Bee Colony Optimization on TSP problem

Author: Yifei Chen (sc22yc)

Project name: A Comparative Study of Bionic Algorithms for Tackling the Traveling Salesman Problem
Other group member for other part of the main project: Yujun Wang (sc223yw), Kunyu Jiang (sc22kj2), Fengyun Wang (ml18f2w)

You do not have to read the full document, just read the part you want to check, I know it is a bit too long. Then menu has provided below.
For use the model go to page10 on the Main model.

This is a part of the project “A Comparative Study of Bionic Algorithms for Tackling the Traveling Salesman Problem”
This is a model based on Bee colony optimization algorithm which was inspired by some of the behavior of bees, and it is specialized to solve the TSP(Travelling salesman problem) without going back to the starting point.

Menu

Environmental requirements:

python3.0+

Content:

just run the only .py file in a IDE (VScode and Thonny are tested)

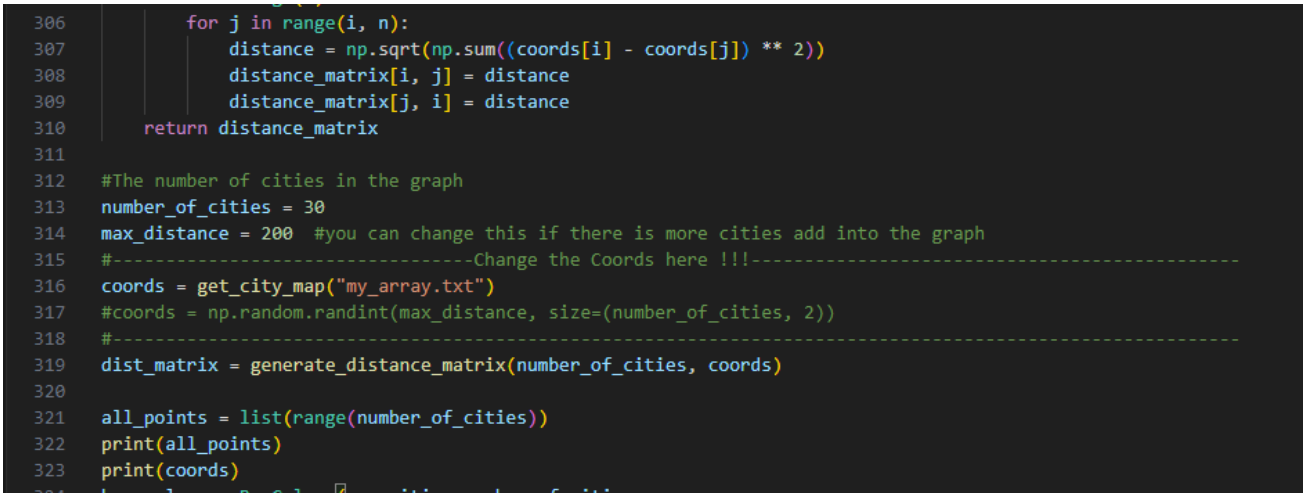
Uses:

the code used following library, you might want to install matplotlib if it has not been installed yet
import numpy
import random
import matplotlib.pyplot

Important Notice:

1. It runs the map store in the txt file by default. If you want to get a random map just go to line 316 to change use the random cords that has been comment out. Looks like following:

```
#-----------------Change the Coords here !!!-------------------------
coords = get_city_map("my_array.txt")
#coords = np.random.randint(max_distance, size=(number_of_cities, 2))
#--------------------------------------------------------------------
```



```
306             for j in range(i, n):
307                 distance = np.sqrt(np.sum((coords[i] - coords[j]) ** 2))
308                 distance_matrix[i, j] = distance
309                 distance_matrix[j, i] = distance
310         return distance_matrix
311
312     #The number of cities in the graph
313     number_of_cities = 30
314     max_distance = 200  #you can change this if there is more cities add into the graph
315     #--------------------------------Change the Coords here !!!-----------------------------------------
316     coords = get_city_map("my_array.txt")
317     #coords = np.random.randint(max_distance, size=(number_of_cities, 2))
318     #---------------------------------------------------------------------------------------------------
319     dist_matrix = generate_distance_matrix(number_of_cities, coords)
320
321     all_points = list(range(number_of_cities))
322     print(all_points)
323     print(coords)
```

```
#The number of cities in the graph
number_of_cities = 30
max_distance = 200  #you can change this if there is more cities add into the graph
#---------------------------------Change the Coords here !!!-----------------------
#coords = get_city_map("my_array.txt")
coords = np.random.randint(max_distance, size=(number_of_cities, 2))
#------------------------------------------------------------------------------------
dist_matrix = generate_distance_matrix(number_of_cities, coords)
```

