# Untitled

August 24, 2024

[1]:
```
# %networkSC_test_poisson.py

'''
Bachelor Arbeit 'Criticality Analysis on Random Network' based on suppy chain␣
 ↪model
from Jianing(Janine) Ye, 67607, Wirtschaftsmathematik

we used the networkx package as the main package to generate the network,
then the regression analysis from package statsmodels.api,
as well as the correlation analysis from numpy.
'''

import networkx as nx
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import pandas as pd
import random
import bezier
import string
import itertools
import collections
import openpyxl as opxl
import os
import sys
import powerlaw
import math
import time
from matplotlib.collections import LineCollection
from scipy import sparse
from scipy import optimize
import numpy as np
import statsmodels.api as sm


# Global variable that we used in G(n,m)model and G(n,p)model
R = int(8  * 7)   # Ammount of [TIER IV]: raw material           * ␣
 ↪Multi-Parameter
```

```
S = int(20 * 7)   # Ammount of [TIER III]: semifinisched products* Multi-Parameter
C = int(4  * 7)   # Ammount of [TIER II]: components             * Multi-Parameter
M = int(2  * 7)   # Ammount of [TIER I]: modules                 * (Multi-Parameter // 2)
pR = 0.05  * 1   # the probability would not change
pS = 0.025 * 1
pC = 0.02  * 1
pM = 0.02  * 1
E_12 = R  / 2 * 4 # = #(node von R) / 2                        * (Multi-Parameter // 2)
E_23 = S  / 2 * 4 # = #(node von S) / 2                        * (Multi-Parameter // 2)
E_34 = M  / 2 * 4 # = #(node von M) / 2                        * (Multi-Parameter // 2)
pE = 0

# the occupation probability, later will be set as 0.9 as in the paper
rou = 0.85

# the number of the figure that we want to output
num_of_fig = input()
print(type(num_of_fig))

# Cummulatvie product function
def multiply(*list):
    result = 1
    for num in list:
        result = result * num
    return result

# create the network of part [TIER I] with the method GNP(N,p)
def GNP(N, p, string):
    edges = itertools.combinations(range(N), 2)
    G = nx.DiGraph()
    # add nodes into the network
    G.add_nodes_from(range(N))

    # add edges into the network
    for e in edges:
        if random.random() < p:
            G.add_edge(*e)

    # rename the nodes in the network
    # change the label of nodes in TIER from 'NUMBER' into 'ALPHABET+NUMBER'
```

```
    mapping = {}
    for n in range(N):
        node = list(G.nodes)
        a = node[n]
        a = str(a)
        mapping[n] = string + a

    G = nx.relabel_nodes(G, mapping)

    return G


# combine two parts [TIER I] & [TIER II] with the method GNL(N,L)
def TRANS(R, S, C, M, pR, pS, pC, pM, E_12, E_23, E_34):
    SC = nx.Graph()

    # create the random network with the possibility of the part [TIER I]
    T1 = GNP(M, pM, "M-")

    # create the random network with the possibility of the part [TIER II]
    T2 = GNP(C, pC, "C-")
    T2 = RENAME(T2, C, "C-")

    # create the random network with the possibility of the part [TIER III]
    T3 = GNP(S, pS, "S-")
    T3 = RENAME(T3, S, "S-")
    # create the random network with the possibility of the part [TIER IV]
    T4 = GNP(R, pR, "R-")
    T4 = RENAME(T4, R, "R-")

    # set both nodes in T1 and T2 into the Graph T
    SC.add_nodes_from(T1.nodes, layer = 3)
    # SC.add_edges_from(T1.edges, layer = 3)
    SC.add_nodes_from(T2.nodes, layer = 2)
    # SC.add_edges_from(T2.edges, layer = 2)
    SC.add_nodes_from(T3.nodes, layer = 1)
    # SC.add_edges_from(T3.edges, layer = 1)
    SC.add_nodes_from(T4.nodes, layer = 0)
    # SC.add_edges_from(T4.edges, layer = 0)


    BIPARTITE(SC, T1, T2, E_12)
    BIPARTITE(SC, T2, T3, E_23)
    BIPARTITE(SC, T3, T4, E_34)

    return SC, T1, T2, T3, T4
```

```
# connect between two Tiers
def BIPARTITE(SC, T1, T2, E):
    n1list = list(T1)
    n2list = list(T2)
    n2list = n2list[:-1]

    edge_count = 0
    while edge_count < E:
        u = random.choice(n2list)
        v = random.choice(n1list)
        # add more conditions how to build the network properly
        if u == v or SC.has_edge(u, v):
            continue
        else:
            SC.add_edge(u, v)
            edge_count += 1

    return SC, T1, T2


# rename the nodes with "node name"|"dual sourcing name"...
def RENAME(T1, M, string):
    new_name = {}
    redundanced_node = []
    node = list(T1.nodes)
    for n in range(M):
        u = node[n]
        for m in range(M):
            v = node[m]
            if(T1.has_edge(u, v)):
                redundancy = u.strip(string)
                if(len(redundancy) == 1):
                    redundancy = redundancy[0]
                elif(len(redundancy) > 1):
                    idx = redundancy.find("|")
                    # print(idx)
                    if(idx == -1):
                        redundancy = redundancy
                    else:
                        redundancy = redundancy[:idx]
                new_name[v] = v + "|" + redundancy
                T1 = nx.relabel_nodes(T1, new_name)
                node = list(T1.nodes)
            new_name = {}

    return T1
```

```
# arrange the node into multi-partite position
def POSITION(SC, par):
    pos = nx.multipartite_layout(SC, subset_key = 'layer', align = "vertical",␣
 ↪scale = 10)

    n = list(SC.nodes)
    for i in range(len(pos)):
        key = n[i]
        # print("key :", key)
        pos_n = pos[key]
        # print("pos_n :", pos_n)
        pos_n[0] = float(pos_n[0] * 10)
        pos_n[1] = float(pos_n[1] + par)

    return pos


# calculate the performan of the graph with and without one node
def PERFORM(SC, T1, T2, T3, T4, without_node):

    n1list = list(T1.nodes)
    n2list = list(T2.nodes)
    n3list = list(T3.nodes)
    n4list = list(T4.nodes)

    # calculate all adjazent edges of without_node
    if without_node != None:
        neighbor = list(SC.neighbors(without_node))

    SC_copy = SC.copy()

    partition = [n1list, n2list, n3list, n4list]
    perform_G = nx.algorithms.community.partition_quality(SC_copy, partition)

    # delete without_ndoe from the network
    if without_node == None:
        partition = [n1list, n2list, n3list, n4list]
        perform = nx.algorithms.community.partition_quality(SC_copy, partition)
        return perform
    elif "M" in without_node:
        n1list.remove(without_node)
        SC_copy.remove_node(without_node)
    elif "C" in without_node:
        n2list.remove(without_node)
        SC_copy.remove_node(without_node)
```

```
    elif "S" in without_node:
        n3list.remove(without_node)
        SC_copy.remove_node(without_node)
    elif "R" in without_node:
        n4list.remove(without_node)
        SC_copy.remove_node(without_node)
    else:
        return SyntaxError

    partition = [n1list, n2list, n3list, n4list]
    perform = nx.algorithms.community.partition_quality(SC_copy, partition)

    SC_copy.add_node(without_node)
    for i in range(len(neighbor)):
        v = neighbor[i]
        SC_copy.add_edge(without_node, v)

    return perform


def CRITICAL_1(mapping_perform, SC, T1, T2, T3, T4):

    perform_all = PERFORM(SC, T1, T2, T3, T4, None)[1]
    perform_all = round(perform_all, 2)
    nlist = []
    nlist = ADD_IN_LIST(nlist, T1)
    nlist = ADD_IN_LIST(nlist, T2)
    nlist = ADD_IN_LIST(nlist, T3)
    nlist = ADD_IN_LIST(nlist, T4)

    for j in mapping_perform:
        without_node = j
        perform = PERFORM(SC, T1, T2, T3, T4, without_node)[1]
        perform = round(perform, 2)
        mapping_perform[j] = str(perform_all - perform)

    return mapping_perform


# adding nodes to the node list
def ADD_IN_LIST(nlist, T1):

    n1list = list(T1.nodes)
    L = len(n1list)
    for i in range(L):
        nlist.append(n1list[i])
