When I was making this program I wanted to make sure I tried to develop an efficient use of this graph, the addVertex method utilizes a hash table (std::unordered\_map), making sure that the insertion of vertices with an average time complexity of **O(1)**. This design choice allows for rapid expansion of the graph regardless of its size.

Another thing to bring up is the addEdge method employs a vector (std::vector), providing constant-time **O(1)** complexity for adding edges. This straightforward approach ensures a quick establishment of connections between existing vertices, which is essential for dynamically building and modifying the graph structure.

When I was attempting to calculate the shortest path (shortestPath method) between two vertices used Dijkstra's algorithm with a priority queue. In the worst case, this algorithm operates with a time complexity of O((V + E) log V), where V is the vertices and E is the edges. The efficient handling of priority updates ensures swift pathfinding even in dense graphs.

Making the minimum spanning tree (minimumSpanningTree method) used Kruskal's algorithm, primarily sorting edges by weight. This algorithm achieves a time complexity of **O(E log E)**, optimized to **O(E log V)** with appropriate data structures. This is supposed to make it suitable for efficiently identifying the optimal set of edges connecting all vertices without cycles.

In conclusion, the implemented graph operations in the C++ program are designed to handle various tasks within this graph for efficiency, these are crucial for applications such as route planning and network optimization in the delivery simulation game. I was trying to ensure that the program performs optimally in scenarios where computational efficiency is vital for real-time decision-making and gameplay experience.