Distance Optimization: Travelling Salesman Problem

Kavya Kushnoor -with Dr. Gene Tagliarini

Abstract:

In this paper, I have examined the travelling salesman problem where an optimum total distance between fourteen cities is to be found. The cities have been randomly generated on a 1000*1000 units Cartesian plane using the random function in Python. Since this is a NP problem with no known solution with certainty, I have considered four distinct algorithms for comparison using these 14 points. In conclusion, the efficiency of each method has been compared. For an objective measurement, metrics like average, median, standard deviation, maximum and minimum values have been considered for a sample size of 30 different permutations to analyze where the algorithms stand in comparison. A visual representation of the analysis has been included.

Introduction:

Travelling salesman problem is an infamous computational problem with many possible solutions. For this paper, the size of n is 14. The possible solutions are hence 13 factorial - a whooping 6,227,020,800 possible solutions. Despite that, we can see how developing few algorithms which are close to optimal is better than using random strategies or the average of a random sample. I have considered 4 different approaches to solve this problem. A cross-comparison with statistical mean and median of 100000 samples has also been taken. A histogram representing the same is represented. 4 graphs representing the four different approaches has also been displayed for analysis.

Background:

The travelling salesman problem has an old history dating back to the 1800s. It has been laid out by the Irish and British mathematicians W.R. Hamilton and Thomas Kirkman. The most popular approaches for solving it have been the brute force algorithm and the nearest neighbor heuristic.

Since the number of cities in my paper has a large value of 14, the brute force method has been ruled out due to its inefficiency and excessive requirement of time to calculate permutations. It has however been used to gather generic statistical metrics like the mean, median, standard deviation, etc. To this end, it has been used without the aid of data storage structures like lists, dictionary or hash tables. The calculations have been made based on storage in variables to facilitate speed.

The 4 different algorithmic approaches that have been considered do a more in-depth analysis of the various possible strategies with which the problem can be solved. TSP is considered as an NP-hard problem. It has (n-1)! possibilities for n cities. In this case 13! is equal to 6,227,020,800

possible solutions. Most known solutions use approximate heuristics with no certain optimal solution. However, the usefulness of probable solution cannot be undermined.

A comparison of the algorithmic solutions with the statistical metrics of a sample size of 30 indicates that some of the algorithmic solutions are in fact relatively more optimal than the average/ median considered. A visual representation of the analysis will be examined in the paper. We will also walk through the logical steps taken for each algorithm along with the python code that facilitates the expedited process.

Experimental Design:

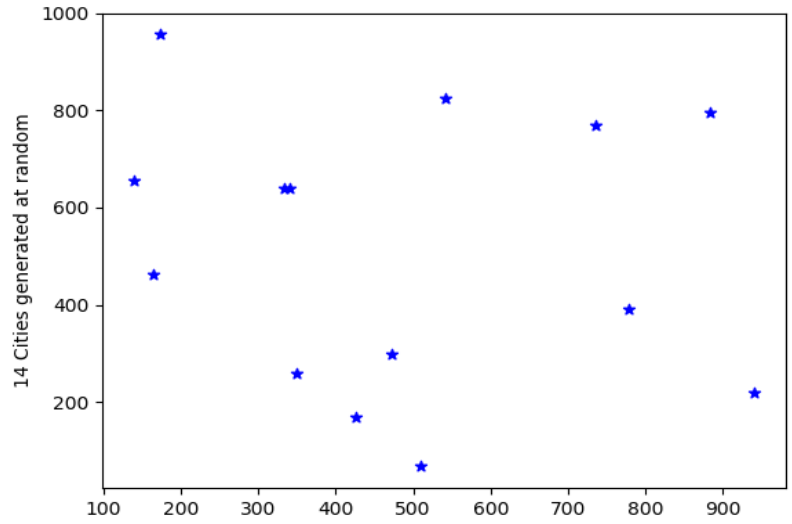
The experiment has a myriad of building functions which add up to the calculation of the optimum distance in the final stages. Following is the enumeration of the foundational functions:

1. A list of 14 city tuples has been generated at random using the cityList() function.

```
def cityList():
    cityList1 = []
    for x in range(1, 15):
        x = random.randint(0, 1001)
        y = random.randint(0, 1001)
        city = x, y
        cityList.append(city)
    return cityList1
```

For simplicity, a single list with hardcoded values has been used throughout the assignment for simplicity and uniformity of findings throughout the analysis. The list is:

```
cityList = [(541, 826), (334, 640), (736, 769), (350, 258),
(173, 956), (779, 392), (164, 461), (941, 218), (884, 795),(340,
640), (426, 168), (473, 298), (510, 67), (139, 655)]
```



2. A function for calculating the distances between any two random cities has been identified. The Euclidian formula, an equivalent to the Pythagorean Theorem, has been used for its calculation.

```
d(p,q) = sqrt((q1-p1)**2 + (q2-p2)**2)
```

For a city whose distance has to be calculated from itself, a value of infinity has been set instead of 0 to facilitate getting minimum distance in the optimal distance calculation. The distance function is:

```
def distance(p1, p2):
    #Calculate disance between 2 given points
    if p1 == p2:
        distance = math.inf
    else:
        distance = math.sqrt(((p1[0] - p2[0]) ** 2) + ((p1[1] -
p2[1]) ** 2))
    return round(distance, 2)
```

3. The maximum distance has been taken as the distance of the diagnol of the entire Cartesian plane times 13. This is a representative of the largest possible Hamiltonian cycle that can be undertaken to represent the maximum value.

```
def max_distance():
    #13 times the diagnol representing maximum value
    return round(distance((0, 0), (1000, 1000)) * 13, 2)
```

4. The total distance of all points in any given list has been calculated with the total_distance function. It returns a sum of distances of each point from the next point in the list until it reaches the [-1] index towards the end.

```
def total_distance(points):
    #Sum of distances between all points in a given list
    return round(sum([distance(point, points[index + 1]) for
index, point in enumerate(points[:-1])]), 2)
```

5. The distance of each point from the origin is then calculated. This will be further used to arrange the points in an ascending order of their distances from the origin. A total distance is calculated.

```
def origin_dist(p1):
    # Distance of each point from the origin
    return distance((0, 0), p1)
```

Algorithms:

Algorithm1: Sort the cities based on ascending order of their distances from the origin. Calculate total distance.

Algorithm2: Randomly calculate the distance based on each city that has been created in the random list of cities.

Algorithm3: Create a matrix to visualize total distance and calculate next closest city by referring to the matrix.

Algortihm4: Use permutations to check next closest city. High speed is used because no data structures/ variable have been implemented.

Algorithm1: Sorted Cities

First, I created a dictionary where the distance from origin is key and the respective city is value. The keys are then sorted in ascending order. The total distance is calculated.

```
def cityDict():
    dict = {}
    for x in cityList:
        dict[origin_dist(x)] = x
    return dict

def sortedCity():
    sortedCity1 = []
    dict = cityDict()
    for key in sorted(dict.keys()):
        sortedDict = dict[key]
        sortedCity1.append(sortedDict)
    return sortedCity1
```

Algorithm2: Random total distance

```
total_distance(cityList)
```

Algorithm3: Optimal Distance based on matrix

The distance of each city from the other city has been calculated and placed in a 14*14 matrix.

```
def eachDistance():
    a = []
    for x in sortedCity:
        for y in sortedCity:
            item = distance(x, y)
```

```
            a.append(item)
    return a


a = eachDistance()
def matrix():
    mat = []
    for z in range(14):
        row = []
        for q in range(14):
            row.append(a.pop())
        mat.append(row)
    return mat


def print_matrix():
    c = np.reshape(a, (14, 14))
    return c
```

