2013 12th International Conference on Machine Learning and Applications

Learning Finite-State Machines: Conserving Fitness Function Evaluations by Marking Used Transitions

Daniil Chivilikhin
University ITMO
St. Petersburg, Russia
Kronverksky pr., 49
Email: chivdan@rain.ifmo.ru

Vladimir Ulyantsev
University ITMO
St. Petersburg, Russia
Kronverksky pr., 49
Email: ulyantsev@rain.ifmo.ru

***Abstract*—This paper is dedicated to the problem of learning finite-state machines (FSMs), which plays a key role in automata-based programming. Metaheuristic algorithms commonly applied to this problem often use FSM mutations (small changes in the FSM structure) for solution construction. Most of them do not employ the specifics of FSMs in their work. We propose a new simple method for improving performance of these algorithms. The basic idea is to mark those transitions of FSMs that were used during fitness evaluation. Then, if a FSM mutation changes a transition that was not used in fitness evaluation, the fitness function value need not be calculated for the mutated FSM. This observation allows to conserve fitness evaluations, which often have high computational costs. The proposed method has been incorporated into several traditional and recent FSM learning algorithms based on evolutionary strategies, genetic algorithms and ant colony optimization. Experimental results are reported showing that the new method significantly improves performance of two methods based on evolutionary strategies and ant colony optimization.**

***Keywords*-automata, inference, machine learning, optimization**

I. Introduction

In the paradigm of automata-based programming [1], [2] finite-state machines are used to describe behavior of software systems. A software system is described as a set of interacting *automated-controlled objects*. Each automated-controlled object consists of a controlled object and a finite-state machine. The controlled object is characterized by *events* it can generate and *actions* that control it. In response to events supplied by the controlled object the FSM generates output actions, which are relayed to the controlled object, calling its functions or methods. Automata-based programming is particularly efficient for systems with complex behavior, i.e. systems that may react differently to the same input events depending on the history of interactions in the past. An example of such a system is a rather simple alarm clock [1], [3]. The key advantage of automata-based programming over traditional programming paradigms is that automata-based programs can be automatically verified [4] using model checking [5].

One of the major issues in automata-based programming is inferring a FSM that would operate the controlled object in a desired and proper way. Manual construction of FSMs for automata-based programs is a hard task and may well be impossible for systems with complex behavior. A possible way to counter this problem is to use search optimization techniques such as evolutionary computation to automatically learn FSMs with proper behavior. When following the path of learning FSMs with metaheuristic search algorithms, a *fitness function* is introduced. The fitness function is usually a real-valued function defined on the considered set of FSMs. Its values are proportional to the closeness of the behavior of the FSM to the desired one.

There are two major ways to define a fitness function: based on comparing the FSM's behavior to a standard recorded in test examples [3], [6], [7] or based on modeling in some environment [8], [9]. Both of these ways may lead to high computational costs of a single fitness function evaluation, which directly leads to an increased cost of building an optimal FSM. That is why it is important to find ways to decrease the number of needed fitness function evaluations as much as possible.

In this work we propose a way to conserve fitness function evaluations which is applicable to all FSM learning algorithms that use FSM mutations. The method is based on taking into account the specifics of FSMs in learning algorithms. An example of a work in which information about FSM transitions was used in the FSM learning algorithm is [10]. The proposed method was incorporated into an evolution strategy, a genetic algorithm and a recent FSM learning method MuACO*sm* [3], [11] based on ant colony optimization [12]. Experimental results describing the influence of the proposed method on the algorithms' performance are reported.

II. Learning Finite-State Machines With Mutation-Based Metaheuristics

In this paper we concentrate on Mealy finite-state machines. A Mealy FSM is formally defined as a six-tuple $(S, s_0, \Sigma, \Delta, \delta, \lambda)$, where S is a set of states, $s_0 \in S$ is the start state, Σ is a set of input events and Δ is a set of output actions. $\delta : S \times \Sigma \to S$ is the *transitions* function and $\lambda : S \times \Sigma \to \Delta$ is the *actions* function. An example of a FSM with three states, $\Sigma = \{x, !x\}$, $\Delta = \{z1, z2\}$ is shown on Fig. 1 (ignore transition colors for now). In our work we use full transition and output tables to represent FSMs. A transition table stores the next state for each combination of input events and start states. An output table stores the output action for all combinations of input events and start states.

