Learning Actions with Symbolic Literals and Continuous Effects for a Waypoint Navigation Simulation

Morgan Fine-Morris[1], Bryan Auslander[2], Hector Muños-Avila[1], Kalyan Gupta[2]

[1] Lehigh University, Bethlehem PA 18015, USA
[2] Knexus Research Corporation, National Harbor MD 20745, USA

Abstract. We present an algorithm for learning planning actions for waypoint simulations, a crucial subtask for robotics, gaming, and transportation agents that must perform locomotion behavior. Our algorithm is capable of learning operator's symbolic literals and continuous effects even under noisy training data. It accepts as input a set of preprocessed positive and negative simulation-generated examples. It identifies symbolic preconditions using a MAX-SAT constraint solver and learns numeric preconditions and effects as continuous functions of numeric state variables by fitting a logistic regression model. We test the correctness of the learned operators by solving test problems and running the resulting plans on the simulator.

Keywords: Learning action models, MAX-SAT, Logistic regression.

1 Introduction

A recurring problem in applying automated planning technology to practical problems is the need to manually encode domains with a collection of actions that generalize all possible actions. Automated learning of such operators is a possible solution to the problem of manual operator encoding and has been a recurrent research topic as we will discuss in the related work section.

In this paper, we are interested in the automated learning of planning actions for waypoint simulations. Waypoint simulations are a crucial subtask for mobile networks [1], gaming agents [2], and robotics agents [3] that must perform locomotion behavior. Learning operators in this domain requires reasoning with continuous-valued variables with durative actions. Learning operators in this domain requires combining the following capabilities simultaneously:

- Learning symbolic literals, whose arguments can take values from a predefined collection, such as vehicle(veh26).
- Learning continuous-valued effects, such as y=f(30), fuel(veh26,y), indicating that veh26 fuel level changes as a function of f.
- Learning under noise data, e.g. a state with the literal position(veh26,loc2) when veh26 is at location loc3. Noise can be introduced when a human operator creates the traces.

- Use of simulation to learn and test the resulting actions. While the learner is independent of the domain, we test the learned operators with the simulation from which we learn them.

In this paper we present an automated action-learning system that tackles these learning requirements simultaneously. To the best of our knowledge, this is the first work on learning action models combining symbolic literals and continuous effects under noisy training data.

The remainder of the paper is as follows: we first discuss related work; next we present the waypoint-navigation simulation used to generate the training data and to test the results of the learner; afterwards we discuss a description of the symbolic-learning procedure; next, we discuss a description of the procedure to learn numeric-durative preconditions and effects, accompanied by an example learned operator; then, we discuss a description of the experiments used to test the learned operators and the results; and finally, discuss future directions.

2 Related Work

Arora *et al.* [4] summarizes the current available algorithms for learning actions. The two algorithms most similar to our own in terms of learning numeric elements of action models are PlanMiner-O2 [5] and ERA [6], however both algorithms differ with respect to our work in substantial ways. ERA learns a mapping of potentially noisy inputs to a set of numeric categories corresponding to a discrete set of output values (one value per category). They use 4 categories for the input: 1-road, 2-grass, 3-dirt and 4-rocks. The sensors might return a reading such as 1.5, indicating that the terrain is either road or grass. The classification scheme might learn that an input of 1.5 should map to the return value for category 1, meaning a road; and based on the classified category, the algorithm returns the speed that a vehicle will be able to travel in that kind of terrain. Succinctly, ERA will learn the effects as a constant function $f(x_1, ..., x_n) = c$, where x_1, …, x_n are input variables and c is computed as the mean of all values of training examples with input $x_1, ..., x_n$. PlanMiner-O2 learns numeric preconditions, and in the effects increases or decreases numeric values by a constant factor. Succinctly, it will learn a function $f(x_1) = x_1 \pm c$ for an input variable x_1. In contrast to these two works, we are learning the numeric preconditions and effects as a continuous-valued function $f(x_1, ..., x_n)$ on input numeric variables $x_1, ..., x_n$ in addition to the symbolic preconditions.

