**Applied Intelligence manuscript No.**
(will be inserted by the editor)

# Discovering Relational and Numerical Expressions from Plan Traces for Learning Action Models

**José Á. Segura-Muros · Raúl Pérez · Juan Fernández-Olivares**

Received: date / Accepted: date

**Abstract** In this paper, we propose a domain learning process build on a machine learning-based process that, starting from plan traces with (partially known) intermediate states, returns a planning domain with numeric predicates, and expressive logical/arithmetic relations between domain predicates written in the planning domain definition language (PDDL). The novelty of our approach is that it can discover relations with little information about the ontology of the target domain to be learned. This is achieved by applying a selection of preprocessing, regression, and classification techniques to infer information from the input plan traces. These techniques are used to prepare the planning data, discover relational/numeric expressions, or extract the preconditions and effects of the domain's actions. Our solution was evaluated using several metrics from the literature, taking as experimental data plan traces obtained from several domains from the International Planning Competition. The experiments demonstrate that our proposal –even with high levels of incompleteness–correctly learns a wide variety of domains discovering relational/arithmetic expressions, showing F-Score values above 0.85 and obtaining valid domains in most of the experiments.

**Keywords** Automated Planning, Planning Domain Learning, Machine Learning, Symbolic Regression

José Á. Segura-Muros
Universidad de Granada, Spain
E-mail: josesegmur@decsai.ugr.es
ORCiD: 0000-0002-7011-755X

Raúl Pérez
Universidad de Granada, Spain
E-mail: fgr@decsai.ugr.es
ORCiD: 0000-0002-1355-1122

Juan Fernández-Olivares
Universidad de Granada, Spain
E-mail: faro@decsai.ugr.es
ORCiD: 0000-0002-7391-882X

## 1 Introduction

Planning domains are the core element of any automated planning (AP) problem. When using AP, the quality of the plans obtained for a given problem is directly related to the quality of the domain used to generate them. Planning domains are formal specifications of the planning problem, usually hand-coded by knowledge engineers. Designing a planning domain is a cumbersome task that requires extensive knowledge about the problem to be modeled. A solution to lessen this issue is to use domain learning techniques. Domain learners can extract domain models from scratch using the information from previously solved planning problems. Actual state-of-the-art domain learners show deficiencies with the expressivity of the planning domains learned, showing problems when learning domains with relations between the elements of the domain.

In order to address this issue, we present the PlanMiner domain learning algorithm. PlanMiner takes a set of observations as input (in the form of plan traces) and extracts a set of numeric action models from them. PlanMiner is able to learn arithmetic and relational expressions between the numeric variables of the planning domain. Then, these expressions can be used to define the preconditions and effects of the action models. PlanMiner implements a hybrid learning method that combines hierarchically classification and techniques to achieve this. Expressions are learned using symbolic regression during a preprocessing step of the information contained in the input data. Once preprocessed the data, a classification algorithm is used in order to get PDDL numerical action models from them. PlanMiner has been tested using a variety of benchmark domains extracted from the International Planning Competition (IPC). The learned action models were not only measured with a series of literature metrics but their problem-solving ability was also tested.

To the best of our knowledge, the main contribution of this paper to the domain learning area is the definition of a new domain learning technique with the ability to discover relations between numeric variables in a PDDL [10] planning domain. Another contribution of our work is that PlanMiner is designed to learn from partially-known plan traces. PlanMiner was tested using data with uncertainty to prove its resistance to low-quality data. Results showed that the algorithm presented in this paper is able to learn under high levels of data incompleteness.

The rest of the paper is organized as follows. Section 2 will cover previous works related to the one here presented. In section 3 we will explain every background concept needed to understand how our solution works and how we pose the domain learning problem. Section 4 presents and explains the description of our domain learner algorithm. Section 5 contains the experimental setup and the results of our domain learning solution. Finally, section 6 presents future work and conclusions.

## 2 Related Work

Domain learning is an open problem widely addressed by several approaches [17] in the field of automated planning [26]. These approaches can be sorted by the requirements of the domains that are trying to learn (such as domains with continuous numerical variables or temporal constraints), the source, type, and quality of the input data, or the representation language of the planning models (like PDDL,

PDDL+ [8] or OCL [3]). Comparing different approaches is difficult, as the data requirements for each one may hugely differ and also because some metrics can not be applied to measure its results. As this work is based on the learning deterministic domains from plan traces with partially-known information, we will discuss in the next lines only similar approaches.

ARMS [32] learns STRIPS planning domains from partially-known intermediate states. ARMS generates a set of logic formulas and defines a weighted propositional satisfiability problem. Then using a MAX-SAT [12] solver ARMS obtains the best subset of logic formulas that define the action models of the domain to be learned. LAMP [34] contemplates a similar approach, but instead of a MAX-SAT solver uses Markov logic networks to find the most suitable formulas to define the action models. AMAN [33] finds a domain model that best explains a set of plan traces by fitting a collection of models to each one. Using a different set of traces to evaluate the models by calculating a predefined reward function that updates the weight of the learned models. In the end, the model with the highest weight is selected as the final model.

Opmaker [24] is a solution to induce operators from action sequences, domain knowledge and user interaction. For each action to be learned, it asks to input, if needed, the target state that would be achieved after the action execution, thus creating its operator schema step-by-step. It is implemented inside a graphic tool called Graphical Interface for Planning with Objects (GIPO), which facilitates user interaction [16]. Opmaker2 [23] improves Opmaker by eliminating the necessity for intermediate state information. Opmaker2 is able to automatically infer the information provided by the user, and then it proceeds in the same fashion as Opmaker. Opmaker2 computes the intermediate states by using a combination of heuristics and inference from the partial domain model and input data.

LOCM [6] reduces the input needed to learn planning domains to only a set of plans without information about the intermediate states of the execution. LOCM uses finite state machines to learn static relations between the objects of the actions' headers of the input plans. These statics relations are used to create the action models. NLOCM [11] improves the LOCM algorithm by including a series of procedures to learn action costs. This is achieved by pairing a plan cost to every input plan and using constraint programming to infer the cost of each action.

FAMA [1] implements a new approach that makes flexible the needed input to learn planning domains. The input needed to learn domains using this algorithm ranges from a set of plans, with its initial and goal states, to only a pair of initial and goal states without any actions. The learning is achieved by creating a hybrid process to learn incomplete actions and simulate intermediate states to feed the plan traces on the fly.

Finally, Suárez-Hernández et al. [29] presents STRIPS-based compilation to learn action signatures from minimal information with an SAT-based [5] planning system.

Each of these approaches focuses on learning STRIPS-like action models but put aside the expressivity of the learned domains. Domain learners able to deal with expressions (either relational or arithmetic) are scarce in the state of the art. E.g. Jiménez et al. [20] addresses this issue with a domain learner able to fit a regression tree for each numerical function. These trees are converted into conditional structures by assigning values to the numerical attributes of the domain's actions (called fluents) according to the values of the rest of attributes. The main

issue of this approach is that it can not generalize the numeric expressions learned, creating a conditional clause for each possible value a numeric fluent can take.

## 3 Background

In this section, we will introduce the background knowledge needed to fully understand this paper. This background knowledge will cover concepts about action schemes and plan traces used in the learning process. Finally, we will define how we address the learning problem.

### 3.1 Action schema

A PDDL action schema can be defined as a tuple $< Header, Pre, Eff >$ where *Header* contains the name and parameters of the action, *Pre* are the preconditions that must be true to allow the execution of the action –that is what elements hold in the state to be able to apply the action–and *Eff* the effects of the action in the world after being executed –namely, what elements change as a result of applying that action.–Preconditions and effects are defined as a conjunction of predicates. A predicate is a statement of the form $p(arg_1, arg_2, ..., arg_n)$ where $p$ is the name of the predicate and $arg_i$ an object of the world. Each predicate has a value associated: *True* or *False* in the case of logical predicates, or a numerical value in the case of numerical predicates.

PDDL codes the expressions (relational or arithmetic) following the mathematical prefix notation. While this notation can be cumbersome to read by a human, it relieves the task of parsing the expression by a computer. Arithmetic/relational expressions are constructed as $(Op\ A\ B)$ where $Op$ is a relational or arithmetic operator and $A$ and $B$ the operands of the expressions. Relational operators contemplated by PlanMiner are: $=, <, \leq, >, \geq$. On the other hand, PDDL uses the basic arithmetic operators when defining arithmetic expressions. The operands used in the expressions can be numeric fluents, constant values or other arithmetic expressions. PDDL implements some operators to signal if a numerical predicate $A$ must be increased, decreased or assigned to a given value $B$.

