Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)

Unsupervised Classification of Planning Instances

Javier Segovia
Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
javier.segovia@upf.edu

Sergio Jiménez
Computing and Information Systems
University of Melbourne
Parkville, Victoria 3010, Australia
sjimenez@unimelb.edu.au

Anders Jonsson
Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
anders.jonsson@upf.edu

Abstract

In this paper we introduce a novel approach for unsupervised classification of planning instances based on the recent formalism of planning programs. Our approach is inspired by structured prediction in machine learning, which aims at predicting structured information about a given input rather than a scalar value. In our case, each input is an unlabelled classical planning instance, and the associated structured information is the planning program that solves the instance. We describe a method that takes as input a set of planning instances and outputs a set of planning programs, classifying each instance according to the program that solves it. Our results show that automated planning can be successfully used to solve structured unsupervised classification tasks, and invites further exploration of the connection between automated planning and structured prediction.

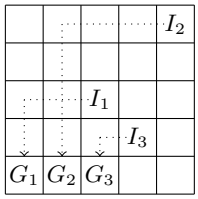
Introduction

Machine Learning (ML) tasks can be broadly classified into two categories according to the available learning source:

- *Supervised learning*. The learning source is a set of inputs labeled with desired outputs. The learning task is to compute a function that maps inputs into desired outputs.
- *Unsupervised learning*. The learning source is a set of inputs, but no labels are available. The learning task is to identify structural patterns for the given set of inputs.

Each input typically represents an object encoded as an assignment of values to a finite set of features. Each output is a scalar that can be either an integer (in the case of classification tasks) or a real value (in the case of regression tasks).

The demand of applications with complex perception abilities, such as the interpretation of natural language or image scenes, is pushing research in ML towards more expressive representations of learning tasks that relax standard assumptions. A prominent example is *structured prediction* (Bakir et al. 2007) in which the outputs represent more complex structural information about inputs, information whose size can even be exponential in the input size.



(a)

```
0. dec(x)
1. goto(0,!(x = x_G))
2. dec(y)
3. goto(2,!(y = y_G))
4. end
```

(b)

Figure 1: (a) Three different instances of grid navigation; (b) a planning program Π_1 that solves all of them.

Our work is particularly inspired by two recent approaches that synthesize *programs* in order to output structured information about inputs (Lake, Salakhutdinov, and Tenenbaum 2015; Ellis, Solar-Lezama, and Tenenbaum 2015). Programs are a natural knowledge representation for many domains, are easy to interpret by humans, and can compactly express relatively complex operations on inputs.

In this paper we ask the following question: given an unlabelled input set of planning instances, is it possible to *classify* the planning instances according to some criterion? Since planning instances include actions, it makes sense to classify them according to *behaviors* rather than feature values. Similar to existing approaches for structured prediction, our approach is to automatically synthesize programs that generate behaviors, taking advantage of the recent formalism of *planning programs* (Jiménez and Jonsson 2015; Segovia-Aguas, Jiménez, and Jonsson 2016a).

From an ML perspective, our approach can be thought of as treating the goal condition of each planning instance as the output of an unknown planning program applied to the initial state. The aim then becomes to automatically synthesize a planning program that outputs the goal (i.e. solves the instance), and cluster instances that are solved by the same program (i.e. that display the same behavior).

As an illustrating example, consider the problem of

navigating through a grid from an initial position to a goal position. Figure 1(a) represents three different planning instances of this type. These instances can all be represented using the same variables and actions: variables x and y that represent the current position, variables x_G and y_G that represent the goal position, and actions $\{\texttt{dec(x)}, \texttt{dec(y)}, \texttt{inc(x)}, \texttt{inc(y)}\}$ that decrement/increment the value of x or y. Although these three instances have different initial states and goals, they are solved by the planning program Π_1 in Figure 1(b): independent of the grid size decrement x until reaching the goal column, and decrement y until reaching the goal row. Note that we could solve these instances using three different planning programs, in which case they would no longer be considered to display the same behavior. This illustrates that, to classify the set of input instances, we should attempt to use as few programs as possible.

Our main contribution is to show that classical planning offers an alternative way to synthesize programs for structured prediction. On one hand, planning programs are more expressive than the programs synthesized in previous approaches. Ellis, Solar-Lezama, and Tenenbaum (2015), for example, only consider programs generated by an acyclic grammar, a restriction not shared by planning programs. On the other hand, the planning programs we consider here are *deterministic* rather than *probabilistic*, which limits their applicability when inputs are noisy.

The rest of the paper is organized as follows. The background section introduces existing notation and concepts that we exploit in our work. We then describe our approach for unsupervised classification of planning instances. In the following section we show how to model noise-free unsupervised classification tasks as planning problems. After that we present the results from empirical experiments, describe how our approach is related to existing approaches in the literature, and conclude with a discussion about future work.

