Traveling Salesman Problem: Closest Edge Insertion Heuristic

Jacob Taylor Cassady
Computer Engineering & Computer Science
Speed School of Engineering
University of Louisville, USA
jtcass01@louisville.edu

1 INTRODUCTION

The Traveling Salesman Problem (TSP) is a well-known non-deterministic polynomial-time hard problem that has been studied within mathematics since the 1930s. The "salesman" is given a list of cities with their locations and is asked the shortest route to travel to each city once and then return to the starting point. A program was developed using Python 3.7 and accompanying 3rd party libraries: NumPy, Pandas, and matplotlib to determine the shortest path.

2 APPROACH

The approach taken to solving the TSP was to use a closest-edge insertion heuristic. Development of the algorithm was aided by a graphical user interface (GUI). Throughout this document cities from TSP will be referred to as "vertices" and the route between a pair of vertices as an "edge."

2.1 ALGORITHM

The algorithm implemented has two main parts to its forward progression. First off it chooses a new vertex to be inserted into the route. Secondly, the route "lasso's" the vertex to update the route with the new vertex in the most optimal way. A graphical representation of this lasso technique can be found in **Figure 1**.

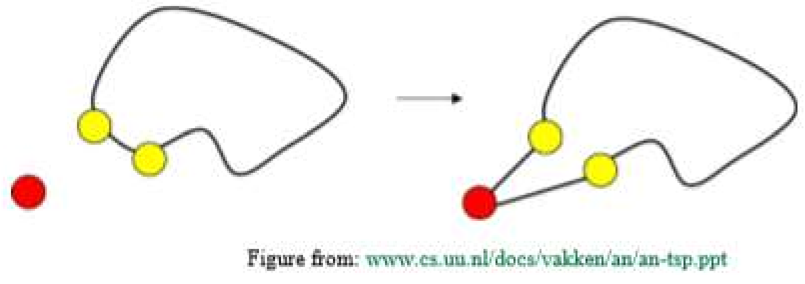


Figure 1 : Vertex Lasso Example

2.1.1 Choose Next Vertex

The first starting vertex for the algorithm is chosen by the user. For this paper's purposes, starting vertices were chosen at random. To choose the next vertex to be entered, all unvisited vertices are iterated over. For each unvisited vertex, the minimum distance to the set of visited vertices and edges is calculated. This minimum distance is found by iterating over the edges and vertices within the route and calculating each. After calculating the minimum distance to the route for each unvisited vertex, the closest unvisited vertex is chosen. For use during the next algorithm, the closest item to the vertex is also returned rather that be a vertex within the route or an edge within the route. For the exact implementation of the choose_next_vertex function, please refer to **Figure 5** in the Appendix of this paper.

2.1.2 Lasso Vertex

The Lasso function implements the addition and deletion of edges to optimally rope an additional vertex as shown in **Figure 1** above. The lasso function takes two parameters: a vertex and the closest item to the vertex. This closest item is assumed to be within the route and of type Edge or Vertex. If the closest item is of type Vertex, the route uses a simple goto() function to 'lasso' the new vertex onto the end of the route. On the other hand, if the closest item is of type Edge, each insertion possibility with respect to the edge is calculated and the minimum distance path is chosen. These paths include the vertex being inserted before the edge, in the middle of the edge, and after the edge within the graph. After the minimum path is determined, Edge objects are created to represent the new path, previous unneeded edges are deleted, and the new Edge objects are inserted. For the exact implementation of the lasso function, please refer to **Figure 6** in the Appendix of this paper.

2.2 GRAPHICAL USER INTERFACE

To aid in the development of the algorithm, a GUI was developed to track the updates to the program's state. The GUI has 5 command buttons and a graph display. The command buttons include: "Step Forward", "Step Backward", "Run Simulation", "Finish Simulation", and "Show All Edges" or "Show Current Route". For reference to the GUI during different states of the algorithm's execution, please refer to **Figures 11-17** in the Appendix of this paper.

2.2.1 Step Forward

The step forward button calls step_forward on the algorithm. In the case of the greedy closest edge algorithm, it finds the nearest vertex to the route and lassos it.

2.2.2 Step Backward

The step backward button calls step_backward on the algorithm. It reverses the previous step forward operation.

2.2.3 Run Simulation

The run simulation button iterates through the algorithm's step_forward function until the algorithm raises a True done flag. It shows each step in the simulation as it occurs.

2.2.4 Finish Simulation

The finish simulation button iterates through the algorithm's step_forward function until the algorithm raises a True done flag. It does not show each step in the simulation as it occurs, only the final product.

2.2.5 Show All Edges / Show Current Route

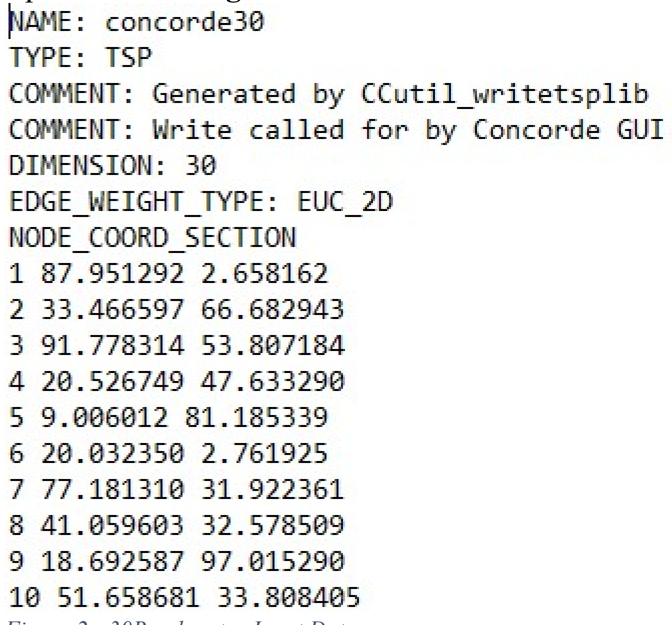
This button toggles between showing the graph with all edges connected to showing only the vertices and the route traveled. The text on the button toggles to match its use case.

3 RESULTS

The Greedy algorithm with a Closest Edge Insertion Heuristic was successfully implemented. The algorithm produces a route that visits each city while minimizing the route distance. The algorithm did not have any issues running with only 4 GB of RAM on datasets with up to 40 cities. No mitigation techniques were needed to reduce program memory usage or improve runtime efficiency.

3.1 DATA

The algorithm was tested using four different datasets each generated with a number of randomly located cities. The files tested had 11, 12, 30, and 40 cities. Within the datafile, cities are enumerated, and x and y coordinates are provided. The input data was formatted like the example shown in **Figure 2** below.



```
NAME: concorde30
TYPE: TSP
COMMENT: Generated by CCutil_writetsplib
COMMENT: Write called for by Concorde GUI
DIMENSION: 30
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
5 9.006012 81.185339
6 20.032350 2.761925
7 77.181310 31.922361
8 41.059603 32.578509
9 18.692587 97.015290
10 51.658681 33.808405
```

Figure 2 : 30Random.tsp Input Data

3.2 RESULTS

The results from processing the different files using the previously described algorithm can be found in **Table 1** shown below.