A general statement of the FSM learning problem follows. Let $N_{\text{states}} = |S|$ be the number of states, Σ be the set of input events and Δ be the set of output actions of the target FSM. Let $X_{N_{\text{states}},\Sigma,\Delta}$ be the set of all FSMs with parameters $(N_{\text{states}}, \Sigma, \Delta)$. The fitness function $f\colon X_{N_{\text{states}},\Sigma,\Delta} \to \mathbb{R}$ is defined on the considered set of FSMs. Its values are proportional to the proximity of the FSM's behavior to the desired one. Then, the goal is to find an FSM $A \in X_{N_{\text{states}},\Sigma,\Delta}$ such that its fitness function value $f(A)$ is greater than or equal to some predefined boundary value.

Mutation-based metaheuristics for learning FSMs, e.g. genetic algorithms and evolution strategies, use FSM mutations for solution construction. An FSM mutation is a small change in the FSM structure. The mutation operators we use work as follows.

First, a state s and an input event e are selected uniformly randomly from the sets of all states S and input events Σ, respectively. The first mutation operator changes the target state the transition (s, e) leads to. The new end state is selected uniformly randomly from the set of all states not including the old end state. The second mutation operator works similarly, but changes the output action performed on the selected transition.

III. Conserving Fitness Evaluations

The proposed procedure for conserving fitness function evaluations is incorporated into the process of fitness evaluation itself and consists of two parts. First, when the fitness value of an FSM A is evaluated the set of transitions $T(A)$ used in this process is memorized. Next, when the FSM A is mutated, one of its transitions t is replaced by another transition. Let the mutated FSM be A'. Logically, if transition t was not used during fitness evaluation of FSM A, then $t \notin T(A)$. Therefore, if $t \notin T(A)$, then the modification of this transition will not change the FSM's behavior. Consequently, the fitness function value $f(A')$ of the modified FSM A' will be equal to the fitness value $f(A)$ of the original FSM A. This means that by simply assigning A' the fitness value $f(A)$ instead of calculating $f(A')$ from scratch we will be saving computational resources.

For example, consider the FSM on Fig. 1. Let the start state of the FSM be state "1". Transitions that were made after receiving the sequence of events $(x, !x, !x, x)$ are marked red and bold. The transition that is marked blue and dashed is the one changed by a mutation. The mutated FSM is shown on Fig. 2. If supplied with the same sequence of events, it will use the same transitions as the original FSM.

The theoretical limitations of the proposed idea are straightforward. First, the idea may prove helpful only when mutations cause small changes in the FSM structure. Second, the proposed approach is only applicable for deterministic fitness functions. That is, we cannot use it for fitness functions that use randomization because the fitness function value of the same FSM may differ from one calculation to another.

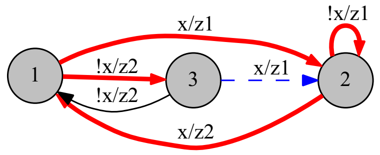

Fig. 1. An example of a FSM. Transitions used during fitness evaluations are marked red and bold, the mutated transition is marked blue and dashed

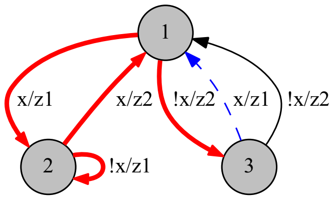

Fig. 2. An example of a mutated FSM. Transitions used during fitness evaluations are marked red and bold, the mutated transition is marked blue and dashed

IV. Example Problems

A. Artificial Ant Problem

The goal in the Artificial Ant problem [9] is to induce a FSM that would optimally control an agent in a certain game. The game is performed on a square toroidal field 32×32 cells. Some cells of the field contain pieces of "food" which are distributed along a certain trail. The trail contains turns and gaps – cells that do not contain food. The Santa Fe trail we use is shown on Fig. 3. This trail contains a total of 89 pieces of food. The Artificial Ant is initially located in the upper left

