CS 151

Graph Implementations
Announcements

Prelab 9 is due now
Test #2 graded & returned today
• generally very well done, good job!
A graph $G=(V,E)$ is a set of vertices $V$ and set of edges $E$.
- directed graph has directed edges $e=(u,v)$ that are ordered pairs of $V$
- undirected graphs have undirected edges $e=\{u,v\}$ of unordered pairs of $V$

Two nodes are neighbours or adjacent if they are connected by an edge.

A path is a sequence of nodes with the property that each consecutive pair in the sequence is connected by an edge. A simple path is a path that does not repeat nodes. A length of a path is the number of edges in that path.

A graph is connected if there is a path between every pair of nodes. A directed graph is strongly connected if there is a directed path between every pair of nodes, and weakly connected if the underlying undirected graph is connected.

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, and
- the subset is not part of some larger set with the connected property.

The degree of a node is the number of edges incident to the node.
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 len 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 conn’d component
- `getAvgDegree()` - determine the average degree of all vertices
- `findSpanningTree()` - find the fewest number of edges of G to keep so that graph is still connected
- `isAcyclic()` - return whether the graph contains no cycles
- `traverse()` - iterate through nodes in an ordered fashion
Facts About Graphs

Fact 1: For any graph $G=(V,E)$, $\sum_{v \in V} \deg(v) = 2|E|$

Proof. Every edge $e=\{u,v\}$ in $G$ is counted twice in the sum of the degrees.

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 ($n=|V|$, $m=|E|$), $m \in O(n^2)$

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.
Therefore, $m \leq \binom{n}{2} = \frac{n(n-1)}{2} \in O(n^2)$

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
     \[
     e[u][v] = \begin{cases} 
     1 & \text{if } (u, v) \in E \\
     0 & \text{if } (u, v) \notin E 
     \end{cases}
     \]
   - or, if you have edge costs:  
     \[
     e[u][v] = \begin{cases} 
     c_e & \text{if } (u, v) \in E \\
     \infty & \text{if } (u, v) \notin E 
     \end{cases}
     \]

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

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

Graph class with adjacency matrix:

```java
int[][] edges
int numNodes

int size() -- return numNodes (or edges.length) RT: O(1)

bool isEmpty() -- return size()==0 RT: O(1)

int getDegree(Vertex u)
    indexU = u’s index in the graph
    degU = sum of num. of non-zero entries in indexU row
    return degU RT: O(n) + time to find u’s index

bool addEdge(Vertex u, Vertex v, [int cost])
    indexU, indexV = u and v’s index in graph, resp.
    + time to find u’s index
    set edges[indexU][indexV] = 1 or cost
    if graph undir, set edges[indexV][indexU] = 1 or cost
    RT: O(1) + time to find u’s index

bool isEdge(Vertex u, Vertex v)
    indexU, indexV = u and v’s index in graph, resp.
    return ( edges[indexU][indexV] != 0 ) RT: O(1) + time to find u’s index
```
2. adjacency list

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

Graph class with adjacency list:

```java
LL<Vertex> nodes // we’ll change this later
int numNodes

Vertex class:
LL<Edge> adj
String name

Edge class:
Vertex start // may not need this if inside Vertex class
Vertex dest
int/double cost

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

boolean equals( Object o ) // if dest same, it’s the same edge
    return ((Edge)o).dest.equals( this.dest )
```
Graph Implementations

Graph with adjacency list:

Vertex class:
   LL<Edge> adj
   String name
   // may add more class members such as prev, dist, scratch

Vertex( name )                  this.name = name

int degree()                return adj.size()

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

bool equals( Object o )
   return ((Vertex)o).name.equals( this.name )

bool 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:
   LL<Vertex> nodes

- 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
```

int size() -- return numNodes (or vMap.size()) RT: O(1)

bool isEmpty() -- return (size() == 0) RT: O(1)

private Vertex getVertex( String vName ) RT: O(1)
```
    Vertex v = vMap.get( vName )
    if( v == null )
        vMap.put( vName, new Vertex( vName ) )
    return v
```

int getDegree( String vName ) RT: O(1)
```
    Vertex v = vMap.get( vName )
    if( v == null ) throw exception
    return v.degree()
```
Graph Implementations

Graph with adjacency list:

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

addEdge( String uName, String vName, int cost)  
  Vertex u = getVertex( uName )
  Vertex v = getVertex( vName )
  u.addEdge( v, cost )
  // if G undir, also v.addEdge( u, cost )  
  RT: O(1)

bool isEdge( String uName, String vName )  
  Vertex u = vMap.get( uName )
  Vertex v = vMap.get( vName )
  if( u or v is null )     return false
  return (u.isNbr( v ))
  RT: O(n)