MuACOsm – A New Mutation-Based Ant Colony Optimization Algorithm for Learning Finite-State Machines

Daniil Chivilikhin
Saint Petersburg National Research University of Information Technologies, Mechanics and Optics
Kronverkskiy pr., 49
St. Petersburg, Russia
chivilikhin.daniil@gmail.com

Vladimir Ulyantsev
Saint Petersburg National Research University of Information Technologies, Mechanics and Optics
Kronverkskiy pr., 49
St. Petersburg, Russia
ulyantsev@rain.ifmo.ru

ABSTRACT

In this paper we present MuACO*sm* – a new method of learning Finite-State Machines (FSM) based on Ant Colony Optimization (ACO) and a graph representation of the search space. The input data is a set of events, a set of actions and the number of states in the target FSM. The goal is to maximize the given fitness function, which is defined on the set of all FSMs with given parameters. The new algorithm is compared with evolutionary algorithms and a genetic programming related approach on the well-known Artificial Ant problem.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.6.e [**Computing Methodologies**]: Induction

General Terms

Algorithms, Experimentation

Keywords

finite-state machine, learning, induction, ant colony optimization

1. INTRODUCTION

In the area of search-based software engineering [10, 16, 15] search optimization techniques are used for automatic program generation. The most common algorithms in use are various evolutionary algorithms. Along with LISP S-expressions and program trees, one of the possible program representations is a finite-state machine.

Automata-based programming [27] is a relatively new programming paradigm that uses FSMs as key components of software systems. A software system in this paradigm consists of a finite-state machine and an automated-controlled object. The FSM receives input events from the environment. Upon receiving an event the FSM makes a transition to one of its states and, possibly, produces a sequence of output actions. These output actions control the automated-controlled object. Although a complex software system may contain many automated-controlled objects, in this work we focus on software systems with only one such object.

The problem of inducing FSMs and other automata types received a significant amount of research over the years. In [29], Spears and Gordon used evolutionary strategies to evolve FSM controllers for the Competition for Resources problem. In [23], Lucas and Reynolds evolved finite-state transducers from test examples with a simple random mutation hill climbing algorithm (RMHC).

Some interesting results were achieved in evolving deterministic finite automata (DFA). A DFA is basically an FSM without output actions, but with two disjoint sets of states instead of one – rejecting states and accepting states. DFA are most commonly used in language recognition problems.

In [24], Lucas and Reynolds compared an evolutionary strategy (ES) with the deterministic Evidence-Driven State Merging algorithm (EDSM) [21] on the problem of inducing DFA from samples of labeled data. It was found that the ES is more effective than the EDSM when the number of states is relatively small.

Later, in [14], Gomez introduced a hybrid method combining incremental learning and evolution for the problem of DFA induction. The main idea of this method is to sort the examples from the training set by length in ascending order. The training set is then divided into a number of blocks. The algorithm first tries to build a DFA consistent only with the first block and then consecutively adds new blocks.

Over the last few decades, many bio-inspired metaheuristic techniques were developed, evolutionary computation being only one of them, for example:

- Ant Colony Optimization (ACO) [11, 13, 30, 12];
- Particle Swarm Optimization (PSO) [18, 28];
- Artificial Bee Colony Algorithm [25, 26];
- Firefly Algorithm [34].

To our knowledge, none of these techniques have ever been applied to FSM induction. In paper [7] we proposed an algorithm for FSM induction which is based on an Ant Colony Optimization algorithm – the MuACO*sm* algorithm. The performance of the new algorithm has already been compared with RMHC, ES and GA on a couple of benchmark

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

problems [7, 8], including test-based extended finite-state machine induction and the Artificial Ant problem [17, 20] with the John Muir trail.

Next, the authors of [5] compared MuACO*sm* with a GA and a (μ, λ)-ES on the problem of inducing FSMs for controlling an unmanned aircraft. They found that MuACO*sm* allowed to build good FSMs faster than the other algorithms, including the GA originally used for this problem in [2].

In this work we present a new improved version of our algorithm and an extensive experimental evaluation on a benchmark problem – the Artificial Ant problem – with the more common Santa Fe trail.

