Exploration into the Traveling Salesman Problem and Genetic Algorithms

Khang Vu

301255281

CMPT 310 Spring 2020

Simon Fraser University

I. Source Code File:

The included file *tsp.py* is a modified version of Dr. Toby Donaldson's framework. The changes are as follow:

- 2 crossover functions added
- 1 selection function added

Since Dr. Donaldson's framework is a single python file, running it will run the current best genetic algorithm that I could make.

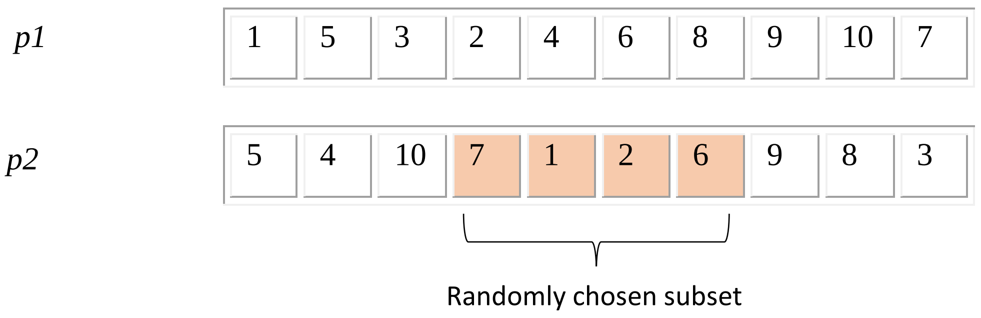
II. Crossover functions:

a. My own function:

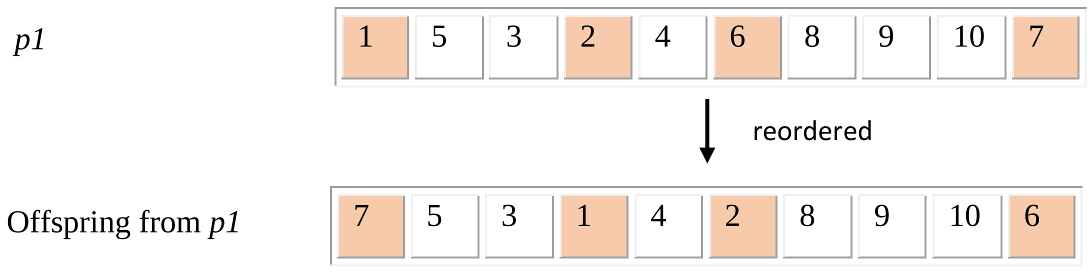
My own crossover function was written in part to understand the framework provided by Dr. Donaldson. It was not meant to be effective, but as I will show later, it was.

Given two parents *p1* and *p2*, the function will choose randomly a subset (any size) of *p2*. The subset elements are searched for in *p1* and reordered to match the subset ordering, though their positions in *p1* are unmoved.

For example:



Then,



A second offspring is also generated from p2 using the same method with subset of *p1*.

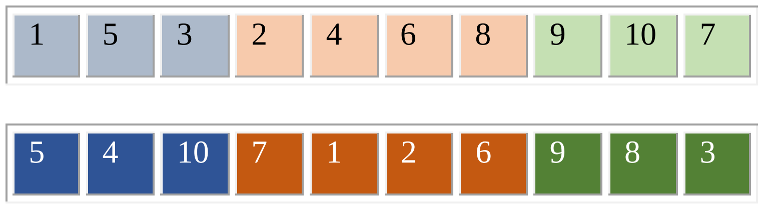
b. Two Point Crossover with Replacement:

The Two Point Crossover with Replacement (TPXwR) method was taken from a paper by Ammar Al-Dallal [1].

