



VIT[®]

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

A Comparison in the Implementation of The Travelling Salesman Problem using Brute – Force and Ant Colony

Optimisation

*J Component Report for the course Data Structures and
Algorithms (CSE2003)*

Submitted By:

NAME	REG. NO.	SLOT
AMAN ANAND	19BCE0521	B1
AKSHAT TRIPATHI	19BCE0527	B1
AYUSH KHARE	19BCE0498	B1

In partial fulfilment for the award of the degree of

Bachelor of Technology

In

COMPUTER SCIENCE ENGINEERING

Under the guidance of

Faculty: Prof. Raghavan R

School of Computer Science and Engineering

Academic Year: 2019 - 2020

Review One

Abstract:

Problem statement (Travelling Salesman Problem - TSP) –

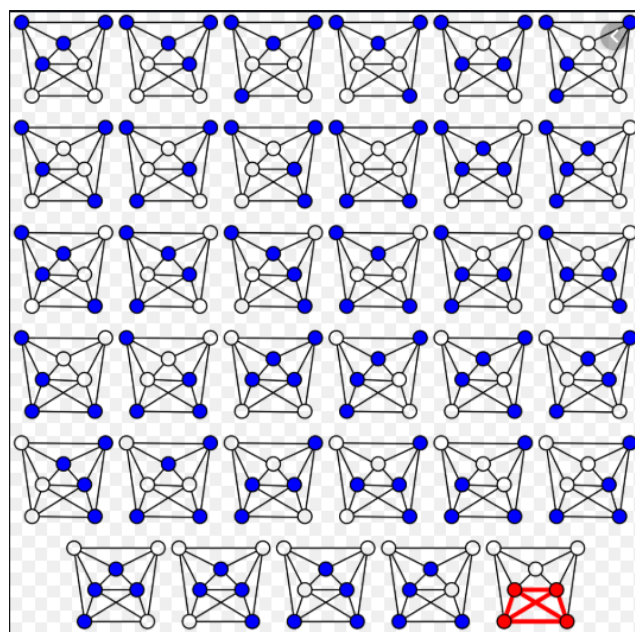
Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Measure the time required by Brute Force Strategy and Ant Colony Optimisation to determine the shortest path possible.

The Travelling Salesman Problem is one of the best known NP-hard problems, which means that there is no exact algorithm to solve it in polynomial time. The minimal expected time to obtain optimal solution is exponential. So, for that reason, we usually use heuristics to help us to obtain a “good” solution.

Brute Force Technique:

Introduction-

Brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.



This type of algorithm usually involves the direct search of the solution by the compiler constrained to the problem criteria. This algorithm exploits every possible solution available and proceeds in a way so as to select the most optimal solution.

Although this may prove to be a fruitituous approach regarding small data sets, it is a very time consuming and compiler over-bearing algorithm with respect to large sets of data. Brute force algorithm has various disadvantages which are derived from the above listed problems. These include:

- Combinatorial Explosion (or) the curse of dimensionality
- Time Complexity
- Memory Storage Efficiency

To overcome the above mentioned complexities and disadvantages we have adapted the ACO technique, which has its own set of problems, but is comparatively efficient with respect to the Brute Force Technique.

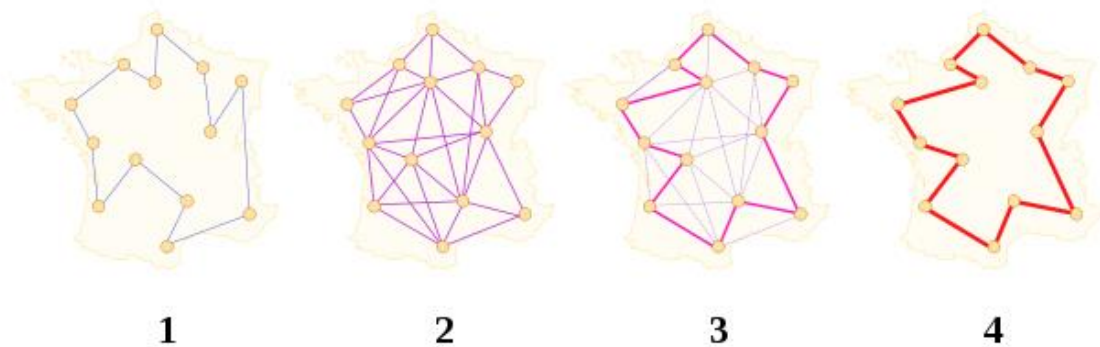
Ant Colony Optimization (ACO) Technique:

Introduction-

In the natural world, ants of some species (initially) wander randomly, and upon finding food return to their colony while laying down pheromone trails. If other ants find such a path, they are likely not to keep travelling at random, but instead to follow the trail, returning and reinforcing it if they eventually find food.

Over time, however, the pheromone trail starts to evaporate, thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison, gets marched over more frequently, and thus the pheromone density becomes higher on shorter paths than longer ones. Pheromone evaporation also has the advantage of avoiding the convergence to a locally optimal solution. If there were no evaporation at all, the paths chosen by the first ants would tend to be excessively attractive to the following ones. In that case, the exploration of the solution space would be

constrained. The influence of pheromone evaporation in real ant systems is unclear, but it is very important in artificial systems. The overall result is that when one ant finds a good (i.e., short) path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leads to many ants following a single path. The idea of the ant colony algorithm is to mimic this behaviour with "simulated ants" walking around the graph representing the problem to solve.



Given an n-city TSP with distances d_{ij} , the artificial ants are distributed to these n cities randomly. Each ant will choose the next to visit according to the pheromone trail remained on the paths just as mentioned in the above example. However, there are two main differences between artificial ants and real ants:

(1) The artificial ants have "memory"; they can remember the cities they have visited and therefore they would not select those cities again.

(2) The artificial ants are not completely "blind"; they know the distances between two cities and prefer to choose the nearby cities from their positions. Therefore, the probability that city j is selected by ant k to be visited after city i could be written as follows:

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{s \in allowed_k} [\tau_{is}]^\alpha \cdot [\eta_{is}]^\beta} & j \in allowed_k \\ 0 & \text{otherwise} \end{cases}$$

where τ_{ij} is the intensity of pheromone trail between cities i and j , α the parameter to regulate the influence of τ_{ij} , η_{ij} the visibility of city j from city i , which is always set as $1/d_{ij}$ (d_{ij} is the distance between city i and j), β the parameter to regulate the influence of η_{ij} and allowed k the set of cities that have not been visited yet, respectively.

