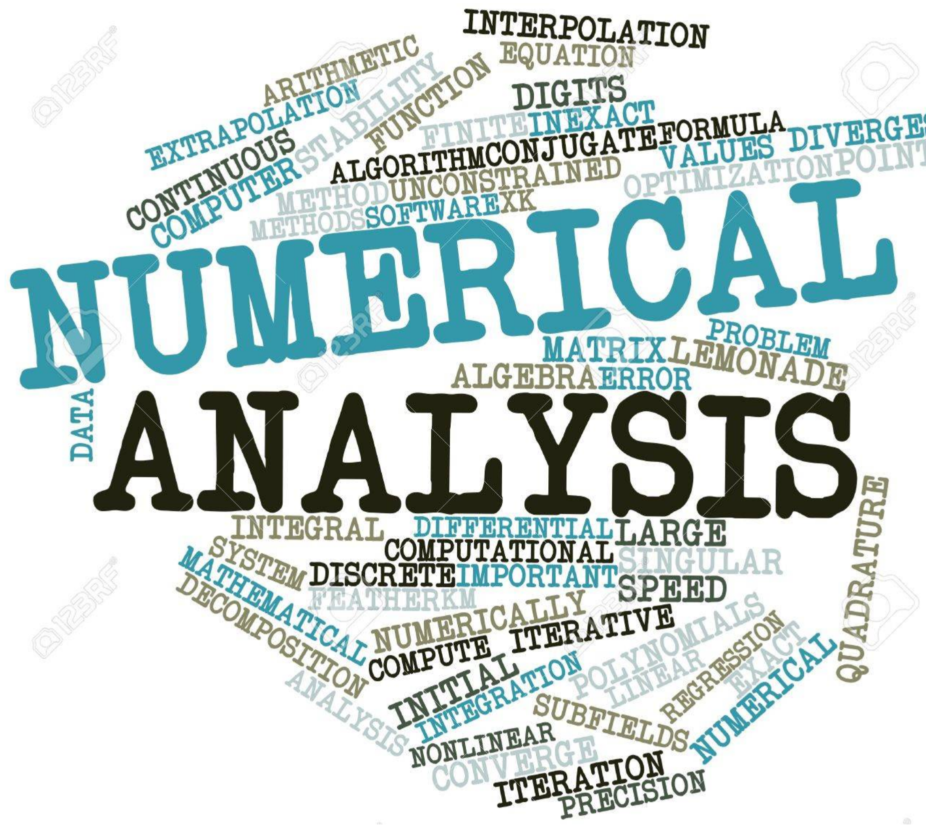
CS510 NUMERICAL ANALYSIS – Rutgers University

POWER METHOD & TRIANGLE ALGORITHM

Programming Project III

Yanbo Fang(yf228) & Xuenan Wang(xw336)



Introduction

In this question, we need to build a random stochastic matrix according to a given graph. So, initially, we build a matrix A to represent the edges between different nodes in the graph, if there is an edge from node_i to node_j, the A_(i,j) would 1, otherwise it would be zero. Then for each row, normalization, if there exists dangling nodes, add average weights

1/n to the dangling nodes, n is the number of nodes in the graph, this can avoid the weights of the dangling nodes become very large after enough iterations.

Power Method

Power is an eigenvalue algorithm, which means given a matrix, the method can produce the eigenvector of the largest eigenvalue of the matrix by iterations. In question 1, we need to solve Mx = x by power method. Firstly, let me prove why power method is useful in solving this question. Because M is a column stochastic matrix, the biggest eigenvalue of M is 1, and power method can find the eigenvector of the biggest eigenvalue, in other words, the eigenvector of 1.

We can use iterations to find the solution as the following, at first, we use np.random.rand() to generate primal x and normalize x, then use np.dot(M, x) to generate x', replace x as x', then do the calculation again, and get a new x', then replace x as x' again, continue to do calculation until reaching the limit of iteration number. Finally output the error of power method by calculate abs(Mx-x).sum(), if the error is close to 0, that means the effect of power method is great.

For question2, we need to use power method to generate a symmetric matrix, so my plan is to design a special graph, in this graph, every node has edges to all other nodes include itself, so the M would be filled with 1/n, n is the number of nodes in the graph, then generate n by n matrix X, and normalize X by column, then using power method on M and X, finally it would generate a n by n matrix, which is a symmetric matrix, this is the result

And actually, we misunderstood problem 2 as using power method on a symmetric matrix to solve Mx = x question, so we did this part of the job also, so please allow me to introduce this part of work also. Because it is difficult to design a symmetric column stochastic matrix after normalization, we used computer to generate this kind of matrix M, then we use power method on it, then it output the result, just like question 1

Triangle Algorithm

In question 3, we need use Triangle Algorithm to solve Mx = x, we would use the same graph as in question 1, and generate the same column stochastic matrix M, then we transferred the x in the right to the left, so the equation becomes (M-I)x = 0, M-I is consisted of v_1, v_2,..., v_n, then we let P = 0, and generate a primal x just like what we did in question 1, and then normalization. Then we use M-I to multiply x, and we get result P', then we find the closest v_i to P, and we set v_i as pivot.

Then we can do iterations as the following, first calculating a_star as np.dot((P-P').T, pivot - P')/math.pow(norm(pivot-p'),2), then we update x by if j is not equal to i(pivot), then x_j = (1-a_star)x_j, else, x_i=(1-a_star)x_j+a_star, then update P' as the result of np.dot(M-I, x), then for any vertices in v_i in M-I, if (norm(P' - v_i)) larger than (norm(P-v_i)), then we set pivot as v_i.

