Project 1
Sudarshana Jagadeeshi

Contents

Objective

Approach

Algorithms
- Shortest path based fast
- Shortest path- Dijkstra

Results
- Figures
- Analysis

Appendix
- Annotated Source Code
- README
- Sources

Objective

We are given a fully connected directed graph with no self-loops. We are also given a and b values, which are NxN matrices representing the unit costs and demands respectively.

The problem at hand is to assign capacities to each of the N*(N-1) links in the graph such that the cost (as defined in the Algorithms section) is minimum.

Approach

To initialize the a matrix, we do the following. For each row i, generate k unique random values not equal to i. Set those elements a_ik to be 1. The rest should be set to 250. To intialize the b matrix, we take my student ID s, and set b_ij to be the absolute value of the ith digit of s minus the jth digit of s.

Then we are ready to create the directed graph, whose weights are the unit costs a. We use the library NetworkX for this purpose. We then proceed to run the algorithm as it is described in the following section. The shortest path implementation is left to the NetworkX library.

Finally, the graph is plottedd using NetworkX and matplotlib.

Algorithms

Shortest path based fast

We seek to minimize the defined as the sum over all nodes- the demand b between them multiplied by the sum of the capacities a that are on the shortest path between them.

From the handout:

- The optimum cost can be expressed explicitely. Let E_{kl} be the set of edges that are on the min cost $k \to l$ path. Then, according to the above, the optimal cost is:

$$Z_{opt} = \sum_{k,l} \left(b_{kl} \sum_{(i,j) \in E_{kl}} a_{ij} \right).$$

[1]

Note that a capacity of zero means no flow can be passed along that link. This means the link will not be built.

To minimize the cost, we use the approach described in the handout "An Application to Network Design":

- Find a minimum cost path between each pair k, l of nodes, with edge weights a_{ij}. This can be done by any standard shortest path algorithm that you met in earlier courses.
- Set the capacity of link (i, j) to the sum of those b_{kl} values for which (i, j) is on the min cost path found for k, l.

[1]

We are unioning the edges of the shortest paths for all pairs of nodes, adding their capacities in the process. Edges with capacity zero are not part of a shortest path for any nodes k,l in the graph.

Shortest path- Dijkstra

Here, the weights used for the graph are the unit costs a.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

[4]

In initialize-single-source, the source is set to be at a distance of 0, and all other nodes at distance infinity. In addition, each nodes predecessor is set to Null.

S is the set of visited nodes, and Q is the queue. Once a node leaves Q, it is placed into S.

The algorithm has a loop that runs until every node is visited. In each iteration, the minimum weight vertice is selected, and each of its adjacents are relaxed.

Relaxation works as follows: when the distance of u plus the weight between u and v is less than the distance of v, the distance of v is updated (because a better path was found).

