TSP report

Zaina Ramadan
Omid Fattahi Mehr
Stefano Wirth
Marin Petric

November 2023

1 Introduction

The purpose of this project is to implement an algorithm that finds an approximate solution to the Traveling Salesman Problem (TSP), which is a known NP-hard problem. TSP states that given a list of locations or cities and the cost between them, find the shortest path that visits each city exactly once and returns to the origin city.

2 Algorithm

The algorithm used to find an approximate solution to the Traveling Salesman Problem is the Christofides algorithm [1]. The main steps involved in the algorithm are, given a graph G:

1. Create a minimum spanning tree T from G
2. Find the set of vertices with odd degrees O in the T
3. Find the minimum-cost perfect matching M of the induced subgraph given by the graph containing all vertices in O only
4. Merge T with M to get a new graph with all vertices having even degrees
5. Form an Eulerian circuit E
6. Make the circuit E a Hamiltonian circuit H, to skip repeated vertices

The first two steps described have been implemented in the tsp.cpp file, the minimum-cost perfect matching implementation used is an implementation from Vladimir Kolmogorov [3] and makes up most of the files, excluding main.cpp, tsp.cpp and util.cpp, and the rest is implemented in the main.cpp file. In the algorithm, vertices are represented as integers and are numbered sequentially from zero to $N-1$ as they are read from the input, where N is the number of vertices in the graph. The edges are represented as pairs of integers $(v1, v2)$, where $v1$ and $v2$ represent the vertices connected by the edge, stored in a tuple.

2.1 Minimum Spanning Tree

In order to find the minimum spanning tree, Prim's algorithm is used [4]. The implementation can be found in the findMST() function. The function takes as input the number of vertices and the distance matrix, which contains the distances from each vertex to all other vertices. In the function, the MST is stored as a list of edges, that are contained in the MST. A priority queue is used to achieve good running times, where priority is given to the closest vertex not already added to the MST. This achieves in practice a fast running time, since we do not have to recheck all connections between the growing MST and the rest each update, only if our added vertex has a lower distance to a given vertex than its previous lowest distance parent.

2.2 Finding the vertices with odd degree

To find the vertices with odd degree, the findOddVertices() function is used that takes the MST (return value from findMST()) as input. The basic process is to loop through each edge in the MST and have a counter for each vertex, counting the number of edges connected to a vertex. Then the vertices with an odd count are selected and added to the list of odd vertices, which is returned. Additionally, findEdges() is used to find the edges connected between two odd vertices.

2.3 The minimum-cost Perfect Matching

For the minimum-cost Perfect Matching we utilized an efficient implementation by Vladimir Kolmogorov, which is described in the following publication [3]. The implementation is based on Edmonds's algorithm for solving the min-cost Perfect Matching and combines the idea of "variable dual updates" [2] with the use of priority queues. To obtain the perfect matching, we create a PerfectMatching object that receives the set of odd edges as input. We then call the solve() and getSolutions() methods to retrieve M.

2.4 TSP Tour

To create a tour from the MST with added perfect matching edges, we use a recursive algorithm. We first turn the MST into an adjacency list, since this is the fastest data structure for our operations, and then call the recursive step function. In one step, this function chooses and removes edges from a starting vertex (by popping from the list) as long as they exist and steps to the end vertex of that edge. It then recursively calls itself with the ending vertex as the new starting vertex. After the recursive call, it adds the starting vertex to the tour. This works because of the following observation: The algorithm will continually recursively call itself until it arrives at a vertex with no more edges. Because all vertices have even degree, this must be the starting vertex. It then returns part of the tour until a point where other edges could be recursively

used (a "crossing"), and starts from there again until it must end at that vertex again. It then outputs more of the tour. Each series of outputs form a cycle, so the end result is a cycle, where by the nature of each edge being stepped once and then removed, must be eulerian.

3 Optimizing Christofides algorithm

3.1 2-Opt

To optimize the tour produced from the Christofides algorithm, we implemented a 2-opt local search algorithm, whose main idea is to take a tour with paths that intersect and reorder them in such a way that they do not. The 2-opt algorithm takes as input the TSP tour generated by Christofides and the original cost of that tour. Then, for 2 randomly selected and distinct edges, we calculate δ, the difference in the current distance of the edges, and the distance if we were to swap the edges. If δ is negative, that would mean that swapping the edges would result in a shorter distance. After swapping the edges, we construct the new tour and calculate the cost for it.

3.2 3-Opt

To optimize the tour further, we implemented a 3-opt local search algorithm. For this, we enumerated all the possible permutations of three edges that could not have been found by 2-opt. If any of these permutations have lower cost, we swap the edges and construct the new tour. The reversing and reordering process necessary to construct the new tour were optimised by creating two parallel arrays and copying the vertices in the new order from one to the other interchangeably. This removed the need for copying some of the same vertices twice (as would be necessary if we had only one array).

3.3 Tuning

A few observations were made to optimise the heuristic improvements of the tour and tune some hyperparameters. First, we noted that 2-Opt converged quite quickly to its local optimum. We therefore introduced a phase where only 2-Opt would be used. We also batched the runs of 2-Opt because measuring the time otherwise had significant cost. For the 3-Opt, we mixed it with some runs of 2-Opt to account for the fact that 3-Opt itself did not consider 2-Opt swaps and after a 3-Opt change, 2-Opt could have found some new simple updates. We also observed that usually, 3-Opts would be most likely profitable where vertices were close in the tour, so we introduced a sampling of vertices close together. This yielded significant improvements in our tests but only minimal changes to the Kattis score. We concluded that this was most probably due to Kattis having more smaller graphs on which this optimisation didn't help much and therefore we decided to use this optimisation only for large vertex sets. We then also left a chance that we would still view non-close vertices even

for large vertex sets. The hyperparameters we settled on through a process of educated guessing were: 500 ms 2-Opt in batches of 10000, the rest mixed 1000 2-Opt, 9000 3-Opt until 1990 ms, random vertices in 3-Opt if $n \leq 700$ and a 10% chance if above. The spread of view for close vertices $\frac{n}{10}$.

4 Tests

We used both the small handcrafted test-set from the Kattis page and a large $n = 1000$ randomly generated test-set in the bounding box 10^6.

References

[1] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum*, 3, 1976.

[2] William Cook and Andre Rohe. Computing minimum-weight perfect matchings. *INFORMS journal on computing*, 11(2):138–148, 1999.

[3] Vladimir Kolmogorov. Blossom v: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1:43–67, 2009.

[4] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389—1401, 1957.