**Homework 4 – Bonus questions**

1. Gremlin command to create the graph

graph=TinkerGraph.open()

g=graph.traversal()

g.addV('CS101').property(id,'CS101').property('is a pre-req of','CS201').as('101').addV('CS201').property(id,'CS201').property('is a pre-req of','CS220,CS334').as('201').addV('CS220').property(id,'CS 220').property('is a pre-req of','CS420').property('co-req','CS420').as('220').addV('CS4 20').property(id,'CS420').as('420').addV('CS334').property( id,'CS334').property('is a pre-req of','CS681,CS400').as('334').addV('CS681').property(id,'CS 681').as('681').addV('CS400').property(id,'CS400').property ('is a pre-req of','CS526').property('is a co-req of', 'CS526').as('400').addV('CS526').property(id,'CS526').as('526').addE('requires prereq').from('201').to('101').property(id,1).addE('requires prereq').from('220').to('201').property(id,2).addE('requires prereq').from('420').to('220').property(id,3).addE('is a co-req of').from('420').to('220').property(id,4).addE('requires prereq').from('334').to('201').property(id,5).addE('requires prereq').from('681').to('334').property(id,6).addE('requires prereq').from('400').to('334').property(id,7).addE('requires prereq').from('526').to('400').property(id,8).addE('is a co-req of').from('526').to('400').property(id,9)

Explanation:

I have used TinkerGraph.open() to create an initial graph which will have no edges or vertices and ‘g’ is a traverser.

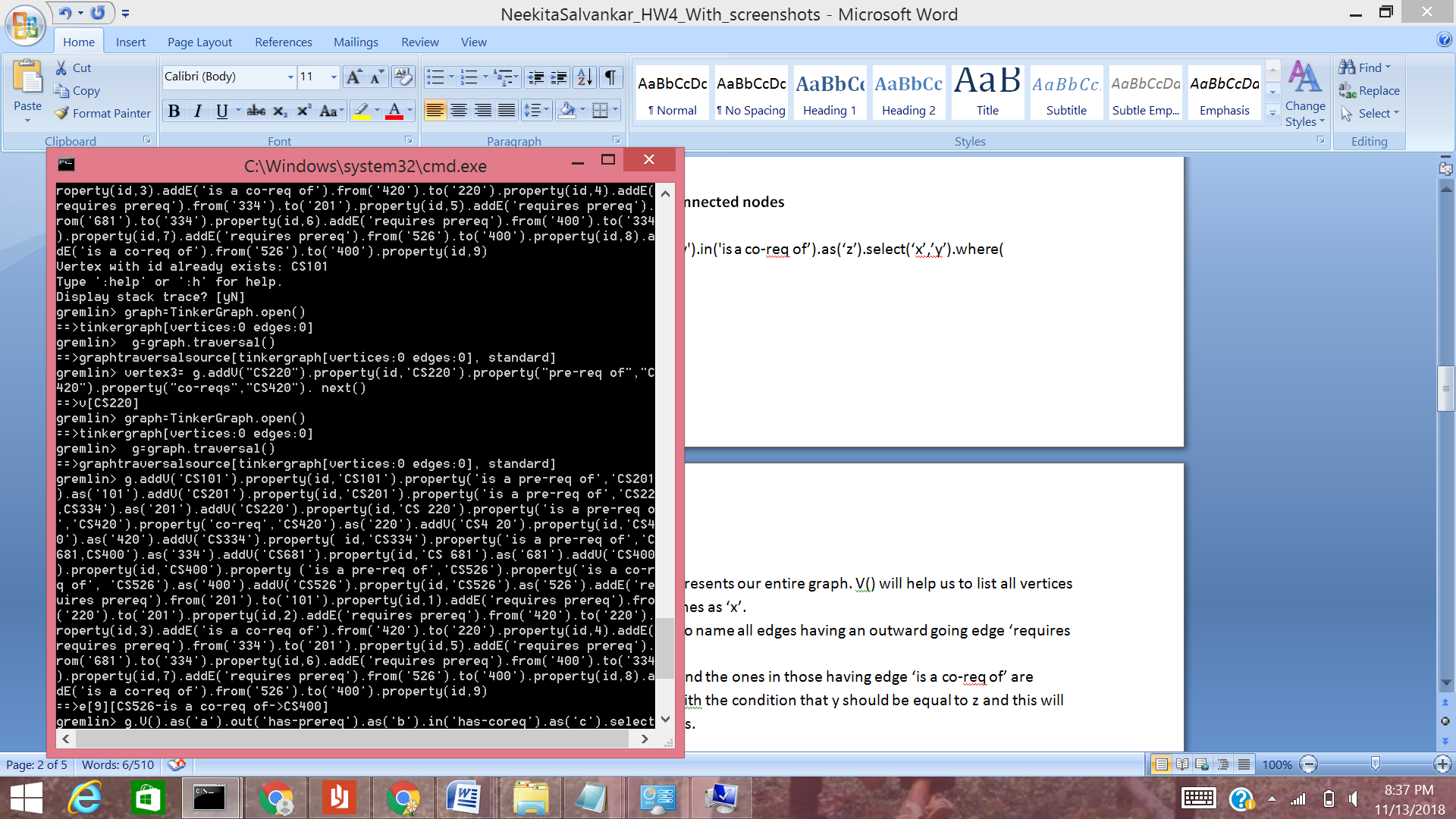
.addV() creates a vertex for my graph and .property() will add a property to it.

