CSE5001 Fall 2021 Project Report

李岱峰 Daifeng Li, 12132268
李卓伦 Zhuolun Li, 12132273
余成明 Cheng ming, 12132371
刘念 Nian Liu, 12132347
Equal Contribution

December 23, 2021

Contents

1 Introduction

The Travelling Salesman Problem (TSP) is one of the most common optimization problems in operational research. One of its variations is the multiple Travelling Salesman Problem (mTSP), where disjoint routes for m salesmen traveling to n cities have to be found. The routes should cover all cities and every salesman should visit at least one city, originating from and returning to the same unified start city, which is called depot here. The quality of the routes can be evaluated using qualitative and quantitative measures, such as distance, time, cost, customer satisfaction, salesman qualification, workload balance to name a few. In the current large number of studies, many algorithms have been introduced to solve the TSP with its variations, and relatively few approaches were applied to handle the mTSP.

The importance of the mTSP arises from the fact that it is a well-known combinatorial optimization problem that models many real-life problems. The mTSP has been tackled using many different approaches; most of them are nature-inspired algorithms of which Genetic Algorithms (GA) is the most typical one. Most of the GA-based approach for solving the mTSP has used a single chromosome for representation. The new approach presented as a *baseline* here is a so-called multi-chromosome technique, which separates the salesmen from each other [3].

In this report, we analyzed the methodology of *baseline* and add in a more efficient algorithm called AC-GA that optimized the initial route to improve the performance. Using the six requested instances, AC-GA shows better results than *baseline* all the time while only increasing time cost no more than twice. Evaluation is explained in detail in Chapter 4.

2 Analysis of algorithms

2.1 Baseline Method

The *baseline* is a so-called multi-chromosome technique, in which each salesman is represented by a chromosome. The route is generated and assigned to salesmen completely randomly to obtain the primitive population. Then mutation is applied to each individual in the population and Iteratively select the individual with minimum fitness as the new population. In this case, the fitness value is the total cost of the transportation, i.e. the total length of each round trip. The algorithm finishes if the stop criteria, i.e. the number of iterations, is satisfied.

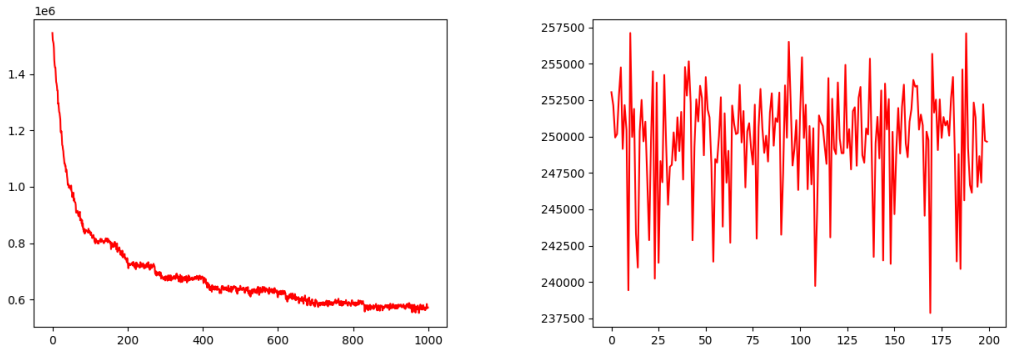
2.2 Ideas for improvement

Since *baseline* is using GA to solve the mTSP problem, the generation of the primitive population has a large impact on the efficiency and effectiveness of the algorithm. Otherwise, the process of mutation lacks interpretability, so the optimization can only be considered in accordance with initialization.

When the cities a salesman needs to visit are fixed, the mTSP problem can be split into several independent TSP problems. ACO is a considerable approach for solving the TSP problem, so we use ACO to pre-process the population. The superiority of applying ACO is that the path of each salesman in our pre-processed primitive population is better than that of *baseline* in which cities are randomly assigned to generate the primitive population, an intuitive example is given here.

