Levi Stone

Kalp Patel

001039052

00776088

Optimizing the Genetic Algorithm Through Improved Mutation Heuristics

Levi Stone

Kalp Patel

Executive Summary

Our idea for this project is to implement an optimization to the Genetic Algorithm regarding its mutation phase. The genetic algorithm is an Artificial Intelligence algorithm that is essentially a simulation of a population in nature as reproduction occurs and the population changes over time. It is used to find solutions to problems when one does not necessarily care about the path to the solution itself, only finding it. There are several steps to the genetic algorithm such as binary string representation, initial population creation, crossover, mutation, selection of parents, evaluation fitness function, and finally the termination point. We will specifically be changing the mutation aspect of the algorithm but for a summary of all steps to the algorithm, the reader can check the proposed algorithm section. An important aspect to understand though regarding this summary, is binary string representation itself. Binary string representation essentially means that we will represent a target (or our goal) as a binary string (a string of binary numbers; either 0 or 1) and will also represent our individuals of our population genetic code, as binary strings as well. This makes the implementation of our fitness function (or a function used to evaluate the fitness, or the correctness of the individual's genetic code when compared to our goal) easier to implement and maintain.

The mutation step of the genetic algorithm typically plays a very small role in the overall process. It is a step that is generally only reached under 1% of the time (for each individual in our population) as mutations in nature itself are not that common. We use this step to ensure that we do not get stuck on a specific binary string in a population and to also randomize our offspring a bit each generation. It works by creating a random number between some range of numbers and when our random number is smaller than some other predetermined number, we mutate that individual (this is how we change the mutation rate, or the probability than any one specific individual's genetic code will mutate). Essentially, what we do after that is flip a random number of bits in random positions (or change random numbers in the genetic code). So for example, if we have selected an index of our binary string that contains a one, we will flip it to a zero and vice versa. This is where our optimization change comes into effect. The change we have implemented to this step is that as opposed to flipping entirely random bits of our individual's binary string, we will instead only flip random bits that do not match our target or essentially only flip a random number of incorrect bits in random indexes.

Background information

In order to discover the best solution(s) to a given computing problem that maximizes or minimizes a specific function, genetic algorithms are a sort of optimization method. The first genetic algorithm was developed by John Holland in 1975. A lot of the syntax used in genetic algorithms comes from biology since they are intended to replicate biological processes. In the generic algorithm the strongest chromosome survives while the weakest gets discarded. Genetic algorithms reflect the genetic make-up and biological development of biological organisms. In recent years, machine learning and AI have increasingly used genetic algorithm. Genetic algorithm uses the "survival of fittest" approach.

Define the problem

The problem we are fixing with the genetic algorithm is essentially the run speed by optimizing and making it run faster. For target's with relatively small binary string goals, this impact will still be noticeable but not as significant. For instance, we have two programs in which we have implemented this change for experimental results. One of the programs is used to solve the n-Queens problem (a problem corresponding to the correct placement of n Queens on a n x n chess board such that none of the Queens are attacking each other) and the other program is used to "crack" (or find the correct representation) of a String. The n queens problem has a binary string goal of size 24 while the string cracker has a binary string goal size of 115. This makes the string cracker much more likely to have to iterate through many more generations before finding the goal. In fact, this program typically has to iterate through somewhere above 5000 generations before finding the goal string (before our change). The problem with this is it is extremely slow to iterate through thousands of generations to find a solution. We want to cut down this time and speed with our change to the mutation portion of this genetic algorithm.

Brief Literature Reviews

Genetic Algorithms (GA) are a family of well-known optimization meta-heuristics, and extensive research has been done on their behavior. The genetic algorithm is a member of the family of evolutionary algorithms. In order to provide approximations for optimization issues, evolutionary algorithms draw inspiration from nature and maily mimic what happens in natural evolution.

Your Proposed Algorithms and Applications

Our proposed algorithm, as discussed before, is an optimization (speed wise) to the genetic algorithm. As stated in the executive summary, we will first summarize the different

