Evolving Mathematical Functions with Genetic Algorithms - A Team Project on Artificial Intelligence

ZHIGUANG XU, CHRIS NEASBITT, JARED SUMMERS, BILL CONKLING
Department of Math and Computer Science
Valdosta State University
Valdosta Georgia, U.S.A.

Abstract - *In contrast to most of the senior level Artificial Intelligence (AI) courses where AI is taught in the jargon of theoretical computer science with crowd of complex matrix algebra and differential equations, I demystify the subjects and demonstrate that most fundamental ideas behind AI like Neural Networks, Genetic Algorithms, and Fuzzy Logic, etc, are wonderfully simple and straightforward. I give students team projects as opportunities to be exposed to how intelligent systems are actually built up and implemented. In this paper, we discuss such a project developed by a group of three students for evolving mathematical functions with Genetic Algorithms (GA) in Java. In the project, GA is used to solve a symbolic optimization problem where sequence of instructions collectively forming a candidate math function is evolved to represent a reasonably complicated target math function. Students were very interested and deeply involved in the development of this project since they actually applied the GA theory in practice. And this is the best way AI can be taught and grow, as I sincerely believe.*

Keywords: Generic Algorithms, Artificial Intelligence, Fitness Evaluation

1 Introduction

Traditionally, Artificial Intelligence (AI) is taught as a senior level theoretical computer science course. Most of the literature on AI is expressed in the jargon of computer science, and crowed with complex matrix manipulations and differential equations [1]. This, of course on one hand, makes the subjects sufficiently respectful, but on the other hand, lacks of the applied opportunities that the students can take to implement the ideas behind those complicated mathematical formulas and solve some real-world problems. My goal of teaching AI is to demystify some popular AI algorithms so that our undergraduate computer science students without hard-core math background can still witness that the AI-based intelligent programs they write themselves are able to solve otherwise-hard-to-solve problems. As a result, students very much attracted in this way are more likely willing to make more efforts to explore AI in depth and show their creativity in contributing to this exciting area in the future.

In this paper, I will discuss a team project three of my students did when they took my CS4820 Artificial Intelligence course in the fall of 2005. In the project, students were asked to use GA to solve a symbolic optimization problem where sequence of instructions collectively forming a candidate math function is evolved to represent a reasonably complicated target math function. The candidate function, which naturally becomes a GA individual/chromosome, is evaluated on a STack-based virtual Machine (STM). The target function is received in the form of human-readable strings and compiled directly into Java bytecode via Java Expressions Library (JEL). The fitness value of each individual/chromosome function, which is what GA tries to optimize, is the difference between the resulting values generated by STM to what is expected per our JEL-parsed target function. Details of the project will be presented in the subsequent sections in this paper. As witnessed by the running outputs, our GA system is able to quickly converge to the desired target, given that the values of various GA parameters are appropriately set up. (Besides the GA project, in the same course, there are also other two AI projects on Artificial Neural Networks and Fuzzy Logic respectively, which I will discuss in other papers.) Students are motivated to create a generalized genetic algorithm simulation that follows the principles of object-oriented design to deliver extensibility and ease of use. In this project, the interface between the AI core and its application (i.e. the evolution of sequence of instructions to a math formula) is clearly established. Therefore, the intact core could in the future potentially solve problems of interest to students such as Hamiltonian Path problems and others in the realms of linear algebra and graph theory. This project comes with a simple Graphical User Interface (GUI). The GUI allows the user to run the program without any knowledge of the command prompt. It offers easy access to change any variable without needing to know any commands or use any command line arguments. For any user not familiar with the command line they can still use the program and see very clear output. The running output data are automatically redirected to *scilab*, a free *matlab*-like scientific software package, and plotted as figures for further performance evaluation.

In the following, I start to present implementation of GA as a general tool. Important GA components like GA operators, GA fitness evaluation process, etc, are explored in depth. Then I will discuss a Stack-based virtual Machine (STM) based on which the candidate functions generated by GA are evaluated. The STM idea is inspired by a similar example in [2]. Java Expressions Library (JEL) will be introduced next as a handy tool that gives the project the ability to accept any string-based

target function that users type in at runtime without the use of any external file or the necessity of restarting the program. Finally, some sample running output figures obtained by executing the project and plotted in scilab are presented and discussed. Apparently, GA is able to successfully converge to some fairly complicated target functions in some reasonable time. We will conclude the paper with our conclusions and future developments. Figure 1 shows an overview of the project.

