Project 4: Traveling Salesman Problem – Genetic Algorithm

Nick York
CES Department
Speed School of Engineering
University of Louisville, USA
neyork01@louisville.edu

1. Introduction

The Traveling Salesman Problem ("the TSP") was introduced in Project 1. A quick recap: the TSP requires finding the minimum cost route ("best route") for a salesman to visit a list of cities without visiting a city more than once, except for the starting city. The salesman must return to their starting city at the end. Each city is represented by an xy-coordinate pair and numbered starting at one. The coordinates are provided by a .tsp file ("TSP file"). The cost to travel between two cities is the Euclidean distance between them, calculated as:

$$cost = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

Project 4 required implementing a genetic algorithm ("the heuristic"). The heuristic is based on theory of evolution, wherein the fittest individuals reproduce and pass on their genes. Specifically, the heuristic encodes data points as genes and collections of genes as chromosomes. Populations are collections of chromosomes that change over generations as new chromosomes replace old ones. Chromosomes have a chance to reproduce each generation based on their fitness. Fitness is determined by a fitness function related to the problem domain. During reproduction, a crossover operator exchanges parts of two chromosomes to produce one or more child chromosomes. Afterward, a mutation operator changes one or more genes for one or more child chromosomes based on a mutation rate. The implementation of the crossover and mutation operators varies across genetic algorithms. Crossover and mutation repeat until the child population is the same size as the parent population. The child population replaces the parent population and repeats these steps until a termination criterion is met.

The goal of Project 4 was to write a program ("the program") to find a chromosome with the minimum cost ("the optimal chromosome") for a given set of genes using the heuristic. In terms of Project 4, cities are encoded as genes and routes as chromosomes. The order of genes in a chromosome implicitly define connections between cities. Genes can be in any order provided they do not repeat. Populations are initialized with randomly generated chromosomes. The heuristic runs a fixed number of times to compile statistics for four combinations of population size and mutation rate. Each run lasts for a fixed number of generations. The chromosome with the minimum cost is the optimal chromosome for the given combination.

2. Approach

Overview

I wrote a Python program that allows the user to specify which TSP file to use. Next, the program reads and parses the coordinate pairs in the TSP file. To implement the heuristic itself, I defined three classes: Gene, Chromosome, and Population.

Gene takes a label to identify the gene and a coordinate pair as arguments and assigns them to *label* and *coords*.[1] Chromosome takes a list of Genes and assigns them to *genes*. Next, Chromosome calls find_cost() to calculate cost using Equation 1 and assigns the result to *cost*. Then, Chromosome divides 1 by cost and assigns it to *fitness*. The rationale for defining fitness this way is that a smaller cost (indicating a shorter path) will yield a larger decimal, which equates to a greater probability of being selected for reproduction.

Lastly, Population takes a population size, mutation rate, and list of Genes. Population assigns population size and mutation rate to *population_size* and *mutation_rate*. These two attributes determine the number of Chromosomes in the Population and the probability of mutation. The list of Genes is passed to the Chromosome constructor to initialize a list of Chromosomes. These Chromosomes are initialized in a for-loop that iterates *i* times, where *i* is equal to *population_size*. Each Chromosome is appended to a list named *chromosomes*. An attribute *elites* is assigned *population_size* divided by 5. *elites* specifies the number of fittest Chromosomes to retain across each Population and ensures the fittest Chromosomes are not loss due to the randomness of crossover and mutation. Lastly, *mean_list*, *best_list*, and *worst_list* are initialized to empty lists and *optimal* to None. *mean_list*, *best_list*, and *worst_list* store the mean, min, and max cost for any Chromosome that belongs to Population for each generation. *optimal* is assigned the Chromosome with the minimum cost across all generations.

The create_genes() function takes the list of coordinate pairs parsed from the TSP file and iterates over them to retrieve each pair and its position in the list (to use as a label) to create a list of Genes. The list of Genes is returned and assigned to genes variable. generations is set to 1000 and num_of_runs to 20. generations specifies how many generations of populations to create before returning from the heuristic, and functions as the termination criterion.