### 3.2 Plan traces

A plan trace is an ordered set of grounded actions (plan) plus a collection of associated states. A plan is the sequence of steps necessary to achieve a goal. The associated states are snapshots of the world in a given point during the execution of the plan. A world state is represented as a conjunction of predicates using first-order logic. Each action has two associated states: one that represents the world at the start point of the action (pre-state) and other for the end point of the action (post-state). States' observations can be partially taken. A partial observation of a state can lack several predicates (even all) of the world at the moment of making the observation.

Table 1 shows an example of a plan trace extracted from a plan obtained from the Rovers [21] domain. *Rovers* is a planning domain presented in the IPC

| **Start** | **End** | *Action* |
|---|---|---|
| **0** | **1** | *(goto rov1 wp1 wp2)* |
| **1** | **2** | *(goto rov1 wp2 wp3)* |

(a) Plan.

| **Index** | *Predicates* |
|---|---|
| **0** | *(at rov1 wp1)* $\wedge$ *(¬ (at rov1 wp2))* $\wedge$ *(¬ (at rov1 wp3))* $\wedge$ *(¬ (scanned wp3))* $\wedge$ *(= (bat_usage rov1) 3)* $\wedge$ *(= (energy rov1) 450)* $\wedge$ *(= (dist wp1 wp2) 50)* $\wedge$ *(= (dist wp2 wp3) 80)* |
| **1** | *(¬ (at rov1 wp1))* $\wedge$ *(at rov1 wp2)* $\wedge$ *(¬ (at rov1 wp3))* $\wedge$ *(¬ (scanned wp3))* $\wedge$ *(= (bat_usage rov1) 3)* $\wedge$ *(= (energy rov1) 300)* $\wedge$ *(= (dist wp1 wp2) 50)* $\wedge$ *(= (dist wp2 wp3) 80)* |
| **2** | *(at rov1 wp3)* $\wedge$ *(¬ (at rov1 wp1))* $\wedge$ *(¬ (at rov1 wp2))* $\wedge$ *(¬ (scanned wp3))* $\wedge$ *(= (bat_usage rov1) 3)* $\wedge$ *(= (energy rov1) 60)* $\wedge$ *(= (dist wp1 wp2) 50)* $\wedge$ *(= (dist wp2 wp3) 80)* |

(b) States list.

Table 1: Extract of a plan trace. The trace is an extract of a solution plan for a specific problem in the Rovers domain. In this plan, a rover (rov1) moves from wp1 to wp3, traversing wp2. Subtable (a) displays the plan (sequence of actions) executed, Start and End columns display the index of associated states before and after applying a given action. Subtable (b) shows the set of intermediate states associated with each action of the plan during the execution. Index references a state in the Start/End columns of Subtable (a). Note that the same state may appear fulfilling different roles for different actions (pre-state or post-state).

where a group of rovers must traverse an alien planet taking samples in different waypoints scattered around the planet. The actions defined in the domain include travel between waypoints, scan a waypoint searching for samples, and use the rover's solar panels to recharge its batteries. *Rovers* consume battery whenever they move or use its tools to scan a waypoint. Battery use when moving depends on the distance between waypoints.

### 3.3 Problem definition

PlanMiner addresses the problem of action schema learning by transforming it to a classification problem [18]. The idea behind this transformation is as follows: In order to characterise an action we need to know (a) what elements hold in the state to be able to apply the action (preconditions) and (b) what elements that were in the state before applying the action change as a result of applying that action (effects). (a) Obtaining the preconditions can be approached as a classification problem since what we are looking for are what elements are common to all the states where that action was applied, i.e., to all the pre-states. Therefore, it consists of finding all features that characterize the pre-states of a given action. (b) Obtaining the effects also poses a problem of classification since we want to find which features belong to the post-states. But, unlike (a), here we will eliminate from the description those features that did not change with respect to the pre-states, since the effects are aimed to represent the world change produced by the given action. In constrast to other techniques, classification techniques allow

PlanMiner to deal with uncertainty in the input data, as well as, categorical and numerical information. Classical statistical methods [22] would achieve the goal of learning a set of features, but when facing uncertainty or big sets of data this kind of techniques falls short. First-order inductive learning techniques [27] would let PlanMiner fit STRIPS-like models but these techniques are not able to deal with numerical information.

Summarizing the learning process, for each action, our solution groups the information of the states and then displays them in an attribute-value matrix. These matrices display as attributes the predicates of the states and as instances the states themselves. Each instance has a class label given his role in the Plan Trace (pre-state or post-state). Over these matrices, a pipeline of preprocessing and regression techniques is applied in order to discover new knowledge. This knowledge describes how the elements of the predicates of the states relate between themselves and are included in the matrices as new attributes of the datasets. At the end of the learning process, a classification algorithm extracts a set of hypotheses that contains the key features that fit the states of the world included in the datasets. Those hypotheses are finally used to fill the preconditions and effects of the action's schemes.

## 4 Methodology

The following section will describe the PlanMiner domain learning algorithm. First, the general outline of the learning process will be described, presenting an illustrative example using a well-known benchmark planning domain. Then, each component of PlanMiner will be described in detail.

### 4.1 Overview

Our domain learner has been designed to learn action schemes with relational and arithmetic relations from a set of input plan traces with partially-known states. The learning process can be summarized in the following steps:

1. **Dataset Extraction.** Takes a set of input plan traces and outputs a collection of datasets. Data structures used as input in typical machine learning algorithms are different from the PDDL format, so a translation process must be carried on to convert the input plan traces into datasets.
2. **Discovery of new information.** Taking the information contained in the datasets generated in the previous step, PlanMiner applies symbolic regression techniques to generate new knowledge and enrich them. This step is crucial, because to be able to learn arithmetic and relational expressions new knowledge has to be explicitly encoded in the datasets.
3. **Classification models acquisition.** Using the datasets as input, PlanMiner relies on the use of a classification algorithm to fit a classification model for each one. The hypotheses contained in the classification models define the key features to model a set of intermediate states.
4. **Planning domain generation.** Finally, the classification models are processed to get a set of action models. In order to obtain an action schema, preconditions

Action: *(goto rov1 wp1 wp2)*

- **pre-state:** *(at rov1 wp1)* ∧ *(¬ (at rov1 wp2))* ∧
  *(¬ (at rov1 wp3))* ∧ *(¬ (scanned wp3))* ∧
  *(= (bat_usage rov1) 3)* ∧ *(= (energy rov1) 450)* ∧
  *(= (dist wp1 wp2) 50)* ∧ *(= (dist wp2 wp3) 80)*
- **post-state:** *(¬ (at rov1 wp1))* ∧ *(at rov1 wp2)* ∧
  *(¬ (at rov1 wp3))* ∧ *(¬ (scanned wp3))* ∧
  *(= (bat_usage rov1) 3)* ∧ *(= (energy rov1) 300)* ∧
  *(= (dist wp1 wp2) 50)* ∧ *(= (dist wp2 wp3) 80)*

Fig. 1: State transition of the *(goto rov1 wp1 wp2)* extracted from Table 1 plan trace.

and effects are extracted from each classification model. The final output of PlanMiner is a PDDL domain obtained by joining the learned action models.

In order to illustrate the whole learning process, we will show various examples based on *Rovers* domain (Table 1).

### 4.2 Dataset extraction

The format of the planning information contained in the input plan traces is not usable by any standard classification or regression algorithm. In order to overcome this, our approach takes the actions and states contained in each plan trace provided as input and adapts it to a typical classification input data structure. To achieve this, first our procedure takes each action $a$ contained in the input plan traces and extracts a *state transition* $(s_1, a, s_2)$. $s_1$ is a snapshot of the world just before executing the action (*pre-state*), while $s_2$ is a observation of the world just after executing the action (*post-state*). A given state can be the pre-state in a state transition and a post-state in a different one. In Figure 1 we show an example a state transitions for the action *(goto rov1 wp1 wp2)*. In this example, we can see how the states relate to the action of the example plan trace.