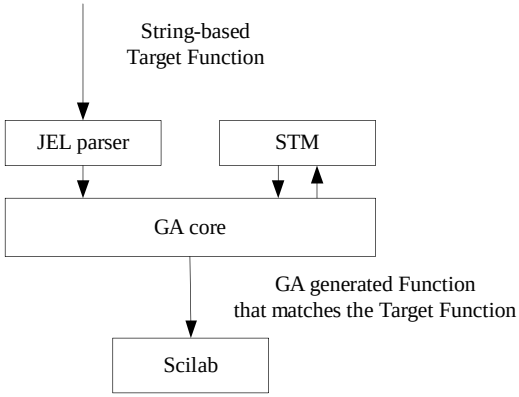

Fig. 1 Overview of the project

2 Implement Genetic Algorithms (GA) as A General Tool

In natural evolution, species search for increasingly beneficial adaptations for survival within their complex environments. The search takes place in the species' chromosomes where changes, and their effects, are graded by the survival and reproduction of the species. Survival of the fittest in nature is the ultimate fitness function [1]. The Genetic Algorithms (GA), developed by John Holland [5] to simulate the natural evolution, is a search algorithm that operates over a population of encoded candidate solutions to solve a given problem. In this project, students are asked to implement in Java a full-fledged GA core as a *general* tool that they can utilize to tackle with various problems. Figure 2 [1] illustrates the flow chart of a basic GA. Some other interesting developments in GA could be found in [3,4,6]

2.1 GA Individuals

First of all, MAX_CHROMS numbers of individuals (or interchangeably called chromosomes) are generated. Each chromosome contains MAX_PROGRAMS genes, where MAX_CHROMS and MAX_PROGRAMS are typically application-dependent and adjustable constants. Each gene, typed as an integer, is initially assigned a value randomly picked from a valid range, which is also application-dependent. Note, for simplicity, in this project, MAX_CHROMS, MAX_PROGRAMS, and valid range of gene values are all defined as constants and we will discuss them later, but in reality, different chromosomes can have different number of genes, and each gene can be initialised from different sets of values.

2.2 GA Termination Criteria

Right after every new generation of chromosomes including the initial one is populated, the fitness value of each chromosome is evaluated based on some application-defined process. We will discuss such a process later in section 3.

