The Traveling Salesperson Problem
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/tsp/tsp.pdf

---

A State Capital Road Trip.

"TSP"

- *The Traveling Salesperson Problem (TSP) is an optimization problem;*
- *It seeks the shortest round trip visiting a set of cities;*
- *The problem has an "impossible" complexity of n!;*
- *Versions of this problem need to be solved for applications ranging from transportation (Federal Express package delivery) to circuit printing (minimizing the travel of the drill or print head);*
- *Many ideas have been suggested for approximate solutions.*

To follow along with the three examples I will discuss, you might want to copy the following files from the class website:

city locations	code
five_position.txt	five.py
fifteen_position.txt	fifteen.py
fortyeight_position.txt	fortyeight.py

1 The Traveling Salesperson Problem

The traveling salesperson problem supposes that a salesperson has been assigned to make a sales trip that visits all the cities on a list, to return then to home base. The additional constraint is that, out of all possible such trips, the shortest one should be chosen.

Since we like to think small, it would seem that, in the case of a list of 5 or 6 cities, this is a simple matter of listing all the possible trips, measuring each one, and choosing the shortest. This is known as a "brute force" approach, because it takes a simple-minded procedure that is guaranteed to work, as long as we have enough time. But, as it turns out, we can only solve "baby-sized" problems this way.

If there are n cities (including the home city), then the salesperson has $n-1$ choices for the first visit, $n-2$ for the second, and so on. It thus turns out that there are essentially $(n-1)!$ possible trips to be considered. Any time you see a factorial in a formula, you have to be worried, because factorials explode in size very quickly. Suppose, for instance, that the salesperson plans to visit every state capital in the US. Then we are talking about something like 49! distinct itineraries to consider. You should easily see that this number, whatever it is, has more than 49 digits, and is far beyond the size of anything we can deal with computationally.

But since these kinds of problems arise in practical business and industrial applications, they have to be solved, or at least solve approximately. And so there are a number of very sensible ideas for trying to find decent estimates of a TSP itinerary.

In this discussion, we will look at some sample problems, and how to implement various techniques for estimating a good solution.

2 A tiny problem

We begin by examining a tiny problem, with which we can start to explore the TSP. We will assume we have five cities, named Aspen, Boston, Clayton, Dayton, and Eaton, with (x, y) map coordinates as follows:

```
10  0
 0 70
90 30
60 80
30 50
```

It will be useful to compute a city-to-city distance table:

```
distance = np.zeros ( [ city_num, city_num ], dtype = int )
for i in range ( 0, city_num ):
  for j in range ( 0, city_num ):
    distance[i,j] = int ( np.linalg.norm ( position[i,:] - position[j,:] ) )
```

and if we round the distances to integers, we get

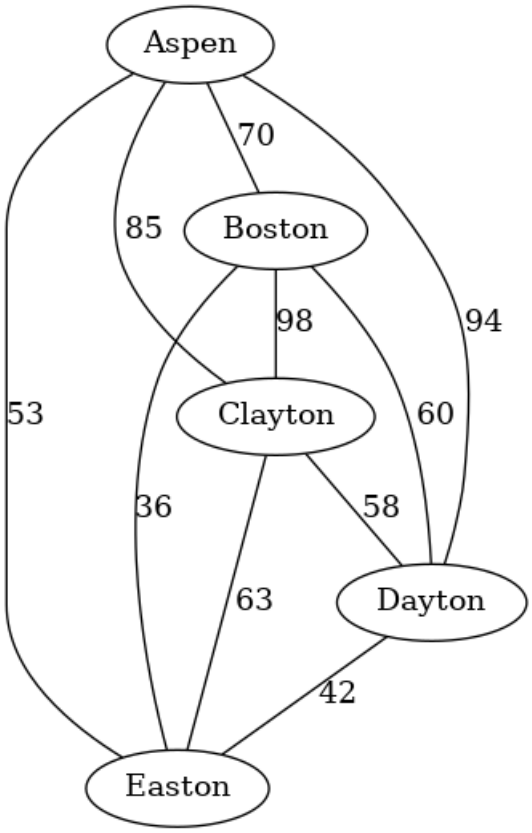
```
 0 70 85 94 53
70  0 98 60 36
85 98  0 58 63
94 60 58  0 42
53 36 63 42  0
```

We can visualize our data using *graphviz*, with the distances (rounded to whole numbers) marked along the connecting edges:

```
from graphviz import Graph
names = [ 'Aspen', 'Boston', 'Clayton', 'Dayton', 'Easton' ]
dot = Graph ( format = 'png' )
for n in names:
  dot.node ( n )
for i in range ( 0, city_num ):
  for j in range ( i + 1, city_num ):
```

```
        dot.edge ( names[i], names[j], str( distance[i,j] ) )
    dot.render ( 'five_distance.dot', view = False )
```

which results in the following picture:



Now we suppose that a traveler wishes to start at Aspen and make a your of the other cities, ending back at Aspen. The traveler is an optimizer, and wishes to make this trip in the most efficient way possible, which in this case simply means that the itinerary should be as short as possible. This is a version of the *traveling salesperson problem*, often abbreviated to `TSP`. While it may seem a simple-minded puzzle, variations of this problem constantly arise in business, transportation, design. Even in the printing of computer circuit boards, it is desired to minimize the travel of the printing head as it moves from spot to spot.