2. Sometimes this algorithm will fall into a local optimal solution, please restart the hive (rerun this program directly).

3. Sometime when rerun the code it can not display the final fitness graph, please just clear the output, stop and restart the code.

The basic idea of this model is:

1. Initialize a group of worker bees, and assign a random initial path to each worker bee

2. Calculate the fitness value of each worker bee to determine the current optimal solution.

3. After the worker bees return to the nest, they will choose some elite bee to stay and assign random path to other worker bee.

4. send scout bees to conduct a local search greedily on a small part of the paths of some worker bees randomly.

5. After the scout bees return to the nest, they send out the observer bees, and the current part some elite worker bees with the best fitness will directly enter the next generation

6. Onlooker bees use roulette wheel to randomly select bees

7. Onlooker bees use crossover and mutate to explore new paths based on existing paths, and then return to the nest.

8. Repeat steps 2-7 until the predetermined number of hive iterations is reached.

Code Structure:

There is a class for individual bee, curr_path represent their current path and compute path distance method use for calculate the current distance of the bee

Bee:

```
class Bee:
    def __init__(self, curr_path):
    def compute_path_distance(self, path):
```

BeeColony:

After that is Bee colony class include all the function needed for three types of bees to run:
Here is all the method of BeeColony, I'm going to explain each function in the following document.

```
class BeeColony:
    def __init__(self, num_cities, num_bees, num_epochs, elite_rate,
    local_search_rate,mutate_rate, neighborhood_size,limitation,scout_bee_rate):
    def get_best_path(self):
        #get current best path among all the bees

    def compute_path_distance(self, path):
        #get the distance of one bee

    def get_average_distance(self):
        #get the average distance for all the bees

    def send_employee_bees(self):
        #send the employee bees

    def greedy_algorithm(self, path):
        #greedily find the best path in a given city list

    def send_scout_bee(self):
        #send the scout bee

    def roulette_wheel_selector(self, probs):
        #do a roulette wheel selection on all the bees

    def crossover(self, path1, path2):
        #do a crossover on two given path to generate two new path

    def mutate(self, path,mutate_rate):
        # random select one to two position in the path to mutate

    def send_onlooker_bees(self):
        # send the onlooker bees to change path
```

Functions in the BeeColony:

1.

```
def __init__(self, num_cities, num_bees, num_epochs, elite_rate,
    local_search_rate,mutate_rate, neighborhood_size,limitation,scout_bee_rate):
```

This initialize the Bee colony
num_cities : the number of city in the graph
num_bees : number of bees total inside the coloney
num_epochs : number of iteration that the coloney are going to go through
elite_rate : the percentage of the bees that are going to survive to the next generation
mutate_rate : the rate to mutate for the outlooker bee
scout_bee_rate : the percentage of the scout bee
local_search_rate : the posibility for the scout bee to do local search
neighborhood_size : the range to rotate city for the scout bee when do the local search
limitation : the number of chance that the scout bee are going to try
valid_cities: list of index of all the citys in the graph
population : all the bee objects
best_path : The current best path amoung all the bees inside the population
best_distance : The current best distance amoung all the bees inside the population
average_distance : The current average distance amoung all the bees inside

