Evolutionary Computation: The P-Median Problem

Luke Runnels

April 24, 2023

1 Introduction

1.1 The P-Median Problem

The intuitive premise behind the P-Median problem is finding all of the 'p' centers in a graph of 'blobs.'

Specifically, the graph of blobs are simply represented as a set of cartesian coordinates $\mathbf{V} = [(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$. Assuming that any coordinate in $\mathbf{V}$ can be assigned as a potential median, the goal of the p-median problem to be the subset of coordinates $V^{'} \subseteq \mathbf{V}$ where $|V^{'}| = \mathrm{P}$ such that the sum of the distances between each of the remaining coordinates in $V - V^{'}$ and the coordinates in $V^{'}$ are minimized.

2 Chromosome & Fitness Representation

2.1 Chromosome Representation

Given a vector of n cartesian coordinates $\mathbf{V} = [(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$, the chromosome is represented as a binary string of length n. The "ith" bit in the binary string corresponds to the coordinate pair (x_i, y_i). If the "ith" bit is 0, then (x_i, y_i) is in the remaining coordinate set $\mathbf{V} - V^{'}$. However, if the "ith" bit is 1, then (x_i, y_i) is in the selected coordinate set $V^{'}$. Since this problem is trying to locate "p" median coordinates from $\mathbf{V}$, the chromosome can only have "p" 1 bits exactly.

2.2 Handling Infeasible Chromosomes

Since the chromosome can only have "p" 1 bits exactly, there are cases of infeasible chromosomes in this problem. The two cases of infeasibility occur when the chromosome has more than "p" 1 bits or less than "p" 1 bits.

2.2.1 Handling more than "p" 1 bits.

If the chromosome had more than p 1 bits, then the procedure was performed until the chromosome had exactly "p" 1 bits.

Given the current chromosome state with p+n 1 bits, for every selected coordinate (x_i, y_i), a temporary chromosome was constructed with (x_i, y_i) from the selected coordinate set. Then, the fitness value from the temporary chromosome was calculated. If the fitness value of the temporary chromosome happened to be lower than the current chromosome, then the coordinate (x_i, y_i) would be keep tracked of. Once all of the selected coordinates had been scanned, the coordinate that minimized the fitness value the most was permanently removed from the chromosome, which results a chromosome containing p+n-1 1 bits. This procedure works to remove as much distance between the selected coordinates and remaining coordinates as possible.

2.2.2 Handling less than "p" 1 bits.

If the chromosome mad less than p 1 bits, then the following procedure was until performed until the chromosome had exactly "p" 1 bits.

Given the current chromosome state with p-1 1 bits, for every remaining coordinate (x_i, y_i), a temporary chromosome was constructed with (x_i, y_i) from the remaining coordinate set added. Then, the fitness value from the temporary chromosome was calculated. If the fitness value of the temporary chromosome happened to be higher than the current chromosome, then the coordinate (x_i, y_i) would be keep tracked of. Once all of the remaining coordinates had been scanned, the coordinate that maximized the fitness vaalue most was permanently added to the current chromosome, which resulted in a chromosome containing p-n+1 1 bits. This procedure works to add as little distance between the selected coordinates and the remaining coordinates as possible.

2.3 Fitness Function

Given the set of selected coordinates and remaining coordinates, the fitness function is simply the euclidean distance between each remaining coordinate and all of the selected coordinates.

Mathematically, the fitness function can be written as

$$\text{Fitness} = \sum_{i=1}^{|N-P|} \sum_{j=1}^{|P|} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{1}$$

where $|P|$ is the cardinality of the selected set, $|N - P|$ is the cardinality of the remaining set, (x_i, y_i) represents a remaining coordinate, and (x_j, y_j) represents a selected coordinate.

3 Operators Used

The basic framework of a genetic algorithm includes a selection operator for picking chromosomes proportionally to their fitness, a crossover operator for refining the chromosomes to increase their fitness through exploitation, and a mutation operator for making small

adjustments to a chromosome for exploration. This paper presents two selection operators, three crossover operators, and two mutation operators.

3.1 Selection Operators

For this project, the roulette selection and rank selection operators were chosen for experimentation. It should be noted that the p-median problem is a minimization problem. So, in the example fitness vectors illustrated below, the numerator and denominator terms will be interchanged in the implementation.

3.1.1 Roulette Selection

The idea behind roulette selection is to "spin" a wheel and select fitness values roughly proportional to their fitness. For example, given a sorted vector of fitness values $\mathbf{F} = (5, 10, 15)$, roulette selection will first sum up all of the fitness values, which results in 30. Then, fitness vector is adjusted with $\mathbf{F}^{adjust} = (5/30, 10/30, 15/30) = (0.16, 0.33, 0.50)$. A random number between 0 and 1 is chosen, and the first value from $\mathbf{F}^{adjust}$ that is greater than the random number will be the fitness value chosen, and subsequently the chromosome that is associated with the chosen fitness value is selected.