We consider the problem of learning FSMs in the most general way. The target FSM is defined by the maximum number of accessible states N, a set of events Σ and a set of actions Δ. The target model problem that the FSM has to solve is formalized with the use of a real-valued fitness function f which is defined on the set of all finite-state machines with parameters (N, Σ, Δ). The goal is to find an FSM A with a value of the fitness function f not less than some predefined boundary value f_b:

$$f(A) \geq f_b.$$

2. 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 *transition* function and $\lambda : S \times \Sigma \to \Delta$ is the *action* function.

An example of an FSM is shown on Fig. 1. Each transition is marked with an event (before the slash) and an action (after the slash). The start state is state 1.

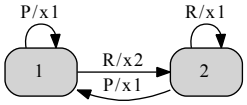

Figure 1: An example of an FSM with two states, event set $\Sigma = \{P, R\}$, and actions set $\Delta = \{x1, x2\}$

A mutation of an FSM is a rather small change in its structure. For example, a mutation can change a transition's output action or destination state. In this work we consider two following FSM mutation types.

- **Change transition end state**. For a random transition in the FSM, the transition's end state is set to another state selected uniformly randomly from the set of all states S.
- **Change transition action**. For a random transition in the FSM, the transition's output action is set to another action selected uniformly randomly from the set of actions Δ.

In our algorithm we use a rather straightforward representation of FSMs – full transition tables. The transition function and action function are stored in the form of tables. An end state and an action are stored for each state and event combination. The start state of all FSMs is always the first state, so it need not be stored. Table 1 demonstrates a representation of the example FSM shown on Fig. 1. A cell of the table should be read like "(end state, action)".

Table 1: A representation of the FSM on Fig. 1

	Event	
State	P	R
1	$(1, x1)$	$(2, x2)$
2	$(1, x1)$	$(2, x1)$

3. ACO OVERVIEW

Ant Colony Optimization algorithms are a family of metaheuristics inspired by the foraging behaviour of real ants. The first ACO algorithm called Ant System was proposed by Marco Dorigo in 1991 [11, 13] and was used to tackle the traveling salesman problem. Since then, a number of ACO models were developed by many researchers, some of them being the Ant Colony System [12], the MAX MIN Ant System [30] and the Rank-based Ant System [4]. These algorithms were applied to a number of combinatorial optimization problems, including:

- sequential ordering problem;
- knapsack problem;
- bin packing;
- classification rule mining;
- quadratic assignment problem.

In order to apply ACO to some combinatorial problem, one must devise a graph representation of the search space. In ACO solutions are built by a colony (set) of artificial ants which use a stochastic strategy. Solutions can be represented either as paths in a graph called the *construction graph*, or simply by graph nodes. Each edge (u, v) of the graph (u and v are nodes of the graph) has an assigned *pheromone value* τ_{uv} and can also have an associated *heuristic information* η_{uv}. The pheromone values are modified by the ants in the process of solution construction, while the heuristic information is assigned initially and is not changed.

In general, an ACO algorithm consists of three operations which are repeated until a viable solution is found or a stop criterion is met.

1. `ConstructSolutions`. Each ant explores the graph following a certain path. It chooses the next edge to visit according to the pheromone value and heuristic information of this edge. When an edge has been selected, the ant appends it to its path and moves to the next node. Commonly, the ants continue exploring the graph until each of them has built a complete solution to the problem.

2. `UpdatePheromones`. Pheromone values on all graph edges are modified. 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 amount of pheromone that each ant deposits on a graph edge depends on the quality of the solution built by this ant, which is measured by the fitness function value of this solution.

3. `DaemonActions` (Optional). Some procedure is executed performing actions that cannot be done by individual ants. An example of such a procedure is local optimization.

When we apply ACO to FSM generation we have to deal with huge graphs, sometimes consisting of several millions of nodes. To deal with this, we apply a variation of the *expansion technique* from [1] – we limit the lengths of the ant paths to reduce the size of the graph we store in memory.

4. MUTATION-BASED ACO FOR LEARNING FINITE-STATE MACHINES

In this section we provide a full description of the new algorithm. First we describe the representation of the search space. Second, some specific features of MuACO*sm* are explained. Next, all ACO steps of our algorithm are detailed. Finally, we review the most significant differences between the proposed algorithm and its previous version described in [7, 8].

4.1 Search Space Representation

The search space – a set of all FSMs with specified parameters (N, Σ, Δ) – is represented in the form of a directed graph G with the following properties.

