SEMINAR 3

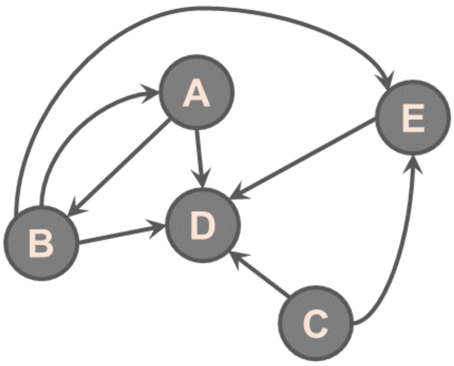
GRAPHS

Week 8 – Seminar 3

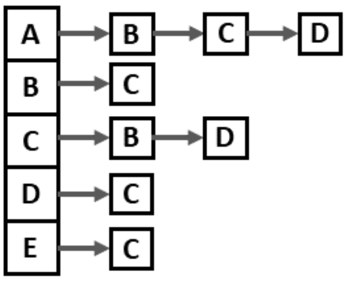
Exercise 1:

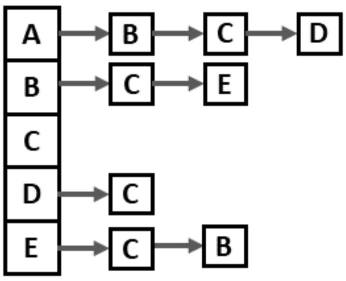
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Exercise 2:



Exercise 3:





Exercise 4:

Brute force

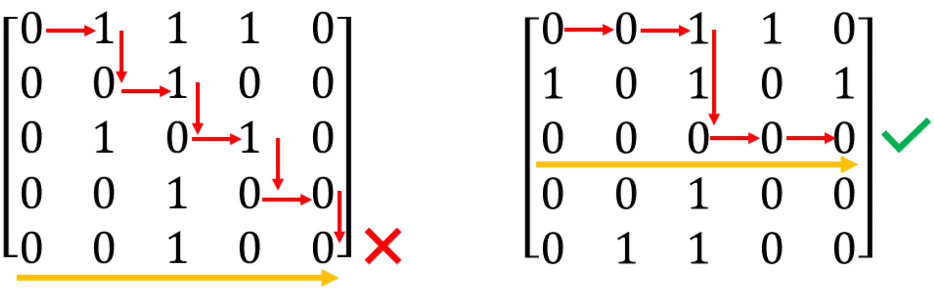
```
Function influencer(network:Graph):int
    v = network.vertices().size() // the number of vertices
    for i := 0 to v-1 do
        followers = 0
        follows = 0
        for j := 0 to v-1 do
            if network[j][i] = 1 then
                followers += 1
            if network[i][j] = 1 then
                follows = 1
                break // no need to continue, not influencer
            endif
        endfor
        if followers = v-1
           and follows = 0 then
            return i // no need to continue, found influencer
        endif
    endfor

    return -1
```

In the worst case we need to go through each vertex and each edge, and therefore the complexity is $O(n + e)$ where n is the number of vertices, and e is the number of edges. In a dense graph, $e \approx n^2$.

A better approach

The idea is for a given vertex v, check if it satisfies the properties of an influencer with the succeeding vertices. If a vertex prove that v is not an influence, this vertex becomes the next possible candidate. If none of the succeeding candidates cannot disprove that v is a candidate, then v is the only possible candidate. As a final step, we need to check if any of the preceding vertices disprove that v is an influencer, if none exists, then v is an influencer. This is shown below, where the graph on the left-hand side has no influencer, and the network on the right-hand side where vertex 2 is an influencer.



```
Function influencer(network:Graph):int
    v = 0
    numberVertices = network.vertices().size()
    while v < numberVertices -1 do
        candidate = true
        j = v + 1
        // in the while loop we try to find the next
        // person that does not follow v or v follows
        while j < numberVertices do
            if network[v][j]=1 or network[j][v]=0 then
                candidate = false
                v = j
                break // no need to continue,
                      // not an influencer
            endif
            j += 1
        endwhile
        if candidate then // the only possible candidate
             // try to disprove it is an influence
             for j := 0 to numberVertices -1 do
                 if j ≠ v and (network[v][j]=1
                    or network[j][v]=0) then
                     return -1 // not an influencer
                 endif
             endfor

             // could not be disprove, therefore an influencer
             return v
         endif
    endwhile

    return -1 // if we arrive here, no influencer found
```

At most we have to go through each vertex twice, so the complexity is $O(n)$ where n is the number of vertices in the network.

A simpler way to write the pseudo code can be achieved by using two for loops. One of your peers found this solution:

```
Function influencer(network:Graph):int
    candidate = 0
    numberVertices = network.vertices().size()
    for user = 1 to network.vertices().size()-1 do
        if network[candidate][user]=1
           or network[user][candidate]=0 then
                candidate = user
        endif
    endfor

    for user = 1 to candidate-1 do
        if network[candidate][user]=1
           or network[user][candidate]=0 then
                return -1
        endif
    endfor
    return candidate
```

Exercise 5:

For this exercise, we should use the Breadth-First-Search algorithm. In addition, we need to store the distance for each node encountered (when encountered for the first time).

```
Function connexionDegree(network:Graph,
                         source:Node,
                         target:Node): int
    toProcess := empty queue // of pair (Node, distance)
    toProcess.enqueue((source,0))
    visited:=[false, false, …, false] // size|V|
    visited[source] := true
    while toProcess not empty do
        (current, distance) := toProcess.dequeue()
        if distance > 3 then // remember, BFS searches
                             // vertices by layers
            return 0
        else if current == target then
            return distance
        endif
        for next in current.neighbours() do
            if not visited[next] then
                toProcess.enqueue((next, distance+1))
                visited[next] := true
            endif
        endfor
    endwhile
    return 0 // if we arrive here, not linked
```