3.1.2 Rank Selection

The idea behind rank selection is similiar to roulette selection. However, given a sorted vector of fitness values $\mathbf{F} = (5, 10, 15)$, the adjusted fitness vector is $\mathbf{F}^{adjust} = (1/3, 2/3, 3/3)$. Essentially, this method literally "ranks" the fitness values from 1 and n. A same "wheel" selection procedure from roulette selection for selecting a chromosome is performed.

3.2 Crossover operators

For this project, the single point, double point, and uniform crossover operators were chosen for experimentation.

3.2.1 Single point crossover

Single point crossover takes two parent binary strings and a random splice point to produce two children. For the first child, it inherits the bits from the first parent left of the splice point and the bits from the second parent right of the splice point. Similiarly, the second inherits the bits from the second parent left of the splice point and the bits from the first parent right of the splice point.

3.2.2 Double point crossover

Double point crossover takes two parent binary strings and two random splice points to produce two children. For the first child, it inherits the bits from the first parent to the left of the first splice point and to the right of the second splice point. However, for the bits inbetween the splice points, it inherits the bits from the second parent. Similarly, in the

second child, it inherits bits from the parent left of the first splice point and right of the second splice point. However, for the bits inbetween the splice points, it inherits the bits from the first parent.

3.2.3 Uniform point crossover

Uniform crossover takes two parent binary strings and a randomized binary string to produce two children. For the first child, if the current bit from the uniform binary string is 0, then the bit from the first parent is inherited. Else, if the current bit from the uniform binary string is 1, then the bit from the second parent is inherited. In the second child, the procedure is similar, except when the current bit in the uniform binary string is 0, the bit from the first parent is inherited and vice versa for the second parent.

3.3 Mutation operators

For this project, the simple and four nearest neighbor operators were chosen for experimentation. These two operators were originally sourced from Yaser's work that surveys different facility location problems through evolutionary computation techniques[2].

3.3.1 Simple operator

In this operator, the random "0" and "1" bit are interchanged. This effectively removes one random coordinate from the selected set and replaces it with a random coordinate from the remaining set.

3.3.2 Four Nearest Neighbors/Hyper Heuristic operator

In this operator, all of the 1 bits are examined through the following heuristic. Given the four closest neighboring coordinates to the current selected coordinate, a temporary chromosome is constructing by removing the current selected coordinate and replaced with a close neighbor. If the constructed chromosome has a better fitness than the current chromosome, the constructed chromosome and fitness is tracked. At the end of this procedure, the chromosome with the best fitness is returned.

4 Parameters Used

4.1 Genetic Algorithm

For the genetic algorithm, a population size of 25 was chosen with 100 iterations. The crossover rate was set to 100% and the mutation rate was set to 5%. For every generation, elitism or copying the two fittest chromosomes is applied as well. Combinations of all of the operators described in section 3 were experimented on.

4.2 Simulated Annealing & Foolish Hill Climbing

For both simulated annealing and foolish hill climbing, the initial temperature was set to 10 and the initial number of maximum iterations was set to 200. The α parameter for reducing the temperature was set to 0.95 and the β parameter for increasing the number of iterations was set to 1.01. Combinations of the mutation operators described in section 3.3 were used as the perturbation functions.

5 Datasets Used

5.1 Small & Medium Datasets

Both the small and medium datasets were created by hand to ensure a obvious optimal. The small dataset has a solution with p=4 in a set of 20 coordinates. The medium dataset has a solution with p=8 in a set of 72 coordinates. The optimal fitness for both the small and medium datasets are 5.316 and 51.233 respectively. The optimal solutions for both datasets are illustrated below.

