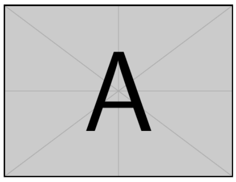
Does the Euclidean TSP intersect the NNG?

Gaurish Telang
gaurish108@gmail.com

October 15, 2020
9:22am



SYNOPSIS

Does the Euclidean TSP for a finite set of points P share an edge with P's nearest neighbor graph? [1] Or its k-NNG? Or the Delaunay Graph? Or indeed any poly-time computable graph spanning the input points? We investigate this question experimentally by checking the validity of this conjecture for various instances in TSPLIB, for which the optimal solutions have been provided.

DESCRIPTION

This question suggested itself to the author while working on the Horsefly problem, itself is a generalization of the famously NP-hard Travelling Salesman Problem [2]. One line of attack was to get at some kind of "structure theorem" by identifying a candidate set of "good" edges from which a near-optimal solution to the horsefly problem could be constructed. But first off, would this approach work for the special case of the TSP? Answering *"$TSP \cap NNG \stackrel{?}{=} \varnothing$"* seemed like a good place to start. However, all attempts at constructing counter-examples in which the intersection is *empty* have, thus far, failed. And so has a cursory literature search. Bill Cook (the author of Concorde) on hearing about this problem from Prof. Mitchell said that, if true, it could be used to speed up some of the existing TSP heuristics.

To spur our intuition, we investigate the conjecture experimentally in this short report [3] using TSPLIB and Concorde in tandem. TSPLIB is an online collection of medium to large size instances for the Euclidean, Metric and other several variants of the TSP for which optimal solutions have been obtained using powerful heuristics implemented in libraries like Concorde or Keld-Helsgaun; the certificate of optimality for these instances (as always!) comes from comparing the tour-length of the computed against a lower bound computed by those very heuristics.

[1] In this article, we will assume the NNG to be undirected i.e. after constructing the nearest neighbor graph for a point-set we will throw away the directions of the edges.

[2] In this report by "TSP", we mean TSP-cycle and not TSP-path, although the question is still interesting for the path case. One reason for focusing only on the path case, is that the TSPLIB bank only mentions optimal cycle solutions and not optimal path solutions, which can be structurally quite different! Also Concorde, the main library used to generate any TSP solutions also outputs cycles.

[3] This report has been written as a literate program to weave together the code, explanations and generated data into the same document. Feedback on the author's preliminary stab at literate programming is most welcome!

For starters, we investigate the following questions [4]: for each symmetric 2-D Euclidean TSP instance from TSPLIB for which we have an optimal solution, does

- ❖ $TSP \cap (k\text{-})NNG \stackrel{?}{=} \varnothing$, for $k = 1, 2, \ldots$
- ❖ $TSP \cap \textit{Delaunay Graph} \stackrel{?}{=} \varnothing$
- ❖ For question 1, in the cases that the intersection is non-empty, what fraction (a fourth?, a third?)of the n edges of a TSP-tour share its edges with the k-NNG does the TSP intersect for various values of k?
- ❖ Are there any structural patterns observed in the intersections? Specifically, does *at least* one edge of the nearest neighbor graph have an edge with a *vertex* incident to the convex hull? [5] More generally, is this true for every layer of the "onion"?

See also Appendix A for a running wishlist of questions that come out during discussions.

As an aid in constructing possible counter-examples, a GUI interface is provided to mouse-in points and then run the Concorde heuristic on it.

The Python 3.7+ code used to generate the data and figures in this paper has been attached to this pdf. If you don't have a Python distribution please download the freely available Anaconda distro, that comes with all the "batteries included".

Instructions for running the code have been relegated to the appendix. All development and testing was done on a Linux machine; minimal modification (if at all!) would be needed to run it on Windows or Mac. In any event, the boring technical issues can be hashed out on Slack.

Yalla, let's go!

[4] Experimental answers to other questions will be barnacled to the report as it keeps (hopefully!) growing

[5] This indeed seemed to be the case in all the author's failed attempts at a counter-example, and so are looking for a proof/disproof for this special vase of the conjecture

Contents

1 Overall structure of `tspnng.py`

The `tspnng.py` file at a high level divided into the following chunks, each of which is expanded upon in the coming sections. The `main.py` file used to run the `main()` function from the command-line is more of a scratchpad for testing the functions in this file, and later pointing the main to the appropriate test harnesses inside the `tspnng.py` file. Hence `main.py` will be developed independently of this document for convenience because it will be subject to continuous changes. .

