Traveling Salesman Problem

Coursework #2

CS454 AI Based Software Engineering

Oct 10th 2019

20170504 Juan Lee

1. Introduction

This assignment is to make a solver for Traveling Salesman Problem (TSP), which is known as NP problem so that we cannot solve TSP in polynomial time (under $P \neq NP$). So, the purpose of this assignment is to lower the result as many as possible using stochastic algorithms and heuristics.

I have used four different algorithms, Genetic Algorithm (GA), Greedy Algorithm, Ant Colony Optimization (ACO), and modified Minimum Spanning Tree (MST) algorithm. Note that MST algorithm is deterministic, and I used this algorithm only for generating initial data for other algorithms or for examining the escape from local optima.

1.1 Environment

Since this assignment is highly affected by computing power, so I note my testing environment. All the codes are written in Python 3.7, and tested on macOS Mojave 10.14.6 with processor 2.2GHz Intel Core i7 and memory 16GB 2400MHz DDR4.

Also, I used `a280.tsp` for examination because it contains proper (not so many not so little) number of nodes to compute in my computer. However, some nodes in `a280.tsp` have exactly same value of x and y, so it does not satisfy triangular inequality. I added some error handling for this case. (For example, immediately move to other node if they are in same location)

1.2 Code Structure

This is the code structure, and all the python programs are in `tsp` folder. `a280.tsp`, `att532.tsp` and `rl11849.tsp` are test files. `tsp.py`is executable file for all the algorithms. I will explain it in **1.3**. `aco.py` is for Ant Colony Optimization algorithm. See function aco if you want to look into my algorithm. `ga.py` is for Genetic Algorithm, starting from function ga. `greedy.py` is greedy algorithm, starting from function greedy. `mst.py` is modified version of Minimum Spanning Tree algorithm, starting from function mst.

`README.md` is markdown version of this report, and `report.pdf` is this file.

```
tsp
├── tsp
│   ├── a280.tsp
│   ├── att532.tsp
│   ├── rl11849.tsp
│   ├── tsp.py
│   ├── aco.py
│   ├── ga.py
│   ├── greedy.py
│   └── mst.py
├── README.md
└── report.pdf
```

Note that all the algorithms can be run separately, but I recommend you to follow **1.3**.

1.3 How to Run TSP Solver

This is how you start my TSP Solver with all default values,

```
$ cd ./tsp
$ python3 tsp.py rl11849.tsp
```

I provide many useful flags to control algorithm, constants, strategy, or system functions.

1.3.1 General Flags

-a alg: choose algorithm. alg is one of `aco`, `ga`, `greedy`, or `spanning`. greedy is default.

-h: show help

-l: print log. False is default.

1.3.2 Ant Colony Optimization Flags

> For detailed information, read related chapter.

-p size: the number of ants. 10 is default.

-w weight: pheromone weight. 0.5 is default. length weight is set (1-weight)

-f size: the number of generations. 100 is default.

-i value: initial pheromone. 1 is default

-e length: estimated shortest tour for ACO. 1000000 is default

-r rate: evaporation rate. 0.1 is default

-x strategy: examination strategy. one of `tester` and `mst`. mst is default.

1.3.3 Genetic Algorithm Flags

> For detailed informaion, read related chapter.

-p size: the size of population. 150 is default

-w rate: selection rate. 0.1 is default.

-f size: the number of generations. 100 is default.

-x value: crossover strategy. value is one of `my`, `cx`, `pmx`, and `no`. cx crossover is default.

-s value: selection strategy. value is one of `overselect` and `elitism`. overselect is default.

-m rate: mutation rate. 0.05 is default.

-gl length: gene max length for my crossover algorithm. 100 is default.

-ie rate: ratio of maintaining best solution at initializing

-im rate: ratio of generating mutated second-best solution at initializing.

1.3.4 Greedy Algorithm Flags

> For detailed information, read related chapter.

-i node: initial node. randomly chosen node is default.

1.3.5 Example Usages

This is an example of tsp solver solving `rl11849.tsp` using Ant Colony Optimization algorithm with one ant, almost (0.9) depending on length, and run 1000 generations.

```
$ python3 tsp.py rl11849.tsp -a aco -p 1 -f 0.1 -g 1000
```

This is an example of tsp solver solving `rl11849.tsp` using Greedy Algorithm starting from node 52. (now deterministic)

```
$ python3 tsp.py rl11849.tsp -a greedy -i 52
```

2. Genetic Algorithm

Genetic Algorithm is a bio-inspired algorithm from the theory of evolution. From randomly generated origin population, their children inherits their parents' genes. The genes are mutated, crossover-ed, and selected from their parents. For TSP, a permutation of n nodes represents a gene. You can see the code in `ga.py`.

