COMP5511
AI Concepts
LIN, Sen
21042832g

Description:

The Traveling Salesman Problem (TSP) is a classical combinatorial NP-Hard problem. This problem is about a salesman who traverses *n* cities cyclically, starting from a city and ending at the same city, with all cities in the graph traversed exactly once. The distances between each city are different. For the project, I will consider symmetric TSP first, introduce the methodologies I used for my implementation of Genetic Algorithm from scratch in Python. Then, I will discuss the effects of changing the parameters of the GA. I will also discuss the effect on changing the size of the cities in the original data pool. In the extension section, asymmetric TSP and Sequential Order Problem will be discussed, according to the result produced by the code.

Methodology:

Parameters:

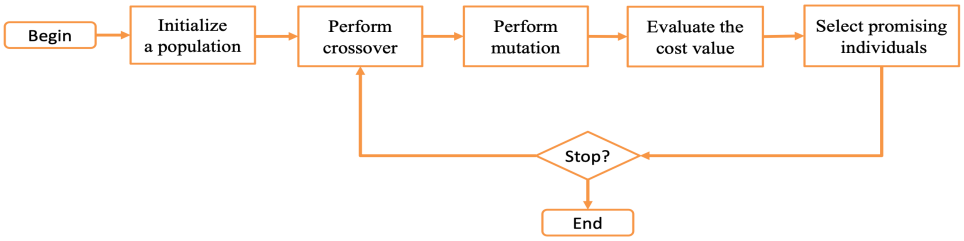
The implementation is written in Python 3.10, with the following default settings:

```
PROJECT_DEFAULT_DATA = {1: (0.3642, 0.7770),
                        2: (0.7185, 0.8312),
                        3: (0.0986, 0.5893),
                        4: (0.2954, 0.9606),
                        5: (0.5951, 0.4647),
                        6: (0.6697, 0.7657),
                        7: (0.4353, 0.1709),
                        8: (0.2131, 0.8349),
                        9: (0.3479, 0.6984),
                        10: (0.4516, 0.0488)}

DEFAULT_POP_SIZE = 30

DEFAULT_SELECTION_SIZE = 20

MUTATION_RATE = 0.03

MAX_ITERATION = 10000
```

PROJECT_DEFAULT_DATA: the default data given by the project description
DEFAULT_POP_SIZE: The population size of the project
DEFAULT_SELECTION_SIZE: The population size of the mating pool, aka the selected population
MUTATION_RATE: The default mutation rate of the population

General flow chart:

The implementation follows the given flow chart.



Encoding:

The encoding of the individual city is defined by number 1-10. The information of 10 cities is stored as tuple in the dictionary.

The chromosome is defined by a list of length 10, each element of list represents a city. The elements inside list of cities are distinct.

The random encoding is achieved by “random.sample()” with length of the Default data.

Initialize population:

Described by “create_default_populations()” in “actions.py”. This function will create a random, population pool with distinct chromosomes. The population pool size is pre-defined in the “defaultData.py”. The function will also calculate the fitness of the chromosome at the same times and pushes them into population pool as tuple.

For each iteration of the algorithm, the population pool will be sorted by ascending order of the cost before starting the iteration. The mating pool will be cleaned up in order to store the new selection results from new iteration.

Fitness:

The fitness function chosen here is simply the cost of the route. The cost of the route is calculated by the sum of the straight-line distance of the sibling cities in a chromosome. Fitness function will also calculate the distance of tail to head, to make chromosome a closed loop.

The calculation is illustrated by the code snippets below:

```
def calculate_distance(city1, city2):
    x = city1[0] - city2[0]
    y = city1[1] - city2[1]
    return math.sqrt(x ** 2 + y ** 2)

def fitness(route):
    score = 0
    for index, city_num in enumerate(route):
        next_index = index + 1
        if next_index == len(PROJECT_DEFAULT_DATA):
            next_index = 0
        next_city = route[next_index]
        score = score + calculate_distance(PROJECT_DEFAULT_DATA[city_num]
                                           , PROJECT_DEFAULT_DATA[next_city])
    return score
```

Selection and Crossover Rate:

The project performs selection using weight average selection by rank.

