

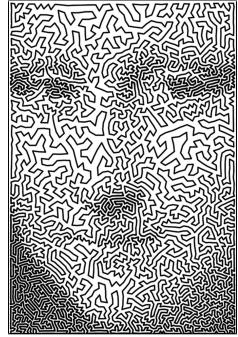
Does the TSP intersect the NNG?

Gaurish Telang

gaurish108@gmail.com

October 29, 2020

3:29am



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 tried was to get at some kind of structure theorem by “guessing” 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 “ $TSP \cap NNG = \emptyset$ ” seemed like a good place to start.

It is easy to construct point-sets where the segment joining the closest pair of points need not lie along the TSP tour. See [Figure 1](#)

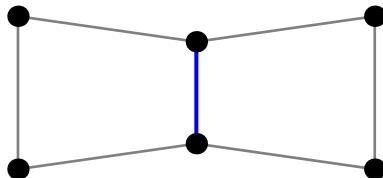


Figure 1: Example where the segment (blue) joining the closest pair of points does not lie along the TSP tour (gray)

One can also easily construct a family of instances, where the nearest neighbor graph intersects the TSP tour, exactly twice as shown in [Figure 2](#).

However, all attempts at constructing examples where the intersection with the 1-NNG is *empty* failed. ³ 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. ⁴ There was also a very much related

¹In 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.

²In 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 Concorde library (to the author’s knowledge) computes only optimal cycle solutions and *not* optimal path solutions!

³Notice that [Figure 1](#) is not a counterexample!

⁴The author believes this will be a useful reference for future work

discussion thread on David Eppstein's [webpage](#). A small counter-example to Michael Shamos' conjecture from his unpublished notes — that the TSP is a *subgraph* of the Delaunay — is given near the bottom of that link.

But the thread says nothing about whether the DT *must* intersect the TSP at least a certain fraction of times, or indeed even once. ⁵

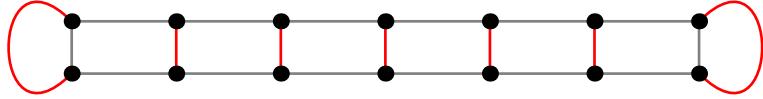


Figure 2: Example where the nearest neighbor graph shares *only* two edges with the TSP. The edges of the abstract nearest neighbor graph (red) are superimposed on top of the TSP tour (gray) through the given points. But note that the TSP tour shares each of its edge with the 2-NNG.

See this [blogpost](#) on the topic, which talks about using them for generating heuristically good (no bounds are given) TSP cycles. Another approach using del tris is taken in [this technical report](#)

Knowledge of some family of easily computed edges that are necessarily part of a TSP solution could potentially be used to speed up some of the existing solutions to TSP using combinatorial optimization methods; see, e.g., Concorde [[App+09](#)] and other papers of Bill Cook ⁶.

To spur our intuition, we investigate the conjecture experimentally in this short report ⁷ using TSPLIB and Concorde in tandem. TSPLIB [[Rei91](#)] is an online collection of medium to large scale instances of the Metric, the Euclidean and a few other variants of the TSP Concorde can compute the optimal solutions in nearly all the instances; the certificate of optimality — as always! — coming from the comparsion of the computed tour-length against a lower bounds (also computed by Concorde).

For starters, we investigate the following questions ⁸: for each symmetric 2-D Euclidean TSP instance from TSPLIB for which we have an optimal solution, does

- ❖ $TSP \cap (k\text{-})NNG \stackrel{?}{=} \emptyset$, for $k = 1, 2, \dots$
- ❖ $TSP \cap \text{Delaunay Graph} \stackrel{?}{=} \emptyset$
- ❖ For question 1 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*

⁵Perhaps, we can follow up with Dillencourt or Eppstein if they have notes on this?

⁶The 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 [[App+09](#)] or Helsgaun's methods [[Hel00](#)] use 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 very fast. But 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 (the pigeon-hole principle on steroids!) or something from Ramsey Theory to prove the existence of an intersection??

⁷This report has been written as a literate program [[Knu84](#); [Ram08](#)] to weave together the code, documentations, explanations, references and generated data into the same document. Brickbats and bouquets on the author's preliminary stab at Literate Programming are most welcome.

⁸Experimental answers to other questions will be barnacled onto the report as it keeps growing.

one edge from the intersection with the 1-NNG have one of its *vertices* on the convex hull? ⁹
 More generally, is this true for every layer of the onion, and not just the outer layer (i,e, the convex hull)?

See also the Appendix ?? 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 various tests on the points inputted.

If you don't have Python 3.7+ on your machine, download the free [Anaconda](#) distro of Python; it comes with most of the batteries included. See Appendix A for instructions on how to install and run the code.

The image of the mona-lisa is taken from [[Sci](#)].

⁹This indeed seemed to be the case in all the author's failed attempts at a counter-example, and so a proof/disproof of this conjecture would be helpful

Contents

	Page
1 Overall structure of <code>tspnng.py</code>	5
2 Data Generation	6
2.1 TSPLIB data-sets	7
2.2 Synthetic data-sets	10
3 Data Storage	14
4 Setting up TSPNNGInput class	15
5 Setting up the Interactive Canvas	16
6 Generating various geometric graphs	25
6.1 k -NNG	25
6.2 Delaunay Triangulation	27
6.3 Minimum Spanning Tree	28
6.4 The Onion	29
6.5 Gabriel Graph	32
6.6 Urquhart Graph	34
6.7 Traveling Saleman Tour (Cycle)	36
6.8 Graph Powers	39
7 Rendering the graphs	39
8 Finding common edges between two graphs	41
9 Checking Hamiltonicity	42
10 Perturbing points by prescribed factor randomly	43
11 Calculating the Length of a Polygonal Chain	43
12 Data Analysis Functions	44
13 Computing the Maximally Dispersed Vertex Cover of a Graph	45
14 Experiments	49
14.1 Intersection behavior for TSPLIB instances	49
14.2 Intersection behavior for uniformly randomly generated instances	52
14.3 Intersection behavior for TSPLIB instances	56
Appendices	57
Appendix A Installing and running the Code	57

Appendix B Catalog of symmetric 2D Euclidean instances inside TSPLIB	60
B.1 Pictures of TSPLIB Euclidean Instances	62
Appendix C Machine Details	72

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. .

5 $\langle \text{tspnng.py } 5 \rangle \equiv$

- $\langle \text{Headers } 6a \rangle$
- $\langle \text{Data Generation } 6b \rangle$
- $\langle \text{Generic utility classes and functions } 11 \rangle$
- $\langle \text{Functions for plotting and interacting } 16 \rangle$
- $\langle \text{Functions for generating various graphs } 25 \rangle$
- $\langle \text{Functions dealing with intersecting two geometric graphs } 41a \rangle$
- $\langle \text{Experiments } 49 \rangle$

6a $\langle Headers \text{ 6a} \rangle \equiv$ (5)

```

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 prettytable import PrettyTable

from sklearn.cluster import KMeans
import argparse, os, sys, time
from colorama import init, Fore, Style, Back
init() # this line does nothing on Linux/Mac,
        # but is important for Windows to display
        # colored text. See https://pypi.org/project/colorama/
import yaml
import shutil

```

2 Data Generation

6b $\langle Data\ Generation \text{ 6b} \rangle \equiv$ (5)

- $\langle TSPLIB\ data \text{ 8} \rangle$
- $\langle Synthetic\ data \text{ 10} \rangle$

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 3: Screenshot of the opening page of [Rei91]

2.1 TSPLIB data-sets

Figure 3 is a screenshot of the entire opening page of [Rei91] that should more than suffice as an intro to this popular set of benchmarks for various TSP-like problems.¹⁰

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 precisely on 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 script¹¹ 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. *Unless otherwise noted, we will restrict our attention to this directory when talking about 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 4 for some examples.

¹⁰Prof. Sandor Fekete has a much larger collection of interesting TSP data-sets, I believe?

¹¹`tsplib_to_yaml.py` in that same directory

This chunk implements two functions: the first one returns the full path names of each of the Euclidean instances in a list and the second one reads in a TSPLIB instance (identified by its file-name e.g. 'berlin52.yaml') in the `euclidean_instances_yaml` directory and returns a list of 2D points for that instance.

8 $\langle TSPLIB\ data\ 8 \rangle \equiv$ (6b)

```
def get_names_of_all_euclidean2D_instances(dirpath=\
    "./sym-tsp-tsplib/instances/euclidean_instances_yaml/" ):

    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_instances_yaml/"):

    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
```

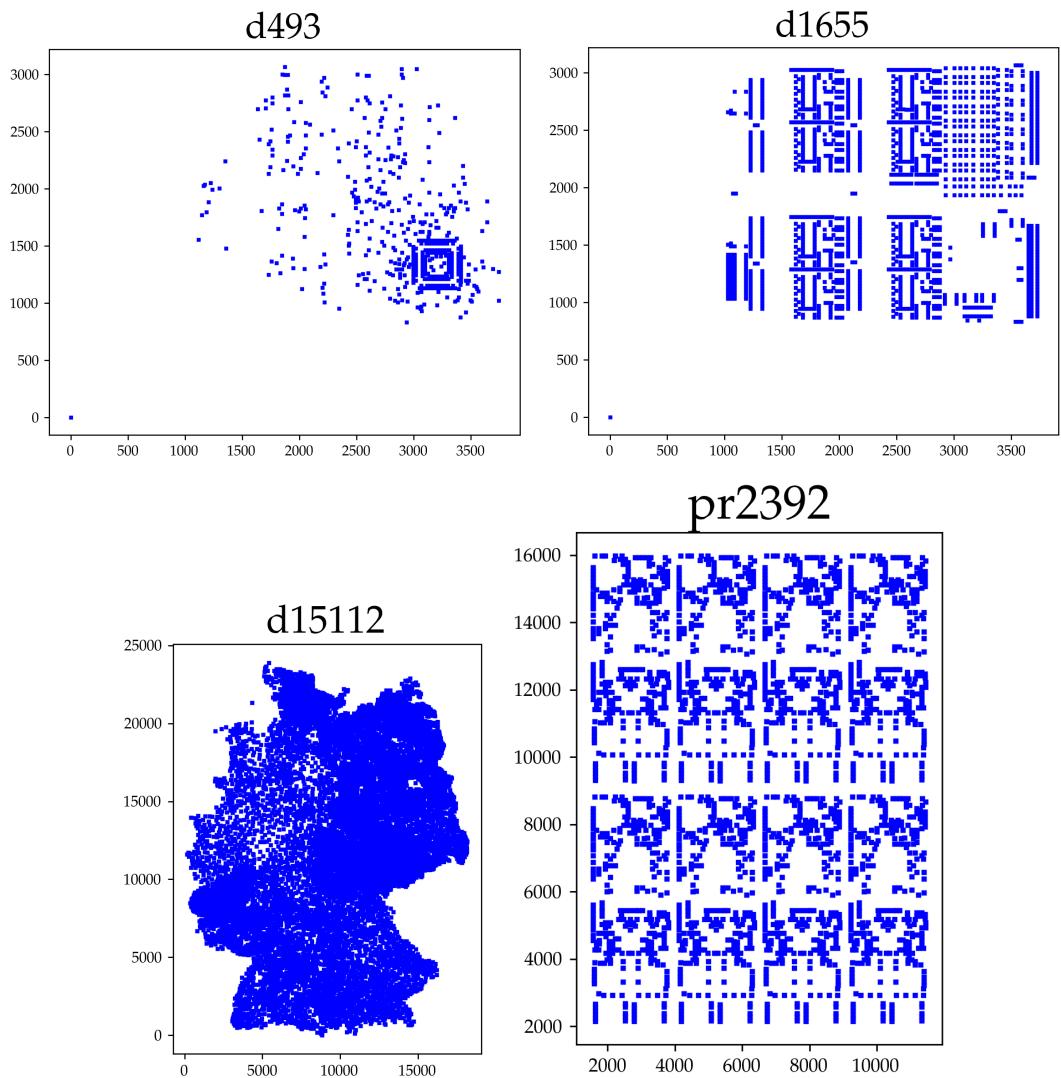


Figure 4: Instances of four TSPLIB data sets for the Symmetric TSP with 2D Euclidean Metric

2.2 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.¹² 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.

10 *(Synthetic data 10)≡* (6b)

```

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()
```

¹²A somewhat similar method was used in Jon Bentley's experimental TSP paper

```
    points.extend(remaining_pts)  
    return points
```

Other kinds of data might include multimodal distributions. I will use a natural method of generating these: uniformly sample k points from the unit-square, and create a normally distributed point set with $\lfloor n/k \rfloor$ points per point-cloud. (the k points sampled are not included into the cloud.)

Finally the points are shifted and scaled so that they all fall within the unit-square.

¹¹ *Generic utility classes and functions* 11 \equiv (5) 12 ▷

```

def multimodal_points(numpts, nummodes=4, sigma=0.05):

    modes = sp.rand(nummodes, 2).tolist()
    points = []
    for mode in modes:
        mx, my = mode
        samp_x = np.random.normal(mx, sigma, int(numpts/nummodes))
        samp_y = np.random.normal(my, sigma, int(numpts/nummodes))
        points.extend([np.asarray(pt) for pt in zip(samp_x, samp_y)])

    num_points_rem = numpts%nummodes
    if num_points_rem != 0:
        xrs = np.random.rand(num_points_rem)
        yrs = np.random.rand(num_points_rem)
        points.extend([np.asarray(pt) for pt in zip(xrs,yrs)])

    points = shift_and_scale_to_unit_square(points)
    return points

```

The next chunk implements the `scale_to_unit_square` function used in the last line. We implement this separately since we expect to use it while generating other point-sets.

```
12  ⟨Generic utility classes and functions 11⟩+≡
      (5) ◀11 13▶
def shift_and_scale_to_unit_square(points):

    # make all coordinates positive by shifting origin
    points = [np.asarray(pt) for pt in points]
    min_x  = min([x for (x,_) in points])
    min_y  = min([y for (_,y) in points])
    m      = min(min_x, min_y)
    origin = np.asarray([m,m])

    # scale to unit-square
    points = [pt - origin for pt in points]
    max_x  = max([x for (x,_) in points])
    max_y  = max([y for (_,y) in points])
    scale   = max(max_x,max_y)
    points = [pt/scale for pt in points]

return points
```

The next function creates a series of points placed uniformly along concentric circles, whose radii are in arithmetic progression. If there are any leftover points to generate, they are generated randomly uniformly inside the unit-square.

```
13   ⟨Generic utility classes and functions 11⟩+≡                               (5) ◀12 14a▷
    def concentric_circular_points(numpts, numrings):
        numpts_per_ring = int(numpts/numrings)
        points         = []
        center          = np.asarray([0.5,0.5])
        for ring in range(numrings):
            radius = (ring+1)*0.5/(numrings+1)
            print("Radius computed is ", radius)

            angles = [idx * 2*np.pi/numpts_per_ring for idx in range(numpts_per_ring)]
            xs     = [center[0] + radius * np.cos(theta) for theta in angles ]
            ys     = [center[1] + radius * np.sin(theta) for theta in angles ]
            points.extend([np.asarray(pt) for pt in zip(xs,ys)])

        num_points_rem = numpts%numrings
        if num_points_rem != 0:
            xrs = np.random.rand(num_points_rem)
            yrs = np.random.rand(num_points_rem)
            points.extend([np.asarray(pt) for pt in zip(xrs,yrs)])

    return points
```

Points placed along a grid are extremely useful as examples when considering TSP like problems. Hence we generate such grid points next. As usual left-over points will be generated randomly inside the unit-square.

