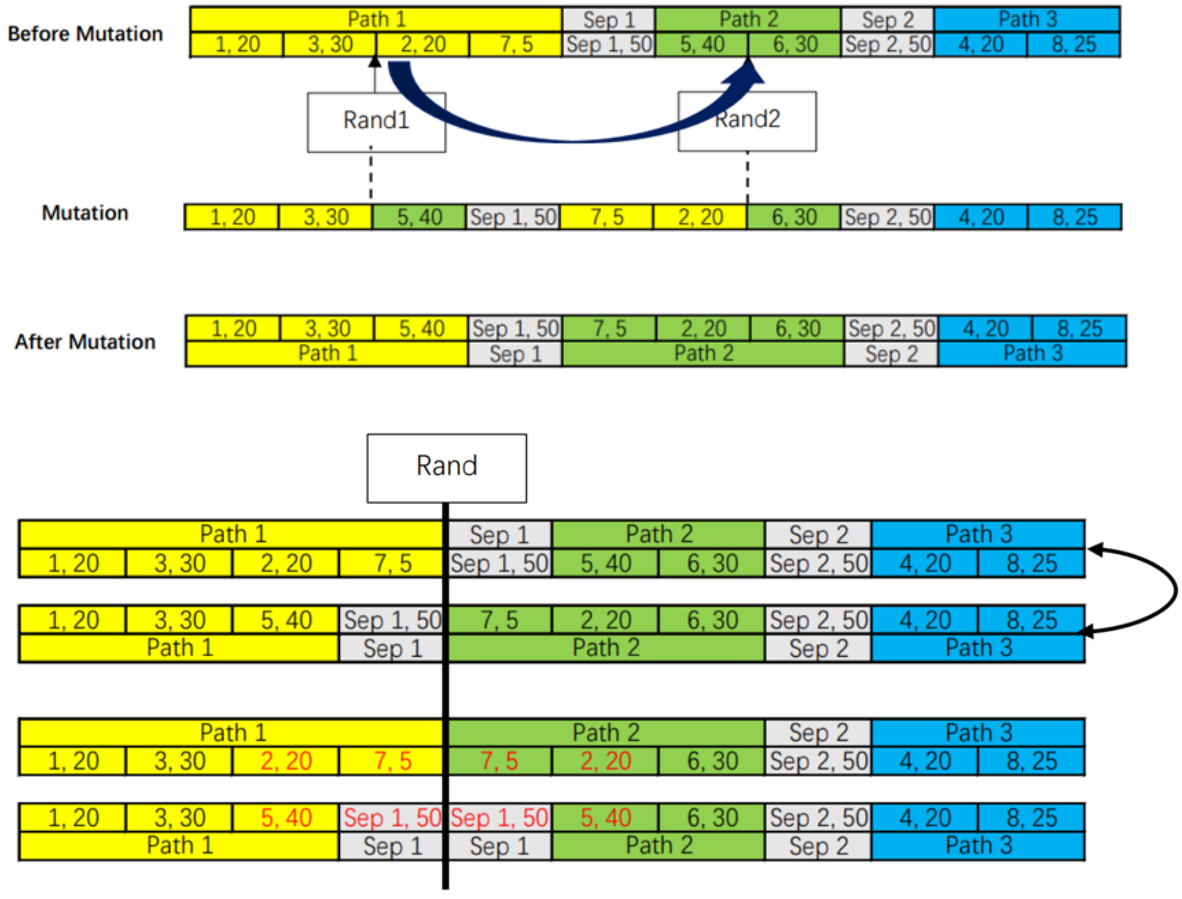
Practice of Genetic Algorithms in VRP Problems

