Learning Finite-State Machines with Classical and Mutation-Based Ant Colony Optimization: Experimental Evaluation

Daniil Chivilikhin
St. Petersburg National Research University of Information Technologies, Mechanics and Optics
Computer Technologies Department
St. Petersburg, Russia
Kronverksky pr., 49
Email: chivdan@rain.ifmo.ru

Vladimir Ulyantsev
St. Petersburg National Research University of Information Technologies, Mechanics and Optics
Computer Technologies Department
St. Petersburg, Russia
Kronverksky pr., 49
Email: ulyantsev@rain.ifmo.ru

***Abstract*—The problem of learning finite-state machines (FSM) is tackled by three Ant Colony Optimization (ACO) algorithms. The first two classical ACO algorithms are based on the classical ACO combinatorial problem reduction, where nodes of the ACO construction graph represent solution components, while full solutions are built by the ants in the process of foraging. The third recently introduced mutation-based ACO algorithm employs another problem mapping, where construction graph nodes represent complete solutions. Here, ants travel between solutions to find the optimal one.**

In this paper we try to take a step back from the mutation-based ACO to find out if classical ACO algorithms can be used for learning FSMs. It was shown that classical ACO algorithms are inefficient for the problem of learning FSMs in comparison to the mutation-based ACO algorithm.

***Keywords*-finite-state machine, automata, induction, inference, machine learning**

I. Introduction

Ant colony optimization [1] is a metaheuristic inspired by the foraging behavior of ants. In this paper we use ACO to tackle the problem of learning FSMs, which plays a key role in automata-based programming [2], [3]. This programming paradigm proposes to use FSMs as key components of software systems. The approach is useful for systems with complex behavior, i.e. systems that can react differently to the same events depending on the history. Examples of such systems are network protocols and control systems. However, manual construction of FSMs for such systems can be hard or even impossible. Therefore, various search optimization techniques are used are used to solve this problem automatically. One of the greatest advantages of automata-based programming is that programs designed using this paradigm can be automatically verified using model checking [4] which is impossible for other types of programs. That is, one can automatically check temporal logic (e.g. *LTL*, *Linear Time Logic*) properties formulated for a program.

ACO uses a special graph called the *construction graph* to build solutions. In order to solve an arbitrary combinatorial problem with ACO, it is typically reduced, or mapped, to another problem in which the goal is to find a maximum or minimum cost path in the construction graph. Nodes of the construction graph represent solution components, while full solutions correspond to paths in the graph. For problem mappings performed in this canonical manner and for certain types of ACO algorithms such as $ACO_{bs,\tau_{\min}}$, convergence in value has been proven. That is, it was shown that $ACO_{bs,\tau_{\min}}$ will find some optimal solution if given enough resources [5].

In this work we show that canonical mapping and classical ACO algorithms with proven convergence are ineffective for the problem of learning FSMs. Another recently introduced ACO-based algorithm MuACO*sm* [6] produces more promising results, though no convergence guarantees currently exist. The benchmark problem we consider is the quite common Artificial Ant problem [7].

II. Learning Finite-State Machines

A finite-state machine is 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 finite-state machine is shown on Fig. 1.

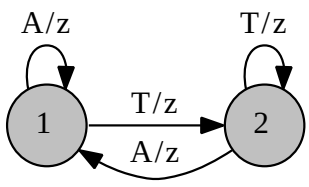

Fig. 1. An example of a finite-state machine with two states, two events (T, A) and a single output action z

Quality evalutation of FSMs is encapsulated into a real-valued *fitness function* f. The closer the FSM's behavior to the desired one, the larger the value of f. Thus, the FSM learning

problem is: given the number of states N_{states}, set of input events Σ and set of actions Δ find a FSM with parameters $(N_{\text{states}}, \Sigma, \Delta)$ with a large enough fitness function value.

The problem of learning FSMs and other types of automata, such as finite-state transducers (FST) and definite finite automata (DFA), is commonly solved using various metaheuristic techniques. For instance, in [8] a random mutation hill climbing evolutionary algorithm is applied to the problem of learning deterministic finite automata, which are somewhat similar to finite-state machines, from sets of labeled strings. The same authors use evolutionary algorithms to learn finite-state transducers from test examples [9]. In [10] a genetic algorithm is applied to learning extended finite-state machines from test examples and *LTL*-formulae. Finite-state machines are learned with an evolutionary algorithm in [11] for the Competition for Resources problem.

