1. We developed the graph with all its functionalities like Creating a node,Inserting and deleting edges and nodes. Using the graph library we loaded the graph data presented in the textbook.

2. Analysis of various search algorithms evaluated on randomly generated graphs representing networks connected with varying probabilities of edges. The algorithms include BFS, DFS, UCS. We benchmarked these algorithms to determine their efficiency in terms of pathfinding capabilities and execution time across different graph complexities.

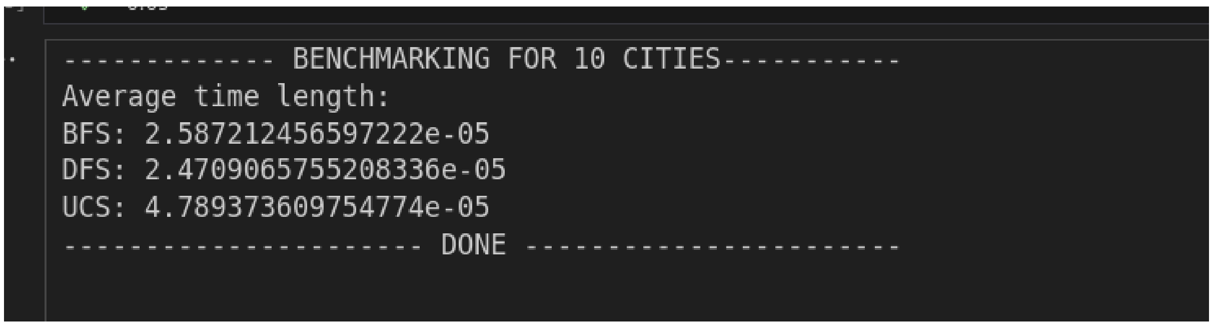
Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors.

Depth-First Search(DFS algorithm) is a recursive algorithm that uses the backtracking principle. It entails conducting exhaustive searches of all nodes by moving forward if possible and backtracking, if necessary. To visit the next node, pop the top node from the stack and push all of its nearby nodes into a stack.

Uniform-cost search(UCS) is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.

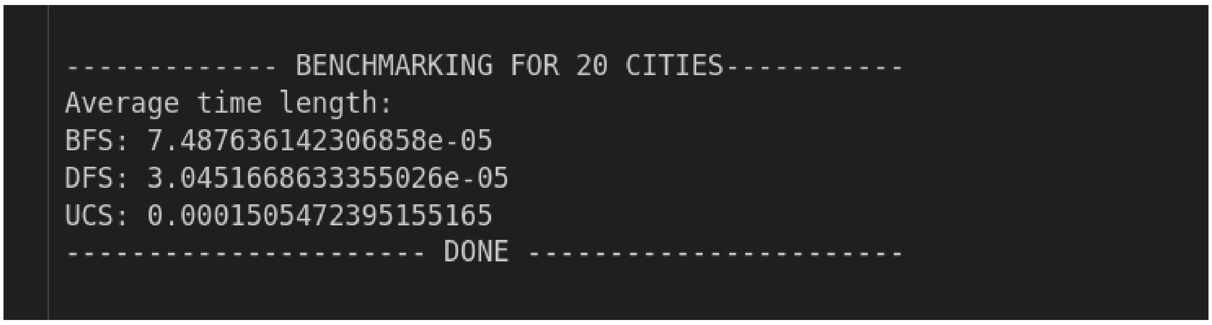
Graphs were generated with node counts set to 10, 20, 30, and 40. Each node was connected to others based on edge probabilities of 0.2, 0.4, 0.6, and 0.8. Nodes were assigned random coordinates, simulating geographical locations, which were used to compute heuristic values for certain algorithms.

Benchmarking
The performance of each algorithm was evaluated by measuring the time taken and the path lengths found between 10 randomly selected nodes within each graph. Each experiment was run 5 times to compute an average for both time and path length.



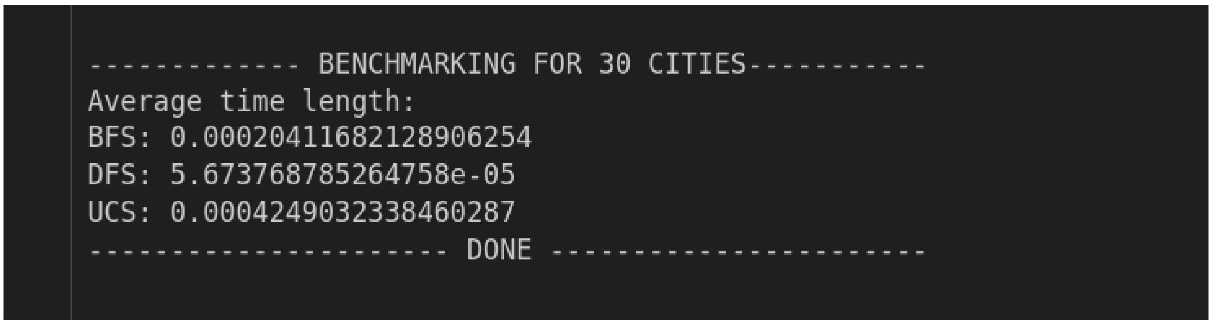
```
------------- BENCHMARKING FOR 10 CITIES-----------
Average time length:
BFS: 2.587212456597222e-05
DFS: 2.4709065755208336e-05
UCS: 4.789373609754774e-05
--------------------- DONE ----------------------
```

DFS has the lowest execution times compared to other algorithms. When the selected node is 10 it is faster than BFS and UCS.



