A GENETIC ALGORITHM FOR FINITE STATE AUTOMATA

Aviral Takkar
Computer Engineering Department, Delhi Technological University(Formerly Delhi College of Engineering),
Shahbad Daulatpur, Main Bawana Road, Delhi-110042.India
aviraltakkar@gmail.com

Abstract

The genetic algorithm is described, including its three main steps: selection, crossover, and mutation. A comparison between implementation costs and running times of regular expressions matching a string is then made. The aim of this paper is to describe and analyze the behavior of an implementation of a non-deterministic finite-state acceptor using a genetic algorithm.

Keywords **Genetic Algorithm, Finite state acceptor, NFA.**.

1. INTRODUCTION

Genetic algorithms are computing algorithms constructed in analogy with the process of evolution. In biology, the gene is the basic unit of genetic storage [1]. Within cells, genes are strung together to form chromosomes. The simplest possible sexual reproduction is between single-cell organisms. The two cells fuse to produce a cell with two sets of chromosomes, called a diploid cell. The diploid cell immediately undergoes meiosis. In meiosis, each of the chromosomes in the diploid cell makes an exact copy of itself. Then the chromosome groups (original and copy) undergo crossover with the corresponding groups, mixing the genes somewhat. Finally the chromosomes separate twice, giving four haploid cells. Mutation can occur at any stage, and any mutation in the chromosomes will be inheritable. Mutation is essential for evolution. There are three types relevant to genetic algorithms: point mutations where a single gene is changed, chromosomal mutations where some number of genes are lost completely and inversion where a segment of the chromosome becomes flipped

2. GENETIC ALGORITHMS

The Genetic Algorithms follows the method of haploid sexual reproduction. [2]The population is a set of individual binary integers such as 1001011. Each individual represents the chromosome of a life-form. There is some function that determines how "**fit**" each individual is, and another function that selects individuals from the population to reproduce. The two selected chromosomes **crossover** and split again. Next, the two new individuals **mutate**. The process is then repeated a certain number of times. Let's consider the highlighted notions in more detail.

Fitness is a measure of the goodness of a chromosome, that is, a measure of how well the chromosome fits the search space, or solves the problem at hand. The fitness f is a function from the set of possible chromosomes to the positive reals.

Selection is a process for choosing a pair of organisms to reproduce. The selection function can be any increasing function, but we will concentrate on fitness-proportionate selection, whose selection function is the probability function

$$p_s(x_i) = \frac{f(x_i)}{\sum_{k=1}^{n} f(x_k)}$$

on the population $\{x_{1,} x_2 \dots \dots .. x_n\}$.

Crossover is a process of exchanging the genes between the two individuals that are reproducing. There are several such processes, but only one-point crossover is considered here, a process that is both standard and simple. A random integer i is selected uniformly between 1 and n. This is the place in the chromosome at which, with probability p_c, crossover will occur. If crossover does occur, then the chunks up to i of the two chromosomes are swapped. For example, the chromosome 11111111 when crossed with 00101010 at $i = 1$ gives the chromosomes 00101111 and 11111010. The **Crossover Rate** controls the probability with which a pair of chromosomes in a population will undergo a crossover.

Mutation is the process of randomly altering the chromosomes. Say that p_m is the probability that bit i will be flipped. Let i vary from 1 to n. For each i a random number is selected uniformly between 0 and 1. If the number is less than p_m, then the bit is flipped. The **Mutation Rate** controls the probability with which bits of a chromosome may be mutated.

Using the preceding notions, the Genetic Algorithm may be described as:

1. Start with a population of n random individuals each with l-bit chromosomes.
2. Calculate the fitness $f(x)$ of each individual.
3. Choose, based on fitness, two individuals and call them parents. Remove the parents from the population.
4. Use a random process to determine whether to perform crossover. If so, refer to the output of the crossover as the children. If not, simply refer to the parents as the children.
5. Mutate the children with probability p_m of mutation for each bit.
6. Put the two children into an empty set called the new generation.
7. Return to Step 2 until the new generation contains n individuals. Delete one child at random if n is odd. Then replace the old population with the new generation. Return to Step 1.