Once the state transitions for a given action have been extracted, the algorithm proceeds to calculate the schema form [32] of every state. Schema forms are calculated by selecting a state transition and taking each instance of the parameters in its action and substituting it by a given token every time it appears as an argument in any of the predicates of its associated states. When every parameter has been substituted by a token, *irrelevant predicates* are erased from the states. Irrelevant predicates are predicates that have not undergone at least one substitution during the translation to schema form. An exception of this rule are the predicates with no arguments, which always are considered relevant. As will be explained in section 4.4, the classification algorithm will select among the relevant predicates in order to keep the ones needed to model the states. The example showed in Figure 2 displays the schema form of the states associated with the generic action *(goto ?arg1 ?arg2 ?arg3)* action. As can be seen, the actions have been equated with the independence of their parameters.

For each different action in the plan traces, a dataset is created. Two actions are different if its headers (the action's name plus arguments after applying the

Action: *(goto ?arg1 ?arg2 ?arg3)*

- **pre-state:** *(at ?arg1 ?arg2)* ∧ *(¬ (at ?arg1 ?arg3))* ∧
  (¬ (at ?arg1 wp3)) ∧ (¬ (scanned wp3)) ∧
  *(= (bat_usage ?arg1) 3)* ∧ *(= (energy ?arg1) 450)* ∧
  *(= (dist ?arg2 ?arg3) 50)* ∧ (= (dist ?arg3 wp3) 80)

- **post-state:** *(¬ (at ?arg1 ?arg2))* ∧ *(at ?arg1 ?arg3)* ∧
  (¬ (at ?arg1 wp3)) ∧ (¬ (scanned wp3)) ∧
  *(= (bat_usage ?arg1) 3)* ∧ *(= (energy ?arg1) 300)* ∧
  *(= (dist ?arg2 ?arg3) 50)* ∧ (= (dist ?arg3 wp3) 80)

Fig. 2: Schema form of a *(goto ?arg1 ?arg2 ?arg3)* action before erasing irrelevant predicates (underlined).

| (at ?arg1 ?arg2) | (at ?arg1 ?arg3) | (bat_usage ?arg1) | (energy ?arg1) | (dist ?arg2 ?arg3) | (scanned ?arg3) | *Class* |
|---|---|---|---|---|---|---|
| *True* | *False* | 3 | 450 | 50 | $MV$ | $pre-state$ |
| *False* | *True* | 3 | 300 | 50 | $MV$ | $post-state$ |
| *True* | *False* | 3 | 300 | 80 | *False* | $pre-state$ |
| *False* | *True* | 3 | 60 | 80 | *False* | $post-state$ |

Table 2: Dataset associated with the *(goto ?arg1 ?arg2 ?arg3)* action. Table shows how the state transitions of the *(goto ?arg1 ?arg2 ?arg3)* actions defined in Table 1 are displayed as an attribute-value matrix. Each state in the state transitions is included as an instance in the dataset, where its predicates are displayed as attributes, and the class labels define its role in the state transition. Note that state 1 (Table 1) appears twice in the dataset (instances 2 and 3) with different values and different class label.

schematization process), are different. The datasets contain the information coded in the states of the state transitions whose attributes are the predicates of the states. Each instance of a dataset is a state, and its values are the values associated with each predicate: a logical value if the predicate is a logic value or a number if the predicate is numerical. The instances of the dataset are categorised by assigning them a class' label given by its relation with a given action (pre-state or post-state) with the action whose dataset is being modelled, creating a binary classification problem. In order to fill each dataset, the state transitions associated with a given action are taken, and its states are displayed as instances of the dataset. These instances are labelled given their role in a certain state transition, which may lead to the same state to appear with slightly different information in several instances of the dataset. In those cases where a predicate that is modelled as an attribute in the dataset doesn't appear in a given state, the value assigned in its instance is a missing value ($MV$) token. For example, the predicate (scanned ?x) is part of the relevant predicates of the action *(goto ?arg1 ?arg2 ?arg3)*, but is missing in many of the concrete state transitions of such action in the plan traces. Therefore its absence in such states is represented as an MV (see Table 2). This leads to interpret the lack of a value in an instance by following the open world assumption [28], meaning that if there is a missing value in an instance of a state, it is considered as unknown instead of false, so it can not be evaluated.

### 4.3 Discovery of new information

For a human with some expert knowledge about the *Rovers* planning problem, extracting a model for the pre-states and the post-states presented in Table 2 would be trivial. For example, it is easy to detect that for the *(goto ?arg1 ?arg2 ?arg3)* action its pre-states have to contain a predicate *(at ?arg1 ?arg2)* that must be *True* (It appears in every pre-state and changes to false in it's associated post-states), that the predicate *(scanned ?arg3)* can be deprecated (It is missing in half of the instances of the dataset and remains constant over the execution of the action when it appears) or that *(energy ?arg1)* decreases whenever the action is executed. With the information contained in the dataset, only the first two hypotheses can be extracted as features of the examples using a classification algorithm. In order to let a classification algorithm learn the rest of the hypothesis (I.E. how *(energy ?arg1)* changes) of the problem, new information must be discovered and explicitly encoded.

The only information about the ontology of the problem accessible to PlanMiner is the actions' headers contained in the plan traces, which presents a big handicap when discovering new information. Creating new information using brute force is not a viable option because there is a risk of increase without control the size of the learning problem. Also, when trying to learn relations between the elements of the dataset (and hence the predicates of the planning problem) an excessive creation of knowledge will lead to the emergence of spurious relations. Spurious relations contain useless information that makes the resolution of the learning problem very difficult. To overcome this, we divide the process of discovering new information in 3 steps:

1. Calculation of the difference between the numerical attributes of the dataset before and after executing an action. This step will produce new synthetic attributes (containing information about how a variable changes throughout the execution of a plan) that will be added to the dataset.
2. Fitting of arithmetic expressions that model the differences of the numerical attributes.
3. Discovery of the relational expressions that link the different elements of the problem.

Before advancing to the next step the new information is filtered to detect useless or redundant information. This helps to keep under control the over-information produced in every step.

#### *4.3.1 Numerical attributes changing.*

As a beginning step prior to being able to learn complex information, we calculate the set of $\Delta$ values associated with each numerical attribute of the dataset. $\Delta(attribute)$ is defined as

$$\delta_i \in \Delta : \delta_i = x_{pre,attribute} - x_{post,attribute}$$

where *pre* and *post* are the instances of the states associated with the *ith* state transition. If $x_{pre,attribute}$ or $x_{post,attribute}$ is missing, $\delta_i$ can not be calculated and

| $\Delta(bat_usage?arg1)$ | $\Delta(energy?arg1)$ | $\Delta(dist?arg2\ ?arg3)$ |
|---|---|---|
| $3-3=\mathbf{0}$ | $450-300=\mathbf{150}$ | $50-50=\mathbf{0}$ |
| $3-3=\mathbf{0}$ | $300-60=\mathbf{240}$ | $80-80=\mathbf{0}$ |

Table 3: $\Delta$ values extract of Table 2 dataset.

is substituted by a missing value token. Table 3 shows an example of the calculation of $\Delta$ values.

Once the $\Delta$ values have been obtained, we can discriminate those attributes that contain helpful information and are worthy of further exploration during the information discovery process. A $\Delta(attribute)$ is irrelevant to the learning process if each $\delta_i$ is equal to 0. An irrelevant $\Delta(attribute)$ means that its associated attribute is not affected by a given action and are discarded before beginning with the next step. Relevant $\Delta$ values are included in the dataset as a new attribute.

*4.3.2 Arithmetic expressions fitting.*

$\Delta$ values provide information about how an attribute changes when applying an action, but only when this information changes on a fixed value (linear functions). Dealing with more complex changes in the attributes (those that, for example, require algebraic functions to be explained) require new (more complex) solutions. The idea behind our solution goes through taking a $\Delta$ value and, using regression techniques, try to fit an arithmetic expression using the rest of attributes of the problem as predictive variables of the formula fitted. Among the multiple fields of machine learning, regression analysis [4] is the field in charge of predicting a continuous-valued output from a set of continuous variables. Our solution is interested in those regression algorithms that output the most interpretable models possible. Standard regression techniques generate regression models based on linear regression expressions and, in many cases, the relationships established in planning domains between numerical predicates are not linear, which means that this approach is not adequate for PlanMiner. This reason led us to develop our work using symbolic regression techniques.

Symbolic regression (SR) is a type of regression analysis [30] which searches the space of arithmetic expressions trying to find a suitable formula that fits a goal set of values. Genetic programming [19] (GP) is the de facto technique used to solve SR problems, but GP algorithms tend to find alternative formulas far from the original. Plus, the random component of the genetic algorithms used in the learning process hinder the replay of the results. These issues led us to check other SR approaches [31].