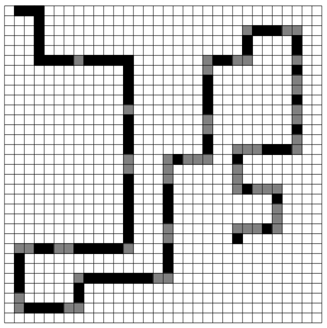
Fig. 3. The Santa Fe trail

cell of the field and is "looking" east. The ant can determine whether the next cell contains a piece of food or not. In our problem statement the ant is given at most 400 steps. On each step it can turn left, turn right of move forward. If the cell to which the ant moves contains a piece of food, the ant "eats" it. The objective is to find a FSM that will allow the ant to eat all 89 pieces of food in the allotted amount of steps. The fitness function has the form:

$$f = n_{\text{food}} + \frac{400 - s_{\text{last}} - 1}{400},$$

where n_{food} is the number of food pieces eaten by the ant and s_{last} is the number of the step on which the last piece of food has been eaten.

In this problem the set of input events Σ consists of two elements: F (the next cell contains a piece of food) and $!F$ (the next cell does not contain a piece of food). There are three possible output actions: L (turn left), R (turn right) and M (move one cell forward).

B. Test-based EFSM induction

An extended finite-state machine (EFSM) is defined as a seven-tuple $(S, s_0, Z, \Sigma, \Delta, \delta, \lambda)$, where S is a set of states, $s_0 \in S$ is the start state, Z is a set of Boolean input variables, Σ is a set of input events, Δ is a set of output actions, $\delta\colon S \times \Sigma \times 2^Z \to S$ is the transitions function and $\lambda\colon S \times \Sigma \times 2^Z \to \Delta^*$ is the actions function. An example of an EFSM is shown on Fig. 4.

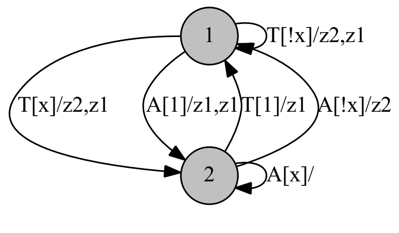

Fig. 4. An example of an extended finite-state machine

The goal in this problem is to build an EFSM with predefined behavior, which is determined by a set of *test examples*, or tests, T. Each i-th test example consists of an input sequence $In[i]$ and a corresponding output sequence $Ans[i]$. Input sequences $In[i]$ consist of pairs of an input event from Σ and a Boolean formula, output sequences $Ans[i]$ consist of elements from the set of outputs Δ.

An EFSM is said to be *consistent* with test $T_i = \{In[i], Ans[i]\}$ if the EFSM produces the sequence $Ans[i]$ on its output if given the sequence $In[i]$ as input. The problem is to find an EFSM with a preset number of states consistent with all test examples from T.

In this problem we treat EFSMs as plain FSMs, which is made possible by utilizing the *smart transition labeling* algorithm desribed in [13] and also by treating combinations of event and Boolean formula as simply an event. The main idea of the labeling algorithm consists in using EFSM *skeletons* – EFSMs, in which transitions are marked with the numbers of necessary output actions instead of the output actions themselves. An EFSM skeleton of the EFSM from Fig. 4 is shown on Fig. 5. The smart transition labeling algorithm takes an EFSM skeleton and the set of test examples as input and produces an EFSM in which transitions marked with output actions in an optimal way with respect to tests T.