The authors of this paper recently introduced a new method of learning FSMs called MuACO*sm*, which is based on ant colony optimization. This new method was compared with a number of evolutionary techniques for learning FSMs on several benchmark problems [6], [12], [13] and proved to be more effective than genetic algorithms and evolutionary strategies. However, no attempt was made to learn or induce finite-state machines using classical ACO yet. In this paper we provide a way to apply classical ant colony optimization algorithms to the problem of learning FSMs and compare our mutation-based ACO method with classical ACO algorithms.

III. Classical ACO for Learning FSMs

A. General Scheme of Classical ACO Algorithms

Classical ACO algorithms, as described in [1], tackle combinatorial problems formalized as $\left(\hat{S}, \hat{f}, \Omega\right)$, where $\hat{S}$ is a set of *candidate solutions*, $\hat{f}$ is the *objective function* and Ω is a set of *constraints* defining feasible candidate solutions. The goal is to find a globally optimal feasible solution s^*.

This combinatorial problem is mapped on another problem defined in the following way. Let $C = \{c_1, \ldots, c_K\}$ be a finite set of *components* and $X = \{x = \langle c_{i_1}, c_{i_2}, \ldots, c_{i_h}, \ldots \rangle, |x| \leq n < +\infty\}$ be a set of *problem states*. Then, the set of feasible candidate solutions $\hat{S}$ is a subset of X. $\tilde{X} \subset X$ is a set of feasible states, i.e. states $x \in X$ that can be completed to a solution satisfying the constraints Ω. The set of feasible solutions is $\tilde{S} = \tilde{X} \cap S$ and $S^* \subset \tilde{S}$ is a non-empty set of optimal feasible solutions. Next, the *construction graph* $G_C = (C, L)$ is a graph with a set of nodes C and a set of connections L which fully connect C. The goal in the mapped problem is to find a maximum cost path in G_C with respect to $\hat{f}$ which satisfies the constraints Ω and corresponds to a feasible solution.

In ACO each connection, or edge (u, v) of the construction graph has an associated *pheromone value* τ_{uv} and can have an associated *heuristic information* η_{uv}. Pheromone values play a role of the colony's long-term memory, while heuristic information represents some apriori knowledge about the problem. Solutions are built by agents called ants, which are probabilistic procedures determining how to add components to the current solution depending on pheromone values and heuristic information. An ACO algorithm typically consists of three steps that are repeated until an optimal solution is found or the resources allotted to the algorithm are depleted. Examples of such stopping criteria include a fixed number of algorithm iterations, a fixed number of fitness evaluations or convergence of a fixed number of ants to the same path.

On the first step called `ConstructAntSolutions` a colony of ants traverse the graph to build solutions. Each ant is placed on some node of the construction graph. The ant adds components to its current solution by traversing the graph until it has built a complete solution. It selects the next node to visit according to some probabilistic rule. Ants continue traversing the graph until each of them has built a complete solution to the problem.

On the second step called `UpdatePheromones` pheromone values on connections L are updated. A particular pheromone value can increase if the edge it is associated with has been traveled by an ant or it can decrease due to pheromone evaporation. The third step called `DaemonActions` is optional. It may be used to perform operations that cannot be executed by individual ants, possibly using domain knowledge.

B. Classical ACO Problem Reduction

In our problem statement defined in the previous section, the set of candidate solutions $\hat{S}$ is the set of all FSMs with parameters $(|S| = N_{\text{states}}, \Sigma, \Delta)$ and the objective function $\hat{f}$ is the fitness function f. The constraints Ω are that a FSM has to be deterministic, that is, it should contain exactly one transition for each start state $i \in S$ and event $e \in \Sigma$. The set of components C is defined as a set of all possible FSM transitions:

$$C = \{t = \langle i, j, e, a \rangle, i, j \in S, e \in \Sigma, a \in \Delta\}, \qquad (1)$$

where $i \in S$ is the transition's start state, $j \in S$ is the end state, $e \in \Sigma$ is an event and $a \in \Delta$ is an output action. Therefore, the construction graph G_C contains exactly $|C| = |S|^2 \cdot |\Sigma| \cdot |\Delta|$ nodes. The set of connections L fully connects the components. An example of the described construction graph is given on Fig. 2. Green connections were already followed by the ant, red connections are forbidden due to constraints and blue connections are allowed for the ant to choose from. If the ant chooses a certain connection it will build the FSM from Fig. 1.