```

```
    return nlist


# looking for the adjacent node in one tier in the graph
def ADJ_IN_ONE_TIER(SC):
    node_list_a = []
    for node in SC.nodes:
        for edge in SC.edges:
            if (node == edge[0]) or (node == edge[1]):
                node_list_a.append(node)
    node_list_a = list(set(node_list_a))
    return node_list_a


# checking in each ter how many adjacent nodes there is
def NUM_OF_NODE_AIOT(node_list_a): # AIOT: adjacent in one tier
    number_of_node_R = 0
    number_of_node_S = 0
    number_of_node_C = 0
    for node in node_list_a:
        if (node[0] == "R"):
            number_of_node_R += 1
        elif (node[0] == "S"):
            number_of_node_S += 1
        elif (node[0] == "C"):
            number_of_node_C += 1
    return number_of_node_R, number_of_node_S, number_of_node_C


# printing the node list according to there first letter
def NODE_LIST(SC, string):
    node_list = []
    for node in SC.nodes:
        if (node[0] == string):
            node_list.append(node)
    node_list = list(reversed(node_list))
    return node_list


# trans the network in to matrix
def NETW_INTO_MATRIX(SC_a, R, S, C):
    SC_ad = nx.adjacency_matrix(SC_a)
    SC_A = SC_ad.todense()
    SC_A_flip180 = SC_A.reshape(SC_A.size)
    SC_A_flip180 = SC_A_flip180[::-1]
    SC_A_flip180 = SC_A_flip180.reshape((R+S+C+M, R+S+C+M))
```

```
    SC_A_R = [row[:R] for row in SC_A_flip180[:R]]
    SC_A_R = np.array(SC_A_R)
    SC_A_R_flip180 = SC_A_R.reshape(SC_A_R.size)
    SC_A_R_flip180 = SC_A_R_flip180[::-1]
    SC_A_R_flip180 = SC_A_R_flip180.reshape((R, R))

    SC_A_S = [row[R:(S+R):1] for row in SC_A_flip180[R:(S+R):1]]
    SC_A_S = np.array(SC_A_S)
    SC_A_S_flip180 = SC_A_S.reshape(SC_A_S.size)
    SC_A_S_flip180 = SC_A_S_flip180[::-1]
    SC_A_S_flip180 = SC_A_S_flip180.reshape((S, S))

    SC_A_C = [row[(S+R):(S+R+C):1] for row in SC_A_flip180[(S+R):(S+R+C):1]]
    SC_A_C = np.array(SC_A_C)
    SC_A_C_flip180 = SC_A_C.reshape(SC_A_C.size)
    SC_A_C_flip180 = SC_A_C_flip180[::-1]
    SC_A_C_flip180 = SC_A_C_flip180.reshape((C, C))

    return SC_A_R_flip180, SC_A_S_flip180, SC_A_C_flip180


# finding all node on the x=0 axis
def List_at_x_axis(node_list_R, node_list_S, SC_a, SC_d, SC_e, minus_par,␣
 ↪pos_cont2, list_on_2nd_line):
    for node in node_list_R:
        list_var = list(SC_d.neighbors(node))
        list_var_b = list(SC_a.neighbors(node))
        for node1 in list_var:
            if ("M" in node1) and (len(list_var_b) == 0):
                pos_cont2[node] = (0, (1 + minus_par))
                # print("pos_cont2[", node, "] = ", pos_cont2[node])
                list_on_2nd_line.append(node)
                SC_e.add_edge(node, node1)
                minus_par = minus_par + 1

            elif ("C" in node1) and (len(list_var_b) == 0):
                pos_cont2[node] = (0, (1 + minus_par))
                # print("pos_cont2[", node, "] = ", pos_cont2[node])
                list_on_2nd_line.append(node)
                SC_e.add_edge(node, node1)
                minus_par = minus_par + 1


    for node in node_list_S:
        list_var = list(SC_d.neighbors(node))
        list_var_b = list(SC_a.neighbors(node))
        for node1 in list_var:
```

```
            if ("M" in node1) and (len(list_var_b) == 0):
                pos_cont2[node] = (0, 1 + minus_par)
                # print("pos_cont2[", node, "] = ", pos_cont2[node])
                list_on_2nd_line.append(node)
                SC_e.add_edge(node, node1)
                minus_par = minus_par + 1

    return list_on_2nd_line, SC_e, pos_cont2


# finding the parallel supply chain and rearrange their position
def parallel_supply_chain(node_list_R, node_list_S, SC_a, SC_d,␣
 ↪node_list_pos_cont, len_node_list_pos_cont, SC_e, pos_cont, pos_cont2):
    for node in node_list_R:
        list_var = list(SC_d.neighbors(node))
        list_var_b = list(SC_a.neighbors(node))
        for node1 in list_var:
            if ("M" in node1) and (len(list_var_b) == 0):
                name_node1 = node1 + "|" + node1[-1]
                SC_e.add_node(name_node1)
                pos_cont[name_node1] = (pos_cont[node1][0], pos_cont2[node][1])
                SC_e.add_edge(node, name_node1)
                SC_e.remove_edge(node, node1)
                if(node_list_pos_cont.index(node1) < len_node_list_pos_cont-1):
                    node3_idx = node_list_pos_cont.index(node1)
                    node3 = node_list_pos_cont[node3_idx+1]
                    SC_e.add_edge(name_node1, node3)

                if(node_list_pos_cont.index(node1) != len_node_list_pos_cont):
                    node2_idx = node_list_pos_cont.index(node1)
                    node2 = node_list_pos_cont[node2_idx-1]
                    SC_e.add_edge(node2, node)
                    pos_cont[node] = (pos_cont[node2][0], pos_cont2[node][1])
                elif(node_list_pos_cont.index(node1) == len_node_list_pos_cont):
                    pos_cont[node] = (pos_cont[node2][0], pos_cont2[node][1])


            elif ("C" in node1) and (len(list_var_b) == 0):
                name_node1 = node1 + "|" + node1[-1]
                SC_e.add_node(name_node1)
                pos_cont[name_node1] = (pos_cont[node1][0], pos_cont2[node][1])
                SC_e.add_edge(node, name_node1)
                SC_e.remove_edge(node, node1)

                if(node_list_pos_cont.index(node1) < len_node_list_pos_cont-1):
                    node3_idx = node_list_pos_cont.index(node1)
                    node3 = node_list_pos_cont[node3_idx+1]
```

```
                    SC_e.add_edge(name_node1, node3)

                if(node_list_pos_cont.index(node1) != len_node_list_pos_cont):
                    node2_idx = node_list_pos_cont.index(node1)
                    node2 = node_list_pos_cont[node2_idx-1]
                    SC_e.add_edge(node2, node)
                    pos_cont[node] = (pos_cont[node2][0], pos_cont2[node][1])


    for node in node_list_S:
        list_var = list(SC_d.neighbors(node))
        list_var_b = list(SC_a.neighbors(node))
        for node1 in list_var:
            if ("M" in node1) and (len(list_var_b) == 0):
                name_node1 = node1 + "|" + node1[-1]
                SC_e.add_node(name_node1)
                pos_cont[name_node1] = (pos_cont[node1][0], pos_cont2[node][1])
                SC_e.add_edge(node, name_node1)
                SC_e.remove_edge(node, node1)
                if(node_list_pos_cont.index(node1) < len_node_list_pos_cont-1):
                    node3_idx = node_list_pos_cont.index(node1)
                    node3 = node_list_pos_cont[node3_idx+1]
                    SC_e.add_edge(name_node1, node3)

                if(node_list_pos_cont.index(node1) != len_node_list_pos_cont):
                    node2_idx = node_list_pos_cont.index(node1)
                    node2 = node_list_pos_cont[node2_idx-1]
                    SC_e.add_edge(node2, node)
                    pos_cont[node] = (pos_cont[node2][0], pos_cont2[node][1])
                elif(node_list_pos_cont.index(node1) == len_node_list_pos_cont):
                    pos_cont[node] = (pos_cont[node2][0], pos_cont2[node][1])

    return pos_cont, SC_e


# build the contracted production network
def CONTRACTED_POS(pos, SC_A_R, IDX_h, node_list, list_on_2nd_line):
    print("SC_A:\n", SC_A_R)
    # SC_A_R = [

    node_list_rdy = []
    node_hor = []
    node_ver = []
    node_hor_to_ver = []

    # list_on_2nd_line = []
```

```
    for i in range(IDX_h):
        for j in range(i, IDX_h):
            if SC_A_R[i][j] == 1:
                # the node in the j-th horizontal line directing to the node in␣
↪the i-th vertical line
                if(node_list[j] not in list_on_2nd_line) and (node_list[i] not␣
