GRAPH ORIENTATION PROBLEM

ALGORITHMIC ASPECTS OF TELECOMMUNICATION NETWORKS

TE 6385.001

V. PURNA CHANDRA VINAY KUMAR

PXV171630

Table of Contents

INTRODUCTION 3
PROBLEM STATEMENT 3
Tasks: 3
APPROACH 3
ALGORITHM 4
Test Examples: 7
Time Complexity 9
FLOW CHART 10
REFERENCES 11
APPENDIX 11
READ ME: 13

INTRODUCTION

In the given problem, given an undirected simple graph, the orientation of the graph is found such that the maximum outdegree of the graph is minimized.

Out degree of the node is defined as the number of directed edges that point out of the node and the action of assigning directions to the undirected graph is referred to as *orientating* the graph. Our job is to find the orientation of the graph such that maximum out degree is minimized.

PROBLEM STATEMENT

The aim of the project is to assign the directions to the undirected edges and to minimize the maximum outdegree of the graph. The specific tasks of the project includes:

Tasks:

- ➢ To design an algorithm to solve for orientation of the undirected graph.
- ➢ Analyzing the running time of the algorithm, in terms of $O(.)$ notation. The algorithm must run in polynomial time, in terms of graph algorithm.
- ➢ Proving the correctness of the algorithm.
- ➢ To write a program that implements the algorithm.
- ➢ Demonstrating the correctness of the algorithm and program by taking some test examples of different sizes, and different structure of the networks.

APPROACH

The approach in designing the algorithm that solves for orientation of the graph is as follows:

- ➢ The program takes the graph matrix and the number of nodes as input. The algorithm works for two types of graphs.
    - o *Case1:* The graph that is complete i.e. all the nodes of the network are connected to every other node. In this case only the number of nodes is given as input and graph matrix is not required
    - o *Case 2:* The graph that is not completely connected in which case the graph matrix that defines the graph connectivity and the number of nodes is required as input to the algorithm.
- ➢ The maximum outdegree of the node is solved by using the fact that the outdegree $odeg(i)$ any node in the graph cannot be greater than or equal to $\max\_\deg = \left\lceil \frac{N}{n} \right\rceil$ i.e.

$odeg(i) \leq \left\lceil \frac{N}{n} \right\rceil$ where $'N'$ denotes the number of links or edges in the graph, $'n'$ denotes the number of nodes of the graph. $\lceil . \rceil$ represents the round the fraction to the nearest highest integer value.

$$odeg(i) \leq \left\lceil \frac{N}{n} \right\rceil$$

- For case 1, i.e. when the graph is complete, the algorithm constructs the graph matrix and the process of finding the orientation is done.
- If outdegree of any node in the graph is greater than $\left\lceil \frac{N}{n} \right\rceil$, then the direction of the link is inverted.
- The algorithm runs over the links instead of nodes so that the time complexity of the algorithm becomes $O(kn)$ which is generalized as $O(n)$.
- The algorithm is tested for different graphs of different nodes and structures.

ALGORITHM

The programming language Python v.3.6.4 is used to solve the orientation of the graph.

<u>High Level Description</u>:

1. The algorithm takes number of nodes $'n'$ and the graph matrix as inputs.
2. For case 1, the program constructs the directed graph matrix M_{ixj} which assigns the edges outwards or inwards for all nodes $'i > j'$

$$M_{i \times j} = \begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The 1's in the matrix represents the orientation of the link from node i to the node j and 0's denote the vice versa.

3. For case 2, the graph matrix and the number of nodes is given as input. The direction of the link is specified by inserting 1 for corresponding row and column. The links that does not exist must be inserted with -1 (self-loops may be neglected and inserted with 0's).

$$M_{i\times j} = \begin{array}{cccccc} \cdot & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 0 & -1 \\ 3 & -1 & 0 & 1 & 0 & 0 \\ 4 & -1 & 1 & -1 & 1 & 0 \end{array}$$

4. The number links for case 1 the number of links $'\mathrm{N}'$ for node $'\mathrm{n}'$ is $N = \frac{\mathrm{n(n-1)}}{2}$ and for the case 2 the number of links is counted by iterating over the links.
5. The upper triangular elements of the graph matrix are saved to a list. The elements in the represents all the links in the graph. -1 in the list denotes that the link does not exist. The index of the list $\overline{M}[t]$ is mapped to the corresponding nodes i, j of graph matrix M_{ixj}.