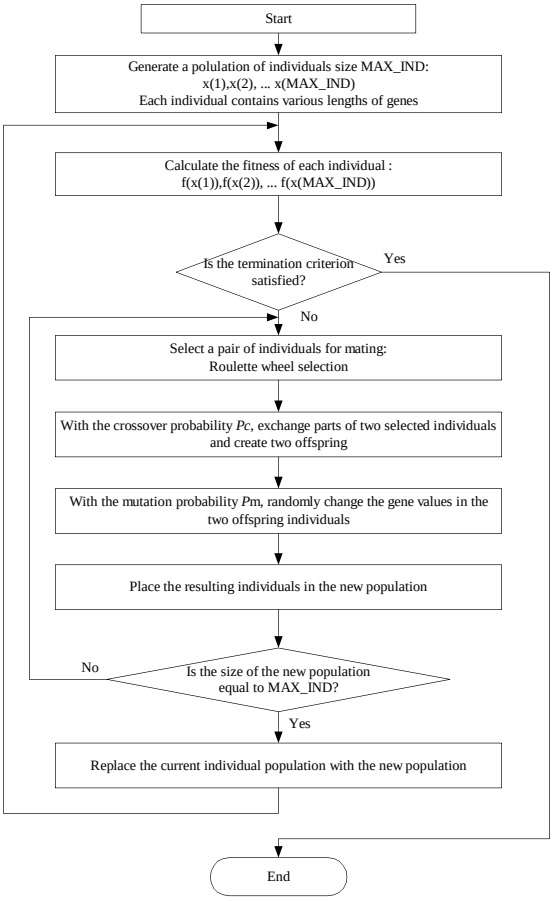

Fig. 2 Flow Chart of a Basic GA

Next, we need to determine if it is time to terminate the program, which is typically governed by a constant MAX_GENERATIONS. When GA has experienced MAX_GENERATIONS number of generations, we quit. But sometimes, it is hard to set up this threshold value as a fixed figure – on one hand, if we set it too large, when testing simple functions, you have to wait a unnecessarily long time until the program stops by itself even though you knew it has converged and solved the problem long time ago; on the other hand, if you set it too small, when testing relatively more complicated functions, you have to take the risk of terminating the GA program too early. A better way is to initialize the constant with a large number first, and adjust it dynamically while the program is running (we call this accelerated learning). As students did in this project, accelerated learning has been implemented by tracking a run which corresponds to times when the system has reached and sustained a peak of maximum fitness value with few interruptions, after a

sufficiently long run the system may terminate prior to the MAX_GENERATIONS.

2.3 Selecting Parents: Roulette Wheel Selection

If neither the condition for accelerated learning is satisfied, nor has the maximum number of generation passed, we need to select a pair of chromosomes as parents for mating from the current population. Parent chromosomes are selected with a probability related to their fitness. The higher the fitness value, the healthier the chromosome is, and the better chance it is selected. The most popular selection algorithm is *Roulette Wheel Selection* (RWS). It operates on the principal that a chromosome's chances of being selected are proportional to that chromosome's fitness compared to the *overall* population. Unfortunately, this classic RWS won't work well for our GA system with such a large number of chromosomes (sometimes 3000+ chromosomes per generation). Therefore in this project, we calculate this probability as a chromosome's fitness divided by the *max* fitness value in the current generation. We then check to ensure that the chromosome under study is at least greater than the minimum fitness of the generation. In other words, we don't select from the least fit of the population (elitist in some manner). We then generate a random number (between 0 and 1) and compare it to our probability value. If the random number is less than the probability value, we select the parent. Otherwise, we continue to the next chromosome. The parent selection process is very important to GA evolution because it makes sure that the next generation is better than its predecessor with respect to the distance to the target and that evolution is going towards the right direction.

2.4 GA Operators

Once two parents are selected with probability related to their fitness values, the following GA operators are applied for reproducing offspring chromosomes.

- We first check to see if we are to perform the *crossover* operation governed by the probability *Pc* (typical valued as 0.7~0.8). If so, we calculate the crossover point by random based on the length of the chromosomes. Then the two parent chromosomes break at the crossover point and exchange the chromosome parts. Mare sure the crossover point is neither the first nor last gene of the chromosome since if so there will be no crossover actually occurring. Note, it is perfectly possible that the offspring chromosomes are just exact clones of their parents if we choose not to crossover.
- The next step is to perform *mutation*. Each gene has a very small chance to be simply redefined to a new valid value, based on the mutation probability *Pm* (typical valued as 0.01~0.02). The role of mutation is to provide a guarentee that the GA search is not trapped on a local optimum. The mutation probability *Pm* can not be assigned a harmfully big number since if so it will make the system unstable. The range of valid values for genes is application dependent and will be discussed in section 3.

The parent selection and offspring reproduction process described above continues until the size of the new generation reaches MAX_CHROMS, at which time we replace the old generation with the new one. Generation after generation, GA attempts to maintain the balance between the exploration for generating new chromosomes and exploitation of discovered information which fits the environment best. As a result, GA is expected to solve and optimise solutions to problems that are otherwise very hard to solve.

3 Stack-based Virtual Machine (STM)

As described in section 2, it is up to the application what the range of valid gene values is, what a chromosome (i.e. a sequence of genes) represents, and how to evaluate the fitness of a chromosome.

Table 1: Simple Instruction Set

Instruction	Description
DUP	Duplicate the top of the stack (A => AA)
SWAP	Swap the top two elements of the stack (AB => BA)
MUL	Multiply the top two elements of the stack (2 3 => 6)
ADD	Add the top two elements of the stack (2 3 => 5)
OVER	Duplicate the second item on the stack (AB => BAB)
NOP	No operation (filler)

Consider a simple instruction set for a stack architecture on a virtual computer [2]. The virtual machine has no registers, only a stack for which instructions can manipulate the values on the stack. Our virtual machine recognizes only 6 instructions, shown in the table 1. These instructions are very simple, but can be used to solve a variety of functions. For example, if we want to compute the square of the top element of the stack, the following instruction sequence could be used (assuming the top of the stack contains our input value): *DUP_MUL*. This sequence duplicates the top of the stack, then multiplies the two together, and finally pushes the product back to the top of the stack. Note, *NOP* can be inserted anywhere in the instruction sequence without changing the definition of the corresponding function (e.g. *NOP_DUP_ NOP_ NOP_ MUL* is the same as *DUP_ MUL*).