This program would iterates until norm(P-P') less or equal than theta * norm(P-pivot), theta is a constant number. Finally it would generate the result x, we can calculate the error of the result by output Mx - x, and compare it with the error of power method in question 1 to see the difference between these two algorithms.

Other Algorithms

In addition, we also achieve Jacobi, Gauss Seidel and SOR algorithm and compare them with power method algorithm and Triangle Algorithm to find their difference.

Jacobi

The Jacobi method is a classic iterative algorithm for determining the solution of a strictly diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate in plugged in. We just initialize x randomly, then compute x' according to linear equations, then replace x as x', then do the calculation again and update x. The process will iterate until the result converges.

Gauss Seidel

The Gauss Seidel Algorithm is an updated version of Jacobi algorithm, so this method can be used to solve linear equations, by comparison with Gauss Seidel algorithm always uses the newest parameter, so this algorithm can converge more quickly. We write the code at the base of Jacobi code, when the algorithm iterates, just always use the newest x when an element in x got updated.

SOR

Compared with Jacobi and Gauss Seidel Algorithm, SOR algorithm has a more quickly converge speed, because SOR algorithm uses a constant factor named relaxation factor, the idea of SOR is that when update x, previous information can be used. In our code, we set relaxation factor as 0.75

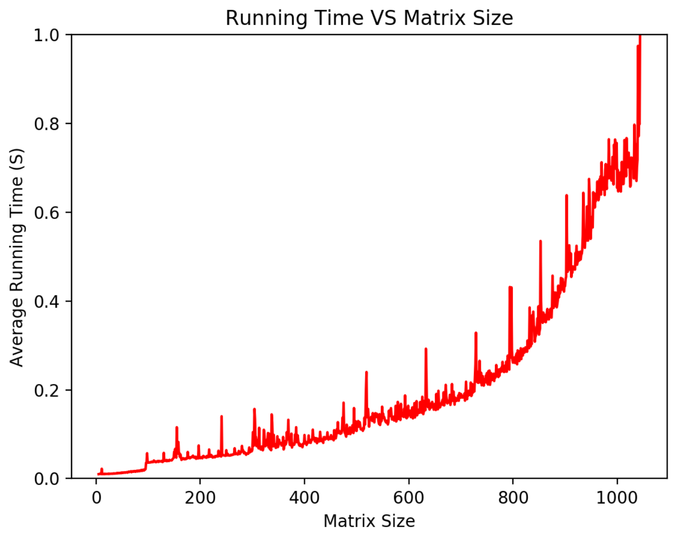
Experiment

Power Method

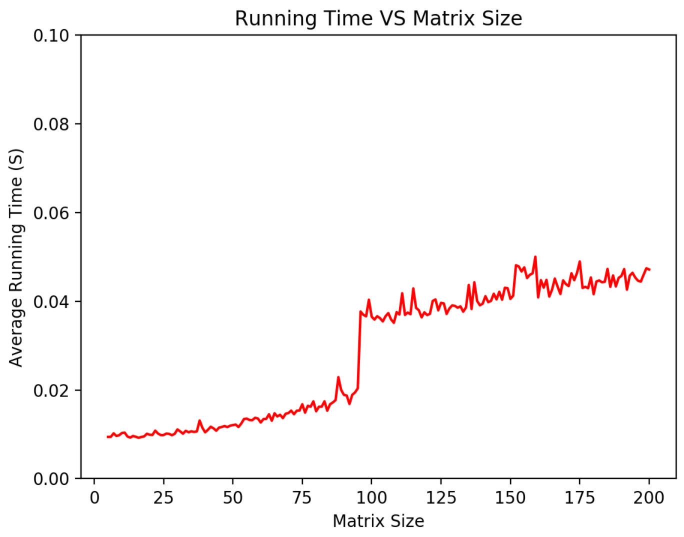
The key point of this programming project is to solve the problem $Mx = x$. We will analyze some properties during solving this problem. Since we are using random generated matrix, we should run single experiment for multiple times and take the average value. We run each experiment for at least 20 times, mostly 50 times.

Running Time VS Matrix Size

We generate a matrix of size n, using power method to solve $Mx = x$ and get the running time using (Python)Time.Time() function. We do this for each matrix size of n for 20 times and compute the average running time, and then plot the figure.

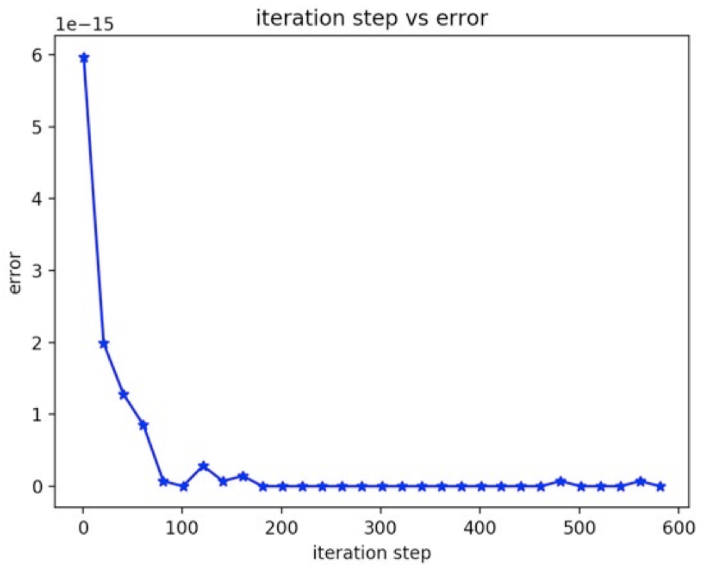


We can see from the above figure, we can see that running time increase non linearly along as the matrix size goes up. We find that during size of 2-200, there's a sudden increasement of running time and we wonder what happened there. Therefore, we have the next figure.