At the beginning, l ants are placed to the n cities randomly. Then each ant decides the next city to be visited according to the probability p_{ij}^k given by Eq. (1). After n iterations of this process, every ant completes a tour. Obviously, the ants with shorter tours should leave more pheromone than those with longer tours.

Therefore, the trail levels are updated as on a tour each ant leaves pheromone quantity given by Q/L_k , where Q is a constant and L_k the length of its tour, respectively. On the other hand, the pheromone will evaporate as the time goes by. Then the updating rule of τ_{ij} could be written as follows:

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

$$\Delta\tau_{ij} = \sum_{k=1}^l \Delta\tau_{ij}^k$$

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

Where t is the iteration counter, $\rho \in [0, 1]$ the parameter to regulate the reduction of τ_{ij} , $\Delta\tau_{ij}$ the total increase of trail level on edge (i, j) and $\Delta\tau_{ijk}$ the increase of trail level on edge (i, j) caused by ant k , respectively. After the pheromone trail updating process, the next iteration $t + 1$ will start.

The ACO technique comes under the category of solving the TSP problem statement using a meta-heuristic approach to the solution. Meta-heuristic simply implies the compilers ability to learn from its' previously traversed routes and apply it to make a partially educated guess regarding to remaining part of the algorithm.

There are many disadvantages to approaching the problem using ACO, but this algorithm gives us a much more efficient time complexity $O(n \log n)$ with respect to the brute force technique $O(n^2)$.

Some of the drawbacks of this algorithm are:

- Probability distribution can change for each iteration.
- Have a difficult theoretical analysis.
- Have uncertain time to convergence.
- Have dependant sequences of random decisions.
- Have more experimental than theoretical research.

Review Two

Drawbacks of Brute-Force Algorithm:

We as a team have mainly focused on overcoming three of the major drawbacks of the brute force algorithm used to implement the Travelling Salesman Problem. These drawbacks include:

a) Combinatorial Explosion (or) the curse of dimensionality

Combinatorial Explosion (or commonly referred to as the curse of dimensionality) refers to the large number of combination which arise when one undertakes the Brute- Force method to solve a problem. Travelling Salesman Problem essentially find a route between the cities to visit. These cities are present as vertices in a graph and the “paths” traversed by the salesman refer to the edges in the graph. Now, we know that in a closed polygon the number of edges formed between the n vertices can be brought about by the formula nC_2 , as from the n vertices we choose any two vertices to join which forms an edge. When employing the brute force method, it judges each and every path and then finds the optimum route for the salesman. In essence, the brute force method has nC_2 iterations in the program and additional computational time (although small) to find the optimal path. In small groups of vertices brute force can

be applied as the value of n is small and the computation time is still small. But if we were extrapolate the above theory to a large set of vertices (as is often the case, when we consider a large “city”) the value of n increases significantly. This causes the value of nC_2 to increase manifold and causes the computational time to also increase manifold. This drawback which results in compilation time of the program sky-rocketing, is called as the curse of dimensionality or combinatorial explosion.

b) Time Complexity

As discussed above, in large “cities” or graph when we employ the Travelling Salesman Algorithm, we find that the computational time has increased significantly. It is common knowledge that the time complexity of program varies directly with the number of iterations in a program. As explained in the previous mentioned point, the more the number of cities that we include in the problem statement, the greater is the number of iterations that we have to undergo. And as mentioned above, it is clear that more the number of iterations, the more is the time complexity of the program. When implementing this program for a huge city the value of “ n ” increases severely and the time complexity becomes mountainous decreasing the efficiency of the program.

c) Memory Storage Efficiency

In addition to the above two problems it can easily be deduced that one startlingly clear problem becomes the storage of all the combination that the brute force algorithm produces, so that we can come back and analyse these combination so as to furnish the most optimum path. Again, with such a heavy load of data, the efficiency

in storage reduces significantly, hence reduces the neatness of the program and its extent of efficiency.

Proposal to overcome the above adversities.

To reduce the impact of the above mentioned drawback and to some extent even overcome it, we have undertaken using a slightly different algorithmic approach to the travelling salesman problem. The algorithm that our team chose is the ACO (Ant Colony Optimization) Technique. ACO has been covered in detail in the previous review, but we would like to shed some light in the less highlighted areas. Like any other algorithm, ACO also has its set of drawbacks. To quickly summarize them, they include:

a) Probability distribution can change for each iteration.

After each iteration of the ACO the probability distribution for the next path changes according to the formula.

b) Have a difficult theoretical analysis.

c) Have uncertain time to convergence.

The time taken by the program to converge to the end point of the graph and produce an optimum path for the result is variable as the probability of the next path taken changes after each iteration.

d) Have dependant sequences of random decisions.

Navigating past the above mentioned difficulties, ACO still provides a more optimal result in cases where the brute force algorithm falters majorly. Whereas brute forces algorithm faces difficulties like, time complexity, and combinatorial explosion when the number of

vertices or “cities” in the graph increases past a certain point, ACO algorithm takes the large number of cities in stride and this algorithm gives us a much more efficient time complexity **$O(n \log n)$** with respect to the brute force technique **$O(n^2)$** .

The way this algorithm essentially works is:

At the beginning, l ants are placed to the n cities randomly. Then each ant decides the next city to be visited according to the probability p_{ij}^k . After n iterations of this process, every ant completes a tour. Obviously, the ants with shorter tours should leave more pheromone than those with longer tours. Therefore, the trail levels are updated as on a tour each ant leaves pheromone quantity given by Q/L_k , where Q is a constant and L_k the length of its tour, respectively. On the other hand, the pheromone will evaporate as the time goes by. Where t is the iteration counter, $\rho \in [0, 1]$ the parameter to regulate the reduction of τ_{ij} , $\Delta\tau_{ij}$ the total increase of trail level on edge (i, j) and $\Delta\tau_{ijk}$ the increase of trail level on edge (i, j) caused by ant k , respectively. After the pheromone trail updating process, the next iteration $t + 1$ will start.