3.1 STM instructions to GA Chromosomes

The range of valid values for each GA gene is naturally the set of six STM instructions above. When being initialized or mutated, GA genes are not allowed to have any other out-of-range value. Consequently, a GA chromosome is constructed as a sequence of valid STM instructions, which potentially respresents a STM-encoded mathematical function that aims at matching the user-provided target function. For example, if the stack contains three independent variables *x*, *y*, and *z*, from to to bottom, the chromosome `MUL_SWAP_DUP_SWAP_OVER_NOP_ADD_ADD_NOP_SWAP_NOP_ADD` will represent $f(x, y, z) = xy + 3z$. The figure 3 shows the dynamics of this chromosome.

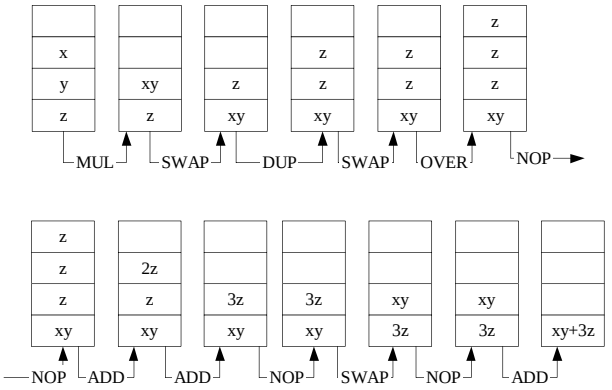

Fig. 3 A GA Chromosome Evaluated on the STM

Remember in section 2.1, we mentioned MAX_PROGRAM as the length of chromosomes and MAX_CHROMS as the size of a GA generation. It turned out that 12 and 3000 are some reasonable values from them respectively. They are surely adjustable.

3.2 Fitness Evaluation

The same group of input independent variable values will be presented to both the JEL-parsed user-defined target function and the candidate function in the form of a STM-based GA chromosome (section 3.1). As a result, two output values are computed.

Now, how do we evaluate the fitness of this chromosome? This is where creativity comes into play. There are tons of methods out there we can choose from. But what we need to keep in our mind is – what we eventually want is *the smaller the difference between the target output and the output produced by the function encoded in the chromosome (or the smaller the error), the larger the fitness of that chromosome*. Next, I will present what we did, and it seemed to work.

- If the STM that evaluated the candidate function exited successfully (no error was reported), we give it a TIER1 value (1, for example) and continue to the next step. Otherwise we just give it a zero and quit.
- If only one value was left on the stack, we add a TIER2 value (20, for example) to the current fitness and continue. Otherwise we quit with just a TIER1 value.
- If the top of the stack was the correct value (the same as the target output), we add in a TIER3 value (400, for example) to the current fitness. Otherwise, we add to it a faction of the TIRE3 value with regard to the *error*, i.e., $\frac{1}{1+error} \times TIER3$ where *error* is defined as the absolute difference between the two output values.

To avoid a chromosome from providing the correct answer, but working on one particular group of input variable values only, we test the chromosome a number of times with different combinations of input variable values (e.g. 10 times defined by the constant COUNT). From the discussion above, we can easily see that the largest possible fitness value is MAX_FIT = (TIER3 * COUNT) + (TIER2 * COUNT) + (TIER1 * COUNT) if the result generated by the chromosome matches the target perfectly every time we run the test.

Once we finish evaluating the fitness values of all the chromosomes in the current generation by repeating the process above, new parents are selected, GA operators are applied, and a new generation of chromosomes are born. The whole program is eventually completed either when we hit the MAX_GENERATIONS threshold (3000 is enough for simple function like x+y+z, but we could tune it larger for some more complicated functions), or the condition for accelerated learning is satisfied (section 2.2).