In the largest instance (Pr226), the minimum distance based on the randomly generated primitive population is over 1.5 million, and even after 1000 iterations by GA, the minimum distance is around 600 thousand, as shown in Figure 1(a). (The horizontal axis is the number of iterations and the vertical axis is the distance)

In contrast, after only 20 iterations of ACO, the minimum distance based on the ACO-preprocessed primitive population is only about 250 thousand, as shown in Figure 1(b), which is almost only $\frac{1}{6}$ of the previous one. In this case, we can see how much the quality of the initial population affects the algorithm. Additionally, the formation of Figure 1(b) is because the result is close to the optimal or better solution and therefore fluctuates in a region without convergence.



(a) Baseline Method

(b) AC-GA Method

Figure 1: Convergence Effect

3 Explanation of our algorithm

3.1 Modifications

Before applying a new algorithm to optimize *baseline*, we first did some modifications to it.

Random number generation The random number generation method is at line 106 of the source code in *ga-logic.py*. The original code will make the *if* condition always holds, resulting in a mutation operation for each generation. Compared to operating mutation every time, it would be more reasonable to set a certain probability of mutation rate, which is also in line with the original design intention of *baseline*. At the same time, the execution position of this criterion is not reasonable, even without mutation, the code will still carry out the initialization of mutation. Therefore, we modified the criterion as follows, and changed the judgment position.

```
#original code of baseline
if random.randrange(1) < mutationRate:

#our modification
if random.randrange(100)/100 < mutationRate:
    cls.mutate(newPopulation.getRoute(i))
```

Store distances using a two-dimensional matrix In the source code, the *distanceTo (self, db)* function in *dustbin.py* is called when the city distance needs to be calculated. This function needs to be called frequently during the running of the algorithm, but the way it is taken in *baseline* is to calculate the distance directly each time, which brings a large computational overhead. Our improvement is that the distances between points in the dataset are stored as a two-dimensional matrix, e.g. the distance between city i and city j is recorded in the row i and column j of the matrix. After reading the coordinates of cities in dataset, the distance matrix is obtained by running the newly added code only once. Later, when the *distanceTo (self, db)* function is called, it simply returns the corresponding value in the distance matrix.

```
#original code of baseline
def distanceTo (self, db):
    xDis = abs(self.getX() - db.getX())
    yDis = abs(self.getY() - db.getY())
    dis = math.sqrt((xDis*xDis) + (yDis*yDis))
    return dis

#our modification
dist_map = np.zeros((numNodes,numNodes))
for i in range(numNodes):
    for j in range(numNodes):
        if i != j:
            dist_map[i][j] = math.sqrt((X[i] - X[j])**2 + (Y[i] - Y[j])**2)
        else:
            dist_map[i][j] = 100000

def distanceTo (self, db):
        dis = dist_map[self.getIndex()][db.getIndex()]
        return dis
```

3.2 Ant colony optimization

Ant Colony Optimization (ACO) is a swarm intelligence algorithm, which is a group of unintelligent or slightly intelligent individuals (Agents) that exhibit intelligent behavior by collaborating with each other, thus providing a new possibility for solving complex problems. Ant colony optimization was first proposed by Italian scholars Colorni A, Dorigo M, etc. in 1991. [1] After more than 20 years of development, ant colony optimization has made great progress in theoretical as well as applied research.

A new approach using ACO to solve mTSP named AC2OptGA was presented in Youssef et al. [2] AC2OptGA is a combination of three algorithms: Modified Ant Colony, 2-Opt, and Genetic Algorithm. We took the basic idea of it as a reference and designed AC-GA.