3. FINITE AUTOMATON

3.1. *Basic Definitions*

A deterministic finite state automaton (DFA) is a machine $D = (\sum, Q, q_0, F, \delta)$ where $\sum$ is a finite non-empty alphabet, Q is a finite set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accepting states, $\delta: Q \times \sum \rightarrow Q$ is the transition function.

A nondeterministic finite state automaton (NFA) is a machine $N = (\sum, Q, q_0, F, \delta)$ defined in the same way except that $\delta: Q \times \sum \rightarrow P(Q)$.

D accepts $x \in \sum^*$ iff $\exists p_0 \ldots \ldots p_{|x|} \in Q$ such that $p_0 = q_0 \; \forall \, i \in \{1, \ldots\ldots, |x|\}, p_i = \delta(p_{i-1}, x_i), p_{|x|} \in F$.

And similarly, N accepts $x \in \sum^*$ iff $\exists p_0 \ldots \ldots p_{|x|} \in Q$ such that $p_0 = q_0 \; \forall \, i \in \{1, \ldots\ldots, |x|\}, p_i = \delta(p_{i-1}, x_i), p_{|x|} \in F$.

The language $L(D)$ (or $L(N)$) of D (or N) is the set of strings D (or N) accepts. Two machines are equivalent if they have the same language [3].

3.2. *NFAs are exponentially more powerful than DFAs*

With respect to complexity, NFAs and DFAs are quite different.

Lemma 1: For each $n \in N$, there is a $n + 1$ state NFA whose minimal equivalent DFA has $\geq 2^n$ states.

Proof: Let $\sum \in \{0,1\}$ and let $L = \{x1y | x \in \sum^*, y \in \sum^{n-1}\}$. Then L is the language of NFA N that may be described graphically. N has the states $0 \ldots \ldots n$ where 0 is the start state, and an edge from $i - 1$ to i for $i \in \{1 \ldots . n\}$ and an edge from 0 to 0. The edge from 0 to 0 is labelled 0; 1, the edge from 0 to 1 is labelled 1, and the edge from $i - 1$ to i is labelled 0,1 for $i \in \{2 \ldots . n\}$. The only accept state is n.

Now suppose indirectly that some DFA D with $k < 2^n$ states has language L. Then by the pigeonhole principle, there are some two distinct strings $x, y \in \sum^n$, such that D ends up in the same state when run on x as when run on y. x, y are different, so there is some $i \in \{1 \ldots \ldots n\}$ such that $x_i \neq y_i$. Suppose $x_i = 0$ and y_i=1. Let z be any string of length $i - 1$, say $z = 1^{i-1}$. Then $xz \notin L$, $yz \in L$ and yet $D(xz) = D(yz)$, a contradiction[3].

Note also that the standard subset construction transforming an NFA into a DFA will produce a DFA D equivalent to N and with 2^{n+1} states [4].

3.3. *Implementation and running times of existing algorithms*

There are at least three different algorithms that decide if and how a given regular expression matches a string.

The oldest and fastest relies on a result in formal language theory that allows every nondeterministic finite automaton (NFA) to be transformed into a deterministic finite automaton (DFA). The DFA can be constructed explicitly and then run on the resulting input string one symbol at a time. Constructing the DFA for a regular expression of size m has the time and memory cost of $O(2^m)$, but it can be run on a string of size n in time O(n). An alternative approach is to simulate the NFA directly, essentially building each DFA state on demand and then discarding it at the next step. This keeps the DFA implicit and avoids the exponential construction cost, but running cost rises to O(m n). The explicit approach is called the DFA algorithm and the implicit approach the NFA algorithm. Adding caching to the NFA algorithm is often called the "lazy DFA" algorithm or just the DFA algorithm without making a distinction. These algorithms are fast, but using them for recalling grouped sub expressions, lazy quantification, and similar features is tricky. [5][6]

The third algorithm is to match the pattern against the input string by backtracking. Its running time can be exponential, which simple implementations exhibit when matching against expressions like (a|aa)*b that contain both alternation and unbounded quantification and force the algorithm to consider an exponentially increasing number of sub-cases.

