Creating Adaptive Sequences with Genetic Algorithms to Reach a Certain State in a Non-Deterministic FSM

Carlos Molinero, Manuel Núñez
Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid,
Madrid, Spain, 28007
Email: molinero@fdi.ucm.es,mn@sip.ucm.es

Robert M. Hierons
Department of Information Systems and Computing,
Brunel University
Uxbridge, Middlesex, UB8 3PH United Kingdom
Email: rob.hierons@brunel.ac.uk

***Abstract*—This paper aims to construct an evolutionary system, based on genetic algorithms, to solve the problem of univocally reaching a target state in a non-deterministic Finite State Machine. Our approach proposes the creation of an *adaptive sequence*, which is a tree of input and outputs that contains the possible behaviors of the non-deterministic Finite State Machine, through a Genetic Algorithm. Essentially, we will characterize the DNA of the individuals as an adaptive sequence and allow the population to evolve until a solution is found. To assure the validity of our approach, we compare it with other methodologies such as hillclimbing and random. We show that the Genetic Algorithm obtains a higher rate of success in creating the adaptive sequences.**

I. Introduction

Evolution is the way life adapts to its environment. If this environment is the landscape formed by the solution of a problem, then this adaptation may serve us to find a way to solve it. This idea is what led evolutionary approaches to be used as search techniques.

Testing ([1], [2]) is an important part of the production of new systems since it takes a high percentage of the time and cost of developing them. Therefore, being able to create new methods that allow us to discover faults in the implementation is a necessity. *Reaching a specific state* is a fundamental part of the testing process because it allows the tester to move the implementation to that state and continue the testing of a certain part of a system, such as a specific component of an embedded system. If we solve this problem in a deterministic FSM, the solution is found in linear time. However, in the context of a non-deterministic FSM, this problem belongs to the EXPTIME complete category, therefore the use of heuristic methods is a must. Moreover, since non-determinism may arise in several ways, as when abstracting data from protocols or when composing a system out of several individual components, to produce test suites for non-deterministic implementations is a problem of practical importance.

Research partially supported by the Spanish MICINN project TESIS (TIN2009- 14312-C02-01), the UK EPSRC project Testing of Probabilistic and Stochastic Systems (EP/G032572/1).

Testing non-deterministic finite state machines produces several complications during the development and application of the test suite. One of them is that we can no longer state equivalence between machines. The equivalence relationship to be established will be called *r-equivalence* and consists in considering that the traces of the implementation conforms to those of the specification (for a detailed and formal explanation, see [3], [4], [5]). Usually, when in a non-deterministic context we will use the *all weather conditions* assumption, which means that if enough inputs are applied to the machine then we will observe every output available in the implementation.

Several studies have already considered the problem of defining test suites for non-deterministic machines (see for example [3], [6], [5], [7], [8]) and shown that this problem is computationally costly. This paper provides an automated way to find adaptive sequences that allow the tester to move through a non-deterministic finite state machine in such a way that always reaches a certain target state.

Next, we intuitively describe the main aspects of the problem that we solve in this paper.

An *adaptive sequence* (see for example [4], [9], [6]) is a tree such that the unique edge that leaves its root will be labeled by an input (to be applied to the $ndFSM$), and it will reach a state such that from this state outgoing edges labeled by outputs (returned from the $ndFSM$) will depart arriving at one state each from where a new input will depart and so on. Therefore, an adaptive sequence is applied by taking the first input and introducing it to the machine, observing the output produced, reaching the node labeled with that output, to next introduce the input that departs from that node. This approach is intended to be used when a tester needs to move the $ndFSM$ to a certain state from a known starting point in order to continue with the execution of a test, or to test a specific property that arises from that target state.

In order to create this adaptive sequence we use the evolutionary technique of Genetic Algorithms (GA), and we propose a methodology for its evolution in the context of our problem. GAs have shown a good performance in search and

978-1-61284-063-5/11/$26.00 ©2011 IEEE

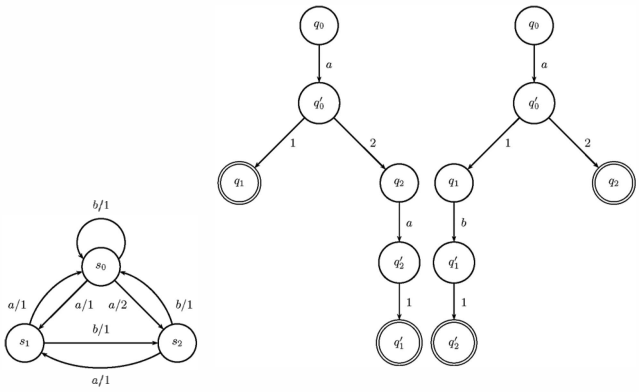


Figure 1. An observable $ndFSM$ (left) and two possible adaptive sequences, one in which we try to univocally reach state s_1 (center) and another in which we try to univocally reach state s_2 (right).

optimization problems and there are a number of papers where GAs are used in testing (e.g., [10], [11], [12], [13]). These approaches usually represent the test data generation problem as an optimization problem and use heuristics to generate test cases.

We will study different evolution methodologies for our proposed GA, in order to understand in which ways this population is able to attain a higher level of adequacy towards its environment. To do so, we run different experiments where we will study whether it is better that the descendants inherit the properties that led their ancestors to get a higher value of adaptation or to randomly restart these values upon the generation of a new candidate solution. Moreover, we will also study what level of randomness in the choosing of the next node to be added to the adaptive sequence is adequate.