2.1 Initialization

```
# initialize
try:
  population = initializeFromExisting("solution.csv", nodes, POPULATION_SIZE)
except:
  population = initializeWithRandom(nodes, POPULATION_SIZE)
```

For genetic algorithm, since it takes too much time to compute from very initial, I made that initializing can begin from an existing file. If there exists `solution.csv`, it creates population using the solution, or, creates population randomly with size of `POPULATIN_SIZE`.

`initializingFromExisting` uses two constants, `INITIALIZE_SAME_AS_EXISTING_RATE` and `INITIALIZE_MUTATE_ONCE_FROM_EXISTING_RATE`, which are given by flags `-ie` and `-im` respectively. `-ie` is the ratio of genes which are same as the solution and `-im` is the ratio of genes which is made by one mutation from the solution.

2.2 Generations

```
# generations
m = 99999999999999
for i in range(GENERATION):
  population, mDist, mPath = select(nodes, population)

  if m > mDist:
    m = mDist
    saveToFile("solution.csv", mPath)

print(mDist)
```

For each generation, it simply selects from population, and assigns the results to population back. All the other processes like mutation or crossover happen in the select function. Also, at the end of each operation, if the current population makes shorter distance than saved current shortest value, it stores the sequence of nodes to `solution.csv` file.

The number of generations is in `GENERATION` variable which can be controlled by `-f` flag.

2.3 Selection

For the selection algorithm, I implemented two strategies, elitism and overselect. Elitism is a way to choose only elites from the population, and all the non-elites genes are discarded. This strategy is good for solving TSP faster, however, it easy falls down to local optima and difficult to escape it. The only way to escape the local optima is mutation in this case.

Overselect is a way to use both elites and non-elites. It generates $(1-r) \times 100\%$ of new poplutions using top $r \times 100\%$ of parents. Also, by using bottom $(1-r) \times 100\%$ of parents, generates $r \times 100\%$ of new population. So, if we set $r < 0.5$, it means that we assume some of top parents are more productive than other bottom parents, and this makes sense in aspect of the theory of evolution. For my algorithm, I use overselect strategy as a default selection algorithm.

All the other operations like mutation and crossover happen in the selection, thus both selection strategies have same structure like below:

```
rp1 = random.choice(bottom)
rp2 = random.choice(bottom)
np1, np2 = crossOver(rp1, rp2)
np1 = mutate(np1, GENE_MUTATE_RATE)
np2 = mutate(np2, GENE_MUTATE_RATE)
newpopulation += [np1, np2]
```

You can control the selection strategy by giving `-s` flag. If not given, overselect is default strategy.

2.4 CrossOver

For the crossover algorithm, I implemeneted three strategies, cx, pmx2, and my own strategy. The cycle crossover (CX) strategy is literally choose crossing genes cyclic, staring from the fixed first gene.

```python
# cxCrossOver: path, path -> two paths
# - cross over using cycle crossover algorithm
def cxCrossOver(x1, x2):
    y1 = [-1] * len(x1)
    y2 = [-1] * len(x2)

    y1[0] = x1[0]
    y2[0] = x2[0]
    i = 0

    # do once first
    while x2[i] not in y1:
        j = x1.index(x2[i])
        y1[j] = x1[j]
        y2[j] = x2[j]
        i = j

    for i in range(len(y1)):
        if y1[i] == -1:
            y1[i] = x2[i]
            y2[i] = x1[i]

    return (y1, y2)
```