A visual representation of the matrix is as follows:

```
mat = [[math.inf, 150.27, 344.4, 729.0, 579.81, 416.45, 565.65,
571.42, 758.04, 644.93, 818.45, 793.7, 776.46, 757.31],
       [150.27, math.inf, 203.16, 593.24, 587.9, 379.44, 416.48,
422.19, 607.79, 539.45, 737.48, 649.65, 676.24, 640.4],
       [344.4, 203.16, math.inf, 390.29, 727.78, 494.97, 273.86,
278.29, 436.86, 532.36, 759.63, 524.74, 667.97, 599.25],
       [729.0, 593.24, 390.29, math.inf, 1065.11, 827.85,
357.41, 354.65, 302.91, 723.16, 950.73, 495.08, 827.62, 720.09],
       [579.81, 587.9, 727.78, 1065.11, math.inf, 237.74,
734.36, 739.28, 913.33, 474.79, 456.69, 814.11, 517.42, 592.35],
       [416.45, 379.44, 494.97, 827.85, 237.74, math.inf,
504.21, 509.44, 691.93, 320.11, 421.88, 618.86, 418.07, 449.44],
       [565.65, 416.48, 273.86, 357.41, 734.36, 504.21,
math.inf, 6.0, 201.56, 366.95, 597.69, 251.03, 479.77, 382.13],
       [571.42, 422.19, 278.29, 354.65, 739.28, 509.44, 6.0,
math.inf, 195.58, 369.17, 599.42, 246.86, 480.88, 382.33],
       [758.04, 607.79, 436.86, 302.91, 913.33, 691.93, 201.56,
195.58, math.inf, 488.88, 695.26, 195.6, 565.28, 449.59],
       [644.93, 539.45, 532.36, 723.16, 474.79, 320.11, 366.95,
369.17, 488.88, math.inf, 233.94, 349.36, 138.24, 129.34],
       [818.45, 737.48, 759.63, 950.73, 456.69, 421.88, 597.69,
599.42, 695.26, 233.94, math.inf, 524.36, 131.37, 249.16],
       [793.7, 649.65, 524.74, 495.08, 814.11, 618.86, 251.03,
246.86, 195.6, 349.36, 524.36, math.inf, 393.06, 275.33],
       [776.46, 676.24, 667.97, 827.62, 517.42, 418.07, 479.77,
480.88, 565.28, 138.24, 131.37, 393.06, math.inf, 117.8],
```

```
        [757.31, 640.4, 599.25, 720.09, 592.35, 449.44, 382.13,
382.33, 449.59, 129.34, 249.16, 275.33, 117.8, math.inf]]
```

An optimum path is then created by looping through each of the rows and finding the minimum distance and appending it the optimum list. The cities that are repeated, are appended to another list for which a permutation is used to find the smallest total distance and a final optimum path is created.

```
def opt_path():
    opt_path = [0]
    for z in mat:
        opt_index = z.index(min(z))
        if opt_index not in opt_path:
            opt_path.append(opt_index)
        else:
            pass
    return opt_path


def rem_cities():
    rem_cities = []
    for x in range(0, 13):
        if x not in opt_path:
            rem_cities.append(x)
    return rem_cities

rem_cities = rem_cities()
def opt_pathPoints():
    opt_pathPoints = []
    for x in opt_path:
        opt_pathPoints.append(sortedCity[x])
    rem_citiesPoints = []
    for x in rem_cities:
        rem_citiesPoints.append(sortedCity[x])
    rem_citiesPoints.append(opt_pathPoints[-1])
    newRemList =
(optimized_travelling_salesman(rem_citiesPoints,
opt_pathPoints[-1]))
    opt_pathPoints.pop()
    for x in newRemList:
        opt_pathPoints.append(x)
    return opt_pathPoints
```

Algorithm4: Optimum path without matrix

This algorithm uses permutations to find the next closest city and uses it to create an optimum path.

```
def optimized_travelling_salesman(points, start=None):
    if start is None:
        start = points[0]
    must_visit = points
    path = [start]
    must_visit.remove(start)
    while must_visit:
        nearest = min(must_visit, key=lambda x: distance(path[-
1], x))
        path.append(nearest)
        must_visit.remove(nearest)
    return path
```

Findings:

A ranking of each of these algorithms would be in the following order based on their optimality:

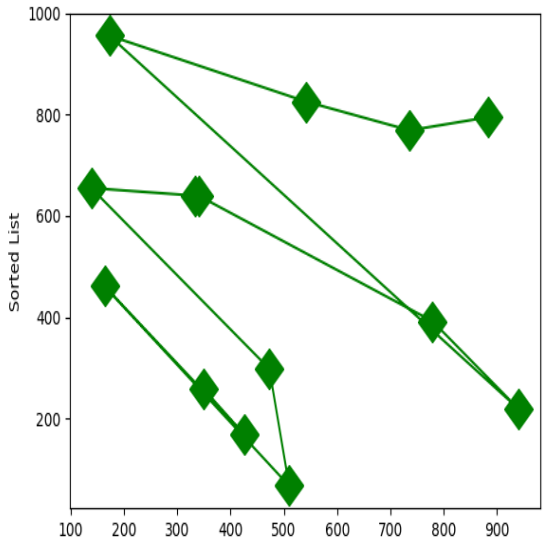
1. Algorithm 4 – Without Matrix
2. Algorithm 1 – Sorted Cities Algorithm
3. Algorithm 3 – With Matrix
4. Algorithm 2 – Random Computation

The statistical metrics have also been computed for a more objective comparison of the performance of these algorithms.

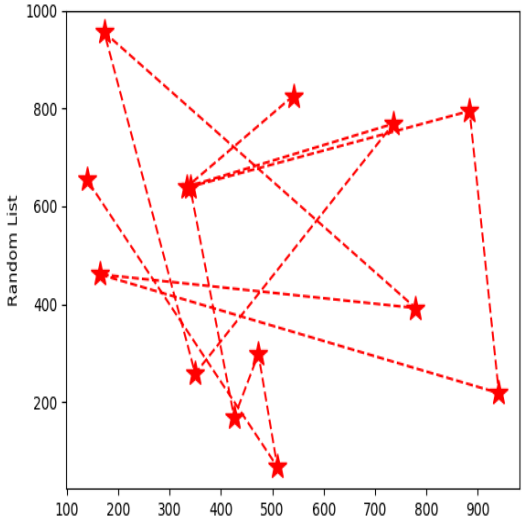
- Minimum distance: 3140.32
- Maximum distance: 18384.73
- Total: 682078738.68
- Average: 6820.85
- Standard Deviation: 508.48
- Median: 7232.98
- Mode : None (All values in sample are unique – 13 factorial solutions)

Graphs:

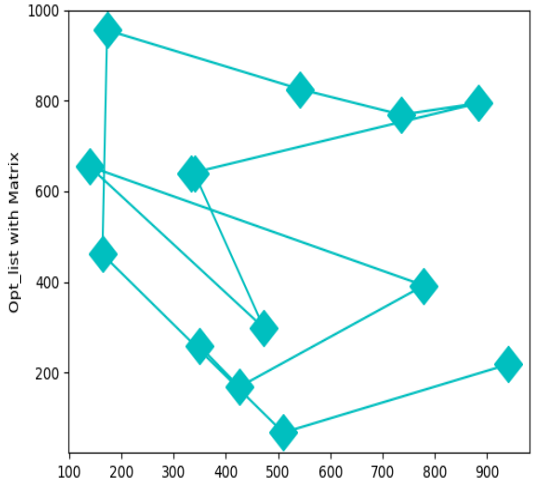
Sorted Algorithm
Total Distance: 4510.4



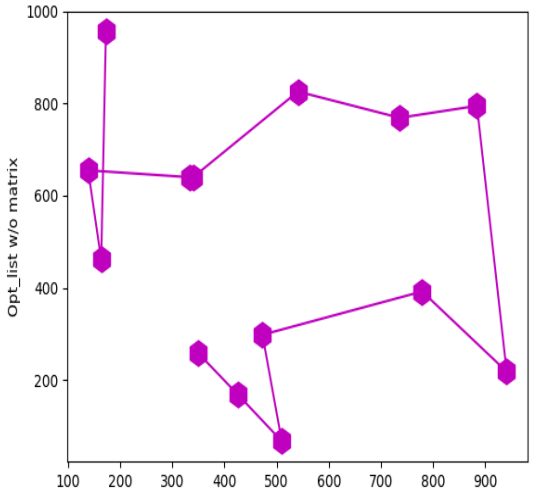
Random Algorithm
Total Distance: 7014.46

