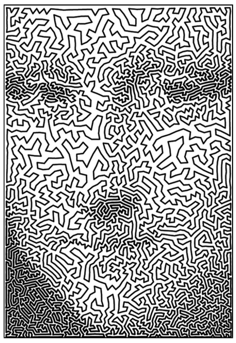

Does the TSP intersect the NNG?

Gaurish Telang
gaurish108@gmail.com

October 16, 2020
1:43pm

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

The question posed in the title came about 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

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* 4c⟩
⟨*Generic utility classes and functions* 9a⟩
⟨*Functions for plotting and interacting* 10⟩
⟨*Functions for generating various graphs* 16⟩
⟨*Functions dealing with intersecting two geometric graphs* 21d⟩
⟨*Testing hypotheses* 22c⟩

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, os, sys, time
from colorama import Fore, Style, Back
import yaml
```

2 Data Generation

4c ⟨*Data Generation* 4c⟩≡ (4a)

⟨*TSPLIB data* 6⟩
⟨*Synthetic data* 8⟩

TSPLIB data-sets

Figure 1 is a screenshot of the entire opening page of [Rei91] that should more than suffice as an intro to this popular benchmark data-set for various TSP-like problems.

TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types. Instances of the following problem classes are available.

Symmetric traveling salesman problem (TSP)
Given a set of n nodes and distances for each pair of nodes, find a roundtrip of minimal total length visiting each node exactly once. The distance from node i to node j is the same as from node j to node i.

Hamiltonian cycle problem (HCP)
Given a graph, test if the graph contains a Hamiltonian cycle or not.

Asymmetric traveling salesman problem (ATSP)
Given a set of n nodes and distances for each pair of nodes, find a roundtrip of minimal total length visiting each node exactly once. In this case, the distance from node i to node j and the distance from node j to node i may be different.

Sequential ordering problem (SOP)
This problem is an asymmetric traveling salesman problem with additional constraints. Given a set of n nodes and distances for each pair of nodes, find a Hamiltonian path from node 1 to node n of minimal length which takes given precedence constraints into account. Each precedence constraint requires that some node i has to be visited before some other node j.

Capacitated vehicle routing problem (CVRP)
We are given $n-1$ nodes, one depot and distances from the nodes to the depot, as well as between nodes. All nodes have demands which can be satisfied by the depot. For delivery to the nodes, trucks with identical capacities are available. The problem is to find tours for the trucks of minimal total length that satisfy the node demands without violating truck capacity constraint. The number of trucks is not specified. Each tour visits a subset of the nodes and starts and terminates at the depot. (Remark: In some data files a collection of alternate depots is given. A CVRP is then given by selecting one of these depots.)

Except, for the Hamiltonian cycle problems, all problems are defined on a complete graph and, at present, all distances are integer numbers. There is a possibility to require that certain edges appear in the solution of a problem.

Figure 1: Screenshot of the opening page of [Rei91]

In this document we will be interested in that subset of instances corresponding to the Symmetric TSP with the standard Euclidean Metric. Pages 9 through 11 of [Rei91] contain 4-column tables with all Symmetric TSP instances. We will be focusing on precisely those instances which have their 3rd column marked `EUC_2D`.

The entire symmetric TSP data-set has been downloaded into the

`./sym-tsp-tsplib/instances/sym-tsp-tsplib/instances/tsplib_symmetric_tsp_instances/`

directory. After writing a small Python script [7] the subset of `EUC_2D` instances were converted into the convenient YAML format and copied into the

`./sym-tsp-tsplib/instances/sym-tsp-tsplib/instances/euclidean_instances_yaml/`

directory. For all practical purposes, unless otherwise notes, *we will retrict our attention to this directory when talking about working with TSPLIB data.*

To see what the point-sets look like peep into the folder `tsplib_euc2d_pictures_of_instances` contained in the top level directory of the code. Note that the numbers affixed to each instance name indicate the number of points in that instance. See Figure 2 for some examples.

This chunk implements two functions: the first one returns the full path names of each of the Euclidean instances in an list and the second one reads in a TSPLIB instance (identified by its file-name e.g.

[7] `tsplib_to_yaml.py` in that same directory

'berlin52.yml') in the euclidean_instances_yaml directory and returns a list of 2D points for that instance.

6 ⟨*TSPLIB data* 6⟩≡ (4c)