I have also used .as() to label the vertex.

.from(), .to() will create an edge between 2 nodes and we will add a name to it using .addE()

More explanation:

I have created each vertex using .addV() and assigned names to them from vertiex 1 to 8 and also assigned 3 types of properties: id, name and where it is a pre-req of some other subject. I have used .as() to label the vertex for easy use while creating edges. I have created edges using from() and to() inside which I have used label names of the vertices to access them. .addE() has 2 types: requires pre-req and is a co-req for. The latter one is used when there are double connections



2. **Write a query to output the double connected nodes**

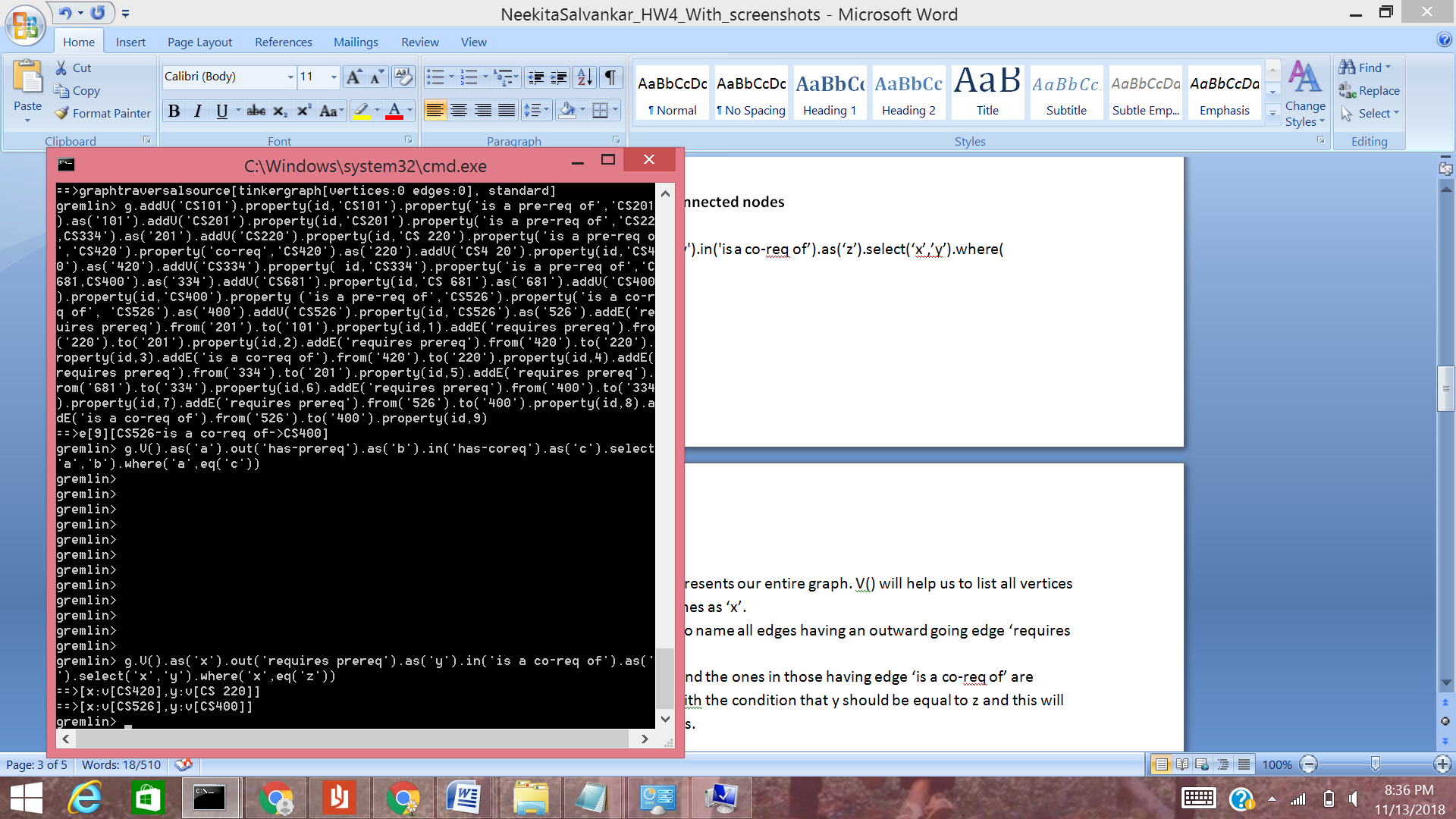
g.V().as('x').out('requires prereq').as('y').in('is a co-req of’).as(‘z’).select(‘x’,’y’).where( ‘x’,eq(‘z’))

Explanation:

We are using g as a traverser and it represents our entire graph. V() will help us to list all vertices in the graph and all of them will be names as ‘x’.

We use .out('requires pre-req').as(‘y’) to name all edges having an outward going edge ‘requires pre-req’ as ‘y’.

Now these ‘y’ vertices are considered and the ones in those having edge ‘is a co-req of’ are named as ‘z’. We then select x and y with the condition that y should be equal to z and this will help us get the double connected nodes.



