Traveling Salesman Problem: Brute Force

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 for solving the TSP was brute force. Throughout this document cities will be referred to as "vertices" and the route between the vertices as "edges." A simplified version of the algorithm designed is shown in Figure 1 of the Appendix.

A "Route" is initialized at the starting vertex. A recursive function is then used to take all available routes from that vertex with each stop at an adjacent vertex making a deeper call to the same recursive function. Once the Graph's vertices have been visited, the route returns to the starting vertex and is kept until all routes have been generated. Finally, the routes are compared to find the route with minimum distance traveled. For graphs with less than 10 vertices, a list of routes is kept in RAM and quickly compared at the end. Since the amount of routes grows the algorithms runtime factorially, the routes need to be moved to hard disk at 10 or more and compared after being generated.

3. Results

The algorithm successfully produces a minimum route; although, there are some drawbacks. Since the algorithm is recursive in nature and the TSP grows with factorial complexity, memory use and computational cost becomes an issue quickly as you increase the number of cities. Mitigation techniques for the memory burden are described in the approach section of this document.

3.1 Data

The algorithm was tested using 9 different datasets each with a different number of cities starting at 4 and ending at 12. Cities are enumerated and x and y coordinates are provided. The input data was formatted like the example below:

```
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
```

3.2 Results

Please refer to the following table for the optimal paths computed for each input file (note: filenames tagged with * denotes the routes were written to disk instead of being kept in RAM during generation):

Filename	Number of Cities	City Order	Distance Traveled	Algorithm Runtime
Random4.tsp	4	0, 3, 1, 2, 0	215.086	0.0029s
Random5.tsp	5	0, 1, 4, 2, 3, 0	139.133	0.0259s
Random6.tsp	6	0, 1, 2, 3, 4, 5, 0	118.969	0.0858s
Random7.tsp	7	0, 1, 6, 2, 5, 4, 3, 0	63.863	0.5166s
Random8.tsp	8	0, 5, 7, 3, 4, 1, 2, 6, 0	310.982	3.9385s
Random9.tsp	9	0, 6, 5, 2, 4, 1, 8, 3, 7, 0	131.028	35.0865s
Random10.tsp*	10	0, 1, 6, 5, 7, 4, 8, 9, 3, 2, 0	106.786	449.1541s
Random11.tsp*	11	0, 5, 9, 10, 7, 8, 6, 4, 2, 3, 1, 0	252.684	4768.5063s
Random12.tsp*	12	0, 7, 1, 2, 11, 3, 8, 4, 9, 5, 6, 10, 0	66.085	57517.0233s

A graph of the number of cities within a file vs. the average algorithm runtime is shown in Figure 2 of the Appendix. Graph visualizations were created for each file's input/output to ensure the results were reasonable. These can be found in the Appendix as well, Figures 3 through 20.

4. Discussion

As you can see from the graph within the appendix and the table above, the algorithm's runtime grows very quickly as the number of cities increase. This is a consequence of the algorithm's factorial complexity. Runtime could be further reduced by using the logging technique mentioned in the Approach section of this document while spawning a separate process for each route after it visits its second city. Depending on the number of processor cores available, you could speed up the runtime by a factor equal to the number of CPU cores utilized with some limitations for datasets with more cities than available CPU cores.

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

6. Appendix

```python
def brute_force_solution(graph, current_vertex_id=0, distance_traveled = 0):
    # Recursive function for trying all adjacent vertices.
    def try_all_open_routes_from_current_route(route):
        # Initialize Routes to keep track of all attempted routes.
        routes = np.array([])
        # Start at the current vertex id location
        current_vertex = route.graph.vertices[current_vertex_id]

        # For each adjacent vertex that has not been visited
        for adjacent_vertex in current_vertex.get_unvisited_adjacent_vertex_ids():
            # copy the route so far
            new_route = deepcopy(route)
            # goto the current adjacent_vertex
            new_route.goto(adjacent_vertex.vertex_id)

            # if all vertices have been visisted
            if(new_route.graph.finished()):
                # goto the starting point
                new_route.goto(current_vertex_id)
                # append the route to the list of completed routes
                routes = np.concatenate((routes, new_route), axis=None)
            else: # if not,
                # Recall the recursive function using the updated route.
                routes = np.concatenate((routes, try_all_open_routes_from_current_route(new_route)), axis=None)

        # After all adjacent vertices have been visisted recursively, return the list of routes
        return routes

    # Initialize the route
    route = Route(list([]), graph)
    # goto the current vertex id
    route.goto(current_vertex_id)

    # Initialize a list of routes
    routes = np.array([])

    # Recursively try all open routes from the current route, advancing when possible.
    routes = np.concatenate((routes, try_all_open_routes_from_current_route(route)), axis=None)

    # Identify the route with minimum distance traveled
    return min(routes)
```