↪in list_on_2nd_line):
                    # write down the index of the horizontal line
                    node_hor.append(j)
                    # write down the index of the horizontal line
                    node_ver.append(i)
                    node_hor_to_ver.append((j, i))
                    # (j,i)  <=>  j-->i
                    # with an edge from j---->i

    # since j ---> i
    # the i-th vertical(node) should be on the right handside of the j-th␣
↪horizontal(node)
    # so we set the position of the VERTICAL nodes AT FIRST
    # count how many times the element is in the list
    result_ver = collections.Counter(node_ver)
    result_hor = collections.Counter(node_hor)
    var = -100

    # if there is no nodes connecting in the same Tier, then STOP
    if(len(node_hor_to_ver) == 0):
        return pos, node_list_rdy, IDX_h


    node_mid_h = [x for x in node_hor if x in node_ver]
    node_mid_h = sorted(node_mid_h)
    node_mid_h = list(node_mid_h)
    node_mid_v = [x for x in node_ver if x in node_hor]
    node_mid_v = sorted(node_mid_v)
    node_mid_v = list(node_mid_v)


    if(len(node_mid_h) != 0):
        node_v = [x for x in node_ver if x not in node_mid_v]
        node_h = [x for x in node_hor if x not in node_mid_h]
        result_m_v = collections.Counter(node_mid_v)
        result_m_h = collections.Counter(node_mid_h)
        result_v = collections.Counter(node_v)
        result_h = collections.Counter(node_h)

        for node_w in node_mid_h:
            var_mid_h = result_hor[node_w]
```

```
            var_mid_v = result_ver[node_w]
            node_left = []
            node_right = []

            h_list = [j for j,val in enumerate(node_hor) if val == node_w]
            IDX_v = 0
            param = 0
            for j in h_list:
                node_v = node_hor_to_ver[j][1]
                # in node_hor_to_ver: [1]--> vertical
                left = [node_hor[j] for j,val in enumerate(node_ver) if val ==␣
↪node_v]
                if(len(left) > 1):
                    node_left = [node for node in left if node != node_w]
                if(node_v not in pos.keys()):
                    pos[node_v] = (IDX_h-1, IDX_v)
                    IDX_v = IDX_v - 1
                    node_list_rdy.append(node_v)
                    param = param + 1
            if(param > 0):
                IDX_h = IDX_h - 1

            if(node_w not in pos.keys()):
                pos[node_w] = (IDX_h-1, 0)
                node_list_rdy.append(node_w)
                IDX_h = pos[node_w][0]
                IDX_v = pos[node_w][1]
            IDX_v = 0
            for node_s in node_left:
                if(node_s not in pos.keys()):
                    pos[node_s] = (IDX_h, IDX_v - 1)
                    node_list_rdy.append(node_s)
                    IDX_v = pos[node_s][1]


            v_list = [j for j,val in enumerate(node_ver) if val == node_w]
            IDX_v = 0
            param = 0
            for k in v_list:
                node_u = node_hor_to_ver[k][0]
                # in node_hor_to_ver: [0]--> horizontal
                right = [node_ver[j] for j,val in enumerate(node_hor) if val ==␣
↪node_u]
                if(len(right) > 1):
                    node_right = [node for node in right if node != node_w]
                    IDX_v_r = 0
                for node_t in node_right:
```

```
                    if(node_t not in pos.keys()):
                        while ((IDX_h-1, IDX_v_r) in pos.values()):
                            IDX_v_r = IDX_v_r - 1
                        pos[node_t] = (IDX_h-1, IDX_v_r)
                        node_list_rdy.append(node_t)
                        IDX_v_r = IDX_v_r - 1
                        IDX_h = pos[node_t][0]
                node_right = []
                if(node_u not in pos.keys()):
                    if ((IDX_h-1, IDX_v) not in pos.values()):
                        IDX_v = 0
                    pos[node_u] = (IDX_h-1, IDX_v)
                    IDX_v = IDX_v - 1
                    node_list_rdy.append(node_u)
                    param = param + 1

                if(param > 0):
                    IDX_h = IDX_h - 1


            node_mid_h=[]

    # we use the FOR-loop to check which node should be on the right-est
    for i in range(len(node_ver)):
        node = node_ver[i]
        adjnode = node_hor[i]

        var_node = result_ver[node] # how many times the node is connected in␣
↪one tier
        var_adjnode = result_hor[adjnode]
        node_right = []

        j_list = [j for j,val in enumerate(node_hor) if val == adjnode]
        IDX_v = 0
        param = 0
        for j in j_list:
            if(node_ver[j] not in pos.keys()):
                pos[node_ver[j]] = (IDX_h-1, IDX_v)
                IDX_v = IDX_v - 1
                param = param + 1
                node_list_rdy.append(node_ver[j])
        if(param > 0):
            IDX_h = IDX_h - 1

        k_list = [k for k,val in enumerate(node_ver) if val == node]
        IDX_v = 0
        param = 0
```

```
        for k in k_list:
            right = [node_ver[j] for j,val in enumerate(node_hor) if val ==␣
 ↪node_hor[k]]
            if(len(right) > 1):
                node_right = [node for node in right if node != node_hor[k]]
                IDX_v_r = IDX_v
            for node_t in node_right:
                if(node_t not in pos.keys()):
                    while ((IDX_h-1, IDX_v_r) in pos.values()):
                        IDX_v_r = IDX_v_r - 1
                    pos[node_t] = (IDX_h-1, IDX_v_r)
                    node_list_rdy.append(node_t)
                    IDX_v_r = IDX_v_r - 1
                    IDX_h = pos[node_t][0]
                    IDX_v = 0
            node_right = []
            if(node_hor[k] not in pos.keys()):
                if ((IDX_h-1, IDX_v) not in pos.values()):
                    IDX_v = 0
                pos[node_hor[k]] = (IDX_h-1, IDX_v)
                IDX_v = IDX_v - 1
                param = param + 1
                node_list_rdy.append(node_hor[k])
            if(param > 0):
                IDX_h = IDX_h - 1

    return pos, node_list_rdy, IDX_h


# if the nodes are in the same tier then we arrange the position from left to␣
 ↪right
def CONTRACTED_POS_CONT_AIOT(pos_cont, pos_cont_new, IDX, node_list,␣
 ↪node_list_rdy, list_on_2nd_line, string):
    if(string == "R"):
        add_par = 0
        l = list(range(R))
    elif(string == "S"):
        add_par = R
        l = list(range(S))
    elif(string == "C"):
        add_par = R + S
        l = list(range(C))

    l = np.array([x for x in l if x not in node_list_rdy])
    for x in node_list_rdy:
        key = node_list[x]
        pos_cont_new[key] = (pos_cont[x][0] + add_par, pos_cont[x][1])
```

```
    for i in l:
        key = node_list[i]
        if(pos_cont.get(i) == None) and (key not in list_on_2nd_line):
            pos_cont_new[key] = (IDX - 1 + add_par, 0)
            IDX = IDX - 1
    return pos_cont_new


# fill the contracted production network with the failed positon for tier M
def CONTRACTED_POS_CONT_FILL(pos_cont, SC):
    idx = R+S+C
    for node2 in SC.nodes: #reposition the nodes WITHOUT adjazent nodes in one␣
 ↪Tier
        if("M" in node2):
            pos_cont[node2] = (idx, 0)
            idx += 1
    return pos_cont


# connection between tier R and tier C/M, connection between tier S and tier M
def ADD_EDGE_TO_NEXT_TIER(node, string1, number, next_tier, SC_d, SC_b):
    if (string1 in node):
        neighbours = list(SC_b.neighbors(node))

        neighbours_1 = neighbours.copy()
        if(string1 == "S") and (len(neighbours) != 0):
            l = len(neighbours)
            for node1 in neighbours:
