**Maximizing Code Coverage with Genetic Algorithms**

GitHub Link: <https://github.com/jacksoncasey00/genetic-algorithm>

One of the most important aspects of testing applications is achieving proper code coverage. After all, if not all lines are code are executed during your tests, then how can you be sure that the code you are testing is free of bugs? However, achieving maximum code coverage is more difficult than it seems. The more complex your code gets, the more difficult it is for humans to test every edge case. This is especially true in black-box testing, where the user is not familiar with the source code and can only view inputs and output responses. This means that there is a high demand for automated tools that generate test cases to achieve maximum code coverage. This project attempts to create such a tool using a genetic algorithm.

Before I get into the details, I want to quickly explain what a genetic algorithm does. A genetic algorithm is an algorithm designed to generate test inputs by mimicking evolutionary biology. In other words, by running the test cases through a process that simulates natural selection, we can generate test cases to cover all parts of the code. The basic genetic algorithm process goes like this. First, we generate a population of random inputs representing test cases. Then, we run each test case on our function we are testing and assign each test case a fitness score depending on how well it performs. In this case, fitness is dependent on how high the code coverage is. Once we run all the test cases, we select a subset of the population with the highest fitness scores. After that, we produce a set of child test cases by performing crossover and mutation with the selected parent population. Finally, we combine the child population with the selected population and repeat this process.

For this specific project, I decided to focus on functions with string inputs. I did this for a few reasons. One, I wanted to focus on creating an algorithm that works well, not one that covers all possible inputs but has bad results. With this, I could tailor my application to strings and make it as accurate as I can. Second, I always see integer inputs being used as examples of genetic algorithms working and I was curious if a genetic algorithm could handle string inputs just as well. For running this algorithm, you simply run main, specifying the name of the function you are testing, how many generations you want the algorithm to run for, the population size per generation, and the conditions you set for the algorithm to stop. You can either set the algorithm to stop once all branches have been executed at least once, or when 100% code coverage is reached. If neither condition is met, execution will end when the max number of generations is reached.

The genetic algorithm was tested on eight different functions and evaluated based on the average code coverage achieved per run. To walk you through an example run, I will use one of my sample functions, called “testProgram”:

A screenshot of a computer program

Description automatically generated

The purpose of this program is to spell out the word ‘generic’ by checking each character for the correct letter. The idea is that the closer the test inputs get to spelling the word, the more code coverage is achieved. I will now walk you through a test execution of this function on the genetic algorithm.

When the algorithm is first called, it just generates a bunch of random strings of varying length:

A screen shot of a computer

Description automatically generated

You’ll notice, however, that strings of the correct length are showing a higher code coverage percentage. Within a few generations you will get an output that looks like this:

A screen shot of a computer

Description automatically generated

Now, all that’s left is to get the characters right. These next few screenshots show the progression of the test inputs getting closer and closer to the optimal input until it finally reaches the max code coverage:

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

As you can see, this particular execution took 160 generations to reach the optimal test case, which is not ideal. Future work will need to look into making the genetic algorithm more efficient.

Outside of this testing program, 7 other simple programs were tested on the genetic algorithm. All 8 functions were tested on their average code coverage when given 100 generations and a population of 20. These are the results:

|  |  |
| --- | --- |
| **Function** | **Avg. Code Coverage** |
| Test Program | 95% |
| Function A | 65% |
| Function B | 100% |
| Function C | 95% |
| Function D | 83% |
| Function E | 100% |
| Function F | 100% |
| Function G | 100% |

The results seem to show that this genetic algorithm performs well on these example programs. However, it is important to remember that these example programs are quite simplified, and performance is likely to change on more complex programs.

There is much that can still be done to improve this algorithm. I will go over a few ideas on where to take this project in the future. First, and most obvious, it is important to allow functions with all different kinds of inputs to be tested on with this genetic algorithm. Crossover and mutation will have to be treated differently in order to handle this, but it is an important next step to take for this project. Next, various features in the algorithm must be refined. To name a few possibilities, the mutation rate could vary depending on the code coverage, and test case selection could be tournament based or rank based to add a little more nuance to the process. We could also drastically increase the usability of this program by making it easier to add your own function to test and providing a more detailed and accurate report on the genetic algorithm performance. Finally, a much wider range of programs need to be used to test this genetic algorithm. That way, we can be sure that it is as good as it possibly can be.

While not perfect, this genetic algorithm has shown to be useful in generating test cases to achieve high code coverage. Of course, more work will need to be done in order to have a satisfactory genetic algorithm that can handle most functions. However, I have learned a lot from this project, and I hope that in the future, a genetic algorithm is created that can fully automate the test generation process and achieve maximum code coverage.