Geometric Connected Dominating Set Problem

Final Project Report

Camp Steiner

April 29, 2024

CS 4623 - R. Wainwright

Introduction

This report details the results of applying several evolutionary computation strategies to the Geometric Connected Dominating Set problem. The Geometric Connected Dominating Set problem (GCDS), also known as the "radar problem", seeks to find the minimum subset of a graph such that every vertex in the graph is either a member of the subset or located within some specified Euclidean distance of a vertex in the subset. [1] As the moniker implies, the GCDS can be modeled as a graph of locations where the specified subsets are locations of radio or cellular towers, with the goal of providing coverage for every town in the area. For this project specifically, the graph is treated as if every vertex was connected, so any given subset is theoretically possible and no care is given to the presence of connecting edges. I approached this problem by creating a Simple Genetic Algorithm, which trained a population of chromosomes to find the optimal solution and mutated and operated on the chromosomes to approach a better solution, a Simulated Annealing algorithm, which repeatedly operated on a single chromosome to bring it near to the optimal solution, and a "Foolish" Hill Climbing algorithm, which modified my Simulated Annealing algorithm to remove the ability to accept a worse solution.

Chromosome and Fitness

As recommended by [3], I chose to represent my chromosome as a bitstring of length n, where n is the number of vertices in the graph. A `1` in the bitstring at a given position represents a vertex which is a part of the subset (a radar station), while a `0` at a given position represents a vertex which is not part of the subset and must be covered by a member of the subset (a consumer). My initial population was a randomly generated population. As I chose to use a bitstring representation, I was able to take advantage of the relationship between binary and decimal numbers for many parts of this problem, including representing a randomly generated chromosome as `random(0,` $2^{length} - 1$`)`.

I modeled my fitness function after the one described in [2]. Yaser describes a possible

fitness function for a minimization approach to the GCDS problem as

$$\frac{1}{(X * 0.8) + (Y * 0.2)} \tag{1}$$

, where X represents the number of points covered in the solution, Y represents the size of the chosen subset, and the two coefficients are experimentally derived in order to push the solution to prioritize graph coverage over subset size. I included this fitness function in my code, but also created my own fitness function that takes into account this fitness value, but also applies an additional penalty to infeasible solutions, which I defined as any solution that does not result in total coverage of every single point in the graph. My modified fitness function was

$$\frac{1}{(X * \text{coeff}_X) + (Y * \text{coeff}_Y)} + N * Z \tag{2}$$

where X and Y and their respective coefficients represented the same logic, N represented the number of nodes that were left uncovered by the particular chromosome, and Z represented a scaling factor applied to all finesses to make the 'kick' operator in Simulated Annealing function properly. As this was part of a minimization fitness equation, the addition of a value based on the number of uncovered nodes heavily penalized infeasible solutions.

Operators

I chose to implement both Roulette and Ranked selection as my reproduction operators. Both functions were implemented as minimization problems, wherein the raw fitnesses were inverted before being used to calculate the share of the roulette wheel in roulette selection and higher ranked chromosomes were given a lower weight during ranked selection.

For crossover functions, I implemented both single-point and uniform crossover, as my chromosome was a bitstring and those functions were best suited for bitwise operations. My single-point crossover function chose a random index along the length of a chromosome, and, given two parent chromosomes, returned two child chromosomes, one with the first portion of parent 1 and the second portion of parent 2, and the other the inverse. My uniform crossover generated a random bitmask and iterated over each bit sequentially,

forming the first child chromosome from parent 1 if the bitmask indicated a 0 and parent 2 if the bitmask indicated a 1, with the second child chromosome performing the inverse selection. I chose to implement elitism in my solution via in-place elitism, and allowed the top few chromosomes in each generation to bypass crossover and mutation and be placed directly into the resulting child pool.

The mutation functions I implemented were operations specifically designed for bitstrings, and as such were not order-based mutations like the ones we discussed in class. My first mutation function was a single bit flip, which chose a random bit in the bitstring and flipped its value. My second mutation function was a multiple bit flip, which iterated over each bit of the chromosome and assigned each a random chance to be flipped or not. This mathematically should result in the same total mutation as a single bit flip, as each bit had a $\frac{1}{\text{chromosome length}}$ chance to be inverted, but allowed for the possibility of no mutation or a multi-bit mutation to inspire exploration.