As it has been highlighted several times in the project, ACO possess some drawbacks. Keeping that in mind, ACO is far more efficient than brute force algorithm for our problem statement, because of its ability to output not only an optimal path but to do so in a stipulated and controlled time frame. It's time efficiency has a huge headway compared to the brute force algorithm. If we were to take an example of a student studying for an exam, if he were to randomly memorize his notes and then try to piece together the data while sitting in the exam room summarizes the brute force method. In ACO, the student takes the path in which he memorizes his notes in the correct and most optimal order, often omitting redundant information so as to have an easier time recalling the concepts during the exam.

The above points are well founded in theory but are supported by code that goes on to provide overwhelming proof to the light that we have tried to shine on ACO Technique.

IMPLEMENTATION OF TSP USING BRUTE FORCE:

CODE (PYTHON 3 ; matplotlib and colorama libraries required).

main.py

```
from timeit import Timer
from graph_tsp import graph_tsp
def runCode():
    var = str(input("Enter filename (with no extension): "))
    T = graph_tsp(var)
t1 = Timer("""runCode()""", """from __main__ import runCode""")
print("Time for Completion (seconds): " '{0:.1f}'.format(t1.timeit(1)))
```

graph_tsp.py

```
from itertools import permutations
import csv
import logging
from colorama import Fore, Back, Style
import matplotlib.pyplot as plt
class graph_tsp():
    def __init__(self, filename):
        try:
            self.logger = self.setupLogger()
            self.filename = filename

# Open CSV file and read each row into a list of lists
        with open('data/' + self.filename + '.csv', newline='') as file:
            reader = csv.reader(file)
            points_original = list(map(list, reader))
```

```

# Remove header column
del points_original[0]

# Remove city index from the points
cities = []
for point in points_original:
    cities.append(point.pop(0))

# Convert string values to float
points = []
for item in points_original:
    point = []
    for values in item:
        point.append(float(values))
    points.append(point)

# Show Plot for original data provided
title = 'The TSP Original Data Route'
self.plotIt(points, title)
self.logger.info('CSV file data/' + self.filename + '.csv was
read...')

# Print distance
print("""Original Distance: {} \nRecommended Tour Route: {}
\nMinimum distance:
{}""".format('{0:.1f}'.format(self.total_distance(points)),tuple(points),
'{0:.1f}'.format(self.total_distance(self.calculate_TSP(points)))))
except Exception as e:
    self.logger.error(Fore.RED + 'Error reading CSV data/' +
self.filename + '.csv file... %s' % str(e) + Style.RESET_ALL)
def setupLogger(self):
    logfile = 'logs/app_log.txt'

# Format log to add time, level and message

```

```

logging.basicConfig(level=logging.WARNING)
logFormatter = logging.Formatter('%(asctime)s - %(levelname)s -
-: %(message)s')
logger = logging.getLogger(__name__)
# Add handler to send logs to log file
handler = logging.FileHandler(logfile)
handler.setFormatter(logFormatter)
handler.setLevel(logging.INFO)
logger.addHandler(handler)
logger.info('Logs written in %s' % logfile)
return logger
def distance(self, point_one, point_two):
    complete = False
    try:
        distance = ( (point_one[0] - point_two[0]) ** 2.0 +
(point_one[1] - point_two[1]) ** 2.0) ** 0.5
        complete = True
        return distance
    except Exception as e:
        self.logger.error(Fore.RED + 'Error calculating distance
between points... %s' % str(e) + Style.RESET_ALL)
    finally:
        if complete:
            self.logger.info('Distance between all points has been
calculated...')
def total_distance(self, points):
    # Determine if the function has been completed
    complete = False
    try:
        total_distance = sum([self.distance(p, points[value + 1]) for
value, p in enumerate(points[:-1])])
        complete = True
        return total_distance
    except Exception as e:

```

```

        self.logger.error(Fore.RED + 'Error calculating total distance
between points... %s' % str(e) + Style.RESET_ALL)
    finally:
        if complete:
            self.logger.info('Total distance between all points has been
calculated...')
    def calculate_TSP(self, points):

# Determine if the function has been completed
    complete = False

# Starting position
    start_position = points[0]

# Places to visit
    visit_list = points

# Set tour to starting position
    tour = [start_position]

# Remove the starting position
    visit_list.remove(start_position)
    try:

# Loop through places to visit to find min distance
        while visit_list:
            key_value = lambda y: self.distance(tour[-1], y)
            # Find nearest city
            nearest_city = min(visit_list, key = key_value)

# Append nearest city to the tour and remove it from the visit list
            tour.append(nearest_city)
            visit_list.remove(nearest_city)
        complete = True

```

```

# Show Plot for the evaluated min tour route order
    title = 'The TSP Calculated Minimum Route'
    self.plotIt(tour, title)
    return tour
except Exception as e:
    self.logger.error(Fore.RED + 'Error calculating minimum
distance between points and finding total path... %s' % str(e) +
Style.RESET_ALL)
    finally:
        if complete:
            self.logger.info('Minimum distance between all points
and path have been calculated...')
    def plotIt(self, tour_points, title):
        try:

# Grab X and Y values of points
            x = []
            y = []
            for values in tour_points:
                x.append(values[0])
                y.append(values[1])

# Append the first item to the end to complete the loop
            x.append(x[0])
            y.append(y[0])

# Setup plot
            plt.plot(x, y, 'b--')
            plt.xlabel('Longitude (X)')
            plt.ylabel('Latitude (Y)')
            plt.title(title)
            plt.grid(True)

# Plot TSP route
            plt.show()

```

```

self.logger.info('Plot created of route... ')
except Exception as e:
    self.logger.error(Fore.RED + 'Error plotting route. Check
data provided and/or graph settings... %s' % str(e) + Style.RESET_ALL)

```

INPUT DATA:

