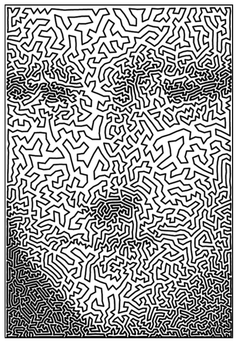


Does the TSP intersect the NNG?

Gaurish Telang
gaurish108@gmail.com

October 21, 2020
12:22 Noon

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.

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

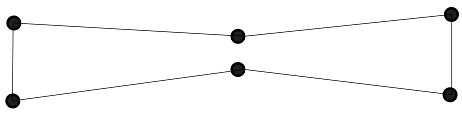


Figure 1: Example where the segment joining the closest pair of points does not lie along the TSP tour

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 A small counter-example to Michael Shamos' conjecutre from his unpublished notes — that the TSP is a *subgraph* of the Delaunay — is given near the bottom of that link.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

[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 Concorde library (to the author's knowledge) computes only optimal cycle solutions and *not* optimal path solutions!
[3]Notice that Figure 1 is not a counterexample!
[4]The author believes this will be a userful reference for future work

fraction of times, or indeed even once. [5]

See also this blogpost on the topic, which talks about using the Delaunay Triangulations for generating heuristically good (no bounds are given) TSP tours. 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

- ❖ $TSP \cap (k\text{-})NNG \stackrel{?}{=} \varnothing$, for $k = 1, 2, \ldots$
- ❖ $TSP \cap$ Delaunay Graph $\stackrel{?}{=} \varnothing$
- ❖ 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* 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 C 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

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

[6] 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 [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??

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

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

[9] This 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

comes with most of the batteries included. See Appendix B for instructions on how to install and run the code.

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* 5⟩
⟨*Generic utility classes and functions* 10a⟩
⟨*Functions for plotting and interacting* 11⟩
⟨*Functions for generating various graphs* 18⟩
⟨*Functions dealing with intersecting two geometric graphs* 29c⟩
⟨*Testing hypotheses* 30c⟩

4b ⟨*Headers* 4b⟩≡ (4a) 13b▷

```
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 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
```

2 Data Generation

5 ⟨*Data Generation* 5⟩≡ (4a)

  ⟨*TSPLIB data* 7⟩
  ⟨*Synthetic data* 9⟩

2.1 TSPLIB data-sets

Figure 2 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]

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 2: 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 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 script [11] 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 retrict 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 3 for some examples.

[10] Prof. 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 an list and the second one reads in a TSPLIB instance (identified by its file-name e.g. `'berlin52.yml'`) in the `euclidean_instances_yaml` directory and returns a list of 2D points for that instance.

7 ⟨*TSPLIB data* 7⟩≡ (5)

```
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
```