                if("R" in node1):
                    neighbours_1.remove(node1)
        elif(string1 == "C") and (len(neighbours) != 0):
            l = len(neighbours)
            for node1 in neighbours:
                if("R" in node1) or ("S" in node1):
                    neighbours_1.remove(node1)
        neighbours = neighbours_1

        if (len(neighbours) == 0):
            par = 1
        for node2 in neighbours:
            if(string1 in node2):
                par = 1
            else:
                par = 0
        if(par == 1):
            rand_idx = random.randrange(number)
            node_rand = next_tier[rand_idx]
```

```
            SC_d.add_edge(node, node_rand)
            SC_b.add_edge(node, node_rand)
    return SC_d, SC_b


# comparing the nodes to find the critical nodes
def comparing_critical_node(all_path, orig_path):
    critical_node = orig_path
    for p in all_path:
        critical_node = list(set(critical_node) - (set(orig_path) - set(p)))

    return critical_node


# bubble sourt for the nodes
def bubble_sort(length, list_pos_cont):
    for i in range(length-1):
        swapped = False
        for j in range(length-i-1):
            key1 = list_pos_cont[j][0]
            coor1 = list_pos_cont[j][1]
            key2 = list_pos_cont[j+1][0]
            coor2 = list_pos_cont[j+1][1]
            if coor1[0] > coor2[0]:
                swapped = True
                list_pos_cont[j] = (key2, coor2)
                list_pos_cont[j+1] = (key1, coor1)
        if not swapped:
            return list_pos_cont


# plot the degree histogram of the graph
def plot_degree_dist(G, m, color):
    plt.subplot(2,2,m)
    degree_hist = nx.degree_histogram(G)
    degree_hist = np.array(degree_hist, dtype = float)
    degree_prob = degree_hist/G.number_of_nodes()
    plt.bar(np.arange(degree_prob.shape[0]), degree_prob, color = color)
    plt.plot(np.arange(degree_prob.shape[0]), degree_prob, '-', color = color)

    plt.tick_params(axis='both', which = 'major', labelsize = 8)
    plt.ylabel('Frequency')
    plt.title('Degree Distribution')


# plot the probablity mass function of the graph
def probability_mass_fct(G, SC):
```

```
    plt.subplot(2,2,2)
    plt.title('Probability Mass Function')
    deg_G = dict(G.degree()).values()
    deg_distri_G = collections.Counter(deg_G)
    deg_SC = dict(SC.degree()).values()
    deg_distri_SC = collections.Counter(deg_SC)

    x_G = []
    y_G = []
    for i_G in sorted(deg_distri_G):
        x_G.append(i_G)
        y_G.append(deg_distri_G[i_G]/len(G))

    x_SC = []
    y_SC = []
    for i_SC in sorted(deg_distri_SC):
        x_SC.append(i_SC)
        y_SC.append(deg_distri_SC[i_SC]/len(SC))

    print(max(x_SC), max(x_G))

    plt.plot(x_G, y_G, linewidth = 2, label = 'scale-free network')
    plt.plot(x_SC, y_SC, linewidth = 2, label = 'barrel-shaped network')
    plt.legend(loc = 0, shadow = True, ncol = 1, facecolor = '#F5F5F5')

    plt.xlabel('Degree')
    plt.ylabel('P(X = x)')

    plt.yscale('linear')
    plt.xscale('linear')
    plt.xticks(fontsize=10)
    plt.yticks(fontsize=10)




# multiple linear regression analysis
def multi_linear_reg(perform_SC_b, corr_x, corr_y, corr_z):
    y = np.array(list(perform_SC_b.values()), dtype = 'float')
    x = np.array([list(corr_x), list(corr_y), list(corr_z)], dtype = 'float')
    ones = np.ones(len(x[0]))
    X = sm.add_constant(np.column_stack((x[0], ones)))
    for element in x[1:]:
        X = sm.add_constant(np.column_stack((element, X)))
    results = sm.OLS(y, X, missing = 'drop').fit()
    return results
```

```
# rearranging the edges for dual sourcing ata diversified suppliers and place␣
 ↪then into a new Digraph
def dual_sourcing_at_diversified_suppliers(SC_a, pos_cont, node_list_pos_cont):
    SC_h = nx.DiGraph()
    last_key = 'OEM'
    for edge in SC_a.edges:
        coord = (pos_cont[edge[0]][0] + 1, 0)
        key = [key for key, value in pos_cont.items() if value == coord]
        key_2 = []
        while(key == []):
            coord = (coord[0] + 1, 0)
            key = [key for key, value in pos_cont.items() if value == coord]
        if(key != last_key):
            key_0 = [node_list_pos_cont[j-1] for j, val in␣
 ↪enumerate(node_list_pos_cont) if val == key[0]]
            new_x = pos_cont[key_0[0]][0]
            key_1 = [key for key, value in pos_cont.items() if value == (new_x␣
 ↪- 1, 0)]
            while(len(key_1) == 0):
                new_x = new_x - 1
                key_1 = [key for key, value in pos_cont.items() if value ==␣
 ↪(new_x - 1, 0)]
            key_2 = [key for key, value in pos_cont.items() if value[0] ==␣
 ↪pos_cont[key_1[0]][0] and value[1] != 0]
            if len(key_2) == 0:
                SC_h.add_edge(key_1[0], key[0])
            elif len(key_2) > 0:
                SC_h.add_edge(key_2[-1], key[0])
            last_key = key
    return SC_h


# calculating the criticality which need to be faster but it still took a long␣
 ↪time to go through the paths FOR-loop
def cal_criticality(SC_g, SC_f, critical_node, pos_cont, all_path, orig_path):
    start_time = time.time()
    criticality_1st = 0

    if(sys.getsizeof(all_path) == 0):
        return criticality_1st

    # ------------------------------need to  be faster
    start_t = time.time()
    criticality = []
    for idx in range(len(critical_node)-1):
```

```
        path_crit = []
        if(critical_node[idx+1] not in SC_f.neighbors(critical_node[idx])):
            source = critical_node[idx]
            target = critical_node[idx+1]
            p = [len(p[1:-1]) for p in nx.all_simple_paths(SC_g, source =␣
 ↪source, target = target)]
            if not ((len(p) == 1) and (p[0] == 0)):
                for l in p:
                    add_var = math.pow(rou, l)
                    path_crit.append(add_var)
                criticality.append(path_crit)
    end_t = time.time()

    criticality_crit = math.pow(rou, len(critical_node))

    crit = []
    for n in range(len(criticality)):
        crit_m = 0
        crit_n = criticality[n]
        var1 = []
        var2 = []
        var2.append(1)
        par2 = 0
        for m in range(len(crit_n)):
            var2.append(1-crit_n[m])
            prod = multiply(*(var2[:-1])) * crit_n[m]
            var1.append(prod)
        crit.append(var1)


    crit_1_ord = []
    for i in range(len(crit)):
        summ = sum(crit[i])
        crit_1_ord.append(summ)

    criticality_1st = criticality_crit * multiply(*crit_1_ord)

    end_time = time.time()

    return criticality_1st


# parrallel supply chain before calculate the centrality metrics
def remove_parallel_node(DC):
    DC_n = DC.copy()
    for data in DC.items():
        if('|' in data[0]) and ('S' not in data[0]) and ('R' not in data[0]):
```

```
            idx_of_bar = data[0].index('|')
            if(data[0][idx_of_bar-1] == data[0][idx_of_bar+1]):
                DC_n.pop(data[0])
        elif(data[0] == 'Start') or (data[0] == 'OEM'):
            DC_n.pop(data[0])
    return DC_n


# write the OLS regression analysis results into txt file
def write_into_txt(p_e, p_ln, fit_network): #, reg_result_perf,␣
 ↪reg_result_crit):
    with open('./BA_data/Gnm_test.txt', 'a+') as file:
        data =␣
 ↪'-------------------------------------------------------------------------------------\n{
 ↪format(num_of_fig)
        data = data +␣
 ↪'-------------------------------------------------------------------------------------\n\
        data = data + 'R:{}\nS:{}\nC:{}\nM:{}\npR:{}\npS:{}\npC:{}\npM:{}\nE_12:
 ↪{}\nE_23:{}\nE_34:{}\nrou:{}\n\n'.format(R, S, C, M, pR, pS, pC, pM, E_12,␣
 ↪E_23, E_34, rou)
        data = data + 'Comparing power-law with exponential Distribution...