Filename	Distance traveled	Algorithm Runtime	Route Order	Insert order
Random30.tsp	509.213	0.11372s	8, 10, 15, 30, 1, 24, 7, 17, 21, 3, 19, 28, 23, 16, 11, 12, 2, 14, 22, 27, 9, 5, 26, 18, 4, 29, 13, 25, 6, 20, 8	8, 10, 16, 11, 12, 18, 4, 29, 13, 25, 6, 20, 2, 14, 26, 5, 9, 27, 22, 17, 7, 30, 24, 15, 1, 23, 19, 3, 21, 28, 8
Random40.tsp	625.171	0.27526s	1, 24, 30, 7, 17, 40, 10, 16, 23, 21, 3, 19, 32, 28, 34, 36, 37, 11, 12, 18, 2, 14, 22, 27, 9, 5, 26, 35, 4, 33, 31, 13, 25, 6, 20, 29, 38, 8, 39, 15, 1	1, 24, 30, 15, 7, 17, 40, 10, 16, 11, 12, 18, 8, 39, 35, 4, 33, 38, 29, 31, 13, 25, 6, 20, 37, 26, 14, 2, 23, 34, 19, 36, 28, 32…
Random11.tsp	252.684	0.007s	1, 6, 10, 11, 8, 9, 7, 5, 3, 4, 2, 1	…
Random12.tsp	66.085	0.00797s	1, 8, 2, 3, 12, 4, 9, 5, 10, 6, 7, 11, 1	…

Table 1 : Algorithm Performance Per Test File

The table above shows the algorithm successfully completed even 40 cities in a little over half a second. Although this is over twice as long as it took to complete 30 cities, it is still relatively quick given the number of possibilities a brute force algorithm would have to sift through for the same dataset.

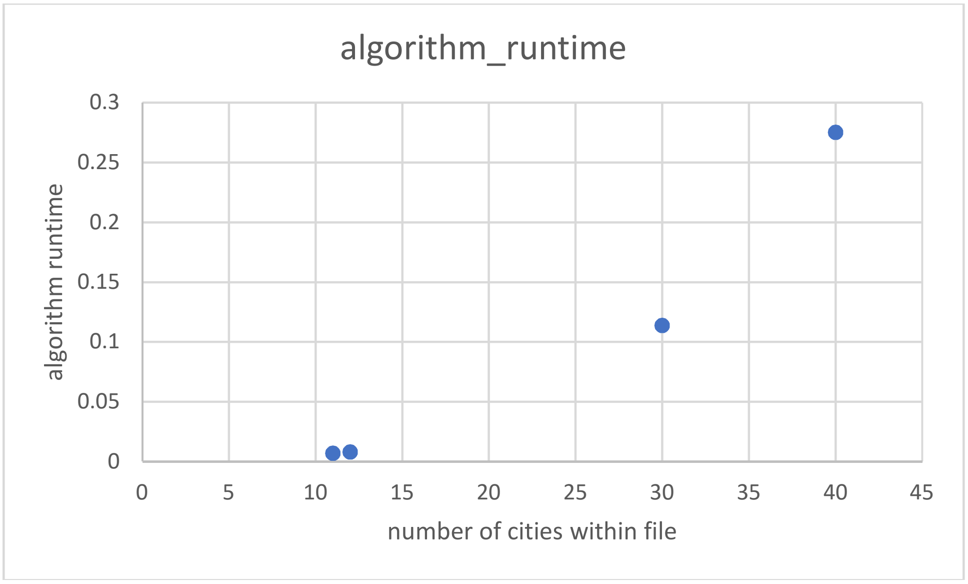


Figure 3 : Algorithm Runtime v. Number of Cities Within File

4 Discussion

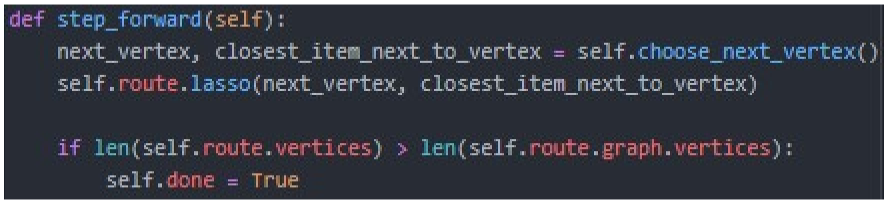
A graph of the number of cities within a file versus the algorithm runtime was presented in **Figure 3** above. As you can see it still looks like it is exponentially shaped but with a less steep curve than brute force. This shows the greedy algorithm using the closest edge insertion heuristic can handle growing datasets much better than the brute force approach while still providing an answer that lacks crossing edges.

Several equations exist for computing the distance between a line, given two points, and a third point outside of the line. A problem found when implementing this algorithm is that many of those equations assume the line extends infinitely past the segment bound by the two points including the equation described here. This made understanding the behavior of the program and calculations within challenging at first.

5 References

Wikipedia, Traveling Salesman Problem - https://en.wikipedia.org/wiki/Travelling_salesman_problem#History
NumPy Documentation - https://docs.scipy.org/doc/
Pandas Documentation - https://pandas.pydata.org/pandas-docs/stable/
Matplotlib Documentation - https://matplotlib.org/3.1.1/contents.html
Minimum Distance Between a Point and a Line - http://paulbourke.net/geometry/pointlineplane/

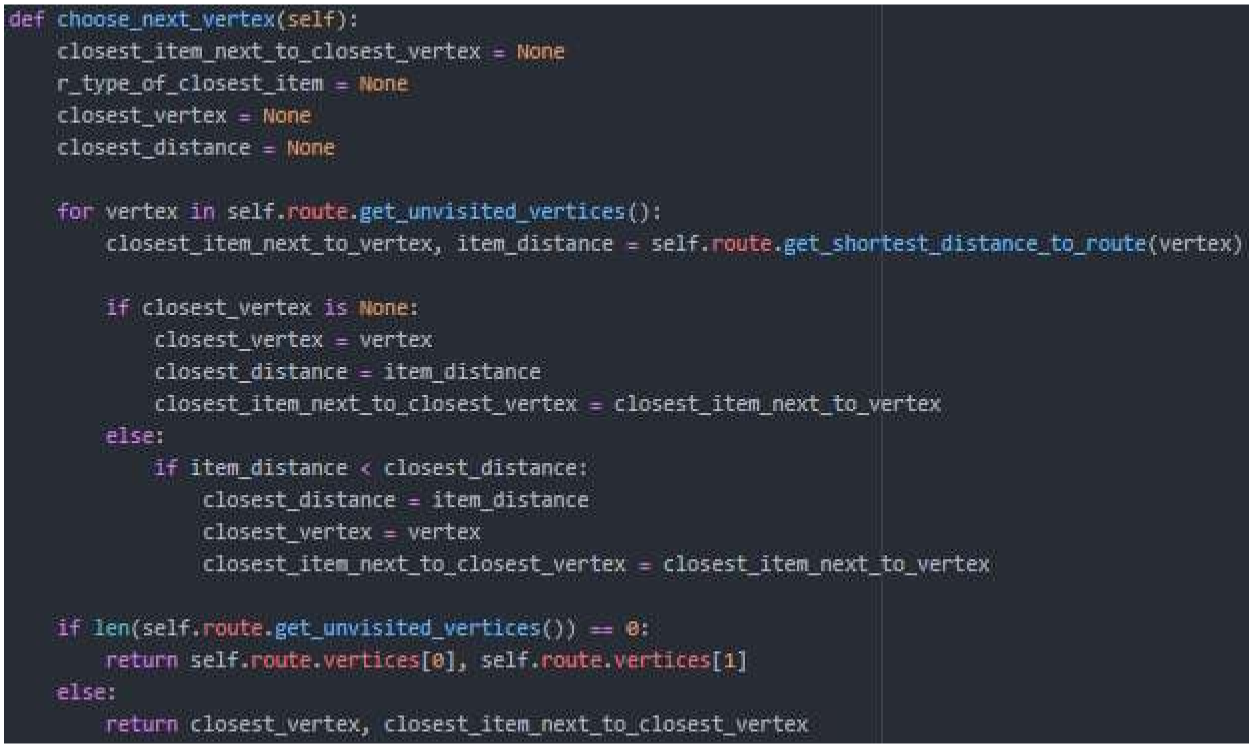
6 APPENDIX



```
def step_forward(self):
    next_vertex, closest_item_next_to_vertex = self.choose_next_vertex()
    self.route.lasso(next_vertex, closest_item_next_to_vertex)

    if len(self.route.vertices) > len(self.route.graph.vertices):
        self.done = True
```

