Final Project

Dane Copple, Zach Shumate, Adam Vest
Computer Engineering & Computer Science
Speed School of Engineering
University of Louisville, USA
dacopp03@louisville.edu, znshum01@louisville.edu, apvest01@louisville.edu

1. Introduction

This project wanted the students to use the Genetic Algorithm and Wisdom of Crowds ideas taught over the previous two projects, while allowing them to apply the ideas to any NP-complete problem of their choice. The NP-complete problem the students chose is the 3-SAT problem, which is a special case of the Satisfiability problem (SAT problem). A SAT problem has boolean variables and boolean clauses that said variables are used in. The focus of this problem is to have the entire sequence of clauses evaluate to being True by finding the right values for each of the variables. In the 3-SAT problem, however, an added restriction of having exactly 3 literals in each clause present. The methodology for both cases involves examining each clause where variables are used and finding the value that would have all of the variable's clauses evaluate to True, or at the very least help them down that path. The issue here, is that these problems need to be solvable in the first place due to the how complex they can get with enough growth. This is why solvable test data was gathered from http://www.cs.ubc.ca/ and used within the code (full link in the References section).

2. Approach

The project involved making a GA with a WoC implementation, so naturally that was the approach needed for the solution. The input arguments were the first thing that were added to the code, and there's quite a few arguments to work with. The only required argument is the --data_path, which is a user defined path to the input .cnf file. The rest of the arguments have default values. Should the user want to adjust these values, they can do so by stating the one of

the arguments below along with a value, such as “--data_path=./folder/file.cnf”. Should one of the input arguments not be an option (like for the various strategies), it will throw an error and exit the code saying which option was invalid. Please note that all of the arguments are lower case.

The --initialization_strategy is in charge of how the population is generated. Though, there is only one method in use, so it defaults to “random”. This simply creates a population with random values for each variable in the arrays. The --population_size dictates how large the population is, and it’s entirely user defined. It defaults to 100 if there is no specific input. The --selection_strategy is in charge of picking the mating pairs between the population, though there is only the bin selection, which the program defaults to. The --number_of_bins goes along with this argument, and it dictates how many bins are allowed for the selection process. The bin number defaults to 5.

The --crossover_strategy is in charge of the crossover method used between mating pairs for reproduction, and its options are random, standard, and greedy. “Random” is a function that runs through the array size of both parents and selects one gene from each parent for each variable value, much like a coin toss. “Standard” creates two children by randomly appending values from both parents to each child. After the two children are made, they are tested using the fitness function and the winner between the two is chosen. “Greedy” follows the same idea, but rather than doing it for each child, the function does it for each value in the array. So if one parent has 0 for a value while the other has 1, the function checks which value satisfies more clauses in the 3SAT problem and picks it. This argument defaults to “random”.

The --mutation_strategy is in charge of the mutation function used on each child that mutates, and it can be either point or pairswap. “Point” is the default, and it flips the value of a randomly chosen bit within the child. “Pairswap” swaps the values of up to 5 randomly selected variables within the child. The --mutation_prob goes along with the mutation_strategy, and it dictates the probability of mutation within the children. It defaults 0.2, which is a 20% chance to mutate within our mutation function.

The --woc_strategy is in charge of which WoC function is used for the program, but the only value is basic, which it defaults to. The basic function creates an array of values and has the population vote on which value should be used. If a person has the value 1 for a variable, the array would add 1 to the variable’s spot in the array, while if it was 0, it would subtract 1. The function selects 1 or 0 if the values are positive and negative, respectively. Once the program has run through the population, it decides on which value should be used and generates a new WoC array accordingly.

The --generations_limit is in charge of the number of generations the program will run without having any improvement. This is goes hand in hand with the --improvement_limit, which dictates how many more clauses need to be satisfied for the generations_limit to reset. The default values for these are 500 and 1, respectively.

The last two arguments are for the user to use. The --visualize_results argument creates line graph of the best child from each generation until termination. This is set to 1 (or on) as default. The --save_results argument simply saves the graph created, and it is set to 0 (or off) by default.