2.

```
def get_best_path(self):
        #get current best path among all the bees
        return min(self.population, key=lambda bee:
bee.curr_path_distance).curr_path
```

This function not use for any optimization, just use to track the best path in the bee colony\

3.

```
def compute_path_distance(self, path):
        #get the distance of one bee
        distance = 0
        visited_cities = set()
        for i in range(len(path)-1):
            if path[i] == path[i+1] or path[i] in visited_cities:
                distance = float('inf')
                break
            visited_cities.add(path[i])
            distance += dist_matrix[path[i], path[i+1]]
        return distance
```

Comput a given path for example [0,3,6,4] it going to look at the distance matrix to add up all the distance

4.

```
def get_average_distance(self):
        #get the average distance for all the bees
```

```
        fitness = []
        for bee in self.population:
            curr_bee_dis = self.compute_path_distance(bee.curr_path)
            fitness.append(curr_bee_dis)
        average_distance = sum(fitness)/len(fitness)
        return average_distance
```

It is also not use for optimization and only use for keep track of the current average distance.

5.

```
def send_employee_bees(self):
        #send the employee bees
        num_elite = int(self.elite_rate * self.num_bees)
        elite_bees = sorted(self.population, key=lambda x:
x.curr_path_distance)[:num_elite]
        for i in range(num_elite,self.num_bees):
            new_path = np.random.permutation(self.valid_cities)
            new_bee = Bee(new_path)
            new_distance = self.compute_path_distance(new_path)
            if new_distance < self.best_distance:
                self.best_distance = new_distance
                self.best_path = new_path
            self.population[i] = new_bee
            self.population[i].curr_path_distance = new_distance
        for i in range(num_elite):
            self.population[i] = elite_bees[i]
```

This generate a group of work bee and choose number of elite got from the last generation and keep them not to be randomly generate a new path. And all other working bee are going to get a random new path(like a new choice)

6.

```
def greedy_algorithm(self, path):
        #greedily find the best path in a given city list
        unvisited_cities = set(path[1:])
        curr_city = path[0]  # keep the start city
        new_path = [curr_city]

        while unvisited_cities:
            nearest_city = min(unvisited_cities, key=lambda next_city:
dist_matrix[curr_city, next_city])
            new_path.append(nearest_city)
            unvisited_cities.remove(nearest_city)
            curr_city = nearest_city
        return np.array(new_path)
```

This is a greedy algorithm greedily find shortest distance of a given path with some of the cities

inside, This function will been use by the scout bee.

7.

```
def send_scout_bee(self):
        #send the scout bee
        max_trial=self.limitation * self.num_bees
        num_trial = 0
        while num_trial < max_trial:
            for i in range(self.num_bees):
                if random.uniform(0, 1) < self.scout_bee_rate:
                    bee_curr_path = self.population[i].curr_path
                    start = np.random.randint(0, number_of_cities)
                    end = np.random.randint(start+1, number_of_cities + 1)
                    subpath  = bee_curr_path[start:end]
                    if random.uniform(0, 1) < self.local_search_rate:
                        roll_num=random.randint(0, self.neighborhood_size)
                        roll_num=int(random.uniform(-roll_num, roll_num))
                        subpath = np.roll(subpath, roll_num)
                    greedy_path = self.greedy_algorithm(subpath)
                    new_path=[]
                    new_path.extend(bee_curr_path[:start])
                    new_path.extend(greedy_path)
                    new_path.extend(bee_curr_path[end:])
                    new_distance = self.compute_path_distance(new_path)
                    if new_distance < self.population[i].curr_path_distance and
new_distance!=0:
                        self.population[i].curr_path = new_path
                        self.population[i].curr_path_distance = new_distance
                        num_trial = 0  # Reset the counter for trials without
improvement
                    else:
                        num_trial += 1
```

This is the scout bee, it going to be run by a given limitation, and there is a possibility for a scout bee to choose the path of one bee. And it going to randomly choose part of the path of a work bee blank it, and it have certain rate the roll the that part of path like a local search and then use the greedy algorithm above to aggressively find a shorter path of the given part. And them merge the path together again, if the new path is shorter, give the path to that work bee.