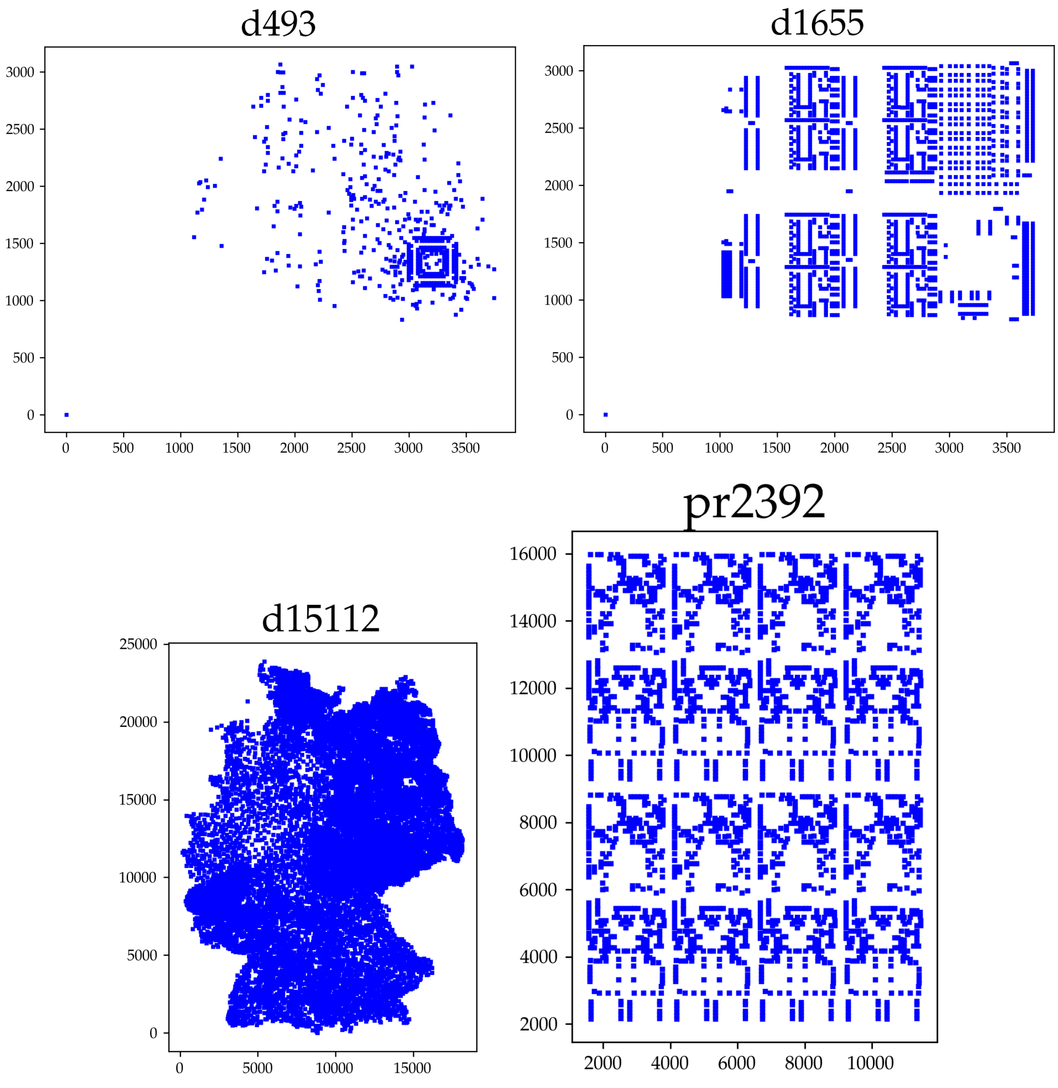


Figure 3: Instances of four TSPLIB data sets for the Symmetric TSP with 2D Euclidean Metric

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.

9 ⟨*Synthetic data* 9⟩≡ (5)

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

def non_uniform_points(numpts):

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

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

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

        points.extend(zip(loc_pts_x, loc_pts_y))

    num_remaining_pts = numpts - cluster_size * numcenters
    remaining_pts = sp.rand(num_remaining_pts, 2).tolist()
```

[12] A somewhat similar method was used in Jon Bentley’s experimental TSP paper

```
    points.extend(remaining_pts)
    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.

10a ⟨*Generic utility classes and functions* 10a⟩≡ (4a) 10b ▷

```
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.

10b ⟨*Generic utility classes and functions* 10a⟩+≡ (4a) ◁ 10a

```
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.

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

```
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.

12 ⟨*Functions for plotting and interacting* 11⟩+≡ (4a) ◁11 13a▷

```
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.

13a ⟨*Functions for plotting and interacting* 11⟩+≡ (4a) ◁12 17b▷

```
def wrapperkeyPressHandler(fig,ax, run):
       def _keyPressHandler(event):
               if event.key in ['n', 'N', 'u', 'U']:
                     ⟨Enter type of point set to generate 16a⟩
               elif event.key in ['t' or 'T']:
                     ⟨Compute TSP and find common edges with various spanning graphs 14⟩
               elif event.key in ['i', 'I']:
                     ⟨Compute spanning graph 15⟩
               elif event.key in ['x', 'X']:
                     ⟨Clear all line segments from the canvas 16b⟩
               elif event.key in ['c', 'C']:
                     ⟨Clear all states and the canvas 17a⟩
       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.

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.

13b ⟨*Headers* 4b⟩+≡ (4a) ◁4b

```
from prettytable import PrettyTable
```

14 ⟨*Compute TSP and find common edges with various spanning graphs* 14⟩≡ (13a)

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

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(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.

15 ⟨*Compute spanning graph* 15⟩≡ (13a)

```
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"            +\
                     "(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 == '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)

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("---------------------------------------")
print("Number of edges in " + algo_str + " graph (TOTAL)                           :", len(geom
```

```
print("Number of edges in " + algo_str + " graph which are also in Concorde TSP   :", len(comm
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.

16a ⟨*Enter type of point set to generate* 16a⟩≡ (13a)

```
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()
```

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‘.

16b ⟨*Clear all line segments from the canvas* 16b⟩≡ (13a)

```
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‘.

17a ⟨*Clear all states and the canvas* 17a⟩≡ (13a)

```
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.

17b ⟨*Functions for plotting and interacting* 11⟩+≡ (4a) ◁13a 28a▷

```
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] [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 grap have the same numbering across all graphs: namely, the order in which the points occur in the `points` array argument.

6.1 k-NNG

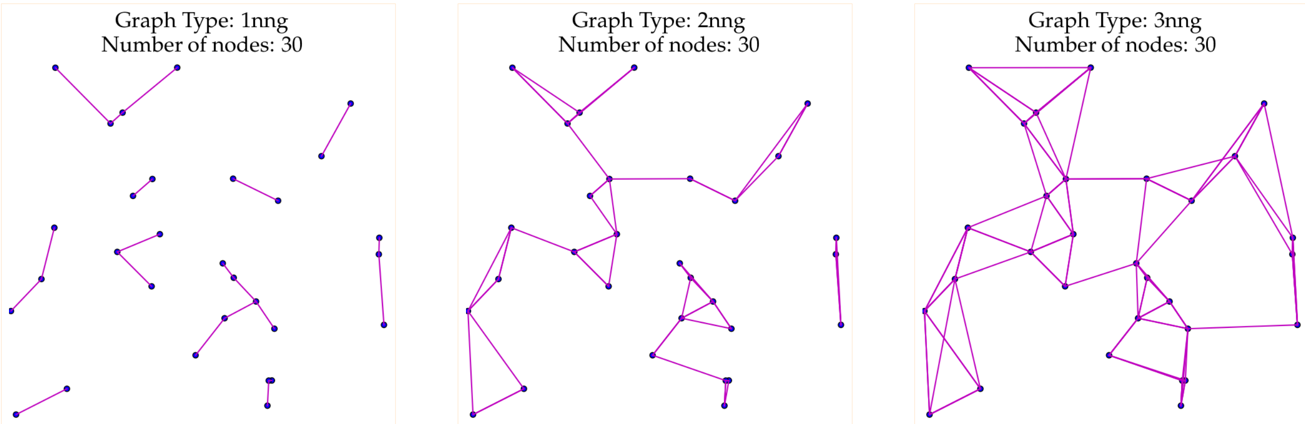


Figure 4: Generating the k-NNG graphs with Scikit-Learn for various value of k on the same set of 30 randomly generated points. Note that we are considering these graphs as undirected.

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 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.

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