```
def get_names_of_all_euclidean2D_instances(dirpath="./sym-tsp-tsplib/instances/euclidean_insta

    inst_names = []
    for name in os.listdir(dirpath):
        full_path = os.path.join(dirpath, name)
        if os.path.isfile(full_path):
            inst_names.append(name)
    return inst_names

def tsplib_instance_points(instance_file_name, dirpath="./sym-tsp-tsplib/instances/euclidean_i

        print(Fore.GREEN+"Reading " + instance_file_name, Style.RESET_ALL)
        with open(dirpath+instance_file_name) as file:
            data = yaml.load(file, Loader=yaml.FullLoader)
            points = np.asarray(data['points'])

        return points
```

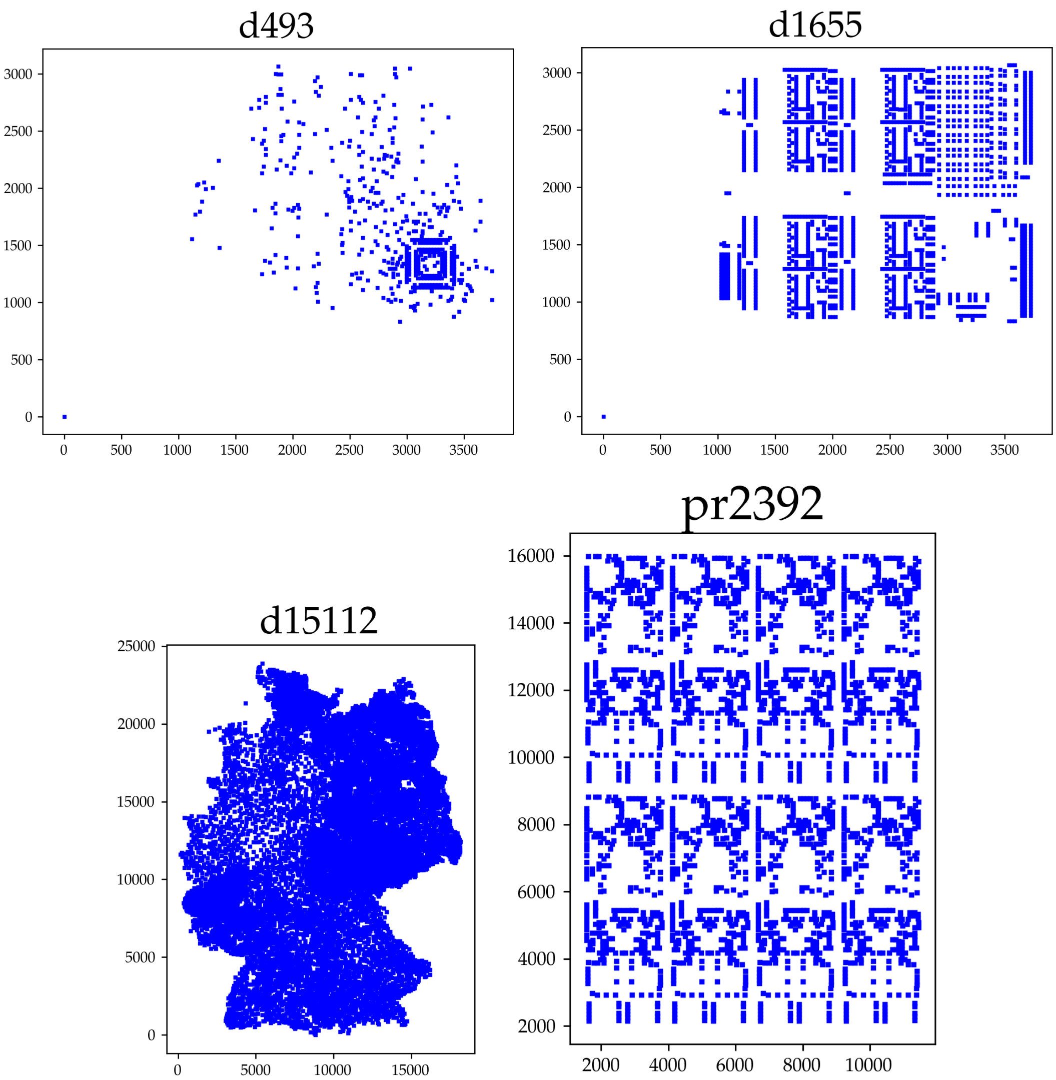

Figure 2: Instances of four TSPLIB data sets for the Symmetric TSP with 2D Euclidean Metric

Synthetic data-sets

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.

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. [8] 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.

8 ⟨*Synthetic data* 8⟩≡ (4c)

```
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)
```

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

```
    return points
```

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.

9a ⟨*Generic utility classes and functions* 9a⟩≡ (4a) 9b▷

```
def write_to_yaml_file(data, dir_name, file_name):
   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.

