高等演算法期末報告

題目: Travelling Salesman problem

使用軟體: Python

姓名:葉冠宏

學號:108753208

指導教授: 郭桐惟

一. Problem description

Goal:

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

二. The method

2-1 method1: Christofides algorithm(3/2 approximation algorithm)

In computer science and operations research, approximation algorithms are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with provable guarantees on the distance of the returned solution to the optimal one.

Travelling salesman problem is a NP-hard problem, so we use the approximation algorithm to approximate the optimal solution when the samples are large.

Demonstrate how they are applied to this problem

Step1: Find a minimum spanning tree T

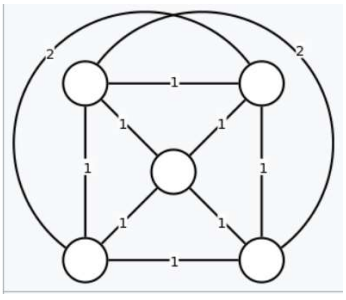
Step2: Let O be the set of vertices with odd degree in T. Find a minimum-cost perfect matching M

Step3:Add the set of edges of M to T. Find a Eulerian tour.

Step4:Shortcut the Eulerian tour to make a Hamiltonian Cycle.

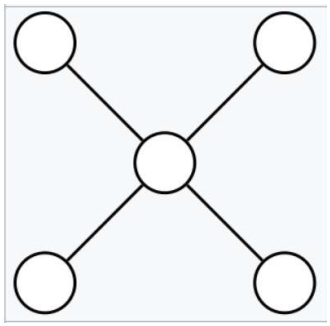
Example:

Step1:



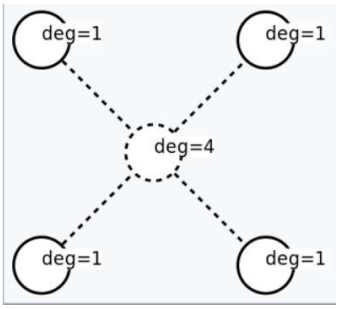
Given: complete graph whose edge weights obey the triangle inequality

Step2:



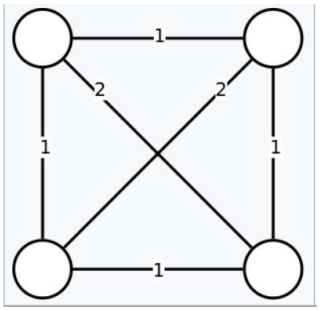
Calculate minimum spanning tree T

Step3:



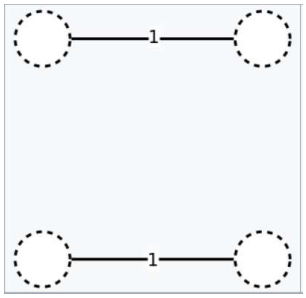
Calculate the set of vertices O with odd degree in T

Step4:



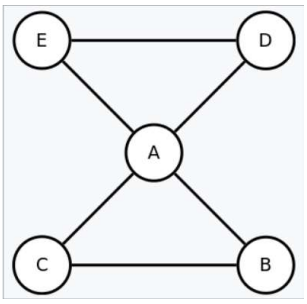
Form the subgraph of G using only the vertices of O

Step5:



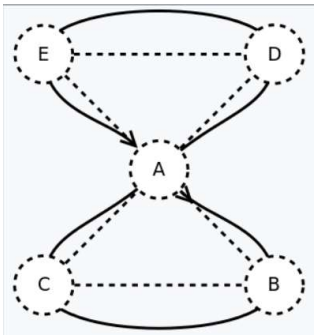
Construct a minimum-weight perfect matching M in this subgraph

Step6:



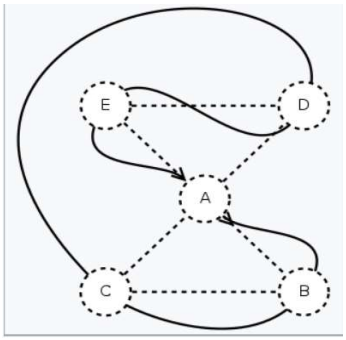
Unite matching and spanning tree T ∪ M to form an Eulerian multigraph

Step7:



Calculate Euler tour

Step8:



Remove repeated vertices, giving the algorithm's output

2-2 method2: Brute force

Demonstrate how they are applied to this problem

1.Generate all (n-1)! Permutations of cities.

2.Calculate the cost of every permutation and keep track of the minimum cost permutation.

3.Return the permutation with minimum cost.

2-3 method3: Simulated Annealing

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete (e.g., the traveling salesman problem). For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to exact algorithms such as gradient descent or branch and bound.

Demonstrate how they are applied to this problem

- Let $s = s_0$
- For $k = 0$ through k_{max} (exclusive):
  - $T \leftarrow$ temperature($(k+1)/k_{max}$)
  - Pick a random neighbour, $s_{new} \leftarrow$ neighbour(s)
  - If $P(E(s), E(s_{new}), T) \geq$ random(0, 1):
    - $s \leftarrow s_{new}$
- Output: the final state s

1. generate a random tour for Hamilton cycle

2.set temperature as (number of city)^2, epoch as 1000

3.if temperature is less than the bench mark, return the result obtained so far.

4.Otherwise, find the neighbouring path by randomly selecting 2 cities and interchanging the 2 cities only.

5.determine the delta cost by calculating the difference of neighbouring path and the current path.

6.if sigmoid(delta_cost/temperatur) is larger than the benchmark, then replace the current tour with the neighbouring tour. And if the cost is better than the best tour cost obtained before, then update the best tour with the tour. Also, the temperature will cool down as time keeps increasing.

