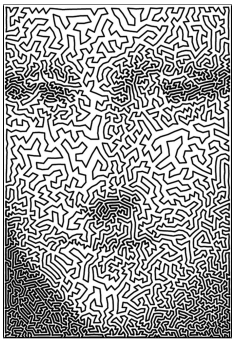
Does the TSP intersect the NNG?

Gaurish Telang
gaurish108@gmail.com



October 16, 2020
3:57am

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 and for other synthetic data-sets (e.g. uniformly and non-uniformly generated points) for which we can compute optimal or near-optimal tours using Concorde.

DESCRIPTION

This question suggested itself to the author while working on the Horsefly problem, 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 examples in which the intersection is *empty* failed . And so did a literature search. The closest matching reference we found was [HS14] which *eliminates* edges that cannot be part of a Euclidean TSP tour on a given instance of points, based on checking a few simple, local geometric inequalities.

Bill Cook, the author of Concorde[App+09], on hearing about this problem from Prof. Mitchell said that, if true, it could be used to speed up some of the existing experimental TSP heuristics. [3]

To spur our intuition, we investigate the conjecture experimentally in this short report [4] 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;

[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 edge directions.

[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] Note that the landmark PTAS'es for the TSP, such as those of Mitchell [Mit99] and Arora[Aro96], are too complicated to be put into code (yes, even Python!). On the other hand, the Concorde library uses a whole kitchen-sink of practical techniques such as k-local swaps, branch-and-bound, branch-and-cut to generate near-optimal (if not optimal) tours relatively quickly. However,it would be interesting to investigate the behavior of the various graphs with respect to the techniques used in the PTAS'es of Mitchell and Arora. Maybe we can augment them with the probabilistic method to prove the existence of an intersection??

[4] This report has been written as a literate program [Knu84; Ram08] 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!

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.

For starters, we investigate the following questions [5]: 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 fifth?)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 from the intersection have a *vertex* incident to the convex hull? [6] 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; it comes with most of the batteries included. You will also need to install a couple of other packages. See Appendix I.

Yalla, what are we waiting for?! Let's go!

[5] Experimental answers to other questions will be barnacled onto the report as it grows

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

Contents

Page

1 Overall structure of `tspnng.py` 4

2 Data Generation 5

3 Data Storage 7

4 Setting up TSPNNGInput class 7

5 Setting up the Interactive Canvas 8

6 Generating various geometric graphs 14

6.1 k-NNG 14

6.2 Minimum Spanning Tree 14

6.3 Delaunay Triangulation 15

6.4 TSP tour 15

7 Rendering the graphs 16

8 Finding common edges between two graphs 16

9 Hypothesis testing! 16

Appendices 17

Appendix I Running the Code 17

Appendix II Laundry-list of Questions/Variants/Conjectures 18

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. .

4a ⟨*tspnng.py* 4a⟩≡

⟨*Headers* 4b⟩
⟨*Data Generation* 5⟩
⟨*Generic utility classes and functions* 7a⟩
⟨*Functions for plotting and interacting* 8⟩
⟨*Functions for generating various graphs* 14a⟩
⟨*Functions dealing with intersecting two geometric graphs* 16b⟩
⟨*Testing hypotheses* 16c⟩

4b ⟨*Headers* 4b⟩≡ (4a)

```
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
import networkx as nx

from sklearn.cluster import KMeans
import argparse, sys, time
from colorama import Fore, Style, Back
```

2 Data Generation

Alongside TSPLIB we will also be using synthetic data-sets 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.

5 ⟨*Data Generation* 5⟩≡ (4a)
⟨*Synthetic data* 6a⟩
⟨*TSPLIB data* 6b⟩

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. [7] The size of the square is proportional to the distance of the sampled point from the boundary of the unit square. Thus you will often see tight clusters near the boundary as you increase the number of input points (`‘numpts‘`). This was done to make sure all points get generated in the unit square. This would make it convenient for the purposes of plotting. Other non-uniform point-generation schemes will later be considered depending on which direction our investigation proceeds.

6a ⟨*Synthetic data* 6a⟩≡ (5)

```
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 = sp.rand(num_remaining_pts, 2).tolist()
    points.extend(remaining_pts)
    return points
```

This chunk principally just reads in TSPLIB data and massages it into a format appropriate for the current code.