Figure 1: Brute Force Algorithm

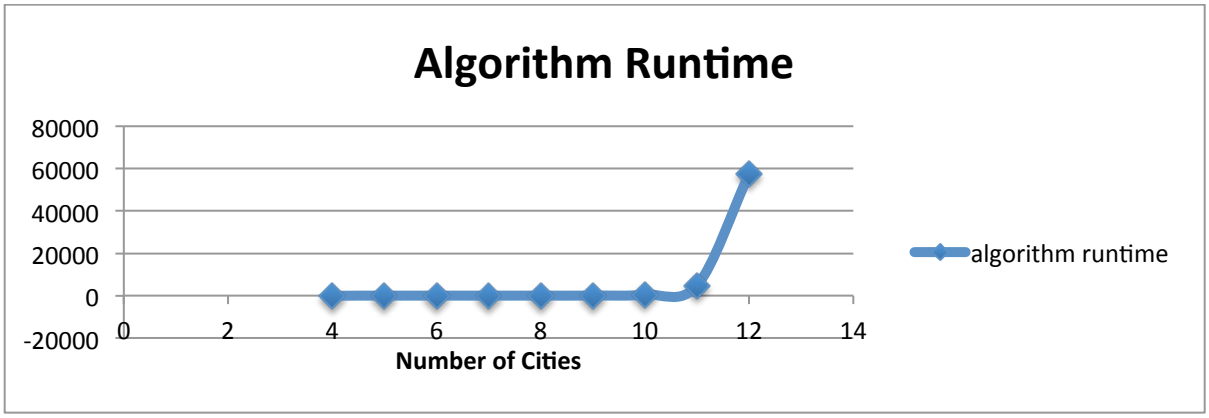

Figure 2: Average Algorithm Runtime Graph

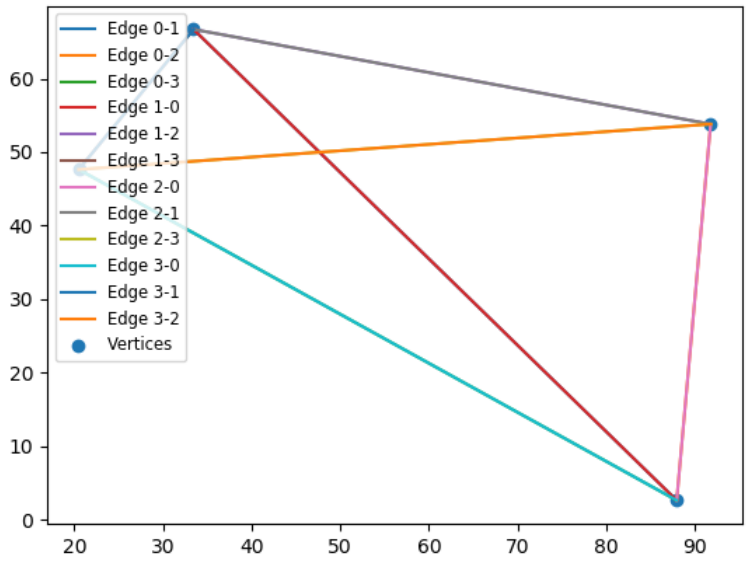

Figure 3: Random4.tsp Input

