Project Two

Problem 1: Single-source Shortest Path Algorithm

Problem 1 Background

The objective of this portion of the project is to implement an algorithm that will find the shortest path from a source node to all other nodes in a weighted graph. The graph can be undirected or directed and have no negative edges. For this objective I implement Dijkstra's algorithm, a greedy algorithm.

Problem 1: Pseudocode

The pseudocode for this algorithm is below (figure 1):

```
Dijkstra Algorithm (graph, source):
    Distance[source] = 0
    Predecessors[source] = inf
    priorityQueue.push((0, source))
    while priorityQueue is not empty:
        (currentDistance, CurrentNode) = priorityQueue.pop()
        If currentNode is visited:
            Continue
        For each neighbor, weight in graph[currentNode]:
            distanceFromSource = currentDistance + weight
            if distanceFromSource < distance[neighbor]:
                distance[neighbor] = distanceFromSource
                predecessors[neighbor] = currentNode
                priorityQueue.push((distanceFromSource, neighbor))
        mark currentNode as visited
    return distance, predecessors
```

To begin, we create a distance array that stores the shortest distance from the source node and set the distance to source node to zero and all other nodes to infinity. The we create a visited array to keep track of nodes that have been visited and a priority queue to select the node with the smallest distance. Next, we add the source node to the priority queue with a distance of zero. The main loop will consist of removing the node with the smallest distance from the priority queue and if the current node has already been visited we skip it and continue to the next node, while the priority queue is not empty. The next step involves something called relaxation, for each neighbor of the current node we calculate the distance from the source to the neighbor and if this distance is smaller than the current stored distance then we update it with the new distance and update the priority queue with the new distance for the neighbor. We then mark the node as visited and repeat the loop. Once all nodes have been visited the distance array should contain the shortest distances from the source node to all other nodes.

Problem 1 Data Structures

The data structure I used to implement the algorithm is a priority queue from the heapq library. To store the graph, I used an adjacency list that I implemented using dictionaries. For the remaining supporting data, I used lists to store everything.

Problem 1 Analysis of Runtime

Since I used an adjacency list the time complexity for Dijkstra's Algorithm is O((V+E) log V), where V is the number of vertices and E is the number of edges. This time complexity is due to the fact that in the worst case each node is visited once and for each of those nodes all of its outgoing edges in the priority queue are considered. Each insertion and extraction of the priority queue takes log V time. This may happen E times and additionally there may be up to V operations to update distance in the priority queue.

Problem 1 Results

Below are visual representations of the graphs used and the results that I achieved with my implementation of Dijkstra's Algorithm:

The first graph use is undirected and weighted with 9 nodes and 12 edges:

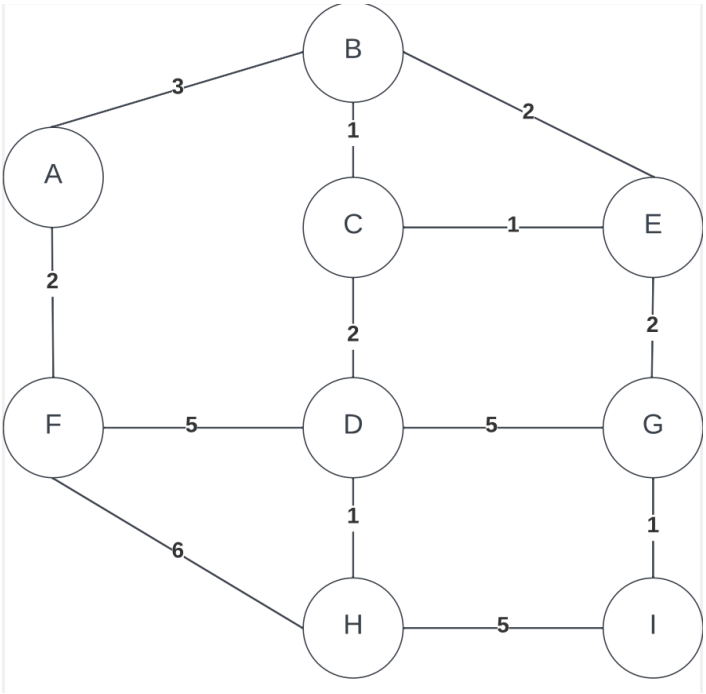

Figure 2. Graph 1: Undirected and weighted.

```
Shortest path to A: ['A'], distance: 0
Shortest path to B: ['A', 'B'], distance: 3
Shortest path to F: ['A', 'F'], distance: 2
Shortest path to C: ['A', 'B', 'C'], distance: 4
Shortest path to E: ['A', 'B', 'E'], distance: 5
Shortest path to D: ['A', 'B', 'C', 'D'], distance: 6
Shortest path to G: ['A', 'B', 'E', 'G'], distance: 7
Shortest path to H: ['A', 'B', 'C', 'D', 'H'], distance: 7
Shortest path to I: ['A', 'B', 'E', 'G', 'I'], distance: 8
```

Figure 3. Graph 1 SSSP results

The second graph used is undirected and weighted with 9 nodes and 12 edges:

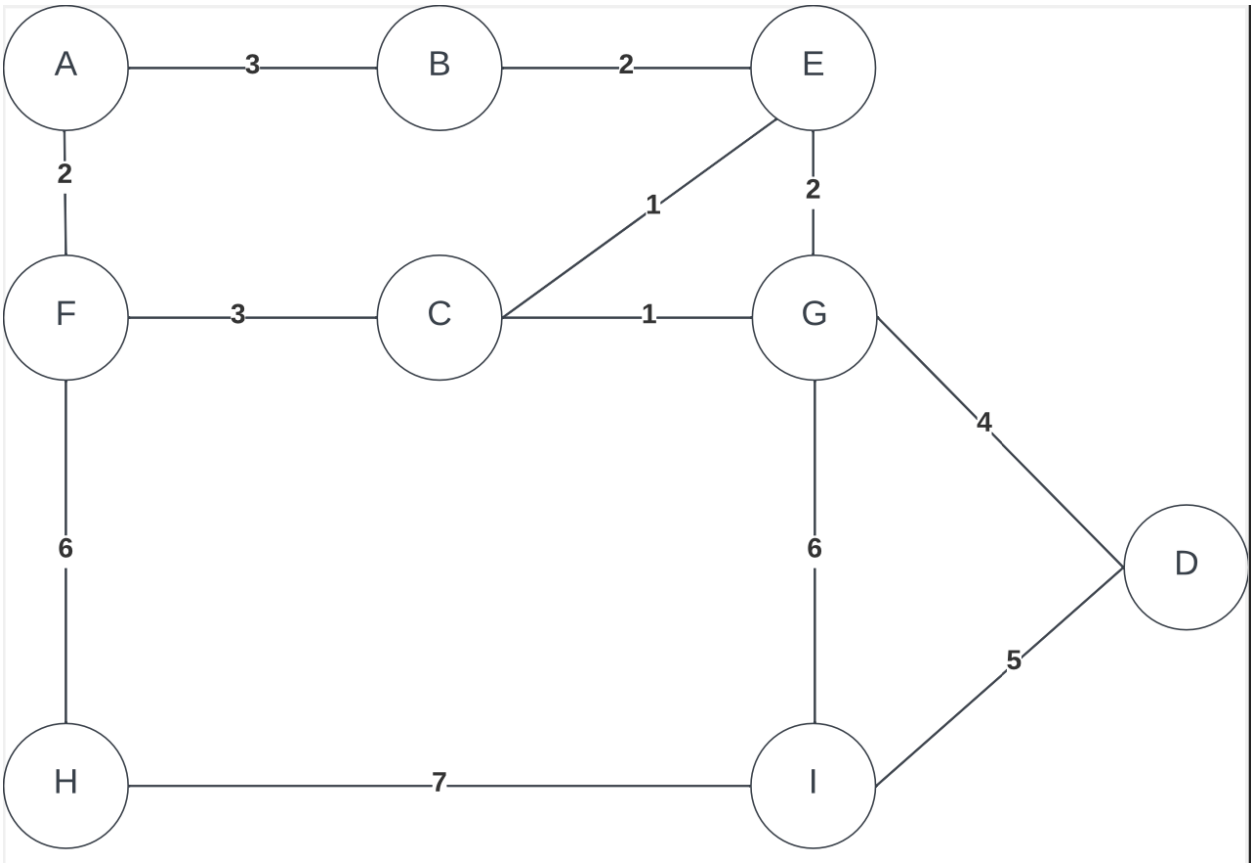

Figure 4. Graph 2: Undirected and weight.

```
Shortest path to A: ['A'], distance: 0
Shortest path to B: ['A', 'B'], distance: 3
Shortest path to F: ['A', 'F'], distance: 2
Shortest path to E: ['A', 'B', 'E'], distance: 5
Shortest path to C: ['C'], distance: inf
Shortest path to G: ['A', 'B', 'E', 'G'], distance: 7
Shortest path to D: ['D'], distance: inf
Shortest path to I: ['A', 'B', 'E', 'G', 'I'], distance: 13
Shortest path to H: ['A', 'F', 'H'], distance: 8
```

Figure 5. Graph 2 SSSP results.

The third graph used is directed and weighted with 9 nodes and 15 edges:

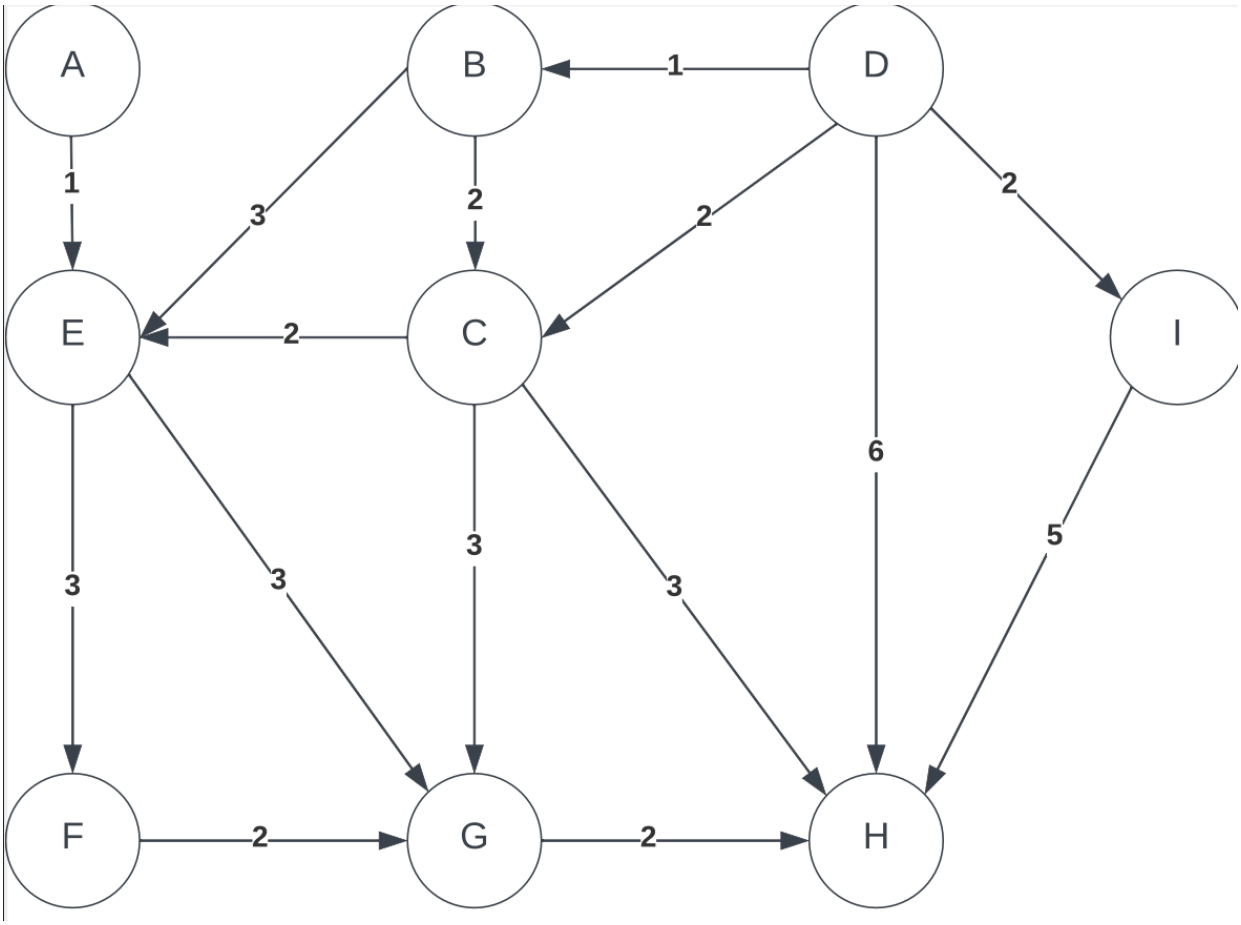

Figure 6: Graph 3: Directed and weighted

```
Shortest path to A: ['A'], distance: 0
Shortest path to E: ['A', 'E'], distance: 1
Shortest path to B: ['B'], distance: Unreachable
Shortest path to C: ['C'], distance: Unreachable
Shortest path to G: ['A', 'E', 'G'], distance: 4
Shortest path to H: ['A', 'E', 'G', 'H'], distance: 6
Shortest path to D: ['D'], distance: Unreachable
Shortest path to I: ['I'], distance: Unreachable
Shortest path to F: ['A', 'E', 'F'], distance: 4
```

Figure 7. Graph 3 SSSP results

The final graph used is directed and weighted with 9 nodes and 12 edges:

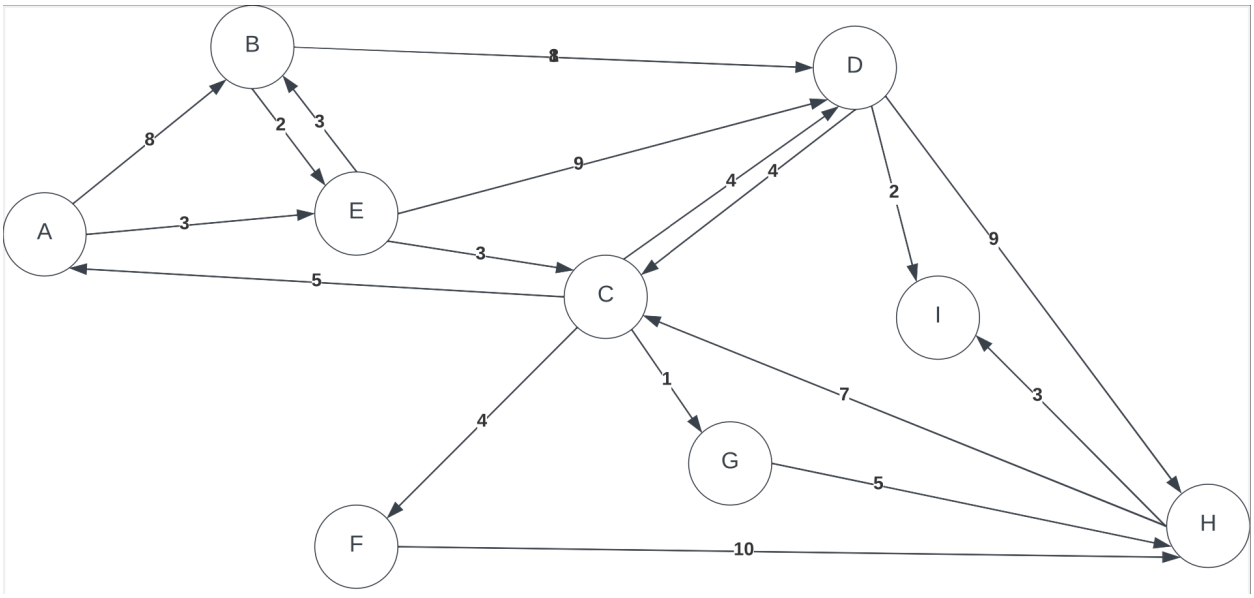

Figure 8. Graph 4: Directed and weighted.

```
Shortest path to A: ['A'], distance: 0
Shortest path to B: ['A', 'E', 'B'], distance: 5
Shortest path to E: ['A', 'E'], distance: 3
Shortest path to D: ['A', 'E', 'C', 'D'], distance: 10
Shortest path to C: ['A', 'E', 'C'], distance: 6
Shortest path to G: ['A', 'E', 'C', 'G'], distance: 7
Shortest path to F: ['A', 'E', 'C', 'F'], distance: 10
Shortest path to I: ['A', 'E', 'C', 'D', 'I'], distance: 12
Shortest path to H: ['A', 'E', 'C', 'G', 'H'], distance: 12
```

Figure 9. Graph 4 SSSP results

Problem 2: Minimum Spanning Tree

Problem 2 Background

The objective of problem 2 is to implement an algorithm that will find the minimum spanning tree given a connected, undirected, weighted graph. The algorithm I chose to implement is Kruskal's algorithm. See below for the pseudocode:

```
Kruskal's Algorithm (graph):
Edges = []

For vertex in graph.keys():
        For neighbor, weight in graph[vertex].items():
```

```
edges.append((vertex, neighbor, weight))

edges = SortEdges(edges)

parent  = {}
rank = {}

for vertex in graph.keys():
        parent[vertex] = vertex
        rank[vertex] = 0

result = []

for edge in edges:
        src, dest, weight = edge
        src_parent = Find(parent, src)
        dest_parent = Find(parent, dest)

        if src_parent != dest_parent:
                result.append(edge) # Add the edge to the MST
                Union(parent, rank, src_parent, dest_parent)

Return result
```

First step in the algorithm is to Initialize an empty list to store all edges in the graph. Next, convert the graph dictionary into a list of edges with their weights. For every vertex we check its neighbor and add to edges list and sort edges in ascending order of their weights. We continue by Initializing an empty dictionary to store the parent of each vertex and initialize an empty dictionary to store the rank of each vertex. For each vertex in the graph, we will also add each vertex as its own parent and set its rank to zero. Then we check if adding this edge forms a cycle, if it doesn't then we add the edge to the result and merge the two disjoint sets. Finally, we return the edges of the minimum spanning tree.

Problem 2 Data Structures

Here I used disjoint set data structure in order to implement Kruskal's algorithm. The disjoint set purpose is to track the set-ID of a node. When a an edge is found which connects two nodes from

two different disjoint sets then we merge them using the Union operation. I also used a list to store the edges and use the sortEdges method to sort them.

Problem 2 Analysis of Runtime

This algorithm's runtime depends on the time it takes to sort the edges and the time it takes to process each one. Sorting the edges takes O(E log E) time, where E is the number of edges in the graph. The reason for this is because this algorithm sorts the edges in ascending order of their weights using an algorithm like quicksort.

The disjoint set data structure is used to check for any cycles and then merge the disjoint sets. This operation takes O(E) because we must process each edge once and perform union-find operations on the vertices.

Problem 2 Results

Below are the graphs used and the results from my algorithm implementation.

The first graph used for MST is undirected and weighted with 9 nodes and 12 edges:

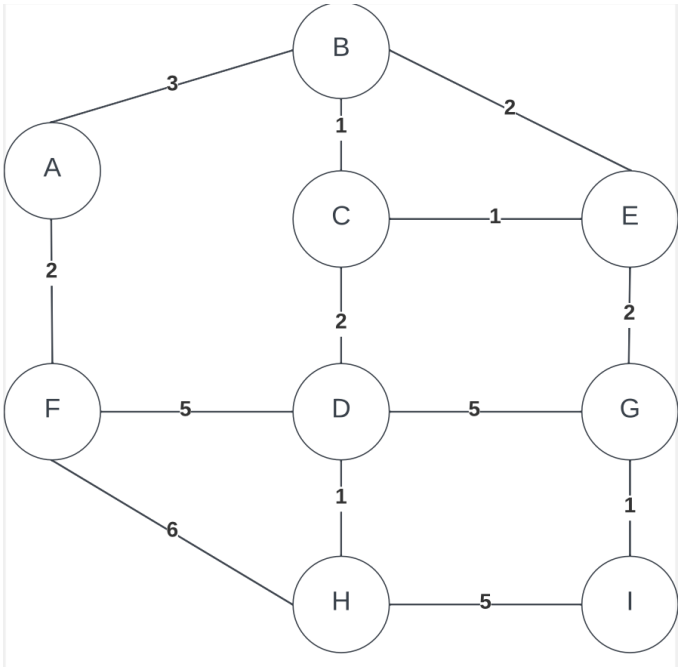

Figure 10. Graph 1: Undirected and weighted.

```
B --> C : 1
C --> E : 1
D --> H : 1
G --> I : 1
A --> F : 2
C --> D : 2
E --> G : 2
A --> B : 3
Minimum Spanning Tree Cost: 13
```

Figure 11. Graph 1 MST results.

The second graph used for MST is undirected and weighted with 9 nodes and 12 edges:

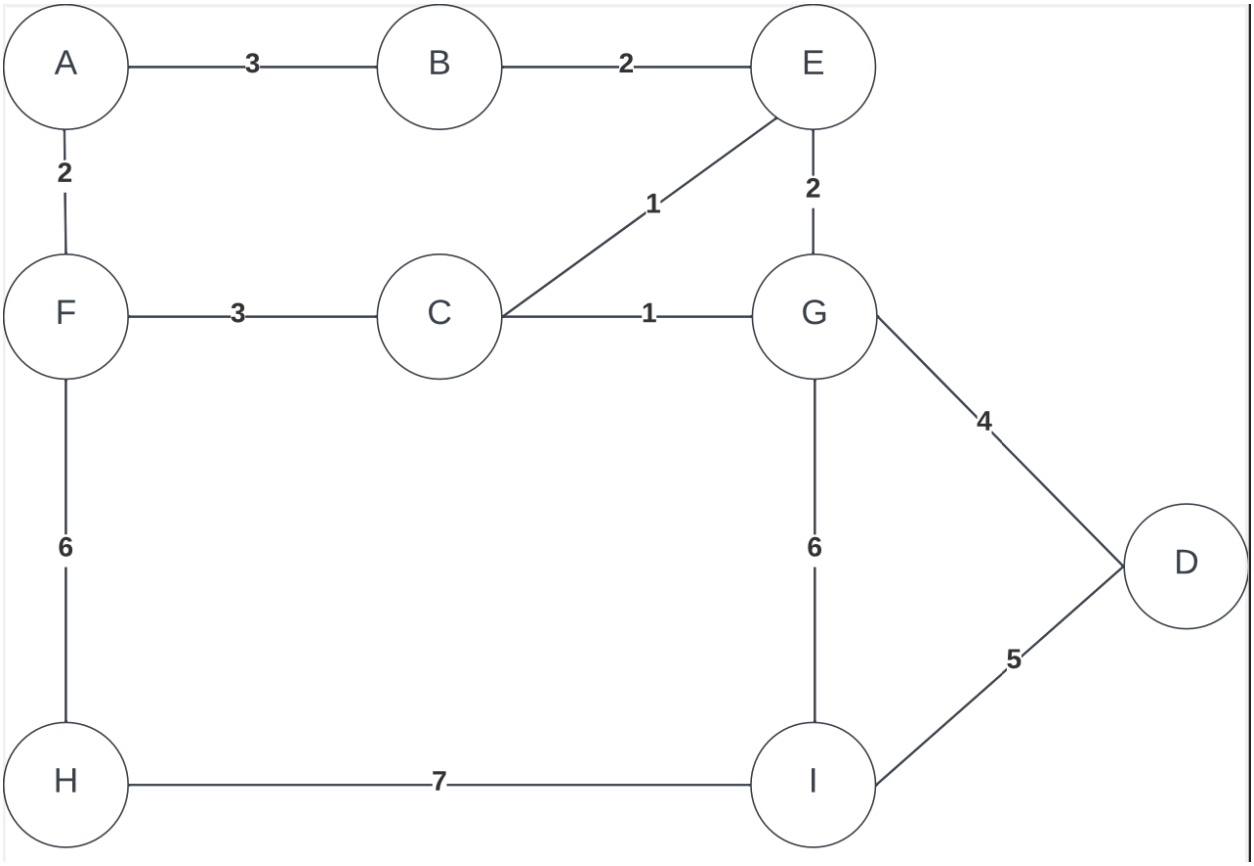

Figure 12. Graph 2: Undirected and weight

```
C --> E : 1
C --> G : 1
A --> F : 2
B --> E : 2
A --> B : 3
D --> G : 4
D --> I : 5
F --> H : 6
Minimum Spanning Tree Cost: 24
```

Figure 13. Graph 2 MST results

The third graph used for MST is undirected and weight. With 9 nodes and 14 nodes.

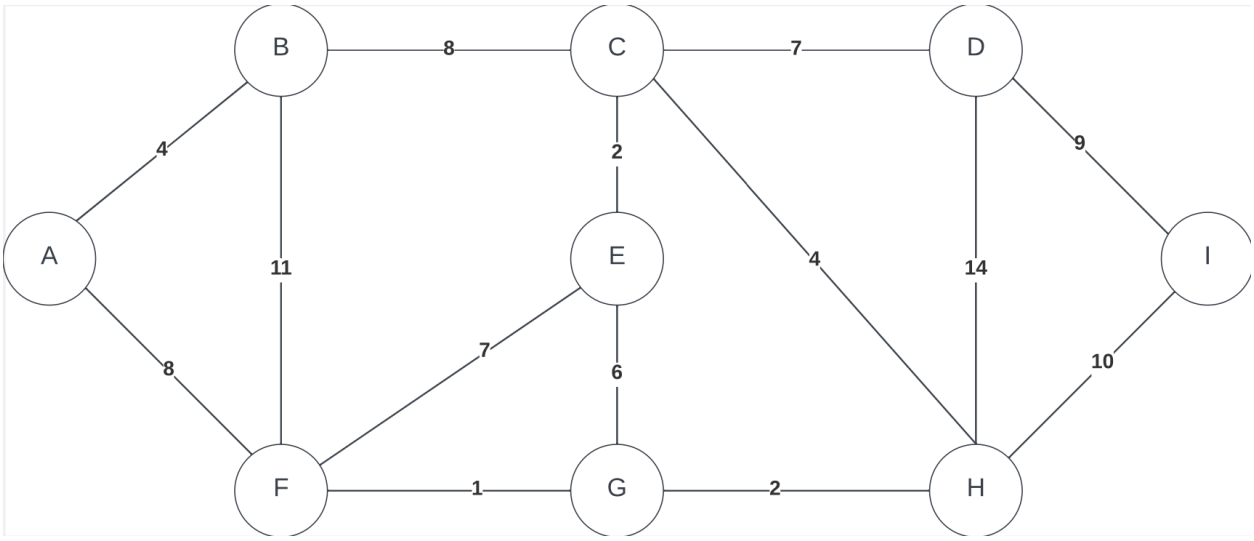

Figure 14. Graph 5: Undirected and weighted.

```
F --> G : 1
C --> E : 2
G --> H : 2
A --> B : 4
C --> H : 4
C --> D : 7
A --> F : 8
D --> I : 9
Minimum Spanning Tree Cost: 37
```