6b ⟨*TSPLIB data* 6b⟩≡ (5)

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

3 Data Storage

YAML[BKEI09] is a convenient serialization and data-interchange format that we will be using for serializing output data of different experiments onto disk. Python has particularly good libraries for dealing with YAML. Basically, YAML records data in a format similar to a Python dictionary. Infact the `yaml` module provides a function that transparently encodes 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.

7a ⟨*Generic utility classes and functions* 7a⟩≡ (4a) 7b▷

```
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)
```

4 Setting up TSPNNGInput class

The following class is used to keep track of the points inserted thus far, along with any other auxiliary information. It basically functions as a convenience wrapper class around the main input data (basically a bunch of points in $\mathbb{R}^2$) and a wrapper function around various graph generators such as TSP, Delaunary, k-NNG etc.

7b ⟨*Generic utility classes and functions* 7a⟩+≡ (4a) ◁7a

```
class TSPNNGInput:
      def __init__(self, points=[]):
          self.points            = points

      def clearAllStates (self):
          self.points = []

      def generate_geometric_graph(self,graph_code):
           pass
```

5 Setting up the Interactive Canvas

The following set of code blocks create an interactive matplotlib canvas onto which the user can insert points, and then run the appropriate algorithm to visualize the intersection of the TSP and various graphs.

We first set up the run handler function (each "run" corresponds to a run of the code on a particular data-set generated synthetically) by connecting the keyboard and mouse handlers to the canvas.

8 ⟨*Functions for plotting and interacting* 8⟩≡ (4a) 9▷

```
def run_handler():
    fig, ax =  plt.subplots()
    run = TSPNNGInput()

    ax.set_xlim([xlim[0], xlim[1]])
    ax.set_ylim([ylim[0], ylim[1]])
    ax.set_aspect(1.0)
    ax.set_xticks([])
    ax.set_yticks([])

    mouseClick   = wrapperEnterRunPointsHandler(fig,ax, run)
    fig.canvas.mpl_connect('button_press_event' , mouseClick )

    keyPress     = wrapperkeyPressHandler(fig,ax, run)
    fig.canvas.mpl_connect('key_press_event', keyPress   )
    plt.show()
```

There are two principal callback functions `wrapperEnterRunPointshandler` and `wrapperkeypresshandler` used in the code above. These encode the interaction between the mouse and keyboard to the matplotlib canvas.

First we define the call back function for mouse-clicks. Double-clicking the left mouse button (denoted as "button 1" in the matplotlib world) inserts a small circle patch representing a point.

9 ⟨*Functions for plotting and interacting* 8⟩+≡ (4a) ◁8 10▷

```
xlim, ylim = [0,1], [0,1]
def wrapperEnterRunPointsHandler(fig, ax, run):
    def _enterPointsHandler(event):
        if event.name      == 'button_press_event'     and \
           (event.button   == 1)                       and \
            event.dblclick == True                     and \
            event.xdata  != None                       and \
            event.ydata  != None:

             newPoint = np.asarray([event.xdata, event.ydata])
             run.points.append( newPoint  )
             print("You inserted ", newPoint)

             patchSize  = (xlim[1]-xlim[0])/140.0

             ax.add_patch( mpl.patches.Circle( newPoint, radius = patchSize,
                                               facecolor='blue', edgecolor='black'  ))
             ax.set_title('Points Inserted: ' + str(len(run.points)), \
                          fontdict={'fontsize':25})
             applyAxCorrection(ax)
             fig.canvas.draw()

    return _enterPointsHandler
```

Now a call-back function for keyboard. Pressing `'i'` or `'I'` on the keyboard further prompts the user to insert a 2 or 3 letter code to indicate which graph should span the points.

10 ⟨*Functions for plotting and interacting* 8⟩+≡ (4a) ◁9 13▷

```
def wrapperkeyPressHandler(fig,ax, run):
        def _keyPressHandler(event):
                if event.key in ['i', 'I']:
                        ⟨Enter spanning graph 11⟩
                elif event.key in ['n', 'N', 'u', 'U']:
                        ⟨Enter type of point set 12a⟩
                elif event.key in ['c', 'C']:
                        ⟨Clear all states and the canvas 12b⟩

        return _keyPressHandler
```

We now elaborate on the chunks in `wrapperkeypresshandler`, and implement the boring technicalities. You can skip ahead to the next sections, at this point, if you wish.