- The nodes of G are associated with FSMs.
- Let u be a node associated with FSM A_1 and v be a node associated with FSM A_2. If machine A_2 lays within one mutation from A_1 then G contains edges $u \to v$ and $v \to u$. Otherwise, nodes u and v are not connected with an edge, except for the case discussed further in section 4.8.
- Therefore, for each pair of FSMs A_1 and A_2 and the corresponding pair of nodes u and v, there exists a path in G from u to v and also from v to u.

An example of a small part of the search space is shown on Fig. 2. Circles represent the nodes associated with FSMs. The meaning of solid and dashed edges will be explained in section 4.5.2. Each edge is marked with a mutation written in the following notation.

- Transitions function mutation:

$$T\colon \text{(state, event)} \to \text{“new end state”}$$

- Actions function mutation:

$$A\colon \text{(state, event)} \to \text{“new action”}$$

4.2 MuACOsm Specif cs

Although the general scheme of MuACO*sm* complies with the classical ACO algorithm, there are some major differences. Firstly, due to the nature of the problem, the construction graph we use differs significantly from construction graphs used in other ACO applications. Commonly, the edges and nodes of the graph represent solution *components*. Full solutions are built by the ants in the process of foraging. On the opposite, in our case, nodes of the graph represent *complete* solutions themselves. The ants travel between solutions, which are actually built by an external local search procedure – FSM mutation.

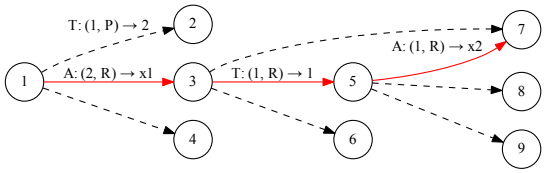

Figure 2: An example of a small part of the search space.

Secondly, our construction graphs can be extremely large – up to several millions of nodes. Such graphs cannot be fully built or stored in memory. In fact, if we were to build a full construction graph we would effectively perform a brute force search. To counter this problem, the algorithm starts off with only one node in the construction graph – a randomly generated solution. When the ants traverse the graph, they determine points in the search space where new solutions should be generated by the local search procedure.

4.3 Initial Solution Generation

In the beginning of the algorithm a random initial solution is generated to become the root of the construction graph. All first-generation ants will start building paths from the root of the graph. Although generation of random initial solutions is the simplest and most straightforward way, it is not the most efficient one.

If the initial solution has a low fitness value, MuACO*sm* can spend several extra iterations while escaping the low-fitness area of the search space. Another way would be to generate a number of random solutions and pick the best one. However, we found to be most efficient the use of a simple random mutation hill climber for a small number of fitness evaluations.

4.4 Heuristic Information

As discussed before in Section 3, each edge of the construction graph can have an associated value called heuristic information. Consider an edge $(u \to v)$, where u and v are nodes of the construction graph G. The heuristic information η_{uv} is calculated as follows:

$$\eta_{uv} = max\left(\eta_{\min}, f(v) - f(u)\right),$$

where $\eta_{\min}$ is a small positive constant value. The $\eta_{\min}$ lower bound ensures that the heuristic information is always positive.

4.5 Path Construction

The path construction procedure consists of two phases. In the first phase a number of start nodes are selected from among all the graph nodes. In the second phase, the ants, starting from selected nodes, traverse the graph.

4.5.1 Start nodes selection

We have experimented with several strategies for start nodes selection, including starting the ants from nodes of the best-so-far ant path and starting the ants from nodes selected with the “roulette wheel” algorithm [3] from among all the nodes in the graph. However, the most efficient strategy

we found was to launch all the ants from the node associated with the best-so-far solution.

4.5.2 Next node selection

Let the artificial ant be located in a node u associated with FSM A. If this node has adjacent edges, then the ant selects the next node v to visit according to the rules discussed below. If node u does not have any children, then the next node is always selected using the first rule.

1. **Expansion.** With a probability of p_{new} the ant attempts to construct new edges of the graph by making N_{mut} mutations of FSM A. The procedure of processing a single mutation of machine A is as follows:
   - construct a mutated FSM A_{mut};
   - find a node t in graph G associated with A_{mut}; if G does not contain such a node, construct a new node and associate it with A_{mut};
   - add an edge (u, t) to G.

