Making Friends Report

Bao Trung Hoang

April 25, 2025

1 Results

Briefly comment the results, did the script say all your solutions were correct? Approximately how long time does it take for the program to run on the largest input? What takes the majority of the time?

The solution passes all the test case, including the largest one. The program takes about 1 minute to run on the largest input. Most of the time is spent on the largest test case, which is the last one.

The process tasks most of the time would be the adjacency list traversal. For each newly visited node, we need to check all the neighbors of that node. This is costly because it has the time complexity of $O(n)$, which will became significant for the largest input, containing $3 * 10^6$ nodes.

2 Implementation details

How did you implement the solution? Which data structures were used? Which modifications to these data structures were used? What is the overall running time? Why?

The solution is implemented in Python 3.12.3, adapted to the given data. The following data structures are used for the solution:

- The adjacency list is used to present the graph.
  - The index position of the list represents the node number.
  - The value at that index is a list of all neighbors of that node.
  - A neighbor is presented as a tuple of the neighbor node number and the weight of the edge connecting them.
- The priority queue is used to store the edges with the smallest weights being processed.
  - An empty list is used for this purpose.
  - The **heapq** module is used to transform the list into the priority queue.

  - The **heapq.heappush()** function is used to add a new edge to the queue, which ensures that the edge with the smallest weights is retrieved fist when using the **heapq.heappop()** function.
  - The edges connected to a defined first node will be added to the queue. The data structure stored in the queue is **(weight, (source_node, destination_node))**, which is a tuple.

- The **visited** list is used to keep track of the nodes that have been visited.
- The **nodes** list used to store all the node numbers available in the graph.

The following modifications are made to the data structure for the given data:

- The node number is shifted by 1 to match the index position of the adjacency list.
- The priority queue is organized based on the value of the first element, which is the weight of the edge.

Why does the algorithm produce the minimal spanning tree?

The algorithm produces the minimal spanning tree due to the implementation of Prim's algorithm. For each node, the algorithm will always choose the safe edge that connects two nonempty sets, which is **nodes** and **visited**. The process will continue until all the nodes are vissted, ensuring that the minimum cost for the spanning tree. This happens because only the edge with minimal cost is selected at each step.

As mentioned above, the edges are selected when it connects **nodes** (non-visted) and **visited** sets. As soon as the edge is selected, the node number of the destination node would be added to the **visited** set, and would not be visited again. This ensures that each node is only visited once.

The process guarantees that no cycles are formed, the tree will connect all the nodes if there is a path between arbitrary two nodes. By using the priority queue, and by adding only the minimal cost at each step, the minimal cost for the whole tree is guaranteed.

What is the time complexity of the algorithm?

The time complexity for the algorithm is $O(E{*}log(V))$, where E is the number of edges and V is the number of nodes. The time complexity is calculated based on the following:

- Creating the adjacency list takes $O(E)$ time.
- Creating a new priority queue and adding the edges takes $O(N)$
- For each node in the graph, the distance to its neighbors will be checked. This has the time complexity of $O(E * log(N))$, which can explain as follows:

- The time complexity of adding edges to the priority queue is $O(log(n))$, where n is the size of the queue.
- The maximum number of edges is $V * (V - 1)/2$, which can be approximated to $O(V^2)$.
- The insertion operation is at most E, which leads to the time complexity of $O(E * log(V^2)) = O(E * 2log(V) = O(E * log(V)))$
- The same reasoning can be applied to the extraction process.

We can add up the time complexity of each step to get the overall time complexity, and the dominance term is $O(E * log(V))$. So the overall time complexity is $O(E * log(V))$.

What happens if one of the edges have chosen to include collapses? Might there be any problems with that in real application?

There are some consequences if one of the edges is removed from the graph, which are as follows:

- The tree will become disconnected, there will be no path between one node with the rest of the tree. This breaks the property of the spanning tree, which is that will be a path to every arbitrary nodes.
- To restore the connectivity of the tree, the entire algorithm is required to run again. This could lead to a less optimal result.

In real world applications, this could lead to additional workload for other nodes. This required immediate rerouting.

The real world applications of the algorithm and the requirements for the problems

Some real world applications for the algorithm are as follows:

- Designing telecommunication networks
- Creating road network that connects all the locations

The requirements for the problems are as follows:

- The data to the problem have to be presented as connected, undirected, weighted graph.
- The goal here is to find the minimum cost the entire operation.
- No cycles are required.
- The weights for each edge must be positive and meaningful.