LOUGA: Learning Planning Operators Using Genetic Algorithms

Jiří Kučera and Roman Barták(✉)

Faculty of Mathematics and Physics, Charles University,
Malostranské nám. 25, Praha, Czech Republic
bartak@ktiml.mff.cuni.cz

Abstract. Planning domain models are critical input to current automated planners. These models provide description of planning operators that formalize how an agent can change the state of the world. It is not easy to obtain accurate description of planning operators, namely to ensure that all preconditions and effects are properly specified. Therefore automated techniques to learn them are important for domain modelling.

In this paper, we propose a novel method for learning planning operators (action schemata) from example plans. This method, called LOUGA (Learning Operators Using Genetic Algorithms), uses a genetic algorithm to learn action effects and an ad-hoc algorithm to learn action preconditions. We show experimentally that LOUGA is more accurate and faster than the ARMS system, currently the only technique for solving the same type of problem.

Keywords: Planning · Learning · Action models · Genetic algorithms

1 Introduction

Automated planning deals with the problem of finding a sequence of actions that transfer the world from the current state to a desired state. It is a model-based method, where the model formally describes how the actions are changing states of the world. Hence an important aspect of automated planning is obtaining a proper model of actions. In this paper we deal with classical (STRIPS) planning where actions are defined via preconditions and postconditions (effects), each being a set of predicates. The problem that we are addressing in the paper is how to learn these sets of preconditions and postconditions automatically from examples of plans.

There exist various approaches to acquisition of planning domain models. The early works such as EXPO [3] or later works such as STRIPS-TraceLearn [9] improve action models incrementally after observing some problem during plan execution. Another approach learns from expert traces and subsequent simulations [10]. Frequently, the acquisition problem consists of finding the domain model from examples of plans, which is also the topic of this paper. In other words, the problem is to learn a correct state transition function according to

K. Yoshida and M. Lee (Eds.): PKAW 2018, LNAI 11016, pp. 124–138, 2018.
https://doi.org/10.1007/978-3-319-97289-3_10

observed sequences of actions and states. The system ARMS [11] uses partially specified plans as its input, namely each plan consists of the initial state, a sequence actions, and goal predicates. Intermediate states might also be partially specified. Using MAX-SAT, ARMS learns the preconditions and effects of actions. The follower of ARMS called AMAN [12] allows some actions in plans to be wrongly recognized. LOCM [2] and LOCM2 [1] do not use a predicate model of world states but they rather learn finite-state automata for objects in the world. These automata describe how properties of objects are being changed by actions. Similarly, Opmaker2 [6] learns actions as methods to change properties (states) of involved objects and it requires some invariant formulas describing propositions that must be true in any state. ASCoL [5] and LC_M [4] both only extend the LOCM system. ASCoL learns static preconditions for LOCM and LC_M extends it to work with missing and noisy data. Finally LAMP [13] learns more complex action models with quantifiers and logical implications.

We solve the same problem as ARMS, but we relax the condition of knowing the goal predicates. The proposed system LOUGA (Learning Operators Using Genetic Algorithms) learns from valid sequences of actions (not necessarily from plans reaching certain goals as ARMS). We assume that the initial state and a valid sequence of actions is given as input. LOUGA can also exploit partially specified intermediate states and a final state to find more accurate models. We use a classical genetic algorithm to learn the effects of actions and an ad-hoc algorithm to learn the preconditions of actions. We will show experimentally that LOUGA produces more accurate domain models and it is also faster than ARMS.

2 Background and Problem Specification

We work with classical STRIPS planning that deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal condition. World states are modeled as sets of predicates that are true in those states and actions are changing validity of certain predicates.