Another class is made for the SAT problem in particular. This class is used for generating the sequence of clauses for the population to solve, in addition to being in charge of the fitness function for the populace. Being a class makes it very easy to work with and call when needed, so it's mainly there for less headaches when programming.

Classes aside, the main workflow of the program begins with reading in user arguments and translating the data from the .cnf file into arrays for the variable and clauses. From there, the population is generated at random and put into the fitness function. The fitness function finds the costs of each person in the populace by checking how many clauses are satisfied, then decides who should mate based off of bin selection. Once enough parent pairs have been made, the program moves over to the reproduction aspect, where the crossover and mutation functions are done. Both of these have default values, but can be mixed and matched as the user desires based on the strategies that are available.

After the new children are made, they are each tested by the fitness function and checked against the current best child. If one of them exceeds the best child in terms of number of correct clauses, then that child will become the new best child and the generation counter will be reset to 0. Otherwise, the program will iterate up a generation. After the best child is or is not found, the children become the new parents and the GA begins anew.

It repeats in a while loop until the generation without improvement count is met, or if a solution to the problem is found. Once either of those situations happens, if the user has it enabled, a line graph will display the best children across all generations relative to their cost, 0 meaning the best cost (since all clauses will have returned True). If the user has the saving function enabled, it will also save the graph in the images folder on the computer.

3. **Results** (How well did the algorithm perform?)

3.1 **Data** (Describe the data you used.)

The test data used for this project was provided from a website that hosted benchmark problems from the SATLIB Solvers Collection. We tested problems that contained a variety of different variable and clause counts that were stored in .cnf files. From this site we used Uniform Random 3-SAT problems that contained 20 variable and 91 clauses, 125 variables and 538 clauses, as well as 250 variables and 1065 clauses; each of these groups were a folder containing at least 100 test files. All of the data used in our tests were satisfiable problems.

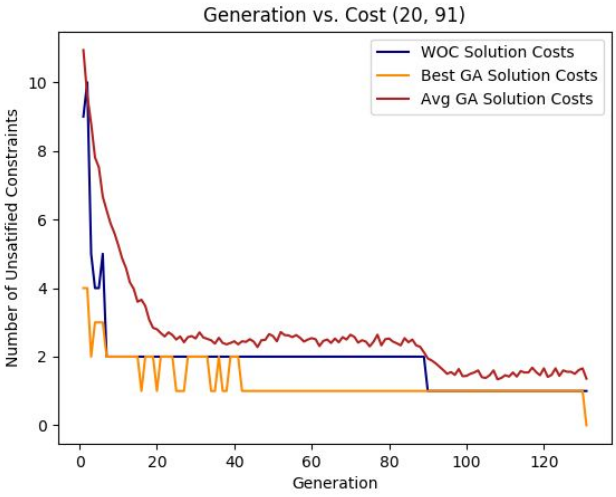
3.2 **Results** (Numerical results and any figures or tables.)

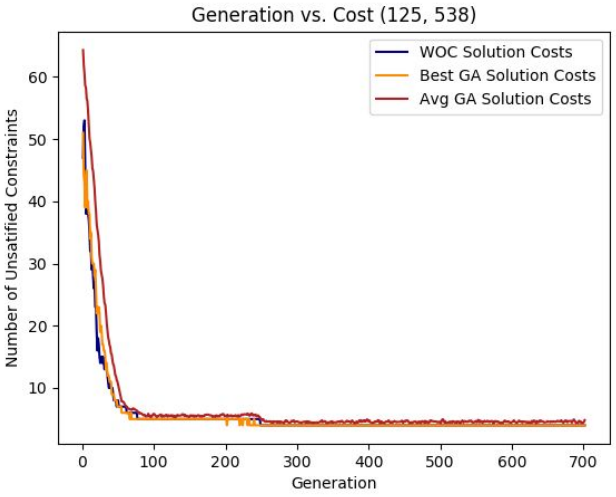
Below are the results of experiments comparing three different crossover operators for solving the 3SAT problem using a genetic algorithm. The examined crossover operators (random, standard, and greedy) were tested on problems of three different sizes: (1) 20 variables and 91 clauses to satisfy, (2) 125 variables and 538 clauses to satisfy, and (3) 250 variables and 1065 clauses to satisfy. For all reported results, the genetic algorithm population was initialized randomly, parents to mate were selected using the bin based selection technique with 5 bins (k=5), and a point mutation strategy was used with a mutation probability of 20% for the mutation operator. The stopping criterion for the algorithm was 500 generations without improvement (i.e., without the genetic algorithm satisfying an additional clause), and a population size of 100 individuals was used. In the tables below, the best GA solution refers to the overall best solution (in terms of number of satisfied clauses) that was found by the genetic algorithm; similarly, the best WoAC solution refers to the overall best WoAC solution produced during the course of the algorithm. The final average GA solution refers to the average number of clauses satisfied by all of the individuals in the population of the genetic algorithm during its final generation. The reported results are the average of three different trials each performed on different 3SAT problem instances.