Before each selection session, the population pool will be sorted by cost (calculated by fitness()) in ascending order.

The probability of selecting this chromosome is defined by rank/sum_of_rank. By the definition of problem, the cost of the chromosome should be as small as possible. Therefore, for each chromosome, the smaller the cost, the larger the rank number is. The chosen chromosome will

be pushed into the mating pool. The number of chromosomes chosen is defined by the default parameter DEFAULT_SELECTION_SIZE which is also the size of the mating pool.
For each selection iteration, there will be a random floating number generated, denoted as "target". The population pool will be traversed from head, and the accumulated probability will be calculated by adding current probability of the current chromosome. Once the accumulated probability is larger than larger than "target", the chromosome is treated as chosen, and will be pushed into mating pool. At this time, current choosing iteration will be terminated, a new choosing iteration will start. If the chosen chromosome is duplicated with a member in the mating pool,
The choosing iteration will keep running until the size of the mating pool is the same as the default selection size.
After the selection session, the mating pool used for performing crossover will be ready. The pool will contain "DEFAULT_SELECTION_SIZE" of distinct chromosomes.
For the simplicity of my implementation, the crossover rate in this project will be replaced by: DEFAULT_SELECTION_SIZE/DEFAULT_POP_SIZE. It can be interpreted as, for default population generated, only (crossover rate) of members in the population are selected to mate.

Crossover:

Before talking about breeding strategy chosen, the crossover strategy should be talked about. The crossover method used is recombination for order-based representation. There will be selected parents. Two different cross points generated when crossover, and the gene snippet will be cut between those two points from parent1. Then get_offspring() function will combine the cut gene with parent2, by filling the elements in parent2, except those elements already appeared in the gene, from the tail of the gene by order. The code snippets are shown below to provide a better explanation of the crossover logic used in this project.

```
def crossover(parent1, parent2):
    # cross point
    cp = random.randint(0, 9)
    while True:
        cp2 = random.randint(0, 9)
        if cp2 != cp:
            break
    if cp < cp2:
        start = cp
        end = cp2
    else:
        start = cp2
        end = cp
    gene = parent1[start:end]
    return get_offstring(gene, start, end, parent2)
```

```
# crossover uses order based representation
def get_offstring(gene, start, end, parent):
    parent_copy = list(parent)
    orig_length = len(parent_copy)
    result = [0] * orig_length
    for index, item in enumerate(gene):
        parent_copy.remove(item)
        result[start + index] = item
    for index2, item2 in enumerate(parent_copy):
        filling_index = end + index2
        if filling_index >= orig_length:
            filling_index = filling_index - orig_length
        result[filling_index] = item2
    return result
```

Breed:

The breeding process for the project is: for each of the member in mating pool, the member will select a random member inside mating pool, excluding itself. The crossover result will be pushed into children pool, and the number of crossover result will be the same as mating pool, which is the length of the children pool.

Mutation:

The mutation rate is defined in “defaultData.py”. For each child in children pool, the system will generate a random floating number within [0,1] for it. Once the random number is smaller than the mutation rate, the mutation will occur for that specific child

The mutation strategy used is simple list swapping. The random generator will generate two different numbers within the length of the route, and swap two positions.

Elitism:

After all the previous steps are down, the algorithm will pop out the lower ranked population from the population pool and append the members children pool to the population pool with its fitness() to form a new, better population pool. A full iteration is completed at this point.

Result:

The result below is based on 10-TSP problem.

