Genetic Programming of Finite Automata

Kaan Aksoy

INTRODUCTION

Genetic programming (also called *evolutionary programming*) is a set of techniques that draws influence from the biological process of evolution. In particular, genetic programming borrows the concepts of *mutation* and *crossover* from the natural world. By simulating evolution as it occurs in nature (albeit in a simplified fashion), genetic programming allows for the automatic generation of programs to solve complex tasks (Poli et al. 2008).

The programs that are evolved through genetic programming can be represented in many forms. Oftentimes, they are represented as a tree of nodes, where inner nodes represent various operators, and leaves represent constants and inputs to the program. On the contrary, this project aims to represent programs as *deterministic finite automata* (DFAs), which are a simple model of computation heavily discussed in theoretical computer science.

DFA IMPLEMENTATION

In formal theory, DFAs are defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with the following properties:

1. Q: The finite set of states.
2. Σ: The finite alphabet of symbols.
3. δ: $Q \times \Sigma \to Q$, the transition function.
4. $q_0 \in Q$: The start state.
5. $F \subseteq Q$: The accept states.

The Java implementation of DFAs used in this project follows the formal definition very closely. The states Q are numbered from 0 to $|Q| - 1$, and the alphabet Σ is simply a character array. In particular, this project uses a simple binary alphabet of just $\Sigma = \{0, 1\}$. The start state q_0 is represented as an integer, and the accept states F are represented as a boolean array of length $|Q|$. A value of *true* in the ith entry of this array indicates that the ith state is accepting.

More interesting is the representation of the transition function δ, which is stored as a 2D array. Below is an example representation of the transition function for a DFA of only two states (labeled A and B):

δ	0	1
A	A	B
B	B	B

In this example DFA, seeing a *0* while in state A results in no change in the state, and seeing a *1* while in state A results in a transition to state B. Furthermore, state B loops on itself if it sees either *0* and *1*.

It is simple to simulate a DFA on some input string w. The simulation begins in the start state q_0 and changes states according to the transition function δ for each symbol in w. Once the simulation is completed, the DFA accepts if it is in an accept state; otherwise, it rejects.

GENETIC PROGRAMMING IMPLEMENTATION

This project utilizes an *evolutionary algorithm* (EA) to evolve DFAs. At a high level, the EA implementation behaves as follows (Poli et al. 2008):

1. Create an initial population of random DFAs.
2. Repeat until an optimal DFA has been found *or* the maximum number of epochs have been exhausted:
    - Calculate the *fitness* of all DFAs in the current population.
    - Create a new (initially empty) population of DFAs.
    - Add the best DFAs (in terms of *fitness*) from the current population to the new population.
    - Apply *crossover* to create new children DFAs to be added to the new population.
    - Randomly mutate DFAs in the new population.
    - Replace the current population with the newly created population.
3. Store or display the best DFA in the final population of DFAs.

The following sections discuss in further detail each of the steps in the above EA implementation.

Initialization

Initialization involves generating an initial population of DFAs with random start states, transition functions, and accept states. The number of DFAs in the population is specified as a parameter to the EA.

Fitness Calculation

During each iteration (or *epoch*) of the genetic algorithm, DFAs in the current population are ranked according to their *fitness*, which is a measure of their efficacy. In the case of this project, DFAs are given a fitness score based on how accurately they recognize the specified *language*. For example, this language could be the set of all binary strings containing *100*, or the set of all binary strings divisible by 5.

The fitness of a DFA is calculated by first running the DFA on a *training set* of example inputs for which the expected output (accept or reject) is already known. Then, the number of correct classifications is counted and divided by the total number of training examples. This calculation can be formalized as follow:

$$\frac{1}{|S|}\sum_{w \in S} |y(w) - \hat{y}(w)|$$

In the above equation, S represents the training set of input strings, $y(w)$ represents the expected output for string w (1 for accept, and 0 for reject), and $\hat{y}(w)$ represents the output of the DFA under consideration when run on string w. The lowest possible fitness is 0 (meaning the DFA gives the wrong output for every input), and the highest possible fitness is 1 (meaning the DFA gives the correct output for every input).

