Using Genetic Algorithms to Solve Computer Science Olympiad Optimization Problems

Group Code: #0030

Can Almelek, Özgür Güzeldereli
Uskudar American Academy - Prof. Dr. Hulusi Behçet Cad., Necile Cevdet Apt., 26/6, Istanbul, 34728, Turkey
Uskudar American Academy - Ethem Efendi Cad., Nidapark No:22, Erenkoy, Kadikoy, Istanbul, 34728, Turkey

Abstract

Genetic algorithms are random search algorithms based on biological evolution theory. In this study, the utilization of genetic algorithms in solving optimization problems from informatics olympiads is investigated. It is hypothesized that genetic algorithms will be able to propose solutions to the optimization problem within a predetermined percent margin of the actual solution. A genetic algorithm is implemented in the C# programming language to test this hypothesis. Three tests are conducted and their data is examined with various parameters taken into consideration. Consequently, the implemented genetic algorithm has been demonstrated to successfully solve the varying number of inputs (500, 1000, 2000) within an acceptable range. It is also concluded that several improvements and optimizations have to be made to utilize this algorithm in a competitive landscape as time and memory constraints are exceeded. Furthermore, the tests concede that the number of inputs and the amount of generations required to converge on an optimal solution are directly correlated. Therefore this outcome should be taken into consideration when designing and tuning genetic algorithms with various parameters, if the problem is aimed to be solved within time and memory constraints of the competitive landscape.

Keywords

Competitive Programming; Genetic Algorithms; Olympiad in Informatics; Optimization

Introduction

Computer science has become the epitome of global development into a more modern and enhanced society. With countless advancements in the fields of Artificial Intelligence (AI), Virtual Reality (VR), Augmented Reality (AR), robotics technologies, etc; computer science has substantiated its crucial role in the new age. Therefore, there is a constant endeavor to improve the effect of computer science by coming up with new methods as well as educating more people. To be precise, the percentage of high schools offering computer science courses in the US has increased from 35% to 51% in the last 3 years [1]. This is a

testament to the fact that more education institutes, students, teachers, and individuals emphasize computer science and its potential in the future.

As computer science's popularity prospers, there have been numerous means to assess one's knowledge and skills on a computer. There are coding websites, mainly Codeforces and Hackerrank designed to impose problem-based coding with occasional contests in a competitive manner. This system of problem-based coding in a competitive manner is essentially called competitive programming. The more broad rendition would be Olympiad in Informatics on regional and international scales. The world's most prestigious computer science competition is the International Olympiad in Informatics (IOI), attracting 327 contestants from 97 countries in IOI 2019 Baku, Azerbaijan; who have passed several stages in national contests. [2] In these contests, there are common problem types such as Graph Theory, Game Theory, recursion which are regularly displayed in problem tasks. One of the key objectives is problems that require finding the best solution from all feasible solutions, optimization problems. There are numerous algorithms designed to solve optimization problems with the motive of being the most optimal solution. Genetic algorithms (GAs) are one of the most fundamental algorithms that incorporate biology in genetic classification into computer science as a viable solution method. [3]

This study demonstrates the utilization of genetic algorithms in solving optimization problems from the olympiad in informatics. It is hypothesized that genetic algorithms would be able to propose a solution within a percent margin of accuracy to olympiad problems which are conventionally solved by other means, given the correct parameters.. Within this paper, the structure of genetic algorithms will be explained, optimization problems are going to be defined, our implementation of a genetic algorithm will be shown, and with the given implementation an optimization problem will be attempted to be solved.

Discussion

Genetic Algorithms

Genetic Algorithms are optimization algorithms that are inspired by natural selection. Genetic algorithms were first designed by John Holland at the University of Michigan in the early 1970s. [4] They involve processes such as crossover and mutation to optimize for a constraint. Genetic algorithms can be used to generate feasible and high-quality solutions for search and optimization problems. Although the structure of genetic algorithms can vary slightly with their implementation, most, if not all, genetic algorithms have several commonalities: initialization, selection, offspring generation, mutation, and termination.