PlanMiner implements an SR algorithm using an informed graph search algorithm [13] with the objective of incrementally building a valid expression that fits the problem's goal. An overview of the SR algorithm implemented can be seen in Algorithm 1. Summarizing, the algorithm takes as the root node an empty expression (∅) and a set of numerical attributes (dataset dat), and adding new operators and operands to existing nodes, creates new states that represent new formulas. New formulas are added to a pool of created formulas. The algorithm selects a new

**Algorithm 1** Symbolic regression algorithm

**Input**: Ø formula, Dataset dat
**Output**: Best Formula Found

```
 1: open_set := {}
 2: successors := {}
 3: open_set := generate_formulas(Ø, dat).
 4: end := False.
 5: while ¬ end do
 6:    current := best(open_set) .
 7:    if h(current, dat) < stop_value then
 8:       best_sol := current.
 9:       end := True.
10:    else
11:       successors := generate_formulas(current, dat).
12:       open_set := open_set ∪ successors.
13:    end if
14: end while
15: return best_sol
```

formula from the pool and repeats the process until meeting a suitable formula for the set of goal values.

In order to guide the search, for a state that represents the arithmetic expression $x$, the algorithm uses an heuristic function

$$h(x, Goal) = 100\% * MAPE(pred(x), Goal) * |x|$$

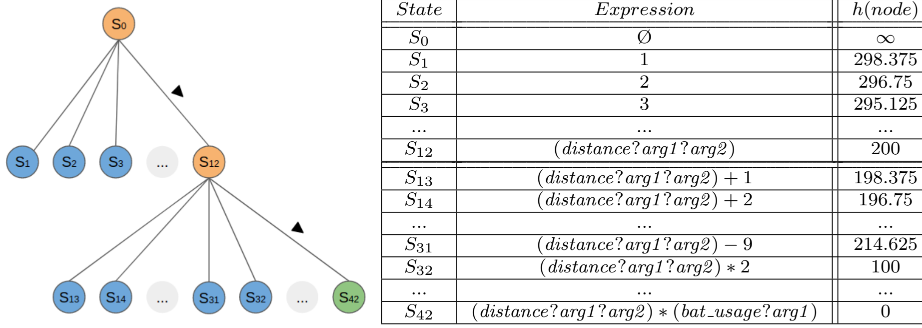
MAPE (mean absolute percentage error) is a measure of difference between two continuous variables (pred(x) and Goal) and is calculated as

$$MAPE(pred(x), Goal) = \frac{\sum_{i=1}^{|Goal|} |pred(x)_i - goal_i|}{Goal_i}$$

where $pred(x)$ are the forecasted values obtained with the arithmetic expression $x$ paired with the *Goal* set of values. Finally, $|x|$ is the size of the arithmetic expression (the number of operands and operators in the expression).

New states are created from a parent state by adding an operand paired with an operator to the arithmetic expression defined in it. Arithmetic operators used by the regression algorithm are +,-,*,/, while operands may be a constant integer or an attribute from the dataset. For each iteration of the algorithm, every possible combination of operators and operands is used to create new successor states. The search ends when a state with a heuristic value close zero is found or a certain amount of time has passed. During the experimentation of this work, the authors set the algorithm stop values as 0.02 for the heuristic and 300 seconds until timeout.

If the search algorithm does not find suitable expression in the given time the $\Delta$ value associated is erased from the dataset. This is due to the supposition that if the search algorithm is incapable of finding a viable pattern in the elements of the $\Delta$ value, then it is full of arbitrary values and no useful information can be obtained from it. If a suitable formula is found, the $\Delta$ value used as the goal is updated with it in the dataset. Figure 3 presents a brief example of $\Delta(energy?arg1)$ formula's search process.



| *State* | *Expression* | *h(node)* |
|---|---|---|
| $S_0$ | Ø | ∞ |
| $S_1$ | 1 | 298.375 |
| $S_2$ | 2 | 296.75 |
| $S_3$ | 3 | 295.125 |
| ... | ... | ... |
| $S_{12}$ | $(distance?arg1?arg2)$ | 200 |
| $S_{13}$ | $(distance?arg1?arg2) + 1$ | 198.375 |
| $S_{14}$ | $(distance?arg1?arg2) + 2$ | 196.75 |
| ... | ... | ... |
| $S_{31}$ | $(distance?arg1?arg2) - 9$ | 214.625 |
| $S_{32}$ | $(distance?arg1?arg2) * 2$ | 100 |
| ... | ... | ... |
| $S_{42}$ | $(distance?arg1?arg2) * (bat_usage?arg1)$ | 0 |

Fig. 3: Example graph for the search process of the $\Delta(energy?arg1)$ (Table 3) expression. Visited nodes (orange), expanded nodes (blue) and goal nodes (red) are shown in the figure. Formulas created in each node are displayed in the right table of the figure. The heuristic values are calculated using the information contained in Table 2.

#### *4.3.3 Relational expressions creation.*

Before finalizing the knowledge discovering process, the relational expressions are created following a straightforward procedure. This procedure takes two numerical attributes of the dataset and creates a new one by pairing them with a relational operator. The relational operators used by our algorithm are $=$, $<$ and $>$.

Finally, the new logical attributes must be checked if they are relevant. The relevance of a logical attribute is calculated by testing if every value in the new attribute is different. If the truth value of the relational expression remains constant, i.e. all rows in the attribute have the same value (either true or false), the attribute can be discarded, as there's no useful information included in it. Relevant attributes are included in the dataset (Table 4).

### 4.4 Classification models acquisition

The datasets extracted from the plan traces, enriched with the new information discovered and added to them, contain information about the examples that represent states of the world. These examples can be generalized into a collection of features with the information needed to represent all of them. Classification techniques allow PlanMiner to retrieve every shared feature from each state and fully recreate an essential state that models all.

Classification is the subfield of machine learning focused on the search of features that define a collection of instances of a given class. Classification is a prolific field with several different techniques [18]. We are more interested in the interpretability of the classification models rather than its performance, so classifiers that create models easy to interpret by humans are very valuable to our solution. This discards the use of complex black-box models in favour of other types of machine learning classification models. A highly interpretable model eases the translation process of classification models to planning models in the later stages

| *(energy ?arg1)* | $\Delta(energy?arg1)$ | (>*(energy ?arg1)* $\Delta(energy?arg1)$) |
|---|---|---|
| 450 | 150 | *True* |
| 300 | 150 | *True* |
| 300 | 240 | *True* |
| 60 | 240 | *False* |

Table 4: New attribute (>*(energy ?arg1)* $\Delta(energy?arg1)$) created from the data contained in the dataset of Table 2.

```
IF
(at ?arg1 ?arg2) = True ∧ (at ?arg1 ?arg3) = False ∧
(bat_usage ?arg1) = 3 ∧
(> (energy ?arg1) Δ(energy ?arg1)) = True
THEN pre-state

IF
(at ?arg1 ?arg2) = False ∧ (at ?arg1 ?arg3) = True ∧
(bat_usage ?arg1) = 3 ∧
Δ(energy ?arg1) =
  (dist ?arg2 ?arg3) * (bat_usage ?arg1)
THEN post-state
```

Fig. 4: Classification models of the *(goto ?arg1 ?arg2 ?arg3)* action shown in Table 2 plus the information added in the different processes of Section 4.3.

of the learning process. Finally, we are interested in classification models that contain the maximum information possible. Among the different type of classification models, white-box [7] descriptive models best fits our requirements. Descriptive models contain all key features of the examples, rather than the minimum number of features that form other kinds of models, and white-box models are easy to interpret.

The domain learning process presented in this paper is independent of the classification algorithm used to extract the models. The only requisite set by our approach is that the classification algorithms must accept attribute-value matrices. To test this work we selected NSLV [9]. NSLV is a classification algorithm based on inductive rule learning [25]. Inductive rule learning is mainly used in problems where a single rule cannot model the whole problem. Given how we define the learning problem (a binary classification problem with two classes: pre-state and post-state) in this paper NSLV would output two rules for each dataset: One to model the pre-states and another one to model the post-states.