The user should type the code enclosed in the brackets (e.g. '`dt`' for delaunay triangulation) to generate the indicated graph that spans the points.

11 ⟨*Enter spanning graph* 11⟩≡ (10)

```
algo_str = input(Fore.YELLOW + "Enter code for the graph you need to span the points:\n" + Sty
                     "(knng) k-Nearest Neighbor Graph        \n"            +\
                     "(mst)  Minimum Spanning Tree           \n"            +\
                     "(dt)   Delaunay Triangulation         \n"             +\
                     "(tsp)  TSP\n")
algo_str = algo_str.lstrip()

if algo_str == 'knng':
      k_str = input('===> What value of k do you want? ')
      k     = int(k_str)
      geometric_graph = get_knng_graph(run.points,k)

elif algo_str == 'mst':
     geometric_graph = get_mst_graph(run.points)

elif algo_str == 'dt':
      geometric_graph = get_delaunay_tri_graph(run.points)

elif algo_str == 'tsp':
     geometric_graph = get_tsp_graph(run.points)

else:
      print(Fore.YELLOW, "I did not recognize that option.", Style.RESET_ALL)
      geometric_graph = None

render_graph(geometric_graph,fig,ax)
fig.canvas.draw()
```

If you want to enter a uniformly or non-uniformly distributed point-set in the unit-square press ‘u‘ or ‘n‘ respectively after being prompted.

12a ⟨*Enter type of point set* 12a⟩≡ (10)

```
numpts = int(input("\nHow many points should I generate?: "))
run.clearAllStates()
ax.cla()
applyAxCorrection(ax)

ax.set_xticks([])
ax.set_yticks([])
fig.texts = []

if event.key in ['n', 'N']:
        run.points = non_uniform_points(numpts)
else :
        run.points = uniform_points(numpts)

patchSize  = (xlim[1]-xlim[0])/140.0

for site in run.points:
    ax.add_patch(mpl.patches.Circle(site, radius = patchSize, \
                facecolor='blue',edgecolor='black' ))

ax.set_title('Points : ' + str(len(run.points)), fontdict={'fontsize':40})
fig.canvas.draw()
```

If you want to wipe the canvas and the data inside the run clean, then as indicated in the following code chunk just press ‘c‘.

12b ⟨*Clear all states and the canvas* 12b⟩≡ (10)

```
run.clearAllStates()
ax.cla()

applyAxCorrection(ax)
ax.set_xticks([])
ax.set_yticks([])

fig.texts = []
fig.canvas.draw()
```

Often the `ax` object has to be reset and cleaned of the various segment and circle patches, or even resetting the aspect ratio of the `ax` object to be 1.0. These "cleanup" functions that were called in some of the code blocks above are implemented next.

13 ⟨*Functions for plotting and interacting* 8⟩+≡ (4a) ◁10 16a▷

```
def applyAxCorrection(ax):
      ax.set_xlim([xlim[0], xlim[1]])
      ax.set_ylim([ylim[0], ylim[1]])
      ax.set_aspect(1.0)

def clearPatches(ax):
    for index , patch in zip(range(len(ax.patches)), ax.patches):
        if isinstance(patch, mpl.patches.Polygon) == True:
            patch.remove()
    ax.lines[:]=[]
    applyAxCorrection(ax)

def clearAxPolygonPatches(ax):

    for index , patch in zip(range(len(ax.patches)), ax.patches):
        if isinstance(patch, mpl.patches.Polygon) == True:
            patch.remove()
    ax.lines[:]=[]
    applyAxCorrection(ax)
```

6 Generating various geometric graphs

For manipulating abstract graphs we use the NetworkX [HSSC08] [8]. This section deals with generating the various geometric graphs using packages like Scipy and Sklearn and then converting them into a NetworkX graph with the necessary edge and node attributes. Note that all the nodes in the abstract constructed below have the same numbering across all grap have the same numbering across all graphs: namely, the order in which the points occur in the `points` array argument.

k-NNG

14a ⟨*Functions for generating various graphs* 14a⟩≡ (4a) 14b▷

```
def get_knng_graph(points,k):
    points = np.array(points)

    ### -> Make the graph here
    knng_graph = None
    return knng_graph
```

Minimum Spanning Tree

14b ⟨*Functions for generating various graphs* 14a⟩+≡ (4a) ◁14a 15a▷

```
def get_mst_graph(points):
    points = np.array(points)

    ### -> Make the graph here
    mst_graph = None
    return mst_graph
```

[8] already available inside the Anaconda Python distribution by default

Delaunay Triangulation

Using the example code on Scipy docs here. After obtaining the list of edges we convert it into a NetworkX graph with any important edge and vertex attributes.