3. Q**uery to output all the ancestors of a given vertex**

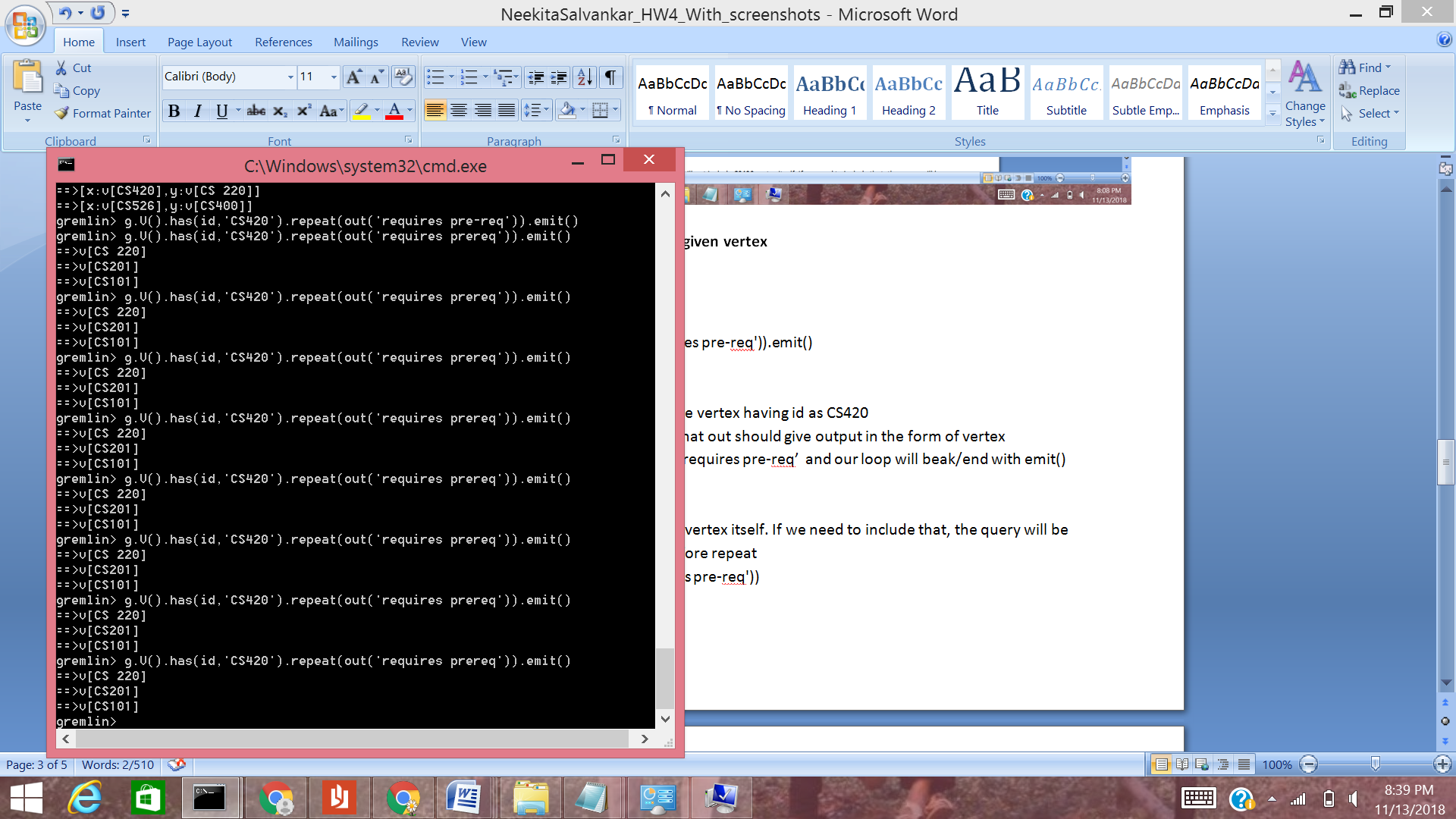
I have taken CS420 as an example

g.V().has(id,'CS420').repeat(out('requires prereq')).emit()

**Explanation**: g is used to traverse

.V().has(id, ‘CS420’) is used to access the vertex having id as CS420

I have used repeat with the condition that out should give output in the form of vertex connected to a edge having the name ‘requires pre-req’ and our loop will beak/end with emit()



4. **Query to output max depth from a given node**

Taking the start node as CS201

g.V().has(id,'CS201').emit().repeat(\_\_.in('requires prereq')).tail(1).path().unfold().count()

**Explanation:**

g is the traverser representing our graph and g.V() will list all the vertices.

emit() is used immediately to emit and use CS201 immediately as we require it to calculate the overall depth. After emitting CS201, it will enter the loop of repeat. The repeat condition requires verticies having incoming edges with label ‘requires prereq’

I have used \_\_in() for getting a top to bottom path and path() helps in getting the same(first vertex to last vertex)

Count() will count the number of lines, that is, the depth and unfold will help to display vertices line by line

Tail() is used to limit the objects. Tail(1) will give the last object only

