Evolving EFSMs Solving a Path-Planning Problem by Genetic Programming

Maxim Buzdalov
Saint-Petersburg National Research University of IT, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia
mbuzdalov@gmail.com

Andrey Sokolov
Saint-Petersburg National Research University of IT, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia
ansokolmail@gmail.com

ABSTRACT

In this paper, we present an approach to evolving of an algorithm encoded as an extended finite-state machine that solves a simple path-planning problem — finding a path in an unknown area filled with obstacles using a constant amount of memory — by means of genetic programming. Experiments show that in 100% of cases a reasonably correct EFSM with behavior similar to one of the BUG algorithms is evolved.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Algorithms, Experimentation

Keywords

genetic programmming, path-planning problem, bug algorithms, finite-state machine

1. INTRODUCTION

Automata-based programming [7, 8, 4] is a programming paradigm which proposes to design and implement software systems as systems of interacting automated controlled objects. Each automated controlled object consists of a controlling EFSM and a controlled object.

The main idea of automata-based programming is to distinguish control states and computational states. The number of control states is relatively small, each of them differs qualitatively from the others and defines the actions. The number of computational states can be very large (even infinite), they differ from each other quantitatively and define only results of the actions but not the actions themselves.

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

Complex programs can be designed using the automata decomposition [7]. This idea can be transformed to an approach of separately evolving strategies represented as EFSMs for different situations, then evolving a top-level EFSM that decides which of these strategies to use. This paper describes an approach for evolving a relatively low-level EFSM, which can be seen as a part of complex automata-based navigation system.

1.1 Extended finite-state machine

An EFSM has a set of *control* states, a transition function and an action function. A controlled object has *computational* states and interfaces with the FSM by *commands* and *input variables*.

The EFSM takes events and input variables as input. Generally, they can come from other parts of the system as well as from the controlled object. After receiving an event and values of the input variables, the FSM transits to some control state and issues commands (or *output actions*) to the controlled object, which, in turn, can change its computational state.

1.2 Path planning with incomplete information

In [6], a family of BUG algorithms, which solve the most restricted class of path planning problems with incomplete information, is introduced.

A two-dimensional scene filled with unknown obstacles with continuous boundaries of finite size is given. An *agent* is a point-sized robot which needs to reach a certain *target location*. The agent has the knowledge of its own coordinates and the coordinates of the target location. It has a *sensor* that tells the agent if it hits an obstacle. The agent has a constant amount of memory to operate. In particular, it can not store the part of the scene it has already visited.

It is shown in [6] that a number of algorithms exist which can drive the agent to the target location if it is possible and determine if the target location is unreachable otherwise in finite time. In [5], the ideas of BUG algorithms were extended to the case of limited distance sensors.

2. PROBLEM DESCRIPTION

In this research we consider a simplified version of the problem described in [6]. Here, the scene is an infinite square grid. Each cell of this grid is either free or contains an obstacle.

The agent occupies an entire cell. The *location* of the

agent is defined by its coordinates and *direction*, which can be one of "north", "west", "south", "east". We denote the next cell in this direction to be *adjacent* to the agent. The externally visible actions of the agent on each turn are limited to the following:

- move forward to the adjacent cell;
- rotate 90 degrees clockwise;
- rotate 90 degrees counter-clockwise;
- terminate and say it has reached the target location;
- terminate and say the target location is unreachable;
- do nothing.

If an agent tries to move to a cell which contains an obstacle, it is said to be "crashed", and the process stops. If agent terminates and says it has reached the target location, but it is not located there, or it says that the target location is unreachable while it is not, it is considered to operate incorrectly.

The agent knows its own coordinates and direction, and the coordinates of the target location. It is able to sense if there is an obstacle in the adjacent cell. In addition, the agent has an internal memory, where it is able to store its location (coordinates and direction) at a certain moment in the past.

The part of the agent described above forms a controlled object, and the part that actually determines how the agent behaves is the controlling EFSM. To design this EFSM, we need to describe the input variables and the output actions first.

2.1 Input Variables

The input variables may depend only on the information known to the agent:

- X_t, Y_t — the target location;
- X_a, Y_a, D_a — the agent's coordinates and direction;
- X_s, Y_s, D_s — the saved coordinates and direction;
- X_j, Y_j — the coordinates of the adjacent cell (a function of X_a, Y_a, D_a);
- O — is there an obstacle in the adjacent cell.