   After all N_{mut} mutations have been performed, the ant selects the best newly constructed node v and moves to that node.
2. **ACO path selection.** With a probability of $(1 - p_{\text{new}})$ the ant stochastically selects the next node from the existing successors set N_u of node u. Node v is selected with a probability defined by the classical ACO formula:

$$p_{uv} = \frac{\tau_{uv}^{\alpha} \cdot \eta_{uv}^{\beta}}{\sum_{w \in N_u} \tau_{uw}^{\alpha} \cdot \eta_{uw}^{\beta}},$$

   where $v \in N_u$ and $\alpha, \beta \in [0, 1]$.

Solid edges on Fig. 2 mark a possible path of an ant while the dashed edges represent those mutations that were made by some ant but were not followed by any ant.

4.6 Colony Strategies – Managing the Ants

While the ant's path selection rule defines the strategy of individual ants, there is also a strategy that controls the whole colony. This strategy defines when and how each ant is launched. The two common ant colony strategies are discussed below.

- **Consecutive ant colony.** All ants of the colony are launched consecutively. Each ant runs until it decides to stop, then the next ant is launched and so on.
- **Parallel ant colony.** Here, ants take turns to make one move each until all the ants decide to stop. In most ACO applications this strategy is not any different from the consecutive strategy, because pheromone values are updated after all ants are done traversing the graph. However, it is not the case in our approach, because the ants *modify* the construction graph while traversing it.

Regardless of the concrete strategy, the colony uses the following mechanisms to avoid stagnation:

- each ant in the colony is given at most n_{stag} steps to make without an increase in its best fitness value; when the ant exceeds this number, it is stopped;
- the whole colony of artificial ants is given at most N_{stag} iterations to run without an increase in the best fitness value; after this number of iterations is exceeded, the algorithm is restarted.

4.7 Pheromone Update

There are several strategies of pheromone update described in the literature, including the Ant System pheromone update, elitist pheromone update and rank-based pheromone update. However, for the problem of FSM induction we have devised a new pheromone update rule we call the *global elitist min-bound pheromone update* based on the MAX MIN rule [30] and the elitist rule [11], that fits our problem better than any other strategy.

For each graph edge (u, v) we store τ_{uv}^{best} – the best pheromone value that any ant has ever deposited on edge (u, v). For each ant path, a sub-path is selected that spans from the start of the path to the best node in the path. The values of τ_{uv}^{best} are updated for all edges along this sub-path. Next, for each graph edge (u, v), the pheromone value is updated according to the formula:

$$\tau_{uv} = max(\tau_{\min}, \rho\tau_{uv} + \tau_{uv}^{\text{best}}),$$

where $\rho \in [0, 1]$ is the evaporation rate and $\tau_{\min}$ is an empirically selected minimum pheromone bound (we use a value of 0.001).

4.8 Daemon Actions

As discussed in Section 4.5, an ant does not always select a node with a higher fitness value than its current one. This feature allows the algorithm to efficiently escape local maxima. However, such behaviour can decrease performance of other ants, forcing them to reach better solutions through a series of worse ones. To counter this, the "*nondecreasing paths*" algorithm was introduced. For each ant's path we select a fitness-nondecreasing sequence of nodes – sequence $(n_0 \dots n_l)$ such that:

$$\forall i \in [0 \dots l-1], \forall j > i : f(A_{n_j}) \geq f(A_{n_i}).$$

From nodes of this sequence we can build a path, connecting its nodes with corresponding mutations. Then, on the stage of pheromone update, pheromone is updated for all the nondecreasing paths the way it would be updated for regular ant paths.

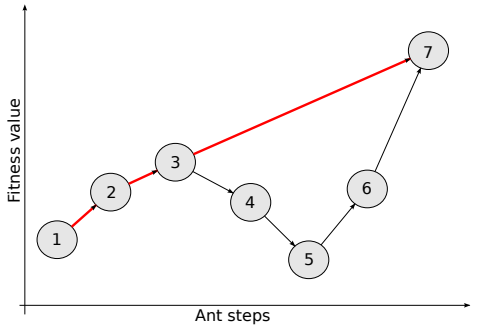

Figure 3: An example of an ant path $(1, 2, 3, 4, 5, 6, 7)$ **and the corresponding nondecreasing path** $(1, 2, 3, 7)$ **(marked red and bold).**

An example of an ant path and a corresponding nondecreasing path is shown on Fig. 3. Steps 2 and 3 in the example led to an increase in the ant's best fitness. Then, steps 4 and 5 led to a decrease in its current fitness. Step 6 was more succesful than 4 or 5, but still worse than the best step 3. At last, step 7 became the best solution found by the ant. Therefore, the original path is $(1, 2, 3, 4, 5, 6, 7)$ and the nondecreasing path would be $(1, 2, 3, 7)$.