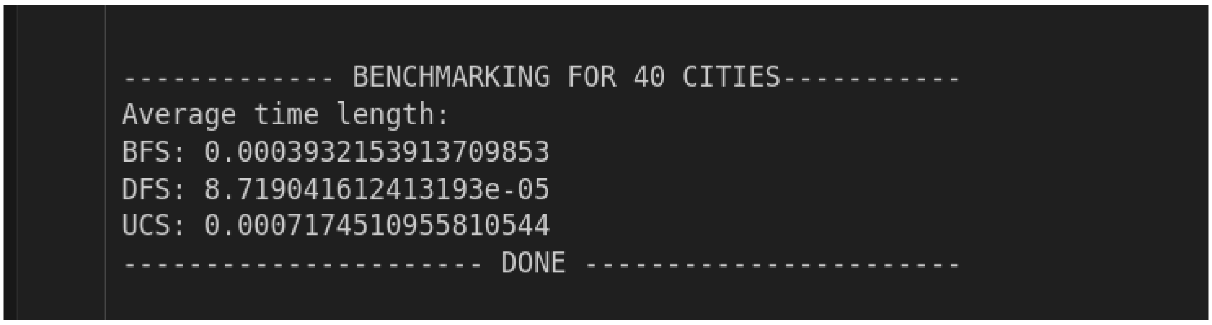
```
------------- BENCHMARKING FOR 20 CITIES-----------
Average time length:
BFS: 7.487636142306858e-05
DFS: 3.0451668633355026e-05
UCS: 0.0001505472395155165
--------------------- DONE ----------------------
```

When the node is 20 the DFS has also the lowest execution times compared to other algorithms. It is faster than BFS and UCS.



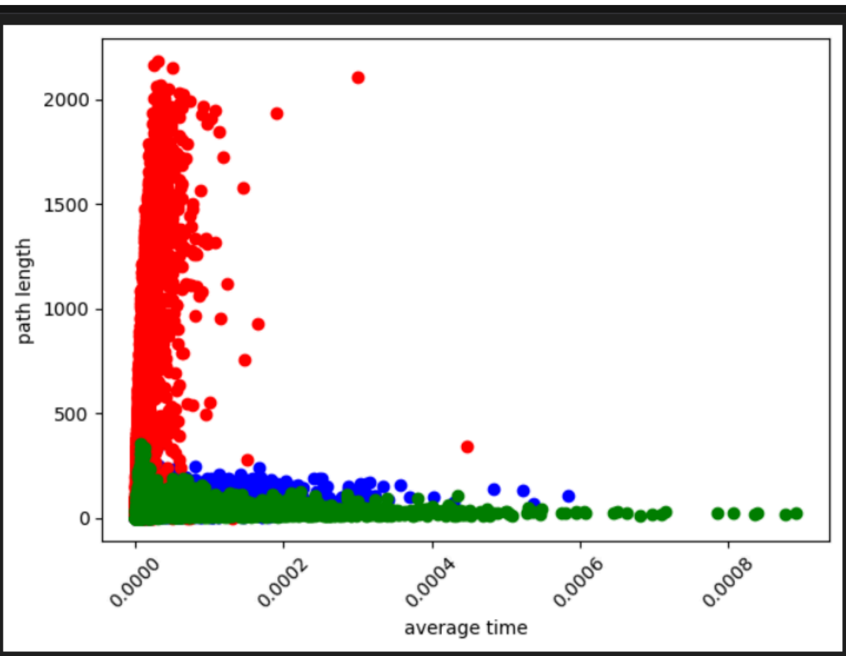
```
------------- BENCHMARKING FOR 30 CITIES-----------
Average time length:
BFS: 0.00020411682128906254
DFS: 5.673768785264758e-05
UCS: 0.0004249032338460287
--------------------- DONE ----------------------
```

When the node is 30 the DFS has also the lowest execution times compared to other algorithms. It is faster than BFS and UCS.