 ↪{}\n'.format(p_e)
        data = data + 'Comparing power-law with lognormal Distribution...
 ↪{}\n\n'.format(p_ln)
        data = data + 'alpha:{}, xmin:{}\n\n'.format(fit_network.power_law.
 ↪alpha, fit_network.power_law.xmin)
        data = data +␣
 ↪'================================================================================'
        # data = data + '\t\t\t\tPerformance'
        # data = data + str(reg_result_perf.summary2(yname = 'Performance',␣
 ↪xname = ['DC', 'BC', 'EC', 'const']))
        # data = data + '\n\n'
        # data = data +␣
 ↪'================================================================================'
        # data = data + '\t\t\t\tCriticality'
        # data = data + str(reg_result_crit.summary2(yname = '1st Criticality',␣
 ↪xname = ['DC', 'BC', 'EC', 'const']))
        # data = data + '\n\n'
        data = data +␣
 ↪'-------------------------------------------------------------------------------------\n{
 ↪format(num_of_fig)
        data = data +␣
 ↪'-------------------------------------------------------------------------------------\n\
        print("Written successfully...", num_of_fig)
        file.write(data)
    file.close()
```

```
# write the OLS regression analysis results as table into xlsx file
def write_into_xlsx(reg_result_perf, reg_result_crit, fit_network, sheet_name,␣
 ↪p_e, p_ln, idx):
    coefficients = list(reg_result_perf.params)
    p_values = list(reg_result_perf.pvalues)
    r_squared = reg_result_perf.rsquared
    std_err = list(reg_result_perf.bse)
    t_values = list(reg_result_perf.tvalues)
    cond_no = reg_result_perf.condition_number

    wb = opxl.load_workbook('./BA_data/OLS_SUMMARY.xlsx')
    ws = wb.active
    ws.title = sheet_name
    ws.cell(row = idx + 1, column = 1, value = num_of_fig)
    ws.cell(row = idx + 1, column = 2, value = 'Performance - Regression␣
 ↪Analysis')
    ws.cell(row = idx + 2, column = 2, value = 'Coefficients')
    ws.cell(row = idx + 2, column = 3, value = 'Standartd Error')
    ws.cell(row = idx + 2, column = 4, value = 't-values')
    ws.cell(row = idx + 2, column = 5, value = 'P-values')
    ws.cell(row = idx + 2, column = 6, value = 'R-squared')
    ws.cell(row = idx + 4, column = 6, value = 'Cond.No.')

    ws.cell(row = idx + 2, column = 1, value = 'alpha')
    ws.cell(row = idx + 3, column = 1, value = fit_network.power_law.alpha)
    ws.cell(row = idx + 4, column = 1, value = 'xmin')
    ws.cell(row = idx + 5, column = 1, value = fit_network.power_law.xmin)

    for i in range(len(coefficients)):
        ws.cell(row = idx + i+3, column = 2, value = coefficients[i])
        ws.cell(row = idx + i+3, column = 3, value = std_err[i])
        ws.cell(row = idx + i+3, column = 4, value = t_values[i])
        ws.cell(row = idx + i+3, column = 5, value = p_values[i])
    ws.cell(row = idx + 3, column = 6, value = r_squared)
    ws.cell(row = idx + 5, column = 6, value = cond_no)

    # ====================== #

    coefficients = list(reg_result_crit.params)
    p_values = list(reg_result_crit.pvalues)
    r_squared = reg_result_crit.rsquared
    std_err = list(reg_result_crit.bse)
    t_values = list(reg_result_crit.tvalues)

    # ws.cell(row = idx + 7, column = 1, value = num_of_fig)
```

```
    ws.cell(row = idx + 1, column = 8, value = 'Criticality - Regression␣
 ↪Analysis')
    ws.cell(row = idx + 2, column = 8, value = 'Coefficients')
    ws.cell(row = idx + 2, column = 9, value = 'Standartd Error')
    ws.cell(row = idx + 2, column = 10, value = 't-values')
    ws.cell(row = idx + 2, column = 11, value = 'P-values')
    ws.cell(row = idx + 2, column = 12, value = 'R-squared')
    ws.cell(row = idx + 4, column = 12, value = 'Cond.No.')

    for i in range(len(coefficients)):
        ws.cell(row = idx + i+3, column = 8, value = coefficients[i])
        ws.cell(row = idx + i+3, column = 9, value = std_err[i])
        ws.cell(row = idx + i+3, column = 10, value = t_values[i])
        ws.cell(row = idx + i+3, column = 11, value = p_values[i])
    ws.cell(row = idx + 3, column = 12, value = r_squared)
    ws.cell(row = idx + 5, column = 12, value = cond_no)

    ws.cell(row = idx + 7, column = 2, value = 'PL or E')
    ws.cell(row = idx + 7, column = 3, value = p_e)
    ws.cell(row = idx + 7, column = 8, value = 'PL or LN')
    ws.cell(row = idx + 7, column = 9, value = p_ln)

    print("Written successfully...", num_of_fig)

    wb.save('./BA_data/OLS_SUMMARY.xlsx')

def main(args=None):

    SC, T1, T2, T3, T4 = TRANS(R, S, C, M, pR, pS, pC, pM, E_12, E_23, E_34)
    SC_a = SC.copy()
    SC_b = SC.copy()
    SC_b.add_edges_from(T2.edges, layer = 2)
    SC_b.add_edges_from(T3.edges, layer = 1)
    SC_b.add_edges_from(T4.edges, layer = 0)


    pos = POSITION(SC, 0)

    # SC_a contains only the edges conneting two nodes in one tier
    # SC contains only the edges conneting two nodes in different tiers
    # SC_b contains only the edges conneting all nodes


    mapping_perform = pos.copy()
    mapping_perform = CRITICAL_1(mapping_perform, SC_a, T1, T2, T3, T4)
    mapping_perform = CRITICAL_1(mapping_perform, SC_b, T1, T2, T3, T4)
    perform_SC_b = mapping_perform
```

```
    mapping_perform = CRITICAL_1(mapping_perform, SC, T1, T2, T3, T4)



    edge = SC_b.edges
    edgelist= list(edge)
    length = len(edgelist)
    SC_a.remove_edges_from(edge)
    for i in range(length):
        edge_uv = edgelist[i]
        node_u = edge_uv[0]
        node_v = edge_uv[1]
        if node_u[0] == node_v[0]:
            SC_a.add_edge(node_u, node_v)
            SC_b.remove_edge(node_u, node_v)


    R_node_list = []
    S_node_list = []
    C_node_list = []
    M_node_list = []
    for u in SC.nodes:
        if("R" in u):
            R_node_list.append(u)
        elif("S" in u):
            S_node_list.append(u)
        elif("C" in u):
            C_node_list.append(u)
        elif("M" in u):
            M_node_list.append(u)

    adj_nodes = []
    for edge1 in SC_a.edges: # for edges in one tier
        node1 = edge1[0]
        node2 = edge1[1]
        par = 0
        for edge2 in SC.edges: # for edges in different tiers
            node3 = edge2[0]
            node4 = edge2[1]
            if(node1 == node3):
                # if the starting point of two edges are the same
                # then connect the other nodes in the same tier into the next␣
↪tier
                SC.add_edge(node2, node4)
                SC_b.add_edge(node2, node4)
            elif(node2 == node3):
                SC.add_edge(node1, node4)
```

```
                SC_b.add_edge(node1, node4)
    for edge in SC_b.edges:
        adj_nodes.append(edge[0])
        adj_nodes.append(edge[1])

    SC_c = SC_b.copy()
    edges = SC.edges()
    SC_d = SC_c.copy()
    # ADDING AN EXTRA POSSIBILITY CONTAINS ALL NODES ON THE RIGHT HANDSIDE