To be able to enhance the comprehension of what a $ndFSM$ and an *adaptive sequence* are we will show in Figure 1 a $ndFSM$ (left) and two adaptive sequences, one reaching state 1 and another reaching state 2.

The rest of the paper is organized as follows. In Section II we introduce some auxiliary notation. In Section III we will explain the proposed methodology. In Section IV we will show the results of our experiments. Finally in Section V we will present our conclusions and mark future lines of research concerning this issue.

II. Preliminaries

In this section we introduce some notation and general intuitions of the topics that will be used throughout the rest of the paper.

We begin by reviewing the main features behind genetic algorithms since they will be used along the paper.

Genetic algorithms [14] are a technique based on the laws of natural evolution proposed by Darwin. Generally, these algorithms codify a possible solution to a given problem in the form of a DNA sequence of one of the candidate solutions and then evolve the population to achieve an optimal solution. The operators for any GA are *reproduction* of the population, *mutation* of the genes and *selection of the fittest*. Mutation allows the population to evolve towards a higher fitness. Maintaining several solutions in the genetic pool and mixing them by using the reproduction operator helps to explore a bigger part of the landscape of the solution and therefore avoid local maxima, escaping stagnation.

Intuitively, the pseudo-code of a genetic algorithm follows these steps:

1) Initialize the population.
2) while the stop criterion is not met do.
   a) Judge the fitness of the population using the heuristic.
   b) Select the fittest specimens and restrict the number of the population.
   c) Reproduction of the population.
   d) Mutate the specimens.
3) Present a solution.

There do exist several drawbacks to the use of genetic algorithms. The biggest one is, of course, a proper definition of the fitness function. Since this is an approximation, one can always argue that a better heuristic may exist. In general, we can observe how important it is to have a heuristic landscape without plateux to allow the population to evolve (see for example [15]). Another of its problems is that these functions should be relatively easy to compute in order to allow the minimal time to elapse between two generations of the genetic pool. This fact allows the system to have a bigger number of generations, which takes us closer to finding an ideal individual. Another of the problems is that there is no way to know when to stop evolving the solution. This problem is dependant on the specific problem, because there may be a way to predict that we are not going to find a better solution than the one already present in the pool of candidate solutions.

Next, we present some definitions of the elements and functions that will be used throughout the paper.

Definition 1: A non-deterministic observable finite state machines ($ndFSM$ from now on), is a mathematical representation of a system that has inputs and outputs and it is composed of states and transitions between them. They are represented through a tuple (S, I, O, s_0, s_t, Tr) where:

- S is a finite set of states.
- I is the input alphabet.
- O is the output alphabet.
- $s_0 \in S$ is the initial state.
- $s_t \in S$ is the target state.
- Tr is the set of transitions. A transition $t_i \in Tr$ is a tuple $t_i = (s_1, i, o, s_2)$ where $s_1, s_2 \in S$, $i \in I$, $o \in O$ and it will be represented throughout the text by $s_1 \xrightarrow{i/o} s_2$.

□

We say that M is observable if there do not exist two different transitions (s, x, y, s_1) and (s, x, y, s_2), with $s_1 = s_2$.

Next, we define *d-reachibility*. A state s is *deterministically reachable* (see [3]), which is denoted by *d-reachable*, in the context of a non-deterministic FSM when there exists an input sequence that applied to the initial state can only return the

state we are looking for. Only a subset of the states are *d-reachable* in a $ndFSM$. Our purpose is to use an adaptive sequence that would allow to deterministically (univocally) reach any state in the non-deterministic machine. This can be achieved in two cases. One often appears when all the leaves (end nodes) of the tree that forms the adaptive sequence represent the target state of the $ndFSM$. The second case arises when the tree is infinite but every leaf that can be reached after a finite sequence represents the target state. Of course, since this perfect solution will seldomly appear, we introduce the concept of a sequence *d-α-reaching* a state, being α the percentage of times that the adaptive sequence is able to reach that target state.

We will define a function such that given $M = (S, I, O, s_0, s_t, Tr)$, computes the distance from every other node to the target state by using Dijkstra's shortest path algorithm. The type of the function is $d_M^{s_t} : S \rightarrow \mathbf{N} \cup \{nil\}$ and it performs the following operations.

1) Create a graph $G = (V, E)$, where V is the set of nodes and E is the set of transitions, such that V will have one element for each state contained in S. The set E will have one transition for each transition contained in Tr with the peculiarity that it will be inverted, that is, the end node will be switched with the start node. This will create a graph that is an inversion of the original machine M.
2) From the node in V representing the target node $s_t \in S$, calculate the tree formed by the Dijkstra shortest path algorithm, and store the distances to each node in $d_M^{s_t}$. Any node not appearing in this tree will have a value of nil.

Let us note that the computational complexity associated with this function is $O(E + V * log(V))$.

We will next formally define our adaptive sequences.