$$\overline{M}[t] = M[i,j] = [(0,1),(0,2),\dots,(0,4),(1,2),\dots,(1,4),(2,3),\dots,(2,4),(4,3)]$$

6. The list index $\overline{M}[0]$ corresponds to the matrix $M(0,1)$, and $\overline{M}[4]$ to $M(1,2)$ soon, and $\overline{M}[9]$ corresponds to $M(4,3)$. Let $'t'$ denotes the index of link state list $'\overline{M}'$ which corresponds to the nodes i, j of matrix M, t can be mapped to i, j with an algorithm as follows:

```
i,j  =  0,0
for t in links:
        If j = nodes-1:
           i++
        j = (t+(i+1)*(i+2)/2)%nodes
```

The above matrix is represented as list as follows

$$\overline{M}[t] = [1,1,-1,-1,0,1,0,0,-1,0]$$

7. The two lists outdegree and indegree stores the outdegree and indegree of the node. The length of the list is equal to the number of nodes.
8. The algorithm iterates over all the edges in the network. If the list contains 1, the outdegree of the node i generated from step six, is incremented. If the link contains 0, then the in degree of the node i is incremented.
9. If the outdegree or indegree of the node i is greater than $\max\_deg$, then direction of the link is inverted i.e. the value in the list of $\bar{M}[t]$ is inverted to 1 if it contains 0 and 0 if it contains 1. The loop is skipped if it encounters -1.
10. The process is also repeated for the node j.
11. The list $\bar{M}[t]$ is then mapped to the matrix $M[i, j]$ which is the required graph.

Pseudo Code

Case 1: For case 1 the input to the program is number of nodes. As the graph is complete, the number of links in the graph is $\frac{n\times(n-1)}{2}$

```
def construct_graph(number_of_nodes):
    construct a link_state list $\bar{M}[t]$
    for $t$ in all links $\bar{M}$:
        map $i, j$
        if(outdegree of node $i$ > $\max\_deg$ or indegree of node $i$ > $\max\_deg$)
            orient the link $t$ in opposite direction
        else:
            update outdegree $i$ if t is 1
            update indegree $i$ if t is 0
        if(outdegree of node $j$ > $\max\_deg$ or indegree of node $j$ > $\max\_deg$)
            orient the link $t$ in opposite direction
        else:
            update outdegree $j$ if $t$ is 1
            update indegree $j$ if $t$ is 0

        map list $\bar{M}$ to matrix $M$.
```

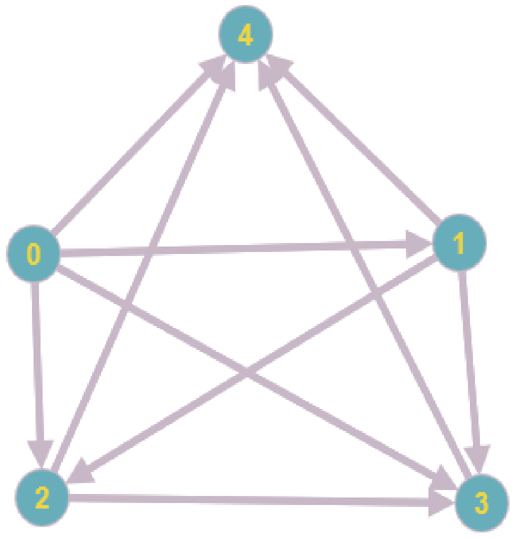
Case 2: For case the number of links are counted from the given matrix which adds an extra 'for' loop in the algorithm.

```
$N$ = 0
for $t$ in all links $\bar{M}$:
    map $i, j$ from $t$
    $\bar{M}[t] = M[i, j]$
    If($M[i, j]$ not equal to -1)
        Increment no of links $N$

max_$deg$ = $\left\lceil \frac{N}{n} \right\rceil$
```