For my Simulated Annealing and Hill Climbing algorithm, I chose to use these two mutation functions as my perturbation functions, as the chromosome was still represented as a bitstring. I was able to save on development time for my Simulated Annealing and Hill Climbing algorithms by modeling them closely off of my genetic algorithm and using the same function references in all three instances.

Parameters

For my Simple Genetic Algorithm, I varied the parameters often to try and experimentally determine what would aid in finding the optimal solution. I found that a population size of 50 chromosomes was a fair tradeoff between finding the optimal solution and saving on processing time and memory usage, particularly on smaller datasets. I chose to feature a crossover rate of 100%, as I did not feel that variable size populations would be a benefit in this problem, and a mutation rate of 5%, as suggested in class. I elected to select the 3 most elite chromosomes to enter the child pool without mutation.

I chose to terminate my genetic algorithm if one of the following conditions was met:

- The number of generations exceeded a defined generation cap, or
- The wall time exceeded a defined time cap, or
- The variance between all chromosomes in the population was below a defined variance floor.

I created my genetic algorithm in such a way that these parameters could be specified upon creation for each instance of the GA, so that I could experiment with different values and their effects. For the sample problem I created, I found that terminating after either 100 generations, 10 minutes of wall time, or a sample variance of 5 was more than sufficient to find the optimal solution.

For my Simulated Annealing algorithm, I chose values of $\alpha = 0.95$ and $\beta = 1.01$ at the recommendation of classroom material and online reading. My heuristic function was the fitness function described above, my own modification of Yaser's minimization fitness function, and I incorporated the same fitness coefficients as in the simple genetic algorithm.

My genetic algorithm also contained parameter information about the dataset that it was trying to solve, so that the GA code was highly modular and generalizable and could be applied to different variations of the GCDS problem with little modification required. These parameters included the number of nodes in the dataset, the unsolved graph represented as an adjacency matrix encoding distance between nodes, and the radius of how far a radar node covered other nodes when selected to be a part of the subset. This information was used in the fitness function to represent how good of a solution a given chromosome was. The exact parameters used for each algorithm can be found in the exports that I created for each run.

Datasets

To test the functionality of my genetic algorithm, I first crafted a toy problem that I expected to find the solution to every single time. My toy problem took the form of a graph 25 units by 25 units large, with multiple clusters of nodes in each corner and the

middle. The nodes within each cluster were 2 units away from each other, so that when ran with a node radius of 2.5 units, there was only one valid solution, seen below.

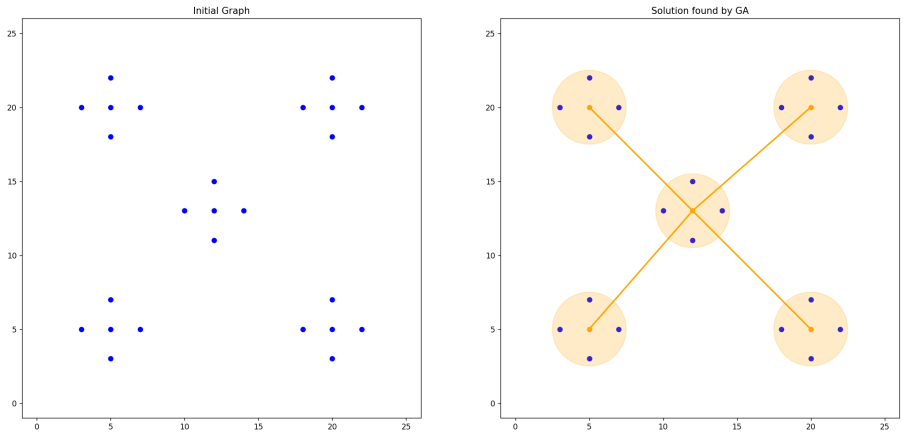

Figure 1: Graph and solution to the toy problem

After verifying that my genetic algorithm, simulated annealing algorithm, and foolish hill climbing algorithm were all able to find this optimal solution to this toy problem, I then randomly generated a dataset of 50 points on the same 25x25 grid, and ran the algorithms with the same node radius of 2.5 units.