Background

This section defines the planning models that we rely on in this work: *classical planning with conditional effects*, that we use to model the inputs of the unsupervised classification task addressed in this paper, and *generalized planning*, that we use to model sets of planning instances whose solutions share a common structure.

Classical Planning with Conditional Effects

Conditional effects make it possible to repeatedly refer to the same action even when the precise effects depend on the current state. Conditional effects are important components of generalized plans because, as shown in conformant planning (Palacios and Geffner 2009), they can adapt the execution of a sequence of actions to different initial states.

We use F to denote a set of propositional variables or *fluents* describing a state. A literal l is a valuation of a fluent $f \in F$, i.e. $l = f$ or $l = \neg f$. A set of literals L on F represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). We use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents.

Given L, let $\neg L = \{\neg l : l \in L\}$ be the complement of L. A *state* s is a set of literals such that $|s| = |F|$, i.e. a total assignment of values to fluents. The number of states is then $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in STRIPS planning.

We consider the fragment of classical planning with conditional effects that includes negative conditions and goals. Under this formalism, a *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions with conditional effects. Each action $a \in A$ has a set of literals $\mathsf{pre}(a) \in \mathcal{L}(F)$ called the *precondition* and a set of conditional effects $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two sets of literals $C \in \mathcal{L}(F)$ (the condition) and $E \in \mathcal{L}(F)$ (the effect).

An action $a \in A$ is applicable in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in s. The result of applying a in s is a new state $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

Given a planning frame $\Phi = \langle F, A \rangle$, a *classical planning instance* is a tuple $P = \langle F, A, I, G \rangle$, where $I \in \mathcal{L}(F)$ is an initial state (i.e. $|I| = |F|$) and $G \in \mathcal{L}(F)$ is a goal condition. A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied following the application of π in I.

In our classification tasks the inputs are planning instances associated with a given planning frame Φ. Given a planning frame $\Phi = \langle F, A \rangle$, we use $\Gamma(\Phi) = \{(F, A, I, G) : I \in \mathcal{L}(F), |I| = |F|, G \in \mathcal{L}(F)\}$ to denote the set of all planning instances that can be instantiated from Φ by defining an initial state I and a goal condition G.

Generalized Planning

Our definition of the generalized planning problem is based on that of Hu and De Giacomo (Hu and De Giacomo 2011), who define a generalized planning problem as a finite set of multiple individual planning instances $\mathcal{P} = \{P_1, \ldots, P_T\}$ that share the same observations and actions. Although actions are shared, each action can have different interpretations in different states due to conditional effects.

A solution Π to a generalized planning problem $\mathcal{P}$ is a generalized plan that solves every individual instance P_t, $1 \leq t \leq T$. In the literature, generalized plans have diverse forms that range from *DS-planners* (Winner and Veloso 2003) and *generalized polices* (Martín and Geffner 2004) to finite state machines (Bonet, Palacios, and Geffner 2010; Segovia-Aguas, Jiménez, and Jonsson 2016b). Each of these representations has its own syntax and semantics but they all allow non-sequential execution flow to solve planning instances with different initial states and goals.

In this work we restrict the above definition for generalized planning in two ways: 1) states are fully observable, so observations are equivalent to states; and 2) each action has the same (conditional) effects in each individual problem. As a consequence, the individual instances $P_1 = \langle F, A, I_1, G_1\rangle, \ldots, P_T = \langle F, A, I_T, G_T\rangle$ are classical planning instances that share the same planning frame $\Phi = \langle F, A\rangle$ and differ only in the initial state and goal, i.e. each individual instance P_t belongs to the set $\Gamma(\Phi)$.

Our definition of generalized planning problems is related to previous works on planning and learning that extract and reuse general knowledge from different instances of the same domain (Fern, Khardon, and Tadepalli 2011; Jiménez et al. 2012). However, we impose a stronger restriction on the individual planning instances since they do not only share the set of predicates but also the fluents, implying that they have the same state space. In addition, we assume that the solutions to generalized plans are in the form of planning programs, as defined in the next section.

Planning Programs

A planning program is a sequence of planning actions enhanced with *goto instructions*, i.e. conditional constructs for jumping to arbitrary locations of the program (Jiménez and Jonsson 2015; Segovia-Aguas, Jiménez, and Jonsson 2016a).