In addition to the discussions of PlanMiner-O2 and ERA, Arora *et al.* [4] presents an overview of some 30 other works on learning action models. Our overall claim of novelty versus existing work is that this is the first action-learning system that combines learning symbolic literals and durative effects under noisy training data.

Our approach for learning the symbolic preconditions builds on ideas from the ARMS operator-learning algorithm [7], which also uses MAX-SAT. Unlike ARMS, we assume full state observability as given by the simulation. As a result, we did not need to formulate hypothesizing causal links between actions. Thus, instead of action traces $s_0\ a_1\ s_1 \ldots s_n$ partially annotated with intermediate states, s_i, the input to MAX-SAT in our algorithm are collections of triples *(s,a,s')*, where *s* and *s'* are the (complete) observed states before and after executing *a*. As a result, with an adequate threshold Θ and sufficient training examples, the false positives drop to nearly zero and the false negatives to zero. The crucial difference is that our algorithm also learns durative actions whereas ARMS learns symbolic literals only.

The EXPO system was designed to learn missing preconditions and effects or even complete operators [8]. EXPO does this by having expectations of when an action should (or should not) be solvable (respectively when a plan should or should not be generated). When a discrepancy occurs between the agent's expectations of the actions and the observed effects of the action, it examines the history of the action's applications to fix it. For instance, looking for missing conditions when the action was successfully applied by accident. Such a process is complementary to our work, when the operators could be further refined after learning the initial collection of operators. Unlike our work, EXPO is strictly symbolic and was not designed to deal with noise in the training data.

The TRAIL system [9] follows similar ideas of completing operators but it does so in a non-deterministic domain where there can be multiple outcomes. It does so by maintaining possible execution trees and assuming a teacher is available that can provide examples on-demand, allowing the learner to extrapolate missing knowledge. It can model durative actions by projecting the effects over multiple time steps and learn the projected intervals for numeric values. It requires substantial background knowledge including the goals that the agent is pursuing, used for performing inductive logic programming [10], and the means to generate traces even when the actions are incomplete by using teleo-reactive trees [11]. The most important differences versus our work is twofold. First, we can deal with noisy training data. Second, we are able to learn the changes in values of durative actions as functions.

Pasula, *et al.* [12] presented an action learning system for stochastic domains and tested the learned actions for MDP planning tasks. It can deal with noise because of the stochastic nature of the domain. In our case, we can cope with noise even though the learned action model is deterministic. Furthermore, we learn durative effects.

Lindsay & Gregory [13] did a study on the domains used in the international planning competition (IPC) and identified a collection of numeric constraints that are typically used in those competitions. This collection is used to learn numeric conditions by using negative and positive examples. The negative examples are used to eliminate constraints that are inconsistent with the training data and the positive examples are used to tune the resulting constraints. In our case, we are learning the continuous function $f(x_1, ..., x_n)$ on input numeric variables $x_1, ..., x_n$ in addition to the symbolic preconditions.

Walsh & Littman's [14] system learns STRIPS operators augmented with the Web description language so they can be used for semantic web service composition. In contrast to our work, it assumes no noise in the training data and no durative effects. It also provides bounds to the operator learning problem: in the general case, learning preconditions may require an exponential number of plan prediction mistakes (PPMs). Informally, a PPM counts the number of times the operator was applied incorrectly. If the maximum number of preconditions of operators is known to be bounded by k, then the number PPM is bounded by a polynomial on k.

3 Waypoint Navigation Simulation

We implemented a simulator for agents to perform logistics and transportation tasks using the python discrete event simulation framework *SimPy*.[1] The simulator models agents or vehicles that can travel to various locations, via a network of roads (edges) that connect them. The agents navigate between waypoints in a geographic area represented by a two-dimensional coordinate system or grid as shown in Figure 1. It depicts 17 locations on a road network or graph. Each location is marked as a vertex on the network and labelled as *loc1* through *loc17*. The edges between the vertices indicate valid roads that can be travelled. The graph is not complete and travel from one location to another may require multiple transits. Depending on agent speed, traveling between waypoints can take different times.