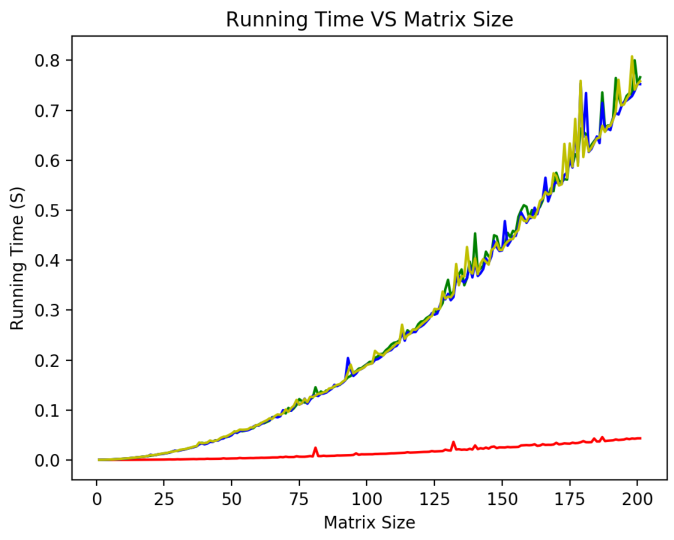
We can clearly see that when matrix size is approaching 100, the corresponding running time increase suddenly. At the matrix size of 50, the running time increase smoothly and also at a relatively low time expense, therefore we decided to run our experiment all at matrix size of 50.

Error VS Iteration Step

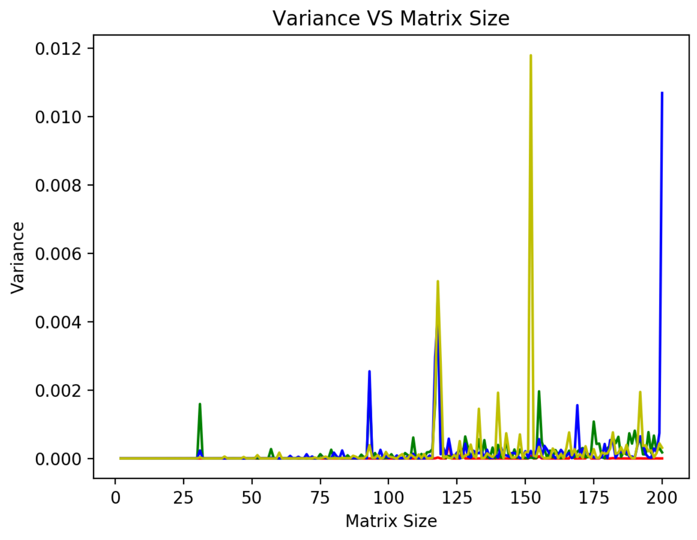


We compute computational error and plot it along the iteration step number going up. We see that at the iteration of 200 times, the algorithm converge to a stable level. And that's also the iteration number we will use for further experiments.

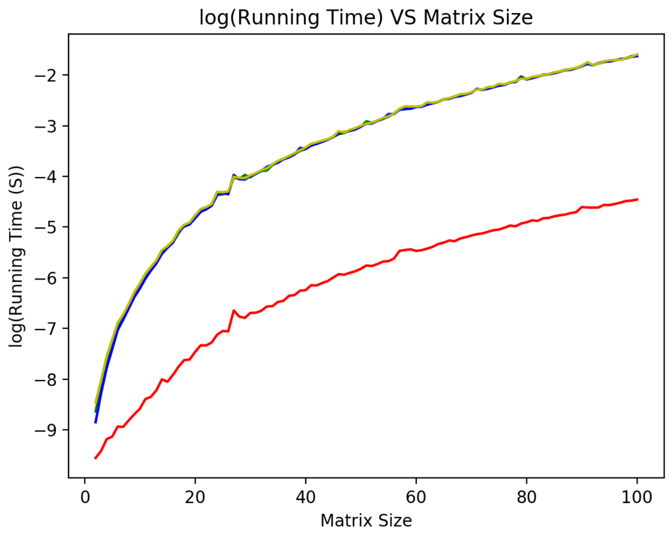
Comparison with Jacobi, Gauss Seidel and SOR



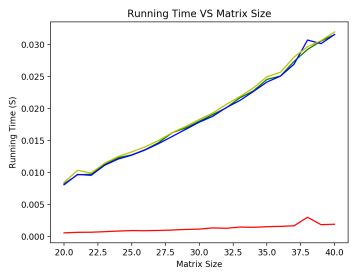
Above is the figure we have when comparing running time of different algorithms. RED line is power method, GREEN line is Jacobi, BLUE line is Gauss Seidel and YELLOW line is SOR. The colors we use to represent different algorithms will be the same in the future. For each experiment in this section, we will first generate a matrix randomly and solve $Mx = x$ using different algorithms. For each matrix size of n, we will repeat above procedures for 50 times and take the average value. We can clearly see that power method is clearly faster than the other three algorithms.

