The Cap K-Center Problem

Christopher Schaefer, 4/25/22

1. Introduction

I am solving the Capacitated K-center Problem using a genetic algorithm The Capacitated K-center Problem is a basic facility location problem. Given a set of vertices on a graph, the goal is to locate K facilities on the graph and assign vertices to these facilities to minimize the maximum distance from a vertex to its assigned facility. This problem is known to be NP-hard.

I defined the problem space as a 100 x 100 Cartesian coordinate plane. Given a set of coordinates, an edge-weighted graph using those coordinates as nodes and the distances between them as edges is created and can be used for the fitness of a genetic algorithm that locates the facilities and assigns them vertices.

I coded the solution to my problem using Python. I have the best understanding and have used Python the most in projects at Tulsa, so I figured using it as my language for the project would make the most sense. I realize there are some tradeoffs in efficiency using Python, but I consider it worth it for the purposes of this project.

2. Chromosome and Fitness

The chromosome for my problem is represented as a string of numbers. In the code, this string of numbers is manipulated as a list, so it is coded as such so there's not a constant back and forth switching between a string and a list data type. In the chromosome, the first K alleles represent the facilities that have been assigned for that chromosome. The rest of the numbers in the chromosome represent the vertices, and they are assigned to the facilities based on their position in the chromosome. The index of the facility a particular vertex is assigned to is determined by integer division of their index location over the number of facilities, or

$$Integer(\frac{Vertex\ Index\ Location}{Number\ of\ Facilities} - 1).$$

The crossover was coded in such a way to make infeasibles impossible. Originally, I allowed infeasibles and punished them in the fitness function, however this was hurting solutions and runtime so I got rid of them entirely. The original infeasible punishment is still present in the code.

The fitness function for this problem and for my solution are simple, the sum of the lengths of all the edges, as the best solution for this problem is the one with the shortest edge lengths. In practice, I created a weighted graph that contained the edge lengths between each vertex. This graph was used during the calculation for the fitness of each chromosome to get the distance between any two points and add the appropriate edge to the fitness sum.

3. Operators

The first selection function used was tournament. In my tournament function two candidates were randomly selected from the current population pool. The fitnesses of these two candidates were compared, and the candidate with the higher fitness was added to the pool of parents. Originally there was a chance for either candidate to win based on their fitness (the closer their fitness, the more

random chance either was selected), however this was complicating the problem and actually making results worse in practice, so it was removed.

The second selection function was Roulette selection. This function created a roulette wheel with weighted probabilities based on the fitness of each chromosome in the population (lower fitness = larger slice of the wheel). Chromosomes would be randomly selected and inserted into the parent pool based on a random selection of the weighted roulette wheel until the parent pool is filled.

The first crossover method used was singlepoint crossover. Two parents were taken from the parent pool at random. In this crossover, a random split point was chosen in the chromosome. Alleles from before the split point were copied over from the parents to their children respectively. After the split point, alleles in each child were added from the opposite parent in order. If one allele was already present in a child, then it was skipped. After this operation was finished, if any child was missing alleles, they were randomly added from alleles not already present in the children. This removed infeasibles and as a result of removing the infeasibles overall made the solution better.

The second crossover method implemented the use of an exchange vector. Two parents were taken and removed from the parent pool at random. Alleles of the first parent were chosen at random to be exchanged. The corresponding alleles in the second parent were then identified, and the locations of these alleles were swapped in their children. This crossover also does not allow for duplicates/repeats in the child chromosomes. The children created from this crossover were inserted into the child pool.

The first mutation operator I used was a simple pairwise exchange. Each child was selected and 10% (the rate of mutation) of the time the child was chosen for mutation. The mutation operator selected two alleles in the chromosome at random and swapped them, no matter the impact on fitness. This chromosome was then reinserted into the child pool.

The second mutation operator was a nearest neighbor mutation operation. For each allele in a child chosen to be mutated, I used the weighted edge graph to find the node closest to it. These two alleles were swapped in the chromosome. If the newly mutated chromosome is more fit than the original chromosome, then it replaces the original. Otherwise the original chromosome is kept intact. Afterwards, this chromosome was then reinserted into the child pool.

The Yaser paper described an N4N mutation that was better than other described mutations. I began to try and implement this hyper-heuristic, but it proved too difficult and the implementation affected runtime extremely, so I did not implement it in the final version of my project. This N4N mutation is what my nearest neighbor mutation is based off of.