The set of input variables used in this research is described in the Table 1. The number of input variables is kept relatively small, while the complexity of their computation is quite low.

2.2 Output actions

The output actions are the same as the possible actions of the agent visible from the outside except for one action which is dedicated to saving the current location of the agent to the memory. All output actions are listed in Table 2.

3. GENETIC PROGRAMMING

In this research, the controlling FSM for the agent is evolved using genetic programming.

Table 1: Input variables

Label	Mnemonic identifier	Formula
x1	CAN_MOVE_FORWARD	not O
x2	IS_MOVE_FORWARD_COOL	$dist(X_j, Y_j, X_t, Y_t) < dist(X_a, Y_a, X_t, Y_t)$
x3	IS_AT_FINISH	$X_a = X_t$ and $Y_a = Y_t$
x4	IS_AT_SAVED	$X_a = X_s$ and $Y_a = Y_s$ and $D_a = D_s$
x5	IS_BETTER_THAN_SAVED	$dist(X_a, Y_a, X_t, Y_t) < dist(X_s, Y_s, X_t, Y_t)$

Note: $dist(X_1, Y_1, X_2, Y_2) = |X_1 - X_2| + |Y_1 - Y_2|$.

Table 2: Output actions

Label	Mnemonic identifier	Description
z1	DO_NOTHING	Do nothing
z2	MOVE_FORWARD	Move to the adjacent cell
z3	ROTATE_POSITIVE	Rotate clockwise
z4	ROTATE_NEGATIVE	Rotate counter-clockwise
z5	SAVE_POSITION	Save current agent's location
z6	REPORT_REACHED	Terminate and say the target is reached
z7	REPORT_UNREACHABLE	Terminate and say the target is unreachable

3.1 FSM Representation

In the proposed method, Mealy EFSM is used [8], where output actions depend on the values of the input variables and the state of the EFSM. The EFSM is represented as an array of numbered states. Start state has the number 0.

Each state is represented by a decision tree [3]. Each non-leaf node of the decision tree holds a reference to an input variable used to define which of the children subtrees to use. If the value of the input variable in the node is true, one proceeds to the "true" subtree, if not, to the "false" one. Each leaf of the tree holds a description of the transition (the output action and the number of next state).

The use of decision trees is motivated by the fact that, in each state, for most of the cases only a few of input variables is used, so the decision tree representation is more compact than the full table representation [3].

In this research, two crossover operators are used. The first one exchanges decision trees for each state with the probability of 0.2. The second one exchanges two randomly selected subtrees in corresponding trees for each state with the probability of 0.2. These two operators are chosen equiprobably.

Two mutation operators are used. The first one for each state with the probability of 0.2 replaces the whole decision tree in it with a randomly generated one. The second one for each state with the probability of 0.2 replaces a randomly selected subtree with a randomly generated subtree of the same size. These two operators are chosen equiprobably.

3.2 Fitness Function

In this research, the fitness function is based on simula-

tion. An EFSM is tested by putting an agent controlled by this FSM on a *field* — a scene with a finite number of finite obstacles, and with initial and target locations specified — and running the agent until it stops or any other reasonable condition is satisfied.

We propose the following fitness function design. At every moment of time there are several fields generated in advance. The fitness function of an EFSM depends on several measures of a path of an agent which is controlled by that FSM. These measures include the length of the path, the shortest distance from a path to the target, and the like. When the best FSM in the generation correctly passes all these fields, new fields are generated randomly in such a way that this FSM works incorrectly on every such field. After that, these fields are added to the list of fields, individuals are reevaluated and the evolution continues.

The termination criterion, that is, when the best EFSM does pass any possible field correctly, is equivalent to the situation when it is impossible to generate a next set of fields. To decide it formally is quite hard, so a simpler approach is used: if after a significantly large number (the order of 10^4) of randomly generated fields no is field found that proves the current EFSM to be wrong, it is considered to be correct. The actual correctness is to be proved later by a human. In practice, no EFSMs were found which passed this "torture test" and appeared to be incorrect.

