Connor Williams

Dr. Wainwright

CS-4623 Evolutionary Computation

April 24, 2023

Introduction

The P-Median problem is an NP-hard problem. This is a facility layout problem in which P facilities are chosen from a set of potentials in order to minimize the total distance between customers and facilities. P-Median has a wide range of real-world applications that include distribution facility layouts. This also serves as the basis for a harder problem, the Capacitated P-Median problem. In the capacitated version of this problem, each facility has a specific capacity and every customer has a request. This increases the difficulty as there adds a secondary determination of which customers go to what facilities, instead of always picking the closest. For the purposes of this project, we will be solving the uncapacitated version.

Chromosome and Fitness

For the Chromosome representation, I decided to make a bit string in which 1’s denote facilities and 0’s denote customers. I chose this representation for a few reasons. Since I am dealing with the uncapacitated variation of the problem, there is no specific need for a permutation-based approach as every customer will always go to the nearest facility. This means that permutation-based approaches are more likely to create issues and hide optimals than they are to help find them. A simpler binary string representation allows for easy computation of the total distances and gives us access to special mutation and crossover operations that could not be done with a permutation-based approach. For the fitness function, I used the summation of all the distances from customers to the nearest facility. The formula for this can be expressed as: where xP and yP represent every currently chosen facility. Infeasible solutions can be present after crossover, but in this problem, they are not particularly useful. Since every infeasible solution would have a number of facilities greater than what is present, to offset the potential fitness gains of the infeasible chromosomes they would have to be made near obsolete to avoid convergence on infeasible solutions and expanding infeasibility. To handle this fixup was applied at the crossover step that ensured that every solution contained exactly p facilities. The initial population is created by making a population of chromosomes with all 0’s and then for each chromosome randomly selecting and assigning bits to 1 equal to the number of facilities to be determined.

Operators

*Selection:*

For the selection operators I use Rank selection and Roulette selection. Rank selection works as a way to stratify data in a way that does not prematurely converge on local optimal. Each chromosome in a population is assigned a rank and selection is done across the ranks using them as integer ranges. For example, if there is a Population of 3 chromosomes with ranks 1, 2, and 3 then the integer ranges would be 1 for the first chromosome, 2-3 for the next, and 4-6 for the last. This gives a more normal distribution regardless of the range of fitnesses represented in the population. This can lead to slower convergence on good solutions, however, as they are not chosen as greedily as in other algorithms. For my implementation of this algorithm, I created a ranking matrix and then generated a random number from 1:. Then I sum the ranks until the randomly generated number is less than or equal to the current sum of ranks.  As for Roulette selection, it functions in a very similar manner, but instead of assigning ranks, it determines a percentage based on its fitness relative to the whole. For minimization, this is:   where f(Chrom[n]) is the fitness of any given chromosome. Then a random number(double) is generated from 1:sum of percentages(solves any floating point rounding problems, but is always close to 100) and a similar process to Rank selection is done, but summing the percentages instead of the ranks. This gives a different effect from the Rank selection since the choices are skewed towards more fit chromosomes more heavily. This selection technique will struggle when the differences in fitness values are small, however.  *Crossover:*

For the crossover operators I decided on uniform and two-point. The uniform crossover operator takes two bit-strings in and creates a third by randomly generating a binary string with equal chances of 0’s and 1’s in every position. Then two children (C1, C2) are created from these binary strings which we will call P1, P2, and U. For every bit in U if it is a 0 C1 will receive the bit from P1 and C2 will receive the bit from P2. If this bit is a 1, this is reversed. This performs several operations that are likely to lead to good results. First, it ignores locality within a chromosome, which, for the chromosomal representation used in this problem, likely helps perform more meaningful exploration. It also creates inverse children to explore even more efficiently. For two-point crossover, two “cut points” are randomly selected along the binary strings. The sections between the cut points are completely copied to the opposite child, and the sections that are left out of the cut points are copied to their respective children. This is meant to preserve any potentially fit subunits that may emerge within chromosomes, but since there is no inherent value in the locality of the bits this will likely not come into effect. It does allow for a meaningful exploration as cut dimensions are random, so the crossover can select subparts from anywhere on the chromosome.

*Perturbation (Mutation):*

The mutation functions for binary string representations are relatively limited. Most are just variations on bit flip so I have done something similar. The first perturbation function is standard bit flip. In this, a bit is randomly selected, and its value is inverted. For the purposes of the P-Median problem, this means that bits must be flipped in pairs to avoid infeasible solutions. This effectively means that a bit is selected from the customers and a bit is selected from the facilities and both are inverted. This function will likely work very well as a perturbation function since it provides a minimal change to exploit what is already probably a decent solution. The main problem I saw with this was scale. Since the bit flip is set to a single pair of bits depending on the number of total stations and facilities that are being searched the percentage change of a mutation operation can vary wildly. To solve this, I created N-Bit Flip which considers the number of facilities and then randomly flips some percentage of them, along with their customer counterparts. Hopefully, this will result in faster and more consistent exploitation across the varying data sizes.