Initialization

All genetic algorithms start with an initialization step. Genetic algorithms consist of a population of individuals. Individuals carry a gene sequence or DNA. The DNA of an individual represents a solution to the problem that is aimed to be solved and the DNA should be structured accordingly. Therefore each individual is required to be initialized to ensure

they have at least some DNA. Although many techniques could be applied to initialize the individuals, two techniques are most prevalent. Of these two methods, assigning random genes to individuals is the most common. The other method is seeding the individual which signifies giving random genes that could be precise to the optimal value of the system or the problem. Initializing the individuals is quintessential to a genetic algorithm. Randomly assigning gene values makes genetic-diversity possible within the population which is key to spanning the entirety of the solution set of a particular problem.

Selection

After individuals are initialized, the selection process takes place. Besides a DNA sequence, the other metric every individual has is the fitness value of their DNA. The fitness value of an individual determines how optimal it is, or how close the individual is to an optimal solution relative to other individuals. The fitness value is calculated via a fitness function which is a problem-specific function determining the optimality of a solution that is proposed by the genes of an individual. Therefore it is important to form a feasible fitness function and it is the hardest process that is involved in constructing a genetic algorithm. If the fitness function fails to produce high-quality fitness values, the genetic algorithm will not be able to create high-quality solutions to the problem. After a valid fitness function is created, every individual's fitness is calculated. Then the individuals are sorted in the population according to their fitnesses and a percentage of lowest fitness individuals is discarded. Some low-fitness individuals are left in the population to ensure the genetic diversity within the population.

Offspring generation

After the selection process, a new empty generation is created and filled via the crossover of the previous generation's individuals. Crossover is the process in which the genes of an individual are mixed with another individual's genes to create offspring. The process involves selecting random points in the genes of the individuals. Then two empty children are created. The genes of the children are set to be the gene of one of the parents (one for each child). In the points that are randomly selected, the parent that the child is getting the gene from switches to the other parent thus crossing over. This process can be repeated multiple times to ensure more diverse results. Crossover's main purpose is to generate new offspring from high fitness individuals to further increase the fitness of the population. By selecting from high fitness individuals and crossing over their genes, a gene sequence with a higher fitness level is created.

Mutation

As with crossover, the mutation is a genetic operator that is used in creating higher fitness solutions in a genetic algorithm. The mutation is the process of randomly changing the

gene of the individual. There could be several consequences of this action. Firstly a higher fitness solution could be reached which is the optimal case. Another case would be a lower fitness solution that could be reached in which case the mutated child will not be able to pass its genes to the next generation. Either way, genetic diversity is created within the population to ensure that the system doesn't converge to a local optimum. By randomly changing genes of the children of the new generation, enough new genes are inputted to the system to widen the search space of the algorithm. In elitist genetic algorithms where the fittest individuals of each generation are passed directly to the next generation, the mutation is also excluded from elite individuals to protect the progress of the genetic algorithm as mutations can lead to a decrease in fitness.

Termination

Termination is the last step of a genetic algorithm. The termination of the genetic algorithm may depend on several factors and several termination conditions. For example, the number of generations could be constrained as well as the time spent on the problem. Other constraints include but are not limited to reaching a certain fitness level, the solution satisfies the bare minimum requirements of the problem, or a point is reached where the successive generations no longer produce new results. Usually, after termination, a solution is outputted as a solution to the problem or optimization. Data from each generation can be used to plot data graphs and the evolution of the population can be analyzed to draw further conclusions.

Pseudocode of a Basic Genetic Algorithm

1. Initialization - All individuals of the population are initialized with n-length genes which represent a solution to the problem.
2. Fitness - Fitness of each individual is calculated using a fitness function.
3. New generation - Follow proceeding steps to create a new generation.
4. Selection - A percent of the population is selected. The higher the fitness the higher the chance of selection.
5. Crossover - From the selected individuals, randomly select 2 parents. Crossover their genes to make 2 new offspring.
6. Accepting - Place new offsprings in a new population
7. Elitism - If the genetic algorithm is elitist, select the highest fitness individuals of the population and also place them into the new population.
8. Replace - Use the new generated population for another run of the algorithm.
9. Test - Test for a termination condition. If the condition is met, terminate the algorithm and return the necessary data as a solution to the problem or optimization.
10. Repeat - Go to step 2.