3.2.1 Possible Run Results on a Single Field

Consider the ways a simulation of a single FSM on a single field may end in.

An agent may either:

- stop by crashing into an obstacle;
- stop by issuing a `REPORT_REACHED` output action;
- stop by issuing a `REPORT_UNREACHABLE` output action;
- run forever.

The first case is self-contained. In the second case, the actual result, i.e. whether the agent works correctly, depends on the agent may or may not stand at the target location, so there are "truly reached" and "wrongly reached" outcomes.

The third case is more difficult to analyse. If the agent says that the target is unreachable, but in fact it is not, then it is definitely wrong. However, if the target is really unreachable, the agent may be wrong as well. Consider the case illustrated in Fig. 1, where the target is unreachable and the agent stops using the suitable output action. However, if we slightly modify this case (Fig. 2) by removing some of the obstacles which the agent did not test, the target becomes reachable, so the agent is actually wrong.

To overcome this problem, we introduce the concept of "mandatory cells". For a field with an unreachable target we first compute an area that is accessible by an agent. The target cell will then be contained inside an unreachable connected component. Each free cell having a common edge with this component is said to be *mandatory* for the agent to visit. If an agent does not visit at least one of the mandatory cells, its `REPORT_UNREACHABLE` message is considered wrong. Strictly speaking, this is a heuristic that may potentially reject some correct agents, but in practice a correct agent with a constant memory which may miss some of mandatory cells is too difficult to build, so this heuristic is feasible.

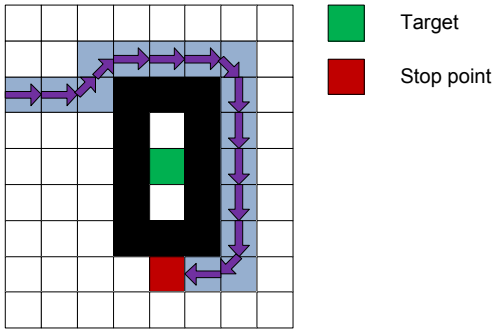

Figure 1: Wrong `REPORT_UNREACHABLE`, case 1

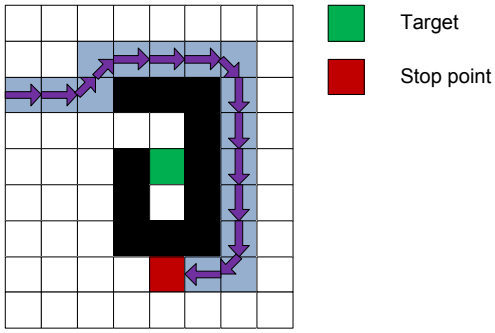

Figure 2: Wrong `REPORT_UNREACHABLE`, case 2

There are quite a number of possible scenarios that either yield the fourth case or look very similar to it. To avoid complex analysis of trajectories, we use the following heuristic: the agent is said to be "running away" if it moves further than five cells away from the rectangular area that contains start and target locations and all the obstacles. The first motivation is that nearly all BUG algorithms either follow a boundary of an obstacle or move in the direction of the target, so it is unusual for them to move too far from the described area. The second motivation is that the correct EFSMs that are able to move too far away seem to be more complex than the correct ones which do not do that.

All possible run results are listed in Table 3. By applying heuristics described above, we guarantee that simulation of any FSM on any field will take finite time.

3.2.2 Fitness Functions for Single and Multiple Fields

In this research, the fitness function of a FSM on a single field takes the path of the FSM on this field and the run result and returns a floating-point value, depending on a number of indicators of the given path. The code that evaluates the fitness function is accessible in the code repository at [1]. The less the function, the better the EFSM. The fitness function for multiple fields is defined as an average over fitness functions for each of the fields.