14a *(Generic utility classes and functions 11) +≡* (5) ◀13 14b ▷

```

def rectangular_grid_points(numpts, numrows):
    numcols = int(numpts/numrows)

    points = []
    for i in range(numrows):
        for j in range(numcols):
            print([i,j])
            points.append(np.asarray([j,i]))

    points = shift_and_scale_to_unit_square(points)

    num_points_rem = numpts-numrows*numcols
    if num_points_rem != 0:
        xrs = np.random.rand(num_points_rem)
        yrs = np.random.rand(num_points_rem)
        points.extend([np.asarray(pt) for pt in zip(xrs,yrs)])

    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.

14b *(Generic utility classes and functions 11) +≡* (5) ◀14a 15 ▷

```

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.

15 *⟨Generic utility classes and functions 11⟩+≡* (5) ◀14b 42▶

```
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.

16 ⟨*Functions for plotting and interacting* 16⟩≡

(5) 17▷

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

    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([])

    patchSize = (xlim[1]-xlim[0])/130.0

    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()

    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.

```
17 <Functions for plotting and interacting 16>+≡ (5) ▷16 18▷
      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.

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.

First we compute the TSP and then print a table mentioning how many of its edges are common to other standard graphs. See <https://pypi.org/project/prettytable/> for more information on the prettytable module used to output data to terminal.

For computing the k -NNG we compute from $k = 1$ upto a value of $k = 2 + \sqrt{n}$ where n is the number of input points. This allows a “natural” way to scale the value of k for increasing values of n .

`19` *(Compute TSP and find common edges with various spanning graphs 19)≡* (18)

```
tsp_graph = get_concorde_tsp_graph(run.points)
graph_fns = [(get_delaunay_tri_graph, 'Delaunay Triangulation (D)'), \
              (get_mst_graph, 'Minimum Spanning Tree (M)'), \
              (get_onion_graph, 'Onion'), \
              (get_gabriel_graph, 'Gabriel'), \
              (get_urquhart_graph, 'Urquhart')]

from functools import partial
for k in range(1,5):
    graph_fns.append((partial(get_knng_graph, k=k), str(k)+'_NNG'))

tbl = PrettyTable()
tbl.field_names = ["Spanning Graph (G)", "G", "G \cap T", "T", "(G \cap T)/T"]
num_tsp_edges = len(tsp_graph.edges)

for ctr, (fn_body, fn_name) in zip(range(1,1+len(graph_fns)), graph_fns):
    geometric_graph = fn_body(run.points)
    num_graph_edges = len(geometric_graph.edges)
    common_edges = list_common_edges(tsp_graph, geometric_graph)
    num_common_edges_with_tsp = len(common_edges)

    tbl.add_row([fn_name, \
                num_graph_edges, \
                num_common_edges_with_tsp, \
                num_tsp_edges, \
                "{perc:3.2f}%".format(perc=1e2*num_common_edges_with_tsp/num_tsp_edges)+ '%' ])

print("Table of number of edges in indicated graph")
print(tbl)
render_graph(tsp_graph,fig,ax)
fig.canvas.draw()
```

In a kind of “dual” demo, we now compute and render the various geometric graphs, and then mention how many edges each graph has in common with the TSP. Thus we can explore the intersection of the TSP with a graph from the point-of-view of both the TSP and the graph.

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.

20 ⟨Compute spanning graph 20⟩≡

```
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"      +\
                  "(onion) Onion                      \n"      +\
                  "(gab)   Gabriel Graph            \n"      +\
                  "(urq)   Urquhart Graph          \n"      +\
                  "(dt)    Delaunay Triangulation     \n"      +\
                  "(conc)  TSP computed by the Concorde TSP library \n" +
                  "(pytsp) TSP computed by 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 == 'onion':
    geometric_graph = get_onion_graph(run.points)

elif algo_str == 'gab':
    geometric_graph = get_gabriel_graph(run.points)

elif algo_str == 'urq':
    geometric_graph = get_urquhart_graph(run.points)

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

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

elif algo_str == 'pytsp':
    geometric_graph = get_py_tsp_graph(run.points)
```

```
elif algo_str in ['d','D']:  
  
    build_delaunay_forest(run.points,stopnum=10)  
    sys.exit()  
  
else:  
    print(Fore.YELLOW, "I did not recognize that option.", Style.RESET_ALL)  
    geometric_graph = None  
  
tsp_graph = get_concorde_tsp_graph(run.points)  
common_edges = list_common_edges( tsp_graph, geometric_graph)  
print(Fore.YELLOW+"-----"+Style.RESET_ALL)  
print("Number of edges in " + algo_str + " graph : ", len(geometric_graph))  
print(Fore.YELLOW, "\nNumber of edges in " + algo_str + " graph which are also in Concorde TSP : ", len(common_edges))  
print("Number of edges in " + "Concorde TSP : ", len(tsp_graph.edges))  
print("-----", Style.RESET_ALL)  
  
  
ax.set_title("Graph Type: " + geometric_graph.graph['type'] + '\n Number of nodes: ' + str(len(geometric_graph)))  
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.

22

⟨Enter type of point set to generate 22⟩≡

(18)

```

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)
elif event.key in ['u', 'U']:
    run.points = uniform_points(numpts)
elif event.key in ['m', 'M']:
    nummodes = int(input(Fore.YELLOW+"How many modes do you want in the distribution?"+Style.RESET_ALL))
    sigma = float(input(Fore.YELLOW+"What do you want the standard deviation of the 1"))
    run.points = multimodal_points(numpts,nummodes=sigmamodes,sigma=sigma)
elif event.key in ['o', 'O']:
    numrings = int(input(Fore.YELLOW+"How many rings do you want?"+Style.RESET_ALL))
    run.points = concentric_circular_points(numpts,numrings)
elif event.key in ['g', 'G']:
    numrows = int(input(Fore.YELLOW+"How many rows do you want?"+Style.RESET_ALL))
    run.points = rectangular_grid_points(numpts,numrows)
else:
    print("I did not understand that option. Please type one of 'n', 'u', 'm', 'o', 'g'")

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()
```

Sometimes, you just want to clear the edges of the network from the graph, so that a new graph can be rendered in its place on the points. For that, you need to press ‘x’ or ‘X’.

23a *⟨Clear all line segments from the canvas 23a⟩≡* (18)

```
print(Fore.GREEN, 'Removing network edges from canvas', Style.RESET_ALL)
ax.lines=[]
applyAxCorrection(ax)
fig.canvas.draw()
```

If you want to wipe the canvas and the point-cloud data (and everything else ...) clean, then press ‘c’.

23b *⟨Clear all states and the canvas 23b⟩≡* (18)

```
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.

```
24   <Functions for plotting and interacting 16>+≡           (5) ◀18 39a▷
      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)
```

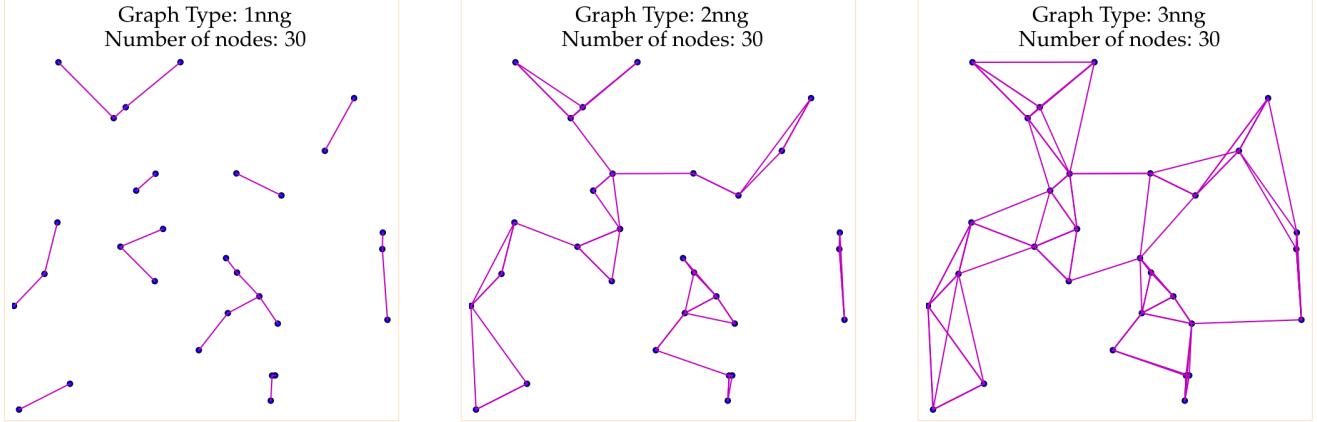


Figure 5: Generating the k -NNG graphs with Scikit-Learn for various value of k on the same set of 30 randomly generated points. Note that we are considering these graphs as undirected.

6 Generating various geometric graphs

For manipulating abstract graphs we use the NetworkX [HSSC08]¹³. 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 graph have the same numbering across all graphs: namely, the order in which the points occur in the `points` array argument.

6.1 k -NNG

We use the nearest neighbor routine from the Scikit-learn [Ped+11] library. The documentation for the various nearest neighbor methods implemented therein can be found at <https://bit.ly/3nTQkqV>. Note that for the nearest-neighbor function of sklearn the k - nearest-neighbors of a point includes the point itself. Thus we use $(k + 1)$ in the argument to the `NearestNeighbors` function below, and take the last k elements in the list returned — the neighbors of a point are reported by that function in increasing order of distance from that point.

25

(Functions for generating various graphs 25)≡

(5) 27▷

```
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 = []
```

¹³already available inside the Anaconda Python distribution by default

```
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
```

6.2 Delaunay Triangulation

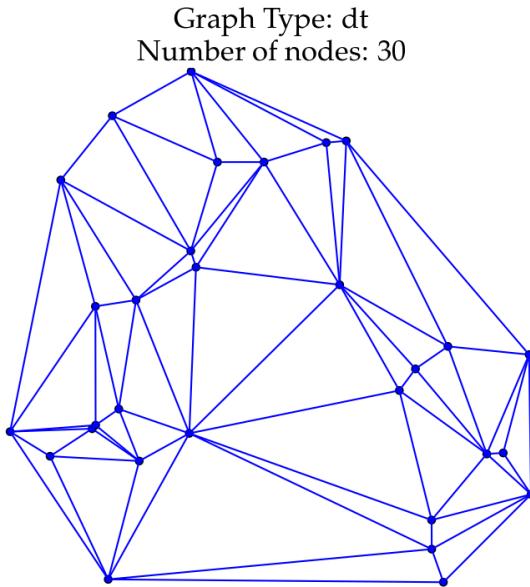


Figure 6: Example of a Delaunay Triangulation computed by SciPy on 30 randomly generated points

We use the [blackbox routine](#) for computing this graph implemented in Scipy [Vir+20].

```
27   <Functions for generating various graphs 25>+≡          (5) ◀25 ▶28>
    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

```

6.3 Minimum Spanning Tree

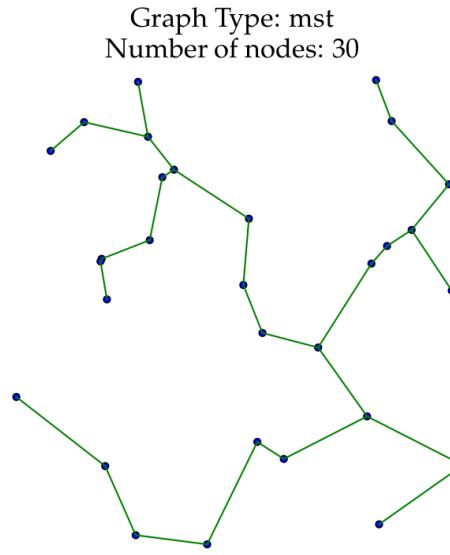


Figure 7: Example of a Minimum Spanning Tree computed by NetworkX on 30 randomly generated points

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 on an abstract weighted undirected graph. Note that along with the Kruskal method (used below), both Prim's and Boruvka's algorithms have also been implemented in that library.

28

⟨Functions for generating various graphs 25⟩+≡

(5) ▲27 ▲29 ▷

```

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

```

6.4 The Onion

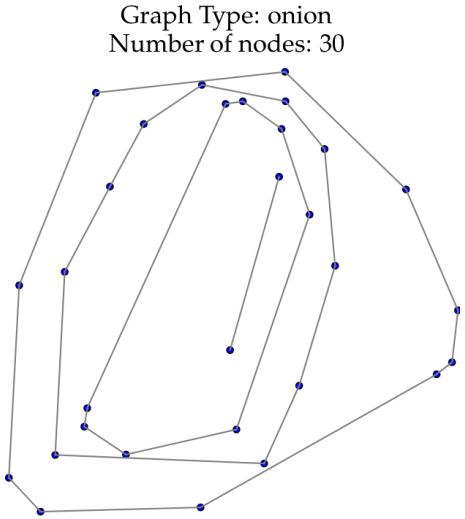


Figure 8: Example of the Onion graph computed with QHull (through SciPy) on 30 randomly generated points

Here we compute successive convex-hull of the point-set: compute the convex hull of the points, delete the hull points, compute the convex hull of this smaller point-set, repeating this process till we run out of points.

The resulting sequence of convex layers form a graph called the onion.

29

⟨Functions for generating various graphs 25⟩+≡

(5) ▲28 ▲32 ▷

```
def get_onion_graph(points):
    from scipy.spatial import ConvexHull
    points      = np.asarray(points)
    points_tmp  = points.copy()
    numpts     = len(points)
    onion_graph = nx.Graph()
    numpts_proc = -1

    ⟨ Definition of circular_edge_zip 30c ⟩
    while len(points_tmp) >= 3:
        ⟨ Generate convex hull of points remaining in points_tmp 30a ⟩
        ⟨ Update onion_graph 30b ⟩
        ⟨ Remove points reported in the convex hull from points_tmp 30d ⟩

        if len(points_tmp) == 2:
            ⟨ Join two remaining points by an edge in onion_graph 31a ⟩
        elif len(points_tmp) == 1:
            ⟨ Add the remaining points as a node in onion_graph 31b ⟩
```

```
onion_graph.graph['type'] = 'onion'
return onion_graph
```

Note that the convex hull is computed by Scipy using the Qhull library as mentioned in the [docs](#).