4.9 Differences from Previous Version

There are three major differences between the algorithm described above and its previous version reported in [7, 8]. First, the use of a (1+1)-evolutionary strategy for improving the random initial solution. The first version of MuACO*sm* used the random initial solution as the root node of the construction graph, which led to using up many fitness evaluations on the early stages of the search.

Second, the use of heuristic information defined on edges of the construction graph as the absolute difference of fitness values of adjacent nodes. The original algorithm did not employ any heuristic information.

And third, the introduction of a lower bound for pheromone values which allowed to avoid the unbounded decrease of pheromone values on those graph edges that were not visited right after being built.

5. EXPERIMENTAL EVALUATION

The performance of the first version of MuACO*sm* has previously been evaluated on three problems [5, 7, 8]:

- test-based extended finite-state machine induction [32, 31];
- the Artificial Ant problem with the John Muir trail [17];
- test-based induction of FSMs with continuous and discrete output actions for unmanned aircraft control [5].

Preliminary experiments with that version of MuACO*sm* were performed and results were compared with genetic algorithms [5, 7], random mutation hill climbing and evolutionary strategies [5, 8]. Comparison of the algorithms in terms of numbers of fitness evaluations needed to find the optimal solution showed that MuACO*sm* either outperforms mentioned evolutionary algorithms or shows similar performance. In this work we consider the Artificial Ant problem with a more common Santa Fe trail [20].

5.1 Artif cial Ant Problem

The Artificial Ant problem, introduced by Jefferson in [17] is a common benchmark problem often used for performance evaluation of metaheuristic algorithms.

The problem is to find a strategy controlling an agent (called an Artificial Ant) in a game performed on a square toroidal field 32 by 32 cells. Some cells of the field contain "food" pellets, which are distributed along a certain trail. The trail usually contains turns and gaps.

There are three common trails (i.e. problem instances) for this problem – the John Muir Trail, Santa Fe trail and Los Altos Hills trail. In this work we focus on the first two trails shown on Fig. 4. Black squares indicate the food, white squares are empty and gray squares show gaps in the trail.

The Santa Fe trail has a length of 144, contains 21 turns and 55 gaps. The John Muir trail has a smaller length of 121, but contains as much as 33 turns and 38 gaps. Both trails contain 89 pellets of food.

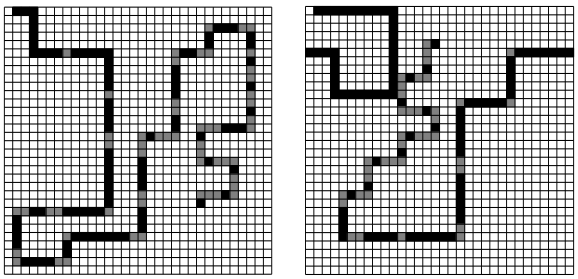

Figure 4: The Santa Fe (on the left) and John Muir (on the right) food trails.

In both trails the ant's initial position is the leftmost upper cell and it is initially "looking" east. The ant receives events from the field – it can determine whether the next cell contains a piece of food or not. On each step it can turn left, turn right or move forward, eating a piece of food if the next cell contains one. The maximum number of steps the artificial ant is allowed to make varies in different experimental setups.

In this problem there are two input events – F (the next cell contains food) and $!F$ (the next cell does not contain food) – and three output actions: L (turn left), R (turn right) and M (move forward).

The goal in this problem is to build a finite-state machine that would allow the artificial ant to eat all pieces of food in the allowed number of steps. The fitness function we use is defined in the following way:

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

where n_{food} is the number of food pellets eaten by the ant, $s_{\max}$ is the maximum number of steps the ant is allowed to make and s_{last} is the number of the step on which the ant ate the last piece of food.

Algorithm parameter values that were used in the experiments are presented in Table 2.

Table 2: MuACO*sm* parameter values.

Parameter	Value	
	Santa Fe	**John Muir**
Colony type	Parallel	Parallel
N	5	10
ρ	0.5	0.5
n_{stag}	50	50
N_{stag}	100	200
N_{mut}	20	60
p_{new}	0.6	0.6
α	1.0	1.0
β	1.0	1.0

5.2 Experiments on the Santa Fe Trail

Many published results are available for the Artificial Ant problem with the Santa Fe Trail. The best results were achieved in [9] by two approaches:

- Memorized-Random-Tree-Search, GP + Subroutines (43000 fitness evaluations);

- Memorized-Random-Tree-Search, Random Search + Subroutines (20696 fitness evaluations).