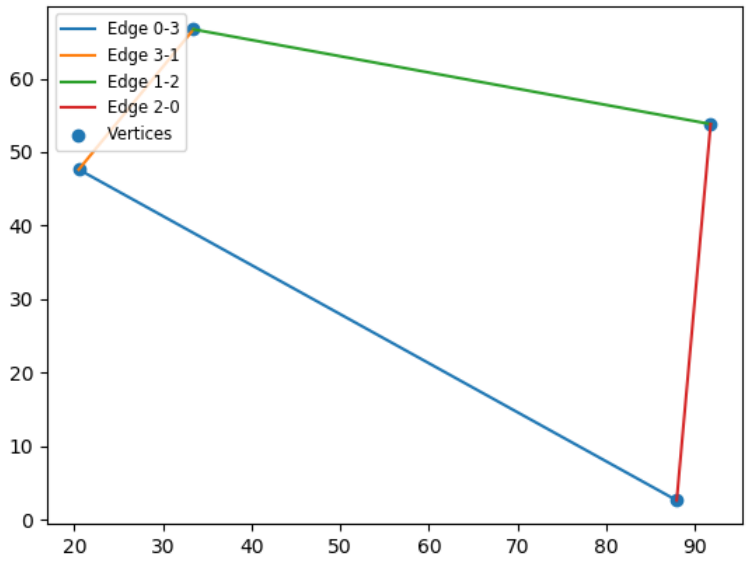

Figure 4: Random4.tsp Output

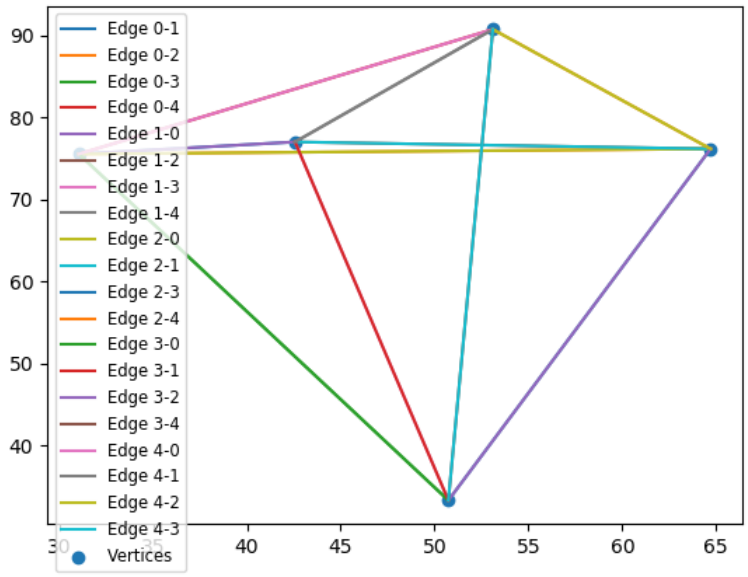

Figure 5: Random5.tsp Input

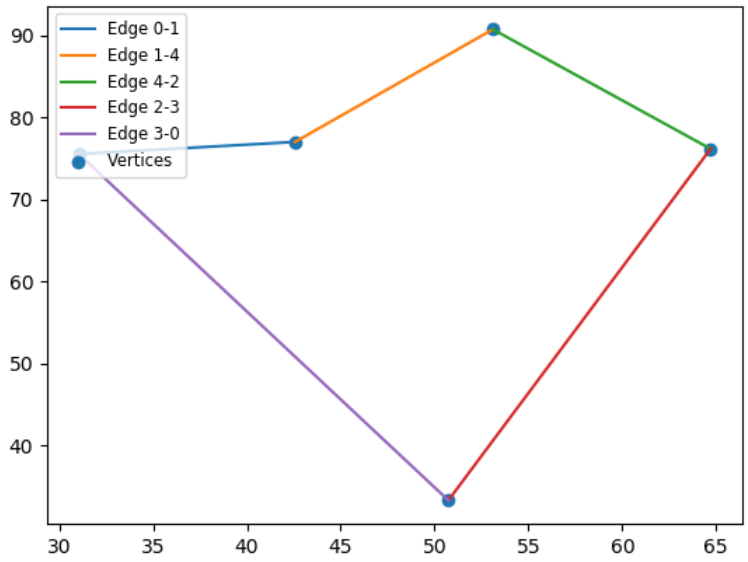

Figure 6: Random5.tsp Output