Results

Figures

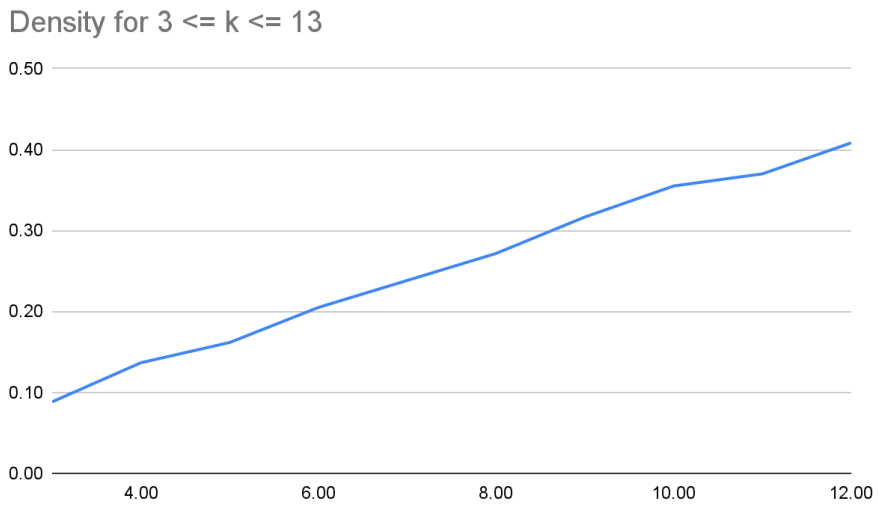

Fig 1.1 Density graph for various k values

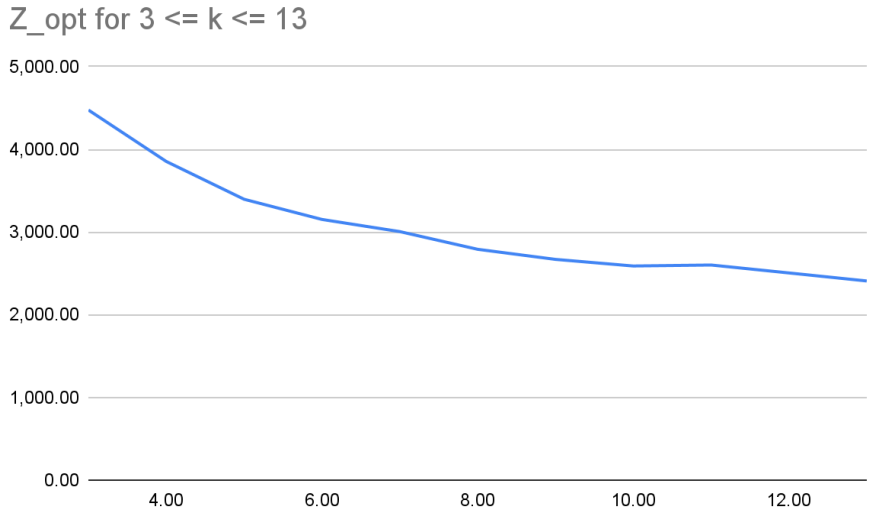

Fig 1.2 Optimal cost graph for various k values

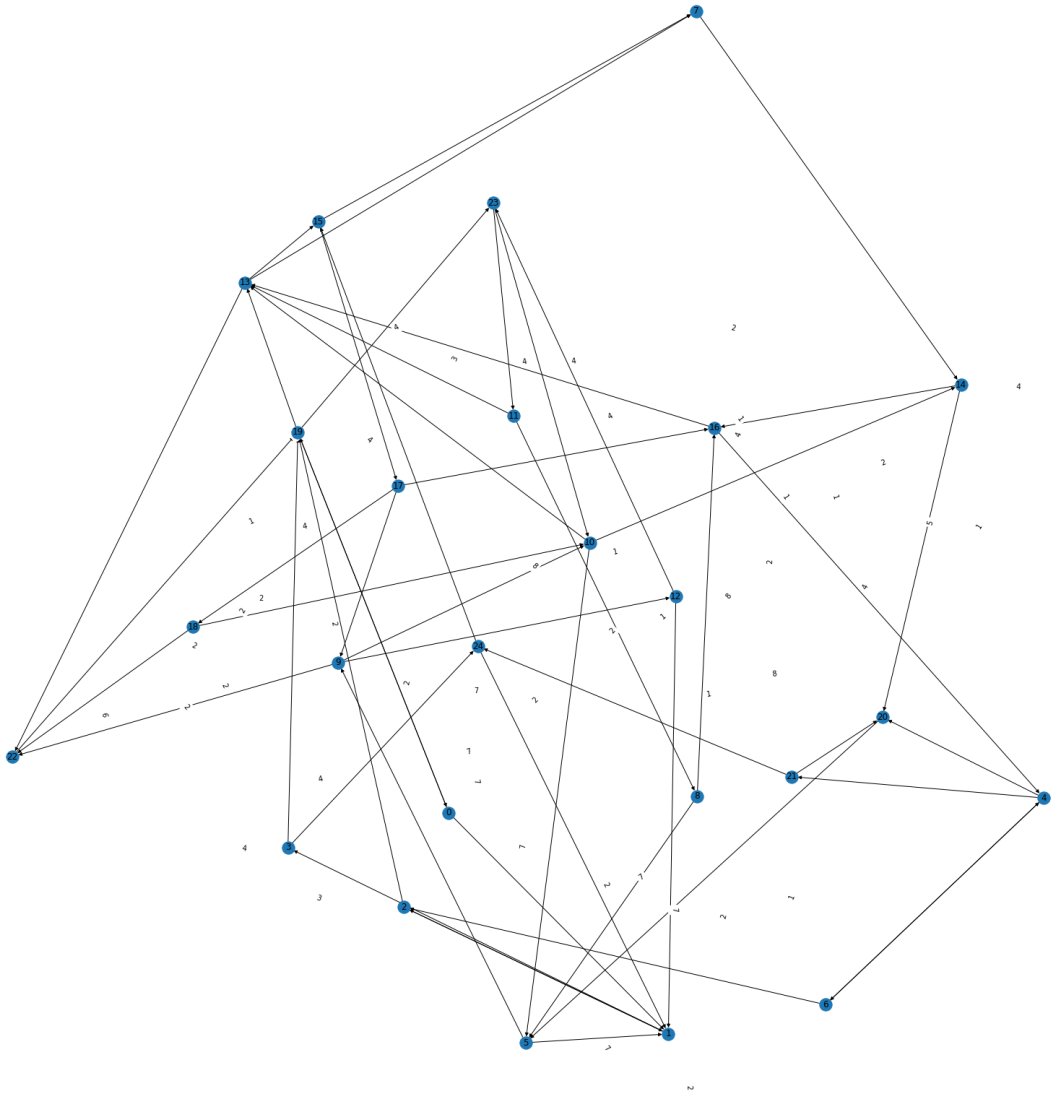

Figure 1.3 k=3 Graph

