Time Complexity Analysis of Offline-6

Kazi Md. Raiyan (2405103)

March 1, 2026

1 Functions

- STL `sort(...)`: Sorts n cities in the acsending order based on their x-coordinates.
- `connect_cities(...)`: Connects the cities recursively in the divide-and-conquer approach with sufficiently low-cost edges.
- `connect_cities_naively(...)`: Serves as the base case for the recursive `connect_cities(...)` function, which uses bruteforce to check all possible edges and greedily add the lowest-cost edge to connect the cities.
- `connect_split_cities(...)`: Connects the cities across the dividing line in the divide-and-conquer approach by checking all possible edges within a certain range from the dividing line and adds the lowest-cost edge to connect the cities across the dividing line.
- `UndirectedGraph::add_edge(...)`: Adds an edge between two nodes in the graph.
- `UndirectedGraph::is_connected(...)`: Checks whether the two passed nodes are connected in the graph using Depth First Search (DFS)

2 Time Complexity Analysis

2.1 STL `sort(...)`

This is the built-in sorting function of C++, which uses a comparison-based sorting algorithm. Therefore, its time complexity is $O(n \log n)$ for sorting n elements.
This is used to sort n cities. Thus this function contributes $\boldsymbol{O(n \log n)}$ to the overall time complexity of the algorithm.

2.2 `UndirectedGraph::add_edge(...)`

This function adds an edge between two nodes in the graph. It involves updating the adjacency list. An `unordered_map` is used to store the adjacency list, and the average time complexity of insertion in an `unordered_map` is $O(1)$. Therefore, the time complexity of this function is $\boldsymbol{O(1)}$.

2.3 `UndirectedGraph::is_connected(...)`

This function checks whether two nodes are connected in the graph using Depth First Search (DFS). In the worst case, DFS visits all nodes and edges in the graph. Therefore, its time complexity is $\boldsymbol{O(n+e)}$, where n is the number of nodes and e is the number of edges in the graph.

2.4 connect_cities_naively(...)

This function is used to connect m cities in the bruteforce manner.
Step 1: First, it uses two nested loops to check all possible pairs of cities and add the edges in a vector, which takes $\boldsymbol{O(m^2)}$ time.
Step 2: Then, it sorts the vector of edges based on their costs. The cost function is a simple arithmetic operation, which takes constant time. Therefore, sorting the edges takes $O(m^2 \log m^2) = \boldsymbol{O(m^2 \log m)}$ time.
Step 3: Finally, it iterates through the sorted edges and adds them to the graph until all cities are connected. In every iteration, it check whether the two cities are already connected using the `UndirectedGraph::is_connected(...)` function, which takes $O(m)$ time in the worst case. If not connected, adds the edge to the vector and the graph using the `UndirectedGraph::add_edge(...)` function, which takes constant time. Since there can be at most $O(m^2)$ edges, this step takes $\boldsymbol{O(m^3)}$ time in the worst case.
Overall: The overall time complexity of the `connect_cities_naively(...)` function is dominated by the last step, which is $\boldsymbol{O(m^3)}$, where m is the number of cities in the base case.

2.5 connect_split_cities(...)

This function is used to connect m cities across the dividing line in the divide-and-conquer approach. It simply uses two nested loops to check all possible pairs of cities across the dividing line within the m range and finds the minimum cost edge. Therefore, the time complexity of this function is $\boldsymbol{O(m^2)}$.

2.6 connect_cities(...)

This is the main recursive function that connects the cities using the divide-and-conquer approach.
Base Case: If the number of cities is less than or equal to m. It calls the `connect_cities_naively(...)` function, which takes $O(m^3)$ time.
Divide: It divides the cities into two halves and recursively calls itself on each half.
Merge: After connecting the cities in the left and right halves, it calls the `connect_split_cities(...)` function to connect the cities across the dividing line, which takes $O(m^2)$ time.
Recurrence Relation: Thus, the time complexity of this function can be expressed as the following recurrence relation:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(m^2) & n > m \\ O(m^3) & n \leq m \end{cases}$$

where n is the number of cities in the current subproblem and m is the number of cities in the base case.

Recursion Tree Approach:

- **Leaf nodes (base cases):**
  The cities are divided into two halves until the number of cities in the subproblem is less than or equal to m.
  Therefore, $m \approx \frac{n}{2^k}$, where k is the height of the recursion tree. Therefore, the number of leaves in the recursion tree is $2^k \approx \frac{n}{m}$.
  So, the total time taken by the base cases is $O(m^3) \cdot \frac{n}{m} = O(nm^2)$.

- **Internal nodes (merge steps):**
  Each internal node of the recursion tree takes $O(m^2)$ time to connect the cities across the dividing line. The number of internal nodes is roughly the same as the number of leaves. Therefore, the total time taken by the internal nodes is $O(m^2) \cdot \frac{n}{m} = O(nm)$.

- **Overall:**
  As per the problem statement, $2 \leq m \leq 50$. Hence we can treat m as a constant. Therefore, total time taken by the base cases is $O(n)$ and the total time taken by the internal nodes is also $O(n)$.
  Therefore, the overall time complexity of the `connect_cities(...)` function is $\boldsymbol{O(n)}$ for sufficiently large n.

Master Theorem Approach:

As m is a small constant, the recurrence relation can be simplified to

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

for sufficiently large n. Using the Master Theorem, we have $a = 2$, $b = 2$, and $f(n) = O(1)$. Therefore, $\log_b a = \log_2 2 = 1$. So, this is the first case of the Master Theorem. Thus, the solution to the recurrence is $T(n) = O(n^{\log_b a}) = O(n)$.
Thus, the overall time complexity of the `connect_cities(...)` function is $\boldsymbol{O(n)}$ for sufficiently large n.

2.7 Overall Time Complexity

The overall time complexity of the algorithm is dominated by the **sorting step**. Therefore, the overall time complexity of the algorithm is $\boxed{\boldsymbol{O(n \log n)}}$ for sufficiently large n.