9b ⟨*Generic utility classes and functions* 9a⟩+≡ (4a) ◁9a

```
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.

10 ⟨*Functions for plotting and interacting* 10⟩≡ (4a) 11 ▷

```
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. Note that each mouse click clears the canvas and freshly draws the input point-set from scratch. This helps with modifying an existing input to check how solution changes.

11 ⟨*Functions for plotting and interacting* 10⟩+≡ (4a) ◁10 12▷

```
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])/130.0

             ax.clear()

             for pt in run.points:
                  ax.add_patch( mpl.patches.Circle( pt, 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.

12 ⟨*Functions for plotting and interacting* 10⟩+≡ (4a) ◁11 15b▷

```
def wrapperkeyPressHandler(fig,ax, run):
        def _keyPressHandler(event):
                if event.key in ['i', 'I']:
                        ⟨Enter spanning graph 13⟩
                elif event.key in ['n', 'N', 'u', 'U']:
                        ⟨Enter type of point set 14⟩
                elif event.key in ['c', 'C']:
                        ⟨Clear all states and the canvas 15a⟩

        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.

13 ⟨*Enter spanning graph* 13⟩≡ (12)

```
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"              +\
                    "(pytsp) TSP computed with the pure Python TSP library \n" +
                    "(conc)  TSP computed with the pure Python TSP library \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 == 'pytsp':
     geometric_graph = get_py_tsp_graph(run.points)

elif algo_str == 'conc':
     geometric_graph = get_concorde_tsp_graph(run.points)

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


common_edges = list_common_edges(get_concorde_tsp_graph(run.points), geometric_graph)
print(Fore.YELLOW, "-----LIST OF EDGES OF COMPUTED GRAPH COMMON TO TSP-------")
for i, edge in zip(range(1,1+len(common_edges)),common_edges):
     print(i, '-->', edge)
print("-------------------------------------------")
print("Number of edges of the indicated geometric graph that are common to the Concorde TSP: "
print("-------------------------------------------", Style.RESET_ALL)
```

```
ax.set_title("Graph Type: " + geometric_graph.graph['type'] + '\n Number of nodes: ' + str(len
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.

14 ⟨*Enter type of point set* 14⟩≡ (12)

```
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 generated: ' + str(len(run.points)), fontdict={'fontsize':25})
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'.

15a ⟨*Clear all states and the canvas* 15a⟩≡ (12)

```
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.

15b ⟨*Functions for plotting and interacting* 10⟩+≡ (4a) ◁12 21a▷

```
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] [9]. 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

We use the nearest neighbor routine from the Scikit-learn [Ped+11] library. The documentation for the various nearest neighor methods implemented therein can be found at `https://bit.ly/3nTQkqV`. Note that k- nearest-neighbors of a point includes the point itself. Thus we use $(k + 1)$ in the argument to the `NearestNeighbors` function below.

16 ⟨*Functions for generating various graphs* 16⟩≡ (4a) 17 ▷

```
def get_knng_graph(points,k):
     from sklearn.neighbors import NearestNeighbors

     points     = np.array(points)
     coords     = [{"coods":pt} for pt in points]
     knng_graph = nx.Graph()
     knng_graph.add_nodes_from(zip(range(len(points)), coords))

     nbrs = NearestNeighbors(n_neighbors=(k+1), algorithm='ball_tree').fit(points)
     distances, indices = nbrs.kneighbors(points)

     edge_list = []
     for nbidxs in indices:
          nfix = nbidxs[0]
          edge_list.extend([(nfix,nvar) for nvar in nbidxs[1:]])

     knng_graph.add_edges_from(  edge_list  )

     knng_graph.graph['type']    = str(k)+'nng'
     knng_graph.graph['weight'] =  None # TODO, also edge weights for each edge!!!
     return knng_graph
```

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

Delaunay Triangulation

We use the blackbox routine for computing this graph implmeneted in Scipy [Vir+20].