Numeric result:

The result converges at around 2.5832. The result is verified by simulation with large iteration settings. i.e., setting the maximum iteration to 50000.

```
49998 ([3, 7, 10, 5, 6, 2, 9, 1, 4, 8], 2.5832228012801766)
49999 ([3, 7, 10, 5, 6, 2, 9, 1, 4, 8], 2.5832228012801766)
Simulation takes: 0:02:25.290043
```

Convergence:

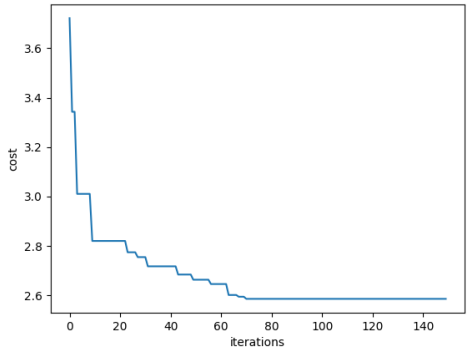
For 10 cities TSP, the result converges generally before 200 iterations with:

```
# default population size of GA
DEFAULT_POP_SIZE = 50

# default mating pool size of GA
DEFAULT_SELECTION_SIZE = 30

# default mutation rate of GA
MUTATION_RATE = 0.03

# maximum iteration of the simulation
MAX_ITERATION = 200
```

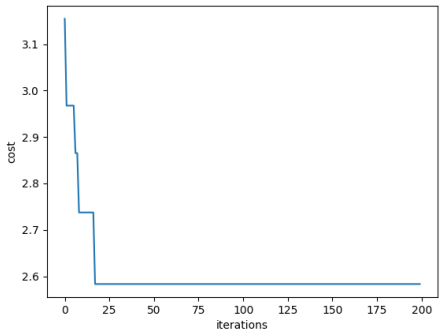
The convergence speed is random due to the fact that the population pool is randomly generated. But generally, it will go down and converges to 2.5382 before 200th iteration.

```
198 ([3, 7, 10, 5, 6, 2, 9, 1, 4, 8], 2.5832228012801766)
199 ([3, 7, 10, 5, 6, 2, 9, 1, 4, 8], 2.5832228012801766)
Simulation takes: 0:00:00.147936
```

A typical convergence diagram under this setting looks like:



Sometimes, the simulation converges faster, for example:



One reasonable guess of the fast convergence is because of the samples in the population pool is randomly chosen. The samples generated in this run may have good quality already, so the children they produce will have better quality at the early stage.

Routes:

The best route is not unique. From the parameters above, the following routes all have the best cost.

[3, 7, 10, 5, 6, 2, 9, 1, 4, 8] 3

[1, 9, 2, 6, 5, 10, 7, 3, 8, 4] 1

[9, 1, 4, 8, 3, 7, 10, 5, 6, 2] 9

From observations, we can find that for all the best children, they all have the same two cities' combinations. Since we treated this problem as a symmetric problem, the distance between two cities is the same, regardless of the direction.

For example, for the first route above, the salesman starts from city 3 -> 7, in second and third routes listed above, we can all find 3->7 and 7->3. Then, for the next path 7->10, the second and third routes also have 7->10 and 10->7.

So, for the given problem, the number of routes is based on the connections of the following tuples

(3, 7), (7, 10), (10, 5), (5, 6), (6, 2), (2, 9), (9, 1), (1, 4), (4, 8), (8, 3)

Discussion:

Change of Default Population Size:

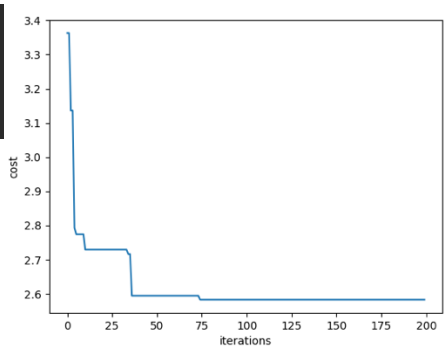
Under 10 cities TSP problem, the first parameter to discuss will be the default size of the population pool.

```
# default population size of GA
DEFAULT_POP_SIZE = 100

# default mating pool size of GA
DEFAULT_SELECTION_SIZE = 50

# default mutation rate of GA
MUTATION_RATE = 0.03

# maximum iteration of the simulation
MAX_ITERATION = 100
```