The figures below show the implementation of this algorithm:

```
tour = random_tour(num_of_cities)
best_tour = tour
temperature = num_of_cities ** 2
num_of_epochs = 1000
for time in range(num_of_epochs):
    for _ in range(1000):
        if temperature < 0.1e-100:
            return best_tour, cost(num_of_cities, distance_matrix, best_tour)

        neighbour = random_neighbour(num_of_cities, tour)
        delta_cost = cost(num_of_cities, distance_matrix, neighbour) - cost(num_of_cities, distance_
        if random.uniform(0, 1) < sigmoid(delta_cost / temperature):
            tour = neighbour
            if cost(num_of_cities, distance_matrix, tour) < cost(num_of_cities, distance_matrix, bes
                best_tour = tour
            temperature = cooling(temperature, time)

return best_tour, cost(num_of_cities, distance_matrix, best_tour)
```

```
def random_neighbour(num_of_cities, tour):
    """
    Two city exchange
    """
    new_tour = tour.copy()
    random_city = random.sample(range(num_of_cities), 2)
    new_tour[random_city[0]], new_tour[random_city[1]] = tour[random_city[1]], tour[random_city[0]]
    return new_tour
```

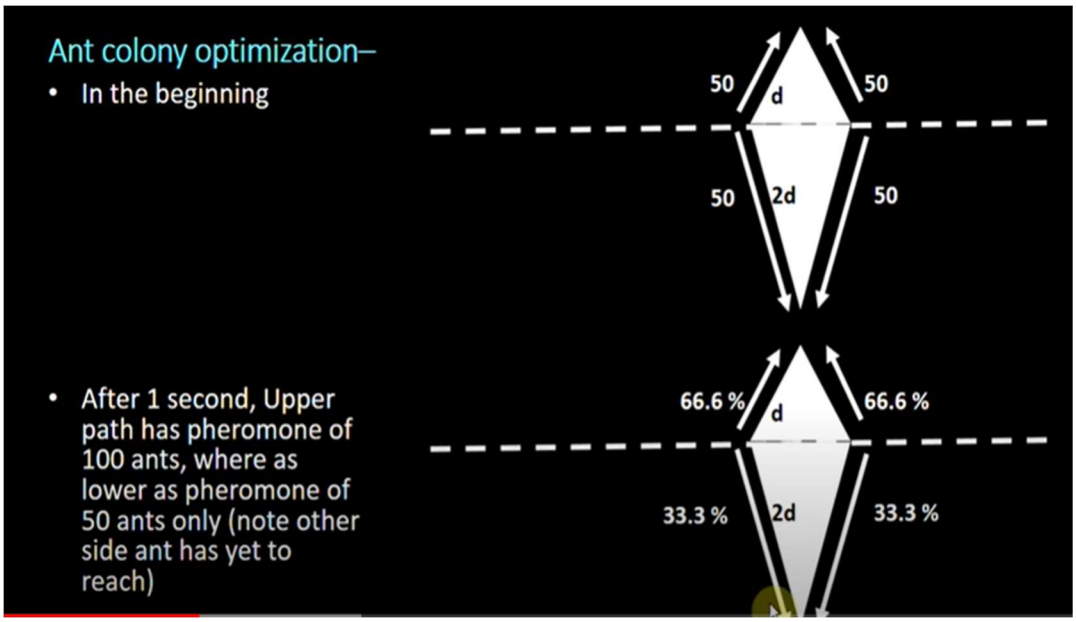
```
def sigmoid(x: float):
    try:
        return 1 / (1 + math.exp(x))
    except OverflowError:
        return 0.99


def cooling(temperature, time):
    return temperature / (time + 1)
```

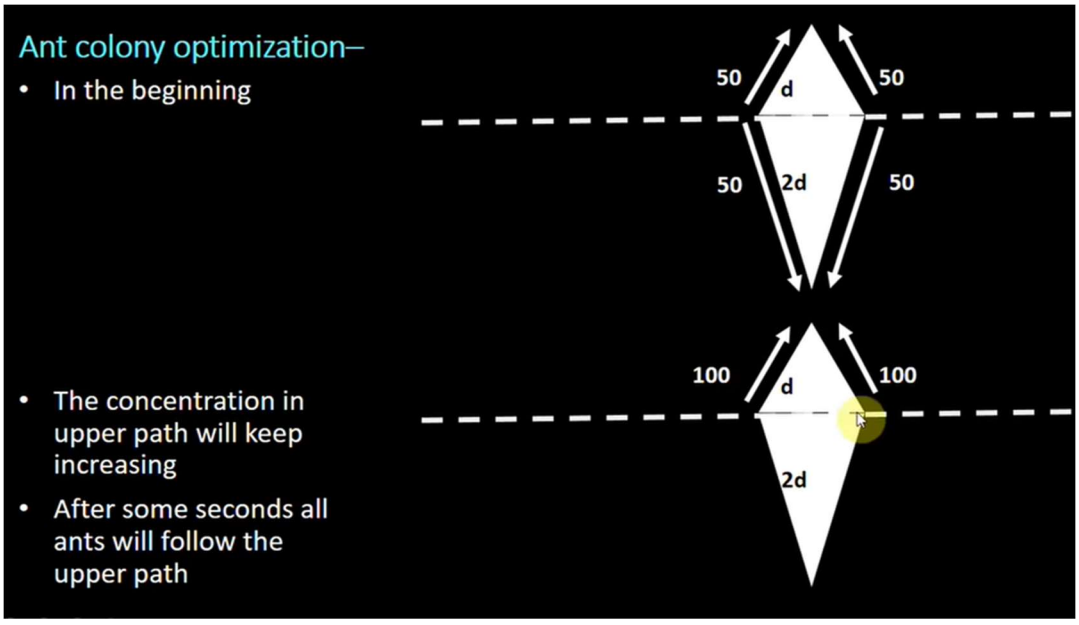
2-4 method4: Ant Colony Optimization Algorithm