My randomly generated dataset favored clusters of nodes as opposed to a uniformly random distribution, as such a graph was better suited for the GCDS problem. I performed small fixups on this dataset by relocating nodes that were unreasonably far away from any other nodes. The sample dataset and a relatively well-performant solution can be seen below, but as this class deals with large problems and verifying all possible subgraphs would take $2^{50} - 1$ iterations, I cannot guarantee that this solution is optimal.

I next generated a medium-sized dataset of 75 nodes on a graph 37 units by 37 units of size, which was to be run with a node radius of 3.75 units. This resulted in a computationally more expensive but theoretically equivalent graph to the previous dataset. The generated graph and a possible solution found by a genetic algorithm are seen below.

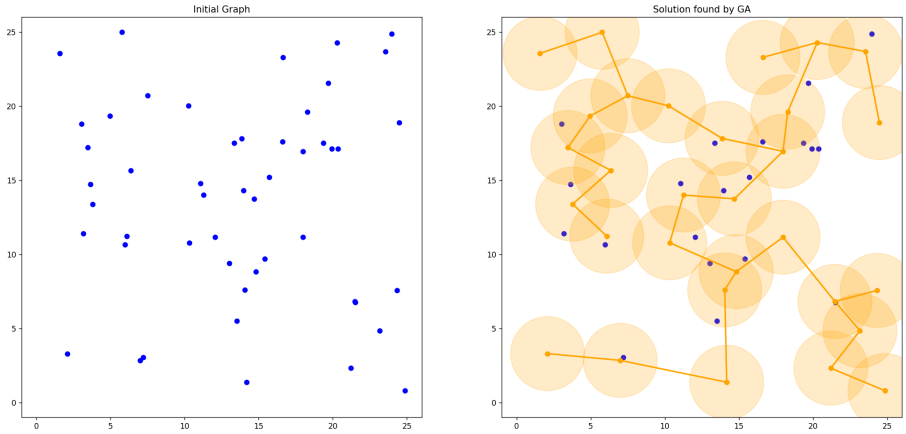

Figure 2: Graph and possible solution to the small problem

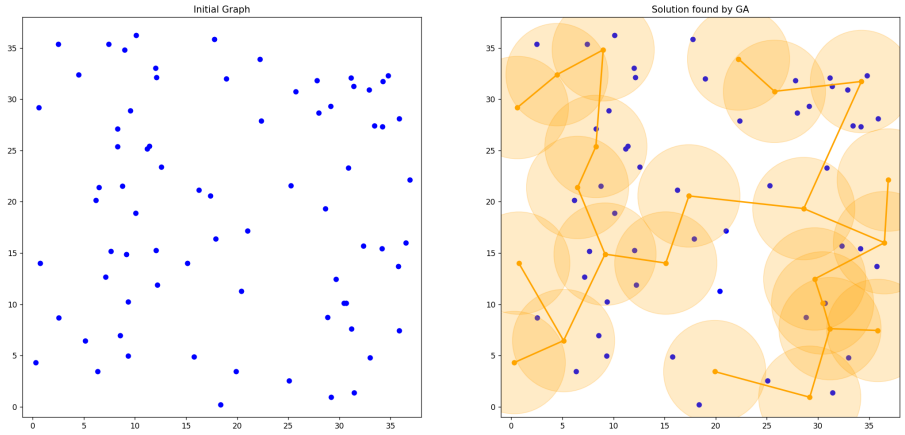

Figure 3: Graph and possible solution to the medium problem

Finally, I generated a large dataset consisting of 100 nodes on a graph 50 units by 50 units, which was to be run with a node radius of 5 units, double in every dimension compared to the small dataset. The generated graph and a possible solution are found below.