Although these approaches use LISP S-expression representation of programs, the comparison we perform is still feasible because an S-expression can be transformed into a finite-state machine, for example using an algorithm described in [19]. Furthermore, we know of no approaches that would build solutions for the Santa Fe trail faster than the approach in [9].

In our experimental setup we varied both the number of accessible states in the target FSM and the maximum number of steps the artificial ant is allowed to make. The experiment was run 10,000 times for each case to achieve statistically meaningful results. MuACO*sm* found an FSM capable of eating all pellets of food in each run. Fig. 5 shows the mean number of fitness evaluations used to find the optimal solution for each FSM size and maximum number of ant steps $s_{\max}$.

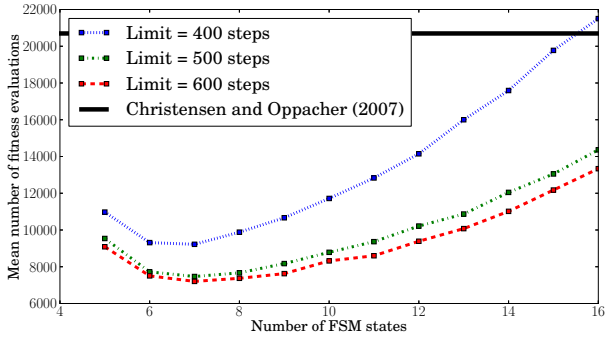

Figure 5: Mean fitness evaluations for different FSM sizes and limits on the number of artificial ant steps.

Results on Fig. 5 show that MuACO*sm* was able to achieve a computational effort which is less than the effort achieved by any other published algorithm. In particular, the smallest computational effort of 7203 mean fitness evaluations for 600 ant steps was achieved for target FSMs with seven states. It is approximately three times less than the mean effort of 20696 published in [9]. A state diagram of one FSM with five states is shown on Fig. 6. The start state on all FSM diagrams in this paper is always state 1.

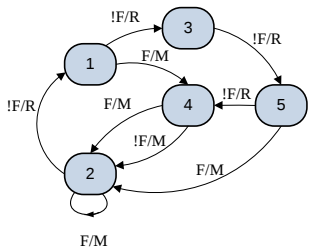

Figure 6: FSM with five states for the Santa Fe trail that allows the ant to eat all food in 394 steps. Unused transitions are not shown on the diagram.

5.3 Experiments on the John Muir Trail

For the John Muir trail we compare our results with those achieved with an evolutionary algorithm in [6] and with GA in [33]. In [6] a rather easy experimental setup is used, which allows the artificial ant to run for 600 steps. A much harder setup is considered in [33], which only allots the artificial ant with 200 steps. All compared algorithms were run 100 times until completion.

5.3.1 *Easy setup: 600 steps*

In our first experiment we compared our results with those presented in [6]. The authors of [6] used an evolutionary algorithm without crossover. They generated both non-modular (regular) and modular FSMs. Basically, a modular FSM can contain other FSMs encapsulated into its states. The evolutionary algorithm used mutations that changed the number of states in the FSMs. Therefore, FSMs in the populations had variable numbers of states. FSMs in the population had about 12 states when achieving perfect scores. In case of modular FSMs, main FSMs had about seven states, while sub-FSMs contained as much as about 13 states.

We, on the contrary, did not build modular FSMs. Instead, we varied the number of FSM states from 7 to 16 in order to complete the comparison.

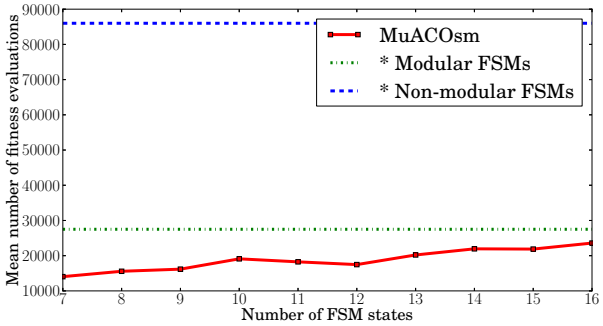

Figure 7: John Muir trail (easy): mean fitness evaluations for different FSM sizes. (*) Chellapilla and Czarnecki (1999) [6]

Results achieved in [6] are as follows. For the modular case, perfect FSMs were achieved in 96% of runs, while generation of the perfect machine took a mean of 27500 fitness evaluations. For the non-modular case, perfect machines were evolved only in 88% of runs, and the mean number of fitness evaluations rose up to 86000.

