CMPT 310 Artificial Intelligence Survey

SFU Surrey Campus

Spring 2020

Final Project

Solving the TSP with Genetic Algorithms

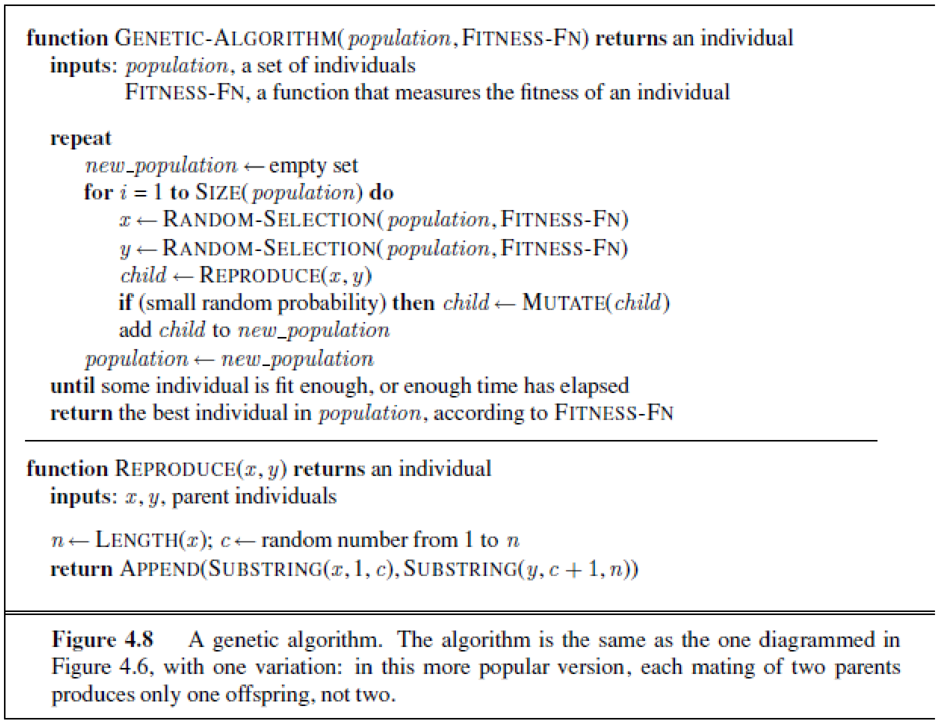
Student Name: Zeyong Jin

SFU ID: 301353174

Computing ID: zeyongj@sfu.ca

1. Introduction of The Genetic Algorithm Framework

In this project I wrote a class called GA_TSP in *tsp.py*, which implemented the genetic algorithm framework for solving the TSP problem. Basically, the framework is modified based on the pseudocode written in Figure 4.8 of the Norvig and Russel textbook.



```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

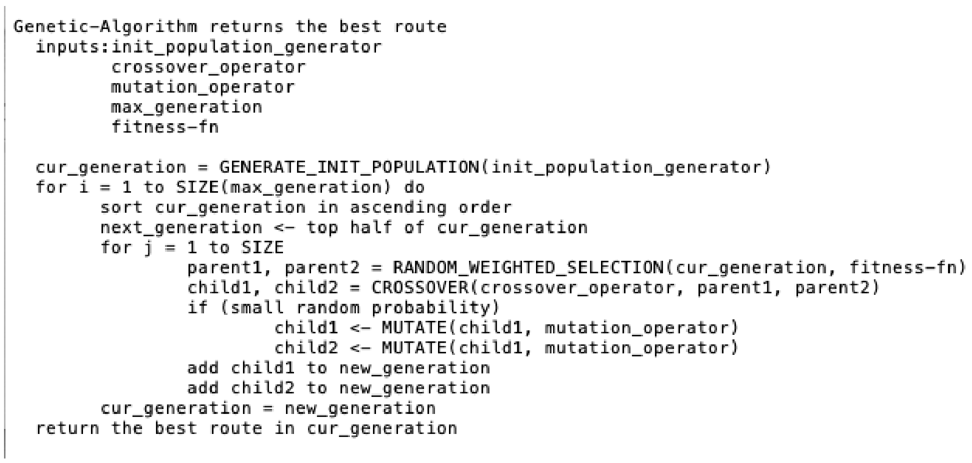
```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

It by default uses roulette wheel selection.

The following figure 1.1 provides the pseudocode:



```
Genetic-Algorithm returns the best route
  inputs:init_population_generator
         crossover_operator
         mutation_operator
         max_generation
         fitness-fn

  cur_generation = GENERATE_INIT_POPULATION(init_population_generator)
  for i = 1 to SIZE(max_generation) do
        sort cur_generation in ascending order
        next_generation <- top half of cur_generation
        for j = 1 to SIZE
                parent1, parent2 = RANDOM_WEIGHTED_SELECTION(cur_generation, fitness-fn)
                child1, child2 = CROSSOVER(crossover_operator, parent1, parent2)
                if (small random probability)
                        child1 <- MUTATE(child1, mutation_operator)
                        child2 <- MUTATE(child1, mutation_operator)
                add child1 to new_generation
                add child2 to new_generation
        cur_generation = new_generation
  return the best route in cur_generation
```

Figure 1.1

1.1 Framework Parameters Introduction

The framework has the freedom to select different init populations, crossover operators, and mutation operators.

