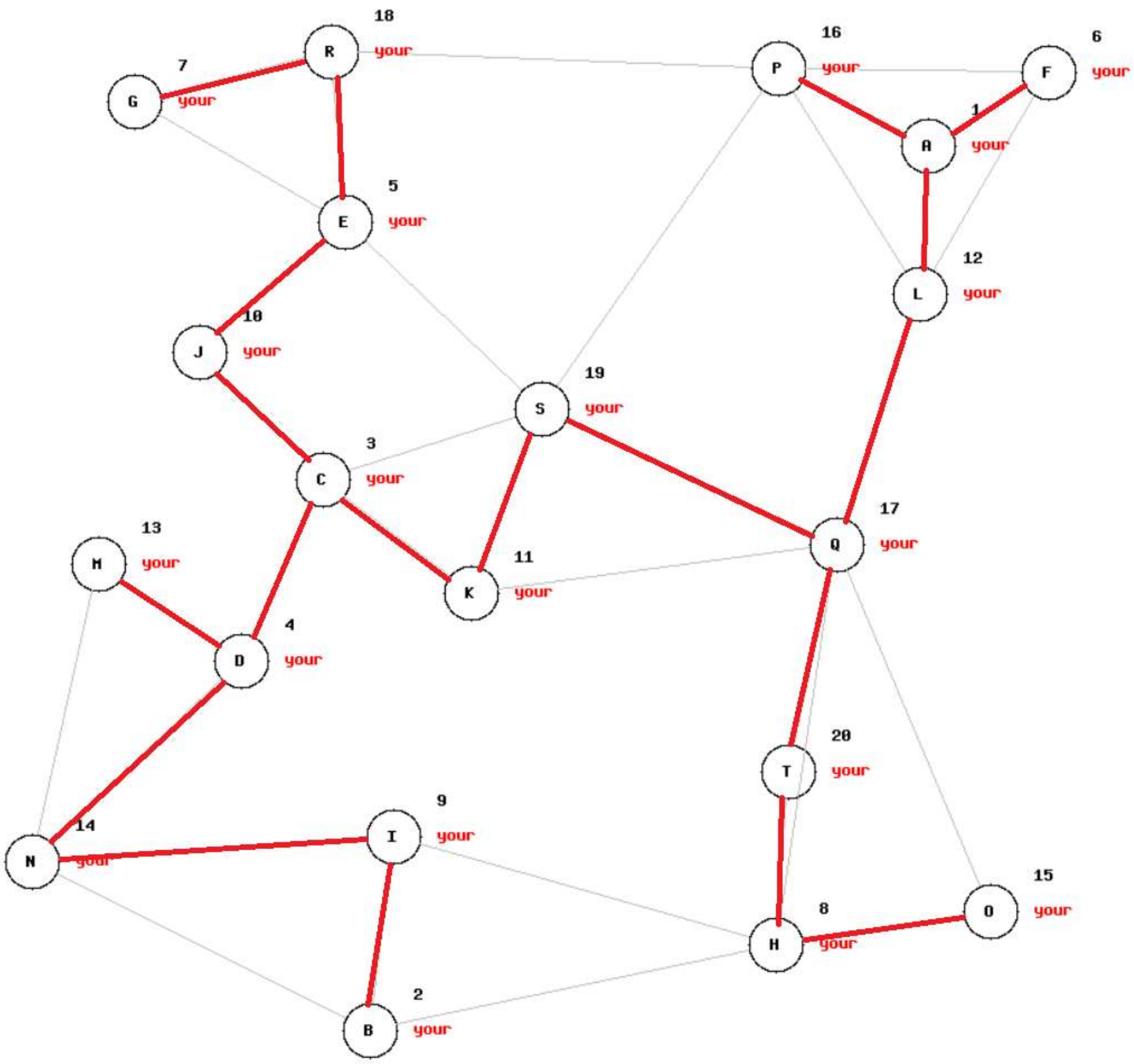
In [48]:
```
from IPython.display import Image

Image("MST.png")
```

Out[48]:



I highlighted the Minimum Spanning Tree that I manually found with Prim's algorithm from point A with red color. If I start from other nodes, e.g. "K" - it should yield the same cost tree, as the MST is unique. Prim's algorithm can descripted like this, firstly, we initialize a tree with a single vertex, chosen arbitrarily from the graph. Then, we grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree and then we repeat the previous action (until all vertices are in the tree).