The fitness function in this problem is based on Levenshtein string distance [14] or edit distance and is calculated in the following way [13]. First, the EFSM skeleton and the test set T are passed to the smart transition labeling algorithm

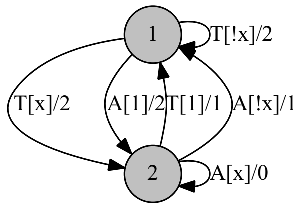

Fig. 5. An example of an EFSM skeleton.

and the optimally labeled EFSM is acquired. Next, each input sequence $In[i]$ is passed to the EFSM and the resulting output sequence $Out[i]$ is memorized. This output sequence is compared with the reference sequence $Ans[i]$ by calculating the following value:

$$f_1 = \frac{1}{|T|} \sum_{i=1}^{|T|} \left(1 - \frac{\mathrm{ED}\left(Out[i], Ans[i]\right)}{\max\left(\mathrm{len}\left(Out[i]\right), \mathrm{len}\left(Ans[i]\right)\right)}\right), \quad (1)$$

where $|T|$ denotes the number of test examples, $\mathrm{len}(s)$ denotes the length of sequence s and $\mathrm{ED}\left(s_1, s_2\right)$ is the edit distance between sequences s_1 and s_2. The final expression for the fitness function takes into account the number of used EFSM transitions $n_{\text{transitions}}$:

$$f = \begin{cases} 10 \cdot f_1 + \dfrac{1}{M}\left(M - n_{\text{transitions}}\right), & \text{if } f_1 < 1 \\ 20 + \dfrac{1}{M}\left(M - n_{\text{transitions}}\right), & \text{if } f_1 = 1, \end{cases} \quad (2)$$

where M is a constant which is guaranteed to be greater than the maximum possible number of EFSM transitions. In our experiments we set $M = 100$. The values of this fitness function are greater for EFSMs that are consistent with all tests and have less transitions and smaller for EFSMs that are not consistent with all tests and have more transitions.

V. Considered Metaheuristic Algorithms

A. $(1, \lambda)$-*Evolutionary Strategy*

In $(1, \lambda)$-ES the population consists of a single individual. On each iteration the algorithm performs λ random mutations of the current solution resulting in λ modified FSMs. The type of mutation is selected uniformly randomly from the set of two possible mutation types. The current solution is replaced with the best newly constructed solution.

B. Genetic Algorithm

We use a classical genetic algorithm with a fixed population size n_{pop}. In addition to the already described mutation operators, in the Artificial Ant problem the genetic algorithm uses a problem-specific crossover operator proposed in [15]. An elitist selection rule is used – the best n_{elit} % individuals of the current generation directly pass to the next generation. Then, the next generation is filled as follows. Two parent individuals are randomly selected from the current generation. Either a mutation or a crossover operator is applied to the selected individuals. The two resulting individuals are added to the next generation.

Two additional mechanisms are used to prevent stagnation: small and large population mutation. In the small population mutation all individuals apart from the best 10 % are mutated. In the large population mutation each individual is either mutated or replaced by a randomly generated one. Small and large population mutations are executed when the best fitness value does not increase during $n_{\text{stag,small}}$ and $n_{\text{stag,large}}$ generations, respectively.

The proposed method here is applied only after mutations. That is, we assume that crossover will most likely affect used FSM transitions, so there is no point in using our method here.

C. Ant Colony Optimization

Another algorithm we consider is the recently proposed FSM learning method [3], [11], [16] called MuACO*sm*, which is based on ant colony optimization [12]. The main idea of the method is to represent the search space in the form of a directed graph. The nodes of the graph are associated with FSMs while edges correspond to FSM mutations. The method uses a new type of an ant colony optimization algorithm to find solutions in this graph.

Let u and v be two nodes of the graph, which is called the *construction graph*. Each edge (u, v) of this graph has an associated pheromone value τ_{uv} and a heuristic information value η_{uv}. The heuristic information is calculated as the absolute difference of fitness values of the start and the end nodes of an edge. Pheromone values are modified by the ants in the process of solution construction.

The algorithm starts off with an empty graph. A random FSM is generated and added to the graph. Then, on each iteration a colony of N_{ants} ants builds solutions. Each ant has a fixed number of n_{stag} steps it can make without an increase in its best fitness value. On each step it selects the next node of the graph to visit. Ants use two rules to select the next node.

