KTH Royal Institute of Technology

School of Electrical Engineering and Computer Science

Project TSP Report

December 3, 2018

Autumn Term 2018

Navid Farhadi

Franz Fuchs

Arturs Kurzemnieks

Alexandre Peinot

Kattis submission ID: 3520421 (score: 35.02778)

1 Introduction

Given a set of n cities and the distances between them, the Travelling Salesman Problem (TSP) consists in finding the shortest circuit visiting all the cities only once. In this project, we study the Euclidian version of TSP in two dimensions. In this variant of the problem, all the cities are represented by their coordonnates x and y and the distance between two cities is the Euclidian distance between them (rounded to the nearest integer in the case of the project).

This report present the different algorithm we have implemented for obtaining the initial tour of this problem and for optimizing it. It shows the results obtained and compare the performance of the different solutions.

2 Implementation

2.1 Initial Tour

2.1.1 Christofides' Algorithm

The algorithm of Christofides is not a brandnew invention. It is an approximation algorithm that achieves to find a tour with worst case length $ALG \leq 1.5 \cdot OPT$. The Christofides algorithm consists of major parts that are explained in the following text.

Construction of a Minimum Spanning Tree

A Minimum Spanning Tree (MST) is subset of edges in a graph that produces a fully connected graph with minimum weight. It is important to note that there does not have to be a unique solution for the MST problem. There can be multiple sets of edges, which lead to equal total cost. There are several algorithms to solve that problem. The most common ones need $O(m \cdot log(n))$ time. One example is the algorithm of Prim. We decided to implement this one because one member of our group had previous experience with that algorithm. The algorithm of Prim has an initial node and creates the tree originating from that node. Once a vertex is processed the shortest path has been found and the incoming edge is added to the set of MST edges. Prim's algorithm is fairly simple to implement in C++ because the algorithm is based on a priority queue. The standard library for C++ offers various structs to use out of the box. We decided to use the `std::make_heap` function in combination with the basic data structure `std::vector<std::pair<int,double>>`

and a comparator struct that puts the smallest reachable vertex on the top of the heap.

Finding of Vertices with Odd Degree

After finding one solution for the MST problem, the algorithm of Christofides says to sort out all the vertices that do not have an odd degree in the MST edges set. This check can be done in $O(m+n)$. First we iterate over all MST edges and increase for the two vertices the degree. After that, we sort out all vertices with an even degree. Since the amount of MST edges is always less equal than the amount vertices, we can simplify the runtime of finding all odd vertices to $O(n)$. The formula $E_{MST} \leq n$ holds because of the construction criterion of MST.

Minimum-weight Perfect Matching

The next step of Christofides involves finding a minimum-weight perfect matching for the odd-degree subgraph acquired in the previous step. A perfect matching for a graph is a set of edges so that no vertices are shared between the edges but all vertices are covered, i.e., every vertex is covered by exactly one edge in the matching. By the Handshaking lemma, the odd-degree subgraph from the previous step must have an even number of vertices, therefore no verttices are left without a pairing and a perfect matching is always possible. As multiple different matchings can be possible, the aim of this step is to produce a minimum-weight matching with the least total cost of the edges.

Since this is a very complex problem, we opted to use an existing and publicly available implementation of the Edmond's Blossom algorithm. This implementation, called *Blossom V*, was described and developed by Vladimir Kolmogorov and follows a $O(n^2m)$ time bound [1]. We incorporated it into our codebase in a trimmed and minimized form, adding a wrapper function to map between the input and output format of the implementation and the data structures used by us.

Finally an Eulerian multigraph is constructed as a union $T \cup M$ of the perfect matching M and the previously obtained minimum spanning tree T. Given our choice of data structures, the union can easily be implemented as a concatenation of the two vectors representing the perfect matching and MST.

Eulerian Circuit & Algorithm Output

The final two steps of Christofides algorithm required us to implement an algorithm to check whether or not the given multigraph $T \cup M$ contains an Eulerian circuit. Prior to doing this, the data structure containing the graph $T \cup M$ is modified from `vector<array<int,2>>` to `unordered_map<int,vector<int>>`. This is a basic implementation of a seperate chaining hash table. The reason that we made the

decision to do this is because `unordered_map` has $O(1)$ look-ups which is useful when checking to see whether or not the graph contains an Eulerian circuit and finding the Eulerian circuit.