8.

```
def roulette_wheel_selector(self, probs):
        #do a roulette wheel selection on all the bees
        roulette_wheel = random.uniform(0, sum(probs))
        current = 0
        for i, p in enumerate(probs):
            current += p
```

```
            if roulette_wheel <= current:
                return i
```

This is a roulette wheel selector for the onlook bee, is should have larger probability to get one with larger probs, in this case is the one with longer path distance. As I have wrote in the report, this is the job of onlook bee to make sure the bee colony does not fall into a local optimize solution.

9.

```
def crossover(self, path1, path2):
        #do a crossover on two given path to generate two new path
        path_length = len(path1)
        child1 = [-1] * path_length
        child2 = [-1] * path_length
        start = np.random.randint(0, path_length)
        end = path_length
        child1[start:end],child2[start:end] = path1[start:end],path2[start:end]
        unvisit_cities1 = [i for i in path2 if i not in child1]
        unvisit_cities2 = [i for i in path1 if i not in child2]
        for i in range(path_length):
            if child1[i] == -1:
                child1[i] = unvisit_cities1.pop(0)
            if child2[i] == -1:
                child2[i] = unvisit_cities2.pop(0)
        return child1, child2
```

This is simply a crossover method which usually appears in a generation algorithm here it uses to do a random cross over for make sure it not fall into a local optimal.

10.

```
def mutate(self, path,mutate_rate):
        # random select one to two position in the path to mutate
        if random.uniform(0, 1) < mutate_rate:
            for i in range(random.randint(1,2)):
                j, k = random.sample(range(len(path)), 2)
                path[j], path[k] = path[k], path[j]
        return path
```

This is a mutate method for onlooker to random a little bit on the path

11.

```
def send_onlooker_bees(self):
        # send the onlooker bees to change path
        fitness = []
        for bee in self.population:
            curr_bee_dis = self.compute_path_distance(bee.curr_path)
            fitness.append(curr_bee_dis)
```

```
        self.average_distance = sum(fitness)/len(fitness)
        num_elites = int(len(self.population) * self.elite_rate)
        elites = sorted(self.population, key=lambda x: x.curr_path_distance,
reverse=False)[:num_elites]
        new_population = elites[:]
        #move elites bees to the next generation first before look by onlooker bees
        while len(new_population) < len(self.population):
            # select parents by roulette wheel selection
            parent1 = self.roulette_wheel_selector(fitness)
            parent2 = self.roulette_wheel_selector(fitness)
            # crossover
            child1, child2 = self.crossover(self.population[parent1].curr_path,
self.population[parent2].curr_path)
            # mutation
            child1 = self.mutate(child1,self.mutate_rate)
            child2 = self.mutate(child2,self.mutate_rate)
            child1_fitness = self.compute_path_distance(child1)
            child2_fitness = self.compute_path_distance(child2)
            if child1_fitness < fitness[parent1]:
                new_population.append(Bee(child1))
            else:
                new_population.append(self.population[parent1])
            if child2_fitness < fitness[parent2]:
                new_population.append(Bee(child2))
            else:
                new_population.append(self.population[parent2])

        self.population = new_population
```

Here the main propose of the onlooker bee is to let the bee colony not fall into a local optimal so it going to do a roulette wheel selector on worker bees and have probabilities to use crossover and/or mutate to the work bee path if the new path is shorter then it going to choose the new path for the worker bee.

That is all the function for Bee colony class.

And there are some other global function for generate the map and distance matrix.

The map reader

```
def get_city_map(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
    print(len(lines))
```

```
    coords = []
    for y in range(len(lines)):
        curr_line=lines[y].split()
        for x in range(len(curr_line)):
            if curr_line[x] == '1':
               coords.append([y,x])
    return np.array(coords)
```

This function was used for get a map from the text file with 0 and 1s in it which 1 represent a city.
It will return a np array of all the coordinate of the city inside the map.