    S_C_M_node_list = []
    S_C_M_node_list.extend(S_node_list)
    S_C_M_node_list.extend(C_node_list)
    S_C_M_node_list.extend(M_node_list)
    C_M_node_list = []
    C_M_node_list.extend(C_node_list)
    C_M_node_list.extend(M_node_list)
    for node in SC_a.nodes:
        SC_d, SC_b = ADD_EDGE_TO_NEXT_TIER(node, "R", S+C+M, S_C_M_node_list,␣
↪SC_d, SC_b)
        SC_d, SC_b = ADD_EDGE_TO_NEXT_TIER(node, "S", C+M, C_M_node_list, SC_d,␣
↪SC_b)
        SC_d, SC_b = ADD_EDGE_TO_NEXT_TIER(node, "C", M, M_node_list, SC_d,␣
↪SC_b)
    SC_d.remove_edges_from(edges)



    SC_ad = nx.adjacency_matrix(SC_a)
    SC_A = SC_ad.todense()
    SC_A_flip180 = SC_A.reshape(SC_A.size)
    SC_A_flip180 = SC_A_flip180[::-1]
    SC_A_flip180 = SC_A_flip180.reshape((R+S+C+M, R+S+C+M))

    SC_A_R = [row[:R] for row in SC_A_flip180[:R]]
    SC_A_R = np.array(SC_A_R)


    plt.figure(1)
    nx.draw_networkx_edges(SC, pos = pos)
    nx.draw_networkx_edges(SC_a, pos = pos, edge_color = 'red', connectionstyle␣
↪= "arc3, rad = 0.5", arrows = True)
    nx.draw_networkx_edges(SC_d, pos = pos, edge_color = 'blue',␣
↪connectionstyle = "arc3, rad = -0.5", arrows = True)
    nx.draw_networkx_nodes(SC, pos = pos, node_size = 500, node_color =␣
↪'black', node_shape = 'o')
```

```
    nx.draw_networkx_nodes(SC, pos = pos, node_size = 450, node_color = 'w',␣
↪node_shape = 'o')
    nx.draw_networkx_labels(SC, pos = pos, font_size = 10, font_color = 'black')


    pos_cont = {}
    # check how many nodes are adjacent to another node in its tier
    node_list_a = ADJ_IN_ONE_TIER(SC_a)

    # number of node in one Tier that is adjacent
    number_of_node_R, number_of_node_S, number_of_node_C =␣
↪NUM_OF_NODE_AIOT(node_list_a)

    idx_R = number_of_node_R
    idx_S = number_of_node_R + number_of_node_S
    idx_C = number_of_node_R + number_of_node_S + number_of_node_C

    node_list_R = NODE_LIST(SC, "R")
    node_list_R = node_list_R[::-1]
    node_list_S = NODE_LIST(SC, "S")
    node_list_S = node_list_S[::-1]
    node_list_C = NODE_LIST(SC, "C")
    node_list_C = node_list_C[::-1]
    node_list_M = NODE_LIST(SC, "M")
    node_list_M = node_list_M[::-1]

    pos_cont2 = {}
    list_on_2nd_line = []
    minus_par = 0
    SC_e = nx.DiGraph()
    SC_e.add_nodes_from(SC_a)
    list_on_2nd_line, SC_e, pos_cont2 = List_at_x_axis(node_list_R,␣
↪node_list_S, SC_a, SC_d, SC_e, minus_par, pos_cont2, list_on_2nd_line)

    print(pos_cont2)

    # turn the network in to Matrix and focus only on Tier R
    SC_A_R, SC_A_S, SC_A_C = NETW_INTO_MATRIX(SC_a, R, S, C)

    pos = {}
    pos_cont = {}
    pos_cont_new = {}
    pos_cont, node_list_rdy, IDX = CONTRACTED_POS(pos, SC_A_R, R, node_list_R,␣
↪list_on_2nd_line)
    pos_cont = CONTRACTED_POS_CONT_AIOT(pos_cont, pos_cont_new, IDX,␣
↪node_list_R, node_list_rdy, list_on_2nd_line, "R")
```

```
    pos = {}
    pos_cont, node_list_rdy, IDX = CONTRACTED_POS(pos, SC_A_S, S, node_list_S,␣
↪list_on_2nd_line)
    pos_cont = CONTRACTED_POS_CONT_AIOT(pos_cont, pos_cont_new, IDX,␣
↪node_list_S, node_list_rdy, list_on_2nd_line, "S")

    pos = {}
    pos_cont, node_list_rdy, IDX = CONTRACTED_POS(pos, SC_A_C, C, node_list_C,␣
↪list_on_2nd_line)
    pos_cont = CONTRACTED_POS_CONT_AIOT(pos_cont, pos_cont_new, IDX,␣
↪node_list_C, node_list_rdy, list_on_2nd_line, "C")
    pos_cont = CONTRACTED_POS_CONT_FILL(pos_cont, SC)

    '''

    SC_f = nx.DiGraph()
    list_pos_cont = list(pos_cont.items())
    length = len(list_pos_cont)
    list_pos_cont1 = list_pos_cont.copy()
    # delete the node not at the same line
    for l in range(length):
        if(list_pos_cont[l][1][1] > 0):
            list_pos_cont1.remove(list_pos_cont[l])

    list_pos_cont = list_pos_cont1
    length = len(list_pos_cont)
    list_pos_cont = bubble_sort(length, list_pos_cont)
    # adding edges between nodes at y=0 axis
    list_at_0 = [list_pos_cont[l] for l, val in enumerate(list_pos_cont) if␣
↪val[1][1] == 0]
    list_under_0 = [list_pos_cont[l] for l, val in enumerate(list_pos_cont) if␣
↪val[1][1] < 0]
    for l in range(len(list_pos_cont)):
        node1 = list_pos_cont[l][0]
        if(pos_cont[node1][0] < R+S+C+M-1):
            node2 = list_pos_cont[l+1][0]
            SC_f.add_edge(node1, node2)


    pos_cont = dict(list_pos_cont)
    pos_cont.update(pos_cont2)

    node_list_pos_cont = []
    for i in list_pos_cont:
        node_list_pos_cont.append(i[0])
    len_node_list_pos_cont = len(node_list_pos_cont)
```

```
   SC_h = dual_sourcing_at_diversified_suppliers(SC_a, pos_cont,␣
↪node_list_pos_cont)

   pos_cont, SC_e = parallel_supply_chain(node_list_R, node_list_S, SC_a,␣
↪SC_d, node_list_pos_cont, len_node_list_pos_cont, SC_e, pos_cont, pos_cont2)

   SC_g = SC_e.copy()
   SC_g.add_edges_from(SC_f.edges)
   SC_g.add_edges_from(SC_h.edges)
   SC_g.add_node('Start')
   pos_cont['Start'] = (-2, 0)
   start_node = list_pos_cont[0][0]

   SC_f.add_edge('Start', start_node)
   SC_g.add_edge('Start', start_node)
   node_list_pos_cont.insert(0, 'Start')
   SC_g.add_node('OEM')
   pos_cont['OEM'] = (R+S+C+M+1, 0)
   node_list_pos_cont.append('OEM')
   end_node = [n for n in SC_g.nodes if pos_cont[n][0] == (R+S+C+M-1)]
   for n in end_node:
       SC_g.add_edge(n, 'OEM')
       SC_f.add_edge(n, 'OEM')
   '''
   '''
   orig_path = nx.all_simple_paths(SC_f, source='Start', target = 'OEM')
   all_path = nx.all_simple_paths(SC_g, source='Start', target = 'OEM')
   orig_path = orig_path.__next__()
   critical_node = comparing_critical_node(all_path, orig_path)
   critical_node = [node for node in node_list_pos_cont if node in␣
↪critical_node]

   criticality_1st_graph = cal_criticality(SC_g, SC_f, critical_node,␣
↪pos_cont, all_path, orig_path)


   criticality_matrix = {}
   SC_i = SC_g.copy()
   node_SC_i = list(SC_i.nodes())
   start_time = time.time()
   for idx_i in range(len(SC_i)-2):
       node_to_remove = node_SC_i[idx_i]
       if('|' in node_to_remove) and ('S' not in node_to_remove) and ('R' not␣
↪in node_to_remove):
           idx_of_bar = node_to_remove.index('|')
           if(node_to_remove[idx_of_bar-1] == node_to_remove[idx_of_bar+1]):
```

```
                continue

        if(node_to_remove in critical_node):
            criticality_1st_without_i = 0
        else:
            SC_i.remove_node(node_to_remove)
            all_path = nx.all_simple_paths(SC_i, source='Start', target = 'OEM')
            orig_path = nx.all_simple_paths(SC_f, source='Start', target =␣
↪'OEM')
            orig_path = orig_path.__next__()
            if(node_to_remove not in orig_path):
                orig_path = all_path.__next__()
            criticality_1st_without_i = cal_criticality(SC_i, SC_f,␣
↪critical_node, pos_cont, all_path, orig_path)
        diff = criticality_1st_graph - criticality_1st_without_i