For this experiment, the parameter settings are shown below:

The default population size is set at 100 and 200, respectively. Here is the simulation result:

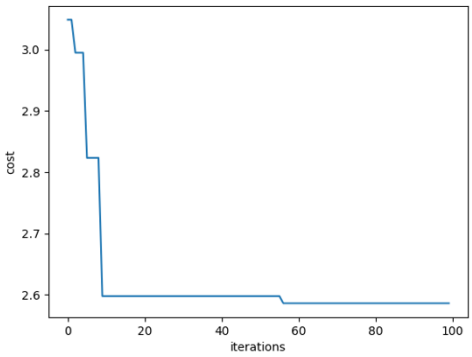
```
98 ([4, 1, 9, 2, 6, 5, 10, 7, 3, 8], 2.5832228012801766)
99 ([4, 1, 9, 2, 6, 5, 10, 7, 3, 8], 2.5832228012801766)
Simulation takes: 0:00:00.152004
```



When the population size is at 100, the algorithm converges quickly, and the graph shows that the convergence point will be before 100th iterations.

When the population size increases to150, the convergence point will generally be later than when the population size is 100, and the best solution is not guaranteed at 100th iterations.

```
98 ([3, 8, 4, 1, 9, 2, 6, 5, 7, 10], 2.5862176675048807)
99 ([3, 8, 4, 1, 9, 2, 6, 5, 7, 10], 2.5862176675048807)
Simulation takes: 0:00:00.140474
```



For the above simulation result and graph, we can see that when population size increases to 150, GA needs more iterations to find the best result. This may be based on the crossover rate. When population size increases, the crossover rate actually goes down.
From above observations, we can conclude that, the number of initial populations of GA does matter, it will affect the iterations needed for GA to find the best result. The more initial populations, the more iterations needed for the simulation.

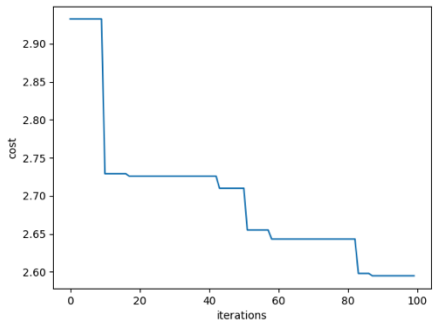
Change of Default Selection Size

For default selection size, as explained in the methodologies section, it an interpretation the crossover rate of the Genetic Algorithm. Crossover rate is equal to DEFAULT_SELECTION_SIZE/DEFAULT_POP_SIZE. In this experiment, the following parameters will be used:

```
# default population size of GA
DEFAULT_POP_SIZE = 150

# default mating pool size of GA
DEFAULT_SELECTION_SIZE = 25

# default mutation rate of GA
MUTATION_RATE = 0.03

# maximum iteration of the simulation
MAX_ITERATION = 100
```

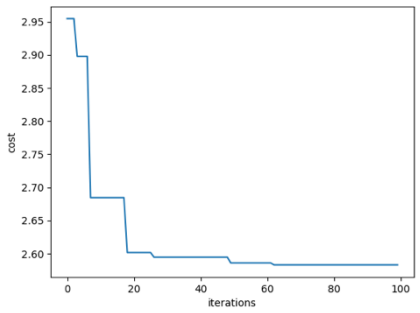
In this simulation, the default selection size chosen will be 25 and 100, respectively, to clearly see the effect brought by different cross rates. The length of the cities list is still 50.
When DEFAULT_SELECTION_SIZE is 25 the simulation result is:

```
98 ([6, 9, 1, 4, 8, 3, 7, 10, 5, 2], 2.594919685833151)
99 ([6, 9, 1, 4, 8, 3, 7, 10, 5, 2], 2.594919685833151)
Simulation takes: 0:00:00.122392
```



