(Object key)`**
  - **`put(K key, V value)`**

JCF implementation

- Because the Java gods love us, they provided two main implementations of the **`Map`** interface
- **`HashMap<K,V>`**
  - **Hash table** implementation
  - To be useful, type **`K`** must have a meaningful **`hashCode()`** method
- **`TreeMap<K,V>`**
  - **Balanced binary search tree** implementation
  - To work, type **`K`** must implement the **`compareTo()`** method
  - Or you can supply a comparator when you create the **`TreeMap`**

Code example

- Let's see some code to keep track of some people's favorite numbers

```
Map<String,Integer> favorites = new TreeMap<>();

favorites.put("John", 42); // Autoboxes int value
favorites.put("Paul", 101);
favorites.put("George", 13);
favorites.put("Ringo", 7);
if (favorites.containsKey("George"))
     System.out.println(favorites.get("George"));
```

JCF Set

- Java also provides an interface for sets
- A set is like a map without values (only keys)
- All we care about is storing an unordered collection of things
- The Java interface for sets is **`Set<E>`**
  - **`E`** is the type of objects being stored
- Any Java class that implements this interface can do the important things that you need for a set
  - **`add(E element)`**
  - **`contains(Object object)`**

Time trials

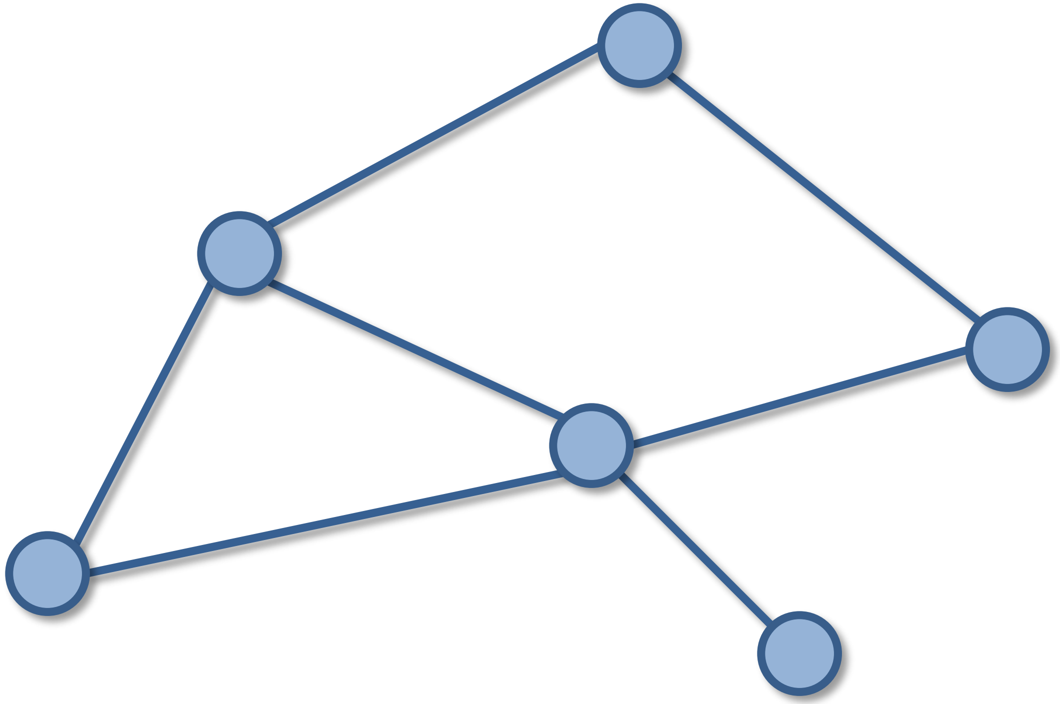
- Let's compare the speed of a tree with the speed of a hash table
  - We can generate 1,000,000 random numbers
  - We can add this list of numbers to a `TreeSet` and to a `HashSet`
  - Then, we can test each one to see if other random numbers can be found inside

Graphs

Definitions

What is a graph?