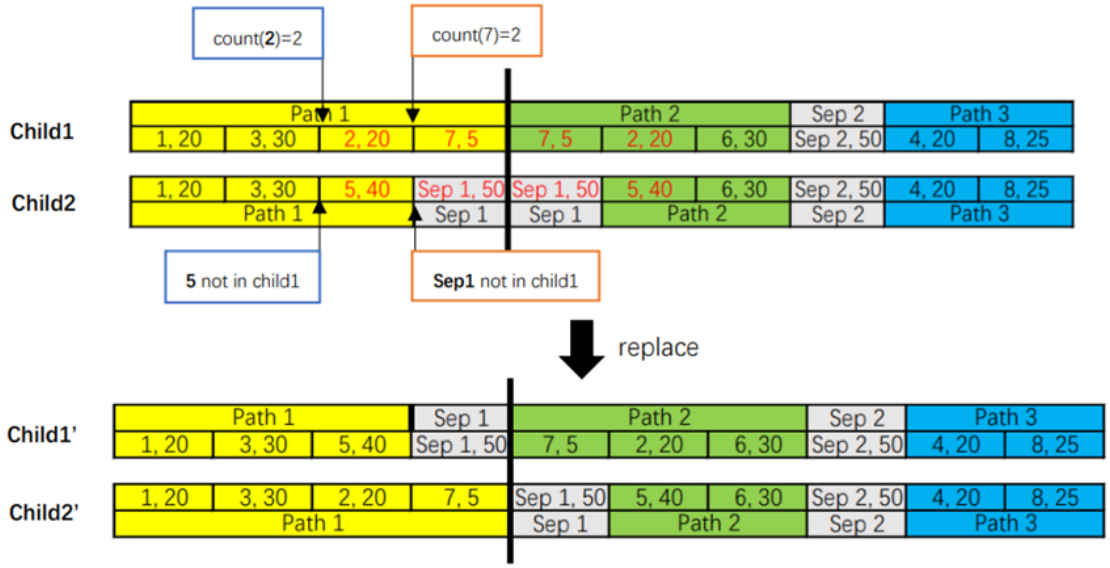
Abstract

The vehicle routing problem (VRP) is a very challenging task. In this paper, we use genetic algorithm to solve VRP.

First, we designed the gene and chromosome forms, and we can prove that our designed forms can satisfy the three criteria of genetic algorithm gene: **Completeness**, **soundless** and **no redundancy**.

Second, we determined the chromosome mutation in a way and the method of chromosome crossover and designed an adjustment method for the presence of illegal chromosomes in the crossover:





Then, we designed the decode method of chromosome and the fitness calculation method of chromosome.

Finally, to validate our model, we did a series of experiments with control variables and found the optimal parameters of the model, which proved that our model can effectively reduce the cost of transportation, and our optimal parameters can be optimized by 10% on top of previous.

Requirements:

- Python 3.x
- Numpy
- Pandas
- Matplotlib

Training the model

After running all the class and function code, the model is trained through the following interface:

```
result = execute_algorithm(k, need, fitness, nG = 200, nP = 100, pD=0.8, pM = 0.05)
```

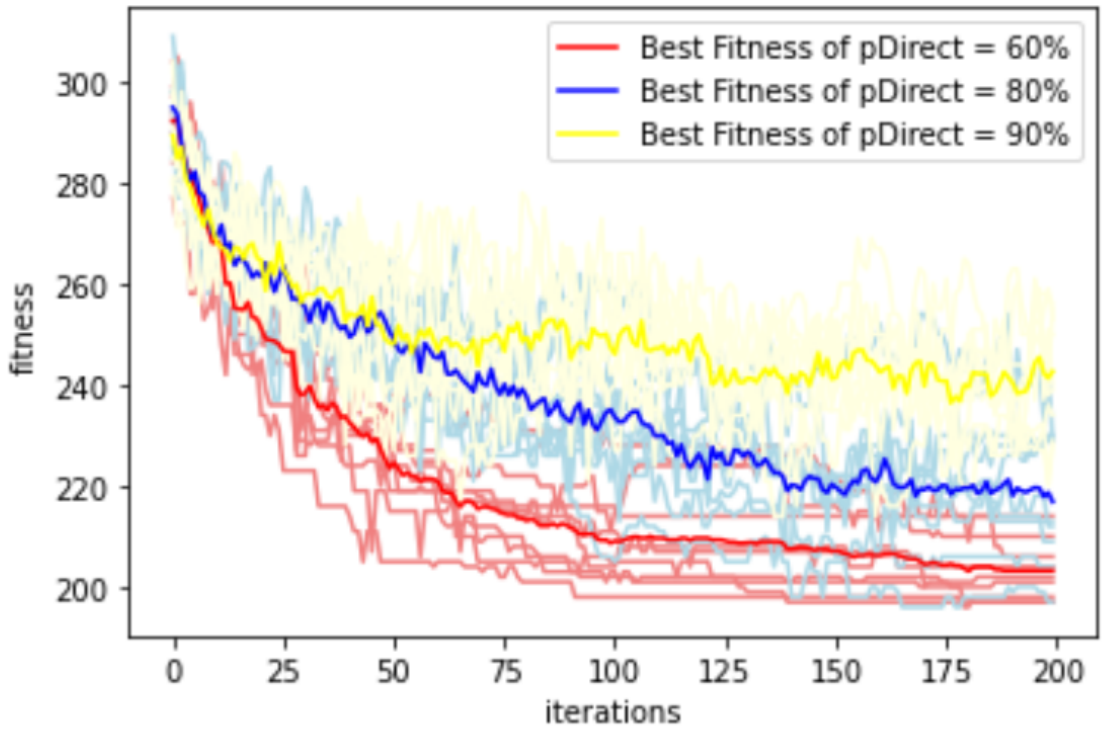
where k is the number of instances running simultaneously, need is the list of city demands, fitness is the fitness function, nG is the number of generations of model training, default is 200, nP is the number of individuals in the population, default is 100, pD is the proportion of the population that goes directly to the next generation, default is 80%, pM is the probability of individual chromosome variation, default is 5%, and the above various parameters can be modified.

