Travelling Salesman Problem - Greedy Algorithm

John Murray

Computer Science

Speed School of Engineering

jmmurr02@louisville.edu

1. Introduction

The Travelling Salesman Problem (TSP) is a problem in which the goal is the find the most efficient path between all of the nodes such that the path creates a Hamiltonian path. This report discusses using a modified greedy-heuristic to add nodes into the specified route one at a time. In this approach, we will be creating a true Hamiltonian path.

2. Approach

The solution was written in Ruby with a graphical front-end written with Shoes. It is important to note that Ruby has no built-in graphic UI libraries or toolkits. Thus, third-party solutions must be used. The framework that I am using is called Shoes. It is a minimalistic framework for creating GUI applications in Ruby.

Beyond the GUI implementation, the theoretical workings of the algorithm are very simple and can be broken down into a set of steps:

1. Select a random start node s
2. Find the node closest to s which we will call s'
3. Create a cyclic path between s and s' by:
    a. Create an edge from s to s'
    b. Create an edge from s' to s
4. Until all nodes are in the path
    a. For each edge e in the set of edges in the path
        i. For each node n in the set of nodes that have not been added to the path
            1. Find an e and n that have the shortest distance between them
            2. Alter the two nodes connected to e (n_{e1} and n_{e2}) such that each now have an edge that connect to n
            3. Delete edge e

That's a fairly simple and straightforward algorithm. However, it does not lead to the most optimal result (or even the same result) every time. Since it is a greedy algorithm, it is likely that it will find a solution before it finds the most optimal solution. However, the solution that it finds will depend on the heuristic in terms of optimality.

Choosing the First Node
Since we are choosing the first node randomly, this could result in a different solutions when run twice over the same data-set. This however, is a purposeful design choice. While it may not be the best solution if you only plan on running the algorithm once, it is useful when searching for more optimal solutions. Since this algorithm is fairly fast, there isn't much harm in running it multiple times over a single data-set (if the data-set is small enough). The randomness in choosing the first result should yield solutions of varying optimality. You could then choose the most optimal solution. Depending on your application area, the time taken to compute this could be worth it for a more optimal solution.