The rules learned by NSLV follow the structure detailed below:

$$\textbf{IF } C_1 \text{ and } C_2 \text{ and} \ldots \text{and } C_m \textbf{ THEN } Class \text{ is } B$$

$$\text{with } \textbf{weight w}$$

where $C_i$ is a sentence "$X_n$ is $A$", with $A$ a label (or a set of labels) of the domain of the variable $X_n$. Each $C_i$ is a feature of the examples extracted to model the class $B$. $X_n$ is an attribute of the problem. The labels $A$ of each variable depends on the type of attribute: True or False for logical attributes or a number for numerical attributes. The rules are weighted by counting the percentage of instances of the dataset covered by it.

| Parameter | Value |
|---|---|
| **Size of genetic population** | 40 |
| **Number of iterations** | 500 |
| **Mutation probability** | 0.01 |
| **Crossover probability** | 1 |

Table 5: NSLV parameter setting.

**IF**
(at ?arg1 ?arg2) = *True* ∧
(at ?arg1 ?arg3) = *False* ∧
(bat_usage ?arg1) = 3 ∧
(>(energy ?arg1) $\Delta$(energy ?arg1)) = *True*
**THEN** *pre-state*

→

(at ?arg1 ?arg2) ∧
(¬ (at ?arg1 ?arg3)) ∧
(= (bat_usage ?arg1) 3) ∧
(>(energy ?arg1)
(* (distance ?arg1 ?arg2)(bat_usage ?arg1)))

**IF**
(at ?arg1 ?arg2) = *False* ∧
(at ?arg1 ?arg3) = *True* ∧
(bat_usage ?arg1) = 3 ∧
$\Delta$ (energy ?arg1) =
(dist ?arg2 ?arg3) * (bat_usage ?arg1)
**THEN** *post-state*

→

(¬ (at ?arg1 ?arg2)) ∧
(at ?arg1 ?arg3) ∧
(= (bat_usage ?arg1) 3) ∧
(decrease (energy ?arg1)
(* (distance ?arg1 ?arg2)(bat_usage ?arg1)))

Fig. 5: Pre-state and post-state meta-model from the rules of Figure 4.

Starting from an empty ruleset and a training set of instances, a new rule that covers a subset of these instances is generated and added to the ruleset iteratively. NSLV uses a steady-state genetic algorithm (GA) to select which selection of tuples $<X_i, A>$ define the antecedent of the rule that fits the highest number of instances of the training set. The instances covered by this new rule are penalized by erasing them from the training set of instances. This penalization mechanism helps NSLV to guide the GA to learn new rules that explain uncovered instances of the dataset. Table 5 shows the parameters of NSLV during the rule learning process.

### 4.5 Planning domain generation

The final phase of the learning process consists of the conversion of the classification models (Figure 4) into planning actions. The implementation of this process is an *ad-hoc* procedure that depends on the classification algorithm used and the desired format of the output planning models. In our particular case, we want to translate a classification rule into a world's description written in PDDL. Given the similarities of both models (recall Section 3.1), this process is trivial. On the one hand, NSLV rule's antecedent contains a set of tuples $<X, A>$ joined by a conjunction operator. Each tuple represents the value A of the problem's attribute X. On the other hand, PDDL displays world states as a set of predicates linked by a conjunction operator(Figure 5).

As said earlier, tuples $<X, A>$ represent the essential predicates that define a state of the world. Those states contain the minimum essential information to model all the pre- or post-states for every appearance of the action of the plan traces. From these states, the preconditions and effects of the action are extracted.

**Preconditions:**
(at ?arg1 ?arg2)) $\wedge$
($\neg$ (at ?arg1 ?arg3)) $\wedge$
(=(bat_usage ?arg1) 3) $\wedge$
(>(energy ?arg1)
(* (distance ?arg1 ?arg2) (bat_usage ?arg1)))

**Effects:**
($\neg$ (at ?arg1 ?arg2)) $\wedge$
(at ?arg1 ?arg3) $\wedge$
(decrease (energy ?arg1)
(* (distance ?arg1 ?arg2) (bat_usage ?arg1)))

Fig. 6: Preconditions and effects learned from the models of Figure 5. As can be noted, preconditions are created directly from the pre-state model, while effects are the steps necessary to obtain the post-state model from it.

Action's preconditions (Figure 6) are the set of features of the world that must hold in order to apply the action, so it can be obtained directly by displaying the information of the pre-state model as a conjunction of predicates. Action's effects represent how the action changes the world after applying it, so they must be obtained by calculating the steps necessary to transform the pre-state into the post-state. These steps are the addition and deletion list of logical predicates and the assignment/increment/decrement of continuous values of numerical predicates. By comparing the pre-state and post-state we can check which logical predicates must be added (were false in pre-state and true in post-state) and deleted (were true in pre-state and false in post-state). Including increments and decrements of numerical fluents is a straightforward process as $\Delta$ values contain explicitly this information, so it only needs to translate them into the PDDL format.

## 5 Experiments and Results

This section is directed to present the results obtained from carrying out different experiments with the aim to prove the performance of the method presented in this work. To achieve so, this section is divided into two blocks: The first part explains in-depth the experimental setup and the metrics used during the experimental process, and the second part shows the results of these experiments and analyses the performance of PlanMiner in them.

### 5.1 Experimental Setup

A wide range of domains from the IPC was selected as the source of experimental input data. For each domain, a collection of 100 random-generated problems were solved. Then, these problems were used to obtain the input plan traces used in the experimental process. Problems were solved with Metric-FF planner [14]. To test the capabilities of our solution dealing with partially-known information, input plan traces were modified by including incompleteness in them. Incompleteness was included by randomly selecting a certain percentage of predicates from every state in every plan trace used as input and erasing them. In order to avoid randomness to

Table 6: Input domains characteristics. The first table shows the STRIPS domains used in the first part of the experimentation, the second table shows the numerical domains used in the second part of the experimentation. Domains characteristics (from left to right) are the number of actions of the domains, the number of parameters of the actions, the number of logic predicates and the number of parameters of the predicates. In the particular case of the numerical domains, the number of numerical fluents is included in the table.

| Domain | \|Actions\| | max action arity | \|logical predicates\| | max predicate arity |
|---|---|---|---|---|
| BlocksWorld | 4 | 2 | 5 | 2 |
| Depots | 5 | 4 | 6 | 6 |
| DriverLog | 6 | 4 | 6 | 2 |
| ZenoTravel | 5 | 6 | 4 | 2 |

| Domain | \|Actions\| | max action arity | \|logical predicates\| | \|numerical predicates\| | max predicate arity |
|---|---|---|---|---|---|
| Depots | 5 | 4 | 6 | 4 | 2 |
| DriverLog | 6 | 4 | 6 | 4 | 1 |
| Rovers | 10 | 6 | 26 | 2 | 3 |
| Satellite | 5 | 4 | 8 | 6 | 2 |
| ZenoTravel | 5 | 3 | 2 | 8 | 2 |

affect the results, a 5 folds cross-validation is used in the learning process. The set of input problems were separated into 5 disjoint subsets (folds). 4 folds were used for learning a planning domain, while the remaining fold was reserved to be used with the VAL [15] tool during the validity measure. Experiments were run 5 times, changing the validation fold in every run. For a set of input problems, the result of the experimentation was the mean value of the measures for every run. In these experiments, The parameters of FF-Metric, VAL and the reference algorithms are set as default as noted by its authors during the whole experimentation process.

#### *5.1.1 Evaluation criteria*

The metrics used in the experimentation process are precision, recall, F-Score and, in the second battery of experiments, validity. Precision and recall [2] are defined as:

$$Precision = \frac{tp}{tp + fp}$$

$$Recall = \frac{tp}{tp + fn}$$

where *tp* is is the number of true positives (predicates that correctly appear in the action schema), *fn* is the number of false negatives (number of missing predicates), and *fp* is the number of false positives (number of extra predicates). Precision quantifies how many learned elements of the domain surplus. On the other hand, recall gauges the number of missing elements in the learned domain. Both precision and recall measures range from 1 (best) to 0 (worst).