        criticality_matrix[node_to_remove] = diff
        SC_i = SC_g.copy()
    end_time = time.time()


    # plot the contracte production network
    plt.figure(6)
    nx.draw_networkx_edges(SC_h, pos = pos_cont, edge_color = 'blue',␣
↪connectionstyle = "arc3, rad = 1.0", arrows = True)
    nx.draw_networkx_edges(SC_e, pos = pos_cont, edge_color = 'green',␣
↪connectionstyle = "arc3, rad = 0.3", arrows = True)
    nx.draw_networkx_edges(SC_f, pos = pos_cont, edge_color = 'black', arrows =␣
↪True)
    nx.draw_networkx_nodes(SC_g, pos = pos_cont, node_size = 500, node_color =␣
↪'black', node_shape = 'o')
    nx.draw_networkx_nodes(SC_g, pos = pos_cont, node_size = 450, node_color =␣
↪'w', node_shape = 'o')
    nx.draw_networkx_labels(SC_g, pos = pos_cont, font_size = 10, font_color =␣
↪'black')


    # ========================================================================= #
    # if necessary, here is the Performance for Contracted Production Network
    SC_k = SC_g.copy()
    SC_k.remove_node('Start')
    SC_k.remove_node('OEM')
    SC_j = SC_k.copy()
    mapping_perform = pos_cont
    mapping_perform.pop('Start')
    mapping_perform.pop('OEM')
    for node in SC_k.nodes:
```

```
        if('|' in node) and ('S' not in node) and ('R' not in node):
            idx_of_bar = node.index('|')
            if(node[idx_of_bar-1] == node[idx_of_bar+1]):
                SC_j.remove_node(node)
                mapping_perform.pop(node)

    mapping_perform = CRITICAL_1(mapping_perform, SC_j, T1, T2, T3, T4)
    perform_SC_g = mapping_perform
    # ========================================================================== #
    #                         Betweennesss based on SC_b                         #
    # ========================================================================== #
    plt.figure(5, figsize = (16, 6))
    colors = list('rgbcmyk')
    DC = nx.algorithms.degree_centrality(SC_b)
    BC = nx.algorithms.betweenness_centrality(SC_b)
    EC = nx.algorithms.eigenvector_centrality(SC_b, max_iter = 1000)
    HC = nx.hits(SC_b)[1]

    # plt.subplot(2,2,1)
    # plt.title('Degree Centrality')
    for data in DC.items():
        x = data[0]
        y = data[1]
        if(y == 0):
            y = y + 10e-5
        # plt.scatter(x, y, color = 'lightblue')

    # plt.subplot(2,2,2)
    # plt.title('Betweenness Centrality')
    for data in BC.items():
        x = data[0]
        y = data[1]
        if(y == 0):
            y = y + 10e-5
        # plt.scatter(x, y, color = 'lightblue')

    # plt.subplot(2,2,3)
    # plt.title('Eigenvector Centrality')
    for data in EC.items():
        x = data[0]
        y = data[1]
        if(y == 0):
            y = y + 10e-5
        # plt.scatter(x, y, color = 'lightblue')
    plt.subplot(1,2,1)
    plt.title('Correlation Analysis - Centrality - multi-partite Network')
    corr_x = np.array(list(DC.values()))
```

```
    corr_y = np.array(list(BC.values()))
    corr_z = np.array(list(EC.values()))
    corr_w = np.array(list(HC.values()))

    r_DC_BC = np.corrcoef(corr_x, corr_y)[0,1]
    r_DC_EC = np.corrcoef(corr_x, corr_z)[0,1]
    r_BC_EC = np.corrcoef(corr_y, corr_z)[0,1]
    r_DC_HC = np.corrcoef(corr_x, corr_w)[0,1]
    r_BC_HC = np.corrcoef(corr_y, corr_w)[0,1]
    r_EC_HC = np.corrcoef(corr_z, corr_w)[0,1]
    plt.text(0.1, 0.5, 'DC')
    plt.text(0.1, 0.3, 'BC')
    plt.text(0.1, 0.1, 'EC')
    plt.text(0.3, 0.7, 'BC')
    plt.text(0.5, 0.7, 'EC')
    plt.text(0.7, 0.7, 'HC')
    plt.text(0.275, 0.5, '%.2f'%r_DC_BC)
    plt.text(0.475, 0.5, '%.2f'%r_DC_EC)
    plt.text(0.675, 0.5, '%.2f'%r_DC_HC)
    plt.text(0.475, 0.3, '%.2f'%r_BC_EC)
    plt.text(0.675, 0.3, '%.2f'%r_BC_HC)
    plt.text(0.675, 0.1, '%.2f'%r_EC_HC)
    # plt.savefig('./BA_data/Centrality/network/
↪centrality_correlation_1x_CPN_{}'.format(num_of_fig))

    # print("====length perform_Network: ", len(perform_SC_b),␣
↪type(perform_SC_b), "====")
    # reg_result_perf = multi_linear_reg(perform_SC_b, corr_x, corr_y, corr_z)
    # print(reg_result_perf.summary())
    # print("====length of criticality_CPN: ", len(criticality_matrix),␣
↪type(criticality_matrix), "====")
    # reg_result_crit = multi_linear_reg(criticality_matrix, corr_x, corr_y,␣
↪corr_z)
    # print(reg_result_crit.summary())


    # ========================================================================== #
    #                          Betweennesss based on SC_g                        #
    # ========================================================================== #
    SC_i = SC_g.copy()

    # plt.figure(2, figsize = (12, 6))
    colors = list('rgbcmyk')
    DC = nx.algorithms.degree_centrality(SC_g)
    DC = remove_parallel_node(DC)
    BC = nx.algorithms.betweenness_centrality(SC_g)
    BC = remove_parallel_node(BC)
```

```
    EC = nx.algorithms.eigenvector_centrality(SC_g, max_iter = 1000000)
    EC = remove_parallel_node(EC)
    HC = nx.hits(SC_g, max_iter = 100000)[1]
    HC = remove_parallel_node(HC)

    # print("DC | ", DC, '\n')
    # plt.subplot(2,2,1)
    # plt.title('Degree Centrality')
    for data in DC.items():
        x = data[0]
        y = data[1]
        if(y == 0):
            y = y + 10e-5
        # plt.scatter(x, y, color = 'lightblue')
    # print("BC | ", BC)
    # plt.subplot(2,2,2)
    # plt.title('Betweenness Centrality')
    for data in BC.items():
        x = data[0]
        y = data[1]
        if(y == 0):
            y = y + 10e-5
        # plt.scatter(x, y, color = 'lightblue')
    print("EC | ", EC)
    # plt.subplot(2,2,3)
    # plt.title('Eigenvector Centrality')
    for data in EC.items():
        x = data[0]
        y = data[1]
        if(y == 0):
            y = y + 10e-5
        # plt.scatter(x, y, color = 'lightblue')
    plt.subplot(1,2,2)
    plt.title('Correlation Analysis - Centrality - Contracted Network')

    corr_x = np.array(list(DC.values()))
    corr_y = np.array(list(BC.values()))
    corr_z = np.array(list(EC.values()))
    corr_w = np.array(list(HC.values()))
    # print(len(corr_x), len(corr_y), type(corr_x))
    r_DC_BC = np.corrcoef(corr_x, corr_y)[0,1]
    r_DC_EC = np.corrcoef(corr_x, corr_z)[0,1]
    r_BC_EC = np.corrcoef(corr_y, corr_z)[0,1]
    r_DC_HC = np.corrcoef(corr_x, corr_w)[0,1]
    r_BC_HC = np.corrcoef(corr_y, corr_w)[0,1]
    r_EC_HC = np.corrcoef(corr_z, corr_w)[0,1]
    plt.text(0.1, 0.5, 'DC')
```

```
    plt.text(0.1, 0.3, 'BC')
    plt.text(0.1, 0.1, 'EC')
    plt.text(0.3, 0.7, 'BC')
    plt.text(0.5, 0.7, 'EC')
    plt.text(0.7, 0.7, 'HC')
    plt.text(0.275, 0.5, '%.2f'%r_DC_BC)
    plt.text(0.475, 0.5, '%.2f'%r_DC_EC)
    plt.text(0.675, 0.5, '%.2f'%r_DC_HC)
    plt.text(0.475, 0.3, '%.2f'%r_BC_EC)
    plt.text(0.675, 0.3, '%.2f'%r_BC_HC)
    plt.text(0.675, 0.1, '%.2f'%r_EC_HC)
    plt.savefig('./BA_data/Centrality/centrality_correlation_1x_CPN_{}'.