Formally, let P be a set of all predicates modeling properties of world states. Then a state $S \subseteq P$ is a set of predicates that are true in that state (every other predicate is false). Each action a is described by four sets of predicates $(B_a^+, B_a^-, A_a^+, A_a^-)$, where $B_a^+, B_a^-, A_a^+, A_a^- \subseteq P, B_a^+ \cap B_a^- = \emptyset, A_a^+ \cap A_a^- = \emptyset$. Sets B_a^+ and B_a^- describe positive and negative preconditions of action a, that is, predicates that must be true and false right before the action a. Action a is applicable to state S iff $B_a^+ \subseteq S \wedge B_a^- \cap S = \emptyset$. Sets A_a^+ and A_a^- describe positive (add list) and negative (del list) effects of action a, that is, predicates that will become true and false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a will be $\gamma(S, a) = (S \setminus A_a^-) \cup A_a^+$. If an action a is not applicable to state S then $\gamma(S, a)$ is undefined. In this work we use additional assumptions about the applicability of actions, namely $A_a^- \subseteq S$ and $A_a^+ \cap S = \emptyset$. The first assumption says that if an action deletes some predicate from the state then this predicate should be

present in the state. Similarly, if an action adds some predicate to the state then the predicate should not be in the state before. These assumptions can be easily included in the action model as $A_a^- \subseteq B_a^+$ and $A_a^+ \subseteq B_a^-$.

In practice, operators are used in the domain model rather than actions. Operator can be seen as a parameterized action. Each operator has a set of attributes and specifies preconditions and effects as predicates over these attributes:

```
(:action move
   :parameters (?o - object ?m - place
                ?l - place)
   :precondition (at ?o ?m)
   :effect (and (at ?o ?l)
                (not (at ?o ?m))))
```

Actions are obtained by substituting constants for the attributes. The planning domain model is then specified by the set of predicates and the set of operators. PDDL modeling language [7] is the most widely used language for modeling planning domains; we will use syntax of that language in our examples.

In our approach, we assume two types of input information. First, there is a partial planning domain model consisting of a set of predicates and a set of operators with attributes but without the description of preconditions and effects. The second type of input is a set of plans, where each plan consists of the initial state and a valid sequence of actions. Partially specified intermediate states or a goal state might also be provided. The information about states can be in three forms: a predicate was observed in the state, a predicate was observed not to be in the state, or the state was fully observed. We do not make any other assumptions about the input data unlike ARMS that presumes that some effect of every action is used by some later action or in the goal state. The task is to complete the domain model by learning preconditions and effects of operators such that the provided input plans are valid plans according to this domain model.

3 LOUGA

The proposed learning approach works in two main stages. First, we will learn action effects using a standard genetic algorithm [8] with some extensions. Second, we will complete the learned action model by learning action preconditions using a polynomial ad-hoc algorithm.

In the following text, we will detail these steps. For the genetic algorithm, we need to encode action effects to a genome. We will show, how to reduce the number of possible genomes by eliminating those violating conditions imposed on action effects. We will also define the fitness function that guides the genetic algorithm and we will show some methods to help the genetic algorithm when being stuck in a local optima. Next, we will show that it is possible to learn the effects predicate by predicate rather than all together. Finally, we will present the method for learning action preconditions.

3.1 Genome Model

Genetic algorithms work with individuals, each individual encoding a solution candidate. In our case, an individual describes effects of operators. First, we generate a list of all operator-predicate pairs such that the operator can use the predicate in its add or delete lists. This property can be easily verified by checking that all attributes of the predicate are among the attributes of the operator. We assume that attributes are typed though this assumption can be relaxed as we will show in the section on experiments. Each operator-predicate pair will be associated with one of three values:

- 0: predicate is not in operator's add and del lists,
- 1: predicate is in operator's add list (positive effect),
- 2: predicate is in operator's delete list (negative effect).

The individual will be the sequence of numbers that corresponds one-to-one to the description of effects of operators.

For example, let us have a model with the following predicates and operators:

```
(:predicates
  (at ?o - (either object briefcase)
      ?l - place)
  (empty ?b - briefcase)
  (free ?o - object)
  (in ?o - object ?b - briefcase)
)
(:action move
  :parameters (?b - briefcase ?m - place
               ?l - place)
)
(:action put-in
  :parameters (?o - object ?p - place
               ?b - briefcase)
)
```

For this model, LOUGA generates the following pairs:

```
1. ((at ?b ?m), (move ?b ?m ?l))
2. ((at ?b ?l), (move ?b ?m ?l))
3. ((empty ?b), (move ?b ?m ?l))
4. ((at ?o ?p), (put-in ?o ?p ?b))
5. ((at ?b ?p), (put-in ?o ?p ?b))
6. ((empty ?b), (put-in ?o ?p ?b))
7. ((free ?o),  (put-in ?o ?p ?b))
8. ((in ?o ?b), (put-in ?o ?p ?b))
```

Hence, each individual will be described by a list of length eight. For example, the individual with genome '210 11000' corresponds to the model in which operator

(*move* ?*b* ?*m* ?*l*) has predicate (*at* ?*b* ?*m*) in its del list and predicate (*at* ?*b* ?*l*) in its add list and operator (*put-in* ?*o* ?*p* ?*b*) has predicates (*at* ?*o* ?*p*) and (*at* ?*b* ?*p*) in its add list.

