Big O(Richard)

**Big O Analysis**

Our DFS algorithm (Graph::DFS())has a big-o performance of O(). This is because this is a recursive function that has each vertex perform a comparison on each of its adjacent vertices to find the shortest edge. Therefore in the worst case scenario each vertex would compare against each other vertex leading to a performance of O(). This means that increasing the amount of vertices increases the performance time quadratically.

Our BFS algorithm (Graph::BFS()) has a big-o performance of O(). This is because this function uses a queue to traverse the graph in layers with each vertex performing a comparison on each of its adjacent vertices. Therefore in the worst case scenario each vertex would compare against each other vertex leading to a performance of O(). This means that increases in the amount of vertices increase the performance time quadratically.

Our MST algorithm (Graph::PrimMST()) has a big-o performance of O(). This is because this function finds the smallest edge repeatedly by comparing each edge to each other edge. Since each edge compares against each other edge, the performance is O(). This means that increases in the amount of edges results in a quadratic increase in performance time.

Our Dijkstra’s algorithm (Graph::Dijkstra()) has a big-o performance of O(). This is because each vertex performs a comparison on each of its adjacent edges to find the shortest edge. Therefore in the worst case scenario each vertex would have comparisons against each other vertex, resulting in a performance of O(). This means that increasing the amount of vertices increases the performance time quadratically.

Our method for taking a complete trip (MainWidow::on\_completeTripButton\_clicked()) has a big-o performance of O(n). This is because in the complete trip each vertex is visited only once meaning that the number of comparisons is proportional to the amount of vertices, giving this function a performance of O(n). This means that increases in the amount of vertices increase the performance time linearly.

**Data structures**

The first data structure we used was a vector. We use vector’s several times throughout the program. For example, we use a vector of WeightEdge’s in our DFS algorithm to store the weights of all of the edges.

The second data structure we used was a queue. We use a queue in our BFS algorithm to visit each vertex layer by layer. This specific queue was implemented using a vector of QStrings.

The third data structure we used was an array. We use arrays many times in our project. For example, we use them in all of our traversal functions as bool arrays to store whether or not each vertex has been visited.

Additionally, we used a map to hold the souvenirs in the cart menu. It specifically held the Souvenirs struct.