3.3 Algorithm Scheme

To evolve an EFSM using fitness function and genetic operators described above, a genetic algorithm from the framework [2] was used. The number of individuals in generation

Table 3: Possible run results on a single field

Mnemonic identifier	Description
`REACHED`	Agent stopped in target location with `REPORT_REACHED`
`WRONGLY_REACHED`	Agent stopped outside of target location with `REPORT_REACHED`
`UNREACHABLE`	Agent visited all mandatory cells and stopped with `REPORT_UNREACHABLE`
`WRONGLY_UNREACHABLE`	Agent stopped with `REPORT_UNREACHABLE`, but not all mandatory cells were visited, or the target is reachable
`CRASHED`	Agent crashed into an obstacle
`LOOPED`	Agent looped
`RAN_AWAY`	Agent moved too far from the area with obstacles

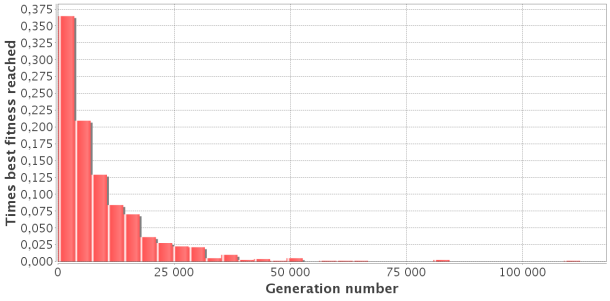

Figure 3: Performance Histogram

is set to 100. To escape from stagnation, after some preliminary experiments it was decided to restart the algorithm every 5000 generations, if the EFSM solving the problem is not found yet. Two selection operators were used, the roulette selector and the tournament selector, which showed similar performance.

4. EXPERIMENT AND RESULTS

There were 800 runs of the genetic algorithm conducted. There were five states in each EFSM. Each run was continued until an EFSM solving the problem (i.e. the one which stands a torture test of 10^4 random fields) had been found.

There were generated 800 FSMs, each solving the problem. They were classified by similarity of paths on several random fields (common for all EFSMs) to find how much different algorithms there are. If several differences between these algorithms are ignored (i.e. changes of EFSM state while not moving or rotating, or extra rotations while standing at the same cell), then it appeared that there were 404 FSMs that move around the obstacles clockwise, and 396 FSMs moving counter-clockwise, so both versions evolve equiprobably.

Fig. 3 shows a histogram describing the performance of the GA, where each bucket corresponds to a range of generation numbers, and the height of a bucket is the relative number of runs that evolved a correct EFSM using the corresponding number of generations.

Trajectory analyses show that the evolved EFSMs generally follow a strategy which can be described as a "blind Tangent bug for Manhattan distance". Initially it heads towards the target until it hits an obstacle. Then the current point is saved and the agent starts to circumvent the obstacle until either it is possible to move towards the target again, or the saved point is reached (which is the evidence of the unreachable target). The differences between EFSMs, besides the direction of the obstacle traversal, is in the initialization code and in optimality of analysing the neighborhood.

5. CONCLUSIONS

In this paper, a method for evolving controlling EFSMs using genetic programming to solve a simple path-planning problem is proposed. This method was tested on a problem of path planning with incomplete information, and was shown to successfully evolve different solutions to the problem. The method can be seen as a part of a framework for designing hierarchies of EFSMs for control of complex navigation systems.

6. FUTURE WORK

The future work is planned in the following directions:

- introducing EFSM simplification and invariant proofs to genetic algorithms;
- using more input variables to see if resulting programs are more efficient;
- testing EFSM representations other than decision trees;
- searching for similar problems and test the proposed method on them.

7. REFERENCES

[1] Source code for the fitness function. http://goo.gl/XGwPq.

[2] Watchmaker framework for evolutionary computation. http://watchmaker.uncommons.org/.

[3] V. Danilov and A. Shalyto. Genetic programming method for induction of finite state machines represented as decision trees. In *Proceedings of XI International Conference on Soft Computations and Measurements*, pages 248–251, 2008.

[4] V. Gurov, M. Mazin, A. Narvsky, and A. Shalyto. Tools for support of automata-based programming. *Programming and Computer Software*, 33(6):343–355, 2007.

[5] I. Kamon, E. Rivlin, and E. Rimon. A new range-sensor based globally convergent navigation algorithm for mobile robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, number 1, pages 429–435, 1996.

[6] V. Lumelsky and A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitraty shape. *Algorithmica*, 2:403–430, 1987.

[7] N. Polikarpova and A. Shalyto. *Automata-based Programming, 2nd Edition (in Russian)*. Piter, 2011.

[8] A. Shalyto. Logic control and reactive systems: Algorithmization and programming. *Automation and Remote Control*, 62(1):1–29, 2001.