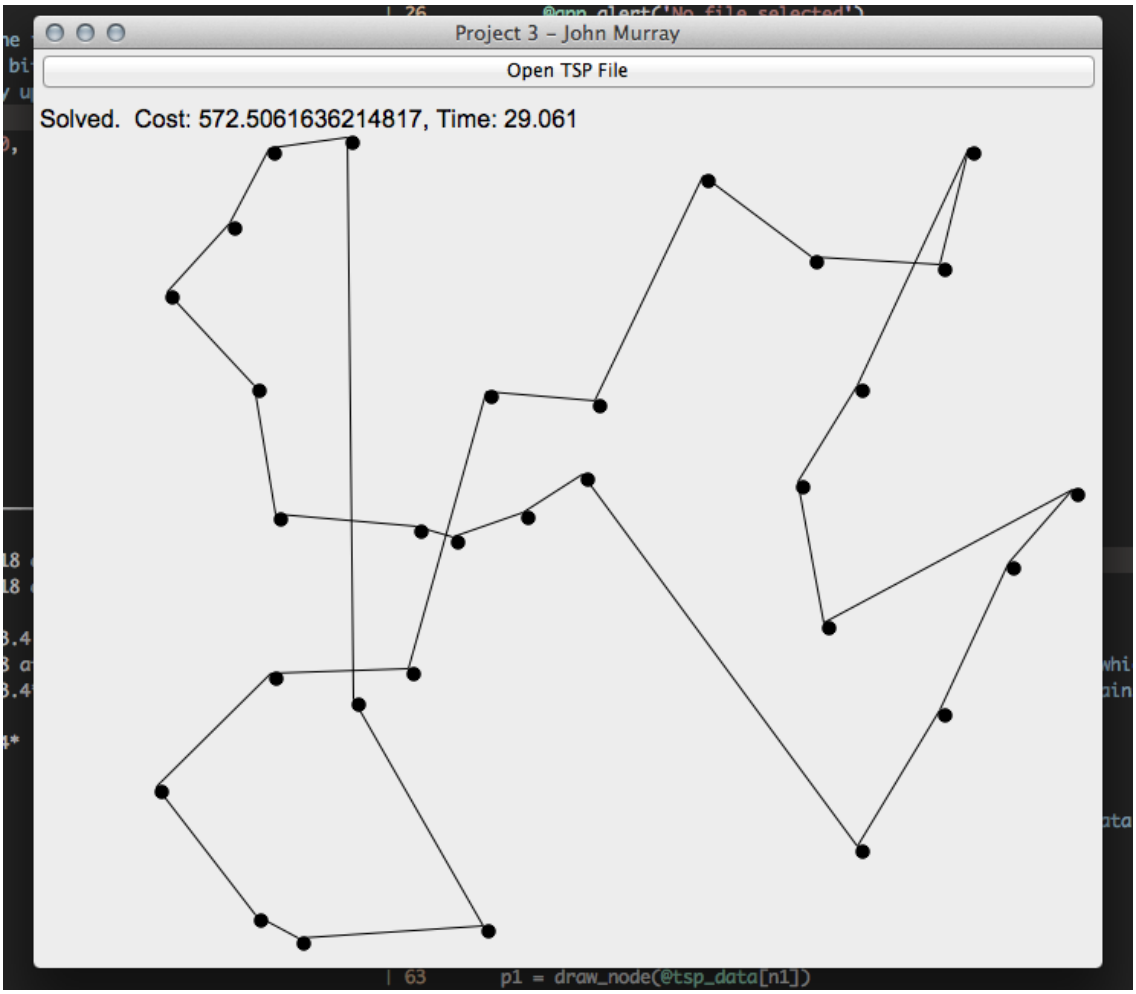
Another thing to consider is that the data could conform to a certain pattern such that choosing the same positional node (eg. the first or last node) as a starting position could lead to less optimal results than choosing randomly. By choosing randomly we should be able to, on average, solve for the average-case-optimality irrespective of the configuration or ordering of the data.

Choosing the Second Node
It makes sense, when choosing the second node, to find the node closest to the initial/starting node. However, the real importance is in how the edges are structured from the beginning. Since our algorithm works by replacing one edge with two (and therefore including another node) we have to be sure that we already have a Hamiltonian path from the beginning. It is important to realize that when we choose the second node, we then create a cyclic path such that each time a new node is added in, we still have a complete (cyclic) path.

Choosing the k^{th} Node
Once the first two nodes have been chosen, an algorithm is applied for choosing each successive node. For each edge currently in the path, the algorithm calculates the distance from the edge to each node. This is then done for all edges and the node/edge pair that has the shortest distance between them is added to the solution. This is then repeated until there are no nodes left that are not part of the solution-path.