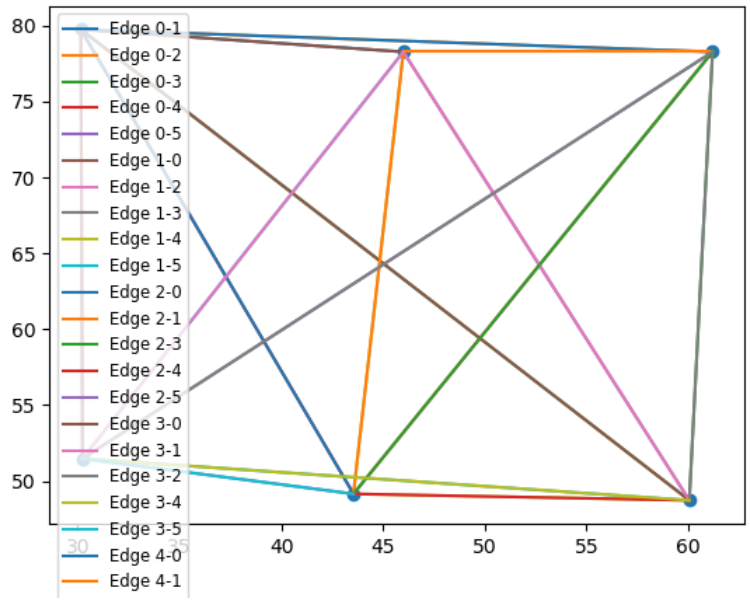

Figure 7: Random6.tsp Input

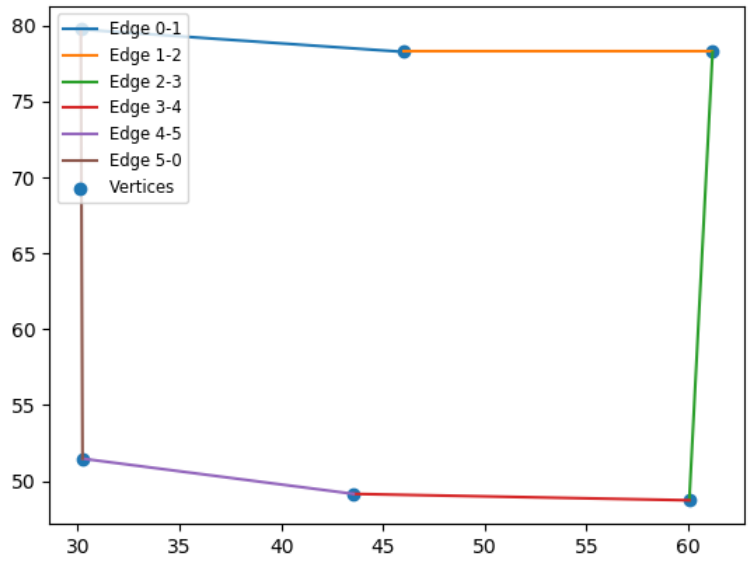

Figure 8: Random6.tsp Output

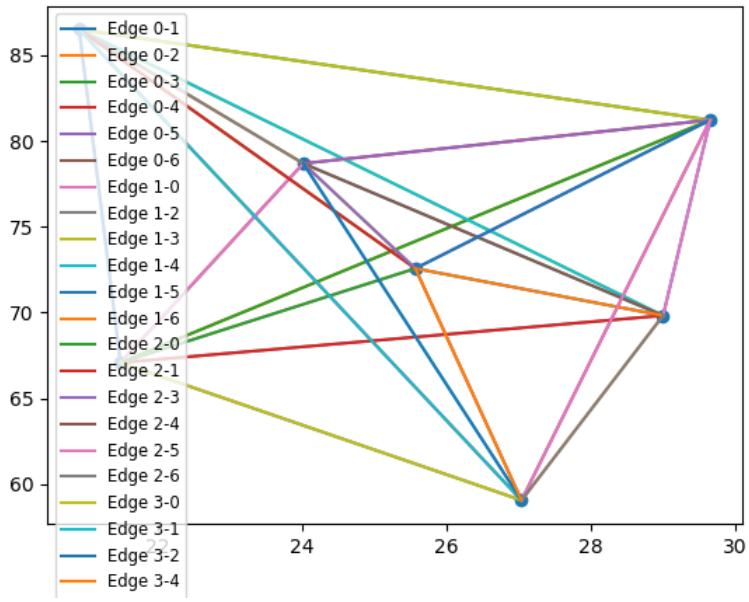

Figure 9: Random7.tsp Input