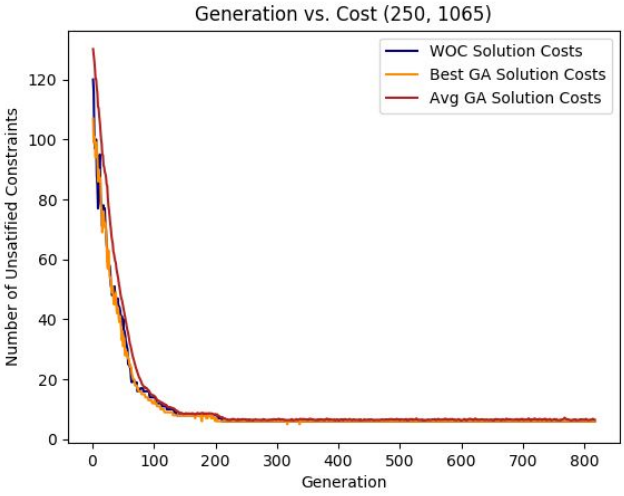
Random Crossover				
Problem Size (Number of Variables, Number of Clauses)	**Best GA Solution Number of Unsatisfied Clauses**	**Final Average GA Solution Number of Unsatisfied Clauses**	**Best WoAC Solution Number of Unsatisfied Clauses**	**Execution Time (s)**
(20, 91)	.66	2.48	1.66	3.75
(128, 538)	5	5.78	5	84.89
(250, 1065)	8.33	9.22	8.66	161.24

Standard Crossover				
Problem Size (Number of Variables, Number of Clauses)	**Best GA Solution Number of Unsatisfied Clauses**	**Final Average GA Solution Number of Unsatisfied Clauses**	**Best WoAC Solution Number of Unsatisfied Clauses**	**Execution Time (s)**
(20, 91)	0	1.52	1	.49
(128, 538)	4.33	4.71	4.33	188.14
(250, 1065)	9	9.31	9	439.39

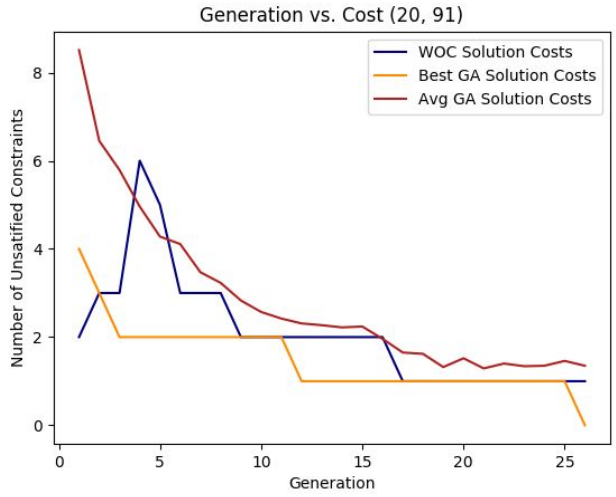
Greedy Crossover				
Problem Size (Number of Variables, Number of Clauses)	**Best GA Solution Number of Unsatisfied Clauses**	**Final Average GA Solution Number of Unsatisfied Clauses**	**Best WoAC Solution Number of Unsatisfied Clauses**	**Execution Time (s)**
(20, 91)	2	4.08	4	9.08
(128, 538)	24	26.18	26	59.97
(250, 1065)	44	47.12	47	186.62

