Genetic Algorithms to solve a Scheduling Problem

[*]Solving class scheduling problems using Genetic Algorithms in Node.js (Activity 2d)

Ângelo Teixeira
FEUP

Porto, Portugal
up201606516@fe.up.pt

***Abstract*—The aim of this work is to attempt to solve a scheduling problem with use of genetic algorithms and analyze the results regarding performance metrics, namely by quantifying (execution time) and qualifying them (gives the optimal solution). This case study consists of assigning classes to existing time slots, so that there are few collisions as possible. A collision happens when a student is registered in two classes on the same slot (at the same time).**

***Index Terms*—Scheduling Problems, Artificial Intelligence, Genetic Algorithms**

I. Introduction

Genetic Algorithms are being increasingly used, due to the ease with which they find solutions to complex problems, even though they can be less efficient, when compared to specialized, and thus optimized, algorithms. This article describes the work done in order to solve a scheduling problem, using genetic algorithms, with varying parameters, such as *population size* and *mutation rate*. The performance of the executions with different parameters was measured and is further analyzed in the end of this article. The implementation of the algorithms, as well as the more thorough problem description is presented in the following chapters.

II. Problem Description

The presented problem consisted of assigning a set of classes to a set of time slots. Each class has students registered to it. When a student is enrolled in more than one class in the same time slot, it is considered to be a collision. The goal is to minimize the collisions. A solution is considered optimal if there are no collisions.

III. Implementation Details

In order to better manipulate the data and apply the genetic algorithm operations (selection, crossover and mutation) the following formulation was chosen:

A. DNA representation

The **DNA** class, which is responsible for storing a "solution", which consists of an array where each element represents the slot for the course identified by its index.

For example, a solution [1,1,1,2,2,2,3,3,3,4,4,4] means that courses 0, 1 and 2 are on the first slot, the courses 3, 4 and 5 are on the second slot, the courses 6, 7 and 8 are on the third slot, and courses 9, 10 and 11 are on the last slot.

It has the capability of calculating its fitness, based on how few collisions there are in the solution.

B. Population

The **Population** class is responsible for the genetic algorithm lifecycle.

It contains methods to generate a random initial population (set of N different DNA elements), to select the ones to reproduce for the next generation, the generation of the next generation itself, using crossover of selected parents and mutation of the the children, and finally, a method to evaluate the current generation, used to terminate the population, when a solution is found.

In order to return always the best solution found, the population keeps the best solution so far stored, based on its fitness, so that, if the optimal solution is not found in the generation limit, the best so far is returned.

C. Fitness Function

The fitness of a solution must reflect how good that solution is. With that in mind, the used approach was to minimize the collisions of a student having two classes at the same time. The used formula is:

$$F = N/(C + N) \quad (1)$$

being F the fitness, N the number of students in the problem, and C the amount of collisions.

This allows for a value in [0,1] range. The fitness value of 1 (perfect solution) only happens when the amount of collisions is 0. However, for a same number of students, a solution with less collisions is better.

D. DNA Selection

The selection of the *parents* that lead to a new generation is based on a probability that increases with their fitness. This method allows for a more "natural selection" which selects an individual with a probability proportional to its relative fitness. For a population of individuals with 0.5, 0.2, 0.2, 0.1 and 0 fitness values, the probability of being selected is 50%, 20%, 20%, 10% and 0%, respectively.

E. *DNA Crossover*

When two *parents* are selected, their DNA is mixed to create a new one, called the *child*. To do this, the first half of a *parent* is joined with the second half of the other *parent*

F. *DNA Mutation*

In order to maintain the variation in the population, its often necessary to mutate some of the values of some solutions. So, with a defined probability, some genes of a *children* are randomly changed, allowing for variation of the solutions themselves.

IV. Experiments and Results

To analyze the results, the following variables were considered:

- Elapsed Time (ms)
- Fitness of the best solution (1 means the optimal was found)
- Number of Generations (Max: 1000)
- Mutation Rate
- Population Size

The following charts show bubbles representing the data. The larger the bubble, the more generations it took. Different colors mean different fitness values, and the bubble position refers to its value related to the chart's axis.

On the first experiment (see Fig. 1), the population size was changed, with a constant mutation rate, to see how the result were affected by having a larger population, or a smaller one. It's possible to note that, by having a smaller population, the execution time is faster, however, the results are not that great when compared to larger populations. This has to do with the variation in the population. A larger population will probably have the most different individuals and the combinations can produce better results, avoiding local maxima.

As we can see, the larger the population gets, the better is the fitness, and for larger values, it often reaches the optimal solutions (green bubbles), with a smaller number of generations. That, on the other hand, brings a cost in execution time, that seems to grow exponentially. In the end, it is all about a tradeoff, but one thing is certain: to get good results, a small population will not serve.

In the next experiments, the varying parameter was the mutation rate, for fixed population sizes.

In Fig. 2, the mutation rate was changed for a small population size, to see if had enough impact on the optimal solution finding. It had not. The algorithm was trapped in a local maximum (0.923076923076923 fitness value) and could not mutate enough even with a 100% chance because the population was so small.

As we increase the initial population size, better results appear. In Fig. 3, we can see an optimal solution for a mutation rate of 10% and 90%. It is also noticeable that for higher mutation rates, the best solution found is better, due to the higher variation inserted in the population.

With a population of 500, in Fig. 4, the results are not much different. The difference being that the overall fitness values are higher, due to the higher initial variation of the population (more individuals cause more variation)

When we have a really high number of individuals, though, in the thousands, the results look much more promising, as the optimal solution is found regardless of the mutation rate (Fig. 5)

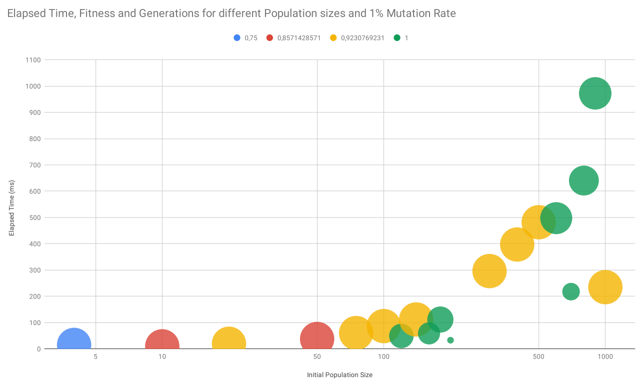

Fig. 1. Elapsed Time, Fitness and Generations for different Population sizes and 1% Mutation Rate

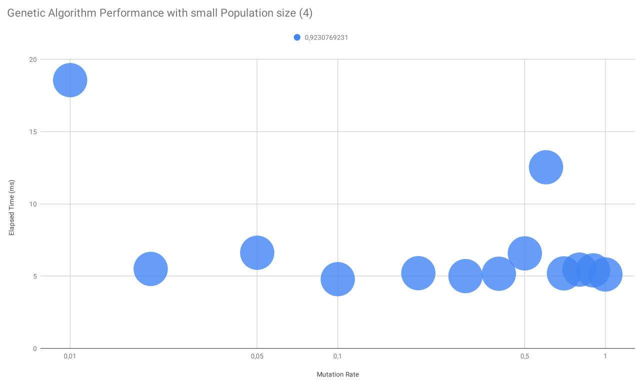

Fig. 2. Mutation Rate, Time, Fitness and Generations for an Initial Population of 4

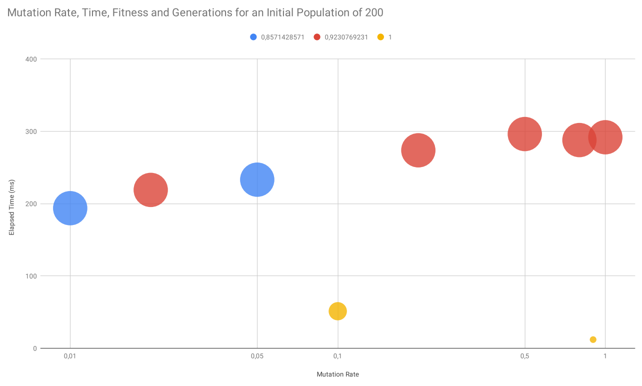

Fig. 3. Mutation Rate, Time, Fitness and Generations for an Initial Population of 200

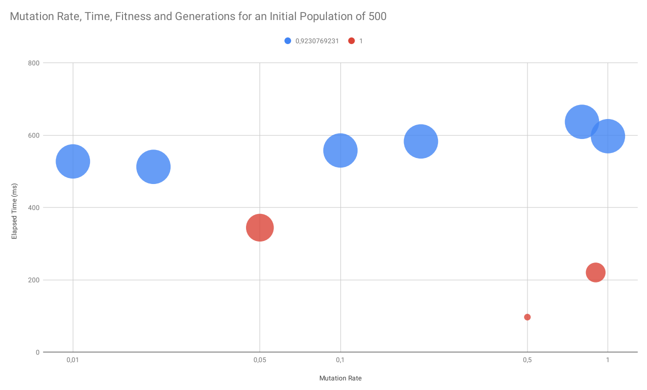

Fig. 4. Mutation Rate, Time, Fitness and Generations for an Initial Population of 500

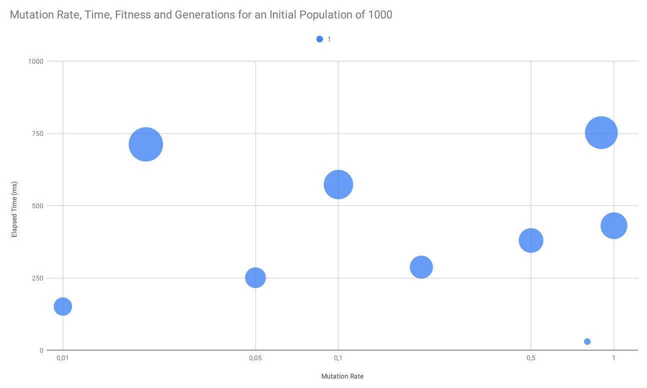

Fig. 5. Mutation Rate, Time, Fitness and Generations for an Initial Population of 1000

One thing can also be noted: The time complexity gained by the increase on the mutation rate is not as near as steep as the increase caused by the population size, while giving a small increase in result quality.

V. Conclusions

With this project, I was able to experiment the problem solving capabilities of a genetic algorithm, and tinker with its parameters. I found out that genetic algorithms are a great way of solving complex problems without much *man-work* - for this whole project I must have coded no more than 2-3 hours - and I was able to play with various aspects of the algorithm, regarding the way that reproduction or mutation worked, allowing for a lot of creativity.

In the context of Artificial Intelligence, there is certainly a place for Genetic Algorithms, as they provide a unique and natural way of finding a solution, that can be applied to various problems, not only path-finding or scheduling, and be used together with a Neural Network to evolve the algorithm itself, forming a strong solution to hard problems.

In a more technical aspect, what I learned is that, in order to have the optimal solution, one must have a big population size, with a lot of variation. That will, however, affect the execution time, and for bigger problems and/or inputs, that can be a bottleneck.