Instructions: (Please read carefully and follow them!)

Try to solve all problems on your own. If you have difficulties, ask the instructor or TAs.

The implementation of the optimization algorithms in this lab will involve extensive use of the `numpy` Python package. It would be useful for you to get to know some of the functionalities of `numpy` package. For details on `numpy` Python package, please consult https://numpy.org/doc/stable/index.html

For plotting purposes, please use `matplotlib.pyplot` package. You can find examples in the site https://matplotlib.org/examples/.

Please follow the instructions given below to prepare your solution notebooks:

- Please use different notebooks for solving different Exercise problems.
- The notebook name for Exercise 1 should be `YOURROLLNUMBER_IE684_Lab1_Ex1.ipynb`.
- Similarly, the notebook name for Exercise 2 should be `YOURROLLNUMBER_IE684_Lab1_Ex2.ipynb`, etc.

There are only 2 exercises in this lab. Try to solve all problems on your own. If you have difficulties, ask the Instructors or TAs.

ALL QUESTIONS ARE COMPULSORY in this sheet. You can either print the answers using `print` command in your code or you can write the text in a separate text tab. To add text in your notebook, click `+Text`. Some questions require you to provide proper explanations; for such questions, write proper explanations in a text tab. Some questions require the answers to be written in LaTeX notation. Please see the demo video to know how to write LaTeX in Google notebooks. Some questions require plotting certain graphs. Please make sure that the plots are present in the submitted notebooks.

After completing this lab's exercises, click File → Download .ipynb and save your files to your local laptop/desktop. Create a folder with name `YOURROLLNUMBER_IE684_Lab1` and copy your `.ipynb` files to the folder. Then zip the folder to create `YOURROLLNUMBER_IE684_Lab1.zip`. Then upload only the `.zip` file to Moodle. There will be PENALTY for students who do not follow the proper naming conventions in their submissions.

Please check the **submission deadline announced in moodle.**

In this and the upcoming sessions, we will start with discussion and implementation of algorithms for solving discrete optimization problems. Broadly speaking, discrete optimization is a branch of optimization where some or all the variables used are members of a discrete set (for example binary variables or integers) as opposed to a continuous set (for example an interval in $\mathbb{R}$). In this sheet, we will take a look at two discrete optimization problems namely the Set Cover Problem and Shortest Path Problem.

1 Set Cover Problem

1.1 Description

The Set Cover Problem is fundamental in computer science and optimization, falling under the category of discrete optimization. It is often encountered in various real-world scenarios where resources need to be allocated efficiently. The problem can be described as: Given a universe U of n elements and a collection S of subsets of U, the Set Cover Problem asks for the smallest subset of S whose union equals U.

Let us consider a simple example to illustrate the Set Cover Problem:

- Universe: $U = \{1, 2, 3, 4, 5, 6\}$
- Collection of subsets: $S_1 = \{1, 2, 3\}$, $S_2 = \{2, 4\}$, $S_3 = \{3, 4, 5\}$, $S_4 = \{4, 5, 6\}$

In this example, the minimum set cover would be $C = \{S_1, S_4\}$, as the union of S_1 and S_4 covers all elements in the universe U, and $|C| = 2$.

The Set Cover Problem has numerous applications across various domains, including:

- Resource allocation and scheduling problems
- Facility location and network design
- DNA sequencing and bioinformatics
- VLSI design optimization
- Data compression and information retrieval
- Auction Theory and Mechanism Design

1.2 Problem Statement

There are eight locations in Chandanpur City. The city needs to build minimum number of fire stations needed to ensure there is atleast one fire station within 25 minutes (driving time) of each location. The driving time required between different locations is given in the table below:

Exercise 1 (40 marks) *1. Formulate the problem as an integer programming (linear programming with binary variables) problem. Argue it is a set cover problem.*

2. Implement a greedy algorithm (you can use the one given above) and note the solution.

3. Will this algorithm always gives an exact optimal solution? Argue

Table 1: 8x8 Table with Labels

	A	B	C	D	E	F	G	H
A	0	20	25	30	36	40	55	24
B		0	27	15	30	35	60	29
C			0	35	45	45	50	19
D				0	25	30	35	55
E					0	35	20	51
F						0	20	51
G							0	31
H								0