To implement the constraints, the k-th ant uses its internal memory to store its path of traversed components L_t^k. Let the k-th ant be located at node $u \in C$ with a set of adjacent nodes N_u. First, the ant applies the constraints to determine possible nodes it can move to. To do that, it scans its memory along with N_u and forms a set of components $\hat{N}_u$ such that no transition $t \in \hat{N}_u$ has a start state $y \in S$ and event $\varepsilon \in \Sigma$ where $\exists r \in L_t^k \colon r.i = y \wedge r.e = \varepsilon$. The next node $v \in \hat{N}_u$ is

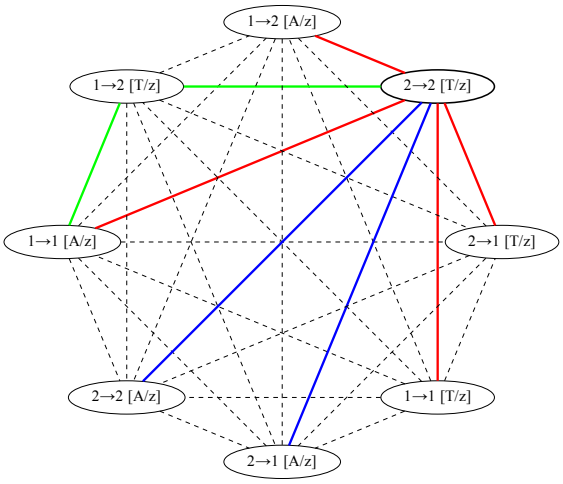

Fig. 2. An example of a classical ACO construction graph for learning FSMs. Green connections were already followed by the ant, red connections are forbidden due to constraints and blue connections are allowed

selected with a probability:

$$p_v = \frac{\tau_{uv}^{\alpha}}{\sum\limits_{w \in \hat{N}_u} \tau_{uw}^{\alpha}}, \tag{2}$$

where $\alpha \in (0, +\infty)$ is a parameter representing the significance of pheromone values. This equation does not take into account heuristic information because there seems to be no meaningful way to define it for the problem of learning FSMs with canonical mapping. From the start, this seems to be a major drawback of the classical ACO approach to learning FSMs.

C. Classical ACO Algorithms Used

We use the Elitist Ant System (EAS) algorithm [14] and the $ACO_{bs,\tau_{\min}}$ algorithm [5]. Both algorithms follow the general scheme described in section III-A. A colony of N_{ants} ants is used to construct solutions. The optional `DaemonActions` procedure is not used in either of the algorithms.

On every iteration during the `ConstructAntSolutions` step each ant traverses the construction graph until it builds a complete solution, i.e. a deterministic FSM. On each step the ant selects the next node of the construction graph from the successor nodes of the current node with respect to constraints. The probability of selecting a particular node is calculated using equation (2).

The `UpdatePheromones` procedure in the considered algorithms is as follows. In the EAS the best-so-far solution s^{best} deposits pheromone along with all solutions on the current iteration. Pheromone values are updated according to the formula:

$$\tau_{uv} = (1-\rho)\tau_{uv} + \sum_{k=1}^{N_{\text{ants}}} \Delta\tau_{uv}^{k} + w_{\text{elit}} \cdot \Delta\tau_{uv}^{\text{best}}, \tag{3}$$

where $w_{\text{elit}} \in [0, 1]$ defines the weight of the elitist solution, $\rho \in [0, 1]$ is the pheromone evaporation rate,

$$\Delta\tau_{uv}^{k} = \begin{cases} f(s_k), (u,v) \in L_t^k \\ 0, \text{otherwise} \end{cases} \tag{4}$$

and

$$\Delta\tau_{uv}^{\text{best}} = \begin{cases} f(s^{\text{best}}), (u,v) \in L_t^{\text{best}} \\ 0, \text{otherwise} \end{cases}, \tag{5}$$

where s_k is the k-th solution on the current iteration and L_t^{best} is the set of connections traversed by the best-so-far ant. Pheromone update in $ACO_{bs,\tau_{\min}}$ is similar, but only the best-so-far solution deposits pheromone with $w_{\text{elit}} = 1$. Furthermore, pheromone values in both algorithms are kept above a lower bound $\tau_{\min}$, which was chosen to have a value of 0.1 in all performed experiments.

