COSC 3P71 Assignment 2: Traveling Salesman Problem

*Note: Sub-titles are not captured in Xplore and should not be used

Ibrahim Hashmi
Computer Science
Brock University
Milton, Canada
ih17px@brocku.ca

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

A genetic algorithm is a search technique inspired by Darwin's theory of natural selection that simulates evolution to solve optimization problems[1]. These search heuristics look for the most optimal solutions by choosing only the most fit individuals from the current generation to act as parents, and produce offspring that make up the population representing the next generation. These individuals can be represented in a variety of ways based on the context of the given problem. This report takes into account the classic Traveling Salesman Problem, which involves being given a list of `n` cities and the distances between them, and finding the shortest possible path that a salesman can take to visit each city exactly once before returning to the initial city they started from.

II. Background

Before applying a Genetic Algorithm to a problem, it's important to first determine how individuals (also referred to as **chromosomes**) will be represented, as well as what exactly it is that they represent.[1] In the context of this report, each individual/chromosome is a possible route through the cities that a salesman can use, and is represented in the form of a list of numbers with each number associated with a city. For example, one route of `5` cities could be `[1,2,3,4,5]`.

A. Initial Population and Elitism

At the start of the program, an **initial population** is created and filled with randomly generated chromosomes. This population is taken and **elitism** is applied to it, which involves sending the top `m`% of highest-fitting chromosomes from the current population into the next generation's population, with m being no more than `7%` (in our case, the top `7%` of chromosomes are taken and sent straight into the next generation).

B. Evaluation of Fitness

Since our algorithm is searching for the path of cities with the **shortest** distance, we use a fitness function `evaluate()` that takes a chromosome (list of cities) as a parameter returns the total distance that the salesman would be traveling if he were to follow that route. The total distance is calculated by iterating through each city in the chromosome and calculating the euclidean distance between them in the order that they appear in. The euclidean distance is calculated using the following formula:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

C. Selection

For our genetic algorithm, we used **Tournament Selection** to determine which individuals were fit enough to be chosen as parents for producing offspring. This selection process involves randomly selecting `k` individuals from a population, and from them choosing the one with the **highest fitness** to be selected as a parent[1]. The value of `k` can be no less than `2` and no greater than `5`[1].

D. Recombination

Two percentages taken as a parameters specify the *rate* at which selected chromosomes are being recombined and mutated before being sent into the next generation's population. The process of recombination was done using **one** of two different crossover methods: **Uniform Order Crossover** and **One-Point Crossover**:

Uniform-Order Crossover uses a bit mask to take genetic information from two parents and combine them to produce children. In our implementation, we randomly generated a list of `1`s and `0`s to act as the bit mask, and initialized both children lists as `child1=["-","-","-","-","-"]`, `child2=["-","-","-","-","-"]`. For every index in the bitmask that contained a `1` we replaced the value in child1 at that specific index with a corresponding value from parent1

that is missing from child1, and did the same with child2 but this time getting the values from parent2:

```
for index in range(bitmask):
    child1[index] = parent1[index]
    child2[index] = parent2[index]
```

We then replaced every occurrence of "-" in each child with a value from the remaining parent they *hadn't* taken values from yet:

for city in parent2:
  if city is not in child1:
    replace first occurrence of "-" in child1 with city

The second crossover method was One-Point Crossover, which places a point randomly in each parent (same spot for both parents) and takes every value left of that point in one parent and combines it with every value right of that point in the second parent to produce the first child, and vice verse to produce the second child:

```
crossover point = second index
Parent1 = [0 0| 0 0 0]
parent2 = [1 1| 1 1 1]
----------------------
child1 = [0 0 1 1 1]
child2 = [1 1 0 0 0]
```

After the recombination process, there is a chance one or both of the selected individuals (either the newly produced offspring or the originally selected chromosomes, depending on whether they were recombined) are put through a **mutation** process. The possibility of this occurring depends on the mutation rate% mentioned earlier that was obtained as a parameter. If an individual is chosen to go through mutation, then one city from within that chromosome is taken from it's original position and relocated to a different, random position. After this mutation process is complete, the chromosome is then sent into the next population being generated. If an individual does *not* get mutated, then they are simply sent straight into the new population.