1 3639 1315	40 3143 3421	79 3928 3029	118 3896 1656
2 3569 1438	41 3130 2973	80 4186 3037	119 3918 2179
3 3904 1289	42 2765 3321	81 3322 1916	120 3972 2136
4 3506 1221	43 2545 2357	82 3429 1908	121 4029 2498
5 3237 1764	44 2611 2275	83 3176 2150	122 3766 2364
6 3089 1251	45 2860 2862	84 3229 2367	123 3896 2443
7 3238 1229	46 2801 2700	85 3402 2912	124 3796 2499
8 4172 1125	47 2370 2975	86 3402 2510	125 3478 2705
9 4020 1142	48 1084 2313	87 3468 3018	126 3810 2969
10 4095 626	49 4177 2244	88 3356 3212	127 4167 3206
11 4403 1022	50 3757 1187	89 3044 3081	128 3486 1755
12 4361 73	51 3488 1535	90 3140 3550	129 3334 2107
13 4634 654	52 3374 1750	91 2769 2492	130 3587 2417
14 2846 1951	53 3326 1556	92 2348 2652	131 3507 2376
15 3054 1710	54 3258 911	93 2778 2826	132 3264 2551
16 2562 1756	55 3646 234	94 2126 2896	133 3360 2792
17 2291 1403	56 4089 1387	95 1890 3033	134 3439 3201
18 2012 1552	57 4196 1044	96 3538 3298	135 3526 3263
19 682 825	58 4312 790	97 3712 1399	136 3012 3394
20 518 1251	59 4685 830	98 3493 1696	137 2935 3240
21 1332 695	60 4720 557	99 3791 1339	138 3053 3739
22 4016 1715	61 4153 426	100 3376 1306	139 2284 2803
23 4087 1546	62 2831 2099	101 3188 1881	140 2577 2574
24 4062 2220	63 3086 1516	102 3814 261	141 2592 2820
25 4061 2370	64 2716 1924	103 3583 864	142 2401 3164
26 4201 2397	65 2751 1559	104 4297 1218	143 1304 2312
27 3777 2095	66 1779 1626	105 4116 1187	144 3470 3304
28 3888 2261	67 1478 267	106 4252 882	
29 3678 2463	68 278 890	107 4386 570	
30 3789 2620	69 3715 1678	108 4643 404	
31 3862 2839	70 4181 1574	109 4784 279	
32 4263 2931	71 3929 1892	110 3007 1970	

33 3492 1901	72 3751 1945	111 1828 1210	
34 3479 2198	73 4207 2533	112 2061 1277	
35 3318 2408	74 4139 2615	113 2788 1491	
36 3296 2217	75 3780 2212	114 2381 1676	
37 3394 2643	76 3594 2900	115 1777 892	
38 3101 2721	77 3676 2578	116 1064 284	
39 3792 3156	78 4029 2838	117 3688 1818	

OUTPUTS AND SCREENSHOTS:

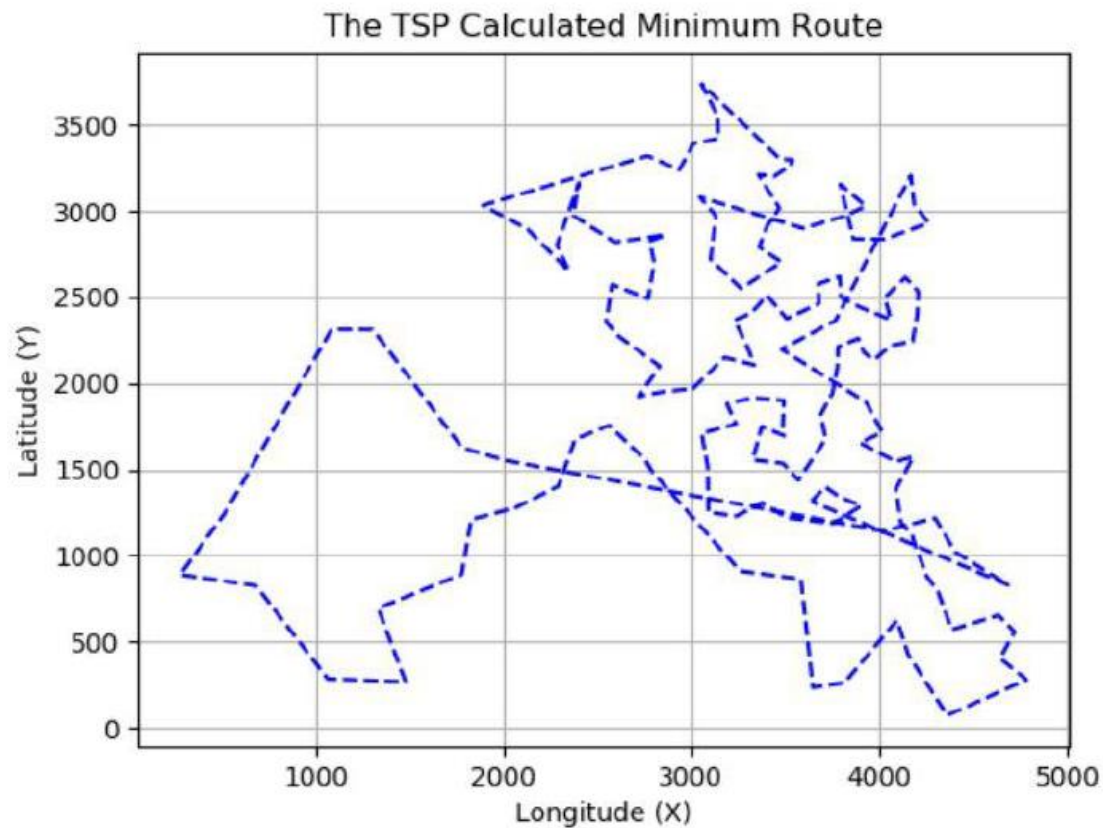
```

Python 3.6.6 Shell
File Edit Shell Debug Options Window Help

Python 3.6.6 (v3.6.6:4c1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Bhuvanjeet\Desktop\TSP-master\Main.py =====
Enter filename (with no extension): dataf
Original Distance: 94231.5
Recommended Tour Route: ([3639.0, 1315.0], [3569.0, 1438.0], [3904.0, 1289.0], [3506.0, 1221.0], [3237.0, 1764.0], [3089.0, 1251.0], [3238.0, 1229.0], [4172.0, 1125.0], [4020.0, 1142.0], [4095.0, 626.0], [4403.0, 1022.0], [4361.0, 73.0], [4634.0, 654.0], [2846.0, 1951.0], [3054.0, 1710.0], [2562.0, 1756.0], [2291.0, 1403.0], [2012.0, 1552.0], [682.0, 825.0], [518.0, 1251.0], [1332.0, 695.0], [4016.0, 1715.0], [4087.0, 1546.0], [4062.0, 2220.0], [4061.0, 2370.0], [4201.0, 2397.0], [3777.0, 2095.0], [3888.0, 2261.0], [3678.0, 2463.0], [3789.0, 2620.0], [3862.0, 2839.0], [4263.0, 2931.0], [3492.0, 1901.0], [3479.0, 2198.0], [3318.0, 2408.0], [3296.0, 2217.0], [3394.0, 2643.0], [3101.0, 2721.0], [3792.0, 3156.0], [3143.0, 3421.0], [3130.0, 2973.0], [2765.0, 3321.0], [2545.0, 2357.0], [2611.0, 2275.0], [2860.0, 2862.0], [2801.0, 2700.0], [2370.0, 2975.0], [1084.0, 2313.0], [4177.0, 2244.0], [3757.0, 1187.0], [3488.0, 1535.0], [3374.0, 1750.0], [3326.0, 1556.0], [3258.0, 911.0], [3646.0, 234.0], [4089.0, 1387.0], [4196.0, 1044.0], [4312.0, 790.0], [4685.0, 830.0], [4720.0, 557.0], [4153.0, 426.0], [2831.0, 2099.0], [3086.0, 1516.0], [2716.0, 1924.0], [2751.0, 1559.0], [1779.0, 1626.0], [1478.0, 267.0], [278.0, 890.0], [3715.0, 1678.0], [4181.0, 1574.0], [3929.0, 1892.0], [3751.0, 1945.0], [4207.0, 253.0], [4139.0, 2615.0], [3780.0, 2212.0], [3594.0, 2900.0], [3676.0, 2578.0], [4029.0, 2838.0], [3928.0, 3029.0], [4186.0, 3037.0], [3322.0, 1916.0], [3429.0, 1908.0], [3176.0, 2150.0], [3229.0, 2367.0], [3402.0, 2912.0], [3402.0, 2510.0], [3468.0, 3018.0], [3356.0, 3212.0], [3044.0, 3081.0], [3140.0, 3550.0], [2769.0, 2492.0], [2348.0, 2652.0], [2778.0, 2826.0], [2126.0, 2896.0], [1890.0, 3033.0], [3538.0, 3298.0], [3712.0, 1399.0], [3493.0, 1696.0], [3791.0, 1339.0], [3376.0, 1306.0], [3188.0, 1881.0], [3814.0, 261.0], [3583.0, 864.0], [4297.0, 1218.0], [4116.0, 1187.0], [4252.0, 882.0], [4386.0, 570.0], [4643.0, 404.0], [4784.0, 279.0], [3007.0, 1970.0], [1828.0, 1210.0], [2061.0, 1277.0], [2788.0, 1491.0], [2381.0, 1676.0], [1777.0, 892.0], [1064.0, 284.0], [3688.0, 1818.0], [3896.0, 1656.0], [3918.0, 2179.0], [3972.0, 2136.0], [4029.0, 2498.0], [3766.0, 2364.0], [3896.0, 2443.0], [3796.0, 2499.0], [3478.0, 2705.0], [3810.0, 2969.0], [4167.0, 3206.0], [3486.0, 1755.0], [3334.0, 2107.0], [3587.0, 2417.0], [3507.0, 2376.0], [3264.0, 2551.0], [3360.0, 2792.0], [3439.0, 3201.0], [3526.0, 3263.0], [3012.0, 3394.0], [2935.0, 3240.0], [3053.0, 3739.0], [2284.0, 2803.0], [2577.0, 2574.0], [2592.0, 2820.0], [2401.0, 3164.0], [1304.0, 2312.0], [3470.0, 3304.0])
Minimum distance: 34732.0
Time for Completion (seconds): 71.7
>>>

```


GRAPH DEPICTING THE PATH TAKEN:



IMPLEMENTATION OF TSP USING ACO.

CODE (PYTHON 3 ; matplotlib library required).

main.py

```
import math
```

```
from aco import ACO, Graph  
from plot import plot
```

```
def distance(city1: dict, city2: dict):  
    return math.sqrt((city1['x'] - city2['x']) ** 2 + (city1['y'] - city2['y'])  
    ** 2)
```

```

def main():
    cities = []
    points = []
    with open('./data/chn144.txt') as f:
        for line in f.readlines():
            city = line.split(' ')
            cities.append(dict(index=int(city[0]), x=int(city[1]),
y=int(city[2])))
            points.append((int(city[1]), int(city[2])))
    cost_matrix = []
    rank = len(cities)
    for i in range(rank):
        row = []
        for j in range(rank):
            row.append(distance(cities[i], cities[j]))
        cost_matrix.append(row)
    aco = ACO(10, 100, 1.0, 10.0, 0.5, 10, 2)
    graph = Graph(cost_matrix, rank)
    path, cost = aco.solve(graph)
    print('cost: {}, path: {}'.format(cost, path))
    plot(points, path)

if __name__ == '__main__':
    main()

```

aco.py

```
import random
```

```

class Graph(object):
    def __init__(self, cost_matrix: list, rank: int):

```

```

"""
:param cost_matrix:
:param rank: rank of the cost matrix
"""

self.matrix = cost_matrix
self.rank = rank
# noinspection PyUnusedLocal
self.pheromone = [[1 / (rank * rank) for j in range(rank)] for i in
range(rank)]

```

```

class ACO(object):
    def __init__(self, ant_count: int, generations: int, alpha: float,
beta: float, rho: float, q: int,
        strategy: int):
        """
        :param ant_count:
        :param generations:
        :param alpha: relative importance of pheromone
        :param beta: relative importance of heuristic information
        :param rho: pheromone residual coefficient
        :param q: pheromone intensity
        :param strategy: pheromone update strategy. 0 - ant-cycle, 1 -
ant-quality, 2 - ant-density
        """

        self.Q = q
        self.rho = rho
        self.beta = beta
        self.alpha = alpha
        self.ant_count = ant_count
        self.generations = generations
        self.update_strategy = strategy

    def _update_pheromone(self, graph: Graph, ants: list):
        for i, row in enumerate(graph.pheromone):

