# Genetic Algorithm Project Report

Rylee Buchert

April 2022

## 1 Introduction

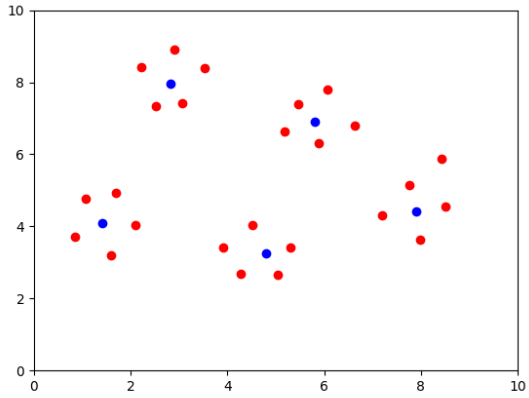
Genetic algorithms are a search heuristic in computer science which take inspiration from real-life evolutionary processes. Based around the ideas of natural selection and survival of the fittest, genetic algorithms have demonstrated strong performance in finding solutions to large optimization/search problems. Utilizing biological-inspired operators such as crossover and mutation, these algorithms are capable of navigating extensive search spaces to find high-quality solutions.

In this project, a genetic algorithm was implemented and tested on the P-Median problem. The results of these experiments highlight the algorithm's strength in finding quality solutions in a relatively small number of iterations. In order to garner an understanding of the genetic algorithm's performance, it was compared to two other search heuristics: simulated annealing and hill-climbing. The genetic algorithm exhibited robust search capabilities, finding optimal solutions in many experimental tests.

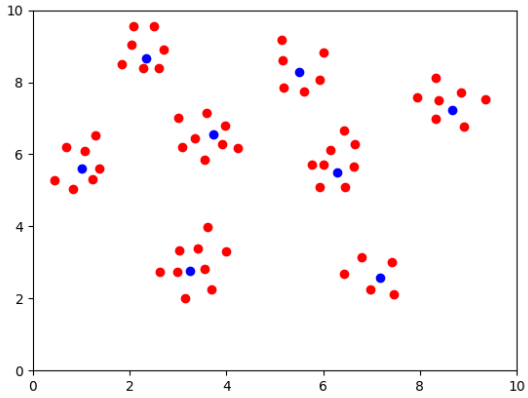
## 2 P-Median Problem

The goal of the P-Median problem is to select $p$ "facilities" in an undirected graph which minimize the total Euclidean distance between the remaining points and the selected facilities. This problem models many real-life situations such as choosing locations for warehouses or stores which service the most potential customers. There are many variations of this problem, including different graph types and "capacitated" facilities, but only the simple version of the problem was tested in this project.

To verify the genetic algorithm's search accuracy, multiple datasets of varying sizes were utilized. I began by constructing "toy" datasets with known optimal solutions, allowing me to compare the algorithm's results to the known answer. The first two datasets ("small" and "medium" sizes) contained 5 / 8 centers and 30 / 65 coordinates, respectively. Utilizing a simple matplotlib script in Python, I was able to plot the coordinates and save the data for later testing. The toy datasets are depicted in Figure 1.



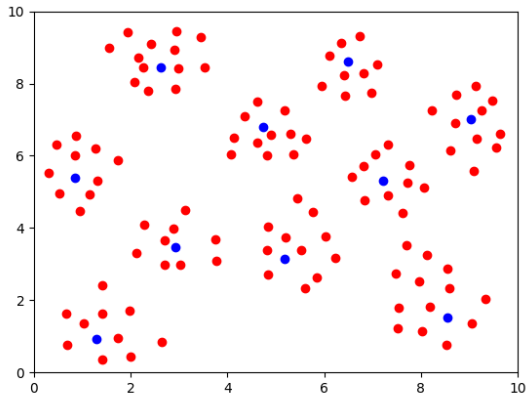

(a) "Small" Dataset



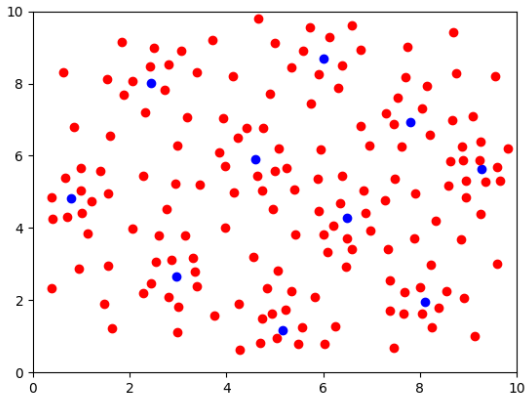

(b) "Medium" Dataset

Figure 1: Toy Datasets

After verifying the genetic algorithm's capabilities on the smaller datasets, I began constructing more complex problems with more coordinates and less clearly-defined solutions. Two "large" datasets were created both with 10 vertices and 115 / 170 vertices, respectively. The large datasets are shown in Figure 2. The blue points in these graphs show the anticipated centers when I created the data, but do not represent the optimal solutions.