4 Java Expression Library (JEL)

To increase the flexibility and ease of use of our system, we employ the Java Expressions Library (JEL) developed by Konstantin L. Metlov and offered free of charge under the GNU General Public License. Utilizing the JEL, we are able to receive equations in the form of strings and evaluate them. Methods of the JEL receive the string expression and convert it into a Java class that is then dynamically loaded into the virtual machine, ready for evaluation. Please refer to http://galaxy.fzu.cz/JEL/ for more information on JEL.

5 Running Result Analysis

To present the output results we have run the system on the indicated function several times and with various parameters to indicate how different program lengths and probabilities affect the runtime and success of the algorithm. Particularly, in the following, *N* represents *MAX_CHROMS*, *Pc* represents the probability of crossover, and *Pm* represents the probability of mutation. Basically, *N* and *Pm* are the two major GA paparemters we want to play with.

- *f(x,y,z) = xy+z^3*

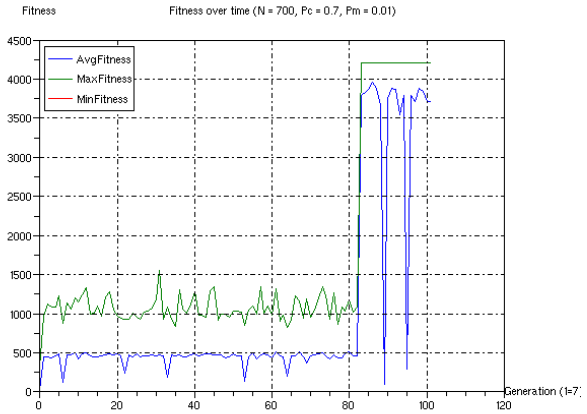

Fig. 4 Time to converge: 4400+ generations

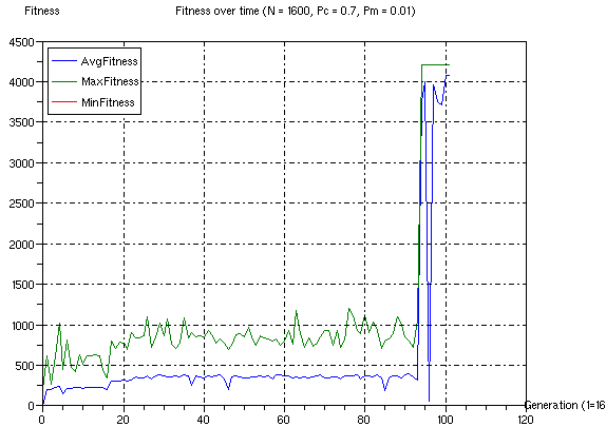

Fig. 5 Time to converge: 1600+ generations

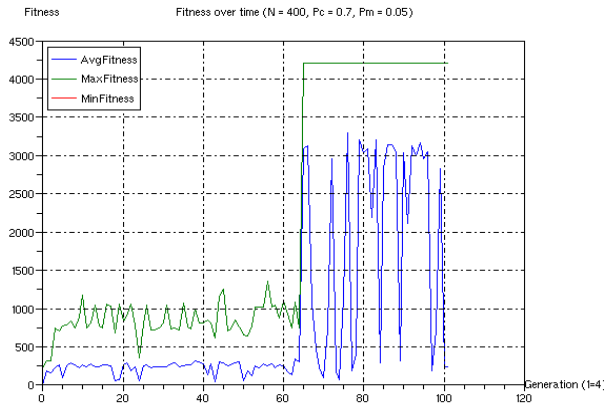

Fig. 6 Time to converge: 400+ generations

The performance of GA described above depends largely upon the following two factors: the level of the population diversity, i.e., the number of different GA chromosomes. The larger the number of chromosomes (*N*) in a GA generation, the more diversified they are, the better the chances our system will converge to the target (shown in figures 4 and 5). Another one is the extent to which GA individuals are able to interact with each other to produce effective offspring. This is mainly achieved through GA operators, in particular the mutation. Changing the probability of mutation (*Pm*) from 0.01 to 0.05 had a noticeable positive effect on the speed to converge (shown in figure 6).