3 ⟨*tspnng.py* 3⟩≡

⟨*Header statements* 4a⟩
⟨*Data Generation* 4b⟩
⟨*Generic utility classes and functions* 6a⟩
⟨*Functions for plotting* 6b⟩
⟨*Functions for generating various graphs* 6c⟩
⟨*Functions for testing various hypotheses* 6d⟩

4a ⟨*Header statements* 4a⟩≡ (3)

```
from matplotlib import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import rc
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

import scipy as sp
import numpy as np
import random
from colorama import Fore, Back, Style
```

2 Data Generation

Alongside TSPLIB we will principally be using synthetic data i.e.uniform and non-uniform point-sets generated inside the unit-square $[0, 1] \times [0, 1]$. Note that each point is represented as a numpy array of size 2.

4b ⟨*Data Generation* 4b⟩≡ (3)

⟨*Synthetic data* 5a⟩
⟨*TSPLIB data* 5b⟩

This chunk generates uniform and non-uniform point sets in $[0,1] \times [0,1]$. To generate non-uniform point-sets we basically take a small set of uniformly distributed random points in the square, place a small square centered around each such random point and then generate the appropriate number of points uniformly inside each of those squares. [6]

5a ⟨*Synthetic data* 5a⟩≡ (4b)

```
def uniform_points(numpts):
     return  sp.rand(numpts, 2).tolist()

def non_uniform_points(numpts):

    cluster_size = int(np.sqrt(numpts))
    numcenters   = cluster_size
    centers      = sp.rand(numcenters,2).tolist()
    scale, points = 4.0, []

    for c in centers:
        cx, cy = c[0], c[1]
        sq_size      = min(cx,1-cx,cy, 1-cy)

        loc_pts_x    = np.random.uniform(low  = cx-sq_size/scale,
                                         high = cx+sq_size/scale,
                                         size = (cluster_size,))
        loc_pts_y    = np.random.uniform(low = cy-sq_size/scale,
                                         high = cy+sq_size/scale,
                                         size = (cluster_size,))

        points.extend(zip(loc_pts_x, loc_pts_y))

    num_remaining_pts = numpts - cluster_size * numcenters
    remaining_pts = scipy.rand(num_remaining_pts, 2).tolist()
    points.extend(remaining_pts)
    return points
```

This chunk principally just deals with massaging the TSPLIB data into a format appropriate for the current code.

5b ⟨*TSPLIB data* 5b⟩≡ (4b)

[6] A similar technique was used in Jon Bentley's experimental TSP paper

YAML[BKEI09] is a convenient serialization and data-interchange format that we will be using principally for serializing data onto disk. Python has particularly good libraries in dealing with YAML. Basically, YAML stores data similar to a Python dictionary. Infact the `yaml` module provides an function to transparently encode any (appropriate) Python dictionary into a YAML file. In the function below, the `data` argument is a dictionary, and `dir_name` and `file_name` are strings.

6a ⟨*Generic utility classes and functions* 6a⟩≡ (3)

```
def write_to_yaml_file(data, dir_name, file_name):
    import yaml
    with open(dir_name + '/' + file_name, 'w') as outfile:
            yaml.dump( data, outfile, default_flow_style = False)
```

6b ⟨*Functions for plotting* 6b⟩≡ (3)

6c ⟨*Functions for generating various graphs* 6c⟩≡ (3)

6d ⟨*Functions for testing various hypotheses* 6d⟩≡ (3)

2 References

[BKEI09] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. "Yaml ain't markup language (yaml™) version 1.1". In: *Working Draft 2008-05* 11 (2009).

Appendices

I Complete installation instructions

On GNU/Linux systems

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Windows 10 + Cygwin

In this example several keywords will be used which are important and deserve to appear in the Index. Terms like generate and some will also show up. Terms in the index can also be nested

II Reading Chunk numbers

From left to right. This is how you should learn to read numbers generated by noweb when definiing a new or extending a chunk.

III Layout of source files

Generate this using some bash commands to give overview of the project tree.