4. IMPLEMENTATION OF NFA USING A GENETIC ALGORITHM

4.1. *Main Idea*

The main idea behind implementing a NFA using genetic algorithm is as follows

- Accepting a string using a NFA N is essentially a process of discovering a path $\{q_0 q_i q_j \dots \dots q_F\}$, if it exists, through the transition diagram of N, where the path is a set of edges (which may be repeated) originating at the start state q_0 and terminating at one of the final states $q_f \in \mathrm{F}$, where $q_i, q_j \in Q$ for $0 \leq i, j \leq n$ and i may or may not be equal to j.
- Let there be n states in N. Consider an input string $w = \{w_1 w_2 \dots \dots w_m\}$ over the alphabet $\sum$. If w is accepted by N, then there will be a path in N, $p = \{q_0 q_i q_j \dots \dots q_F\}$, where each edge $q_i q_j$, for $0 \leq i, j \leq n$, will be labelled w_k for $1 \leq k \leq m$.
- Thus, the question of acceptance is equivalent to asking whether a path exists in the NFA N whose edges have the labels corresponding to the letters of the input string.
- All possible paths through the NFA, originating at the start state, and terminating at one of the final states form a <u>search space</u> for a given input string.

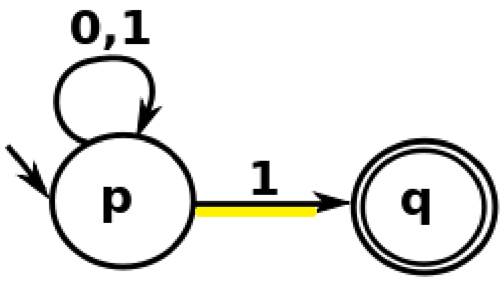

Fig. 1. An NFA N, for $w = 1$, the path through the N is pq.

4.2. *Representation of NFA*

The NFA N may be represented as a transition table, in the form of a 2-D array, with the rows being indexed by the state number $(0,1,2 \dots n)$ and the columns by the input symbols. (Say $\sum = \{0,1\}$). This is illustrated in Table 1.

TABLE I
TRANSITION TABLE T_n FOR N

Symbol	0	1
P	P	PQ
Q	-	

4.3. *Application of genetic algorithm*

To apply the genetic algorithm, each possible path in the search space is encoded in the form of a binary string, which represents some chromosome c. A typical chromosome may look like

$$1\,0\,0\,0\,0\,1\,1\,0\,0\,1\,0\,1\,0\,0\,1\,1\,0\,0\,0\,1\,1\,1\,0\,1\,0\,0$$

For a fixed gene length g, each chromosome can be parsed such that a series of c/g number of “transition numbers”, T, be obtained from it.

Let NFA N be represented by the transition table Tn. Each entry of the transition table is a set of states such as $\mathrm{Tn[state][input\ symbol]} = \alpha$. For example, in Fig. 2, $\mathrm{Tn[P][1]} = \mathrm{PQ}$. Thus $\alpha = \mathrm{PQ}$. The transition from state P on input symbol 1 to state P may be call transition 1, and the transition from state P on input symbol 1 to state Q may be called transition 2.

Thus, the transition numbers for state P on input 1 are $\mathrm{T} = \{1,2\}$, signifying transition to set of states PQ. In a similar fashion, the transition numbers for other states may be obtained.

The transition numbers $T = \{t_1 t_2 \ldots\ldots t_{\frac{c}{g}}\}$ for the current state are used along with current state and current input symbol to determine the next edge to be traversed, to reach the next state. The algorithm for traversing $\mathbf{N}$ given a chromosome $\mathbf{c}$, and an input string $\mathbf{w}$, and thus calculating the fitness of the chromosome $\mathbf{c}$, is now presented.