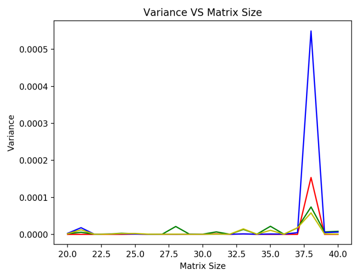


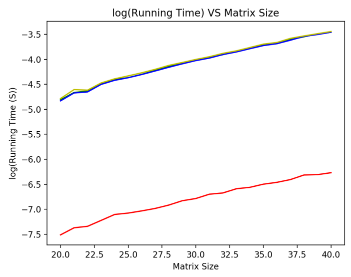
Above is the variance of four algorithms at different matrix size. We can barely see the red line, which indicates that power method has a more stable running time comparing with other algorithms.



We also took a look at matrix size and running time in logarithmic scale. We can see that the running time is not in log(n) time. To further illustrate this problem, we will redo this experiment during matrix size of 20-40.

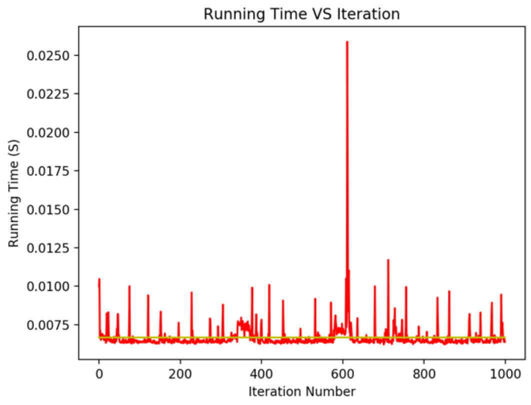


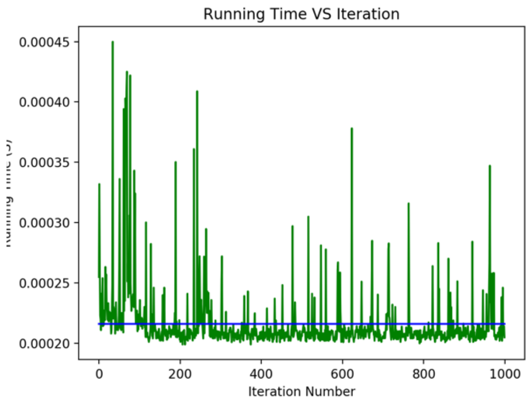




We can see from above three figures that between matrix size of 20 and 40, the running time is basically log(n) and power method still got a better performance.

Apply Power Method on Symmetric Matrix



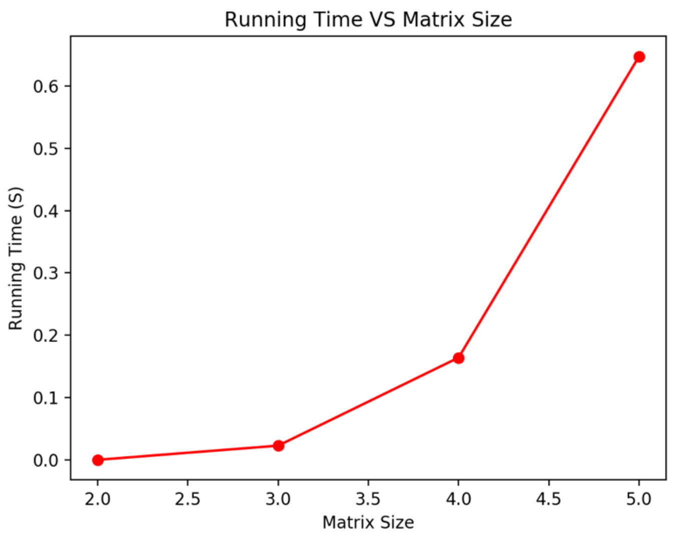


We apply power method both on random matrix and symmetric matrix. Left(**RED**) one is random matrix and right(**GREEN**) is symmetric matrix. It's clear that symmetric matric got a better running time compared to random matrix. Variance for random matrix running time is 8.55905404366638E-07 and for symmetric matrix is 7.31300723089134E-10. Since there's a huge spike on random matrix at iteration number around 600, we wonder if that's the reason causing higher variance. Therefore, we repeat this process for 100 times and see at what percentage that random matrix got a higher variance.

Based on our experiment, random matrix got a 100% higher variance than symmetric matrix. We are not saying this is proof of anything, we are only presenting our findings and results from our experiments.

Triangle Algorithm

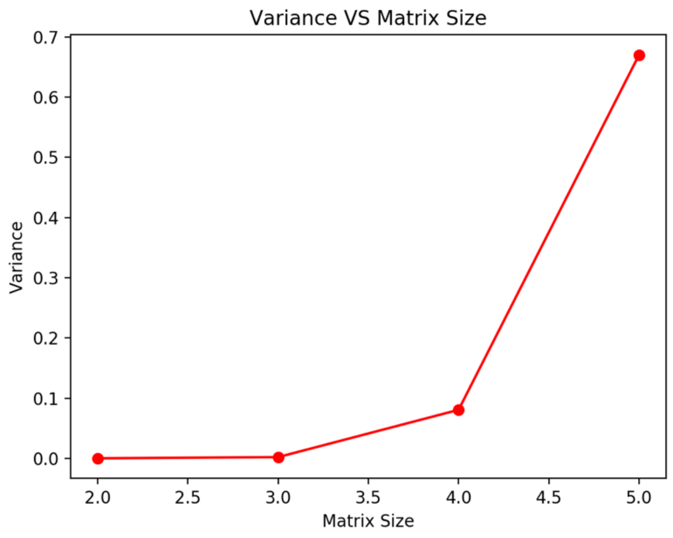
Running Time



Triangle algorithm is taking way more time to compute comparing to other algorithms, so we cannot cross compare it with others.