(a) "Large" Dataset 1




(b) "Large" Dataset 2

Figure 2: Large Datasets

# 3 Genetic Algorithm

## 3.1 Chromosome and Fitness

For the P-Median problem, the genetic algorithm's chromosome representation is very straight-forward. The chromosome is a string of 0's and 1's the length of the number of vertices in the graph. Only $p$ digits in the chromosome can be 1 since those alleles represent the "facilities" or centroids within the graph. Because of this, it is easy to check for infeasible chromosomes since the sum of the values should always be equal to five. The initial chromosome is generated by randomly choosing $p$ indexes in the string to be the centers, while the rest are assigned 0.

The fitness function is used to evaluate the strength of each chromosome. To calculate fitness, each of the non-center points are assigned to the closest centroid and the total Euclidean distance from each point to the centers is calculated. This was implemented by looping over all 0's in the chromosome and choosing the closest center according to Euclidean distance. The distance between the points is summed up and returned, allowing all chromosomes in the pool to be compared. After scoring all of the chromosomes in the initial pool, the two highest fitness-valued ones are selected and automatically inserted into the next generation. This concept is known as 'elitism' and has been shown to improve the genetic algorithm's performance and convergence time.

## 3.2 Selection

The first operator applied to the chromosome pool is selection, a way of choosing the chromosomes for the parent pool. I implemented three selection methods for the GA: Roulette, Rank, and Tournament selection. Roulette selection operates by assigning probabilities to each chromosome according to their fitness value and randomly selecting parents according to those probabilities. The steps of the method include calculating the total sum of all the chromosome scores, normalizing the scores by dividing the total by each score, and using the normalized values to assign percentages. Using numpy's random choice function, a Roulette wheel is simulated and the method returns the selected parents.

Rank selection follows a somewhat similar structure to Roulette, except it uses ranks instead of fitness values to assign percentages to the chromosomes. This method works by sorting the chromosomes by fitness in ascending order and ranking them 1 to 100. By summing the ranks and normalizing the values, the percentages can be calculated and the parent pool is generated with the probabilities. The final selection mechanism is Tournament, which operates by choosing two parents randomly from the pool. With 75% probability, the higher fitness parent is inserted into the pool and the lower fitness parent is chosen 25% of the time. This process is repeated 100 times until the parent pool is full.

## 3.3 Crossover

The next operator used is crossover, a way of producing 'child' chromosomes by combining the genetic material of the parents. Similar to selection, I implemented three crossover methods for the GA: Single-Point, Double-Point, and Uniform crossover. Single-Point crossover

works by randomly choosing two parents from the pool and an index/crossover point in the chromosome. The two children chromosomes are created by taking the alleles to the left of the crossover point from parent 1 and combining them with the alleles to the right of the crossover point in parent 2. The opposite is performed to produce child 2.

After both children chromosomes are produced, the 'fix_up' method is called to ensure the chromosomes are not infeasible. This method works by seeing if the number of 1's in the children chromosomes does not equal $p$. In the event that there are more than $p$ centers, the method randomly chooses an index of one center and converts it to a zero. This is repeated until only $p$ values is 1. A similar process is used if the number of centers is less than $p$: the method randomly chooses a 0 index and converts it to 1, repeating until the chromosome is feasible.

The Double-Point crossover is very similar to Single-Point, except two crossover indexes are chosen instead of just one. In this method, the alleles between the two crossover points are inserted into the opposite parent to produce the children chromosomes. The fix up method is then called if either child chromosome is infeasible. The final crossover operator is uniform crossover, which utilizes a 'u' string to dictate which alleles are taken from the parents to produce the children. After randomly generating a string of 0's and 1's (the 'u' string), the value at each index represents which parent to select the alleles from. For child 1, a '0' indicates that the allele should be taken from parent 1 and a '1' suggests the allele comes from parent 2. The opposite applies when creating child 2.

## 3.4 Mutation

The mutation operator for this genetic algorithm is very simple. For each chromosome in the pool produced by crossover, one random bit is flipped with 5% probability. This operator is used to mix up the solution for a few number of chromosomes, helping the algorithm jump out of local minima with the goal of finding the global optimal solution in the search space. This mutation operator combined random bit flips in the fix up method helps to prevent the genetic algorithm from getting stuck in a non-optimal solution for too long.