In computer science and operations research, the ant colony optimization algorithm (ACO) is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs. Artificial ants stand for multi-agent methods inspired by the behavior of real ants. The pheromone-based communication of biological ants is often the predominant paradigm used. Combinations of artificial ants and local search algorithms have become a method of choice for numerous optimization tasks involving some sort of graph, e.g., vehicle routing and internet routing.

As an example, ant colony optimization is a class of optimization algorithms modeled on the actions of an ant colony. Artificial 'ants' (e.g. simulation agents) locate optimal solutions by moving through a parameter space representing all possible solutions. Real ants lay down pheromones directing each other to resources while exploring their environment. The simulated 'ants' similarly record their positions and the quality of their solutions, so that in later simulation iterations more ants locate better solutions. One variation on this approach is the bees algorithm, which is more analogous to the foraging patterns of the honey bee, another social insect.

Demonstrate how they are applied to this problem





Ant colony optimization for TSP–

Travelling sales man problem

- Sales man has to start from one place
- Go to all other city just once (A)
- And come back to original city
- TSP is all about finding the least cost (distance) path with above conditions (B)

Ant colony algorithm for TSP

- If there are n cities, then l distinct ant's start from any of these n cities, randomly
- These ants have two distinctive advantage over real ants
  - ✓ They have memory – not to visit the cities, they have visited (condition A)
  - ✓ They know the distance of the cities and tend to choose nearby city (if every thing same)
  - ✓ And if distance of two paths are same, they tend to choose path with more pheromone (just like real ant)
- There probability (P_{ij}^{k}) to select city j by an ant k, sitting at city i is given by

Ant colony optimization for TSP–

- There probability (Pjk) to select city j by an ant k, sitting at city i is given by

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

- 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 $i = 1/d_{ij}$ (d_{ij} is the distance between city i and j
- β the parameter to regulate the influence of η_{ij}
- $Allowed_k$ the set of cities that have not been visited by k yet

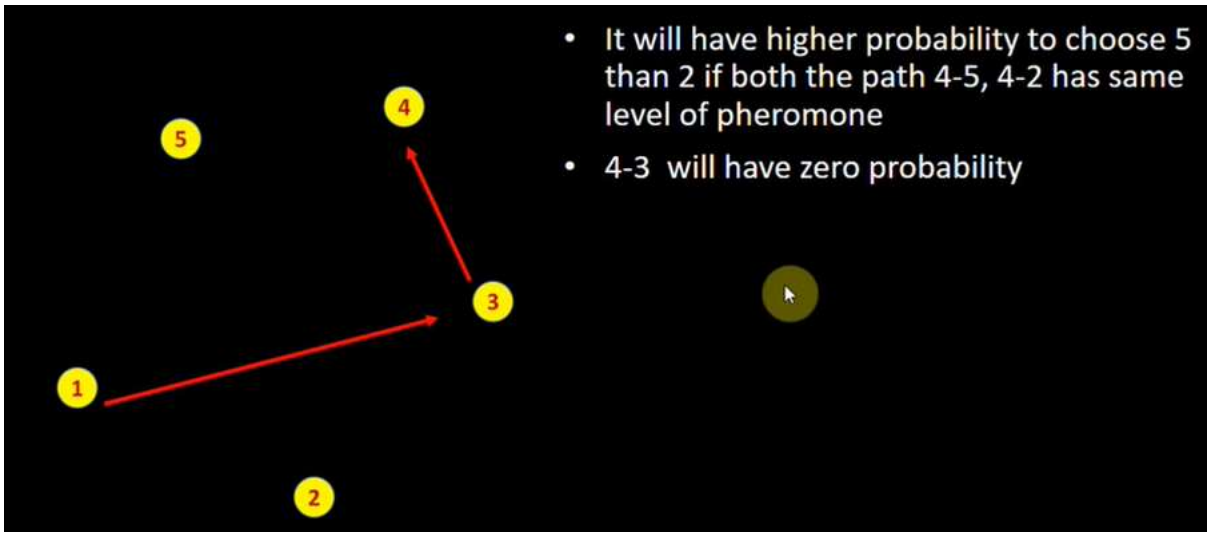
Ant colony optimization–

- After each ant completes n iteration, the complete tour is obtained
- The pheromones are updated in such a way that shorter path gets more pheromone than longer path
- The pheromone update formula is given by
- Where L_k the length of its tour,
- t is the iteration counter
- $\rho \in [0, 1]$ is the evaporation factor
  - ✓ It regulated the reduction of pheromone
- $\Delta\tau_{ij}$ the total increase of trail level on edge (i, j) by all ants
- $\Delta\tau_{ij}^{k}$ the increase of trail level on edge (i, j) caused by ant k,