Figure 4: TravelingSalesman.GreedyAlgorithm.step_forward()



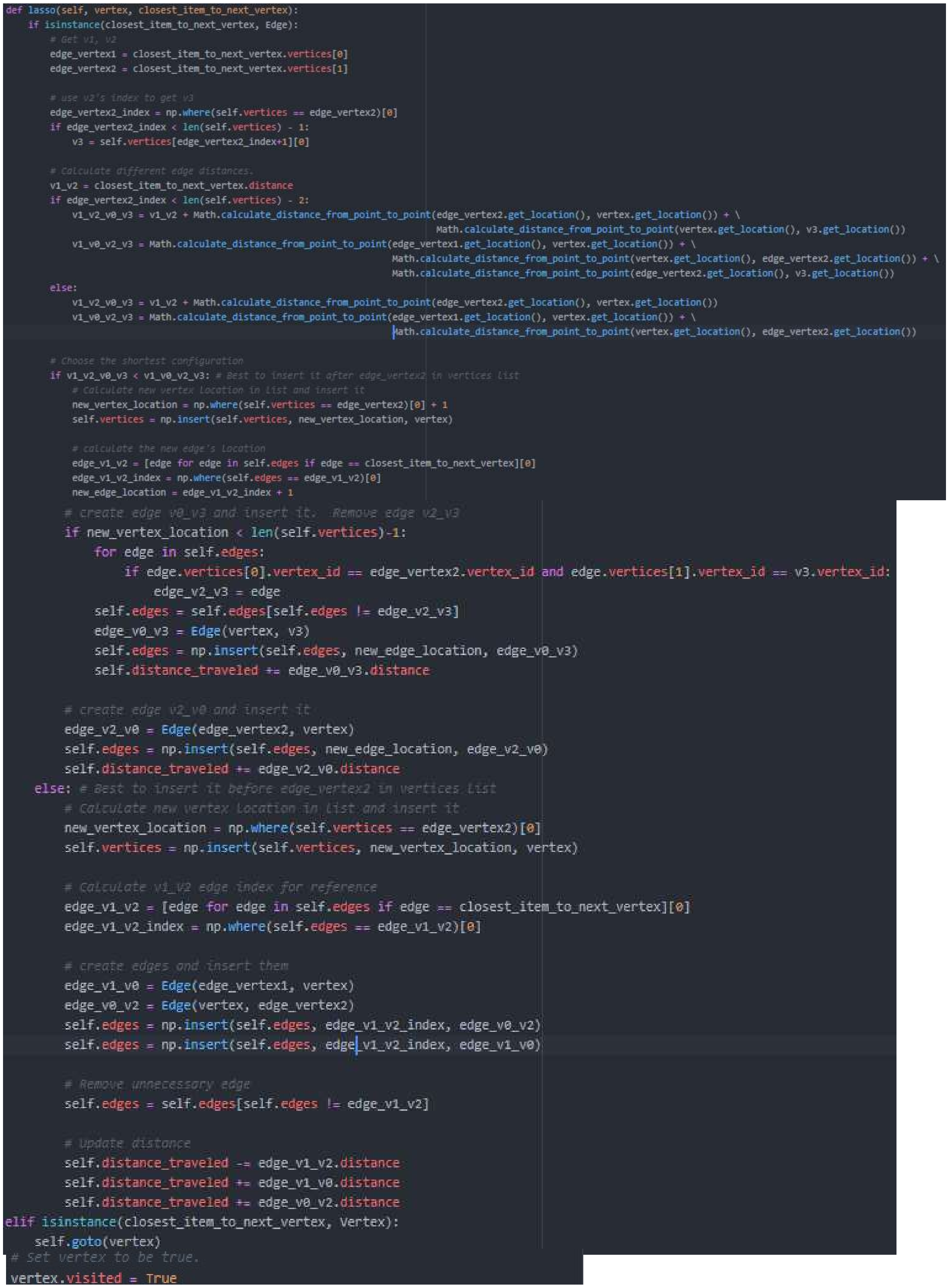
```
def choose_next_vertex(self):
    closest_item_next_to_closest_vertex = None
    r_type_of_closest_item = None
    closest_vertex = None
    closest_distance = None

    for vertex in self.route.get_unvisited_vertices():
        closest_item_next_to_vertex, item_distance = self.route.get_shortest_distance_to_route(vertex)

        if closest_vertex is None:
            closest_vertex = vertex
            closest_distance = item_distance
            closest_item_next_to_closest_vertex = closest_item_next_to_vertex
        else:
            if item_distance < closest_distance:
                closest_distance = item_distance
                closest_vertex = vertex
                closest_item_next_to_closest_vertex = closest_item_next_to_vertex

    if len(self.route.get_unvisited_vertices()) == 0:
        return self.route.vertices[0], self.route.vertices[1]
    else:
        return closest_vertex, closest_item_next_to_closest_vertex
```