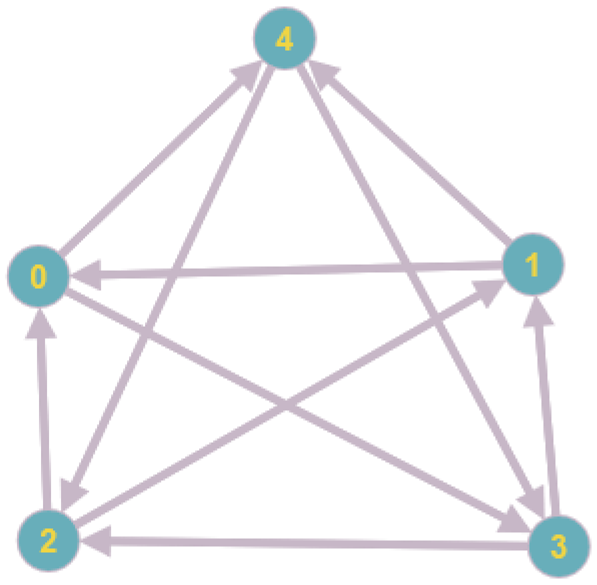
Test Examples:

- Consider a complete graph of 5 nodes with number of links 10 and whose graph matrix is as follows.



$$M_{i\times j} = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 0 \end{array}$$

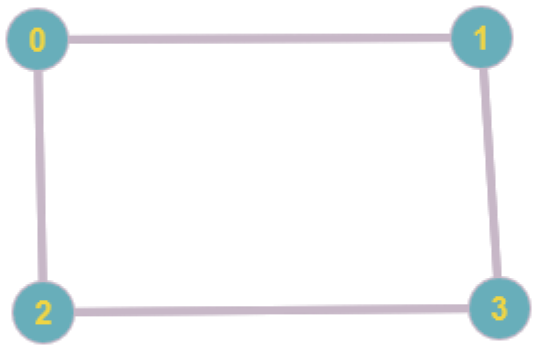
- The matrix $M_{i\times j}$ is mapped to $\bar{M} = [\,1,1,1,1,1,1,1,1,1,1]$. As the node 0 has the highest outdegree and node 4 has the highest indegree, while iterating over the links starting from the link $(0,1)$, $(0,2)$ soon, the direction of the link is inverted if the outdegree of the node ‘0’ is greater than max_deg which is $\frac{10}{5} = 2$.
- Therefore, the output graph of the algorithm looks like,



$$M_{i\times j} = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 \end{array}$$

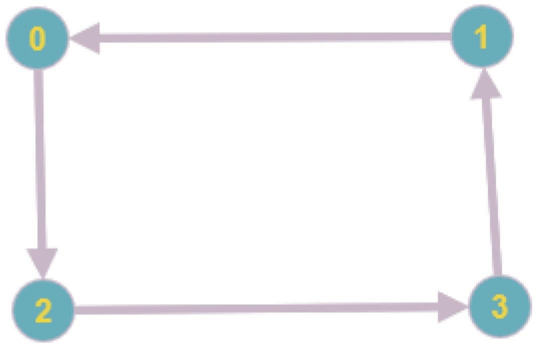
Where each node has the maximum outdegree of 2.

- Consider another example of 4 nodes where each node is connected to only two of the adjacent nodes. Hence, the number of link of the network is 4. The undirected graph and corresponding graph matrix is,



$$M_{i\times j} = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ 0 & -1 & 0 & 0 & -1 \\ 1 & 0 & -1 & -1 & 0 \\ 2 & 0 & -1 & -1 & 0 \\ 3 & -1 & 0 & 0 & 0 \end{array}$$

- Since the maximum outdegree of every node should be equal to one, on applying algorithm the outgraph matrix would be,



$$M_{i\times j} = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 1 & -1 & 0 & 0 \\ 2 & 0 & 0 & -1 & 1 \\ 3 & 0 & 1 & 0 & -1 \end{array}$$

- The algorithm is best suited for all the undirected complete and incomplete graphs. The inputs to the complete graph is only the number of nodes and the program constructs the graph matrix and for incomplete graph the program takes number of nodes and the graph matrix as input and orient the graph by minimizing the maximum out degree.

