Learning Finite-State Machines with Ant Colony Optimization

Daniil Chivilikhin and Vladimir Ulyantsev

Saint-Petersburg National Research University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia
chivilikhin.daniil@gmail.com

Abstract. In this paper we present a new method of learning Finite-State Machines (FSM) with the specified value of a given fitness function, which is based on an Ant Colony Optimization algorithm (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 and the goal is to maximize the given fitness function, which is defined on the set of all FSMs with given parameters. Comparison of the new algorithm and a genetic algorithm (GA) on benchmark problems shows that the new algorithm either outperforms GA or works just as well.

1 Introduction

Finite-state machines can be applied to various problems. FSMs have proven to be a good representation of agent strategies [1]. They are also used as efficient representations of large dictionaries [2]. In a programming paradigm called automata-based programming [3] FSMs are used as key components of software systems.

The problem of inducting finite-state machines for a given fitness function has drawn the attention of many researchers. The common approach to this problem is the use of various evolutionary algorithms (EA). In [1] Spears and Gordon used evolutionary strategies (ES) to learn FSMs for the Competition for Resources problem. In [4] and [5] a GA was used for inducting a FSM from test examples with a special crossover operator based on tests. In [2] Lucas and Reynolds used an EA to learn deterministic finite automata from labeled data samples. Another GA was used in [6] to build an optimal solution of the John Muir food trail problem. EA have proven to be efficient in cases when FSMs cannot be built heuristically.

The optimization problem we are solving is formulated in the following way: given the number of states N, a set of events Σ and a set of actions Δ build a FSM with the specified target value of the fitness function f.

We propose a new local-search heuristic method of learning FSMs based on ACO and compare it with GA in terms of performance.

2 ACO Overview

In ACO, the solutions are built by a set of artificial ants which use a stochastic strategy. The solutions can be represented either as paths in the graph, or simply by graph vertices. Each edge (u, v) of the graph (u and v are vertices of the graph) has an assigned pheromone value τ_{uv} and can also have an associated heuristic distance η_{uv}. The pheromone values are modified by the ants in the process of solution construction, while the heuristic distances are assigned initially and are not changed. An ACO algorithm consists of three major steps which are repeated until a viable solution is found or a stop criterion is met. In the first step — `ConstructSolutions` — each ant explores the graph following a certain path. The ant chooses the next edge to visit according to the pheromone value and heuristic distance of this edge. When an edge has been selected, the ant appends it to its path and moves to the next node. In the next stage — `UpdatePheromones` — the pheromone values of 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 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. On the last (optional) stage — `DaemonActions` — some procedure is executed performing actions that cannot be performed by individual ants.

When we apply ACO to FSM generation we have to deal with huge graphs, sometimes consisting of several millions vertices. In our work we apply a variation of the *expansion technique* introduced in [7] – we limit the ant path lengths to reduce the size of the graph we store in memory.

3 Finite-State Machines and Search Space Representation

We formally define a finite-state machine as a sextuple $(S, \Sigma, \Delta, \delta, \lambda, s_0)$. Here, S is a set of states, Σ is a set of input events, Δ is a set of output actions. δ is a transition function mapping a state and an event to another state, i.e. $\delta(s, e) = t$, where $s, t \in S$, $e \in \Sigma$. λ is a transition function mapping a state and an event to an output action, i.e. $\lambda(s, e) = a$, where $s \in S$, $e \in \Sigma$, $a \in \Delta$ and s_0 is the initial state.

Informally speaking, a mutation of a FSM is a small change in its structure. In this work we consider two 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.

The search space, which is a set of all FSMs with the specified parameters, is represented in the form of a directed graph G with the following properties:

- the vertices of G are associated with FSMs;
- let u be a vertex associated with FSM A_1 and v be a vertex 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.

For each pair of FSMs A_1 and A_2 and the corresponding pair of vertices u and v, there exists a path in G from u to v and also from v to u.

4 The Proposed Algorithm

The overall scheme of our algorithm complies with the classical ant colony optimization algorithm with a few modifications. First we generate an initial random solution. While stagnation is not reached, the artificial ants select paths in the graph, building it in process and deposit pheromone on the graph edges.

4.1 Path Construction

First we need to select a node for each ant to start from. If the graph is empty, we generate a random initial solution (a FSM) and place all the ants into the node associated with that solution. This random initial solution is generated by randomly defining the transition functions of a FSM with a fixed number of states. If the number of nodes in the graph is greater than zero, then the start nodes for the ants are selected randomly from the list of nodes in the best path — a path traveled by a certain ant leading to the solution with the highest fitness function value.

Let the artificial ant be located in a node u associated with FSM A. If this node has successors, then the ant selects the next node v to visit according to the rules discussed below — search space expansion and stochastic path selection.

Search Space Expansion. With a probability of p_{new} the ant attempts to construct new edges of the graph by making N_{mut} mutations of A. The procedure of processing a single mutation of machine A is as follows:

- construct a mutated FSM A_{mutated};
- find a node t in graph G associated with A_{mutated}. If G does not contain such a node, construct a new node and associate it with A_{mutated};
- add an edge (u, t) to G.