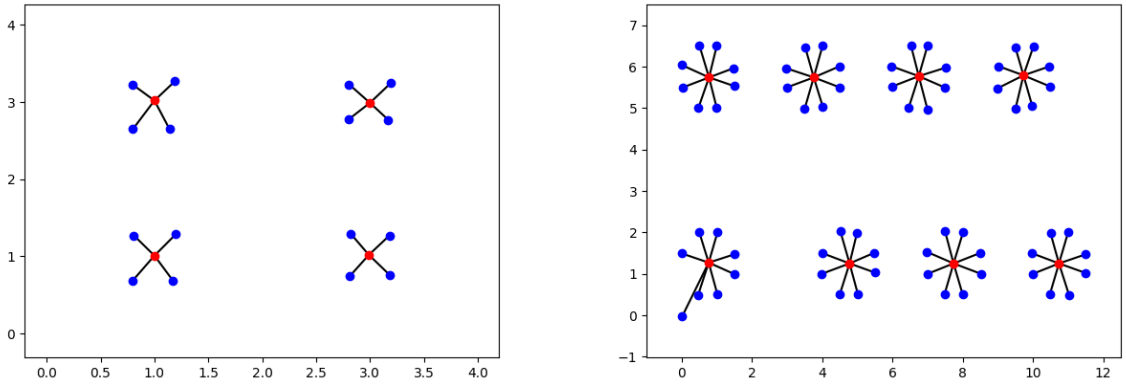

Figure 1: The small/toy dataset with 20 coordinates(left) and the medium dataset with 72 coordinates(right)

5.2 Large Datasets

Both of the large datasets were generated through as isotropic Gaussian blobs. For easy implementation, I utilized the 'make_blobs' module from scikit-learn [1]. The first dataset has a solution with p=14 in a set of 210 coordinates. The second dataset has a solution with p=15 in a set of 240 coordinates. Sample solutions from both datasets are illustrated below.

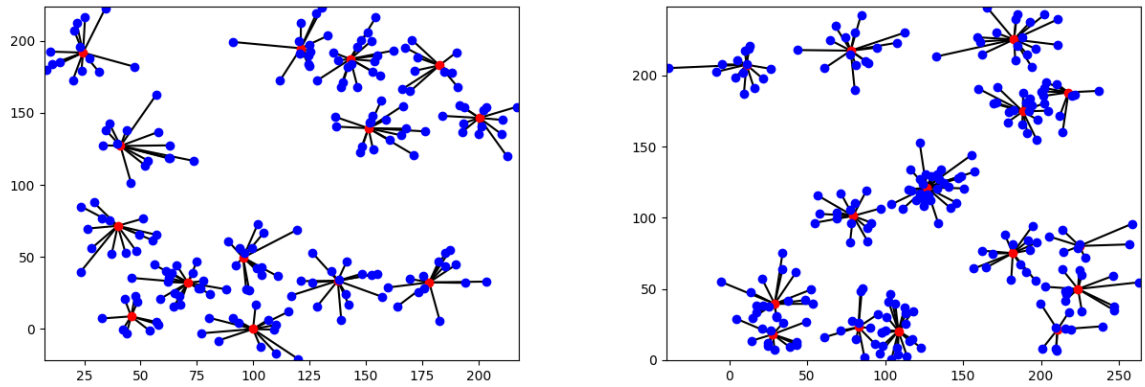

Figure 2: The two large datasets with 210 coordinates(left) and 240 coordinates(right)

6 Computational Results

The following tables illustrate the best fitness from the last generation after the genetic algorithm ran 100 iterations or after simulated annealing/foolish hill climbing lowered the temperature below 1. For the small and medium datasets, the cases that did not reach the optimal fitness are highlighted in red. For the large datasets, the case that reached the most optimal value is highlighted in green.

6.1 Genetic Algorithm Performance

Performance			
Selection	Crossover	Mutation	Best Fitness
roulette	single_point	simple	5.316
roulette	single_point	hyper_heuristic	5.316
roulette	double_point	simple	5.316
roulette	double_point	hyper_heuristic	5.316
roulette	uniform	simple	5.316
roulette	uniform	hyper_heuristic	5.316
rank	single_point	simple	5.316
rank	single_point	hyper_heuristic	5.316
rank	double_point	simple	5.316
rank	double_point	hyper_heuristic	5.316
rank	uniform	simple	5.316
rank	uniform	hyper_heuristic	5.316

Table 1: Performance of the genetic algorithm on the small dataset

Performance			
Selection	Crossover	Mutation	Best Fitness
roulette	single_point	simple	51.233
roulette	double_point	simple	51.233
roulette	single_point	hyper_heuristic	51.233
roulette	double_point	hyper_heuristic	51.233
roulette	uniform	simple	51.233
roulette	uniform	hyper_heuristic	51.233
rank	single_point	simple	51.233
rank	double_point	simple	51.233
rank	single_point	hyper_heuristic	51.233
rank	double_point	hyper_heuristic	51.233
rank	uniform	simple	51.233
rank	uniform	hyper_heuristic	51.233