Given a planning frame $\Phi = \langle F, A\rangle$, a basic planning program is a sequence of instructions $\Pi = \langle w_0, \ldots, w_n\rangle$. Each instruction w_i, $0 \leq i \leq n$, is associated with a *program line* i and is drawn from the set of instructions $\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{end\}$, where $\mathcal{I}_{go} = \{\, goto(i', !f) : 0 \leq i' \leq n, f \in F\}$ is the set of goto instructions.

In other words, each instruction is either a planning action $a \in A$, a goto instruction $goto(i', !f)$ or a termination instruction end. A termination instruction acts as an explicit marker that program execution should end, similar to a return statement in programming.

The execution model for a planning program Π consists of a *program state* (s, i), i.e. a pair of a planning state $s \subseteq F$ and a program counter $0 \leq i \leq n$ whose value is the current program line. Given a program state (s, i), the execution of instruction w_i on line i is defined as follows:

- If $w_i \in A$, the new program state is $(s', i+1)$, where $s' = \theta(s, w_i)$ is the result of applying action w_i in planning state s, and the program counter is incremented.
- If $w_i = goto(i', !f)$, the new program state becomes $(s, i+1)$ if $f \in s$, and (s, i') otherwise.
- If $w_i = end$, execution terminates.

To execute a planning program Π on a planning problem $P = \langle F, A, I, G\rangle$, the initial program state is set to $(I, 0)$, i.e. the initial state of P and program line 0. The program Π *solves* P if and only if execution terminates and the goal condition holds in the resulting program state (s, i), i.e. $G \subseteq s \wedge w_i = end$.

Executing Π on P can fail for three reasons:

1. Execution terminates in program state (s, i) but the goal condition does not hold, i.e. $G \not\subseteq s \wedge w_i = end$.
2. When executing an action $w_i \in A$ in program state (s, i), the precondition of w_i does not hold, i.e. $\mathsf{pre}(w_i) \not\subseteq s$.
3. Execution enters an infinite loop that never reaches an end instruction.

This execution model is *deterministic* and hence a planning program can be viewed as a form of compact reactive plan for the family of planning problems defined by the planning frame $\Phi = \langle F, A\rangle$. Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ defined on a common planning frame $\Phi = \langle F, A\rangle$, a planning program Π is a generalized plan that solves $\mathcal{P}$ if and only if Π solves each planning instance P_t, $1 \leq t \leq T$.

Jiménez and Jonsson (2015) introduced a compilation from generalized planning to classical planning for computing planning programs. Briefly, given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ and a number of program lines n, the compilation outputs a classical planning instance $P_n = \langle F_n, A_n, I_n, G_n\rangle$. P_n is defined such that any plan π that solves P_n simultaneously programs the instructions of a planning program Π and simulates the execution of Π on each individual planning instance P_t, $1 \leq t \leq T$, thus validating that Π solves $\mathcal{P}$.

The compilation defines two new sets of fluents:

- $F_{pc} = \{\mathsf{pc}_i : 0 \leq i \leq n\}$, the fluents that encode the program counter.
- $F_{ins} = \{\mathsf{ins}_{i,w} : 0 \leq i \leq n, w \in A \cup \mathcal{I}_{go} \cup \{\mathsf{nil}, \mathsf{end}\}\}$, the fluents that encode the instructions on different program lines, where nil denotes an empty line.

Formally the components of P_n are defined as:

- $F_n = F \cup F_{pc} \cup F_{ins} \cup \{\mathsf{done}\}$,
- $A_n = \{\mathsf{P}(a_i), \mathsf{R}(a_i) : a \in A, 0 \leq i < n\}$
  $\cup \{\mathsf{P}(\mathsf{go}^f_{i,i'}), \mathsf{R}(\mathsf{go}^f_{i,i'}) : goto(i', !f) \in \mathcal{I}_{go}, 0 \leq i < n\}$
  $\cup \{\mathsf{P}(\mathsf{end}_{t,i}), \mathsf{R}(\mathsf{end}_{t,i}) : 1 \leq t \leq T, 0 < i \leq n\}$,
- $I_n = I \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\} \cup \{\mathsf{pc}_0\}$,
- $G_n = \{\mathsf{done}\}$.

Here, a_i is an action that simulates instruction $a \in A$ on line i, $\mathsf{go}^f_{i,i'}$ is an action that simulates instruction $goto(i', !f) \in \mathcal{I}_{go}$ on line i, and $\mathsf{end}_{t,i}$ is an action that simulates the termination instruction end on line i for the individual instance P_t,$1 \leq t \leq T$. The precondition of $\mathsf{end}_{t,i}$ verifies that the goal condition G_t of P_t is satisfied, and the effect is to reset the program state to $(I_{t+1}, 0)$, if $t < T$, or to add the fluent done required in the goal condition, if $t = T$. Each such action α has two versions: $P(\alpha)$, for *programming* α, and $R(\alpha)$, for *repeating* the execution of α, and the action set A_n includes both versions.

