**CSCI 585 HW 4**

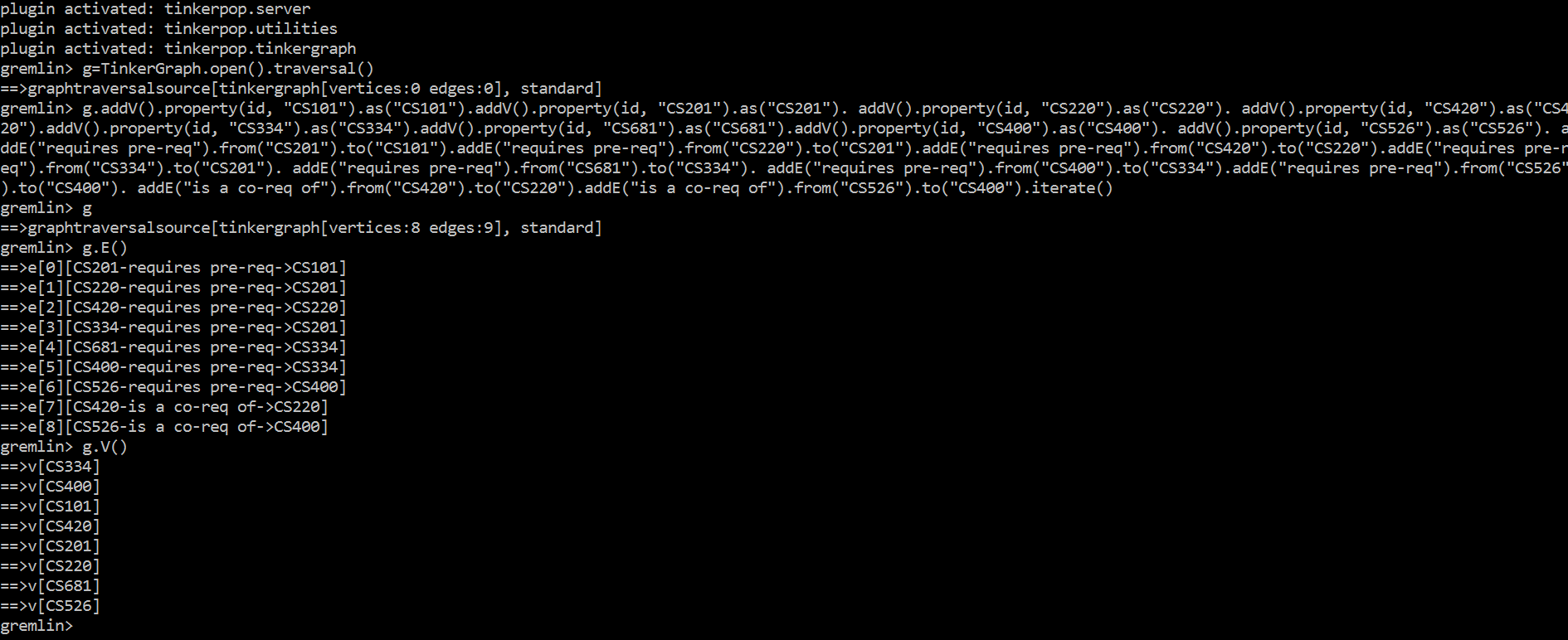
Q1)

Query-

g=TinkerGraph.open().traversal()

g.addV().property(id, "CS101").as("CS101").addV().property(id, "CS201").as("CS201"). addV().property(id, "CS220").as("CS220"). addV().property(id, "CS420").as("CS420").addV().property(id, "CS334").as("CS334").addV().property(id, "CS681").as("CS681").addV().property(id, "CS400").as("CS400"). addV().property(id, "CS526").as("CS526"). 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("CS681").to("CS334"). addE("requires pre-req").from("CS400").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-



Explanation-

1. I have written the query for graph creation as a 2-line query.

2. Now, the initial query creates an instance of graph as g and opens it and the second query chains all the addVertex(addV) and addEdge(addE) queries to accommodate in a single query.

3. addV () adds vertices to the chain.

4. property () step is used to define the property of a node like it’s id, label, etc.

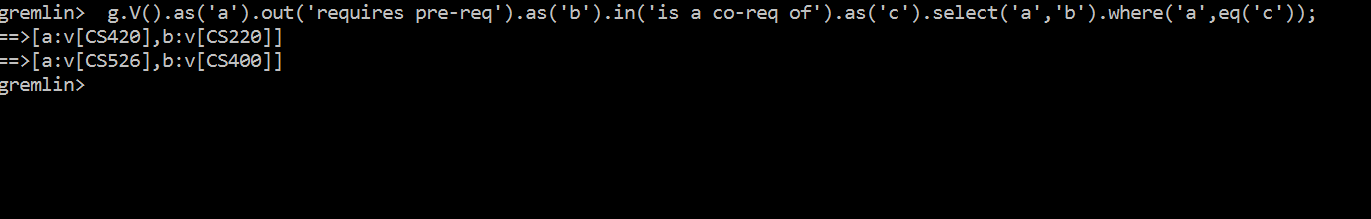
5. addE () step is used to create edge from the .from( ) node to the .to( ) node .