We will start our consideration of the TSP by looking at how to solve our five city problem; then we will realize that as the number of cities grows, the problem becomes impossibly hard to solve exactly.

3 Solution by brute force

Given that there are many possible itineraries for our five city tour, how can we find the shortest one? For many problems, our approach would normally be to find some formula that the best solution satisfies, or some way of quickly sorting through a sequence of better and better solutions. However, for the TSP, there are not many obvious ways to guess something close to the best itinerary.

On the other hand, we know there is a solution, because there are only finitely many different itineraries, and so there must be a shortest length. With five cities, there are actually only $5! = 120$ itineraries to consider;

this is the number of permutations that can be formed from five objects. (In fact, we can actually cut this down to $4!/2 = 12$ if we think about it!)

So a simple-minded approach is simply to generate every possible permutation, measure the length of the corresponding path, and report the permutation corresponding to the shortest length. Because this approach makes no attempt to take advantage of any patterns or structure in the problem, it is known as the *brute force method*.

Presumably, our itinerary can be described as a permutation, so we need a function that can report the length of the corresponding path:

```
def path_length ( distance, perm ):
  mileage = 0.0
  c0 = perm[-1]
  for c1 in perm:
    mileage = mileage + distance[c0,c1] )
    c0 = c1
  return mileage
```

and we need a way to generate all the itineraries.

```
  from itertools import permutations
  perms = permutations ( np.arange ( city_num ) )
  mileage_shortest = np.inf
  for perm in perms:
    mileage = path_length ( distance, perm )
    if ( mileage < mileage_shortest ):
      perm_shortest = perm
      mileage_shortest = mileage
```

The `itertools` library allows us to generate every permutation of a given list, one at a time. So the value of `perm` will start at [0,1,2,3,4], then [0,1,2,4,3] and so on, all the way to [4,3,2,1,0].

Almost immediately, our program returns to report that the best itinerary is has mileage 291, using the cities in the sequence $(0, 1, 4, 3, 2)$, or by name:

```
Aspen => Boston => Easton => Dayton => Clayton => Aspen
```

4 Replacing brute force by greed

We are actually going to want to work with a problem involving 48 cities. We have heard that the problem gets more difficult as the number of cities increases. So we prepare an intermediate challenge involving 15 cities, whose locations are defined in the file *fifteen.txt*. However, when we apply the code that worked on *five.txt*, we are frustrated as we wait a minute, five minutes, 10, 15 minutes with no result. Should this be a surprise?

The 5 city problem required generating $10! = 120$ permutations. For the 15 city problem, there are $15! = 1,307,674,368,000$, a value we can collect by calling `math.factorial(15)`. We might not be afraid to ask for a million permutations; a billion seems like a lot, but here we are asking for more than a trillion. If our original program solved 120 permuations in 1 second, then our new program might be expected to give us an answer in

```
1,307,674,368,000/120/60/60/24/365 = 345 years
```

So we can safely assume that the brute force method is already a completely hopeless approach at size 15. If we want any hope of an answer, we need a new approach. Aside from brute force, there are no algorithms that can guarantee a solution of TSP. Therefore, we are looking instead for *heuristics*, that is, ideas or approaches that seem like a reasonable way to get close to an answer.

The approach we will consider next is called the *greedy TSP algorithm*. Instead of having a big strategy, the greedy algorithm takes a very short-sighted approach:

1. Initialize shortest path length as ∞, that is, `np.inf`;
2. Consider each city as the starting point for a tour;
3. Select the next edge of the tour as the shortest edge from your current location to an unvisited city;
4. Repeat until you return to your starting city.
5. Remember this tour if it is the shortest so far;

For our fifteen city problem, the code finds a greedy estimate in three steps.

```
Greedy tour starting at city 0  costs  70
Greedy tour starting at city 4  costs  65
Greedy tour starting at city 8  costs  58
```