Note that it is possible to encode operator's preconditions in the same way, but we will present a more efficient method to learn operator's preconditions later.

3.2 Pre-processing

The genome model specifies the search space that the genetic algorithm will explore. We can reduce this space further by eliminating individuals violating constraints of the model. This is done by exploring the example plans and identifying predicates that cannot be present in the add or del lists of specific operators. LOUGA simulates execution of the plan and for each state it finds two sets of predicates: the first set contains predicates that are definitely in the current state and the second set contains predicates that can possibly be in the current state, but it is not certain. Algorithm 1 describes how these sets are constructed and used.

Algorithm 1. Removing possible values of some genes.

Input: plan P; array M representing possible values of genes
Output: modified array M

```
1: Q ← predicates from initial state
2: R - empty set of predicates
3: for all actions a from P do
4:     generate a set of predicates X, which a can use
5:     for all p ∈ X do
6:         if p ∈ Q then                          ▷ p is definitely in current state
7:             M[(a, p),add] = false
8:             R = R ∪ {p}
9:             Q = Q \ {p}
10:        else if p ∉ R then                     ▷ p is definitely not in the state
11:            M[(a, p),del] = false
12:            R = R ∪ {p}
13:        end if
14:        if predicates S were observed after a then
15:            Q = Q ∪ S
16:            R = R \ S
17:        end if
18:        if predicates S were observed missing after a then
19:            Q = Q \ S
20:            R = R \ S
21:        end if
22:    end for
23: end for
```

Initially, the first set Q contains all predicates from the initial state (line 1) and the second set R is empty (line 2). LOUGA then goes through the actions in the order specified by the plan. For each action it generates the set of all predicates that the action can use. If some predicate is present in the state before the action, LOUGA marks that the action cannot have that predicate in its add list (line 7). If a predicate is definitely not present in that state, LOUGA marks that the action cannot delete it (line 11). All predicates generated for the action are then added to the second set and removed from the first one if they were present in it. If there are some predicates observed in the state after the action, all of these predicates are added to the first set and removed from the second one. After that, LOUGA continues with the next action until it processes the whole plan. The justification of this process is as follows. If some predicate can be modified by the action then that predicate can possibly be part of the next state. If some predicate is in the state and it is not modified by the action then the predicate stays in the state. Also, information about observed predicates can be exploited there (lines 14–21).

For example let us assume this short plan:

```
(:state
  (empty b1)
  (at b1 home)
  (free pencil)
  (at pencil home)
  (at rubber home)
  (free rubber))
(put-in pencil home b1)
(move b1 home office)
```

We know that there are exactly six predicates in the initial state. Action (*put-in pencil home* $b1$) can work with predicates (*at pencil home*), (*at* $b1$ *home*), (*empty* $b1$), (*free pencil*) and (*in pencil* $b1$). Pairs made of these predicates and operator *put-in* correspond to genes 4–8. Predicates (*at pencil home*), (*at* $b1$ *home*), (*empty* $b1$) and (*free pencil*) are definitely present in the state before action *put-in*, which means that the action cannot add them. As a result, genes 4–7 will have disabled value 1 during evolution. Predicate (*in pencil* $b1$) is not in the initial state, which means that the action cannot delete it and therefore gene 8 will have disabled value 2 during evolution.

After processing the action we move all these predicates to the second set (line 8) and delete them from the first set (line 9), if they are present there. Now the sets contain these predicates:

```
first set Q (definitely in the state)
  (at rubber home)
  (free rubber)
second set R (possibly in the state)
  (empty b1)
  (at b1 home)
  (free pencil)
  (at pencil home)
  (in pencil b1)
```

The next action is (*move b*1 *home office*). Operator *move* corresponds to genes 1–3, that means that the action can use predicates (*at b*1 *home*), (*at b*1 *office*) and (*empty b*1). Predicate (*at b*1 *office*) is in none of the sets, therefore the action cannot delete it and gene 2 cannot have value 2. Other predicates are already in the second set, so genes 1 and 3 will remain unchanged. We add predicate (*at b*1 *office*) to the second set and continue with the next action (if any).