- Vertices (Nodes)
- Edges

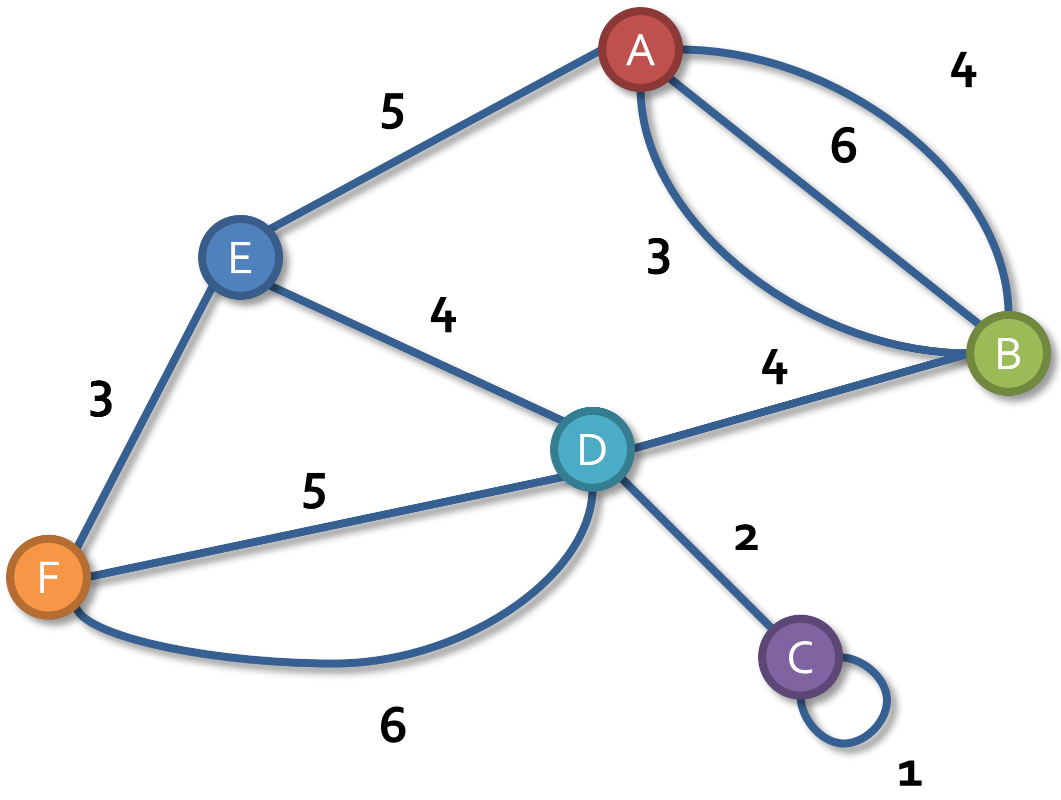


Adjacency

- If two nodes are connected by an edge, they are **adjacent**
- The number of nodes adjacent to a particular node is called its **degree**

Lots of flavors of graphs

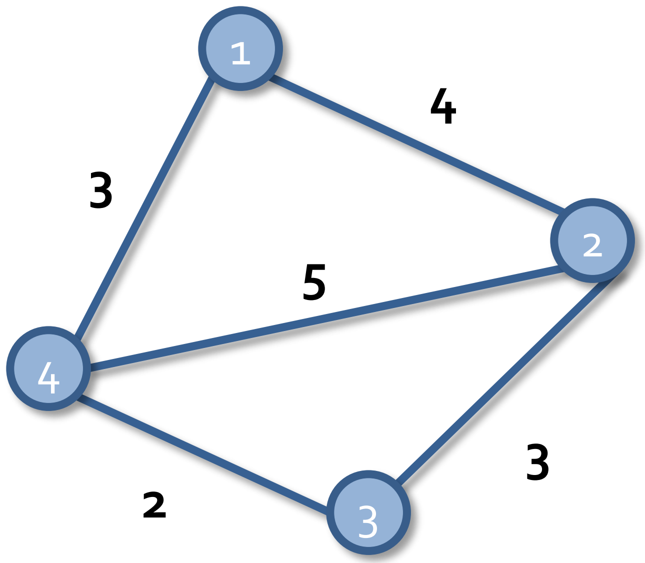
- Labeled
- **Weighted**
- Colored
- Multigraphs



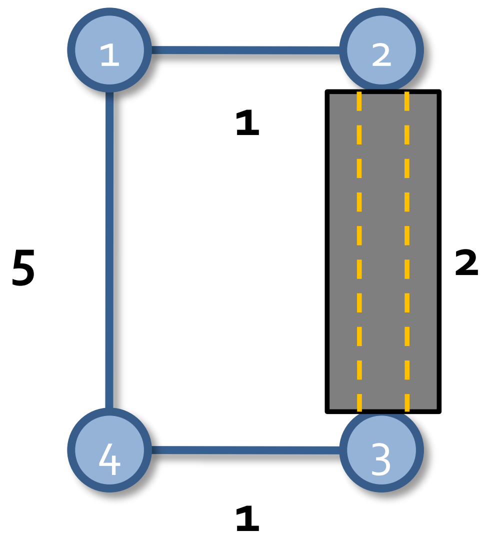
Triangle inequality

- When a weighted graph obeys the triangle inequality, the direct route to a node is always fastest