F-Score is a measure of the domain's general performance. F-Score is the harmonic mean between precision and recall and is calculated as

$$F\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Overall, F-Score is a grade about how good is the learned domain on average. F-Score was used as opposed as domain's error rate in this experimentation. Error

Table 7: Precision, recall and F-Score experimental results. For each experiment mean and standard deviation of the 5 folds metrics values are shown.

| Domain | Incompleteness | μPrecision | σPrecision | μRecall | σRecall | μF-Score | σF-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.949 | 0.00 | 1.00 | 0.00 | 0.963 | 0.00 |
| | 10% | 0.962 | 0.01 | 1.00 | 0.00 | 0.970 | 0.005 |
| | 50% | 0.953 | 0.022 | 0.996 | 0.010 | 0.962 | 0.015 |
| | 90% | 0.884 | 0.024 | 0.969 | 0.021 | 0.912 | 0.014 |
| DriverLog | 0% | 0.918 | 0.017 | 0.980 | 0.018 | 0.948 | 0.017 |
| | 10% | 0.907 | 0.036 | 0.973 | 0.027 | 0.939 | 0.032 |
| | 50% | 0.895 | 0.032 | 0.966 | 0.023 | 0.929 | 0.026 |
| | 90% | 0.784 | 0.050 | 0.960 | 0.027 | 0.862 | 0.036 |
| Rovers | 0% | 0.775 | 0.000 | 1.000 | 0.000 | 0.873 | 0.000 |
| | 10% | 0.774 | 0.003 | 1.000 | 0.000 | 0.872 | 0.002 |
| | 50% | 0.458 | 0.045 | 0.867 | 0.022 | 0.599 | 0.039 |
| | 90% | 0.350 | 0.031 | 0.809 | 0.036 | 0.488 | 0.027 |
| Satellite | 0% | 1.000 | 0.000 | 1.000 | 0.000 | 1.000 | 0.000 |
| | 10% | 0.993 | 0.014 | 0.993 | 0.014 | 0.993 | 0.009 |
| | 50% | 0.857 | 0.036 | 0.853 | 0.033 | 0.852 | 0.024 |
| | 90% | 0.852 | 0.079 | 0.833 | 0.086 | 0.844 | 0.082 |
| ZenoTravel | 0% | 0.812 | 0.000 | 1.000 | 0.000 | 0.896 | 0.000 |
| | 10% | 0.703 | 0.026 | 1.000 | 0.000 | 0.825 | 0.018 |
| | 50% | 0.620 | 0.075 | 0.715 | 0.021 | 0.662 | 0.042 |
| | 90% | 0.521 | 0.097 | 0.676 | 0.021 | 0.585 | 0.065 |

rate as defined in [34] is calculated by counting the number of wrongly learned predicates (extra or missing predicates) and dividing them by the total number of possible predicates in a given action. When taking into account arithmetic expressions the total number of possible predicates is infinite (as there is an infinite combination of operators and operands to create arithmetic expressions), making it unfit to measure our experimentation. Precision, recall and F-Score were chosen because they were the metrics that better fit our goal of measuring the learned domains and test its quality by comparing them with the original domains used to create the input data.

Validity is a metric that checks the learned domain's problem-solving capabilities. Validity is calculated by taking a plan obtained with a handmade domain and trying to replay it with a learned domain. A successful recreation of the plan (final state of the recreation equal to problem's goal state) means that the domain is valid. Validity is measured using the automatic plan validation tool VAL [15] from the IPC.

## 5.2 Results and Discussion

### *5.2.1 Comparison with reference algorithms*

The reference algorithms selected to analyze the performance of PlanMiner are ARMS [32], AMAN [33], OpMaker2 [23] and FAMA [1]. Those are some of the most spread domain learning systems in the literature. These are approaches able to learn deterministic planning domains from certain levels of uncertainty in the input data, but they are able to only learn STRIPS-like actions. Thus we selected a set of domains of this type to compare the approaches. Table 6 shows the domains used during this step of the experimental process as well as its characteristics, and Figure 7 compares ARMS, AMAN, OpMaker2, FAMA and PlanMiner in terms of F-Score.

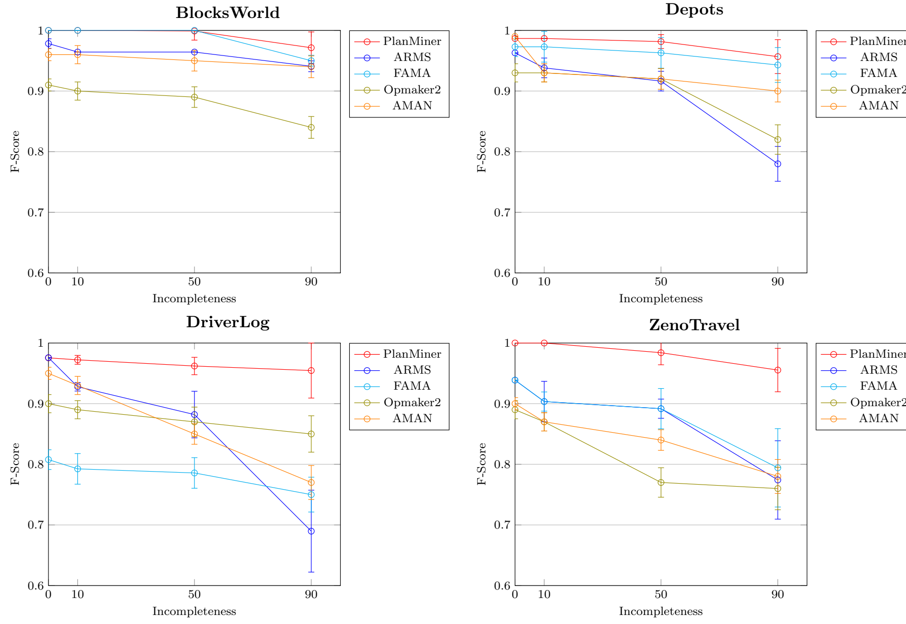



Fig. 7: F-Score comparison charts between PlanMiner and the reference algorithms. From top to bottom and left to right, results are shown of BlocksWorld, Depots, DriverLog and ZenoTravel domains.

At worst cases, PlanMiner equals the performance of the reference algorithms, but the differences between PlanMiner and the reference algorithms increase with higher levels of incompleteness. Results show that learning STRIPS-like domains, PlanMiner presents high resilience to incompleteness, maintaining stable F-Score values, in contraposition with the performance of some of the algorithms used.

Inspecting the results by domains we can observe that:

- In Blocksworld domain F-score above 0.9 points with every algorithm, with higher levels of incompleteness, only OpMaker2 drops below that threshold. PlanMiner presents perfect scores except for 90% incompleteness. FAMA, the closest algorithm in terms of performance, presents similar issues but shows the worst performance at 90% incompleteness (around 0.1 points of F-Score).
- In Depots domain PlanMiner presents a non-perfect F-Score rating. This is caused by the inclusion of bidirectional matrices that represent the problem's world, as PlanMiner learns some redundant information. An in-depth study of this issue will be discussed later in the next subsection, as well as a solution for it. Overlooking these errors, PlanMiner maintains a stable performance facing incompleteness, presenting problems only in the experiments with the highest number of missing data. Comparing with other approaches PlanMiner dominates them in every experiment, with FAMA presenting the narrowest gap between results (0.1 points at 90% incompleteness). ARMS, OpMaker2 and AMAN present an overall performance around 0.9 F-Score with important drops in the most complex experiments.

- In Driverlog domain PlanMiner shows a similar behaviour as the one viewed with the Depots domain, both in terms of performance and resilience to incompleteness. Driverlog domains use the similar bidirectional matrices that Depots, hindering the overall performance of PlanMiner. Reference algorithms present a worse performance than PlanMiner over the experimentation, with some algorithms presenting 0.2 or more points difference of F-Score at 90% incompleteness.
- In Zenotravel domain PlanMiner shows a clear dominance over the rest of reference algorithms. The closest algorithms in terms of performance are ARMS and FAMA, that show results 0.08 points below PlanMiner's results. This difference widens to 0.2 points in the most complex experiments. AMAN and OpMaker2 algorithms start with worse results, but they don't show falls so sharp.

#### *5.2.2 Numeric planning domains learning*

Results showed in tables 7 and 8 exhibit the overall performance of PlanMiner learning numerical domains. The domains used in this experimentation are the numeric versions of the Depots, DriverLog, Rovers, Satellite and ZenoTravel domains (table 6). Roughly, as can be seen, PlanMiner presents good results until levels of 50% of incompleteness. PlanMiner maintains F-Score levels above 0.8 even in the worst cases. Beginning with 50% of incompleteness these results worsen in the Rovers and ZenoTravel domains. Even so, PlanMiner shows recall levels almost flawless in most situations. The Precision results are, on average, lower than the recall results and then the F-Score results resent. PlanMiner is able to obtain valid domains with some levels of incompleteness in 4 out of 5 domains. Validity is the hardest metric to meet, as a single missing or extra effect in an action, can make the whole domain unable the reproduce the test plans.