The partially-mapped crossover 2 (PMX2) strategy is a way to map some part of child with proper part of parent. The two crossover points are randomly chosen.

```python
# pmxCrossOver
def pmxCrossOver(p1,p2):
    y1 = [-1] * len(p1)
    y2 = [-1] * len(p2)

    a = random.randint(1, len(p1) - 1)
    b = random.randint(a, len(p1))

    for i in range(a, b):
        y1[i] = p2[i]
        y2[i] = p1[i]

    for i in (list(range(a)) + list(range(b, len(p1)))):
        if p1[i] in y1:
            t = p2[i]
            while (t not in p1[a:b]) or (t in y1):
                t = p2[p1.index(t)]
            y1[i] = t
        else:
            y1[i] = p1[i]
```

```
    for i in (list(range(a)) + list(range(b, len(p1)))):
        if p2[i] in y2:
            t = p1[i]
            while (t not in p2[a:b]) or (t in y2):
                t = p1[p2.index(t)]
            y2[i] = t
        else:
            y2[i] = p2[i]

    return y1, y2
```

I made my own, which chooses some part with randomly chosen starting point and randomly chosen length, and generate a child by mapping the part to another parent's genes containing same element as chosen one of the first parent with maintaining the order.

```
def myCrossOver(p1, p2):
    s = random.randint(0, len(p1)-1)
    e = random.randint(s, s + GENE_MAX_LENGTH)
    e = min(e, len(p1) - 1)
    np1 = p1[:]
    np2 = p2[:]
    exchangePart = p1[s:e+1]

    idx1 = s
    idx2 = 0
    while idx1 < e + 1:
        if np2[idx2] in exchangePart:
            np1[idx1], np2[idx2] = np2[idx2], np1[idx1]
            idx1 += 1
        idx2 += 1
    return (np1, np2)
```

You can select crossover algorithm by giving `-x` flag. If not given, cx is default strategy.

2.5 Mutation

Mutation happens very simply. It randomly chooses two nodes from the path and exchange them with the probability of r. You can give mutation rate by giving `-m` flag.

```python
# mutate: path, r -> path
# - mutate if p < r
def mutate(path, r):
    if random.random() < r:
        id1 = random.choice(path)
        id2 = random.choice(path)
        while id2 == id1:
            id2 = random.choice(path)

        idx1 = path.index(id1)
        idx2 = path.index(id2)
        path[idx1], path[idx2] = path[idx2], path[idx1]
    return path
```

2.6 Results

I compare and contrasts the strategies by using `a280.tsp` test file.

2.6.1 Different Population Size

Without changing any other strategies, I firstly change the population size and examine the performance.

Trial	Size = 10	Size = 100	Size = 1000
1	30460.455649649724	25352.835163090258	21041.18658386337
2	30729.02696903852	24939.693109770862	19314.39413706647
3	31683.85967369266	25002.99973184998	19986.05824402357
4	29757.224718621335	24733.104581769403	19453.74995919088
5	30482.94349892524	25341.2388277846	19189.361699694135
6	30781.89791055486	24803.997093830018	20149.337492665614
7	30018.89614246253	24574.891680816807	21160.422942459885
8	30672.166803238688	24847.465597062444	21514.984989394805
9	31109.233633432763	25525.746371621968	19000.181694585208
10	30872.232802828064	24175.81454369313	19808.01556129264
Mean	30656.7938	24929.7787	20061.7693

The results show that the larger the size is the shorter the distance is. This shows that the larger population has higher probabilty of making a good new population, and this is quite obvious result for genetic algorithm since it randomly generates initial population. Better gene can be generated randomly if we generate many.

2.6.2 Selection Strategies

With the population size 10 and size 1000, I compare and contrast two selection strategies, overselect and elitism. Since elitism is good for making local optima in short time and overselect is good for escaping the local optima, I choose two different population sizes which have big gap between them.

Trial	Elitism 10	Elitism 1000	Overselect 10	Overselect 1000
1	30510.194875667552	21616.246254442594	30460.455649649724	21041.18658386337
2	29279.592644474524	20484.561476339168	30729.02696903852	19314.39413706647
3	31183.97859062004	19428.42690080714	31683.85967369266	19986.05824402357
4	29796.39836439586	21286.108913483822	29757.224718621335	19453.74995919088
5	31875.82138852167	20364.306127363405	30482.94349892524	19189.361699694135
6	30882.78732962493	20548.12156375085	30781.89791055486	20149.337492665614
7	31683.44879050855	20740.65354475836	30018.89614246253	21160.422942459885
8	31764.152436486384	20028.290764121946	30672.166803238688	21514.984989394805
9	31405.15422500531	21174.842252258568	31109.233633432763	19000.181694585208
10	28465.795302109942	21449.55394124934	30872.232802828064	19808.01556129264
Mean	30684.7324	20712.1112	30656.7938	20061.7693

I think the table shows interesting results. For the smaller population, the result is not that different. Even for some trials, elitism overwhelms overselection. However, for the larger population, it shows a little difference. We might be able to say, overselection works for larger group.