IV. Mutation-based ACO for Learning FSMs

The mutation-based ACO, which was first introduced in [6] for learning FSMs, uses a different type of problem mapping, even though it resembles classical ACO in the way solutions are constructed. In the mutation-based ACO algorithm for learning FSMs called MuACO*sm*, nodes of the construction graph G represent complete solutions instead of solution components. Edges of the construction graph correspond to *mutations* of FSMs – small changes in the FSM structure. Two FSM mutation types are used: for a random transition in the FSM we change either the action performed on this transition or the state it leads to. The ants travel between solutions by mutating FSMs associated with construction graph nodes.

More formally, nodes u and v of G can be connected with an edge (u, v) if FSM A_2 associated with v can be acquired from FSM A_1 associated with u using one mutation operation. Consequently, any FSM can be transformed into any other FSM with certain mutations. Therefore, a fully constructed graph G will contain a path of mutations connecting any two FSMs. However, full construction of G is infeasible since it would effectively lead to performing a brute force search. An example of the construction graph used in MuACO*sm* is shown on Fig. 3. Each edge of the graph is marked with a mutation written in the following notation:

- Tr: $(s_1, e) \to s_2$ means that the end state of transition from state s_1 with event e has to be set to state s_2;
- Out: $(s_1, e) \to z$ means that the output action of transition from state s_1 with event e has to be set to z.

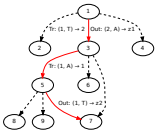

Fig. 3. An example of a small part of the construction graph used in MuACO*sm*

The construction graph is initially empty. First, a random initial FSM with a fixed number of states N_{states} is generated. This is done by choosing a random action and

destination state for each FSM state and input event. The random solution is added to the construction graph and becomes its first node. Then, `ConstructAntSolutions` and `UpdatePheromones` procedures are executed until an optimal solution is found or computational resources are depleted.

On the `ConstructAntSolutions` step, all ants are first placed on the node associated with the best-so-far solution. Each ant, being located at node u, uniformly randomly selects one of the following rules for determining the next node.

1) **Construct new solutions**. The ant constructs exactly N_{mut} mutations of its current solution associated with node u. Each mutated solution is added to the graph G, if not already present there: a new node t is constructed and connected to u with edge (u, t). The ant selects the best constructed node, i.e. node associated with a FSM with the largest fitness function value, and moves to that node.
2) **Probabilistic selection**. The next node v is selected from the set of adjacent nodes N_u according to the formula:

$$p_v = \frac{\tau_{uv}^{\alpha} \cdot \eta_{uv}^{\beta}}{\sum\limits_{w \in N_u} \tau_{uw}^{\alpha} \cdot \eta_{uw}^{\beta}}, \tag{6}$$

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.

Termination conditions for individual ants and the whole colony are defined in the following way. Each ant is allowed to make n_{stag} steps without an increase of its best fitness value before it is stopped. Similarly, the colony is given N_{stag} iterations to run without an increase in the best fitness value before the algorithm is restarted. On each colony iteration, after all ants have finished building solutions, pheromone values are updated by the `UpdatePheromones` procedure as follows. 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. Then, for each graph edge (u, v) pheromone values are updated according to the formula:

$$\tau_{uv} = (1 - \rho)\tau_{uv} + \tau_{uv}^{\text{best}}. \tag{7}$$

For more detailed information about MuACO*sm* and its comparison with different evolutionary computation techniques see [13].