Triangle Inequality

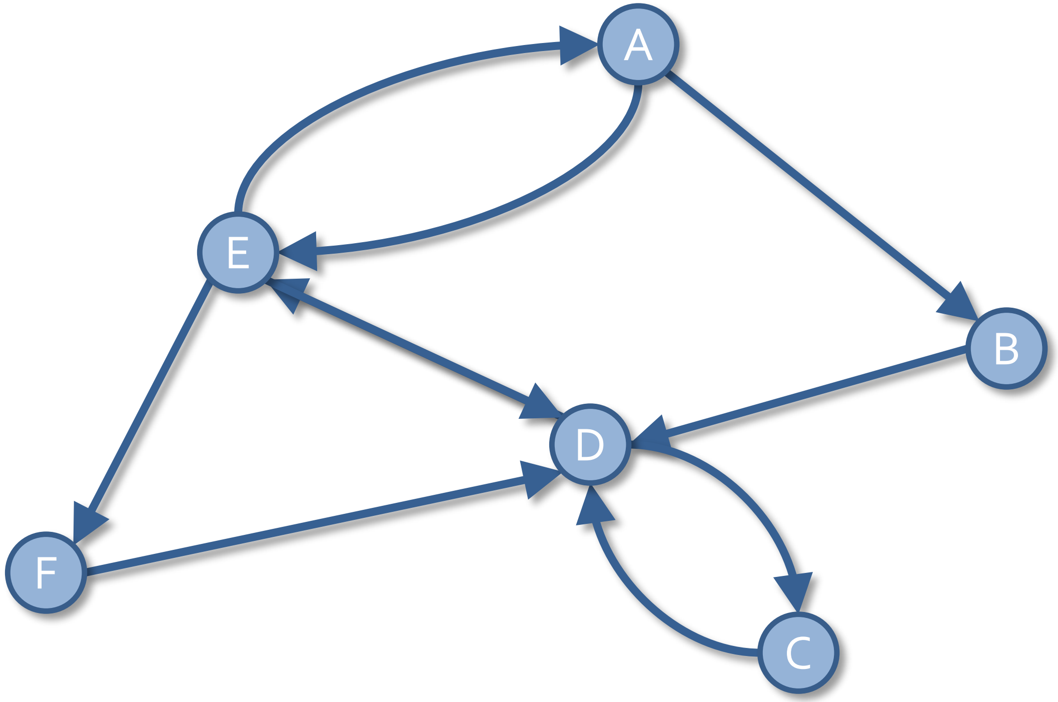


No Triangle Inequality



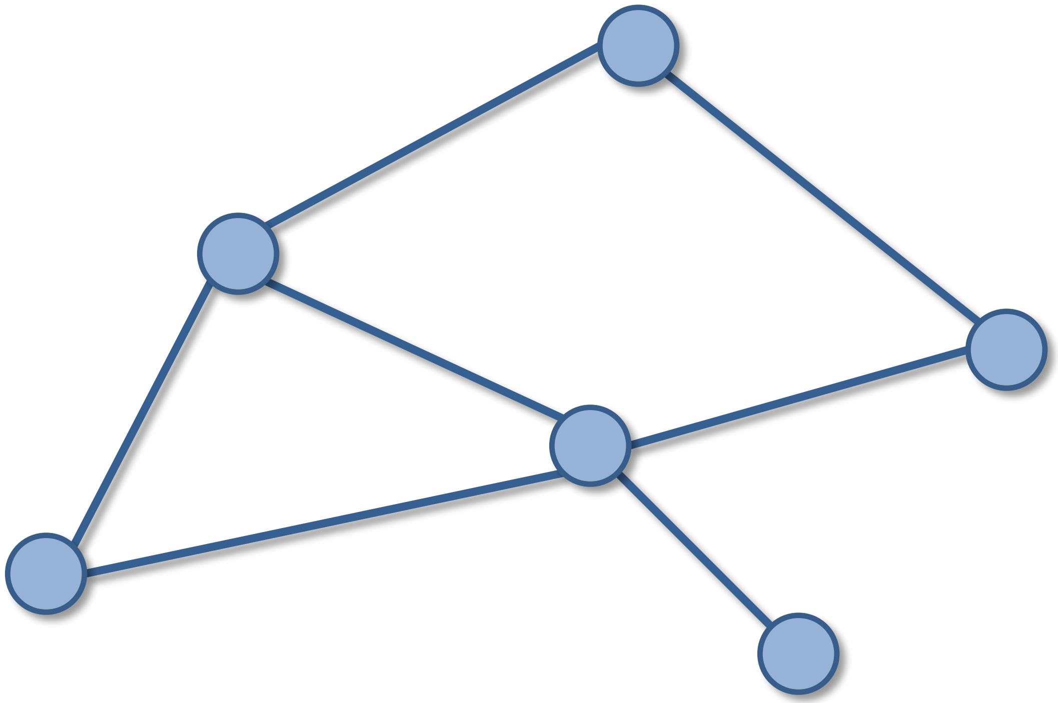
Directed graphs

- Some graphs have edges with direction
- Example: One way streets
- Reachability?