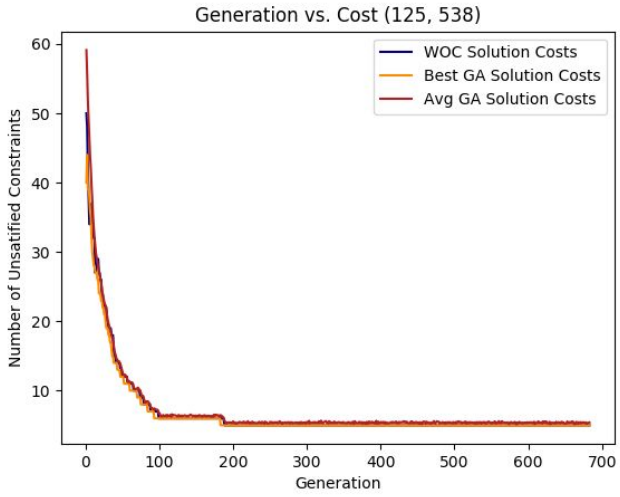


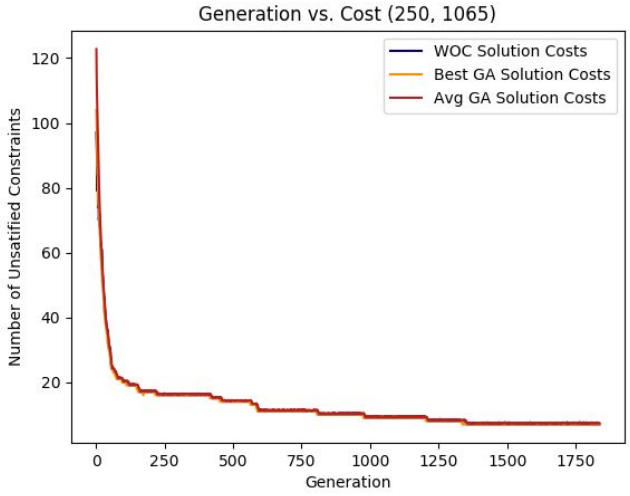




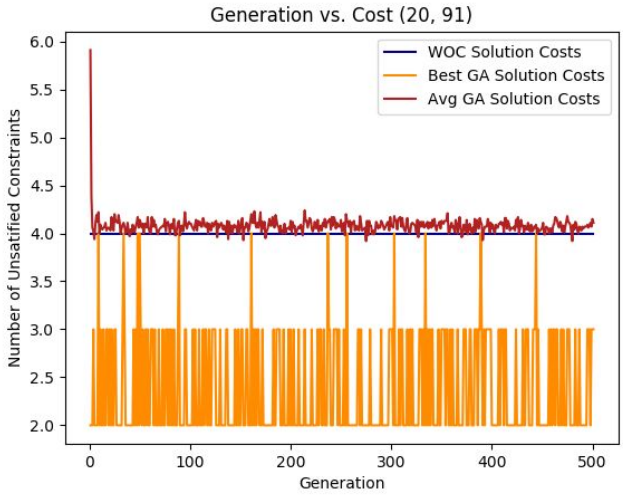
Figs 1-3. Sample Random Crossover Result Graphs