When DEFAULT_SELECTION_SIZE is 100, the simulation result is shown below:

```
98 ([10, 5, 6, 2, 9, 1, 4, 8, 3, 7], 2.5832228012801766)
99 ([10, 5, 6, 2, 9, 1, 4, 8, 3, 7], 2.5832228012801766)
Simulation takes: 0:00:00.240401
```



From two plots, we can see that DEFAULT_SELECTION_SIZE will affect the result of the simulation drastically. When the DEFAULT_SELECTION_SIZE is small (25), the simulation does not give the optimal route before 100th iteration. At this point, the crossover rate is low, which makes sense for the result to be non-optimal. When the crossover rate is high, aka when DEFAULT_SELECTION_SIZE of the simulation is large (100), we can see that the simulation converges fast, and much faster than the simulation with DEFAULT_SELECTION_SIZE = 25. In this case, the simulation will give optinal results before 100th iteration.

From the above two sections (Default population size and default selection size), we can conclude that with higher crossover rate, the simulation will converge faster. When crossover rate is higher, more populations will be chosen to mate, and more high-quality children will be produced each round. In this case, higher quality descendants are more likely to be produced.

More Cities

In order to simulate the case of more cities, I introduced 10 more random generated cities into the data list, together with the 10 given cities to form new data set. In "defaultData.py". In the code, the tester can switch between old data and new data by comment out either one.

The default setting of the simulation of more cities is:

```
# default population size of GA
DEFAULT_POP_SIZE = 150

# default mating pool size of GA
DEFAULT_SELECTION_SIZE = 100

# default mutation rate of GA
MUTATION_RATE = 0.03

# maximum iteration of the simulation
MAX_ITERATION = 20000
```

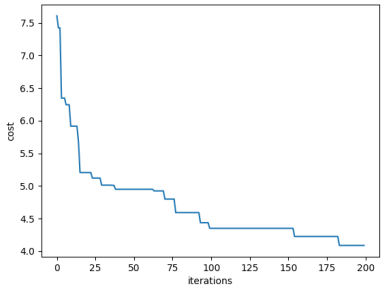
The parameter setting is the same as *Change of Default Selection Size* section: The simulation result is demonstrated as follow:

```
19998 ([15, 8, 11, 4, 17, 14, 2, 6, 18, 20, 5, 19, 16, 10, 12, 7, 13, 3, 9, 1], 3.5709498661950074)
19999 ([15, 8, 11, 4, 17, 14, 2, 6, 18, 20, 5, 19, 16, 10, 12, 7, 13, 3, 9, 1], 3.5709498661950074)
Simulation takes: 0:00:58.803337
```

The result is given by 20000 iterations.

From previous simulation results of 10-cities TSP, 10-cities TSP will give optimal result before 200 iterations. However, for 20-cities TSP, 200 iterations are not enough for 20-cities TSP to find the best solution:

```
198 ([4, 17, 14, 2, 18, 6, 20, 12, 7, 10, 16, 19, 5, 13, 3, 9, 1, 15, 8, 11], 4.088739420038408)
199 ([4, 17, 14, 2, 18, 6, 20, 12, 7, 10, 16, 19, 5, 13, 3, 9, 1, 15, 8, 11], 4.088739420038408)
Simulation takes: 0:00:00.446653
```



From the previous plot, we can see that the 20-cities TSP is still far from convergence at 200 iterations. But for 10-cities TSP, from the previous result, it only takes around 25 iterations to find the best solution. From the result of two simulations, we can conclude that when the number of cities increases in the datset, the number iterations required to get the optimzed result increases drastically.