III. Experimental Setup

A function named `run_ga()` was created to launch the algorithm to determine the optimal city route. This function has `7` parameters that must be given a value in order to run the algorithm. These algorithms (in order of appearance from left to right) are:

- `generations` : number of generations in the experiment
- `kValue` : k-value used in tournament selection
- `seed` : the random number seed used in experiments
- `elitismRate` : % of highest-fitting chromosomes sent to new population
- `crossoverRate` : % chance of individuals being recombined
- `mutationRate` : % chance of individuals being mutated
- `popSize` : length of each population

To obtain the results used in this report, this function must be called multiple for each experiment, with the parameters being adjusted each time according to the crossover and mutation rates being tested. A `k-value` of `2` and `elitismRate` value of `7` were set for all the experiments. The total number of generations for each experiment was determined based on the text file that was being used to pull the cities from:

- *"ulysses22.txt"* →`generations = 50`
- *"eil51.txt"* →`generations = 60`

`popSize` is also dependant on which text file is being used, as a population of `1000` was used for ulysses22.txt, while a population of `300` was used for eil51.txt.

The two parameters `crossoverRate` and `mutationRate` are set to whichever combination of crossover and mutation rates we are testing for, and each combination is tested 5 times with a different random number `seed` each time.

For example, to test for a crossover rate of `100%` and mutation rate of `10%` using the *eil51.txt* text file, the function should be called by running the following:

```
  run_ga(60,2,0,7,100,10,300)
run_ga(60,2,1,7,100,10,300)
run_ga(60,2,2,7,100,10,300)
run_ga(60,2,3,7,100,10,300)
run_ga(60,2,4,7,100,10,300)
```

To test the same rates using the *ulysses.txt* text file, an example of a function call would be:

```
  run_ga(50,2,0,7,100,10,1000)
run_ga(50,2,1,7,100,10,1000)
run_ga(50,2,2,7,100,10,1000)
run_ga(50,2,3,7,100,10,1000)
run_ga(50,2,4,7,100,10,1000)
```

To switch between crossover methods, simply disable whichever one is not being used by commenting it out using the `#` symbol on `lines 275,276`. In the example below, the Uniform Order Crossover is disabled while the One-Point Crossover is enabled:

```
child1,child2 = crossover_1pt(parent1,parent2)
#child1,child2 = uox(parent1,parent2)\\
```

A. Results

The results of the experiments carried out using the *ulysses22.txt* file indicate that the crossover rate of 100% combined with a mutation rate of 10% gave us the single most optimal route for the Traveling Salesman (given our parameters), as it found the city route [15, 6, 7, 12, 13, 14, 8, 22, 18, 4, 17, 2, 3, 1, 16, 21, 20, 19, 10, 9, 11, 5] and it's fitness score of `75.5088177`, which is lower than any other fitness score that was found throughout the experiments.

However, when we averaged the best fitness scores across all runs for each crossover-mutation rate parameter, it was apparent that the custom parameter of crossover rate = 92% and mutation rate =8% actually found the shortest overall routes consistently (see *Fig.1*).

Between the two crossover operators, the One-Point Crossover method consistently out-performed the Uniform Order Crossover throughout the experiments throughout all the tested parameters except for when crossover rate = 90% and mutation rate=0%, where Uniform Order was able to find a shorter path.

On the other hand, the experiments carried out using the *eil51.txt* file indicate that the crossover rate of 92% combined with the mutation rate of 8% managed to find the shortest possible route, as it found the path of cities [14, 7, 26, 8, 31, 22, 2, 21, 29, 20, 35, 36, 3, 28, 13, 19, 40, 41, 24, 43, 23, 25, 47, 34, 16, 32, 27, 48, 18, 46, 12, 15, 17, 4, 38, 9, 50, 49, 37, 6, 51, 1, 11, 5, 10, 39, 30, 33, 45, 44, 42] and it's fitness score of `758.4047213`.