EX2: In the code below g is converted graph to a matrix. The printed matrix is

In [20]:
```
g = [[0,1,0,0,0,1,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0],
     [0,1,0,0,1,0,1,0,0,0,0],
     [0,0,1,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,1],
     [0,0,0,0,1,0,0,0,0,0,0],
     [0,0,0,0,1,0,0,0,0,1,0],
     [0,0,0,0,0,1,0,0,0,0,0],
     [0,0,0,0,0,0,1,0,0,0,0],
     [0,0,0,0,0,0,0,0,1,0,0],
     [0,0,0,0,1,0,0,0,0,0,0]]

g_ex4 = g
def extendone(matrixgraph):
    response = matrixgraph
    for i in range(0, len(response)):
        for j in range(0, len(response[i])):
            if response[i][j] == 1:
                response[i]=[response[i][a] | response[j][a] for a in range(0,
 len(response[i]))]
    return response

# multiply_matrix(g,g,hop2)
print(extendone(extendone(extendone(g))))
# print (multiply_matrix(hop2,g,hop3))
```

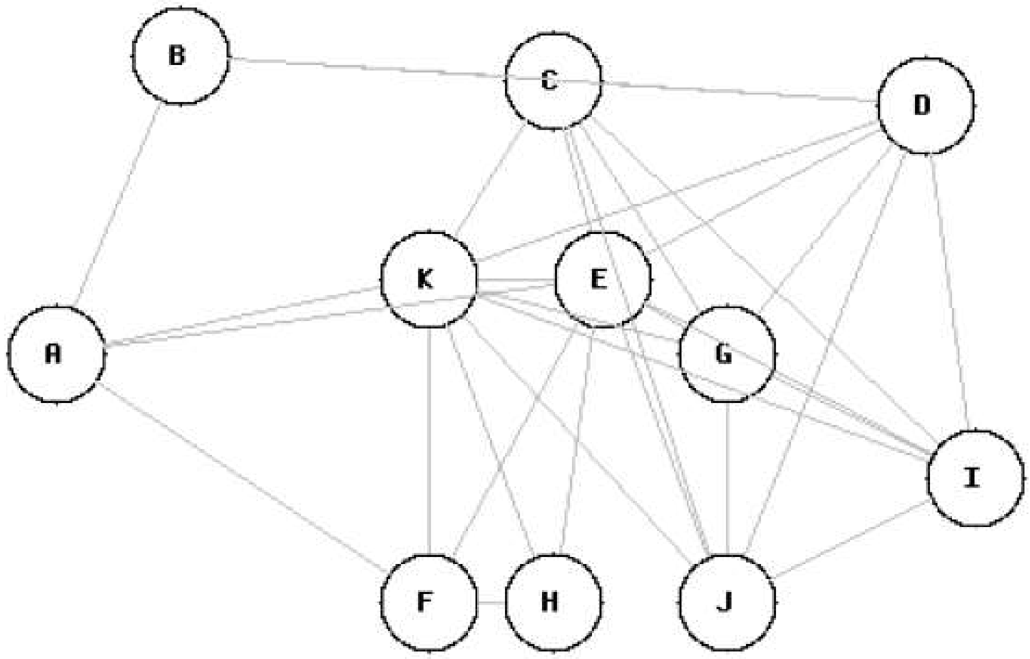
```
[[0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1,
0, 0, 1, 0, 1, 0, 1, 1, 1], [0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1], [0, 0, 0, 0,
1, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 0,
1, 0, 1, 1, 1], [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 0, 1, 0,
1, 1, 1], [0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1], [0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
1]]
```

This the graph where the original edges are replaced by "3-hop" edges

In [21]:
```
from IPython.display import Image

Image("ex1.png")
```

Out[21]:



EX3: Again to get transitive closure of the graph, we use the adjacency matrix. Given a directed graph, finding transitivive closure means whether a vertex j is a path(reachable) from another vertex i for all vertex pairs (i, j) in the given graph. So to find all possible path legthes we should put 1 to diogonals.

