TSP-report

David Östling, Mohamed Mahdi, Hamid Noroozi, Filip Döringer Kana

October 2022

Contents

1 Introduction

During this project, the group was tasked with creating an approximate solution for the already known *traveling salesman problem (TSP)* [1] with the goal of creating as good a solution as possible in under two seconds running time. In particular, this regards the 2d version of the TSP problem and it is given by n cities where the distance d_{ij} between each city i and j is given by the euclidean distance between the two cities. In this case, the distance is also always rounded to the nearest integer. The goal is to find the shortest tour with the condition that all cities are visited exactly once, given the time limitations.[1].

2 Algorithm idea

The algorithm consists of three parts. First, an initial tour is found using a greedy nearest neighbor approach [3]. This initial tour is then iterated upon with a local search heuristic until no more local improvements can be made. Afterwards, if there is time left, the local heuristic is run again but from a different starting tour. Finally, only the best tour is kept. Below, each of these approaches will be explained in more detail.

2.1 Greedy Nearest neighbor

The nearest neighbor algorithm that produces an initial tour is a very simple naive function running in $O(n^2)$ [3]. It involves picking a starting vertex and then iteratively adding the vertex closest to last vertex added to the tour. Thus when n-1 vertices have been picked, the final one is the last remaining vertex. This property of the algorithm can make it produce very inefficient results and is why this procedure is done only in order to generate a starting tour.

2.2 Two-Opt

Once a tour is constructed it can be improved using the local optimization algorithm 2-opt [4]. Local optimization algorithms iteratively locate a local optima. The steps consist of picking two arches in the tour to connect and calculate the new cost. If a changed route's new cost is favorable it is kept and the previous route is connected in reverse. This is repeated up until implemented limitation, could be that there are no more improvements found or time restraint reached. Figure 1 shows the implemented two-opt swap function. If swapping the edges decreases the total length of the tour, the swap is improved. Notice that the edges in between the swapped edges are reversed. The implementation of the swap is based on the work in [5].

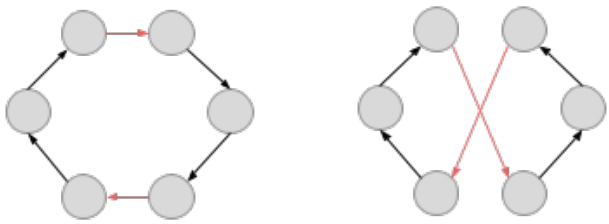

Figure 1: *Example of two edges swapping*

2.3 Randomization

Since 2 opt is a local search heuristic that moves from one state to a neighboring state, it can easily get stuck in a local minimum where no improvements can be made, yet the optimal value still has not been found. Thus, after running 2-opt on the tour produced by the greedy nearest neighbor algorithm, we randomly generate a new starting tour and run 2-opt again. This procedure is repeated as many times as possible during the given time limit and the overall best tour is kept. This will increase the chance that we pick a starting tour where 2-opt can find a tour that is closer to a global optimum. Also, in order to not randomly generate the same starting tour again, we hash the tour that is randomly generated and save the hash in a set. Whenever we generate a random tour we then also check if we have previously generated it and if that is the case, we simply discard it.

3 Pseudo-code

3.1 Two-Opt swap

Algorithm 1 two_opt_swap

Require: $edges[\,]$ & $costs\_matrix[\,][\,]$ & $first$ & $second$
Ensure: Swaps edges if cost is improved
  $e\_1 \leftarrow edges[first]$
  $e\_2 \leftarrow edges[second]$
  $v1 \leftarrow$ first vertex of e_1
  $v2 \leftarrow$ second vertex of e_1
  $v3 \leftarrow$ first vertex of e_2
  $v4 \leftarrow$ second vertex of e_2
  $currentCost \leftarrow costs[v1][v2] + costs[v3][v4]$
  $newCost \leftarrow costs[v1][v3] + costs[v2][v4]$
  if $newCost < CurrentCost$ **then**
    $edges[first] \leftarrow (v1, v3)$
    $edges[second] \leftarrow (v2, v4)$
    Reverse all edges between e_1 and e_2
  end if

