# Solar Sales Documentation Guide

Developer: Immanuel Morris

**Program Goals:**
The goal of this program is to provide a sales person a list of routes and cities in which to complete a sales attempt. Home base and the starting point is Riverside, and neighboring cities include: Moreno Valley, Perris, and Hemet.

**Program Outline**
In addition to providing routes, the program also offers paths that could be considered "shortest" from origin. It also provides a city matrix of all mentioned cities and the distance (represented in mileage) between each one. With the help of the matrix, the user (sales person) can decide which path might be best to start at, when they begin their workday, or they can choose which path to take, after making their first or second stop. Lastly, this program provides a list of city adjacencies, by showing which cities are next to each other based off the given list. This may be helpful if there is a new hire whom may not be familiar with this particular part of the Inland Empire. The user will be able to use this option to plan their route accordingly.

**Algorithm Implementation/Explanation:**
In the initial part of the program, its necessary to create a custom data structure to be implemented inside of a class. So a structure type named Node is created, and nested with 2 integer categories: value, and cost. Then within the same nest, we call on the struct Node, turn it into a pointer, and point to the next node.

Secondly, another custom data structure is created called edge, and this variable has 3 integers: source, destination, and weight. Each value will contain appropriate information relative to the distance in which each city is from each other.

After each custom structure is created, the Graph Class is established and implemented. Initially, it calls on the Node pointer (established earlier) and connects it to a custom function for adjacency - which contains a mix the values inside of the Edge and Node structure. Specifically, this function contains: value, weight, and the Node pointer (which points to a custom variable called head). Within the function we use the Node structure to point to a newly allocated Node, and have that point to value and weight, and head (the 3 parameters of the function). Ultimately this returns the 3 characteristics of a newly created node. Towards the end of the class definition we also establish a variable for the number of nodes within the graph.

Next, we get to the public information of the Graph class (no private info) and once again call upon the Node structure to point to a variable called "head". After this, a graph constructor is created for the head and has parameters for each city to be defined. Within this constructor lie the parameters for the struct Edge, and two integer values (these will represent 2 cities). The “head” variable is allocated for new memory and points to a new Node. Following this, a loop is then created for all verticies to be represented as “heads”, and then an additional loop is created for edges to be created between them. Within this second loop (which IS NOT nested) the source, destination, and weight values are all generated and values stored within each iteration. After this process is done, the Node struct is called upon yet again, and pointing to a new node which calls upon the adjacency function established earlier (it is then deconstructed at the end).

Next in the algorithm explanation there is a custom function that prints out a visual list of arrows and statements (based off of the class groundwork). There are two of the functions created with slightly variating characteristics. The first one prints out  arrows that can be visually seen by the user, and the second one prints out comments and mileage for the user. It should be noted that the first printList function needs only a head value as a parameter, and the second printList needs a head value and integer value.

We also need to have edges between two points so there’s a custom function called AddEdge that allows you to place any two cities, and uses the push_back call to compare how close they are to each other.

Continuing down the list, there’s a function called printGraph that actually prints the graph list based off of the Add edge function earlier. PrintGraph parameters contain an array, and single integer value (Vertex) and creates a loop to cycle through each city. After the loop is conducted the information uses an auto keyword to automatically place each result next to each other, based off of their adjacency. A second variation of printGraph function is used, but it has the same inner working of the first one, the only exception is the insides pertain to the “shortest route” option chosen.

Next we have what might be the most important implementation of the program, which is the custom function called menuSelection. This drives the main program, and is the meat (if you will) of the entire program. This function consists of a do-while loop that provides the user with menu options, so that they may make the best choice dependant on their needs of the sale. In a list of 5 options choices are given in such a user can consider in which path to start. After the menu is displayed, a bit of housekeeping is designed to keep the user within the parameters of the 5 options only.

Choice 1:
Should the user choose option 1 (Travel Route options) a map displays of each city an how they are connected. Within the first choice the algorithm takes an array container, and pairs each city 2 by 2. It then takes the amount of vertices (cities) to compare and compares the amount of edges to connect. Next, the graph class is called upon, and the parameters for edges, and vertices is filled in. Lastly a visual dashed line is displayed along with calling the printList function containing the elements from the graph class head looped elements.

Choice 2:
Should the user choose option 2, a matrix field map is displayed with accompanying mileage between 2 cities. Similarly, to option 1, the Edge class called which contains the edge data

structure which holds an array or container of 3 variables: 2 cities and the literal mileage between them. It then takes the amount of vertices (cities) to compare and compares the amount of edges to connect. Next, the graph class is called upon, and the parameters for edges, and vertices is filled in. Lastly a visual dashed line is displayed along with calling the printList function containing the elements from the graph class head looped elements, along with an extra parameter for mileage.