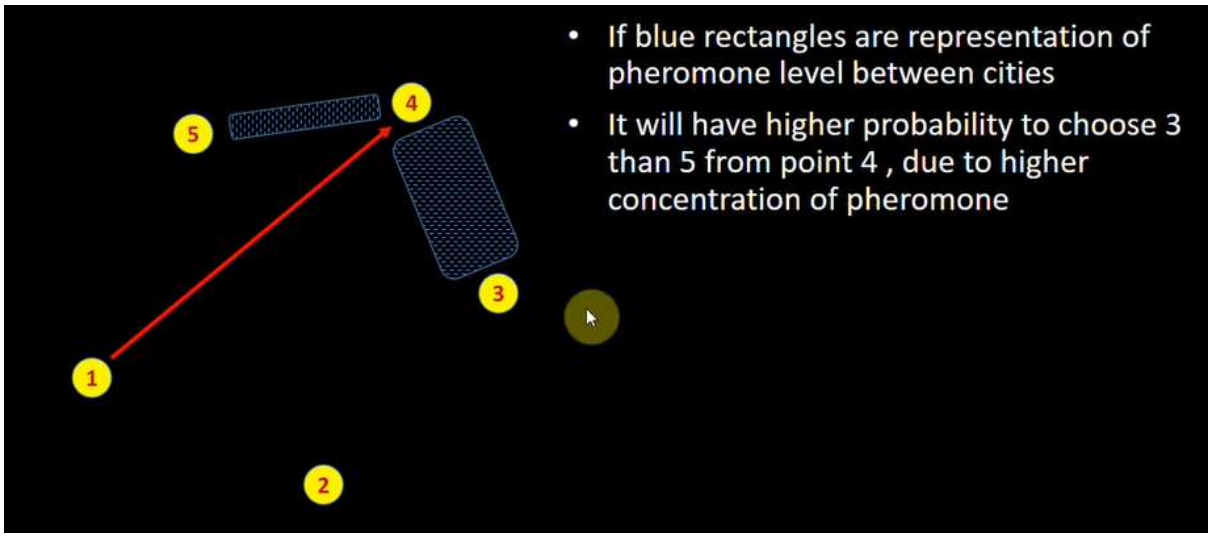
$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

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

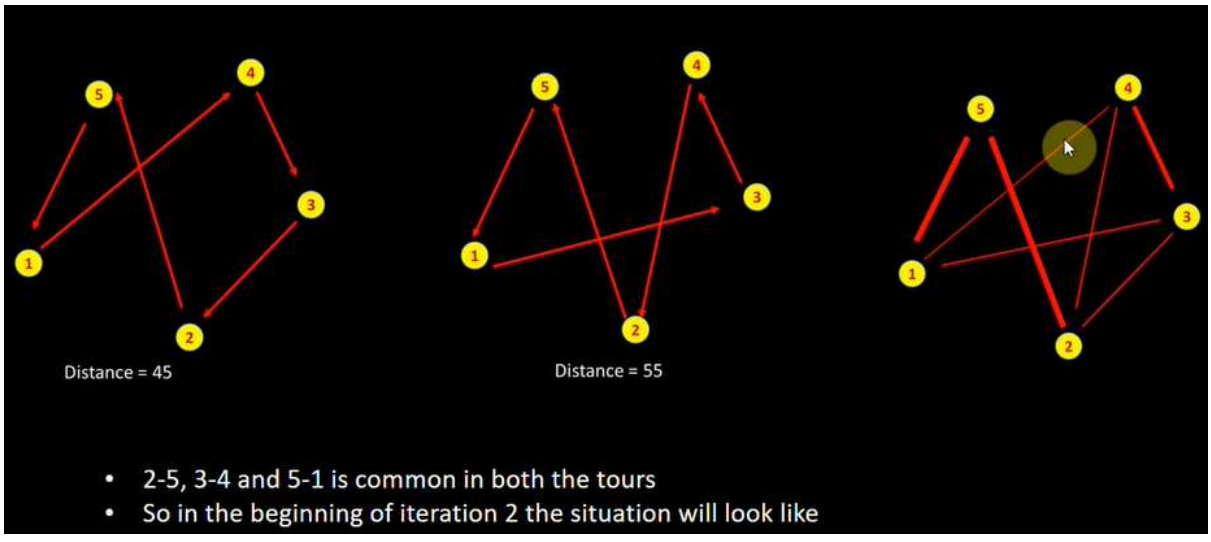
$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i,j) \\ 0 & \text{otherwise} \end{cases}$$



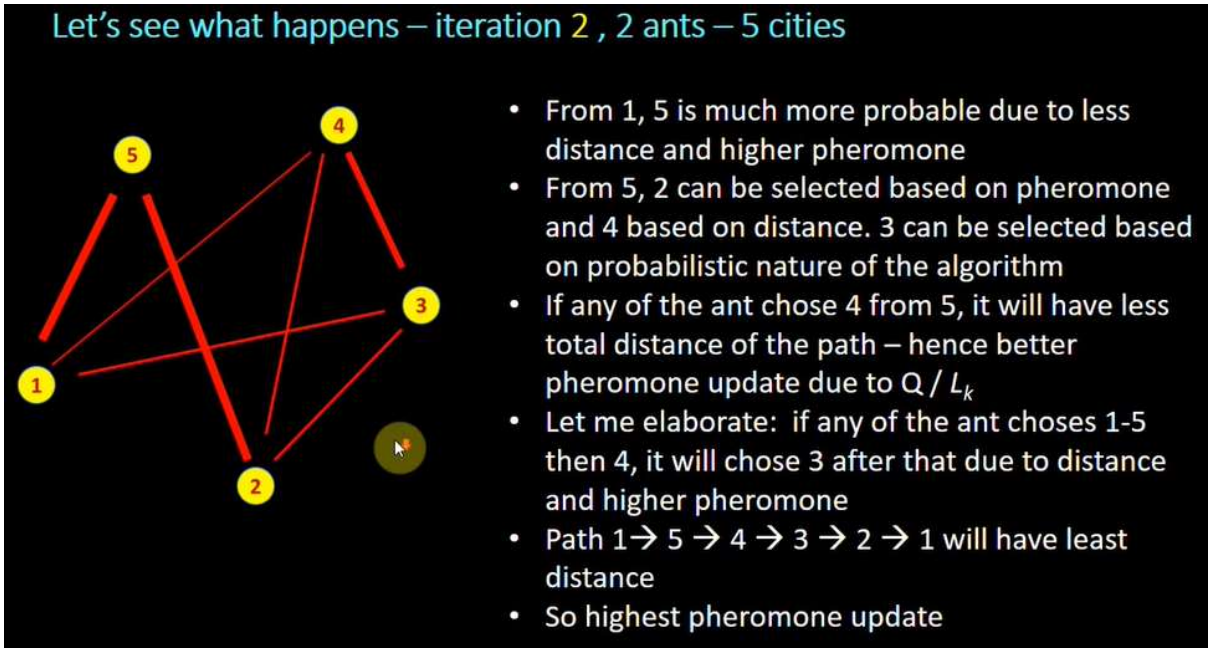
- It will have higher probability to choose 5 than 2 if both the path 4-5, 4-2 has same level of pheromone
- 4-3 will have zero probability

