Travelling Salesperson Problem

Evan Kelly
Student Number: 5331799

Table of Contents

1.0 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.

2.0 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.

3.0 Results

Small datasets

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

```
Solving with Nearest Neighbour:
berlin52.tsp
Shortest found tour length: 8980.000000
Tour:
1
22
49
32
36
35
34
39
40
38
37
48
24
5
15
6
4
25
46
44
16
50
20
23
31
18
3
19
45
41
8
10
9
43
33
51
12
28
27
26
47
13
14
52
11
29
30
21
17
42
7
2
-1
```

```
Solving with Greedy 3-Opt:
berlin52.tsp
Shortest found tour length: 7542.000000
Tour:
16
29
50
20
23
30
2
7
42
21
17
3
18
31
22
1
49
32
45
19
41
8
9
10
43
33
51
11
52
14
13
47
26
27
28
12
25
4
6
15
5
24
48
38
37
40
39
36
35
34
44
46
-1
```

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.

```
Solving with Greedy 3-Opt:
berlin52.tsp
Shortest found tour length: 8007.000000
Tour:
16
50
20
23
```

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.

eli101.tsp – Optimal solution distance: 629

```
Solving with Nearest Neighbour:
eil101.tsp
Shortest found tour length: 817.000000
Tour:
1
69
27
101
53
58
40
21
73
72
74
22
75
56
39
23
67
```

```
Solving with Greedy 3-Opt:
eil101.tsp
Shortest found tour length: 641.000000
Tour:
75
41
22
74
72
73
21
40
58
2
57
15
43
42
14
38
```

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

A280.tsp – Optimal solution 2579

```
Solving with Nearest Neighbour:
a280.tsp
Shortest found tour length: 3139.000000
Tour:
1
280
2
3
279
278
4
277
276
275
274
273
272
271
16
17
18
19
20
```

```
Solving with Greedy 3-Opt:
a280.tsp
Shortest found tour length: 2690.000000
Tour:
222
221
220
217
218
215
216
213
214
211
212
207
210
209
252
255
256
257
```

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

```
Solving with Nearest Neighbour:
d493.tsp
Shortest found tour length: 43632.000000
Tour:
1
2
4
5
6
7
11
10
9
8
12
19
17
18
16
15
14
```

```
Solving with Greedy 3-Opt:
d493.tsp
Shortest found tour length: 36166.000000
Tour:
471
470
469
468
467
465
443
431
432
425
424
409
396
389
377
381
368
```

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

Large datasets

ul1060.tsp – Optimal solution 224094

```
Solving with Nearest Neighbour:
u1060.tsp
Shortest found tour length: 306108.000000
Tour:
1
3
4
1060
1059
1058
1057
1056
1055
1052
1054
1053
31
30
12
29
28
27
32
1051
1050
```

```
Solving with Greedy 3-Opt:
u1060.tsp
Shortest found tour length: 231448.000000
Tour:
429
398
397
396
395
394
393
392
391
274
390
389
276
275
273
270
271
272
268
269
```

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

```
Solving with Nearest Neighbour:
    Progress update: 515 cities visited, current tour distance: 11676.00
    Progress update: 731 cities visited, current tour distance: 15829.00
    Progress update: 898 cities visited, current tour distance: 19343.00
    Progress update: 1039 cities visited, current tour distance: 22795.00
    Progress update: 1164 cities visited, current tour distance: 25264.00
    Progress update: 1277 cities visited, current tour distance: 28690.00
    Progress update: 1381 cities visited, current tour distance: 32990.00
^Z
```

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.

University of Heidelberg, (2007). Optimal solutions for symmetric TSPs.
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html