3.3 Fitness Function

The genetic algorithm uses a fitness function which evaluates the error rate of the model represented by the individual. We assume three types of errors:

- **add error:** an action tries to add a predicate that is already present in the world state,
- **del error:** an action tries to delete a predicate that is not currently present in the world state,
- **observation error:** a predicate was observed in a state in the original plan, but it is not present in the corresponding state of the plan executed according to the current model, or there is a predicate in a state that should not be present according to observations about the corresponding state in the original plan.

Formally we can define these errors as follows: let S be a state of a plan executed according to the model represented by the individual, T be a set of predicates that were observed in the corresponding state in the input plan, N be a set of predicates that were observed not to be present in the corresponding state, a be the action performed from state S and $p \in A_a^+$, $q \in A_a^-$, $s \in S$ and $t \in T$ be some predicates. Add error occurs when $p \in S$, del error occurs when $q \notin S$, and observation error occurs when $t \notin S$, $s \in N$ or – if T was marked as a fully-observed state – when $s \notin T$.

After all plans are processed, the fitness value of the individual is defined using this formula:

$$(1 - (error_{add} + error_{del})/(total_{add} + total_{del})) * (1 - error_{obs}/total_{obs}),$$

where $error_{add}$, $error_{del}$ and $error_{obs}$ are the numbers of corresponding errors, $total_{add}$ and $total_{del}$ are the numbers of add and delete operations performed in simulation, $total_{obs}$ is the total number of observations about intermediate and goal states plus the number of surplus predicates in fully-observed states.

We tried a version of the fitness function that treated all three types of errors identically, but it turned out not to be ideal. When there were too many or too few observations in input data, evolution could get stuck in a local optima that favors good add and delete error rates over the observation error rate or vice versa. Treating observation errors separately solves this problem. We also tried to split the add and delete errors, but that had a marginal effect on efficiency. Obviously, the fitness value 1 means a perfect individual.

3.4 The Genetic Algorithm

LOUGA uses a classical genetic algorithm with one-point crossover and mutation [8]. We tried more sophisticated versions of those operators but we did not find any that would perform significantly better than the standard versions. We extended the standard algorithm by two additional operators applied when the population stagnates for some time (i.e. when the best individual is of certain age).

The first operator is basically 1-step local search starting from the best individual's genome to find all options how to change one gene to get a better individual. Genes are picked one by one and for every gene, every possible value is tried and resulting individuals are evaluated. As every gene has at most three possible values, there are at most $2 * N$ candidate genomes, where N is the length of genome. All individuals that performed better than the current best individual are added to the population.

The second operator is applied when even the local search cannot find a better individual. It stores the best individual and restarts the population. Next time before restarting it tries to use the information from previous runs by crossing the current best individual with the stored genomes from previous runs. If it breaks the stagnation, evolution goes on as before until it starts stagnating again or a perfect individual is found. If the algorithm cannot find a better local optimum even after multiple restarts, the operator deletes the local optima list and the genetic algorithm starts from scratch.

3.5 Learning Effects Predicate by Predicate

In complex[1] domain models, individuals' genomes can be too long for the genetic algorithm to work effectively. LOUGA solves this problem by learning operators' lists separately for each predicate type. It means that an instance of the genetic algorithm is run for each predicate type separately. In each instance, genomes are built only from those operator-predicate pairs that use the correct predicate type and the fitness function ignores observations of predicates of types other than the current predicate type.

This method generates the same genome as if all predicates were learned at once, the learning process is only split into multiple parts. These parts are independent to each other because occurrence of a predicate of one type cannot affect whether occurrence of a predicate of another type is incorrect or not. Therefore this method is sound and yields the same outcome as the standard approach.

3.6 Learning Preconditions

After the add and del sets are learned, the sets of preconditions are generated. LOUGA goes through every plan and for every operator and every relevant predicate it counts the number of cases where the predicate was present before the action was performed and the number of cases where it was not present. After every plan is processed, a positive precondition is created for every such pair that the predicate was always present before the action was performed, and a negative precondition is created for every such pair that the predicate was never present before the action was performed. If evolution gives a perfect individual, this method yields proper precondition lists. Algorithm 2 describes this process formally.

