Team Name: **[MORKS]**
University Name: **[Sri Lanka Institute of Information Technology]**
Team Members: **[Shazaan Ashraff, Sandali Sandagomi, Himasha Hettiarachchi]**

Case A Solution: Optimizing Delivery Routes

Introduction

The problem represents a city as a directed weighted graph, which requires calculating the shortest delivery times from a central warehouse to multiple delivery points. The city consists of intersections and roads with traffic delays. This real-world algorithmic problem mirrors the classical shortest path algorithm in graph theory.

After consideration, the graph was noticed to have non-negativity weights, and it was a no brainer to use the Dijkstra's algorithm. Also, the algorithm was implemented with priority queues to address the scaling possibilities. This report elaborates the thought process, design implementation, edge case handling and optimizations of space and time in this algorithm design.

The Thought Process

Problem visualization: The problem was first analyzed well and considered, the city grid is then modelled as a directed graph where the intersection are nodes and roads are edges with weights representing traffic delays. The main objective was to find the shortest path from a single source (warehouse) to multiple delivery points (K destinations)

Algorithm selection: To compute the shortest distance a graph traversal theory was necessary Dijkstra's algorithm was an intuitive choice. The different other choices and the reasonings with each of them will be explained separately. We also optimized the algorithm by using a priority queue (min-heap) for edge relaxation.

Challenges: Efficient handling up to 10^5 intersections and roads while balancing time and space complexity for optimal performance

Why Dijkstra's algorithm and priority queue

A simpler approach would have been BFS or DFS which would require listing out all paths and selecting the shortest path, which could result in exponential time complexity $O(2^N)$ for large graphs.

We also considered the bellman ford algorithm, it also computes shortest path also was well suited for graphs with negative weights, but since we did not need to accommodate negative weights (Realistically delays or distances cannot be negative in road networks) we decided to not use it since it had O (N.M) time complexity.

Floyd-Warshall Algorithm was also considered but it was an overkill for a single source shortest path problem and worked well only for paired paths.

The A* algorithm needed a guide to search which is not applicable in this problem. And finally, after all consideration we concluded using Dijkstra's algorithm for this problem solution.

We also searched for an efficient way to handle the nodes during each iteration, first we considered using a set since it was easy to implement but soon realized it was very hard to extract the smallest distance. In contrast priority queue (min heap) always maintains the smallest element at the top, ensuring O (log N) operations. This makes it a good choice for algorithm implementation.

Handling Edge Cases

1. **No nodes or Edges:** If the graph is completely empty no path exists to deliver. “No delivery points specified” message will be displayed.
2. **No delivery points:** If no delivery points are provided, there is nothing to compute. "No delivery points specified” will be displayed.
3. **Single Node:** Because the source is the only node and delivery point, it will return the source distance to itself: zero.
4. **Disconnected Graph**: If nodes in the graph are not connected to the source, those nodes are not reachable from the source. “Delivery point is unreachable from the source” message will be displayed.
5. **Multiple Components**: since two different disconnected graphs cannot be travelled. “Delivery point is unreachable from the source” message will be displayed.
6. **Negative Weights (invalid for Dijkstra):** Since the question is regarding the shortest distance, we did not account for negative distancing due to it being practically wrong in this scenario (negative traffic delay doesn’t exist.) We have added an edge case just in case, which will exit the program with a message prompt. "Invalid input: Negative edge weights detected. Dijkstra's algorithm requires non-negative weights."
7. **Zero-Weight Edges:** it will consider zero weights and give the shortest routes for the distance.
8. **Source Not in Graph:** If the source is missing from the graph, you cannot compute the delivery path. "Delivery point is unreachable from the source." Is displayed because the source is not part of the graph.
9. **Source is a Delivery Point:** The distance from the source to itself is zero. Therefore, the output will be zero.
10. **Self-Loops and cycles (A node have an edge pointing to itself):** This does not affect the shortest path calculation because Dijkstra’s algorithm anyway ignores it.
11. **Nodes with the Same Weight:** If two or more nodes have the same shortest distance during the algorithm's execution, they will all be added to the priority queue with the same priority (weight).The priority queue will process these nodes in any order, but the algorithm will still correctly calculate the shortest paths to their neighbors. Therefore, it will not affect Dijkstra’s algorithm.