Statistics

num_of_runs specifies how many times to run each heuristic. Running the heuristic multiple times allows us to compile statistics that are more reflective of the heuristic's performance. To test how performance varies with different parameters, I decided to vary the population size and mutation rate used by the heuristic. A list called population_sizes is initialized with 100 and 500. A second list mutation_rates is initialized as 0.01 and 0.1. Each of these lists specify parameters to vary. The program will produce four datasets for each combination of parameters. summary_statistics is initialized as an empty list to use to store each datasets' statistics.

The next section of code takes place inside a nested for-loop. The outer for-loop iterates over population_sizes and the inner for-loop iterates over mutation_rates. The nested for-loop will run the genetic algorithm twenty times for each combination of population size and mutation rate. To compile statistics across runs, several variables are initialized. best_runs and worst_runs are

[1] Class attributes are italicized to differentiate them from regular variables.

initialized to empty lists to store the min and max cost across runs. running_time is initialized to 0 and is used to track the time to run the heuristic across runs. results_list and stats_list each are initialized to an empty list and store results and statistics across each run returned by the heuristic itself.

A for-loop iterates *i* times, where *i* is equal to num_of_runs. Inside the for-loop, parameters is assigned a list of the current population size and mutation rate, plus generations. parameters is used to specify how the heuristic should operate. time() is called and the return value assigned to start_time. time() is called again after he heuristic runs, start_time is subtracted from it, and the result is assigned to running_time. Between the calls to time(), genetic_algorithm() is called with arguments parameters and genes.[2] genetic_algorithm returns two values that are assigned to results and stats. results includes the optimal Chromosome and cost for the first and every one hundred generations. stats includes the mean, min, and max cost for every generation, plus the min and max cost across all generations. results and stats are appended to results_list and stats_list. The min and max costs across all generations appended to best_runs and worst_runs. This process repeats for *i* runs before exiting the for-loop.

At this point, a single combination of population size and mutation rate has been used to compile a dataset of heuristic results. A helper function get_best_index() is called and passed best_runs to return the index of the best run across all twenty runs, assigning it to best_index. best_index is used to retrieve the optimal solution from results_list and the best run from best_runs and assigns each to best_result and best. A second helper function get_worst() is called and passes worst_runs to return the worst run across all twenty runs, assigning it to worst. The sum of best_runs is taken and divided by num_of_runs to calculate the mean across runs and assigns it to mean. Meanwhile, best_runs is used a second time to calculate the standard deviation. For each value in best_runs, mean is subtracted from value and the result squared. Once all the values are used, the sum of the squared results is taken and divided by num_of_runs. This takes the standard deviation across all runs. These statistics are combined with population size and mutation rate and appended to summary_statistics.

At this point, the nested for-loop has iterated once. The nested for-loop iterates three more times for the remaining combinations, and their dataset's statistics are appended to summary_statistics. The statistics are compiled and saved to a CSV file for review.

Genetic Algorithm

Now that the method of compiling statistics has been described, we can move onto the inner workings of genetic_algorithm(). genetic_algorithm() takes two arguments: parameters and genes. parameters contains population size, mutation rate, and generations, which are assigned to population_size, mutation_rate, and generations (always one thousand). genes is passed to Population() along with population_size and mutation_rate to initialize the initial population and assigns the Population object to population. results is initialized to an empty list to store the generation number, genes, and optimal cost for the first and every two hundred generations.

Next, a for-loop iterates *i* times, where *i* is generations. Inside the for-loop are the Population class methods. These class methods implement the heuristic are called in the following order: selection(), crossover(), mutation(), and evaluate(). First, selection() is called and takes no

[2] I will describe the inner workings of genetic_algorithm() after discussing how the program compiles statistics.

arguments. selection() zips a list of chromosome costs with *self.chromosomes*. Then, it sorts that list based on cost from small to large and returns just the sorted list of *self.chromosomes*. selection is initialized to an empty list to store the Chromosomes that are selected for reproduction. A for-loop iterates *i* times, where *i* is *self.elites* (always a fifth of the population size) and pops the Chromosomes from *self.chromosome*, which returns the fittest Chromosomes to ensure they survive and reproduce. The popped Chromosomes are appended to selection.

A second for-loop iterates *i* times, where *i* is *self.population_size* divided by two, minus *self.elites*. Inside the for-loop, a list of fitness values is retrieved from unselected Chromosomes and assigned to fitness_list. Next, a random Chromosome is selected using the random module's choices() function. choices() selects a single Chromosome from *self.chromosomes* using fitness_list for weights. The result is that the more fit Chromosomes have a higher probability of being selected. The result is popped by its value from *self.chromosomes* and appended to selection. selection() returns selection so crossover can occur.