The perturbation functions I used in Simulated Annealing were the same as the mutation functions I used for my Genetic Algorithm. The first perturbation function used in my Simulated Annealing was pairwise exchange. The perturbation operator selected two locations in the solution at random and swapped them. Then, if this solution was better than the current solution, or if an exception was to be made, this solution is set to be the new solution

The second perturbation function used in my Simulated Annealing was a nearest neighbor exchange. For each location in the solution, I used the weighted edge graph to find the node closest to it. These two locations were swapped in the solution. If the new solution is more fit than the original solution, then it replaces the original. If an exception was to be made, this solution is also then set to be the new solution

4. Parameters

The population size used in all cases with my algorithm was 100 with a crossover rate of 0.9, or 90%. The mutation rate was 0.1, or 10%. These values are the same used by Yaser in his paper. Adjusting these values in either direction either didn't change the outcome of the solution, or made it worse. I decided to terminate my program based on time and the number of generations. This mainly had to deal with the computing power I have available to me. Terminating based on the fitness itself caused the runtime to be excessive in some cases, and basing termination on the time and the generations allowed me to collect data more efficiently. I ensured to run the program for enough generations that it would converge to some solution it wouldn't improve upon without significant dedication of resources and time.In every Toy case I let the program run for 10,000 generations. For small cases I let the program run for 10,000 generations. I ran the program for 5,000 generations. I left in the option for the program to terminate when it reached a certain fitness, and it did meet this condition in some cases (specifically the toy cases).

For Simulated Annealing, my initial temperature was 20.0, my initial iterations were 1000. My α and β qwew 0.98 and 1.02, respectively, and I terminated the program when the temperature reached 1 or less than 1.

5. Datasets

My toy data set used a set of 20 coordinates representing a fairly uniform distribution.

x=[10, 0, 0, 20, 20, 10, 0, 0, 20, 20, 90, 80, 80, 100, 100, 90, 80, 80, 100, 100]

y=[10, 0, 20, 0, 20, 90, 80, 100, 80, 100, 10, 0, 20, 0, 20, 90, 80, 100, 80, 100]

The optimal for this data set is known to be 226.3. I contrived it to be calculable by hand and easy to assign. My algorithm was able to obtain the optimal for the toy data set. Simulated Annealing was also able to reach this optimal.

My second data set was small. It was generated randomly. I found that existing datasets online were either too large to be usable or unreadable/uninterpretable. For these reasons, I generated my own vertices/coordinates to be used in my program. The small dataset contained 50 data points, and can be found in the testData.txt document. The optimal for this set was unknown. (I estimated the optimal to be around 1600 based on an expected average distance, but this could vary greatly).

My Third data set was medium. Similar to the small dataset, it was generated randomly for the same reasons. The medium dataset contained 100 data points, and can be found in the testData.txt document. The optimal for this set was unknown. (I estimated the optimal to be around 3200 based on an expected average distance, but this could vary greatly).

In Yaser's paper, their small data set was ~300 nodes and their large dataset was ~600 nodes. Even attempting to use data sets this large, the time taken to run my algorithm is absurd, so the datasets chosen are more appropriate.

6. Tables of Results for the Cap-K Center Problem

(Each program was run 5 times)

Dataset Name/Size: Toy Nodes: 20 K: 4 Optimal value (if known): 226.2

		Best Edge Length Sum	Number of Generations/Perturbations (initial)	Average Best Edge Length Sum	Avg Time Taken (s)
SGA	Tournament Singlepoint Pairwise	226.2	10000	319.4	49.8
	Tournament Singlepoint Nearest N	303.8	10000	350.2	99.5
	Tournament Vector Pairwise	339.4	10000	360.5	54.4
	Tournament Vector Nearest N	226.2	10000	237.2	105.57
	Roulette Singlepoint Pairwise	226.2 (in <100 generations)	10000	226.2	156.9
	Roulette Singlepoint Nearest N	226.2	10000	277.9	315.5
	Roulette Vector Pairwise	226.2	10000	318.2	571.1
	Roulette Vector Nearest N	226.2	10000	276.0	329.8
SA	Pairwise $\alpha = 0.98$ $\beta = 1.02$ $T_0 = 10$ $I_0 = 1000$	226.2	1000	232.3	22.3
	Nearest N $\alpha = 0.98$ $\beta = 1.02$ $T_0 = 10$ $I_0 = 1000$	230.2	1000	267.7	30.1
Foolish	"Foolish" Hill Climbing Pairwise	415.7	1000	810	20.6
	"Foolish" Hill Climbing Nearest N	245.6	1000	516.4	29.1