Elitism

Elitism is the concept of selecting the best DFAs of the current population (in terms of fitness) to be placed in the new population. The number of DFAs to be moved is specified as a parameter to the EA, and is often called the *elitism offset*.

Elitism is important for the success of EAs because it guarantees that progress towards finding an optimal DFA is not lost when transitioning from one population to the next.

Crossover

Crossover is the process of combining two *parent* DFAs in order to create a new *child* DFA that hopefully has a higher fitness than its parents. It is akin to the biological idea that children share and borrow DNA from both of their parents.

The first step in crossover is to select two parent DFAs from the population. Every DFA in the population does not have the same probability of being selected. Rather, selection of parents is done through *roulette selection*, also known as *fitness proportional selection*. In roulette selection, the probability of selecting a DFA is proportional to its fitness, meaning that more fit DFAs are more likely to be selected.

Once two parent DFAs have been selected, the next step is to combine the two into a new child DFA. In order to decide what the child inherits from which of its two parents, first an arbitrary *crossover point* is selected. The child inherits everything before the crossover point from its first parent, and everything after the crossover point from its second parent. The resulting child DFA's transition function and accept states thus become a mix of its two parents' transition functions and accept states.

Below is an example of crossover being applied to the accept states of two parent DFAs. Recall that accept states are encoded as a boolean array:

Parent 1	**T**	**F**	T	F
Parent 2	F	F	**T**	**T**
Child	T	F	T	T

In the above example, the vertical line represents the crossover point for the accept states of the two parent DFAs. The child's accept states are built such that entries to the left of the crossover point are taken from the first parent, and entries to the right are taken from the second parent.

Mutation

As the name suggests, *mutation* is the process of randomly changing certain aspects of a DFA (for example, its accept states or transitions). Mutation is important to EAs because it helps maintain diversity within the population. Otherwise, there would be a possibility of reaching a population where all DFAs are identical and no more progress can be made (since crossover would no longer create new DFAs).

Although mutation is very useful, mutating DFAs too frequently can cause the EA to lose the progress it has made and never reach an optimal solution. The probability of mutating should thus be fine tuned as a parameter to the EA.

Termination

Once the EA has created an optimal DFA (with a fitness of 1) or the maximum number of epochs have been exhausted, the EA is terminated. At this point, the final population represents the best DFAs that the EA has managed to create for the given language.

EXPERIMENTS

To test the implementation discussed above, an evolutionary algorithm was employed to try building a DFA for the following language:

$$L = \{w \in \{0,1\}^* | \text{ w is a multiple of 5 in binary}\}$$

A total of 100 training examples were provided to the EA, with 50 of these being multiples of 5. For the initial test, the EA was run 14 times with the following parameters:

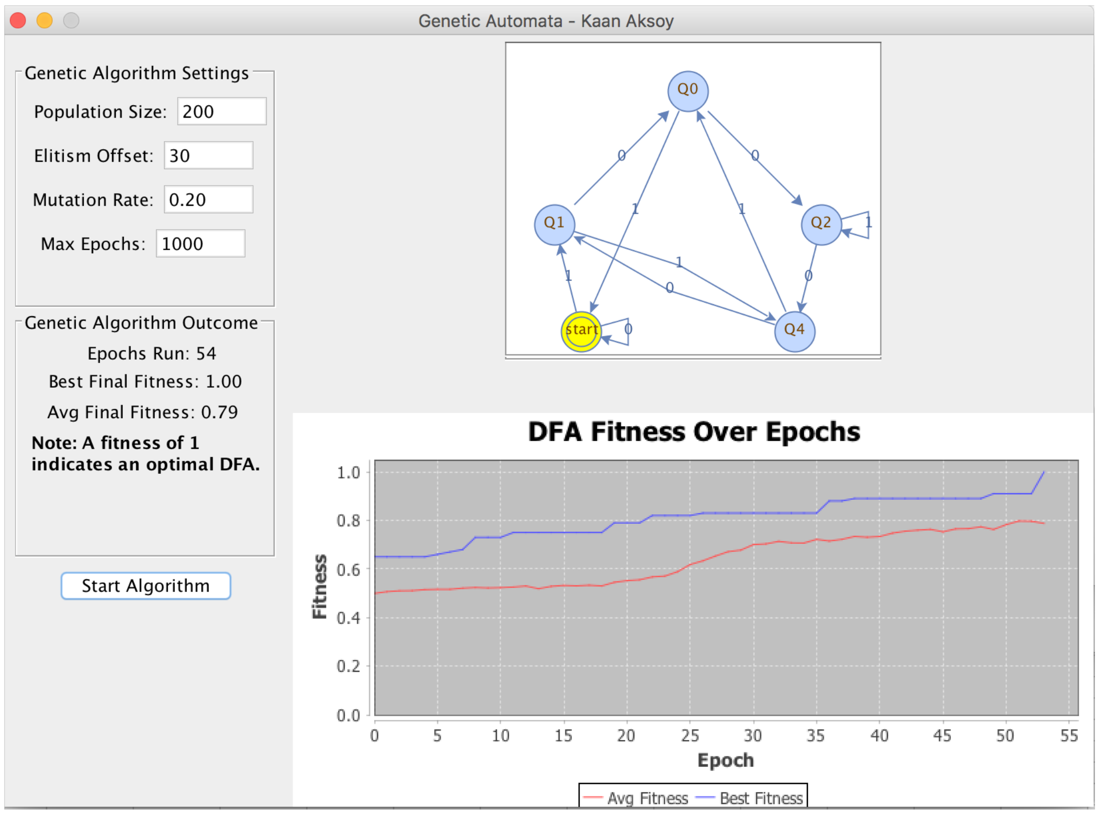
- *Population Size*: 200
- *Elitism Offset*: 30
- *Mutation Rate*: 0.20
- *Maximum Epochs*: 1000

The results of these initial tests were as follows:

Trial	Epochs Run	Final Fitness	Trial	Epochs Run	Final Fitness
1	54	1.0	8	72	1.0
2	85	1.0	9	243	1.0
3	72	1.0	10	68	1.0
4	59	1.0	11	1000	0.74
5	73	1.0	12	1000	0.79
6	45	1.0	13	47	1.0
7	51	1.0	14	45	1.0

As shown in the above chart, 12 of the 14 EA trials successfully located an optimal DFA for the language. The remaining 2 trials reached their maximum limit of 1000 epochs without finding an optimal DFA. The average number of epochs run for each trial was 198.27. Although these values could certainly be improved by fine tuning the EA parameters, they are nevertheless promising and confirm that the EA implementation is working properly.

A screenshot of the Java application running the first trial is shown below:


Genetic Automata - Kaan Aksoy

Genetic Algorithm Settings

Population Size: 200

Elitism Offset: 30

Mutation Rate: 0.20

Max Epochs: 1000

Genetic Algorithm Outcome

Epochs Run: 54

Best Final Fitness: 1.00

Avg Final Fitness: 0.79

Note: A fitness of 1 indicates an optimal DFA.

On the left side of the screen, the user is able to specify the various parameters for the EA, and is also given a brief overview of the outcome of the trial.

The top of the screen shows the best DFA created throughout the EA trial. Note that the DFA created in the trial depicted above takes the intuitive approach of keeping one state for each possible result of n mod 5. This is interesting to note because most people would do the same when designing a DFA for the language of multiples of 5.

The bottom of the screen shows a graph of how fitness changed across epochs during the trial. The blue line shows the fitness of the *best* DFA in each epoch, while the red line shows the *average* fitness of the entire population in each epoch.

CONCLUSION

The aim of this project was to employ genetic programming to automatically build DFAs for particular languages. The project was a success in that the implementation is capable of creating DFAs for fairly complicated languages (such as multiples of 5). In the future, it would be interesting to apply this evolutionary algorithm to other models of computation, such as *pushdown automata* or even *Turing machines*. Additionally, further analysis could be performed to determine the optimal parameter values (such as mutation rate) for the evolutionary algorithm.

REFERENCES

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming*, <`http://www.gp-field-guide.org.uk`> (With contributions by J. R. Koza).