

CS 5381 Analysis of Algorithms

Assignment – 4

- Ayyappa Reddy – R11888944

Objective: Implementation of Prim's and Kruskal's algorithm to implement Prim's and Kruskal's algorithms for finding Minimum Spanning Trees (MSTs) in each graph, and to analyse and compare the performance of these algorithms under different conditions.

Part 1: Implementation

Deliverable 1: Submit a well-documented source code package containing two separate files: one for implementing Prim's algorithm and another for Kruskal's algorithm. Each file should include a clearly defined function, `prim_mst(graph)` for Prim's algorithm and `Kruskal_mst(graph)` for Kruskal's algorithm. These functions should take a graph as input and return its minimum spanning tree (MST). The code must be well-commented to explain the logic, and the graph input can be in the form of an adjacency list or matrix, as preferred.

1. Implement Prim's Algorithm as a function `prim_mst(graph)`.

Solution:

```
prim_mst.py > ...
1 import heapq
2 #importing the heapq module to utilize the priority queues of min heap for the edge selections.
3 def prim_mst(graph): #prim_mst will calculate the minimum spanning tree of a graph by using the prim's algorithm.
4
5     num_vertices = len(graph) # Number of vertices in the graph.
6     if num_vertices == 0:
7         return []
8
9     mst = [] # Initialize the list to store the edges of the MST.
10    start_vertex = 0 # Start the algorithm from the first vertex (index 0).
11    visited = set([start_vertex]) # A set to keep track of visited vertices.
12
13    #Initialize a priority queue (min heap) to store edges with their weights.
14    #The MST represented as a list of tuples (start_vertex, end_vertex, weight).
15    #Only include edges that are present (graph[start_vertex][j] is not None).
16    #The graph is represented as an adjacency matrix where graph[i][j] represents the weight of the edge between vertices i and j.
17    #If there's no edge, it's represented by None.
18
19    edges = [(graph[start_vertex][j], start_vertex, j) for j in range(num_vertices) if graph[start_vertex][j] is not None]
20    heapq.heapify(edges) #transforms a list into a heap in linear time and the edges list becomes a min heap and ensuring the edge with the
21    #smallest weight is at the root of the heap.
22
23    # Main loop to construct the MST
24    while len(visited) < num_vertices and edges:
25        weight, frm, to = heapq.heappop(edges) # Select the edge with the minimum weight
26        if to not in visited:
27            visited.add(to) # Mark the vertex as visited
28            mst.append((frm, to, weight)) # Add the edge to the MST
29
30            # Add new edges to the priority queue from the newly added vertex
31            for next_vertex in range(num_vertices):
```

Fig1. prim_mst algorithm

```

32         if next_vertex not in visited and graph[to][next_vertex] is not None:
33             heapq.heappush(edges, (graph[to][next_vertex], to, next_vertex))
34
35     return mst #Returns the minimum spanning tree.
36
37 #graph with the following vertices and edges
38 graph = [
39     [0, 2, None, 6, None, None, None],
40     [2, 0, 3, 8, 5, None, None],
41     [None, 3, 0, None, 7, None, None],
42     [6, 8, None, 0, 9, 4, None],
43     [None, 5, 7, 9, 0, None, 1],
44     [None, None, None, 4, None, 0, 2],
45     [None, None, None, None, 1, 2, 0]
46 ]
47
48 #computing and calculating the total weight of the minimum spanning tree
49 minimum_spanning_tree = prim_mst(graph) #Calculating the minimum spanning tree by using prim's algorithm.
50 total_weight = sum(edge[2] for edge in minimum_spanning_tree) #calculates the total weight of the Minimum spanning tree.
51 print("Total weight of minimum spanning tree:", total_weight) #This displays the total weight of the minimum spanning tree.
52 print("Minimum spanning tree for prim_mst(graph):", minimum_spanning_tree) #This will displays the minimum spanning tree with the edges.

```

Fig2. Graph Input of Prim's algorithm

```

c:/Users/harle/OneDrive/Desktop/Algos/prims_mst.py
Total weight of minimum spanning tree: 17
Minimum spanning tree for prim_mst(graph): [(0, 1, 2), (1, 2, 3), (1, 4, 5), (4, 6, 1), (6, 5, 2), (5, 3, 4)]

```

Fig3. Output for prim_mst algorithm

2. Implement Kruskal's Algorithm as a function `Kruskal_mst(graph)`.

Solution:

```

kruskal_mst.py > kruskal_mst
1 class DisjointSet:
2     #Initializes the disjoint set with each vertices in its own set.
3     def __init__(self, vertices):
4
5         #Creating a disjoint set where each vertex is its own parent initially.
6         self.parent = {vertex: vertex for vertex in vertices}
7         #Initializin the rank of each vertex as 0.
8         self.rank = {vertex: 0 for vertex in vertices}
9
10    def find(self, vertex):
11        #Finds the root of the set that a vertex belongs to.
12
13        #Recursive function to find the root parent of a vertex with the path compression.
14        if self.parent[vertex] != vertex:
15            self.parent[vertex] = self.find(self.parent[vertex])
16        return self.parent[vertex]
17
18    def union(self, vertex1, vertex2):
19        #Merges two sets that the vertex belongs to.
20
21        #Finding the root parents of the sets that vertex1 and vertex2 belongs to
22        root1 = self.find(vertex1)
23        root2 = self.find(vertex2)
24
25        if root1 != root2:
26            #Union by rank: Attaching the set with lower rank to the one with higher rank
27            if self.rank[root1] < self.rank[root2]:
28                self.parent[root1] = root2
29            elif self.rank[root1] > self.rank[root2]:
30                self.parent[root2] = root1

```

Fig4. Kruskal Algorithm

```

kruskal_mst.py > kruskal_mst
31         else:
32             #If the ranks are equal, attach one set to the other and increment rank.
33             self.parent[root2] = root1
34             self.rank[root1] += 1
35
36 def kruskal_mst(graph):
37
38     #Calculates the minimum spanning tree (MST) of a graph using Kruskal's algorithm.
39     #Each tuple is an edge in the MST and contains (start_vertex, end_vertex, weight).
40
41     #Determine the total number of vertices in the graph by calculating the length of the graph list.
42     #this values represents the total number of vertices present in the given graph structure.
43     num_vertices = len(graph)
44
45     # Initialize an empty list named 'edges'.
46     # This list will store the edges extracted from the graph's adjacency matrix to be used in Kruskal's algorithm.
47     edges = []
48
49     #Extracting all edges from the adjacency matrix
50     #graph (list of lists): A graph represented as an adjacency matrix.
51     for i in range(num_vertices):
52         for j in range(i + 1, num_vertices): # To avoid duplicate edges
53             if graph[i][j] is not None and graph[i][j] != float('inf'):
54                 edges.append((graph[i][j], i, j))
55
56     # Sorting edges based on weight
57     edges.sort(key=lambda x: x[0])
58
59     # Initialize disjoint sets for each vertices
60     ds = DisjointSet(range(num_vertices))
61     mst = []

```

Fig5. Kruskal_mst algorithm

```

kruskal_mst.py > kruskal_mst
62
63     # Iterate through sorted edges and add them to MST if they don't form a cycle
64     for weight, start, end in edges:
65         if ds.find(start) != ds.find(end):
66             ds.union(start, end)
67             mst.append((start, end, weight))
68
69     return mst
70
71 #graph with the input of edges by using adjacency matrix
72 graph = [
73     [None, 4, None, None, None, None, None, 8, None],
74     [4, None, 8, None, None, None, None, 11, None],
75     [None, 8, None, 7, None, 4, None, None, 2],
76     [None, None, 7, None, 9, 14, None, None, None],
77     [None, None, None, 9, None, 10, None, None, None],
78     [None, None, 4, 14, 10, None, 2, None, None],
79     [None, None, None, None, None, 2, None, 1, 6],
80     [8, 11, None, None, None, None, 1, None, 7],
81     [None, None, 2, None, None, None, 6, 7, None]
82 ]
83
84 #computing and calculating the total weight of the minimum spanning tree
85 minimum_spanning_tree = kruskal_mst(graph) #Finding the minimum spanning tree
86 total_weight = sum(edge[2] for edge in minimum_spanning_tree) #calculating the total weight of Minimum spanning tree edges
87 print("Total weight of minimum spanning tree:", total_weight) #Displays the total weight of the Minimum spanning tree.
88 print("Minimum spanning tree for kruskal_mst(graph):", minimum_spanning_tree) #displays the minimum spanning tree edges.