When the best fitness scores for each crossover-mutation parameter were averaged, it was found that the same parameters of crossover=92%, mutation=8% was also consistently better than the other parameters at finding city routes with shorter paths. As we can see in Figure 2, the only parameters that managed to beat it were crossover=100%, mutation=10% using One-Point Crossover, and crossover=100%, mutation=0% using Uniform Order Crossover.

When analyzing the performances of the crossover operators, once again we noticed that One-Point Crossover managed to out-perform Uniform Order Crossover, however not by much. As we can see in Figure 2, One-Point Crossover managed to perform better using 3 of the 5 tested parameters, while Uniform Order Crossover was slightly better in the remaining 2.

B. Discussion and Conclusions

Separate experiments were carried out using two significantly different crossover methods to test out which of the following parameters led to the more optimal solutions:

- Crossover rate = 100%, mutation = 0%
- Crossover rate = 100%, mutation = 10%
- Crossover rate = 90%, mutation=0%
- Crossover rate = 90%, mutation=10%
- Crossover rate = 92%, mutation = 8%

Although the results of the experiment clearly show which crossover-mutation rates performed better as well as which crossover methods led to more optimal results, it must be noted that the results could change by adjusting certain parameters. Certain experiments implied that even lower routes could possibly be found if a higher number of generations were used for experiments, especially with the experiments involving *ulysses22.txt*. Thus, the only finding from the experiments that was significant was the fact that the custom parameters we set of crossover=92% and mutation=8% consistently out-performed the rest of the given parameters.

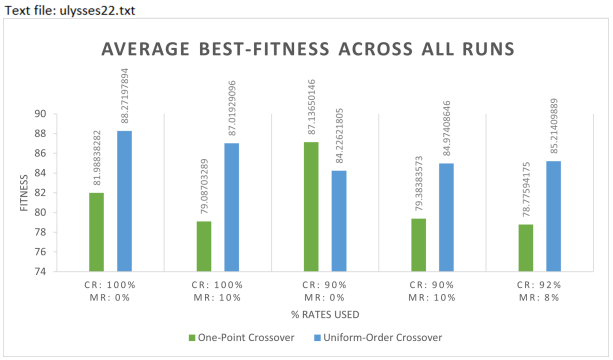

Fig. 1. Experiment results using the file ulysses.txt

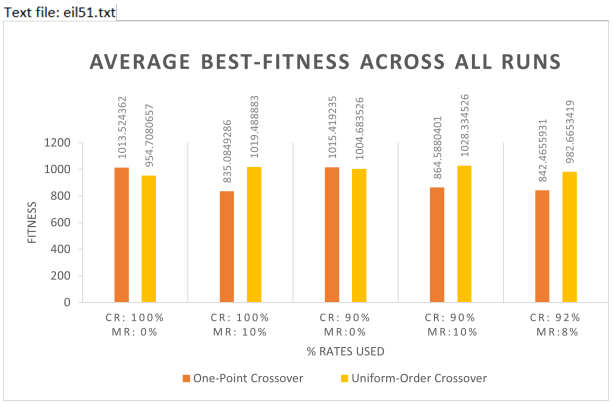

Fig. 2. Experiment results using the file eil51.txt

t-Test: Two-Sample Assuming Equal Variances

	One-Point X	*UOX*
Mean	81.2743389	85.94113
Variance	12.3687767	2.74999
Observatic	5	5
Pooled Va	7.55938313	
Hypothesi	0	
df	8	
t Stat	-2.683772	
P(T<=t) on	0.01388162	
t Critical o	1.85954804	
P(T<=t) tw	0.02776324	
t Critical tv	2.30600414	

Fig. 3. T-test of ulysses22.txt using best average fitnesses

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	914.2164	997.9761
Variance	8494.257	883.9914
Observatic	5	5
Pooled Va	4689.124	
Hypothesi	0	
df	8	
t Stat	-1.93401	
P(T<=t) on	0.044583	
t Critical o	1.859548	
P(T<=t) tw	0.089167	
t Critical tv	2.306004	

Fig. 4. T-test of eil51.txt using best average fitnesses

REFERENCES

[1] B. Obuki Berman Lecture Slides - COSC 3P71 Introduction to Artifical Intelligence

[2]