Results of experimental runs for MuACO*sm* as well as plotted mean results from [6] are presented on Fig. 7. These results show that MuACO*sm* is about four-five times as fast as the EA from [6] in the non-modular case. Furthermore, MuACO*sm* can build non-modular FSM solutions for this problem about two times faster than the EA that builds modular FSMs.

5.3.2 *Hard setup: 200 steps*

In this experiment the number of FSM states was also varied from 7 to 16. FSMs with less than seven states were not considered due to the fact that no FSM with less than

seven states can solve the problem in 200 steps as it was proven in [33].

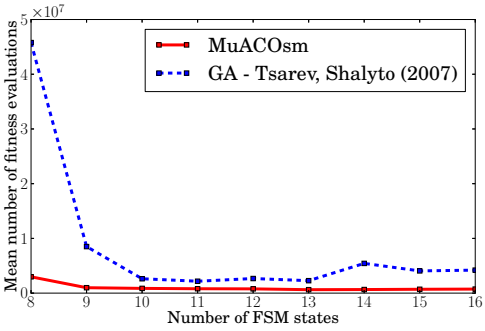

Figure 8: John Muir trail (hard): mean fitness evaluations for different FSM sizes.

For FSMs with seven states, MuACO*sm* achieved a mean number of approximately 31×10^6 fitness evaluations, while GA showed as much as 1799×10^6 fitness evaluations. Thus, MuACO*sm* is approximately 60 times faster than GA on the minimal FSMs. Results for other FSM sizes presented on Fig. 8 show that in these cases MuACO*sm* is also several times better than GA. We should also mention that the previous version of MuACO*sm* allowed to build a solution for FSMs with seven states using as much as approximately 200×10^6 fitness evaluations, and that only with a modified fitness function. One of the generated FSMs with seven states is shown on Fig. 9.

6. DISCUSSION AND FUTURE WORK

Experimental results presented in section 5 demonstrate that the proposed MuACO*sm* algorithm is quite successful in the application to the considered problem. Furthermore, the proposed method proved to be successful in solving more complex problems such as generating FSMs with continuous and discrete output actions from test examples [5].

However, it is, of course, not clear if the proposed method will be successful on other classes of problems involving FSMs and other automata. In particular, the method should certainly be compared with EA on the problems of learning random deterministic finite automata [24] and finite state transducers from test examples [23]. This will give a picture of how the proposed method performs on two large classes of problems. One more interesting direction is the exploration

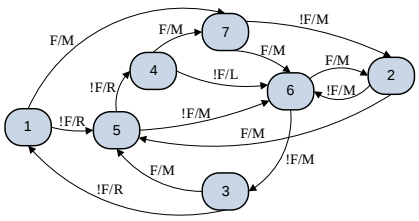

Figure 9: FSM with seven states for the John Muir trail that allows the ant to eat all food in 189 time steps.

of MuACO*sm*'s properties, influence of particular parameters on the algorithm's efficiency, possibly considering a very simple example problem.

Another issue that needs attention is selection of the algorithm's parameters. For single-instance problems such as the Artificial Ant problem, these parameters can be selected manually or using a full factorial experiment design. However, if we were to deal with classes of problems (e.g. learning random DFA), we could employ some procedure for automatic algorithm configuration such as the `irace` package [22].

In this particular work the values of parameters were first selected manually for the easier Santa Fe field. Next, for the John Muir field we shifted three exploration parameters towards more intensive exploration – we substantially increased the values of N, N_{mut} and N_{stag}.

Furthermore, we think that it might be fruitful to apply the proposed method to other combinatorial problems apart from learning finite-state machines. Indeed, the only three things that we need in order to apply the proposed algorithm to some combinatorial problem is a problem instance model, a set of mutation operators and a fitness function.

Last but not least, the apparent successful experience with applying ACO to FSM induction encourages us to apply other swarm intelligence techniques to this problem.