Figure 5: TravelingSalesmane.GreedyAlgorithm.choose_next_vertex()



```python
def lasso(self, vertex, closest_item_to_next_vertex):
    if isinstance(closest_item_to_next_vertex, Edge):
        # Get v1, v2
        edge_vertex1 = closest_item_to_next_vertex.vertices[0]
        edge_vertex2 = closest_item_to_next_vertex.vertices[1]

        # use v2's index to get v3
        edge_vertex2_index = np.where(self.vertices == edge_vertex2)[0]
        if edge_vertex2_index < len(self.vertices) - 1:
            v3 = self.vertices[edge_vertex2_index+1][0]

        # Calculate different edge distances.
        v1_v2 = closest_item_to_next_vertex.distance
        if edge_vertex2_index < len(self.vertices) - 2:
            v1_v2_v0_v3 = v1_v2 + Math.calculate_distance_from_point_to_point(edge_vertex2.get_location(), vertex.get_location()) + \
                                  Math.calculate_distance_from_point_to_point(vertex.get_location(), v3.get_location())
            v1_v0_v2_v3 = Math.calculate_distance_from_point_to_point(edge_vertex1.get_location(), vertex.get_location()) + \
                          Math.calculate_distance_from_point_to_point(vertex.get_location(), edge_vertex2.get_location()) + \
                          Math.calculate_distance_from_point_to_point(edge_vertex2.get_location(), v3.get_location())
        else:
            v1_v2_v0_v3 = v1_v2 + Math.calculate_distance_from_point_to_point(edge_vertex2.get_location(), vertex.get_location())
            v1_v0_v2_v3 = Math.calculate_distance_from_point_to_point(edge_vertex1.get_location(), vertex.get_location()) + \
                          Math.calculate_distance_from_point_to_point(vertex.get_location(), edge_vertex2.get_location())

        # Choose the shortest configuration
        if v1_v2_v0_v3 < v1_v0_v2_v3: # Best to insert it after edge_vertex2 in vertices list
            # Calculate new vertex location in list and insert it
            new_vertex_location = np.where(self.vertices == edge_vertex2)[0] + 1
            self.vertices = np.insert(self.vertices, new_vertex_location, vertex)

            # calculate the new edge's location
            edge_v1_v2 = [edge for edge in self.edges if edge == closest_item_to_next_vertex][0]
            edge_v1_v2_index = np.where(self.edges == edge_v1_v2)[0]
            new_edge_location = edge_v1_v2_index + 1
            # create edge v0_v3 and insert it.  Remove edge v2_v3
            if new_vertex_location < len(self.vertices)-1:
                for edge in self.edges:
                    if edge.vertices[0].vertex_id == edge_vertex2.vertex_id and edge.vertices[1].vertex_id == v3.vertex_id:
                        edge_v2_v3 = edge
                self.edges = self.edges[self.edges != edge_v2_v3]
                edge_v0_v3 = Edge(vertex, v3)
                self.edges = np.insert(self.edges, new_edge_location, edge_v0_v3)
                self.distance_traveled += edge_v0_v3.distance

            # create edge v2_v0 and insert it
            edge_v2_v0 = Edge(edge_vertex2, vertex)
            self.edges = np.insert(self.edges, new_edge_location, edge_v2_v0)
            self.distance_traveled += edge_v2_v0.distance
        else: # Best to insert it before edge_vertex2 in vertices list
            # Calculate new vertex location in list and insert it
            new_vertex_location = np.where(self.vertices == edge_vertex2)[0]
            self.vertices = np.insert(self.vertices, new_vertex_location, vertex)

            # Calculate v1_v2 edge index for reference
            edge_v1_v2 = [edge for edge in self.edges if edge == closest_item_to_next_vertex][0]
            edge_v1_v2_index = np.where(self.edges == edge_v1_v2)[0]

            # create edges and insert them
            edge_v1_v0 = Edge(edge_vertex1, vertex)
            edge_v0_v2 = Edge(vertex, edge_vertex2)
            self.edges = np.insert(self.edges, edge_v1_v2_index, edge_v0_v2)
            self.edges = np.insert(self.edges, edge_v1_v2_index, edge_v1_v0)

            # Remove unnecessary edge
            self.edges = self.edges[self.edges != edge_v1_v2]

            # Update distance
            self.distance_traveled -= edge_v1_v2.distance
            self.distance_traveled += edge_v1_v0.distance
            self.distance_traveled += edge_v0_v2.distance
    elif isinstance(closest_item_to_next_vertex, Vertex):
        self.goto(vertex)
    # Set vertex to be true.
    vertex.visited = True
```

Figure 6 : Route.lasso(vertex, closest_item_to_vertex)