steps of the genetic algorithm (excluding mutation, since it was already covered) in case the reader is unclear on any of the steps. We already discussed binary string representation in the summary but our programs represent their target strings as binary strings (one of which is 24 in length with each 3 bits corresponding to the position of a queen in a row, and the other of which is 114 in length with each 5 bits corresponding to an index of an array of characters that stores our possible letters for cracking our string) Initial population creation is exactly what it sounds like, it is creating a random initial population of N individuals (not to be confused with creating a new generation as this is only called once in the process). In our programs, we use a population of 30 individuals per generation for our n-Queens problem and a population of 50 individuals per generation for our string cracker. Crossover is the process of creating an offspring for a new generation after randomly selecting its two parents from our current generation. It works by taking a random cross-section of one of the parent's binary strings and inserting it into the same position of the other parent's binary string (to create the individual's new unique binary string). Selection of parent's refers to the way we select the parent's of our current offspring. There are several ways to do this but the way we are doing it in these programs is by selecting the top 10% of our previous generation and breeding them to create a new population. Our evaluation fitness function for these programs is to compare every index of the individual's current binary string to our target string and give it a score based on the percentage correct. Lastly, our termination point (or the point in which the reproduction ceases) is when our fitness reaches our desired value for a single individual in the current population.

Implementation and Experimental Results

As stated before, we added this optimization to two programs utilizing the genetic algorithm. For the code of the mutation method before and after the change, check the end of the paper. As for our experimental results, we ran each of the algorithm's (the genetic algorithm without the change and with the change) 10 times each and got the average generation count that was required to find the solution for both programs and these are our output results. Code snippet and results of the experiment are provided below.

```
public static String mutation2 (individual i1, String target) {
    Random rand = new Random();
    boolean runner = true;
    int num =0;
    while (runner == true) {
        if (!i1.values.isEmpty()) {
            num = rand.nextInt(Math.abs(i1.values.length()));
            runner = false;
        }
        else {

        }
    }
    String newvalues = "";
    for (int i=0; i <num; i++) {
        int bit = rand.nextInt(i1.values.length());
        if(i1.values.charAt(bit) != target.charAt(bit)) {
        if(i1.values.charAt(bit) == '0') {
            if(bit ==0) {
                newvalues = "1" + i1.values.substring(bit+1);
            }
            else if (bit == i1.values.length()) {
                newvalues = i1.values.substring(0, i1.values.length()-1) + "1";
            }
            else {
                newvalues = i1.values.substring(0, bit) + "1" + i1.values.substring(bit+1);
            }
        }
        else {
            if(bit ==0) {

                newvalues = "0" + i1.values.substring(bit+1);
            }
            else if (bit == i1.values.length()) {
                newvalues = i1.values.substring(0, i1.values.length()-1) + "0";
            }
                else{
            newvalues = i1.values.substring(0, bit) + "0" + i1.values.substring(bit+1);

            }

        }
        }
    }
    return newvalues;
}
```

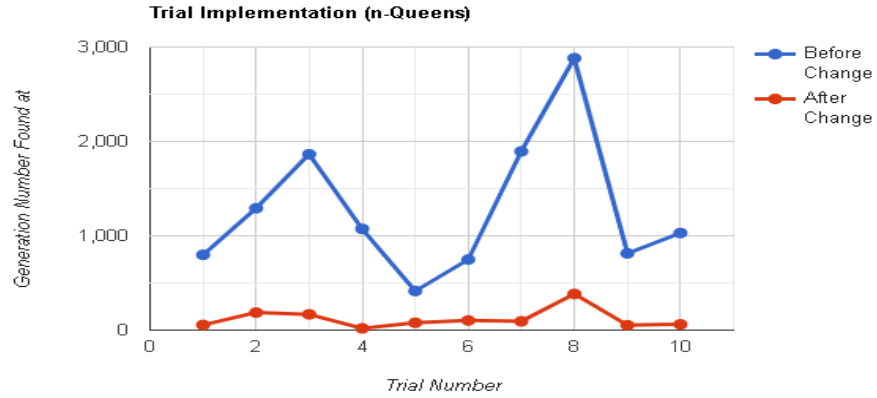
```
Correct solution found at index 36 of generation 7262
Correct solution found at index 13 of generation 7919
Correct solution found at index 19 of generation 7946
Correct solution found at index 10 of generation 6476
Correct solution found at index 22 of generation 8587
Correct solution found at index 46 of generation 7213
Correct solution found at index 42 of generation 8735
Correct solution found at index 33 of generation 7340
Correct solution found at index 8 of generation 7402
Correct solution found at index 16 of generation 4949
Correct solution found at index 1 of generation 676
Correct solution found at index 33 of generation 777
Correct solution found at index 2 of generation 861
Correct solution found at index 27 of generation 541
Correct solution found at index 28 of generation 647
Correct solution found at index 38 of generation 746
Correct solution found at index 1 of generation 511
Correct solution found at index 2 of generation 775
Correct solution found at index 23 of generation 776
Correct solution found at index 38 of generation 736
Average generation count w/o change = 7382
Average generation count w change = 704
```