After all N_{mut} mutations have been made, the ant selects the best newly constructed node v and moves to that node.

Stochastic 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:

$$p_{uv} = \frac{\tau_{uv}^{\alpha} \cdot \eta_{uv}^{\beta}}{\sum_{w \in N_u} \tau_{uw}^{\alpha} \cdot \eta_{uw}^{\beta}}, \tag{1}$$

where $v \in N_u$ and $\alpha, \beta \in [0, 1]$. In our algorithm all the heuristic distances η_{uv} are considered to be equal and do not influence path selection, β does not influence path selection and α equals one at all times.

If u does not have any successors, then the next node is selected according to the search space expansion rule with a probability of 1.0.

4.2 Controlling Graph Growth

We use the following mechanisms for controlling the graph growth rate:

- each ant 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.

4.3 Pheromone Update

For each graph edge (u, v) we store $\Delta\tau_{uv}^{\text{best}}$ – 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 $\Delta\tau_{uv}^{\text{best}}$ are updated for all edges along this sub-path. Next, for each graph edge (u, v), the pheromone value is updated according to the classical formula:

$$\tau_{uv} = \rho\tau_{uv} + \Delta\tau_{uv}^{\text{best}}, \tag{2}$$

where $\rho \in [0, 1]$ is the evaporation rate.

5 Experimental Evaluation

To evaluate the efficiency of our algorithm, we applied it to the following problems:

- inducting finite-state transducers based on test examples [4] [5];
- inducting a finite-state machine for the John Muir food trail problem [6].

5.1 Inducting FSMs from Test Examples

Fitness Evaluation Method. The input data for this problem is the set of possible events, the set of possible actions, the number of states in the target FSM and a set of test examples. Each of the test examples consists of an events sequence `Inputs[i]` and a corresponding actions sequence `Actions[i]`. For fitness function evaluation, each events sequence `Inputs[i]` is given as input to the FSM, and the resulting output sequence `Outputs[i]` is recorded. After running the FSM on all test examples, the following value is calculated:

$$FF_1 = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \frac{ED\left(Outputs[i], Actions[i]\right)}{max\left(\left|Outputs[i]\right|, \left|Actions[i]\right|\right)}\right) \tag{3}$$

Here, n is the number of test examples and $ED(s_1, s_2)$ denotes the edit distance between strings s_1 and s_2. The final expression for the fitness function has the form:

$$FF_2 = 100 \cdot FF_1 + \frac{1}{100}\left(100 - n_{\text{transitions}}\right) \tag{4}$$

Here, $n_{\text{transitions}}$ is the number of transitions in the FSM. Note that we do not evolve the output sequences on the FSM transitions — we use a technique introduced in [5] and used in [4] called smart transition labeling. The idea is to evolve the numbers of output actions for each transition instead of the output sequences themselves. The output sequences for each transition are selected optimally using a simple algorithm based on test examples.

Alarm Clock Problem Description. An alarm clock has three buttons for setting the current time, setting the alarm to a specific time and turning it on or off. The alarm clock has two operation modes – one mode for setting the current time and another one for setting the alarm time. When the alarm is off, the user can push the buttons H and M to increment the hours and minutes of the current time respectively. If the user presses the A button, the clock is switched to the alarm setting mode. In this mode the same buttons H and M are used to adjust the time when the alarm should sound. When the user presses the A button in this mode, the alarm is set to go off when the current time will be equal to the alarm time. When the alarm is ringing, the user can press the A button to switch it off, however it will automatically turn off after one minute. The alarm clock also contains a timer that increments the current time each minute. This system has four input events and seven output actions.

We compare the efficiency of our algorithm with the results obtained in [6] using a GA. The goal of the experiment was to generate a heuristically built FSM that satisfies all the test examples, has three states and 14 transitions.

Experiment. In the experiment we searched for the solution among FSMs with four nominal states following the experimental setup in [5]. Our algorithn had the following values of parameters: N — 5, ρ — 0.5, n_{stag} — 20, N_{stag} — 4, N_{mut} —20, p_{new} — 0.6.

We have performed 1000 runs of our algorithm and also 1000 runs of the GA and measured the average number of fitness evaluations used to generate the target FSM. Results show that the ACO algorithm required an average of 53944 fitness evaluations, while GA required more than twice as much — an average of 117977 fitness evaluations. The transition diagram of one of the constructed FSMs is shown on Figure 1. The start state on this diagram and all the other diagrams in this paper is always state 1.

5.2 The Food Trail Problem

Problem Description. The food trail problem, described in [8], is considered to be a benchmark problem for testing the performance of evolutionary algorithms. The objective is to find a program controlling an agent (called an ant) in

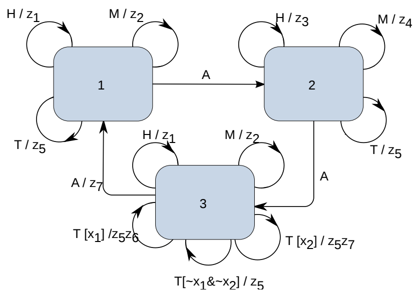

Fig. 1. Finite-state machine for the alarm clock problem

a game performed on a square toroidal field 32 by 32 cells. There are 89 pieces of food (apples) distributed along a certain trail in this field. In the beginning of the game, the ant is located in the leftmost upper cell and is looking east. The field, food trail and the ant's position at the beginning of the game are shown on Figure 2. The black squares indicate the food, the white squares are empty and the gray squares show the trail.