I got the same answer for all these cases. 1) GGG...;

For this case, after 2 times multiplication I got the answer.

2) Making use of the following property - G2 = GG, G4 = G2G2, G8 = G4*G4

Finding G4, I got the answer.

3) with Warshall algorithm. I used Warshall algorithm which updates the current graph but not to build new one.

In [22]:
```
g = [[1,1,0,0,0,1,0,0,0,0,0],
     [0,1,0,0,0,0,0,0,0,0,0],
     [0,1,1,0,1,0,1,0,0,0,0],
     [0,0,1,1,0,0,0,0,0,0,0],
     [0,0,0,0,1,0,0,1,0,0,1],
     [0,0,0,0,1,1,0,0,0,0,0],
     [0,0,0,0,1,0,1,0,0,1,0],
     [0,0,0,0,0,1,0,1,0,0,0],
     [0,0,0,0,0,0,1,0,1,0,0],
     [0,0,0,0,0,0,0,0,1,1,0],
     [0,0,0,0,1,0,0,0,0,0,1]]

hop = [[0 for j in range(11)] for i in range(11)]

def multiply_matrix(g1,g2,g3):
    for i in range(11):
        for j in range(11):
            for k in range(11):
                if g1[i][k]&g2[k][j]:
                    g3[i][j] = 1
                    break
    return g3
def warshall_matrix(g):
    for i in range(11):
        for j in range(11):
            for k in range(11):
                if g[i][k]&g[k][j]:
                    g[i][j] = 1
    return g
#1
h =  multiply_matrix(g,g,hop)
h =  multiply_matrix(h,g,hop)
print (h)
#2
h =  multiply_matrix(g,g,hop)
# print (h)
h =  multiply_matrix(h,h,hop)
print (h)
#3
warshall_matrix(g)
```

```
[[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1,
1, 0, 1, 1, 1, 1, 1, 1, 1], [0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1], [0, 0, 0, 0,
1, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 1,
1, 1, 1, 1, 1], [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 0, 1, 1,
1, 1, 1], [0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1], [0, 0, 0, 0, 1, 1, 0, 1, 0, 0,
1]]
[[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1,
1, 0, 1, 1, 1, 1, 1, 1, 1], [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0,
1, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 1,
1, 1, 1, 1, 1], [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 1, 1, 1,
1, 1, 1], [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 1, 1, 0, 1, 0, 0,
1]]
```

```
Out[22]: [[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1],
          [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1],
          [0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1],
          [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
          [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
          [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
          [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
          [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
          [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
          [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1]]
```

EX4:

In [23]:
```
import random

def randomwalk(matrix, walk):
    nodes_prob = {}
    for i in range(0, len(matrix)):
        nodes_prob[chr(i+65)] = 0
    rand = random.randint(0, len(matrix)-1)
    cur = random.randint(0, len(matrix)-1)
    for j in range(0, walk):
        if matrix[cur][rand] == 1 or cur == 1:
            nodes_prob[chr(cur+65)] += 1
            cur = rand
        rand = random.randint(0, len(matrix)-1)

    return nodes_prob

def randomwalk_80_20(matrix, walk):
    nodes_prob = {}
    for i in range(0, len(matrix)):
        nodes_prob[chr(i+65)] = 0
    rand = random.randint(0, len(matrix)-1)
    cur = random.randint(0, len(matrix)-1)

    for j in range(0, walk):
        choice = random.random()
        if choice < 0.2:
            cur = random.randint(0, len(matrix)-1)
        else:
            if matrix[cur][rand] == 1:
                nodes_prob[chr(cur+65)] += 1
                cur = rand
            rand = random.randint(0, len(matrix)-1)
    return nodes_prob

print (randomwalk(g_ex4, 1000000))
print (randomwalk_80_20(g_ex4, 1000000))
```

```
{'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 90848, 'F': 0, 'G': 0, 'H': 0, 'I': 1,
'J': 0, 'K': 90814}
{'A': 10872, 'B': 0, 'C': 13770, 'D': 13122, 'E': 34419, 'F': 10103, 'G': 237
78, 'H': 9476, 'I': 23521, 'J': 23447, 'K': 34568}
```