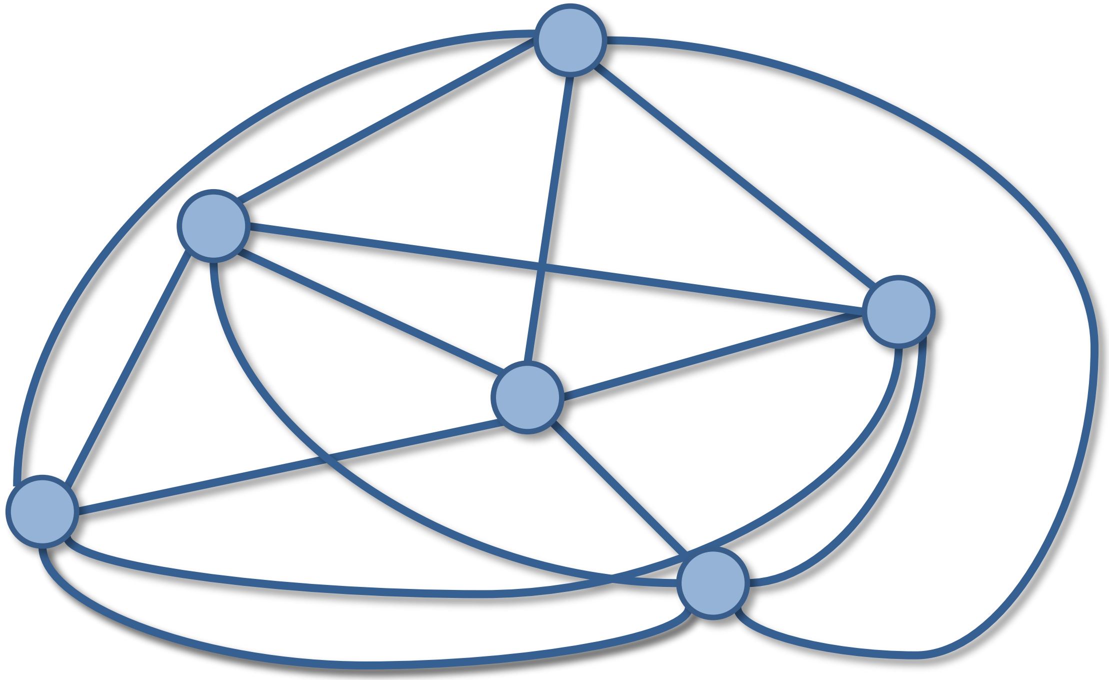
Connected graphs

- Often we talk about connected graphs
- But, not all graphs have to be connected



The other extreme

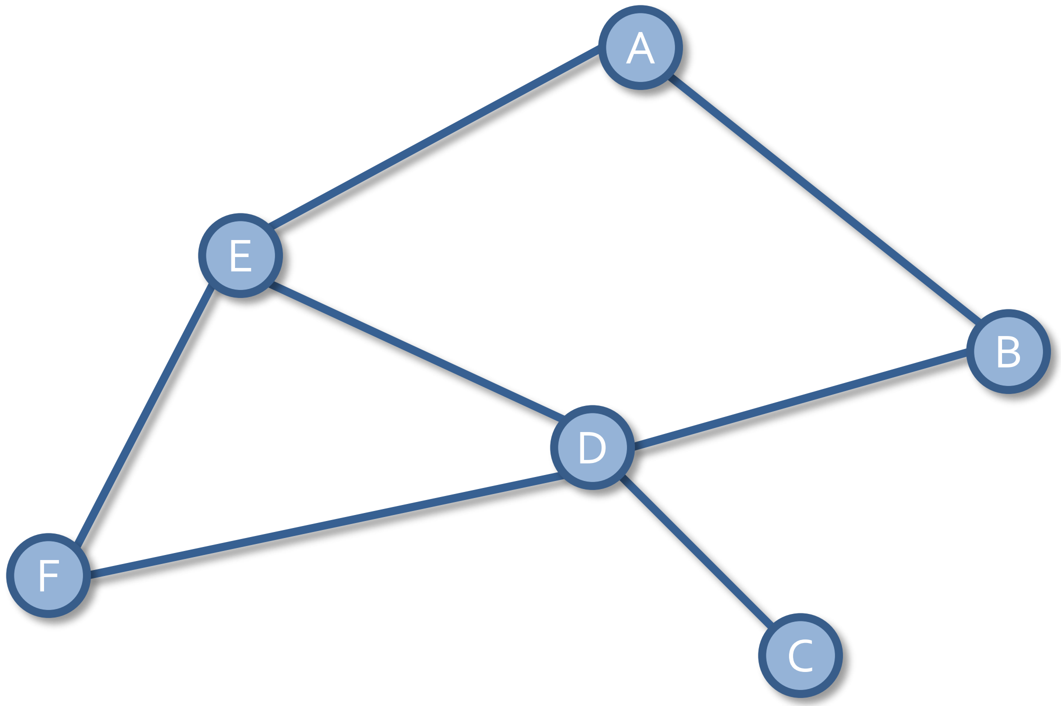
- Complete graphs
- Every node is connected to every other
- How many edges in a complete graph with ***n*** nodes?