4 Results of Experiments

We evaluated experimentally the contribution of components of LOUGA and we compared LOUGA to ARMS, which is the only other technique solving the same problems. All experiments were run on laptop with Intel Core i5-2410 2.3 GHz processor and 8 GB of RAM. We used five classical domains from planning competitions, namely Blocksworld, Briefcase, FlatTyre, Rover, and Freecell. The Blocksworld domain is a classical planning domain, where a robotic hand builds towers of blocks. In the Briefcase domain, the task is to transport items between locations using a briefcase. The FlatTyre domain deals with the problem of repairing a flat tyre using various tools. The Rover domain models a Mars rover taking pictures and samples. The Freecall domain is an encoding of a card game.

[1] As 'complex' models we consider models that have a large number of predicate types and operators. Such models usually have long genomes so the genetic algorithm has to search through a large hypothesis space.

Algorithm 2. Generation of precondition lists

Input: genome G; set of input plans Q
Output: model M

```
create model M represented by G
Y, N - integer fields indexed similarly as G; all fields initially 0
for all P ∈ Q do
    s ← initial state of P
    for all a ∈ P do
        for all predicate p, which can be generated by a do
            g ← index of gene corresponding to (p,a)
            if p ∈ s then
                Y[g]++
            else
                N[g]++
            end if
        end for
        s ← s after performing a according to M
    end for
end for
for all gene g ∈ G; g corresponds to predicate p and operator o do
    if G[g] = 0 then
        if N[g] = 0 && Y[g] > 0 then
            add p to pre_o
        else if N[g] > 0 && Y[g] = 0 then
            add (not p) to pre_o
        end if
    else if G[g] = 1 then
        add (not p) to pre_o
    else if G[g] = 2 then
        add p to pre_o
    end if
end for
```

The domains were selected because of their various difficulties of how their models can be learned. The easiest ones are the Blocksworld and Briefcase domains, a bit harder is the FlatTyre domain and the hardest ones are the Rover and Freecell domains. Their basic characteristics are given in Table 1.

For each experiment, we randomly generated 200 valid sequences of actions (plans) and performed five-fold cross-validation test by splitting them in five equal parts and running algorithms five times. During each run we used four groups as learning data and the fifth group as a test set. Plans generated for the first three domains had usually 5–8 actions. For domains Rover and Freecell, we generated random walks (without a preset goal) that had about 15–20 actions.

Most tables show runtimes and error rates of generated models (smaller numbers are better). We define errors in similar way as described in the section about fitness function of LOUGA. Add and delete error rates are calculated by dividing the number of errors by the number of performed add or delete actions. Since

Table 1. Comparison of domain models used in experiments.

	Briefcase	Blocksworld	FlatTyre	Rover	Freecell
# object types	3	1	5	7	3
# predicate types	4	5	12	25	11
# operators	3	4	13	9	10
Avg. size of effect lists	3.3	4.5	2	3	5.6
Avg. parameters of operators	2.66	1.5	2.33	4	4.9

ARMS does not work with negative observations, we only evaluate fulfillment of those observations that state which predicates were definitely present in state. Observation error rate is therefore calculated by dividing the number of unfulfilled observations by the total number of predicates observed in intermediate and final states.

The size of population was set to 10, the threshold for local search was set to 7, the threshold for crossover with individuals from previous runs to 10, the threshold for population restart was 15 and mutation probability was 5% with 10% chance for a gene to be switched. From our internal tests we saw that benefits of having bigger population do not outweigh the longer computational time, so we kept the population sizes low. Keeping thresholds high did not provide much benefit neither, because the population did not usually break stagnation in reasonable time anyway.

4.1 Efficiency of Predicate by Predicate Approach

In the first experiment, we will show the effect of splitting the learning problem to multiple smaller problems, where each predicate is learned separately. Both versions of LOUGA reliably find flawless solutions, so we present the runtimes only (Table 2).

Table 2. Performance of LOUGA learning a model predicate by predicate compared to basic version (runtime measured in seconds).