17 ⟨*Functions for generating various graphs* 16⟩+≡ (4a) ◁16 18a▷

```
def get_delaunay_tri_graph(points):
     from scipy.spatial import Delaunay
     points       = np.array(points)
     coords       = [{"coods":pt} for pt in points]
     tri          = Delaunay(points)
     deltri_graph = nx.Graph()

     deltri_graph.add_nodes_from(zip(range(len(points)), coords))

     edge_list = []
     for (i,j,k) in tri.simplices:
         edge_list.extend([(i,j),(j,k),(k,i)])
     deltri_graph.add_edges_from(  edge_list   )

     total_weight_of_edges = 0.0
     for edge in deltri_graph.edges:
           n1, n2 = edge
           pt1 = deltri_graph.nodes[n1]['coods']
           pt2 = deltri_graph.nodes[n2]['coods']
           edge_wt = np.linalg.norm(pt1-pt2)

           deltri_graph.edges[n1,n2]['weight'] = edge_wt
           total_weight_of_edges = total_weight_of_edges + edge_wt

     deltri_graph.graph['weight'] = total_weight_of_edges
     deltri_graph.graph['type']   = 'dt'

     return deltri_graph
```

Minimum Spanning Tree

From elementary CG, we know that the MST of a set of points in the plane is a subset of the delaunay triangulation. Thus to compute the MST, it suffices to compute the MST of the corresponding delaunay triangulation. See this page for a documentation of the code in NetworkX used to compute the MST. Note that along with the Kruskal method (used below), both Prim's and Boruvka's algorithms have also been implemented.

18a ⟨*Functions for generating various graphs* 16⟩+≡ (4a) ◁17 18b▷

```
def get_mst_graph(points):

    points = np.array(points)
    deltri_graph = get_delaunay_tri_graph(points)
    mst_graph = nx.algorithms.tree.mst.minimum_spanning_tree(deltri_graph, algorithm='kruskal
    mst_graph.graph['type']   = 'mst'
    return mst_graph
```

Traveling Saleman Tour (Cycle)

We use two separate independent routines that each compute the TSP. One is the `tsp` module available at `https://pypi.org/project/tsp/` the other being, Concorde, through its Python interface (whose github page can be accessed at `https://github.com/jvkersch/pyconcorde`. Anedoctally speaking the first solver works relatively quickly on point-sets upto size 30. Because of its simplicity, we used it in the intial stages of writing this report. It is clearly not competitive with Concorde (which can solve a 300 size instances in a couple of seconds), but it offers a useful backup routine, in the event that your machine faces problems installing the PyConcorde library.

* Using the `tsp` library

18b ⟨*Functions for generating various graphs* 16⟩+≡ (4a) ◁18a 20▷

```
def get_py_tsp_graph(points):
    import tsp
    points = np.array(points)
    coords = [{"coods":pt} for pt in points]
    ⟨Generate TSP cycle and convert into NetworkX graph 19a⟩
    ⟨Compute weight of each edge and total edge weight 19b⟩
    ⟨Set graph attributes 19c⟩
    return tsp_graph
```

19a ⟨*Generate TSP cycle and convert into NetworkX graph* 19a⟩≡ (18b)

```
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 = list(zip(idxs_along_tsp, idxs_along_tsp[1:])) + \
                  [(idxs_along_tsp[-1],idxs_along_tsp[0])]
tsp_graph.add_edges_from(  edge_list  )
```

19b ⟨*Compute weight of each edge and total edge weight* 19b⟩≡ (18b)

```
total_weight_of_edges = 0.0
for edge in tsp_graph.edges:

      n1, n2 = edge
      pt1 = tsp_graph.nodes[n1]['coods']
      pt2 = tsp_graph.nodes[n2]['coods']
      edge_wt = np.linalg.norm(pt1-pt2)

      tsp_graph.edges[n1,n2]['weight'] = edge_wt
      total_weight_of_edges = total_weight_of_edges + edge_wt
```

19c ⟨*Set graph attributes* 19c⟩≡ (18b)

```
tsp_graph.graph['weight'] = total_weight_of_edges
tsp_graph.graph['type']   = 'pytsp'
```

* Using the Pyconcorde library

This library is a thin interface around Concorde. Installing Pyconcorde automatically installs Concorde and other required libraries such as QSOpt. Instructions for installation are given in Appendix I.

Note that the Concorde solver works only on points with integer coordinates. Since our synthetic datasets will be generated inside the unit-square, we scale by the amount `scale_factor` and then rounded to an integer using `int()`. For a sufficiently large value `scaling_factor`, ordering of points reported by Concorde should be the same as if the algorithm was run on the unscaled points.

