Richard Dong
301317247
April-19-2020

CMPTt 310 Surrey Spring 2020

Final Project: TSP with Genetic Algorithm

Richard Dong
301317247
April-19-2020

Introduction:

The genetic algorithm framework is borrowed from the tsp.py given by class. The framework can be broken down into the following parts: Initial Population generation, Fitness Function, Selection, Crossover, Mutation. The initial population generation was given by the tsp.py . It functions by creating a list of 1 -> 1000, and then shuffling it to form one permutation. This is repeated "population size" times.

The given framework includes the insertion method of "*elitism", to some degree*. This involves picking the best permutation and carrying it directly over to the next. *"elitism"* has shown to get the best fittest population, when compared to roulette wheel selection and tournament selection, but not by much [1]. By using the top 33% best permutation carry over to the new generation, as well as being the parents for the next generation, it is hoped that better children will be produced and still have more new different permutations be created.

```
Fitness_function(current_generation):
        For score of each permutation in current_generation:
        Fit_score.append(1/score)
End

Standadize_fitness(fitness_score):
        Val ← sum of fitness_score
        For score in fitness_score:
                score = score/sum
End
```

The fitness function is rather simple. By taking the actual score of the permutation then set fitness score to 1/score inverses it. The lower the score is, the higher the fitness score will be leads to a higher chance of being selected as being a parent. I still wanted roulette selection to be used for picking parents, so I standardized the fitness score, and relied on random selection based on probability.

Richard Dong
301317247
April-19-2020

Crossover Function

After the selection of the parents, they need to be recombined to form new children. The crossover function is an important part that can greatly affect the results obtained. I have used the following crossover functions in hopes of getting the best solution: PMX, OX, CX2 and my own made crossover. A simple explanation of the crossovers will be provided.

1. Partially Mapped Crossover (PMX) [3]
   1.1 How does it work:
      1.1.1 Select 2 random points 0<=i<j<permutation length
      1.1.2 Look at second parent values in same index position
      1.1.3 Swap the value in position of index i with the value of second parent in position i of first parent
      1.1.4 Repeat steps 1.1.2 and 1.1.3 until i = j

   1.15 Repeat from beginning for the second offspring but with swapped parents.

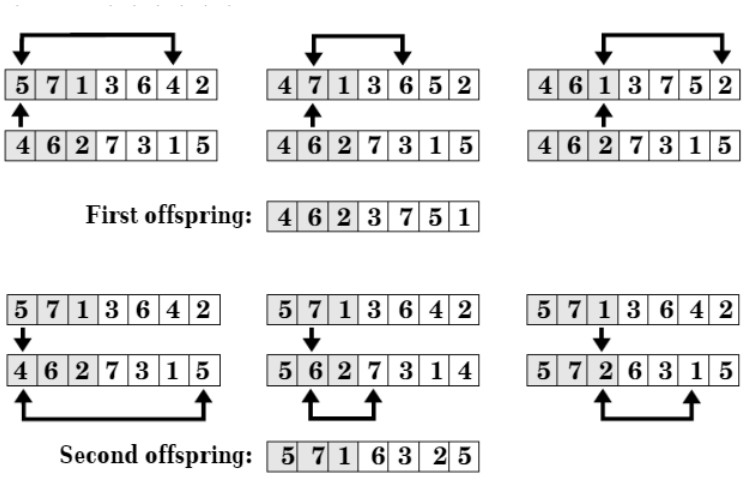


Figure taken from [3]

2. Ordered Crossover (OX) [4]
   2.1 How does it work
      2.1.1 Select 2 random points i, j; 0<=i<j<permutation_length
      2.1.2 Copy parent1[i:j] to first offspring, in the same index too
      2.1.3 Start from position j in parent 2, and go traverse to the end and again from the beginning (if needed too) . Insert any elements not currently in the first offspring in the same way as you traverse (this keeps the order of elements to be the same)