	Pred. by pred.	Basic version
Briefcase	0.22 ± 0.15	0.86 ± 0.64
Blocksworld	0.81 ± 0.66	22.78 ± 40.38
FlatTyre	2.26 ± 0.74	111.76 ± 116.06
Rover	4.11 ± 0.33	$\gg 600$
Freecell	5.73 ± 1.1	$\gg 600$

As expected, the predicate-by-predicate mode performs significantly faster than the basic version. Moreover as the standard deviation indicates, the predicate-by-predicate mode is also more stable. Runtimes of the basic version varied greatly, some runs were even 10 times longer than others. Genetic algorithms usually suffer from such behavior because of the randomness of the method. The predicate-by-predicate mode works more consistently thanks to evolution having a clearer direction. We can say that it generates only one property at a time even though it is composed of many genes. The basic version works with all properties together and improvement in one direction can go hand in hand with step back in other.

4.2 Comparison of GA and Hill Climbing

In the second experiment, we compared the genetic algorithm with the hill climbing approach. In particular, we compared three setups:

- **LOUGA** - the standard version with 10 individuals
- **HC** - hill climbing with 1 individual and local search in every generation; restart when stuck in local optimum
- **GA** - genetic algorithm without local search operator

For the FlatTyre domain, we used inputs with only goal predicates in ending states, so the problem had many solutions. In the other domains, we used plans with complete initial and ending states.

The results (Table 3) show that the genetic algorithm without the local search operator performs much worse than the other two setups. Pure hill climbing performs better on inputs where there are many possible solutions. However if we use complex domains, there is an advantage in incorporating GA, because local search takes a lot of time on big genomes.

Table 3. Runtimes [s] of LOUGA, hill-climbing and a genetic algorithm.

	LOUGA	HC	GA
Briefcase	0.22	0.88	0.36
Blocksworld	0.81	2.22	1.78
Flat tyre (ambiguious)	2.26	1.82	4.2
Rover	4.11	6.92	34.54
Freecell	5.73	11.11	51.52

4.3 Efficiency of Using Types

We assume that objects (constants) are typed, which reduces the number of candidate predicates for preconditions and effects. LOUGA also works with models without types so our next experiment shows the effect of typing on efficiency.

Table 4. Performance of LOUGA on models with and without typing.

	Genome length		Runtime [s]	
	Types	NoTypes	Types	NoTypes
Briefcase	26	32	0.22 ± 0.15	0.97 ± 0.47
Blocksworld	16	108	0.81 ± 0.66	1.91 ± 0.76
FlatTyre	67	405	2.26 ± 0.74	11.47 ± 3.65
Rover	201	2796	4.11 ± 0.33	97.86 ± 7.67
Freecell	291	1481	5.73 ± 1.1	30.7 ± 3.96

The results (Table 4) show that LOUGA can handle domain models without types, thought efficiency decreases significantly. The table also shows the increased size of the genome when types are not used. The added genes (predicates) can be split in two groups. The first group consists of genes that use the unary predicates describing types. These genes do not add any difficulty to the problem, because they are not used in any add or delete lists and thus LOUGA immediately finds a trivial solution for those predicates. The second group consists of genes that use the original predicates. These genes do make the problem noticeably harder. In the Rover domain significantly more of these genes were created, because this domain has more operators and predicate types, and therefore more operator's parameter-predicate's parameter pairs were created by removing typing and more genes needed to be added.

4.4 Comparison to ARMS

Finally, we compared performance of LOUGA and ARMS [11], which is still the most efficient system for this kind of problem. We used two settings there. First, we used example plans with complete initial states, goal predicates, and a small number of predicates observed in intermediate states (every predicate will have 5% chance to be observed). This is the kind of input ARMS was created for. Second, we used plans with complete initial and ending states but no information about intermediate states, which is input that suits LOUGA well. We present the comparison for three domains only where both systems found solutions.

Tables 5 and 6 clearly indicate that LOUGA outperforms ARMS both in terms of runtime and quality of obtained models. From the data we can also see that ARMS has some problems generating delete lists. In many cases, there were zero predicates in delete lists in total. We assume that it is due to ARMS not having enough information about which predicates need to be deleted. LOUGA has less trouble generating those lists thanks to the assumption that a predicate has to be deleted before it can be added again to the world. But in some cases in the first experiment the learned delete lists were not the same as the delete lists of the original model, because the information about what has to be deleted was not sufficiently present in the plans. In the second experiment LOUGA knew that every predicate in the ending states was observed, so it typically found the original models.

