A Heuristic Solution to the Metric Travelling Salesperson Problem

Rosie Bartlett
lvff38@durham.ac.uk

November 4, 2021

Abstract

In this paper I will present a heuristic solution to a variant of the TSP(travelling salesperson problem) knows as the metric TSP or ΔTSP , where all points can be represented on a two dimensional Euclidian plane. This new approach uses an approach inspired by physics simulations, where we stretch a path around the set of points s.t. all points are either on this path or bounded inside the polygon of this path. Through bisectors of edges, margins, and normals with edges through points, the remaining points are placed along edges, and thus finding a heuristic solution to the given problem

Contents

1 A introduction to the ΔTSP

The travelling salesperson problem(TSP) is a problem based on finding the shortest path through points. This is typically given as a set of distances between points. As such, the TSP is typically considered to be a problem falling under the branch of graph theory. There is no known origin of the problem (mathematically or otherwise).

Exact solutions to the TSP are quite difficult to come by. One method is brute force, taking every possible route and finding the shortest one. This has a time complexity of $O(n!)$, so becomes exceedingly slow and impractical the more points are included. Various other exact solutions have been proposed improving on the brute force method such as the Held-Karp algorithm which has a time complexity of $O(n^2 2^n)$[1].

Many heuristic algorithms have also been discovered for the TSP. One of note is the ant colony system. Described in 1993 by Marco Dorigo, this algorithm copies the behaviour of ants in nature when travelling to and from food sources.

Instead of finding exact or heuristic solutions for the TEP, many researchers have found solutions (exact or heuristic) for special cases of the TSP. One of these special cases if the metric TSP or ΔTSP. This special case occurs when all points can be positioned on a 2 dimensional plane and retain the correct distances between all points. This could also be described as all points obeying the triangle rule. This states that for three unique points p_a, p_b, and p_c:

$$|p_b - p_a| + |p_c - p_b| \geq |p_c - p_a| \tag{1}$$

Notation	Description
P	Matrix of points
p_i	The point $(p_{i,1}, p_{i,2})$
$x(p_i)$	The x coordinate of point p_i
$y(p_i)$	The y coordinate of point p_i
$\theta_{i,j}$	The angle of the line from p_i to p_j where $\theta = 0$ is the horizontal and $\theta_{i,j} \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
$R(\theta)$	The rotation matrix $\begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}$
n	The total number of points
B	Set of indexes representing the bounding polygon
b_i	The position (not index) of the i^{th} point in B
e_i	The edge from b_i to b_{i+1}
R	The set of indexes of points in P but not in B, $R := [1, n] \setminus B$
s_i	The set of indexes of points that are bounded by the bisector triangle associated with edge e_i

Figure 1: Notation used throughout this paper

[1]It is worth noting here that whilst the algorithm improves on time complexity, the space complexity does increase from $O(n^2)$ (brute force) to $O(n2^n)$ (Held-Karp)

2 The proposed algorithm

I will now present my proposed heuristic algorithm to a general ΔTSP problem. The algorithm is in two sections, the bounding section and compression section, and will be detailed as such. This section will also use an example matrix to demonstrate the two sections of this algorithm. The original matrix is shown in fig. 2

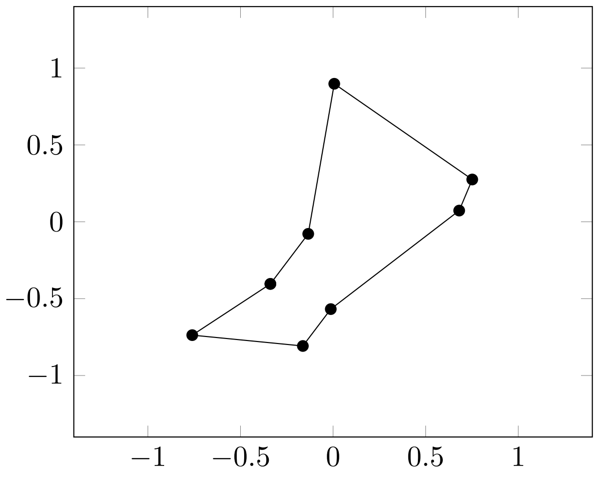


Figure 2: Solved problem with $n = 8$ and a shortest path distance of 1.86633 to 5 decimal places. This will be used as the example matrix throughout

2.1 Inspiration

This algorithm is inspired by considering a ΔTSP problem as a set of immovable points enclosed in a flexible divider, separating two areas of differing pressure. The solution to the problem being the path of the divider when changing the pressure difference to have a higher pressure on the outside of the divider. This will obviously not be the exact solution, and simulating this would be difficult to put it mildly. Instead, I have taken several major concepts from this idea and used it to inspire this algorithm.