```
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 = []
```

[13] already 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

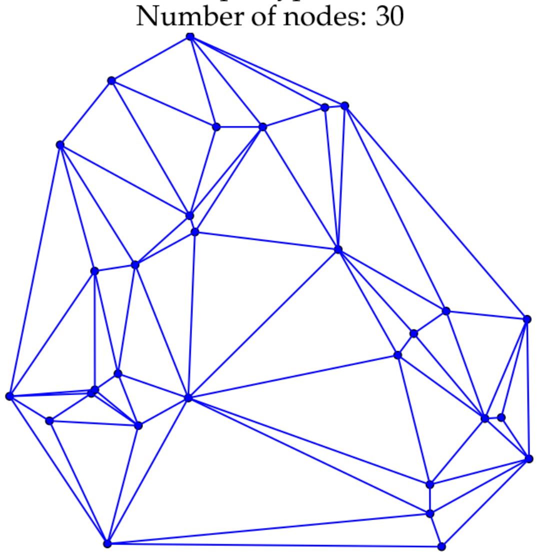


Figure 5: Example of a Delaunay Triangulation computed by SciPy on 30 randomly generated points

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

20 ⟨*Functions for generating various graphs* 18⟩+≡ (4a) ◁18 21▷

```
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

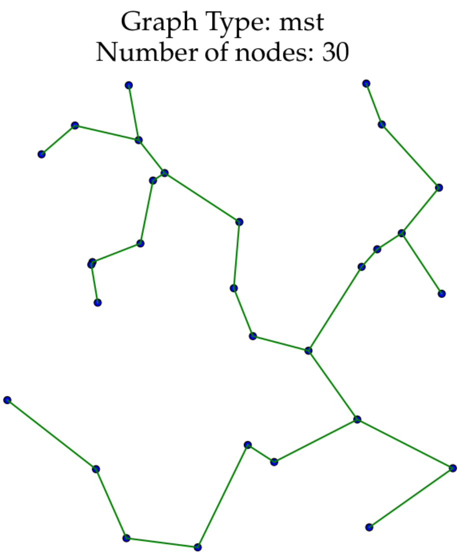


Figure 6: Example of a Minimum Spanning Tree computed by NetworkX on 30 randomly generated points

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.

21 ⟨*Functions for generating various graphs* 18⟩+≡ (4a) ◁20 22▷

```
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

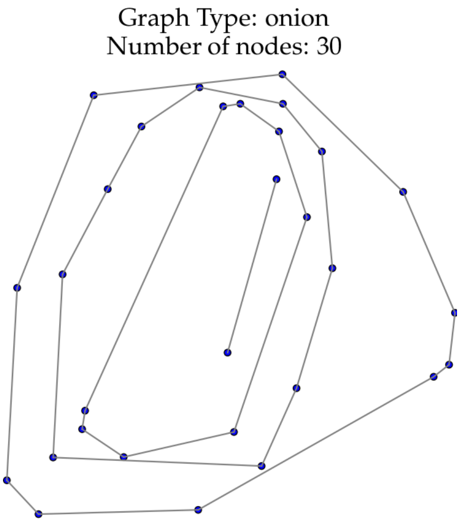


Figure 7: Example of the Onion graph computed with computed with QHull (through SciPy) on 30 randomly generated points

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.

22 ⟨*Functions for generating various graphs* 18⟩+≡ (4a) ◁21 25▷

```
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 23c⟩
     while len(points_tmp) >= 3:
            ⟨Generate convex hull of points remaining in points_tmp 23a⟩
            ⟨Update onion_graph 23b⟩
            ⟨Remove points reported in the convex hull from points_tmp 23d⟩

     if len(points_tmp) == 2:
           ⟨Join two remaining points by an edge in onion_graph 24a⟩
     elif len(points_tmp) == 1:
           ⟨Add the remaining points as a node in onion_graph 24b⟩
```

```
    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.