Table 2: Performance of the genetic algorithm on the medium dataset

Performance			
Selection	Crossover	Mutation	Best Fitness
roulette	single_point	simple	3099.552
roulette	single_point	hyper_heuristic	3106.682
roulette	double_point	simple	3019.064
roulette	double_point	hyper_heuristic	3036.123
roulette	uniform	simple	3085.690
roulette	uniform	hyper_heuristic	3076.164
rank	single_point	simple	3007.203
rank	single_point	hyper_heuristic	2995.996
rank	double_point	simple	3013.167
rank	double_point	hyper_heuristic	2995.996
rank	uniform	simple	2998.481
rank	uniform	hyper_heuristic	2995.996

Table 3: Performance of the genetic algorithm on the first large dataset

Performance			
Selection	Crossover	Mutation	Best Fitness
roulette	single_point	simple	3735.735
roulette	single_point	hyper_heuristic	3657.720
roulette	double_point	simple	3669.624
roulette	double_point	hyper_heuristic	3637.976
roulette	uniform	simple	3735.332
roulette	uniform	hyper_heuristic	3640.597
rank	single_point	simple	3645.974
rank	single_point	hyper_heuristic	3644.480
rank	double_point	simple	3695.973
rank	double_point	hyper_heuristic	3640.896
rank	uniform	simple	3669.410
rank	uniform	hyper_heuristic	3636.792

Table 4: Performance of the genetic algorithm on the second large dataset

6.2 Simulated Annealing/Foolish Hill Climbing Performance

Performance		
Foolish	Perturbation	Best Fitness
FALSE	simple	7.112
TRUE	simple	5.316
FALSE	hyper_heuristic	5.316
TRUE	hyper_heuristic	5.316

Table 5: Performance of the simulated annealing/foolish hill climbing on the small dataet

Performance		
Foolish	Perturbation	Best Fitness
FALSE	simple	58.255
TRUE	simple	51.233
FALSE	hyper_heuristic	104.829
TRUE	hyper_heuristic	78.523

Table 6: Performance of the simulated annealing/foolish hill climbing on the medium dataset

Performance		
Foolish	Perturbation	Best Fitness
FALSE	simple	2998.188
TRUE	simple	3064.832
FALSE	hyper_heuristic	3850.611
TRUE	hyper_heuristic	3716.352

Table 7: Performance of the simulated annealing/foolish hill climbing on the first large dataset

Performance		
Foolish	Perturbation	Best Fitness
FALSE	simple	3644.774
TRUE	simple	3652.689
FALSE	hyper_heuristic	5045.427
TRUE	hyper_heuristic	3888.081

Table 8: Performance of the simulated annealing/foolish hill climbing on the second large dataset

7 Discussion & Conclusion

These results are quite interesting. In general, it seems that the genetic algorithm produced the best results. Unfortunately, simulated annealing and foolish hill climbing failed to find the optimal fitness values for the small and medium datasets. The reason as to why these algorithms failed to find them will be discussed after genetic algorithms.

Within the genetic algorithm parameters, any combination that involved the rank selection and hyper heuristic mutation performed the best. However, this combination of rank selection and heuristic mutation combined with uniform crossover seemed to produce the best results. Given that rank selection ignores the proportionality of fitness, uniform crossover provides the best opportunity for change from the crossover methods, and hyper heuristic mutation provides a greedy fitness heuristic to the algorithm, this combination's performance seems reasonable.

Despite the good performance with hyper heuristic mutation in the genetic algorithm, its performance is quite different with simulated annealing and foolish hill climbing. In general, any combination involving the hyper heuristic perturbation function produced worse results. It should be noted that the hyper heuristic function is a greedy mutation function. So, on its own, simulated annealing and foolish climbing were effectively reduced to greedy algorithms, which explains the bad results.

Surprisingly, simulated annealing itself produced the least impressive results. Foolish hill climbing seemed to produce better results than simulated annealing for the small datasets. This does make reasonable, as the fitness landscape is most likely flatter with a small dataset. So, a purely greedy algorithm will perform better in this case. Even without a greedy mutation operator, simulated annealing and foolish climbing don't provide any opportunity for exploitation what is known the state space. So, the worse results from these two algorithms do seem reasonable in this problem.

With that being said, for the large datasets, foolish hill climbing combined with hyper heuristic mutation perturbation the worst results in this study. As mentioned before, foolish hill climbing with hyper heuristic perturbation is effectively a greedy algorithm. Therefore, the particularly bad results from this combination is not very surprising.

Ultimately, for this problem and chromosome representation, I would recommend a genetic algorithm with rank selection, uniform crossover, and hyper heuristic mutation as it balances heuristics and randomness to ultimately find a good optimal fitness value.

References

[1] Sklearn.datasets.make_blobs.

[2] Yaser Alkhalifah and Roger L Wainwright. A genetic algorithm applied to graph problems involving subsets of vertices. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 1, pages 303–308. IEEE, 2004.