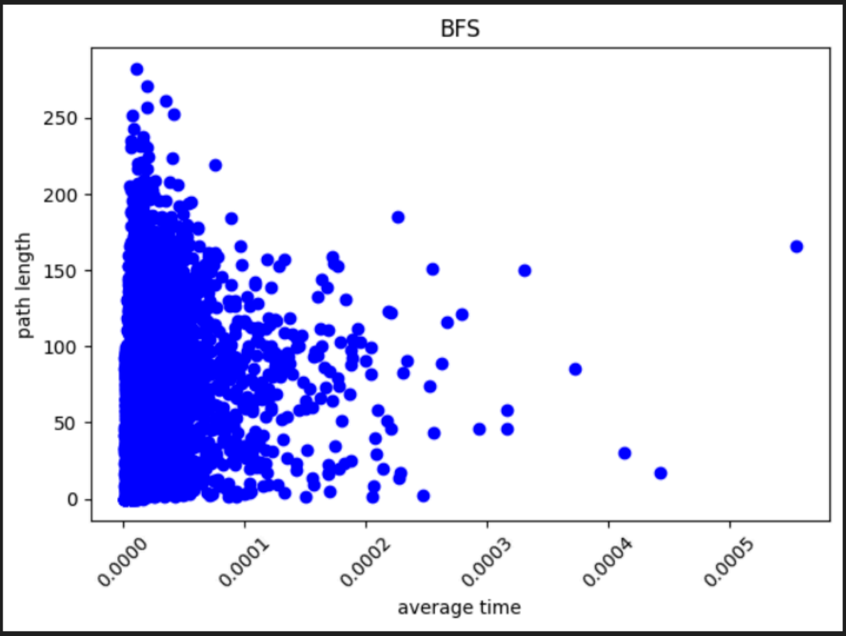


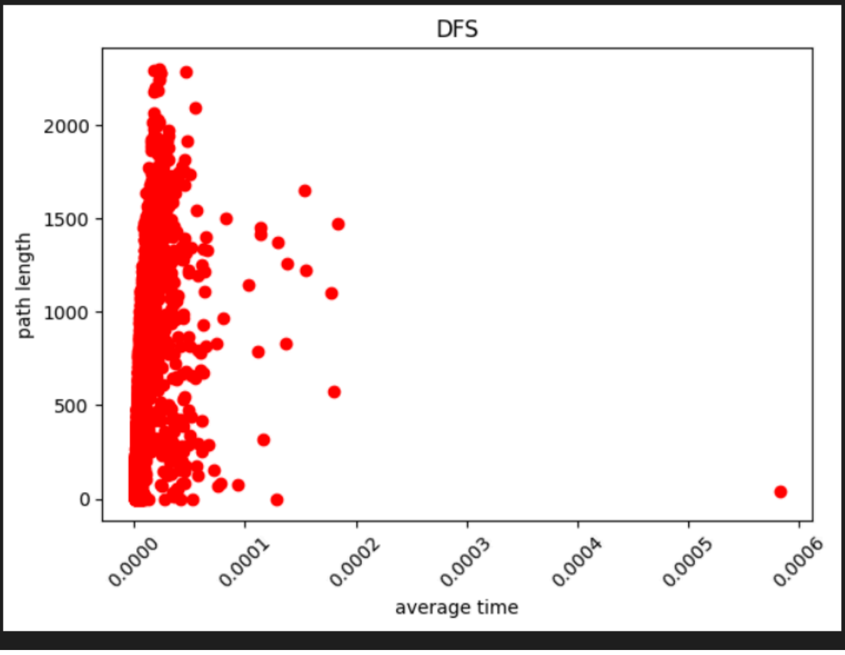
```
------------- BENCHMARKING FOR 40 CITIES-----------
Average time length:
BFS: 0.0003932153913709853
DFS: 8.719041612413193e-05
UCS: 0.0007174510955810544
--------------------- DONE ----------------------
```

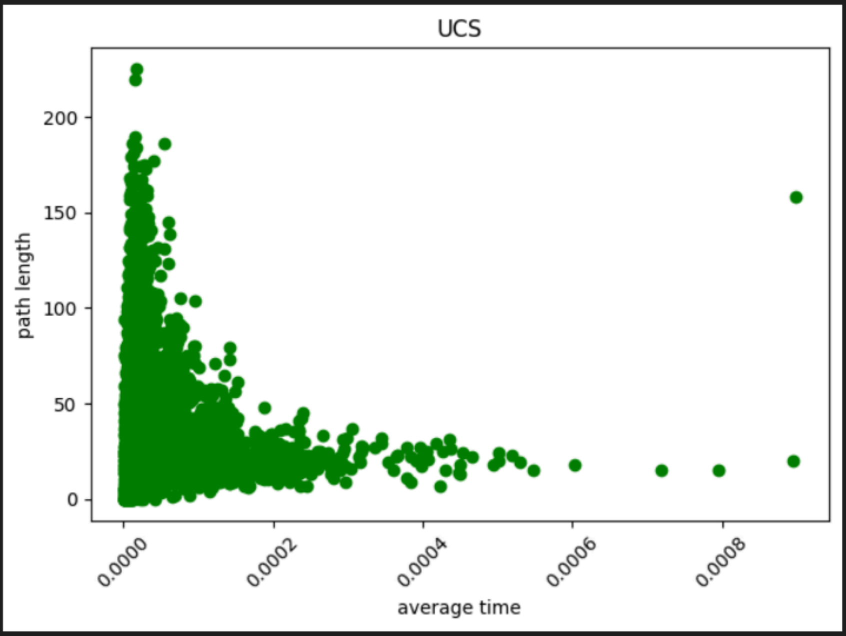
In case node is 40 the DFS has also the lowest execution times compared to other algorithms. It is faster than BFS and UC