V. Case Study: Artificial Ant Problem

A. Artificial Ant Problem Description

In the Artificial Ant problem [7] the goal is to build an FSM optimally controlling an agent in a game performed on square toroidal field divided into 32×32 cells. In order to avoid confusing this artificial ant with ants used in ACO algorithms, we will call it the agent. The field contains 89 pellets of food, or apples, distributed along a certain trail. The agent is initially located in the leftmost upper cell and is "looking" east. It can determine whether the next cell contains a piece of food (event F) or not (event $!F$). In this work we use the Santa Fe field shown on Fig. 4 instead of the John Muir field originally used in [7] and considered in [6]. Black cells contain food, white cells are empty and gray cells depict the optimal trail.

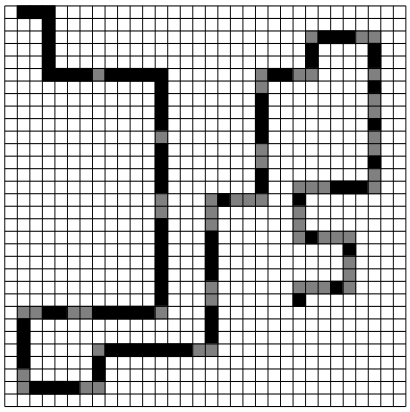

Fig. 4. The Santa Fe field

The goal is to build a FSM controlling the agent that will allow it to eat all 89 pellets of food in no more than s_{max} steps. On each step the agent can turn left (action L), turn right (action R) or move forward (action M). If the cell to which the agent moves contains a pellet of food, the agent eats it. We use a fitness function that takes into account both the number of eaten pellets n_{food} and the number of the step s_{last} on which the last pellet was eaten:

$$f = n_{\text{food}} + \frac{s_{\text{max}} - s_{\text{last}} - 1}{s_{\text{max}}}. \tag{8}$$

B. Tuning Algorithm Parameters

Parameter values for the EAS, $ACO_{bs,\tau_{\text{min}}}$ and MuACO*sm* were selected by performing a full factorial experiment considering the Artificial Ant problem with $s_{\text{max}} = 600$ and FSMs consisting of five states. Tuning was performed on a personal computer with an *Intel i7 3.4 GHz* processor. For tuning each algorithm we selected certain levels of each parameter's value and executed the algorithm with all parameter value combinations. Each algorithm was run 50 times on each parameter value combination. Each run was limited to a maximum of 30000 fitness evaluations. To assess the successfulness of the algorithms we used the *success rate*, which is defined as simply the ratio of experimental runs in which an optimal solution with a fitness value greater than or equal to 89 was found. Parameter value sets were compared according to the success rate over these 50 experiments.

After performing the full factorial experiments for each algorithm we selected the parameter value set that yield the highest success rate. The total tuning times for tuning are: 18392 sec. for $ACO_{bs,\tau_{\text{min}}}$, 66346 sec. for EAS and 11174 sec. for MuACO*sm*. This means that the classical ACO algorithms

were given more time for tuning which theoretically puts them in a more favorable position than MuACO*sm* which used the least time for tuning. Parameter value levels used in the full factorial experiments are listed in Table I. Parameter values that were selected as the best are highlighted in bold.

TABLE I
FULL FACTORIAL DESIGN OF EXPERIMENTAL SETUP: PARAMETER VALUE LEVELS FOR ALL ALGORITHMS. BEST PARAMETER VALUES ARE HIGHLIGHTED IN BOLD

Parameter	$ACO_{bs,\tau_{\min}}$	EAS	MuACO*sm*
ρ	**0.1**, 0.5, 0.9	0.1, **0.5**, 0.9	**0.1**, 0.5, 0.9
N_{ants}	1, **5**, 10	**1**, 5, 10	**5**, 10
α	1, 3, **5**	1, **3**, 5	**1**, 5
β	–	–	1, **5**
w_{elit}	–	0.1, 0.5, **0.9**	–
n_{stag}	–	–	5, **10**, 20
N_{stag}	–	–	5, 10, **20**
N_{mut}	–	–	5, **10**
p_{new}	–	–	**0.5**

C. Experiments

Benchmarking experiments were performed for FSMs with $N_{\text{states}} \in [5, 10]$ and $s_{\max} = 400$. We intentionally used a lower value of $s_{\max}$ so that we test the algorithms on problems that are harder than the one we tuned them on. Each experiment was run for a maximum of 30000 fitness evaluations and was repeated 100 times. For each number of FSM states we recorded the mean running time, mean number of fitness evaluations and success rate over all experimental runs. Fig. 5 shows the success rate of the compared algorithms, Fig. 6 shows mean numbers of used fitness evaluations and Fig. 7 shows the mean execution time using a logarithmic scale.