```

```

        for j, col in enumerate(row):
            graph.pheromone[i][j] *= self.rho
        for ant in ants:
            graph.pheromone[i][j] += ant.pheromone_delta[i][j]

# noinspection PyProtectedMember
def solve(self, graph: Graph):
    """
    :param graph:
    """
    best_cost = float('inf')
    best_solution = []
    for gen in range(self.generations):

# noinspection PyUnusedLocal
        ants = [_Ant(self, graph) for i in range(self.ant_count)]
        for ant in ants:
            for i in range(graph.rank - 1):
                ant._select_next()
            ant.total_cost += graph.matrix[ant.tabu[-1]][ant.tabu[0]]
            if ant.total_cost < best_cost:
                best_cost = ant.total_cost
                best_solution = [] + ant.tabu

# update pheromone
            ant._update_pheromone_delta()
            self._update_pheromone(graph, ants)

        # print('generation #{}', best cost: {}, path: {}'.format(gen, best_cost,
        best_solution))
        return best_solution, best_cost

class _Ant(object):

```

```

def __init__(self, aco: ACO, graph: Graph):
    self.colony = aco
    self.graph = graph
    self.total_cost = 0.0
    self.tabu = [] # tabu list
    self.pheromone_delta = [] # the local increase of pheromone
    self.allowed = [i for i in range(graph.rank)] # nodes which are
allowed for the next selection
    self.eta = [[0 if i == j else 1 / graph.matrix[i][j] for j in
range(graph.rank)] for i in
        range(graph.rank)] # heuristic information
    start = random.randint(0, graph.rank - 1) # start from any node
    self.tabu.append(start)
    self.current = start
    self.allowed.remove(start)

def _select_next(self):
    denominator = 0
    for i in self.allowed:
        denominator += self.graph.pheromone[self.current][i] **
self.colony.alpha * self.eta[self.current][
        i] **
self.colony.beta

# noinspection PyUnusedLocal
probabilities = [0 for i in range(self.graph.rank)] # probabilities
for moving to a node in the next step
for i in range(self.graph.rank):
    try:
        self.allowed.index(i) # test if allowed list contains i
        probabilities[i] = self.graph.pheromone[self.current][i] **
self.colony.alpha * \
        self.eta[self.current][i] ** self.colony.beta / denominator
    except ValueError:
        pass # do nothing

```

```

# select next node by probability roulette
    selected = 0
    rand = random.random()
    for i, probability in enumerate(probabilities):
        rand -= probability
        if rand <= 0:
            selected = i
            break
    self.allowed.remove(selected)
    self.tabu.append(selected)
    self.total_cost += self.graph.matrix[self.current][selected]
    self.current = selected

# noinspection PyUnusedLocal
def _update_pheromone_delta(self):
    self.pheromone_delta = [[0 for j in range(self.graph.rank)] for i in
range(self.graph.rank)]
    for _ in range(1, len(self.tabu)):
        i = self.tabu[_ - 1]
        j = self.tabu[_]
        if self.colony.update_strategy == 1: # ant-quality system
            self.pheromone_delta[i][j] = self.colony.Q
        elif self.colony.update_strategy == 2: # ant-density system

# noinspection PyTypeChecker
            self.pheromone_delta[i][j] = self.colony.Q /
self.graph.matrix[i][j]
        else: # ant-cycle system
            self.pheromone_delta[i][j] = self.colony.Q / self.total_cost

```

plot.py

```
import operator
```

```
import matplotlib.pyplot as plt
```

```
def plot(points, path: list):
```

```
    x = []
```

```
    y = []
```

```
    for point in points:
```

```
        x.append(point[0])
```

```
        y.append(point[1])
```

```
    # noinspection PyUnusedLocal
```

```
    y = list(map(operator.sub, [max(y) for i in range(len(points))], y))
```

```
    plt.plot(x, y, 'co')
```

```
    for _ in range(1, len(path)):
```

```
        i = path[_ - 1]
```

```
        j = path[_]
```

```
    # noinspection PyUnresolvedReferences
```

```
        plt.arrow(x[i], y[i], x[j] - x[i], y[j] - y[i], color='r',
```

```
length_includes_head=True)
```

```
    # noinspection PyTypeChecker
```

```
        plt.xlim(0, max(x) * 1.1)
```

```
    # noinspection PyTypeChecker
```

```
        plt.ylim(0, max(y) * 1.1)
```

```
    plt.show()
```

INPUT DATA:

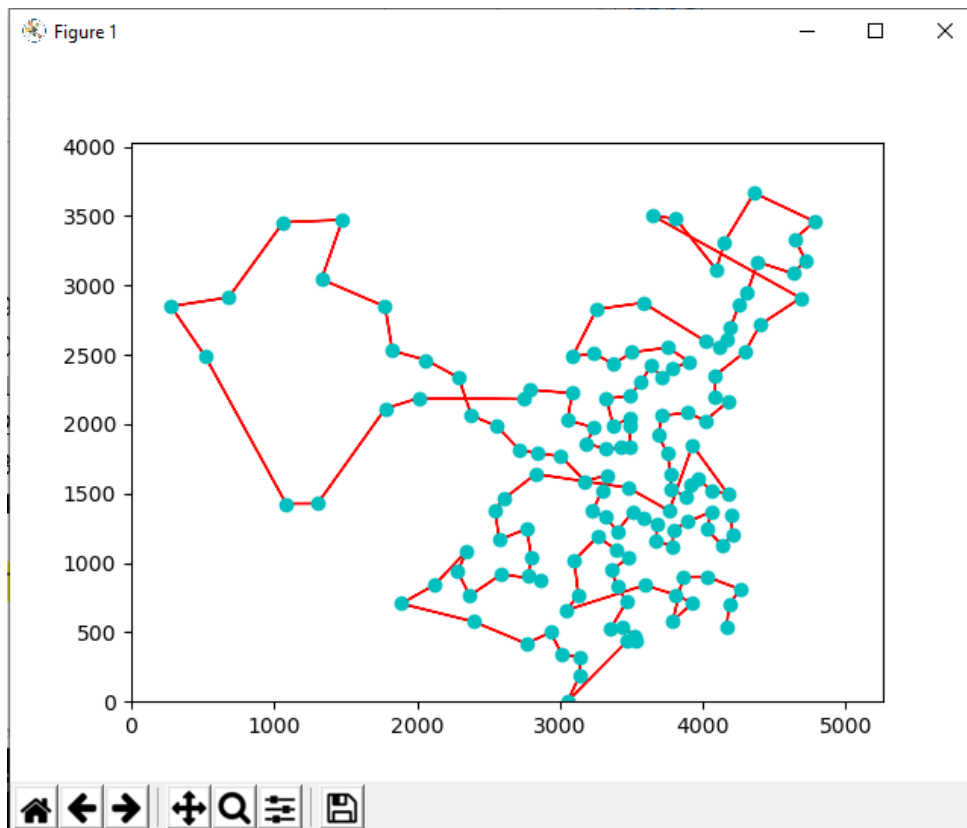
1 3639 1315	40 3143 3421	79 3928 3029	118 3896 1656
2 3569 1438	41 3130 2973	80 4186 3037	119 3918 2179
3 3904 1289	42 2765 3321	81 3322 1916	120 3972 2136
4 3506 1221	43 2545 2357	82 3429 1908	121 4029 2498
5 3237 1764	44 2611 2275	83 3176 2150	122 3766 2364
6 3089 1251	45 2860 2862	84 3229 2367	123 3896 2443
7 3238 1229	46 2801 2700	85 3402 2912	124 3796 2499
8 4172 1125	47 2370 2975	86 3402 2510	125 3478 2705
9 4020 1142	48 1084 2313	87 3468 3018	126 3810 2969
10 4095 626	49 4177 2244	88 3356 3212	127 4167 3206
11 4403 1022	50 3757 1187	89 3044 3081	128 3486 1755
12 4361 73	51 3488 1535	90 3140 3550	129 3334 2107
13 4634 654	52 3374 1750	91 2769 2492	130 3587 2417
14 2846 1951	53 3326 1556	92 2348 2652	131 3507 2376
15 3054 1710	54 3258 911	93 2778 2826	132 3264 2551
16 2562 1756	55 3646 234	94 2126 2896	133 3360 2792
17 2291 1403	56 4089 1387	95 1890 3033	134 3439 3201
18 2012 1552	57 4196 1044	96 3538 3298	135 3526 3263
19 682 825	58 4312 790	97 3712 1399	136 3012 3394
20 518 1251	59 4685 830	98 3493 1696	137 2935 3240
21 1332 695	60 4720 557	99 3791 1339	138 3053 3739
22 4016 1715	61 4153 426	100 3376 1306	139 2284 2803
23 4087 1546	62 2831 2099	101 3188 1881	140 2577 2574
24 4062 2220	63 3086 1516	102 3814 261	141 2592 2820
25 4061 2370	64 2716 1924	103 3583 864	142 2401 3164
26 4201 2397	65 2751 1559	104 4297 1218	143 1304 2312
27 3777 2095	66 1779 1626	105 4116 1187	144 3470 3304
28 3888 2261	67 1478 267	106 4252 882	
29 3678 2463	68 278 890	107 4386 570	
30 3789 2620	69 3715 1678	108 4643 404	
31 3862 2839	70 4181 1574	109 4784 279	
32 4263 2931	71 3929 1892	110 3007 1970	
33 3492 1901	72 3751 1945	111 1828 1210	
34 3479 2198	73 4207 2533	112 2061 1277	
35 3318 2408	74 4139 2615	113 2788 1491	
36 3296 2217	75 3780 2212	114 2381 1676	
37 3394 2643	76 3594 2900	115 1777 892	
38 3101 2721	77 3676 2578	116 1064 284	

39 3792 3156	78 4029 2838	117 3688 1818	
--------------	--------------	---------------	--

OUTPUTS AND SCREENSHOTS:

```
C:\WINDOWS\py.exe
cost: 34687.79240456095, path: [78, 125, 30, 77, 79, 31, 126, 38, 134, 95, 143, 133, 87, 86, 84, 132, 36, 124, 85, 34, 8
3, 35, 128, 82, 109, 13, 63, 61, 43, 42, 139, 90, 45, 92, 44, 140, 46, 141, 41, 136, 135, 39, 89, 137, 40, 88, 37, 131,
33, 130, 129, 28, 76, 29, 123, 122, 121, 74, 27, 118, 119, 23, 48, 25, 24, 120, 73, 72, 70, 71, 26, 116, 68, 117, 21, 22
, 69, 55, 104, 7, 56, 105, 57, 106, 12, 59, 107, 108, 11, 60, 9, 101, 54, 102, 53, 6, 5, 62, 14, 4, 100, 80, 81, 32, 127
, 97, 51, 52, 50, 1, 0, 96, 98, 2, 49, 8, 103, 10, 58, 3, 99, 112, 64, 15, 113, 16, 17, 65, 110, 111, 114, 20, 66, 115,
18, 67, 19, 47, 142, 94, 93, 138, 91, 75]
```

GRAPH DEPICTING THE PATH TAKEN:

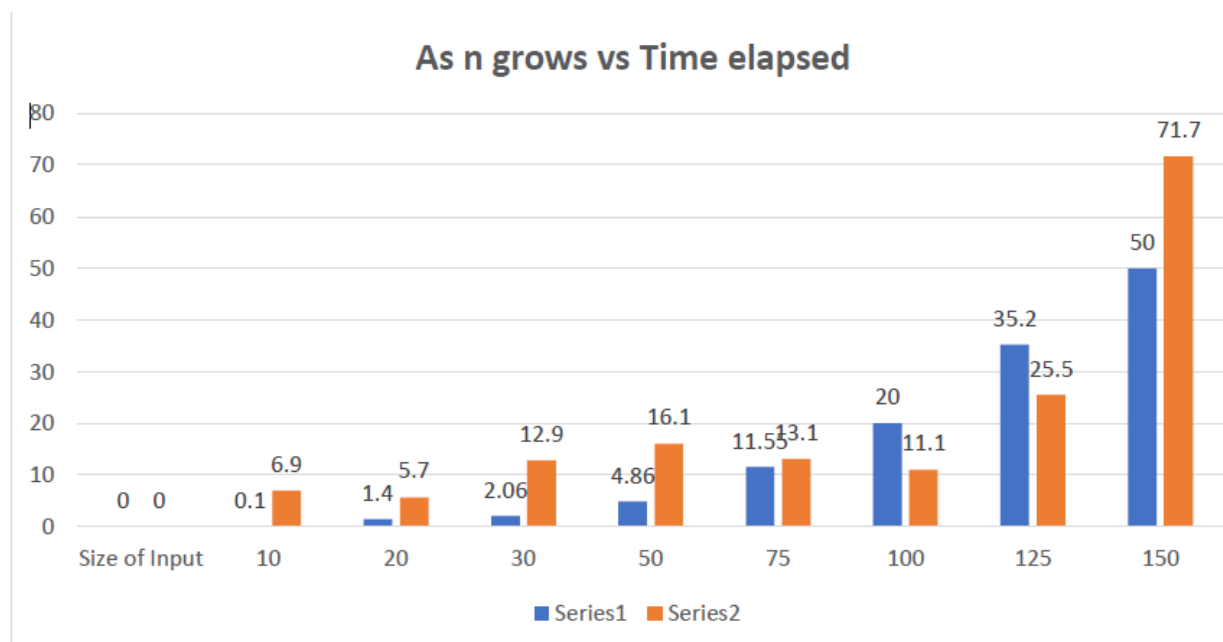


Comparison of Results from ACO and Brute Force Techniques :

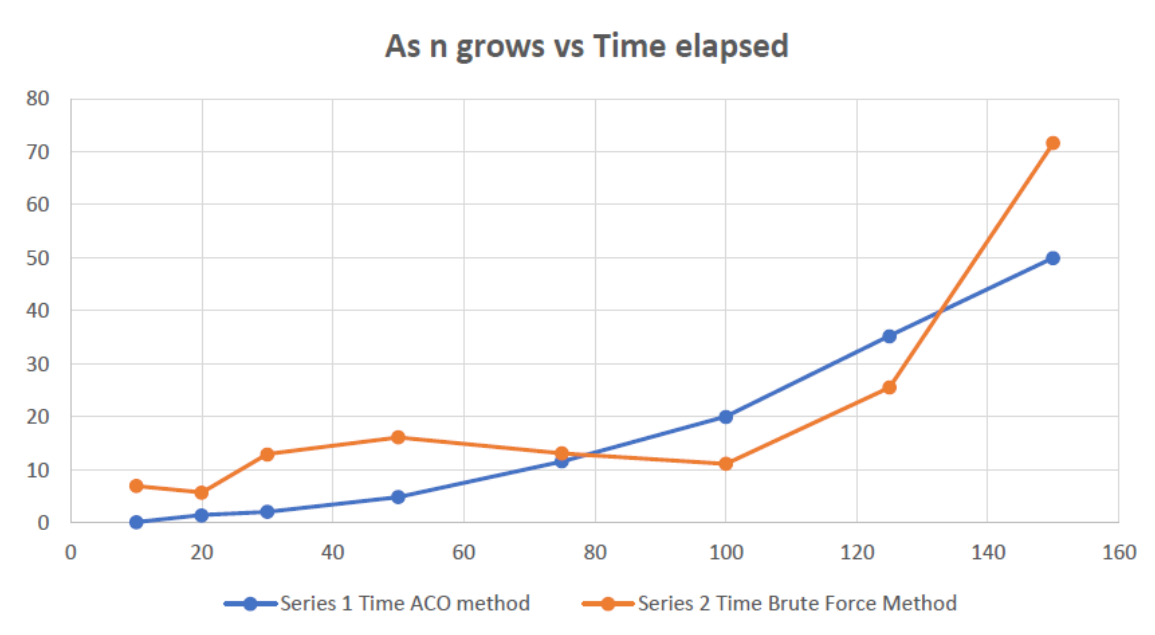
Table:

Size of Input(As n grows)	Time Elapsed in ACO method (Series 1)	Time Elapsed in Brute force method(Series 2)
10	0.1	6.9
20	1.4	5.7
30	2.06	12.9
50	4.86	16.1
75	11.55	13.1
100	20	11.1
125	35.2	25.5
150	50	71.7

Bar Graph:



Line Graph:



Conclusion:-

As shown above through both theory as well as our programming, we as a team have come to the conclusion that our problem statement of the Travelling Salesman Problem, is executed more efficiently through the Ant Colony Optimization Technique as compared to the brute force technique. Although it necessary to highlight that we have used the ACO for the input data that include 144 cities. The ACO performs efficiently at this state due to the large number of cities or “n” (as discussed before in the project review). Had it been for a comparatively smaller number of cities brute force has shown to be significantly more effective than the ACO. Our findings are also in accordance with the years of research that have gone behind this meta-heuristic technique. We would sincerely like thank our Professor Raghavan R. who helped us in our project, every researcher whose hard work has gone into the data that fuelled this project and the everlasting and undying support that comes from our parents and peers.

Bibliography (References):-

- <https://stackoverflow.com/questions/5003400/travelling-salesman-problem>
- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- <https://www.sciencedirect.com/science/article/pii/S1002007108002736>
- https://en.wikipedia.org/wiki/Brute-force_search#:~:text=The%20main%20disadvantage%20of%20the,be%20the%20given%20number%20n.
- <https://en.wikipedia.org/wiki/Metaheuristic#:~:text=In%20computer%20science%20and%20mathematical,information%20or%20limited%20computation%20capacity.>
- <https://www.dummies.com/programming/big-data/algorithms-computing-costs-following-heuristics/>
- https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms
- https://www.researchgate.net/figure/Advantages-and-disadvantages-of-Ant-Colony-Optimization-ACO_tbl2_317296448

Base Papers:-

- An ant colony optimization method for generalized TSP problem
Jinhui Yang a, Xiaohu Shi a, Maurizio Marchese b, Yanchun Liang a,*
- Ant colony optimization for real-world vehicle routing problems
From theory to applications
A.E. Rizzoli, · R. Montemanni · E. Lucibello · L.M. Gambardella

Book Referred:-

- Introduction to Algorithms
By:- Thomas H. Cormen ,Charles Leiserson, Ron Rivest, and Cliff Stein

!! THANK YOU !!