The simulator supports multiple vehicles to concurrently perform logistics actions such as trucks performing activities in tandem. The vehicles can perform two actions: transit and refuel. A transit action enables a vehicle to move between locations if an edge connects them. A refuel action refills a vehicle's fuel tank if it is in a location. We model these activities with continuous valued variables. Vehicles have average travel speed and fuel consumption rate. The simulator consumes inputs in the form of world states comprising vehicles, locations, and roads along with a schedule comprising list of transit and refuel actions. The simulator generates connection networks randomly.

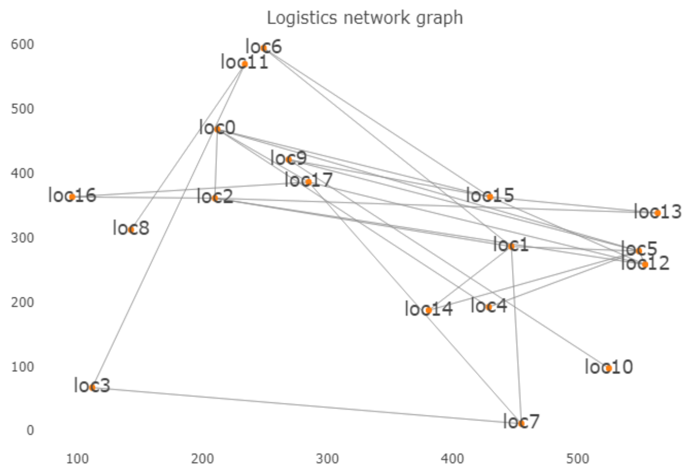

Fig. 1. Example logistics road network.

The simulator executes the input schedule and outputs a simulation trace in commonly-used JavaScript Object Notation (JSON) format[2] as shown in Figure 2. It shows a fragment of the simulation trace obtained by executing a transit action from *loc12* to *loc5*. Every time an action starts or finishes the entire simulation state is logged

[1] https://simpy.readthedocs.io/en/latest/
[2] https://www.json.org/

into the trace as training data and a means of replaying the simulation. The trace shows the completion of the transit action for *veh0*; *veh0* is now at position 5. Similarly, *veh1* is at an edge between *loc6* and *loc1*. The trace notes the status of each vehicle such as rate of fuel consumption, current gas level, and total gas capacity. It enforces domain constraints at runtime and will return a failed trace if an invalid action is attempted.

4 Overview of Learning Algorithm

There are two requirements for the learning algorithm: (1) positive and negative examples and (2) background knowledge on numeric values used to infer numeric-durative effects. From these requirements as a starting point, the procedure performs the following steps: (1) simulate positive examples by randomly generating a road network and performing random walks from locations in the network; (2) Convert JSON traces into predicate form; (3) Convert numeric arguments into constants to prepare for learning symbolic literals; (4) Construct clauses (which enforcing constraints) from positive examples and run these clauses on a MAXSAT solver to learn the operator's symbolic literals; (5) Generate negative examples; (6) Use the positive examples, negative examples and background knowledge to generate training data for a logistic regression learner to generate the operator's durative effects.

We provided an example of the JSON output in Figure 2. The conversion into predicate form is mostly straightforward and we omit the details for the sake of space.

```
"action": {
   "action_id": "17",
   "end_location": "loc5",
   "start_location": "loc12",
   "type": "TransitAction",
   "vehicle": "veh0"},
  "state": {
   "edges": [{
    "end": {"name": "loc5"},
    "name": "edgeloc0loc5",
    "start": {"name": "loc0"},
    "traffic_speed": 60.0 }, ...],
   "locations": [{"name": "loc5", "x": 549, "y": 279}, ...]
   "vehicles": [{
    "gas_tank_capacity": 30,
    "gas_tank_level": 17,
    "gph": 1.0,
    "name": "veh1",
    "position": "edgeloc6loc1"}, {
    "gas_tank_capacity": 30,
    "gas_tank_level": 19.56,
    "gph": 1.0,
    "name": "veh0",
    "position": "loc5"}]},
"status": "finished", "time": 26.0
```

Fig. 2. Example JSON simulation trace fragment.