30a $\langle \text{Generate convex hull of points remaining in points_tmp 30a} \rangle \equiv$ (29)

```
hull          = ConvexHull(points_tmp)
pts_on_hull   = [points_tmp[i] for i in hull.vertices]
coords        = [{"coods":pt} for pt in pts_on_hull]
```

30b $\langle \text{Update onion_graph 30b} \rangle \equiv$ (29)

```
new_node_idxs = range(numpts_proc+1, numpts_proc+len(hull.vertices)+1)
onion_graph.add_nodes_from(zip(new_node_idxs, coords))
onion_graph.add_edges_from(circular_edge_zip(new_node_idxs))
```

Given a set of node ids of a graph provided as a list of integers, the following function, returns a cycle of edges with successive nodes joined in the order provided. e.g. $[1, 2, 3] \rightarrow [(1, 2), (2, 3), (3, 1)]$. Convenient to have this defined separately.

30c $\langle \text{Definition of circular_edge_zip 30c} \rangle \equiv$ (29)

```
def circular_edge_zip(xs):
    xs = list(xs) # in the event that xs is of the zip or range type
    if len(xs) in [0,1] :
        zipl = []
    elif len(xs) == 2 :
        zipl = [(xs[0],xs[1])]
    else:
        zipl = list(zip(xs,xs[1:]+xs[:1]))
    return zipl
```

30d $\langle \text{Remove points reported in the convex hull from points_tmp 30d} \rangle \equiv$ (29)

```
numpts_proc  = numpts_proc + len(hull.vertices)
rem_pts_idxs = list(set(range(len(points_tmp)))-set(hull.vertices))
points_tmp   = [ points_tmp[idx] for idx in rem_pts_idxs ]
coords        = [{"coods":pt} for pt in points]
```

There are two edge cases: when only two points and one point remain. These cases, cannot be handled by Qhull (It reports an error at the terminal, saying it needs at least three points must be provided as input). Hence the separate treatment in the following two chunks.

When two nodes, remain, we just join them by an edge.

31a $\langle \text{Join two remaining points by an edge in onion_graph 31a} \rangle \equiv$ (29)

```
p, l = numpts_proc+1, numpts_proc+2
onion_graph.add_node(p)
onion_graph.add_node(l)
onion_graph.nodes[p]['coods'] = points_tmp[0]
onion_graph.nodes[l]['coods'] = points_tmp[1]
onion_graph.add_edge(p,l)
```

No edges to add here, just the node.

31b $\langle \text{Add the remaining points as a node in onion_graph 31b} \rangle \equiv$ (29)

```
l = numpts_proc+1
onion_graph.add_node(l)
onion_graph.nodes[l]['cood'] = points_tmp[0]
```

6.5 Gabriel Graph

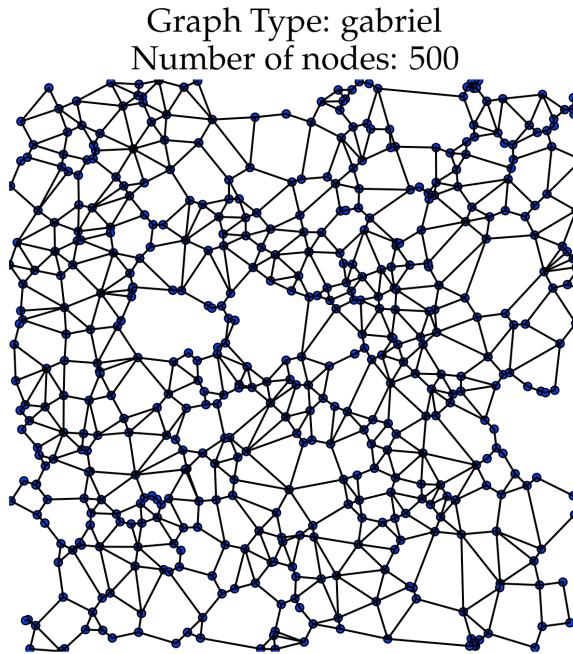


Figure 9: An example of the gabriel graph generated by the routine in this section on 500 uniformly distributed random points

Gabriel graph

From Wikipedia, the free encyclopedia

In [mathematics](#) and [computational geometry](#), the **Gabriel graph** of a set S of points in the [Euclidean plane](#) expresses one notion of proximity or nearness of those points. Formally, it is the [graph](#) G with vertex set S in which any points $p \in S$ and $q \in S$ are adjacent precisely if they are distinct, i.e. $p \neq q$, and the closed [disc](#) of which [line segment](#) \overline{pq} is a [diameter](#) contains no other elements of S . Gabriel graphs naturally generalize to higher dimensions, with the empty disks replaced by empty closed [balls](#). Gabriel graphs are named after [K. Ruben Gabriel](#), who introduced them in a paper with [Robert R. Sokal](#) in 1969.

Figure 10: The Definition of the Gabriel Graph from Wikipedia

Thanks Sam for writing this!!

32 ⟨*Functions for generating various graphs* 25⟩+≡ (5) ▲29 ▲34

```
def get_gabriel_graph(points):
    from scipy.spatial import Voronoi

    def ccw(A,B,C):
        return (C[1]-A[1]) * (B[0]-A[0]) > (B[1]-A[1]) * (C[0]-A[0])
    def intersect(A,B,C,D):
        return ccw(A,C,D) != ccw(B,C,D) and ccw(A,B,C) != ccw(A,B,D)
```

```

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

vor = Voronoi(points)
center = vor.points.mean(axis=0)

for (p1, p2), (v1, v2) in zip(vor.ridge_points, vor.ridge_vertices):
    if v2<0:
        v1, v2 = v2, v1
    if v1 >= 0: # bounded Voronoi edge
        if intersect(vor.points[p1], vor.points[p2],
                     vor.vertices[v1], vor.vertices[v2]):
            gabriel.add_edge(p1,p2)
        continue
    else: # unbounded Voronoi edge
        # compute "unbounded" edge
        p1p2 = vor.points[p2] - vor.points[p1]
        p1p2 /= np.linalg.norm(p1p2)
        normal = np.array([-p1p2[1], p1p2[0]])

        midpoint = vor.points[[p1, p2]].mean(axis=0)
        direction = np.sign(np.dot(midpoint - center, normal)) * normal
        length = max(2*np.linalg.norm(vor.points[p1]-vor.vertices[v2]),
                    2*np.linalg.norm(vor.points[p2]-vor.vertices[v2]))
        far_point = vor.vertices[v2] + direction * length

        if intersect(vor.points[p1], vor.points[p2],
                     vor.vertices[v2], far_point):
            gabriel.add_edge(p1,p2)
gabriel.graph['type'] = 'gabriel'
return gabriel

```

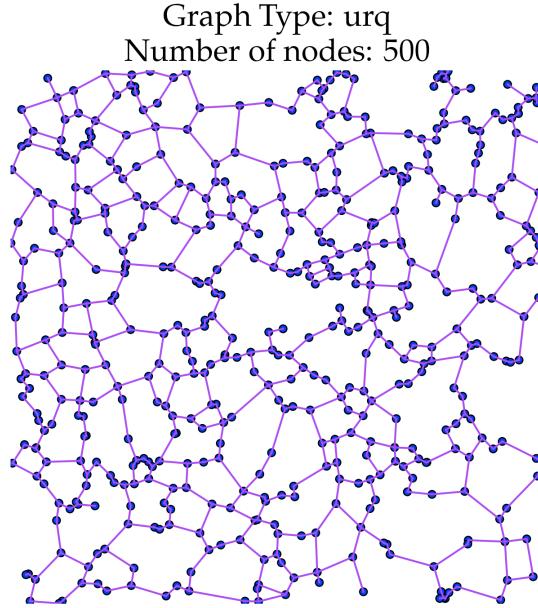


Figure 11: An example of the Urquhart graph on uniformly generated 500 randomly generated points

6.6 Urquhart Graph

the Urquhart graph of a set of points in the plane, is obtained by removing the longest edge from each triangle in the Delaunay triangulation.

Thanks to Sam again for this routine!

```
34 <Functions for generating various graphs 25>+≡ (5) ◀32 ▶36
def get_urquhart_graph(points):
    from scipy.spatial import Delaunay
    points      = np.array(points)
    coords      = [{"coods":pt} for pt in points]
    tri         = Delaunay(points)
    urq_graph = nx.Graph()

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

    edge_list = []
    longest_edge_list = []
    for (i,j,k) in tri.simplices:
        edges = [(i,j),(j,k),(k,i)]
        norms = [np.linalg.norm(points[j]-points[i]),
                 np.linalg.norm(points[k]-points[j]),
                 np.linalg.norm(points[i]-points[k])]
        zipped = zip(edges,norms)
        sorted_edges = sorted(zipped, key = lambda t: t[1])
        longest_edge = sorted_edges[-1]
        if longest_edge[1] > 1.0:
            edge_list.append(longest_edge)
        else:
            longest_edge_list.append(longest_edge)
```

```
longest_edge_list.append(sorted_edges[2][0])
edge_list.extend([(i,j),(j,k),(k,i)])
urq_graph.add_edges_from( edge_list )
urq_graph.remove_edges_from( longest_edge_list )

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

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

urq_graph.graph['weight'] = total_weight_of_edges
urq_graph.graph['type'] = 'urq'

return urq_graph
```

6.7 Traveling Saleman Tour (Cycle)

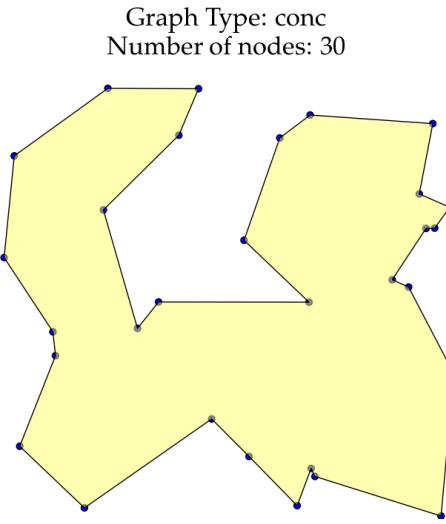


Figure 12: Example of an optimal TSP tour computed with Concorde on 30 randomly generated points

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 serves as a useful backup routine, in the event that a machine faces problems with the installation of PyConcorde.

* Using the `tsp` library

```
36   ⟨Functions for generating various graphs 25⟩+≡ (5) ▷34 38▷
    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 37a⟩
        ⟨Compute weight of each edge and total edge weight 37b⟩
        ⟨Set graph attributes 37c⟩
        return tsp_graph
```

37a \langle Generate TSP cycle and convert into NetworkX graph 37a $\rangle \equiv$ (36)

```
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 )
```

37b \langle Compute weight of each edge and total edge weight 37b $\rangle \equiv$ (36)

```
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
```

37c \langle Set graph attributes 37c $\rangle \equiv$ (36)

```
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 for the EUC_2D cases, the Concorde solver works only on points with integer coordinates. Since our synthetic data-sets 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.

Note that Concorde crashes when you pass it only three points. Probably something to do with its internals. Of course for the case of one or two points, the package explicitly informs us that we must pass a longer list.

38 *Functions for generating various graphs* 25) +≡ (5) ▲36

```
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]

    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
```

6.8 Graph Powers

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).

For the special case of the TSP, we render it as a polygon patch, because the interior needs to be colored.

39a \langle Functions for plotting and interacting 16 $\rangle + \equiv$ (5) ▷24

```
def render_graph(G,fig,ax):
    if G is None:
        return
    {Set up edge colors depending on graph type 39b}
    if G.graph['type'] not in ['conc', 'pytsp']:
        {Iterate through graph edges and draw as segments 40a}
    else:
        {Draw tour as polygon patch 40b}

    ax.axis('off') # turn off box surrounding plot
    fig.canvas.draw()
```

39b \langle Set up edge colors depending on graph type 39b $\rangle \equiv$ (39a)

```
edgecol = None
if G.graph['type'] == 'mst':
    edgecol = 'g'
elif G.graph['type'] == 'onion':
    edgecol = 'gray'
elif G.graph['type'] == 'gabriel':
    edgecol = (153/255, 102/255, 255/255)
elif G.graph['type'] == 'urq':
    edgecol = (255/255, 102/255, 153/255)
elif G.graph['type'] in ['conc', 'pytsp']:
    edgecol = 'r'
elif G.graph['type'] == 'dt':
    edgecol = 'b'
elif G.graph['type'][-3:] == 'nng':
    edgecol = 'm'
```

40a \langle Iterate through graph edges and draw as segments 40a $\rangle \equiv$ (39a)

```
#for elt in list(G.nodes(data=True)):
#    print(elt)

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)
```

Because the *interior* of the tour has to be colored, we render it as a polygon patch, and not just as a bunch of edges. Since I've stored the tour as a generic graph, the `.edges` data member of such a container, does not necessarily report the edges in the order encountered along the TSP.

But this order can trivially be extracted using depth first search.

40b \langle Draw tour as polygon patch 40b $\rangle \equiv$ (39a)

```
from networkx.algorithms.traversal.depth_first_search import dfs_edges
node_coods = []
for (nidx1, nidx2) in dfs_edges(G):
    node_coods.append(G.nodes[nidx1]['coods'])
    node_coods.append(G.nodes[nidx2]['coods'])

node_coods = np.asarray(node_coods)
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection

# mark the nodes
xs = [pt[0] for pt in node_coods]
ys = [pt[1] for pt in node_coods]
ax.scatter(xs,ys)

polygon = Polygon(node_coods, closed=True, \
                  facecolor=(255/255, 255/255, 102/255, 0.4), \
                  edgecolor='k', linewidth=1)
ax.add_patch(polygon)
```

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.

41a *⟨Functions dealing with intersecting two geometric graphs 41a⟩*≡ (5) 41b▷

```
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 each edge from one of the graphs and check whether it exists in the other.