5 How rare are short paths?

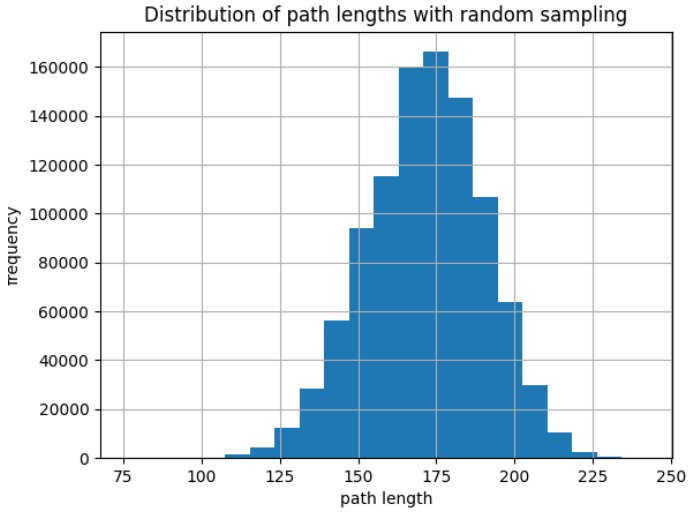
We might think that we can stumble upon a solution by simply randomly sampling a lot of permutations, and taking the best one we encounter. This could be a reasonable approach if short solutions are relatively common. Let's continue with our 15 city problem, generate 1,000,000 permutations, compute the length of each, and make a histogram. This would suggest whether coming close by random selection is going to be a viable option. We already have one piece of evidence: at least one path is of length 58.

Sampling might be an effective idea as long as short paths are not extremely rare. To see this issue, suppose we are trying to find the smallest point (the origin) in a multidimensional sphere of radius 1. In dimension 10, 65% of the points are more than 0.9 from the origin. This rises to 88% in 20 dimensions, and 99% in 50 dimensions. If our TSP problem is like sampling a high dimensional sphere, then random sampling will be a waste of time. Let us try generating sample permutations for our 15 city problem, and see what the data tells us.

samples	shortest path
100	128
1,000	108
10,000	91
100,000	94
1,000,000	76

So, at least for the 15 city case, it looks like if we are very very persistent, we might get close to the path of length 58. However, as the problem size increases, the rate of improvement will drop drastically.

Here is the histogram of all the path lengths found when generating 1,000,000 samples:

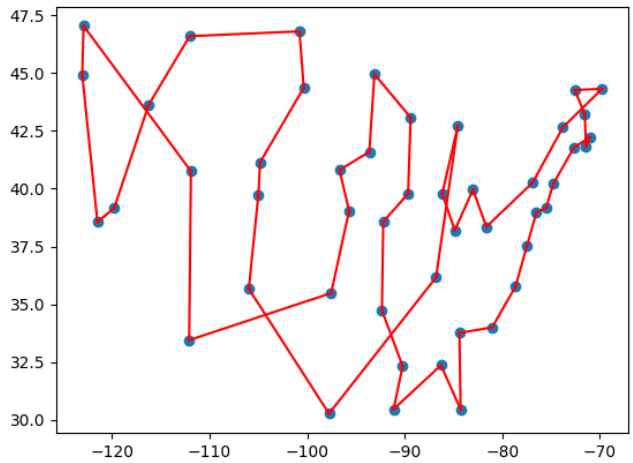


It's clear from the plot that most of the random permutations represent horribly inefficient tours whose average length is often triple the best that we saw. Again, this is for a small problem, and the discrepancy will become much much worse for a longer tour.

6 The forty eight state capital challenge

Now it's time to see if we can get a reasonable solution to what is actually a very small version of the TSP, namely, the challenge to visit all 48 state capitals in the "mainland" US. On the one hand, modern researchers routinely do an excellent job of estimating the solution when tours involve hundreds of thousands of cities; on the other hand, we saw how difficult the 15 city problem was, with more than a trillion permutations to consider. The 48 city problem is simply unimaginably harder than that.

We can certainly get a rough guess for a solution by trying the greedy approach, which simply computes 48 particular permutations and returns the best one. In that case, we get a tour length of 178, but the plot shows that our solution is far from ideal:



Any tour which crosses itself cannot be optimal (at least in realistic Euclidean geometry!).

In our second investigation, we try random sampling.

samples	shortest path
100	744
1,000	633
10,000	626
100,000	588
1,000,000	551

Here, we see that the problem is resisting us. We are quite far away from the greedy solution of 178, and the rate of decrease is not very promising.