Choice 3
Should the user choose option 3, an adjacency list will be displayed. This section starts with initializing a a variable for a total number of vertices (cities). Next a vector integer data type is called upon for ease of use rather than a simple array. This vector then points to newly allocated memory which holds the container for the vertex(city) value. A visual representation of the city list is then displayed, and afterward the addEdge function (established earlier) is called upon. The parameters for this function is filled with the allocated container name and two cities to compare adjacency to. The printGraph function is called upon to display arrow direction, and the container name, and vertex value are implemented as parameters. Since new memory is being allocated, at the end, the running memory is deleted.

Choice 4
Should the user choose option 4, an adjacency list will be displayed (similarly to option 3) but with an additional caveat. In this choice, all adjacency's are based off of the starting point (which is city 0 (Riverside)). This option starts similarly by introducing a local vector variable that is a pointer. This pointer points to newly allocated memory and stored in the variable. A city list is then displayed for the user, and the once again the addEdge function is called upon. The parameters this time contain the variable name for the pointer, the starting city (Riverside), and the neighboring cities. A variation of the printGraph is then implemented which uses 2 parameters: the custom vector variable and the, input variable for the amount of cities.

Choice 5
Should the user choose option 5, it is a simple exit command out of the do-while loop. The while portion has a condition that as long as option 5 is NOT chosen, the program will not end. If option 5 is chosen, there is a goodbye message and the program ends.

For the main portion of the program, there is very little information needed. We simply establish an integer variable that allows for user choice, and call on the menuSelection function to drive the entire program. Once that is done, the program will run smoothly.

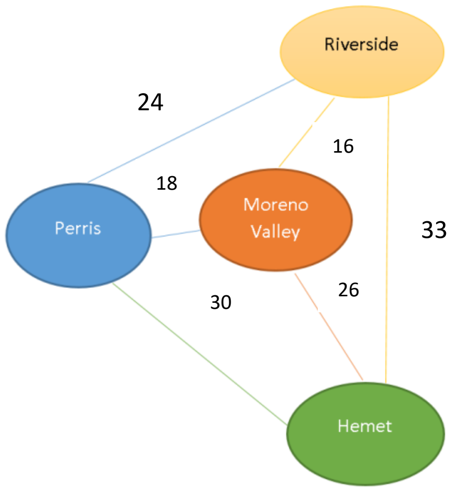
## ***Discrete Structure Importance:***

It's important to note that the custom struct algorithm paves the way for these new data types to be tied in seamlessly into a new class environment. With the power to essentially create one's own ability for ease and simplicity creates an essential characteristic of Discrete mathematics. In addition, merging the concept and usefulness of pointers with vectors for array sizes has amplified and elevated the algorithm to work much more efficiently than taking a more rudimentary approach.

**Program Limitations:**
When it comes to creating new allocated memory it's absolutely essential to delete the new variable because C++ simply cannot handle memory dumping. Also, as the program currently stands, adding a plus 1 to have the cities start from number1 has incurred an error using Visual Studio. For sake of ease and time, the first city starts at number 0 (zero).

# Visual Diagrams:

Below is a base diagram of each city, and the respective miles between them. Please keep in mind, the origin or start point should be Riverside.



**City Organization (0-3)**
Riverside to Moreno Valley, then Perris, Then Hemet
Riverside is 0, Moreno valley is 1, Perris is 2, Hemet is 3