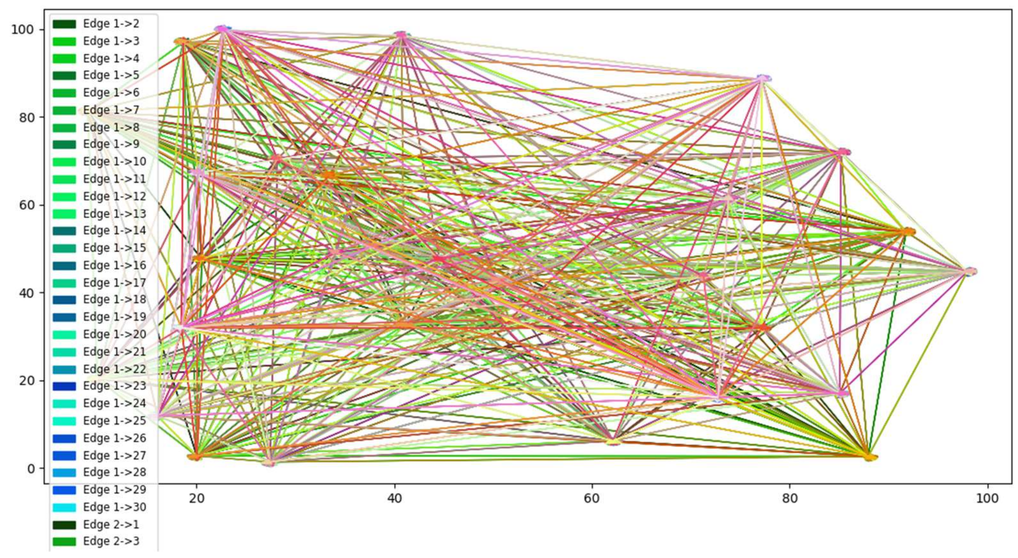


Figure 7 : Random30.tsp Input

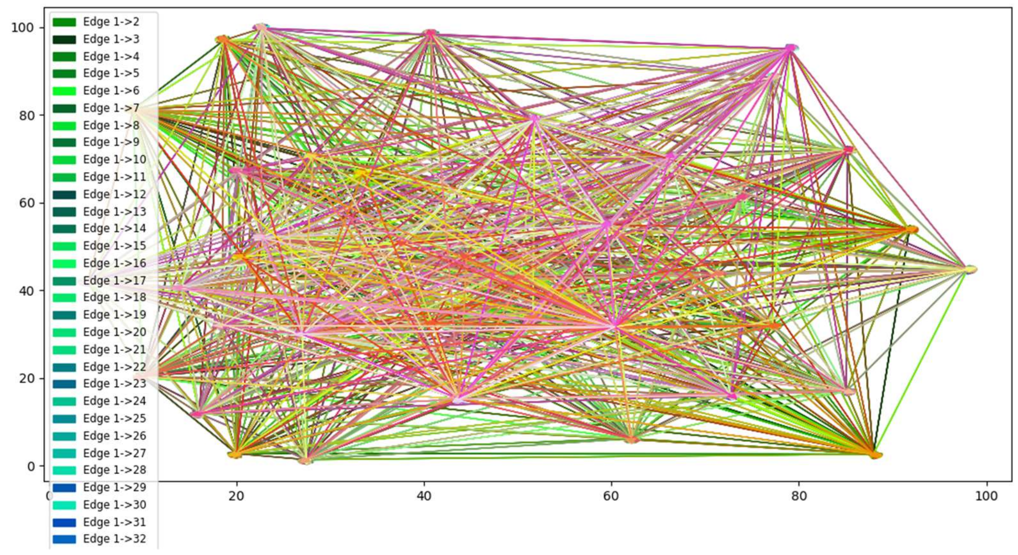


Figure 8 : Random40.tsp Input

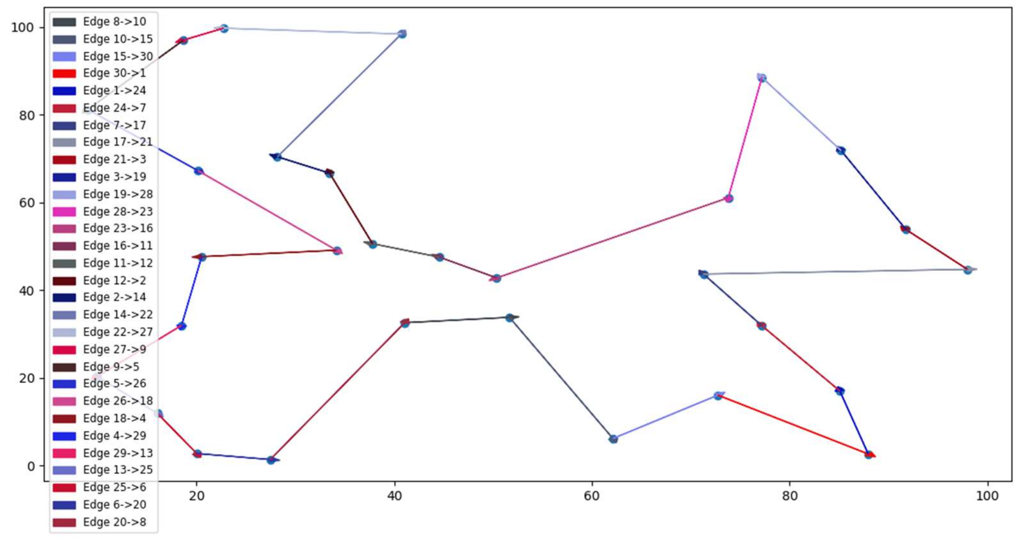


Figure 9 : Random30.tsp Output

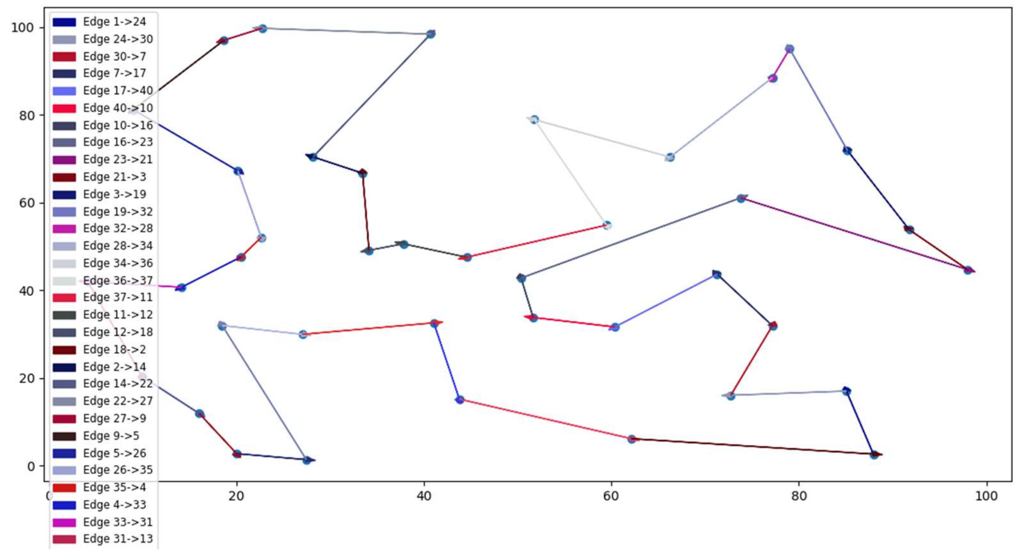


Figure 10 : Random40.tsp Output

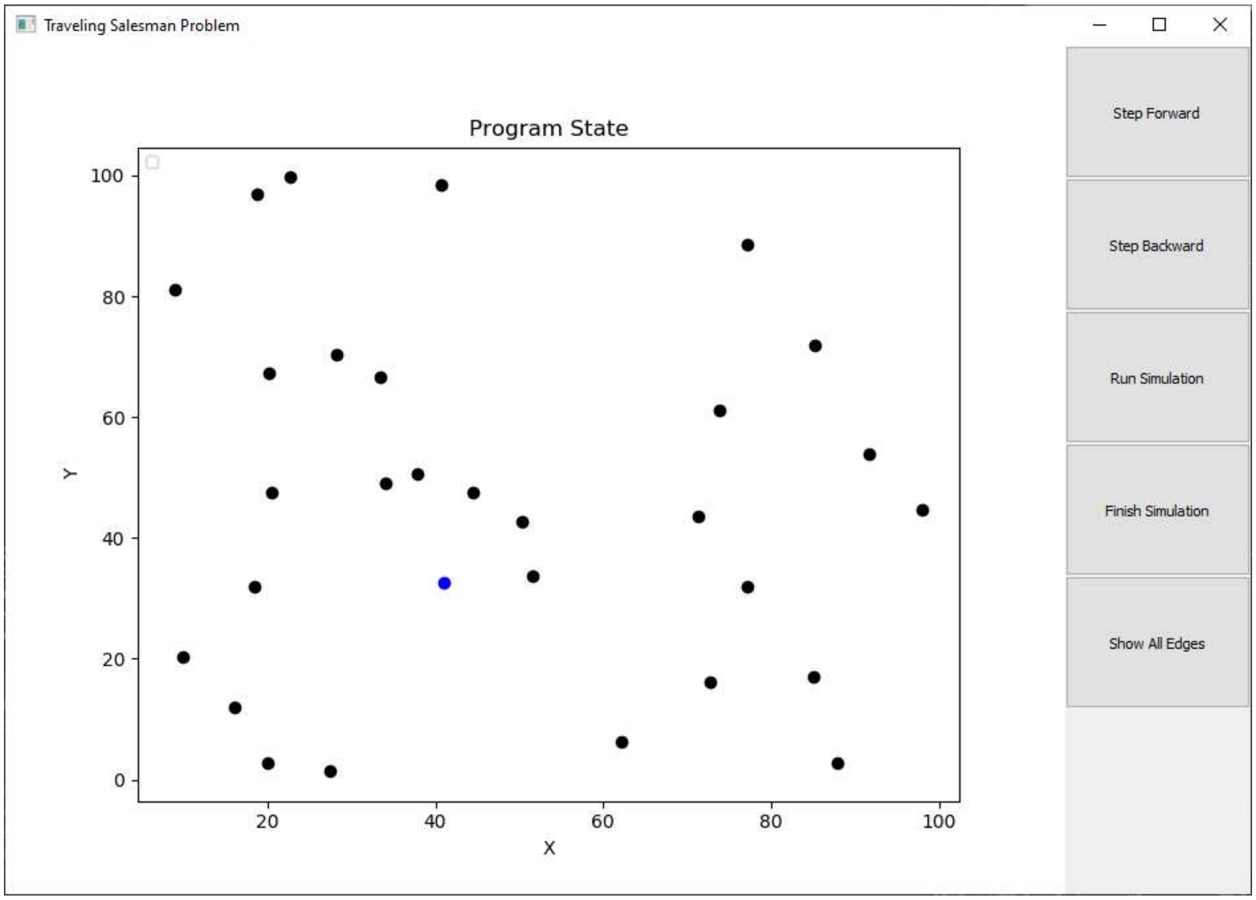


Figure 11 : Program State 0 [Random30.tsp]