Algorithm 1 Greedy algorithm for Set Cover

Require: Subsets $S_1.S_2...S_m$
1: Initialize $I = \{\}$
2: **while** $I \neq U$ **do**
3: : Find $S_i \in \{S_1, S_2....S_m\}$ such that the number of newly added elements is maximum.
4: : Add elements of above picked S_i to I, i.e., I = I $\cup S_i$
5: **end while**
6: **Output:** S_i which are picked.

4. *What if we needed 20 minutes of driving time, does the number of fire stations and the locations change? If yes, enumerate the new solution.*

5. *Suppose it is decided that there has to be a fire station at location 6. How many more minimum fire stations do we need and where (20 mins driving time)?*

2 Shortest Path Problem

When you use Google maps to see how far the nearby restaurant is from IIT main gate, the Google maps algorithm instantly solves a shortest path problem. As the name suggest, the shortest path problem aims to find the shortest path between two vertices in a weighted graph. Given a graph where each edge has a numerical weight, the objective is to determine the path from a specified starting vertex to a designated destination vertex that minimizes the sum of the edge weights along that path. This problem has numerous applications in various fields such as transportation, network routing, and logistics optimization. There are several algorithms to solve this problem efficiently. In the following exercise, we will implement the classic one: Djiktra's algorithm.

2.1 Djiktra's Algorithm:

Dijkstra's algorithm is a graph search algorithm designed to find the shortest path from a given source node to all other nodes in a weighted graph. It accomplishes this by maintaining a list of tentative distances to each node, initially set to infinity for all nodes except the source node, which is set to zero. It also keeps track of a set of visited nodes. (See References)

The algorithm proceeds in iterations, with each iteration selecting the node with the smallest tentative distance from the set of unvisited nodes. This node is then marked as visited, and its neighbors are examined to update their tentative distances if a shorter path is found through the current node. This process continues until all nodes have been visited, or until the target node is reached if a specific destination is provided.

Given below is one implementation of the algorithm where we define a function with two arguments - the input graph (as an adjacency matrix) and a source city.

Listing 1: Dijkstra's algorithm implementation

```
import sys # for storing large number

def dijkstra(graph, source):
    # Number of vertices in the graph
    vertices = ?

    # Initialize a list of distances from source to all other vertices as large number
    dist = [sys.maxsize]*vertices

    # Initialize distance of source vertex from itself as 0
    dist[src]= 0

    # Initialize list to keep track of visited vertices
    visited = [False] * vertices

    # Loop to find shortest path for all vertices
    for _ in range(vertices):
        # Find the vertex with the minimum distance value, which has not been visited yet
        min_dist = ?
        min_dist_index = 0

        for v in range(vertices):
            if dist[v] < min_dist and not visited[v]:
                min_dist = dist[v]
                min_dist_index = v

        # Mark the minimum distance vertex as visited
        ?

        # Update the distance value of the adjacent vertices of the picked vertex

        for v in range(vertices):
            if graph[min_dist_index][v] > 0 and not visited[v] and
            dist[v] > dist[min_dist_index] + graph[min_dist_index][v]:
                dist[v] = dist[?] + graph[min_dist_index][v]

    return dist
```

2.2 Problem Statement:

Bidyut is an aspiring actor from Guwahati who wants to go to Mumbai in search of fame and fortune. He wants to travel on his motorcycle, and that's a long journey. Riding for day 1 he can reach his friend in Siliguri (0). Now he faces a problem. After spending the night in Siliguri, he can choose to go to either his friend in Patna (1), Jamshedpur (2), or Kolkata (3) to spend the second night. Then from there, he can spend his third night in his friend's home at either Varanasi (4) or Raipur (5). Similarly fourth night he can spend in Indore (6) or Nagpur (7)

and then finally from there he can reach Mumbai (8). He prepares the following graph with distances as edge weights (in × 100 km). (Distance from Jamshedpur (2) to Varanasi (4) is 5.4 × 100 km)