Definition 2: Given $M = (S^M, I, O, s_0^M, s_t^M, Tr^M)$, an *adaptive sequence* for M is a tuple $T_M = (S, Tr, s_0, S_I, S_O, S_G, \pi)$ where:

- S is the (possibly infinite) set of states.
- $s_0 \in S$ is the initial state.
- S_I, S_O and S_G conform a partition of S, that is $S_I \cup S_O \cup S_P = S$ and they are pairwise disjoint.
- $Tr \subseteq (S_I \times I \times S_O) \cup (S_O \times O \times S_I)$ is the transition relation.
- π is a function that maps the states from the adaptive sequence to the states of $ndFSM$. $\pi : S \rightarrow S^M$ such that for all $s, s', s'' \in S$ we have that $s \xrightarrow{i} s', s' \xrightarrow{o} s''$ if and only if $\pi(s) \xrightarrow{i/o} \pi(s'')$.

The transition relation and the sets of states fulfill the following conditions:

- S_I is the set of input states. We have that $s_0 \in S_I$. For all input state $s \in S_I$ there exist at most one outgoing transition $s \xrightarrow{i} s' \in Tr$. For this transition we have that $i \in I$ and $s' \in S_O$.
- S_O is the set of output states. For all output state $s \in S_O$ we have that for all $o \in O$ there exists a unique state s' such that $s \xrightarrow{o} s' \in Tr$. In this case, $s' \notin S_O$. Moreover, there do not exist $i \in I$ and $s' \in S$ such that $s \xrightarrow{i} s' \in Tr$.
- S_G is the set of target states. In most cases this set will be formed by a unique state. We say that these states are terminal. That is, for all state $s \in S_G$ we have that there do not exist $a \in I \cup O$ such that $s \xrightarrow{a} s' \in Tr$.

We say that an adaptive input sequence is valid if the graph induced by Tr is a tree with root at the initial state s_0.

□

III. Methodology

One specific feature of our genetic algorithm is that it is not based on a normal DNA bit sequence, but the DNA is itself an adaptive sequence, that is, the DNA is a labeled tree.

We initially consider a population of 25 individuals. Each individual has a parameter chosen at random between 0 and 0.5 that represents its tendency towards choosing a random or a minimum distance node. This parameter does not evolve and it is randomly selected at the creation of each individual. The choice of these decisions was taken by following the results of the comparisons between different GAs to be shown in Section IV. Therefore, the definition of a candidate solution of the population can be represented by a tuple in which, in addition to the DNA and the tendency to choose randomly new nodes (α), we include its fitness value, that is, $cs = (DNA, \alpha, value)$.

The mutation rate is set to one mutation, every generation for each individual. Reproduction is done 10 times in each generation, where two children per crossover are created.

The first step for our algorithm, that will subsequently serve us to compute the fitness value of the individuals, is to compute the distances from every state to the target state.

Note that computing an individual based in Dijkstra's shortest path algoritm (a candidate solution that takes the shortest path to the target) is not computationally costly, and this step, according to our experimental results, increases the quality of the population. We set one of the 25 candidate solutions of the initial population to have the DNA formed by a shortest path to the target. Through a similar reasoning, given that computing a completely random individual is also made in linear time, and that our experiments showed that most of the times random outperformed the Dijkstra method, in each generation we add a random individual to incorporate diversity into our genetic pool.

We use two stop criterions. The first one is that the population finds a perfect solution in which a 100% of the times the target state is reached. If this is not achieved we will stop the search after 500 seconds, although this time can of course be increased with a subsequent improvement of the solution.

Next, we give more details about the algorithm that controls our methodology.

A. Operators

We present here separately the three operators common to every GA: mutation, reproduction and selection of the fittest.

1) Mutation: The mutation operator randomly either adds or deletes an input/output tree to its DNA. In order to choose which node to mutate, the algorithm randomly traverses the DNA until we find a state $s \in S$ with no children. This random traversal is done to choose a node in linear time. Although it would be better to get all the end nodes of the tree formed by the DNA, depending on the branching of the $ndFSM$, this choice could produce an exponential explosion. One positive outcome of our choice is that the DNA will evolve by modifying the nodes that have a higher probability of being the ones returned by applying the adaptive sequence to the machine (nodes that are further away from the origin are less likely to become the final state). A drawback of this method is that sometimes, mutation will try to mutate the target node, in this case, no transformation is made to the DNA and. therefore, mutation rate is lower than one for each generation and for each individual.

If the chosen state is not the target state, mutation can perform two operations, 50% of the times it will delete the node and the rest of the times it will add a subtree to that node. Since the node chosen will always belong to the set S_O, deleting it will cause the deletion of the chosen node, its parent and all the children of its parent.

If, on the contrary, the algorithm chooses to add a subtree, then it checks the subset of inputs available from $\pi(s)$ and takes one of them. Depending on the parameter α, some of the times the node will be randomly chosen and the rest of the times it will choose the closest node to the target. Therefore mutation behaves like a hill-climbing algorithm, but it does not always choose the same states in order to keep different DNAs coexisting in the DNA pool.

Consider that the chosen input is $i \in I$ and that the algorithm adds a state $s_1 \in S_O$ and a transition to Tr $s \xrightarrow{i} s_1$, and for every transition $s_M \xrightarrow{i/o} s'_M \in Tr^M$, where $s'_M \in S^M$, adds states $s_2^o \in S_I$ and transitions $s_1 \xrightarrow{o} s_2^o$.

In order to improve the efficiency of the algorithm, in the case that the creation of a subtree produces a node s' such that $d_M^{s_t}(\pi(s')) = nil$, meaning that there is no way to reach the target node from s', the node is added to a special list of nodes to be deleted in the next mutation, and the mutation is preset for the next generation to automatically delete the subtree that contains such node.

The mutation operator is graphically shown in Figure 2.