For each domain used in the experimentation process, we will discuss its results separately in a proper subsection. Each subsection includes an example of an action learned. These examples highlight (**Bolded** predicates) the most common errors of the solution presented in this paper. These errors can be classified into two types: over-information errors and bias errors. Over-information errors arise when PlanMiner creates pointless expressions that relate various predicates during the information discovery steps. Bias errors occur when there is a lack in the variety of input data that causes to take as true very restrictive clauses. Over-information errors can be avoided by restricting the creation of some expressions by including expert information as input data. While increasing the variety of the data and including several different examples as input would eliminate bias errors.

*5.2.3 Depots*

```
Action:
  LIFT (?arg_0 - hoist ?arg_1 - crate ?arg_2 - surface
        ?arg_3 - place)
Precondition:
  (at ?arg_2 ?arg_3)
  (clear ?arg_1)
  (at ?arg_0 ?arg_3)
  (available ?arg_0)
  (at ?arg_1 ?arg_3)
  (on ?arg_1 ?arg_2)
Effects:
  (¬(available ?arg_0))
  (lifting ?arg_0 ?arg_1)
  (increase (fuel-cost) 1)
```

Depots' results show flawless recall scores. The errors found in the Depots domains are over-information errors. The predicates marked can be inferred from other predicates of the preconditions so it is not necessary to explicitly include them. For example, with the predicates $(at?arg_1?arg_3)$ and $(on?arg_1?arg_2)$ we would infer the predicate $(at?arg_2?arg_3)$, making it redundant and therefore erroneous. Although these errors are scattered in the whole domain and they reduce the precision of the domains and hence lower the F-Score of the domains, they do not affect domains' validity in any situation (even with the highest levels of incompleteness).

*5.2.4 DriverLog*

```
Action:
  DRIVE-TRUCK (?arg_0 - truck ?arg_1 - location ?arg_2 - location
               ?arg_3 - driver)
Precondition:
  (at ?arg_0 ?arg_1)
  (link ?arg_2 ?arg_1)
  (link ?arg_1 ?arg_2)
  (driving ?arg_3 ?arg_0)
Effects:
  (¬ (at ?arg_0 ?arg_1))
  (at ?arg_0 ?arg_2)
  (increase (driven) (time-to-drive ?arg_2 ?arg_1))
```

Like Depots, DriverLog domains present a light widespread problem with domains' precision. The source of these problems is how the connectivity in the world map is defined. To create a connectivity network between locations the domain uses the $(link?arg_1?arg_2)$ predicates. Given that this network is usually defined bidirectionally, for each pair of locations there are two symmetric predicates and PlanMiner can not discern that only one of these two predicates is needed to define the actions. This issue occurs to a lesser extent in the Depots domain and is the cause of the flaws in the STRIPS Depots and Driverlog domains. The addition of a single non-bidirectional connection between two locations would eliminate this problem from the learned domains. These errors do not invalidate the learned domains to reproduce the original plans used as input and therefore they are valid.

At 90% incompleteness, these errors and some recall errors invalidate the learned domains, even that the F-Score maintains above 0.86 points.

*5.2.5 Rovers*

```
Action:
  SAMPLE_SOIL (?arg_0 - rover ?arg_1 - store ?arg_2 - waypoint)
Precondition:
  (at_soil_sample ?arg_2)
  (available ?arg_0)
  (store_of ?arg_1 ?arg_0)
  (empty ?arg_1)
  (equipped_for_soil_analysis ?arg_0)
  (equipped_for_soil_analysis ?arg_0)
  (at ?arg_0 ?arg_2)
  (>= (energy ?arg_0) 3)

Effects:
  (¬ (at_soil_sample ?arg_2))
  (¬ (empty ?arg_1))
  (full ?arg_1)
  (have_soil_analysis ?arg_0 ?arg_2)
  (decrease (energy ?arg_0) 3)
```

The high number of predicates defined in the Rovers domain generates a high quantity of bias and over information errors in the learned domains. For example (*available?arg_0*) for a given rover is always true except when transmitting information to the rover's dock, and PlanMiner is not able to delete them. The predicate (*equipped_for_soil_analysis?arg_0*) is a bias error produced by the lack of different configurations of rovers in the input data. As every rover is equipped to execute every action, PlanMiner learns some spurious relations that lower the precision scores. In contraposition, we can see that recall scores maintain above 0.8 points even at 90% incompleteness. The low precision scores make F-Score resents and hinders the validity of the domains.

*5.2.6 Satellite*

```
Action:
  TURN_TO (?arg_0 - satellite ?arg_1 - direction ?arg_2 - direction)
Precondition:
  (pointing ?arg_0 ?arg_2 )
  (> (fuel ?arg_0) (slew-time ?arg_1 ?arg_2))
  (> (fuel-used) (slew-time ?arg_1 ?arg_2))
  (> (fuel ?arg_0)  (slew-time ?arg_1 ?arg_2))}*)
Effects:
  (¬(pointing ?arg_0 ?arg_2))
  (pointing ?arg_0 ?arg_1)
  (decrease (fuel ?arg_0) (slew-time ?arg_1 ?arg_2))
  (increase (fuel-used) (slew-time ?arg_1 ?arg_2))
```

Satellite starts with flawless results both in precision and recall, but they lower evenly in the absence of information. Action showed as an example is taken from a domain learned using input information with a 10% missing predicates. The lack of information leads PlanMiner to create relations that offer no knowledge, this is

not caused by the characteristics of the domain but is a deficiency of the learning process. These issues do not hinder the capabilities of PlanMiner to learn valid planning domains even at 50% of missing data. Finally, the results show that our approach achieves 0.84 points F-Score at the highest levels of incompleteness.

#### *5.2.7 ZenoTravel*

```
Action:
  FLY (?arg_0 - satellite ?arg_1 - direction ?arg_2 - instrument
    ?arg_3 - mode)
Precondition:
  (at ?arg_0 ?arg_1)
  (> (fuel ?arg_0) (*(distance ?arg_1 ?arg_2) (slow-burn ?arg_0)))
  (> (fuel ?arg_0) (distance ?arg_1 ?arg_2))
  (> (fuel ?arg_0) (distance ?arg_2 ?arg_1))
  (> (fuel ?arg_0) (fast-burn ?arg_0))
  (> (fuel ?arg_0) (slow-burn ?arg_0))
Effects:
  (¬(at ?arg_0 ?arg_1))
  (at ?arg_0 ?arg_2)
  (decrease (fuel ?arg_0)
      (*(distance ?arg_1 ?arg_2 ) (slow-burn ?arg_0)))
  (increase (total-fuel-used)
      (*(distance ?arg_1 ?arg_2) (slow-burn ?arg_0 )))
```

PlanMiner presents some problems with spurious arithmetic and logic relations when learning ZenoTravel domains. These errors appear with the existence of predicates that represent bidirectional connectivity networks (in this particular case $(distance?arg_1?arg_2)$), the main difference between the issues found in Driverlog is that these predicates are numerical predicates rather than logical predicates. This leads to PlanMiner to adjust twice the number of arithmetic and relational expressions. These expressions are virtually equal, but our approach can not discriminate between them. ZenoTravel presents another characteristic that hinders the performance of PlanMiner. This characteristic is the existence of equivalent predicates like $(slow - burn?arg_0)$ and $(fast - burn?arg_0)$. Although these predicates do not have the same information, they represent the same matter. This leads PlanMiner to create more spurious relations. This situation lowers the overall precision score of the learned domains, but do not affect recall scores. PlanMiner is able to obtain flawless recall scores, even in the presence of missing information. This leads to PlanMiner able to maintain the validity of the learned domains even when the number of spurious relations is higher.