23a ⟨*Generate convex hull of points remaining in* `points_tmp` 23a⟩≡ (22)

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

23b ⟨*Update* `onion_graph` 23b⟩≡ (22)

```
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. $[1, 2, 3] \to [(1, 2), (2, 3), (3, 1)]$. Convenient to have this defined separately.

23c ⟨ *Definition of* `circular_edge_zip` 23c⟩≡ (22)

```
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
```

23d ⟨*Remove points reported in the convex hull from* `points_tmp` 23d⟩≡ (22)

```
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.

24a ⟨*Join two remaining points by an edge in* `onion_graph` 24a⟩≡ (22)

```
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.

24b ⟨*Add the remaining points as a node in* `onion_graph` 24b⟩≡ (22)

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

6.5 Traveling Saleman Tour (Cycle)

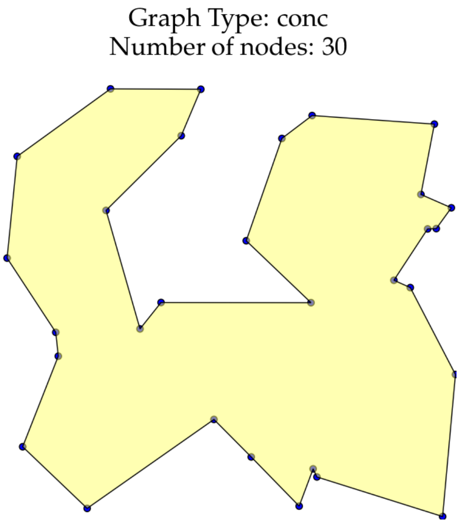


Figure 8: Example of an optimal TSP tour computed with Concorde on 30 randomly generated points

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

25 ⟨*Functions for generating various graphs* 18⟩+≡ (4a) ◁22 27▷

```
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 26a⟩
      ⟨Compute weight of each edge and total edge weight 26b⟩
      ⟨Set graph attributes 26c⟩
      return tsp_graph
```

26a ⟨*Generate TSP cycle and convert into NetworkX graph* 26a⟩≡ (25)

```
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  )
```

26b ⟨*Compute weight of each edge and total edge weight* 26b⟩≡ (25)

```
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
```

26c ⟨*Set graph attributes* 26c⟩≡ (25)

```
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 explcitly informs us that we must pass a longer list.

27 ⟨*Functions for generating various graphs* 18⟩+≡ (4a) ◁25

```
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
```

6.6 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.

28a ⟨*Functions for plotting and interacting* 11⟩+≡ (4a) ◁17b

```
def render_graph(G,fig,ax):
     if G is None:
            return
     ⟨Set up edge colors depending on graph type 28b⟩
     if G.graph['type'] not in ['conc', 'pytsp']:
          ⟨Iterate through graph edges and draw as segments 29a⟩
     else:
          ⟨Draw tour as polygon patch 29b⟩

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

28b ⟨*Set up edge colors depending on graph type* 28b⟩≡ (28a)

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

29a ⟨*Iterate through graph edges and draw as segments* 29a⟩≡ (28a)

```
#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.

29b ⟨*Draw tour as polygon patch* 29b⟩≡ (28a)

```
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

polygon = Polygon(node_coods, closed=True, \
                  facecolor=(255/255, 255/255, 102/255,0.5), \
                  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.

29c ⟨*Functions dealing with intersecting two geometric graphs* 29c⟩≡ (4a) 30a ▷

```
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.

30a ⟨*Functions dealing with intersecting two geometric graphs* 29c⟩+≡ (4a) ◁29c 30b▷

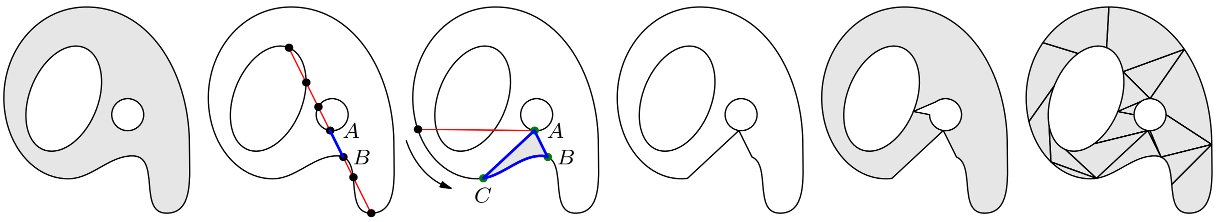
```
def list_common_edges(g1, g2):
     common_edges = []
     for e1 in g1.edges:
          for e2 in g2.edges:
             if  edge_equal_p(e1,e2):
                  common_edges.append(e1)
     return common_edges
```

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

30b ⟨*Functions dealing with intersecting two geometric graphs* 29c⟩+≡ (4a) ◁30a

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

9 Hypothesis testing!



(a) (b) (c) (d) (e) (f)

30c ⟨*Testing hypotheses* 30c⟩≡ (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.

[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/37eObAq.

[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.

[Dil96] Michael B Dillencourt. "Finding Hamiltonian cycles in Delaunay triangulations is NP-complete". In: *Discrete Applied Mathematics* 64.3 (1996), pp. 207–217.