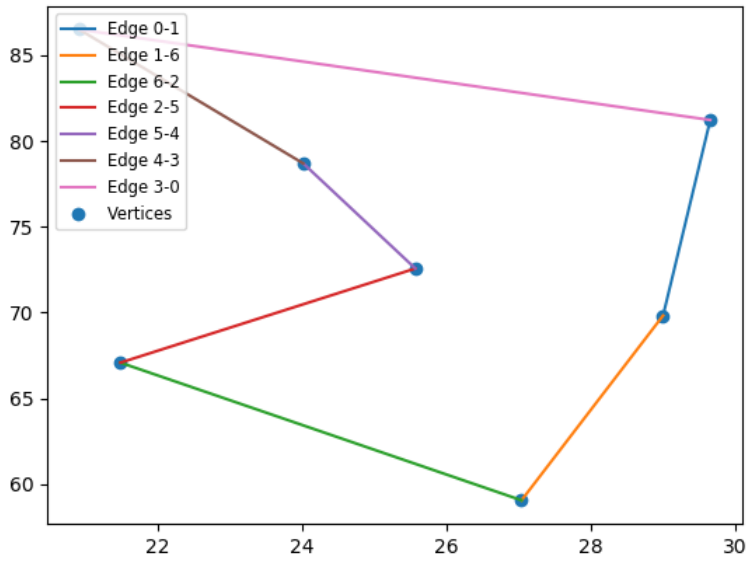

Figure 10: Random7.tsp Output

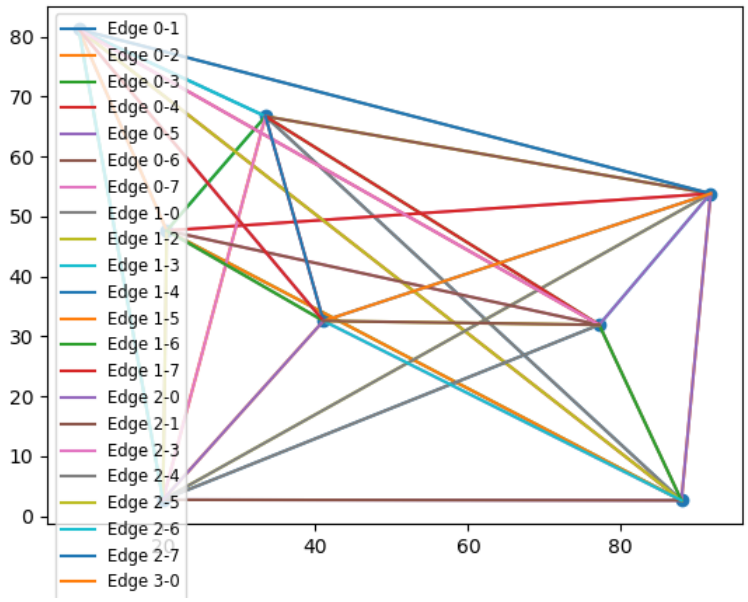

Figure 11: Random8.tsp Input

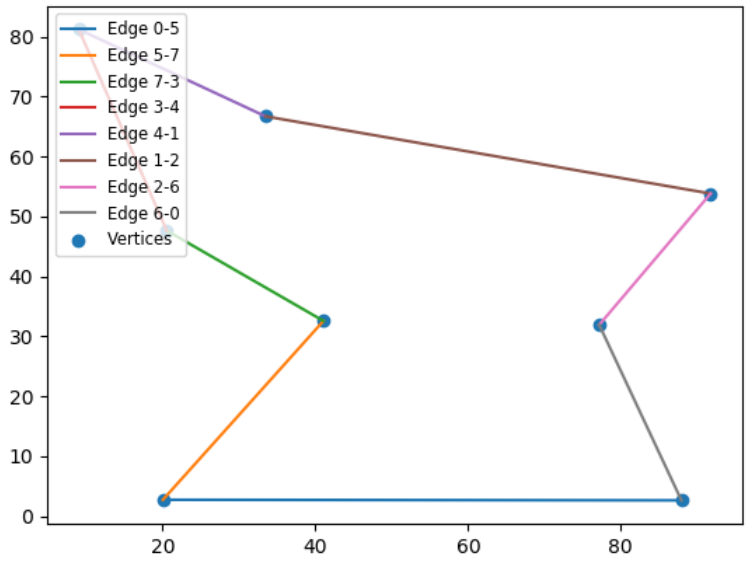