5 Learning Symbolic Literals

5.1 MAX-SAT

The conjunctive normal form CNF-SAT is the decision problem of determining the satisfiability of a given CNF $\phi = (D_1 \wedge D_2 \wedge \ldots \wedge D_n)$, where each clause D_i is a disjunction $(X_1 \vee X_2 \vee \ldots \vee X_m)$ and each literal X_j is either a Boolean variable or its negation. CNF-SAT is NP-complete. MAX-SAT is a variation of CNF-SAT in that each clause D_i has associated a nonnegative weight, $w(D_i)$. The MAX-SAT problem is defined as follows: given a CNF ϕ, find a *subset* of clauses, $D_{1'}, D_{2'}, \ldots, D_{m'}$, in ϕ such that (1) there is an assignment of the Boolean variables making each of these clauses true and (2) W' = w($D_{1'}$)+ $w(D_{2'})$+ ... + $w(D_{m'})$ is maximized. That is, for any other subset of clauses $D_{1''}, D_{2''}, \ldots, D_{m''}$, satisfying Condition 1, then W' ≥ W'', where W'' = w($D_{1''}$)+ $w(D_{2''})$+ ... + $w(D_{m''})$. MAX-SAT is NP-hard. MAX-SAT solvers run in polynomial-time on the number of clauses and approximate a solution to find a collection of clauses that can be satisfied together with the variable Boolean assignments that make this possible. They provide no guarantee that Condition (2) is satisfied. In our work we use the MAX-SAT solver reported in Ansótegui *et al.* [15].[3]

5.2 Using MAX-SAT to Learn Symbolic Literals

We learn symbolic preconditions from positive examples, which we extract from simulator-generated random-walk traces. Positive examples are state-action-state triples, $s_t\ a_t\ \ s_{t+n}$, where s_t occurs immediately before the action starts, and s_{t+n} after the action ends. n ≥ 1 because actions are durative and can interleave, so they may take several states before completion. We replace numeric values with the generic constant *num*, during symbolic learning. We compile potential preconditions from the literals in s_t of each example. We remove unlikely preconditions, namely, literals (1) which share no arguments with the action's signature or (2) which do not appear in more than some fraction, Θ of examples. Condition (1) is a relaxation of the standard STRIPS notation where arguments of literals appearing in the preconditions must be arguments named in the action's signature. Condition (1) requires at least one of the arguments of each precondition of the action TransitAction(v21, l4, l7) to be either v21, l4, or l7.

We use the state-variable action representation. That is, a state is defined as a collection of variables and their assigned values. From state to state, the variable remains the same but their values might change. For instance, the literal *position(v1, loc1)*, represents the state variable "position of vehicle v1" and indicates it is currently assigned to *loc1*.

Since actions are durative, state changes for multiple actions can occur between s_t and s_{t+n} and removing literals helps ensure that no erroneous preconditions are learned. For example,

s_t: position(v1, loc1), position(v2, loc1), ...
a: action(v1, loc1, loc2)
s_{t+n}: position(v1, loc2), position(v2, edge13), ...

[3] https://github.com/jponf/wpm1py

Although *position(v2, loc1)* could be identified as a precondition because it includes *loc1*, unless it occurs in s_t of multiple examples, it will be removed from consideration.

For each remaining literal, we include a clause in our CNF equation asserting that the literal is a precondition; this means that the literal is a candidate to be a precondition of the action and it will be up to MAX-SAT to determine which of those clauses are true. The weight of the clause is the number of examples for which the literal appears in s_t. We add additional clauses in our CNF encoding to enforce the following conditions:

1. If a state-variable in s_{t+n} has a changed value compared to s_t, it must be a true precondition.
2. A state-variable in the effects must have changed its value with respect to its preconditions.
3. All actions have at least one precondition and one effect.