Time complexity

The time complexity of the problem relies on three parts, the construction of the graph, Dijkstra’s algorithm, and how many times we insert and extract elements. Let us break them down and analyze the time complexity for each part.

1. **Graph Construction:**

The roads(edges) and intersections(nodes) are represented by a graph and an adjacency list is used to process this efficiently, and each node has a list of neighboring nodes and the weights (which represent the delay of the roads) connecting them. For each edge(road) we need to add the connecting nodes and the delay as an entry to the adjacency list. Therefore, we need to iterate over all the M edges once and it takes a constant time per edge O (1).

Time complexity is O(M)

2. **Dijkstra's Algorithm**

The two main operations are extracting the minimum element from the priority queue (min-heap) and updating the distances of neighbors.

The priority queue (min-heap) will repeatedly extract the node with the smallest distance and update the distances of its neighbors. The extracting of the minimum node from a priority queue takes O (log N) where N is the number of nodes. And this will be repeated for all node, therefore N extractions in total one for each node.

Time complexity is O (N log N)

Each time we update the distance, we may need to insert a node in the heap, which takes O (log N) this operation is performed M times over M edges.

Time complexity is O (M log N)

3. **Result Extraction:**

Extracting the results for each delivery point takes O(K) where K is the number of delivery points. However, the number of edges is usually larger than the number of delivery points and since it specifies that the grid size can go up to as large as 10^5 we can ignore the addition of O(K) to the equation.

Total Time Complexity:

Best case: It is a sparse graph: M≈N−1 (close to the minimum number of edges). Therefore, few delivery points. The algorithm still processes M edges and N nodes.

Time Complexity: O((M+N) log N)

Worst Case: Dense graph: M≈N^2 (maximum number of edges for a directed graph). All M edges contribute to heap update and all N nodes require heap extraction.

Time Complexity: O((M+N) log N)

Average Case: Some nodes may be unreachable, reducing the number of heap operations.

Time Complexity: O((M+N) log N)

Space Complexity Analysis

This primarily depends on the data structures used to represent the graph, store distances, and manage the priority queue.

The graph is stored as an adjacency list using a defaultdict. Space requirement: O(M) where M is the number of edges, since each edge is stored only once in the adjacency list. A dictionary distances is used to store the shortest distance for each node. Space requirement: O(N) where N is the number of nodes, and each node is entered in the dictionary. The priority queue stores all the nodes that need to be processed along with the weight, in worse case all N nodes are added. Space requirement: O(N)

Total Space complexity: O(N+M)

Future optimization possibilities

1. **Dynamic traffic data:** Add real time updates by running the algorithm again and again or editing the edge weights dynamically.
2. **Divide and conquer:** For very large grids, computations can be done in a partitioned manner. We can compute the shortest paths and merge them as a global addition to reduce the time taken to compute.
3. **Fibonacci heaps:** We can also use Fibonacci heaps instead of the used standard binary heaps, this can reduce the complexity of decrease-key operations, making the algorithm faster for large scale.

Conclusion

This report presents a solution to the logistics optimization problem using the Dijkstra's algorithm, to compute the shortest paths in large scale weighted graphs making it the most suitable choice for the given case. Compared to the alternatives like BFS and DFS and other popular algorithms. The design can be enhanced with parallelization and dynamic updates in scenarios with dynamic traffic conditions and its adaptability for modern logistics.

Link to the code: https://drive.google.com/drive/folders/1ZUFMW3GXPfCFLOKMBYFQmAdh20DTv82d?usp=sharing