Traveling Salesman Problem: Approximation algorithms

Juan Da Silva

June 30, 2021

Contents

1 Introduction

In this report, we will be studying the ***Traveling Salesman Problem (TSP)***: Given an undirected graph $G(V, E)$ with non-negative integer cost $c(u, v)$, for all edges $(u, v) \in E$, find the hamiltonian cycle with minimum cost.

The TSP was mathematically formulated in the 1800s by the Irish mathematician W.R. Hamilton and by the British mathematician Thomas Kirkman. The general form of the TSP appears to have been studied by mathematicians in Vienna and Harvard throughout the 1930s, most notably by Karl Menger, who describes the problem, evaluates the obvious brute-force approach, and notices the non-optimality of the nearest neighbor heuristic:

> *We denote by messenger problem (since in practice this question should be solved by each postman, anyway also by many travelers) the task to find, for finitely many points whose pairwise distances are known, the shortest route connecting the points. Of course, this problem is solvable by finitely many trials. Rules which would push the number of trials below the number of permutations of the given points, are not known. The rule that one first should go from the starting point to the closest point, then to the point closest to this, etc., in general does not yield the shortest route.*

In 1976, Christofides and Serdyukov independently of each other made a big advance in this direction: the Christofides-Serdyukov algorithm yields a solution that, in the worst case, is at most 1.5 times longer than the optimal solution. As the algorithm was simple and quick, many hoped it would give way to a near optimal solution method. However, this hoped for improvement did not immediately materialize, and Christofides-Serdyukov remained the method with the best worst-case scenario until 2011, when a (very) slightly improved approximation algorithm was developed for the subset of "graphical" TSPs. In 2020 this tiny improvement was extended to the full (metric) TSP.

2 Metric TSP

TSP is an NP-complete problem, and therefore there is no known efficient solution. In fact, unless P=NP, there is no good approximation algorithm for the general TSP problem. By modify it slightly, there is a known 2-approximation algorithm for the ***metric TSP***. In metric TSP, we impose a metric $d(\cdot, \cdot) : X \times X \to \mathbb{R}, d(x, y) \mapsto \|x - y\|$. In metric TSP, the cost function satisfies the triangular inequality:

$$c(x, y) = d(x, y) \leq d(x, z) + d(z, y) \quad \forall x, y, z \in V$$

However, even with this constraint, the metric TSP is still an NP-complete problem.

2.1 MST Approximation Algorithm (Double-Tree Algorithm)

We already know how to find minimum spanning trees efficiently. Also, when you remove an edge from a Hamiltonian cycle, you get a spanning tree (not necessarily the minimum spanning tree). Using this idea, we create an approximation algorithm for minimum weight Hamiltonian cycle. The algorithm is as follows: Find the minimum spanning tree T of G rooted at some node r. Let H be a list of vertices visited in pre-order tree-walk of T starting at r. Return the cycle that visits the vertices in the order of H

We will now prove that the MST-based approximation is a 2-approximation for the metric TSP. Let H^* be the optimal Hamiltonian cycle of graph G, and let $c(R)$ be the total weight of all edges in R. Moreover, let $c(S)$ for a list of vertices S be the total weight of the edges needed to visit all vertices S in the order they appear in S.

Lemma 1. *$c(T)$ is a lower bound of $c(H^*)$.*

Proof. Removing any edge from H^* results in a spanning tree. Thus, the weight of MST must be smaller than that of H^*.

Lemma 2. *$c(S') \leq c(S)$ for all $S' \subset S$*

Proof. Consider $S' = S - v$. Without loss of generality, assume that vertex v was removed from a subsequence u, v, w of S. Then in S', we have $u \to w$ rather than $u \to v \to w$. By tringular inequality, we know that $c(u, w) \leq c(u, v) + c(v, w)$. Therefore, $c(S)$ is non-increasing, and $c(S') \leq c(S)$ for all $S' \subset S$.