```

Fig6. Graph Input for Kruskal algorithm

```

c:/Users/harle/OneDrive/Desktop/Algos/kruskal_mst.py
Total weight of minimum spanning tree: 37
Minimum spanning tree for kruskal_mst(graph): [(6, 7, 1), (2, 8, 2), (5, 6, 2), (0, 1, 4), (2, 5, 4), (2, 3, 7), (0, 7, 8), (3, 4, 9)]

```

Fig7. Output for Kruskal_mst algorithm

Part 2: Testing and Validation

Deliverable 2: Provide a comprehensive testing document that includes detailed descriptions and results of at least three different graph test cases used to validate both algorithms. These graphs should represent a variety of types, including a sparse graph, a dense graph, and a graph with varied edge weights. For each test case, the document should clearly present the input graph, the MSTs obtained from both Prim's and Kruskal's algorithms, and a summary of the total weight of each MST. This will demonstrate the correctness and functionality of your implementations.

1. Create at least three different graphs to test your algorithms. These should include:
 - a) A sparse graph with 10-15 nodes.
 - b) A dense graph with 10-15 nodes.
 - c) A graph with varying edge weights, including negative weights, if your implementation allows.
2. Record and analyze the output of each algorithm on these graphs.
 - Run both algorithms on each of the graphs.
 - Record the output MSTs and their total weights.

Solution:

A sparse graph using 14 nodes for the function `prim_mst` of `prims` algorithm:

A sparse adjacency matrix is a representation of a graph where most entries are empty, indicating the absence of edges between most pairs of vertices. In a sparse graph, the number of edges is significantly less than the maximum possible number of edges.

```
# Sparse Graph with 14 nodes (few of the edges compared to nodes):
sparse_graph = [
    # Input of a sparse graph represented as an adjacency matrix
    # Nodes have connections with a few other nodes (few edges compared to nodes)

    [None, 5, None, None, None, None, None, None, None, None, None, None, None, None],
    [5, None, 6, None, None, None, None, None, None, None, None, None, None, None],
    [None, 6, None, 7, None, None, None, None, None, None, None, None, None, None],
    [None, None, 7, None, 8, None, None, None, None, None, None, None, None, None],
    [None, None, None, 8, None, 9, None, None, None, None, None, None, None, None],
    [None, None, None, None, 9, None, 10, None, None, None, None, None, None, None],
    [None, None, None, None, None, 10, None, 11, None, None, None, None, None, None],
    [None, None, None, None, None, None, 11, None, 12, None, None, None, None, None],
    [None, None, None, None, None, None, None, 12, None, 13, None, None, None, None],
    [None, None, None, None, None, None, None, None, 13, None, 14, None, None, None],
    [None, None, None, None, None, None, None, None, None, 14, None, 15, None, None],
    [None, None, None, None, None, None, None, None, None, None, 15, None, 16, None],
    [None, None, None, None, None, None, None, None, None, None, None, 16, None, 17],
    [None, None, None, None, None, None, None, None, None, None, None, None, 17, None]
]
```

Fig8. Graph input for sparse graph of `prim_mst`(graph) algorithm

The above-mentioned graph is the input for the sparse graph of prim's algorithm with 14 nodes. A sparse graph is a type of graph where the number of edges is much less than the maximum numbers of edges that exist between the given number of vertices.

Prim's algorithm is a most used method to find the minimum spanning tree in a weighted undirected graph. The minimum spanning tree is a subset of edges that connects to all the vertices in the graph without forming any cycles and also has the minimum possible total edge weight.

At first, we will input the sparse graph within the form of representing as an adjacency matrix with 14 nodes. The graph exhibits a low density of edges compared to nodes by demonstrating a scenario where few nodes have connections with the other nodes.

```
"c:/Users/harle/OneDrive/Desktop/Algos/sparse_graph(prim).py"
Total weight of minimum spanning tree: 143
Minimum spanning tree for prim_mst(sparse_graph): [(0, 1, 5), (1, 2, 6), (2, 3, 7), (3, 4, 8), (4, 5, 9), (5, 6, 10), (6, 7, 11), (7, 8, 12), (8, 9, 13), (9, 10, 14), (10, 11, 15), (11, 12, 16), (12, 13, 17)]
```

Fig9. Output for prim_mst algorithm for the sparse graph.

The above is the output for the prim_mst sparse graph algorithm which represents the total weight of the graph and the minimum spanning tree of prim's sparse graph. The total weight of the generated minimum spanning tree for the sparse graph is calculated as 143. The minimum spanning tree itself, consisting of edges connecting nodes to form the tree, is listed as follows: [(0, 1, 5), (1, 2, 6), (2, 3, 7), (3, 4, 8), (4, 5, 9), (5, 6, 10), (6, 7, 11), (7, 8, 12), (8, 9, 13), (9, 10, 14), (10, 11, 15), (11, 12, 16), (12, 13, 17)].

The above implementation of the prim's algorithm is efficient to provide the minimum spanning tree for a sparse graph. By selecting edges with at least weight while exploring a limited number of connections, the algorithm produced a minimum spanning tree that optimally connected the nodes with minimal total weight.

A sparse graph using 14 nodes for the function Kruskal_mst of Kruskal's algorithm:

```
#Sparse Graph with 14 nodes (few of the edges compared to nodes) with an adjacency matrix.
sparse_graph = [
    [None, 5, None, None, None, None, None, None, None, None, None, None, None, None],
    [5, None, 6, None, None, None, None, None, None, None, None, None, None, None],
    [None, 6, None, 7, None, None, None, None, None, None, None, None, None, None],
    [None, None, 7, None, 8, None, None, None, None, None, None, None, None, None],
    [None, None, None, 8, None, 9, None, None, None, None, None, None, None, None],
    [None, None, None, None, 9, None, 10, None, None, None, None, None, None, None],
    [None, None, None, None, None, 10, None, 11, None, None, None, None, None, None],
    [None, None, None, None, None, None, 11, None, 12, None, None, None, None, None],
    [None, None, None, None, None, None, None, 12, None, 13, None, None, None, None],
    [None, None, None, None, None, None, None, None, 13, None, 14, None, None, None],
    [None, None, None, None, None, None, None, None, None, 14, None, 15, None, None],
    [None, None, None, None, None, None, None, None, None, None, 15, None, 16, None],
    [None, None, None, None, None, None, None, None, None, None, None, 16, None, 17],
    [None, None, None, None, None, None, None, None, None, None, None, None, 17, None]
]
```

Fig10. Graph input for sparse graph of Kruskal_mst(graph) algorithm

The above graph inputs for the sparse graph that implemented code is effectively applying Kruskal's algorithm to find the minimum spanning tree of a sparse graph represented as

an adjacency matrix with 14 nodes. The algorithm effectively computes the minimum spanning tree, ensuring the total weight of the tree is minimized.

Kruskal's algorithm is used to find the minimum spanning tree in a connected, weighted graph. The MST of a graph is the subset of edges that connects all the vertices together with the minimum of total edge weight and forms a tree without any cycles.

At first, we will input the sparse graph within the form of representing as an adjacency matrix with 14 nodes. Each node has connections with only fewer nodes, resulting in a sparse arrangement of edges compared to node.

```
"c:/Users/harle/OneDrive/Desktop/Algos/sparse_graph(kruskal).py"  
Total weight of minimum spanning tree: 143  
Minimum spanning tree for Kruskal_mst(sparse_graph): [(0, 1, 5), (1, 2, 6), (2, 3, 7), (3, 4, 8), (4, 5, 9), (5, 6, 10), (6, 7, 11), (7, 8, 12), (8, 9, 13), (9, 10, 14), (10, 11, 15), (11, 12, 16), (12, 13, 17)]
```

Fig11. Output for Kruskal_mst algorithm for the sparse graph.

The total weight of the generated minimum spanning tree for the sparse graph is calculated as 143. The minimum spanning tree itself, consisting of edges connecting nodes to from the tree, is listed as follows: [(0, 1, 5), (1, 2, 6), (2, 3, 7), (3, 4, 8), (4, 5, 9), (5, 6, 10), (6, 7, 11), (7, 8, 12), (8, 9, 13), (9, 10, 14), (10, 11, 15), (11, 12, 16), (12, 13, 17)].

This implementation shows that the successful application of Kruskal's algorithm in constructing a minimum spanning tree for a sparse graph scenario, there by highlighting its usefulness in optimizing the edge connections to minimize the total weight in network-like structures with sparse connectivity.