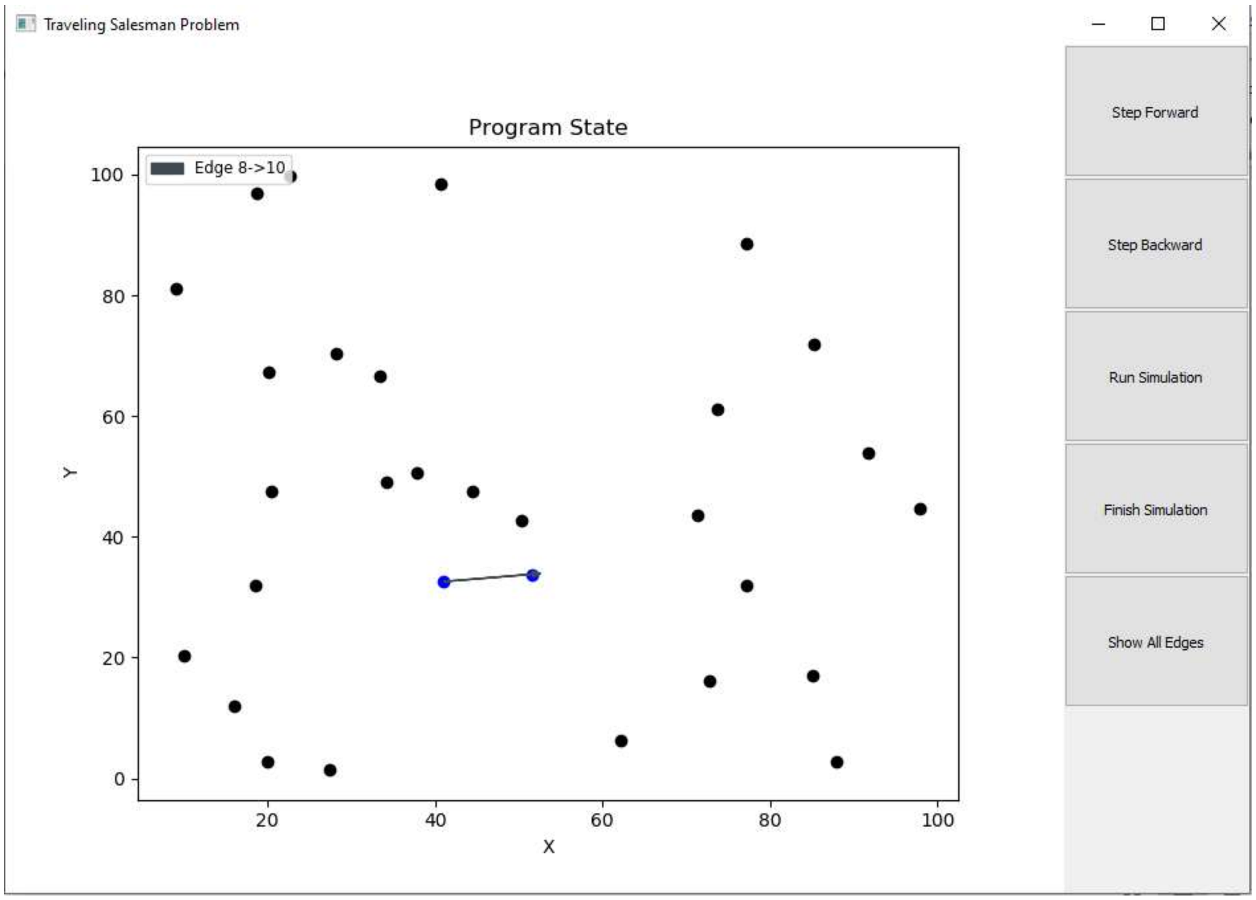


Figure 12 : Program State 1 [Random30.tsp]

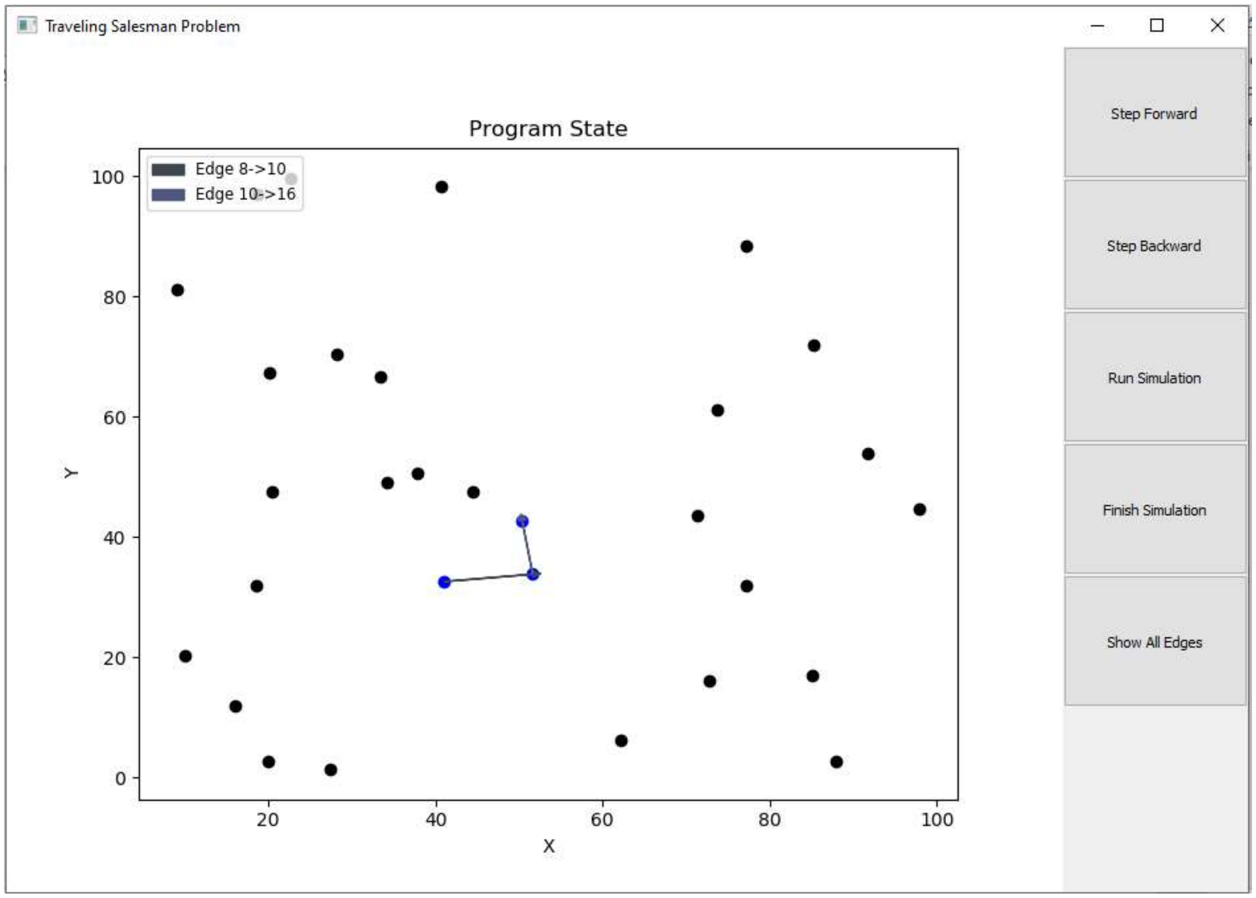


Figure 13 : Program State 3 [Random30.tsp]