Constraint 1 ensures that the preconditions include literals relevant to the action parameters when the state-variable changed from s_t to s_{t+n}. Constraint 2 requires that all effects are literals that underwent changes during the action. Constraint 3 ensures that all actions are non-trivial, i.e., that they effect some change in the state.

6 Learning Durative Preconditions and Effects

Unlike symbolic literals, learning durative effects requires positive and negative examples and background information. We learn a logistic regression model [16] for *gas_tank_level* as a function of the initial tank level and the travel distance, which we use in both preconditions and effects.

Domain Background Information. To compute the gas tank level of a vehicle we need the distance between locations. However, distance is not an explicit attribute in the state, so we provide background information indicating how to calculate distance along an edge by computing the Manhattan distance between its start and end locations.

6.1 Using Logistic Regression

For learning numeric literals, we revert back from the generic constant *num* to the original numeric values in all positive examples. To generate negative examples, we generate traces using random walks in the simulator and remove all refuel actions from the traces. We simulate these modified traces and if for a particular state-action pair, s_i a_{i+1}, in the trace, the action is not executable by the simulator, then the pair (s_i, a_{i+1}) is used as a negative example for action a_{i+1}.

From each example (negative or positive), we extract numeric preconditions and discard any where the value is the same across all examples, e.g., the gallons per hour, *gph*. We use the remaining values (*x* and *y* for start location and end location, *gas_tank_level* for vehicle) and the background information to calculate the distance between the locations and train a Logistic Regression model on distance and *gas_tank_level*, using the positive and negative labels of the examples as our classes. In the effects, we calculate the new value for *gas_tank_level* using the decision function of the trained logistic regression model.

6.2 Example Learned Operator

Figure 3 shows the learned transit operator, TransitAction. The vehicle *?vehicle* will move from location *?start* to location *?end* along edge *?edge* (question marks denoted variables). The start and end of *?edge* are defined in two literals. The effects describe the change in the vehicle's position and the gas tank level.

```
Head:
 TransitAction(?vehicle, ?start, ?end)
Preconditions:
 gas_tank_capacity(?vehicle, ?gtcv)
 gas_tank_level(?vehicle, ?gtlv1)
 gph(?vehicle, ?gphv)
 position(?vehicle, ?start)
 type(?vehicle, "vehicles")
 end(?edge, ?end)
 type(?end, "locations")
 x(?end, ?xe)
 y(?end, ?ye)
 start(?edge, ?start)
 type(?start, "locations")
 x(?start, ?xs)
 y(?start, ?ys)
 ?gtlv2 ← -0.0328 + 0.62*?gtlv1 + -0.199*distance_metric((?xs, ?ys), (?xe, ?ye))
 ?gtlv2 > 0
Effects:
 position(?vehicle, ?end)
 gas_tank_level(?vehicle, ?gtlv2)
```

Fig. 3. Learned TransitAction Operator.

7 Empirical Evaluation

We perform two sets of analysis, the first evaluating the effectiveness of MAX-SAT at learning the symbolic preconditions, the second testing the effectiveness of the whole learning procedure.

7.1 Experimental Setup

To evaluate the effectiveness of the symbolic literal learning component, we generate a pool of 100 simulations, each with a different initial state, containing at least one of each type of action. From each simulation we extract one positive example of each action type to create a pool of 100 positive examples for each action. From this pool of

100, we select 5 sets of 3, 10, and 30 examples and learn the symbolic preconditions for each set. For each set, we also select a set of mislabeled examples to test the effects of incorrect examples (noise) on the system's learning ability. Mislabeled examples are examples which do not contain the appropriate symbolic preconditions, but are treated as positive learning examples. We learn preconditions from each of these sets at three different Θ values, 0.1, 0.2, and 0.3. The performance metrics were false positives (i.e., additional preconditions, not appearing in the ground truth) and false negatives (i.e., missing preconditions). The ground truth is a collection of literals that were manually selected in order to ensure that the operator was correctly learned.

To test the effectiveness of the whole procedure, we select a set of 10 positive examples, 8 negative examples, and the corresponding learned preconditions. We create a set of operators for each Θ value and test these operators on 30 randomly-generated planning problems. The problems are constructed by generating a random state and randomly selecting a vehicle and destination location. The performance metric is the number of test problems for which a valid plan is generated; plan validity is tested by running the plan on the simulator.