3.3 Methodology

We propose a new hybrid approach based on the heuristic-based methods ACO and GA. ACO is used to generate a set of good feasible solutions for the TSP (one salesman). Then, we randomly choose a feasible solution from the set as population. After that, the following steps are the same as *baseline*. The basic steps of ACO are below:

- Setting up multiple ants to search parallelly in the graph.
- Ants release pheromones when passing a path (The amount of pheromone deposited is the inverse of the total route length).
- Ants have a greater probability of choosing the path with the larger pheromone.
- Each ants find a legal path, then this round of iteration ends.

Repeating the above steps, the general iteration stops when the predefined number of iterations is reached or all ants choose the same path. Here the probability function is defined as follows (See Appendix A for detailed code

documentation):

$$p_{ij}^{k} = \begin{cases} \frac{(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}{\sum_{j \in N_i^k}(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})} & \text{if } j \in N_i^k \\ 0 & \text{if } j \notin N_i^k \end{cases}$$

where:

- N_i^k: The set of nodes adjacent to node i that have not been visited yet by the kth ant;
- τ_{ij}^{α}: The amount of pheromone on edge connecting node i to j;
- η_{ij}^{β}: Known as *Heuristic Information*, is the inverse of cost to visit node j from node i;
- α, β: Parameters set by user *a priori* that control the weight/influence τ_{ij} and η_{ij} hold in the formula, respectively.

4 Evaluation

4.1 Wilcoxon Rank Sum Test

In statistics, the Wilcoxon rank-sum test (also called Mann Whitney U test) tests the null hypothesis that two sets of measurements are drawn from the same distribution. The alternative hypothesis is that values in one sample are more likely to be larger than the values in the other sample.

For testing true of false of the hypothesis above, we need to calculate the U statistic of each samples, It is calculated as follows:

Let $X_1, \ldots, X_n$ be an i.i.d. sample from X, and $Y_1, \ldots, Y_m$ an i.i.d. sample from Y, and both samples independent of each other. The corresponding (Mann-Whitney) U statistic is defined as:

$$U = \sum_{i=1}^{n} \sum_{j=1}^{m} S(X_i, Y_j)$$

with

$$S(X,Y) = \begin{cases} 1, & \text{if } X > Y, \\ \frac{1}{2}, & \text{if } Y = X, \\ 0, & \text{if } X < Y. \end{cases}$$

Once we have the U value, we can compare it with the appropriate critical value of U with sample sizes (for example: n1=n2=5) and one-sided or two-sided level of significance (=0.05). The appropriate critical value of U can be found in the *table of Mann-Whitney U critical value* like in this link: http://socr.ucla.edu/Applets.dir/WilcoxonRankSumTable.html

Fortunately, in `scipy` package there is a function called `scipy.stats.wilcoxon` which can help us with the test. This function takes two samples as input and outputs the U statics and p value, where the p value is our needed. If the p value is less than 0.05 we can have a straight conclusion that which samples have statistically larger value.

For example, we have two samples called X and Y, we assume statistically X has larger value than Y. To prove it, do as follows, and if the output `res.pvalue` is less than 0.05, we can conclude that statistically X has larger value than Y.

```
from scipy import stats
# we have one sample called x
# we have another sample called y
res = stats.wilcoxon(x,y, alternative='greater')
print(res.pvalue)
```

4.2 Comparison

Both *baseline* and our proposed algorithm AC-GA are implemented and tested using Visual Studio Code and Anaconda on PC having the following specifications: 3.70GHz AMD Ryzen 5 5600X 6-Core Processor, 32.00 GB RAM, and running under Windows 10. The relevant parameters included are set as follows:

- Number of salesmen = 5
- Number of ants = 100
- Population size = 100
- Mutation rate = 0.5
- Pheromone exponential weight = 1
- Heuristic exponential weight = 2
- Evaporation rate = 0.1
- Number of iterations = 200

For each independent running of the baseline or ours AC-GA method, we collected the *initial distance of the route*, the *time cost* of each running and the *final global minimum distance* that the algorithm generates, like the figure 2 and figure 3 show.

