Graph ADT

**numVertices()**: Returns the number of vertices of the graph.

**vertices()**: Returns an iteration of all the vertices of the graph.

**numEdges()**: Returns the number of edges of the graph.

**edges()**: Returns an iteration of all the edges of the graph.

**getEdge(u, v)**: Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).

**endVertices(e)**: Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.

**opposite(v, e)**: For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.

**outDegree(v)**: Returns the number of outgoing edges from vertex v.

**inDegree(v)**: Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).

**outgoingEdges(v)**: Returns an iteration of all outgoing edges from vertex v.

**incomingEdges(v)**: Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).

**insertVertex(x)**: Creates and returns a new Vertex storing element x.

**insertEdge(u, v, x)**: Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.

**removeVertex(v)**: Removes vertex v and all its incident edges from the graph.

**removeEdge(e)**: Removes edge e from the graph.

---

from Goodrich, Tamassia, Goldwater 6th ed.
Here is an implementation of depth-first search (DFS), using our Graph ADT. The result of the DFS is to fill in the DFS tree. The DFS tree is represented as a Map from vertex -> discovery edge. Assume when DFS() is first called on vertex u that dfsTree is an empty map.

On some inputs, our DFS() is generating a DFS "tree" with cycles in it! Which line has the bug?

<table>
<thead>
<tr>
<th>ANSWER CHOICE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>6</td>
</tr>
</tbody>
</table>
Here is a correct DFS; however we've changed the name and return type because we want to extend the DFS to do something interesting...

Suppose that we want to write a method to detect cycles in undirected graphs. How might we change this method to do so?

Let's assume the graph is connected (i.e., all vertices reachable by a path from u), to avoid changing too much at once.

ANSWER CHOICE

A after line 10 check if u is still the same then return true

B after line 10, add an else case where we return true

C change line 4 to check if known.contains(u) already and if so return true

D after line 6 write "try { g.opposite(v, e); return true; } catch (Exception e) { }"

E after line 10, add an else case where we return true, on line 9 rewrite to "return" before "hasCycle"

F after line 10, add an else case where we return true, on line 9 rewrite to "if (hasCycle...) return true"
#3

Here is the complete hasCycles() method. Why doesn't it work for **directed graphs**? You should create an example directed graph where the method would return the wrong answer.

```java
1. public static <V,E> boolean hasCycle(Graph<V,E> g, Vertex<V> u, 
2. Set<Vertex<V>> known, Map<Vertex<V>, Edge<E>> dfsTree) 
3. {
4.   known.add(u);
5.   for (Edge<E> e : g.outgoingEdges(u)) {
6.     Vertex<V> v = g.opposite(u, e);
7.     if (!known.contains(v)) {
8.       dfsTree.put(v, e);
9.       if (hasCycle(g, v, known, dfsTree)) return true;
10.     } else { return true; } // back edge found;
11.   }
12.   return false;
13.}
```

You may either include a link to the drawing you make on http://sketchtoy.com/

Or you can just write in the edges of the graph:

**Explanation:**

An example directed graph for which hasCycles incorrectly returns true: E = {A->B, A->C, C->D, B->D}

#4

I have a question about the code for DFS or hasCycles.

**Correct Answer:**

True  False

#5

Imagine we build a directed graph where a vertex contains the name of a UI course and there is an edge (X,Y) when X is a pre-requisite for Y.

(treat dotted lines as regular arrows)

Assuming you can take one class per semester, which is a valid course plan?

**ANSWER CHOICE**
#6

Did you know that when you plan your courses, you and your advisor are performing **topological sort**? In topological sort we define a "partial order" $<_S$ over all the vertices in $V$.

An edge $(x,y)$ in $E$ implies that $x <_S y$.

A permutation of the vertices, given as $x_1, x_2, ..., x_n$, is a topological sort if for every constraint $x_i <_S x_j$, $x_i$ comes before $x_j$ in the permutation.

#7

Which statements are both true?

**ANSWER CHOICE**

A. A graph might have 0, 1 or more topological sorts || an acyclic graph has no topological sort

B. A graph has exactly 0 or 1 topological sort || a graph with a cycle has no topological sort

C. A graph might have 0, 1 or more topological sorts || a graph with a cycle has no topological sort
Explanation:

1. As you might have noticed in the course planning example, there could be multiple plans that obey the prerequisites.
2. Proof by contradiction: let there exist a topological sort on G and let G have a cycle with edges \((x_0, x_1), (x_1, x_2), \ldots, (x_{i-1}, x_i), (x_i, x_0)\). Therefore, we must have orders \(x_0 < x_1 < x_2 < \ldots < x_{i-1} < x_i < x_0\), which is impossible.

#8

Give an example of another real-life application that could utilize topological sort.

You should include two things:
1) what do the nodes and edges in the graph represent?
2) why is topological sort useful for such a graph?

Your example should not be related to courses at a university.

Subset of the ideas generated in class:
reading book series you have to read the prequel before the sequel, but you can switch between book series
pancake recipe, where you have to do certain steps before others
advancing in a company through a sequence of job titles; not just a tree b/c different paths may lead you to VP
milestones for completing a project; milestone 1a and 1b might be required before milestone 2

#9

An algorithm for topological sort of a directed acyclic graph (DAG).

1. have a counter called "indegree" for every vertex \(u\), where
   indegree is initialized to \(g.\text{inDegree}(u)\)
2. add all vertices for which indegree=0 to a queue
   while queue is not empty:
   3. pick a vertex \(u\) from the queue
   4. print \(u\)
   5. for each of \(u\)'s predecessors \(v\), decrement \(v\)'s indegree; if \(v\)'s indegree reaches 0, add \(v\) to the queue

For the graph shown, suppose we've run topological sort part way and printed 11, 3 so far. What is the indegree of the other
^ through step 5
I have a question regarding topological sort.

**Correct Answer:**

- True
- False

Consider the weighted (edges labeled with costs) directed graph.
T/F: In general, we can apply breadth-first search to find the lowest cost path (where cost is the sum of the weights on the edges in the path) from d to some other vertex.

Correct Answer:

True  False

#12

We are going to need a modified version of BFS to solve this problem. We'll call the problem of finding the lowest cost path between two vertices: the "shortest paths problem". Next time we will learn 1 or 2 algorithms for finding the shortest paths.

Can you think of a real life situation where solving the shortest paths problem would be useful?

give two things

1. what the nodes, edges, and weight on each edge represent
2. what problem can you solve by finding the shortest paths between vertices?

"traveling salesman" came up a few times. Note that the shortest paths problem is different than the "traveling salesman problem". In Shortest paths we are just finding the shortest path between two nodes. In Traveling salesman problem we are finding the shortest path that "visits all nodes".