Control Variates

For control Variates, we encapsulate a comparison function for it, with the following usage:

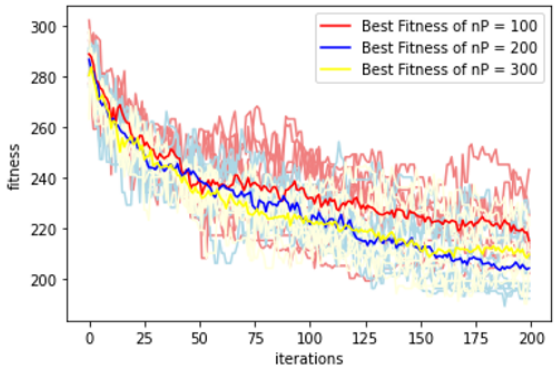
```
# Control Variates for pD

# pDirect = 60%
pd6 = execute_algorithm(genetic_problem_instances, need, lambda y: fitness(y),
pD=0.6)

# pDirect = 80%
pd8 = execute_algorithm(genetic_problem_instances, need, lambda y: fitness(y),
pD=0.8)

# pDirect = 90%
pd9 = execute_algorithm(genetic_problem_instances, need, lambda y: fitness(y),
pD=0.9)

# comparison function
comp_res([pd6, pd8, pd9], ['pDirect = 60%','pDirect = 80%','pDirect = 90%'])
```

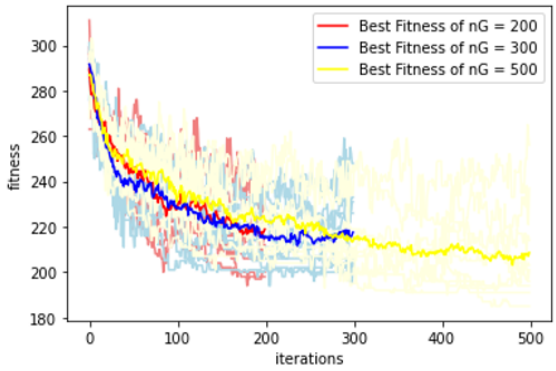
The details of each run of the model are returned as the ***execute_algorithm*** function, so different variables are needed to accept the returns (pd6/8/9), after which these returns is entered into comp_res to generate the comparison results for them, the results of pd 6/8/9 are as follows:

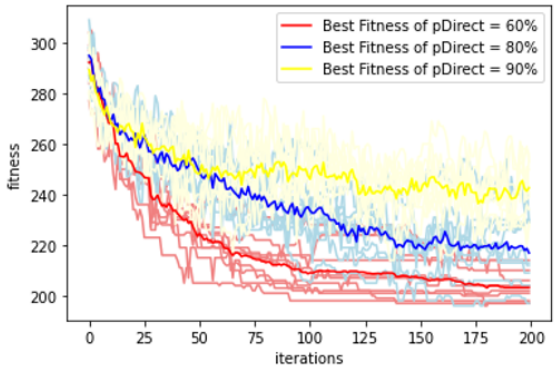


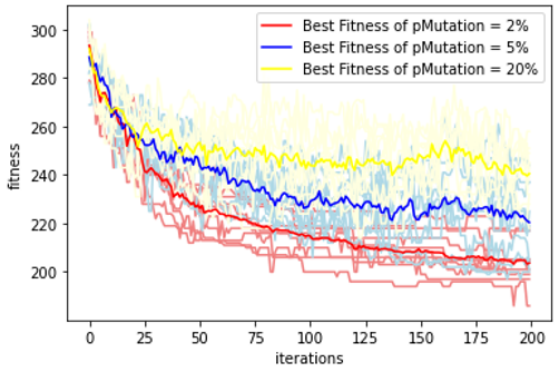
Variable: pDirect	60%	80%	90%
Max Best Fitness	214	230	256
Min Best Fitness	197	197	228
Mean Best Fitness	203.2	216.9	242.7
Max Poplution Fitness	229.4	280.5	302.8
Min Poplution Fitness	208.6	223.4	265.8
Mean Poplution Fitness	218.2	250.9	286.9

Results

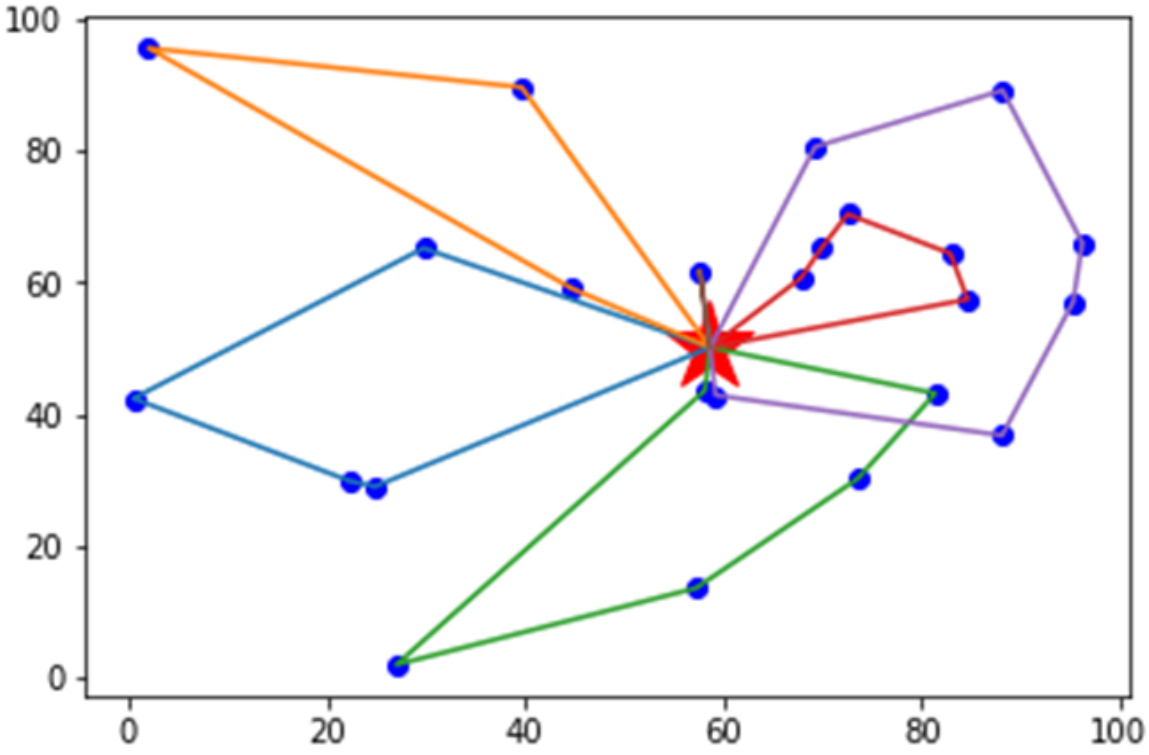








nP	Best	Best_Np	Best_nG	Best_pD	Best_pM
Max Best Fitness	**208**	226	256	214	218
Min Best Fitness	**187**	192	185	197	186
Mean Best Fitness	**194.4**	204	208.6	203.2	203.7
Max Population Fitness	**214**	270	295.9	229.4	230.2
Min Population Fitness	**195**	221.4	216.7	208.6	204.5
Mean Population Fitness	**202**	245.1	240.1	218.2	214.5


100
80
60
40
20
0
0
20
40
60
80
100