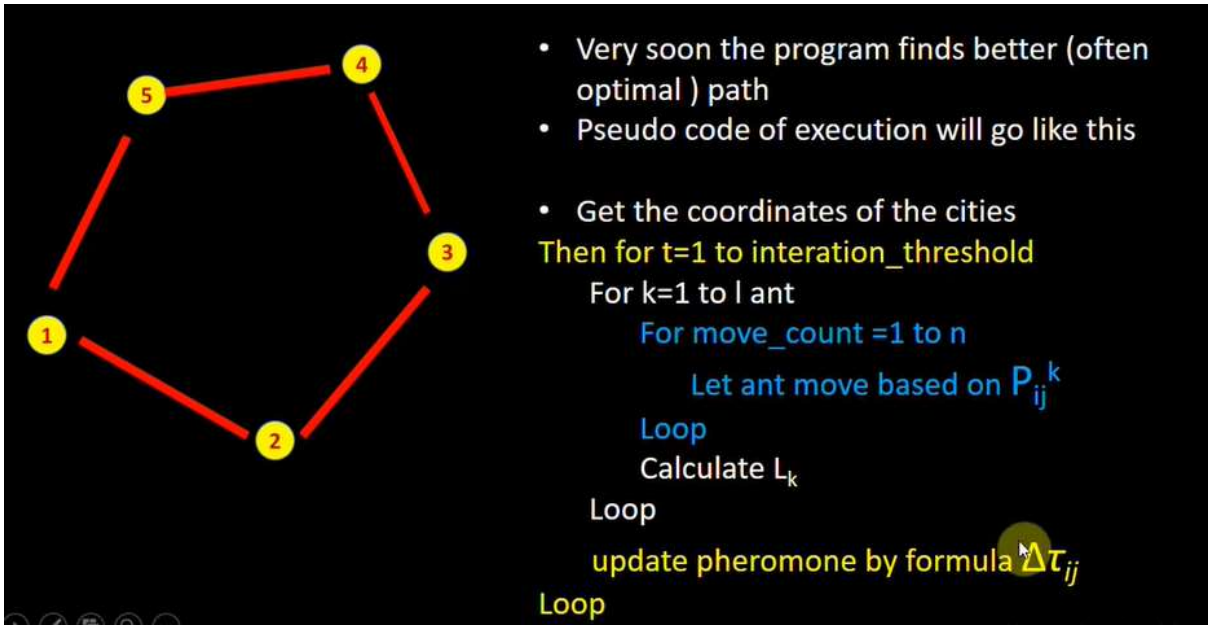


- If blue rectangles are representation of pheromone level between cities
- It will have higher probability to choose 3 than 5 from point 4 , due to higher concentration of pheromone



- 2-5, 3-4 and 5-1 is common in both the tours
- So in the beginning of iteration 2 the situation will look like





三. Experiment

Input data

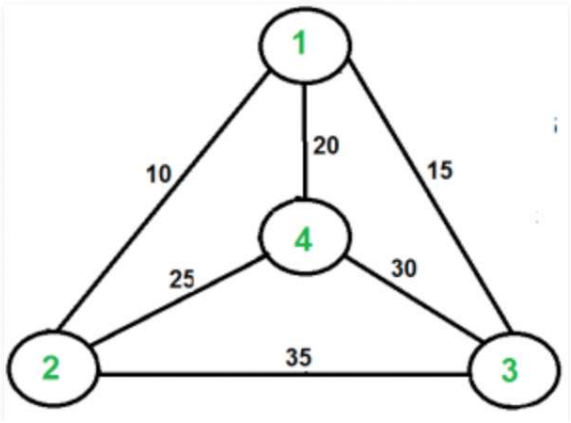
We experiment 5~10 cities for each of the algorithm. We first randomly generate the cities, calculate their distances, and then transform them into adjacency matrix.

```python
def transformcoord(coord):
    V=len(coord)

    graph=[]

    for i in range(V):
        newdis=[]
        mycoo=coord[i]

        for j in range(V):
            tar=coord[j]
            dis=((mycoo[0] - tar[0]) ** 2 + (mycoo[1] - tar[1]) ** 2) ** (1.0 / 2.0)
            newdis.append(int(dis))

        graph.append(newdis)

    return graph
```

```python
def generate(num):
    matr=[]
    random.seed()

    for i in range(num):
        temp=[]

        x=random.randint(-50, 50)
        y=random.randint(-50, 50)
        temp.append(x)
        temp.append(y)

        matr.append(temp)


    return matr
```