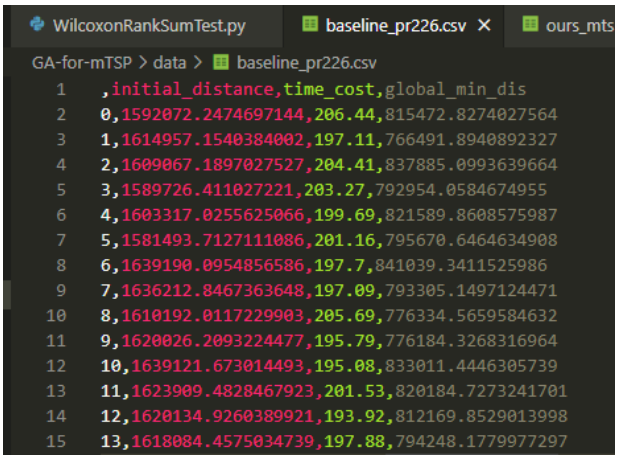


Figure 2: The collected data of baseline method on pr226

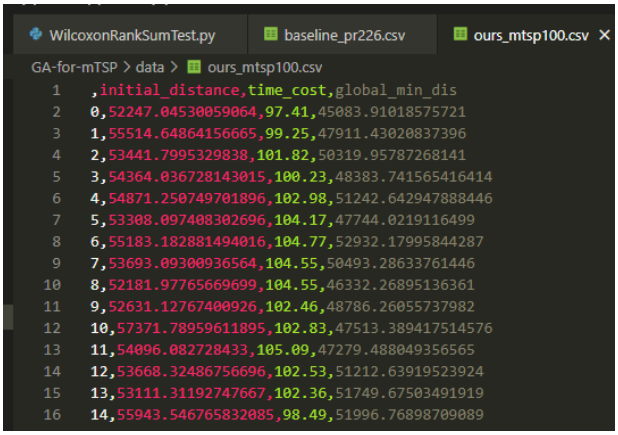


Figure 3: The collected data of ours method on mtsp100

After making 30 independent runs of each method in each 6 instances, we applied a python script on these data we collected, to generate the *mean value of the global minimum distance*, the *mean value of the time cost*, this python file/script is appended at the appendix B, and also in our submission zip file. The mean value of distances and time cost obtained by each method for each instance are summarized in Table 1. Then results were also processed by the *Wilcoxon Rank Sum Test* and all the output results are shown in the figure 4.

Table 1: Table of mean distance & time cost generating by two methods of all instances

Instance	**AC-GA Distance**	**Baseline Distance**	**AC-GA Time (sec)**	**Baseline Time (sec)**
mtsp51	709.18	848.86	45.05	44.54
mtsp100	49272.61	77755.45	101.51	88.61
mtsp150	92156.99	129261.42	176.24	129.39
pr76	232360.23	312480.62	69.47	65.28
pr152	186051.64	473904.26	178.64	131.37
pr226	231886.77	802205.27	318.88	201.94

Since the p-values (approximately 8.7e-7) of all the Wilcoxon rank sum tests are far smaller than 0.05, they are apparently not from the same distribution and also because the mean value of the distance generated by ours AC-GA method is obviously smaller than those generated by the baseline method. Additionally, the time our method spent is less than 2 * CPU runtime of the baselinem, which meets the requirement of the project description, **we can have a conclusion that our method exerts much better performance than the baseline method**.

