ALTERNATIVE BONUS

1. To implement using one or two commands, JSON file is used, which is submitted along with this file.

g=TinkerGraph.open()

Query:

g.io(IoCore.graphml()).readGraph("/Users/siri/Desktop/sample.xml");

g=g.traversal();

Explanation:

- The above command is used to process the xml file and create a graph with vertices and edges in Tinkerpop, here my xml file is sample.xml. Path of the xml file had to be passed into the readGraph() function.

- The graph can be traversed using traversal().

```
guest-wireless-nup-1621-10-121-092-127:apache-tinkerpop-gremlin-console-3.3.0 siri$ bin/gremlin.sh

         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
^[[Aplugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin> g=TinkerGraph.open()
==>tinkergraph[vertices:0 edges:0]
gremlin> g.io(IoCore.graphml()).readGraph("/Users/siri/Desktop/sample.xml");
==>null
gremlin> g=g.traversal()
==>graphtraversalsource[tinkergraph[vertices:8 edges:9], standard]
gremlin> g.V()
==>v[cs526]
==>v[cs681]
==>v[cs220]
==>v[cs101]
==>v[cs420]
==>v[cs201]
==>v[cs334]
==>v[cs400]
gremlin> g.E()
==>e[black7][cs526-edge->cs400]
==>e[orange2][cs526-edge->cs400]
==>e[orange1][cs420-edge->cs220]
==>e[black1][cs201-edge->cs101]
==>e[black2][cs220-edge->cs201]
==>e[black5][cs681-edge->cs334]
==>e[black6][cs400-edge->cs334]
==>e[black3][cs420-edge->cs220]
==>e[black4][cs334-edge->cs201]
```

O/P(Screenshot):

Queries 2, 3, 4 are same as the original queries since they all are single lined queries.

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

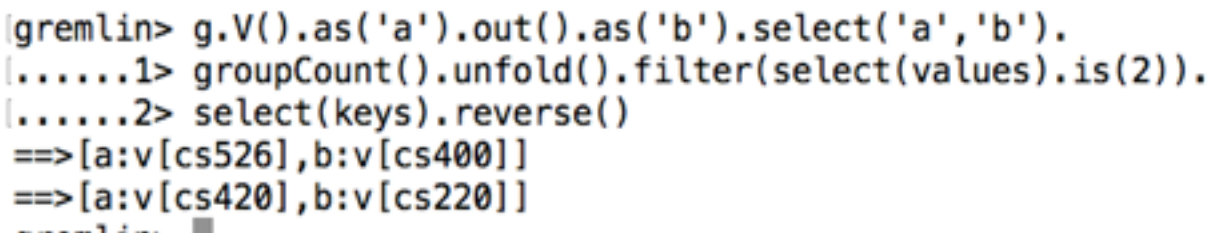
Query

g.V().as('a').out().as('b').select('a','b').groupCount().unfold().filter(select(values).is(2)).
select(keys).reverse()

Explanation:

- The above query outputs only the nodes which are doubly connected i, e a node which contains both a black edge as well as an orange edge.
- In the query, the graph vertices are selected using V().
- out() displays the vertices which have an outgoing edges, it may contain duplicates as well.
- as() names the edges and select() displays the edges as a->b.
- groupCount() counts the repeated vertex pairs from the out() output. and outputs the vertex pairs with their counts.
- unfold() maps the vertex pairs as keys and the group count output as values i, e a hash table structure is formed here as a {key,value} one after the other.
- In the above query the values(groupCounts from above) are selected and passed into the filter() function to filter them with the condition that the groupCounts(values) is equal to 2(doubly connected) using function is().
- From the above output all the keys are selected. Here keys are the vertex pairs. since, the output is not order specific, reverse() is used to print the output in the required order in order to match with the order mentioned in the question.

O/P(Screenshot)



3. Write a query that will output all the ancestors (for us, these would be pre-reqs) of a given vertex.

This query can be done in two ways. Vertex is hardcoded.

Query 1 :

- If we use pre-reqs as a part of out query, then

```
g.V("cs526").repeat( out("requires pre-req" )).emit()
```

Query 2:

- If ancestors had to be found out in a query irrespective of the edge name/property but only based on the outgoing edges upto the root node, then

```
g.V("cs526").repeat( out()).emit().unique()
```

Explanation:

- In the above query, any vertex is taken for which an ancestor had to be found is taken and passed in V(), and repeat() is used to repeat the process of finding an outgoing edge recursively(which are pre-reqs in this case).
- emit() displays the output of repeat() after each iteration.
- In query2, since the name of the edge is not mentioned in the out(), there can be duplicates since all the iterative edges are displayed starting from a pre-req edge as well as co-req edge. Hence unique() is used to display all the unique ancestors till the root**.**

O/P(Screenshot):

```
gremlin> g.V("cs526").repeat(out()).emit().unique()
==>v[cs400]
==>v[cs334]
==>v[cs201]
==>v[cs101]

gremlin> g.V("cs201").repeat(out('requires pre-req')).emit()
==>v[cs101]
gremlin> g.V("cs526").repeat(out('requires pre-req')).emit()
==>v[cs400]
==>v[cs334]
==>v[cs201]
==>v[cs101]
```

4. **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).**

Query:

vertex is hardcoded.

g.V(“cs101”).repeat(__.in()).emit().path().count(local).max()

Explanation:

- In the above query, a vertex is taken and starting from that vertex all the vertices with incoming edges are recursively traversed using repeat(__.in()).
- emit() outputs all the edges in each repeat() step. path() displays the path i, e the traversal starting from the vertex in each step.
- count(local) counts the vertices present in each path. local helps the query to do this. If local is not used, then count() simply displays all the edges it came across in that traversal.
- max() displays the maximum of all the local counts of the traversals.
- If a node is a leaf node, it does not have any maximum depth hence a garbage/negative values in max is displayed.

O/P(Screenshot):

```
gremlin> g.V("cs101").repeat(__.in()).emit().path().count(local).max()
==>5
gremlin> g.V("cs334").repeat(__.in()).emit().path().count(local).max()
==>3
gremlin> g.V("cs526").repeat(__.in()).emit().path().count(local).max()
==>-2147483648
```