Distance matrix generator

```
def generate_distance_matrix(n, coords):
    distance_matrix = np.zeros((n, n), dtype=float)
    for i in range(n):
        for j in range(i, n):
            distance = np.sqrt(np.sum((coords[i] - coords[j]) ** 2))
            distance_matrix[i, j] = distance
            distance_matrix[j, i] = distance
    return distance_matrix
```

This function are going to generate a distance matrix form a given coordinate list.

Use map from the txt file or random generate one

```
number_of_cities = 30
max_distance = 200  #you can change this if there is more cities add into the graph
#---------------------------------Change the Coords here !!!-----------------------
----------------------
#coords = get_city_map("my_array.txt")
coords = np.random.randint(max_distance, size=(number_of_cities, 2))
#---------------------------------------------------------------------------------
----------------------
```

As I said, here you can change the map, as there is only 30 cities inside the txt file, change the number of cities only if you switch to randomly generate a new map.
Also you can change the map size by change the max distance.

Main model (How to use it!!!)

```
bee_colony = BeeColony(num_cities=number_of_cities,
```

```
                    num_bees=100,
                    num_epochs=35,
                    elite_rate=0.05,
                    local_search_rate=0.5,
                    mutate_rate=0.5,
                    neighborhood_size=8,
                    limitation=2,
                    scout_bee_rate=0.1
                    )
```

Here is the main model, the num_bees is the number of bees in the map. It should be higher than the number of city to get a good performance.
Num_epochs is just the number of iteration for the colony
Elite_rate is the percentage of work bee to become elite to survive a generation
Local_search_rate is for the probability of scout bee to do the rotation on the path
Mutate_rate is for the possibility of a onlook be are going to change the path by one or two cities
Neighborhood_size is for the bee to decide the number of cities to rotate
Limitation is the limit for the scout bee to work
Scout_bee_rate stand for the probability that a scout bee are going to choose the path of a worker bee

You can change some of them, but the default should work best on a 30 cities graph

Training

```
#start training
for i in range(bee_colony.num_epochs):
    bee_colony.send_employee_bees()
    bee_colony.send_scout_bee()
    bee_colony.send_onlooker_bees()
    bee_colony.best_path = bee_colony.get_best_path()
    bee_colony.best_distance =
bee_colony.compute_path_distance(bee_colony.best_path)
    bee_colony.average_distance = bee_colony.get_average_distance()
    print(f"Epoch {i+1}: Best distance = {bee_colony.best_distance}, Average
distance = {bee_colony.average_distance}")
    plt.clf()

    x = coords[:, 0]
    y = coords[:, 1]
    best_path=bee_colony.best_path

    for k in range(len(best_path) - 1):
        j = best_path[k]
```

```
        k = best_path[k + 1]
        dx, dy = coords[k] - coords[j]
        plt.arrow(coords[j, 0], coords[j, 1], dx, dy, head_width=0.2,
length_includes_head=True,
                  color='red', linestyle='dashed', linewidth=2, shape='right')
    plt.scatter(x, y)
    for j, city in enumerate(coords):
        plt.text(city[0], city[1], str(j))
    plt.scatter(coords[best_path[0], 0], coords[best_path[0], 1], color='green',
s=100, zorder=3)
    plt.scatter(coords[best_path[-1], 0], coords[best_path[-1], 1], color='orange',
s=100, zorder=3)
    plt.xlabel('X')
    plt.ylabel('Y')
    print()
    plt.title('BCO TSP Solution distance: '+str(bee_colony.best_distance))
    plt.savefig('./results/'+ str(i) +'.jpg')
    epochs.append(i)
    bestdist.append(bee_colony.best_distance)
    averagedist.append(bee_colony.average_distance)
    plt.pause(0.1)
```

This part is for run the Training and it going to generate a path graph for the best path for each generation.