Optimization Problems

As aforementioned, optimization problems are the problems that require finding the best solution out of all feasible solutions. The whole entity of feasible solutions is often referred to as sub-optimal (local) solutions [4]. A specific goal is asked to be met at an optimal value out of all sub-optimal solutions. These problems must provide the variable constraints as specifications necessary for constructing a solution. Accordingly, programmers are generally obligated to find the minimum and maximum values of a function that satisfies the task's objective.

Moreover, optimization problems can be classified in two manners according to their variables: continuous or discrete. In the cases of discrete variables in an optimization problem, an integer, graph, or permutation must be identified out of a countable set. In discrete optimization, some or all of the variables in a model are required to belong to a *discrete set*; this is in contrast to continuous optimization in which the variables are allowed to take on any value within a range of values. The discrete set is mainly initialized as subsets, combinations, graphs, or sequences. On the other hand, the cases of continuous variables require a continuous function to be found. Continuous optimization means finding the minimum or maximum value of a function of one or many real variables, subject to constraints. The constraints usually take the form of equations or inequalities.

In general, a rather popular method for solving optimization problems is Dynamic Programming (DP). DP is essentially a recursive solution with repeated method calls, storing the results of subproblems for the eventual solution. Compared to the aforementioned genetic algorithms, DP solutions usually use a lot of memory and have a longer running time. Even though genetic algorithms are typically harder to implement, efficiency in time and memory capacity are highly valuable assets for competitive programming which makes genetic algorithms more effective in Olympiad in Informatics.

Utilizing Genetic Algorithms for Optimization Problems

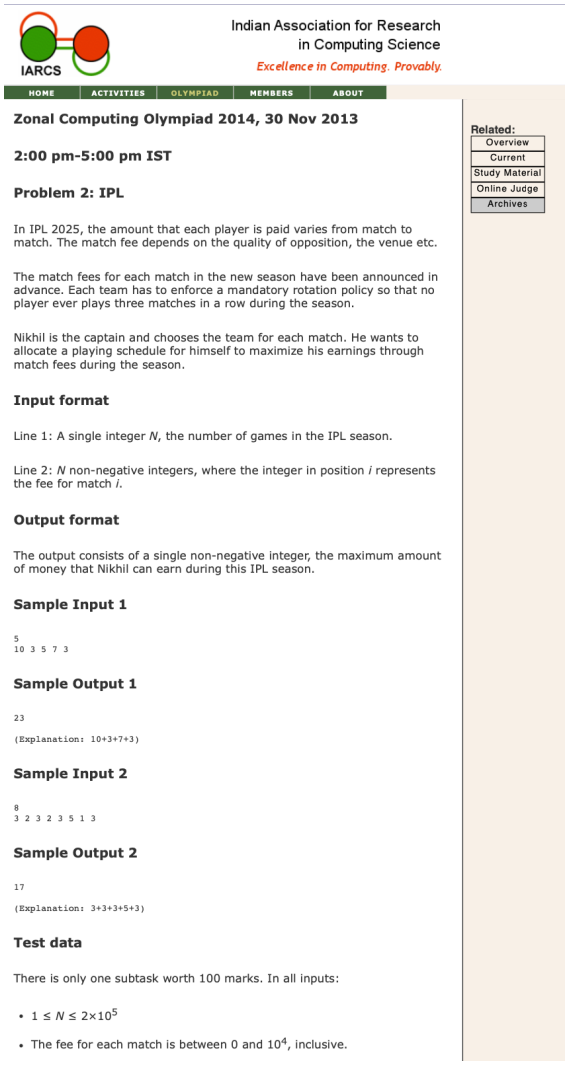
Genetic Algorithms can be considered systematic random search algorithms where the random search algorithm also considers the optimality of previous trials and evolves accordingly. Randomly initializing the individuals causes the proposed solutions to span the search space. With each generation, the solutions then attempt to converge at a global optimum. Mutations add genetic diversity to the gene pool and aim to ensure that the point the population converges is the global optimum. Knowing these it can be inferred that a genetic algorithm may not be able to converge on the most optimal solution even with the best possible parameters as it is still a randomized and probabilistic algorithm. Although that may be the case, it could be expected that the final result of a genetic algorithm is close to the global optimum. For the set of runs that will be conducted in this paper, it is assumed that the problem is solved if the value of the solution proposed by the algorithm is within 5% of the actual solution after 2000 generations. The percentage values of how close the solution is to the actual solution are calculated using the following formula:

$$\left|\frac{proposed\ solution - actual\ solution}{actual\ solution}\right| \times 100$$

Another factor to consider is, genetic algorithms are intrinsically parallel compared to most algorithms which are serial. Serial algorithms only permit linear exploration that does not store every value reached by the algorithm's infer steps. On the contrary, genetic algorithms contain multiple offspring which allows multiple dimensional explorations. Depending on the accuracy of the offspring, some are eliminated, whereas some are pursued as promising solutions. Due to the parallelism, they can operate resembling a graph structure, evaluating multiple traceries at once. Thanks to their structure, particularly problems containing vast amounts of potential solutions are easier to solve which would take an immense amount of time with a simple search algorithm.

Utilization of Genetic Algorithms on a Olympiad in Informatics problem

Figure 1, From [10], ZCO 2014 Problem 4 IPL



IARCS

Indian Association for Research in Computing Science

Excellence in Computing. Provably.

HOME | ACTIVITIES | OLYMPIAD | MEMBERS | ABOUT

Related: Overview | Current | Study Material | Online Judge | Archives

Zonal Computing Olympiad 2014, 30 Nov 2013

2:00 pm-5:00 pm IST

Problem 2: IPL

In IPL 2025, the amount that each player is paid varies from match to match. The match fee depends on the quality of opposition, the venue etc.

The match fees for each match in the new season have been announced in advance. Each team has to enforce a mandatory rotation policy so that no player ever plays three matches in a row during the season.

Nikhil is the captain and chooses the team for each match. He wants to allocate a playing schedule for himself to maximize his earnings through match fees during the season.

Input format

Line 1: A single integer *N*, the number of games in the IPL season.

Line 2: *N* non-negative integers, where the integer in position *i* represents the fee for match *i*.

Output format

The output consists of a single non-negative integer, the maximum amount of money that Nikhil can earn during this IPL season.

Sample Input 1

```
5
10 3 5 7 3
```

Sample Output 1

```
23
(Explanation: 10+3+7+3)
```

Sample Input 2

```
8
3 2 3 2 3 5 1 3
```

Sample Output 2

```
17
(Explanation: 3+3+3+5+3)
```

Test data

There is only one subtask worth 100 marks. In all inputs:

- $1 \le N \le 2 \times 10^5$
- The fee for each match is between 0 and 10^4, inclusive.

Identifying the sections of a problem

First and foremost, it is essential to identify crucial sections of a problem task before analyzing any algorithms or writing code. In this problem titled *IPL,* a certain number of games (N) are given according to the user's input. Afterward, N number of games is expected from the user with each containing a particular fee awarded if played in the game. The problem calls for the maximum amount of money that could be earned for an individual (Nikhil) through the N games. However, there is a condition in which an individual is obligated to play at most 2 consecutive games. Herewith, a certain algorithm must be constructed which maximizes salary while taking at most 2 consecutive games condition into account.

Furthermore, a prediction about the complexity of a problem could be made per the given test data constraints. A maximum of $2*10^5$ number of games along with 10^4 for each game can be inputted by the user. An average computer is able to make approximately $2*10^6$ calculations per second regardless of the operating system. Thus, a brute force algorithm with a time complexity of

$O(n^2)$ would be sufficient for a solution to pass this problem. The time complexity of $O(n^2)$ signifies the worst solution possible for a problem which usually persists in iterating through every single value until the correct answer is reached. Nonetheless, a brute force solution wouldn't pass for this problem if the user could input per se $2*10^8$ number of games. In a hypothetical case, a genetic algorithm would be one of the most ideal solution methods for this problem with a time complexity of O(gnm). Among the time complexity of O(gnm), "g" stands for the number of generations, "n" stands for the population size, whereas, "m" signifies the size of individuals. In brief, although in this particular case it's not imperative, a genetic algorithm solution would be a fast and viable solution to the problem among random search and brute force algorithms.

Genetic Algorithm Implementation

The genetic algorithm is implemented via the programming language C# which is a general-purpose, object-oriented programming language developed by Microsoft. The following section breaks down the code into several sections - namely initialization, selection, crossover, mutation, and termination.

Before the function of the code is analyzed, its inner structure must be understood. First of all, the code starts with a class named *Individual*, which represents one individual in the entirety of the population. Each contains a set of genes that are of template type, a double fitness value, and a chromosome length which is common throughout the entire population. Its implementation is as follows:

```
class Individual<T> {
    public T[] genes; //instance variables
    private int chromosomeLength;
    public double fitness;
    ...
```

All of these instance variables are initialized with a constructor as each instance of the *Individual* class is created. Another structure within the algorithm is the *Population* class. The class contains an array of individuals, a generation number, a population size, a chromosome size, and various probability values for the genetic algorithm to work on. The implementation of its instance variables is as follows:

```
class Population<T> {
    public Individual<T>[] individuals;
    public double generationNumber;
    public Individual<T> fittestMember;
    private int populationSize;
    private int chromosomeSize;
    private double mutationProbability;
    private Func<T> GetRandomGene;
    private Func<T[], double> CalculateFitness;
    private double elitismPercent;
    private double selectPercent;
    ...
```

As evident from the code snippet, the population class also contains two functions: GetRandomGene() and CalculateFitness() function. These functions are provided by the client of these classes and are called within the population's various functions. GetRandomGene() is the function that returns a random gene to initialize the members. CalculateFitness() is a function that returns a double value according to the genes of the individual; the number returned signifies the fitness of that particular gene set or individual. As cognizant of solely the meaning and implementation of these values, it is possible to analyze the code entirely. The first step in the code is the initialization of the individuals. The initialization method iterates through the array of individuals and sets the gene of each individual using the GetRandomGene() method. For the particular problem, the genes of the members are an array of integers. All values in the array are either 1 or 0. The gene sequence represents the sequence of games and each 1 in the gene represents that a match is going to be played while each 0 represents that the match will be passed.

After the initialization of the population is complete, the process of selection begins. Before the selection can order and remove *individuals* from the population, the fitness of each individual has to be calculated. The CalculateAllFitness() function located inside the Population class iterates through all of the population's individuals once more and calls the CalculateFitness() method on their gene sequence and sets their fitness value to be the return value of the method. The CalculateAllFitness() method is implemented as such:

```
public void CalculateAllFitness() {
    foreach (var individual in individuals) {
        individual.fitness = CalculateFitness(individual.genes);
    }
}
```

The CalculateFitness() function is provided by the client of the *Population* class. For this particular problem the fitness is calculated to be the highest fee that can be reached within the gene sequence. A fee is calculated from a series of matches until 3 consecutive matches are played which is derived by the condition of the problem. If 3 consecutive matches are played, the fitness value is reset to 0 and the value it contains before resetting is compared to the *biggestFitness* value. If the fitness calculated from this particular series of matches within the gene sequence is bigger than the *biggestFitness* value, the *biggestFitness* value is set to be the fitness of this series. This process aims to maximize the fee earned from the gene sequence as it attempts to maximize for the largest value of fee possible while also keeping the amount of three consecutive matches played to the lowest possible value.

After all of the fitnesses are calculated the selection takes place. The selection function orders each Individual according to their fitness and a percent of the population whose fitness is lower is removed from the population. The percentage which the population is selected is given as a parameter to the method (selectPercent). The selection method is also a part of the Population class, and it is implemented in the following code snippet:

```
public void Selection(double selectPercent) {
    Individual<T>[] SortedList = individuals.OrderByDescending
                                    (o => o.fitness).ToArray();
    fittestMember = SortedList[0]; /* Fittest member will later be used to
                               collect data about the progress of the algorithm */
```

```
    individuals = SortedList.Take
                ((int)(populationSize * selectPercent)).ToArray();
}
```

After selection is completed, a new generation is created. Another array of individuals is initialized with all of the individuals being empty. Then a percentage of the previous generation is passed onto the next generation starting from highest fitness. This is called elitism. The remaining empty places in the population are filled using the offsprings of the individuals of the previous generation. To produce offspring two parents are selected using weighted random selection which uses the individuals' fitness as weight. After the selection is made, crossover and mutation is applied. The two offspring are placed into the new generation and this process is repeated until the new generation becomes just as populated as the previous generation was. The code for making a new generation is implemented as follows:

```
public void MakeNewGeneration() {
    List<Individual<T>> currentGeneration = individuals.ToList();
    List<Individual<T>> newGeneration = new
                              List<Individual<T>>(populationSize);
    newGeneration = individuals.Take
              ((int)(elitismPercent * populationSize)).ToList(); /* Elitism is
  applied, a percent of the highest fitness individuals are passed onto the next
                                                                    generation */
    Individual<T> parent1, parent2;
    double totalFitness = 0;
    foreach (var ind in currentGeneration){
        totalFitness += ind.fitness; /* Total fitness is calculated to weight
                                                    selection of new parents */
    }
    while(newGeneration.Count < populationSize) {
        parent1 = GetRandomByFitness(totalFitness); /* Weighted random selection
          is applied; higher the fitness, higher the chance of being selected */
        parent2 = GetRandomByFitness(totalFitness);
        var children = parent1.CrossoverReproduction(parent2);
        foreach (var child in children) {
            child.Mutate(mutationProbability, GetRandomGene);
            if (newGeneration.Count < populationSize)
                newGeneration.Add(child);
        }
    }
    individuals = newGeneration.ToArray();
}
```

All of these functions are put together in a PropagateGeneration method to simplify the entire process. PropagateGeneration is defined in the following way:

```
public void PropagateGeneration() {
    CalculateAllFitness();
    Selection(selectPercent);
    MakeNewGeneration();
}
```

The last step is the termination. For this particular problem the termination condition is the number of generations that the algorithm ran for. In the tests made, every algorithm was run for 2000 generations before being terminated. The fitness levels were not adjusted or altered and were taken as is from when the algorithm stopped.

Evaluation of the Solution

A total of three runs were made. Each algorithm is left to run for 2000 generations and with a population of 10000 individuals. Runs were made using N values of 500, 1000, and 2000. In these tests, fitness also represents the proposed solution of a gene sequence which in this case is the highest fee an individual can earn. This works as both the fitness and the solution to the problem are trying to be maximized. In each generation 50 percent of the low fitness population is removed and others are allowed to produce offspring. Mutation chance is 80%. This value can alter with the rate of convergence and prevent it dramatically but it is required to be extremely high to cover the vast search space. Elitism percent is 5% meaning in each generation highest fitness 5% of the population is passed onto the next generation without being subjected to any alteration. The processed data from the runs are entered into *Table 1.* The best fitness for each generation is plotted as a function of generation count and *Graphs 1, 2,* and *3* are created.

Table 1, ZCO 2014 Problem 4 IPL Test I

Solution Qualities	**TEST I**	**TEST II**	**TEST III**
Generation Count:	2000	2000	2000
Population Size:	10000	10000	10000
Inputs	N = 500	N = 1000	N = 2000
Best Fitness	1920717	3794093	7243546
Actual Solution	1950445	3874387	7614676
Percentage Difference	1.5%	2.1%	4.8%
Status	Solved	Solved	Solved