41b *⟨Functions dealing with intersecting two geometric graphs 41a⟩*+≡ (5) ▷41a 41c▷

```
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.

41c *⟨Functions dealing with intersecting two geometric graphs 41a⟩*+≡ (5) ▷41b

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

9 Checking Hamiltonicity

During experiments, we will often require to check if a graph is Hamiltonian. Of course this task is NP -complete, but it is easy to implement a naive backtracking routine that should be competitive enough for dense graphs with upto 50 nodes , or larger but sparser sparse graphs. Such a routine will be particularly helpful for honing hypotheses and generating interesting examples.

Because the author is lazy, he will unabashedly copy and paste code for such a routine from <https://gist.github.com/mikkelman/ab7966e7ab1c441f947b>

If needed, it could be sped up by other algorithmic techniques. But the time for such micro-optimizations, like that of casting Excalibur back into the Lake, is far off.

42 {Generic utility classes and functions 11}+≡ (5) ◀15 43a▷

```
def hamilton(G):
    F = [(G,[list(G.nodes())[0]])]
    n = G.number_of_nodes()
    while F:
        graph,path = F.pop()
        confs = []
        neighbors = (node for node in graph.neighbors(path[-1]))
        if node != path[-1]): #exclude self loops
        for neighbor in neighbors:
            conf_p = path[:]
            conf_p.append(neighbor)
            conf_g = nx.Graph(graph)
            conf_g.remove_node(path[-1])
            confs.append((conf_g,conf_p))
        for g,p in confs:
            if len(p)==n:
                return p
            else:
                F.append((g,p))
    return None
```

10 Perturbing points by prescribed factor randomly

Many point-sets are degenerate, i.e. have many points that are collinear or are co-circular. It thus helps to perturb the points slightly for which this is the case.

The amount of perturbation is controlled by the parameter α (where $\alpha = 0$) corresponds to zero perturbation. Be careful when choosing α ; this will depend on the point-cloud you are perturbing.

43a \langle Generic utility classes and functions 11 $\rangle + \equiv$ (5) ▲42 43b ▷

```
def perturb_points(points, alpha=0.01):
    points = np.asarray(points)

    for i in range(len(points)):
        theta      = random.uniform(0,2*np.pi)
        unitvec   = np.asarray([np.cos(theta), np.sin(theta)])
        points[i] = points[i] + alpha * unitvec

    return points
```

11 Calculating the Length of a Polygonal Chain

We assume for the input that points along a polygonal chain have been provided in the order they occur. The routine computes the length of that same chain.

If the boolean flag `cycle_p` has been set to `True`, then the poly-chain is now assumed to be a cycle i.e. the last point is joined to the first point. So in this case, the routine returns the length of the polygonal chain *plus* the length of this last edge that closes the cycle.

43b \langle Generic utility classes and functions 11 $\rangle + \equiv$ (5) ▲43a 44 ▷

```
def length_poly_chain(points, cycle_p=False):

    points      = [np.asarray(pt) for pt in points]
    seglengths = [np.linalg.norm(p1-p2) for (p1,p2) in zip(points, points[1:])]

    if cycle_p :
        seglengths.append(np.linalg.norm(points[-1]-points[0]))

    return sum(seglengths)
```

12 Data Analysis Functions

Given a stream of repeated data recordings x_1, x_2, \dots, x_n for each fixed set of parameters in experiments, this function reduces this stream to three numbers $(\mu_x - \sigma_x, \mu_x, \mu_x + \sigma_x)$

In slightly more technical language, especially for those who familiar with Haskell, this function accepts input of the type `[Double]` and “reduces” each element (every element in the outer array is of type `[Double]`) with a tuple of three numbers $(\mu - \sigma, \mu, \mu + \sigma)$ which represents an error interval $[\mu - \sigma, \mu + \sigma]$ of a single standard deviation above and below the mean μ of the stream of numbers.

44 *⟨Generic utility classes and functions 11⟩+≡* (5) ◀43b 45 ▷

```
def error_intervals(xss):
    means          = [np.mean(xs) for xs in xss]
    stds          = [np.std(xs)   for xs in xss]
    error_intervals = [ (means[i]-stds[i], \
                        means[i]           , \
                        means[i]+stds[i]) for i in len(xss)]
    return error_intervals
```

13 Computing the Maximally Dispersed Vertex Cover of a Graph

In the following routine we compute the maximally dispersed vertex cover for the supplied graph of the delaunay triangulation on the graph. Returns a list of indices corresponding to the nodes of the supplied graph

The algorithm implemented below has been described in *Minimum Diameter Covering Problems* that can be downloaded from <http://www.ams.sunysb.edu/~estie/papers/diam.ps.gz>. It solves the problem *exactly* on a graph by using 2-sat. See Theorem 4.1, Page 10 from that paper.

The following implements the core routine which uses 2-sat to solve the decision question of computing the existence of a vertex cover with the specified dispersion.

I believe calculating the maximally dispersed vertex cover a graph might yield good approximation algorithms for network design problems when combined with some inefficient dynamic programming algorithm. For instance here is one possible heuristic for computing the TSP. Compute the Delaunay Triangulation, calculate the max dispersed vertex cover, then repeat, by computing the delaunay triangulation of this new vertex cover and so on. Recurse downwards until you have a small point-set and then iteratively and greedily build up the solution from the bottom up. In a sense, I would like to claim maximally dispersed vertex covers of delaunay triangulations form a sort of “universal core-set” but I should of course put my money where my mouth is and give some performance bounds for this rather tall claim. At the moment it looks like a cute framework in which to design algorithms for low-cost networks. If it gives a constant factor approximation algorithm to the TSP, then the resulting tour intuitively looks simple (i.e. does not self intersect). However, Joe just mentioned today via email, that this problem was solved early this year by two guys from Sharif University in Iran, using so-called Snowflake trees. Might this offer a simpler alternative? Let’s see!!

Implementing this routine is a first step in getting good examples and honing my hypotheses. Another reason for inserting this piece of code is I am curious how many edges of the resulting tour which I use this routine to construct shares its edges with the delaunay triangulation.

Exploring the maximally dispersed vertex covers (and getting faster algorithms for them instead of the 2-sat algorithm below that takes $O(n^2 \log n)$) for various “standard” graphs like Delaunay graphs, or MSTs seems like an interesting set of questions to tackle. Can we do $O(n \text{polylog}(n))$ for them? Maybe use Hochbaum Mass for getting good approximations in $O(n \text{polylog}(n))$?

The motivation for looking at Estie’s paper was computing maximally dispersed guards in the annulus to answer Abhishek’s lighting parking lots questions which he faced at Phillips that I reformulated as an MDG problem, that hasn’t been looked at in literature yet. (Sandor’s student has a hardness proof for polygons with holes, but the problem remains of interest in a convex annulus with a single convex hole.)

45

(Generic utility classes and functions 11) +≡

(5) ▲44 47a►

```
def exists_disp_vertex_cover_more_than_bin_guess(graph,points,d):
    from satispy import Variable, Cnf
    from satispy.solver import Minisat
```

```

numpts = len(points)
exp    = Cnf()

⟨Set up variables  $x_i$  for each node of the graph 46a⟩
⟨Set up clauses for each edge 46b⟩
⟨Create a clause for each pair of nodes whose distance is smaller than  $d$  46c⟩
⟨Solve the system with Minisat 46d⟩

```

46a ⟨Set up variables x_i for each node of the graph 46a⟩≡ (45)

```

xs = []
for idx in range(numpts):
    xs.append(Variable( 'x_'+str(idx) ))
print("      Created Variables....")

```

46b ⟨Set up clauses for each edge 46b⟩≡ (45)

```

for (i,j) in list(graph.edges):
    c = Cnf()
    c = xs[i] | xs[j]
    exp &= c
print("      Created Edge Clauses...")

```

46c ⟨Create a clause for each pair of nodes whose distance is smaller than d 46c⟩≡ (45)

```

for i in range(numpts):
    for j in range(i+1, numpts):
        if np.linalg.norm(points[i]-points[j]) < d:
            c = Cnf()
            c = -xs[i] | -xs[j]
            exp &= c
print("      Created All Pair Clauses....")

```

Minisat is an extremely efficient solver for solving large SAT instances, freely available from <http://minisat.se/>. I am using it through the Python interface.

46d ⟨Solve the system with Minisat 46d⟩≡ (45)

```

solver = Minisat()
solution = solver.solve(exp)

if solution.success:
    print(Fore.CYAN, "      2-SAT Solution found!", Style.RESET_ALL)
    return (True, [idx for idx in range(numpts) if solution[xs[idx]] == True])
else:
    print(Fore.RED, "      2-SAT Solution cannot be found", Style.RESET_ALL)
    return (False, [])

```

The following routine calls the function defined above, and uses binary search to hone down onto a good vertex cover. Note that the reason `bin_high` has been set to $\sqrt{2}$ since all points lie inside the unit-square.

47a

(Generic utility classes and functions 11) +≡ (5) ◀45 47b▶

```
def get_max_disp_vc_nodes(graph, points):

    tol          = 1e-6      # configurable
    bin_low, bin_high = 0.0, 2.0 # initial interval limits for binary search
    old_disp_idxs = range(len(points))

    while abs(bin_high - bin_low) > tol:

        assert bin_high >= bin_low
        bin_guess = (bin_high + bin_low)/2.0

        print ("\nBinary search Interval Limits are ", "[", bin_low, " , ", bin_high, "]")
        print ("Guessed dispersion value currently is ", bin_guess)
        print ("Checking for vertex cover with ", bin_guess, " dispersion.....")

        [exist_flag_p, disp_idxs] = exists_disp_vertex_cover_more_than_bin_guess(graph,points)

        if exist_flag_p:
            bin_low      = bin_guess
            old_disp_idxs = disp_idxs
        else:
            bin_high = bin_guess

    return (old_disp_idxs, bin_guess)
```

47b

(Generic utility classes and functions 11) +≡ (5) ◀47a 47c▶

```
def list_points_in_node_order(graph):
    return [graph.nodes[idx]['coods'] for idx in range(len(graph.nodes))]
```

47c

(Generic utility classes and functions 11) +≡ (5) ◀47b 48▶

```
class Forest:
    def __init__(self,rootNodes):
        self.rootNodes = rootNodes

class Node:
    def __init__(self, point, children=[]):
        self.point      = np.asarray(point)
        self.children = children
```

```
48 <Generic utility classes and functions 11>+≡ (5) ▷47c
def build_delaunay_forest(points, stopnum=10):
    from sklearn.neighbors import NearestNeighbors
    points = np.asarray([np.asarray(pt) for pt in points])
    while len(points) >= stopnum :
        del_graph = get_delaunay_tri_graph(points)
        max_disp_vc_nodes, dispersion = get_max_disp_vc_nodes(del_graph, points)
        vcpoints = np.asarray([points[idx] for idx in max_disp_vc_nodes])
        nbrs = NearestNeighbors(n_neighbors=5, algorithm='ball_tree')
        distances, indices = nbrs.kneighbors(vcpoints)
        print(distances)
        print(indices)
        print(len(vcpoints))
        points = vcpoints

    sys.exit()
    return delaunay_forest
```

14 Experiments

14.1 Intersection behavior for TSPLIB instances

49 $\langle Experiments \text{ 49} \rangle \equiv$

(5) 52▷

```
def expt_tsplib_intersection():

    dirName = './tsplib-expts/'
    shutil.rmtree(dirName, ignore_errors=True)
    try:
        os.mkdir(dirName)
        print("Directory " , dirName , " Created ")
    except FileExistsError:
        print("Directory " , dirName , " already exists")

    maxpts_bound      = 1100
    instance_file_names = get_names_of_all_euclidean2D_instances()

    f = open("./slides-beamer/slidedecode.tex", "w")

    for fname in instance_file_names:

        inst_points = shift_and_scale_to_unit_square(tsplib_instance_points(fname))

        if len(inst_points) > maxpts_bound:
            print(Fore.RED, 'Skipping file', fname, ' which has ', len(inst_points) , '>', m
        else:
            print(fname , 'has ' , len(inst_points) , ' points')
            print('...Testing for intersections')
            tsp_graph = get_concorde_tsp_graph(inst_points)

            graph_fns = [(get_mst_graph      , 'MST') , \
                         (get_urquhart_graph , 'Urquhart') , \
                         (get_gabriel_graph , 'Gabriel') , \
                         (get_delaunay_tri_graph, 'Delaunay')]

            from functools import partial
            for k in range(1,4):
                graph_fns.append((partial(get_knng_graph, k=k), str(k)+'-NNG'))

            num_tsp_edges      = len(tsp_graph.edges)
```

```

percentage_intersecn = []

print(Fore.YELLOW,'....COMPUTING INTERSECTIONS OF TSP WITH VARIOUS GRAPHS')
for ctr, (fn_body, fn_name) in zip(range(1,1+len(graph_fns)), graph_fns):
    print(Fore.CYAN+'-----> Intersecting with '+fn_name+Style.RESET_ALL)
    geometric_graph          = fn_body(inst_points)
    num_graph_edges           = len(geometric_graph.edges)
    common_edges               = list_common_edges(tsp_graph, geometric_graph)
    num_common_edges_with_tsp = len(common_edges)
    percentage_intersecn.append(100*num_common_edges_with_tsp/num_tsp_edges)

# plot the tour
fig,ax = plt.subplots()
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([])

render_graph(tsp_graph,fig,ax)
inst_name = fname[:-4]
plt.savefig(dirName+inst_name+'-concorde-tsp.pdf',bbox_inches='tight')
plt.close()

# plot the intersection percentage bar-chart
plt.style.use('ggplot')
x           = [grtype for (_,grtype) in graph_fns]
x_pos        = [i for i, _ in enumerate(x)]

plt.bar(x_pos, percentage_intersecn, color='blue')
plt.xlabel("Graph Type")
plt.ylabel("Percentage of Intersections")
plt.title("Percentage of intersections of TSP with various graphs on " + fname)

plt.xticks(x_pos, x)
plt.savefig(dirName+inst_name+'-intersection-chart.pdf',bbox_inches='tight')

# write slide code to file

slidecode="%%%%%%%%%%%%%\n"
slidecode+="\\begin{frame}{Intersections of Concorde TSP with TSPLIB instances on %s}\n"
slidecode+="\\begin{figure}[ht]\n"

```