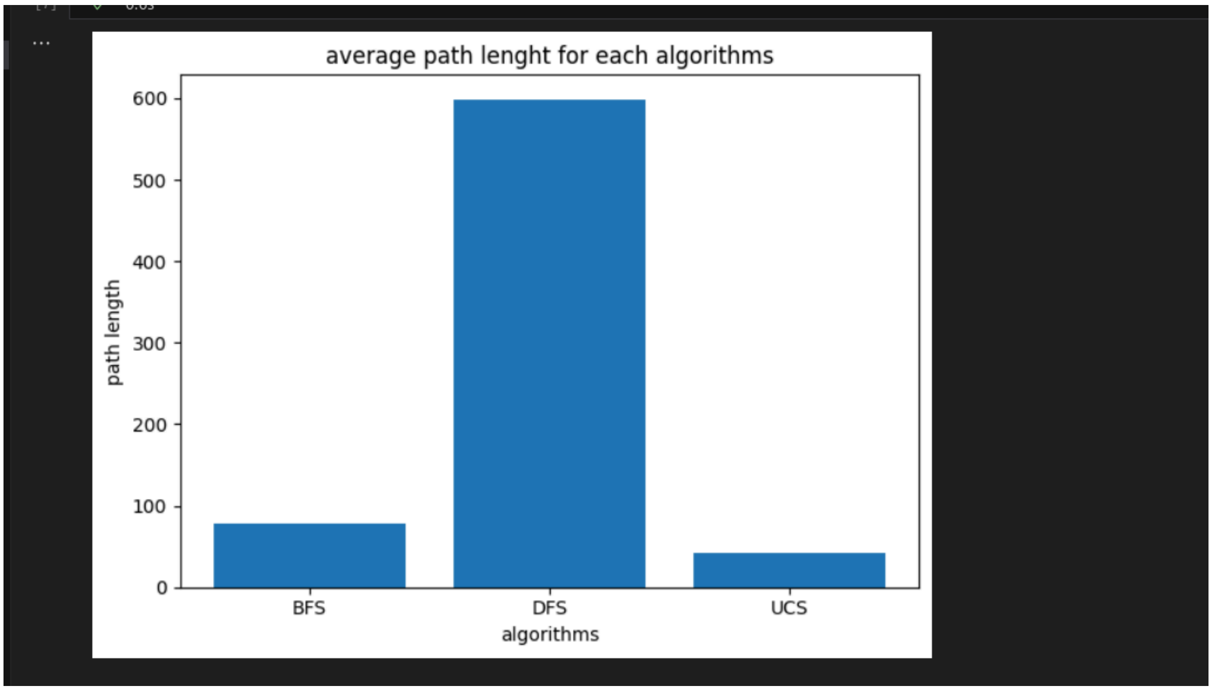


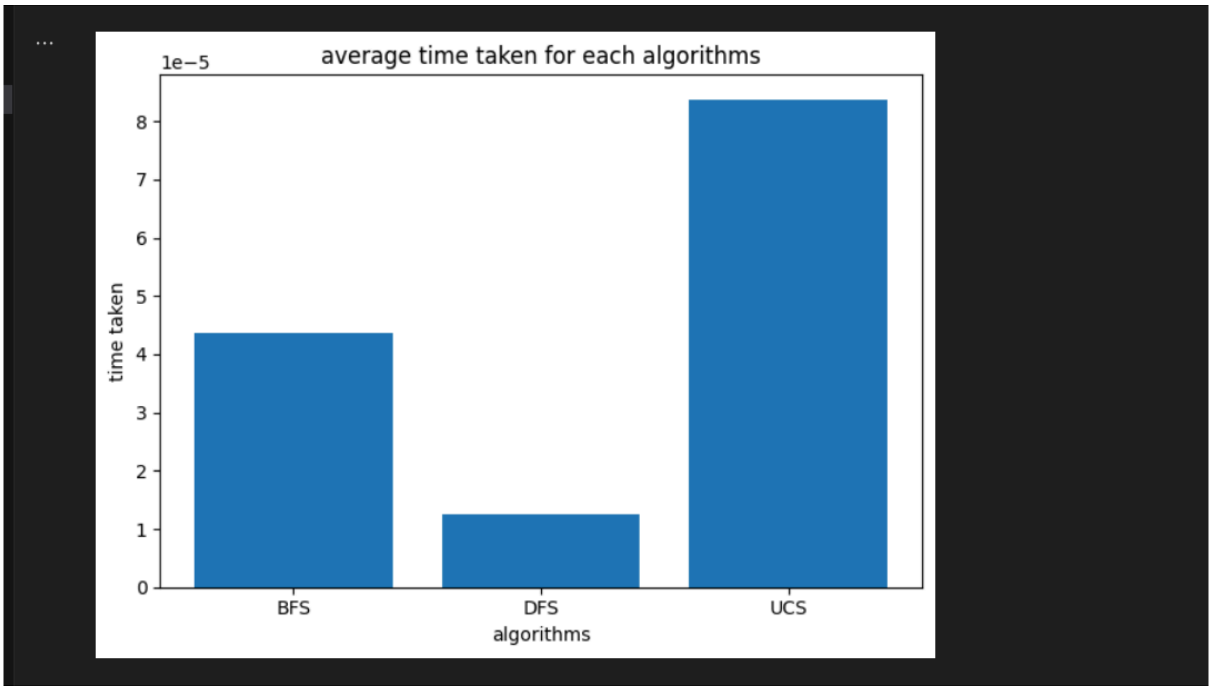
DFS(red color) is the most efficient in terms of computation time which is followed by BFS(blue color) and then UCS(green color).




DFS
path length
2000
1500
1000
500
0
average time
0.0000
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006


UCS
path length
200
150
100
50
0
average time
0.0000
0.0002
0.0004
0.0006
0.0008





BFS tends to have a higher execution time compared to DFS but is generally lower than UCS. Its execution time increases significantly as the number of cities increases, which indicates that it explores many nodes, especially in larger graphs. BFS is complete and optimal for unweighted graphs, meaning it will always find the shortest path if one exists.

DFS consistently has the lowest execution times among the three algorithms across all city sizes. Its time increases with the number of cities but not as sharply as BFS or UCS. While DFS is efficient in terms of speed and memory usage, it is neither complete (it can get stuck in loops) nor necessarily optimal (it may not find the shortest path).

UCS has the highest execution times in all instances. The growth in time is also the steepest as the number of cities increases. This is expected as UCS explores paths in order of their cumulative cost, which requires maintaining a priority queue and can be computationally expensive. UCS is complete and optimal for graphs with any positive edge cost. It is the best choice if finding the least costly path is necessary, regardless of graph size.