Dataset Name/Size: Small Nodes: 50 K: 5 Optimal value (if known): Unknown

		Best Edge Length Sum	Number of Generations/Perturbations (initial)	Average Best Edge Length Sum	Avg Time Taken
SGA	Tournament Singlepoint Pairwise	1533	10000	1633	575
	Tournament Singlepoint Nearest N	1598	10000	1730	574
	Tournament Vector Pairwise	1566	10000	1680	231
	Tournament Vector Nearest N	1680	10000	1716	590
	Roulette Singlepoint Pairwise	1211	10000	1250	1325
	Roulette Singlepoint Nearest N	1404	10000	1426	1720
	Roulette Vector Pairwise	1536	10000	1601	1611
	Roulette Vector Nearest N	1539	10000	1751	1953
SA	Pairwise $\alpha = 0.98$ $\beta = 1.02$ $T_0 = 10$ $I_0 = 1000$	1584	1000	1884	45.6
	Nearest N $\alpha = 0.98$ $\beta = 1.02$ $T_0 = 10$ $I_0 = 1000$	1554	1000	1837	58.6
Foolish	"Foolish" Hill Climbing Pairwis	2112	1000	2462	46.5
	"Foolish" Hill ClimbingNearest	1949	1000	2413	55.3

Dataset Name/Size: Medium Nodes: 100 K: 10 Optimal value (if known): Unknown

		Best Edge Length Sum	Number of Generations/Perturbations (initial)	Average Best Edge Length Sum	Avg Time Taken
SGA	Tournament Singlepoint Pairwise	3665	5000	3698	280.7
	Tournament Singlepoint Nearest N	3826	5000	3826	1016.3
	Tournament Vector Pairwise	3667	5000	3701	318.6
	Tournament Vector Nearest N	3668	5000	3679	1075
	Roulette Singlepoint Pairwise	3164	5000	3191	1477
	Roulette Singlepoint Nearest N	3155	5000	3289	2140.4
	Roulette Vector Pairwise	3859	5000	3934	1705.3
	Roulette Vector Nearest N	3836	5000	3904	2420.1
SA	Pairwise $\alpha = 0.98$ $\beta = 1.02$ $T_0 = 10$ $I_0 = 1000$	4320	1000	4821	118.1
	Nearest N $\alpha = 0.98$ $\beta = 1.02$ $T_0 = 10$ $I_0 = 1000$	3372	1000	4631	146.1
Foolish	"Foolish" Hill Climbing Pairwis	5189	1000	7012	122.1
	"Foolish" Hill ClimbingNearest	4905	1000	6529	139.7

Longer times for the toy problem usually meant that the algorithm didn't always find a good answer and usually ran the full time it was allowed to. Short times mean it always found it, and usually found it quickly. Most combinations of functions could find the optimal solution for the

toy problem at least once. Roulette and singlepoint were certainly the best of their functions. The mutation operators tended to turn into a tossup but pairwise exchange seemed to be slightly favored in most cases.
The algorithm was run on each data set for enough generations to converge to a solution

7. Conclusions

The best performer across the board was Roulette, Singlepoint, Pairwise exchange. In almost every case it performed the best. Overall it seemed that roulette was the best performer in terms of selection. It typically performed better and found a solution better than tournament selection. Roulette was much easier to implement than I had first thought and tended to perform better. Roulette also performed much better with larger data sets on average.

The vector crossover I implemented looked to perform well, but not in a way that significantly affected the algorithm. It did suffer with larger datasets. Singlepoint crossover actually performed better overall. Both mutation operators performed well. The nearest neighbor mutation performed well but took a long time to run. It didn't seem worth it to use the nearest neighbor. Perhaps with a more efficient method or if the operator is actually applied as N4N, it'd be more useful. The population size of 100 worked the best for all cases while working with the GA. Any smaller and results suffered. Any larger and the time increase was significant.

Encoding the chromosome and working with the fitness was especially difficult. I had trouble assigning vertices to locations based on the chromosome. Even when I was sure my math was correct, I had trouble manipulating the weighted graph and the chromosomes. It took a lot of troubleshooting to make it work correctly. I learned a lot of the nuances of genetic algorithms and was involved in their creation. When you're theorizing and doing the math of how the chromosomes and fitness should work out it makes sense. The actual code and how the randomness shakes out isn't as straightforward and human as I thought it would be.

Based on the results of Yaser's paper, actually implementing N4N hyper-heuristic seems like a good idea and it would improve on the algorithm a lot. However, I wasn't able to successfully implement it, and even if I had been I think that the time it would take to run would be astronomical. It's definitely worth studying.

Genetic Algorithms with crossover and mutation are better than Simulated Annealing with just a perturbation, especially for larger problems. Simulated annealing seems fine for smaller problems, especially for those that maybe you don't need to build an entire GA to solve and find an optimal solution. Simulated Annealing was surprisingly easy to implement, and Foolish Hill climbing was expectedly easy to implement from there. Foolish hill climbing reinforces the element of randomness and how it's a necessary element of nature that needs to be replicated in our algorithms, and it's interesting that the results are so much worse.

8. References

Y. Alkhalifah and R. L. Wainwright, "A genetic algorithm applied to graph problems involving subsets of

vertices," *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, 2004, pp. 303-308 Vol.1, doi: 10.1109/CEC.2004.1330871.