20 ⟨*Functions for generating various graphs* 16⟩+≡ (4a) ◁18b

```
def get_concorde_tsp_graph(points, scaling_factor=1000):
     from concorde.tsp import TSPSolver
     points = np.array(points)
     coords = [{"coods":pt} for pt in points]

     #from concorde.tests.data_utils import get_dataset_path
     #fname = get_dataset_path("berlin52")
     #solver = TSPSolver.from_tspfile(fname)
     #solution = solver.solve()

     xs = [int(scaling_factor*pt[0]) for pt in points]
     ys = [int(scaling_factor*pt[1]) for pt in points]
     solver = TSPSolver.from_data(xs, ys, norm='EUC_2D', name=None)
     print(Fore.GREEN)
     solution = solver.solve()
     print(Style.RESET_ALL)

     concorde_tsp_graph=nx.Graph()

     idxs_along_tsp = solution.tour
     concorde_tsp_graph.add_nodes_from(zip(range(len(points)), coords))
     edge_list = list(zip(idxs_along_tsp, idxs_along_tsp[1:])) + \
                    [(idxs_along_tsp[-1],idxs_along_tsp[0])]
     concorde_tsp_graph.add_edges_from(  edge_list  )

     concorde_tsp_graph.graph['type']    = 'conc'
     concorde_tsp_graph.graph['found_tour_p'] = solution.found_tour
     concorde_tsp_graph.graph['weight'] = None ### TODO!!
     return concorde_tsp_graph
```

7 Rendering the graphs

For this we just draw each edge of the geometric graph as a straight line segment between the points(each of which happens to be a node of the graph).

21a ⟨*Functions for plotting and interacting* 10⟩+≡ (4a) ◁15b

```
def render_graph(G,fig,ax):
    if G is None:
        return
    ⟨Set up edge colors depending on graph type 21b⟩
    ⟨Iterate through graph edges and draw as segments 21c⟩
    fig.canvas.draw()
```

21b ⟨*Set up edge colors depending on graph type* 21b⟩≡ (21a)

```
edgecol = None
if G.graph['type'] == 'mst':
    edgecol = 'g'
elif G.graph['type'] in ['conc','pytsp']:
    edgecol = 'r'
elif G.graph['type'] == 'dt':
    edgecol = 'b'
elif G.graph['type'][-3:] == 'nng':
    edgecol = 'm'
```

21c ⟨*Iterate through graph edges and draw as segments* 21c⟩≡ (21a)

```
for  (nidx1, nidx2) in G.edges:
    x1, y1 = G.nodes[nidx1]['coods']
    x2, y2 = G.nodes[nidx2]['coods']
    ax.plot([x1,x2],[y1,y2],'-', color=edgecol)
```

8 Finding common edges between two graphs

It is possible the same edge may exist in both the graphs but the indices recorded in the nodes may be in a different order. Hence, we explicitly define edges from two different graphs on the same set of nodes as being equal, if they are equal as sorted lists.

21d ⟨*Functions dealing with intersecting two geometric graphs* 21d⟩≡ (4a) 22a▷

```
def edge_equal_p(e1,e2):
    e1 = sorted(list(e1))
    e2 = sorted(list(e2))
    return (e1==e2)
```

To find the set of edges common to two graphs on the same set of nodes, we take take each edge from one of the graphs and check whether it exists in the other.

22a ⟨*Functions dealing with intersecting two geometric graphs* 21d⟩+≡ (4a) ◁21d 22b▷

```
def list_common_edges(g1, g2):
     common_edges = []
     for e1 in g1.edges:
          for e2 in g2.edges:
             if  edge_equal_p(e1,e2):
                  common_edges.append(e1)
     return common_edges
```

Finally, just a small function that tests if two graphs intersect.

22b ⟨*Functions dealing with intersecting two geometric graphs* 21d⟩+≡ (4a) ◁22a

```
def graphs_intersect_p(g1,g2):
     flag = False
     if list_common_edges(g1,g2):
          flag = True
     return flag
```

9 Hypothesis testing!

22c ⟨*Testing hypotheses* 22c⟩≡ (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.

[Rei91] Gerhard Reinelt. "TSPLIB—A traveling salesman problem library". In: *ORSA journal on computing* 3.4 (1991), pp. 376–384. URL: https://bit.ly/37e0bAq.

[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.

[Ped+11] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[Vir+20] Pauli Virtanen et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature methods* 17.3 (2020), pp. 261–272.

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 [10]
pip install tsp
git clone https://github.com/jvkersch/pyconcorde
cd pyconcorde
pip install -e .
```

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

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