- $|E| = \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$ is $O(n^2)$

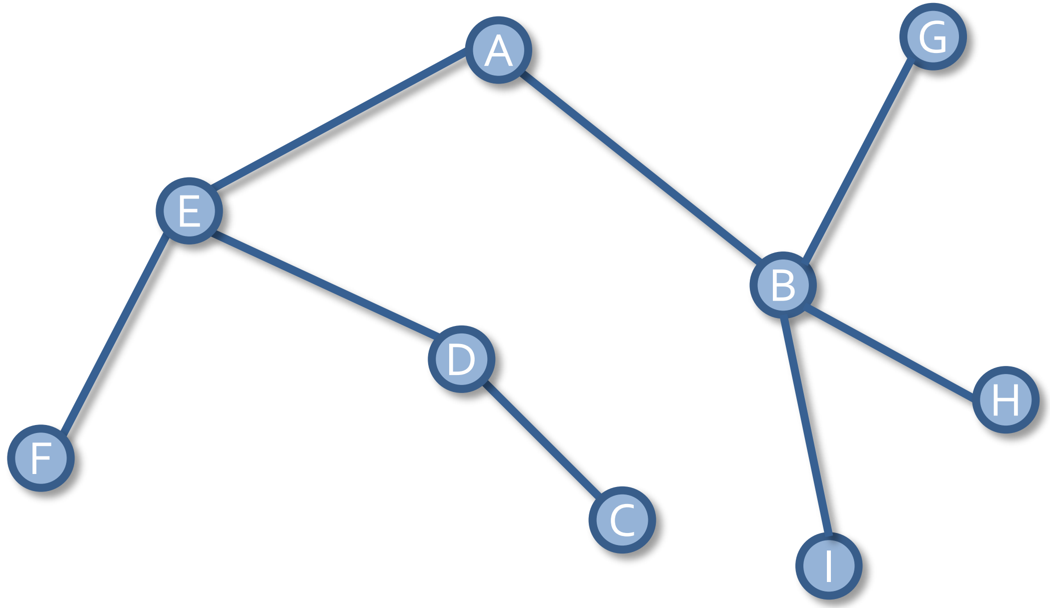
Subgraphs

- We can talk about a part of a graph
- For example, what is the largest complete subgraph in this graph?



Trees

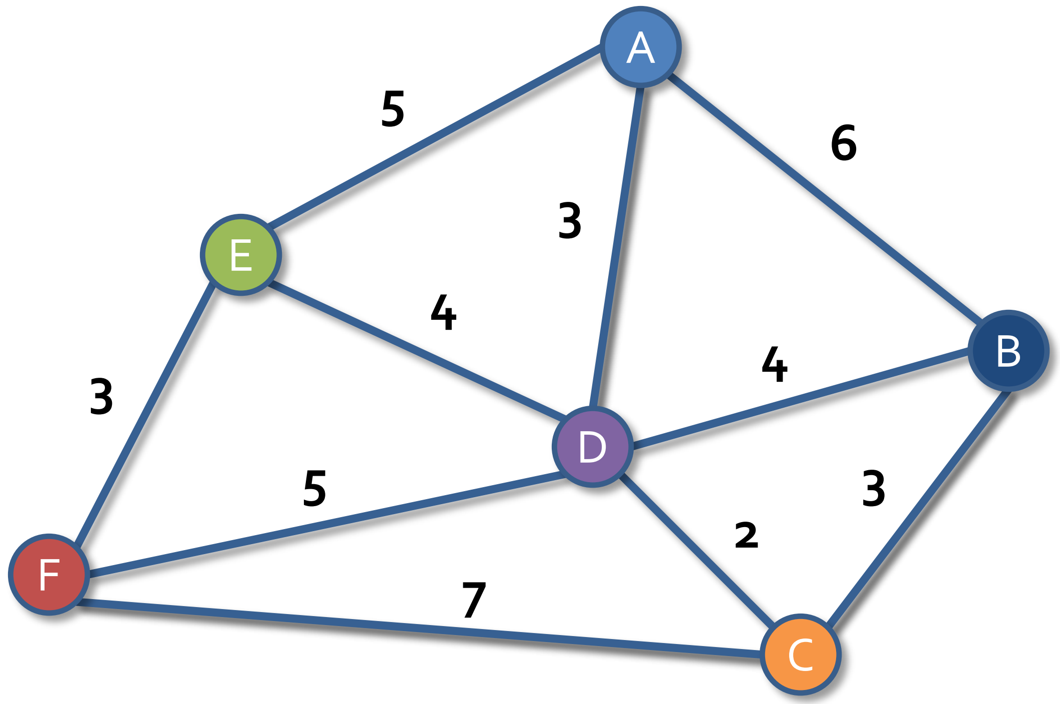
- A **tree** (in the graph sense) is a connected acyclic graph