Solution plans are generated using HTN planning techniques, which direct the selected vehicle to explore the state map until it arrives at the goal location, refueling to ensure a full tank before each transit. To validate a generated plan, we convert to JSON schedule format and simulate from the start state.

7.2 Results

Figure 4 shows the false positive rate for learning symbolic preconditions is non-zero and dependent on both the Θ value and the number of learning examples. Each line represents the combined false positive rate (FPR) for symbolic preconditions for both operators when learning from a set of positive examples and/or a set of positive examples with an additional 10% of examples which are mislabeled, simulating noise in the data.

The FPR changes more between 3 and 10 positive examples than it does between 10 and 30 examples. When learning from 10 or more examples, the largest improvement occurs between theta of 0.1 and 0.2, with little improvement between 0.2 and 0.3.

Interestingly, noise added via mislabeled examples had little impact on the number of false positive preconditions, except at Θ=0.3 for 3 positive examples. The addition of mislabeled examples actually improves the set of preconditions, because it decreases the percent of examples in which the same non-preconditions occur, allowing more non-preconditions to be filtered out.

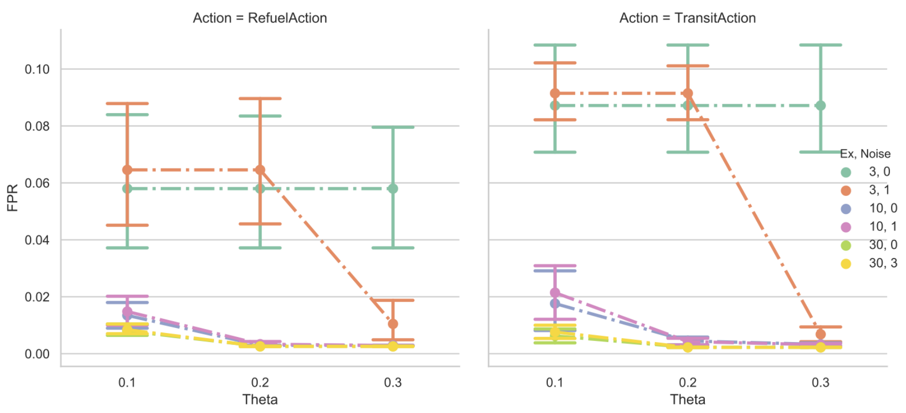

Fig. 4. False positive rates with respect for the Refuel and the Transit action.

Even for Θ=0.3 and 30 training examples, there is a non-zero error rate in the false positives because literals (e.g., *gph*) were the same throughout the examples and the learner acquired them, however we deemed them superfluous because they were constant and present in all training and testing examples.

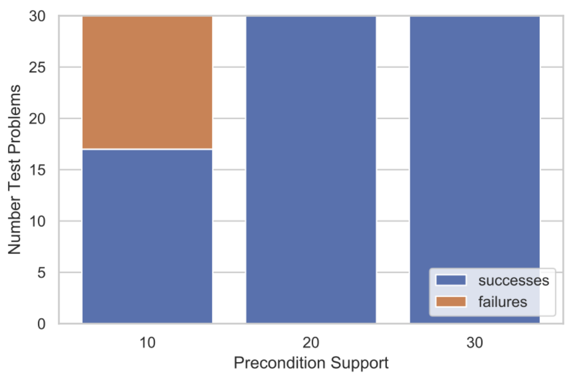

Fig. 5. Number of solved test problems for Θ=10%, 20% and 30%. Solution plans were run in the simulator. All test problems were solved and verified for Θ=20% and 30% and more than half were for Θ=10%.

There were no false negative literals learned in any of the runs, demonstrating the effectiveness of the MAX-SAT procedure.