From many runs of simulations, the GA general gets its optimal route after 8000 iterations.

```
7999 ([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074)
Simulation takes: 0:00:20.968927
```

Comparing to around 25 iterations for 10 cities, we can conclude that TSP is a NP-Hard question. As we all know, TSP is in O(n!). It makes sense for GA to spend much more iterations to find the optimal answer.

Also, after running GA multiple runs for more cities, I found that for larger number of cities, the GA is more unlikely to find distinct result. When the data size is larger, GA is more likely to breed the same result. Like the screenshots captured below, GA will produce the same result many times, until it breeds a new result.

```
multiple_runs() › for i in range(0, count)
main
The number of distinct best solutions found so far is: 4
[([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074),
The number of distinct best solutions found so far is: 4
[([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074),
The number of distinct best solutions found so far is: 4
[([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074),
The number of distinct best solutions found so far is: 4
[([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074),
The number of distinct best solutions found so far is: 4
[([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074),
The number of distinct best solutions found so far is: 4
[([5, 20, 18, 6, 2, 14, 17, 4, 11, 8, 15, 1, 9, 3, 13, 7, 12, 10, 16, 19], 3.5709498661950074),
```

However, with shorter cities list, GA only takes a couple runs to find distinct results. Also, for running the same number of iterations, it takes much shorter time. It's easy to explain this observation, since TSP is a NP-Hard question, and the running time for TSP is O(n!).

```
[([5, 6, 2, 9, 1, 4, 8, 3, 7, 10], 2.5832228012801766), ([2,
The number of distinct best solutions found so far is: 14
[([5, 6, 2, 9, 1, 4, 8, 3, 7, 10], 2.5832228012801766), ([2,
The number of distinct best solutions found so far is: 14
[([5, 6, 2, 9, 1, 4, 8, 3, 7, 10], 2.5832228012801766), ([2,
Simulation takes: 0:00:05.595772
```

Extensions

Asymmetric TSP

The asymmetric TSP calculation in this project is defined as follows:

If the start city number is smaller than destination number, the coordinates for both cities are retrieved from PROJECT_DEFAULT_DATA

If the start city number is larger than destination city number, the coordinates for both cities are retrieved from EXT_ASYM_DATA (defined in extensionData.py)

The parameters setting is the same as symmetric 10-cities simulation above:

```
# default population size of GA
DEFAULT_POP_SIZE = 50

# default mating pool size of GA
DEFAULT_SELECTION_SIZE = 30

# default mutation rate of GA
MUTATION_RATE = 0.03

# maximum iteration of the simulation
MAX_ITERATION = 200
```

Result:

With the random generated data (EXT_ASYM_DATA), one of the optimal routes found is:

```
49998 ([9, 2, 6, 8, 7, 10, 5, 1, 3, 4], 2.615606846204271)
49999 ([9, 2, 6, 8, 7, 10, 5, 1, 3, 4], 2.615606846204271)
Simulation takes: 0:00:17.906882
```

The above result is verified by setting the maximum iterations to 50000:

Discussion

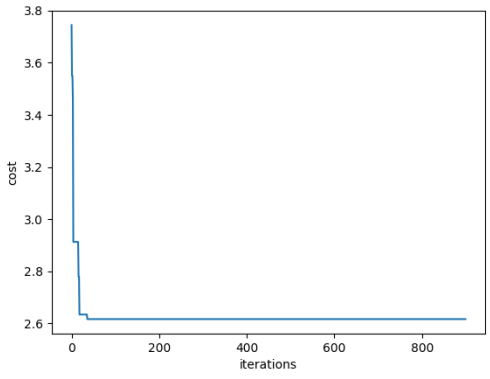
However, using the above parameters, we find that the above parameter cannot guarantee an optimal answer within 200 iterations. With 200 iterations, there are many answers with different cost as shown by the screen captures below.

```
198 ([4, 9, 2, 6, 8, 7, 10, 5, 1, 3], 2.6156068462042716)
199 ([4, 9, 2, 6, 8, 7, 10, 5, 1, 3], 2.6156068462042716)
Simulation takes: 0:00:00.152355
```

```
198 ([4, 9, 2, 6, 7, 10, 5, 1, 3, 8], 2.621723176742725)
199 ([4, 9, 2, 6, 7, 10, 5, 1, 3, 8], 2.621723176742725)
Simulation takes: 0:00:00.148602
```

