Complexity:

Dijkstra's Algorithm:

For the priority queue approach, complexity is O(E log V) where V is the number of vertices and E is the number of edges.

Kruskal's Algorithm:

Kruskal's complexity is O(E log E)

Basic Graph Operations:

O(1) on average, but the worst case is O(E) where E is the number of edges.

Vertex:

```
struct GraphNode {
    std::string name;
    std::vector<Edge *> neighbors;

    GraphNode(const std::string &n) : name(n) {}  //Used strings instead of char to be able name nodes by school names
};                                                // If not done, then the code gets a 'int'-'char' conversion error
```

Edge:

```
struct Edge {
    int weight;
    GraphNode *source;
    GraphNode *destination;

    Edge(int w, GraphNode *s, GraphNode *d) : weight(w), source(s), destination(d) {} //Establishing weight and directionality.
};
```

Dijkstra's:

```cpp
void dijkstra(const std::string &start) {
    if (nodes.find(start) == nodes.end()) return;

    std::unordered_map<GraphNode *, int> distances;
    std::unordered_map<GraphNode *, GraphNode *> previous;

    //priotity queue
    auto cmp = [](const std::pair<GraphNode *, int> &left, const std::pair<GraphNode *, int> &right) {
        return left.second > right.second;
    };
    std::priority_queue<std::pair<GraphNode *, int>, std::vector<std::pair<GraphNode *, int>>, decltype(cmp)> pq(cmp);

    //pseudocode implementation from wiki
    for (const auto &pair : nodes) {
        distances[pair.second] = INT_MAX;
        previous[pair.second] = nullptr;
    }

    distances[nodes[start]] = 0;
    pq.push({nodes[start], 0});


    //perform Dijkstra's algorithm
    while (!pq.empty()) {
        GraphNode *current = pq.top().first;
        pq.pop();

        for (Edge *edge : current->neighbors) {
            GraphNode *neighbor = (edge->source == current) ? edge->destination : edge->source;
            int alt = distances[current] + edge->weight;

            if (alt < distances[neighbor]) {
                distances[neighbor] = alt;
                previous[neighbor] = current;
                pq.push({neighbor, alt});
            }
        }
    }
```

Kruskal's:

```
void kruskal() {
    std::vector<Edge *> result;
    std::unordered_map<std::string, int> nodeIndex;
    std::vector<int> parent(nodes.size(), -1);
    int cost = 0;

    int index = 0;
    for (const auto &pair : nodes) {
        nodeIndex[pair.first] = index++;
    }
    //Sort edges by their weight
    std::sort(edges.begin(), edges.end(), [](Edge *a, Edge *b) {
        return a->weight < b->weight;
    });
    //using helper function find()
    for (Edge *edge : edges) {
        int x = find(parent, nodeIndex[edge->source->name]);
        int y = find(parent, nodeIndex[edge->destination->name]);

        if (x == -1 || y == -1) {
            std::cerr << "Error: Invalid index returned by find function." << std::endl;
            continue;
        }
        //Check if current tree creates a cycle using helper function union()
        if (x != y) {
            result.push_back(edge);
            Union(parent, x, y);
            cost += edge->weight;
        }
    }

    std::cout << "Edges in the minimum spanning tree:" << std::endl;
    for (Edge *edge : result) {
        std::cout << edge->source->name << " - " << edge->destination->name << " (weight " << edge->weight << ")" << std::endl;
    }
    std::cout << "Total Miles : " << cost << std::endl;
}
```

Testing:
Instead of creating different testing functions, I decided to create one big test. This test implements all the functions and should meet all criteria in my Design PDF. I did this as a way to solve the problem that I created in my Design PDF and show how a graph can be useful!

Code:

```
int main() {
    std::cout << "Starting the graph setup..." << std::endl;

    Graph graph;

    // Create nodes
    graph.addNode("PSU");
    graph.addNode("EOU");
    graph.addNode("UofO");
    graph.addNode("LBCC");
    graph.addNode("OSU");
    graph.addNode("LU");

    // Create edges
    graph.addEdge(162, "PSU", "EOU");
    graph.addEdge(40, "PSU", "LU");
    graph.addEdge(72, "PSU", "LBCC");
    graph.addEdge(113, "PSU", "UofO");

    graph.addEdge(123, "EOU", "LBCC");
    graph.addEdge(127, "EOU", "UofO");

    graph.addEdge(43, "UofO", "LBCC");

    graph.addEdge(8, "LBCC", "OSU");
    graph.addEdge(48, "LBCC", "LU");

    graph.addEdge(47, "OSU", "LU");

    std::cout << "Graph setup complete. Displaying the graph:" << std::endl;

    // Display the graph. Can you this output to draw the graph by hand
    graph.display();

    std::cout << "\nRunning Dijkstra's Algorithm from node 'OSU':" << std::endl;

    // Find shortest paths from node 'OSU' to other schools.
    graph.dijkstra("OSU");

    std::cout << "\nRunning Kruskal's Algorithm to find the MST:" << std::endl;

    // Find the minimum spanning tree. This can me used to save mileage
    graph.kruskal();

    return 0;
}
```