Figure 5 shows the number of test problems (out of 30) for which the planner is able to generate plans using operators learned with Θ=0.1, 0.2 and 0.3. The simulator validated all generated plans. For Θ=0.1, slightly more than half of the test problems were solvable by the planner. As we can see from Figure 4, with Θ=0.1 and 30 examples, there are some 15% additional literals in the preconditions, restricting applicability of operators.

8 Final Remarks

Our algorithm successfully learns operators with symbolic literals and continuous effects even under noisy training data. To the best of our knowledge, this is the first action learning algorithm that combines symbolic and numeric fluents and durative effects.

There are a number of possible future directions. First, we provide background knowledge because our simulator doesn't compute the distance explicitly as part of the state information. It would have been easy to modify the simulator and make the distance explicit. Indeed, other waypoint navigation simulators explicitly compute distances. However, in general some kind of background knowledge is needed when computing complex numeric literals, such as durative effects, on conditions such as traffic flow, traffic speed etc. In the future, we want to explore inductive learners such as ILASP which may help induce background knowledge. Second, we want to take into account temporal considerations such as an action starting at time t and a second action starting at a time t+Δ and the effects of both actions contributing towards a third action starting later. For instance, at time t, a hose starts adding water to a container at a certain fixed rate, and later at time t+Δ another hose starts adding water to the same container. Later we open a faucet to drain the container. In this case the rate that the container is filled and drained is a function of time.

References

1. Bettstetter, C., & Wagner, C.: The Spatial Node Distribution of the Random Waypoint Mobility Model. WMAN, 11, 41-58 (2002).
2. Tan, C. H., Ang, J. H., Tan, K. C., & Tay, A.: Online adaptive controller for simulated car racing. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), pp. 2239-2245. IEEE (2008).
3. Bruce, J., & Veloso, M. M.: Real-time randomized path planning for robot navigation. In: Robot Soccer World Cup, pp. 288-295. Springer, Berlin, Heidelberg (2002).
4. Arora, A., Fiorino, H., Pellier, D., Métivier, M., & Pesty, S.: A review of learning planning action models. The Knowledge Engineering Review, 33 (2018).
5. Segura-Muros, J. Á., Pérez, R., & Fernández-Olivares, J.: Learning numerical action models from noisy and partially observable states by means of inductive rule learning techniques. KEPS 2018, 46 (2018).

6. Balac, N., Gaines, D. M., & Fisher, D.: Learning action models for navigation in noisy environments. In: ICML Workshop on Machine Learning of Spatial Knowledge, Stanford (2000, July).
7. Yang, Q., Wu, K., & Jiang, Y.: Learning action models from plan examples using weighted MAX-SAT. Artificial Intelligence, 171(2-3), 107-143 (2007).
8. Gil, Y.: Learning by experimentation: Incremental refinement of incomplete planning domains. In: Machine Learning Proceedings 1994, pp. 87-95. Morgan Kaufmann (1994).
9. Benson, S. S.: Learning action models for reactive autonomous agents. Doctoral dissertation, Stanford University (1996).
10. Lavrac, N., & Dzeroski, S.: Inductive Logic Programming. In: WLP, pp. 146-160 (1994).
11. Nilsson, N.: Teleo-reactive programs for agent control. Journal of artificial intelligence research, 1, 139-158 (1993).
12. Pasula, H. M., Zettlemoyer, L. S., & Kaelbling, L. P.: Learning symbolic models of stochastic domains. Journal of Artificial Intelligence Research, 29, 309-352 (2007).
13. Lindsay, A., & Gregory, P.: Discovering Numeric Constraints for Planning Domain Models. KEPS 2018, 62 (2018).
14. Walsh, T. J., & Littman, M. L.: Efficient learning of action schemas and web-service descriptions. In: AAAI-2008, vol. 8, pp. 714-719. (2008).
15. Ansótegui, C., et al.: Improving SAT-based weighted MaxSAT solvers. In: International conference on principles and practice of constraint programming, pp. 86-101. Springer, Berlin, Heidelberg (2012, October).
16. Hosmer Jr, D. W., Lemeshow, S., & Sturdivant, R. X.: Applied logistic regression, vol. 398. John Wiley & Sons (2013).