Consider now the walk by traversing the tree in pre-order. This walk traverses each edge exactly twice, meaning $c(W) = 2c(T)$. Furthemore, we know that $c(T) \leq c(H^*)$. By Lemma 2, we know that $c(H) \leq c(W)$. Putting it all together, we have: $c(H) \leq c(W) = 2c(T) \leq c(H^*)$

2.2 Christofides Algorithm

The MST algorithm can be improved by slightly modifying the MST. Define an ***Euler tour*** of a graph to be a tour that visits every edge in the graph exactly once. As before, find the minimum spanning tree T of G rooted at some node r. Compute the minimum cost perfect matching M of all the odd degree vertices, and add M to T to create T'. Let H be the list of vertices of Euler tour of T' with duplicate vertices removed. Return the cycle that visits vertices in the order of H.

We will now prove that the Christofides algorithm is a $\frac{3}{2}$-approximation algorithm for the metric TSP. Note that an Euler tour of $T' = T \cup M$ exists because all vertices are of even degree. We now bound the cost of the matching M

Lemma 3. *$c(M) \leq \frac{1}{2}c(H^*)$*

Proof. Consider the optimal solution H' to TSP of just the odd degree vertices of T. We can break

H' to two perfect matchings M_1 and M_2 by taking every other edge. Because M is the minimum cost perfect matching, we know that $c(M) \leq \min(c(M_1), c(M_2))$. Moreover, because H' only visits a subset of the graph, $c(H') \leq c(H^*)$. Therefore, $2c(M) \leq c(H') \leq c(H^*) \Rightarrow c(M) \leq \frac{1}{2}c(H^*)$.

The cost of the Euler tour of T' is $c(T) + c(M)$ since it visits all the edges exactly once. We know that $c(T) \leq c(H^*)$ as before (Lemma 1). Using Lemma 3 along with Lemma 1, we get $c(T) + c(M) \leq c(H^*) + \frac{1}{2}c(H^*) = \frac{3}{2}c(H^*)$. Finally, removing duplicates further reduces the cost by the triangular inequality. Therefore, $c(H) \leq c(T') = c(T) + c(M) \leq \frac{3}{2}c(H^*)$

3 Euclidean TSP

If the distances between vertices can be arbitrary, we already know that no know that no approximation algorithm is possible. If the distances are metric, we have seen a 2-approximation algorithm by doubling a spanning tree and a $\frac{3}{2}$-approximation via Christofides' algorithm. This is the currently best known algorithm for this problem. Hence, we have to restrict the input to the problem to gain better algorithms. When the input numbers can be arbitrary real numbers, the euclidean TSP is a particular case of metric TSP, since distances in a plane obey the triangular inequality. When the input numbers must be integers, comparing lengths of tours involves comparing sums of square-roots. Like the general TSP, euclidean TSP is NP-hard. With rational coordinates and discretized metric, the problem is NP-complete.

3.1 Aroras algorithm

Definition 1. *An instance of euclidean TSP is called ϵ-nice if the following two conditions hold:*

- *Every point has integral coordinates in the interval $[0, O\left(\frac{n}{\epsilon}\right)]^2$*
- *Any two different points have distance at least 4*

It is important to note that without loss of generality, an input instance can be translated and scaled as we like, i.e., all points can be translated by the same vector and the coordinates of each point can be scaled by the same factor. However, the approximation ratio is preserved.

Without loss of generality, we can translate the box such that it is rooted at the origin and scale it such that $L = \lceil \frac{8n}{\epsilon} \rceil$

Lemma 4. *Let I be an arbitrary instance to euclidean TSP. Let $c(I^*)$ denote the cost of the optimum tour in I. We can transform I into an ϵ-nice instance $I_{\epsilon\text{-}nice}$ such that $c(I^*_{\epsilon\text{-}nice}) \leq (1+\epsilon)c(I^*)$*

Proof. Consider the smallest bounding box around the points of instance I. Set the largest side to L. Since the box was the smallest bounding box, there exists two points (on opposite sides of the box) of distance at least L. Hence, the optimum tour is of size at least L. In fact, since the tour has to travel forth and back between those two points, we have that $c(I^*) \geq 2L$.