A dense graph using 14 nodes for the function prim_mst of prim's algorithm:

A dense adjacency matrix is a representation of a graph where most of the vertexes are connected resulting in a matrix that is mostly filled with non-null values (weights) to represent the edges between vertices. In a dense graph, the number of edges is close to the maximum possible number of edges.

```

43 # Dense Graph with 14 nodes (more edges, closer to a complete graph):
44 dense_graph = [
45     # Input of a dense graph represented as an adjacency matrix
46     # Nodes have connections with many other nodes (more edges compared to nodes)
47
48     [None, 5, 9, 2, 4, 1, 6, 7, 10, 8, 12, 3, 2, 4],
49     [5, None, 3, 6, 7, 9, 8, 10, 2, 11, 5, 4, 2, 5],
50     [9, 3, None, 1, 5, 8, 9, 4, 6, 7, 3, 5, 7, 1],
51     [2, 6, 1, None, 9, 7, 5, 3, 4, 6, 2, 8, 1, 7],
52     [4, 7, 5, 9, None, 6, 4, 2, 3, 5, 9, 7, 3, 6],
53     [1, 9, 8, 7, 6, None, 2, 5, 7, 8, 4, 6, 9, 1],
54     [6, 8, 9, 5, 4, 2, None, 1, 3, 2, 9, 7, 8, 3],
55     [7, 10, 4, 3, 2, 5, 1, None, 6, 4, 5, 3, 2, 9],
56     [10, 2, 6, 4, 3, 7, 3, 6, None, 1, 8, 2, 4, 5],
57     [8, 11, 7, 6, 5, 8, 2, 4, 1, None, 7, 6, 3, 2],
58     [12, 5, 3, 2, 9, 4, 9, 5, 8, 7, None, 1, 6, 4],
59     [3, 4, 5, 8, 7, 6, 7, 3, 2, 6, 1, None, 5, 8],
60     [2, 2, 7, 1, 3, 9, 8, 2, 4, 3, 6, 5, None, 7],
61     [4, 5, 1, 7, 6, 1, 3, 9, 5, 2, 4, 8, 7, None]
62 ]

```

Fig12. Graph input for dense graph of prim_mst(graph) algorithm

The above mentioned is the input graph for the dense graph of a prim's algorithm. This dense graph represented by the adjacency matrix shows the connection between nodes. Each row corresponds to a node and the elements in the row represent the weights of the edges from that node to all other nodes. 'None' indicates no direct edge between the nodes. Prim's algorithm finds the minimum spanning tree in a graph by selecting the lowest-weight edges.

At first, the prim's algorithm will calculate the minimum spanning tree of a given dense graph using an adjacency matrix representation. The prim_mst function accepts a graph in the form of a list of lists, where each entry represents the weight of an edge between the nodes.

The dense graph provided consists of 14 nodes, showing the dense connection structure. Each node has multiple edges connecting it to other nodes, resulting in a higher edge density compared to the number of nodes.

```

"c:/Users/harle/OneDrive/Desktop/Algos/dense_graph(prim).py"
Total weight of minimum spanning tree: 18
Minimum spanning tree for prim_mst(dense_graph): [(0, 5, 1), (5, 13, 1), (13, 2, 1), (2, 3, 1), (3, 12, 1), (3, 10, 2), (10, 11, 1), (5, 6, 2), (6, 7, 1), (6, 9, 2), (9, 8, 1), (7, 4, 2), (8, 1, 2)]

```

Fig13. Output for prim_mst algorithm for the dense graph.

After executing the code with the given dense graph, the above is the output for the prim_mst algorithm for the dense graph. The total weight of the resulting minimum spanning tree is calculated to be 18. The minimum spanning tree itself, consisting of edges connecting nodes to form the tree, is listed as follows: [(0, 5, 1), (5, 13, 1), (13, 2, 1), (2, 3, 1), (3, 12, 1), (3, 10, 2), (10, 11, 1), (5, 6, 2), (6, 7, 1), (6, 9, 2), (9, 8, 1), (7, 4, 2), (8, 1, 2)].

The implementation of prim's algorithm demonstrated its ability to efficiently calculate the minimum spanning tree even in a dense graph scenario. Despite the increased edge density, the algorithm effectively produced the optimal solution, by showing its reliability in various graph structures also.

A dense graph using 14 nodes for the function Kruskal_mst of Kruskal's algorithm:

```
# # Dense Graph with 14 nodes (more edges, closer to complete graph) with an adjacency matrix
dense_graph = [
    [None, 5, 9, 2, 4, 1, 6, 7, 10, 8, 12, 3, 2, 4],
    [5, None, 3, 6, 7, 9, 8, 10, 2, 11, 5, 4, 2, 5],
    [9, 3, None, 1, 5, 8, 9, 4, 6, 7, 3, 5, 7, 1],
    [2, 6, 1, None, 9, 7, 5, 3, 4, 6, 2, 8, 1, 7],
    [4, 7, 5, 9, None, 6, 4, 2, 3, 5, 9, 7, 3, 6],
    [1, 9, 8, 7, 6, None, 2, 5, 7, 8, 4, 6, 9, 1],
    [6, 8, 9, 5, 4, 2, None, 1, 3, 2, 9, 7, 8, 3],
    [7, 10, 4, 3, 2, 5, 1, None, 6, 4, 5, 3, 2, 9],
    [10, 2, 6, 4, 3, 7, 3, 6, None, 1, 8, 2, 4, 5],
    [8, 11, 7, 6, 5, 8, 2, 4, 1, None, 7, 6, 3, 2],
    [12, 5, 3, 2, 9, 4, 9, 5, 8, 7, None, 1, 6, 4],
    [3, 4, 5, 8, 7, 6, 7, 3, 2, 6, 1, None, 5, 8],
    [2, 2, 7, 1, 3, 9, 8, 2, 4, 3, 6, 5, None, 7],
    [4, 5, 1, 7, 6, 1, 3, 9, 5, 2, 4, 8, 7, None]
]
```

Fig14. Graph input for dense graph of Kruskal_mst(graph) algorithm

The implementation of a disjoint set data structure to track the sets and efficiently detect cycles during the construction of the minimum spanning tree. The 'DisjoiningSet' class initializes sets for each of the vertex and offers methods to find the root and perform unions based on ranks.

Union Operation: If the vertices are in different sets, we merge these sets into one. This is done by updating the root of one of the sets to point to the root of the other set, effectively combining the two trees into one tree.

The 'kruskal_mst' function receives a dense graph represented as an adjacency matrix and extracts all edges, excluding duplicates and edges with no weight information. The edges are sorted based on their weights in the ascending order. Kruskal's algorithm performs accurately in identifying the minimum spanning tree for the dense graph. Its ability to efficiently handle many edges while maintaining the correctness is evident.

```
"c:/Users/harle/OneDrive/Desktop/Algos/dense_graph(kruskal).py"
Total weight of minimum spanning tree: 18
Minimum spanning tree for Kruskal_mst(dense_graph): [(0, 5, 1), (2, 3, 1), (2, 13, 1), (3, 12, 1), (5, 13, 1),
(6, 7, 1), (8, 9, 1), (10, 11, 1), (1, 8, 2), (1, 12, 2), (3, 10, 2), (4, 7, 2), (5, 6, 2)]
```

Fig15. Output for Kruskal_mst algorithm for the dense graph.

The above is the mentioned figure that shows the output for dense graph of Kruskal_mst algorithm. The total weight of the resulting minimum spanning tree is calculated to be 18. The minimum spanning tree itself, consisting of edges connecting nodes to form the tree, is listed as follows: [(0, 5, 1), (2, 3, 1), (2, 13, 1), (3, 12, 1), (5, 13, 1), (6, 7, 1), (8, 9, 1), (10, 11, 1), (1, 8, 2), (1, 12, 2), (3, 10, 2), (4, 7, 2), (5, 6, 2)]

The Kruskal's algorithm successfully calculated the minimum spanning tree for the given dense graph. Its performance on dense graphs with multiple edges demonstrates its scalability and reliability in finding the optimal solutions.

A graph with varying edge weights, including negative weights for prims algorithm. if your implementation allows.

A Graph with negative edges has negative weights between two edges. Negative edges are common in real-world applications and can represent costs, distances, or other metrics where decrease is meaningful. "None" value indicates that there are no edges and negative values indicate that there are weights with negative edges.

```
#Graph with varying weights(including negative weights) with 14 nodes:
# Representation of a graph with varying weights, including negative weights:
# Each inner list represents the edges from a node to all other nodes.
# - None indicates no edge between the nodes.
# - Negative weights are allowed in this representation.
varying_weights_graph = [
    [None, 3, -2, None, None, None, None, None, None, None, None, None, None, None],
    [3, None, -6, None, None, None, None, None, None, None, None, None, None, None],
    [-2, -6, None, 2, None, None, None, None, None, None, None, None, None, None],
    [None, None, 2, None, -5, None, None, None, None, None, None, None, None, None],
    [None, None, None, -5, None, -5, None, None, 2, None, None, None, None, None],
    [None, None, None, None, -5, None, -9, None, None, None, None, None, None, None],
    [None, None, None, None, None, -9, None, -8, None, None, None, None, None, None],
    [None, None, None, None, None, None, -8, None, 1, None, None, None, None, None],
    [None, None, None, None, 2, None, None, 1, None, -4, None, None, None, None],
    [None, None, None, None, None, None, None, None, -4, None, -10, None, None, None],
    [None, None, None, None, None, None, None, None, None, -10, None, -12, None, None],
    [None, None, None, None, None, None, None, None, None, None, -12, None, -11, None],
    [None, None, None, None, None, None, None, None, None, None, None, -11, None, -13],
    [None, None, None, None, None, None, None, None, None, None, None, None, -13, None]
] # This adjacency matrix represents a graph with 14 nodes and varying edge weights,
# facilitating negative weights between certain nodes.
```

Fig16. Graph input for varying_weights_graph of prim_mst(graph) algorithm

The above-mentioned code is the input graph for varying_weights_graph of prim_mst(graph) algorithm contains an implementation of prim's algorithm to find the minimum spanning tree of a graph represented as an adjacency matrix. Additionally, it includes a graph with 'varying_weights_graph' with 14 nodes and varying edge weights, including negative weights.