Next, we wrote a function to check whether or not the graph contains an Eulerian circuit. This was straightforward since we just needed to check whether or not each vertex has an even degree, since a graph contains an Eulerian circuit if and only if all vertices have an even degree. If the graph contains an Eulerian circuit, then we find the Eulerian circuit. This function was useful when testing our algorithm. In Christofides algorithm, $T \cup M$ is guaranteed to be given to you as an Eulerian multigraph so it isn't necessary to utilize this function in our final implementation, but we found that it did not effect the kattis score so we decided to keep it.

To find the Eulerian circuit we used a simple algorithm that runs in $O(n + m)$ [2]. Once we have an Eulerian circuit, we simply removed all repeated vertices which takes $O(n)$ time to give us the output of the algorithm as a hamiltonian circuit of the original input graph to TSP.

2.1.2 Nearest Neighbor

An initial tour can also be found with the nearest neighbor approach. Starting from a random initial vertex, the next vertex is determined in the following way:

- Compute the distance to each vertex that has not been visited yet
- Choose the vertex with the minimum distance

It can be shown that this approach leads to a tour with worst case length $ALG \leq 2 \cdot OPT$. Our implementation of the nearest neighbor algorithm has a runtime of $O(n^2)$ because there is an iteration over all vertices and another nested iteration over all vertices to find the one with the minimum distance to the current vertex.

2.1.3 Multiple Fragment

The Multiple Fragment (MF) approach also promises to give a good initial tour, but cannot hold as tight bounds as the algorithm of Christofides. The MF approach is based on calculating all edges and sorting them from the lowest weight to the highest one. Starting with the edge that has the lowest weight edges are picked and put into the solution set of edges if they do not create circle in the solution domain. Moreover, each vertex is only allowed to have one incoming and one outgoing edge. In that way, our implementation of the MF approach has a runtime of $O(n^3)$ because

it is possible to have to iterate through all edges and to check at the last edge an circle of length $(n-1)$.

2.2 Optimization

After the obtention of the initial tour, we need to run some optimization algorithm to get as close as possible to the optimal solution. In our project, we have focus our work on three special cases of k-opt: 2-opt, 2H-opt and 3-opt. All of these algorithms belong to the class of local search algorithms. The main issue with these algorithms is that they can easily fall in some local minimum and then never obtain the optimal path.

In practice, these algorithms must run until there is no new improvment. However, due to the Kattis time constraint, we have set a limit in the time execution in order to get the best possible path in the allowed time.

2.2.1 2-opt

The concept of the 2-opt algorithm is really simple. It basically consits in breaking two different edges of the tour and reconnecting the two obtained part of the tour in a way that minimize the distances. In this case, there are only two way of reconnecting the tour. The complete 2-opt algorithm iterates over all the possible pair of edge cutting in the tour.

Our implementation of the algorithm is $O(n^2)$ because it iterates through all pair of edge cutting. However, it is quite fast as it only evaluate the distances which could change by reversing the path between the two cuts and do the reverse only if it improves the tour's distance.

2.2.2 2H-opt

Also called 2.5-opt, this algorithm looks for sets of three vertices $\{t, u, v\}$ in the path and tries relocating the middle vertex u between some other two vertices $\{x, z\}$, and evaluating whether the new triad $\{x, u, z\}$ and directly connected $\{t, v\}$ are resulting in a shorter total path.

Our implementation iterates through all triads of the path, for each additionally iterating through the possible pairs for it to be compared against, resulting in $O(n^2)$ running time. As with 2-opt, the distance evaluation itself is quite fast, but modifying the tour if a movable vertex is found is considerably more expensive. As a

vector is being used for representation of the path, vertex u must be deleted from it, resulting in shifting of elements after its position. It is then reinserted between x and z, again shifting elements z and upwards.

2.2.3 3-opt

The 3-opt optimization consists in cutting 3 different edges of the path and concerving the shortest circuit over the 8 possibilities of reconnecting the three obtained part of the tour. The complete 3-opt algorithm iterates over all the possible set of 3 edge cutting in the tour. Thus, one execution of the algorithm is $O(n^3)$. Once again, we only evaluate, the distances which could change when reconnecting the circuit.

3 Evaluation

4 Conclusion

The Travelling Salesman Problem is a really simple problem in its formulation but it can be very difficult to find an optimal tour by using limited ressources (time, memory). That is why it is important to compute quickly an initial tour that is quite close to the optimal solution and choose well the optimization algorithms to use to go efficiently to the optimal tour.

References

[1] Vladimir Kolmogorov. "Blossom V: A new implementation of a minimum cost perfect matching algorithm." In Mathematical Programming Computation (MPC), July 2009, 1(1):43-67.

[2] Dumitru Ciubatii. "Eulerian Path and Circuit." Retrieved November 15, 2018, from http://www.graphmagics.com/articles/euler.php.