We can see below matrix size of 5, the running time of triangle algorithm is acceptable.

Variance



Variance is also acceptable but not as good as power method. Like running time, variance of triangle algorithm is very sensitive to matrix size and can easily go wild.

Source Code

```
import numpy as np
import math
import copy
import time
import matplotlib.pyplot as plt


def compute_error(x, M):
   return np.dot(M, x)-x

##########################################
#######
# build stochastic matrix
# question1 and question3
##########################################
######
def stochasticmatrix(matrix):
   Hermit = 0
   for i in range(n):  #
      children = 0  # the number of children of the node/
the output degree
      for j in range(n):
         if matrix[i][j] == 1:
            children += 1
      if children != 0:  # dangling node
         # matrix[i] *= (1/children)
         for j in range(n):
            if matrix[i][j] == 1:
               matrix[i][j] = (1 / children)
      else:
         for j in range(n):  # normalization
            matrix[i][j] = (1 / n)
         Hermit = 1

   matrix = np.matrix(matrix)
   if Hermit == 1:
      matrix = matrix.H
   else:
      matrix = matrix.T  # transpose

   return matrix


##########################################
######
# using power method to iterate
# queston2
##########################################
######
def power_method(A, n, round):
   # time_start = time.time()
   e = np.ones((n, 1))  # column vector full with 1
   e_t = e.T
   M = d * stochasticmatrix(A) + (1 - d) * ((e * e_t) / n)  #
calculate M
   # print("\n the M is:")
   # print(M)
   x = np.random.rand(n, 1)  # random primal x
   # x = np.ones((n,1))
   # a = sum(x)
   # for i in range(n):
   #         x[i] = x[i] / a
   # print("\n the primal x is:")
   # print(x)
   # error_array = np.array([])
   # x_old = 0
   for k in range(round):  # interate
      x = np.dot(M, x)
      x /= (sum(x))
      # error = compute_error(x, M)
      # error_array = np.append(error_array, np.var(error))
      # error = x - x_old
      # error_array = np.append(error_array,
np.mean(error))
      # error_array = np.append(error_array, np.var(x))
      # x_old = x
      # a = np.dot(M, x)
      # error_array = np.append(error_array, abs(a.sum() -
x.sum()))
   # x /= (sum(x))
   # print("\n solve Mx=x, x is:")
   # print(x)
   # print('\n')
   # time_end = time.time()
   # error = compute_error(x, M)
   # error_array = 0
   # a = np.dot(M, x)
   # return time_end-time_start


##########################################
######
# using power method to iterate
# question1
##########################################
######
def generate_graph(n):
   A = np.random.rand(n, n)
   for i in range(n):
      for j in range(n):
         if A[i][j] <= 0.5:
            A[i][j] = 0
         else:
            A[i][j] = 1
   # print("A is:")
   # print(A)
   # print("\n")
   return A


##########################################
###
```

```
# generate symmetric matrix
# question2
##########################################
##
def generate_graph_symmetric(n):
    # A = np.ones((n,n))
    valid = 0
    while valid == 0:  # if symmetrix valid == 0, then return
        valid = 1
        A = np.random.rand(n, n)
        for i in range(n):
            for j in range(n):
                if A[i][j] <= 0.5:
                    A[i][j] = 0
                else:
                    A[i][j] = 1

        for i in range(n):  #
            children = 0  # the number of children of the node/
the output degree
            for j in range(n):
                if A[i][j] == 1:
                    children += 1
            if children == 0:  # dangling node
                for j in range(n):  # normalization
                    A[i][j] = (1 / n)

        A = np.triu(A)  # build diagonal matrix
        A += A.T - np.diag(A.diagonal())

        for i in range(n):
            A[i] /= sum(A[i])  # normaliztion

        for i in range(n):  # check diagonal or not
            for j in range(i, n):
                if A[i][j] != A[j][i]:
                    valid = 0
    # print("\n original matrix is:")
    # print(A)
    # print('\n')
    return A


##########################################
###
# use power method to iterate in question2
##########################################
###
def power_method_realmat(A, n):
    # time_start = time.time()
    e = np.ones((n, 1))
    e_t = e.T
    M = d * A + (1 - d) * ((e * e_t) / n)

    # M = np.triu(M)
    # M += M.T - np.diag(M.diagonal())
    # print("M is :")
    # print(M)
    # print('\n')
    x = np.random.rand(n, 1)
    x /= sum(x)
    '''
x = np.triu(x)
x += x.T - np.diag(x.diagonal())
'''
    # x = np.ones((n,n))
    # a = sum(x)
    # for i in range(n):
    #         x[i] = x[i] / a
    # print("\n the primal is:")
    # print(x)
    for k in range(200):
        x = np.dot(M, x)
    # print('\n')
    x /= sum(x)
    # print("\n the final is:")
    # print(x)
    # time_end = time.time()
    # error = compute_error(x, M)
    # error = 0