- *f(x,y,)* = (x*y)+(y^2)+z

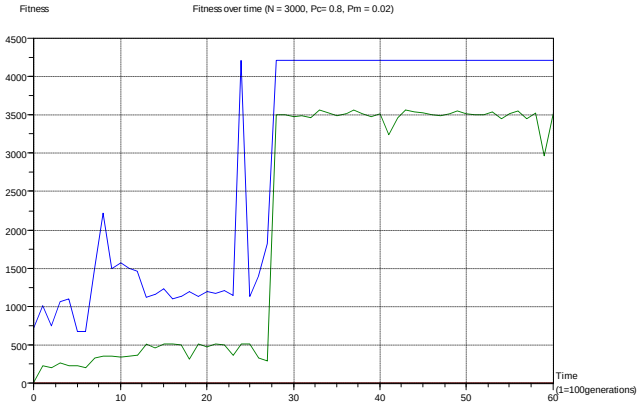

Fig. 7 Time to converge: 6000+ generations

- *f(x,y,)* = x^5+2y

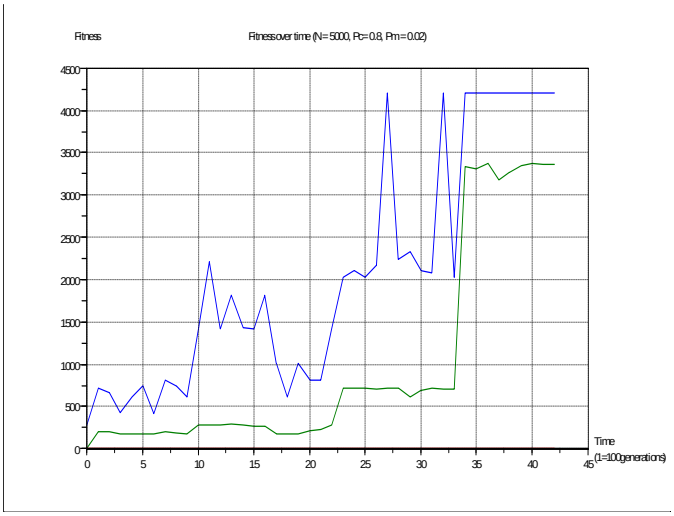

Fig. 8 Time to converge: 4200+ generations

As shown in figures 7 and 8, our GA system is smart enough to evolve some fairly complicated mathematical functions.

6 Conclusions

The computers have become indispensable in our everyday life. They act intelligently to help us in typewriting, car controlling, communication systems, data base systems, so on and on. In addition to taking

advantage of them, our computer science students should be given chances to learn, discover, and even implement those intelligent systems – what makes them intelligent? How they are built? How do we choose the right tool for the job? Etc. In, this paper, I answer these questions by presenting a team project three of my students did when they took my AI course. The system they built took use of Genetic Algorithm – an important branch of AI – to evolve sequence of simple instructions to a complex math function, with the help of Java Expressions Library and a stack-based virtual machine. As demonstrated and discussed above, the project (among other two team projects) made the course a huge success on introducing AI principles to students and retaining them for in-depth developments. Future tasks include expanding the set of instructions recognized by the stack machine such that even more complicated functions can be evolved in a reasonable time and improving the graphical user interface to enhance the usability and flexibility of the system.

7 References

[1] M. Negnevitsky, *Artificial Intelligence, A Guide to Intelligent Systems*, Addison Wesley Press, 2005.

[2] M. T. Jones, *AI Application Programming*, Charles River Media, 2003.

[3] D.S. Burke, K.A. De Jong, J.J. Grefenstette, C.L. Ramsey, and A.S. Wu, "Putting more genetics in genetic algorithms", *Evolutionary Computation*, Vol. 6, No. 1, pp. 387-410, 1998.

[4] G. Syswerda, "Schedule optimisation using genetic algorithms", *Handbook of Genetic Algorithms*, pp. 332-349, 1991.

[5] J. Holland, *Adaptation in Natural and Artificial Systems*, Ann Arbor: the University of Michigan Press, 1975.

[6] Z. Xu, A.S. Wu, "Adhoc-like routing in wired networks with genetic algorithms", *Ad hoc networks 2*, Vol.2, No.3, pp. 255-263, July 2004.