Let the ant be located in node u. According to the first rule, which is applied with a certain probability p_{new}, the ant generates a fixed number of N_{mut} mutations of the current solution. The ant then selects the best newly constructed node (associated with the FSM with the largest fitness value) and moves to that node. According to the second rule, which is used with a probability of $1 - p_{\text{new}}$, the next node v is selected from the set of adjacent nodes N_u with a probability p_v calculated according to the formula:

$$p_v = \frac{\tau_{uv}^{\alpha} \cdot \eta_{uv}^{\beta}}{\sum\limits_{w \in N_u} \tau_{uw}^{\alpha} \cdot \eta_{uw}^{\beta}}, \tag{3}$$

where $\eta_{uv} = max\left(\eta_{\text{min}}, f(v) - f(u)\right)$ and $\alpha, \beta \in (0, +\infty)$ are parameters representing the significance of pheromone values and heuristic information, respectively. $\eta_{\text{min}} = 10^{-3}$ is a constant parameter used to ensure that heuristic information values are always positive.

After all ants have finished building solutions, pheromone values are updated for all graph edges using the following specialized elitist update rule. For each graph edge (u, v) we store τ_{uv}^{best} – the best pheromone value that any ant has ever deposited on this edge. First, for each ant path we select a sub-path that spans from the start to the best node in the path and update values of τ_{uv}^{best} on its edges with the fitness value of the best node in the path. Then, for each graph edge (u, v) pheromone values are updated according to the formula:

$$\tau_{uv} = (1 - \rho)\tau_{uv} + \tau_{uv}^{\text{best}}. \tag{4}$$

To prevent stagnation, the whole ant colony is given a maximum of N_{stag} iterations which it can make without an increase in its best fitness value before the algorithm is restarted. For more detailed information about MuACO*sm* and its comparison with different evolutionary computation techniques see [16].

VI. Tuning Algorithm Parameters

To perform a fair comparison of the considered algorithms we tuned the algorithms' parameters using a full factorial design of experiment. For each parameter of each algorithm we empirically selected minimum and maximum levels of parameter values. Each algorithm was allotted a maximum of $t_{\text{tune}} = 10$ hours of tuning time for the Artificial Ant problem and $t_{\text{tune}} = 20$ hours for the test-based EFSM induction problem. Tuning was executed on an *Intel i7 3.4 GHz* personal computer. For the Artificial Ant problem the algorithms were tuned on a problem instance with $N_{\text{states}} = 5$. Each run of each algorithm was limited to 10000 fitness evaluations. For the test-based EFSM induction problem we used a test set for the alarm clock problem consisting of 38 tests with a total length of input sequences equal to 242 and total length of output sequences equal to 195, where the goal is to find an EFSM with $N_{\text{states}} = 4$ and exactly 14 transitions. Here, algorithms were given at most 30000 fitness evaluation for each run.

The tuning process is as follows. The tuning system first evaluates the approximate running time t_{run} of an algorithm run by performing ten runs and calculating the average execution time over these runs. The approximate number of runs n_{runs} that will be executed is then calculated as $n_{\text{runs}} = \frac{t_{\text{tune}}}{t_{\text{run}}}$. We assume that each of the n_{params} parameters of an algorithm will have n_{levels} value levels. Next, to acquire a reasonable assessment for each algorithm configuration (i.e. parameter set) we will have to perform at least n_{repeats} experiments for this configuration. This leads us to the equation $n_{\text{runs}} = n_{\text{levels}}^{n_{\text{params}}} \cdot n_{\text{repeats}}$ and, consequently, we can calculate the number of levels for each parameter as:

$$n_{\text{levels}} = \exp\left(\frac{\ln \frac{n_{\text{runs}}}{n_{\text{repeats}}}}{n_{\text{params}}}\right).$$

Knowing the minimum and maximum parameter values for each parameter we can then create and execute the full factorial experiment design. In order to select the best found configuration, we recorded the mean number of fitness evaluations over all runs for this configuration and the *success rate*. The success rate is defined as the percentage of runs in which the optimal solution was found. Configurations were first sorted by success rate. If several configurations had a 100 % success rates we selected the configuration with the lowest mean number of fitness evaluations.

All scripts, binaries and data that were used for tuning algorithms in this paper are available online at http://rain.ifmo.ru/~chivdan/icmla-2013/tuning.tar.gz.