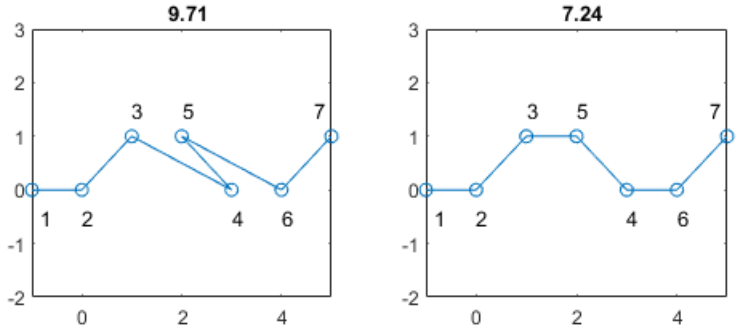
It is time to search for a better approach that tries harder than the greedy algorithm, but more intelligently than random sampling.

7 Jumps and Flips

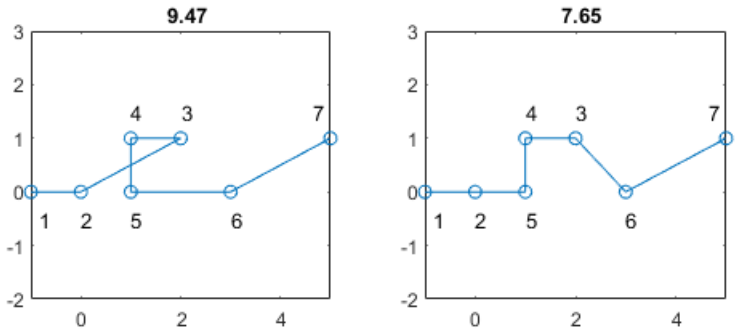
Cleve Moler, the developer of Matlab, discusses a program called `travel` which tries to solve a version of the TSP in his article:
`https://blogs.mathworks.com/cleve/2018/09/17/usa-traveling-salesman-tour/s_tid=srchtitle_tsp_2&doing_wp_cron=1680823886.0916650295257568359375`

Each time it is called, `travel` starts with a random permutation of the cities. Then, for several thousand iterations, it tries to reduce the length of the trip by revising the permutation in each of two simple ways.

One scheme is to move one city to another position in the ordering. This example starts with the order [1:7]. It then moves city 4 to follow city 5, so the order becomes [1 2 3 5 4 6 7]. This reduces the length from 9.71 to 7.24.



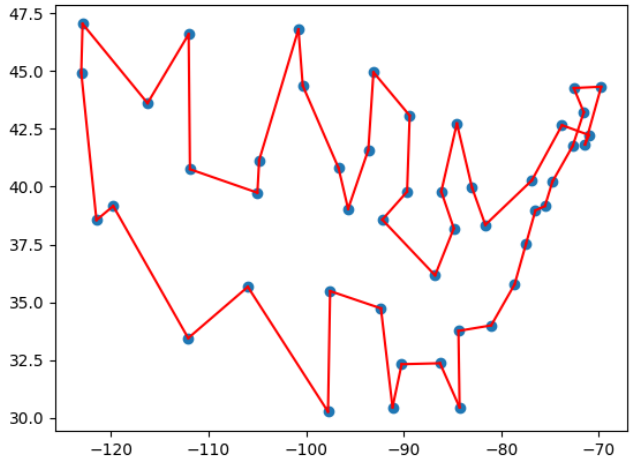
The second scheme is based on the observation that any time a route crosses itself, the length can be reduced by reversing the crossed segment. We don't actually look for crossings, we just try reversing random chunks of the ordering. This example reverses the [3 4 5] in [1:7] to get [1 2 5 4 3 6 7], thereby reducing the length from 9.47 to 7.65.



Because these changes are made at random, it is just as likely that we start with the picture on the right, make our modification, and end up with the picture on the left; that is, we can just as easily make things worse. But the advantage of doing these changes at random is that we avoid having to come up with, and then express computationally, all the tests that would tell us beforehand when our changes would be helpful. Sometimes it can be simpler and cheaper to try many changes at random than to carefully work out in advance what might be a good move.

After trying these random modifications, the program measures the length of the new path, and if it improves the record for shortest path, it becomes the current candidate. Since our random sampling approach didn't do so well, there seems to be no guarantee that this new method will work. The best thing we can do is test it out on our "fortyeight" example, keeping in mind as a comparison the estimated solution found by the greedy algorithm.

By starting with 10,000 random permutations, and applying these simple set of alterations, the `traveler()` code came up with a tour of length 155. If we plot this tour, it looks very good, except for a path crossing up near Maine. This suggests that we haven't found the best solution, but certainly, this looks very close. If we run traveler several times, we will get different results, and we may drive the distance down further.



No reliable algorithm has been found that can produce exact solutions of the TSP; this means that practical people have identified algorithms that work pretty well, and theoreticians continue to search for methods that improve the current algorithms and estimate how far off a given estimate is from the shortest path. The ultimate goal is a deterministic TSP solver, but as with many problems in combinatorics, this may be something that mathematics is just not ready for!