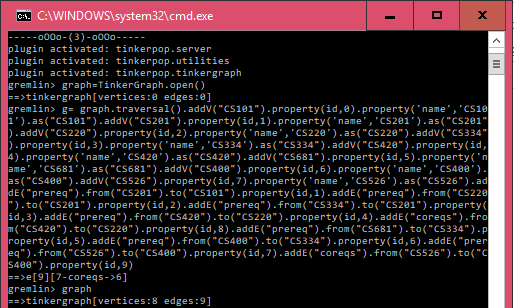
Queries:

**Q1. Write a Gremlin command that creates the above graph [hint - you will also need a 'traversal' for it]. The command could be a multi-statement one, or a single line one (with function chaining).**

graph=TinkerGraph.open()

g= graph.traversal().addV("CS101").property(id,0).property('name','CS101').as("CS101").addV("CS201").property(id,1).property('name','CS201').as("CS201").addV("CS220").property(id,2).property('name','CS220').as("CS220").addV("CS334").property(id,3).property('name','CS334').as("CS334").addV("CS420").property(id,4).property('name','CS420').as("CS420").addV("CS681").property(id,5).property('name','CS681').as("CS681").addV("CS400").property(id,6).property('name','CS400').as("CS400").addV("CS526").property(id,7).property('name','CS526').as("CS526").addE("prereq").from("CS201").to("CS101").property(id,1).addE("prereq").from("CS220").to("CS201").property(id,2).addE("prereq").from("CS334").to("CS201").property(id,3).addE("prereq").from("CS420").to("CS220").property(id,4).addE("coreqs").from("CS420").to("CS220").property(id,8).addE("prereq").from("CS681").to("CS334").property(id,5).addE("prereq").from("CS400").to("CS334").property(id,6).addE("prereq").from("CS526").to("CS400").property(id,7).addE("coreqs").from("CS526").to("CS400").property(id,9)

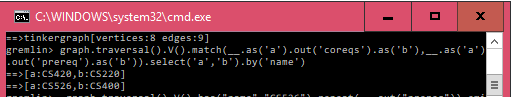


**Explanation:**

* In the above command an instance of TinkerGraph is being created.
* In order to traverse the graph we use graph traversal method named traversal()
* The graph is implemented using chaining method where addition of every vertex and edge (with properties) is separated by a delimiter(‘.’), where I have added 8 vertices (Courses) and 9 edges( relationship) between the courses.
* addV() is used to add vertex
* addE() is used to add edges between already created vertex of the graph
* property() is a common method used by both vertices and edges to define its id or name as per the use
* as() method is used to store the output value into a specific variable specified within “” of as().
* When the chaining command is used in the above manner all the vertices and edges the entire command is executed at the same time.
* The above command can also be written in multistmt fashion. But to make the job simpler we can use chaining.

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

graph.traversal().V().match(\_\_.as('a').out('coreqs').as('b'),\_\_.as('a').out('prereq').as('b')).select('a','b').by('name')

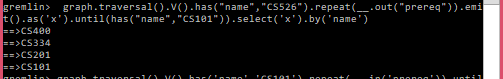


**Explanation:**

* Here in the above command we are trying to find a double relationship in the graph.
* We have represented the relationship in the form of edges.
* There are 2 kind of relationship in the graph (‘prereq’) and (‘coreqs’)
* The goal is to find such vertices that share both relationship
* To do this we use the graph traversal method to traverse the graph.
* For every vertex in the graph to every other vertex in the graph, we check whether there is a relationship of prereq and coreqs between them using match() command.
* match() considers a given vertex as a and checks for its outgoing edge of prereq to b and at the same time it checks that the vertex as a also has its outgoing edge of coereqs to b.
* If yes then we display them by select() by their names.
* select() is used for printing the output
* by() is used to add filter on those output of select()
* In our graph there are 2 pairs which has such relationship.

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

graph.traversal().V().has("name","CS526").repeat(\_\_.out("prereq")).emit().as('x').until(has("name","CS101")).select('x').by('name')



**Explanation:**

* In the above query we are trying to find all the ancestor of a give node.
* Here I have made an assumption of node CS526, you can change it to check for other nodes as well.
* The query is executed by traversing the graph through its ancestor nodes till it reaches the root node.
* To implement this we use repeat()
* Since we are interested in knowing the prereqs we add a filter on prereqs.
* repeat() will keep on running into loop until a particular condition is satisfied by the traverser.
* repeat(\_\_.out(“prereq”)) means for a given node if there is an outgoing edge to a node of relationship prereq, then emit that outgoing edge.
* Here emit() is used to display the node. If we want the vertex no. the emit() command is sufficient.
* But if we want the graph to emit the names of node then, we have to store the nodes emitted as a variable and display it using select() with a by filter on names.
* In our case, it is till the time we reach the root node i.e CS101 the loop will keep on running. For this purpose we use until().

**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). This would give us a total count of the longest sequence of courses that can be taken, after having completed a prereq course.**

graph.traversal().V().has('name','CS101').repeat(\_\_.in('prereq')).until(inE().count().is(0)).path().tail(1).unfold().count()



**Explanation:**

* We want to find out the maximum depth starting from a given node.
* It is calculated for a given node by look for incoming edges with prereq from any node.
* For those node look for incoming edges till there are no incoming edges from any other nodes.
* The above process is repeated in a loop using repeat() and until()
* repeat() will keep on running into loop until a particular condition is satisfied by the traverser.
* The traverser returns the longest path since we traversing every possible solution in the graph by exploring every incoming edge from a node.
* tail() is used for including the vertex in the count.
* unfold() is use to unfold the path of nodes.
* If we don’t write unfold it will just return 1 i.e the longest path but not the individual nodes along the path.
* The no. of nodes in the path is counted using count().