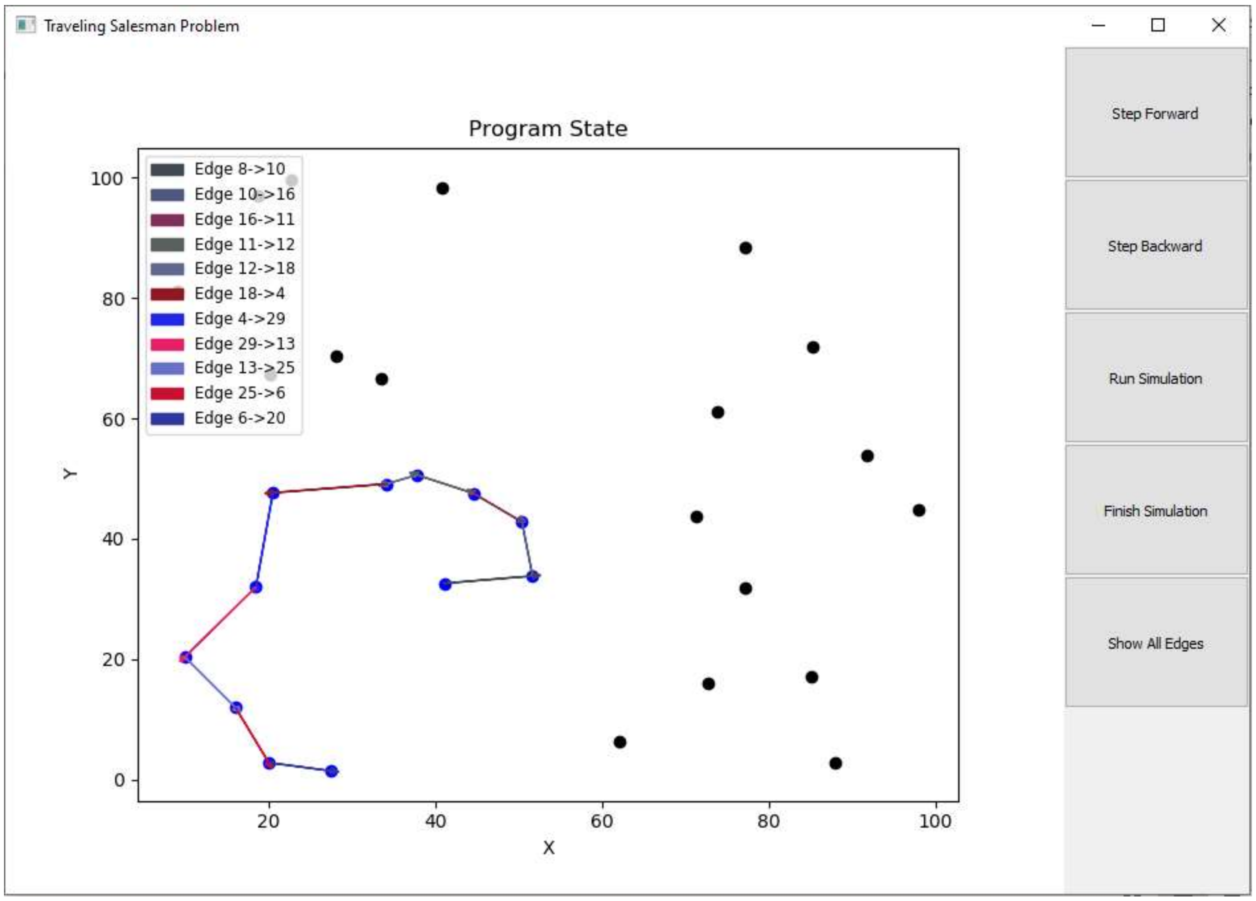


Figure 14 : Program State 11 [Random30.tsp]

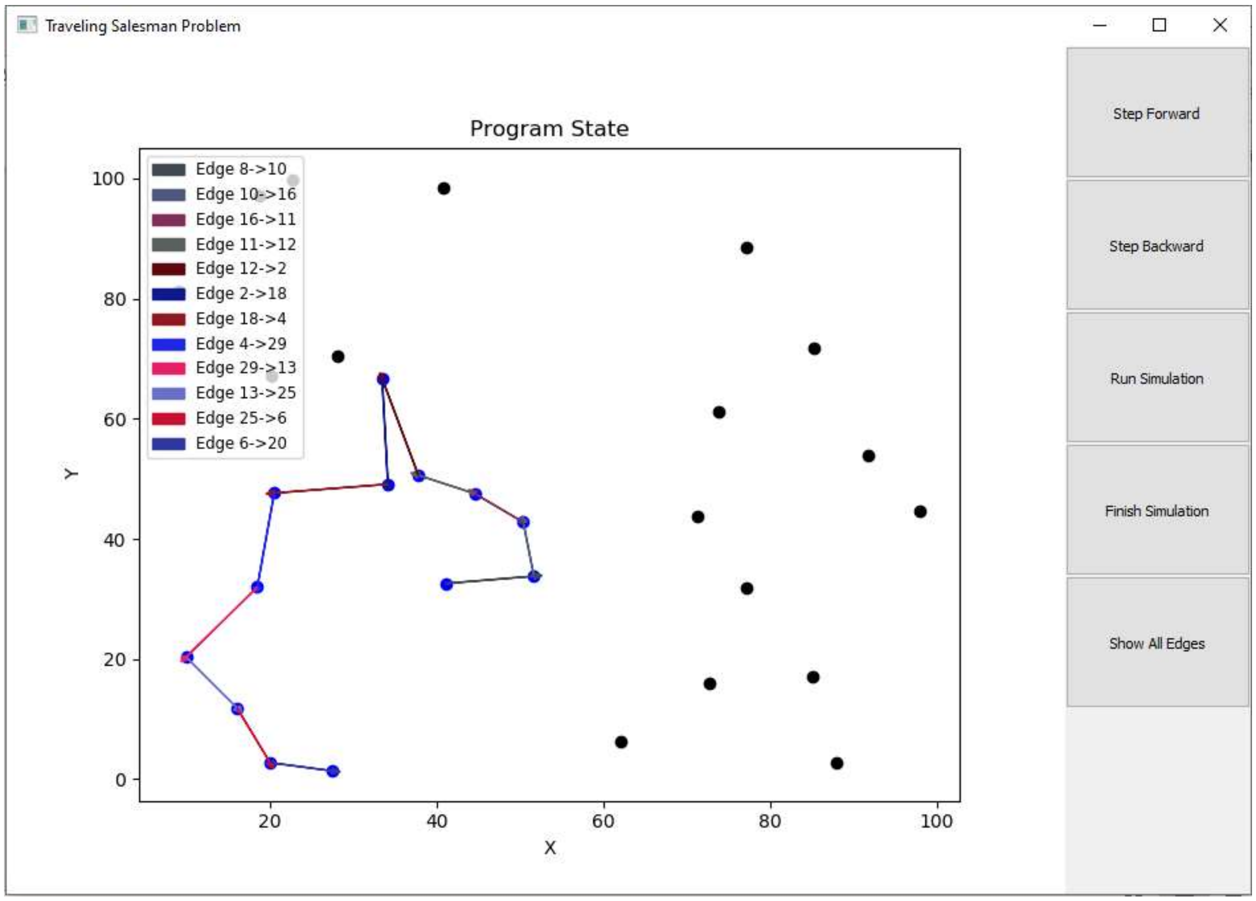


Figure 15 : Program State 12 [Random30.tsp]