## 6 Conclusions and Future Work

In this paper, we have developed a new planning domain learner, named PlanMiner, that uses several machine learning techniques to learn the planning domains with relational and numerical expressions in the preconditions and effects from plan traces with partially-known states. Our domain learner was measured with a battery of experiments that tested the learned domain's F-Score, precision, recall, and validity by solving a set of problems from benchmark planning domains and by comparing it with state-of-the-art domain learning algorithms. The results

Table 8: Validity results. For each experiment mean and standard deviation of the 5 folds of the plans solved are shown. If at least 50% of the test plans are valid, the domain is valid. Validity is calculated by majority vote between folds.

| Domain | Incompleteness | $\mu$Plans solved | $\sigma$Plans solved | Validity |
|---|---|---|---|---|
| Depots | 0% | 1.000 | 0.000 | ✓ |
| | 10% | 1.000 | 0.000 | ✓ |
| | 50% | 1.000 | 0.000 | ✓ |
| | 90% | 0.910 | 0.124 | ✓ |
| DriverLog | 0% | 1.000 | 0.000 | ✓ |
| | 10% | 0.933 | 0.149 | ✓ |
| | 50% | 0.800 | 0.447 | ✓ |
| | 90% | 0.000 | 0.000 | ✗ |
| Rovers | 0% | 0.966 | 0.0745 | ✓ |
| | 10% | 0.000 | 0.000 | ✓ |
| | 50% | 0.000 | 0.000 | ✗ |
| | 90% | 0.000 | 0.000 | ✗ |
| Satellite | 0% | 1.000 | 0.000 | ✓ |
| | 10% | 0.800 | 0.447 | ✓ |
| | 50% | 0.75 | 0.426 | ✓ |
| | 90% | 0.000 | 0.000 | ✗ |
| ZenoTravel | 0% | 1.000 | 0.000 | ✓ |
| | 10% | 0.833 | 0.372 | ✓ |
| | 50% | 0.333 | 0.447 | ✗ |
| | 90% | 0.000 | 0.000 | ✗ |

obtained show that our solution is able to learn valid planning domains, even with high levels of incompleteness in the input states.

There's a wide variety of work lines to improve our domain learning technique in the future. First, we want the improve further the expressivity of the domains learned. This will lead to exploring the possibility of learning planning domains with durative actions. Durative actions are actions that take a certain time to be completed, in contrast to the actions learned in this paper that are considered to be instantaneous. Second, we are considering including noise in the plan traces. The inclusion of noise is difficult, as there exist a wide variety of different techniques in the learning process that have to be attuned in order to achieve satisfactory results. Finally, we are aware of the weak points of our solution. The issue of the spurious information included in the learned domains can hinder the results of PlanMiner, and we are considering a series of techniques to lessen the problem.

**Acknowledgements** This research is being developed and partially funded by the Spanish MINECO R&D Project TIN2015-71618-R and RTI2018-098460-B-I00

## 7 Declarations

### 7.1 Funding

This research is being developed and partially funded by the Spanish MINECO R&D Project TIN2015-71618-R and RTI2018-098460-B-I00

### 7.2 Conflicts of interest/Competing interests

Authors state that there is no conflict of interest

### 7.3 Availability of data and material

Data used in experimentation are available at http://www.icaps-conference.org/index.php/Main/Competitions

### 7.4 Code availability

Code presented in this work is available in https://github.com/Leontes/PlanMiner

## References


1. Aineto, D., Celorrio, S.J., Onaindia, E.: Learning action models with minimal observability. Artificial Intelligence **275**, 104 – 137 (2019). DOI https://doi.org/10.1016/j.artint.2019.05.003. URL http://www.sciencedirect.com/science/article/pii/S0004370218304259
2. Aineto, D., Jiménez, S., Onaindia, E.: Learning strips action models with classical planning. In: International Conference on Automated Planning and Scheduling, ICAPS-18 (2018)
3. Cabot, J., Gogolla, M.: Object constraint language (ocl): a definitive guide. In: International School on Formal Methods for the Design of Computer, Communication and Software Systems, pp. 58–90. Springer (2012)
4. Chatterjee, S., Hadi, A.S.: Regression analysis by example. John Wiley & Sons (2015)
5. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing, pp. 151–158 (1971)
6. Cresswell, S.N., McCluskey, T.L., West, M.M.: Acquiring planning domain models using locm. The Knowledge Engineering Review **28**(2), 195–213 (2013). DOI 10.1017/S0269888912000422
7. Fernandez, A., Herrera, F., Cordon, O., Jose del Jesus, M., Marcelloni, F.: Evolutionary fuzzy systems for explainable artificial intelligence: Why, when, what for, and where to? IEEE Computational Intelligence Magazine **14**(1), 69–81 (2019)
8. Fox, M., Long, D.: Pddl+: Modeling continuous time dependent effects. In: Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, vol. 4, p. 34 (2002)
9. García, D., Gámez, J.C., González, A., Pérez, R.: An interpretability improvement for fuzzy rule bases obtained by the iterative rule learning approach. Int. J. Approx. Reasoning **67**(C), 37–58 (2015). DOI 10.1016/j.ijar.2015.09.001. URL https://doi.org/10.1016/j.ijar.2015.09.001
10. Ghallab, M., Howe, A., Knoblock, C., Mcdermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL—The Planning Domain Definition Language (1998). URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212
11. Gregory, P., Lindsay, A.: Domain model acquisition in domains with action costs. In: ICAPS, pp. 149–157 (2016)
12. Hansen, P., Jaumard, B.: Algorithms for the maximum satisfiability problem. Computing **44**(4), 279–303 (1990)
13. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)
14. Hoffmann, J.: The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. Journal of Artificial Intelligence Research **20**, 291–341 (2003)
15. Howey, R., Long, D.: Val's progress The automatic validation tool for pddl2.1 used in the international planning competition. In: Proceedings of the ICAPS 2003 workshop on The Competition: Impact, Organization, Evaluation, Benchmarks, pp. 28–37. Trento, Italy (2003)
16. Jilani, R., Crampton, A., Kitchin, D.E., Vallati, M.: Automated knowledge engineering tools in planning: state-of-the-art and future challenges. Knowledge Engineering for Planning and Scheduling (2014)

17. Jiménez, S., Rosa, T.D.L., Fernández, S., Fernández, F., Borrajo, D.: A review of machine learning for automated planning. The Knowledge Engineering Review **27**(4), 433–467 (2012)
18. Kotsiantis, S.B., Zaharakis, I., Pintelas, P.: Supervised machine learning: A review of classification techniques. Emerging artificial intelligence applications in computer engineering **160**, 3–24 (2007)
19. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Statistics and Computing **4**(2), 87–112 (1994). DOI 10.1007/BF00175355. URL https://doi.org/10.1007/BF00175355
20. Lanchas, J., Jiménez, S., Fernández, F., Borrajo, D.: Learning action durations from executions. In: Proceedings of the ICAPS. Citeseer (2007)
21. Long, D., Fox, M.: The 3rd international planning competition: Results and analysis. Journal of Artificial Intelligence Research **20**, 1–59 (2003)
22. Matloff, N.S.: From algorithms to Z-Scores: Probabilistic and statistical modeling in computer science. University Press of Florida Gainesville (2009)
23. McCluskey, T.L., Cresswell, S., Richardson, N.E., West, M.M.: Automated acquisition of action knowledge. International Conference on Agents and Artificial Intelligence (ICAART) (2009)
24. McCluskey, T.L., Richardson, N.E., Simpson, R.M.: An interactive method for inducing operator descriptions. In: AIPS, pp. 121–130 (2002)
25. Mitchell, T.M.: Machine Learning, 1 edn. McGraw-Hill, Inc., New York, NY, USA (1997)
26. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
27. Quinlan, J.R.: Learning logical definitions from relations. Machine learning **5**(3), 239–266 (1990)
28. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited, (2016)
29. Suárez-Hernández, A., Segovia-Aguas, J., Torras, C., Alenyà, G.: Strips action discovery (2020)
30. Var, I.: Multivariate data analysis. vectors **8**(2), 125–136 (1998)
31. Worm, T., Chiu, K.: Prioritized grammar enumeration: symbolic regression by dynamic programming. In: Proceedings of the 15th annual conference on Genetic and evolutionary computation, pp. 1021–1028. ACM (2013)
32. Yang, Q., Wu, K., Jiang, Y.: Learning action models from plan examples using weighted MAX-SAT. Artificial Intelligence Journal. p. 107–143 (2007)
33. Zhuo, H.H., Nguyen, T., Kambhampati, S.: Refining incomplete planning domain models through plan traces. In: Twenty-Third International Joint Conference on Artificial Intelligence (2013)
34. Zhuo, H.H., Yang, Q., Hu, D.H., Li, L.: Learning complex action models with quantifiers and logical implications. Artificial Intelligence **174**(18), 1540 – 1569 (2010). DOI https://doi.org/10.1016/j.artint.2010.09.007. URL http://www.sciencedirect.com/science/article/pii/S0004370210001566