After trying many numbers of iterations (in hundreds), the asymmetric TSP in this project will likely guarantee a best result when the maximum iteration is set to 900.

```
898 ([8, 7, 10, 5, 1, 3, 4, 9, 2, 6], 2.615606846204271)
899 ([8, 7, 10, 5, 1, 3, 4, 9, 2, 6], 2.615606846204271)
Simulation takes: 0:00:00.380376
```



Even from the plot given, we can conclude that, asymmetric TSP converges early, but the number of iterations that can guarantee a stable result is very large. From here, we can deem

that asymmetric is much more complicated than symmetric TSP that can produce a stable result before 200th iterations with same parameters.

Sequential Ordering Problem (SOP)

For SOP, there is a field definition "EXT_PATTERN" in "extensionData.py".
The implementation of SOP is available in the code.

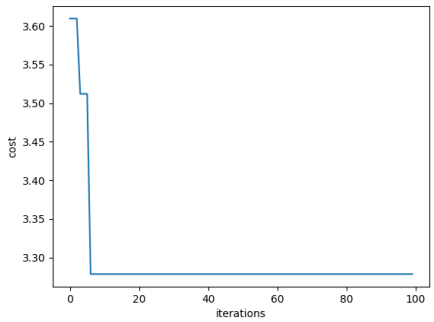
Result

```
1996 ([4, 8, 9, 1, 2, 3, 7, 10, 5, 6], 3.278513233349771)
1997 ([4, 8, 9, 1, 2, 3, 7, 10, 5, 6], 3.278513233349771)
1998 ([4, 8, 9, 1, 2, 3, 7, 10, 5, 6], 3.278513233349771)
1999 ([4, 8, 9, 1, 2, 3, 7, 10, 5, 6], 3.278513233349771)
Simulation takes: 0:00:00.771949
```

For SOP problem, in this simulation, the selected pattern is "[1,2,3]", the best result found is shown at snipped.

```
97 ([8, 9, 1, 2, 3, 7, 10, 5, 6, 4], 3.278513233349771)
98 ([8, 9, 1, 2, 3, 7, 10, 5, 6, 4], 3.278513233349771)
99 ([8, 9, 1, 2, 3, 7, 10, 5, 6, 4], 3.278513233349771)
Simulation takes: 0:00:00.111760
```

```
12 ([10, 5, 6, 4, 8, 9, 1, 2, 3, 7], 3.2785132333497713)
13 ([4, 8, 9, 1, 2, 3, 7, 10, 5, 6], 3.278513233349771)
14 ([4, 8, 9, 1, 2, 3, 7, 10, 5, 6], 3.278513233349771)
```



The result is not unique, and from the experiment, the result will be optimized before 100th iteration, and likely to be convergence before iteration 25th. The reason for this observation is that: since the pattern of the population is defined already, the possible distinct population member will be smaller. Therefore, SOP will converge even faster.
Another fact for the reduced complexity is that if the 10 cities SOP runs many times (>5000). The population pool will not have enough distinct candidates anymore. In this case, the mating pool are not able to be fully filled.

Conclusion:

From the above research and simulations, we can conclude that for TSP problem, the speed and cost is affected by crossover rate. Also, when increasing the population of the data, the number complexity of the problem increases in factorial and the number of iterations increases factorially. Additionally, for asymmetric TSP problem, the complicity will increase due to the different cost for return route. But for Sequential Order Problem, the complexity of the

problem is reduced, since the required pattern is already set, the available distinct route combinations will be lesser.