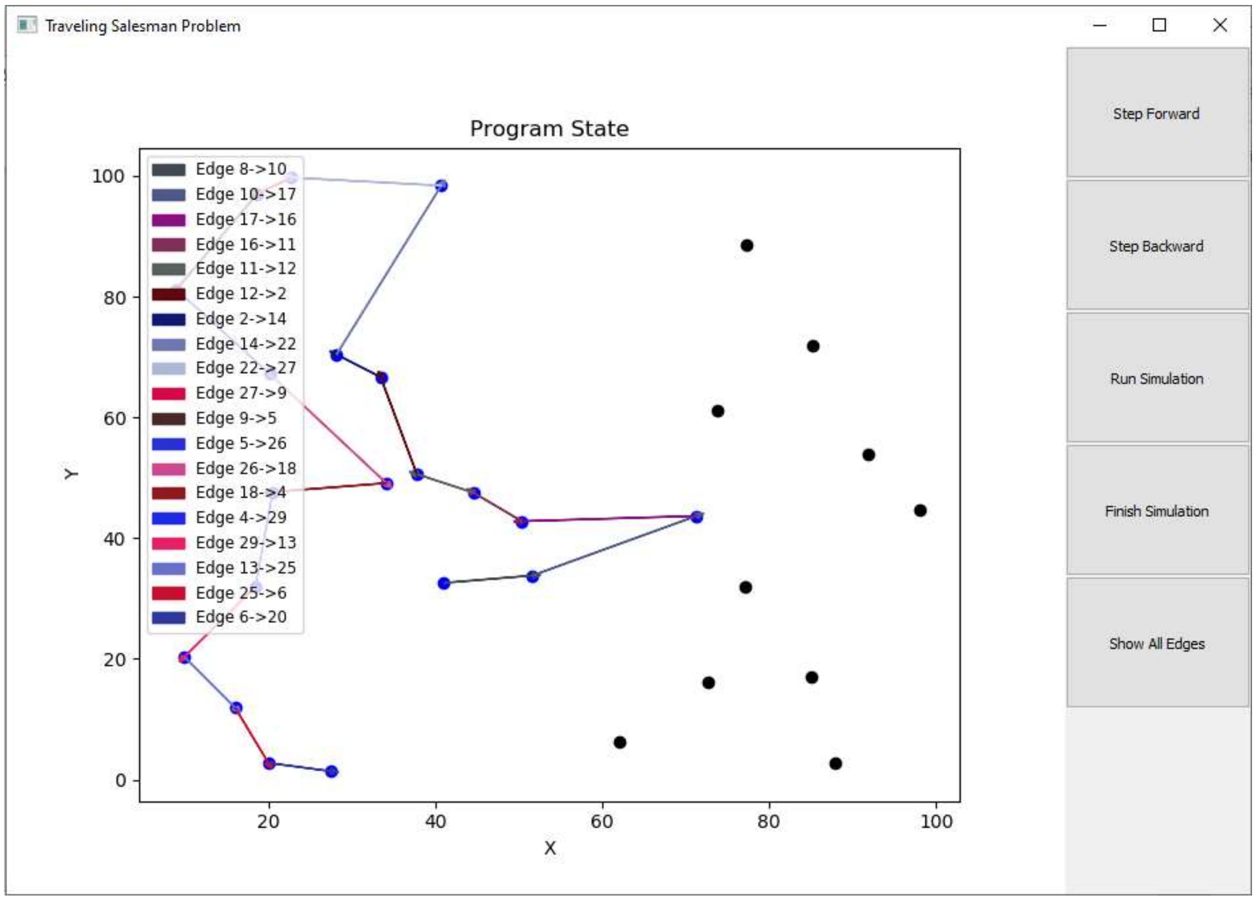


Figure 16 : Program State 19 [Random30.tsp]

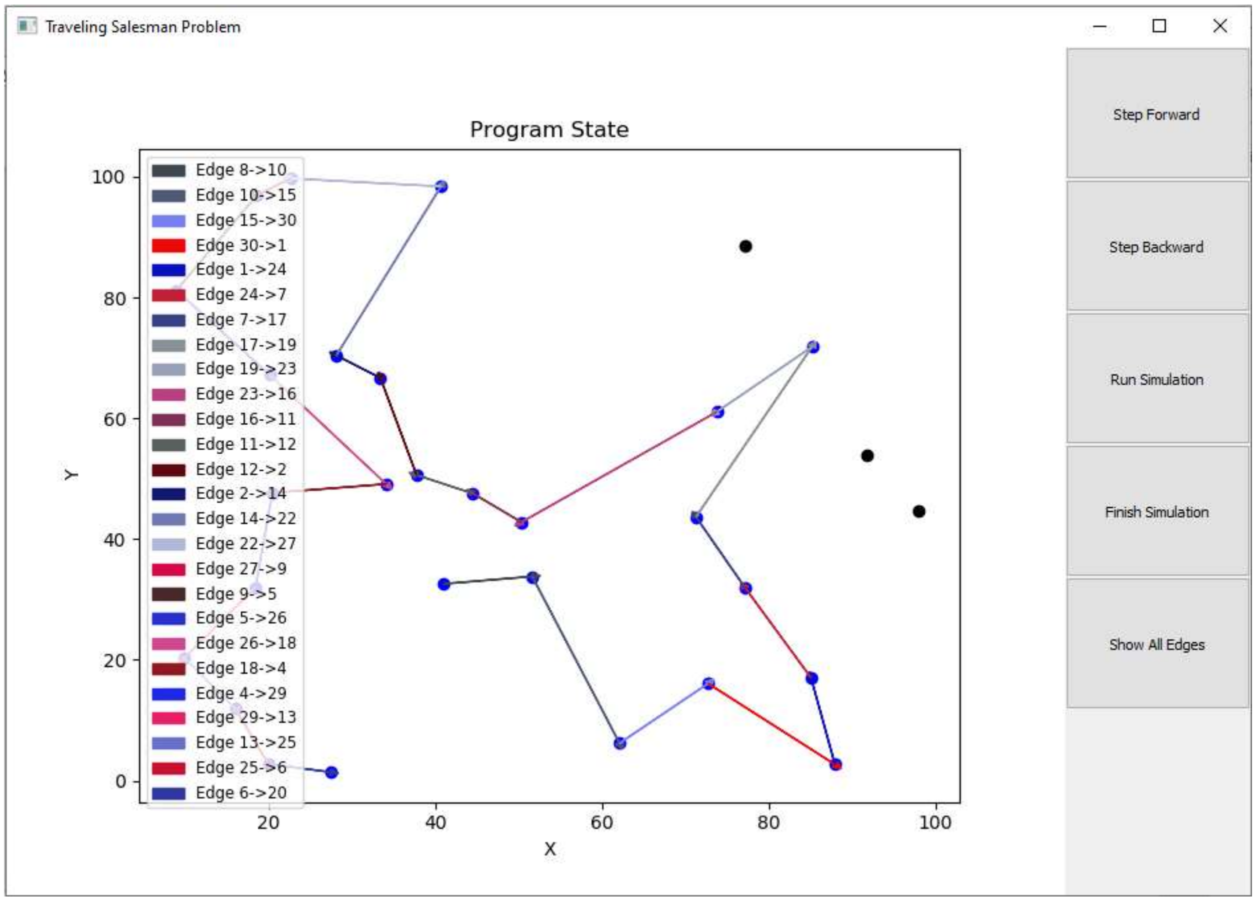


Figure 17 : Program State 26 [Random30.tsp]