AI:SMPS Report

Varun Venkatesh, 21f1000743

March 26, 2024

Abstract

The Traveling Salesman Problem (TSP) is a widely studied and established optimization problem. Linear programming can provide optimal solutions for small instances of the problem in a reasonable amount of time. Nevertheless, due to its classification as NP-hard, solving larger instances of the TSP with an assured optimal solution will require an extensive amount of computation time.

1 Introduction

In this assignment, our objective was to determine solutions for a range of medium to large Travelling Salesman Problems using both euclidean and non-euclidean distance matrices. Rather than applying one of the heuristics taught in the course, I chose to investigate published research that compared different TSP algorithms before beginning the solution process.

2 Literature Review

There are various studies that provide a comparative analysis of different heuristics for the TSP. For instance, Nilsson (2003) conducted a comprehensive evaluation of Nearest Neighbour, Greedy, Insertion, Christofides, k-opt, Lin-Kernighan, Simulated Annealing, Genetic Algorithms, Branch Bound, and Ant Colony Optimization heuristics. The results indicated that, in most scenarios, Lin-Kernighan produced the most optimal solutions, with only a marginal 1% deviation from The Held-Karp Lower Bound. This lower bound refers to the optimal solution for the linear programming relaxation of the integer programming formulation of the TSP. According to Johnson, McGeoch, and Rothberg (1996), the solution can be determined in polynomial time using the Simplex method and a polynomial constraint separation algorithm. However, the 2-opt algorithm was also found to be efficient, with a quick processing time and only a 4% deviation from the Held-Karp Lower Bound.

Likewise, , Kim, Shim, and Zhang (1998) conducted a comparative analysis of the Greedy, various $k-$opt variations, Simulated Annealing, and a neural network. Their findings revealed that the neural network algorithm yielded the best results if computational time was not a constraint. However, in all other circumstances, the $2-$opt algorithm was the recommended choice.

3 Algorithm

3.1 Base Algorithm

The pseudocode for the 2-opt swap procedure, which illustrates how the given route is modified as shown in Figure 1, is presented below. In this algorithm, v1 and v2 represent the initial vertices of the edges that need to be swapped as the route is traversed.

Algorithm 3.1.1: 2-opt, Credit: Wikipedia

```
procedure 2optSwap(route , v1, v2) {
  1. take route[0] to route[v1] and
  add them in order to new_route
  2. take route[v1+1] to route[v2] and
  add them in reverse order to new_route
  3. take route[v2+1] to route[start] and
  add them in order to new_route
  return new_route;
}
```

The $2-$opt algorithm is a method of improving a given solution or path by iteratively swapping pairs of vertices until the path length cannot be further reduced. It is commonly applied to candidate solutions generated by other heuristics. However, for the present task, a randomly generated candidate solution was provided as input to the algorithm.

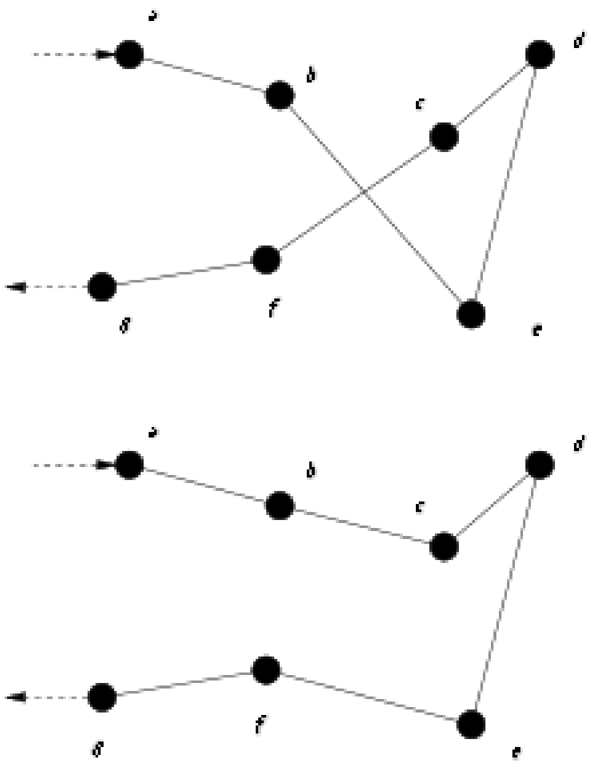

Figure 1: A 2-opt move, Credit: Wikipedia

3.2 Efficient Implementation

The process of constructing a new route and computing its distance can be computationally expensive, typically with a time complexity of $O(n)$, where n represents the number of vertices present in the route. An efficient implementation of $O(1)$ complexity begins by considering a exchange of vertices v1 and v2. If this exchange yields a lower path distance than the previous one, the two vertices are swapped. If not, the algorithm proceeds to v3 and assesses the exchange, and so on, until an improvement is found. Once v1 and v2 are swapped, the algorithm takes this as an initial solution and repeats the process until no further improvements can be made. This implementation of the 2-Opt algorithm makes the swap permanent upon discovering an improvement, resulting in less computational time compared to the 2-Opt algorithm. In Kim et al. (1998), this was referred to as the "Greedy 2-opt".