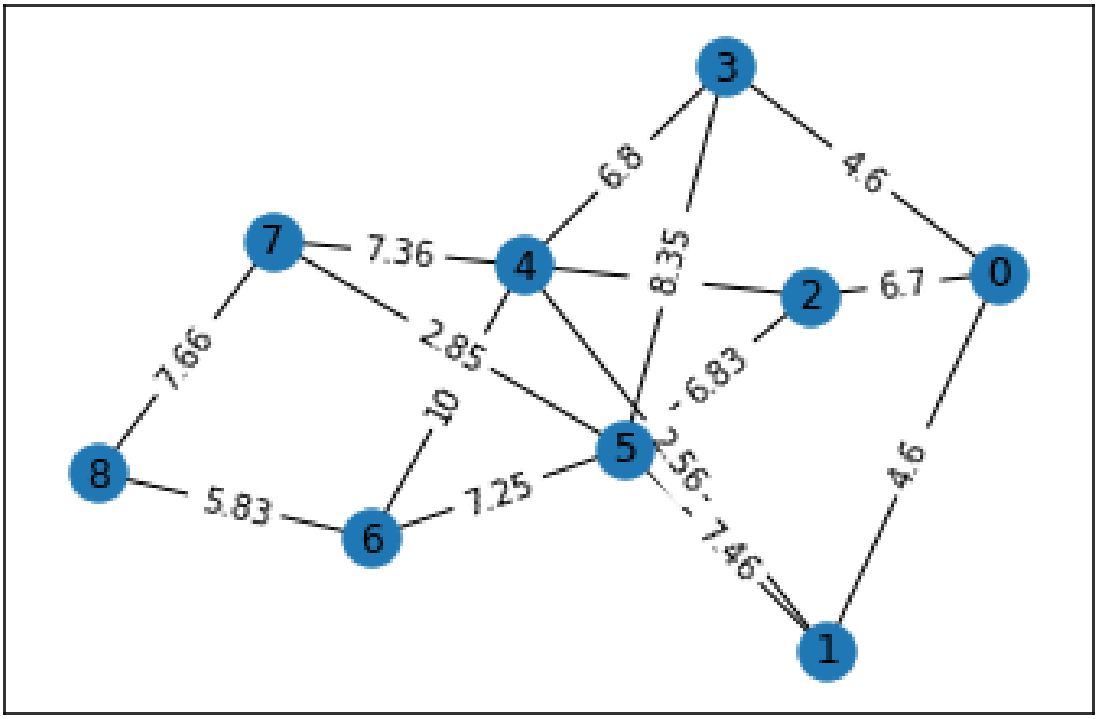

Figure 1: Bidyut's shortest path problem

Exercise 2 (60 marks)

1. *We need to help Bidyut to plan the shortest path in his journey to save his petrol expenses. Implement the above code to find shortest distance from Siliguri to every city. (Hint: use adjacency matrix as data structure).*
2. *Backtrack to find the shortest path from Siliguri to Mumbai using the above result. You can design a code for backtracking for full marks.*
3. *After reaching the second night in the city given by (1), Bidyut learnt from the news that there is roadblock in the path from Varanasi(4) to Nagpur(7). Does his shortest path change ? If yes, what is his new shortest path?*
4. *Seeing his WhatsApp status of travel, Bidyut received a call from his friend in Bhopal that he can spend the fourth night in Bhopal if he wants. Does his optimal path now change? If yes, What is his new optimal path and distance? (Distances from Varanasi:Bhopal = 770 km Raipur:Bhopal = 622 km Bhopal:Mumbai = 760 km) (Note: Using a backtracking code for 2,3,4 correctly will result in full marks)*

References

1. YouTube link for shortest path:
   https://www.youtube.com/watch?v=bZkzH5x0SKU&t=303s
2. Set Cover problem:
   https://optimization.cbe.cornell.edu/index.php?title=Set_covering_problem