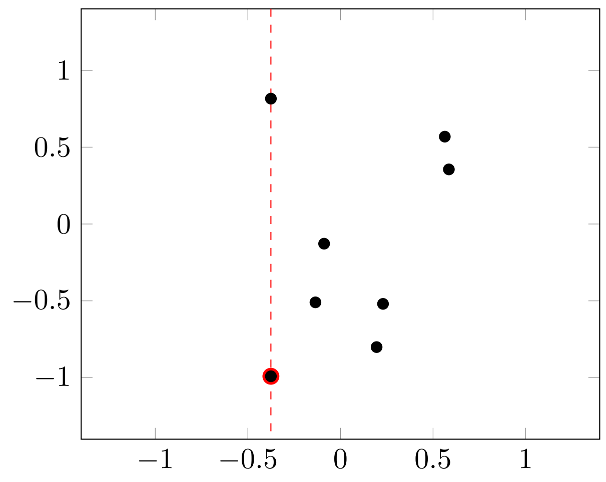
2.2 Bounding process

The bounding process is an iterative process designed to find a bounding path around P s.t. all points not included on the bounding polygon are enclosed by it. It begins as follows:

1. Find the furthest left point p_l s.t. $l = \underset{t\in[1,n]}{\text{argmin}}\ x(p_t)$
2. Find the point p_b s.t. $b = \underset{t\in[1,n],t\neq l}{\text{argmax}}\ \theta_{l,t}$, giving us the previous point in the bounding polygon.
3. We then rotate P by $\theta_{b,l}$ (not $\theta_{l,b}$) meaning that P is now $PR(\theta_{b,l})$

The result of this initial process on the example set is shown in fig. 3a
Once this first step is complete, the iterative process begins and is as follows:

1. If $l = \underset{t\in[1,n]}{\text{argmin}}\ y(p_t)$, then $P = PR(\frac{\pi}{2})$


(a) P after the first rotation with the starting point highlighted, and left bounding line

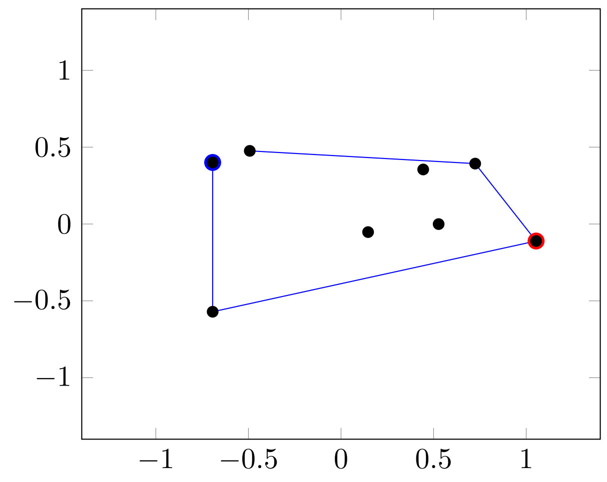

(b) P after the bounding process, with the bounding polygon in blue, with the first bounding point highlighted in blue and the original starting point highlighted for reference.

Figure 3: P just before the iterative stage, and at the end, of the bounding process with important points displayed

2. The next index of the bounding points x is defined to be $x = \underset{t\in[1,n],t\neq l}{\operatorname{argmin}} \ \theta_{l,t}$. This value is appended to the bounding points array and subsequently replaces l, thus making the new p_l be the same as the current p_x
3. Rotate P by $\theta_{l,x}$ s.t. $P = PR(\theta_{l,x})$
4. Redefine l as x to update the leftmost point
5. Repeat the process until $x = b$

At this point the bounding array is rolled 2 places, meaning that the last two values are moved to the front of the array[2]. This places the current first point as the top point on the edge of B currently perpendicular to the vertical as seen in fig. 3b.

The result of the entire process can be see in fig. 3.

2.3 Compression process

This is the final section of the algorithm,

2.3.1 Bisector Triangle Bounding

In this section we will consider the general case of the edge e_i, aligned with the vertical through matrix rotation, and the set of points s_i for which each element $s_{i,j} \in R$. We will also be mostly considering this section in polar coordinates with b_i as the pole, only using Cartesian coordinates for matrix rotation.

In the following, I will refer heavily fig. 4 to show angles, bisectors, and lines to better explain this section. I will refer to these sections in brackets as such (fig. 4 description of attribute).