4.4. ***Algorithm for calculation fitness of a chromosome***

$Tn[i][j] =' \$'$ signifies that there is no transition in NFA $\mathbf{N}$ from state $\mathbf{i}$ on input symbol $\mathbf{j}$.

```
a) Initialize $i, j, state = 0$ and transitions = null
b) Let a buffer $\mathbf{T}$ contain the parsed bits of the chromosome, i.e. $\mathbf{T}$ contains a series of transitions as
   encoded by the chromosome. Let length of $\mathbf{T}$ be $l$.
c) While $i < l$ AND $j < m$ do
     a. If $Tn[state][w_j] \neq '\$'$ AND $T[i] \leq Tn[state][w_j].length$ then
          i. state = $state.T[i]$
          ii. transitions = transitions + $T[i]$
          iii. i++
          iv. j++
     b. else
          i. i++ /*ignore this gene and move on to the next one*/
d) if $state \in F$ AND input is over
     a. $c.fitness = MAX$
e) else if $state \notin F$ AND $i \nleq l$
     a. $c.fitness = MIN$
f) else
     a. $c.fitness = \frac{i}{l} * RANDOM(0,1)$ /*fitness is proportional to the length of chromosome
        examined*/
```

Thus the fitness of a chromosome is calculated.

4.5. ***Algorithm to accept string***

```
a) Initialise Chromosome Population[ ] = new Chromosome[pop_size]
b) $input =$ getInput()
c) Generations_required_to_find_solution = 0
d) for each chromosome in Population[ ] do
     a. chromosome = getRandomBits()
     b. chromosome.fitness =0.0f
e) while(1) do
     a. total_fitness = 0.0f
     b. for each chromosome in Population[ ] do
          i. chromosome.fitness = assignFitness($input, Tn$)
          ii. total_fitness = total_fitness + chromosome.fitness
     c. for each chromosome in Population[  do]
          i. if(chromosome.fitness = MAX)
               1. Return ACCEPTED
     d. Chromosome NextGeneration[ ] = new Chromosome[pop_size]
     e. i=0
     f. while i<pop_size
          i. offspring1 = Roulette(total_fitness,population)/*randomly select two parents*/
          ii. offspring2 = Roulette(total_fitness,population)
          iii. if(RANDOM(0,1) < CROSSOVER_RATE)/*perform crossover*/
```

```
                1. crossover = RANDOM(0,l)
                2. offspring1=offspring1.substring(0,crossover) + offspring2.substring(crossover,l)
                   AND offspring2=offspring2.substring(0,crossover) +
                   offspring1.substring(crossover,l)
          iv.   offspring1.Mutate()/*perform mutation*/
           v.   offspring2.Mutate()
          vi.   offspring1.fitness = 0.0f
         vii.   offspring2.fitness = 0.0f
        viii.   NextGeneration[i++]=offspring1
          ix.   NextGeneration[i++]=offspring2
    g.  Population = NextGeneration
    h.  Generations_required_to_find_solution++
    i.  If (Generations_required_to_find_solution > Max_Allowed_Generations)
           i. RETURN
```

5. Results

This algorithm has been implemented in Java (NetBeans IDE 7.2). It has been found that for those input strings that are accepted by the NFA implemented, the result is usually found within 0 to 5 generations, resulting in minimal running times. However, for those strings not accepted by the NFA, the algorithm does not provide results until the maximum allowable generations have been evaluated. Hence, in those cases, the response time depends upon the maximum allowable generations. Also, due to the unpredictable nature of genetic algorithms, at times a string maybe incorrectly rejected by the algorithm, resulting in decreased accuracy. Such cases have been found to occur when the length of the input string is over a particular threshold value, which is different for different NFAs. The following NFA has been implemented using the above mentioned algorithm.

