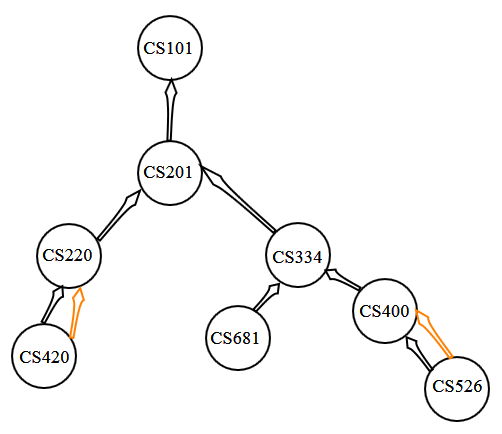
**Home Work - 4**



**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).**

**Solution:**

gremlin> graph = TinkerGraph.open()

==>tinkergraph[vertices:0 edges:0]

**Explanation:**

Creates a memory for the TinkerGraph

gremlin> g = graph.traversal()

==>graphtraversalsource[tinkergraph[vertices:0 edges:0], standard]

**Explanation:**

Provides information for graph traversal by generating a graph traversal source

gremlin> g.addV('CS101').property(id,'CS101').as('v1').addV('CS201').property(id,'CS201').as('v2').addV('CS220').property(id,'CS220').as('v3').addV('CS334').property(id,'CS334').as('v4').addV('CS420').property(id,'CS420').as('v5').addV('CS681').property(id,'CS681').as('v6').addV('CS400').property(id,'CS400').as('v7').addV('CS526').property(id,'CS526').as('v8').addE('requires pre-req').from('v2').to('v1').addE('requires pre-req').from('v3').to('v2').addE('requires pre-req').from('v4').to('v2').addE('requires pre-req').from('v5').to('v3').addE('is a co-req of').from('v5').to('v3').addE('requires pre-req').from('v6').to('v4').addE('requires pre-req').from('v7').to('v4').addE('requires pre-req').from('v8').to('v7').addE('is a co-req of').from('v8').to('v7')

**Output:**

==>e[8][CS526-is a co-req of->CS400]

**Explanation:**

g.addV('CS101').property(id,'CS101').as('v1') – adds a vertex in the graph with id as ‘CS101’. I have given it an alias name ‘v1’ so that it can be referred easily while adding edges.

.addE('requires pre-req').from('v2').to('v1') – adds an edge in the graph with the label 'requires pre-req' going from ‘v2’ to ‘v1’

gremlin> g = graph.traversal()

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



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

**Solution:**

gremlin> g.V().as("a").out().as("b").select("a","b").groupCount().unfold().filter(select(values).is(gt(1))).select(keys)

**Output:**

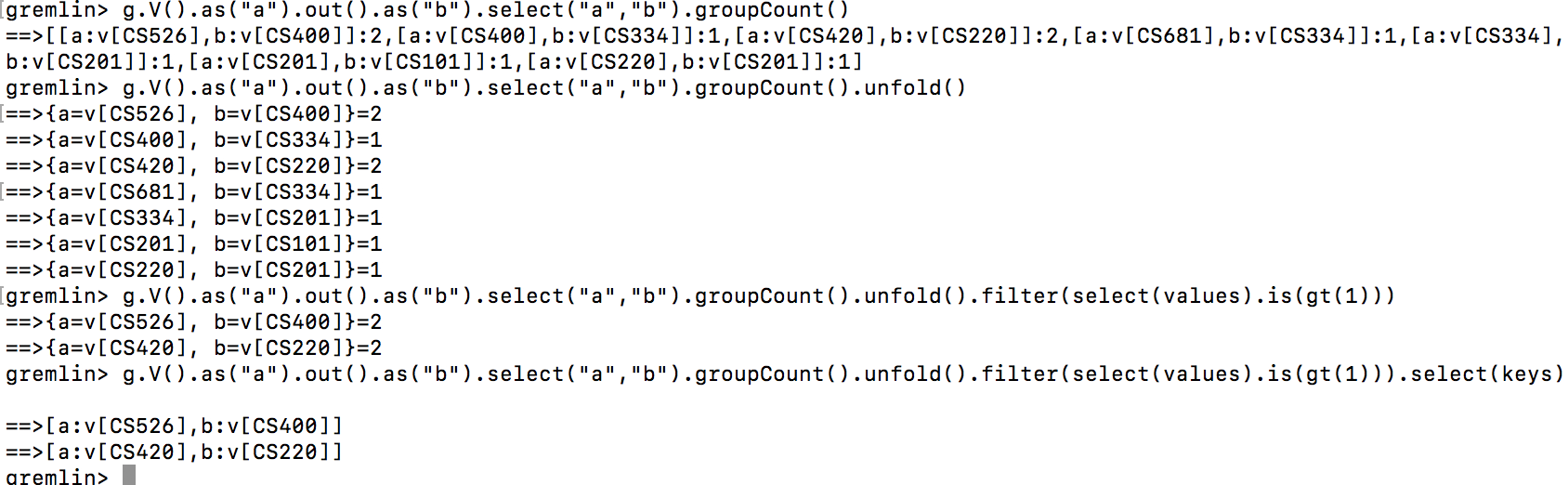
==>[a:v[CS526],b:v[CS400]]