Fitness Graph

```
plt.clf()
plt.plot(epochs, bestdist, linestyle = '-', label = 'Best fitness')
plt.plot(epochs,averagedist, linestyle = '-.', label = 'Average fitness')
plt.xlabel('Distance')
plt.ylabel('epochs')
plt.title('BCO TSP')
plt.legend()
plt.show()
plt.savefig('./results/bee.jpg')
```

finally it going to draw a epoch and fitness graph for the beecolony.

Example Output:

There is some example output inside the directory ./example_output

```
    bestdist.append(bee_colony.best_distance)
    averagedist.append(bee_colony.average_distance)
    plt.pause(0.1)

plt.clf()
plt.plot(epochs, bestdist, linestyle = '-', label = 'Best fitness')
plt.plot(epochs,averagedist, linestyle = '-.', label = 'Average fitness')
plt.xlabel('epochs')
plt.ylabel('Distance')
plt.title('BCO TSP')
plt.legend()
plt.show()
plt.savefig('./results/bee.jpg')
```

```
Shell
Epoch 23: Best distance = 848.2763847936293, Average distance = 974.2703149359234
Epoch 24: Best distance = 848.2763847936293, Average distance = 1014.6764916119783
Epoch 25: Best distance = 848.2763847936293, Average distance = 1006.759717158658
Epoch 26: Best distance = 848.2763847936293, Average distance = 1015.1882542333423
Epoch 27: Best distance = 848.2763847936293, Average distance = 1005.0617247636056
Epoch 28: Best distance = 848.2763847936293, Average distance = 992.5574520154585
Epoch 29: Best distance = 843.2271955833312, Average distance = 1002.1506986464622
Epoch 30: Best distance = 827.8161948177446, Average distance = 1004.9328421113089
Epoch 31: Best distance = 827.8161948177446, Average distance = 978.5167808500981
Epoch 32: Best distance = 827.1019999180039, Average distance = 998.707552141225
Epoch 33: Best distance = 827.1019999180039, Average distance = 978.524522402486
Epoch 34: Best distance = 827.1019999180039, Average distance = 992.816864656687
Epoch 35: Best distance = 822.3790092587052, Average distance = 1024.7196777130418
```