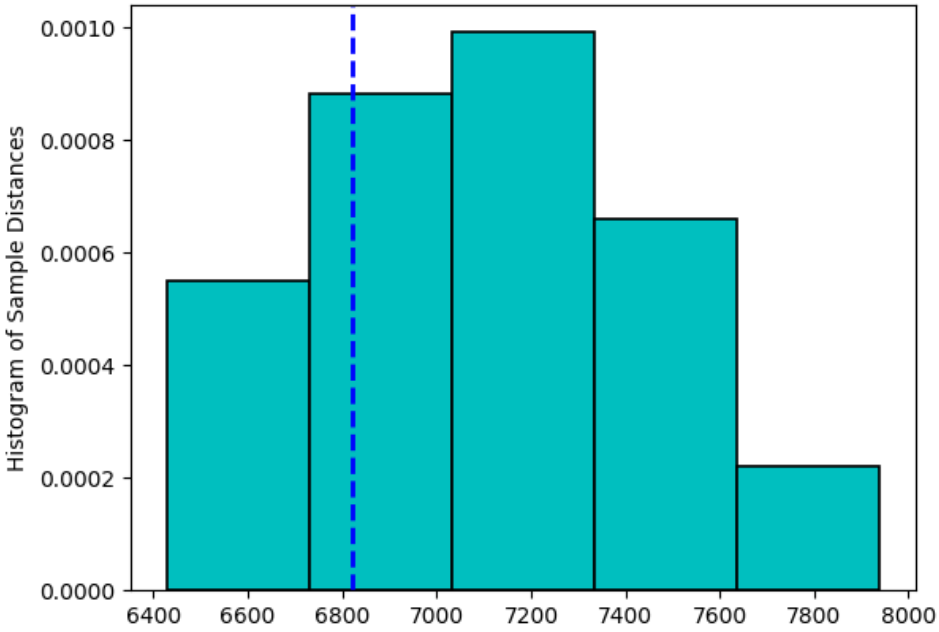


3. With Matrix
Total Distance: 4880.9



4. Without Matrix
Total Distance: 3140.32





Conclusion:

Some of the key takeaways in conclusion are:

1. Using data structures relies on RAM which could slow down the process of computation. It is hence a more effective algorithm if computations can be made on the fly; with or without the use of a variable. It enhances speed.
2. Developing algorithms for optimum costs has proved to be beneficial in 3 out of 4 cases in comparison with generic statistical metrics like mean/ median. Only the random method has exceeded these metrics.
3. Knowing the correct intrinsic motivator (confidence of skill) is the most optimal factor while tackling a difficult problem such as this.

Future Works:

1. Solving this problem using trees.
2. Finding more optimal ways with which to use Python for grazing through the matrix of distances.
3. Trying the TSP with object-oriented programming languages like Java.
4. Understanding complex algorithms with more depth.

References:

- Travelling salesman problem. (2018, January 30). Retrieved February 01, 2018, from <<https://en.wikipedia.org/wiki/Travelling_salesman_problem>>
- Matplotlib.pyplot.plot. (n.d.). Retrieved February 01, 2018, from <<https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html>>
- Travelling salesman using brute-force and heuristics. (n.d.). Retrieved February 01, 2018, from <<https://codereview.stackexchange.com/questions/81865/travelling-salesman-using-brute-force-and-heuristics>>
- OPERATIONS RESEARCH|| Tutorial on TRAVEL SALESMAN PROBLEM || Step -by- Step Procedure. (2016, September 14). Retrieved February 01, 2018, from <<https://www.youtube.com/watch?v=_Qb9YWG5Mmo>>