==>[a:v[CS420],b:v[CS220]]

**Explanation:**

Traverse every vertex in the graph and label it as ‘a’. Select an edge going from ‘a’ to another vertex labelled as ‘b’ using select("a", "b"). Count the number of edges going from vertex ‘a’ to vertex ‘b’ as a group using groupCount(). Now using unfold() convert the result in linear form, this represents a Hashmap kind of structure. For every key-value in the obtained result, select only those key-value pairs whose value is greater than 1 which shows that they are doubly connected nodes. The pairs are filtered using filter() function. From these filtered records select only the keys to obtain the result using select(keys).

The following shows the details of each command:



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

**Solution:**

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

**Output:**

==>v[CS400]

==>v[CS334]

==>v[CS201]

==>v[CS101]

**Explanation:**

Note: Here CS526 is hardcoded.

Start with the vertex that has its id as ‘CS526’ using the function has(). Once we find that vertex, we have go to its outward edge using the function out() (this finds the ancestor) and keep repeating this for each node/vertex until no outward edge for a particular vertex is existing i.e. we have reached the highest ancestor vertex. This is done using the function repeat(). In this process there is a possibility that a particular node can have 2 outward edges (i.e. 'requires pre-req' edge and 'is a co-req of' edge) to another same node. So in order to delete duplicates we use the function dedup(). Finally emit() function is used to display all the selected vertices during the traversal.

**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.**

**Solution:**

gremlin> g.V().has(id,'CS101').repeat(\_\_.in().dedup()).emit().tail(1).path().count(local)

**Output:**

==>5

**Explanation:**

Note: Here CS101 is hardcoded.

Start with the vertex that has its id as ‘CS101’ using the function has(). Once we find that vertex, we go to its incoming edge using the \_\_.in() function and repeat this on every new node that comes in its path using the repeat() function until we reach the leaf node that has no incoming edges. We use dedup() function to delete any duplicate vertices and use emit() function for the emission of the vertices found during the traversal. Out of all the vertices selected, we need the last vertex which will the deepest leaf, so we use tail(1) function. The path() function gives the entire path from ‘CS101’ to the deepest leaf. In order to find the count of the current local object

we use the count(local) function. Thus, we find the total count of the longest sequence of courses that can be taken, after having completed a prereq course.

**ALTERNATIVE BONUS (0.25\*4=1 point).**

**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).**

**Solution:**

gremlin> graph = TinkerGraph.open()

==>tinkergraph[vertices:0 edges:0]

gremlin> g = graph.traversal()

==>graphtraversalsource[tinkergraph[vertices:0 edges:0], standard]

gremlin> g.addV('CS101').property(id,'CS101').as('v1').addV('CS201').property(id,'CS201').as('v2').addV('CS220').property(id,'CS220').as('v3').addV('CS334').property(id,'CS334').as('v4').addV('CS420').property(id,'CS420').as('v5').addV('CS681').property(id,'CS681').as('v6').addV('CS400').property(id,'CS400').as('v7').addV('CS526').property(id,'CS526').as('v8').addE('requires pre-req').from('v2').to('v1').addE('requires pre-req').from('v3').to('v2').addE('requires pre-req').from('v4').to('v2').addE('requires pre-req').from('v5').to('v3').addE('is a co-req of').from('v5').to('v3').addE('requires pre-req').from('v6').to('v4').addE('requires pre-req').from('v7').to('v4').addE('requires pre-req').from('v8').to('v7').addE('is a co-req of').from('v8').to('v7')

**Output:**

==>e[8][CS526-is a co-req of->CS400]

**Explanation:** As above.

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

**Solution:**

gremlin> g.V().as("a").out().as("b").select("a","b").groupCount().unfold().filter(select(values).is(gt(1))).select(keys)

**Output:**

==>[a:v[CS526],b:v[CS400]]

==>[a:v[CS420],b:v[CS220]]

**Explanation:** As above

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

**Solution:**

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

**Output:**

==>v[CS400]

==>v[CS334]

==>v[CS201]

==>v[CS101]

**Explanation:** As above.

**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.**

**Solution:**

gremlin> g.V().has(id,'CS101').repeat(\_\_.in().dedup()).emit().tail(1).path().count(local)

**Output:**

==>5

**Explanation:** As above.