2) Reproduction: The methodology used to choose which individuals to cross is roulette wheel selection, which adds proportional probability of being chosen to the fitness value of an individual. Roulette wheel selection is a procedure where individuals are added to a multiset, each is added a number of times proportional to their fitness value. Once the adding of the candidate solutions is finished, one element is randomly retrieved. In order to eliminate crossing over the same points, and to produce two identical children, We decided to choose the first individual and then eliminate this candidate solution from the roulette wheel before choosing the second individual.

In our setting crossover is complicated since we need to find a specific state in two DNAs that represent the same state in the original specification. Therefore, this operation randomly traverses the DNA of one of the candidate solutions (the mother) and, for every state that it visits, makes another random travel through the sequence of the father. In the case of finding a non-initial state that is coincident through π in the set of states of M and such that the states belong to S_I, then it adds them to a set of possible crossover points. Later on, it randomly chooses a state from that set of crossover points, and exchanges the trees of both candidate solutions.

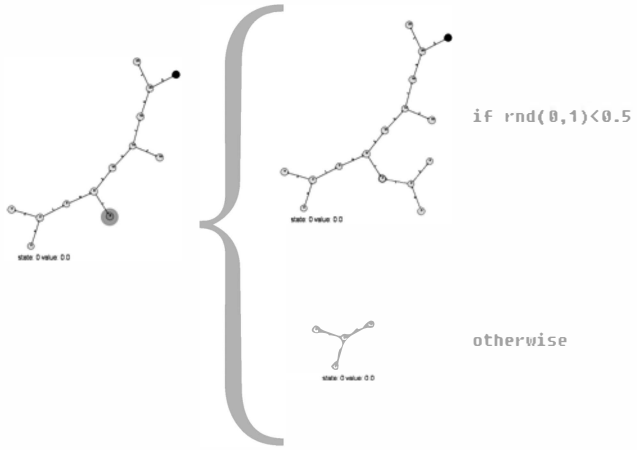


Figure 2. Mutation operator.

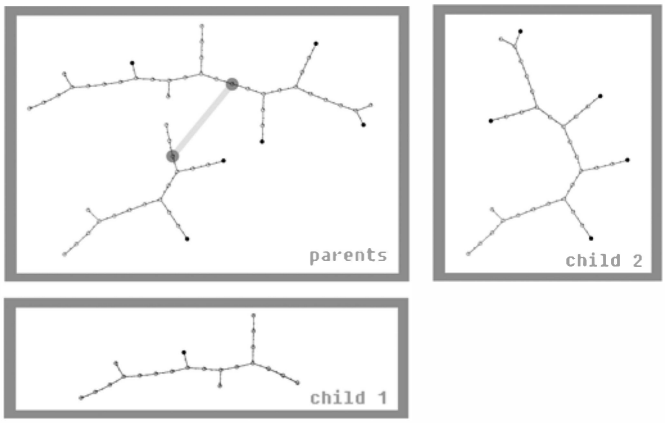


Figure 3. Crossover operator.

A graphical representation of this operator can be found in Figure 3.

3) Selection of the fittest: In order to maintain the same number of individuals in the population, after judging the fitness of each individual, the worst individuals are removed, and the fittest one is passed to the next generation without making any mutations to its DNA to be able to preserve the best solution. This is usually referred in the literature as *elitism*

B. Heuristic

Our fitness function is given by a sampling method. We use the DNAs of each candidate solution for a total of 100 iterations, and measure the number of times that we arrive to the target state. This number of times is multiplied by a coefficient β to increase its relative importance. In our case we have used $\beta = 100$, this value should be equal to the number of nodes (n), since the maximal distance from a node to another is set by this limit, and doing so, we can be sure that we are giving more importance to having the target node in our adaptive sequence. In order to complete this heuristic, every time that it ends up in a state that is not the target, we subtract the distance from that state to the target from the heuristic. Finally, if $d_M^{s_t}(s) = nil$ then we remove 50 points ($n/2$) from the heuristic to penalize the choosing of that node.

Our GA creates adaptive sequences that either have all end nodes as the target state, or it relegates them to a deeper part of the tree. This provides a higher rate of success, since it returns a bigger number of non-deterministic choices that takes us to the goal state. Therefore, if we quantify the choices then, probabilistically, we have a greater chance of arriving to the target. In order to illustrate this, consider that we have Figure 4 where the $ndFSM$ is depicted in the left, and in the right hand side there are two possible adaptive sequences.

Our heuristic considers in a indirect manner, the size of the graph, and the ratio of end nodes that are equivalent to the searched state, and their relative position in the tree. Since the heuristic measures the number of times that we hit the target out of a hundred runs, we have a higher rate of success when there are proportionally more end nodes that are the target and when the end nodes are closer to the root. This is so because, non-determinism probabilistically favors a shorter outcome.

One important fact to note is that our heuristic, since it is a sampling method, does not give us an exact measure of the number of times that our adaptive sequence reaches the target. However, by taking a look at one of the graphs (see for example Figure 7) we can observe that there exist different amplitudes in the oscillation of the measure. Actually the more consistent a measure is with the times measured before, the higher rate of equal end nodes that our sequence has. Therefore, the oscillation in the graph gives us an idea of how close we are to having a perfect adaptive sequence (one that will take us to the target each time).