```
slidecode+=" \\\ncentering\n"
slidecode+=" \\\nincludegraphics[width=4cm]{../tsplib-expts/" + inst_name+"-conco
slidecode+=" \\\nincludegraphics[width=8cm]{../tsplib-expts/" + inst_name+"-inter
slidecode+="\\end{figure}\n"
slidecode+="\\end{frame}\n"
f.write(slidecode)

plt.close()

f.close()
```

Intersection % between Euc. 2D TSP and Various Graphs on Random Uniform points in $[0, 1]^2$

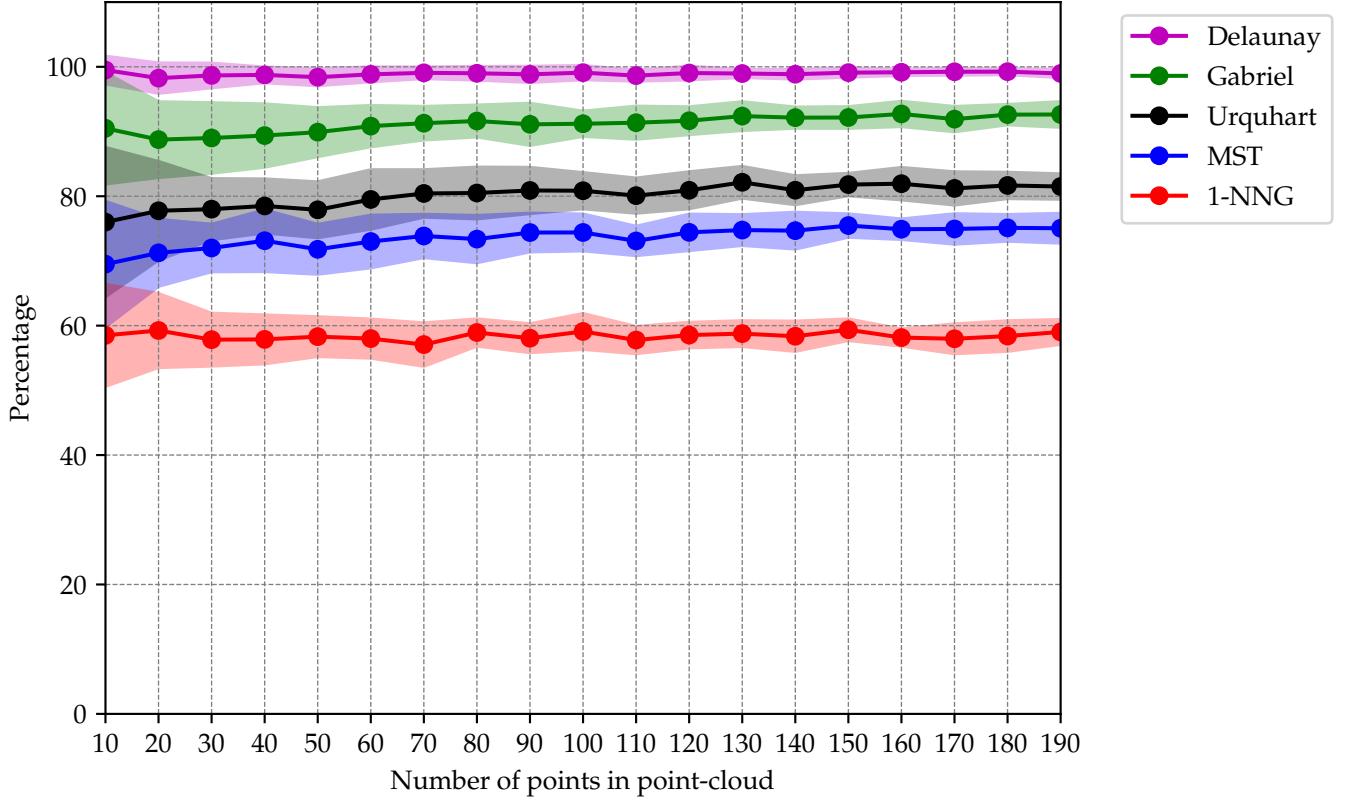


Figure 13: Over uniformly randomly generated instances in $[0, 1]^2$ the percentage of the intersections of 1-nng, mst, delaunay with TSP is plotted.

14.2 Intersection behavior for uniformly randomly generated instances

In this experiment we take point clouds of size `ptsmin`, `ptsmin+skipval`, `ptsmin+2*skipval` ... upto `ptsmax` (or thereabouts) For each such point-cloud size we generate `numrunsper` point-clouds, and then find how often the TSP generated with Concorde and the various graph types intersect one another (in terms of percentage).

For each point-cloud size we plot we calculate the average percentage and then plot the resulting curve. Also to account for error, we also shade the interval lying one standard deviation below the mean and one standard deviation above i.e. $[\mu - \sigma, \mu + \sigma]$ is shaded for each point-cloud size.

52

(Experiments 49)+≡

(5) ▲49

```

def expt_intersection_behavior():
    expt_name          = 'expt_non_uniform_intersection_behavior'
    expt_plot_file_extn = '.pdf'

    ptsmin   = 10
    ptsmax   = 300
    skipval  = 20

```

```

numrunsper      = 15
cols_1nng       = {}
cols_mst        = {}
cols_gabriel   = {}
cols_urquhart  = {}
cols_delaunay  = {}

for numpts in range(ptsmin,ptsmax,skipval):
    print(Fore.CYAN, "\n ===== \n Tackling numpts = ", numpts, "\n=====\n")
    time.sleep(0.50)
    cols_1nng[numpts]      = []
    cols_mst[numpts]        = []
    cols_gabriel[numpts]    = []
    cols_urquhart[numpts]  = []
    cols_delaunay[numpts]  = []

    for runcount in range(numrunsper):
        print(Fore.CYAN, ".....Run number:", runcount, " for numpts = ", numpts, Style.RESET_ALL)
        time.sleep(0.25)

        #pts           = uniform_points(numpts)
        pts            = multimodal_points(numpts, nummodes=4, sigma=0.05)
        nng1_graph    = get_knng_graph(pts,k=1)
        mst_graph     = get_mst_graph(pts)
        gabriel_graph = get_gabriel_graph(pts)
        urquhart_graph = get_urquhart_graph(pts)
        del_graph     = get_delaunay_tri_graph(pts)
        conctsp_graph = get_concorde_tsp_graph(pts)

        cols_1nng[numpts].append(100*len(list_common_edges(nng1_graph,conctsp_graph))/len(nng1_graph))
        cols_mst[numpts].append(100*len(list_common_edges(mst_graph,conctsp_graph)) / len(mst_graph))
        cols_gabriel[numpts].append(100*len(list_common_edges(gabriel_graph,conctsp_graph)) / len(gabriel_graph))
        cols_urquhart[numpts].append(100*len(list_common_edges(urquhart_graph,conctsp_graph)) / len(urquhart_graph))
        cols_delaunay[numpts].append(100*len(list_common_edges(del_graph,conctsp_graph)) / len(del_graph))

fig, ax = plt.subplots()
ax.set_title(r"Intersection \% between Euc. 2D TSP and Various Graphs" "\n" r"on Random")
ax.set_xlim([ptsmin,ptsmax-skipval])
ax.set_ylim([0,110])
ax.set_xlabel("Number of points in point-cloud")
ax.set_ylabel("Percentage")
ax.set_xticks(np.arange(ptsmin,ptsmax,step=3*skipval))

def arithmetic_mean(nums):

```

```

    return sum(nums)/len(nums)

cols_1nng_min = [ min(cols_1nng[key]) for key in sorted(cols_1nng) ]
cols_1nng_max = [ max(cols_1nng[key]) for key in sorted(cols_1nng) ]
cols_1nng_am = np.asarray([ arithmetic_mean(cols_1nng[key]) for key in sorted(cols_1nng) ])
cols_1nng_std = np.asarray([ np.std(cols_1nng[key]) for key in sorted(cols_1nng) ])

cols_mst_min = [ min(cols_mst[key]) for key in sorted(cols_mst) ]
cols_mst_max = [ max(cols_mst[key]) for key in sorted(cols_mst) ]
cols_mst_am = np.asarray([ arithmetic_mean(cols_mst[key]) for key in sorted(cols_mst) ])
cols_mst_std = np.asarray([ np.std(cols_mst[key]) for key in sorted(cols_mst) ])

cols_gabriel_min = [ min(cols_gabriel[key]) for key in sorted(cols_gabriel) ]
cols_gabriel_max = [ max(cols_gabriel[key]) for key in sorted(cols_gabriel) ]
cols_gabriel_am = np.asarray([ arithmetic_mean(cols_gabriel[key]) for key in sorted(cols_gabriel) ])
cols_gabriel_std = np.asarray([ np.std(cols_gabriel[key]) for key in sorted(cols_gabriel) ])

cols_urquhart_min = [ min(cols_urquhart[key]) for key in sorted(cols_urquhart) ]
cols_urquhart_max = [ max(cols_urquhart[key]) for key in sorted(cols_urquhart) ]
cols_urquhart_am = np.asarray([ arithmetic_mean(cols_urquhart[key]) for key in sorted(cols_urquhart) ])
cols_urquhart_std = np.asarray([ np.std(cols_urquhart[key]) for key in sorted(cols_urquhart) ])

cols_delaunay_min = [ min(cols_delaunay[key]) for key in sorted(cols_delaunay) ]
cols_delaunay_max = [ max(cols_delaunay[key]) for key in sorted(cols_delaunay) ]
cols_delaunay_am = np.asarray([ arithmetic_mean(cols_delaunay[key]) for key in sorted(cols_delaunay) ])
cols_delaunay_std = np.asarray([ np.std(cols_delaunay[key]) for key in sorted(cols_delaunay) ])

xs = range(ptsmin,ptsmax,skipval)

# colors of lines corresponding to various graphs
nng1col='r'
mstcol='b'
gabcol='g'
urqcol='k'
dtcol='m'

ax.plot(xs,cols_delaunay_am,'o-', label='Delaunay',color=dtcol)
ax.fill_between(xs, cols_delaunay_am-cols_delaunay_std, \
                cols_delaunay_am+cols_delaunay_std , color=dtcol, alpha=0.3)

ax.plot(xs,cols_gabriel_am,'o-', label='Gabriel',color=gabcol)

```

```
ax.fill_between(xs, cols_gabriel_am-cols_gabriel_std, \
                cols_gabriel_am+cols_gabriel_std , color=gabcol, alpha=0.3)

ax.plot(xs,cols_urquhart_am,'o-', label='Urquhart',color=urqcol)
ax.fill_between(xs, cols_urquhart_am-cols_urquhart_std, \
                cols_urquhart_am+cols_urquhart_std , color=urqcol, alpha=0.3)

ax.plot(xs,cols_mst_am,'o-', label='MST',color=mstcol)
ax.fill_between(xs, cols_mst_am-cols_mst_std, \
                cols_mst_am+cols_mst_std , color=mstcol, alpha=0.3)

ax.plot(xs,cols_1nng_am,'o-', label='1-NNG',color=nng1col)
ax.fill_between(xs, cols_1nng_am-cols_1nng_std, \
                cols_1nng_am+cols_1nng_std , color=nng1col, alpha=0.3)

ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
ax.grid(color='gray',linestyle='-', linewidth=0.5)
plt.savefig(expt_name+expt_plot_file_extn, bbox_inches='tight')
print("Plot File written to disk")
```

14.3 Intersection behavior for TSPLIB instances

14 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.
- [Hel00] Keld Helsgaun. “An effective implementation of the Lin–Kernighan traveling salesman heuristic”. In: *European Journal of Operational Research* 126.1 (2000), pp. 106–130.
- [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>.
- [Sci] *Science and Culture: Solving a math problem to create art / PNAS*. <https://www.pnas.org/content/113/52/14873>. (Accessed on 10/28/2020).
- [BKEI09] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. “Yaml ain’t markup language (yamlTM) 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

A Installing and running the Code

The program can be downloaded from Github: <https://github.com/gtelang/tspnng>. Alternatively open a terminal and run the command, `git clone https://github.com/gtelang/tspnng.git`

The only other prerequisites for running the code, are the [Anaconda](#) distribution of Python 3 and a couple of other packages. To check if the Python executable is in your path ¹⁴ run the command `python --version`. If it succeeds, you have installed Anaconda!

The additional packages required can be installed by:

```
pip install colorama prettytable tsp satispy treelib15
git clone https://github.com/jvkersch/pyconcorde
cd pyconcorde
pip install -e .
```

To run the program, `cd` into the code's top-level folder, then type ¹⁶ any one of (depending on your use case):

- ❖ `python src/main.py --interactive`
- ❖ `python src/main.py --file <filename>`
- ❖ `python src/main.py --tsplibinstance <instancename>`

The ‘--interactive’ flag is for inserting a point-set with the mouse onto the canvas.

The ‘--file <filename>’ is for inserting a custom point set provided in a file onto the canvas (and if needed modifying the point set by adding more points).

When entering input points with a file make sure the file looks similar to the contents of the `foo.yaml` in the home directory of the code whose contents I've also shown in [Figure 14](#). Make sure that all coordinates are scaled so that the points in the file lie inside the unit-square. This makes it convenient for nice plotting and for different experiments.

Finally, very much similar to the last option but only for TSPLIB instances, we have ‘--tsplibinstance <instancename>’ where you just enter the filename corresponding to the appropriate TSPLIB instance. For instance by typing ‘`python src/main.py --tsplibinstance berlin52`’ runs the code on the ‘berlin52’ instance of TSPLIB.

¹⁴and that it is Python 3.7 or above

¹⁵If you don't have superuser access during installation or the installation process seems to get stuck, add the flag `--user` at the end

¹⁶On Windows replace, the forward slash ‘/’ by ‘\’

The code takes care of retrieving the appropriate file corresponding to each instance. Do not type ‘.yaml’ extension or directory name or anything like that when entering in the instance name. e.g. see how the ‘berlin52’ instance was passed to the code in the previous sentence

The list of instance names and the corresponding pictures can be found in can be found in Appendix B.