## 3.5 Parameters

When creating the genetic algorithm, many choices must be made about the structure of the components to ensure it runs effectively. A few parameters, such as the selection/crossover operators, the mutation rate, and number of iterations are specified when the genetic algorithm is instantiated. This allows the user to test several versions of the GA with different parameters to see which work best in producing solutions.

On the other hand, parameters such as the population size and crossover rate were hard-coded into the program. I decided on a population size of 100 for the pools and a crossover rate of 100%, both common choices when implementing genetic algorithms. The final choice of parameters is the number of iterations that the algorithm will run. The final genetic algorithm ran for 1,000 iterations or when the solution chromosome has not changed for 100 generations, whichever comes sooner.

# 4 Simulated Annealing

## 4.1 Operators

The second algorithm implemented in this project was simulated annealing, a probabilistic heuristic used in problems with large search spaces. Similar to genetic algorithms, simulated annealing is based around a solution which is scored by fitness and modified to search the solution space. The main operator in annealing is perturbation, a method used to alter the solution in attempts of finding local/global optimums.

The perturbation function used is very similar to the mutation operator in the genetic algorithm. The perturbation works by randomly selecting an index in the solution and flipping the bit. After the solution is modified, the 'fix up' algorithm is applied which continually flips 0's or 1's until the solution is feasible (meaning the number of 1's is equal to the number of centers). These operations provide two sources of mutation, helping to ensure the algorithm does not get stuck in a local minimum when running.

## 4.2 Parameters

The simulated annealing algorithm contains several parameters which affect the number of iterations and how the algorithm generates solutions. The $\alpha$ and $\beta$ parameters control the changes in temperature and number of iterations for the loop. During each iteration of the outer loop, the temperature is multiplied by $\alpha$ and the number of iterations for the inner loop is multiplied by $\alpha$. For this project, I chose values of 1.02 and 0.98 for $\alpha$ and $\beta$, respectively. The final two parameters are the initial temperature and number of iterations. Following common practice for simulated annealing, I set the initial temperature to 10 and the iterations to 1000.

## 4.3 Foolish Hill-Climbing

The third and final algorithm implemented in this project is the "foolish" hill-climbing algorithm. This algorithm operates almost identically to simulated annealing, except it never accepts a worse solution when executing. Simulated annealing contains a provision where it will accept a worse perturbed solution randomly. The hill-climbing is "foolish" because it only accepts higher-fitness solutions, meaning it is more likely to get caught in local minima.

# 5 Computational Results

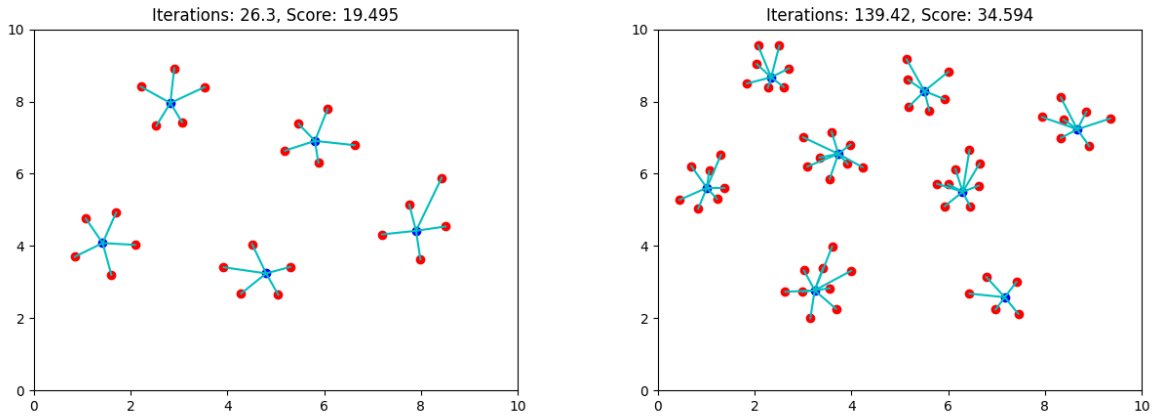
To evaluate the algorithms on the P-Median problem, they were each tested on several datasets of varying sizes. Each of the experiments involved several repetitions of the algorithms, with the average best score and number of iterations/perturbations taken for the tests. The results also include the best score produced by the chromosomes/solutions and associated graphs which show point assignments to the facilities.

The first experiment was conducted on the "toy" datasets which contained fewer points and known optimal solutions. The "small" dataset contained 5 centers and 30 total vertices

while the "medium" data included 8 centers with 65 vertices. The results for the toy tests are depicted in Tables 1 and 2.