For init populations generation, it provides three options:

- random: generate the initial population randomly
- random_2opt: generate the initial population by generating a random population first and then for each of the routes do 2opt. This can normally provides a good initial population
- nearest_neighbor_2opt: generate the initial population by randomly generating a start city first, and then use the nearest neighbor algorithm to find a good solution, and then perform 2opt, repeat until the population size is reached.

For crossover operator, it provides two options:

- pm: Partially-Mapped Crossover
- ox: Ordered Crossover

For mutation operator, it provides three options:

- cim: Centre Inverse Mutation
- rsm: Reverse Sequence Mutation
- 2opt: A modified designed based on the idea of 2opt, basically it will find the first two swappable cities that can shorten the total distance, then do the swap and stop

1.2 Instruction on How to Run the Program tsp.py

The tsp.py is implemented with a command line interface, which allows to set the parameters for running the genetic algorithm through the command line. The following are some options it contains:

- -h (--help): Print command line interface help manual
- -i $arg (or --city_ifile $arg): set the input city dataset filename
- -o $arg (or --solution_ofile $arg): set the output filename for writing the best solution. $arg has default value: best_solution.txt
- -s $arg (or --init_pop_generator $arg): set the initial population generator. Valid options include: random, random_2opt, nearest_neighbor_2opt. $arg has a default value: random
- -c $arg (or --crossover $arg): set the crossover generator. Valid options include: pmx, ox. $arg has a default value: pmx
- -m $arg (or --mutation $arg): set the mutation operator. Valid options include: rand_swap, cim, rsm, 2opt. $arg has a default value: rsm
- -n $arg (or --max_gen $arg): set the maximum number of generations for running the genetic algorithm. $arg has a default value: 10000
- -k $arg (or --max_gen $arg): set the population size. $arg has a default value: 20

Examples of running the program:

1. One can use the following command to print the help manual:

```
python3 tsp.py -h
```

2. Given a dataset named cities1000.txt, one can generate a tsp solution with all the default arguments by using command:

```
python3 tsp.py -i cities1000.txt
```

3. Given a dataset named cities1000.txt, one can specify the number of generations as 100, and population size as 40, by using command:

```
python3 tsp.py -i cities1000.txt -n 1000 -k 40
```

4. Given a data set named cities1000.txt, one can specify the init_pop_generator as random_2opt, crossover as ox, mutation operator as rsm, by using command:

```
python3 tsp.py -i cities1000.txt -s random_2opt -c ox -m rsm
```

2. Ideas and Features Tried in This Project

As mentioned earlier in Section 1, I tried using different initial population generators (e.g., random, random_2opt, nearest_neighbor-2opt), crossover operators (e.g., pmx, ox) and mutation operators (rand-swap, cim, rsm, 2opt), and with different combinations of them. Because of the time complexity, I just used a file of 50 cities for convenience and to find the pattern. Following are some of the results:

Table 2.1: Comparison of different init_pop_generator on a file of 50 cities, with maximum generation 1000, 5000, and 10000, and crossover operator as pmx, mutation operator as rand-swap. The best score for each of them are shown as follows:

	1000	5000	10000
random	19789.73	19228.90	13796.23

random_2opt	10605.62	10532.50	10437.20
nearest_neighbor_2opt	10790.40	10690.38	10545.62

Table 2.1

Table 2.2: Comparison of different crossover operators on a file of 50 cities, with maximum generation 1000, 5000, 10000, and init-pop-generator as random, mutation operator as rand_swap. The best score for each of them are shown as follows:

	1000	5000	10000
pmx	17816.93	13293.65	13161.15
ox	18043.70	16045.40	14955.92

Table 2.2

Table 2.3: Comparison of different mutation operators on a file of 50 cities, with maximum generation 1000, 5000, 10000, and init-pop-generator as random, crossover operator as pmx. The best score for each of them are shown as follows:

	1000	5000	10000
rand_swap	16389.48	15773.59	14958.25
cim	15871.85	12507.38	11054.78

rsm	12315.48	11003.19	10697.74
2opt	11476.92	11190.18	10721.04

Table 2.3

Clearly, with the maximum generation increases, the highest score of each way of solving the problem goes down. And this behavior gave a hint to solve the challenge problem.

3. Challenge Problem Results

As for the challenge problem, my current best score is 209300.97 (seems not as good as the ones who shows their results on the discussion board) and it took me 3 hours to run to get the result. I ran it by selecting “init_pop_generator” as “random”, “crossover_operator” as “pmx”, and “mutation_operator” as “2opt_mutator” in my genetic alrogrithm framework.

In short, I generated the initial population randomly, and then used roulette wheel selection, pmx crossover, and my own designed 2opt mutation operator (as introduced earlier in section 1.1) to generate the result.