Figure 12: Random8.tsp Output

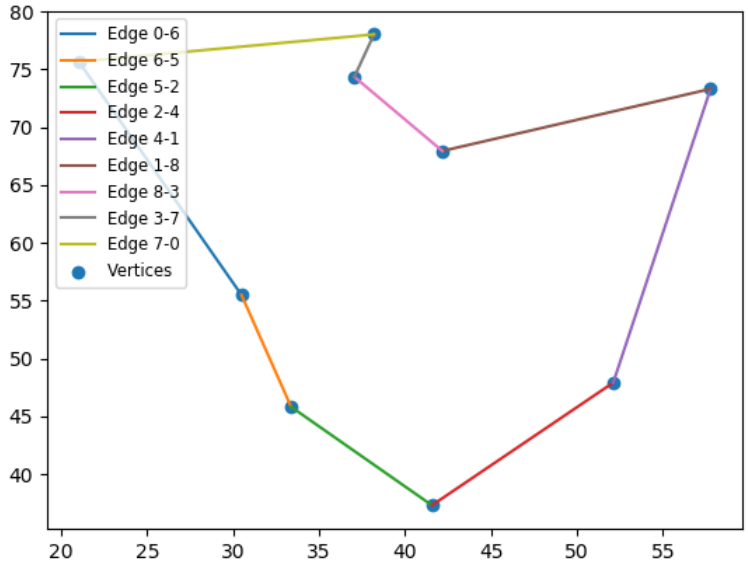

Figure 13: Random9.tsp Input

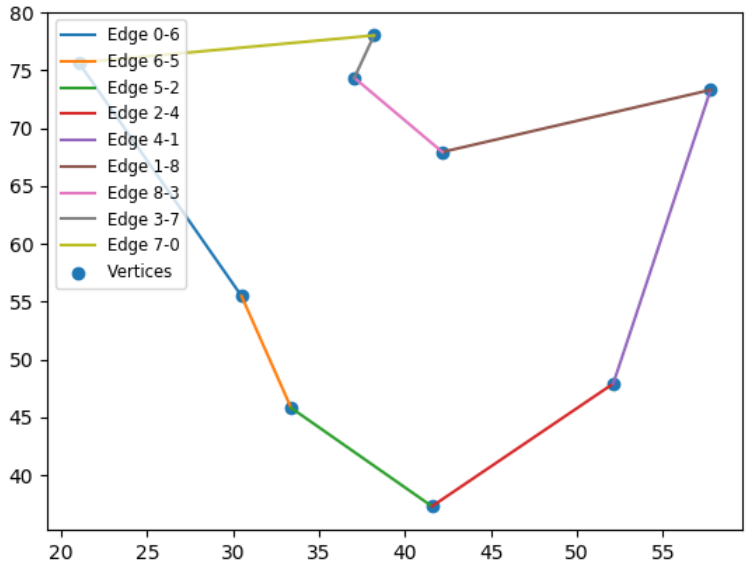

Figure 14: Random9.tsp Output

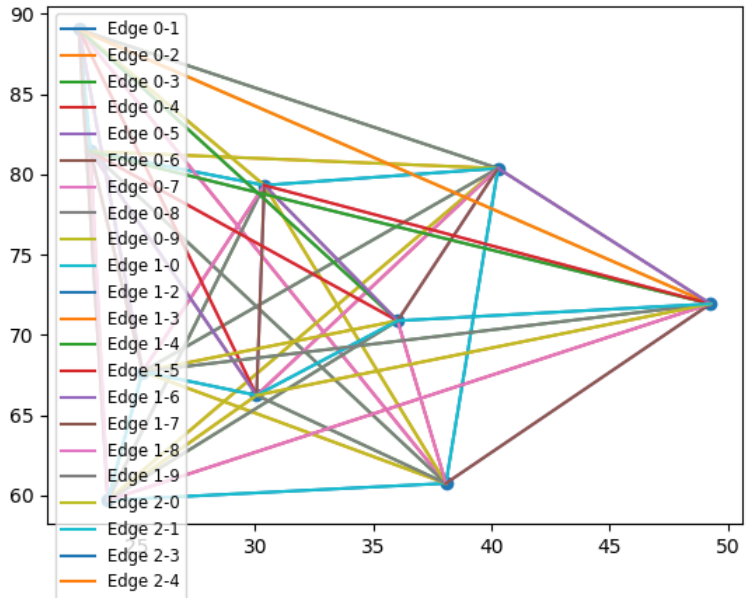