[Geo09] Agelos Georgakopoulos. "A short proof of Fleischner's theorem". In: *Discrete Mathematics* 309.23-24 (2009), pp. 6632–6634.

[LK73] Shen Lin and Brian W Kernighan. "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations research* 21.2 (1973), pp. 498–516. url: https://bit.ly/31pdk7s.

[Pap92] Christos H Papadimitriou. "The complexity of the Lin–Kernighan heuristic for the traveling salesman problem". In: *SIAM Journal on Computing* 21.3 (1992), pp. 450–465.

Appendices

A 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 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.tsp`
- ❖ `berlin52.tsp`
- ❖ `bier127.tsp`
- ❖ `brd14051.tsp`
- ❖ `ch130.tsp`
- ❖ `ch150.tsp`
- ❖ `d1291.tsp`
- ❖ `d15112.tsp`
- ❖ `d1655.tsp`
- ❖ `d18512.tsp`
- ❖ `d198.tsp`
- ❖ `d2103.tsp`
- ❖ `d493.tsp`
- ❖ `d657.tsp`
- ❖ `eil101.tsp`
- ❖ `eil51.tsp`
- ❖ `eil76.tsp`
- ❖ `fl1400.tsp`
- ❖ `fl1577.tsp`
- ❖ `fl3795.tsp`
- ❖ `fl417.tsp`
- ❖ `fnl4461.tsp`
- ❖ `gil262.tsp`
- ❖ `kroA100.tsp`
- ❖ `kroA150.tsp`
- ❖ `kroA200.tsp`
- ❖ `kroB100.tsp`
- ❖ `kroB150.tsp`
- ❖ `kroB200.tsp`
- ❖ `kroC100.tsp`
- ❖ `kroD100.tsp`
- ❖ `kroE100.tsp`
- ❖ `lin105.tsp`
- ❖ `lin318.tsp`
- ❖ `linhp318.tsp`
- ❖ `nrw1379.tsp`
- ❖ `p654.tsp`
- ❖ `pcb1173.tsp`
- ❖ `pcb3038.tsp`
- ❖ `pcb442.tsp`
- ❖ `pr1002.tsp`
- ❖ `pr107.tsp`
- ❖ `pr124.tsp`
- ❖ `pr136.tsp`
- ❖ `pr144.tsp`
- ❖ `pr152.tsp`
- ❖ `pr226.tsp`
- ❖ `pr2392.tsp`
- ❖ `pr264.tsp`
- ❖ `pr299.tsp`
- ❖ `pr439.tsp`
- ❖ `pr76.tsp`
- ❖ `rat195.tsp`
- ❖ `rat575.tsp`
- ❖ `rat783.tsp`
- ❖ `rat99.tsp`
- ❖ `rd100.tsp`
- ❖ `rd400.tsp`
- ❖ `rl11849.tsp`
- ❖ `rl1304.tsp`
- ❖ `rl1323.tsp`
- ❖ `rl1889.tsp`
- ❖ `rl5915.tsp`
- ❖ `rl5934.tsp`
- ❖ `st70.tsp`
- ❖ `ts225.tsp`
- ❖ `tsp225.tsp`
- ❖ `u1060.tsp`
- ❖ `u1432.tsp`
- ❖ `u159.tsp`
- ❖ `u1817.tsp`
- ❖ `u2152.tsp`
- ❖ `u2319.tsp`
- ❖ `u574.tsp`
- ❖ `u724.tsp`
- ❖ `usa13509.tsp`
- ❖ `vm1084.tsp`
- ❖ `vm1748.tsp`

B 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
```
[15]

```
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:

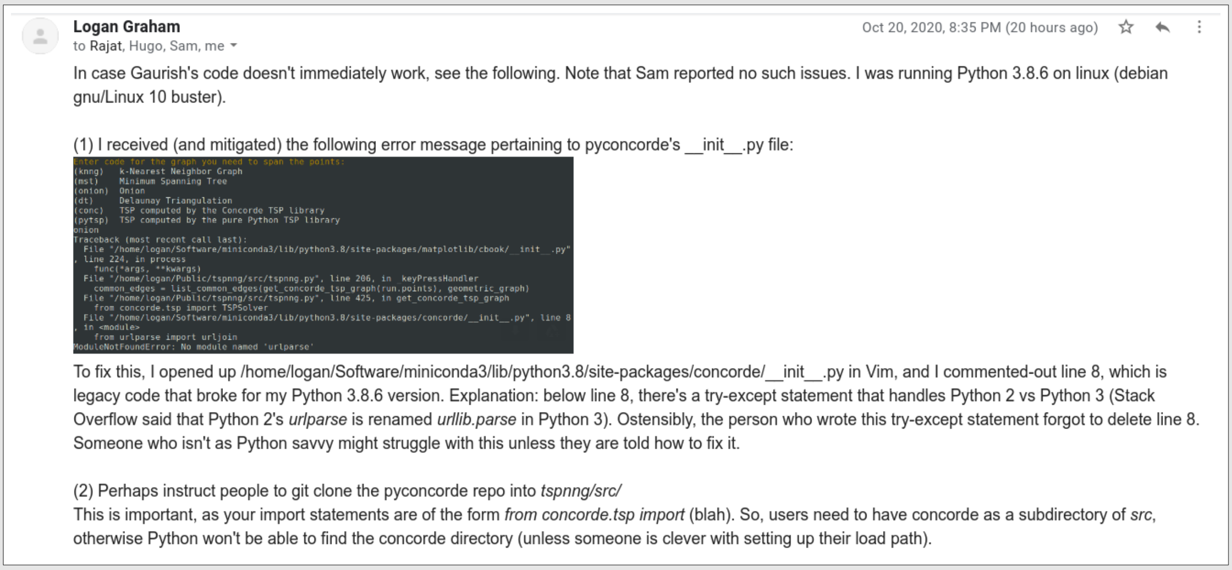
- `python src/main.py -interactive`
- `python src/main.py -file <points.yaml>` TODO!