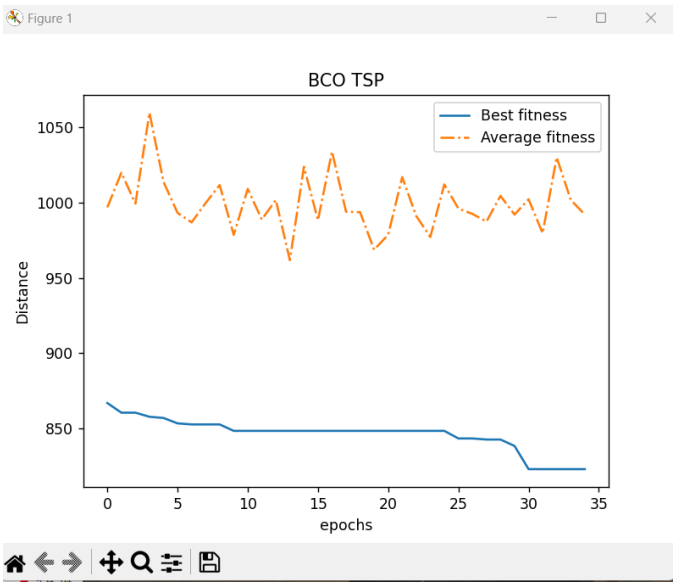
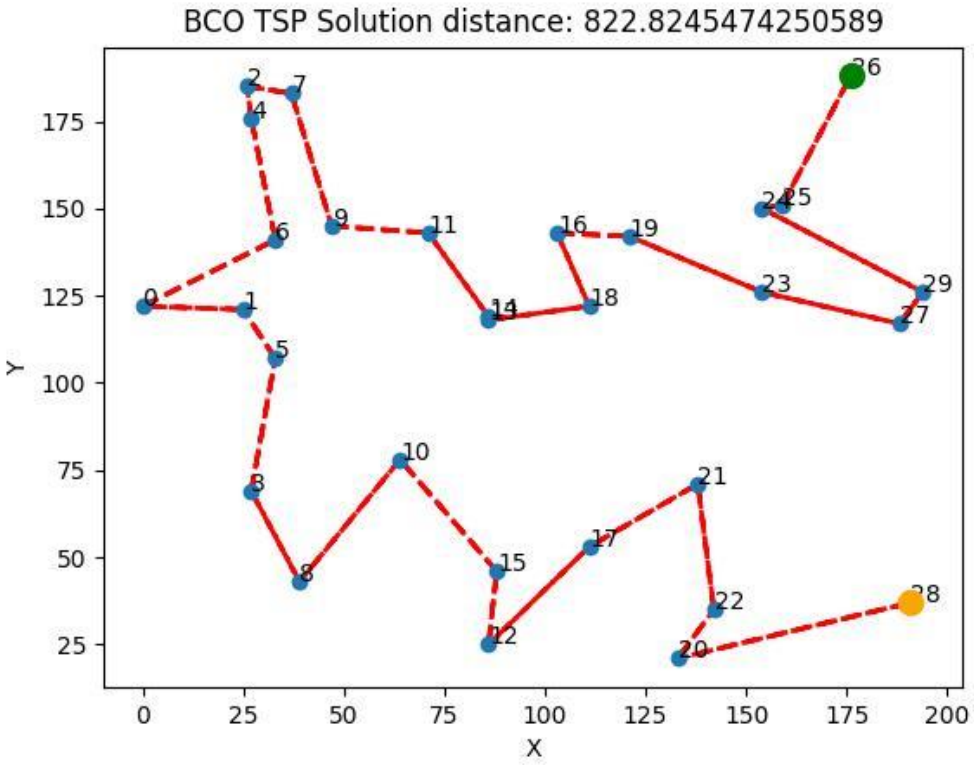
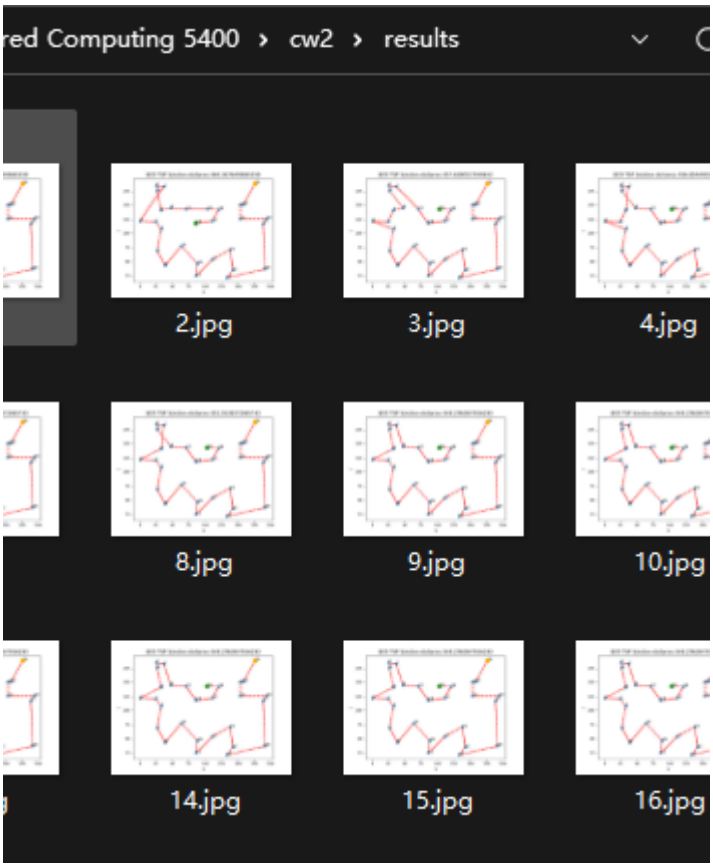
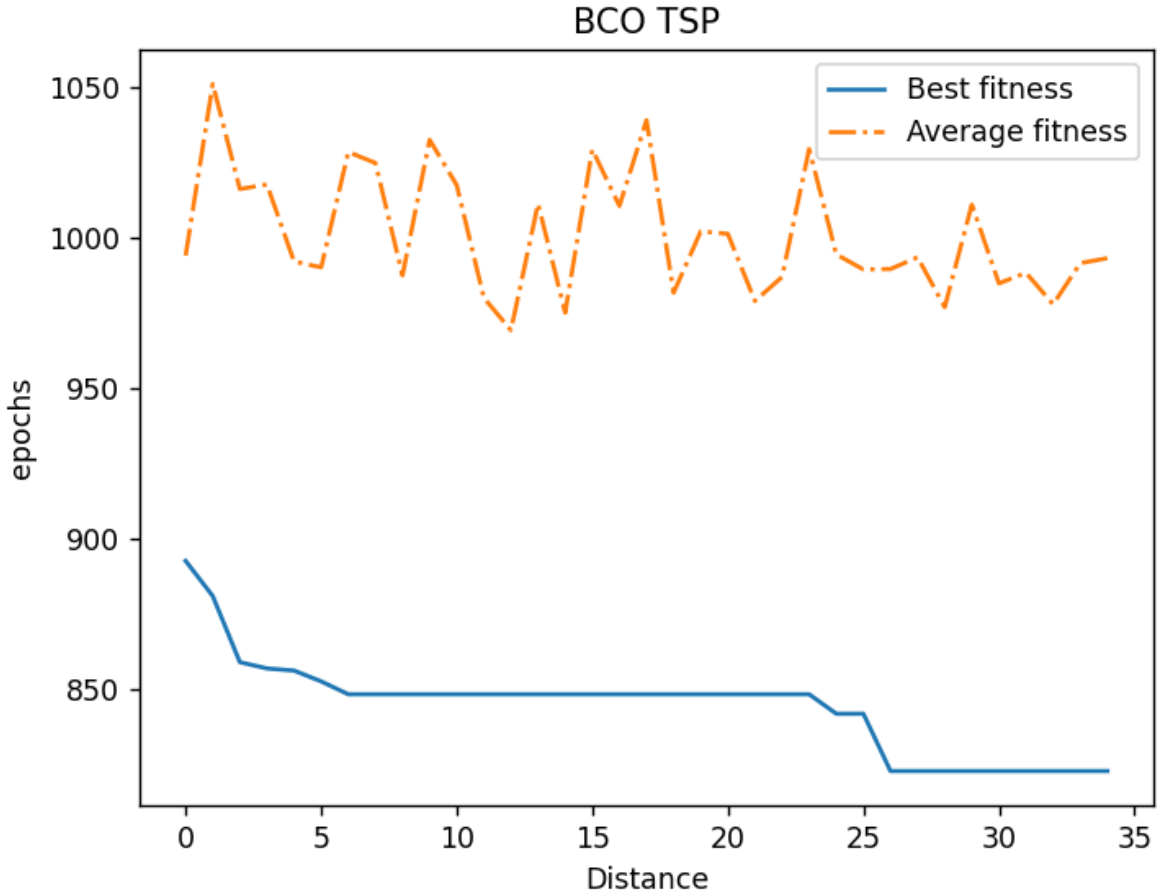


Figure 1
BCO TSP
Best fitness
Average fitness
Distance
epochs
1050
1000
950
900
850
0
5
10
15
20
25
30
35


BCO TSP Solution distance: 822.8245474250589
X
Y
0
25
50
75
100
125
150
175
200


red Computing 5400 › cw2 › results
2.jpg
3.jpg
4.jpg
8.jpg
9.jpg
10.jpg
14.jpg
15.jpg
16.jpg


BCO TSP
Best fitness
Average fitness
epochs
Distance
1050
1000
950
900
850
0
5
10
15
20
25
30
35