15a ⟨*Functions for generating various graphs* 14a⟩+≡ (4a) ◁14b 15b▷

```
def get_delaunay_tri_graph(points):
    points = np.array(points)
    tri    = sp.spatial.Delaunay(points)

    ### -> Make the graph here
    deltri_graph = None
    return deltri_graph
```

TSP tour

15b ⟨*Functions for generating various graphs* 14a⟩+≡ (4a) ◁15a

```
def get_tsp_graph(points):

    import tsp
    points = np.array(points)
    coords = [{"coords":pt} for pt in points]
    t      = tsp.tsp(points)
    idxs_along_tsp = t[1]


    tsp_graph = nx.Graph()
    tsp_graph.add_nodes_from(zip(range(len(points)), coords))

    edge_list = zip(idxs_along_tsp, idxs_along_tsp[1:]) + [(idxs_along_tsp[-1],idxs_along_tsp
    tsp_graph.add_edges_from(  edge_list  )

    print(Fore.RED, list_edges(tsp_graph), Style.RESET_ALL)

    return tsp_graph
```

7 Rendering the graphs

16a ⟨*Functions for plotting and interacting* 8⟩+≡ (4a) ◁13

```
def render_graph(geometric_graph,fig,ax):
    if geometric_graph is None:
        return

    t = np.arange(0.0, 2.0, 0.01)
    s = 1 + np.sin(2 * np.pi * t)
    ax.plot(t, s)
    fig.canvas.draw()
```

8 Finding common edges between two graphs

To find the set of edges common to two graphs on the same set of nodes, we take take an edge from the smaller of the two graphs (i.e. the one with the smaller number of edges) and check whether it exists in the other. Repeat.

16b ⟨*Functions dealing with intersecting two geometric graphs* 16b⟩≡ (4a)

9 Hypothesis testing!

16c ⟨*Testing hypotheses* 16c⟩≡ (4a)

9 References

[HS14] Stefan Hougardy and Rasmus T Schroeder. "Edge elimination in TSP instances". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2014, pp. 275–286. URL: https://bit.ly/3dCFqRS.

[App+09] David L Applegate et al. "Certification of an optimal TSP tour through 85,900 cities". In: *Operations Research Letters* 37.1 (2009), pp. 11–15.

[Mit99] Joseph SB Mitchell. "Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems". In: *SIAM Journal on computing* 28.4 (1999), pp. 1298–1309.

[Aro96] Sanjeev Arora. "Polynomial time approximation schemes for Euclidean TSP and other geometric problems". In: *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE. 1996, pp. 2–11.

[Knu84] Donald Ervin Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[Ram08] Norman Ramsey. *Noweb—a simple, extensible tool for literate programming*. 2008.

[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).

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

Appendices

I Running the Code

Ideally, you will just need to download and install Anaconda and a couple of other Python packages.

To check if the Python executable is in your path and that it is Python 3+, run the command `python --version` inside your terminal emulator. If it succeeds in printing something similar to "`Python 3.7.3`" — as it did on my laptop — then you have successfully installed the Anaconda distro.

The additional packages required can be installed executing in succession the following commands:

```
pip install colorama [9]
pip install tsp
git clone https://github.com/jvkersch/pyconcorde
cd pyconcorde
pip install -e .
```

[9] If you don't have superuser access during installation, add the flag `--user`

To run the code, `cd` into the code's top-level folder, then type any one of the following commands into your terminal.

- `python src/main.py --interactive`
- `python src/main.py --batchtest`
- `python src/main.py --file <points.yaml>`

Please note, while running the code in '`interactive`' mode you will a warning message:

```
CoreApplication::exec: The event loop is already running
```

Please ignore it! It doesn't affect any of the results. Something in the the internals of the Matplotlib that uses the Qt library triggers that warning. I am not sure what. The message doesn't pop up when I use Python 2.7 [10] instead of Python 3.7.

If you have any trouble — or detect a bug! — we can hash things out on Slack, Github or email.

II Laundry-list of Questions/Variants/Conjectures

[10] Note that Python 2 has been deprecated in favour of Python 3+, since January 2020,