In the case of Efficiency DFS is the most efficient in terms of computation time.

3. To compute the centrality of a graph, we'll first define the terms and their formulas. We'll then utilize Python libraries to perform the calculations. The steps involved are:

1. Compute these centralities for each node
2. Report a table containing top-ranked cities in each centrality category
3. Summarize your observations

Degree Centrality

In a graph, degree centrality is a measure of how many edges connect to a node.

Undirected Graphs

C(v) = *deg(v) i.e.* Number of edges connected to node

For undirected graphs, the degree centrality of a node is simply the number of edges connected to that node.

Directed Graphs

For directed graphs, we can calculate the in-degree centrality and out-degree centrality separately.

- In-degree Centrality: Number of edges pointing into the node.
- Out-degree Centrality: Number of edges pointing out of the node.

Closeness Centrality

In a connected graph, the closeness centrality of a node is the average length of the shortest path between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes.

C(v) = $\frac{N-1}{\Sigma u\ d(u,v)}$ where d(u,v) is the distance between the u and v nodes and N is the total number of nodes in the graph.

Katz Centrality

Katz centrality is a generalization of degree centrality. Degree centrality measures the number of direct neighbors, and Katz centrality measures the number of all nodes that can be connected through a path, while the contributions of distant nodes are penalized.

Betweenness Centrality

Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where δst is total number of shortest path from node s to node t and δst(v) is the number of those paths that pass through v.

PageRank Centrality

PageRank is a variant of EigenCentrality, also assigning nodes a score based on their connections, and their connections' connections. The difference is that PageRank also takes link direction and weight into account – so links can only pass influence in one direction, and pass different amounts of influence.
PageRank satisfies the following equation.

$$x_i = \alpha \sum_j a_{ji} \frac{x_j}{L(j)} + \frac{1-c}{N}$$

where

$$L(j) = \sum_i a_{ji}$$

is the number of neighbors of node *j*

Eigenvector Centrality

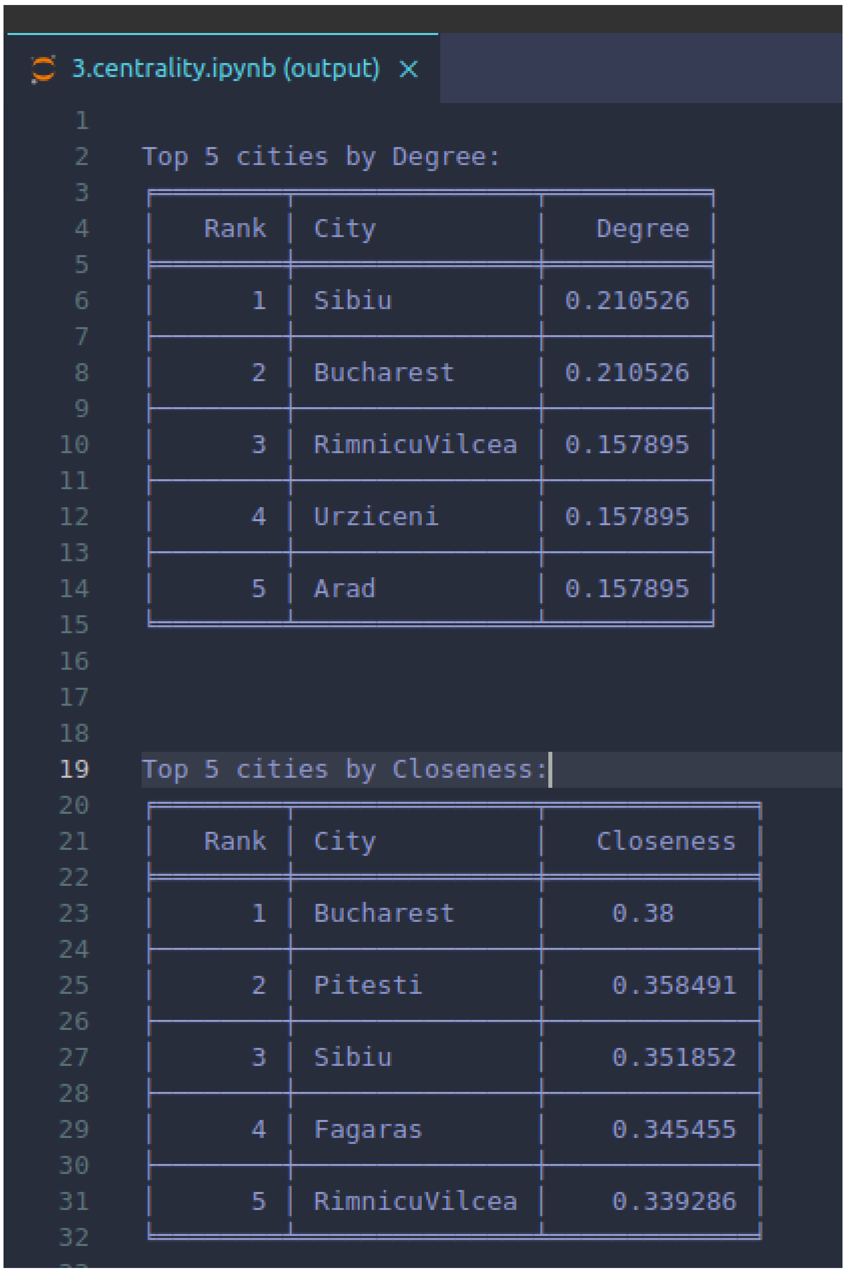
Eigenvector centrality is a measure of the influence of a node in a network.