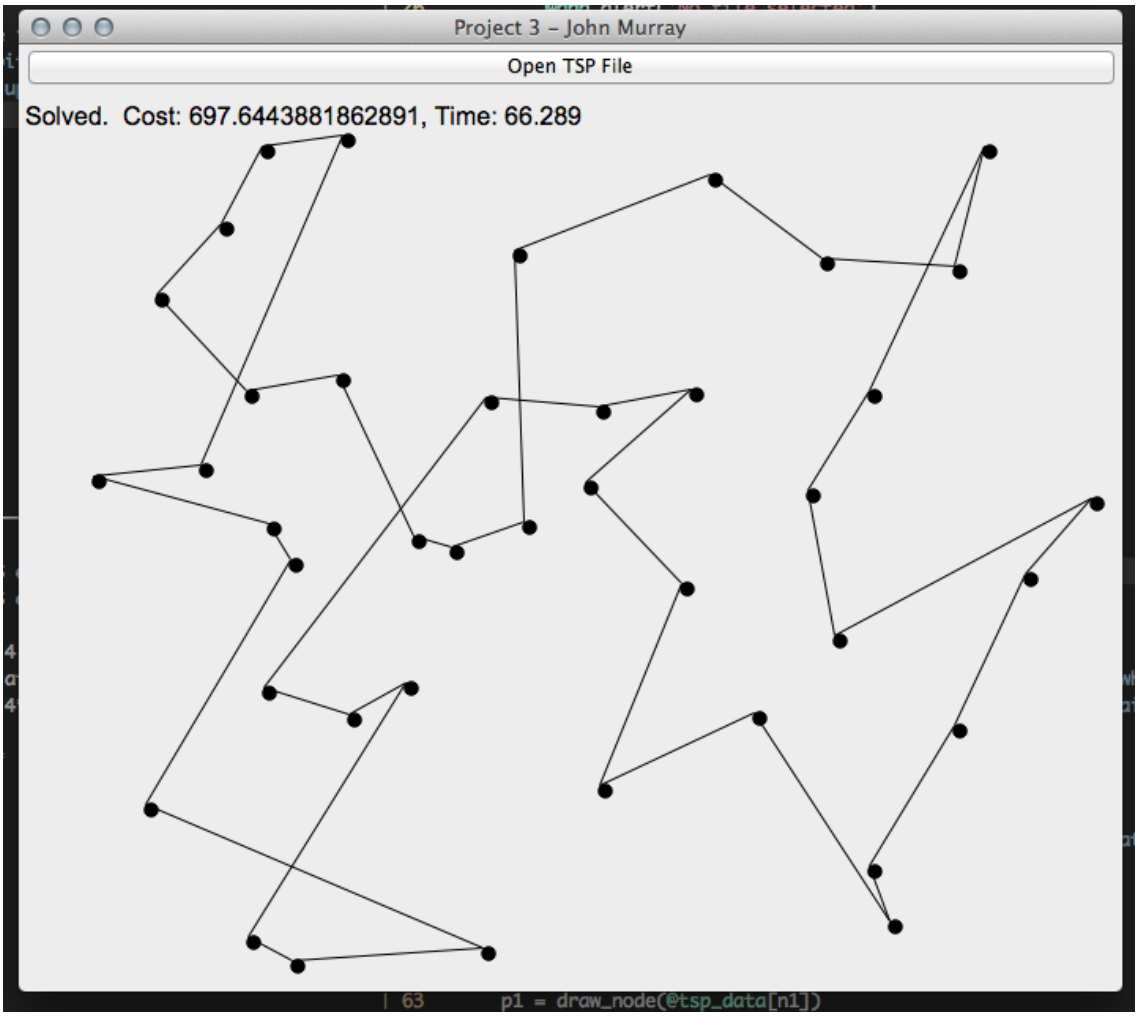
3. The Results
Data
The problem was solved using two TSP files (one with 30 points and one with 40 points). However, the solution should work with any random set of TSP data (however, the size of the data will impact the running-time and needed memory).

The Results
- 30 Points
  - Cost: 572.51
  - Time(ms): 29.06
  - Graph:



40 Points

- Cost: 697.64
- Time(ms): 66.29
- Graph:



4. Discussion

Speed

When compared to the brute-force techniques used in Project 1, this approach is by far a huge win. It was difficult to produce an optimal solution with only 12 nodes using the brute-force approach. And, although these solutions may not be optimal in all cases, it provides a near-optimal solution in a very small amount of time (milliseconds vs minutes/hours/days).

Optimality

This has already been discussed at length above. It should be noted however that this solution does not always produce the most optimal result. It should however, produce near-optimal results at worst.

However, given the speed of the algorithm, multiple passes could be made to try to find the most-optimal solution for the algorithm/heuristic.