Travelling Salesperson Problem

Evan Kelly   
Student Number: 5331799

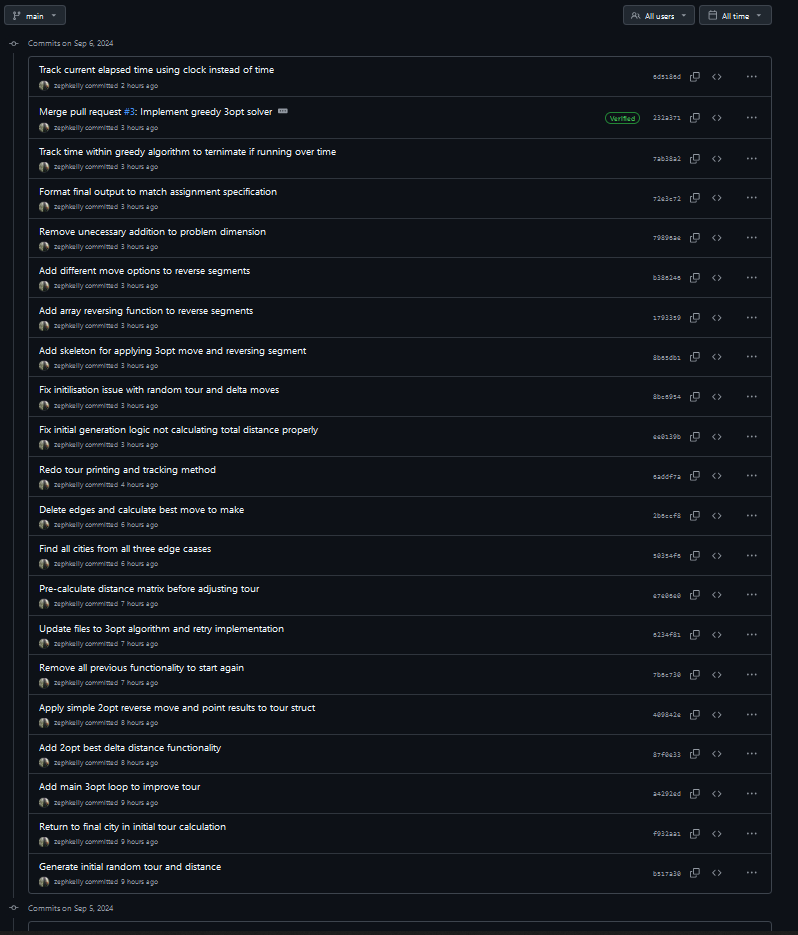
Link to public repository on Github with commit history:  
<https://github.com/zephkelly/griffith-1806ICT>

Table of Contents

[1.0 Problem Description 3](#_Toc488399413)

[2.0 Implementation details 4](#_Toc488399414)

[3.0 Results 5](#_Toc488399415)

# Problem Description

The travelling salesman problem, first encountered in 1832 and later mathematically formalised in the 19th century, aims to find the shortest distance one could travel between a collection of different cities where you can only visit each city once, forming a loop.

Classified as a NP-hard problem, the run time for any algorithm increases in the worst case exponentially with the number of cities you must search, making it an excellent candidate for testing various heuristics to reduce the overall runtime and increase the accuracy of the final solution.

# Implementation details

My TSP Solver is implemented in an object-oriented fashion with a focus on separation of concerns. The program is made in a number of key parts, including an abstract interface for interacting with the underlying implementation of each solver.  
  
My program implements two different solvers:  
  
**Nearest Neighbour**, and **Greedy 3-Opt**

The program is structured like so:

**main:** Manages the overall program flow and is responsible for outputting the values of the final Tour struct

**tsp\_utils:** Shared functionality between different solvers, mainly, the calculation of Euclidian and squared distances.

**tsp\_structures:** The various data structures used throughout the program including the problem representation and the final output representation.