In order to see the affect of time, I tried same experiment with not population size but the number of generation. 10 and 1000 are big different for the number of generation, so I expected I can see the meaningful difference.

Trial	Elitism 10	Elitism 1000	Overselect 10	Overselect 1000
1	30328.240003664523	14407.166153671325	30041.747013673696	14192.781275560304
2	30459.448668885045	14455.094888528356	30018.90631786845	14194.222175471314
3	29439.9270078868	14823.223997142872	29615.73423330859	14083.916770573058
4	30251.078506429414	15400.970945566358	30201.46528504951	14715.058855974881
5	29667.940096647286	14815.471576848886	30018.396171460336	14288.064754602869
6	30125.741034435126	14643.42720543723	29310.688018869536	14722.435966223005
7	29770.876315733636	14755.8422926035	29875.454444900017	14147.219644775054
8	31034.706129616465	14455.626629775428	30360.271005135113	14135.787863837482
9	29584.548985518322	15042.644712940515	30065.683227899404	13978.194673939834
10	30104.780853246684	14740.660006943404	31205.186684889366	14050.51880264258
Mean	30076.7288	14754.0128	30071.3532	14250.8201

Interestingly, or sadly, even the number of generation does not affect much to the selection algorithm. In conclusion, both generation and population size affect the selection algorithm little bit, and overselection makes shorter distance for both cases, but the difference was not big in both cases.

2.6.3 CrossOver Strategies

Without changing other variables, I examined three crossover strategies.

Trial	CX	PMX2	My Own
1	25352.835163090258	22571.49017244644	19269.256850288326
2	24939.693109770862	23382.24185649681	19027.871813272563
3	25002.99973184998	22005.260003693234	19658.50335392552
4	24733.104581769403	22365.456254633915	19667.68415827356
5	25341.2388277846	22401.252021651188	19892.36807519136
6	24803.997093830018	22755.408498791294	20052.63867263101
7	24574.891680816807	22877.140922180253	20490.284175843648
8	24847.465597062444	20942.105003045315	21071.46424847718
9	25525.746371621968	22112.87735018012	19905.79941010982
10	24175.81454369313	23116.77676596921	20664.955929152875
Mean	24929.7787	22453.0009	19970.0827

CrossOver experiment shows very and very interesting results. *I don't know why...* but my own strategy makes the best performance among cx, pmx2, and my own algorithm. I guess this is because of the feature of the dataset. So, I tried it once again with different dataset `att532.tsp`.

Trial	CX	PMX2	My Own
1	1330953.1915611054	1101964.5545933624	1149607.5829424844
2	1293981.4940828322	1166515.1067867687	1149568.20216079
3	1298075.0345198216	1180516.5668649592	1147990.7919919111
4	1302327.7208721414	1160887.0985638432	1125353.1535104103
5	1338449.4152270027	1209918.1285760172	1080512.3612170555
6	1313548.105376266	1152454.7271207392	1140845.7265408672
7	1283112.83597773	1250368.46920412	1101276.007987794
8	1307437.663505304	1160213.9918794332	1128387.7251622607
9	1320188.3696314606	1228162.9111157968	1160686.499887125
10	1315181.3331217095	1162941.3675968556	1112500.2120000038
Mean	1310325.52	1177394.29	1129672.83

Even for this dataset my algorithm makes better performance. This might be because of the small number of generation, so that my algorithm might accelerate selection for the early stages of evolution.

2.6.4 Mutation Rates

Without chainging other variables, I examined the effect of mutation rates. To show the affect, I selected 0.05, 0.5, and 1.

Trial	0.05	0.5	1
1	24771.860059512434	20036.09606684269	17891.016006750808
2	25281.151174722923	20830.187076556307	19174.3713364187
3	24807.884487502546	19912.1680467268	19413.816808463387
4	25551.723397809867	19487.87632149736	18917.30338384056
5	25308.399575240757	19396.925502599086	18771.399731605634
6	25511.663049045615	19846.984863860227	19085.076151436613
7	24745.020596720045	19956.225363605	18466.93846128218
8	25440.98455994306	20312.17824520457	18823.40850489367
9	24575.63612895698	20463.881049606374	19456.371370942266
10	25831.659765189946	20454.065790985558	18405.74611422217
Mean	25182.5983	20069.6588	18840.5448

The result for the mutation rates give another interesting insights. It shows a linear relationship between mutation rate and the distance, the bigger the mutation rate is, the shorter the result is. I guess this is because the small number of generation size. For the early steps of generations, mutation might work as one part of selection or crossover algorithm so that it accelerates the evolution.