    # return time_end-time_start, np.mean(error)


##########################################
###
# triangle algotirhm
# question3
##########################################
###
def triangle(A, n):
    print("\n the input matrix A is:")
    print(A)
    theta = 0.00001
    e = np.ones((n, 1))
    e_t = e.T
    M = d * stochasticmatrix(A) + (1 - d) * ((e * e_t) / n)
    print("\n M is:")
    print(M)
    M = M - np.eye(n)  # solve Mx = x -> (M-I)x = 0
    e = np.ones((n, 1))
    P = np.zeros((n, 1))  # P is zero vector
    pivot = []
    index = 0  # index for pivot
    print("\n M is:")
    print(M)

    alfa = np.random.rand(n, 1)  # primal guess
    alfa /= sum(alfa)
    P_prime = np.dot(M, alfa)  # primal P_primal
    number = float('inf')
    for i in range(n):  # find the closest vector
        if np.linalg.norm(P - M[:, i]) < number:
            # if np.dot((P - P_prime).T, M[:,i]) >= (1/2) *
(math.pow(np.linalg.norm(P), 2) -
math.pow(np.linalg.norm(P_prime), 2)):
            number = np.linalg.norm(P - M[:, i])
            pivot = M[:, i]
            index = i
    # print(np.linalg.norm(P - P_prime))
    # print(np.linalg.norm(P - pivot))
    count = 0  # interation times

    while np.linalg.norm(P - P_prime) > theta *
np.linalg.norm(P - pivot):
        count += 1
        # number = float('inf')
        for i in range(n):
            if np.dot((P - P_prime).T, M[:, i]) >= (1 / 2) * (
                    math.pow(np.linalg.norm(P), 2) -
math.pow(np.linalg.norm(P_prime), 2)):
                # if np.linalg.norm(P - M[:,i]) < number:
                # number = np.linalg.norm(P - M[:,i])
                pivot = M[:, i]
                index = i
                break

        al_star = (np.dot((P - P_prime).T, (pivot - P_prime))) / (
            math.pow(np.linalg.norm(pivot - P_prime), 2))  #
build al_star
        # P_pp = (1 - al_star) * P_prime + al_star * pivot
        for i in range(n):  # update alfa
            if i != index:
                alfa[i] = (1 - al_star) * alfa[i]
            else:
                alfa[i] = (1 - al_star) * alfa[i] + al_star

        P_pp = np.dot(M, alfa)
        P_prime = P_pp  # replace P_prime with P_pp

    # print(al_star)
    # print(P_pp)
    a = np.dot(M + np.eye(n), alfa)
    print("\nthe final result is:")
    print(alfa)

    print("\n the number of interation is:")
    print(count)

    print("\n M * alfa is:")
    print(a)
    # print(sum(np.dot(M + np.eye(n), alfa)))
    distance = 0
    for i in range(n):
        distance += abs(a[i][0] - alfa[i][0])

    print("\n the error is:")
    print(distance)


##########################################
###
# comparison Jacobi
##########################################
###
def Jacobi(A, n, it):  # Jacobi algorithm
    e = np.ones((n, 1))
    e_t = e.T
    M = d * stochasticmatrix(A) + (1 - d) * ((e * e_t) / n)

    M = M - np.eye(n)
    x = np.random.rand(n)  # primal guess
    x /= sum(x)
    y = np.zeros(n)  # intermedate value
    count = 0  # number of iteration

    while count < it:
        count += 1
        for i in range(n):
            temp = 0
            for j in range(n):
                if i != j:
                    temp = temp - (x[j] * M[i, j])
            y[i] = temp / M[i, i]
        x = copy.deepcopy(y)

    print(f"\n the result of {count} iteration is:")
    print(y)
    result = np.dot(M, x)
    distance = 0
    for i in range(n):
        distance += abs(result[0, i])

    print(f"the error of Jacobi iteration is {distance}")


##########################################
###
# comparison Gauss-Seidel
##########################################
###
def Gauss_Seidel(A, n, it):  # Gauss_Seidel algorithm
    e = np.ones((n, 1))
    e_t = e.T
    M = d * stochasticmatrix(A) + (1 - d) * ((e * e_t) / n)

    M = M - np.eye(n)
    x = np.random.rand(n)  # primal guess
    x /= sum(x)
    y = np.zeros(n)  # intermedate value
    count = 0  # number of iteration

    while count < it:
        count += 1
        for i in range(n):
            temp = 0
            for j in range(n):
                if i != j:
                    temp = temp - (x[j] * M[i, j])
            x[i] = temp / M[i, i]

    print(f"\n the result of {count} iteration is:")
    print(x)
    result = np.dot(M, x)
    distance = 0
    for i in range(n):
        distance += abs(result[0, i])

    print(f"the error of Gauss_Sdidel is {distance}")


##########################################
###
# comparison SOR
##########################################
###
def SOR(A, n, it):
    w = 0.7  # relaxation factor
    e = np.ones((n, 1))
    e_t = e.T
    M = d * stochasticmatrix(A) + (1 - d) * ((e * e_t) / n)

    M = M - np.eye(n)
    x = np.random.rand(n)  # primal guess
    x /= sum(x)
    y = np.zeros(n)  # intermediate value
    count = 0  # number of iteration

    while count < it:
        count += 1
        for i in range(n):
            temp = 0
            for j in range(n):
                if i != j:
                    temp = temp - (x[j] * M[i, j])
            y[i] = (1 - w) * y[i] + w * (temp / M[i, i])