   2.14 Repeat this for offspring 2, but with swapped roles of parents

Richard Dong
301317247
April-19-2020

3. Cycle Cross Over2 (CX2) [2]
   3.1 How does it work (taken directly from [5])
      3.1.1 Select 1st bit from second parent as 1st bit of first offspring
      3.1.2 The select bit from step 2 would be found in first parent and pick the exact same position bit which is in second parent, and that bit would be found again in the first parent, and finally, the exact same position bit which is in second parent will be selected for 1st bit of second offspring.
      3.1.3 The select bit from step 2 is found in first parent and take its exact same position bit in the second parent as the next bit for the first offspring
      3.1.4 Repeat steps 3 and 2 until the 1st bit of parent1 shows up in offspring2
      3.1.5 Apply the same steps from 1-4 for the remaining unused bits (keep the bits in the same position.

4. Self-Made Crossover (myX)
   4.1 How it works
      4.1.1 Traverse both parents from beginning to end
      4.1.2 Take any identical bits in the same index and copy it into the offspring of the same index during traversal
      4.1.3 Take all the remaining bit leftover and insert it randomly into the remaining spots

Mutations:

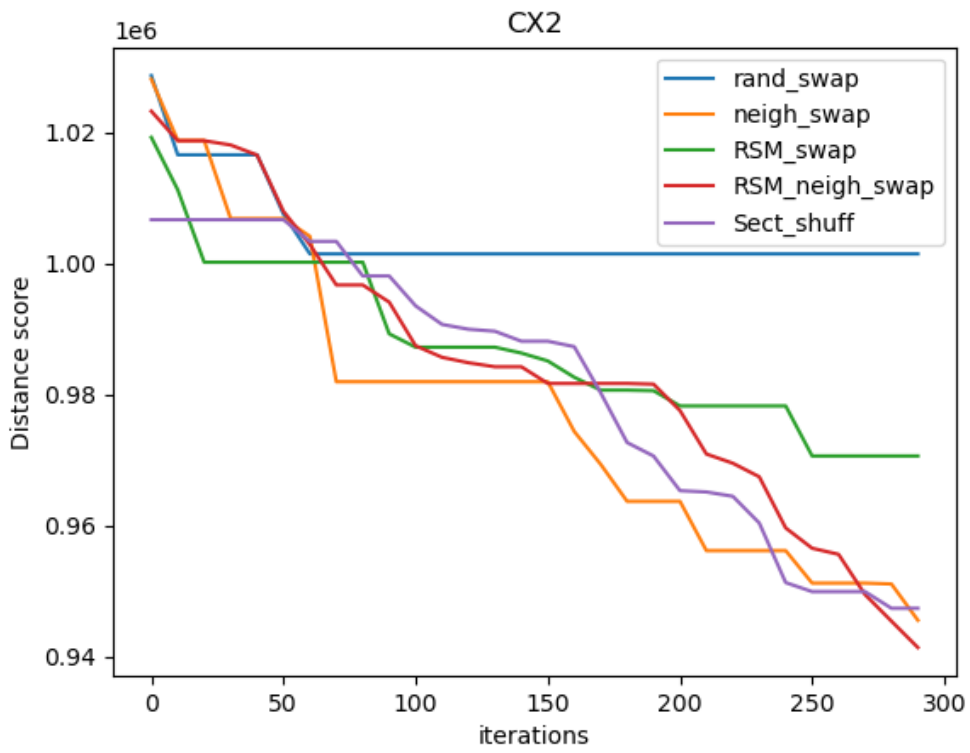
After crossover is done, each permutation is subjected to mutation. The mutation rate is set to 0.2 for all crossovers performed, except for the self-made crossover, which was set to 0.3. The reason for that is the self-made crossover can get stuck in a local minimum more often due to it taking the exact elements of same index from both parents. In order to offset that, we have given its offspring a higher chance mutating.