For crossover(), I implemented several crossover operators, including one from a paper. After testing the effectiveness of each, I decided to use a simple one of the my own design that randomly takes a sequence of Genes of one parent Chromosome and combines it with the Genes from the second parent not in it to create a child Chromosome. The remaining genes from both parents are combined into a second child Chromosome.

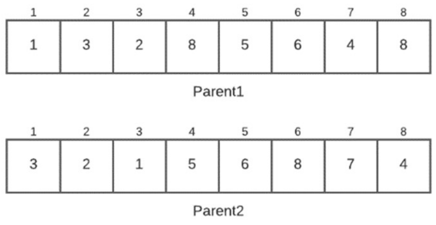

Figure 1. Two parent chromosome selected to reproduce.

First, crossover() takes selection as an argument using the random module's shuffle() function to return a random permutation of the Chromosomes in selection. An empty list, children, is initialized to hold the child chromosomes. From there, an outer for-loop iterates *n* times, where *n* is the number of Chromosomes in selection divided by two due to each iteration incrementing *i* two. Inside the for-loop, *i* and *i*+1 is used to index selection and assign the two chromosomes to parent1 and parent2. Two empty lists, genes1 and genes2, are initialized to store the sequence of genes used to initialize the child chromosomes. Then, the random module's randint() is called twice to return two random integers to index the parent chromosomes, assigning them to index1 and index2. The min and max of the two indices are assigned to start and end.

start and end are used in the first of four for-loops to index parent1, returning the genes corresponding to the range of numbers between start and end. The result is appended to genes1. In the second for-loop, the range of numbers between 0 and start are used to index parent1 and the result is appended to genes2.

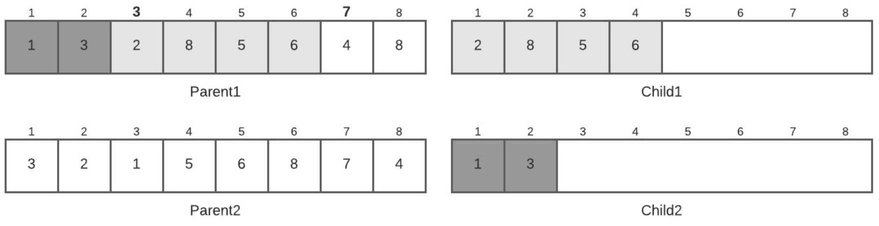

Figure 2. Child chromosomes after the first two for-loops finish. The bold 3 and 7 signify the start and end variables.

For the third for-loop, each gene in parent2 that is not in genes1 is appended to genes1, else they are appended to genes2. The fourth for-loop uses the range of numbers between end and the length of parent1 to index parent1 and the result is appended to genes2. genes1 and genes2 are used to initialize Chromosome objects child1 and child2. The child chromosomes are appended to children. The outer for-loop continues until all the selected chromosomes have reproduced. crossover() returns children for the next step, mutation.

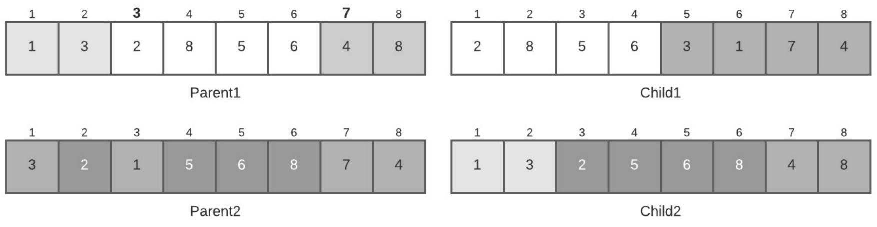

Figure 3. Child chromosomes after remaining for-loops finish. The bold 3 and 7 signify the start and end variables.

mutation() implements Reverse Sequence Mutation (see Resources). mutation() takes children as an argument. First, it finds the length of a Chromosome (based on the number of Genes) and assigns the result to size. Then, a for-loop iterates once for each Chromosome in children. For a Chromosome to mutate, the random module's random() function is called and if the result (a decimal between 0 and 1) is less than or equal to *self.mutation_rate*, then the Chromosome mutates. If the Chromosome mutates, then an empty list, genes, is initialized and start and end indices initialized using the same method as crossover. The start and end indices are used take the sequence of genes between them and reverse their order. A for-loop iterates *n* times, where *n* is size, to update each Gene in Chromosome to its new position based on genes. mutation() returns children to evaluate the new population.