```
points:
  - [0,0]
  - [0.5,0.7]
  - [0.3,0.8]
  - [0.4,0.8]
  - [0.5,0.6]
  - [0.3,0.3]
  - [0.1,0.9]
```

Figure 14: Contents of the sample `foo.yaml` file in the home directory for inputting custom points into the code

UPDATE: (*Thanks Logan!!*) Some of you *might* face problems running the code if you work with the newest Anaconda distribution of Python 3.8.6, rather than Python 3.7.3 which I use.

If you face this issue, please look at the screenshot Figure 15 of Logan’s email to fix the problem. I leave my installation instructions above unchanged, in the event it works for some of you, so that you don’t have to bother mucking about with code-internals.

In any event, Logan’s suggested fix, is simple enough.

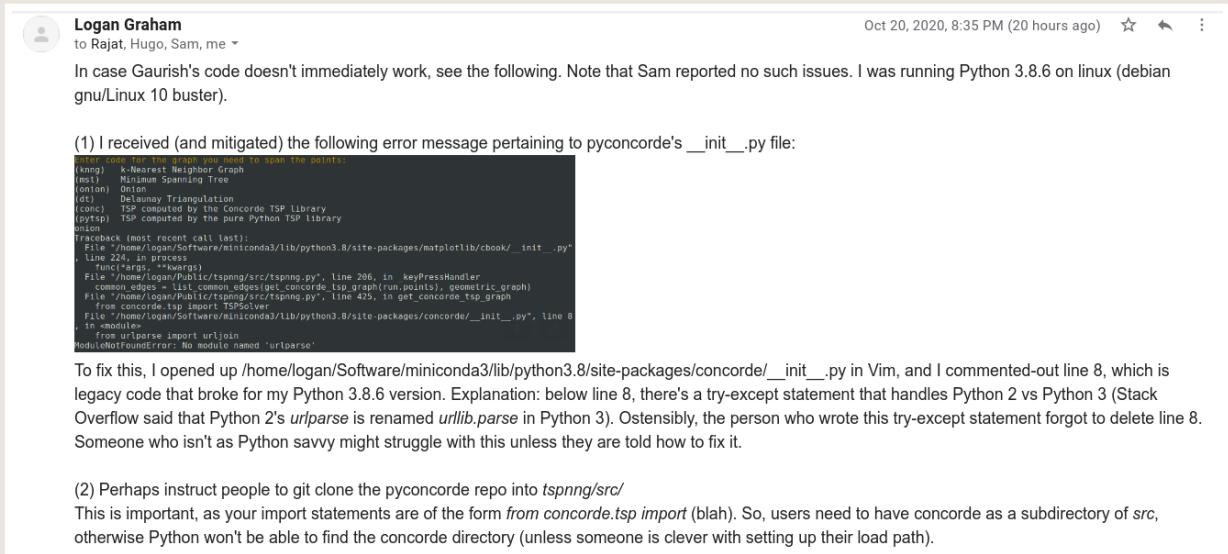


Figure 15: Screenshot of Logan’s email with an installation fix

As mentioned before one can mouse-in points onto a canvas (with double-clicks), or provide points via a file (and then add additional points via a canvas), run various network algorithms and render them onto a GUI canvas. *By the way, make sure the terminal is visible at all times during your interaction with the canvas, as it will often ask for input via prompts or display output information.*

Entering points with mouse clicks

After you mouse in your input points, press ‘**i**’ ; that will open a prompt at the terminal, asking which network do you want computed on those points. Enter the code inside the brackets ‘**(...)**’

Generating large random point-sets on the canvas

If you *don't* want to mouse-in points, and just need random points plastered uniformly across the canvas, press ‘**u**’ , and then type into the terminal the number of points. Other kinds of non-uniform point sets can also be generated using the following commands.

Note that while computing these point-sets, if there are points “left-over”, then the remaining points are added randomly and uniformly to the unit-square.

Non uniform points

(This is kind of badly named, sorry, couldn't think of a better one ¹⁷ *This a scheme I sometimes use when geneating non-uniform random points. Press ‘**n**’ It generates clusters of points, wherein the clusters near the square boundary are “tighter” i.e. have smaller diameter.*

Multi-modal distribution

Another scheme for non-uniform random points, that's more natural than the above one. Press ‘**m**’ and then type into the terminal the number of points, the number of modes, and the spread (σ) around each of the modes. The points are assumed normally distributed around each mode