Time Complexity

The time complexity is defined as the computational complexity that is used to estimate the time taken by the algorithm to execute. The time complexity of an algorithm is usually measured in big O notation typically $O(n), O(n \log n), O(n^a), O(2^n)$ etc. where n is the input size. An algorithm with time complexity of $O(n)$ is a *linear time algorithm,* an algorithm with time complexity $O(n^a)$ for some constant $a \geq 1$ is a *polynomial time algorithm.*

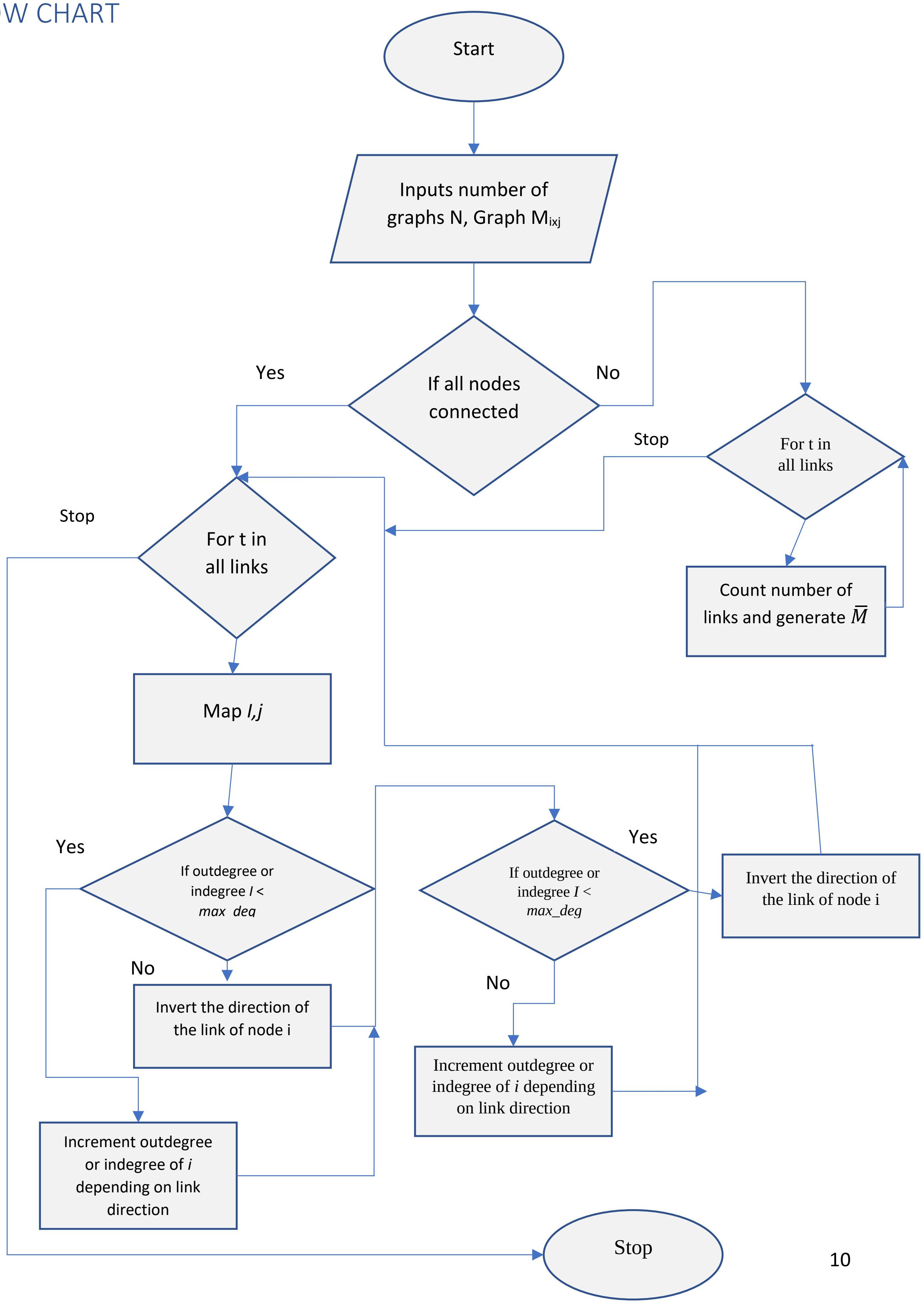
The conditional statements link *if*, usually takes the time complexity of $O(1)$, and *for* loops usually takes time complexity of $O(n)$. Nested loops have the polynomial time complexity with $a > 1$, a being the number of nested loops.

The algorithm consists of one *for* loop for case 1 and two *for* loops for case 2. The number of *if*, conditional statements which has the time complexity of $O(k \times 1)$, k being the number of if statements usually neglected considering the worst-case scenario.