VII. Experimental Study

A. Results: Artificial Ant Problem

The following experimental setup was used. The number of states N_{states} of the target FSMs was varied from 5 to 20. Each algorithm was run either until finding an optimal solution with a fitness value greater than or equal to 89, or until the algorithm exceeded the alloted number of 30000 fitness evaluations. For each number of states each algorithm was run with and without the fitness evaluation conserving procedure (used transition marking). Each experiment was repeated 1000 times and the success rate (percentage of runs in which an optimal solution was found) was recorded. Experimental results in the form of success rate plots are presented on Fig. 6. Solid lines depict the success rates of algorithms combined with the proposed fitness evaluation conserving method, while dashed lines show success rates of algorithms without the proposed method.

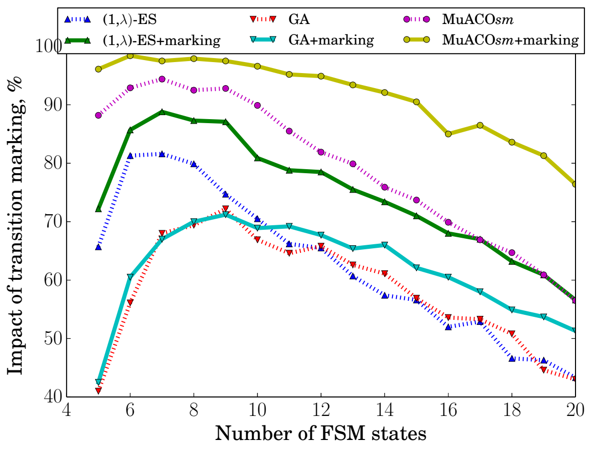

Fig. 6. Success rates of ES, GA and MuACO*sm*, %

The impact of the proposed method of conserving fitness evaluations on algorithm performance is described by the increase of the success rate for each FSM size, which is plotted on Fig 7. One can see that the use of the proposed transition marking technique boosts MuACO*sm* and ES up to 35 %, while results for GA are worse – a maximum of only about 20 %.

To assess the statistical significance of the presented results we applied the ANOVA [17] statistical test. The p-values calculated with ANOVA for fitness distributions of MuACO*sm*, ES and GA are given in Table I. If the p-value for an algorithm is less than 0.05 then the use of transition marking yields a significantly different fitness distribution for

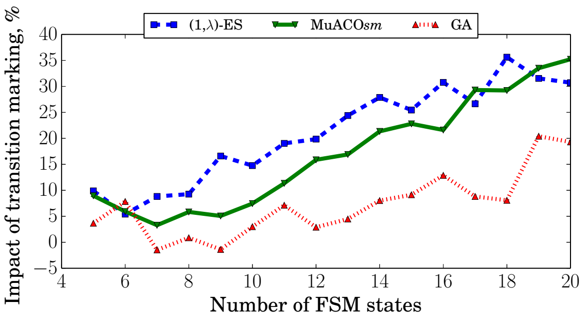

Fig. 7. Impact of using transition marking on ES, GA and MuACO*sm* success rates, %

this algorithm. The provided data indicates that transition marking has a significant impact on MuACO*sm* and ES for all FSM sizes. On the other hand, unfortunately, in most cases the impact of transition marking on the performance of GA is insignificant. Formally speaking, some exclusions from this are for $N_{\text{states}} = 11, 14, 19$ and 20. This was probably caused by the fact that GA uses crossover that changes a lot in the FSMs making it pointless to apply the proposed method of conserving fitness evaluations.