Concentric circular points

Press ‘**o**’ and then type into the terminal number of points and number of rings.

Grid points

Press ‘**g**’ and then type into the terminal, the number of points and then the number of rows of the grid. The number of columns is automatically computed.

Note that after generating these points, you can continue mousing in additional points as part of the input. Useful if you want, say, an example with more or less uniform randomly generated points, except for a few small tight clusters here and there.

Computing TSP directly

¹⁷. To directly compute the TSP cycle on the points (without needing to go through the prompts of the previous step) just press ‘**t**’ .

Canvas should be active during keypresses

Note that when you press, any of the keys above, your matplotlib canvas *must* be an active window ¹⁸ . Only *then* does matplotlib detect the key presses (i.e. execute the appropriate call-back function).

¹⁷This is an option meant more for convenience than anything really; might be useful for trying to detect counter-examples

¹⁸Single click or tap the window title bar with the mouse to make the canvas active.

Modifying input

If you want to insert another point onto the existing point-set, just double-click at that position on the canvas. The computed networks are wiped clean off the canvas, and you can again compute the appropriate networks again as above.

Wiping the canvas clean

If you want the screen and the internal state wiped clean completely — say, to begin tinkering with a fresh set of points — press ‘**C**’.

P.S: You may see a warning — as I do — in the terminal during key-presses:

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 Matplotlib using Qt triggers that message. \(\backslash\)(\(\cup\))/. If you have any trouble — or detect a bug! — we can hash things out on Slack, Github or email.

B Catalog of symmetric 2D Euclidean instances inside TSPLIB

The framed box below contains the names of the subset of TSPLIB files corresponding to symmetric Euclidean 2D instances. The number inside each instance name denotes the number of points e.g. `berlin52.tsp` contains 52 points in \mathbb{R}^2 . Python has excellent YAML data parsers, and so I've converted the TSPLIB files below into YAML format.

These converted files have been stored in the folder

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

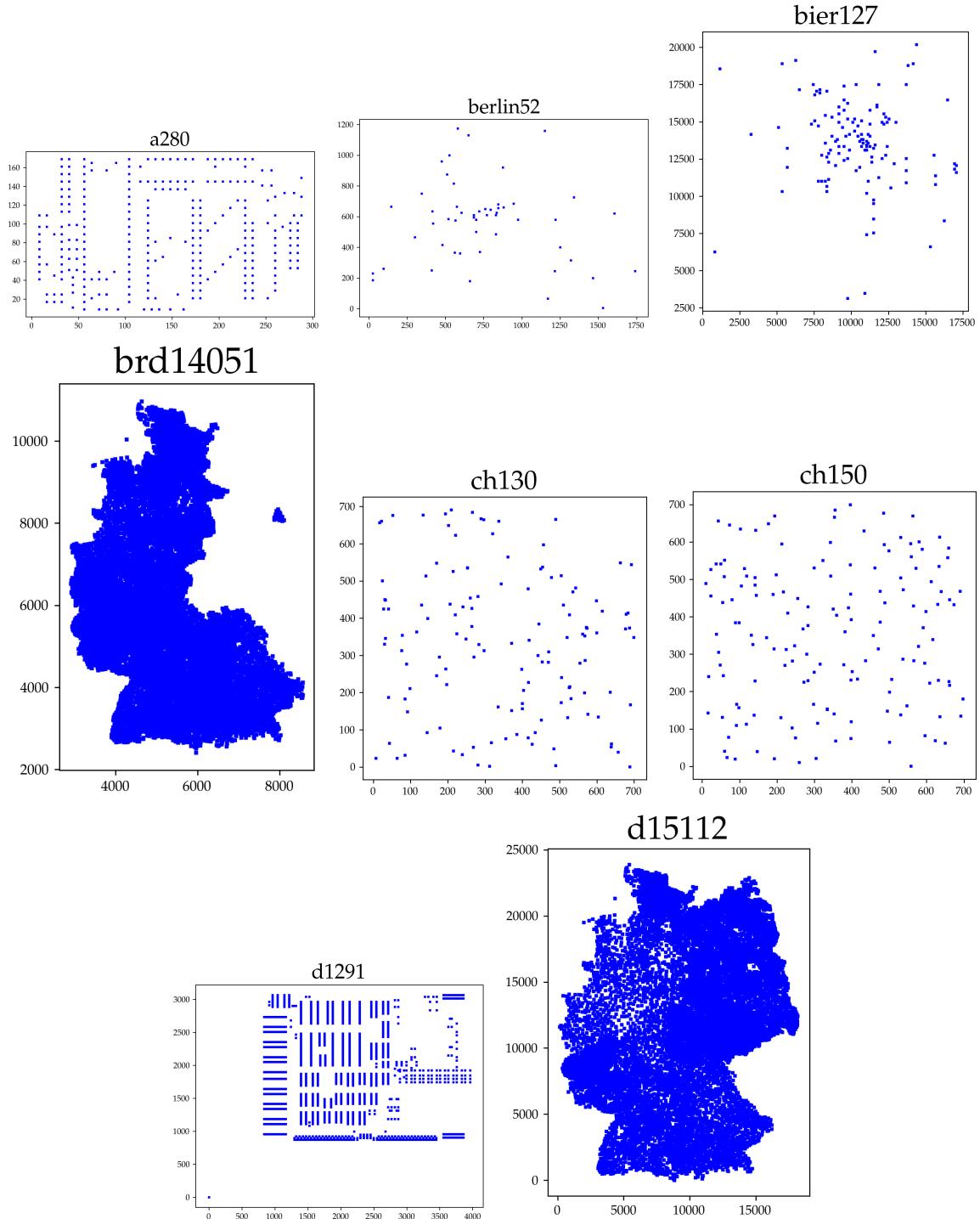
