# Genetic Algorithm for TSP

Shynar Torekhan

TSP is of critical importance for many real life problems. Here, I propose a model based on a genetic algorithm for generating inputs, which can find relatively good solution in significantly less time when compared with brute force approaches. First, design decisions will be described. Secondly, decisions on choice of GA operations will be described, concluding with hyper parameter tuning and bench-marking.

## I. Design decisions

I tried to make code as modular as possible. I created system of classes like Instance, City, Genetic Algorithm, where each class encapsulates methods modifying their fields, which made development easier and efficient. For example, because fitness of an Instance is calculated upon initialization there is no need to recompute fitness each generation for all instances, because some instances came from previous generation. Also, because each Instance holds its fitness value, there is no need to compute fitnesses again during tournament selection, which saves computation time. Program provides interface for controlling parameters. The parameters can be changed in code(the 'default' part), as shown in the Figure 1, or can be manipulated through command line by calling 'python main.py -p path_to_file -s population_size -ms mating_pool_size' and etc. The detailed description of input arguments can be seen in Figure 1 in 'help' part.

For better visualization of results, there is logging feature called by 'python stats.py'. It will plot fitness versus generations graph, which will give a sense whether solution is converging. The example of output graph is presented on Figure 2.

```
# parses command line arguments
def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('-p', type=str, default="a280.tsp", help="path to the input file")
    parser.add_argument('-s', type=int, default=50, help="population size")
    parser.add_argument('-ms', type=int, default=25, help="mating pool size")
    parser.add_argument('-ts', type=int, default=7, help="tournament size")
    parser.add_argument('-e', type=int, default=15, help="elite_size")
    parser.add_argument('-mg', type=int, default=50, help="max generations")
    parser.add_argument('-cr', type=float, default=0.3, help="crossover rate")
    parser.add_argument('-mr', type=float, default=0.3, help="mutation rate")
    args = parser.parse_args()
    return args.p, args.s, args.ms, args.ts, args.e, args.mg, args.cr, args.mr
```

FIG. 1. Parameter control interface

## II. Methodology

For high iteration and quick trial and error, all the tests were made on small population, small number of generations and small instance of cities(a280.tsp). During testing of the influence of one parameter change, all other parameters were frozen.

### A. Mutation

First, I implemented the most basing swapping mutation. Later, I started exploring papers on the best

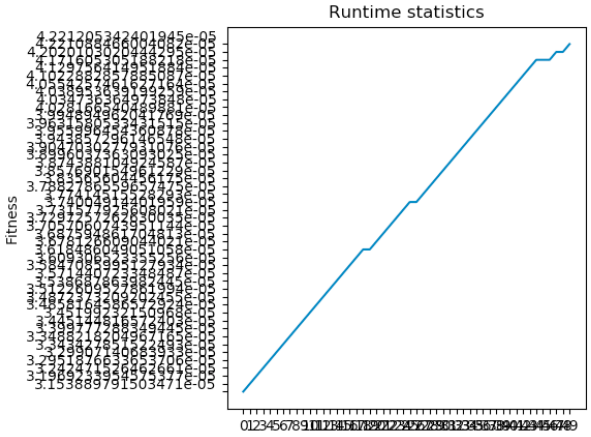


FIG. 2. Logging interface

techniques of mutation. Paper [1] benchmarked several mutation operators presented in figure 3. I implemented the RSM mutation operator as it shows the best results in the graph. However, when I benchmarked RSM with mutation operator, I didn't get significant increase in the performance. So, I decided to keep swapping operator, as it is much less computationally expensive than RSM mutation operator. Both of implementations are available in the code submitted.

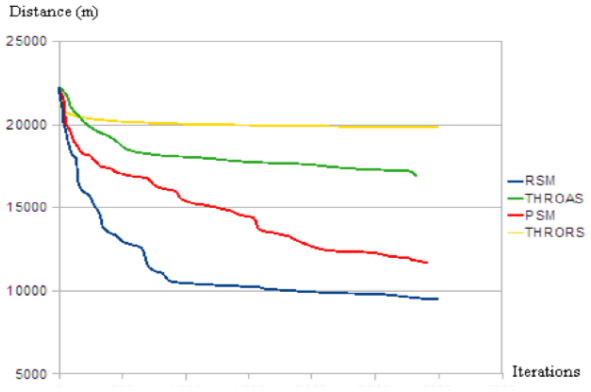


FIG. 3. Comparison of mutation operators [1]

Then, I tuned the mutation rate hyper parameter. Experiments can be seen in Table 1. Thus, mutation rate

of 0.3 was chosen.

| Mutation rate | Percentage increase in average fitness |
|---|---|
| 0.1 | 28.7% |
| 0.2 | 37% |
| 0.3 | 45% |
| 0.4 | 32.3% |

TABLE I. Mutation rate comparisons

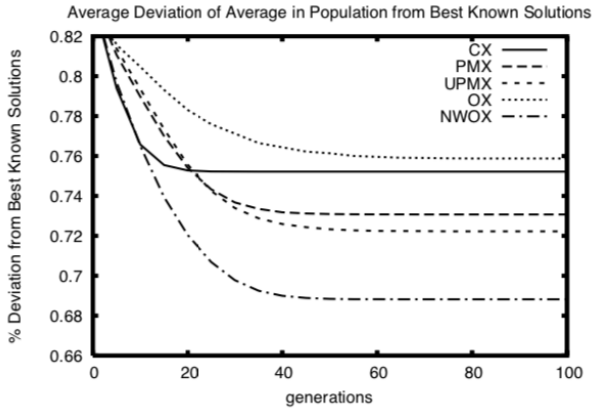


FIG. 4. Comparison of crossover operators [5]

### B. Fitness function

Fitness function is inverse proportional to the distance of the route. So, the shorter the route, the higher the fitness.

### C. Selection

Tournament selection was selected as based on the research results of paper which benchmarked different selection methods for TSP [2]. According to it, tournament selection is the best.

### D. Crossover

First, I implemented XO crossover operator, as it is widely used in the papers. According to [5] the best crossover operator is NWOX, however, its time complexity is bigger when compared with XO. So, I decided to keep XO crossover operator. The benchmarks are shown in figure 4.

### E. Elitism

I experimented with having elitism. Average fitness increase with elitism is 45% when compared to 36% increase without elitism. So, it was decided to introduce the elitism.

### F. Initialization

I explored papers about TSP initialization. In [4] it is written: "As final conclusion of this research it can be highlighted the efficiency of using heuristic initialization functions. Anyway, the excessive use of them could decrease the exploration capacity of the GA, trapping the population in local optimums quickly." So, it was decided to use random initialization

### G. Population size

The bigger the population, earlier the convergence happens. Test with population of 400 significantly outperformed test run with population of 50.

### H. Logging Feature

For better visualization of results, there is logging feature called by 'python stats.py'. It will plot fitness versus generations graph, which will give a sense whether solution is converging. The example of output graph is presented on Figure 2.

[1] Benchmark of different mutation operators
[2] Benchmark of selection operators for TSP
[3] Tutorial on GA
[4] Discussion on initialization
[5] Crossover operations benchmark for TSP