3.2 Two-Opt

Algorithm 2 two_opt

Require: *edges*[ ] & *costs_matrix*[ ][ ] & *start_time*
Ensure:
  threshold ← 1.9
  stop_time ← *current_time* − *start_time*
  improved ← *True*
  while *improved* & *stopTime* < *treshold* **do**
    improved ← *False*
    for (i = 0; i < length(edges)-2; i++) **do**
      for (j = 1; i < length(edges); j++) **do**
        swap ← *two_opt_swap*(*edges*, *cost_matrix*, *i*, *j*)
        if Swap was performed **then**
          improved ← *True*
          i ← *j* + 1
          Break from inner loop
        end if
      end for
    end for
    stopTime ← *currentTime* − *startTime*
  end while

3.3 Randomization

Algorithm 3 Randomization

Require: *tour*[ ]
Ensure: *best_tour*
  threshold ← 1.9
  start_time ← *current_time*
  best_tour ← *two_opt*(*edges*, *cost_matrix*, *current_time*)
  best_cost = cost of best_tour
  hashes ← [ ]
  while *current_time* − *start_time* < *treshold* **do**
    Shuffle the tour
    h ← *hash*(*tour*)
    while h in hashes **do**
      Shuffle the tour
      h ← *hash*(*tour*)
    end while
    new_cost ← cost of tour
    if *new_cost* < *best_cost* **then**
      best_cost ← *new_cost*
      best_tour ← *tour*
    end if
    Append h to hashes
  end while
  return *best_tour*

4 Results

Table 1 shows the scores achieved on Kattis for the individual parts of the algorithm that make up the final solution as well as the final algorithm in the last row of the table.

Algorithm	**Kattis Score**
Greedy	2.990721
Greedy + 2-opt	20.067428
Greedy + 2-opt with randomization	24.630963

Table 1: Kattis results from running the different algorithms

5 Discussion

One of the limitations of this implementation is the chosen programming language. If the implementation would have been done in a more efficient language, such as c++ we could likely have achieved a higher score with similar code. Also, the code was not optimized to reduce the time complexity as much as possible either. Another improvement could have been to use something like simulated annealing combined with 2-opt to see if we could improve the score that way. Also, other local search optimizations could have been implemented, such as 3-opt.

5.1 First implementation

The very first implementation of the code was not meant to score high points in Kattis. It was more a matter of constructing an efficient skeleton and trying our best to get the core functionality right. As a result, the score was pretty low but it served as a nice guideline for the remainder of the project. The group started building from this solution by optimizing it in different ways and after some reconstruction it shaped the final solution.

5.2 Failed attempts

One failed attempt was trying to develop the *Christofides approximation algorithm* [2] that can guarantee a score of $ALG <= 1.5OPT$. We started by implementing Prim's algorithm to find a Minimum spanning tree of the graph. Soon we realized that the work required would most likely not be worth it, and that using local heuristics would most likely be a better choice. Therefore, we scrapped the idea of using Christofides. Another idea was to have our base tour be the *2-approximation* [4] obtained by running a depth first search on an MST of the graph. We tried this idea but found that the tour received that way did not score higher than the greedy algorithm and we decided against using the 2-approximation algorithm to obtain our initial tour.

References

[1] Austrin P., Håstad J., 2000, Chapter 16: Notes for the course advanced algorithms p.129 - 131

[2] Austrin P., Håstad J., 2000, Chapter 16: Notes for the course advanced algorithms p.131 - 132

[3] Austrin P., Håstad J., 2000, Chapter 16: Notes for the course advanced algorithms p.132

[4] Austrin P., Håstad J., 2000, Chapter 16: Notes for the course advanced algorithms p.133 - 136

[5] Englert, M., Röglin, H., & Vöcking, B. (2014). Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP. Algorithmica, 68(1), 190-264