        x = copy.deepcopy(y)

    print(f"\n the result of {count} iteration is:")
    print(y)
    result = np.dot(M, x)
    distance = 0
    for i in range(n):
        distance += abs(result[0, i])

    print(f"the error of SOR iteration is {distance}")


##########################################
###

##########################################
###
n = 5
d = 0.85
iteration = 20
# a = np.matrix('1 1 0; 1 0 1; 0 0 0')
G = np.array([[0, 1 / 2, 1 / 2, 0, 0, 0],
              [1 / 6, 1 / 6, 1 / 6, 1 / 6, 1 / 6, 1 / 6],
              [1 / 3, 1 / 3, 0, 0, 1 / 3, 0],
              [0, 0, 0, 0, 1 / 2, 1 / 2],
              [0, 0, 0, 1 / 2, 0, 1 / 2],
              [0, 0, 0, 1, 0, 0]])
```

```
# print("\n############ This is question 1
################\n")
# A = generate_graph(n)
# B = copy.deepcopy(A)
# power_method(A, n)

# Test for running time

# time_array = np.array([])
# error_array = np.array([])
# matrix_size = 5 #50
# iteration_num = 1000
# iter_round = 200
# for i in range(iteration_num):
#     A = generate_graph(matrix_size)
#     B = copy.deepcopy(A)
#     time_start = time.time()
#     power_method(A, matrix_size, iter_round)
#     time_end = time.time()
#     # print('time cost: ', time_diff)
#     time_array = np.append(time_array, time_end-
time_start)
#     # error_array = np.append(error_array, error)
#
# plt.figure(1)
# plt.plot(range(iteration_num), time_array, 'r')
# plt.plot(range(iteration_num), np.ones(iteration_num) *
np.mean(time_array), 'y')
# # plt.ylim(0, 0.008)
# # plt.text(0.4 * iteration_num, 0.007, 'Power Method,
Random Matrix')
# # plt.text(0.4 * iteration_num, 0.006, 'mean: ' +
str(np.mean(time_array)) + ' s')
# # plt.text(0.4 * iteration_num, 0.005, 'variance: ' +
str(np.var(time_array)))
# plt.xlabel('Iteration Number')
# plt.ylabel('Running Time (S)')
# plt.title('Running Time VS Iteration')
# print('mean: ', np.mean(time_array))
# print('var: ', np.var(time_array))

# time_array = np.array([])
# error_array2 = np.array([])
# matrix_size = 50 #50
# iteration_num = 1000
# iter_round = 10
# for i in range(iteration_num):
#     A = generate_graph(matrix_size)
#     B = copy.deepcopy(A)
#     time_diff, error_array2 = power_method(A,
matrix_size, iter_round)
#     # print('time cost: ', time_diff)
#     time_array = np.append(time_array, time_diff)
#     # error_array2 = np.append(error_array2, error2)
#
# time_array = np.array([])
# error_array3 = np.array([])
# matrix_size = 50 #50
# iteration_num = 1000
# iter_round = 10
# for i in range(iteration_num):
#     A = generate_graph(matrix_size)
#     B = copy.deepcopy(A)
#     time_diff, error_array3 = power_method(A,
matrix_size, iter_round)
#     # print('time cost: ', time_diff)
#     time_array = np.append(time_array, time_diff)
#     # error_array3 = np.append(error_array3, error3)
#
# time_array = np.array([])
# error_array4 = np.array([])
# matrix_size = 50 #50
# iteration_num = 1000
# iter_round = 10
# for i in range(iteration_num):
#     A = generate_graph(matrix_size)
#     B = copy.deepcopy(A)
#     time_diff, error_array4 = power_method(A,
matrix_size, iter_round)
#     # print('time cost: ', time_diff)
#     time_array = np.append(time_array, time_diff)
#     # error_array4 = np.append(error_array4, error4)
#
# time_array = np.array([])
# error_array5 = np.array([])
# matrix_size = 50 #50
# iteration_num = 1000
# iter_round = 10
# for i in range(iteration_num):
#     A = generate_graph(matrix_size)
#     B = copy.deepcopy(A)
#     time_diff, error_array5 = power_method(A,
matrix_size, iter_round)
#     # print('time cost: ', time_diff)
#     time_array = np.append(time_array, time_diff)
#     # error_array5 = np.append(error_array5, error5)

# plt.figure(2)
# plt.plot(range(iter_round), error_array, 'b')
# plt.plot(range(iter_round), error_array, 'bo')
# plt.plot(range(iter_round), error_array2, 'r')
# plt.plot(range(iter_round), error_array2, 'ro')
# plt.plot(range(iter_round), error_array3, 'y')
# plt.plot(range(iter_round), error_array3, 'yo')
# plt.plot(range(iter_round), error_array4, 'g')
# plt.plot(range(iter_round), error_array4, 'go')
# plt.plot(range(iter_round), error_array5, 'm')
# plt.plot(range(iter_round), error_array5, 'mo')
# plt.xlabel('Iteration Number')
# plt.ylabel('Error')
# plt.title('Error VS Iteration')
# plt.show()

# print('average time cost: ', np.mean(time_array))

# # Test for matrix size
# time_array = np.array([])
# time_mean = np.array([])
# time_var = np.array([])
# error_array = np.array([])
# error_mean = np.array([])
# iteration_num = 100
# matrix_size = 5
# matrix_size_limit = 200
# start_size = matrix_size
# time_limit = 0.1 #1
# time_now = 0
# size_step = 1
# # while time_now <= time_limit:
# while matrix_size <= matrix_size_limit:
#     print('matrix size: ', matrix_size)
#     time_array = np.array([])
#     error_array = np.array([])
#     for i in range(iteration_num):
#         A = generate_graph(matrix_size)
#         B = copy.deepcopy(A)
#         time_diff, error = power_method(A, matrix_size)
#         # print('time cost: ', time_end - time_start)
#         time_array = np.append(time_array, time_diff)
#         error_array = np.append(error_array, error)
#     time_mean = np.append(time_mean,
np.mean(time_array))