TABLE I
P-VALUES OF SIGNIFICANCE CALCULATED WITH ANOVA

N_{states}	**MuACO*sm***	**GA**	$(1, \lambda)$-**ES**
5	$9.921 \cdot 10^{-10}$	0.311	$1.724 \cdot 10^{-5}$
6	$1.986 \cdot 10^{-8}$	0.296	0.005267
7	$6.329 \cdot 10^{-4}$	0.868	$1.217 \cdot 10^{-7}$
8	$3.047 \cdot 10^{-6}$	0.841	0.0001253
9	$1.492 \cdot 10^{-7}$	0.603	$1.838 \cdot 10^{-9}$
10	$6.006 \cdot 10^{-8}$	0.239	$7.074 \cdot 10^{-7}$
11	$8.303 \cdot 10^{-12}$	0.001	$4.302 \cdot 10^{-9}$
12	$< 2.2 \cdot 10^{-16}$	0.722	$5.666 \cdot 10^{-8}$
13	$< 2.2 \cdot 10^{-16}$	0.249	$5.057 \cdot 10^{-12}$
14	$< 2.2 \cdot 10^{-16}$	0.043	$5.849 \cdot 10^{-10}$
15	$< 2.2 \cdot 10^{-16}$	0.030	$2.607 \cdot 10^{-10}$
16	$< 2.2 \cdot 10^{-16}$	0.141	$1.118 \cdot 10^{-12}$
17	$< 2.2 \cdot 10^{-16}$	0.069	$8.358 \cdot 10^{-9}$
18	$< 2.2 \cdot 10^{-16}$	0.563	$< 2.2 \cdot 10^{-16}$
19	$< 2.2 \cdot 10^{-16}$	0.004	$5.759 \cdot 10^{-11}$
20	$< 2.2 \cdot 10^{-16}$	0.017	$4.662 \cdot 10^{-12}$

B. Results: Test-Based EFSM Induction

In the experiments on test-based EFSM induction we only studied the impact of the proposed method on the MuACO*sm* algorithm. The experimental setup is as follows. The number of EFSM states N_{states} was varied from 4 to 10. For each value of N_{states} 1000 runs were performed. In each run we first generated a random EFSM with two input events, two output actions, two input variables, $4 \times N_{\text{states}}$ transitions and with a length of output sequences no more than two. For each randomly generated EFSM a random test set with a total length of $150 \times N_{\text{states}}$ was generated. Algorithms were run until acquiring an EFSM consistent with all tests or until 50000 fitness evaluations. Experimental results here are in the form of a success rate plot given on Fig. 8 and a plot of transition marking impact on success rate given on Fig. 9.

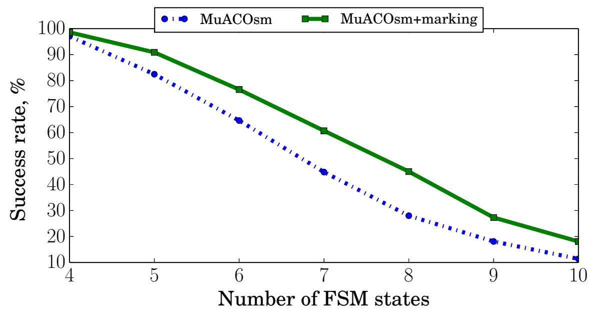

Fig. 8. Success rates of MuACO*sm* on test-based EFSM induction, %

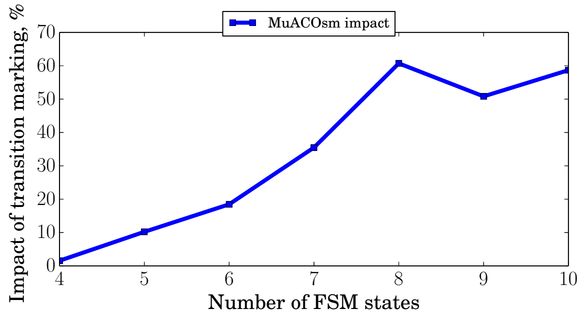

Fig. 9. Impact of using transition marking on MuACO*sm* success rates for test-based EFSM induction, %

As one can see from Fig. 9 the impact of the proposed method on the performance of MuACO*sm* is significant and increases with the growth of N_{states} reaching 60 %. The significance of the differences in performance here was also tested using the ANOVA statistical test. Calculated p-values are given in Table II. These values indicate that the impact of transition marking on the performance of MuACO*sm* is significant.