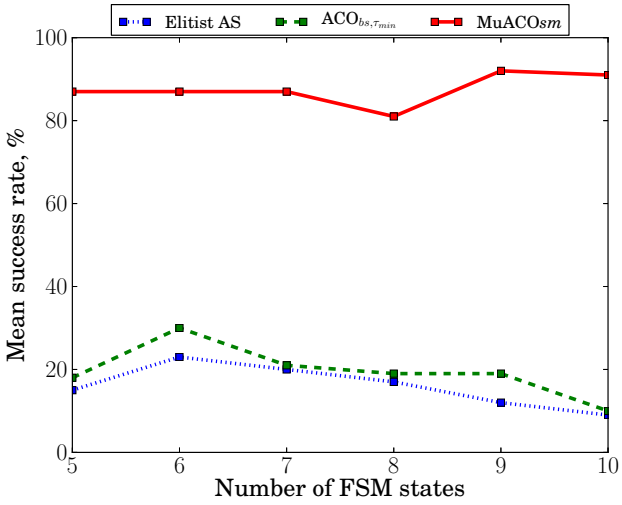

Fig. 5. Success rate of experimental runs for EAS, $ACO_{bs,\tau_{\min}}$ and MuACO*sm*

To check the statistical significance of the acquired results we used the ANOVA [15] statistical test. The test was applied to the $ACO_{bs,\tau_{\min}}$ and MuACO*sm* algorithms. This test calculated the probability that the two algorithms yield the same success rate. ANOVA was run for each value of N_{states}. The resulting p-values of having similar success rates were less than 10^{-4} for each number of FSM states, which means that the difference in the algorithms' performance is statistically significant.

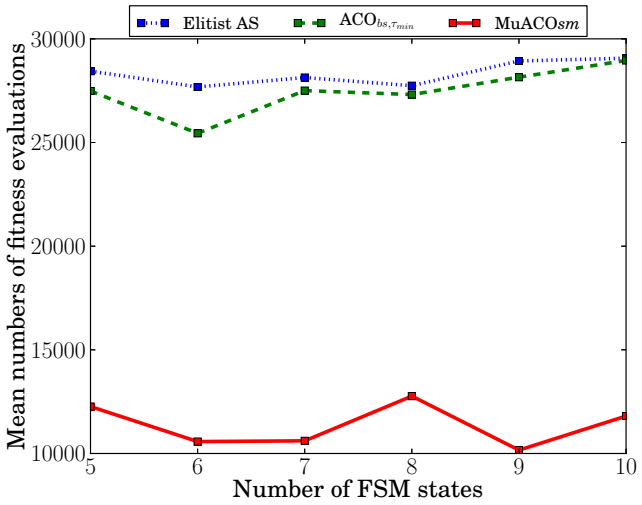

Fig. 6. Mean numbers of fitness evaluations in experimental runs for EAS, $ACO_{bs,\tau_{\min}}$ and MuACO*sm*

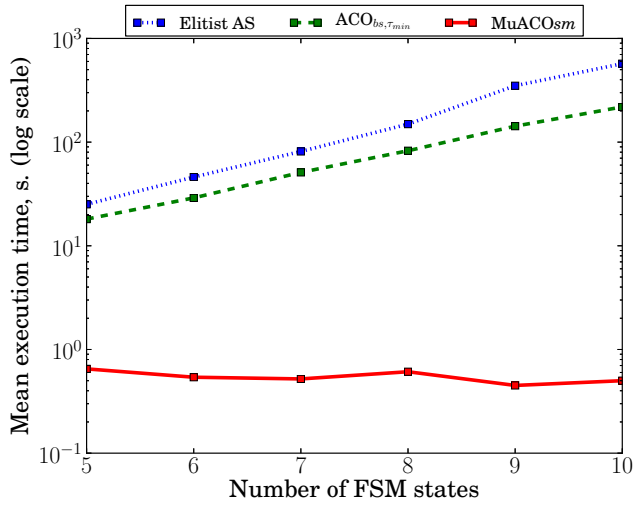

Fig. 7. Mean execution time of experimental runs for EAS, $ACO_{bs,\tau_{\min}}$ and MuACO*sm*