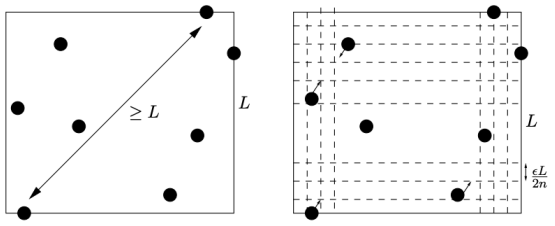

Figure 1: Bounding box with length L and scaled box with fine grid

For ease of exposition, extend the bounding box such that both sides have length L. Now, to obtain instance $I_{\epsilon\text{-nice}}$ we draw a fine grid with spacing $\frac{\epsilon L}{2n}$ into the bounding box and map each point to its closest grid point.

Lets verify that this new instance $I_{\epsilon\text{-nice}}$ is indeed ϵ-nice. Clearly, all points lie on integer coordinates in the respective range since $L = \lceil \frac{8n}{\epsilon} \rceil \in O(n/\epsilon)$. [Answering the question posed during the seminar, when I say that all points lie on integer coordinates, it is meant that all mapped points lie on the mapped integer coordinates]. Furthemore, the grid spacing is $\frac{\epsilon L}{2n} \geq \frac{\epsilon}{2n}\frac{8n}{\epsilon} = 4$ and therefore the distance between any two different points in $I_{\epsilon\text{-nice}}$ is at least 4. Thus, $I_{\epsilon\text{-nice}}$ is ϵ-nice.

It remains to prove that the optimal tours in I and $I_{\epsilon\text{-nice}}$ differ by at most a factor of $(1+\epsilon)$. Lets consider the optimum tour in I. By mapping the points of I to the points in $I_{\epsilon\text{-nice}}$ we moved every point by at most a distance of $\frac{\epsilon L}{2n}$. Thus, every edge between two points changed by at most $\frac{\epsilon L}{n}$. Since there are n edges in any tour, the change in cost is at most ϵL, i.e., we obtained a tour for $I_{\epsilon\text{-nice}}$ of cost at most $c(I^*) + \epsilon L$. Using the fact that $L \leq c(I^*)$, we obtain:

$$c(I_{\epsilon\text{-nice}}) \leq c(I^*) + \epsilon L \leq (1+\epsilon)c(I^*)$$

4 Real-World Applications

TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. The numerous direct applications of the TSP bring life to the research area and help to direct future work.

In the git repository has been implemented a program for route planning using the Christofides algorithm. On the *main.ipynb* file, it is possible to visualize all steps of the algorithm graphically by setting the boolean variable 'debug' to True.

The program is integrated with Google API that can get the geographical location and the distance of the points of interest. Moreover, it allows to display the route between those points.

Furthemore, Google API works in different modes (i.e. driving, walking, bycicling, transit). This

feature gives a more precise calculations of the distances between points of interest depending on how the person plans to travel. This makes this program even more interesting as it encompasses more target people.

Thanks to Google API, it is possible to construct a distance matrix. Even though the distance matrix is not symmetric, it can be modified just by setting the lower triangular part of the matrix equal to the upper triangular part of it.

The algorithm is implemented as explained in detail in section 2.2. Lets review shortly the algorithm:

1. Minimum Spanning Tree
2. Odd Degree Vertices
3. Minimum Weight Perfect Matching
4. Eulerian Circuit
5. Hamiltonian Circuit

The *GoogleMaps* class permits the interaction with the Google API. Concretely, it accesses to the *GooglePlaces* and *googlemaps* libraries. *GoogleMaps* objects are able to search for places, access to the data of different places of interest given the latitude and longitude of this ones, and get address recommendations for the searched places.

The core file of this program is *graph.py*. It accesses to methods of the library *networkx* to facilitate the manipulation and operations with graphs (add/get edges, add/remove/get nodes, get Euler tour, among others). It is the core file in this program, since the problem of finding a circuit that travels through all points of interest is translated into a graph were the points of interest are the vertices and the paths between cities are represented by edges.

To translate such a real world problem into a graph, a function to create a graph from a numpy matrix has been implemented. Note, it is also possible to create a graph from a list of tuples with reference to the node numbers and the edge between nodes. This makes the program not only applicable to the real world application that we are trying to solve (route planning) but also for a general problem with no relation to the original idea. Example of the list of inputs can be found in the *input* folder.

Kruskal's algorithm for finding the minimum spanning tree and some fundamental methods as the minimum weight perfect matching, more complex graph operations like the union of graphs and the calculation of odd degree vertices have been implemented on the *utils.py* file.