- A tree does not have to have a root (unlike tree data structures)
- A tree with ***n*** nodes will always have ***n*** – 1 edges

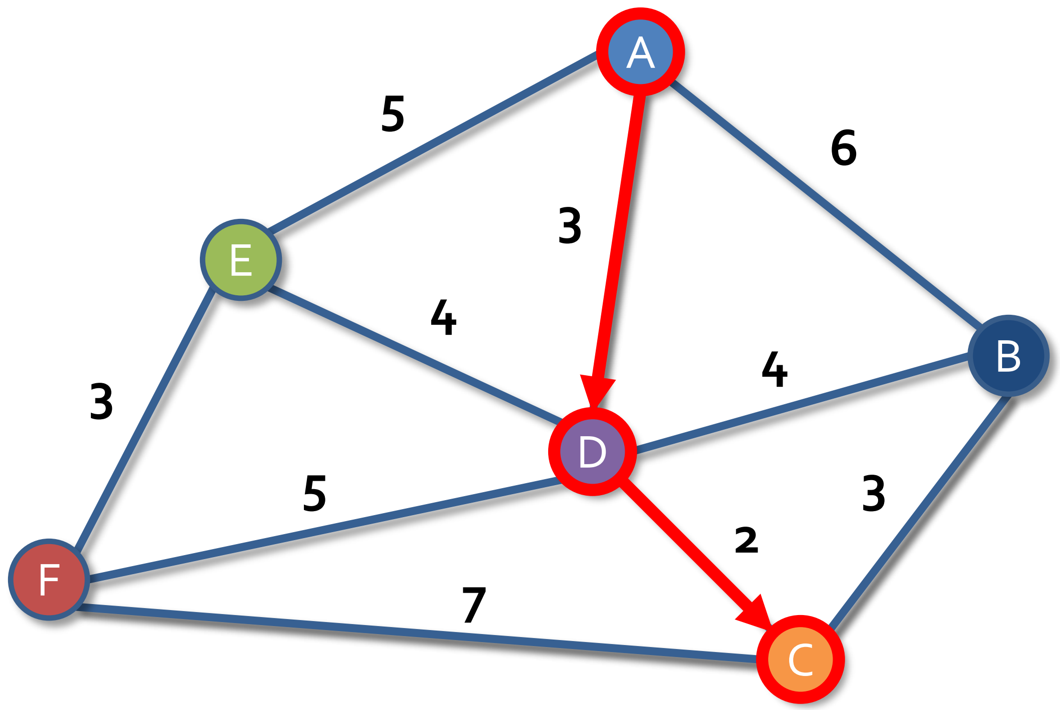
Paths

- A **path** is a sequence of nodes connected by edges
- A **simple path** has no repeated nodes
- A **cycle** is a path with at least one edge whose first and last node are the same
- A **simple cycle** is a cycle with no repeated edges or nodes (except the first and the last)