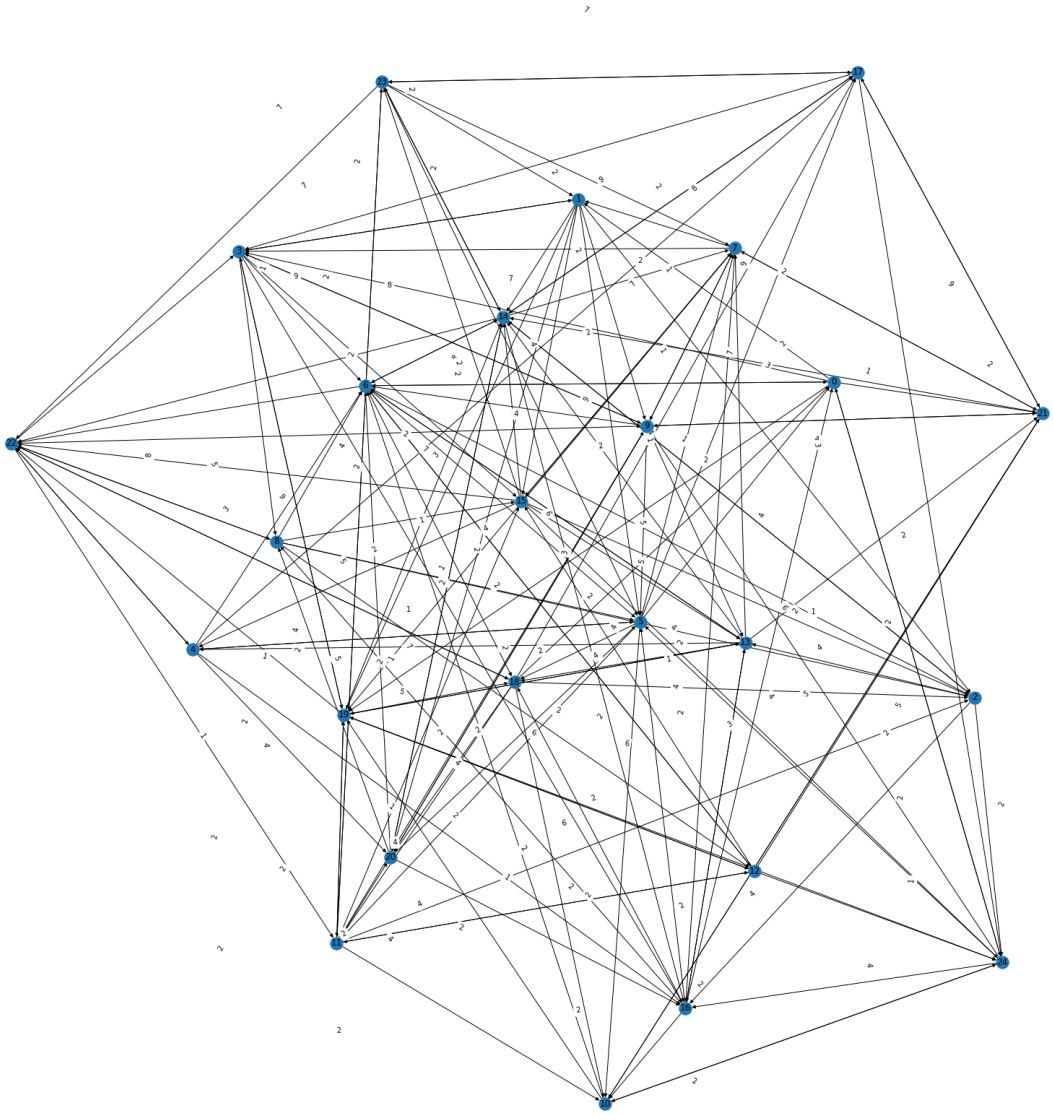

Figure 1.4 k=8 Graph

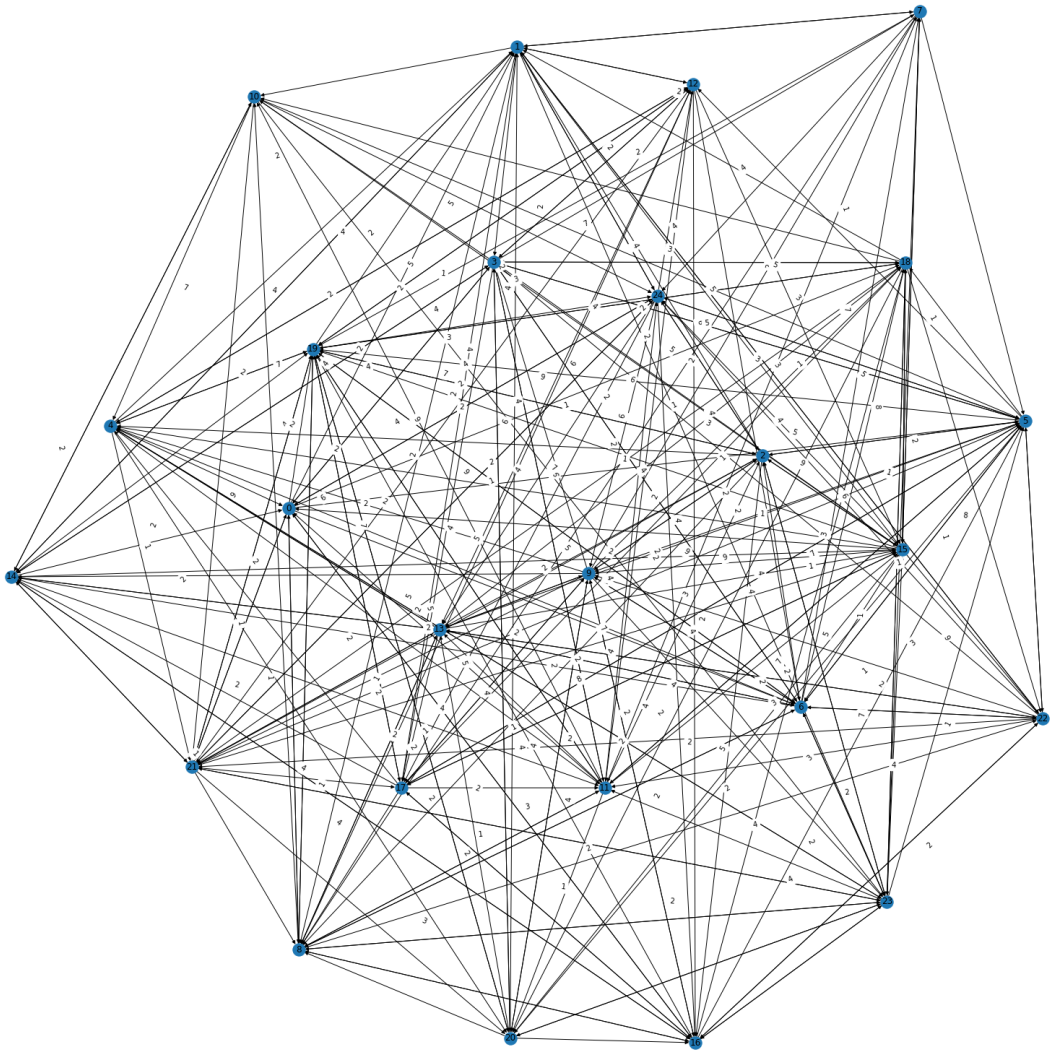

Figure 1.5 k=13 Graph

Analysis

The cost of 250 is prohibitively high, especially in a graph of only 25 nodes. Once k is large enough (in this case k=3), one can reach any node using only low-cost links. If k is increased by one, N*k low-cost links are created. This means that at least N-N*k high-cost links will virtually always be eliminated. This linear expression explains the linear increase in the density for k.