Given two parents *p1* and *p2*, the method randomly splits each parent into three subsections:

Original *p1*

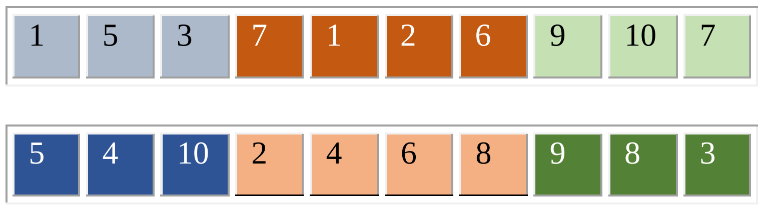
Original *p2*



Then the middle parts are swapped:

p1

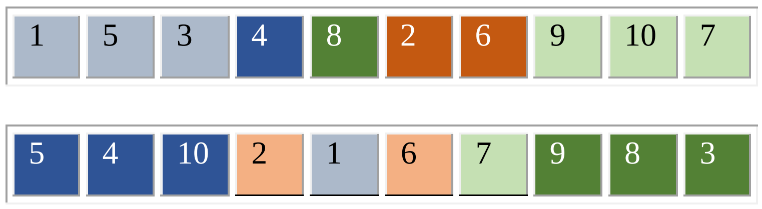
p2



Then duplicates in the swapped parts are fixed by using the first and last parts of the original parent to have valid permutations:

Offspring 1

Offspring 2



To be noted that each subsection's size is randomly chosen. So it is possible that the middle part's size is 0, then the offsprings are the same as the parents.

[1] A. Al-Dallal, "Using Genetic Algorithm with Combinational Crossover to solve Travelling Salesman Problem," *2015 7th International Joint Conference on Computational Intelligence (IJCCI)*, Lisbon, 2015, pp. 149-156.

III. Mutation:

The mutation I used is the simple swapping of a random pair of cities. Each offspring of the two crossover functions described above is applied this mutation.

IV. Selection Function:

Binary Tournament Selection:

This selection function is also taken from the paper by Al-Dallal. However, the author's description for it is very brief, so some implementation details are added by me. The selection process is as follows:

Step 1. The best permutation is retained for the new population.

Step 2. From the top 67% (2/3) of the population, randomly choose 2 permutations and the better permutation is chosen as the first parent, *p1*.

Step 3. Repeat step 1 to get the second parent, *p2*.

Step 4. The parents are then bred using one of the two previously described crossover functions, which gives two offspring.

Step 5. Repeat steps 2-4 to populate the new generation.

This function makes it so that only the best permutation is retained for the next generation. The rest of the current population are used as parents to generate offspring. Thus, the new population contains only offspring (and the best permutation of the previous generation).

V. Exploration History:

a. The first thing I did was, of course, downloading *tsp.py* given by Dr. Donaldson. I took the first day (though not with full effort) to understand the framework.

b. I then played around with the different parameters. I tried a few things:

- Adding many mutations (swapping random pairs).

- Changing population size and max iterations of the provided functions by Dr. Donaldson.

- Combinations of the two.

However, the result never got lower than 1 million in score.

c. I implemented my crossover function described in **II.1**. Still using the crossover search provided by Dr. Donaldson that retains the top 50% of the population, the result never broke 1 million in score, either.

d. I then did some research on TSP and genetic algorithms. I looked at probably 10 different papers, and most of them went over my head. Until I tumbled on the paper by Ammar Al-Dallal. His explanations are clear, so I decided to implement some of his ideas.

e. Using my crossover function and binary tournament selection with randomly generated first generation (1000 iterations on 50 permutations) I was able to obtain a score of 662878, which was a huge break through, as the previous best score was over 1 million.

f. Using TPXwR and binary tournament selection with randomly generated first generation (1000 iterations on 30 permutations), I was able to obtain a score of 638118.

g. The next thing I tried is using the best current record permutation as part of the initial population. I added the permutation with the score of 638118 to the randomly generated first generation. And miraculously, the score got better.

h. Thus, I repeated this method, adding the best current record to the initial randomly generated first generation. And the score kept getting better and better.

i. Due to the complexity of TPXwR, the function took a long time to execute compared to my crossover function. Thus, I decided to give my function a shot. With a short iteration numbers of 100 on 30 permutations, which takes about 10 seconds, I was able to obtain a better record at every run.

j. I then automated this process.

VI. Best Result:

In summary, the method that got me the best score is as follows:

Loop:

- Initialize a new population with 49 random permutations and the best permutation from previous loop iteration.
- Run my crossover function (**section II.a)** with binary tournament selection (**section IV)** for 100 iterations.

In actual runtime, my best result took about 3 hours to complete. Though, for the majority of the time, I was manually inserting the best result into the function and execute it again and again. When I automated the process, it took a lot less time. About 70 iterations of the loop were executed.

To be noted that, the score improved at every loop. However, the improvement started to plateau, so I stopped its execution. Given more time, the algorithm will provide a better score than the one I submitted.