DB Assignment 4: Graph Traversal

Siddhesh Karekar, karekar@usc.edu

NOTE: To achieve the **alternative bonus**, queries 2-4 were done in a single command and query 1 in two commands.

Queries

Q1. Write a Gremlin command that creates the graph.

```
g = TinkerGraph.open().traversal()
g.addV('course').property(id, 'CS101').as('CS101').
addV('course').property(id, 'CS201').as('CS201').
addV('course').property(id, 'CS220').as('CS220').
addV('course').property(id, 'CS334').as('CS334').
addV('course').property(id, 'CS400').as('CS400').
addV('course').property(id, 'CS420').as('CS420').
addV('course').property(id, 'CS526').as('CS526').
addV('course').property(id, 'CS681').as('CS681').
addE('requires pre-req').from('CS201').to('CS101').
addE('requires pre-req').from('CS220').to('CS201').
addE('requires pre-req').from('CS420').to('CS220').
addE('requires pre-req').from('CS334').to('CS201').
addE('requires pre-req').from('CS400').to('CS334').
addE('requires pre-req').from('CS681').to('CS334').
addE('requires pre-req').from('CS526').to('CS400').
addE('is a co-req of').from('CS420').to('CS220').
addE('is a co-req of').from('CS526').to('CS400').iterate()
```

Output

```
gremlin> g
==>graphtraversalsource[tinkergraph[vertices:8 edges:9], standard]
```

Explanation

- The first line creates a new instance of the graph, using `TinkerGraph.open()`.
- g is the traversal object for that graph which stores information about traversing it.
- `addV()` creates a new vertex while specifying the label (‘`course`’)
- `property()` assigns the id or unique identifier as the course name
- `as()` provides a reference to later refer to the vertex by,
- `addE()` creates a new edge with the specified label
- `from()` provides the reference of where the edge is from
- `to()` provides the reference of where the edge goes to
- `iterate()` at the end iterates over all instances in the current traversal

Q2. Write a query that will output JUST the doubly-connected nodes.

```
g.V().as('a').outE().inV().as('b').select('a','b').groupCount().unfold().filter(select(values).is(eq(2))).select(keys)
```

Output

```
==>[a:v[CS526],b:v[CS400]]
==>[a:v[CS420],b:v[CS220]]
```

Explanation

- g is the reference to the graph traversal
- `V()` returns the vertices of the graph
- `as('a')` allows them to be later referenced by ‘a’
- `outE()` selects the outgoing edges of the vertex
- `inV()` selects the incoming head vertex of the edge
- `as('b')` allows them to be later referenced by ‘b’
- `select('a','b')` outputs pairs of ‘a’ and ‘b’ i.e. edges
- `groupCount()` creates a map where they key is a vertex pair and the value is the number of edges between the two vertices
- `unfold()` unrolls the path list to separate out vertex pairs
- `filter(select(values).is(eq(2)))` retains only those entries with value 2
- `select(keys)` outputs the map keys, which is our answer

Q3. Write a query that will output all the ancestors (for us, these would be prerequisites) of a given vertex.

```
g.V().has(id,'CS526').repeat(out().dedup()).emit()
```

Output

```
==>v[CS400]
==>v[CS334]
==>v[CS201]
==>v[CS101]
```

Explanation

- g is the reference to the graph traversal
- `V()` returns the vertices of the graph
- `has()` selects the node signifying a specific course by the given id, here ‘CS526’
- `repeat()` loops through the process specified inside it
- `out()` is given as the argument in repeat, hence g traverses along outgoing edges.
- `dedup()` removes duplicate paths.
- `emit()` outputs the traversed vertices by their IDs which are the course names, as we require.

Q4. Write a query that will output the max depth starting from a given node (provides a count (including itself) of all the connected nodes till the deepest leaf).

```
g.V().has(id,'CS101').repeat(__.in()).emit().path().count(local).max()
```

Output

```
==>5
```

Explanation

- g is the reference to the graph traversal
- `V()` returns the vertices of the graph
- `has()` selects the node signifying a specific course by the given id, here `'CS101'`
- `repeat()` loops through a process specified inside it
- `in()` is given as the argument in repeat, which helps travers all incoming edges to the given node
- `emit()` stops repeat() and outputs the traversed vertices
- `path()` gets the path(s) from repeat().
- `count(local)` counts the nodes in the path(s), which returns the depth of all the nodes in the graph
- `max()` takes the path with the most nodes, which returns the max depth as we require.

Alternative Bonus

Queries 2-4 were done in a single command and query 1 in two commands instead of the Eulerian Circuit problem.