3.1 Compute these centralities for each node.

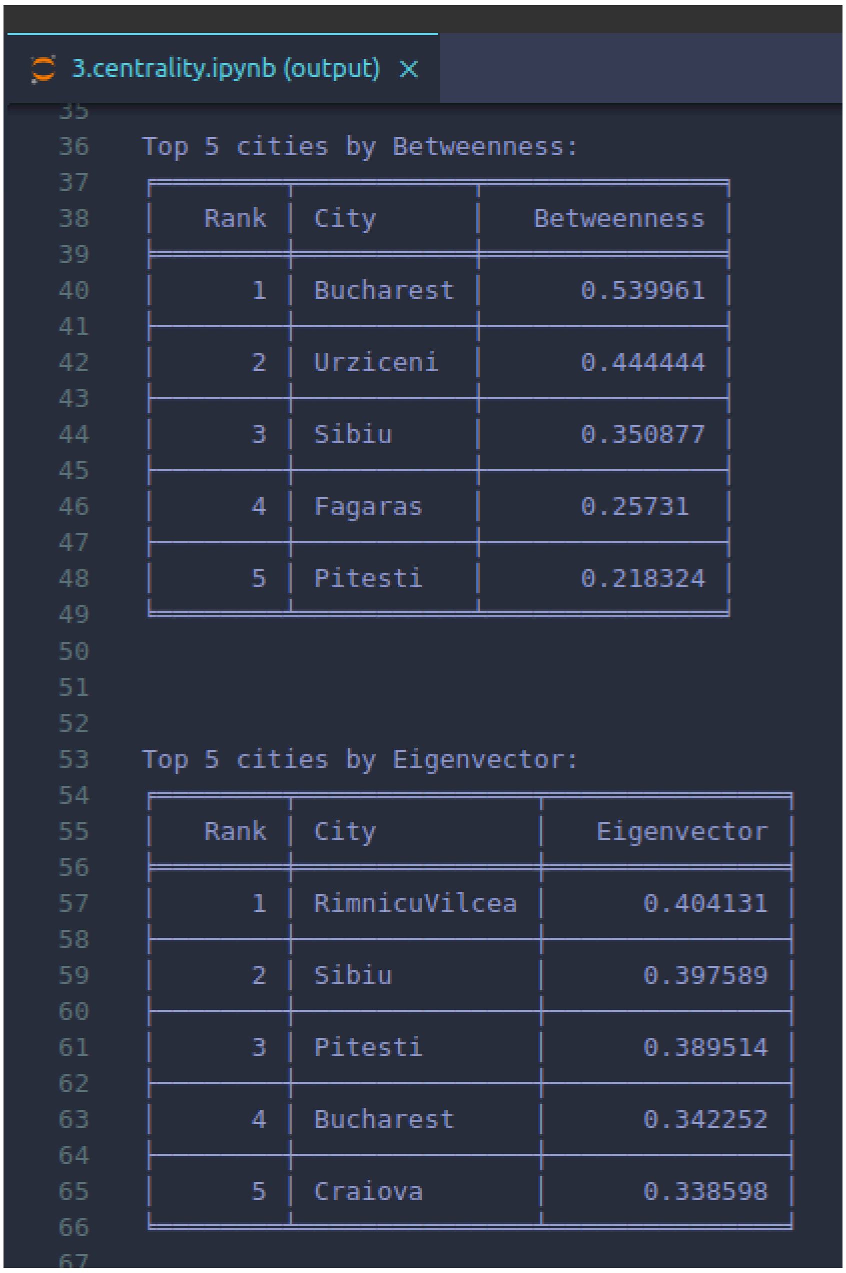
To implement the above centralities I will use a python library called NetworkX.