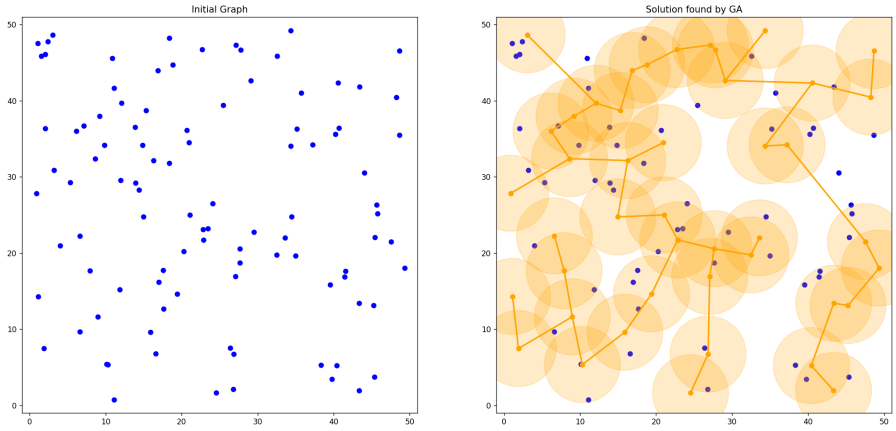

Figure 4: Graph and possible solution to the large problem

Computational Results

The functions described in this table are as follows:

Table 1: Algorithm Functions

Selection I	Roulette Selection
Selection II	Ranked Selection
Crossover I	Single Point Crossover
Crossover II	Uniform Crossover
Mutation I Perturbation I Hill Climbing I	Multiple Bit Flip
Mutation II Perturbation II Hill Climbing II	Single Bit Flip

The results for the algorithms ran on the toy problem are as follows. These results are an

aggregate of running the same algorithm with the same parameters 5 times and finding the best and average solutions of those 5 runs. The best performing solution is highlighted in bold. For the Simulated Annealing and Foolish Hill Climbing algorithms, the "Perturbations" statistic represents how many times the modified chromosome was chosen to replace the current solution during the run.

Table 2: Toy Problem

		Best Solution	Avg. Generations	Avg. Solution
SGA	Selection I Crossover I Mutation I	Fitness: 45.45 Length: 5	Generations: 81.4 Avg. Time: 23.11s	Fitness: 49.28 Length: 6.59
	Selection I Crossover I Mutation II	Fitness: 45.45 Length: 5	Generations: 88.4 Avg. Time: 25.64s	Fitness: 49.97 Length: 6.29
	Selection I Crossover II Mutation I	Fitness: 45.45 Length: 5	Generations: 72.4 Avg. Time: 20.18s	Fitness: 47.72 Length: 6.22
	Selection I Crossover II Mutation II	Fitness: 45.45 Length: 5	Generations: 76.8 Avg. Time: 20.55s	Fitness: 47.68 Length: 5.94
	Selection II Crossover I Mutation I	Fitness: 45.45 Length: 5	Generations: 71.8 Avg. Time: 10.05s	Fitness: 49.05 Length: 6.58
	Selection II Crossover I Mutation II	Fitness: 45.45 Length: 5	Generations: 73.4 Avg. Time: 10.31s	Fitness: 50.17 Length: 6.84
	Selection II Crossover II Mutation I	Fitness: 45.45 Length: 5	Generations: 60.2 Avg. Time: 8.88s	Fitness: 49.97 Length: 7.74

Continued on next page

Table 2: Toy Problem (Continued)

	Selection II Crossover II Mutation II	Fitness: 45.45 Length: 5	Generations: 82.0 Avg. Time: 11.29s	Fitness: 49.29 Length: 6.44
SA	Perturbation I a = 0.95 b = 1.01 T0 = 5.0 I0 = 500	Fitness: 45.45 Length: 5	Perturbations: 2041.6 Avg. Time: 41.66s	Fitness: 45.45 Length: 5.00
	Perturbation II a = 0.95 b = 1.01 T0 = 5.0 I0 = 500	Fitness: 45.45 Length: 5	Perturbations: 224.4 Avg. Time: 48.52s	Fitness: 49.82 Length: 8.00
Foolish	"Foolish" Hill Climbing I	Fitness: 45.45 Length: 5	Perturbations: 17.4 Avg. Time: 43.27s	Fitness: 45.45 Length: 5.00
	"Foolish" Hill Climbing II	**Fitness: 45.45 Length: 5**	**Perturbations: 16.4 Avg. Time: 44.30s**	**Fitness: 47.07 Length: 6.20**

As expected, all algorithms were able to find the solution to the toy problem. The solution to this problem can be seen in in Figure 1. The Foolish Hill Climbing algorithm was technically the most performant algorithm on this problem, as it was able to converge on the optimal solution without ever choosing a worse chromosome, allowing it to find the solution relatively quickly.

The aggregate results for the algorithms ran on the generated small dataset are as follows:

Table 3: Small Problem

		Best Solution	Avg. Generations	Avg. Solution

Continued on next page

Table 3: Small Problem (Continued)

SGA	Selection I Crossover I Mutation I	Fitness: 32.47 Length: 32	Generations: 59.4 Avg. Time: 77.54s	Fitness: 36.75 Length: 30.25
	Selection I Crossover I Mutation II	Fitness: 31.25 Length: 30	Generations: 55.4 Avg. Time: 73.94s	Fitness: 36.24 Length: 31.94
	Selection I Crossover II Mutation I	Fitness: 31.85 Length: 31	Generations: 46.0 Avg. Time: 60.27s	Fitness: 37.84 Length: 31.05
	Selection I Crossover II Mutation II	Fitness: 31.25 Length: 30	Generations: 61.4 Avg. Time: 81.04s	Fitness: 36.80 Length: 31.07
	Selection II Crossover I Mutation I	Fitness: 32.06 Length: 28	Generations: 54.0 Avg. Time: 34.83s	Fitness: 37.24 Length: 28.60
	Selection II Crossover I Mutation II	Fitness: 32.47 Length: 32	Generations: 55.8 Avg. Time: 37.23s	Fitness: 37.23 Length: 30.36
	Selection II Crossover II Mutation I	Fitness: 31.85 Length: 31	Generations: 49.8 Avg. Time: 33.21s	Fitness: 37.77 Length: 31.38
	Selection II Crossover II Mutation II	Fitness: 31.25 Length: 30	Generations: 57.8 Avg. Time: 38.70s	Fitness: 35.76 Length: 30.86

Continued on next page

Table 3: Small Problem (Continued)

SA	**Perturbation I** **a = 0.95** **b = 1.01** **T0 = 5.0** **I0 = 500**	**Fitness: 30.67** **Length: 29**	**Perturbations: 2085.8** **Avg. Time: 242.99s**	**Fitness: 30.90** **Length: 29.40**
	Perturbation II a = 0.95 b = 1.01 T0 = 5.0 I0 = 500	Fitness: 31.25 Length: 30	Perturbations: 121.0 Avg. Time: 243.78s	Fitness: 31.49 Length: 30.40
Foolish	"Foolish" Hill Climbing I	Fitness: 30.67 Length: 29	Perturbations: 18.8 Avg. Time: 245.44s	Fitness: 31.37 Length: 30.20
	"Foolish" Hill Climbing II	**Fitness: 30.67** **Length: 29**	**Perturbations: 18.0** **Avg. Time: 241.07s**	**Fitness: 30.90** **Length: 29.40**

Here, the Simulated Annealing algorithm using multiple bit flip perturbations and the "Foolish" Hill Climbing algorithm using single bit flip perturbations were able to find the optimal solution to the small problem. While the solutions found by the genetic algorithms were slightly less optimal, they converged in less than a third of the time, making them more efficient for this problem. The solution to this problem found by the top performing Simulated Annealing algorithm can be seen below.

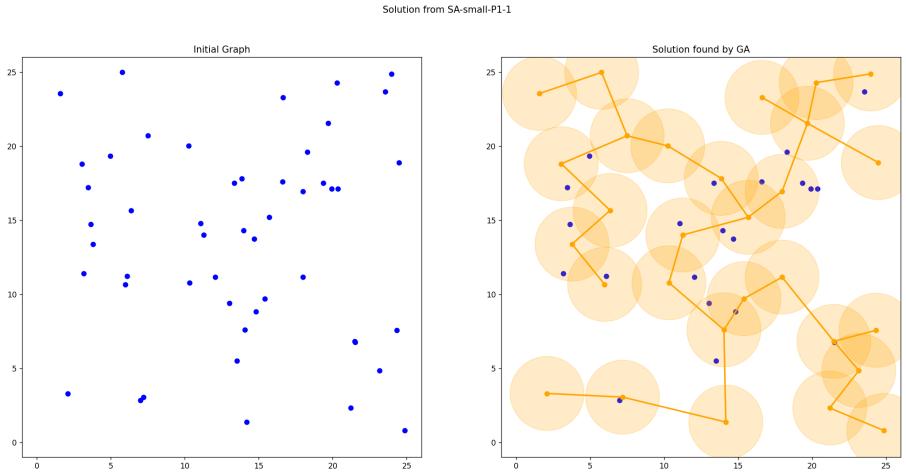

Figure 5: Solution for the small problem: chromosome from SA-small-P1-1

The aggregate results for the algorithms ran on the generated medium dataset are as follows:

Table 4: Medium Problem

		Best Solution	Avg. Generations	Avg. Solution
SGA	Selection I Crossover I Mutation I	Fitness: 21.01 Length: 44	Generations: 43.6 Avg. Time: 119.82s	Fitness: 25.03 Length: 44.34
	Selection I Crossover I Mutation II	Fitness: 21.16 Length: 39	Generations: 48.0 Avg. Time: 128.65s	Fitness: 25.79 Length: 41.71
	Selection I Crossover II Mutation I	Fitness: 20.75 Length: 43	Generations: 43.2 Avg. Time: 116.54s	Fitness: 26.89 Length: 41.56

Continued on next page

Table 4: Medium Problem (Continued)

	Selection I Crossover II Mutation II	Fitness: 19.31 Length: 37	Generations: 38.6 Avg. Time: 105.32s	Fitness: 26.59 Length: 42.75
	Selection II Crossover I Mutation I	Fitness: 19.53 Length: 38	Generations: 47.2 Avg. Time: 64.09s	Fitness: 25.18 Length: 41.33
	Selection II Crossover I Mutation II	Fitness: 21.28 Length: 45	Generations: 42.2 Avg. Time: 59.06s	Fitness: 26.62 Length: 44.02
	Selection II Crossover II Mutation I	Fitness: 20.75 Length: 43	Generations: 35.2 Avg. Time: 49.95s	Fitness: 26.50 Length: 44.36
	Selection II Crossover II Mutation II	Fitness: 20.24 Length: 41	Generations: 39.4 Avg. Time: 53.58s	Fitness: 27.35 Length: 41.88
SA	**Perturbation I** **a = 0.95** **b = 1.01** **T0 = 5.0** **I0 = 500**	**Fitness: 18.66** **Length: 34**	**Perturbations: 2072.0** **Avg. Time: 447.20s**	**Fitness: 18.83** **Length: 34.80**
	Perturbation II **a = 0.95** **b = 1.01** **T0 = 5.0** **I0 = 500**	**Fitness: 18.66** **Length: 34**	**Perturbations: 98.2** **Avg. Time: 447.40s**	**Fitness: 18.96** **Length: 35.40**
Foolish	"Foolish" Hill Climbing I	Fitness: 18.87 Length: 35	Perturbations: 24.6 Avg. Time: 457.48s	Fitness: 19.09 Length: 36.00

Continued on next page

Table 4: Medium Problem (Continued)

	"Foolish" Hill Climbing II	**Fitness: 18.66 Length: 34**	**Perturbations: 23.6 Avg. Time: 450.00s**	**Fitness: 18.96 Length: 35.40**

Again, the Simulated Annealing solution was the most performant, but took an order of magnitude longer to converge than the genetic algorithms. Between the genetic algorithms, there was not an enormous amount of variation in performance, although the algorithms ran using ranked selection over roulette selection were significantly faster at converging in roughly the same number of generations. The solution to this problem found by the top performing Simulated Annealing algorithm can be seen below.

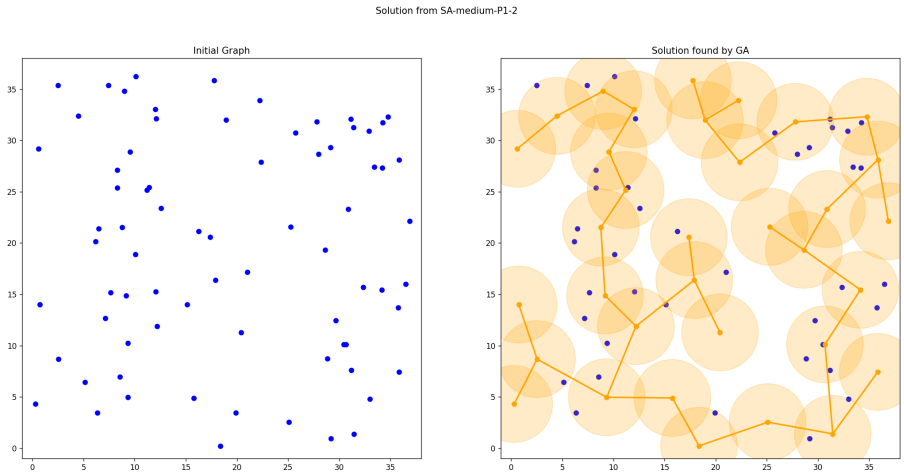

Figure 6: Solution for the medium problem: chromosome from SA-medium-P1-1

The aggregate results for the algorithms ran on the generated large dataset are as follows:

Table 5: Large Problem

		Best Solution	Avg. Generations	Avg. Solution

Continued on next page

Table 5: Large Problem (Continued)

SGA	Selection I Crossover I Mutation I	Fitness: 14.53 Length: 52	Generations: 33.4 Avg. Time: 151.77s	Fitness: 18.15 Length: 53.45
	Selection I Crossover I Mutation II	Fitness: 13.81 Length: 46	Generations: 42.4 Avg. Time: 179.51s	Fitness: 17.43 Length: 52.62
	Selection I Crossover II Mutation I	Fitness: 14.16 Length: 49	Generations: 37.8 Avg. Time: 143.11s	Fitness: 18.36 Length: 54.97
	Selection I Crossover II Mutation II	Fitness: 13.81 Length: 46	Generations: 40.8 Avg. Time: 138.70s	Fitness: 18.91 Length: 53.28
	Selection II Crossover I Mutation I	Fitness: 14.93 Length: 55	Generations: 27.0 Avg. Time: 66.48s	Fitness: 18.01 Length: 55.83
	Selection II Crossover I Mutation II	Fitness: 14.29 Length: 50	Generations: 33.0 Avg. Time: 79.58s	Fitness: 18.30 Length: 54.80
	Selection II Crossover II Mutation I	Fitness: 13.59 Length: 44	Generations: 45.6 Avg. Time: 94.23s	Fitness: 17.91 Length: 52.76
	Selection II Crossover II Mutation II	Fitness: 13.26 Length: 41	Generations: 39.4 Avg. Time: 80.80s	Fitness: 18.40 Length: 51.84

Continued on next page

Table 5: Large Problem (Continued)

SA	**Perturbation I** **a = 0.95** **b = 1.01** **T0 = 5.0** **I0 = 500**	**Fitness: 12.47** **Length: 33**	**Perturbations: 2094.0** **Avg. Time: 709.12s**	**Fitness: 12.58** **Length: 34.20**
	Perturbation II a = 0.95 b = 1.01 T0 = 5.0 I0 = 500	Fitness: 12.56 Length: 34	Perturbations: 81.2 Avg. Time: 592.38s	Fitness: 12.76 Length: 36.00
Foolish	"Foolish" Hill Climbing I	Fitness: 12.56 Length: 34	Perturbations: 35.8 Avg. Time: 623.44s	Fitness: 12.72 Length: 35.60
	"Foolish" Hill Climbing II	Fitness: 12.56 Length: 34	Perturbations: 33.2 Avg. Time: 553.21s	Fitness: 12.74 Length: 35.80

Again, the Simulated Annealing solution ran with multiple bit perturbation found the most optimal solution out of the algorithms. Here, there was a much larger difference in solution length and fitness between the simulated annealing and genetic algorithm solutions, but the genetic algorithms appeared to converge before reaching their 100 generation cap, possibly indicating that they were reaching a local maxima too quickly. The solution to this problem found by the top performing Simulated Annealing algorithm can be seen below.