The graph of the optimal cost shows exponential savings for increase in k. Once again, N*k low-cost links are made for each increase in k. Suppose a link is created from node A to B. There are N-1 nodes that lead to A. There are (N-1)^(N-1) nodes that lead to those nodes. Thus we can see the exponential gains as the path gets longer. Lower k-values force longer paths, so the gains are made faster there, and the returns diminish after.

As expected, we see that visually the graphs are getting denser with increase in k.

Appendix

Annotated Source Code

A link to the code can be found here: https://colab.research.google.com/drive/19FkrLuvBxaBzYltfsPq-GuNx2aTCMXPf?usp=sharing

```python
import networkx as nx
import matplotlib.pyplot as plt
import random

#CONSTANTS
N=25
K=8

#SET SEED
random.seed(2001)

G = nx.DiGraph()
G.add_nodes_from(range(N))
print(G)

b_generator= [2,0,2,1,4,6,0,2,4,9,2,0,2,1,4,6,0,2,4,9,2,0,2,1,4]; #my student ID
print(len(b_generator))
a= [[0 for i in range(N)] for j in range(N)] #unit costs
b = [[0 for i in range(N)] for j in range(N)] #demand
c = [[0 for i in range(N)] for j in range(N)] #capacities, to be set as output

#create b matrix
for i in range(len(b)):
    for j in range(len(b[0])):
      if (i != j):
        b[i][j]= abs(b_generator[i]- b_generator[j]);

#initialize a matrix to 250
for i in range(N):
    for j in range(N):
        if (i != j):
          G.add_edge(i, j, weight= 250, capacity=0)#0 is a placeholder, will be updated later

#create lowcost links
for i in range(N):
    valid= list(range(0,N))
    valid.remove(i) #no self loops

    index= random.sample(valid, K); #random.sample ensures that duplicates are not chosen
    for dest in index:
      G.add_edge(i, dest, weight= 1, capacity=0) #low cost links.

print(G.edges.data("weight"))

#get shortest path between any two nodes
totalcost=0;
for k in range(N):
  for l in range(N):
    if (k == l):
```

```
      c[k][l] = 0; #no self loops
    else:
      #find distance between k and l
      cost, path = nx.single_source_dijkstra(G, k, l, weight='weight')
      totalcost += (b[k][l] * cost)

      if (path.index(k) + 1 == path.index(l)): #i.e. there is a k and l in the list next to each other
        c[k][l] += b[k][l] #add to the capacity

#print capacities
for i in range(N):
  print()
  for j in range(N):
    print(c[i][j], end = " ")

print("TOTALCOST: ", totalcost )
print("OLD NUMBER OF EDGES", G.number_of_edges())

#remove zero edges
for k in range(N):
  for l in range(N):
    if (c[k][l] == 0 and k != l):
      #print(k,l)
      G.remove_edge(k, l)
    elif (c[k][l] != 0):
      G[k][l]['capacity']=c[k][l] #set the capacity attribute of the edge from the c array

print("NUMBER OF EDGES", G.number_of_edges())
print("DENSITY", nx.density(G))

#draw
pos = nx.spring_layout(G) #this is a random
plt.figure(3,figsize=(24,24))
nx.draw(G, pos=nx.spring_layout(G), with_labels = True)
nx.draw_networkx_edge_labels(G,pos,edge_labels=nx.get_edge_attributes(G,'capacity')) #label the graph edges with the capacity attribute

#show and save
plt.savefig("k3graph.png")
plt.show()
```

README

This program was written in python. Just type python3 <filename>.py

You may have to install the requisite libraries.

Sources

[1] An Application to Network Design, Class Handout
[2] https://networkx.org/, NetworkX documentation
[3] https://matplotlib.org/, matplotlib documentation
[4] Introduction to Algorithms (CLRS), Graph algorithm information