The ant 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 target controlling program must allow the ant to eat all 89 apples in no more than 200 steps.

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

The solution proposed in [3] is to build a finite-state machine controlling the ant. The authors of [6] achieved the best-known solution of this problem using GA. The generated FSMs contained seven states and allowed the ant to eat all 89 apples in less than 200 steps. The construction of such a FSM in two different experiments took the GA 160 and 250 million fitness evaluations respectively.

Experiments. We have performed two different experiments on this problem. In the first experiment we tried to search for the solution among FSMs with seven states, copying the experimental setup of [6]. Our algorithm produced two valid solutions after 143 and 221 million fitness evaluations correspondingly. For the second experiment, we chose to expand the search space and search for the target FSM among FSMs with 12 states. We still wanted the solution FSM to contain only seven states, therefore we modified the classical fitness function:

$$f(A) = n + \frac{200 - n_{\text{steps}}}{200} \tag{5}$$

Here, A is the FSM, n is the number of eaten apples and n_{steps} is the number of the step on which the ant ate the last apple. Our modified fitness function takes into account the number of states visited by the FSM in the process of fitness evaluation:

$$f(A) = n + \frac{200 - n_{\text{steps}}}{200} + 0.1 \cdot (M - N) \tag{6}$$

Here, M is the number of states in the initial FSM and N is the number of visited states. We have performed 30 runs of our algorithm with the following values of parameters: N — 10, ρ — 0.5, n_{stag} — 40, N_{stag} — 200, N_{mut} — 40, p_{new} — 0.5. This experiment was performed for the ACO algorithm only, because an experiment with the GA algorithm would require certain changes in its code, which we did not have access to.

Results show that in this case our algorithm only requires an average of 37.29 million fitness evaluations to reach the desired solution. The transition diagram of one of the generated FSMs is shown on Figure 3. This machine allows an ant to eat all food in 189 steps.

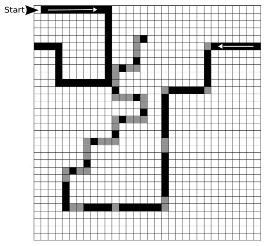

Fig. 2. The field in the food trail problem (John Muir trail)

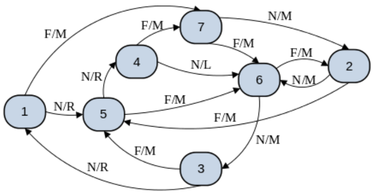

Fig. 3. Finite-state machine for the food trail problem

6 Conclusion

We have developed an ACO-based local-search heuristic method of learning finite-state machines for a given fitness function. ACO is used to find an optimal vertex in a graph, where vertices are associated with FSMs and edges are associated with mutations of FSMs. The efficiency of the proposed algorithm has been compared to GA on the problem of inducting FST from test examples and on the John Muir food trail problem. On both problems our method has either outperformed GA or worked just as well.

References

1. Spears, W.M., Gordon, D.E.: Evolving finite-state machine strategies for protecting resources. In: Proceedings of the International Symposium on Methodologies for Intelligeng Systems, pp. 166–175 (2000)

2. Lucas, S., Reynolds, J.: Learning dfa: Evolution versus evidence driven state merging. In: The 2003 Congress on Evolutionary Computation (CEC 2003), vol. 1, pp. 351–348 (2003)
3. Polykarpova, N., Shalyto, A.: Automata-based programming. Piter (2009) (in Russian)
4. Tsarev, F., Egorov, K.: Finite-state machine induction using genetic algorithm based on testing and model checking. In: Proceedings of the 2011 GECCO Conference Companion on Genetic and Evolutionary Computation (GECCO 2011), pp. 759–762 (2011), `http://doi.acm.org/10.1145/2001858.2002085`, doi:10.1145/2001858.2002085
5. Tsarev, F.: Method of finite-state machine induction from tests with genetic programming. Information and Control Systems (Informatsionno-upravljayushiye sistemy, in Russian) (5), 31–36 (2010)
6. Tsarev, F., Shalyto, A.: Use of genetic programming for finite-state machine generation in the smart ant problem. In: Proceedings of the IV International Scientific-Practical Conference "Integrated Models and Soft Calculations in Artificial Intelligence", vol. (2), pp. 590–597 (2007)
7. Alba, E., Chicano, F.: Acohg: dealing with huge graphs. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computing (GECCO 2007), pp. 10–17 (2007), `http://doi.acm.org/10.1145/1276958.1276961`, doi:10.1145/1276958.1276961
8. Koza, J.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge (1992)