At first, the prim_mst function uses to perform the prim's algorithm to calculate the minimum spanning tree for the "varying_weights_graph", and the edges are computed and stored in the 'minimum_spanning_tree' variable. The total weight of the minimum spanning tree is calculated by summing up the weights of the edges in minimum spanning tree 'total_weight'.

```
"c:/Users/harle/OneDrive/Desktop/Algos/varying_weights_graph(prim).py"
Total weight of minimum spanning tree: -82
Minimum spanning tree for prim_mst(varying_weights_graph): [(0, 2, -2), (2, 1, -6), (2, 3, 2), (3, 4, -5), (4,
5, -5), (5, 6, -9), (6, 7, -8), (7, 8, 1), (8, 9, -4), (9, 10, -10), (10, 11, -12), (11, 12, -11), (12, 13, -
13)]
```

Fig17. Output for prim_mst algorithm for the varying_weights_graph.

The above is the mentioned figure that shows the output for varying_weights_graph of prim_mst algorithm. The total weight of the resulting minimum spanning tree is calculated to be -82. The minimum spanning tree itself, consisting of edges connecting nodes to from the tree, is listed as follows: [(0, 2, -2), (2, 1, -6), (2, 3, 2), (3, 4, -5), (4, 5, -5), (5, 6, -9), (6, 7, -8), (7, 8, 1), (8, 9, -4), (9, 10, -10), (10, 11, -12), (11, 12, -11), (12, 13, -13)].

A graph with varying edge weights, including negative weights for Kruskal's algorithm. if your implementation allows.

```
#Graph with varying weights(including negative weights) with 14 nodes:
# Representation of a graph with varying weights, including negative weights:
# Each inner list represents the edges from a node to all other nodes.
# - None indicates no edge between the nodes.
# - Negative weights are allowed in this representation.
varying_weights_graph = [
    [None, 3, -2, None, None, None, None, None, None, None, None, None, None, None],
    [3, None, -6, None, None, None, None, None, None, None, None, None, None, None],
    [-2, -6, None, 2, None, None, None, None, None, None, None, None, None, None],
    [None, None, 2, None, -5, None, None, None, None, None, None, None, None, None],
    [None, None, None, -5, None, -5, None, None, 2, None, None, None, None, None],
    [None, None, None, None, -5, None, -9, None, None, None, None, None, None, None],
    [None, None, None, None, None, -9, None, -8, None, None, None, None, None, None],
    [None, None, None, None, None, None, -8, None, 1, None, None, None, None, None],
    [None, None, None, None, 2, None, None, 1, None, -4, None, None, None, None],
    [None, None, None, None, None, None, None, None, -4, None, -10, None, None, None],
    [None, None, None, None, None, None, None, None, None, None, -10, None, -12, None],
    [None, None, None, None, None, None, None, None, None, None, None, -12, None, -11],
    [None, None, None, None, None, None, None, None, None, None, None, None, -11, -13],
    [None, None, None, None, None, None, None, None, None, None, None, None, 13, None]
] # This adjacency matrix represents a graph with 14 nodes and varying edge weights,
# facilitating negative weights between certain nodes.
```

Fig18. Graph input for varying_weights_graph of Kruskal_mst(graph) algorithm

The above mentioned is the graph input for displaying the varying_weights_graph for Kruskal's algorithm. This Kruskal's algorithm works by sorting all the edges in ascending order if weight and then iterates through them. This adds each edge of the minimum spanning tree if it doesn't create a cycle. It does this by maintaining od disjoint sets.

The graph includes both positive and negative edge weights, which is acceptable in Kruskal's algorithm as well. The algorithm can handle negative edge weights if there are no negative cycles.

```
"c:/Users/harle/OneDrive/Desktop/Algos/varying_weights_graph(kruskal).py"
Total weight of minimum spanning tree: -82
Minimum spanning tree for Kruskal_mst(varying_weights_graph): [(12, 13, -13), (10, 11, -12), (11, 12, -11), (9, 10, -10), (5, 6, -9), (6, 7, -8), (1, 2, -6), (3, 4, -5), (4, 5, -5), (8, 9, -4), (0, 2, -2), (7, 8, 1), (2, 3, 2)]
```

Fig19. Output for Kruskal_mst algorithm for the varying_weights_graph.

The above is the mentioned figure that shows the output for varying_weights_graph of Kruskal_mst algorithm. The total weight of the resulting minimum spanning tree is calculated to be -82. The minimum spanning tree itself, consisting of edges connecting nodes to form the tree, is listed as follows: [(12, 13, -13), (10, 11, -12), (11, 12, -11), (9, 10, -10), (5, 6, -9), (6, 7, -8), (1, 2, -6), (3, 4, -5), (4, 5, -5), (8, 9, -4), (0, 2, -2), (7, 8, 1), (2, 3, 2)].

The above obtained results for calculating the minimum spanning tree using Kruskal's algorithm on a graph with varying edge weights, including the negative weights.

The performance between the Prim's vs Kruskal's for Sparse graph:

Prim's Algorithm:

Prim's method builds a minimum spanning tree starting from a single vertex, extending it by adding the shortest edge that connects a new vertex to the tree. It consistently chooses the edge with the least weight that reaches an unconnected vertex.

Prim's algorithm typically employs a priority queue to effectively pick the smallest edge at each iteration. Additionally, it keeps an array to keep track of the least possible edge weight for each vertex.

Kruskal's Algorithm:

Kruskal's technique involves arranging all the edges by their weight and sequentially incorporating them into the minimum spanning tree from the smallest to the largest, ensuring that no cycles are formed. It uses a union structure to quickly determine whether adding an edge would create a cycle.

Performance of Prim's:

Prim's is ideal for sparse graphs where the edges are significantly fewer than the maximum number of possible edges. ($|E| \ll |V|^2$). The efficiency of Prim's often comes from the priority queue's ability to quickly select the smallest edge.

Performance of Kruskal's:

Kruskal's is also suitable for sparse graphs, and it is often the algorithm of choice when the graph's edges are presented as a list. Sorting the edges may require significant computation, but the union mechanism's quick cycle checking aids the overall efficiency of the algorithm.

In summary, both the Prim's and Kruskal's algorithms can perform well in sparse graphs. The choice between them may depend on the specific characteristics of the graph, the representation used, and the underlying data structures available for implementation.

The Performance between the Prim's vs Kruskal's for Dense graph:

Prim's Algorithm:

Prim's algorithm is generally more efficient for graphs with many edges because it incrementally builds the minimum spanning tree by adding the smallest edge at each step. The algorithm's efficiency can be enhanced with a priority queue, which can lower its time complexity to $O(E + V \log V)$. This is particularly beneficial in a graph where edges are dense since Prim's algorithm focuses on vertices, which are typically less numerous than edges in a dense graph.

Kruskal's Algorithm:

For graphs that have a large number of edges, Kruskal's algorithm might be less effective due to the necessity of sorting all the edges, which becomes substantially burdensome as the edge count rises. It has a time complexity of $O(E \log E)$, and because the number of edges (E) approaches the square of the number of vertices ($|V|^2$) in dense graphs, the sorting step becomes the expensive and time-consuming.

The Performance between the Prim's vs Kruskal's for Negative edges graph:

Prim's Algorithm:

Prim's method is also effective with negative weights, as it simply requires selecting the edge with the smallest weight connecting the growing tree to the remaining vertices. The algorithm's efficiency remains intact because it is based on the comparison of edge weights, not their absolute magnitudes.

Kruskal's Algorithm:

Kruskal's method of implementation operates efficiently with any edge weights, including negative ones, since it organizes edges by weight before incrementally building the minimum spanning tree. Its functionality is unaffected by negative weights as it does not depend on the positivity of edge weights.

Part 3: Analysis

- Analyze the performance of both algorithms on each graph.
- Discuss the time complexity in the best, average, and worst-case scenarios for each algorithm.