For the small data, all 11 algorithms were able to locate the optimal solution and 7 of them found the optimal in all 50 repetitions of the experiment. The SGA with rank selection and uniform crossover averaged the lowest number of iterations to reach the solution while tournament, single-point averaged the highest number of iterations.

For the medium dataset, 10 of the 11 algorithms located the optimal solution. The SGA with rank selection and uniform crossover once again reached the solution in the fewest number of iterations. In this test, rank crossover averaged 30 less generations than the other crossover methods. The point assignments for the best performing algorithms on the toy datasets are shown in Figure 3.




(a) Small Dataset

(b) Medium Dataset

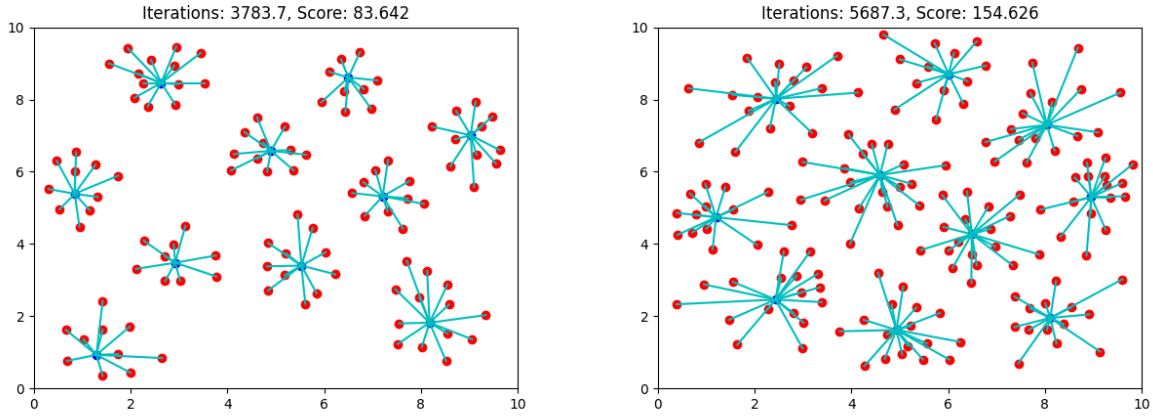
Figure 3: Toy Dataset Results

After verifying the algorithms on the toy datasets, the remaining experiments were conducted on "large" datasets, containing more centers/points than the inital tests and less clearly defined solutions. The results for the two large datasets are shown in Tables 3 and 4. Due to time constraints and the significant computational complexity for my code, I was only able to run 10 repetitions for each algorithm on the large datasets.

Fewer algorithms were able to find the optimal solution for these tests, with only 6 locating the optimal in the first large data and 4 in the second large dataset. Out of the genetic algorithms which found the best solution, tournament selection combined with double-point crossover required the fewest number of generations to reach the solution.

The most perplexing part of the computational results relates to the Simulated Annealing and Foolish Hill-Climbing algorithms. The SA algorithm performed worse than foolish hill-climbing in every test, a result which goes against expectations. While SA was anticipated to under-perform the genetic algorithms, it should generally score better than hill-climbing. Yet, hill-climbing performed best out of all the algorithms tested, including the genetic algorithms. It was able to locate optimal solutions in far fewer iterations than simulated annealing as well.

The most interesting part of this outcome is that foolish hill-climbing averaged the optimal solution for all 4 tests. The foolish algorithm found the optimal solution in every experiment, all 50 iterations on the toy datasets and all 10 on the large data. The optimal solutions found by the foolish algorithm for large data 1 and 2 are shown in Figure 4.




(a) Large Dataset 1

(b) Large Dataset 2

Figure 4: Large Dataset Results

# 6 Conclusions

The genetic algorithms exhibited strong performance in finding solutions to the P-Median problem in the experimental tests. In the majority of experiments, the SGAs were able to locate the optimal solution. Rank crossover combined with uniform selection performed best in the toy data tests, while tournament selection and double-point crossover scored best on the large data.

While the genetic algorithms performed very well, the foolish hill-climbing algorithm performed the best in all tests. This result was not expected, since simulated annealing should be capable of exploring the solution space better than the "foolish" approach. I poured over my code and could not locate any errors with simulated annealing and am still unsure why the results came out this way. These results may have occurred because of the structure of the test data, but I do not know why.