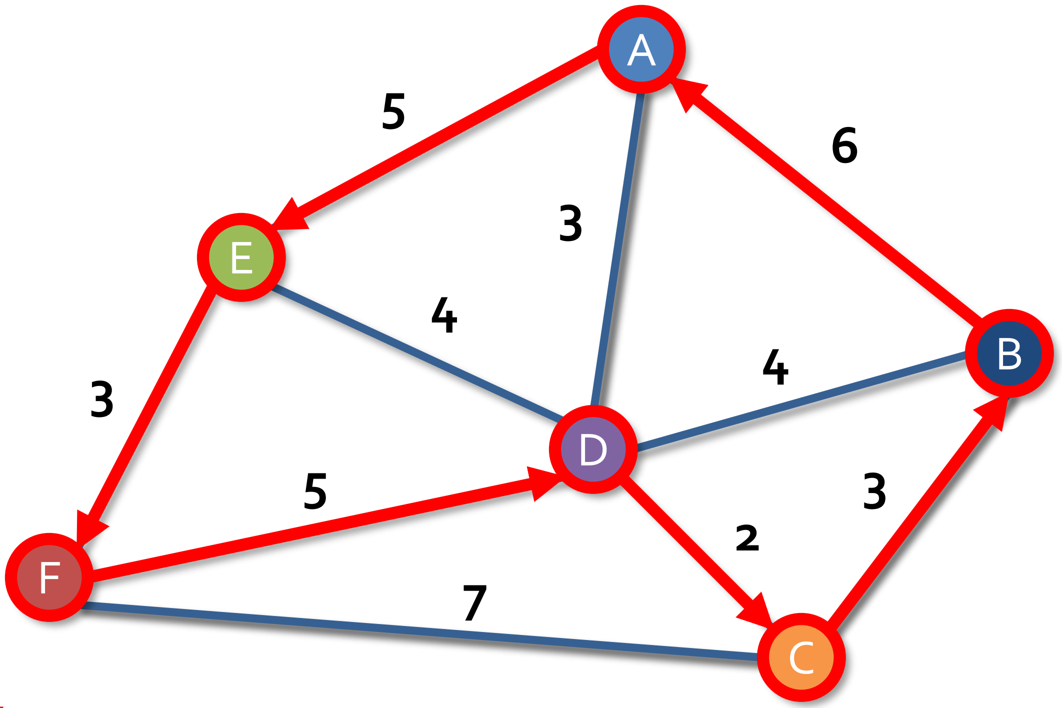
Weighted paths

- Many practical problems look at graphs with weighted edges
- The cost or weight of the path is usually the sum of the edge weights
- This path from A to C costs 5



Tours

- A tour is a path that visits every node and (usually) returns to its starting node



- This tour costs 24

Undirected Graph ADT

- A graph is more abstract than a stack or a queue
  - But we can still think of some general operations we need
- V()
  - Get the number of nodes (vertices)
- E()
  - Get the number of edges
- addEdge(*v*, *w*)
  - Add an edge between node *v* and node *w*
- adjacent(*v*)
  - Get a list of nodes adjacent to *v*

The purpose of graphs

- A graph is generally **not** like a list or a symbol table
- We usually don't want to keep adding and removing data from the graph
- Instead, a graph is a set of relationships
- We want to look at a (usually unchanging) graph and determine various properties of it
- We usually don't care about the efficiency of adding or removing nodes

Implementing the graph ADT

- The book mentions four implementations:
  - **Adjacency matrix**
  - Array of edges
  - **Adjacency lists**
  - Adjacency sets
- We will talk about adjacency matrices and adjacency lists

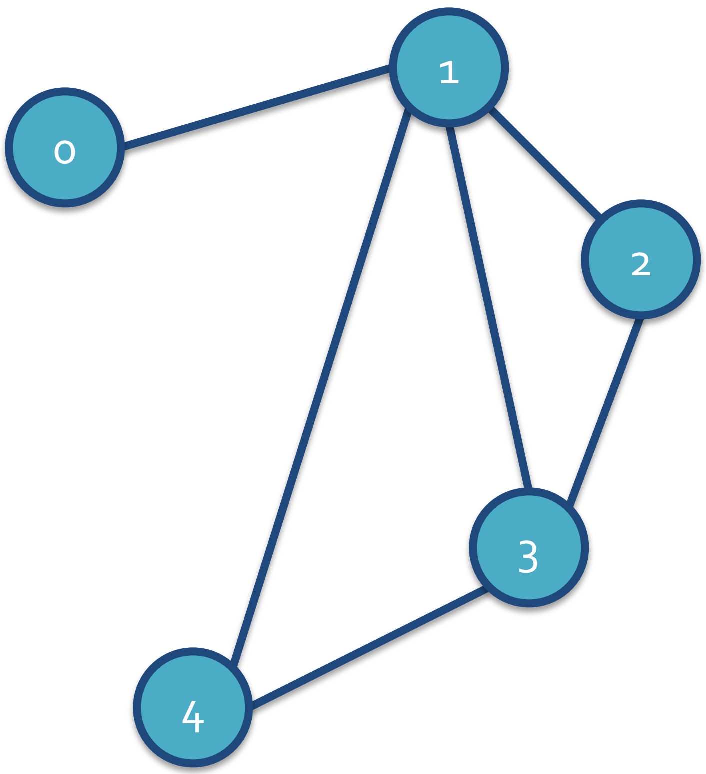
Implementing the graph ADT

- The book mentions four implementations:
  - **Adjacency matrix**
  - Array of edges
  - **Adjacency lists**
  - Adjacency sets
- We will talk about adjacency matrices and adjacency lists