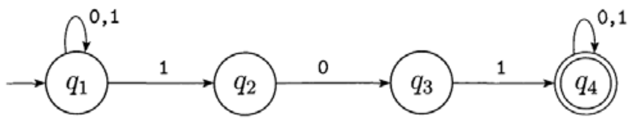

Fig. 2. A NFA accepting strings containing the substring '101'

TABLE 2
OBSERVATION TABLE

Length	*Result*	*Time*	***Generations required to find solution***	*Accuracy*
8	*Accept*	*0.062*	*0*	*1*
14	*Accept*	*0.012*	*0*	*1*
27	*Accept*	*0.04*	*1*	*1*
25	*Accept*	*0.011*	*0*	*1*
28	*Accept*	*0.009*	*0*	*1*
17	*Accept*	*0.008*	*0*	*1*
33	*Reject*	*0.229*	*101*	*1*
35	*Reject*	*0.216*	*101*	*0*
40	*Reject*	*0.22*	*101*	*1*
40	*Reject*	*0.232*	*101*	*1*

Percentage accuracy = 90%.

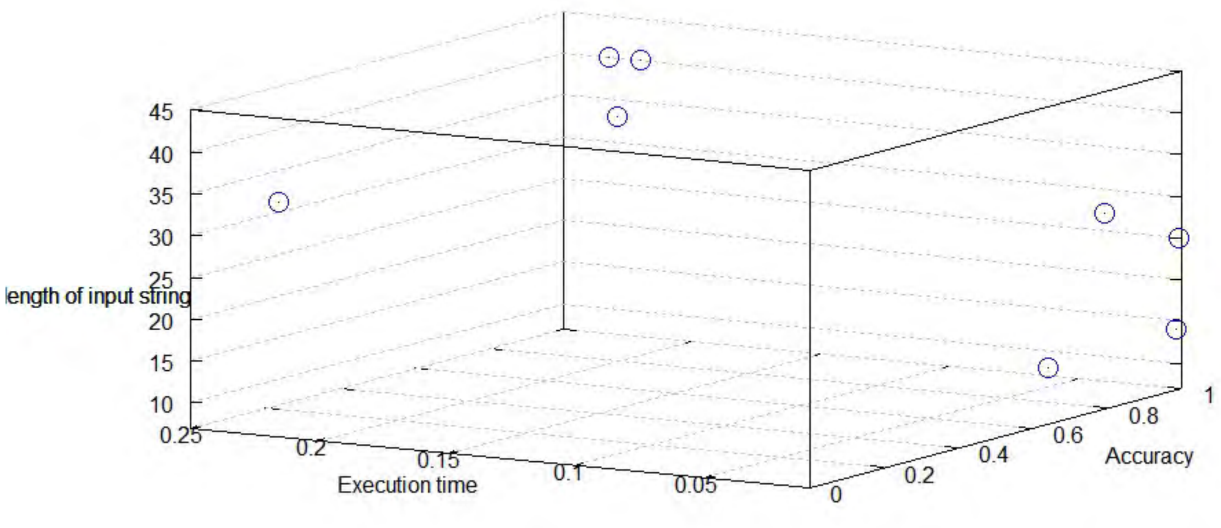

Fig. 3. This graph shows a scatter plot of TABLE 2.

These results have been obtained after specifying the following parameters

- Population Size = 100
- Chromosome Length = 300
- Gene Length = 4
- Maximum allowable runs = 100
- Crossover rate = 0.7f
- Mutation rate = 0.01f
- Radom function = java.math.random();

Thus it is found that although the exact running time of this algorithm on any given NFA and string is hard to predict, relatively small NFAs can be efficiently simulated by this algorithm, and a quick response time for small input string can be obtained by effectively guessing a path through the NFA. At times, a string maybe incorrectly rejected, but repeated experiments show that such cases occur only for relatively longer strings, if they occur at all.

Acknowledgments

I would like to express my deep gratitude to Professor Rajesh K Yadav, Delhi Technological University, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. His willingness to give his time so generously has been appreciated.

References

[1] Purves, W., Orians, G., and Heller, C., Life, the Science of Biology, Sinauer, 1995.
[2] Mitchell, M., An Introduction to Genetic Algorithms, MIT press, 1996
[3] Chris Calabro, http://cseweb.ucsd.edu/~ccalabro/essays/fsa.pdf, 2005
[4] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). Introduction to Automata Theory, Languages, and Computation (2nd ed.). Addison-Wesley.
[5] Cox, Russ (2007). "Regular Expression Matching Can Be Simple and Fast"