[14] and that it is Python 3.7 or above
[15] If you don't have superuser access during installation, add the flag `--user` at the end
[16] On Windows replace, the forward slash '/' by '\'

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 9 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.



Logan Graham Oct 20, 2020, 8:35 PM (20 hours ago)
to Rajat, Hugo, Sam, me

In case Gaurish's code doesn't immediately work, see the following. Note that Sam reported no such issues. I was running Python 3.8.6 on linux (debian gnu/Linux 10 buster).

(1) I received (and mitigated) the following error message pertaining to pyconcorde's __init__.py file:

```
Enter code for the graph you need to span the points:
(knng)   k-Nearest Neighbor Graph
(mst)    Minimum Spanning Tree
(onion)  Onion
(dt)     Delaunay Triangulation
(conc)   TSP computed by the Concorde TSP library
(pytsp)  TSP computed by the pure Python TSP library
onion
Traceback (most recent call last):
  File "/home/logan/Software/miniconda3/lib/python3.8/site-packages/matplotlib/cbook/__init__.py"
, line 224, in process
    func(*args, **kwargs)
  File "/home/logan/Public/tspnng/src/tspnng.py", line 206, in _keyPressHandler
    common_edges = list_common_edges(get_concorde_tsp_graph(run.points), geometric_graph)
  File "/home/logan/Public/tspnng/src/tspnng.py", line 425, in get_concorde_tsp_graph
    from concorde.tsp import TSPSolver
  File "/home/logan/Software/miniconda3/lib/python3.8/site-packages/concorde/__init__.py", line 8
, in <module>
    from urlparse import urljoin
ModuleNotFoundError: No module named 'urlparse'
```

To fix this, I opened up /home/logan/Software/miniconda3/lib/python3.8/site-packages/concorde/__init__.py in Vim, and I commented-out line 8, which is legacy code that broke for my Python 3.8.6 version. Explanation: below line 8, there's a try-except statement that handles Python 2 vs Python 3 (Stack Overflow said that Python 2's *urlparse* is renamed *urllib.parse* in Python 3). Ostensibly, the person who wrote this try-except statement forgot to delete line 8. Someone who isn't as Python savvy might struggle with this unless they are told how to fix it.

(2) Perhaps instruct people to git clone the pyconcorde repo into *tspnng/src/*
This is important, as your import statements are of the form *from concorde.tsp import* (blah). So, users need to have concorde as a subdirectory of *src*, otherwise Python won't be able to find the concorde directory (unless someone is clever with setting up their load path).

Figure 9: Screenshot of Logan's email with an installation fix

B.1 Interactive Mode

In this mode, one can mouse-in points onto a canvas (with double-clicks), 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. Ditto for non-uniformly distributed points: only that you must press `'n'` .

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).

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 compoute 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. ¯_(ツ)_/¯. If you have any trouble — or detect a bug! — we can hash things out on Slack, Github or email.

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

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

C Laundry-list of Questions/Variants/Conjectures

HAMILTONICITY STRUCTURE

We know that the Delaunay Triangulation of a set of points need not be Hamiltonian. In fact *detecting* Hamiltonicity of a Delaunay Triangulation is famously NP-complete [Dil96]

Two useful facts before we proceed:

Folklore

The cube of any connected graph is Hamiltonian. [19].

Fleischner's theorem [Geo09]

The square of any 2-vertex connected graph is Hamiltonian. [20]

And so the following questions are natural:

- Based on the experiments results shown in this report, can we claim $TSP \subseteq DT^k$ or $TSP \subseteq MST^k$ in $\mathbb{R}^2$ for a *small* constant k? I'd wager $k = 2, 3$ or, at worst, some very slowly growing function of n [21]
- For $G = MST^3$, how good is the any (or the shortest??) Hamilton cycle through G in approximating the TSP? [22] Surely, this must be known, right? For the arrangement of n points at the roots of unity suggests that the approximation could be as bad as 3.
- What is the likelihood of MST^2 of n points $\in \mathbb{R}^2$ in general position being Hamltonian? Any characterization of such point-sets?
- Given a set of points in $\mathbb{R}^2$, does *ANY* (weakly/strongly) simple polygon and *ANY* triangulation on those points have an edge in common? This is surely not true, right?!

  UPDATE: Yeah this isn't true. Figure 10 is a nice counter-example due to Hugo.