Here we see that the first method always stays in ’E’ and ’K’, because they act like ’B’ as a trap and there is no going away from that. That is why we will have little number of other nodes. In the second method we have a good proportionality, with a walk of 1000000.

EX5: For the random walk algorithm I decided to multiply the matrix by another matrix. This other matrix is built by the first matrix but each non-zero element on each line is replaced by its probability + 1, as given in the homework (80-20; 60,10,10). After that i divide each line by its sum to have a probability of each edge being taken. As a result we have the weight of each edge in percentage of probability. If we multiply the graph without modifications G->H, B->K, K->H, we get that multiple edges are removed, because their probability goes to zero. With a n of 1000, the probability are very stable, thus n = 1000 is more than sufficient and we stop multiplying. The graph used is the one with the introduced links G->H, B->K, K->H.

In [29]:
```
import numpy
import copy

startgraph = [# A  , B  , C  , D  , E  , F  , G  , H  , I  , J  , K
               [0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0], # A
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0], # B
               [0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0], # C
               [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], # D
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0], # E
               [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], # F
               [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0], # G
               [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0], # H
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0], # I
               [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0], # J
               [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]  # K
             ]

def buildWalk(matrix):
    res = copy.deepcopy(matrix)
    for i in range(0, len(matrix[0])):
        choice = random.random()
        positiveindexes = []
        for j in range(0, len(matrix[i])):
                    if matrix[i][j] == 1:
                        positiveindexes.append(j)
        if len(positiveindexes) == 3:
            if choice < 0.6:
                res[i][positiveindexes[0]] = 2
                res[i][positiveindexes[1]] = 1
                res[i][positiveindexes[2]] = 1
            elif choice < 0.9:
                res[i][positiveindexes[0]] = 1
                res[i][positiveindexes[1]] = 2
                res[i][positiveindexes[2]] = 1
            else:
                res[i][positiveindexes[0]] = 1
                res[i][positiveindexes[1]] = 1
                res[i][positiveindexes[2]] = 2
        elif len(positiveindexes) == 2:
            if choice < 0.8:
                res[i][positiveindexes[0]] = 2
                res[i][positiveindexes[1]] = 1
            else:
                res[i][positiveindexes[0]] = 2
                res[i][positiveindexes[1]] = 1
    return res

def randomWalk(matrix, n):
    sumtotal = 0
    for i in matrix:
        sumtotal += sum(i)
    nmatrix = numpy.array(matrix)
    res = numpy.array(copy.deepcopy(matrix))
    for i in range(0, n):
        res *= numpy.array(buildWalk(matrix))
        for i in range(0, len(res[0])):
```

```
            for j in range(0, len(res[i])):
                res[i][j] /= sum(res[i]) * sumtotal
    return numpy.nan_to_num(res).tolist()

for i in randomWalk(startgraph, 1000):
    print (i)
```

```
[0.0, 0.05876468502354651, 0.0, 0.0, 0.0, 0.00011757104592742669, 0.0, 0.0,
0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.058823529411764705]
[0.0, 0.046377056206473415, 0.0, 0.0, 0.01953182674798378, 0.0, 1.53735048331
08646e-05, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.058823529411764705, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.058823529411764705]
[0.0, 0.0, 0.0, 0.0, 0.058823529411764705, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0059056925624700165, 0.0, 0.0, 0.05545647497378912, 0.
0, 1.391642169410099e-11, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.058823529411764705, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.058823529411764705, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.058823529411764705, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.05876468502354651, 0.0, 0.0, 0.00011757104592742669,
0.0, 0.0, 0.0]
```

What I printed in the end is If we multiply the percentage of each edge choice per node by the total amount of edges we have the overall probability of each edge.