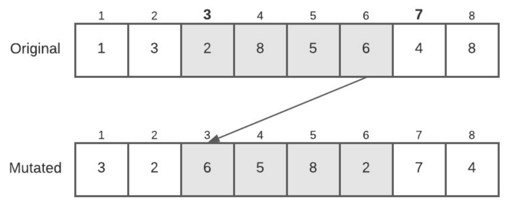

Figure 4. Result of Reverse Sequence Mutation. The bold 3 and 7 signify the start and end variables.

evaluation() takes no arguments and initializes any empty list, costs. Next, for each Chromosome in *self.chromosomes*, the Chromosome's cost is compared to the *self.optimal.cost* and appended to costs. Then, evaluation takes the sum of costs divided by *self.population_size*, the min value of costs, and the max value of costs and appends each to *self.mean_list*, *self.best_list*, and *self.worst_list*. These lists are used to plot the mean, best, and worst costs across generations for a single run of the heuristic. *self.optimal* is compared across generations and the final result is the Chromosome encoding the optimal tour. At this point, the heuristic has run for a single generation and repeats until the stopping criterion (specified by the generations variable) is met.

3.1 Data

For this project the students were provided a single TSP file, Random100.tsp with one hundred coordinates, for us to test our program on.

3.2 Results

The results of the heuristic for all four datasets are written to Random100_Stats.csv under a folder named Results (Figure 5). Figure 6 shows the optimal solution for each of the four datasets. Figure 7[3] has six subplots, each with two subplots; the first subplot shows the optimal tour for population 500 and mutation rate 0.1 across all twenty runs of the heuristic and how its mean, min ("best"), and worst ("max") change over generations. The second subplot plots the optimal tour itself. The second subplot shows how the heuristic improves across generations. Figure 8 shows the optimal tour for each of the four datasets.

Population Size	**100**		**500**	
Mutation Rate	**0.01**	**0.1**	**0.01**	**0.1**
Mean	1279.52	1195.35	992.08	910.66
Min	1190.88	1095.51	924.21	827.20
Max	5481.41	5641.61	5662.60	5634.97
Standard Deviation	42.48	29.27526106	43.05	49.96
Running Time (s)	10.90	15.50	78.54	86.81

Figure 5. Results of running the heuristic twenty times with each combination of population size and mutation rate.

[3] Due to a bug with Matplotlib's animator, the node labels do not display correctly in when animated.