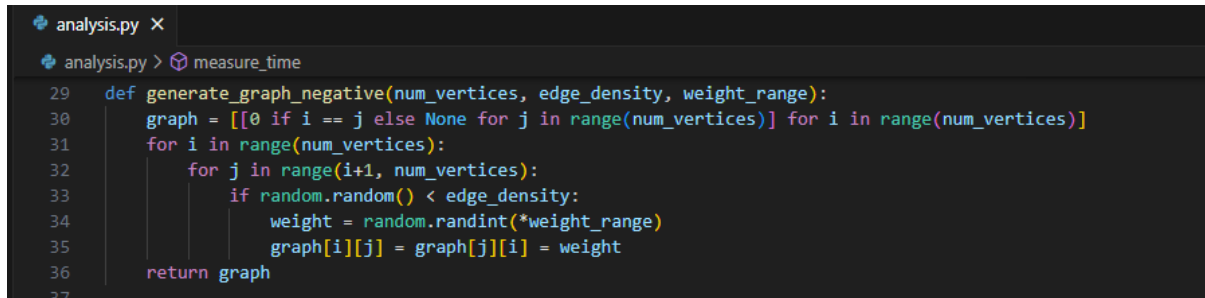
Deliverable 3: The third part requires a detailed analytical report. This report should include a thorough performance analysis comparing Prim's and Kruskal's algorithms, focusing on time complexity in various scenarios (e.g., best, average, and worst-case). Additionally, the report should feature a real-world application case study, demonstrating the practical use of MSTs and arguing which of the two algorithms would be more suitable for the specific scenario. The report should conclude with a comparative discussion of the two algorithms, drawing insights from both the theoretical aspects and the testing results.

Solution:

Testing Results

Let's compare the performance of prim's and Kruskal's algorithm using plot:

At first let's generate with some random values as weights and which also takes that density as input.



```
analysis.py X
analysis.py > measure_time

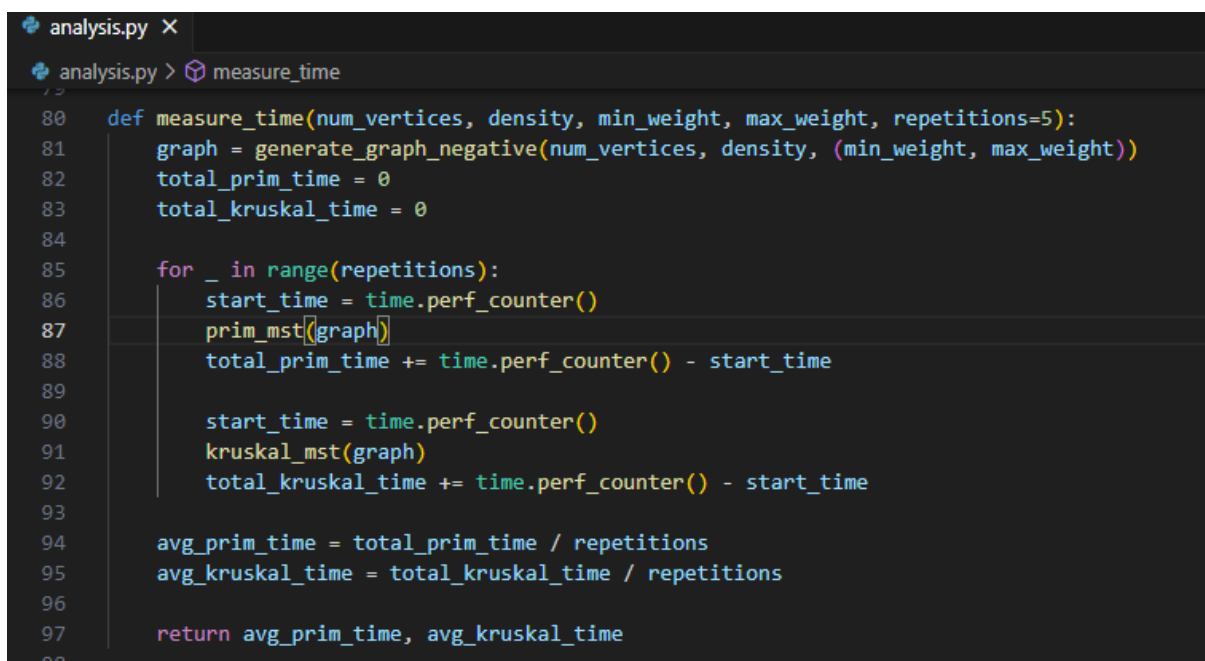
29 def generate_graph_negative(num_vertices, edge_density, weight_range):
30     graph = [[0 if i == j else None for j in range(num_vertices)] for i in range(num_vertices)]
31     for i in range(num_vertices):
32         for j in range(i+1, num_vertices):
33             if random.random() < edge_density:
34                 weight = random.randint(*weight_range)
35                 graph[i][j] = graph[j][i] = weight
36     return graph
37
```

Fig20. Generating random values

This takes the range for weights of edges, number of vertices and edge density as parameter and builds adjacency graph.

This edge density having a range of [0-1] and this is main and primary to generate the sparse and dense graphs. If the density is close to 1, dense graph will be generated and if the density is close to 0 sparse graph will be generated.

Now, let's calculate time taken to perform minimum spanning tree for both the prim's and Kruskal's:



```
analysis.py X
analysis.py > measure_time

80 def measure_time(num_vertices, density, min_weight, max_weight, repetitions=5):
81     graph = generate_graph_negative(num_vertices, density, (min_weight, max_weight))
82     total_prim_time = 0
83     total_kruskal_time = 0
84
85     for _ in range(repetitions):
86         start_time = time.perf_counter()
87         prim_mst(graph)
88         total_prim_time += time.perf_counter() - start_time
89
90         start_time = time.perf_counter()
91         kruskal_mst(graph)
92         total_kruskal_time += time.perf_counter() - start_time
93
94     avg_prim_time = total_prim_time / repetitions
95     avg_kruskal_time = total_kruskal_time / repetitions
96
97     return avg_prim_time, avg_kruskal_time
98
```

Fig21. Performing the calculation for time taken to perform MST.

Now, let's the function calculates the running time for both the prim's and Kruskal's algorithm and let's implement to plot the graph for both the algorithms.

```

analysis.py x
analysis.py > measure_time
99 def running_time_comparison1(density, min_weight, max_weight, repetitions=5):
100     graph_sizes = [i for i in range(10, 500, 20)] # Increase graph sizes for better visibility of time differences
101     prims_times = []
102     kruskals_times = []
103
104     for size in graph_sizes:
105         running_time = measure_time(size, density, min_weight, max_weight, repetitions)
106         prims_times.append(running_time[0])
107         kruskals_times.append(running_time[1])
108         print(f"Graph Size: {size}, Prim's Avg Time: {running_time[0]}, Kruskal's Avg Time:{running_time[1]}")
109
110     plt.figure(figsize=(10, 6))
111     plt.plot(graph_sizes, prims_times, label='Prim\'s Algorithm', marker='o')
112     plt.plot(graph_sizes, kruskals_times, label='Kruskal\'s Algorithm', marker='x')
113     plt.xlabel('Graph Size (Number of Vertices)')
114     plt.ylabel('Average Running Time (Seconds)')
115     plt.title('Comparison of Average Running Times: Prim\'s vs Kruskal\'s Algorithm')
116     plt.legend()
117     plt.show()

```

Fig22. To plot the graph.

This takes the running times of both the prims and Kruskal's algorithm on y-axis and generates a graph with number of vertices on x-axis.

Now let's compare plot for different graphs of types of matrix:

Sparse graph:

In sparse graphs the number of edges is less. Hence, we provide the edge density value as 0.1.

```

119 running_time_comparison1(0.9,-10,15)

```

Fig23. Running time comparison

Now, calling the above build function with the density value of 0.1 for the sparse graph and minimum weight is 1 and the maximum weight is 15. So, this generates the graph with weight values in range of [1-15]

Plot Result:

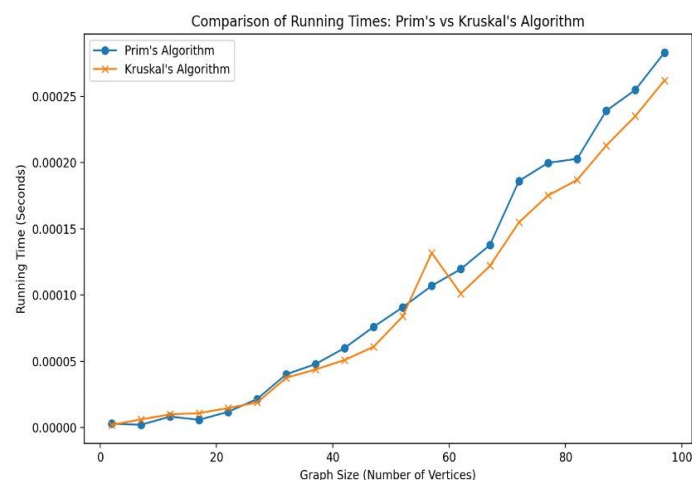


Fig24. Comparison of running time Prim's vs Kruskal's.

Observations for the Sparse graph:

Prim's vs Kruskal's:

In sparse graphs, which have relatively few edges compared to the maximum possible, both prim's and Kruskal's algorithms shows comparable performance graphs. However, Kruskal's tends to have a slight edge in efficiency.

- **Prim's Algorithm:**
For sparse graphs, prim's algorithm is more efficient when using an adjacency list representation, with the time complexity being $O((V+E) \log V)$, which is favourable when there are fewer edges.
- **Kruskal's Algorithm:**
Kruskal's Algorithm generally shows better efficient in sparse graphs due to its sorting-based nature.