Adjacency matrix

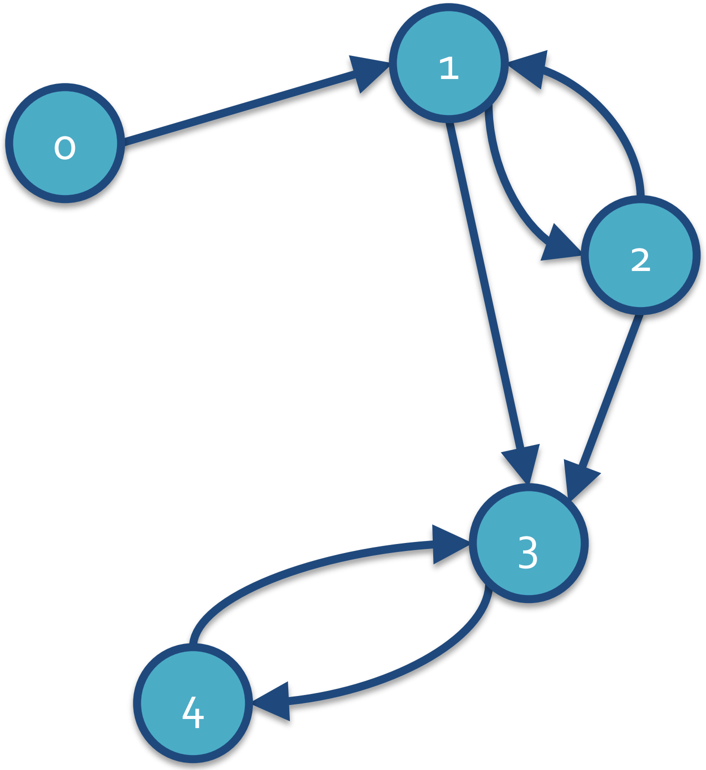
- A simple way of keeping track of the edges in a graph is an **adjacency matrix**
- An adjacency matrix is an n x n matrix where n is the number of nodes
- The number in row i column j is the number of edges between node i and node j
- Undirected graphs have symmetrical adjacency matrices
- The weakness of an adjacency matrix is that it uses $\Theta(n^2)$ space, even for sparse graphs

Adjacency matrix example



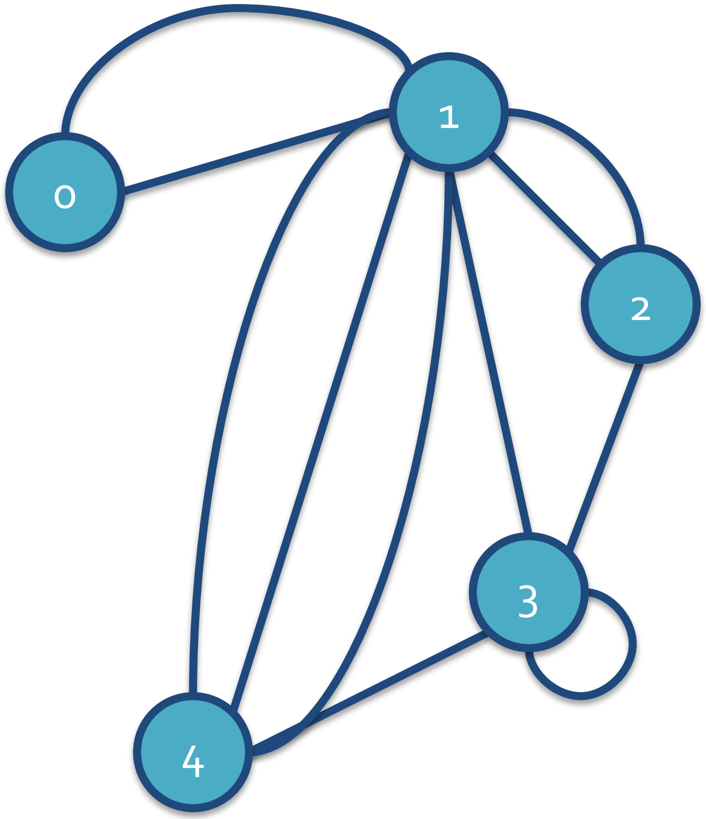
	0	1	2	3	4
0	0	1	0	0	0
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	0	1	0	1	0

Directed graph example



	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	0	0	0	1
4	0	0	0	1	0

Multigraph example



	0	1	2	3	4
0	0	2	0	0	0
1	2	0	2	1	3
2	0	2	0	1	0
3	0	1	1	1	1
4	0	3	0	1	0

Upcoming

Next time…

- Finish representations
- Depth first search
- Breadth first search
- Topological sort
- Connectivity

Reminders

- Keep working on Project 3
- Keep working on Assignment 4
  - Due Friday!
- Read 4.2 and 4.3