Experimental results

Example1: (optimal solution=10+25+30+15=80)



Christofides algorithm:

```
Path for the problem: [0, 1, 3, 2, 0]
Traveling cost: 80
Running time: 0.0059685707092285 16
```

Brute force:

```
Path for the problem: [0, 1, 3, 2, 0]
Traveling cost: 80
Running time: 0.0
```

Simulated annealing:

```
Path for the problem: [1, 0, 2, 3, 1]
Traveling cost: 80
Running time: 0.010998725891113281
```

Ant Colony Optimization:

```
Path for the problem: [0, 1, 3, 2, 0]
Traveling cost: 80
Running time: 0.07703661918640137
```

Example2:

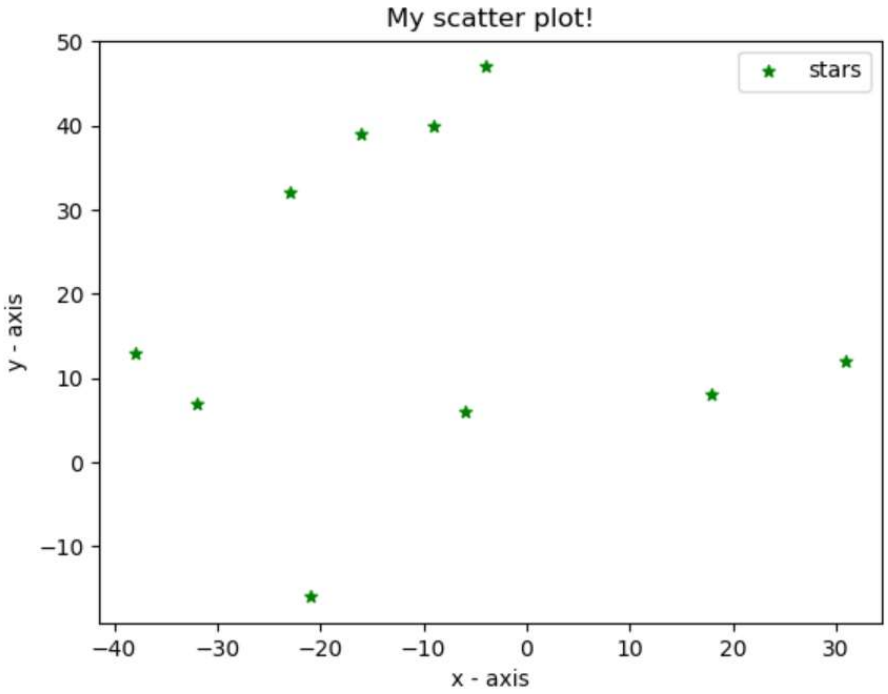
We generate 10 cities, and their coordinates are as follows:

[[-6, 6], [31, 12], [-38, 13], [18, 8], [-23, 32], [-32, 7], [-4, 47], [-21, -16], [-9, 40], [-16, 39]]

Their adjacency matrix is as follows:

[[0, 37, 32, 24, 31, 26, 41, 26, 34, 34], [37, 0, 69, 13, 57, 63, 49, 59, 48, 54], [32, 69, 0, 56, 24, 8, 48, 33, 39, 34], [24, 13, 56, 0, 47, 50, 44, 45, 41, 46], [31, 57, 24, 47, 0, 26, 24, 48, 16, 9], [26, 63, 8, 50, 26, 0, 48, 25, 40, 35], [41, 49, 48, 44, 24,

48, 0, 65, 8, 14], [26, 59, 33, 45, 48, 25, 65, 0, 57, 55], [34, 48, 39, 41, 16, 40, 8, 57, 0, 7], [34, 54, 34, 46, 9, 35, 14, 55, 7, 0]]



Christofides algorithm:

```
Path for the problem: [5, 0, 3, 1, 6, 8, 9, 4, 2, 7, 5]
Traveling cost: 218
Running time: 0.007997989654541016
```

Brute force:

```
Path for the problem: [0, 3, 1, 6, 8, 9, 4, 2, 5, 7, 0]
Traveling cost: 193
Running time: 0.253995418548584
```

Simulated annealing:

```
Path for the problem: [2, 7, 0, 3, 1, 6, 8, 9, 4, 5, 2]
Traveling cost: 203
Running time: 0.012999296188354492
```

Ant Colony Optimization:

```
Path for the problem: [9, 8, 6, 2, 5, 4, 7, 0, 1, 3, 9]
Traveling cost: 267
Running time: 0.17250084877014l6
```

Statistics:

Christofides algorithm:

Christofides	time	cost	approximation ratio
sample5	0	267	1
sample6	0.001	267	1.038910506
sample7	0.0016	288	1.103448276
sample8	0.001	234	1
sample9	0.0012	232	1.004329004
sample10	0.0156	218	1.129533679
sample15	0.0019	338	

We discover that the approximation ratio of Christofides algorithm is about 1.1 when city is below 10, which is really good. Besides, the result of each trial is the same. And the running time of this algorithm is not too much. In fact, it can be run quickly even when the city size is large.