In future experiments, I would have liked to let the algorithms run for more iterations. This would have allowed the models more time to search the solution space for the optimal solution, potentially changing which algorithms performed best. But with the number of repetitions, datasets, and parameter combinations that were tested, I did not have enough time to let the algorithms run for a very long time. Additionally, I would spend more time going over the simulated annealing/hill-climbing algorithms to verify all components are written correctly and figure out why the foolish algorithm performed so well.

# Appendix

| Algorithm | Parameters | Best Score | Average Best Score | Average # of Generations/Perturbations |
|---|---|---|---|---|
| SGA | Roulette Single-Point | 19.495* | 19.495* | 70.24 |
| SGA | Roulette Double-Point | 19.495* | 19.495* | 40.84 |
| SGA | Roulette Uniform | 19.495* | 19.495* | 30.04 |
| SGA | Tournament Single-Point | 19.495* | 19.527 | 72.90 |
| SGA | Tournament Double-Point | 19.495* | 19.495* | 46.06 |
| SGA | Tournament Uniform | 19.495* | 19.495* | 29.40 |
| SGA | Rank Single-Point | 19.495* | 19.544 | 51.06 |
| SGA | Rank Double-Point | 19.495* | 19.495* | 46.92 |
| SGA | Rank Uniform | 19.495* | 19.539 | 26.3 |
| SA | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 19.495* | 23.218 | 39,433.62 |
| Foolish | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 19.495* | 19.495* | 379.20 |

Table 1: Small Dataset Results

| Algorithm | Parameters | Best Score | Average Best Score | Average # of Generations/Perturbations |
|---|---|---|---|---|
| SGA | Roulette Single-Point | 34.594* | 35.872 | 212.36 |
| SGA | Roulette Double-Point | 34.594* | 35.714 | 197.50 |
| SGA | Roulette Uniform | 34.594* | 35.676 | 218.98 |
| SGA | Tournament Single-Point | 34.594* | 35.153 | 183.40 |
| SGA | Tournament Double-Point | 34.594* | 34.656 | 190.28 |
| SGA | Tournament Uniform | 34.594* | 35.004 | 214.72 |
| SGA | Rank Single-Point | 34.594* | 34.975 | 172.26 |
| SGA | Rank Double-Point | 34.594* | 34.836 | 158.44 |
| SGA | Rank Uniform | 34.594* | 34.821 | 152.94 |
| SA | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 35.962 | 39.721 | 343,694.86 |
| Foolish | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 34.594* | 34.594* | 1,281.30 |

Table 2: Medium Dataset Results

| Algorithm | Parameters | Best Score | Average Best Score | Average # of Generations/Perturbations |
|---|---|---|---|---|
| SGA | Roulette Single-Point | 84.086 | 86.552 | 223.40 |
| SGA | Roulette Double-Point | 83.727 | 89.282 | 326.80 |
| SGA | Roulette Uniform | 84.796 | 92.882 | 191.20 |
| SGA | Tournament Single-Point | 83.642* | 84.603 | 315.30 |
| SGA | Tournament Double-Point | 83.642* | 84.239 | 219.40 |
| SGA | Tournament Uniform | 84.525 | 86.976 | 263.80 |
| SGA | Rank Single-Point | 83.642* | 85.314 | 248.90 |
| SGA | Rank Double-Point | 83.642* | 84.874 | 255.60 |
| SGA | Rank Uniform | 83.642* | 85.015 | 271.00 |
| SA | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 86.559 | 90.704 | 358,622.50 |
| Foolish | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 83.642* | 83.642* | 3,783.70 |

Table 3: Large Dataset 1 Results

| Algorithm | Parameters | Best Score | Average Best Score | Average # of Generations/Perturbations |
|---|---|---|---|---|
| SGA | Roulette Single-Point | 156.493 | 159.798 | 280.50 |
| SGA | Roulette Double-Point | 155.73 | 158.315 | 323.40 |
| SGA | Roulette Uniform | 161.915 | 168.685 | 199.80 |
| SGA | Tournament Single-Point | 154.626* | 155.493 | 353.6 |
| SGA | Tournament Double-Point | 154.626* | 155.318 | 271.1 |
| SGA | Tournament Uniform | 156.429 | 161.360 | 250.4 |
| SGA | Rank Single-Point | 155.494 | 157.839 | 234.7 |
| SGA | Rank Double-Point | 154.626* | 155.891 | 357.7 |
| SGA | Rank Uniform | 155.01 | 156.322 | 327.7 |
| SA | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 157.136 | 161.241 | 362,525.30 |
| Foolish | $\alpha$=0.98 $\beta$=1.02 $T_0$=10 $I_0$=1000 | 154.626* | 154.626* | 5,687.30 |

Table 4: Large Dataset 2 Results