Therefore, for *case 1* one the time complexity of the algorithm is $O(n + k)$, k is equal to 3, and for *case 2* the number of for loops is 2 and therefore the time complexity is $O(m \times n + k)$.

Overall, the time complexity of the algorithm for the worst-case scenario is $O(n)$ on the expense of space complexity.

FLOW CHART


Start
Inputs number of graphs N, Graph M_{ixj}
If all nodes connected
Yes
No
For t in all links
Stop
Count number of links and generate $\overline{M}$
For t in all links
Stop
Map *I,j*
If outdegree or indegree *I* < *max_deg*
Yes
No
Invert the direction of the link of node i
Increment outdegree or indegree of *i* depending on link direction
If outdegree or indegree *I* < *max_deg*
Yes
No
Invert the direction of the link of node i
Increment outdegree or indegree of *i* depending on link direction
Stop

REFERENCES

1) https://en.wikipedia.org/wiki/Orientation_(graph_theory)
2) http://graphonline.ru/en/

APPENDIX

```
import numpy as np

class graph_orientation:
    def __init__(self, graph=np.zeros(1), no_of_nodes=2):

        self.no_of_nodes = no_of_nodes
        self.graph = graph
        self.N = int(((self.no_of_nodes*(self.no_of_nodes-1)/2))+0.5)
        self.min_out_degree = int((self.N/self.no_of_nodes)+0.5)
        self.out_degree = np.zeros(self.no_of_nodes)
        self.in_degree = np.zeros(self.no_of_nodes)
        self.all_connected = True
        self.link_state = np.zeros(self.N)

    def update_graph(self,i,j,l):
        swap = {0:1,1:0}
        inverting = lambda t: swap[t]

        if(self.out_degree[i]+self.link_state[l]>self.min_out_degree or

self.in_degree[i]+inverting(self.link_state[l])>self.min_out_degree):
            self.link_state[l] = inverting(self.link_state[l])
        else:
            self.out_degree[i]+=self.link_state[l]
            self.in_degree[i]+=inverting(self.link_state[l])


if(self.out_degree[j]+inverting(self.link_state[l])>self.min_out_degree or
           self.in_degree[j]+self.link_state[l]>self.min_out_degree):
            self.link_state[l] = inverting(self.link_state[l])

        else:
            self.out_degree[j]+=inverting(self.link_state[l])
            self.in_degree[j]+=self.link_state[l]


    def construct_graph(self):
        if(self.all_connected==True):
            self.graph = np.zeros((self.no_of_nodes,self.no_of_nodes))
            swap = {0:1,1:0}
            inverting = lambda t: swap[t]
            i,j = 0,0
            for l in range(self.N):
                if(j==self.no_of_nodes-1):
                    i+=1

                j = int((l+(i+1)*(i+2)/2)%self.no_of_nodes)
                self.update_graph(i,j,l)
                self.graph[i][j] = self.link_state[l]
                self.graph[j][i] = inverting(self.link_state[l])


        else:
            swap = {0:1,1:0}
            inverting = lambda t: swap[t]
            i,j = 0,0
            self.no_of_links = 0
            for l in range(self.N):
                if(j==self.no_of_nodes-1):
                    i+=1

                j = int((l+(i+1)*(i+2)/2)%self.no_of_nodes)
                self.link_state[int(i*self.no_of_nodes+j-((i+1)*(i+2)/2))]
= self.graph[i][j]
                if(graph[i][j]!=-1):
                    self.no_of_links+=1

            i,j = 0,0
            self.min_out_degree = 
int((self.no_of_links/self.no_of_nodes)+0.5)

            for l in range(self.N):
                if(j==self.no_of_nodes-1):
                    i+=1
                j = int((l+(i+1)*(i+2)/2)%self.no_of_nodes)
                if(self.link_state[l]==-1):
                    self.graph[i][j],self.graph[j][i] = 0,0
                    continue

                self.update_graph(i,j,l)
                self.graph[i][j] = self.link_state[l]
                self.graph[j][i] = inverting(self.link_state[l])
```

READ ME:

1. Python v.3.6.4 is used as a programming software.
2. To run the program, the module is saved into a folder and is imported into the python environment for compilation.
3. Python Jupyter notebook environment is used to compile the program.
4. The module is imported to the notebook, then the module is run to generate the plots.
5. Alternatively, the program can be compiled from the terminal window by running the module using the command ‘import filename’ .