[19] It is sufficient to prove this fact for any tree, and then use it on the spanning tree of the given graph

[20] This last theorem certainly applies to Delaunay triangulations of general point-sets. By the square of a delaunay triangulation, I mean to say: throw away the edge weights and consider the square of the underlying unweighted graph. Once the new edges are added, consider them weighted with the natural euclidean distance between their endpoints

[21] $\log(n)$ maybe?

[22] This is a cute vertex analog of the standard edge-doubling based 2-OPT heuristic, but is detecting such a cycle for a small MST power polytime? I'd bet yes. Probably the FPT experts have something to say on this topic.

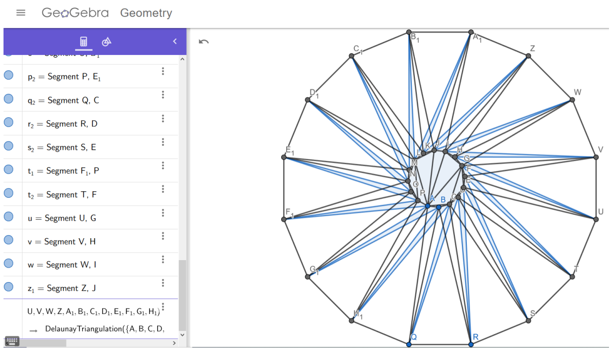


Figure 10: A simple polygon with edges in the complement of the graph of a point set's triangulation

Monkey wrench à la Joe's Universal Guarding Problem: given a set of points P in general position, might there exist a "universal" triangulation of P that shares some edge with *any* simple polygon on P? Maybe Hugo's counter-example of Figure 10 still holds? Qian's beautiful UGP counterexample from her paper may also be of some relevance. If this is not true for all point-sets, is detecting this property NP-hard? Check if some of the gadgets from Demaine et al's hardness proof of the pulley problem are applicable here: they basically showed that detecting whether a set of mutually disjoint disks(not necessarily of the same radii) admit a "pulleygon" — excuse the pun! — is NP-hard .

TSP $\rightsquigarrow$ DELAUNAY

❖ (Worst idea ever??) Suppose by some magic black box (Concorde! ☺) one has obtained the TSP tour through n points. Can we then compute the Delaunay Triangulation (or MST. Recall $MST \subseteq DT$) of the point-set in $O(n)$ time? It certainly seems so going by the high number of edges in the TSP common to the Delaunay and MST as suggested by the tests in this writeup. Surely the $TSP \to MST$ question in $O(n)$ time has been studied? Maybe the Okabe book on Voronoi etc. has something on this?

It's, of course, utter baloney attempting to compute good triangulations or MSTs like this in practice but why should that *that* stop a computer scientist speculating from the comfort of his/her armchair?

In some vague sense might be related to the question of *detecting* if a triangulation is Delaunay is linear time (check with joe, if this problem is trivial/open/known. I mean detecting if a list is sorted is trivially doable in $O(n)$ as opposed to sorting which takes $\Omega(n \lg n)$ in comparsion model. Are there lower bounds in other models (streaming, online etc.) for Delaunay tri detection? Check with Joe, Mayank others.

This reminescent of Melkman (i.e. simple poly chain $\to$ convex hull in $O(n)$). Samir Khuller also had work (SODA 1992, iirc) in which he was able to compute a $O(1)$ (1.25, was it?) approximations to low-weight spanning trees of max degree 3 on point-sets in $\mathbb{R}^2$, from a *given* MST in linear time. His paper hinged overall on a very simple but beautiful generalization of the Δ inequality [23] Might that be exploitable? Check!

Joe mentioned last week there is some nice work by Melhorn et al, on the reconstructibility properties of the TSP back from SOCG 1988(1989?) ; Also Amenta (Crust algorithm) and Dey's works on reconstruction. Maybe we can connect these two lines in some hand-wavy way?

- Is the Min Weight (MW), or even better, the Min-Angle Maximizing (MAM) Triangulation of the TSP and its external pockets close to being Delaunay in some sense[24]? Note that both MW and MAM triangulations of simple polygons can be computed by D.P. Should be fun to try this out experimentally. See also Sandor's ALENEX paper from 2015 where he did something about computing "bad" triangulations (iirc he wanted to minimize maximum angle i.e. make triangle skinny), Maybe we could compare against the triangulations obtained from his code, just to see how bad the TSP $\rightarrow$ Delaunay heuristic is compared to the "bad end" of the spectrum?