Dense Graph:

In dense graph, the number of edges is more. Hence, we give the edge density values as 0.9.

Calling the above build function with the density of 0.9 for the dense graph and the minimum weight is 1 and the maximum weight is 15. So, this generates the graph with weight values in range of [1-15]

Plot for Dense Graph: Number of vertices range from [2-100]

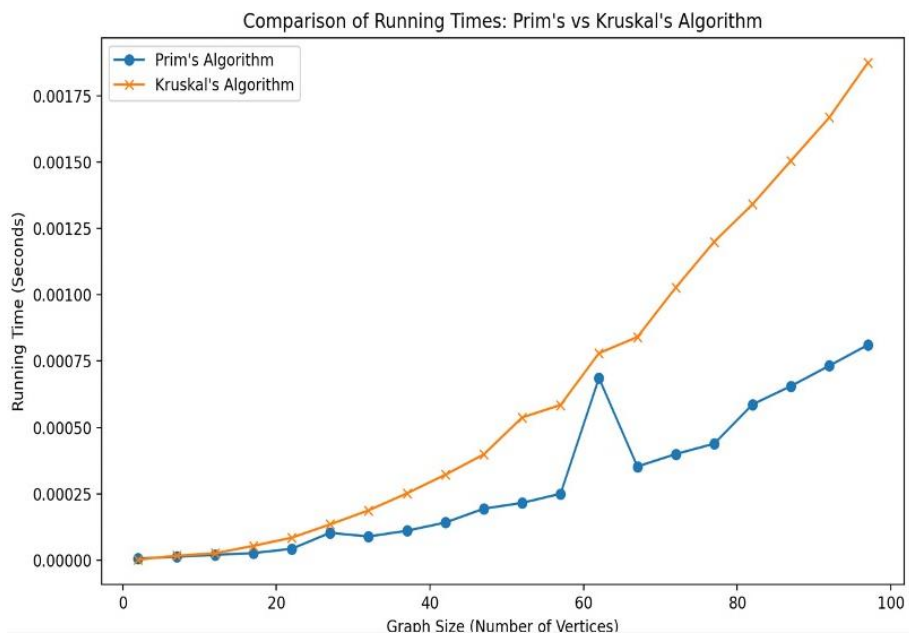


Fig25. Comparison running times: Prim's vs Kruskal's algorithm.

The above graph shows the comparison between prim's and Kruskal's algorithm on the dense graphs.

Dense graphs analysis: prim's vs Kruskal's algorithm

In dense graphs analysis, it's clearly shows that the prim's algorithm operates more efficiently than Kruskal's, particularly as the number of vertices increases.

Prim's algorithm:

It's particularly effective for dense graphs and it incrementally builds a Minimum Spanning Tree (MST), starting from a single vertex and continuously attaching the least expensive edge to a new vertex.

Kruskal's Algorithm:

This algorithm is generally less optimal for dense graphs and the efficiency decreases in dense graphs due to the overhead of managing a multitude of edges.

Performance Cases for Prim's and Kruskal's Algorithms:

Best Case:

Prim's Algorithm: If the graph is already connected, the complexity is $O(V^2)$ for a matrix, or $O(E + V \log V)$ for a list.

Kruskal's Algorithm: With a good sorting algorithm, the complexity can be as good as $O(E \log E)$.

Average Case:

Prim's Algorithm: Usually runs in $O(V^2)$ for a matrix and $O((V + E) \log V)$ for a list.

Kruskal's Algorithm: Typically, the time complexity remains at $O(E \log E)$ when using a robust sorting method.

Worst Case:

Prim's Algorithm: In dense graphs, it can go up to $O(V^2)$ for a matrix and $O((V + E) \log V)$ for a list.

Kruskal's Algorithm: The time complexity can reach $O(E \log E)$ in the worst-case scenario, assuming efficient sorting.

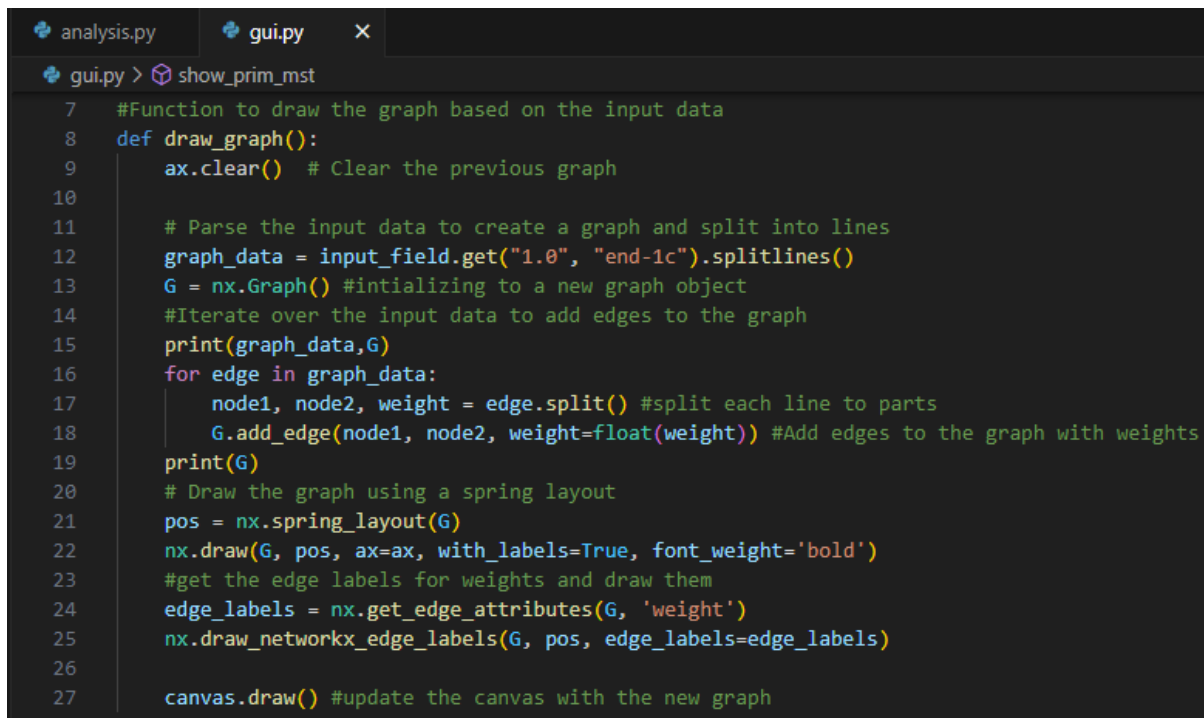
Part 4: Extra Credit (Optional)

Implement a graphical user interface (GUI) where users can draw a graph and see the MST found by each algorithm.

Solution:

Implementing the graphical user interface where users can draw the graph by providing the weights and see the two different mst's found by each algorithm.

The 'draw_graph' function visualizes a weighted graph from user input. It clears the plot, and passes the input for edges and weights, creates a graph, and plots it with labels for nodes and edge weights, updating the display accordingly.



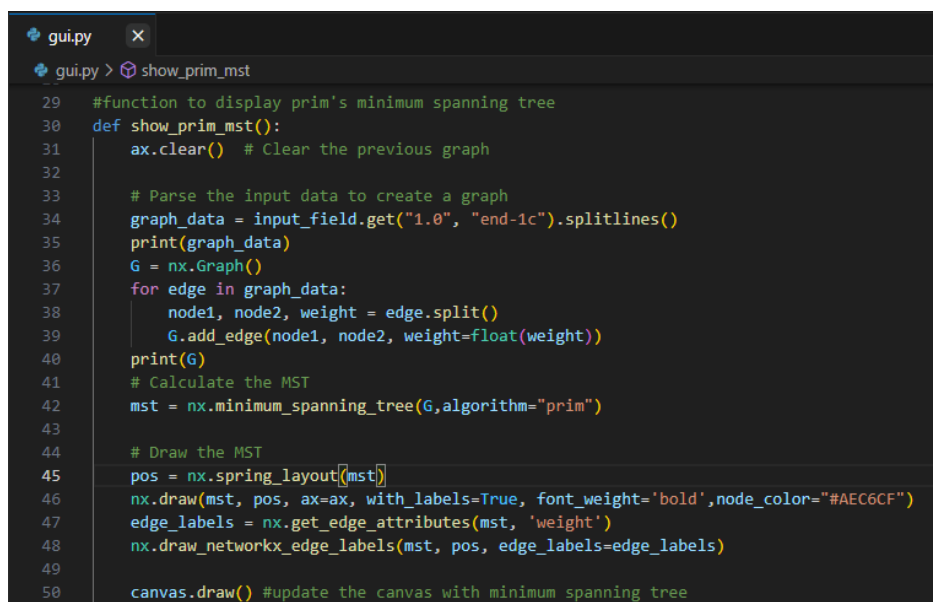
```

analysis.py  gui.py  X
gui.py > show_prim_mst
7  #Function to draw the graph based on the input data
8  def draw_graph():
9      ax.clear() # Clear the previous graph
10
11      # Parse the input data to create a graph and split into lines
12      graph_data = input_field.get("1.0", "end-1c").splitlines()
13      G = nx.Graph() #initializing to a new graph object
14      #Iterate over the input data to add edges to the graph
15      print(graph_data,G)
16      for edge in graph_data:
17          node1, node2, weight = edge.split() #split each line to parts
18          G.add_edge(node1, node2, weight=float(weight)) #Add edges to the graph with weights
19      print(G)
20      # Draw the graph using a spring layout
21      pos = nx.spring_layout(G)
22      nx.draw(G, pos, ax=ax, with_labels=True, font_weight='bold')
23      #get the edge labels for weights and draw them
24      edge_labels = nx.get_edge_attributes(G, 'weight')
25      nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
26
27      canvas.draw() #update the canvas with the new graph