Figure 15: Random10.tsp Input

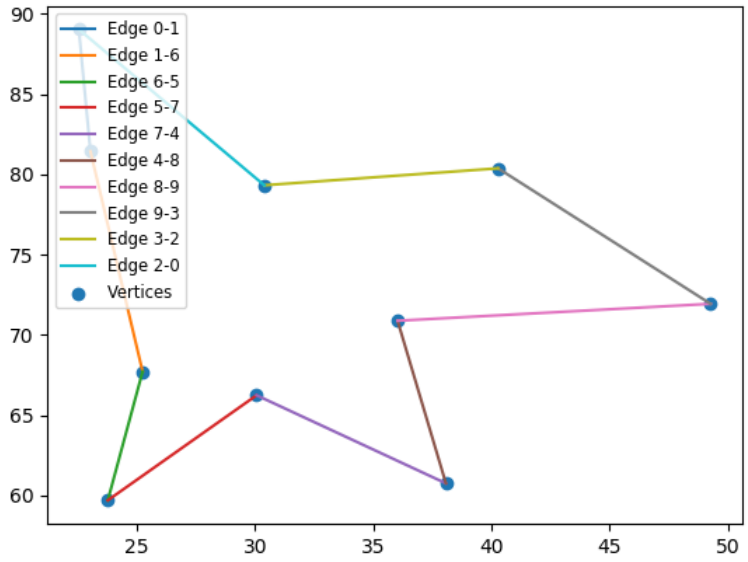

Figure 16: Random10.tsp Output

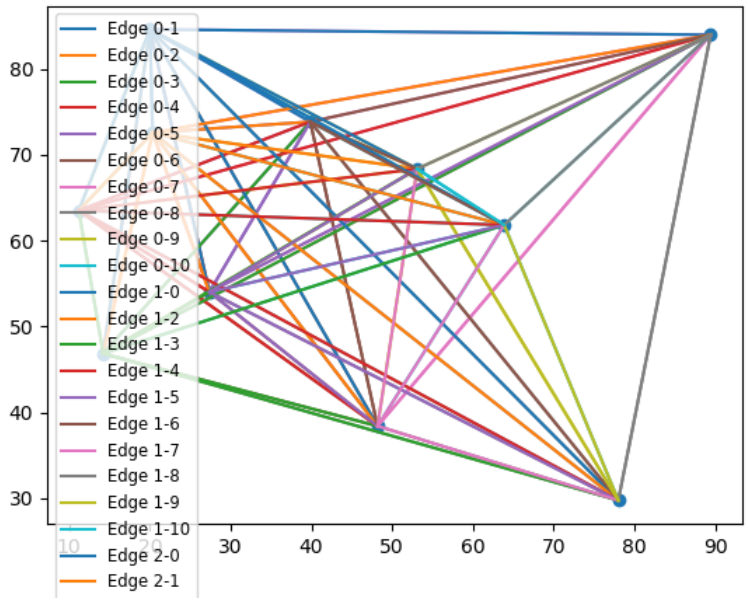

Figure 17: Random11.tsp Input