3.2 The following table shows the top 5 cities by each centrality measure:

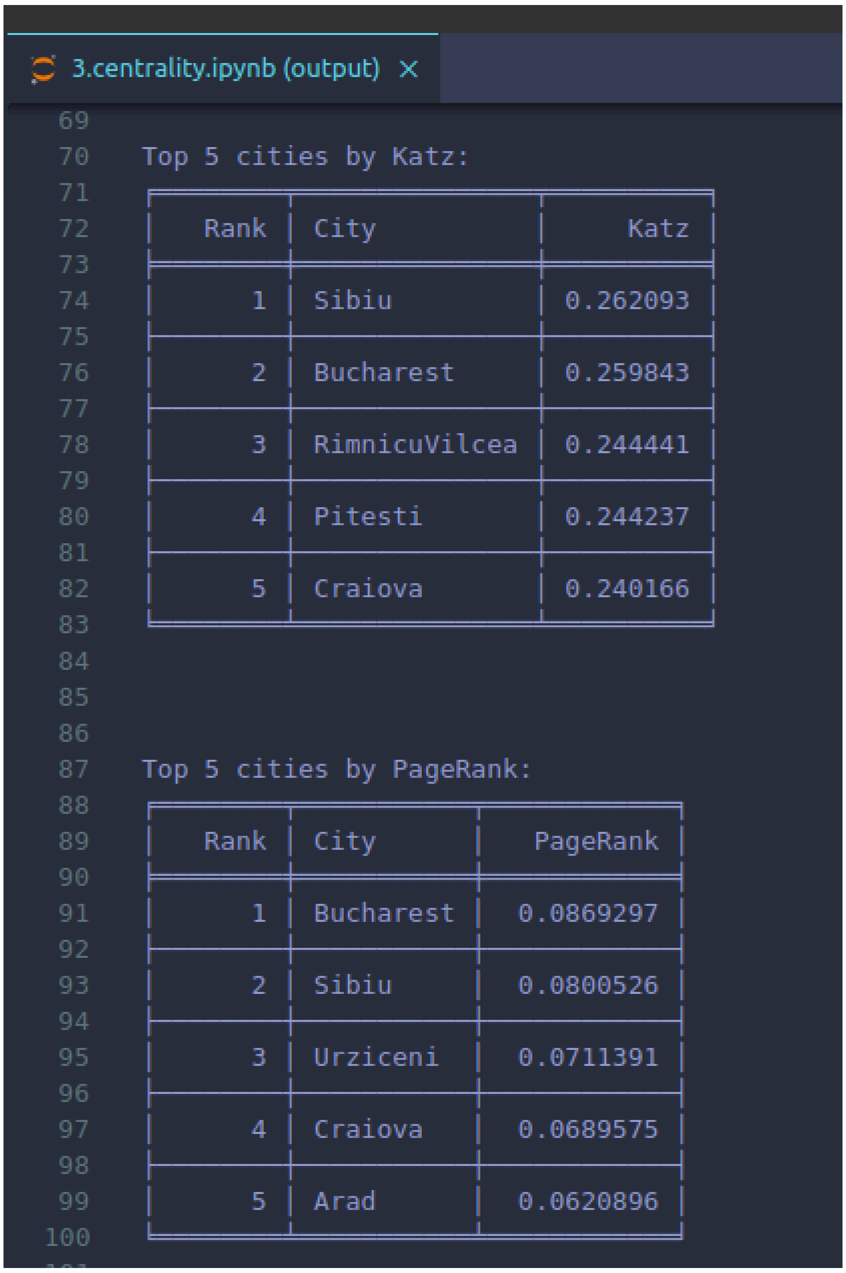
- first for degree and closeness,
- second for betweenness and eigenvector centrality, and
- third for Katz and PageRank.


3.centrality.ipynb (output)

```

Top 5 cities by Degree:
╒════════╤══════════════╤══════════╕
│   Rank │ City         │   Degree │
╞════════╪══════════════╪══════════╡
│      1 │ Sibiu        │ 0.210526 │
├────────┼──────────────┼──────────┤
│      2 │ Bucharest    │ 0.210526 │
├────────┼──────────────┼──────────┤
│      3 │ RimnicuVilcea│ 0.157895 │
├────────┼──────────────┼──────────┤
│      4 │ Urziceni     │ 0.157895 │
├────────┼──────────────┼──────────┤
│      5 │ Arad         │ 0.157895 │
╘════════╧══════════════╧══════════╛



Top 5 cities by Closeness:
╒════════╤══════════════╤═════════════╕
│   Rank │ City         │   Closeness │
╞════════╪══════════════╪═════════════╡
│      1 │ Bucharest    │    0.38     │
├────────┼──────────────┼─────────────┤
│      2 │ Pitesti      │    0.358491 │
├────────┼──────────────┼─────────────┤
│      3 │ Sibiu        │    0.351852 │
├────────┼──────────────┼─────────────┤
│      4 │ Fagaras      │    0.345455 │
├────────┼──────────────┼─────────────┤
│      5 │ RimnicuVilcea│    0.339286 │
╘════════╧══════════════╧═════════════╛
```


3.centrality.ipynb (output)

```
Top 5 cities by Betweenness:
╒════════╤═══════════╤═══════════════╕
│   Rank │ City      │   Betweenness │
╞════════╪═══════════╪═══════════════╡
│      1 │ Bucharest │      0.539961 │
├────────┼───────────┼───────────────┤
│      2 │ Urziceni  │      0.444444 │
├────────┼───────────┼───────────────┤
│      3 │ Sibiu     │      0.350877 │
├────────┼───────────┼───────────────┤
│      4 │ Fagaras   │      0.25731  │
├────────┼───────────┼───────────────┤
│      5 │ Pitesti   │      0.218324 │
╘════════╧═══════════╧═══════════════╛



Top 5 cities by Eigenvector:
╒════════╤══════════════╤═══════════════╕
│   Rank │ City         │   Eigenvector │
╞════════╪══════════════╪═══════════════╡
│      1 │ RimnicuVilcea │      0.404131 │
├────────┼──────────────┼───────────────┤
│      2 │ Sibiu        │      0.397589 │
├────────┼──────────────┼───────────────┤
│      3 │ Pitesti      │      0.389514 │
├────────┼──────────────┼───────────────┤
│      4 │ Bucharest    │      0.342252 │
├────────┼──────────────┼───────────────┤
│      5 │ Craiova      │      0.338598 │
╘════════╧══════════════╧═══════════════╛
```


3.centrality.ipynb (output)

```
Top 5 cities by Katz:
╒════════╤══════════════╤══════════╕
│   Rank │ City         │     Katz │
╞════════╪══════════════╪══════════╡
│      1 │ Sibiu        │ 0.262093 │
├────────┼──────────────┼──────────┤
│      2 │ Bucharest    │ 0.259843 │
├────────┼──────────────┼──────────┤
│      3 │ RimnicuVilcea│ 0.244441 │
├────────┼──────────────┼──────────┤
│      4 │ Pitesti      │ 0.244237 │
├────────┼──────────────┼──────────┤
│      5 │ Craiova      │ 0.240166 │
╘════════╧══════════════╧══════════╛



Top 5 cities by PageRank:
╒════════╤═══════════╤═════════════╕
│   Rank │ City      │    PageRank │
╞════════╪═══════════╪═════════════╡
│      1 │ Bucharest │   0.0869297 │
├────────┼───────────┼─────────────┤
│      2 │ Sibiu     │   0.0800526 │
├────────┼───────────┼─────────────┤
│      3 │ Urziceni  │   0.0711391 │
├────────┼───────────┼─────────────┤
│      4 │ Craiova   │   0.0689575 │
├────────┼───────────┼─────────────┤
│      5 │ Arad      │   0.0620896 │
╘════════╧═══════════╧═════════════╛
```

Observations and generalization from the result of centrality measures:

- Bucharest and Sibiu appear as the most central nodes according to most measures. They both share the highest degree of centrality, indicating they have the most direct connections.