[2]The exact function used here is `np.roll`, documentation for which is available at `https://numpy.org/doc/stable/reference/generated/numpy.roll.html`

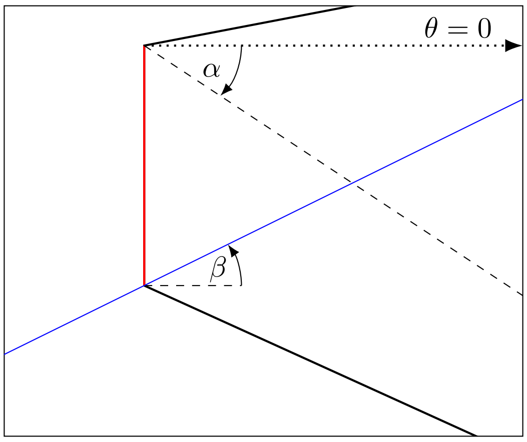


Figure 4: Example bounding polygon section to illustrate various angles and bisectors

We will consider the general case for the edge e_i (fig. 4 vertical red edge). through matrix rotation and how the edge was chosen previously, the following properties are true:

1. All points not on e_i are to the right of the edge
2. b_i (fig. 4 upper point on e_i) is above b_{i+1} (fig. 4 lower point on e_i)
3. e_i is perpendicular to the vertical

We begin by finding α, which is the angle of the angle bisector of e_i and e_{i-1}, and β as the angle of the angle bisector og e_i and e_{i+1}. We then calculate $\theta_{b_i,j} \forall j \in R$ and remove and points for which $\theta_{b_i,j} > \alpha$. We then use eq. (2) to plot the second bisector (fig. 4 blue line) as a polar equation.

$$r(\theta) = ||b_{i+1} - b_i||_2 \frac{\sin\left(\beta - \frac{\pi}{2}\right)}{\sin(\theta - \beta)} \tag{2}$$

$$r(\theta_{b_i,j}) \geq ||p_j - b_i||_2 \forall j \in R \tag{3}$$

From both of these we can then define s_i in eq. (4):

$$s_i = \{j : r(\theta_{b_i,j}) \geq ||p_j - b_i||_2, \theta_{b_i,j} \leq \alpha \forall j \in R\} \tag{4}$$

This process is then repeated for all edges.

2.3.2 Edge compression

A Source code

The following is the source code for the proposed algorithm. It has been written in Python 3.9 using Numpy 1.21.3. No other versions have been tested. All the source code, along with the generated test sets used, are available on GitHub (`https://github.com/RosiePuddles/HeuristicTSP`).

```python
import csv
import sys

import matplotlib.pyplot as plt
import numpy as np


def rotate(angle) -> np.ndarray:
    return np.array([[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.cos(
    angle)]])


def solve(points: np.ndarray):
    """
    Function to heuristically solve a metric TSP problem

    :param points: flattened points in the range [-1,1]
    :return:
    """
    plt.xlim(-1.4, 1.4)
    plt.ylim(-1.4, 1.4)
    points = points.reshape((len(points) // 2, 2))
    # Optional randomisation/shuffling of the original set
    # np.random.shuffle(points)

    # elastic band bounding
    left = np.argmin(points[:, 0])
    points_above = points[np.greater(points[:, 1], points[left][1])]
    first = np.argmax(np.divide(*np.flip(np.subtract(points_above, points[left])
    , axis=1).transpose()))
    theta = np.arctan(np.divide(*np.flip(np.subtract(points[first], points[left
    ])))) - np.pi / 2
    points = np.dot(points, rotate(theta))
    bounding = np.array([left])
    theta_all = np.array([])
    rotated = np.array([])
    while left != first:
        if np.argmin(points[:, 1]) == left:
            rotated = True
            points = np.dot(points, np.array([[0, -1], [1, 0]]))
        left_point = points[left]
        indexes = np.less(points[:, 1], left_point[1])
        points_below = points[indexes]
        bounding = np.append(bounding, [
            np.where(indexes)[0][
                np.argmin(np.divide(*np.flip(np.subtract(points_below,
    left_point), axis=1).transpose()))
            ]], axis=0)
        theta = np.arctan(np.divide(*np.flip(points[bounding[-1]] - left_point))
    ) + np.pi / 2
        theta_all = np.append(theta_all, [theta]) + (np.pi / 2 if rotated else
    0)
        rotated = False
        left = bounding[-1]
        points = np.dot(points, rotate(theta))
    bounding = np.roll(bounding, 2)
    theta_all = np.append([
        np.arctan(np.divide(*np.flip(points[bounding[1]] - points[bounding[0]]))) + np.pi / 2
    ], theta_all)
    plt.plot(*np.append(points, [points[0]], axis=0).transpose(), 'ro--')
    plt.plot(*points[bounding].transpose(), 'b-')

    # Compression
    # Bisector triangle bounding
    remaining = np.delete(np.arange(len(points)), bounding)
    bisector = lambda x: (np.pi - x) / 2
    bounded_indexes = np.array([[]])
    for i in range(len(bounding)):
        pole = points[bounding[i]]
        first_bisector = -bisector(theta_all[i - 1]) / 2
        second_bisector = bisector(theta_all[i])
        point_angles = np.arctan(np.divide())

    plt.show()
    sys.exit(0)


if __name__ == "__main__":
    with open("tests/8.csv", "r") as f:
        read = csv.reader(f)
        for row in read:
            solve(np.array([float(i) for i in row]))
```