Population Size	Mutation Rate	Cost	Genes
100	0.01	1190.883168	['79', '57', '30', '83', '58', '63', '96', '28', '55', '50', '32', '3', '86', '34', '76', '13', '8', '26', '87', '64', '17', '46', '48', '11', '68', '10', '98', '15', '69', '89', '7', '53', '9', '39', '12', '24', '5', '59', '97', '19', '80', '78', '82', '93', '84', '23', '70', '0', '72', '67', '14', '42', '91', '94', '29', '71', '88', '6', '35', '74', '2', '65', '36', '37', '20', '38', '73', '16', '92', '40', '85', '45', '95', '54', '60', '62', '27', '31', '18', '90', '52', '22', '33', '99', '44', '51', '21', '66', '77', '56', '49', '47', '1', '75', '25', '61', '4', '41', '43', '81']
100	0.1	1095.513879	['43', '79', '56', '77', '61', '41', '4', '57', '25', '76', '86', '1', '75', '13', '49', '47', '48', '34', '58', '12', '28', '55', '24', '59', '97', '19', '39', '5', '50', '96', '53', '7', '68', '10', '9', '89', '82', '93', '88', '35', '70', '0', '23', '84', '71', '67', '42', '72', '29', '91', '94', '14', '78', '80', '32', '63', '30', '83', '3', '17', '46', '11', '15', '69', '98', '92', '16', '40', '38', '6', '74', '73', '65', '37', '36', '20', '2', '52', '85', '22', '18', '90', '62', '31', '27', '60', '33', '54', '95', '45', '64', '87', '99', '51', '44', '21', '66', '26', '8', '81']
500	0.01	924.2075914	['54', '60', '27', '18', '90', '62', '31', '44', '51', '99', '95', '64', '81', '56', '43', '79', '57', '4', '41', '61', '63', '58', '30', '83', '32', '12', '24', '39', '5', '97', '59', '19', '80', '78', '67', '14', '94', '91', '0', '70', '23', '72', '42', '29', '71', '84', '6', '35', '88', '37', '36', '65', '20', '40', '52', '2', '38', '74', '73', '16', '92', '98', '10', '11', '17', '46', '68', '15', '69', '93', '82', '9', '89', '7', '53', '50', '55', '28', '96', '3', '34', '76', '25', '77', '8', '26', '21', '66', '87', '49', '13', '75', '1', '86', '47', '48', '45', '85', '22', '33']
500	0.1	827.1970384	['17', '11', '48', '86', '76', '25', '61', '49', '13', '75', '1', '47', '64', '87', '77', '56', '4', '41', '57', '79', '43', '81', '8', '26', '66', '21', '44', '31', '62', '27', '60', '54', '51', '99', '95', '33', '45', '85', '22', '18', '90', '52', '40', '38', '2', '20', '65', '37', '36', '35', '88', '6', '74', '73', '16', '92', '69', '98', '15', '89', '9', '93', '82', '84', '71', '72', '29', '23', '70', '0', '91', '42', '94', '14', '67', '78', '80', '19', '97', '59', '5', '39', '24', '12', '30', '83', '58', '63', '32', '3', '34', '96', '28', '55', '50', '53', '7', '68', '10', '46']

Figure 6. Optimal solution found in each of the four datasets.