1. Random Swapping.
   a. This swapping involves iterating across the permutation, and selecting a random number between 0 and 1
   b. If the number selected is less than the given mutation rate, perform the swap.
   c. The swap begins by selecting a random index and switching the elements with the current index.
2. Neighbor Swapping.
   a. Same as random swap until step c
   b. The swap is done between the element next to it, so if at index i, we swap it with index i+1. If at the end, we swap element at i, with element at 0 index

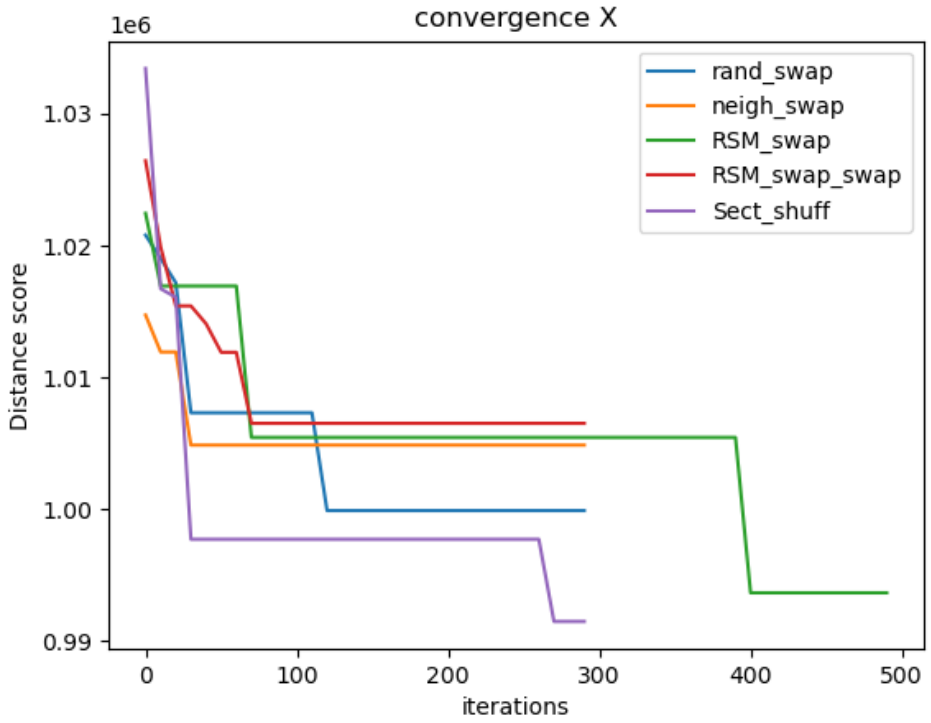
Richard Dong
301317247
April-19-2020

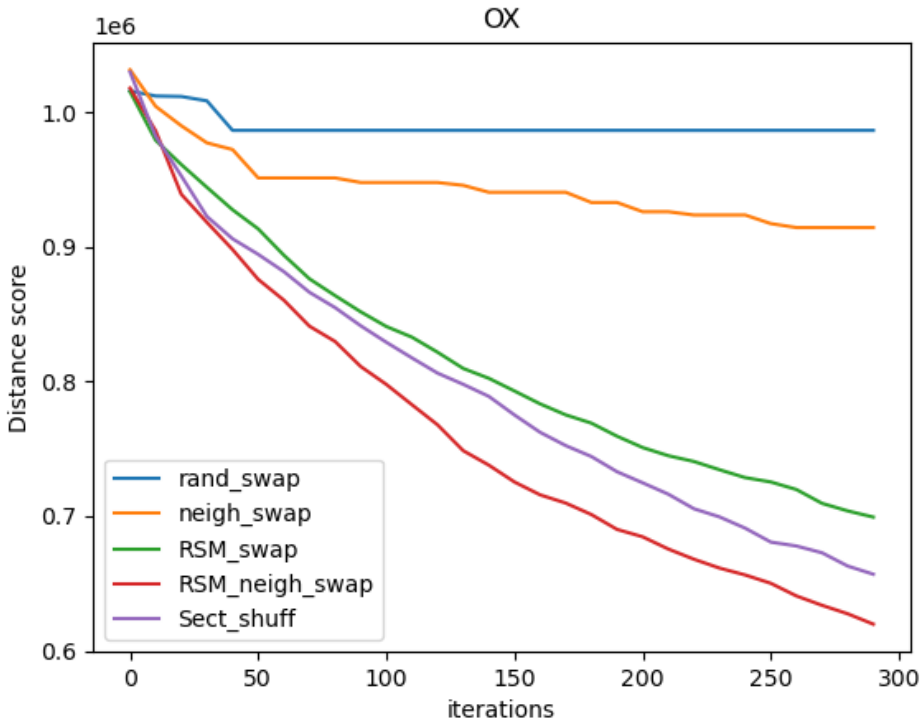
3. Reverse Sequence Mutation (RSM) [4]
    a. Select two points p and q; 0<=p<q<length of permutation-1
    b. Reverse the order of the elements between p and q
4. RSM with neighbor swap
    a. Perform RSM
    b. Then for elements between p and q, perform neighbor swap
5. Section Shuffle
    a. Select two points p,q from 0<=p<q<length of permutation
    b. Randomly shuffle the elements within that section

Results:

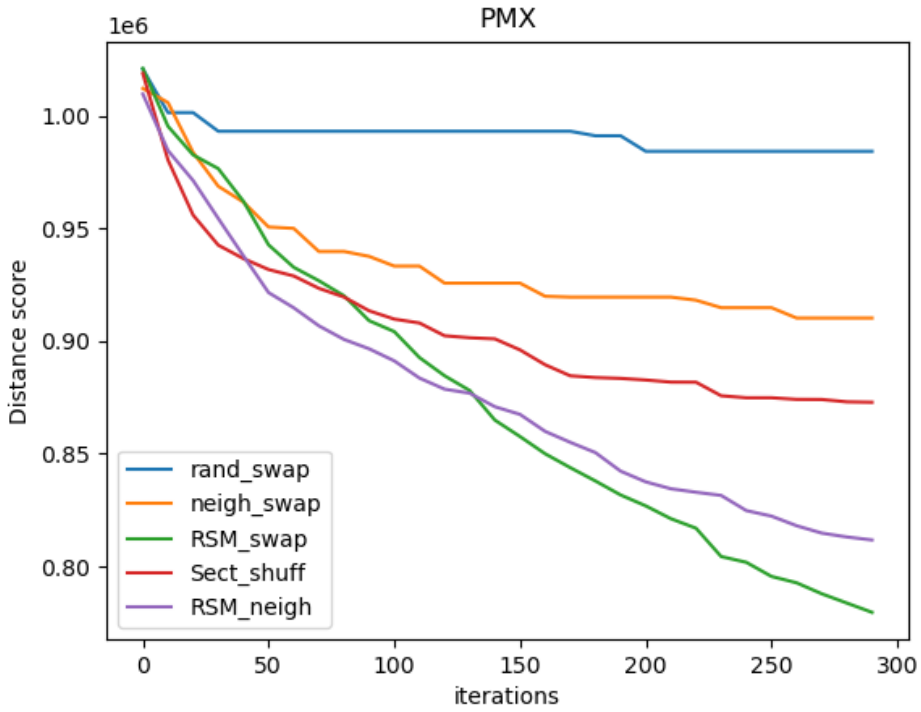


Richard Dong
301317247
April-19-2020





Richard Dong
301317247
April-19-2020



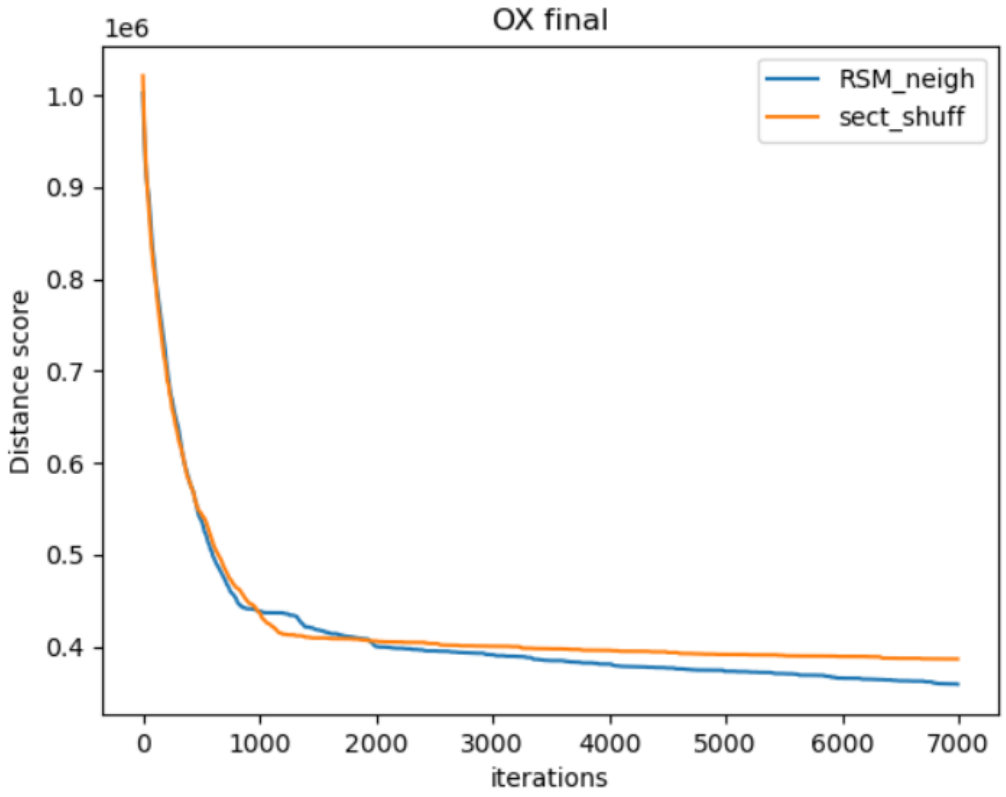
As one can see, with an iteration of 300 and population size of 100, the results has shown

myX < CX2 <PMX < OX

when it comes to solving the TSP. For mutations, varying results were obtained, but RSM swap, with RSM neigh and Sect shuffle has shown to be the best, except for CS2 where neighbor swapping came on top.

Since OX has shown to be the best crossover with the mutations RSM_neigh and Sect_Shuff, I ran this method for a total of 7000 iterations to see what the best score is I can get.

Richard Dong
301317247
April-19-2020



OX performed with RSM_neigh got the score of 358,780.88413986465 with time of 8793 seconds taken. OX with sect_shuff got the score of 386,156.16712054657 with time of 8841 seconds.

Testing out the Code:

To test this the code, please run the file tsp.py. To change any of the parameters, investigate the main function at the bottom. There should be an iteration parameter and population size you can change accordingly. The crossover function and mutation are in the crossover search function at line 252. The crossover functions and mutations are all stored in mutation.py

[1]https://www.researchgate.net/publication/285690217_Comparison_of_parents_selection_methods_of_genetic_algorithm_for_TSP

[2] https://www.hindawi.com/journals/cin/2017/7430125/

[3] https://www.researchgate.net/publication/282732991_A_New_Mutation_Operator_for_Solving_an_NP-Complete_Problem_Travelling_Salesman_Problem

[4] https://arxiv.org/ftp/arxiv/papers/1203/1203.3097.pdf