**Announcements**

- Prelab 9 is due now
- Test #2 graded & returned today
  - Generally very well done. Good job!
  - Test #2 graded & returned today

---

**Graph Implementations**

**Graph Definitions**

- A graph $G = (V, E)$ is a set of vertices $V$ and a set of edges $E$.
- Directed graph has directed edges $e = (u, v)$.
- Undirected graphs have unordered edges $e = (u, v)$.

**Graph Operations**

- $size()$: return the number of nodes in the graph
- $isEmpty()$: return whether there are 0 nodes in the graph
- $isEdge(Vertex v, Vertex w)$: return true if $(v, w)$ is an edge
- $areConnected(Vertex v, Vertex w)$: return whether $v, w$ are connected
- $shortestPath(Vertex v, Vertex w)$: return length of the shortest $v$-$w$ path
- $isConnected()$: return whether graph is one connected component
- $numComponents()$: return the number of connected components
- $findConnComponent(Vertex v)$: return vertices in $v$'s connected component
- $getAvgDegree()$: determine the average degree of all vertices
- $findSpanningTree()$: find the fewest number of edges of $G$ to keep so that $G$ is still connected

---

**Graph Operations**

- $traverse()$: traverse through nodes in an ordered fashion
- $isStronglyConnected()$: determine the strongest degree of all vertices
- $findShortestPath()$: find the shortest $v$-$w$ path

**Summary of Definitions**

- A graph is connected if there is a path between every pair of vertices.
- A graph is strongly connected if there is a directed path between every pair of vertices.
- A graph is weakly connected if there is a directed path between every pair of vertices.
- The degree of a node is the number of edges incident to the node.
- The neighborhood of a node is a set of the nodes adjacent to the node.
- A connected component of a graph is a subset of the nodes such that:
  - Every node in the subset has a path to every other node in the subset.
  - The subset is not part of some larger set with the connected property.

---

**Graph Summary**

- A graph is connected.
- A graph is strongly connected if there is a directed path between every pair of vertices.
- The degree of a node is the number of edges incident to the node.
- The neighborhood of a node is a set of the nodes adjacent to the node.
- A connected component of a graph is a subset of the nodes such that:
  - Every node in the subset has a path to every other node in the subset.
  - The subset is not part of some larger set with the connected property.
- A graph is connected if there is a path between every pair of vertices.
- A graph is strongly connected if there is a directed path between every pair of vertices.
- A graph is weakly connected if there is a directed path between every pair of vertices.
- The degree of a node is the number of edges incident to the node.
- The neighborhood of a node is a set of the nodes adjacent to the node.
- A connected component of a graph is a subset of the nodes such that:
Facts About Graphs

Proof. Every edge \( e = (u, v) \) in \( G \) is counted twice in the sum of the degrees.

Fact 1: For any graph \( G = (V, E) \),

\[
\sum_{e \in E} \deg(e) = \sum_{v \in V} \deg(v) = 2|E|.
\]

Fact 2: In a directed graph,

\[
\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v)
\]

Proof. Every edge \( e = (u, v) \) adds 1 to in-degree sum and 1 to out-degree sum.

Fact 3: In a graph \( G \) with \( n \) nodes and \( m \) edges \((n = |V|, m = |E|)\),

\[
m \in \Theta(n^2)
\]

Proof. At most you can have 1 edge between every pair of vertices.

There are \( \binom{n}{2} \) ways to pick distinct pairs of vertices out of \( n \) of them.

Thus, \( \binom{n}{2} \in \Theta(n^2) \).

A graph is

**dense**

if

\[
m \in \Theta(n^2)
\]

and

**sparse**

if

\[
m \in O(n)
\]

Graph Implementations

1. adjacency matrix

- A graph is an \( n \times n \) matrix (2D array) of edges
- Or, if you have edge costs:

\[
e[u][v] = \begin{cases} c & \text{if } (u, v) \in E \\ \infty & \text{if } (u, v) \not\in E \end{cases}
\]

2. adjacency list

- A graph is a list of vertices
- Each vertex keeps track of its incident edges (equivalently, its adjacent nodes)

### Graph class with adjacency matrix:

- \( \text{int[][] edges} \)
- \( \text{int numNodes} \)

- \( \text{int size() -- return numNodes (or edges.length)} \)
  - RT: \( O(1) \)
- \( \text{bool isEmpty() -- return size() == 0} \)
  - RT: \( O(1) \)
- \( \text{bool isEdge(Vertex u, Vertex v)} \)
  - index\text{U}, index\text{V} = u and v’s index in graph, resp.
  - return ( edges\[index\text{U}\][index\text{V}] != 0 )
  - RT: \( O(n) \)
  + time to find \( u \)’s index
- \( \text{int getDegree(Vertex u)} \)
  - index\text{U} = u’s index in the graph
  - deg\text{U} = sum of num. of non-zero entries in index\text{U} row
  - return deg\text{U}
  - RT: \( O(n) \)
  + time to find \( u \)’s index
- \( \text{bool addEdge(Vertex u, Vertex v, [int cost])} \)
  - index\text{U}, index\text{V} = u and v’s index in graph, resp.
  - set edges\[index\text{U}\][index\text{V}] = 1 or cost
  - if graph undir, set edges\[index\text{V}\][index\text{U}] = 1 or cost
  - RT: \( O(1) \)
  + time to find \( u \)’s index
  + time to find \( u \)’s index