Parameters Used

*Genetic Algorithm:*

For the parameters I used a population size of 100, a crossover rate of 100%, and a mutation rate of 2%. For termination, I decided to use a set number of iterations to terminate as I felt that this would provide more consistent and more easily comparable data. This was set at 200 epochs. Elitism was also implemented for every genetic algorithm.

*Simulated Annealing:*

For these parameters, I chose an alpha of .98, a beta of 1.02, a temperature of 10, and an iteration of 1000. These came from the suggested simulated annealing parameters. They were also carried over to the Foolish Hill Climbing. The perturbation functions for simulated annealing and foolish hill climbing are the same as the mutation operators from the genetic algorithm.

Datasets

I generated a total of six datasets, three of which are toy datasets and three of which are practical datasets. For the toy datasets, I decided to make a way to generate datasets with a known near-optimal solution of any size. I did this by taking in the number of total locations and the number of facilities. Then for each wanted facility I randomly placed it on a grid. Then each station is generated with a specified standard deviation of the seeded locations. This means that a near-optimal can be known for each of these datasets which is the total number of locations multiplied by the standard deviation. For the purposes of these datasets, I did a standard deviation of 3 to make sure the clustering of the data remained consistent and was likely to avoid overlaps, but larger deviations are more likely to make the solutions more difficult to find. This creates problem spaces that have large clusters of data where a near-optimal solution is a facility in every cluster and then finding the best median within each cluster improves on this (Figure 1). For the practical datasets, I generated a problem space of a completely random set of locations (Figure 2). For both of these generation methods, I created three sizes of datasets, one with 25 locations and searching for 4 facilities, one with 50 locations searching for 5, and one with 100 searching for 10.

Computational Results

 

   

For Graph representations of the largest datasets fitness see Appendix Figures 3-4. For Graph representations of solutions on the largest datasets see Figures 5-6. For this problem it seems that a genetic algorithm with a combination of Rank selection and Uniform crossover consistently works best. The mutation operator is likely dependent on the size of the dataset and the proportion of stations to facilities. Simulated annealing did not perform as well as the genetic algorithms, but consistently outperformed foolish hill climbing, especially on the larger, non-contrived datasets. Foolish hill climbing performed the worst overall as was expected, but did manage to perform relatively well on the contrived datasets.

Conclusions

The genetic algorithm seemed to work best for this problem. It consistently performed at the best level, but the specific parameters of the algorithm also seem to have a large impact on the outcome. Rank and Uniform definitely have an advantage given the problem space and chromosomal representation. Rank likely helps to exploit solutions better than Roulette, especially as they are approaching convergence. Uniform gains an advantage by having a more flexible crossover strategy it makes sense that it would outperform two-point in most cases. I was surprised that simulated annealing seemed to struggle with this problem, even in some test cases being outperformed by foolish hill climbing. I think that this is partially because of the nature of the contrived datasets and partially because of the chromosomal representation. Since the chromosome is represented as a bit string the rigidity of the perturbation functions likely struggled to keep up with the flexibility of the crossover operations. As for the contrived datasets, because of the way they are generated, there are good conditions for foolish hill climbing. Once the algorithm finds each cluster it will be able to only select improvements within each cluster easier than simulated annealing likely would. The N-Bit Flip mutations operator I created seemed promising, but probably requires some tuning to make it more viable. If I were to remake it, I would likely have the percentage of the Chromosome that is mutated be inversely proportional to the similarity across chromosomes. This should mean that when the chromosomes get too similar, any chromosome that mutates, mutates more instead of it just being done randomly within a range.

This project helped me learn on a deeper level how important the initial chromosome representation was. It seems to be both a blessing and a curse in that most of the code forms around how the chromosome is represented meaning that if it is a good representation, it can make generating new methods easy, but if it is not a good representation changing it or making new methods for it will likely be taxing. Moving forward in this project I would like to explore how the relative number of facilities to total locations changes how easy it is to find solutions. I would also like to explore how the standard deviation for contrived datasets changes the amount of time required to find the near optimal. I would also like to explore the capacitated version of this problem and see how that changes the representation of the chromosome. For the full datasets as well as more graphs, code, and training data everything is available here: https://github.com/ConnorLWilliams/P-Median\_Genetic\_Algorithm.

Appendix

Figure 1:

Chart, scatter chart

Description automatically generated

Figure 2:

Chart, scatter chart

Description automatically generated

Figure 3:

Figure 4:

Figure 5:

Chart, scatter chart

Description automatically generated

Figure 6:

Chart, scatter chart

Description automatically generated