↪format(num_of_fig))

    '''
    # STILL NEED TO RESET THE CALCULATION FORMULAR FOR CRITICALITY

    # test if the network fit the power law Distribution
    # but we should only use it in a network with large amount of nodes
    # we would like to use it for a small test to find
    # a valuable probability for the poisson Distribution
    # SC.add_edges_from(SC_d.edges)
    plt.figure(3, figsize = (15, 10))


    degree_sequence = sorted([d for n, d in SC_b.degree()], reverse = True)
    degree_count = nx.degree_histogram(SC_b)

    fit_network = powerlaw.Fit(degree_sequence)

    # goodness-of-fit test power law distribution with another alternative␣
↪statistic distribution
    p_comp_e = fit_network.distribution_compare('power_law', 'exponential')
    if (p_comp_e[0] > 0) and (p_comp_e[1] >= 0.5):
        p_e = 'Power Law'
    elif (p_comp_e[0] > 0) and (p_comp_e[1] < 0.5):
        p_e = 'better Power Law, but not a certain answer'
    elif (p_comp_e[0] < 0) and (p_comp_e[1] >= 0.5):
        p_e = 'Exponential'
    elif (p_comp_e[0] < 0) and (p_comp_e[1] <0.5):
        p_e = 'better Exponential, but not a certain answer'

    p_comp_ln = fit_network.distribution_compare('power_law', 'lognormal')
    if (p_comp_ln[0] > 0) and (p_comp_ln[1] >= 0.5):
        p_ln = 'Power Law'
    elif(p_comp_ln[0] > 0) and (p_comp_ln[1] < 0.5):
        p_ln = 'better Power Law, but not a certain answer'
```

```
    elif (p_comp_ln[0] < 0) and (p_comp_ln[1] >= 0.5):
        p_ln = 'Lognormal'
    elif (p_comp_ln[0] < 0) and (p_comp_ln[1] < 0.5):
        p_ln = 'better Lognormal, but not a certain answer'

    write_into_txt(p_e, p_ln, fit_network)


    plt.subplot(2,2,1)
    plt.title('Power Law PDF(probability density function) Fitting')
    fit_network.plot_pdf(color = 'b', marker = 'o', linewidth = 2, label =␣
↪'barrel-shaped network')
    plt.legend(loc = 0, shadow = True, ncol = 1, facecolor = '#F5F5F5')
    plt.xlabel('Observed Value x')
    plt.ylabel('Probability Density P(x)')

    plt.subplot(2,2,3)
    plt.title('Power Law CCDF Fitting')
    fit_network.plot_ccdf(color = 'b', marker = 'o', linewidth = 2, label =␣
↪'barrel-shaped network')
    plt.legend(loc = 0, shadow = True, ncol = 1, facecolor = '#F5F5F5')
    plt.xlabel('Observed Value x')
    plt.ylabel('Probability Density P(X>x)')

    plt.subplot(2,2,4)
    plt.text(0.01, 0.9, "Checking if Poisson Distribution...")
    plt.text(0.01, 0.8, "Variance :%s"%np.var(degree_sequence))
    plt.text(0.01, 0.7, "Mean :%s"%np.mean(degree_sequence))
    plt.text(0.01, 0.55, 'Comparing with exponential Distribution...')
    plt.text(0.01, 0.45, '%s'%p_e)
    plt.text(0.01, 0.35, 'Comparing with lognormal Distribution...')
    plt.text(0.01, 0.25, '%s'%p_ln)
    plt.text(0.01, 0.1, 'R:{}, S:{}, C:{}, M:{}, E_12:{}, E_23:{}, E_34:{}'.
↪format(R, S, C, M, E_12, E_23, E_34))


    G = nx.scale_free_graph(R+S+C+M, alpha = 0.5, beta = 0.3, gamma = 0.2)
    probability_mass_fct(G, SC_b)

    degree_sequence = sorted([d for n, d in G.degree()], reverse = True)
    degree_count = nx.degree_histogram(G)

    # plt.figure(4)
    fit = powerlaw.Fit(degree_sequence)
    print(fit.power_law.alpha)
    print(fit.power_law.xmin)
    plt.subplot(2,2,1)
```

```
   plt.title('Power Law PDF Fitting')
   fit.plot_pdf(color = 'r', marker = 'o', linewidth = 2, label = 'scale-free␣
↪network')
   plt.xticks(fontsize=10)
   plt.legend(loc = 0, shadow = True, ncol = 1, facecolor = '#F5F5F5')
   plt.yscale('linear')
   plt.xscale('linear')
   plt.xticks(fontsize=10)
   plt.yticks(fontsize=10)
   plt.subplot(2,2,3)
   plt.title('Power Law CCDF Fitting')
   fit.plot_ccdf(color = 'r', marker = 'o', linewidth = 2, label = 'scale-free␣
↪network')
   plt.legend(loc = 0, shadow = True, ncol = 1, facecolor = '#F5F5F5')
   plt.yscale('linear')
   plt.xscale('linear')
   plt.xticks(fontsize=10)
   plt.yticks(fontsize=10)
   plt.savefig('./BA_data/Powerlaw/power_law_distribution_1x_CPN_{}'.
↪format(num_of_fig))
   '''
   print("====length perform_CPN: ", len(perform_SC_g), type(perform_SC_g),␣
↪"====")
   reg_result_perf = multi_linear_reg(perform_SC_g, corr_x, corr_y, corr_z)
   print(reg_result_perf.params)
   print(reg_result_perf.summary2(yname = 'Performance', xname = ['DC', 'BC',␣
↪'EC', 'const']))

   print("====length of criticality_CPN: ", len(criticality_matrix),␣
↪type(criticality_matrix), "====")
   reg_result_crit = multi_linear_reg(criticality_matrix, corr_x, corr_y,␣
↪corr_z)
   print(reg_result_crit.summary2(yname = '1st Criticality', xname = ['DC',␣
↪'BC', 'EC', 'const']))

   if (p_e == 'Power Law' or p_e == 'better Power Law, but not a certain␣
↪answer') and ((p_ln == 'Power Law' or p_ln == 'better Power Law, but not a␣
↪certain answer')):
       sheet_name = 'OLS_SUMMARY'

       wb = opxl.load_workbook('./BA_data/OLS_SUMMARY.xlsx')
       sheet = wb[sheet_name]
       idx = len(list(sheet.rows))

       write_into_xlsx(reg_result_perf, reg_result_crit, fit_network,␣
↪sheet_name, p_e, p_ln, idx)
```

```
    else:
        print(" FAILED ...", num_of_fig)
    '''

    # jls_extract_var = plt
    # jls_extract_var.show()
    # plt.savefig('labels.png')


if __name__ == "__main__":
    main()
```

[ ]: