Algorithms in Computational Biology

Genetic Algorithms

Or Amar – 311166169

January 2021

Introduction

Genetic algorithms (GA) are a family of computational models inspired by evolution. Genetic algorithms are commonly used in solving optimizations and search problems, using methods and assumptions from the field of biology such as mutation, selection and crossovers.

GAs were first described by John Holland in the 1960s and further developed by Holland, his students, and colleagues in the 1960s and 1970s. Holland wanted to develop ways in which the mechanisms of natural adaptation and selection might be imported into the field of computer science. Later on, in his book "Adaptation in Natural and Artificial Systems" from 1975, Holland presented the GA as an abstract form of evolution and set the framework for adaptation under GAs.

Setting the framework

I will now describe shortly the terms used in GA's and their meaning. In order to do so, let us imagine we are facing an optimization problem and we wish that our algorithm evolve to have better results. To have an evolutionary process, we need an initial population and some change from one generation to the next. In GAs the "population" consists of candidate solutions, sometimes referred as individuals or phenotypes. Each individual has a set of properties (sometimes referred as chromosomes or genotype) which are exposed to mutation and alternation processes. Usually, the evolutions starts from a population of randomly selected individuals, then each iteration of the algorithm represents the current population which is the "generation". At every generation we evaluate the individuals by looking at the values they emit in the objective function our problem. This value is sometimes referred as the "fitness" of an individual. The fittest individuals are stochastically selected from the current generation and the "genome" of those individuals is modified through processes of mutation and recombination.

We can summarize the terms in the following table:

1	Natural Evolution	Genetic Algorithm
2	Genotype	Coded String
3	Phenotype	Uncoded Point
4	Chromosome	String
5	Gene	String Position
6	Allele	Value at a Certain Position
7	Fitness	Objective Function Value

Table 1 - GA terminology

The canonical genetic algorithm

As mentioned before, the first step in the implementation of any genetic algorithm is to generate an initial population, let us denote the population size with N. In the canonical genetic algorithm, each individual of this population will be represented with a binary string of length L which corresponds to the problem encoding. This string is the genotype/chromosome. In most cases the initial population is generated randomly. Now that we have the initial population, each string of individual is then evaluated and assigned a fitness value. It is useful to distinguish between the evaluation function (the objective function) and the fitness function used by a genetic algorithm. The objective function provides a measure of performance with respect to a particular set of parameters and is independent of the evaluation of any other string. The fitness function on the other hand is always defined with respect to other members of the current population and the fitness function transforms that measure of performance of the objective function into an allocation of reproductive opportunities.

In the canonical GA we normalize the fitness as follows: for every individual i in the population, let f_i be the evaluation by the objective function of the $i'th$ string. Let $\bar{f}$ be the average evaluation of all the individuals at the current population. Therefore, the fitness of the $i'th$ individual is: $F_i := \frac{f_i}{\bar{f}}$. There are other ways to assign the fitness of individuals in the population, but we will not discuss them here.

<u>The canonical GA process</u>

We can view the execution of the GA as a two-stage process. Starting with current population, the first stage is when selection is applied to the current population and as a result, we create a new intermediate population. The next stage is applying recombination and mutation to create the next generation. On the next iteration in the GA, we will use the new population which was formed as our current population. In Fig.1 we can see a sketch of an entire iteration.

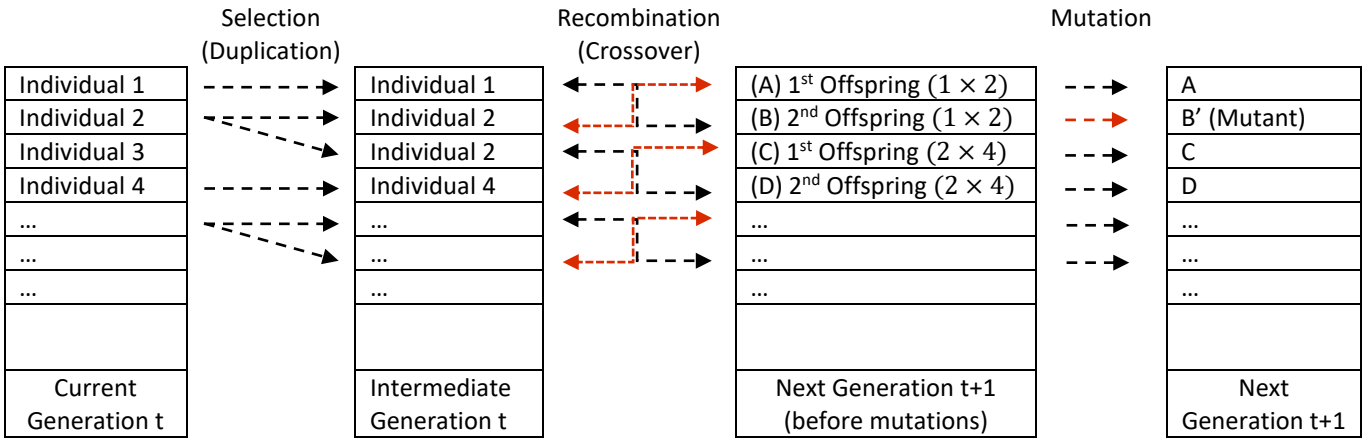

Figure 1 – Process of the canonical algorithm - one generation is broken down into a selection phase, a recombination phase and a mutation phase. This is an illustrated example because strings are assigned to slot randomly

We will first consider the construction of the intermediate population from the current population. After calculating F_i for all the individuals in the current population, selection is carried out. In the canonical GA, the fitness of individuals affects the probability that individuals will continue to the intermediate

generation (The higher the fitness it is more likely to be pulled to the intermediate generation). There are a number of ways to do selection. We will explain one selection process in which we use the remainder stochastic sampling. For each individual i, we use the calculated value F_i. When $F_i = x > 1$ we will directly place $\lfloor x \rfloor$ copies in the intermediate population and the remainder ($x - \lfloor x \rfloor$) will serve us as the probability to place another copy in the intermediate population. Meaning, after directly placing the integer value of each individual in the intermediate population, all the individuals (including those with $F_i < 1.0$) place additional copies in the intermediate population with a probability corresponding to the fractional portion of F_i. Remainder stochastic sampling is most efficiently implemented using a method known as stochastic universal sampling. Assume that the population is laid out in random order as in a pie graph, where each individual is assigned space on the pie graph in proportion to fitness. Next an outer roulette wheel is placed around the pie with N equally spaced pointers. A single spin of the roulette wheel will now simultaneously pick all N members of the intermediate population. This method is also resulting an unbiased selection.

After selection has been carried out the construction of the intermediate population is complete, and recombination and mutation can occur. We pair the individuals (represented in strings) and a crossover is applied to randomly paired strings with a probability denoted P_c. With probability P_c 'recombine' the pair of strings to form two new strings that are inserted into the next population (notice that in some scenarios the new strings might be similar to their “parents” in the intermediate population).

After recombination, we can apply a mutation operator. For each bit in the population, mutate with some low probability P_m. There are several ways to apply mutation and we will not cover them in this paper, but a mutation is a change of a bit in a string corresponding to individual.

After the process of selection, recombination and mutation is complete, the next population can be evaluated. A GA is typically iterated for around 50 to 500 or more generations or until a criterion is matched. The entire process is often called a “run.” At the end of a run, there are often one or more highly fit chromosomes in the population. Since randomness plays a large role in each run, two runs will probably produce different detailed behavior. Therefore, GA researchers often report statistics (such as the best fitness found and generation at which best fitness was found) averaged over many different runs of the GA on the same problem.

Pseudo code

```
generate initial population
compute fitness of each individual

WHILE True:
    /* produce new generation */
    FOR N/2:
        select two individuals from old generation for mating
        /* fitted individuals are more likely to be picked */
        recombine the two individuals to give two offspring
        add mutations /* by mutation factor */
```

compute fitness of the two offspring
insert offspring in new generation

IF population has converged OR threshold reached: /* *depends on implementation* */
break

return fittest individual / average fitness of last generation / string of individual /* *depends on problem* */

Illustrated example GA – mouse in a maze problem (search problem)

Consider the mouse in a maze problem illustrated in Figure 1. The mouse needs to find a sequence of no more than x steps that will move the mouse from the entrance to the cheese. As shown in the figure, each move (Forward, Backward, Left, or Right) is represented by two bits, so a sequence of x moves can be represented by a bit string of length $2x$. Now we can initialize a population of N possible strings (individuals) of size $2x$. The fitness of a specific sequence can be calculated by letting the mouse follow the sequence, and then measuring the number of steps between its final position and the cheese (the exit) where the smaller the distance, the higher the fitness of the sequence. Maximum fitness is attained if the mouse reaches the cheese in x steps or less.

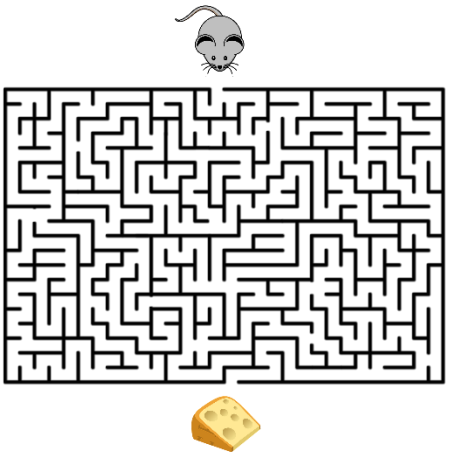

Figure 2 - Mouse in a maze problem in which one candidate solution is a bit string encoding for a series of "forward," "backward," "left," and "right" moves. WLOG, 00 – forward, 01 – backward, 10 – left, 11 – right.

Why does it work?

At a very general level of description, GAs are converging towards good building blocks of solutions in a parallel fashion. The idea is that good solutions are often made from good building blocks. To formalize the idea of building blocks came the notion of schemas. A schema is a set of bit strings made of '0's '1's and '*'s (a wild card). A schema is said to be of order y if the schema has y defined bits ('0's or '1' but no '*'). Schema can denote both subset of strings represented by a template or the template itself, and the meaning should be clear from context. Not every possible subset of length$-L$ bit strings can be described as a schema. There are 2^L possible bit strings of length L and thus 2^{2^L} possible subsets of strings, however there are only 3^L possible schemas. However, that the central tenet of traditional GA theory is that schemas are the building blocks which the GA processes effectively. Let there be a bit string of size L so that string is an instance of 2^L unique schemas. For example, the string 11 is an

instance of **, 1*, *1 and 11 schemas. For the general case, let there be N strings of size L, they are instances of between 2^L (all the strings are identical) and $N \times 2^L$ (all the strings are unique) different schemas. The meaning of this is that in evaluating a population of N strings, the GA is implicitly estimating the average fitnesses of all schemas that are present in the population which known as implicit parallelism. The guiding principle is that the degree of parallelism does not depend on the population size, but on the much larger schemas whose instances are present. The effect of selection is to gradually bias the sampling procedure toward instances of schemas with higher average fitness.

In order to calculate the dynamics of this sample biasing, we will act as follows. Let s be a schema with at least 1 instant present in the population at time (or iteration) t. let $N(s,t)$ be the number of instances of s at time t and let $\bar{u}(s,t)$ be the observed average fitness of the instances of s at time t. Now we wish to compute the expected number of instances of s at the next generation, denoted with $\mathbb{E}\big(N(s,t+1)\big)$. let F_i be as mentioned in "The canonical genetic algorithm" in page 2 and for simplicity we will now ignore the effects of crossovers and mutations. Let $i \in s$ (individual i is an instance of s) then:

$$\mathbb{E}\big(N(s,t+1)\big) = \sum_{i \in s} F_i = \frac{\bar{u}(s,t)}{\bar{f}(t)} \cdot N(s,t)$$