Brute force:

Bruteforce	time	cost
sample5	0	267
sample6	0	257
sample7	0.0009	261
sample8	0.00366	234
sample9	0.035	231
sample10	0.262	193
sample15	>3hr	

The result of the brute force algorithm is the ground truth. It can be run really fast when the city size is below 10. Yet, the running time grows exponentially when it exceeds 10.

Simulated annealing:

Simulated annealing	time	cost	approximation ratio
sample5	6.388	267	
	6.522	267	
	0.186	267	
avg	4.365333	267	1
sample6	0.0482	257	
	0.0129	257	
	0.01	257	
avg	0.0237	257	1
sample7	3.48077	261	
	7.4414	261	
	0.0649	261	
avg	3.662357	261	1
sample8	0.011	234	
	0.0189	234	
	0.0109	234	
avg	0.0136	234	1
sample9	0.0119	232	
	0.1605	238	
	0.01199	254	
avg	0.061463	241.3333333	1.044733045
sample10	0.01	215	
	0.012	234	
	0.0129	223	
avg	0.011633	224	1.160621762
sample15	0.014	479	

The result of simulated annealing algorithm is pretty consistent and has a great approximation ratio when the city size is below 9. But it varies slightly among each trial when the city size becomes larger given the same input. The reason is that it uses randomized factor in the algorithm.

Besides, the running time may be large depending on the data and the starting point of the tour. Overall, it has an approximation ratio around 1.1 and it increases as the sample size increases.

Ant Colony Optimization:

Ant colony	time	cost	approximation ratio
sample5	0.089	318	
	0.0889	339	
	0.0869	321	
avg	0.088267	326	**1.220973783**
sample6	0.105	257	
	0.1029	364	
	0.105	339	
avg	0.1043	320	1.245136187
sample7	0.121	307	
	0.12	328	
	0.117	302	
avg	0.119333	312.3333333	1.196679438
sample8	0.143	421	
	0.1396	359	
	0.137	267	
avg	0.139867	349	1.491452991

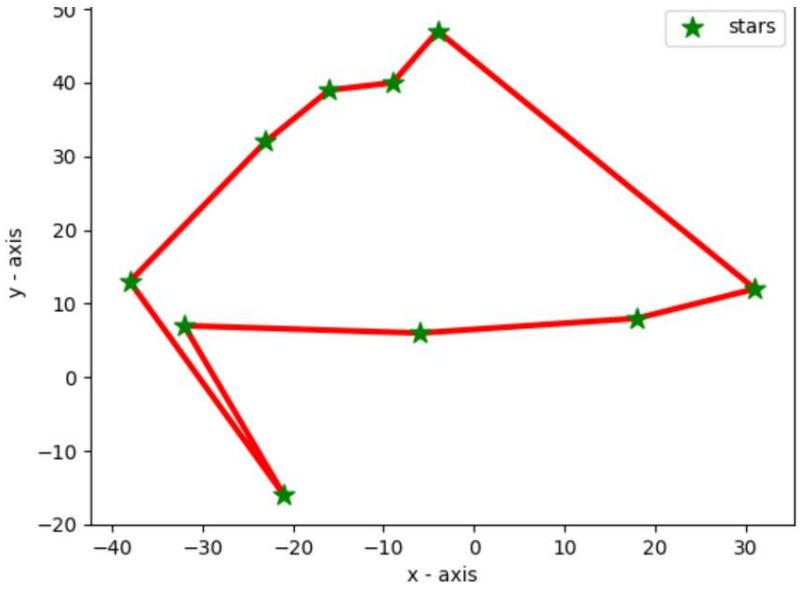
sample9	0.1579	411	
	0.2439	325	
	0.163	435	
avg	0.188267	390.3333333	1.68975469
sample10	0.232	338	
	0.1749	344	
	0.1729	259	
avg	0.193267	313.6666667	1.625215889
sample15	0.2729	446	

Ant colony optimization algorithm has a higher approximation ratio on average. Besides, the result for each trial is unstable. It may vary a lot even when the sample size is low. Overall, it has an approximation ratio around 1.5. And it can be run really fast even when the sample size is large.

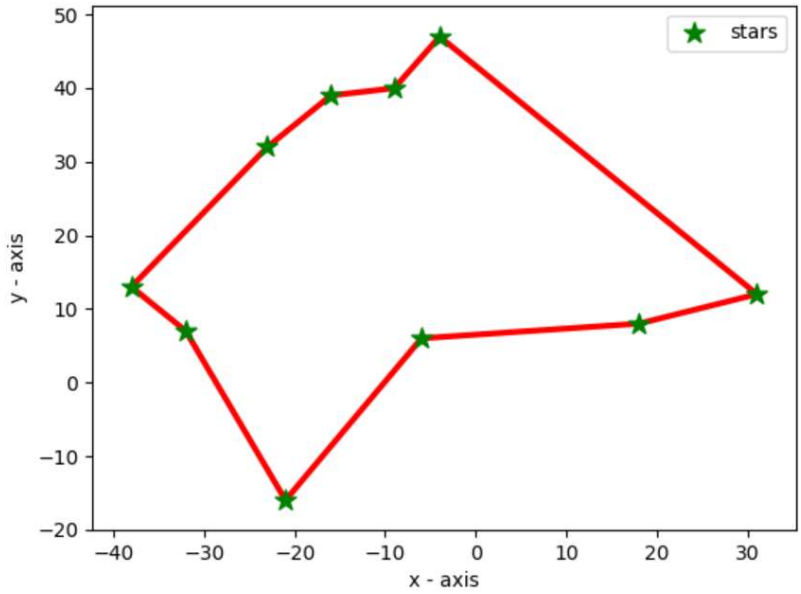
Graph representation of the results:

Example2:

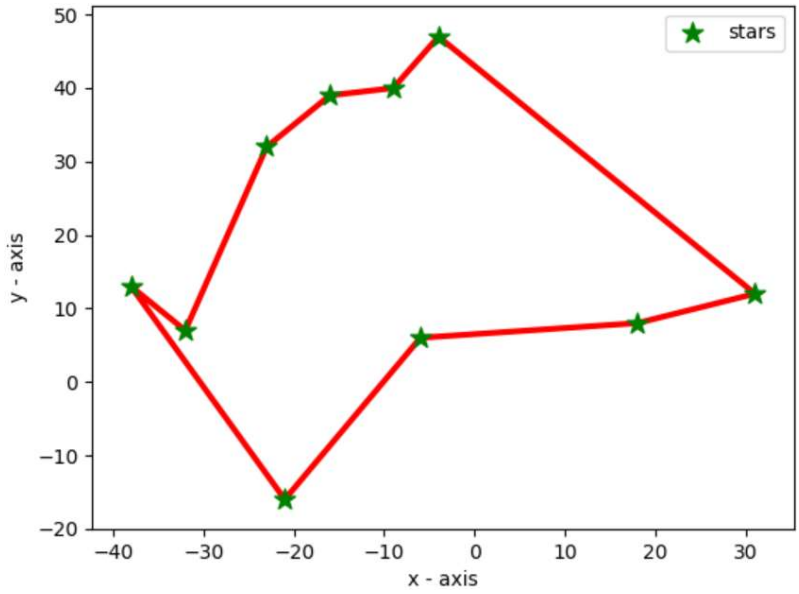
Christofides algorithm



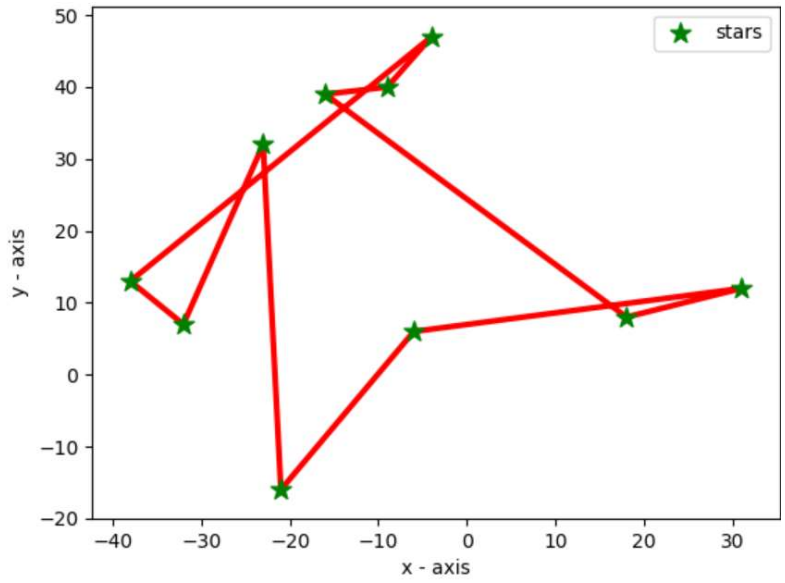
Brute force



Simulated annealing



Ant Colony Optimization



四. Conclusion

In our experiment, Brute force is the ground truth approach. Our reason to use approximation algorithm, such as Christofides algorithm, is to approximate the optimal solution because it takes a lot of computational time when the sample is large. By the mathematical proof, it will have at most 3/2 times of the optimal solution. For simulated annealing algorithm and ant colony optimization algorithm, they are metaheuristic approach to approximate the problem. It involves random factor in the algorithm, so we cannot guarantee it will reach near the optimal solution. Yet, we can approximately have a glimpse of the solution.

From the research, we find that brute force approach may not be helpful in the real world since it will take a lot of time to calculate in the case when city size is above 10.

For the Christofides algorithm, it has a great performance in terms of cost and running time.

For simulated annealing algorithm, it is a nice approach to approximate. However, whether we can find a good solution depends on the starting point and the complexity of the data set. It may take some time due to the search for the neighbouring tour.

For ant colony optimization algorithm, it is a way to approximate the solution. Yet, the results for each trial given the same data may be different due to the nature of

the approach. In addition, the performance is worse than Christofides algorithm and simulated annealing algorithm in terms of the cost.

In practice, especially in the era of big data, it is almost impossible to find the real solution to the problem. Although the approach is mathematically feasible, it will take a lot of time to compute. Hence, approximation method is needed for us to gauge the upper bound and the range of the real solution.

五.Reference

https://en.wikipedia.org/wiki/Travelling_salesman_problem

https://en.wikipedia.org/wiki/Python_(programming_language)

https://en.wikipedia.org/wiki/Approximation_algorithm

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

https://en.wikipedia.org/wiki/Triangle_inequality

https://en.wikipedia.org/wiki/Matching_(graph_theory)

https://en.wikipedia.org/wiki/Factorial

https://en.wikipedia.org/wiki/Sigmoid_function

https://en.wikipedia.org/wiki/Simulated_annealing

https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

https://www.youtube.com/watch?v=C86j1AoMRr0

https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

https://github.com/Retsediv/ChristofidesAlgorithm/blob/master/christofides.py

https://en.wikipedia.org/wiki/Christofides_algorithm

Appendix

To run the code:

Christofides algorithm: please run “christo.py”

Simulated annealing: please run “mainsimu.py”

Brute force: please run “bruteforce.py”

Ant Colony Optimization: please run “mainant.py”