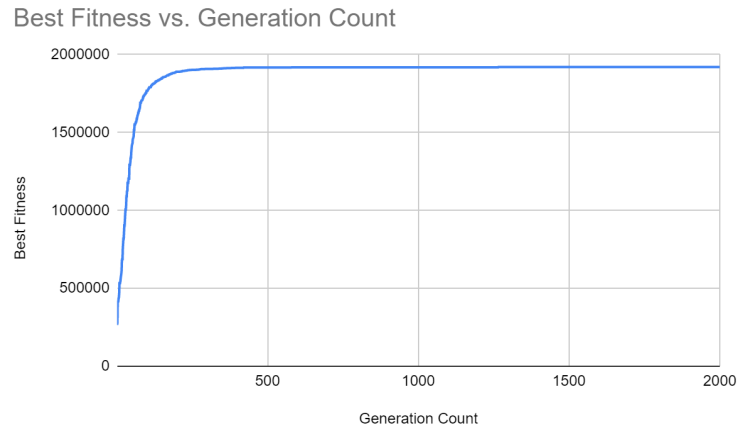
The data suggests that for the tests done, the algorithm is successful in finding an optimal solution to the problem. It should be noted that for the third test the percentage difference is very close to the threshold value. This increase in the percentage difference is caused by inadequate parameters in the genetic algorithm, most notably the generation count and population size. Given a higher generation count, it is expected that the genetic algorithm converges closer to the value, resulting in a lower percentage difference. Another test with the same parameters except for generation count as test 3 is done to show this phenomenon. Note that the actual solution is different from test 3 as all the input values are randomized.

Table 1, ZCO 2014 Problem 4 IPL Test II

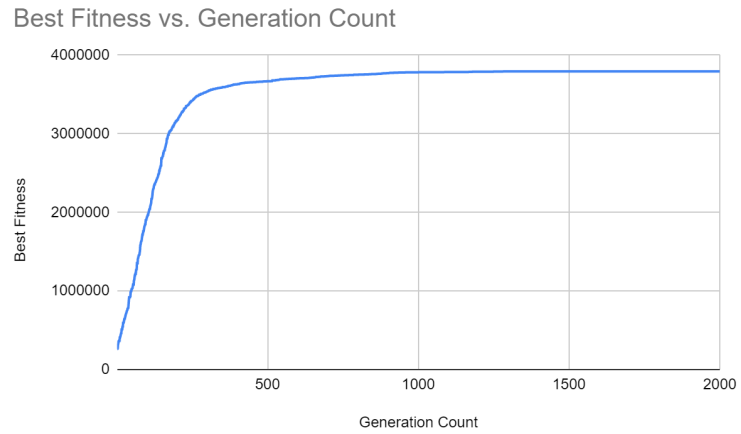
	Generation Count	**Best Fitness**	**Actual solution**	**Percentage Difference**
TEST III (Revised)	4000	7440479	7615239	2.3%

As can be observed from this test, the percentage difference drops significantly when a larger generation count is used. Considering all of these tests, it could be concluded that genetic algorithms are adequate in solving such problems, but require a lot of time and memory to run efficiently. Along with this, as the amount of input values gets bigger, the amount of generation required to converge on an optimal solution increases. This fact must then be taken into consideration when designing problem-specific genetic algorithms. Since the time and memory are usually restrained on olympiad questions, an efficient algorithm must be implemented with the best parameters possible.

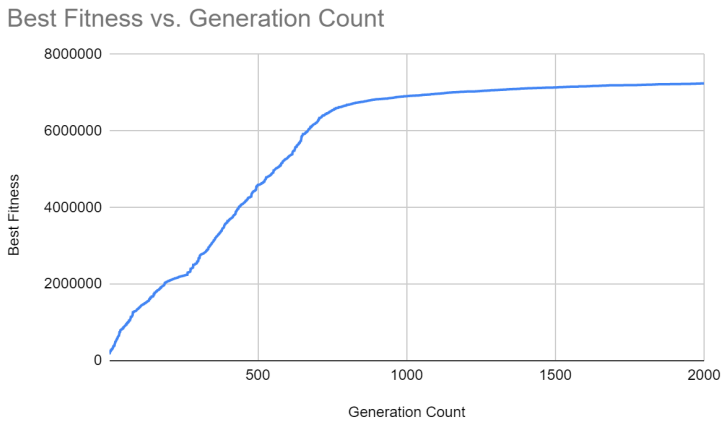
Graph 1, Best Fitness vs Generation Count Graph for Test I