VI. DISCUSSION

Presented experimental results demonstrate that the mutation-based ACO is more efficient than classical ACO in all cases of the considered problem in terms of execution time, success rate and number of required fitness evaluations. Perhaps this was to be expected, since the number of nodes in the classical algorithms' construction graphs increases as N^2_{states} and the number of connections increases as N^4_{states}. Consequently, classical algorithms use up more computational resources for performing walks on the graph than for computing fitness function values. On the contrary, MuACO*sm* mainly uses computational resources for fitness evaluation.

This is demonstrated by the fact that, as can be seen on Fig. 7, mean execution times of both EAS and $ACO_{bs,\tau_{\min}}$ increase exponentially with the increase of the number of FSM states. The mean execution time of MuACO*sm* stays almost constant for all values of N_{states} and seems to depend only on the number of fitness evaluations, which can be derived from comparing MuACO*sm* plots on Fig. 6 and Fig. 7.

To further illustrate this, for each run we calculated the percentage of the execution time which was used for fitness evaluation. The resulting values are presented in Table II. These values allow us to say that $ACO_{bs,\tau_{\min}}$ uses a significant amount of time for its internal computations: almost 97% for FSMs with five states and about 99% for FSMs with ten states. On the contrary, MuACO*sm* uses only about half of time for internal computations. Therefore, if the algorithms were allotted the same computational time instead of the fitness evaluation number limit, classical algorithms would probably perform even worse. Futhermore, when the number of FSM states was increased from five to ten, the fitness time percentage of $ACO_{bs,\tau_{\min}}$ increased almost by a factor of ten, while the fitness time percentage of MuACO*sm* only increased by about 1.2.

TABLE II
PERCENTAGE OF ALGORITHM EXECUTION TIME USED FOR FITNESS EVALUATION

N_{states}	**ACO**$_{bs,\tau_{\min}}$	**MuACO*sm***
5	3.02 %	49.66 %
10	0.34 %	61.4 %

VII. CONCLUSION

Classical and mutation-based ACO algorithms for learning FSMs were presented. It was experimentally shown that classical ACO algorithms based on random walks on construction graphs that consist of solution components are inefficient for the problem of learning FSMs. On the contrary, the mutation-based algorithm MuACO*sm* performed significantly better, which is demonstrated by both execution data such as mean numbers of fitness evaluations and execution time, as well as by a statistical significance test. This is rather unfortunate, since classical algorithms were proven to converge in value, while devising such a proof for the mutation-based algorithm remains a challenging task for us.

Future work includes futher development of the mutation-based MuACO*sm* algorithm to cope with more complex problems such as inferring FSMs from test examples and *LTL*-formulae, as well as devising some kind of a convergence proof for MuACO*sm*.

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] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.
[2] N. Polykarpova and A. Shalyto, *Automata-based programming*. Piter., 2009, in Russian.
[3] A. Shalyto and N. Tukkel', "Switch technology: An automated approach to developing software for reactive systems," *Programming and Computer Software*, vol. 27, no. 5, pp. 260–276, 2001.
[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] T. Stützle and M. Dorigo, "A short convergence proof for a class of ant colony optimization algorithms," *IEEE Transactions on Evolutionary Computation*, pp. 358–365, 2002.
[6] 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. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 268–275. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32650-9_27
[7] 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.
[8] 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.
[9] 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.
[10] 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. [Online]. Available: http://doi.acm.org/10.1145/2001858.2002085
[11] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=646359.690248
[12] 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. New York, NY, USA: ACM, 2012, pp. 603–606. [Online]. Available: http://doi.acm.org/10.1145/2330784.2330883
[13] 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. New York, NY, USA: ACM, 2013, pp. 511–518. [Online]. Available: http://doi.acm.org/10.1145/2463372.2463440
[14] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 26, no. 1, pp. 29–41, 1996.
[15] 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