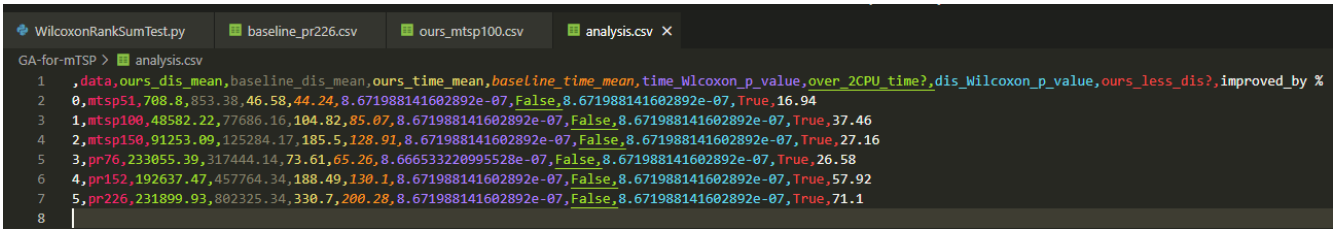


Figure 4: Processed data of all the consequences obtained by each method for each instance

5 Discussion and conclusion

AC-GA produces a wide range of high-performance solutions with small differences between the best, average, and worst. We believe that there is still room for improvement by incorporating smart techniques to build better mTSP solutions.

In this paper, a detailed analysis was given about the two hybrid algorithms in multiple Traveling Salesman Problem. A novel approach, AC-GA, is presented and implemented using Python. The proposed method combines the Ant Colony Optimization and Genetic Algorithm. An experimental study was conducted to evaluate the performance of the proposed approach to solve various mTSP instances. The obtained results show that the AC-GA overperforms *baseline* with all the six instances and the time cost are all controlled within twice the *baseline*.

References

[1] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. 01 1991.

[2] Youssef Harrath, Abdul Fattah Salman, Abdulla Alqaddoumi, Hesham Hasan, and Ahmed Radhi. A novel hybrid approach for solving the multiple traveling salesmen problem. *Arab Journal of Basic and Applied Sciences*, 26(1):103–112, 2019.

[3] András Király and János Abonyi. *A Novel Approach to Solve Multiple Traveling Salesmen Problem by Genetic Algorithm*, volume 313, pages 141–151. 10 2010.

A Code Documentation

All our code can be found in https://github.com/Lenny-Lee-ustb/GA-for-mTSP
The main modification of code is to add the Ant Colony Algorithm in the file *ant.py*. And we make minor changes to the original code to fit our new algorithm. Besides, we fix a serious bug in the original code, which can be even called an error.

- *ant.py*
  The main part of Ant Colony Algorithm(AC).
  Firstly, we create a class *AntColony*, there are two member variables:

```
def __init__(self):
    self.path_best = []
    self.distance_best = []

```

  path_best is to store the best path of each iteration, *distance_best* is to store the shortest distance of each iteration.
  The main function is in the member function *iterate*. In the function, all ants will do the same actions, which is the feature of AC. We can split the whole algorithm into four steps for each ant:

  1. Find a stating city
  2. Choose the next city by the concentration of pheromone.
  3. Update pheromone on its path
  4. Repeat step 2-3 until find a complete path

When all ants have finishing these steps, it marks the end of one iteration. Then we calculate the distance and find the best path.
The default number of iterations in our code is just 20, because the overhead of AC is significantly higher than GA.
The key of AC is how to choose the next city by the concentration of pheromone:

```
for k in range(len(unvisit)):
    # calculate all citys' concentration of pheromone
    protrans[k] = np.power(pheromonetable[visit][unvisit[k]], alpha) * np.power(
        etable[visit][unvisit[k]], bate)
    cumsumprobtrans = (protrans / sum(protrans)).cumsum()
    # choose the next city by probability
    cumsumprobtrans -= np.random.rand()
    k = unvisit[list(cumsumprobtrans > 0).index(True)]
    candidate[i, j] = k
    unvisit.remove(k)
    length[i] += Distance[visit][k]
    visit = k
    length[i] += Distance[visit][candidate[i, 0]]

```

And update the pheromone of each city:

```
changepheromonetable = np.zeros((city_count, city_count))
# calculate the modification
for i in range(AntCount):
    for j in range(city_count - 1):
        changepheromonetable[candidate[i, j]][candidate[i][j + 1]] += Q / length[i]
    changepheromonetable[candidate[i, j + 1]][candidate[i, 0]] += Q / length[i]
# update all pheromone
pheromonetable = (1 - rho) * pheromonetable + changepheromonetable