Graph 2, Best Fitness vs Generation Count Graph for Test II



Graph 3, Best Fitness vs Generation Count Graph for Test III



Conclusion

Thanks to the nature of genetic algorithms, competitive programmers and programming enthusiasts have the upper hand in solving optimization problems. Although genetic algorithms are able to provide optimal solutions for a majority of optimization problems, a couple of drawbacks are visible from the solution of IPL. Firstly, since genetic algorithms require a set of generations to be processed in order to reach the final accurate answer, time and memory constraints are generally exceeded. For instance, in the IPL example, the time limit is stated as 1 second, whereas the memory limit is stated as 32 Megabytes (MB). The solution that is at hand utilizing genetic algorithms requires at least 370 seconds and encompasses 200 MB of memory. Thus, certain implications to the program should be made if genetic algorithms ought to be used in a competitive environment. A suggestion for an implication would be to stop new generation formation when an accurate solution is reached. Genetic algorithm provides a comprehensive search methodology for optimization. The problem of finding the accurate solution in a space with many sub-optimal solutions is a classic problem for all systems that can adapt and learn.

In conclusion, there have been a couple of benefits and drawbacks emerging as genetic algorithms were utilized in this study in order to solve optimization problems from informatics olympiads. It is observed that genetic algorithms are generally able to solve such problems but are unable to comply with the memory and time restrictions. It should also be noted that converging on higher input values, requires more number of generations to be run on the algorithm, or an increased population size. Consequently, a competitor in such informatics olympiads must consider the possibility of using genetic algorithms as a tool for solving the problems yet must also approach the algorithm with caution as the implementation has to be optimized heavily in order to comply with time and memory constraints of the olympiads.

References

[1] Klein, A. More than half of high schools now offer computer science, but inequities persist.https://www.edweek.org/teaching-learning/more-than-half-of-high-schools-now-offer-computer-science-but-inequities-persist/2021/11 (accessed Jan 7, 2022).

[2] IOI Statistics. https://stats.ioinformatics.org/olympiads/2019 (accessed Jan 10, 2022).

[3] Techopedia. What is a genetic algorithm? - definition from Techopedia https://www.techopedia.com/definition/17137/genetic-algorithm (accessed Jan 10, 2022).

[4] Sablier. Coding games and programming challenges to code better https://www.codingame.com/playgrounds/334/genetic-algorithms/history (accessed Jan 30, 2022).

[5] Dynamic Programming https://www.geeksforgeeks.org/dynamic-programming/ (accessed Jan 30, 2022).

[6] Worden, K.; Liu, S.; Das, D.; Avi Ceder, A. Genetic Algorithm https://www.sciencedirect.com/topics/engineering/genetic-algorithm (accessed Jan 11, 2022).

[7] Lecture12-dukecomputerscience. https://www.cs.duke.edu/courses/fall05/cps130/lectures/skiena.lectures/lecture8.pdf (accessed Jan 12, 2022).

[8] Continuous optimization. https://uwaterloo.ca/combinatorics-and-optimization/research-combinatorics-and-optimization/research-areas/continuous-optimization (accessed Jan 12, 2022).

[9] Time complexity of genetic algorithms exponentially ... https://fernandolobo.info/p/gecco00.pdf (accessed Jan 13, 2022). Agnishom. ZCOSolutions/ipl.cpp at master · Agnishom/Zcosolutions.

[10] Indian Computing Olympiad Archives 2012 - IARCS. https://www.iarcs.org.in/inoi/2014/zco2014/zco2014-2b.php (accessed Jan 17, 2022).

[11] Bajpai, P.; Kumar, M. Genetic Algorithm - an Approach to Solve Global Optimization Problems https://www.researchgate.net/ (accessed Jan 20, 2022).