### Graph class with adjacency list:

- A graph is a list of vertices
- Each vertex keeps track of its incident edges (equiv., its adjacent nodes)

\[
\begin{array}{c|c}
\text{A} & \{A, B\} \\
\hline
\text{B} & \{A, B\} \\
\text{C} & \{B, C\} \\
\text{D} & \{B, D\} \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{A} & \{A, B\} \\
\hline
\text{B} & \{B, C\} \\
\text{C} & \{C, D\} \\
\text{D} & \{D, B\} \\
\end{array}
\]

### Graph class with adjacency matrix:

**Graph class with adjacency matrix:**

\[
\begin{array}{cccc}
A & B & C & D \\
\hline
A & 0 & 1 & 0 & 0 \\
B & 1 & 0 & 1 & 1 \\
C & 0 & 1 & 0 & 1 \\
D & 0 & 1 & 1 & 0 \\
\end{array}
\]

**Graph class with adjacency list:**

\[
\begin{array}{c|c}
\text{A} & \{A, B\} \\
\hline
\text{B} & \{B, C\} \\
\text{C} & \{C, D\} \\
\text{D} & \{D, B\} \\
\end{array}
\]

**Graph class with adjacency matrix:**

\[
\begin{array}{cccc}
A & B & C & D \\
\hline
A & 0 & 1 & 0 & 0 \\
B & 1 & 0 & 1 & 1 \\
C & 0 & 1 & 0 & 1 \\
D & 0 & 1 & 1 & 0 \\
\end{array}
\]

**Graph class with adjacency list:**

\[
\begin{array}{c|c}
\text{A} & \{A, B\} \\
\hline
\text{B} & \{B, C\} \\
\text{C} & \{C, D\} \\
\text{D} & \{D, B\} \\
\end{array}
\]

**Graph class with adjacency matrix:**

\[
\begin{array}{cccc}
A & B & C & D \\
\hline
A & 0 & 1 & 0 & 0 \\
B & 1 & 0 & 1 & 1 \\
C & 0 & 1 & 0 & 1 \\
D & 0 & 1 & 1 & 0 \\
\end{array}
\]

**Graph class with adjacency list:**

\[
\begin{array}{c|c}
\text{A} & \{A, B\} \\
\hline
\text{B} & \{B, C\} \\
\text{C} & \{C, D\} \\
\text{D} & \{D, B\} \\
\end{array}
\]
Graph Implementations

**Graph class with adjacency list:**

- `Vertex` class:
  - `LL<Edge>` adj
  - `String` name

- `Edge` class:
  - `Vertex` start
  - `Vertex` dest
  - `int`/`double` cost

  - `Edge( Vertex dest, int cost )`
   - `this.dest = dest`
   - `this.cost = cost`

  - `boolean equals( Object o )`
   - `return ((Edge)o).dest.equals( this.dest )`

**Graph with adjacency list:**

- `Graph` class:
  - `LL<Vertex>` nodes
  - `int` numNodes

- `Vertex` class:
  - `LL<Edge>` adj
  - `String` name

  // may add more class members such as prev, dist, scratch

  - `Vertex( String name )`
   - `this.name = name`

  - `int degree()`
   - `return adj.size()`

  - `addEdge( Vertex w, int cost )`
   - `adj.add( new Edge( w, cost ) )`

  - `boolean equals( Object o )`
   - `return ((Vertex)o).name.equals( this.name )`

  - `boolean isNbr( Vertex w )`
   - `return adj.contains( new Edge( w ) )`

  - `LL<Vertex> nbrs()`
   - `for each e in adj, add e.dest to new LL and return it`

**Graph with adjacency list:**

- `Graph` class:
  - `HashMap<String,Vertex> vMap`
  - `int` numNodes

  - `size()` -- return numNodes (or vMap.size())
  - `isEmpty()` -- return (size() == 0)

  - `private Vertex getVertex( String vName )`
   - `Vertex v = vMap.get( vName )`
   - `if( v == null ) vMap.put( vName, new Vertex( vName ) )`
   - `return v`

  - `getDegree( String vName )`
   - `Vertex v = vMap.get( vName )`
   - `if( v == null ) throw exception`
   - `return v.degree()`

• but user doesn’t usually refer to vertices as Objects—they prefer names
• how can we store vertices by their names & still have quick access?
  Use a HashMap of <String,Vertex> pairs!
Graph Implementations

### Graph with adjacency list:

#### Graph class:

- `HashMap<String, Vertex> vMap`
- `int numNodes`

#### Method:

- `addEdge(String uName, String vName, int cost)`
  - `Vertex u = getVertex(uName)`
  - `Vertex v = getVertex(vName)`
  - `u.addEdge(v, cost)`
  - **RT:** O(1)

- `isEdge(String uName, String vName)`
  - `Vertex u = vMap.get(uName)`
  - `Vertex v = vMap.get(vName)`
  - `return (u.isNbr(v))`  
  - **RT:** O(n)

---

**Graph class:**

```java
Graph() {
    this.numNodes = 0;
    this.vMap = new HashMap<String, Vertex>();
}
```

**addEdge(String uName, String vName, int cost):**

```java
addEdge(String uName, String vName, int cost) {
    Vertex u = getVertex(uName);
    Vertex v = getVertex(vName);
    if (u != null && v != null) {
        // Add edge
        u.addEdge(v, cost);
        v.addEdge(u, cost);
    }
}
```

**isEdge(String uName, String vName):**

```java
isEdge(String uName, String vName) {
    Vertex u = vMap.get(uName);
    Vertex v = vMap.get(vName);
    return (u.isNbr(v));
}
```