Q2)

Query-

g.V().as('a').out('requires pre-req').as('b').in('is a co-req of').as('c').select('a','b').where('a',eq('c'));

Output-



Explanation-

|  |
| --- |
| 1. Gremlin Commands :- 'V()' lists all the vertices of the traverser. 'g.V()' will traverser to iterate through all the vertices in the graph. 'out()' function takes the traverser to the adjacent vertices on an outward edge. The outward edge can be specified as a parameter to the function. ‘in()’ takes the traverser to the adjacent vertices on an inward edge. ‘select()’ selects the specified vertices in the parameter that are passed by the previous functions in the chaining. "where" gives a condition to the query and helps filter out results.  2. Query Explanation:- If 'a' connects 'b' on an out edge 'p' and 'b' connects 'c' on an in edge 'q' then there are two edges namely 'p' and 'q' from 'a' to 'b' only if 'a' == 'c'. Thus, for all the vertices in the graph, I label each vertex as ‘a’ and go out on the edge 'requires pre-req' and label those vertices 'b'. Now, I check all the vertices that have incoming edge 'is a co-req of' on 'b' and label them as 'c'. Then I display vertices 'a' and 'b' such that a=c using select function. |
|  |

|  |
| --- |
|  |
|  |

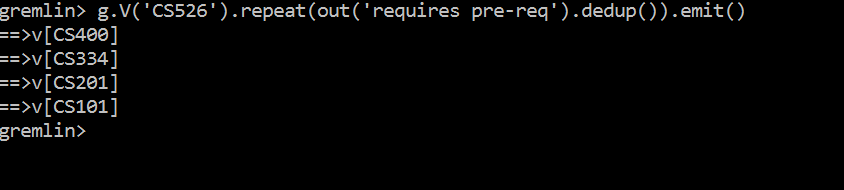
|  |
| --- |
|  |
|  |

Q3)

Query-

g.V('CS526').repeat(out('requires pre-req').dedup()).emit()

Output-



Explanation-

1. This query we find the list of all the ancestors for the node "CS526".

2. .repeat() step is used to repeat a particular sub-query within its parenthesis.

3. In this case, loop statement within repeat() step checks for all the out-going edges with the relationship "requires pre-req" and the loop continues to execute as long as there exists an out edge named ‘requires pre-req’.

4. dedup() step is used to remove duplicates.

5. emit() prints intermediate vertices while traversal.

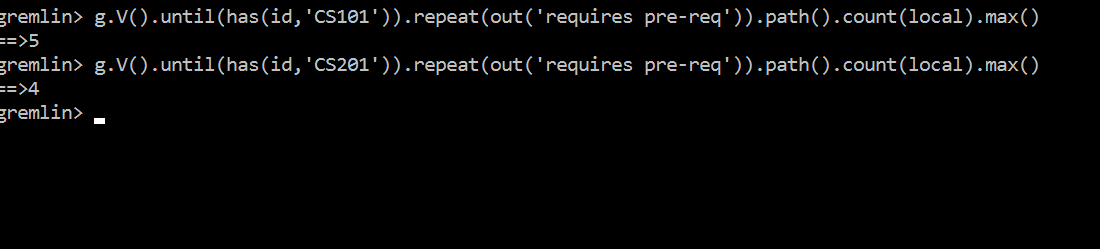
Note- In this query, the position of until() is very important. We can’t use until() before repeat(), as it will include the vertex whose ancestors we have to find.

Q4)

Query-

g.V().until(has(id,'CS101')).repeat(out('requires pre-req')).path().count(local).max()

Output-



Explanation-

1. This query finds the maximum depth that can be reached from a node including itself, CS101 in this case.

2. We can change the id "CS101" to some other id to find depth from that particular id.

3. g.V() is used to run this command from all vertices in the graph. .repeat() step is used to repeat a particular sub-query within its parenthesis until a condition present within the parenthesis of .until() step is true.

The loop runs for all the outgoing edges with relationship "requires pre-req" from all the nodes to "CS101".

4. count(local) gives the count of nodes in each path.

5. path() is used to output all the paths from different nodes to CS101.

6. max() selects the maximum out of these path lengths.

Hence, we get the length of the longest path from CS101.

Note-

a. It is important to place until() before repeat() to also include the starting node in the count.

b. This query returns 1 for all the leaf nodes and a negative value for all the non-existing nodes.