- Bucharest stands out with the highest closeness centrality, suggesting it's the most accessible city, likely because it is centrally located among major cities.
- Bucharest also has the highest betweenness centrality, highlighting its role as a key transit or bridge point in the network. This is indicative of its strategic location in pathways between many other cities.

- Rimnicu Vilcea and Pitesti show significant influence, as evidenced by their high eigenvector centrality, indicating they are connected to other well-connected cities.

- PageRank shows a preference towards Bucharest, suggesting that random walks (or 'traffic' in a real-world analogy) are likely to end up in Bucharest more often than in other cities.

4. Comparing the performance of different graph representations – Adjacency List, Adjacency Matrix, and Edge List.

Time and space complexities of different representations for each operation.

Time Comparison Table

Operation	**Adjacency List**	**Adjacency Matrix**	**Edge List**
Insertion of vertices	O(1)	O(V^2)	O(1)
Insertion of edges	O(1)	O(1)	O(1)
Deletion of vertices	O(V + E)	O(V^2)	O(V + E)
Deletion of edges	O(V)	O(1)	O(E)
Checking for edge existence	O(V)	O(1)	O(E)
Finding neighbors	O(1) (average)	O(V)	O(V)

Space Comparison Table

Representation	**Space Complexity**
Adjacency List	O(V + E)
Adjacency Matrix	O(V^2)
Edge List	O(E)

We examine how each operation behaves in terms of time and space as the size of the input (number of vertices and edges) rises in order to calculate the time and space complexity of each operation and representation.Findings are in terms of Big O notation.

a. Insertion of Vertices:

- **Time Complexity**:
  - Adjacency List: O(1) for each vertex added.
  - Adjacency Matrix: O(V^2) for adding V vertices.
  - Edge List: O(1) for each edge added.
- **Space Complexity**:
  - Adjacency List: O(1) for each vertex added.
  - Adjacency Matrix: O(V^2) for adding V vertices.
  - Edge List: O(1) for each vertex added.

b. Insertion of Edges:
- **Time Complexity**: O(1) for all representations.
- **Space Complexity**:
  - Adjacency List: O(1) for each edge added.
  - Adjacency Matrix: O(1) for each edge added.
  - Edge List: O(1) for each edge added.

c. Deletion of Vertices:
- **Time Complexity:**
  - Adjacency List: O(V + E) for removing a vertex with its associated edges.
  - Adjacency Matrix: O(V^2) for updating all entries in the row and column corresponding to the removed vertex.
  - Edge List: O(V + E) for removing the vertex from the list of edges and updating edge references.
- **Space Complexity**:
  - Adjacency List: O(V + E) for maintaining the vertex and edge lists.
  - Adjacency Matrix: O(V^2) for storing the VxV matrix.
  - Edge List: O(E) for storing the list of edges.

d. Deletion of Edges:
- **Time Complexity**:
  - Adjacency List: O(V) for finding and removing the edge.
  - Adjacency Matrix: O(1) for updating the matrix entry.
  - Edge List: O(E) for finding and removing the edge from the list.
- **Space Complexity**:
  - Adjacency List: O(V + E) for maintaining the vertex and edge lists.
  - Adjacency Matrix: O(V^2) for storing the VxV matrix.
  - Edge List: O(E) for storing the list of edges.

e. Checking for Edge Existence:
- **Time Complexity:**
  - Adjacency List: O(V) for checking the presence of the edge in the adjacency list.
  - Adjacency Matrix: O(1) for accessing the matrix entry.
  - Edge List: O(E) for searching the list of edges.

- **Space Complexity**:
    - Adjacency List: O(V + E) for maintaining the vertex and edge lists.
    - Adjacency Matrix: O(V^2) for storing the VxV matrix.
    - Edge List: O(E) for storing the list of edges.

f. Finding Neighbors of a Vertex:

- **Time Complexity**:
    - Adjacency List: O(1) on average for each neighbor retrieval.
    - Adjacency Matrix: O(V) for accessing all entries in the row/column corresponding to the vertex.
    - Edge List: O(V) for searching and filtering the list of edges.
- **Space Complexity**:
    - Adjacency List: O(V + E) for maintaining the vertex and edge lists.
    - Adjacency Matrix: O(V^2) for storing the VxV matrix.
    - Edge List: O(E) for storing the list of edges.

To compare the performance(efficient) representation depends on the specific operations and requirements of the graph.

- If the graph is sparse and the operations involve vertex insertion/removal or finding neighbors, Adjacency List representation is efficient.
- If the graph is dense and the operations involve checking for edge existence or adjacency, Adjacency Matrix representation is efficient.
- Edge List representation may be suitable when memory is a concern and there is no need for quick edge existence checking.