TABLE II
p-VALUES OF SIGNIFICANCE CALCULATED WITH ANOVA FOR TEST-BASED EFSM INDUCTION

N_{states}	p-value
4	0.019
5	$4.596 \cdot 10^{-8}$
6	$5.329 \cdot 10^{-9}$
7	$6.075 \cdot 10^{-13}$
8	$4.133 \cdot 10^{-15}$
9	$1.938 \cdot 10^{-7}$
10	$3.383 \cdot 10^{-5}$

VIII. Conclusion

A method for conserving fitness evaluations in mutation-based metaheuristic FSM learning algorithms was presented. The proposed method was shown to improve performance of an evolutionary strategy and an ACO-based algorithm MuACO*sm*. Statistical significance of results was evaluated.

Future work includes using MuACO*sm* and other metaheuristics together with the proposed method of conserving fitness evaluations to solve the problem of inducing EFSMs from test examples and temporal logic formulae as in [13]. Results presented in this work indicate that the proposed method of conserving fitness evaluations will most likely significantly increase performance of metaheuristics applied to this problem.

Acknowledgements

Research was supported by the Ministry of Education and Science of Russian Federation in the framework of the federal program "Scientific and scientific-pedagogical personnel of innovative Russia in 2009-2013" (contract 16.740.11.0455, agreement 14.B37.21.0397), University ITMO development program in 2012-2018 and by the University ITMO research project 610455.

References

[1] N. Polykarpova and A. Shalyto, *Automata-based programming*. Piter., 2009, in Russian.
[2] A. Shalyto and N. Tukkel', "Switch technology: An automated approach to developing software for reactive systems," *Programming and Computer Software*, vol. 27, no. 5, 2001.
[3] D. Chivilikhin, V. Ulyantsev, and F. Tsarev, "Test-based extended finite-state machines induction with evolutionary algorithms and ant colony optimization," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, ser. GECCO Companion '12, 2012, pp. 603–606.
[4] S. E. Velder, M. A. Lukin, A. A. Shalyto, and B. R. Yaminov, *Verification of automata-based programs (Verificatsiya avtomatnykh programm)*. Nauka, 2011, in Russian.
[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT press, 1999.
[6] S. Lucas and J. Reynolds, "Learning finite state transducers: Evolution versus heuristic state merging," *IEEE Transactions on Evolutionary Computation.*, vol. 11, no. 3, pp. 308–325, 2007.
[7] A. Alexandrov, A. Sergushichev, S. Kazakov, and F. Tsarev, "Genetic algorithm for induction of finite automata with continuous and discrete output actions," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, ser. GECCO '11, 2011, pp. 775–778.
[8] W. M. Spears and D. F. Gordon, "Evolving finite-state machine strategies for protecting resources," in *Proceedings of the 12th International Symposium on Foundations of Intelligent Systems*, ser. ISMIS '00. London, UK, UK: Springer-Verlag, 2000, pp. 166–175.
[9] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang, "Evolution as a theme in artificial life," *Artificial Life II*, 1991.
[10] S. Lucas and T. Reynolds, "Learning dfa: evolution versus evidence driven state merging," in *Proceedings of the 2003 Congress on Evolutionary Computation. CEC '03*, vol. 1, 2003, pp. 351–358.
[11] D. Chivilikhin and V. Ulyantsev, "Learning finite-state machines with ant colony optimization," in *Proceedings of the 8th international conference on Swarm Intelligence*, ser. ANTS'12, 2012, pp. 268–275.
[12] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.
[13] F. Tsarev and K. Egorov, "Finite state machine induction using genetic algorithm based on testing and model checking," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 759–762.
[14] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.
[15] F. Tsarev and A. Shalyto, "Constructing minimal finite-state machines for the artificial ant problem," in *Proceedings of the 10th International conference on soft calculation and measurement*, ser. SCM'07, vol. 2, 2007, pp. 88–91.
[16] D. Chivilikhin and V. Ulyantsev, "Muacosm: a new mutation-based ant colony optimization algorithm for learning finite-state machines," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, ser. GECCO '13, 2013, pp. 511–518.
[17] R. G. Miller, *Beyond ANOVA: Basics of Applied Statistics (Texts in Statistical Science Series)*. Chapman & Hall/CRC, Jan. 1997. [Online]. Available: http://www.worldcat.org/isbn/0412070111