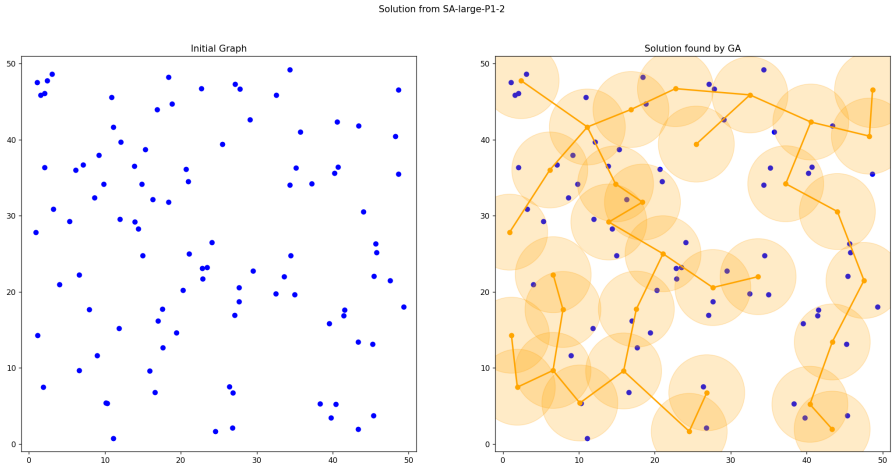

Figure 7: Solution for the large problem: chromosome from SA-large-P1-1

Conclusion

This project taught me a great deal about evolutionary computation algorithms and their potential for finding near-optimal solutions to hard problems. As stated earlier, the search space for this project was as large as 2^{100}, making any sort of deterministic exhaustive algorithm infeasible for finding an answer. However, all of the evolutionary algorithms proved to be able to find the optimal solution when it was known and a near-optimal solution for larger dataset in a matter of minutes.

I dealt with a number of unique errors in the creation of my algorithms. As referenced earlier, when using a direct implementation of Yaser's fitness function, I was seeing fitnesses in the range of 0.04-0.09. This caused the 'kick' operator to have a 99% chance of choosing the worse chromosome during simulated annealing, and the algorithm would never converge. I was able to fix this by adding a scaling factor to the fitness function, which allowed the 'kick' operator to function as intended. I also had to deal with the issue of infeasible solutions, which I solved by adding a penalty to the fitness function for every node that was not covered by the subset. This allowed the genetic algorithm to prioritize feasible solutions over infeasible ones. I also faced a number of issues in representing my

data and results in a fashion that would let me automatically run the algorithms and analyze the results later, rather than monitoring and recording each iteration manually. My solution to this problem, involving saving the entire state of a GA allowing me to reimport it and run more iterations or graphs at any point, is massively over-engineered, but I am fond of how I solved it and the technology will be useful if I ever continue my studies in evolutionary computation in the future.

Overall, I found that the Simulated Annealing algorithm ran with the Multiple Bit Flip perturbation function was the most performant. This performance, however, came with a large time cost, as the algorithm took an order of magnitude longer to converge than the genetic algorithms. The genetic algorithms were able to find near-optimal solutions in a fraction of the time, but were not able to find the optimal solution in the same way that the Simulated Annealing algorithm was. The "Foolish" Hill Climbing algorithm was able to find a near-optimal similar solution to the Simulated Annealing algorithm in a similar time.

I believe this is due to my choice of termination conditions for the genetic algorithm. I chose to terminate when the models converged to within a certain variance threshold of each other, but I believe that this is possibly causing them to converge on a good but not optimal local maxima and then become too similar before any can explore the solution space more and approach the optimal solution. I would like to experiment with either lowering the variance threshold or raising the mutation rate of my genetic algorithms to try and inspire the genetic algorithms to approach the same solution that the Simulated Annealing algorithm was able to find. However, even though the genetic algorithms possibly converged too early and did not find the most optimal solution for larger datasets, they still found a solution within a small margin of error of the optimal solution in a fraction of the time that the Simulated Annealing algorithm took to converge, which is a significant advantage in a real-world application.

To generalize the results of this project, the Multiple Bit Flip mutation/perturbation function appeared to outperform the Single Bit Flip mutation/perturbation function in all cases. The Ranked Selection reproduction operator appeared to outperform the Roulette Selection reproduction operator in terms of fitness and also in terms of processing speed, and Uniform crossover operator appeared to be more performant than the Single Point

crossover, particularly on larger datasets. This aligns with my expectations for this project, which were that functions that increase entropy and diversity of solutions would outperform functions that have a slight effect on a chromosome. If I had more time, I would like to experiment with running my genetic algorithms again with varying parameters and a higher number of generations to see if they could find the same solution as the Simulated Annealing algorithms.

References

[1] Y. Alkhalifah and R.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)* (2004). DOI: 10.1109/cec.2004.1330871.

[2] Yaser Alkhalifah and Roger L Wainwright. *220 Yaser-Review of Four Problems.*

[3] Roger L Wainwright. *180 Possible Projects to pick from.*