Set Covering Problem

# Introduction

This program is designed to solve the Set Covering Problem. The set covering problem is NP-Complete and is described as the attempt, when given a universe, to find the smallest subset of elements whose union equals the universe.

# Chromosome & Fitness

### Representation of Chromosome

The chromosome representation in this program is a binary bit string, though an alpha bit string could also be used given a different implementation. The data sets that I used gave the data in the following format:

Number of rows (m)

Number of columns (n)

The cost of each column c(j) j=1,...,n

For each row i (i=1,...,m)

The number of columns which cover row i followed by a list of the columns which cover row i.

### Encoding Data

I encoded the data into a 2D matrix according to the above scheme. Once the data was read in, I created chromosomes using a random selection of which columns to activate and which to leave off. The chromosome was placed into a 1D array which was placed into the 2D array called Population.

### Fitness Function

The fitness of each chromosome was determined in the fitEval() function. The fitness of an individual i is calculated by Sij = value of the jth bit (column) in the ith row \* cost of the column. If chromosome is infeasible, this function punishes is by multiplying a constant infeasible factor by the number of rows not covered and adding it to the fitness. This severely reduces the chance of the chromosome being chosen for procreation later. High fitness values are undesirable. During the fitness evaluation, the program takes note of the chromosome with the lowest (best) fitness and saves it for later use in the elite() function.

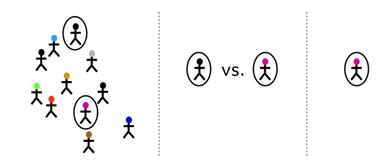
### Infeasibles

Before the fitness function can be evaluated, the population must be checked for infeasible chromosomes. This is done with the function checkFeasible() which is called at the beginning of fitEval(). In order to determine the feasibility of a chromosome in this type of problem, one need only determine whether all rows are covered by the activated columns. This is done using a comparison of the chromosome to the original data set. If all rows are not covered, the number of uncovered rows is returned to the fitness function to be dealt with. Infeasibles are not explicitly removed from the population, only penalized.

# Operators

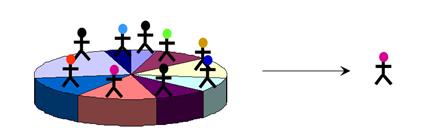
### Tournament Selection

The first selection technique implemented is Tournament Selection. In tournament selection, a portion of the population is chosen to compete in a tournament, in which the fittest member of the tournament is declared the winner. I implemented this with a random selection of members of the population, and chose to have 1/3 of the population compete in each selection. The winners of two separate calls to this function are chosen to be the parents of two children in the next generation. In my implementation, it is possible for the same member of the population to be chosen more than one time to compete in the same tournament.

1

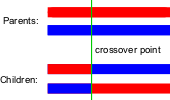
### Roulette Selection

The second technique I implemented is Roulette Wheel selection. In this method, the members of the population are entered into an array based on their fitness. The fittest members of the population receive the most room on the array and are therefore more likely to be selected for procreation. Every member of the population is represented on the array proportional to their fitness. Once the wheel array is filled, a random number is chosen between zero and the array size. This number corresponds to the index of the chosen chromosome and is returned from the method to be the parent. As in tournament selection, this method is called twice and the parents obtained will go on to have two children in the next generation.

1

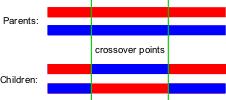
### Single-Point Crossover

The first crossover selected was a simple single point crossover. In this method, the parents chosen in the selection function are passed in. A cut point is randomly selected. A temporary parent is created to hold all values to the left of the cut point from parent 1. Parent 1 is then overwritten in every index left of the cut point by the values in parent 2. Parent 2 is overwritten by the values that the temporary parent held. The end result is that a portion of parent 1 and parent 2 have been swapped. The parents are now the children and are placed into the next generation.

2

## Double-Point Crossover

The second crossover chosen is double point crossover. This is nearly identical to the single point, except two cut points are chosen instead of one. The portion of the parents swapped is all indices between the two cut points, rather than all indices to the left of a single point. The end result is two children with the head and tails of their parents swapped. These children are moved to the next generation.

3

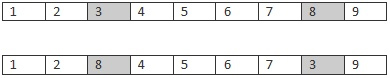
### Mutation by bit flip

The first mutation chosen is a simple bit flip. As this is a binary bit string representation, flipping a bit in a chromosome adds or removes a column from the set. Removing a column may make the chromosome infeasible. This mutation occurs on only one chromosome.

Mutation4

### Mutation by bit swap

The second mutation choice is a bit swap. Two indices in a chromosome are randomly chosen and the values are swapped between them. This has no effect if the values were the same initially. This may make the chromosome infeasible.



### Perturbation Operators in Simulated Annealing

The operators I used for simulated annealing were adding and subtracting columns from the chromosome. This achieved a neighborhood search by avoiding moving too far in the search space. Whether the Add or Remove functions were chosen was determined with a 50/50 probability. Both operators were included in the same Perturb() function for ease of access.

# Parameters Used

**Population Size:** 100 **Crossover Rate:** 70%  
**Mutation Rate:** 5%  
**Termination Condition:** 50 Generations  
**Chance of including a column in initial population creation:** 40%  
**Tournament Selection Population:** 1/3 population  
**β:** .2  
**α:** 5

# Datasets

### Toy Data

I contrived the toy data set for testing of each method and the program as a whole. The matrix obtained from the data set is as follows:

1 1 1 1 0  
 1 1 1 1 1  
 0 0 1 0 1  
 0 1 1 0 0  
 0 1 0 0 0

The optimal solution for this matrix is easy to determine manually. The best chromosome for this set would be 0 1 0 0 1 with a cost of 6. Any potential solution that is feasible must contain index 1 and either 2 or 4 or both.

Both my GA and SA algorithms are able to find this solution with regularity. The GA may find solutions that contain the optimal but are not optimal themselves due to redundancy when not given enough generations to run. This is due to not paring out redundant columns on feasible solutions. This is not an issue in the SA, optimal is nearly always achieved.

### Other data

The other data sets I used were obtained from the OR Library referenced in the assignment. I attached a link to the list in references.6

I used scp41, scp42, and scp43 from that list, but any of the data sets starting with scp should function properly in my program.  
The optimal is NOT known for any of the data sets I found.

# Results

### Trivial Data Optimal Chromosome 01001

|  |  |  |
| --- | --- | --- |
| **SGA** | **Best Cost** | **Average Cost** |
| Tournament  Single Point Crossover  Bit Flip Mutation | 6 | 6.8 |
| Tournament  Single Point Crossover  Bit Swap Mutation | 6 | 8 |
| Tournament  Double Point Crossover  Bit Flip Mutation | 6 | 7.2 |
| Tournament  Double Point Crossover  Bit Swap Mutation | 6 | 7.2 |
| Roulette  Single Point Crossover  Bit Flip Mutation | 6 | 8.8 |
| Roulette  Single Point Crossover  Bit Swap Mutation | 6 | 8.4 |
| Roulette  Double Point Crossover  Bit Flip Mutation | 6 | 8 |
| Roulette  Double Point Crossover  Bit Swap Mutation | 6 | 6.8 |
| **SA** | **Best Cost** | **Average Cost** |
| **β:** .2 **α:** 5  **T**: 100 | 6 | 18.96 |
| **Foolish Hill Climbing** | **Best Cost** | **Average Cost** |
|  | 6 | 20.2 |

### Large Data Optimal Chromosome unknown

|  |  |  |
| --- | --- | --- |
| **SGA** | **Best Cost** | **Average Cost** |
| Tournament  Single Point Crossover  Bit Flip Mutation | 3696 | 3792.6 |
| Tournament  Single Point Crossover  Bit Swap Mutation | 3616 | 3830.4 |
| Tournament  Double Point Crossover  Bit Flip Mutation | 6557 | 3681.4 |
| Tournament  Double Point Crossover  Bit Swap Mutation | 3583 | 3735.8 |
| Roulette  Single Point Crossover  Bit Flip Mutation | 3709 | 3828 |
| Roulette  Single Point Crossover  Bit Swap Mutation | 3609 | 3750.6 |
| Roulette  Double Point Crossover  Bit Flip Mutation | 3723 | 3876.6 |
| Roulette  Double Point Crossover  Bit Swap Mutation | 3631 | 3778.8 |
| **SA** | **Best Cost** | **Average Cost** |
| **β:** .2 **α:** 5  **T**: 100 | 5939 | 6161 |
| **Foolish Hill Climbing** | **Best Cost** | **Average Cost** |
|  | 6280 | 6522 |

### Medium Data Optimal Chromosome unknown

|  |  |  |
| --- | --- | --- |
| **SGA** | **Best Cost** | **Average Cost** |
| Tournament  Single Point Crossover  Bit Flip Mutation | 13970 | 15697.2 |
| Tournament  Single Point Crossover  Bit Swap Mutation | 14224 | 15113 |
| Tournament  Double Point Crossover  Bit Flip Mutation | 14097 | 15062.2 |
| Tournament  Double Point Crossover  Bit Swap Mutation | 14732 | 15544.8 |
| Roulette  Single Point Crossover  Bit Flip Mutation | 14605 | 15519.4 |
| Roulette  Single Point Crossover  Bit Swap Mutation | 14478 | 14935.2 |
| Roulette  Double Point Crossover  Bit Flip Mutation | 15113 | 16027.4 |
| Roulette  Double Point Crossover  Bit Swap Mutation | 14859 | 15392.4 |
| **SA** | **Best Cost** | **Average Cost** |
| **β:** .2 **α:** 5  **T**: 100 | 24765 | 26401 |
| **Foolish Hill Climbing** | **Best Cost** | **Average Cost** |
|  | 25908 | 28415 |

### Small Data Optimal Chromosome unknown

|  |  |  |
| --- | --- | --- |
| **SGA** | **Best Cost** | **Average Cost** |
| Tournament  Single Point Crossover  Bit Flip Mutation | 4473 | 5040 |
| Tournament  Single Point Crossover  Bit Swap Mutation | 4410 | 5103 |
| Tournament  Double Point Crossover  Bit Flip Mutation | 4914 | 5153.4 |
| Tournament  Double Point Crossover  Bit Swap Mutation | 4788 | 5178.6 |
| Roulette  Single Point Crossover  Bit Flip Mutation | 4725 | 4997 |
| Roulette  Single Point Crossover  Bit Swap Mutation | 4788 | 5052.6 |
| Roulette  Double Point Crossover  Bit Flip Mutation | 4536 | 5115.6 |
| Roulette  Double Point Crossover  Bit Swap Mutation | 4347 | 5065.2 |
| **SA** | **Best Cost** | **Average Cost** |
| **β:** .2 **α:** 5  **T**: 100 | 7875 | 8789 |
| **Foolish Hill Climbing** | **Best Cost** | **Average Cost** |
|  | 8127 | 9465 |

# Conclusions

Working with this software was certainly a unique experience for me. The largest learning experience I had from it was that I did not truly understand the genetic algorithm process when I began. In fact, I programmed the SGA in full and left it for 2 weeks before realizing my results didn’t make any sense. Upon returning to the program and researching more in depth implementation instructions, I realized I was not penalizing infeasible chromosomes. Adding a large penalty made the results much more favorable. In the future, the SGA could be perfected by adding a redundancy check for minimizing the cost of the covering set. At this time, the solution is found but not guaranteed to be optimal in all cases.

When comparing the SGA, SA and FHC algorithms, I found the SGA to be the best result by far. The optimal was nearly always found in the contrived data set, and would be found 100% of the time with an increase to the number of attempts made. The different operators did not seem to change the average results much, with a few percentages offset in one direction or another. The most noticeable difference was only in computational time of Roulette vs Tournament. This could likely be attributed to my implementation rather than the operator itself.

I was very pleased with this project and it tied together the subject of this semester very well, which is not something I have found to be true in many of my other programming classes. In order to complete this project successfully, I needed to understand the subject very clearly, requiring careful study of the material as well as outside research.

The problem I chose had very little guidance available regarding particulars of implementation, unlike more common choices like the TSP. This simply meant I spent more time gaining understanding of what exactly I needed my algorithms to do than if I had been able to follow one of the many guides available for other problem types, which was preferable. I believe this project made me a better programmer as the size of it required me to create unit tests, which has not been necessary in the past.

# References

1. <http://www.insisoc.org/metanorms/User_Guide.htm>
2. <http://en.wikipedia.org/wiki/File:SinglePointCrossover.png>
3. <http://www.peltarion.com/doc/index.php?title=Synapse:Genetic_optimizer>
4. <http://www.talkorigins.org/faqs/genalg/genalg.html>
5. <http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5>
6. <http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/>