**tsp\_file\_reader:** Responsible for reading in the data from each .tsp file in a flexible way. This data is represented as a TSPData object which is passed to each solver.

**tsp\_solver:** Abstract-like interface named ‘Solver’ which contains function pointers that call to the underlying solver implementation  
  
**nearest\_neighbour**: Implementation of the nearest neighbour TSP solver, which uses squared distances to search through every city at each iteration to find the next lowest cost city to travel to. Populates a Tour struct on completion.  
  
**greedy-3opt**: Implementation of the greedy 3opt TSP solver, which begins by generating a random tour, then, starting at the first index, it searches 3 other possible ways it can reconnect the given segments to try and find a reduced tour length. Each of these 3 options are analysed and the overall reduction in tour distance is used to select the most appropriate move to make which can range from a simple swap to a more complex deletion of edges and reattachment of new segments to form a new tour. It populates a Tour struct on completion.

# Results

**Small datasets**

**berlin52.tsp – Optimal solution distance: 7542 (from University of Heidelberg)**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

As shown above, the program correctly outputs the required information to the console, including the problem name, the shortest found tour length, and the tour written as a vertical list with no repeating characters and a terminating -1 to indicate the end of the tour.

berlin52 has an optimal tour distance of 7542. The nearest neighbour approach seems to always return tour distances that are within 15-20% of the optimal solution. While the Greedy 3opt approach can (inconsistently due to the initial random tour generation) produce the most optimal solutions or very close to, optimal solutions for small problem sets.  
A screenshot of a computer program

Description automatically generated  
  
  
Here are examples of the Greedy 3opt algorithm not finding the most optimal solution. However, when compared to the nearest neighbour approach, greedy 3opt consistently perform better.

A screenshot of a computer

Description automatically generated**eli101.tsp – Optimal solution distance: 629**  
A screenshot of a computer

Description automatically generated

Again, on small datasets, we can see that the Greedy 3opt algorithm performs significantly better than Nearest Neighbour, finding solutions that are within a few percent of the optimal solution.

**Medium datasets**

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated**A280.tsp – Optimal solution 2579**

Both heuristics finished relatively instantly. Again, we can see that the Greedy 3opt heuristic converges on a far more accurate optimal solution than nearest neighbour.

**d493.tsp – Optimal solution 35002**

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

Greedy 3opt took around 20 seconds to complete, nearest neighbour almost instantly.

**Large datasets**

A screenshot of a computer

Description automatically generated**ul1060.tsp – Optimal solution 224094**

A screenshot of a computer

Description automatically generated

When analysing larger datasets, the Nearest Neighbour heuristic finishes almost instantaneously, while the Greedy 3opt heuristic takes over 4 minutes to complete.

This increased calculation time does result in a much more accurate tour length for Greedy 3opt, however it does make other more computationally efficient heuristics like Nearest Neighbour the better choice in some scenarios.

One thing I have learned from running my two implementations against a large dataset is that there are ways I could have implemented hybrid approaches to solve problems more efficiently. For example, since the nearest neighbour heuristic finishes almost instantly, I could use it to seed the initial tour that feeds into the Greedy 3opt heuristic. This would mean that the starting tour is 80% of the way to being completely optimised, and Greedy 3opt can bring it within a few percent of the optimal solution within a more reasonable timeframe.

**d18512.tsp – Optimal solution 645238**

A screen shot of a computer screen

Description automatically generated

For fun I tried to run the largest data set and it became obvious it would not complete in any reasonable time frame, even for my nearest neighbour approach. There are a number of optimisations I could apply to my nearest neighbour approach, such as (memory space permitting) pre-computing a distance matrix between all of the cities before running the search.

A screenshot of a computer

Description automatically generated

Here is an example of setting a time limit of 10 seconds, when the time limit is reached the program gracefully exits and whatever results were captured are printed to the console.

University of Heidelberg, (2007). Optimal solutions for symmetric TSPs.

<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html>