EX6: h contains uneven 80-20, 60-30-10 probabilities and e describes equal-probabilities links.

In [34]:
```
#     A   B C   D E   F   G   H I J K
h = [[0,  0,0,  0,0,  0,  0,  0,0,0,0],
     [0.8,0,0.6,0,0,  0,  0,  0,0,0,0],
     [0,  0,0,  1,0,  0,  0,  0,0,0,0],
     [0,  0,0,  0,0,  0,  0,  0,0,0,0],
     [0,  0,0.3,0,0,  0,  0.2,1,0,0,1],
     [0.2,0,0,  0,0,  0,  0,  0,0,0,0],
     [0,  0,0.1,0,0,  0,  0,  0,1,0,0],
     [0,  0,0,  0,0.8,1,  0,  0,0,0,0],
     [0,  0,0,  0,0,  0,  0,  0,0,1,0],
     [0,  0,0,  0,0,  0,  0.8,0,0,0,0],
     [0,  0,0,  0,0.2,0,  0,  0,0,0,0]]

#     A   B C   D E   F   G   H I J K
e = [[0,  0,0,  0,0,  0,  0,  0,0,0,0],
     [0.5,0,0.3,0,0,  0,  0,  0,0,0,0],
     [0,  0,0,  1,0,  0,  0,  0,0,0,0],
     [0,  0,0,  0,0,  0,  0,  0,0,0,0],
     [0,  0,0.3,0,0,  0,  0.5,1,0,0,1],
     [0.5,0,0,  0,0,  0,  0,  0,0,0,0],
     [0,  0,0.3,0,0,  0,  0,  0,1,0,0],
     [0,  0,0,  0,0.5,1,  0,  0,0,0,0],
     [0,  0,0,  0,0,  0,  0,  0,0,1,0],
     [0,  0,0,  0,0,  0,  0.5,0,0,0,0],
     [0,  0,0,  0,0.5,0,  0,  0,0,0,0]]


d = [[0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0],
     [0,0.09,0,0,0,0,0,0,0,0,0]]

u = [[1 for i in range(1,12)] for i in range(1,12)]
u
coef = 0.2/11
add = [[0 for i in range(1,12)] for i in range(1,12)]

def add_matrix(g1,g2,g3):
    for i in range(11):
        for j in range(11):
            g3[i][j] = 0.8*(g1[i][j]+g2[i][j]) + coef*u[i][j]
            print (g3[i][j],)
        print ("\n")

print ("the uneven 80-20, 60-30-10 probabilities\n")
add_matrix(h,d,add)
```

```
the uneven 80-20, 60-30-10 probabilities

0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.6581818181818183
0.09018181818181818
0.4981818181818181 7
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.8181818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.2581818181818182
```

```
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.17818181818181822
0.8181818181818182
0.018181818181818184
0.018181818181818184
0.8181818181818182


0.17818181818181822
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.0981818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.8181818181818182
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.6581818181818183
0.8181818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
```

```
0.018181818181818184
0.8181818181818182
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.6581818181818183
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.17818181818181822
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
```

In [35]:
```
print ("\n\nequal-probabilities\n")
add_matrix(e,d,add)
```

```
equal-probabilities

0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.4181818181818182
0.09018181818181818
0.2581818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.8181818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
```

```
0.2581818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.4181818181818182
0.8181818181818182
0.018181818181818184
0.018181818181818184
0.8181818181818182


0.4181818181818182
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.2581818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.8181818181818182
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.4181818181818182
0.8181818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
```

```
0.018181818181818184
0.018181818181818184
0.8181818181818182
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.4181818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184


0.018181818181818184
0.09018181818181818
0.018181818181818184
0.018181818181818184
0.4181818181818182
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
0.018181818181818184
```