3.3 Issues

In view of the fact that 2−opt is a refinement algorithm, it cannot assure achieving the optimal solution. Figure 2 illustrates an instance where the algorithm can provide a locally optimal solution, and if the algorithm was initiated with a different candidate solution, it could have achieved the global optimum instead.

The problem of returning locally optimal solutions by the 2-opt algorithm was addressed by implementing the algorithm with multiple randomly generated candidate solutions. Though it cannot guarantee global optimality, employing a sufficiently large number of iterations with different candidate solutions resulted in solutions that were very close to the global optimum in practice.

In light of the varying number of vertices, a fixed number of iterations or candidate solutions for refinement would not be feasible. In this assignment, the problem statement allowed for a maximum processing time of 300 seconds, which means that only a limited number of candidate solutions could be attempted within this timeframe. For instance, if there were 1000 vertices, only around 50 candidate solutions could be processed within 300 seconds, whereas for smaller numbers of vertices, such as 50, thousands of candidate solutions could be refined.

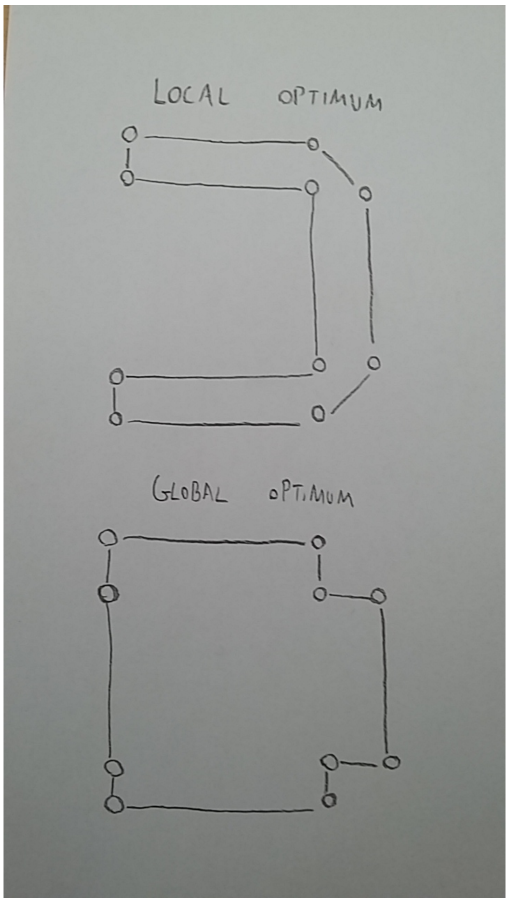

Figure 2: An example of local optima, Credit: J. Schmidt, StackExchange

To address this issue, an exponential decay function was formulated to determine the number of candidate solutions to use as follows:

$$5000 * e^{\frac{-\text{number of vertices}}{500}}$$

As the "Efficient Implementation" of the 2-opt algorithm was used, a 2-opt move was only performed if lengthDelta < 0 where lengthDelta is defined below:

Algorithm 3.3.1: Credit: Wikipedia

```
lengthDelta =
  - dist(route[v1], route[v1+1])
    - dist(route[v2], route[v2+1])
      + dist(route[v1+1], route[v2+1])
        + dist(route[v1], route[v2])
```

The 2-opt algorithm was unable to converge within 300 seconds for problems with floating-point distances, even when the problem consisted of only 25 cities, as the algorithm kept finding two vertices that could be exchanged, resulting in an infinitesimally small negative delta in the length of the path. To address this issue, a threshold value of lengthDelta $< \epsilon$, where ϵ is a small negative number, was used instead.

4 Conclusion

Due to the sheer number of possible paths (25!) for a 25 city problem, I was unable to test the solution of this algorithm against the globally optimum solution. Nevertheless, in comparison to the solutions generated by my peers in the trial runs, the algorithm performed competitively.

For the future I'd like to implement the "Improved Greedy 2-opt" as specified in Kim et al. (1998). The improvement is as follows: First, use the Greedy 2-opt to find a solution. Then, identify the segment between v1 and v2, and compare it with other segments in the network to check for any intersections. If there are no intersections, move on to the segment between v2 and v3 and compare it with subsequent segments in the network. If any two distinct segments intersect, the loop is rerouted. This sequential process is repeated throughout the entire network. It would be worthwhile to examine whether utilizing a candidate solution for the 2-opt generated by the Nearest Neighbour or Greedy Heuristic at the start results in a marked enhancement of the solution produced or a reduction in computational resources utilized.

Disclaimer

Some parts of this academic report may contain borrowed information from other sources, including research papers and articles. The reason for this is to convey the thought process behind my decision to choose a particular algorithm for the problem statement given, and to provide an explanation of the reasoning used by the original authors. All borrowed content has been properly cited and referenced, and the original sources have been acknowledged. This report is not intended to represent original research, but rather a summary and analysis of existing literature related to the chosen algorithm.

References

Johnson, David S, Lyle A McGeoch, and Edward E Rothberg (1996). Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Volume 341, pp. 350. ACM Press San Francisco.

Kim, Byung-In, Jae-Ik Shim, and Min Zhang (1998). Comparison of tsp algorithms. *Project for Models in Facilities Planning and Materials Handling 1*, 289–293.

Nilsson, Christian (2003). Heuristics for the traveling salesman problem. *Linkoping University 38*, 00085–9.