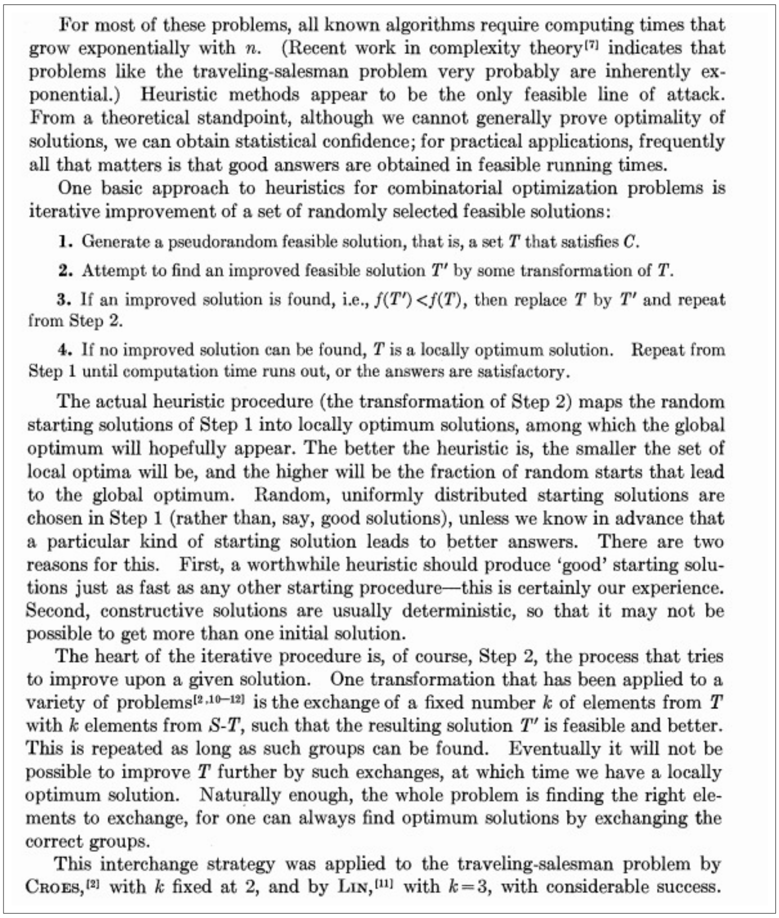
SHOWING [LK73] HAS A CERTAIN FRACTION OF DELAUNAY/NNG EDGES

Here we try to show that the output of the Lin Kernighan heuristic (rather than the optimal TSP) on any set of points has the postulated properties. In many practical cases, the LK heuristic is optimal, so this might be a cut-price (but interesting) substitute of the original question, especially for lower bounds. The LK paper has also documents some nice local properties of the exchanges.

[23] which bounded the perimeter of a triangle in terms of the sums of the distances of a point X to the points of that triangle. When we set X to any of the vertices of the triangle, we recover the standard Δ inequality. I forget the exact coefficients of that cited linear inequality, remember some $3\sqrt{3}$ and a 4 in there.

[24] e.g. how good is the maximum value of the minimum angle in a triangle over all triangles different from OPT

Figure 11 is a screenshot from the opening pages:



For most of these problems, all known algorithms require computing times that grow exponentially with n. (Recent work in complexity theory[7] indicates that problems like the traveling-salesman problem very probably are inherently exponential.) Heuristic methods appear to be the only feasible line of attack. From a theoretical standpoint, although we cannot generally prove optimality of solutions, we can obtain statistical confidence; for practical applications, frequently all that matters is that good answers are obtained in feasible running times.

One basic approach to heuristics for combinatorial optimization problems is iterative improvement of a set of randomly selected feasible solutions:

1. Generate a pseudorandom feasible solution, that is, a set T that satisfies C.

2. Attempt to find an improved feasible solution T' by some transformation of T.

3. If an improved solution is found, i.e., $f(T')<f(T)$, then replace T by T' and repeat from Step 2.

4. If no improved solution can be found, T is a locally optimum solution. Repeat from Step 1 until computation time runs out, or the answers are satisfactory.

The actual heuristic procedure (the transformation of Step 2) maps the random starting solutions of Step 1 into locally optimum solutions, among which the global optimum will hopefully appear. The better the heuristic is, the smaller the set of local optima will be, and the higher will be the fraction of random starts that lead to the global optimum. Random, uniformly distributed starting solutions are chosen in Step 1 (rather than, say, good solutions), unless we know in advance that a particular kind of starting solution leads to better answers. There are two reasons for this. First, a worthwhile heuristic should produce 'good' starting solutions just as fast as any other starting procedure—this is certainly our experience. Second, constructive solutions are usually deterministic, so that it may not be possible to get more than one initial solution.

The heart of the iterative procedure is, of course, Step 2, the process that tries to improve upon a given solution. One transformation that has been applied to a variety of problems[2,10–12] is the exchange of a fixed number k of elements from T with k elements from S-T, such that the resulting solution T' is feasible and better. This is repeated as long as such groups can be found. Eventually it will not be possible to improve T further by such exchanges, at which time we have a locally optimum solution. Naturally enough, the whole problem is finding the right elements to exchange, for one can always find optimum solutions by exchanging the correct groups.

This interchange strategy was applied to the traveling-salesman problem by CROES,[2] with k fixed at 2, and by LIN,[11] with $k=3$, with considerable success.

Figure 11: Screenshot from the Lin Kernighan paper

Page 5 of that paper contains the main sketch of the overall algorithm.

On that note, see [Pap92] where *"It is shown that finding a local optimum solution with respect to the Lin-Kernighan heuristic for the traveling salesman problem is PLS-complete,"* Might have useful nuggets.

D 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	1.10.0-38-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 /home/xxxx/.conda/pkgs
envs directories	/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