```

Fig26.draw_graph ():

The 'show_prim_mst' function is designed to calculate and shows the Prim's minimum spanning tree from a graph that has been previously input by the user. It clears the current graph display, calculate the minimum spanning tree using prim's algorithm via the NetworkX library, and then draws this minimum spanning tree on the canvas. The nodes are displayed with a specific colour and the edges are labelled with their corresponding weights. And then must update the canvas to show the minimum spanning tree with the layout and labels.



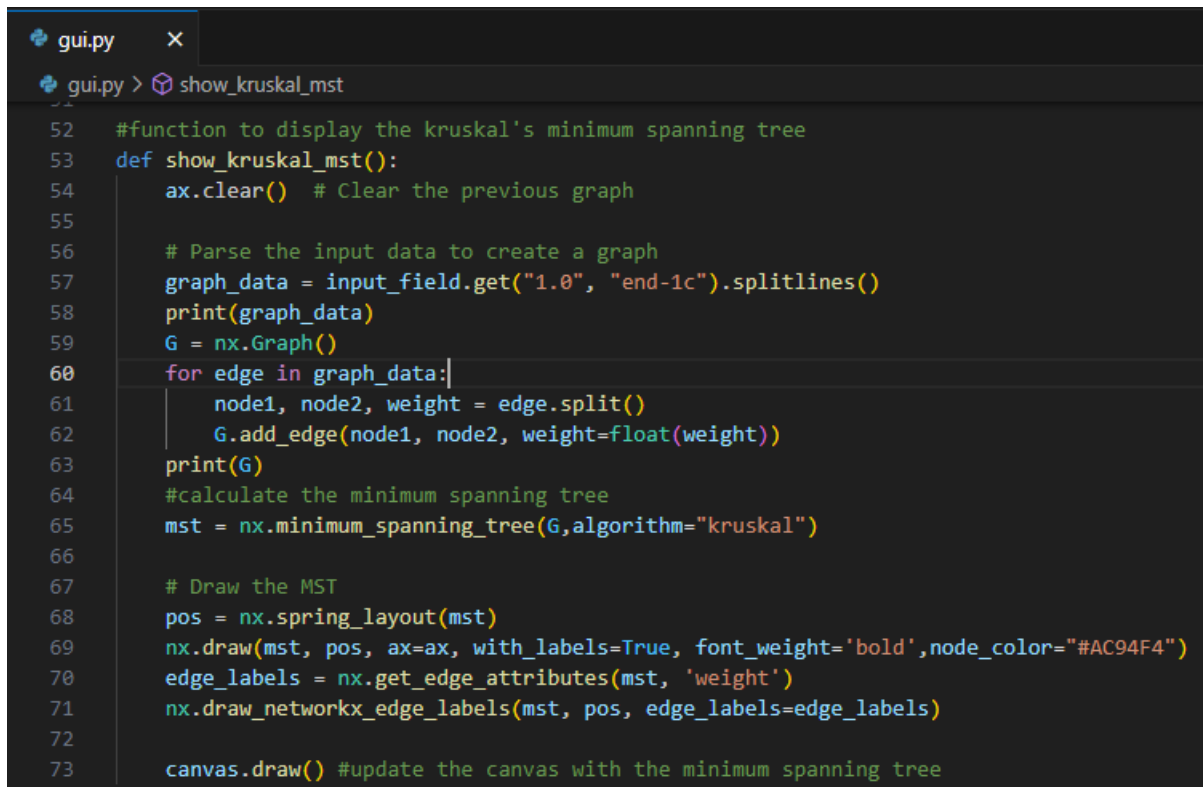
```

gui.py  X
gui.py > show_prim_mst
29  #function to display prim's minimum spanning tree
30  def show_prim_mst():
31      ax.clear() # Clear the previous graph
32
33      # Parse the input data to create a graph
34      graph_data = input_field.get("1.0", "end-1c").splitlines()
35      print(graph_data)
36      G = nx.Graph()
37      for edge in graph_data:
38          node1, node2, weight = edge.split()
39          G.add_edge(node1, node2, weight=float(weight))
40      print(G)
41      # Calculate the MST
42      mst = nx.minimum_spanning_tree(G,algorithm="prim")
43
44      # Draw the MST
45      pos = nx.spring_layout(mst)
46      nx.draw(mst, pos, ax=ax, with_labels=True, font_weight='bold',node_color="#AEC6CF")
47      edge_labels = nx.get_edge_attributes(mst, 'weight')
48      nx.draw_networkx_edge_labels(mst, pos, edge_labels=edge_labels)
49
50      canvas.draw() #update the canvas with minimum spanning tree

```

Fig27.show_prim_mst ():

The 'Show_kruskal_mst' is function calculates and display's the Kruskal's minimum spanning tree for a graph provided by the user. It starts by clearing any graph currently displayed, then calculates the minimum spanning tree using Kruskal's algorithm, which is a part of the NetworkX library. After calculating the minimum spanning tree, it uses a spring layout to position the nodes, draws the minimum spanning tree with a unique node colour, and annotates the edges with their weights. Finally, the canvas is refreshed to show the updated visualization with the minimum spanning tree.



```

gui.py x
gui.py > show_kruskal_mst

52 #function to display the kruskal's minimum spanning tree
53 def show_kruskal_mst():
54     ax.clear() # Clear the previous graph
55
56     # Parse the input data to create a graph
57     graph_data = input_field.get("1.0", "end-1c").splitlines()
58     print(graph_data)
59     G = nx.Graph()
60     for edge in graph_data:
61         node1, node2, weight = edge.split()
62         G.add_edge(node1, node2, weight=float(weight))
63     print(G)
64     #calculate the minimum spanning tree
65     mst = nx.minimum_spanning_tree(G,algorithm="kruskal")
66
67     # Draw the MST
68     pos = nx.spring_layout(mst)
69     nx.draw(mst, pos, ax=ax, with_labels=True, font_weight='bold',node_color="#AC94F4")
70     edge_labels = nx.get_edge_attributes(mst, 'weight')
71     nx.draw_networkx_edge_labels(mst, pos, edge_labels=edge_labels)
72
73     canvas.draw() #update the canvas with the minimum spanning tree

```

Fig28. Show_kruskal_mst ():

The below code is sets up the main window for a Tkinter application designed to visualize weighted graphs and their minimum spanning tree. It initializes the main application window with a title, and then adds a text input widget where users can enter graph data. There are two buttons: one to draw the graph based on the input data and the other two to compute and display the minimum spanning tree using either prim's or Kruskal's algorithm. It also sets up a Matplotlib figure within the Tkinter canvas, which is where the graphs and minimum spanning tree will be showed. Finally, it starts the Tkinter event loop, which keeps the application running and responsive to user actions like button clicks.

```

gui.py x
gui.py > show_kruskal_mst
75 # Set up the main window
76 root = tk.Tk()
77 root.title("Weighted Graph and MST Visualization")
78
79 # Create input field for graph data
80 input_field = tk.Text(root, height=10)
81 input_field.pack()
82
83 # Button to draw graph
84 draw_button = tk.Button(root, text="Draw Graph", command=draw_graph)
85 draw_button.pack()
86
87 # Button to show Prims MST
88 show_mst_button = tk.Button(root, text="Show Prims MST", command=show_prim_mst)
89 show_mst_button.pack()
90
91 # Button to show Kruskals MST
92 show_mst_button = tk.Button(root, text="Show Kruskals MST", command=show_kruskal_mst)
93 show_mst_button.pack()
94
95 # Set up the matplotlib figure and axes
96 fig, ax = plt.subplots()
97
98 # Embed the matplotlib figure in a Tkinter canvas
99 canvas = FigureCanvasTkAgg(fig, master=root)
100 canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=1)
101
102 # Start the Tkinter event loop
103 root.mainloop()

```

Fig29. Plotting for the Tkinter Application

Results:

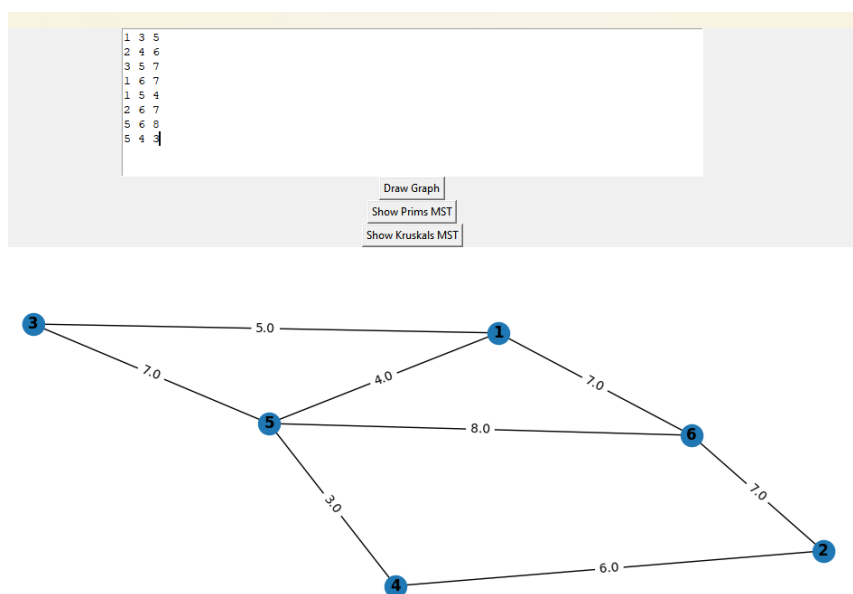


Fig30. Draw graph.

For the above draw graph represents all the nodes that which we have entered and if any node is repeating again, it will take the latest node that represents in the input data provided at the top of the application.

In the above draw graph, we gave total of 8 nodes those are [(1,3,5) (2,4,6) (3,5,7) (1,6,7) (1,5,4) (2,6,7) (5,6,8) (5,4,3)].

Prims mst:

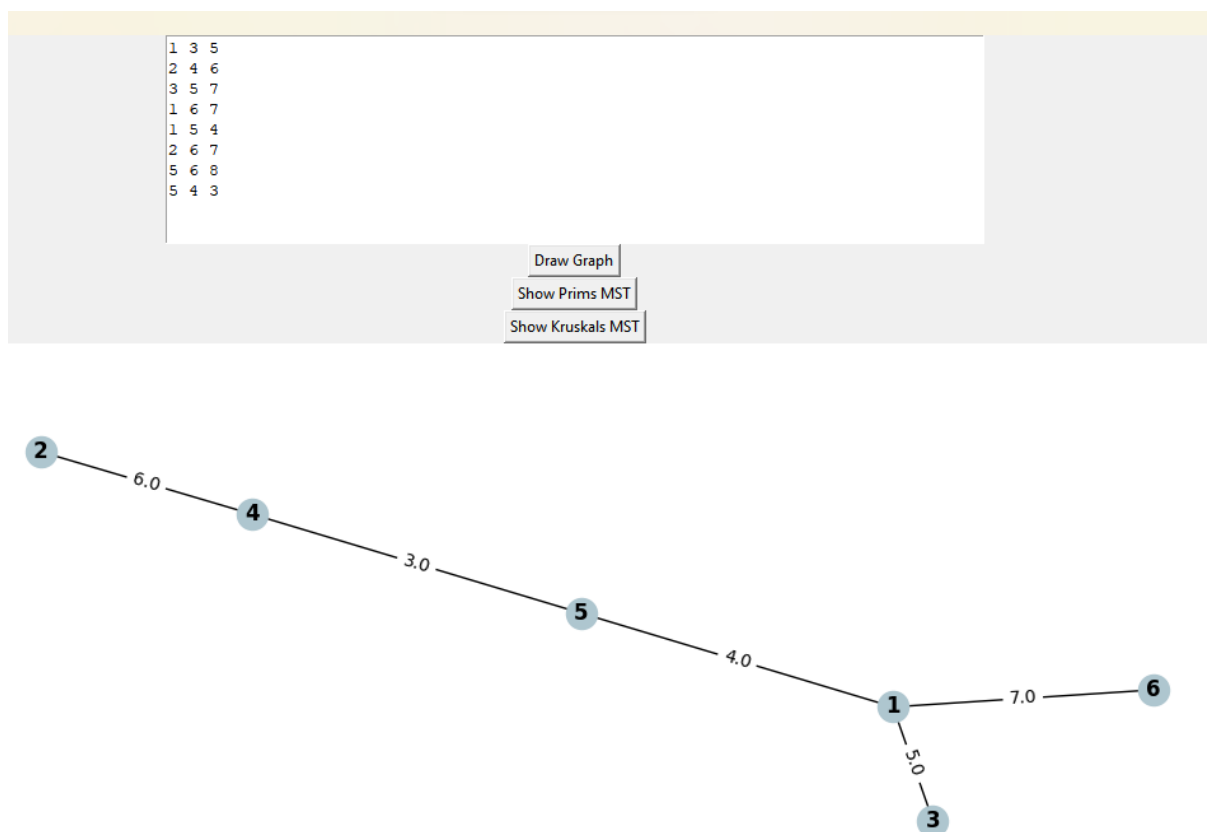


Fig31.Prims_MST

This is the minimum spanning tree generated using prim's algorithm. The edges selected are those that connect all the vertices with the least total weight without forming any cycles. The total weight of prims minimum spanning tree is calculated by summing the weights of all the included edges. This weight is the minimum that can be achieved while connecting all the vertices. The above connected vertices and weights are [(1,3,5) (1,6,7) (1,5,4) (5,4,3) (2,4,6)].

Kruskal's mst:

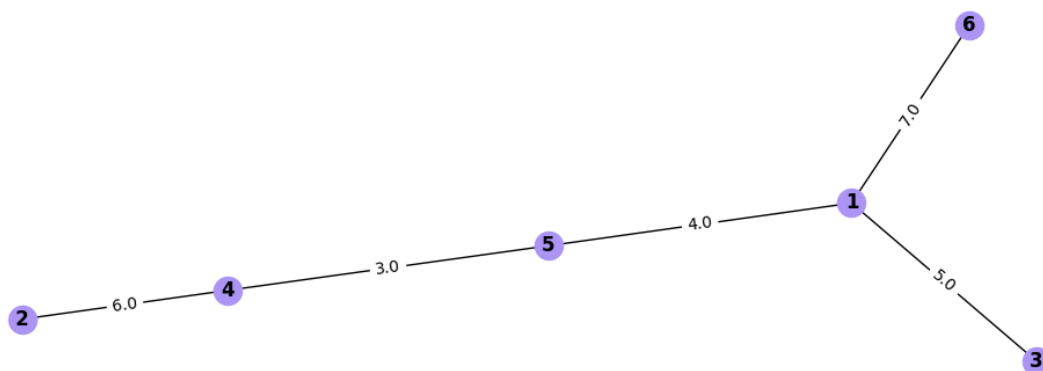
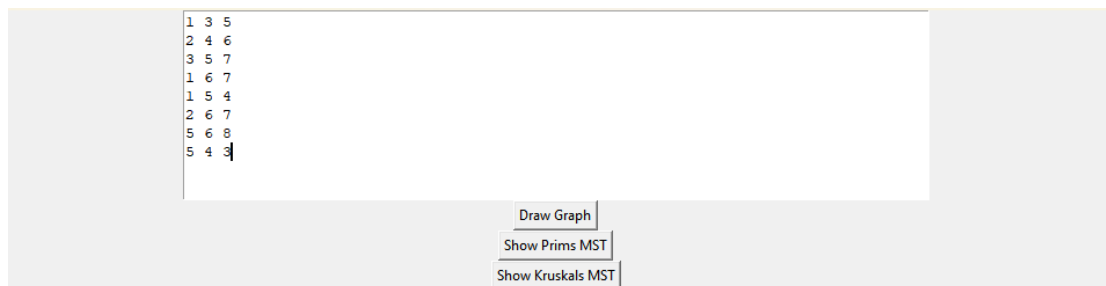


Fig31. Kruskal's mst

From the above result of Kruskal's mst, like prim's mst this tree also connects all the vertices with the minimum total weight. However, it is constructed using Kruskal's algorithm, which selects the shortest edges first, ensuring no cycles formed. The total weight of Kruskal's minimum spanning tree is also the minimum possible for connecting all the vertices like prim's but achieved through a different selection process. In the above the connection of vertices are as follows: [(1,6,7) (1,3,5) (1,5,4) (5,4,3) (2,4,6)].

Both the prim's and Kruskal's Minimum spanning tree often yield the same result, especially in undirected graphs where the edge weights are distinct. However, the methods of constructing these tree differ, as illustrated by the two different approaches in these visualizations. Despite the similar outcomes, both the algorithms differ in their approach. Prim's algorithm builds the minimum spanning tree incrementally, while the Kruskal's algorithm sorts all the edges first and then selects the shortest edges.