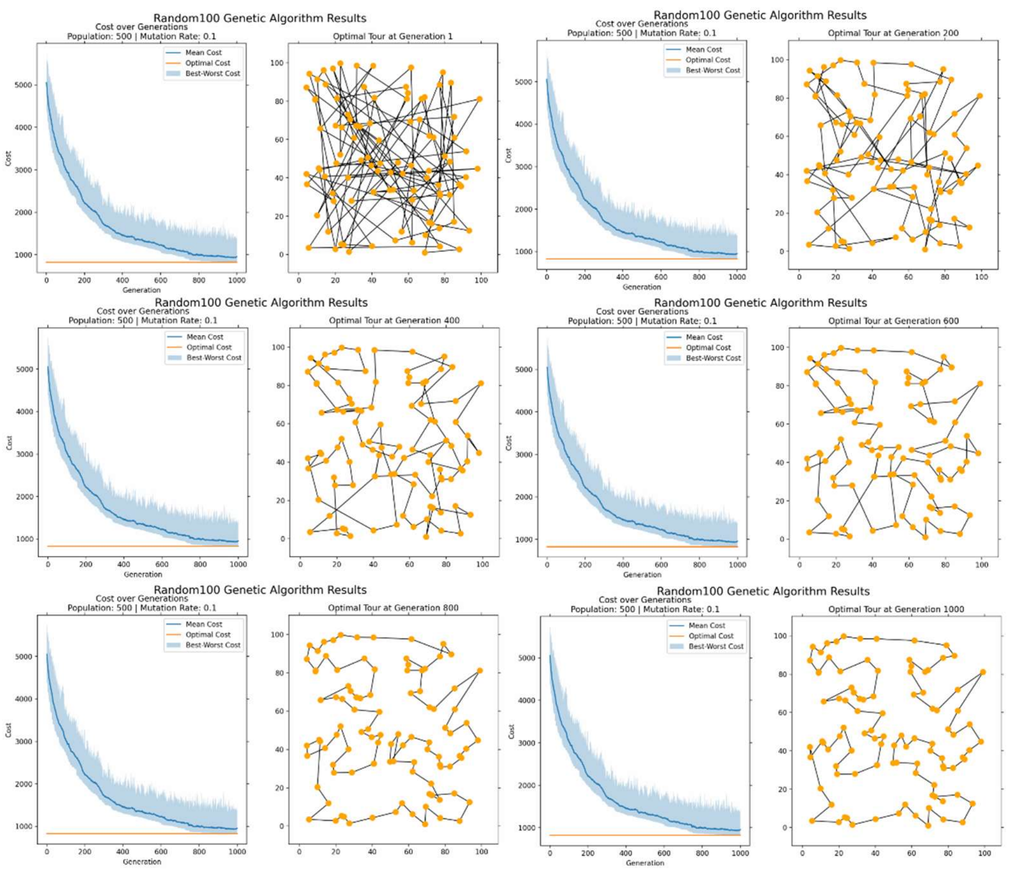

Figure 7. ***Left:*** *Cost over generations for population size 500 and mutation rate 0.1 for the optimal tour from twenty runs.*
Right: *A dynamic plot of the optimal tour at generation 1, 200, 400, 600, 800, and 1000.*

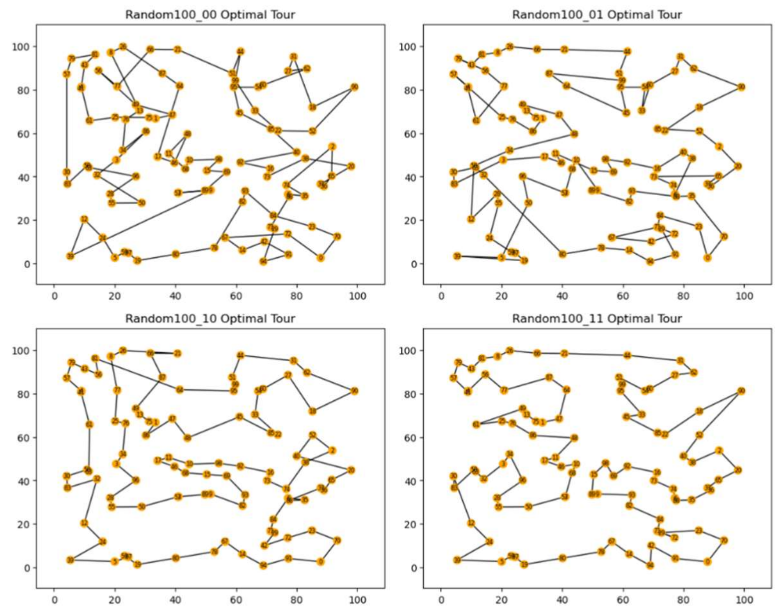

Figure 8. Optimal tour across twenty runs for each combination of parameters.
00: 100, .01 | 01: 100, .1 | 10: 500, .01 | 11: 500, .1

Discussion

My crossover operator and the mutation operator I selected (Reverse Sequence Mutation) worked reasonably well with the other parameters to arrive at near-optimal solutions for each of the four datasets. The most optimal solution was a population of 500 and mutation rate of 0.1. My hypothesis is that a larger population has a greater variety of Chromosomes with unique gene sequences than a smaller population, allowing for the discovery of more optimal solutions. Likewise, a higher mutation rate may create new Chromosomes with further unique gene sequences. As for the stopping criteria, it proved effective for the four datasets, but limits the heuristic to fewer iterations when it may still arrive at a more optimal solution. I used a stopping criteria of one thousand generations to limit the running time so I could perform more runs per datasets.

For running time, the optimal solution came from the fourth dataset and took an average of 86.81 seconds. Across the four datasets, those with a higher mutation rate or a higher population size took longer to run. The optimal solution took longest due to the use of the larger population size and mutation rate. However, the best combination of population size and mutation rate took the longest to run (86.81 seconds versus 10.90 for the minimum average running time). At the same time, the longer running time yielded the most optimal solution, which in depending on the requirements of a given problem may be an acceptable tradeoff.

Compared to previous solutions, the genetic algorithm can handle the largest number of cities. The brute force approach took roughly an hour and a half to find an optimal solution for twelve cities. Due to combinatorial explosion, the brute force approach would never be tractable for one hundred cities. Breadth-first and depth-first search runs in fractions of a second but requires a directed graph and is not guaranteed to find the optimal route. For an open-ended route where a salesman can visit cities in any order, and where the salesman must return to their starting city, the BFS/DFS approach is of limited utility. Compared to closest-edge insertion heuristic, which ran 40 cities finds *the* optimal solution for forty cities in less than ten seconds, the genetic algorithm is sluggish and does not guarantee an optimal solution.

References

Stack Overflow answer explaining concept of elites

Paper detailing Reverse Sequence Mutation