3. Ant Colony Optimization

Ant Colony Optimization is a bio-inspired algorithm from ants. Since ants need to optimize their movements in their colony, they use pheromone. Instead of calculating the shortest distance or memorizing all the paths, they leave pheromone where they already passed. The leaving pheromone is determined by the distance of routes ants traveled with satisfying TSP condition, so that pheromone reinforces the path which leads the path shorter and shorter.

3.1 initializing

For the initialzing, I calculated all the possible distance of edges in `Anthill`, and by `begin` operation, we put ants at the random node in the anthill.

```
anthill = Anthill(nodes)
ants = [Ant(anthill, i) for i in range(ANT_NUMBER)]

for ant in ants:
  ant.begin()
```

3.2 generations

For each generation, ants travel the nodes under TSP conditions, which is *All nodes must be visited exactly once*. Because of this condition all the ants travel same length of route, so that we can find the end of travelling by checking one ant. At the end of each generation, we evaporate some pheromone. This helps to escape local optima.

```
for gen in range(GENERATION):
  while not ants[0].end():
    for ant in ants:
      ant.turn()
    anthill.evaporate()

    # reset
    for ant in ants:
      ant.begin()
```

3.3 ant moves

Basically, ants move to near nodes with the probability of:

$$p_{ij}^k = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{h\in J^k}^{H}\tau_{ih}^{\alpha}\eta_{ih}^{\beta}}$$

, where $\eta_{ij} = \frac{1}{l_{ij}}$ is the visibility of the node j from node i. Weights α and β indicate the weights of length of pheromone. J^k is a set of not visited nodes.

However, for my algorithm due to the different scale of length and pheromone, I re-scaled the distance into $[0, 1]$ range. I thought this is okay since I set a default value of the initial pheromone as 1 so the they are balanced. This calculation code is implemented in the code like below:

```
dist = anthill.edges[(i-1, current-1)]
eta = (anthill.weightMax - anthill.weightMin) / (dist - anthill.weightMin + 1)
tau = anthill.pheromone[(i-1, current-1)]

prob = (eta**WEIGHT_LENGTH) * (tau**WEIGHT_PHEROMONE)
```

3.4 Pheromone

The amount of pheromone ant leaves is calculated as

$$\triangle\tau_{ij}^k = \frac{Q}{L^k}$$

, where Q is the estimated length of shortest tour, and L^k is the length of tour of ant k. This pheromone is evaporated at the end of each generation with the equation of $\tau_{ij}^{t+1} = (1-\rho)\tau_{ij}^{t} + \triangle T_{ij}$.

3.5 Make a Result

ACO Algorithm only tells us how to update the wieght of edges using pheromones. So, we need to make another heuristic to make a result.

I made another ant named tester, which examines the path by following the edges just like other ants.

3.5.1 Tester Ant

The first method is to use a tester ant. Just like other ants, one ant travels at the end of all generations, and calculate the route.

However, I thought this method is not good to calculate the result of the ACO algorithm because this does not fully use the results of pheromone, and still depends on probablity.

3.5.2 MST Algorithm

So I applied MST algorithm here (This is modified version, see **chapter 5** of this report). Instead of edges, I used resulting pheromone to construct the minimum spanning tree.

3.6 Results

3.6.1 Tester Ant vs. MST Algorithm

In order to compare two resulting strategies, I tried a simple examination here. Because of the low running speed of the program, I used three ants and only ten generations.

Trial	Tester Ant	MST
1	29344.900515504803	5011.152731234156
2	31570.233159653544	5226.745089956175
3	32272.264801336354	5060.589234206545
4	31616.03777039962	5169.505968885088
5	30009.35924847931	5144.259770795619
6	29557.284122006185	5242.220592610473
7	30392.243853257718	5301.77732819374
8	30271.63617285205	5256.381032045911
9	30364.589177747304	5311.122735747752
10	31521.634659424817	5076.068229533991
Mean	30692.0183	5179.98227

This is somewhat obvious result, but I thought we do not have to depend on the probability at the end of algorithm, since there is no futher step of optimization.