#     time_var = np.append(time_var, np.mean(time_array))
#     error_mean = np.append(error_mean,
np.mean(error_array))
#     matrix_size += size_step
#     time_now = np.mean(time_array)
#
# plt.figure(2)
# plt.plot(range(start_size, matrix_size, size_step),
time_mean, 'r')
# plt.ylim(0, time_limit)
# plt.xlabel('Matrix Size')
# plt.ylabel('Average Running Time (S)')
# plt.title('Running Time VS Matrix Size')
# plt.show()

# print("\n############# This is question 1
#################\n\n")

# print("\n############# This is question 2
#################\n")
# A_Real = generate_graph_symmetric(matrix_size)
# power_method_realmat(A_Real, matrix_size)

# time_array = np.array([])
# error_array = np.array([])
# # iteration_num = 100
# for i in range(iteration_num):
#     A_Real = generate_graph_symmetric(matrix_size)
#     time_start = time.time()
#     power_method_realmat(A_Real, matrix_size)
#     time_end = time.time()
#     # print('time cost: ', time_diff)
#     time_array = np.append(time_array, time_end-
time_start)
#     # error_array = np.append(error_array, error)
```

```
#
# plt.figure(2)
# plt.plot(range(iteration_num), time_array, 'g')
# plt.plot(range(iteration_num), np.ones(iteration_num) *
np.mean(time_array), 'b')
# # plt.text(0.4 * iteration_num, 0.004, 'Power Method,
General Real Matrix')
# # plt.text(0.4 * iteration_num, 0.003, 'mean: ' +
str(np.mean(time_array)) + ' s')
# # plt.text(0.4 * iteration_num, 0.002, 'variance: ' +
str(np.var(time_array)))
# plt.xlabel('Iteration Number')
# plt.ylabel('Running Time (S)')
# plt.title('Running Time VS Iteration')
# print('mean: ', np.mean(time_array))
# print('var: ', np.var(time_array))

# plt.figure(2)
# plt.plot(range(iteration_num), error_array, 'g')

# plt.show()
#
# print('average time cost: ', np.mean(time_array))

# Test for matrix size
# time_array = np.array([])
# time_mean = np.array([])
# time_var = np.array([])
# error_array = np.array([])
# error_mean = np.array([])
# time_now = 0
# size_step = 1
# matrix_size = 5
# while time_now <= time_limit:
#     print('matrix size: ', matrix_size)
#     time_array = np.array([])
#     error_array = np.array([])
#     for i in range(iteration_num):
#         A_Real = generate_graph_symmetric(matrix_size)
#         time_diff, error = power_method_realmat(A_Real,
matrix_size)
#         # print('time cost: ', time_end - time_start)
#         time_array = np.append(time_array, time_diff)
#         error_array = np.append(error_array, error)
#     time_mean = np.append(time_mean,
np.mean(time_array))
#     time_var = np.append(time_var, np.mean(time_array))
#     error_mean = np.append(error_mean,
np.mean(error_array))
#     matrix_size += size_step
#     time_now = np.mean(time_array)
#
# plt.figure(3)
# plt.plot(range(start_size, matrix_size, size_step),
time_mean, 'r')
# plt.ylim(0, time_limit)
# plt.xlabel('Matrix Size')
# plt.ylabel('Average Running Time (S)')
# plt.title('Running Time VS Matrix Size')
# plt.show()

# print("\n############# This is question 2
#################\n")

# print("\n############# This is question 3
#################\n\n")
# triangle(B, n)
# print("\n############# This is question 3
#################")
#
# print("\n############# This is Jacobi
#################\n")
# Jacobi(A, n, iteration)
# print("\n############# This is Jacobi
#################\n\n")
#
# print("\n############# This is Gauss_Seidel
#################\n")
# Gauss_Seidel(A, n, iteration)
# print("\n############# This is Gauss_Seidel
#################\n\n")
#
# print("\n############# This is SOR
#################\n")
# SOR(A, n, iteration)
# print("\n############# This is SOR
#################\n\n")

# iter = 100
# num_random = 0
#
# for i in range(iter):
#     time_array = np.array([])
#     error_array = np.array([])
#     matrix_size = 5 #50
#     iteration_num = 100
#     iter_round = 200
#     for i in range(iteration_num):
#         A = generate_graph(matrix_size)
#         B = copy.deepcopy(A)
#         time_start = time.time()
#         power_method(A, matrix_size, iter_round)
#         time_end = time.time()
#         # print('time cost: ', time_diff)
#         time_array = np.append(time_array, time_end-
time_start)
#
#     var_random = np.var(time_array)
#
#     time_array = np.array([])
#     error_array = np.array([])
#     # iteration_num = 100
#     for i in range(iteration_num):
#         A_Real = generate_graph_symmetric(matrix_size)
#         time_start = time.time()
#         power_method_realmat(A_Real, matrix_size)
#         time_end = time.time()
#         # print('time cost: ', time_diff)
#         time_array = np.append(time_array, time_end-
time_start)
#
#     var_sym = np.var(time_array)
#
#     if var_random >= var_sym:
#         num_random += 1
# print('var(random)/var(sym): ', num_random/iter)
```