```

- *population.py*
  When the population initialise, we will excute AC firstly and choose the path from AC solution instead of random generating to generate individual.

```
def __init__ (self, populationSize, initialise):
    self.populationSize = populationSize
    if initialise:
        # excute AC when initialise
        ants = AntColony()
        ants.iterate(20)
        ant_path = ants.path_best
        for i in range(populationSize):
            newRoute = Route() # create empty route
            # random choose one path of AC results and generate individual
            choose = random.randint(1, len(ant_path))
            newRoute.generateIndividual(ant_path[choose - 1] + 1)
            self.routes.append(newRoute) # Add route to the population

```

- *route.py*

```
def generateIndividual (self, antpath):
    k=0

    # Original version use a random method to create init generate:
    # ---
    # # put 1st member of RouteManager as it is (It represents the initial node) and shuffle the
    rest before adding
    # for dindex in range(1, RouteManager.numberOfDustbins()):
    #     self.base[dindex-1] = RouteManager.getDustbin(dindex)
    #     random.shuffle(self.base)
    # ---
    # Now we use ant colony algorithm to do this, every time we generate an individual,
    # we random choose one from ant colony.
    # The whole algorithm is in file ant.py

    # print(antpath)
    for i in range(numTrucks):
        self.route[i].append(RouteManager.getDustbin(0)) # add same first node for each route
```

```
        for j in range(self.routeLengths[i]-1):
            Dustbin = RouteManager.getDustbin(int(antpath[k]))
            self.route[i].append(Dustbin)
            k+=1

```

B WilcoxonRankSumTest.py

```
from scipy import stats
import numpy as np
import pandas as pd

data_name = ["mtsp51", "mtsp100", "mtsp150", "pr76", "pr152", "pr226"]
df = pd.DataFrame(columns=['data', 'ours_dis_mean', 'baseline_dis_mean',
                           'ours_time_mean', 'baseline_time_mean', 'time_Wlcoxon_p_value',
                           "over_2CPU_time?", 'dis_Wilcoxon_p_value', 'ours_less_dis?',
                           'improved_by %'])
k = 0
for data in data_name:
    print("Doing Wilcoxon Sum Rank Test on the data " + data + ": ")
    df_baseline = pd.read_csv("./old_data/baseline_" + data + ".csv", sep=",",
                              header=0, names=['initial_distance', 'time_cost', 'global_min_dis'])
    df_ours = pd.read_csv("./old_data/ours_" + data + ".csv", sep=",",
                          header=0, names=['initial_distance', 'time_cost', 'global_min_dis'])

    mean_basline_time = df_baseline['time_cost'].mean()
    mean_ours_time = df_ours['time_cost'].mean()
    mean_baseline_dis = df_baseline['global_min_dis'].mean()
    mean_ours_dis = df_ours['global_min_dis'].mean()

    baseline_dis = df_baseline.iloc[:, 2].values
    ours_dis = df_ours.iloc[:, 2].values
    baselin_time = df_baseline.iloc[:, 1].values
    ours_time = df_ours.iloc[:, 1].values

    res_time = stats.wilcoxon(2*ours_time, baselin_time, alternative='greater')
    res_dis = stats.wilcoxon(baseline_dis, ours_dis, alternative='greater')
    df.loc[k] = [data, round(mean_ours_dis, 2), round(mean_baseline_dis, 2), round(mean_ours_time, 2),
    round(mean_basline_time, 2),
                 res_time.pvalue, True if res_time.pvalue > 0.05 else False,
                 res_dis.pvalue, True if res_dis.pvalue < 0.05 else False,
                 round((mean_baseline_dis-mean_ours_dis)/mean_baseline_dis*100, 2)]
    k = k+1
df.to_csv("analysis.csv")
```