3.6.2 Pheromone Depending Weights

I tried several experiment by controlling pheromone depending weights. 1 means ants entirely depend on pheromone, 0 means ants entirely depend on length. Note that 0 does not mean it is greedy algorithm because ants always choose the way based on the probability. For this test, I choose three values, 1, 0.5, and 0.

Trial	weight = 1	weight = 0.5	weight = 0
1	4843.7716611159385	5212.486673452476	7172.272063841112
2	4885.837124825724	5013.211876883836	7537.035290744606
3	4866.320821127782	5160.095072186534	7553.433035646136
4	4905.381306528206	5285.732278773989	7474.341857244992
5	4881.656896849474	5188.948932083258	7154.505787102157
6	4879.10952817189	5073.5432764749075	7342.27138801812
7	4841.279422375462	5179.611731956752	7323.690940886781
8	4868.411647416126	5180.886085300517	7707.111286945524
9	4829.814225990952	5048.586884890031	7613.015779922371
10	4870.277837036473	5219.242211347652	7081.725652944361
Mean	4867.18605	5156.2345	7395.94031

This gives me interesting insight, which pheromone is obviously working.

4. Greedy Algorithm

Greedy algorithm is a way to select currently best value at each step. For this algorithm, the starting node affects very much to the result, so that I randomly choose one node to start. However, you can give an initial node by `-i` flag.

```
def greedy_from(n, nodes):
    # 51, 1111492.9231858053 # 1 to 62
    path = [n]
    toVisit = list(nodes.keys())
    toVisit.remove(n)
    while len(toVisit) > 0:
        m = 999999999
        mIdx = -1
        for target in toVisit:
            dist = distanceBtw(nodes[target], nodes[path[-1]])
            if dist < m:
                m = dist
                mIdx = target

        toVisit.remove(mIdx)
        path.append(mIdx)
```

```
    return path
```

4.1 Results

I examined greedy algorithm with random starting node with `a280.tsp` and `att532.tsp`.

Trial	a280.tsp	att532.tsp
1	3296.364589364881	110014.87128448064
2	3417.4756412042807	110089.93383821366
3	3177.506070991219	111004.85477994182
4	3133.7552366506497	109946.67910100473
5	3307.514365995899	107503.32509092095
6	3233.598998788573	110016.81611787807
7	3203.4698583413447	107367.82695412835
8	3506.0688852020426	110917.43418611592
9	3176.204206298816	111022.91002522611
10	3204.2546561621757	105927.10649036754
Mean	3265.62125	109381.176

Compared to the other algorithms using same dataset, greedy algorithm shows much better performance. (even it is very fast!) However, as we all already know, greedy cannot ensure to find global optima.

5. Minimum Spanning Tree

> Note that this is deterministic algorithm.
>
> I do not write this algorithm as submission itself, but only used this algorithm for other algorithm, for example, the initial path for genetic algorithm or result maker for ant colony optimization.

I calculated the TSP by adding one simple condition to Prim's algorithm, which is *Degrees of all nodes are up to 2*. This small modification ensures that the result of modified Prim's algorithm satisfies TSP.

5.1 Results

	a280.tsp	att532.tsp	rl11849.tsp
Result	2960.4783808070056	106018.79430856417	1039269.8361605078

The result is even shorter than greedy algorithm, but this might be also one of local optima.

6. Conclusion

> Before I conclude something, it was very sorry not to have enough time to examine very large number of generations. Almost all the evolution algorithm could make good performance under late stage of generations, since the population can be thought as evolved. However, for my case, I cannot spend much time to reach that stage.

As the Results chapter of each algorithm said, the deterministic algorithm performs the best among four algorithms I used. (This is sad). However, As we already know, it is obviously falled in local optima, and this is proved by leaderboard. (I am not on the top) This obviously affects to the ACO algorithm with MST, so ACO with MST gave me the best performance except MST itself.

If I do not consider MST at all, I think the best performance is coming from Genetic Algorithm. The performance here also includes the running time, and GA's running time was much faster than ACO's running time. I think this is because I need to calculate $nAnts \times nGenerations$ for ACO algorithm unlike to GA which only calculates $nGenerations$ times.