**1. What is the big-Oh space complexity of Dijkstra's? Justify your answer.**

Answer:

The big-Oh space complexity of Dijkstra's algorithm is O(V), where V is the number of vertices in the graph. This is because the algorithm maintains a set of vertices (usually implemented as a min-heap) to keep track of the shortest distances found so far. The size of this set is bounded by the number of vertices in the graph. Additionally, the algorithm requires a fixed amount of space to store the distance from the source to each vertex and possibly other information such as the predecessor vertex for each vertex. However, the space required for these additional data structures does not depend on the size of the input and can be considered constant. Therefore, the overall space complexity is dominated by the space required for the vertex set, resulting in O(V).

Proof:

The space complexity analysis considers the amount of additional memory required by the algorithm as a function of the input size. In Dijkstra's algorithm, the input size is primarily determined by the number of vertices in the graph. The algorithm needs to store information about each vertex, such as its current shortest distance from the source. This information is typically stored in an array or similar data structure, whose size is proportional to the number of vertices. The min-heap used to efficiently select the vertex with the smallest distance also requires space proportional to the number of vertices. Therefore, the overall space requirement grows linearly with the number of vertices, resulting in O(V) space complexity.

**2. What is the big-Oh time complexity for Dijkstra's? Justify your answer.**

Answer:

The big-Oh time complexity of Dijkstra's algorithm is O(V^2) in the worst case, where V is the number of vertices in the graph. This occurs when the algorithm is implemented using a simple linear search to find the minimum distance vertex in each iteration. However, with the use of a min-heap to efficiently retrieve the minimum distance vertex, the time complexity can be improved to O((V + E)logV), where E is the number of edges in the graph. This is because inserting and extracting the minimum element from a min-heap takes O(logV) time, and this operation is performed V times (once for each vertex). Additionally, updating the distances of adjacent vertices after relaxing an edge takes O(E) time in total.

Proof :

The time complexity analysis of Dijkstra's algorithm considers the number of operations required to compute the shortest paths from the source to all other vertices. Without using a min-heap, finding the vertex with the minimum distance in each iteration takes O(V) time using a linear search, resulting in a total time complexity of O(V^2) since this operation is repeated V times. However, by using a min-heap, the time to find the minimum distance vertex is reduced to O(logV), leading to an overall time complexity of O((V + E)logV). This improvement is achieved because the min-heap efficiently maintains the vertices in order of their distances, allowing for faster retrieval of the minimum distance vertex. The O(E) term accounts for the time spent relaxing edges and updating distances.

**3. Does your implementation operate correctly? How do you know?**

Answer:

To determine whether my implementation of Dijkstra's algorithm operates correctly, I would follow these steps:

Unit Testing: I would write unit tests for various scenarios, including graphs with different topologies (e.g., directed, undirected, weighted, unweighted), graphs with negative weights (to ensure my implementation correctly handles the inapplicability of Dijkstra's algorithm in this case), and graphs with a single source and multiple destinations. Each test case would verify that the computed shortest paths match the expected results.

Comparison with Known Solutions: For small graphs, I would manually compute the shortest paths using a different method (e.g., breadth-first search for unweighted graphs) and compare the results with those obtained from my Dijkstra's implementation. For larger graphs, I would use existing, trusted implementations of Dijkstra's algorithm as a reference and compare the outputs.

Edge Cases and Exception Handling: I would ensure that my implementation handles edge cases correctly, such as empty graphs, graphs with a single vertex, or graphs where all vertices are disconnected from the source. Additionally, I would test for any potential exceptions that may occur during execution, such as division by zero or accessing out-of-bounds indices, and verify that they are handled appropriately.

Performance Analysis: Although not strictly related to correctness, I would also analyze the performance of my implementation to ensure it scales well with the size of the input. This could involve profiling the code to identify any bottlenecks or inefficiencies and optimizing them accordingly.