IV. Experiments

The implementation of our framework has been done in *Java JDK 1.6*, with *NetBeans* as IDE. For the implementation we have used the graphic library from *processing (http://www.processing.org)*, together with the library *traer physics (http://www.cs.princeton.edu/~traer/physics/)* to help ordering the states in the space, to represent the machines. The executable application can be found in the address *http://www.carlosmolinero.com/reachState.zip*.

We use for our experiments a randomly constructed $ndFSM$s with one hundred nodes (that is, $n = 100$). In order to make sure that it is connected we add a transition between each node in a linear way (node s_i gets connected with node s_{i+1}), and force a branching level of 3, which means that the $ndFSM$ makes $3*n = 300$ attempts to add random transitions (in the case that the input and output is already present in the state the transition is not added because the $ndFSM$ is observable). The set of inputs is $I = \{a, b\}$ while the outputs are $O = \{1, 2\}$.

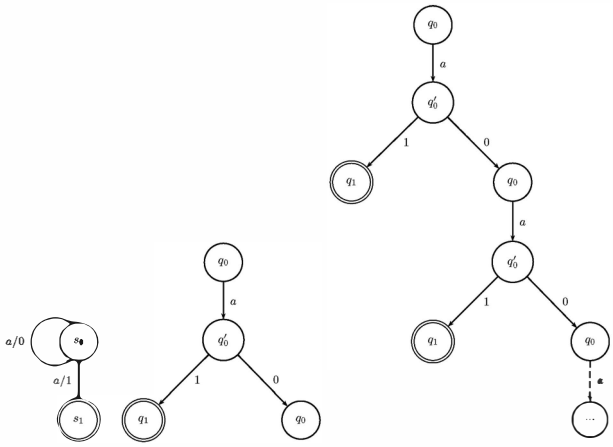


Figure 4. An observable $ndFSM$ (left) in which we try to univocally reach state s_1. And two possible adaptive sequences (center and right).

We will use our first experiment to narrate the differences between our approach and other techniques. Experiment A is represented through two charts, the first chart, in Figure 7, shows the fitness values of the different methods, while the second chart (Figure 8) represents the sizes of the adaptive sequences.

As we can see in Figure 7, our GA performs better than the rest of the methods, as it maximal representative (shown in Figure 6) scores higher than any other method. Random is able to reach very high levels of fitness but never assuring it, and Dijkstra is of a constant value since no updating of the adaptive sequence is performed. The only method that is comparable in quality to the GA is hillclimbing, but as we can see in Figures 10 and 6, and in the chart given in Figure 8, the size of the adaptive sequence created by the GA is much smaller than the one created by hillclimbing. This characteristic not only allows the GA to compute during more generations (the size of the hillclimbing becomes unmanageable at some point) but it also allows the GA to improve its fitness value faster, because having less nodes, allows that a small modification of the tree notoriously improves it. Another thing that we can observe in Figure 7, is that the oscillations in the value are also an indirect effect of the size of the tree. They are produced by the number of different nodes in the tree, together with the measurement method of the fitness value. Since a sampling method is used, 100 executions return different nodes as the end node, if all the end nodes of an adaptive sequence would be equal, then no oscillation would be seen in the graph. Thus, the smaller the adaptive sequence (for a given fitness value) the smaller that the oscillation will be, since the number of

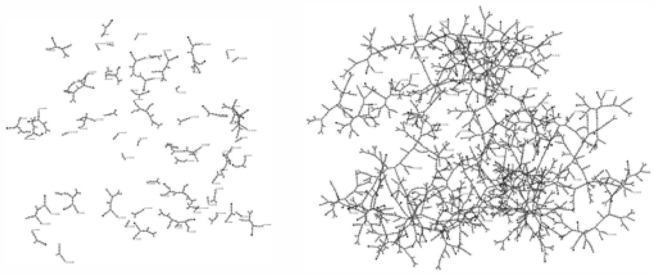


Figure 5. Evolution of the whole population of the GA. An initial stage (left) and the final configuration (right).

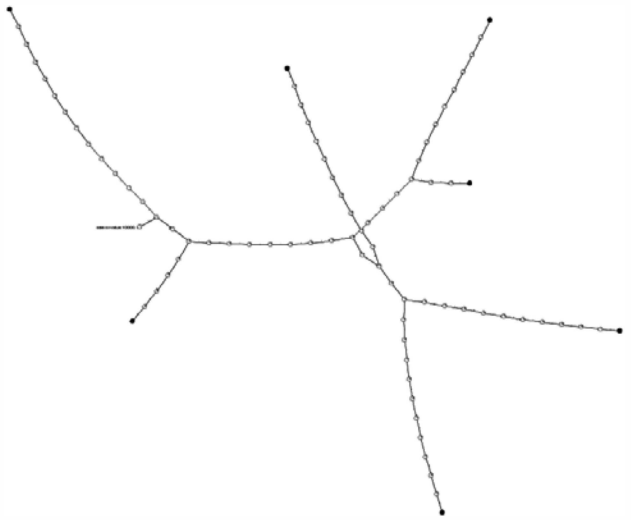


Figure 6. Experiment A, GA reaches the target 100% of the times.