Table 5. Comparison of ARMS and LOUGA systems. Inputs with goal predicates and small number of predicates in intermediate states were used.

ARMS	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0.263	–	0.029	0	6.19
Blocksworld	0.409	0.095	0.039	0.001	28.82
Flat tyre	0.319	0.479	0.342	0.003	504.19
LOUGA	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0	0	0	0	0.29
Blocksworld	0	0	0	0	0.64
Flat tyre	0	0	0	0	1.04

Table 6. Comparison of ARMS and LOUGA systems. Inputs with complete goal states were used.

ARMS	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0.318	–	0.032	0	6.72
Blocksworld	0.331	0.061	0.036	0.014	29.64
Flat tyre	0.336	0.507	0.311	0.005	548.09
LOUGA	Add ER	Del ER	Pre ER	Obs. ER	Runtime [s]
Briefcase	0	0	0	0	0.22
Blocksworld	0	0	0	0	0.81
Flat tyre	0	0	0	0	2.26

5 Conclusions

The paper presents a novel approach to learn planning operators from example plans using a genetic algorithm. We presented several techniques to improve performance of the classical genetic algorithm. First, we suggested the preprocessing technique to restrict the set of allowed genomes. Second, we used the genetic algorithm to learn action effects only while the preconditions are learnt separately using an ad-hoc algorithm. Third, we showed that action effects can be learnt predicate by predicate rather than learning the effect completely using a single run of the genetic algorithm. The presented approach LOUGA achieves much better accuracy and it is faster than the state-of-the-art system ARMS solving the same problem.

Acknowledgements. Research is supported by the Czech Science Foundation under the project P103-18-07252S.

References

1. Cresswell, S., Gregory, P.: Generalised domain model acquisition from action traces. In: Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany, 11–16 June 2011 (2011)
2. Cresswell, S., McCluskey, T.L., West, M.M.: Acquisition of object-centred domain models from planning examples. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, 19–23 September 2009
3. Gil, Y.: Learning by experimentation: incremental refinement of incomplete planning domains. In: Proceedings of the Eleventh International Conference on Machine Learning (ICML 1994), pp. 87–95 (1994)
4. Gregory, P., Lindsay, A., Porteous, J.: Domain model acquisition with missing information and noisy data. In: Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling, ICAPS 2017, pp. 69–77 (2017)
5. Jilani, R., Crampton, A., Kitchin, D.E., Vallati, M.: Ascol: A tool for improving automatic planning domain model acquisition. In: Proceedings of AI*IA 2015, Advances in Artificial Intelligence - XIVth International Conference of the Italian Association for Artificial Intelligence, Ferrara, Italy, 23–25 September 2015, pp. 438–451 (2015)
6. McCluskey, T.L., Cresswell, S.N., Richardson, N.E., West, M.M.: Action knowledge acquisition with Opmaker2. In: Filipe, J., Fred, A., Sharp, B. (eds.) ICAART 2009. CCIS, vol. 67, pp. 137–150. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11819-7_11
7. McDermott, D., et al.: PDDL - the planning domain definition language. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
8. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
9. Shahaf, D., Chang, A., Amir, E.: Learning partially observable action models: efficient algorithms. In: Proceedings of Twenty-First AAAI Conference on Artificial Intelligence, pp. 920–926 (2006)
10. Wang, X.: Learning by observation and practice: an incremental approach for planning operator acquisition. In: Proceedings of the Twelfth International Conference on Machine Learning (ICML 1995), pp. 549–557 (1995)
11. Yang, Q., Wu, K., Jiang, Y.: Learning action models from plan examples using weighted MAX-SAT. Artif. Intell. **171**(2–3), 107–143 (2007)
12. Zhuo, H.H., Kambhampati, S.: Action-model acquisition from noisy plan traces. In: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, 3–9 August 2013, pp. 2444–2450 (2013)
13. Zhuo, H.H., Yang, Q., Hu, D.H., Li, L.: Learning complex action models with quantifiers and logical implications. Artif. Intell. **174**(18), 1540–1569 (2010)