The original files (i.e the ones with `.tsp` extension) can be found in

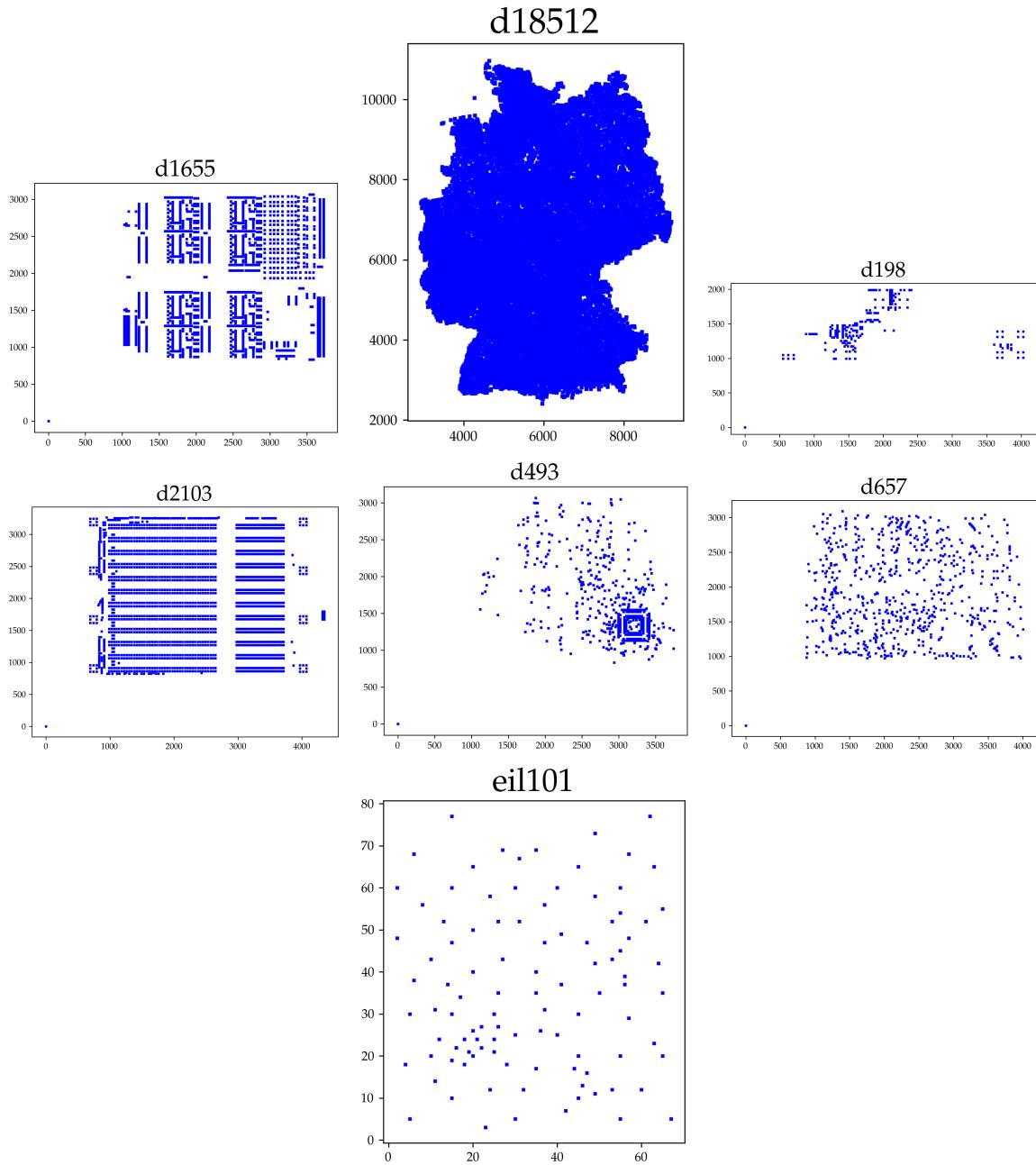
`./sym-tsp-tsplib/instances/tsplib_symmetric_tsp_instances`

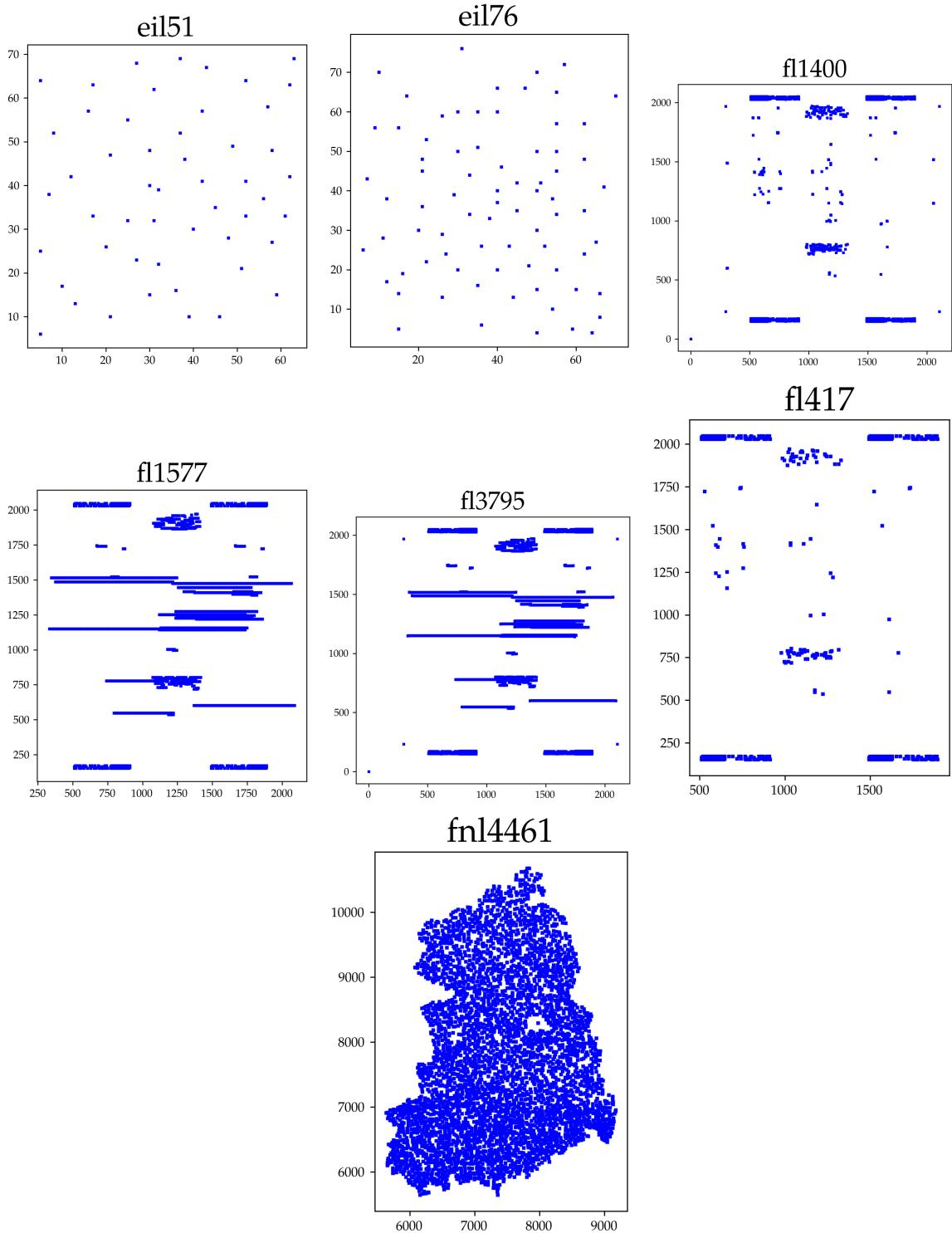
❖ a280	❖ fl3795	❖ pcb442	❖ rl1304
❖ berlin52	❖ fl417	❖ pr1002	❖ rl1323
❖ bier127	❖ fnl4461	❖ pr107	❖ rl1889
❖ brd14051	❖ gil262	❖ pr124	❖ rl5915
❖ ch130	❖ kroA100	❖ pr136	❖ rl5934
❖ ch150	❖ kroA150	❖ pr144	❖ st70
❖ d1291	❖ kroA200	❖ pr152	❖ ts225
❖ d15112	❖ kroB100	❖ pr226	❖ tsp225
❖ d1655	❖ kroB150	❖ pr2392	❖ u1060
❖ d18512	❖ kroB200	❖ pr264	❖ u1432
❖ d198	❖ kroC100	❖ pr299	❖ u159
❖ d2103	❖ kroD100	❖ pr439	❖ u1817
❖ d493	❖ lin105	❖ pr76	❖ u2152
❖ d657	❖ lin318	❖ rat195	❖ u2319
❖ eil101	❖ linhp318	❖ rat575	❖ u574
❖ eil51	❖ nrw1379	❖ rat783	❖ u724
❖ eil76	❖ p654	❖ rat99	❖ usa13509
❖ fl1400	❖ pcb1173	❖ rd100	❖ vm1084
- ❖ fl1577	❖ pcb3038	❖ rd400	❖ vm1748
		❖ rl11849	

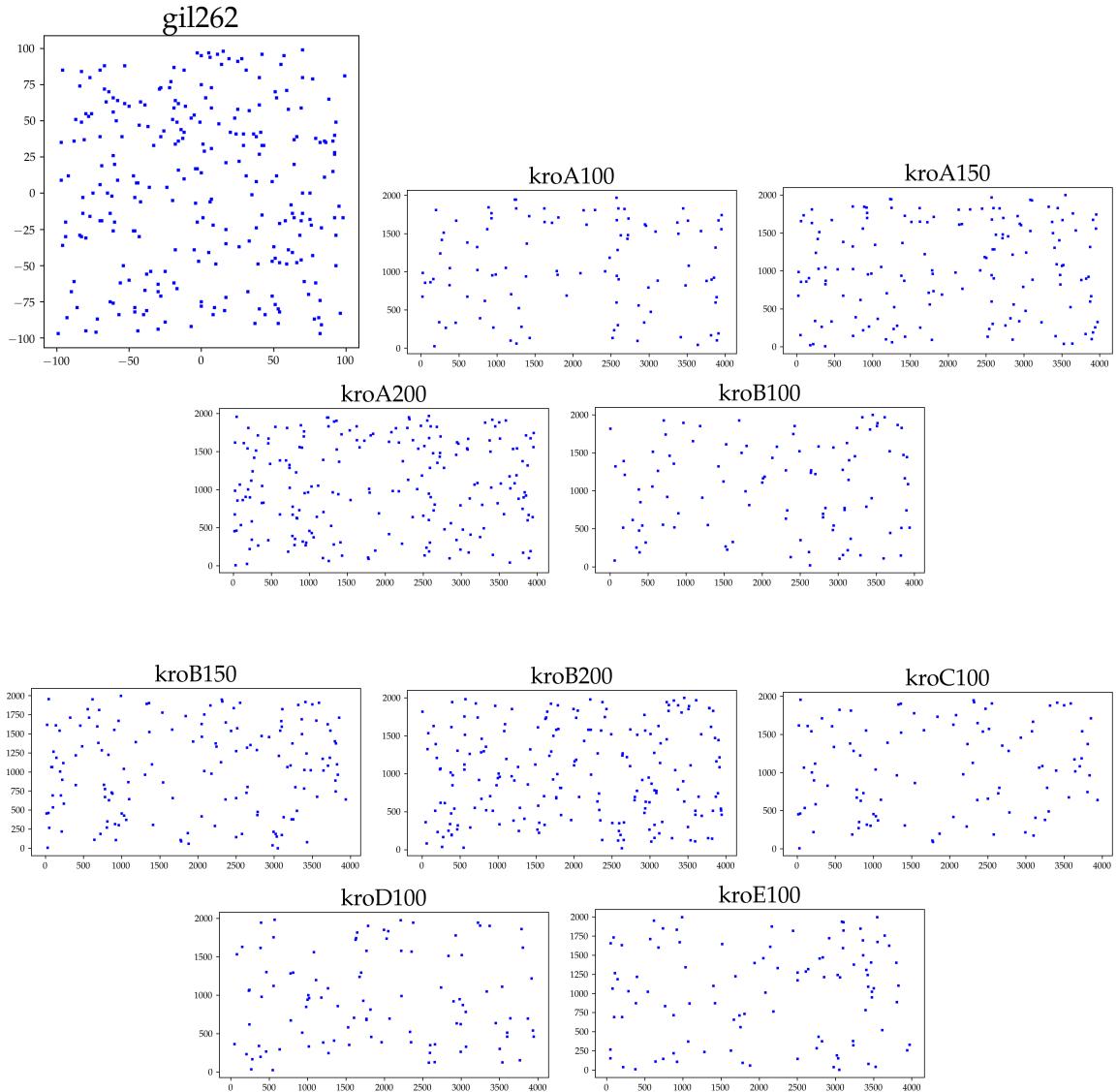
B.1 Pictures of TSPLIB Euclidean Instances

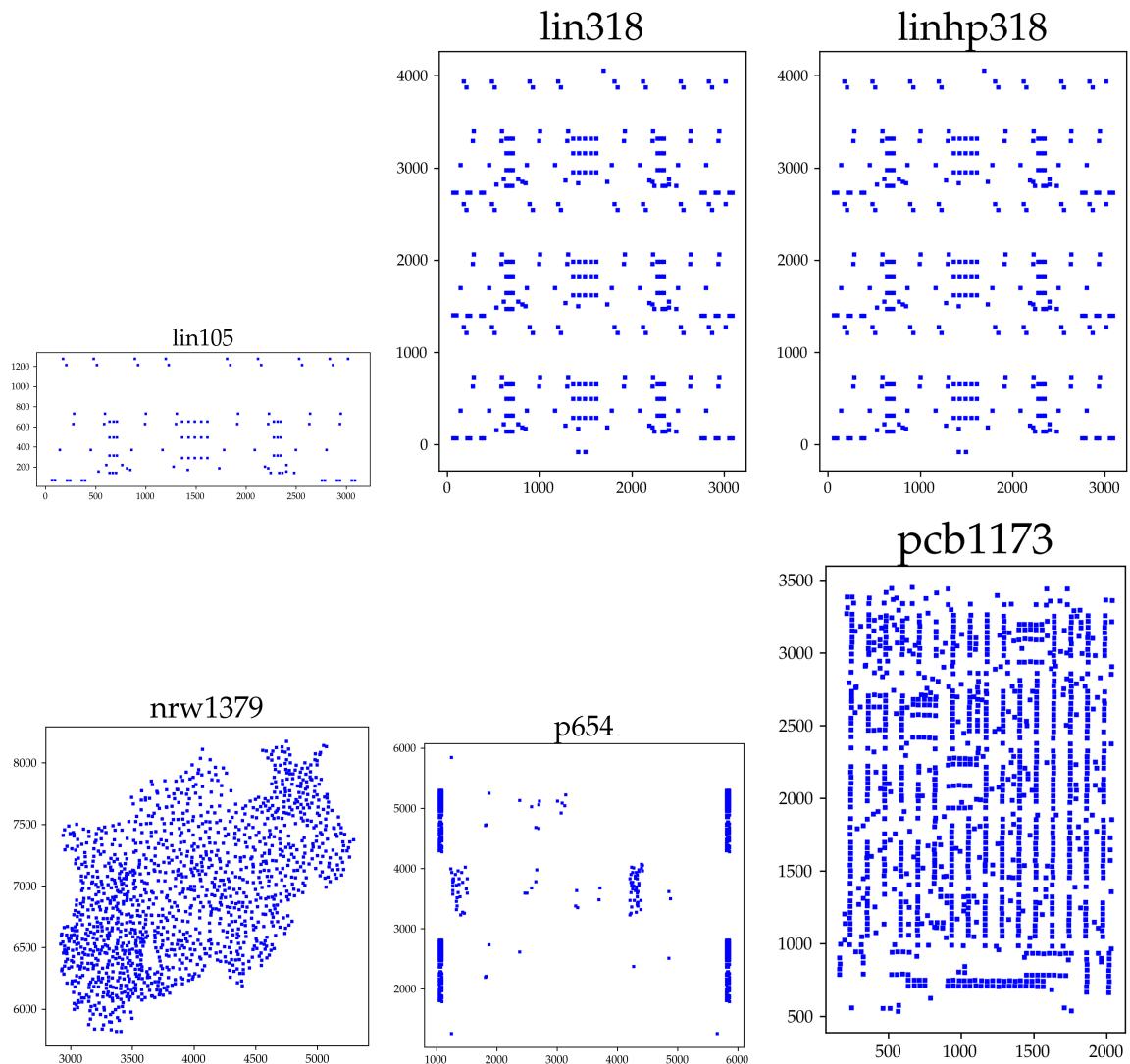
We include here the pictures of all the TSPLIB Euclidean instances.

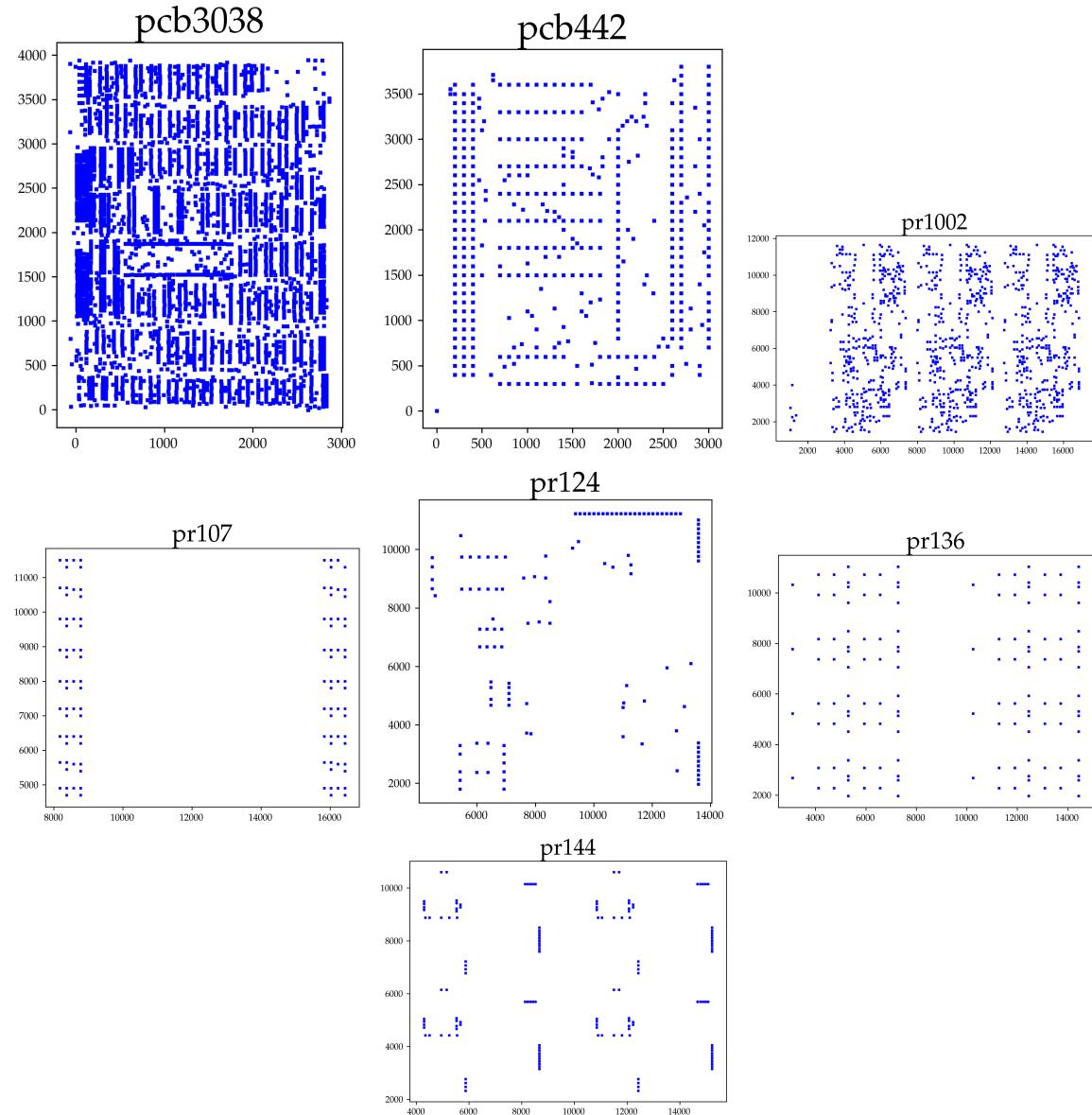


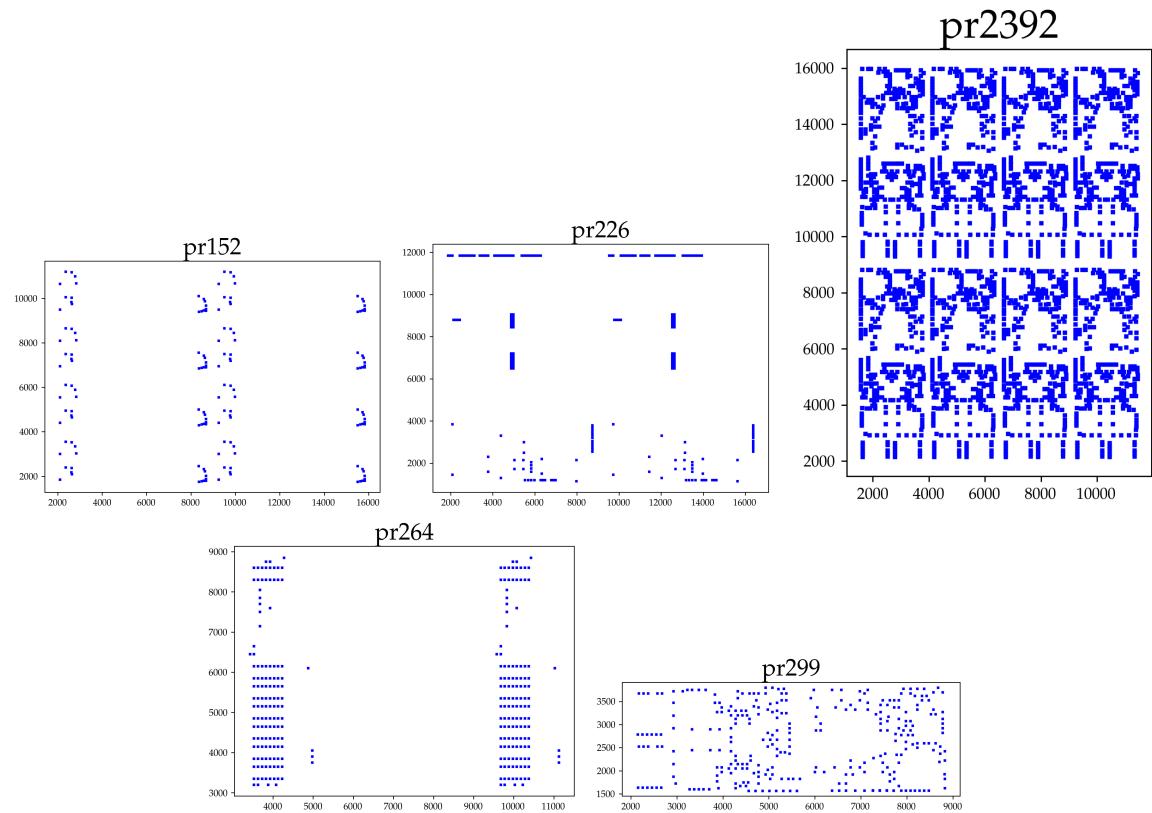


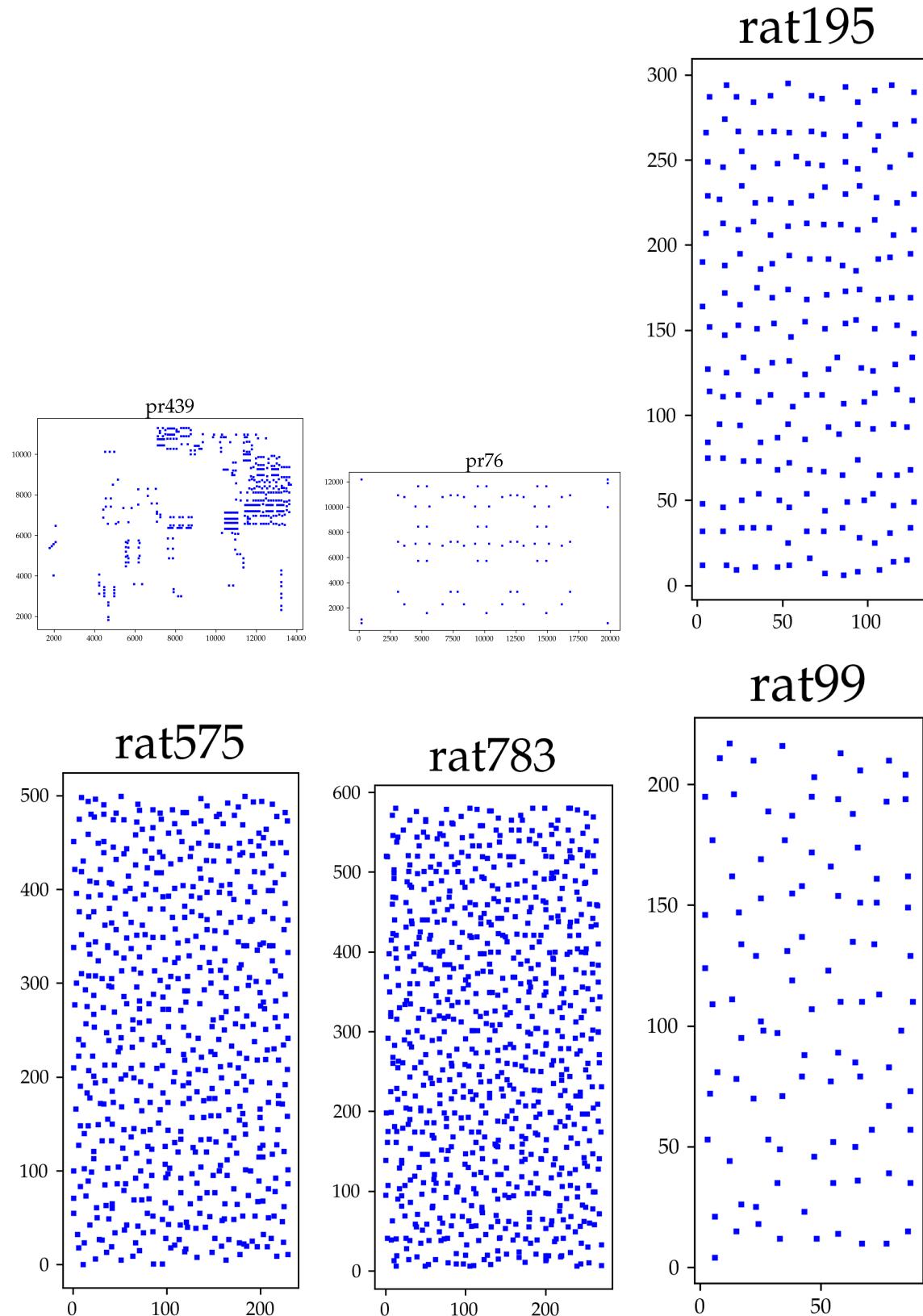


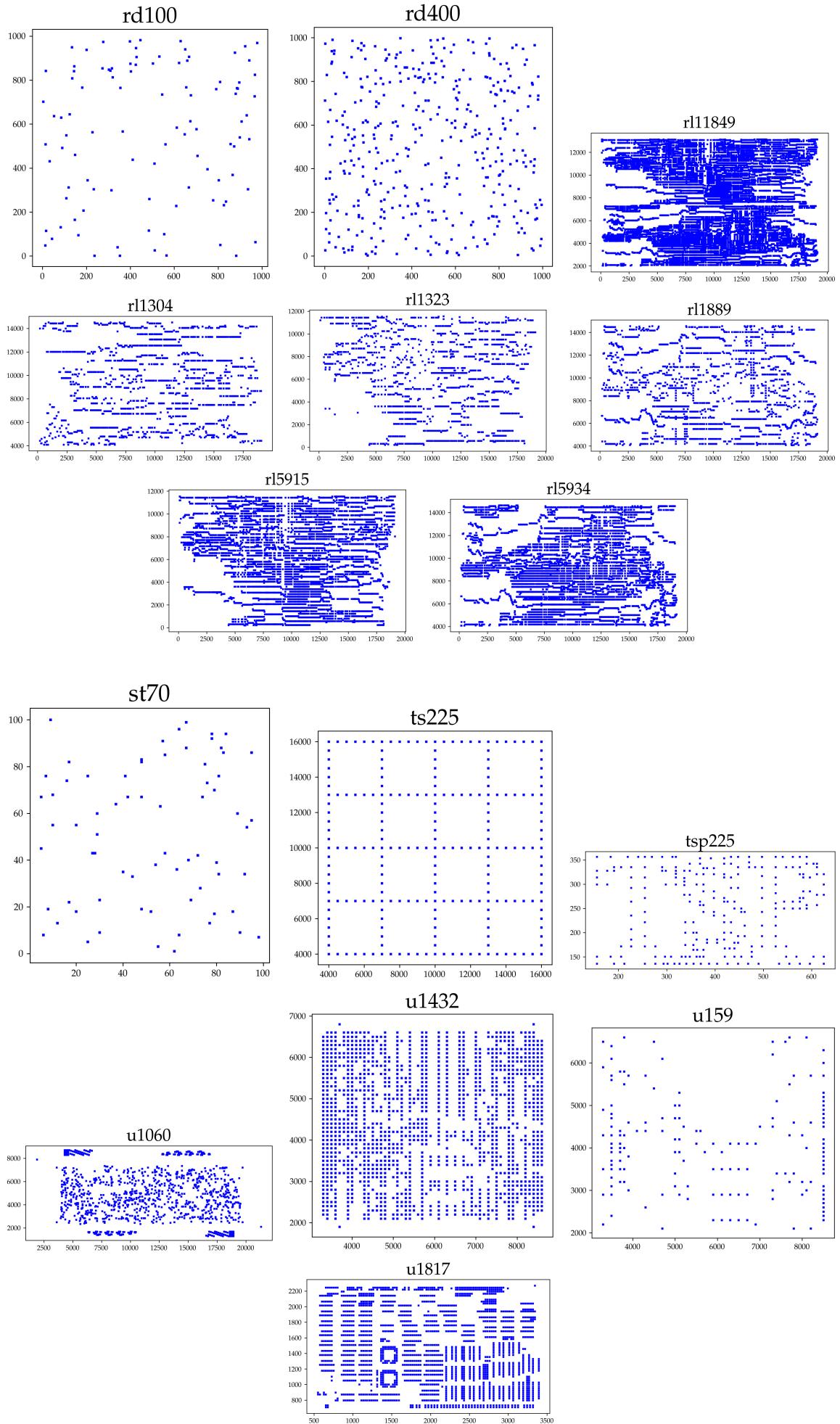


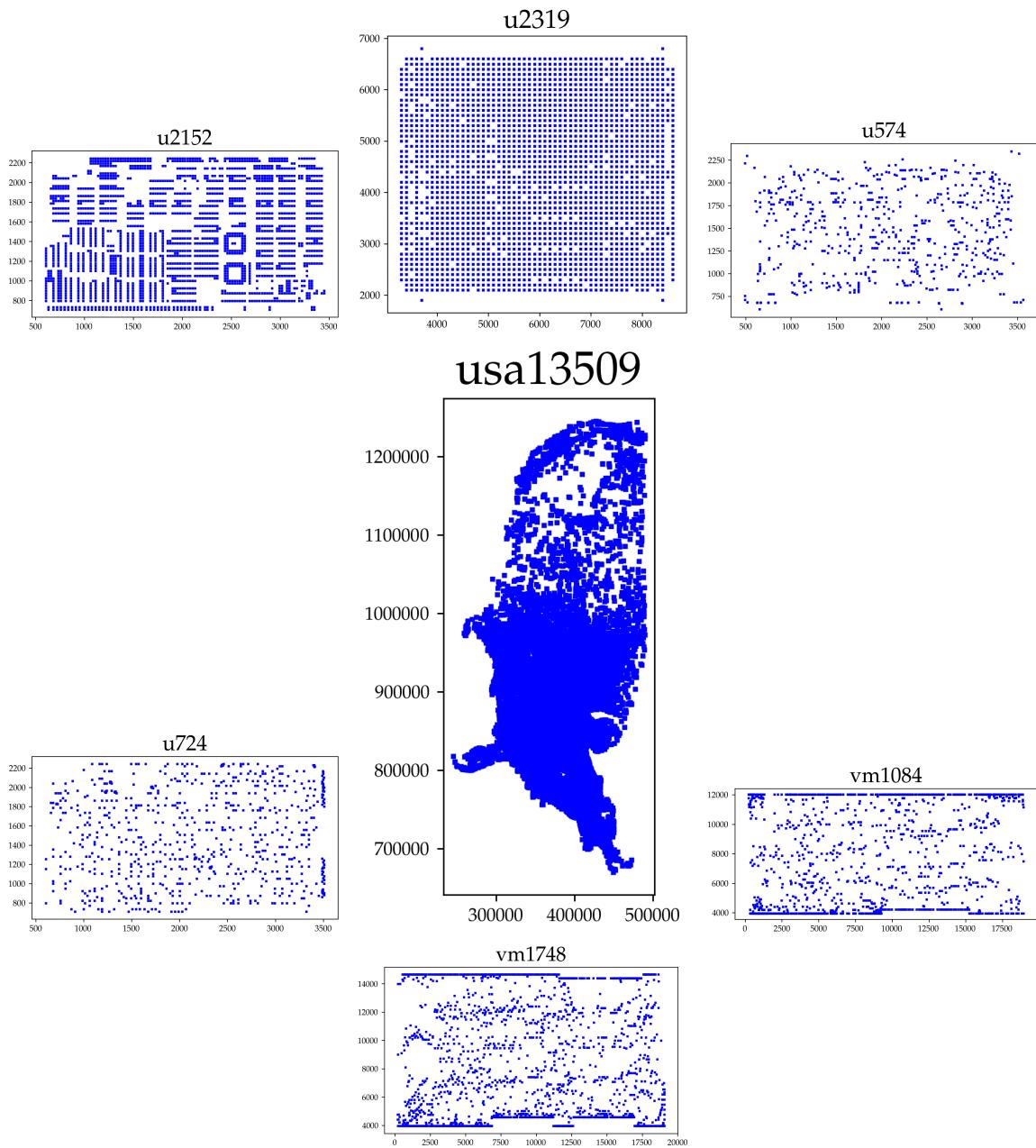












C Machine Details

The code is being developed on a laptop, with the following specs:

Operating System	Linux Mint 18.3 Cinnamon 64-bit
Cinnamon Version	3.6.6
Linux Kernel	4.10.0-38-generic
Processor	Intel® Core™ i5-4300U CPU @ 1.90 GHz x 2
Memory	7.5 GiB
Hard Drives	109.0 GB
Graphics Card	Intel Corporation Haswell-ULT Integrated Graphics Controller

The Anaconda distribution on the machine has the following configuration: (output of ‘`conda info`’)

active environment	None
user config file	/home/xxxx/.condarc
populated config files	
conda version	4.7.10
conda-build version	3.18.8
python version	3.7.3.final.0
virtual packages	
base environment	/home/xxxx/anaconda3 (read only)
channel URLs	https://repo.anaconda.com/pkgs/main/linux-64 https://repo.anaconda.com/pkgs/main/noarch https://repo.anaconda.com/pkgs/r/linux-64 https://repo.anaconda.com/pkgs/r/noarch
package cache	/home/xxxx/anaconda3/pkgs
envs directories	/home/xxxx/.conda/pkgs /home/xxxx/.conda/envs /home/xxxx/anaconda3/envs
platform	linux-64
user-agent	conda/4.7.10 requests/2.22.0 CPython/3.7.3 Linux/4.10.0-38-generic linuxmint/18.3 glibc/2.23
UID:GID	1000:1000
netrc file	None
offline mode	False