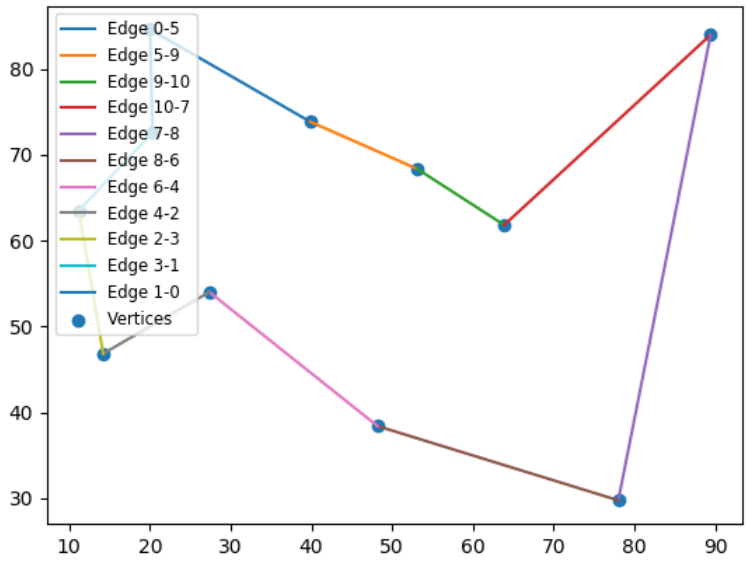

Figure 18: Random11.tsp Output

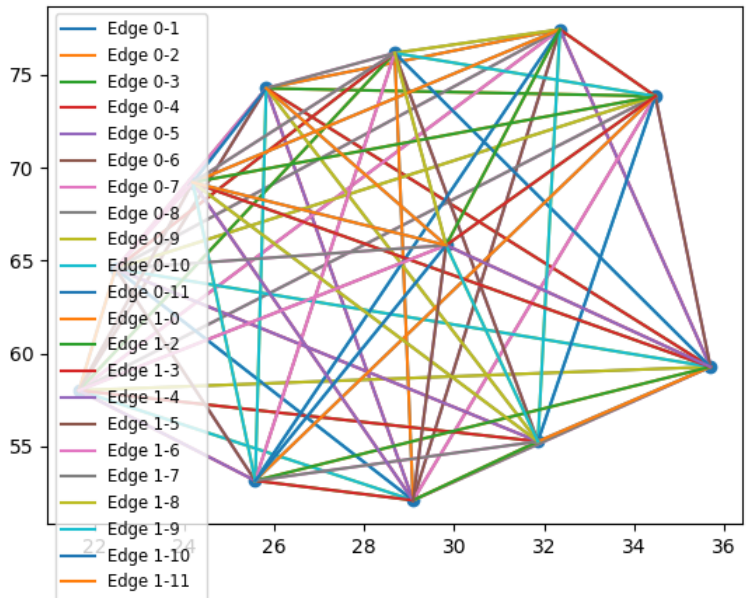

Figure 19: Random12.tsp Input

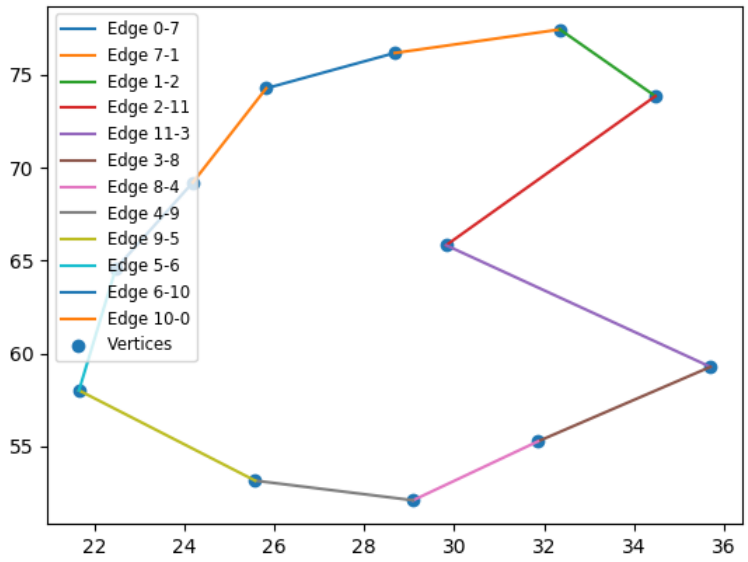

Figure 20: Random12.tsp Output