7. CONCLUSION

We have developed a new method of learning finite-state machines with the use of ACO. The problem of learning FSMs is reduced to the problem of finding an optimal node in a graph, where nodes are associated with FSMs and edges are associated with FSM mutations. The efficiency of the new algorithm in terms of fitness evaluation numbers needed to find the optimal solution was compared with traditional algorithms on the Artificial Ant problem. The comparison showed that the new algorithm is several times faster than other algorithms for the considered problem.

8. REFERENCES

[1] E. Alba and F. Chicano. Acohg: dealing with huge graphs. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 10–17, New York, NY, USA, 2007. ACM.

[2] 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*, GECCO '11, pages 775–778, New York, NY, USA, 2011. ACM.

[3] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[4] B. Bullnheimer, R. F. Hartl, and C. Strauß. A new rank based version of the ant system - a computational study. *Central European Journal for Operations Research and Economics*, 7:25–38, 1997.

[5] I. Buzhinsky, V. Ulyantsev, and A. Shalyto. Test-based induction of finite-state machines with continuous output actions. 2013. To appear.

[6] K. Chellapilla and D. Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation. CEC'99*, volume 2, pages 1349–1356, 1999.

[7] D. Chivilikhin and V. Ulyantsev. Learning finite-state machines with ant colony optimization. In *Proceedings of the 8th international conference on Swarm Intelligence*, ANTS'12, pages 268–275, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] 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*, GECCO Companion '12, pages 603–606, New York, NY, USA, 2012. ACM.

[9] S. Christensen and F. Oppacher. Solving the artificial ant on the santa fe trail problem in 20,696 fitness evaluations. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1574–1579, New York, NY, USA, 2007. ACM.

[10] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEEE Proceedings – Software*, 150(3):161–175, 2003.

[11] M. Dorigo. Optimization, learning and natural algorithms, 1992. PhD thesis.

[12] M. Dorigo and L. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[13] 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*, 26(1):29–41, 1996.

[14] J. Gomez. An incremental-evolutionary approach for learning deterministic finite automata. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation. CEC '06*, pages 362–369, 2006.

[15] M. Harman. Software engineering meets evolutionary computation. *Computer*, 44(10):31–39, 2011.

[16] M. Harman, A. Mansouri, and Y. Zhang. Search-based software engineering: A comprehensive analysis and review of trends, techniques and applications. Technical report, 2009. Tech. report TR-09-03, Dept. of Computer Science, King's College, London.

[17] 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.

[18] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[19] D. Kim. Memory analysis and significance test for agent behaviours. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 151–158, New York, NY, USA, 2006. ACM.

[20] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992. Cambridge, MA, USA.

[21] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference*, ICGI '98, pages 1–12, London, UK, UK, 1998. Springer-Verlag.

[22] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[23] S. Lucas and J. Reynolds. Learning finite state transducers: Evolution versus heuristic state merging. *IEEE Transactions on Evolutionary Computation.*, 11(3):308–325, 2007.

[24] S. Lucas and T. Reynolds. Learning dfa: evolution versus evidence driven state merging. In *Proceedings of the 2003 Congress on Evolutionary Computation. CEC '03*, volume 1, pages 351–358, 2003.

[25] D. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi. The bees algorithm, 2005. Technical Note.

[26] D. Pham, E. Koc, J. Lee, and J. Phrueksanan. Using the bees algorithm to schedule jobs for a machine. In *Proceedings of the Eighth International Conference on Laser Metrology, CMM and Machine Tool Performance*, pages 430–439, 2007.

[27] N. Polykarpova and A. Shalyto. *Automata-based programming*. Piter., 2009. In Russian.

[28] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation. CEC' 98*, pages 69–73, 1998.

[29] 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*, ISMIS '00, pages 166–175, London, UK, UK, 2000. Springer-Verlag.

[30] T. Stützle and H. H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(9):889–914, June 2000.

[31] F. Tsarev. Method of finite-state machine induction from tests with genetic programming. *Information and Control Systems (Informatsionno-upravljayushiye sistemy)*, (5):31–36. In Russian.

[32] 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*, GECCO '11, pages 759–762, New York, NY, USA, 2011. ACM.

[33] F. Tsarev and A. Shalyto. Use of genetic programming for finite-state machine generation in the smart ant problem. Number 2, pages 590–597. Moscow, Phizmatlit, 2007. In Russian.

[34] X. Yang. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2008.