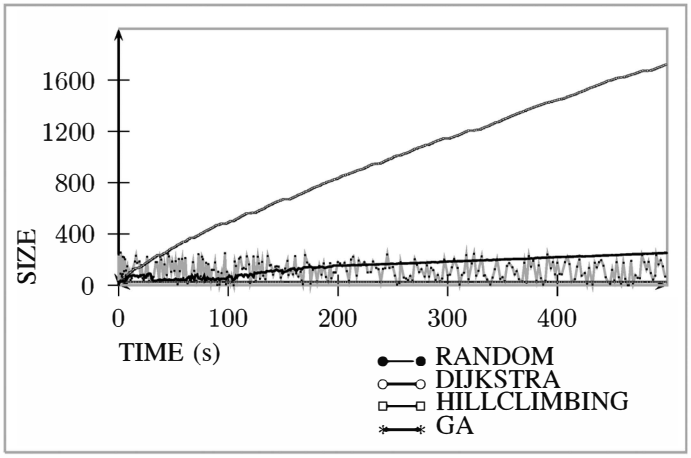


Figure 8. Graph representing the sizes in experiment A.

Figure 9. Experiment A, dijkstra reaches the target only 1% of the times.

end nodes equal to the target proportionally to the rest of the end nodes, has to be larger. In Figure 5 we show the evolution of the population from an initial step to its final configuration.

Remember that the value of the fitness function is based on a sampling method. Therefore, even if it shows that the fitness value has reached a maximum, it does not mean that a perfect solution has been found, but that all the times that the adaptive sequence has run, it has reached the target.

Next we present a short explanation of the other methods used to compare against our methodology. A random creation of an adaptive sequence, a shortest path adaptive sequence created using Dijkstra's shortest path algorithm, and a hill-cimbling approach.

A. Comparison Against Other Techniques

In this section we present a number of techniques that will be used to give the reader an idea of the relative quality of the GA methodology presented in this paper.

1) Random: It creates an adaptive sequence by mutating a random number of times, based in the number of states, with $\alpha = 0$. The mutation operator for random only adds a tree, never deletes.

2) Dijkstra: The Dijkstra individual is created by traversing the non-deterministic FSM from the origin to the target, in the fastest way possible using the Dijkstra's shortest path algorithm. An example of this is shown in Figure 9.

3) Hillclimbing: Hillclimbing as well as the GA, uses Dijkstra's individual as a starting point. From there on it checks all the end nodes of the tree, chooses one of those nodes and tries to add a subtree. If adding the subtree does not improve the fitness value, then it chooses another node and performs the same operation. Since trying to check all the possibilities can be exponential, the algorithm passes to the next generation when it finds a new configuration that is better than the last one. We can see an example of the adaptive sequence created by hillclimbing in Figure 10.

Next, we comment on the results of our experiments. The table shown in Figure 11 clearly shows that the level of achievement of the GA is high in most cases. On average it reaches the target state 83.18% of the times, although some non-deterministic machines proved to be too complex to find a solution by any methodology (see experiment number 5 or even 17). The image of the center of Figure 11, shows that our technique had a good level of coverage, and outperforms in almost any case the rest of the methodologies, it is usu-

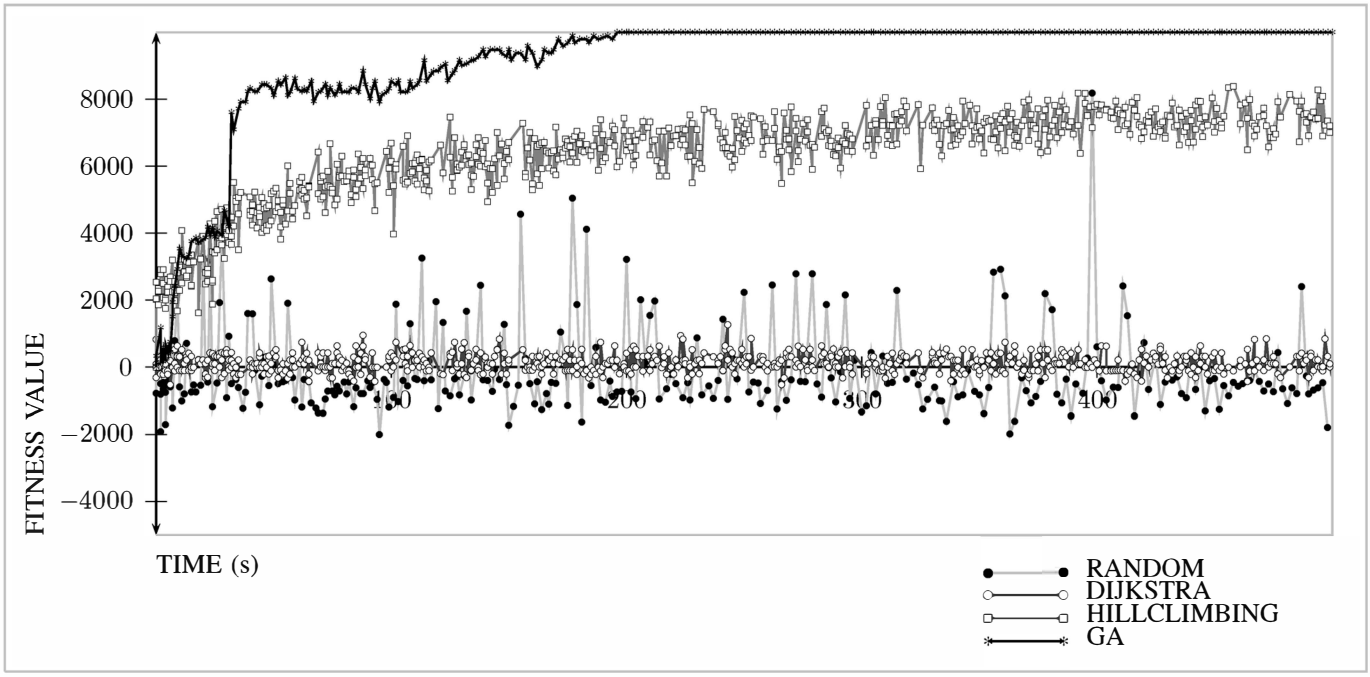
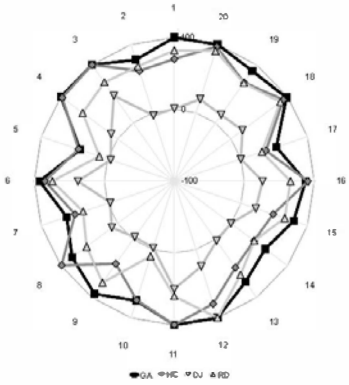


