This code will work for directed graphs too:
An eg input would be:

```
int graph[V][V] = {
    {0, 4, 0, 0, 0, 0, 0, 8, 0},
    {0, 0, 8, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 7, 0, 4, 0, 0, 2},
    {0, 0, 0, 0, 9, 14, 0, 0, 0},
    {0, 0, 0, 0, 0, 10, 0, 0, 0},
    {0, 0, 4, 0, 0, 0, 2, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 1, 6},
    {0, 0, 0, 0, 0, 0, 0, 0, 7},
    {0, 0, 2, 0, 0, 0, 6, 0, 0}};
```

The time complexity of Dijkstra's algorithm is O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph1. The auxiliary space required is O(V) to store the distance values of vertices1.

To improve the time complexity of Dijkstra's algorithm, we can use a min-priority queue instead of a linear search to find the vertex with the smallest tentative distance.

Below is the algorithm based on the above idea:

Initialize the distance values and priority queue.
Insert the source node into the priority queue with distance 0.
While the priority queue is not empty:

- Extract the node with the minimum distance from the priority queue.
- Update the distances of its neighbors if a shorter path is found.
- Insert updated neighbors into the priority queue.

<u>Hence the Time Complexity will be O((V + E) log V) as:</u>
<u>Initialization: O(V)</u>

Setting up distance array (dist), priority queue, and other auxiliary arrays.

<u>Main Loop: O(V log V + E log V)</u>

The main loop runs V times, and in each iteration, the minimum distance vertex is extracted from the priority queue, which takes O(log V) time.
For each vertex, its adjacent vertices are relaxed (distance updated) if a shorter path is found. This can happen at most once for each edge, leading to a total of E relaxation operations.

So, the time complexity of the main loop is O(V log V + E log V), which can be simplified to O((V + E) log V).

The time complexity and auxiliary space complexity you've mentioned, O((V + E) log V) and O(V), respectively, correspond to an optimized version of Dijkstra's algorithm using a priority queue or a min-heap data structure.

Auxiliary Space Complexity: O(V)
dist array: O(V) - To store the shortest distances from the source vertex to all other vertices.
priority queue: O(V) - In the worst case, all vertices can be added to the priority queue.

The relevant data structures for implementing Dijkstra's algorithm are:

Adjacency matrix: A 2D array of size V x V where V is the number of vertices in the graph. If there is an edge between vertex i and vertex j, then the value of the matrix at (i, j) is the weight of the edge. Otherwise, it is 0.
Adjacency list: An array of linked lists where each element of the array represents a vertex and the linked list contains the adjacent vertices and their weights.
Min-priority queue: A data structure that supports the following operations:
Insertion of an element with a key
Deletion of the element with the minimum key
Decrease-key operation to decrease the key of an element1.