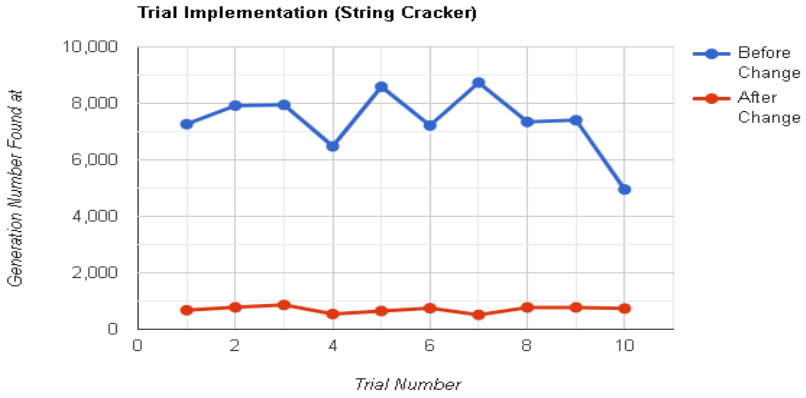
String Cracker output

```
Correct solution found at index 25 of generation 798
Correct solution found at index 17 of generation 1292
Correct solution found at index 11 of generation 1864
Correct solution found at index 6 of generation 1073
Correct solution found at index 21 of generation 414
Correct solution found at index 22 of generation 748
Correct solution found at index 27 of generation 1894
Correct solution found at index 23 of generation 2881
Correct solution found at index 11 of generation 813
Correct solution found at index 2 of generation 1029
Correct solution found at index 2 of generation 55
Correct solution found at index 26 of generation 186
Correct solution found at index 19 of generation 167
Correct solution found at index 22 of generation 19
Correct solution found at index 28 of generation 78
Correct solution found at index 16 of generation 102
Correct solution found at index 1 of generation 93
Correct solution found at index 26 of generation 383
Correct solution found at index 15 of generation 52
Correct solution found at index 0 of generation 61
Average generation count w/o change = 1280
Average generation count w change = 119
```

N-Queens output

As you can see this resulted in an average of a 10x speed increase and 10x less generation's needed to find our target solution.





Advantages and Drawbacks

Our suggested algorithm has undergone changes that affect how it behaves generally (speed wise). We will just flip random bits that do not match our target, or simply only flip a random amount of incorrect bits in random indexes, as opposed to flipping completely random bits of our individual's binary string. The only drawback is that the population is less randomized than normal.

Summary and Conclusion

In conclusion, our goal was to find a way to improve the speed of the genetic algorithm through the mutation portion of the algorithm. We succeeded in finding this improvement and implemented it into two separate programs using the genetic algorithm to run test trials to see the impact of our improvement. From these test trials, we learned that our improvement to the Genetic Algorithm's mutation step, caused our two programs to show up to a 10x speed increase of finding our solution by reducing the number of generations required to find the solution. This

improvement could prove especially useful when implementing a genetic algorithm that is searching for a relatively large binary string target as these types of problems result in many more generation's required to find the solution in the first place as was shown in the string cracker program. In both of our example programs, (n-queens and string cracker) the maximum number of generations required throughout our 10 trial runs of our improved algorithm, are all significantly less than the minimum number of generations required without the improvement. Overall, this change to the mutation step of the genetic algorithm could prove useful for a multitude of different problems (especially ones requiring long solutions to be found thousands of generations in) while it could also be potentially detrimental to others (that rely on the aspect of more randomness throughout the algorithm).

References

S. Lan and W. Lin, "Genetic algorithm optimization research based on simulated annealing," 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016, pp. 491-494, doi: 10.1109/SNPD.2016.7515946.

Matoušek, Radomil. "GAHC: Improved Genetic Algorithm." *SpringerLink*, Springer Berlin Heidelberg, 1 Jan. 1970, https://link.springer.com/chapter/10.1007/978-3-540-78987-1_46.

"Genetic Algorithms." *GeeksforGeeks*, 27 Apr. 2022, https://www.geeksforgeeks.org/genetic-algorithms/.

H. Jiang, G. Xu and Z. Deng, "Research of Multi-objective Optimization Based on Hybrid Genetic Algorithm," 2009 Fifth International Joint Conference on INC, IMS and IDC, 2009, pp. 1996-1999, doi: 10.1109/NCM.2009.136.