Figure 7. Graph representing the fitness value of the different individuals in experiment A.

Comparison								
	HEURISTIC VALUE (%)				SIZES			
#	GA	HC	DJ	RD	GA	HC	DJ	RD
1	100	70.11	1	83.93	252	1722	27	69
2	77.66	61.25	-3.76	68.18	234	1534	9	152
3	100	100	46.96	69.89	156	1340	7	104
4	100	97.92	11.42	61.8	114	2265	10	97
5	41.6	43.87	-4.47	12.52	275	2022	25	73
6	92.34	84.7	37.6	74.21	261	2027	4	104
7	61.52	49.11	-3.81	37.04	254	1225	7	81
8	80.34	98.98	10.21	54.77	253	2350	7	81
9	93.80	42.12	-3.94	74.54	184	1920	26	233
10	75.01	73.95	-2.63	9.98	320	1873	17	89
11	100	100	50.08	60.26	27	52	6	146
12	100	79.57	24.68	100	6	2520	5	100
13	73.11	48.57	2.84	60.98	165	1130	14	235
14	60.54	40.86	0.06	41.29	154	1250	9	156
15	79.19	48.58	22.50	65.99	246	1145	7	98
16	87.53	90.75	26.16	66.03	571	1235	5	231
17	53.40	37.65	-0.1	31.63	201	1241	17	100
18	97.94	92.83	20.45	87.22	308	1158	7	98
19	89.62	69.72	13.96	69.06	529	1356	10	138
20	100	96.87	19.69	90.70	328	1320	9	31
AVERAGE	83.18	71.37	13.45	60.90	241.9	1534.25	11.4	125.45



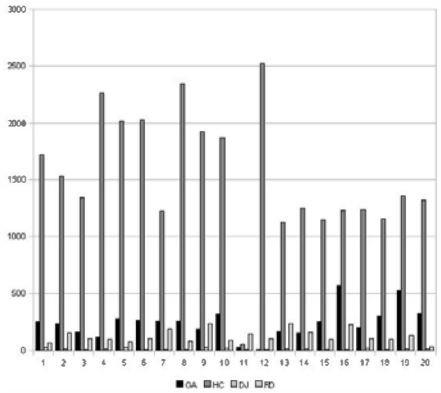


Figure 11. Representation of the 20 experiments. Experimental results (left), fitness values (center) and sizes of the resulting adaptive sequences (right).

ally dependant on the quality of the Dijkstra individual, but examples like 1,2,4, and 9 defy this proposition, since it was the random individual the one that helped the GA to find a solution. This was because the Dijkstra path contained a high level of branching, and this fact probably opened up paths that led to nodes with no connection to the target node, while there was a path that was not the shortest but had less branching. Therefore, the GA probably deleted the Dijkstra solution and continued randomly. This capacity to use the best of both worlds, including the Dijkstra individual as part of the starting population and the use of randomness to go through the search space and look for alternative paths is what makes the GA more consistent, and able to find a good solution through any *ndFSM*. We can also see, that in one of the experiments, number 10, the GA was able to find a good solution, even though the random and Dijkstra individuals scored poorly.

In the vast majority of our experiments, hillclimbing was able to find a solution, on average 71.37% of the times, 12% under the results of the GA, and even in some experiments (number 5, 8 and 16) the hillclimbing method ranks slightly better than the GA. We might therefore think that the hill-climbing solution is as valid as the one produced by the GA but as we can see on the right hand side of Figure 11, the size of the hillclimbing solution is considerably larger than the one of the GA, which makes it less worthy. Note that this methodology was also highly dependent on the capability of

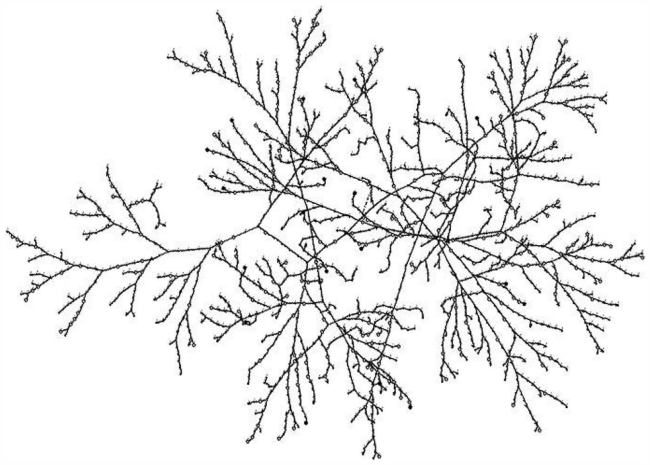


Figure 10. Experiment A, hillclimbing reaches the target 70% of the times.