Equation 1

Adding crossover and mutation, the right side of Eq. 1 can be modified to give lower bound on the expected value. At a single-point crossover event we will say that schema s "survived" under the event if one of the offspring is also an instance of schema s. Let us denote the surviving of s under a crossover event with $S_c(s)$ so we can bound $S_c(s)$ as follows:

$$S_c(s) \geq 1 - P_c\left(\frac{d(s)}{L-1}\right),$$

where P_c mentioned under "The canonical genetic algorithm" in page 3 and $d(s)$ is the distance between the outermost defined bits in the schema s (also known as the defining length). Crossovers occurring within the defining length can destroy s and therefore we get the bounded equation above. In other words, the probability of survival under crossover is higher for shorter schemas. We can do a similar thing to calculate the effects of mutation. Let us set P_m to be as mentioned in page 3, then we will denote the probability that schema s will survive under mutation with $S_m(s)$. Therefore:

$$S_m(s) = (1 - P_m)^{o(s)},$$

Where $o(s)$ is the order of s as mentioned before, which means that for schema s to survive mutation it needs to survive (not change) its first defined bit ($with\ probability\ of\ 1 - P_m$) and its second defined bit and so on, $o(s)$ times.

Now we can apply these effects to Eq. 1 and get:

$$\mathbb{E}\big(N(s,t+1)\big) \geq \frac{\bar{u}(s,t)}{\bar{f}(t)} \cdot N(s,t) \cdot \left(1 - P_c \cdot \left(\frac{d(s)}{L-1}\right)\right) \cdot \big((1 - P_m)^{o(s)}\big)$$

Equation 2 - The Schema Theorem

Eq. 2 can be interpreted to say that short, low-order schemas, with relatively high fitness will receive exponentially increasing number of samples (representatives) over time because their schemas have lower chances of disrupting effects by the crossover and mutation and their fitness increases by a factor of $\frac{\overline{u}(s,t)}{\bar{f}(t)}$ at each generation.

Notice that this is a lower bound, albeit some crossover events and mutation might increase the fitness of an individual, however in this scenario the schema s might not survive (and the schema of the offspring s' will have a better fitness). Mutation serves a great deal with not losing genetic diversity as it forces toward genetic drift-mutation equilibrium and preventing fixation.

Applications of Genetic Algorithms

While the aforementioned algorithm is rather simple, there a many variations on this basic algorithm. Here are some applications of GAs which can be used in problems and models from scientific and engineering fields:

- Optimization: numerical and combinatorial optimization problems such as circuit layout and job-shop scheduling.
- Automatic programming: evolving computer programs for specific tasks, and designing computational structures, such as cellular automata and sorting networks.
- ML: classification and prediction tasks such as the prediction of weather or protein structure. GAs can also evolve aspects of particular machine-learning systems, such as weights for neural networks or rules for learning classifier systems.
- Ecological models: modeling ecological phenomena such as biological arms races, host-parasite co-evolution, symbiosis, and resource flow in ecologies
- Immune system models: modeling aspects of the natural immune system including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
- Economic models: modeling processes of innovation, development of bidding strategies, and emergence of economic markets.

Criticism

Genetic algorithms are surely elegant and easy to comprehend but there are several criticisms of the use of GAs comparing to other optimization algorithms:

- The evaluation of the fitness function in complex problems may be very expensive and this process occurs repeatedly in GAs. For some real-world problems, a single evaluation may require several hours or even days. In these cases, it may be more efficient to make an exact evaluation and use an approximated fitness.
- GAs struggle with dynamic data sets, as genomes tend to converge in processes of genetic drift and the remaining solutions may no longer be valid for later data. To tackle this issue, several methods have been proposed to increase the genetic diversity, e.g., by increasing the probability of mutation when the solution quality decreases drastically.
- For specific problems, better solutions might be found using other algorithms.

Conclusion

GAs are a method which is adaptive to its environment due to their modifying abilities taken from the field of evolution. In general, genetic algorithms can be found solving complex, real world problems, however, several improvements must be made in order to make GAs more generally applicable.

References

1. Whitley, D. (1994). A genetic algorithm tutorial. Statistics and computing, 4(2), 65-85.
2. Yadav, P. K., & Prajapati, N. L. (2012). An overview of genetic algorithm and modeling. International Journal of Scientific and Research Publications, 2(9), 1-4.
3. Beasley, D., Bull, D. R., & Martin, R. R. (1993). An overview of genetic algorithms: Part 1, fundamentals. University computing, 15(2), 56-69.
4. Beasley, D., Bull, D. R., & Martin, R. R. (1993). An overview of genetic algorithms: Part 2, research topics. University computing, 15(4), 170-181.
5. Mitchell, M. (1995). Genetic algorithms: An overview. Complexity, 1(1), 31-39.