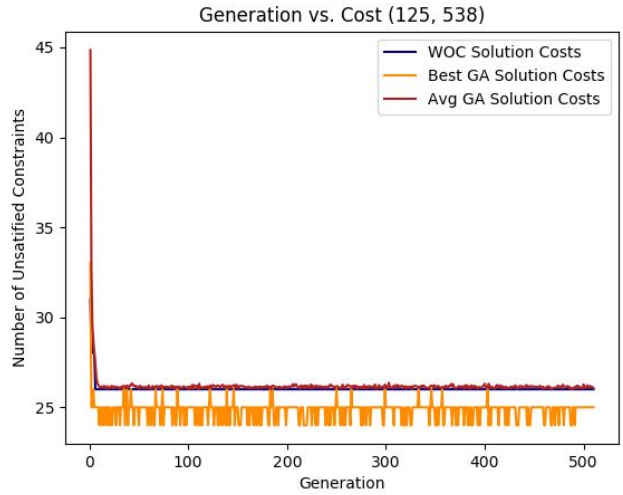


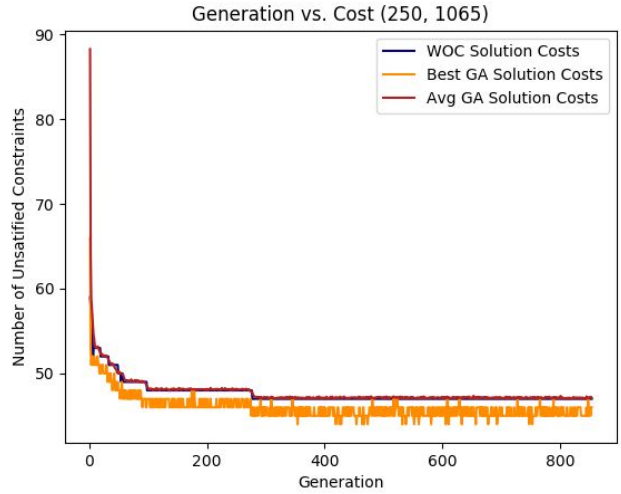




Figs 4-6. Sample Standard Crossover Result Graphs







Figs 7-9. Sample Greedy Crossover Result Graphs

4. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)

The performance of the Wisdom of Artificial Crowds and pure genetic algorithm approaches shown above show some distinct patterns. First, the agreement-based Wisdom of Artificial Crowds always outperformed the average pure genetic algorithm solution but was usually outperformed by or had equal performance to the best genetic algorithm solution. This trend can be seen in both the results graphs and tables. For each of the performed experiments, the average solution in the final generation of the algorithm was found to be outperformed by the best WoAC solution; further, the results graphs indicate that that the WoAC solution produced during a given generation almost always satisfied more clauses than the average GA solution. This trend can be observed in the results graphs as for all but five generations in the course of all the experiments utilizing the random or standard crossover operators the WoAC solution outperformed the average GA solution. Additionally, all five of these generations occurred in experiments with small problem sizes (20 variables and 91 clauses to satisfy), suggesting that the WoAC approach confers additional benefits on larger problem sizes. While the WoAC approach was able to produce better solutions than the average GA individual, it was generally unable to outperform the best individual in the population of the genetic algorithm. In each of the performed experiments, the best GA solution was able to satisfy more or an equal number of constraints to the best WoAC solution; however, for 11 out of 12 trials when the standard or random crossover operators were used on larger problem sizes (125 or 250 variables), the WoAC solution produced a solution of equal quality to the best GA individual. These results suggest that the proposed WoAC approach's performance is bounded by the best individual in the GA but will converge given sufficient time for the mutation/crossover that produced the best individual to be propagated throughout the population.

There were also distinct patterns in the observed performance of the crossover operators used in our experiments. The first and most obvious of these is the almost complete failure of the greedy crossover operator to produce quality solutions. This problem was unable to solve even the smallest problem instances and produced WoAC solutions that had more than 5 times as many unsatisfied constraints on average than when either the random or standard crossover were used. Additionally, though the standard and random crossovers performed similarly in our experiments, the standard crossover seems to perform better on small problem instances while the random operator performed better on large instances in terms of time and solution quality. The standard crossover managed to completely solve the 20 variable problem in all three trials and did so in 13% of the execution time of the random crossover. However, the random crossover operator produced better results on average on the largest problem instance (250 variables, 1065 clauses) and did so in 36% of the execution time of the standard crossover. These results suggest that random crossover operator may result in better exploration of the search

space by escaping local minima while the standard crossover operator may be more capable of exploiting productive solutions identified early in the execution of the algorithm.

5. References

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK3/NODE112.HTM

https://en.wikipedia.org/wiki/Boolean_satisfiability_problem#3-satisfiability

Test data:
http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html