Table I
TABLE COMPARING SEVERAL GAS WITH DIFFERENT RANDOM COEFFICIENTS AND POPULATIONS (LABELED WITH "M") IN WHICH THIS COEFFICIENT WAS HEREDITARY.

Comparison								
fitness value (%)								
1	0	0.5	(0-0.5)	(0-1)	(0.5-1)	(0-0.5)m	(0-1)m	(0.5-1)m
81.16	83.38	81.49	96.89	60.68	76.90	54.55	70.89	79.1
55.46	44.67	47.01	57.45	44.09	44.03	48.18	59.65	48.56
69.53	65.70	63.46	72.19	67.54	65.32	70.39	68.47	70.33
68.89	78.36	75.22	80.65	75.50	75.11	78.49	77.10	70.50
80.54	86.07	82.08	82.48	81.69	82.48	82.12	80.06	88.12

the Dijkstra individual to find a solution, and in some cases (see experiment number 9) in which the shortest path led to high levels of branching, the hillclimbing method was not able to find an optimal solution.

In order to be able to study the effect of either choosing randomly a node or choosing the best node, we have run several experiments in which we tested different degrees of randomness. The populations were initialized with the values of 1, 0 or 0.5, we also added populations with coefficients in the ranges (0-0.5), (0-1), (0.5-1). Finally, in order to be able to discern which mechanism was better to obtain this coefficient in a new individual, we decided to also test some versions in which the coefficient was not randomly started when a new children was created, but it was a hereditary characteristic, these populations are labeled with a "m" (mixed). The results shown in Table I indicate that in most cases the best population, and generally the one that had a good outcome in the cases that was not the best (the most consistent population), was the one initialized with the range of (0-0.5) and that did not used the hereditary mechanism. This is because having a mostly random population allows to explore a bigger part of the landscape of the solution, this, added to keeping some individuals choosing sometimes the closest node to the target, allows the GA to find a correct solution. Moreover, it also helps the GA, since reproduction will be more useful when crossing complete different individuals, like the ones that will be formed by continuously taking random choices.

V. CONCLUSIONS AND FUTURE WORK

We have created an evolutionary approach based on genetic algorithms, to solve an important problem in the world of testing. We have compared our methodology with other reasonable approaches, and experimental results showed that our approach is the most consistent and the one that found the best solution in most of the cases (specifically, out of 20 runs, there were 2 cases in which both methodologies found a perfect solution, in 15 cases GA outperformed the hillclimbing algorithm and in 3 cases, it was the hillclimbing methodology the one that found the best solution). We envisage several paths for future development. One of them would be to experiment on whether using a GA, in which *islands* of different populations will separately search from a different initial state, and we will consider sub-target states or use different random coefficients and then mix the best individuals from each solution. Another line for future work will be directing mutation towards modifying the subtree with less fitness value. Finally it is necessary to test different types of non-deterministic machines, in order to decide in which cases the GA methodology performs better.

REFERENCES

[1] G. Myers, *The Art of Software Testing*, 2nd ed. John Wiley and Sons, 2004.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] A. Petrenko, N. Yevtushenko, and G. v. Bochmann, "Testing deterministic implementations from their nondeterministic FSM specifications," in *9th IFIP Workshop on Testing of Communicating Systems, IWTCS'96*. Chapman & Hall, 1996, pp. 125–140.

[4] R. Hierons, "Testing from a non-deterministic finite state machine using adaptive state counting," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1330–1342, 2004.

[5] F. Ipate, "Testing against a non-controllable stream X-machine using state counting," *Theoretical Computer Science*, vol. 353, no. 1, pp. 291–316, 2006.

[6] R. Alur, C. Courcoubetis, and M. Yannakakis, "Distinguishing tests for nondeterministic and probabilistic machines," in *27th ACM Symp. on Theory of Computing, STOC'95*. ACM Press, 1995, pp. 363–372.

[7] R. Hierons, "Adaptive testing of a deterministic implementation against a nondeterministic finite state machine," *The computer Journal, Oxford Journals*, vol. 41, no. 5, pp. 349–355, 2008.

[8] ——, "Reaching and distinguishing states of distributed systems," *SIAM Journal on Computing*, vol. 39, no. 8, pp. 3480–3500, 2010.

[9] M. Gromov, N. Yevtushenko, and A. Kolomeets, "On the synthesis of adaptive tests for nondeterministic finite state machines," *Programming and Computer Software*, vol. 34, pp. 322–329, 2008.

[10] M.Harman and P.McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[11] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving transformation sequences using genetic algorithms," in *4th IEEE Int. Workshop on Source Code Analysis and Manipulation, SCAM'04*. IEEE Computer Society Press, 2004, pp. 65–74.

[12] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian, "Computing unique input/ouput sequences using genetic algorithms," in *3rd Int. Workshop on Formal Approaches to Software Testing, FATES'03, LNCS 2931*. Springer, 2004, pp. 169–184.

[13] K. Derderian, R. Hierons, M. Harman, and Q. Guo, "Automated Unique Input Output sequence generation for conformance testing of FSMs," *Computer Journal*, vol. 49, no. 3, pp. 331–344, 2006.

[14] D. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.

[15] P. McMinn, "Search-based software test data generation: a survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.