**Edge Connection:**
Edges: {{0,1},{0,2},{0,3},{1,2},{1,3},{2,3}}

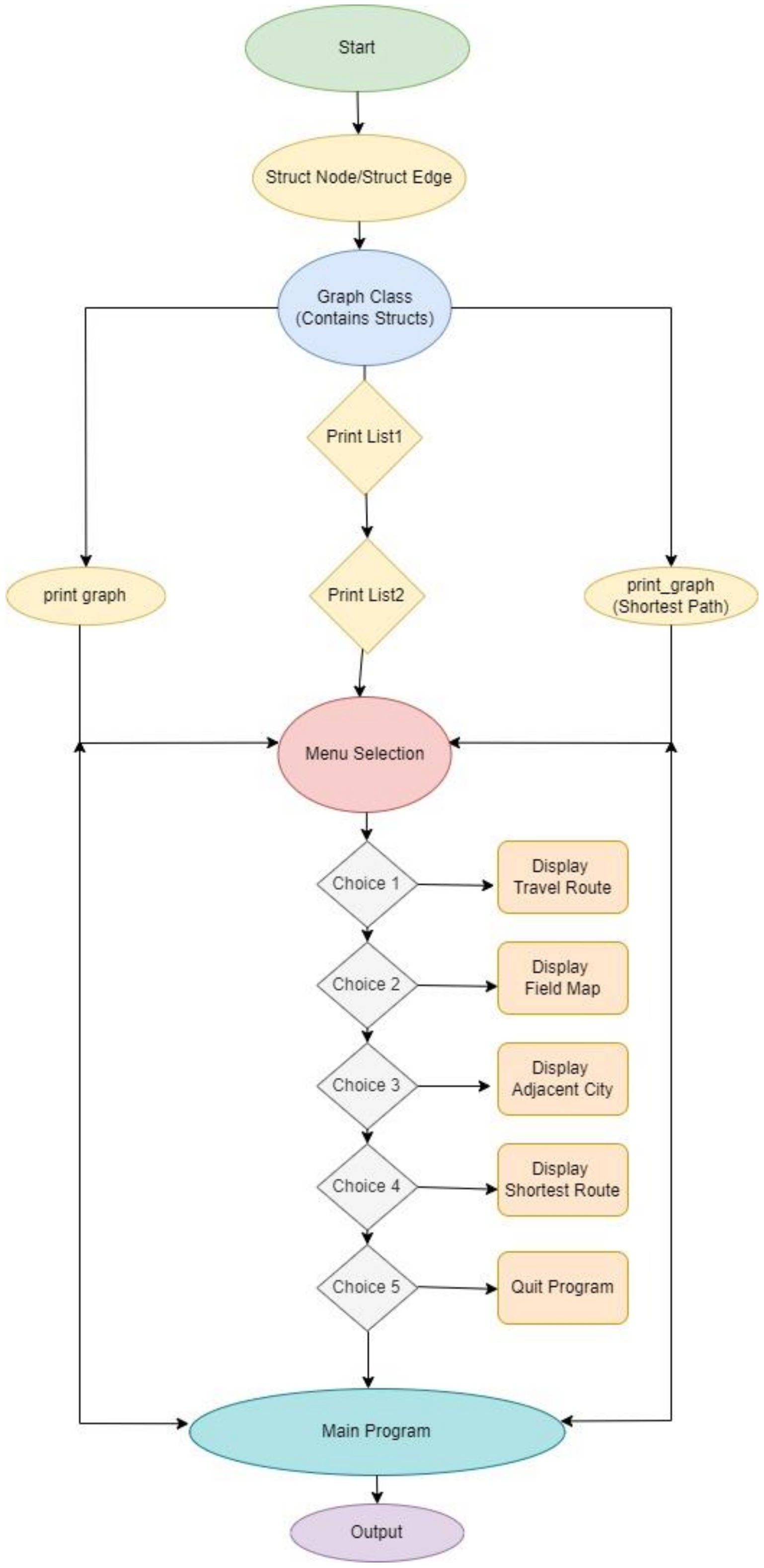
**Adjacency List Between Cities:**
0(Riverside) is adjacent to 1 (MoVal), 2(Perris),3 (Hemet
1(MoVal) is adjacent to 0(Riverside), 2 (Perris), 3(Hemet)
2(Perris) is adjacent to 0(Riverside), 1(MoVal), 3(Hemet)
3(Hemet) is adjacent to 0(Riverside), 1(MoVal), 2(Perris)

**Adjacency Matric (With Distance in Miles):**

| Vertex(Cities | Riverside | Moreno Valley | Perris | Hemet |
|---|---|---|---|---|
| Riverside | 0 | 15 | 24 | 33 |
| Moreno Valley | 16 | 0 | 18 | 26 |
| Perris | 24 | 18 | 0 | 30 |
| Hemet | 33 | 26 | 30 | 0 |

## Algorithm Diagram (Base Level)

Below is a flowchart of how the major components of the program are connected from start to finish. Major intricacies can be found when opening the .cpp file.


Start
Struct Node/Struct Edge
Graph Class (Contains Structs)
Print List1
print graph
Print List2
print_graph (Shortest Path)
Menu Selection
Choice 1
Display Travel Route
Choice 2
Display Field Map
Choice 3
Display Adjacent City
Choice 4
Display Shortest Route
Choice 5
Quit Program
Main Program
Output