Figure 15. Graph 5 MST results

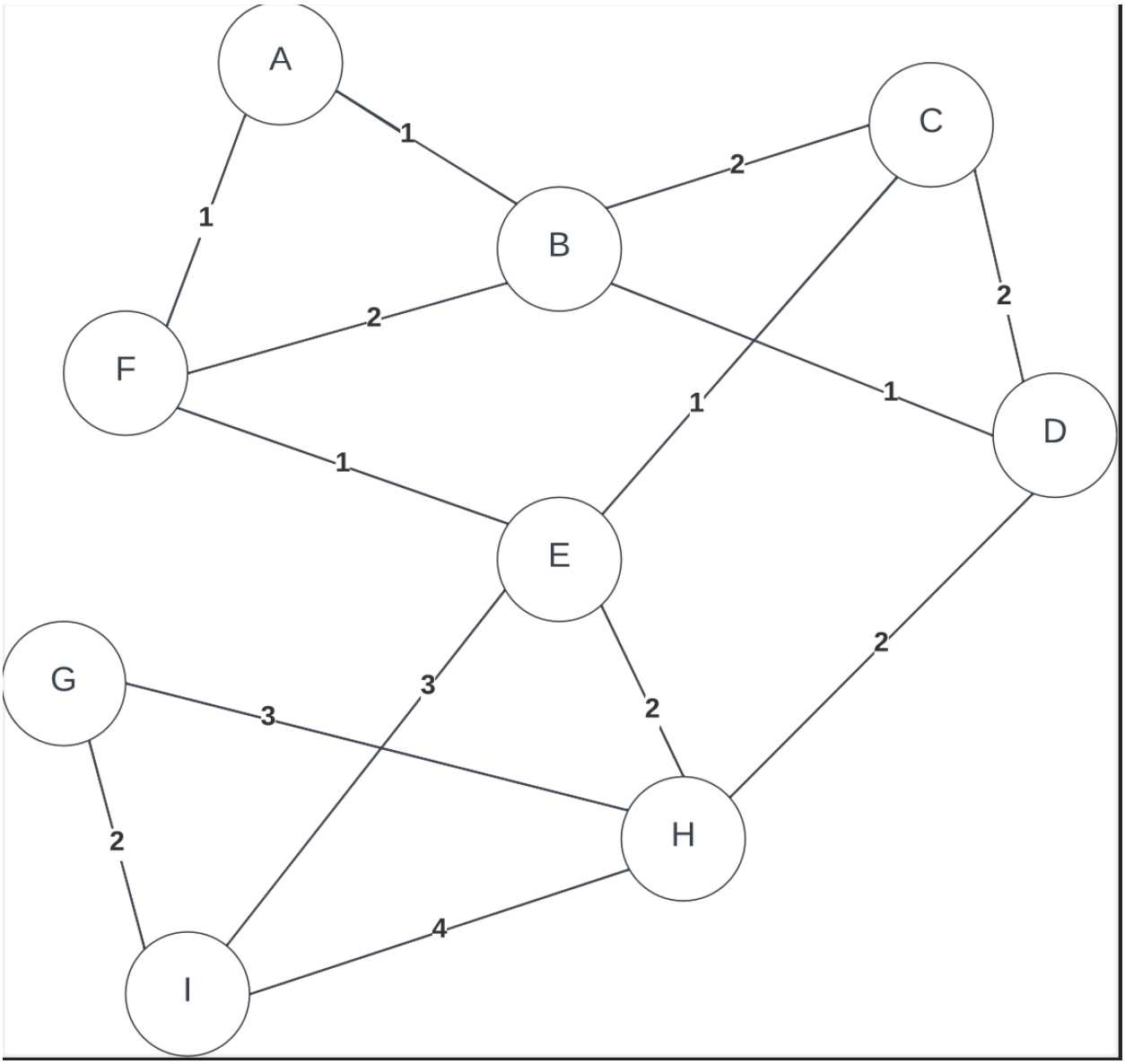

Figure 16. Graph 6: Undirected and weighted

```
A --> B : 1
A --> F : 1
B --> D : 1
E --> F : 1
B --> C : 2
D --> H : 2
G --> I : 2
E --> I : 3
Minimum Spanning Tree Cost: 13
```

Figure 17. Graph 6 MST results.