The compiled planning instance P_n can be solved using an off-the-shelf classical planner, and the resulting solution plan π can be used to extract a planning program Π that provably solves the generalized planning problem $\mathcal{P}$. A classical planner detects all three failure conditions described above, making it particularly suitable for computing planning programs.

Classification of Planning Instances

In this section we describe our approach to classification of planning instances. The input is a set of unlabelled planning instances $E = \{P_1, \ldots, P_T\}$. Just as in the definition of generalized planning, we assume that each input instance P_t, $1 \leq t \leq T$, is instantiated from a common planning frame $\Phi = \langle F, A \rangle$, i.e. $P_t \in \Gamma(\Phi)$.

Unlike generalized planning, the aim here is not to learn a *single* generalized plan for solving the instances in E. Rather, we want to *classify* the instances in E according to some criterion and, since the input instances in E are unlabelled, this is an unsupervised classification task.

The most common approach to unsupervised classification is *clustering*. Clustering addresses the classification of a set of unlabelled examples in such a way that examples within the same class (cluster) are more *similar* to each other than to those in other classes (clusters).

Our approach to unsupervised classification of planning instances is to consider two different planning instances as *similar* if their solutions share a common structure, i.e. both instances can be solved using the same generalized plan. In other words, a given planning instance belongs to a class if the generalized plan, that acts as the class prototype (Liu, Jiang, and Kot 2009), solves that instance. For instance, Figure 1(b) shows the program Π_1 that solves the three planning instances illustrated in Figure 1(a) and represents the prototype for this cluster of different planning instances.

Formally, the unsupervised classification task we consider in the paper is defined by a tuple $\langle \Phi, E, m \rangle$, where

- $\Phi = \langle F, A \rangle$ is a classical planning frame.
- $E = \{P_1, \ldots, P_T\}$ is a finite set of unlabelled examples drawn from Φ, i.e. $P_t \in \Gamma(\Phi)$, $1 \leq t \leq T$.
- m is the number of clusters, implicitly defining a set of *class labels* $C = \{c_1, \ldots, c_m\}$.

A *model* for the unsupervised classification task $\langle \Phi, E, m \rangle$ is defined as a tuple $\langle \mathcal{S}, f \rangle$, where:

- $\mathcal{S} = \{\Pi_1, \ldots, \Pi_m\}$ is set of generalized plans, one for each class label.
- $f : E \rightarrow C$ is a partial function on $\Gamma(\Phi)$ that assigns a class label $f(P_t) \in C$ to each input instance $P_t \in E$.

The model $\langle \mathcal{S}, f \rangle$ is *valid* if and only if, for each class label $c_j \in C$ and each input instance $P_t \in E$ such that $f(P_t) = c_j$, the corresponding generalized plan Π_j solves P_t. Another way to put this is that a valid model makes no classification errors with respect to the input instances in E.

Intuitively, the function f defines a *partition* of the input instances into clusters, and each cluster corresponds to a generalized planning problem $\mathcal{P}_j$, $1 \leq j \leq m$, whose planning instances are the input examples mapped to that cluster and whose solution is given by the generalized plan Π_j.

We can now define two different tasks associated with unsupervised classification of planning instances:

1. Given an unsupervised classification task $\langle \Phi, E, m \rangle$, compute a valid model $\langle \mathcal{S}, f \rangle$.
2. Given an unsupervised classification task $\langle \Phi, E, m \rangle$, a model $\langle \mathcal{S}, f \rangle$ and a planning instance $P \in \Gamma(\Phi) \setminus E$ (i.e. instantiated from Φ but not included among the inputs), classify P by determining its class label.

This second task can also be understood as plan recognition (Ramırez and Geffner 2010) using plan libraries in the form of generalized plans and where inputs are not sequences of observations but (initial state, goal) pairs. In what follows we describe our approach for carrying out each of these two tasks.

Π_0:
```
0. choose(Π1|Π2)
1. end
```

Π_2:
```
0. inc(x)
1. goto(0,!(x=n))
2. inc(y)
3. goto(2,!(y=n))
4. end
```

Figure 2: An unsupervised classification model with two clusters of planning instances represented as a planning program with a choice instruction $choose(\Pi_1|\Pi_2)$, with Π_1 given in Figure 1(b).

Computing Classification Models

This section shows how to compute a valid model $\langle \mathcal{S}, f \rangle$ for a given unsupervised classification task $\langle \Phi, E, m \rangle$.

Our approach is extending the planning program formalism and the compilation described in the previous section. In more detail we represent both $\mathcal{S}$ (the set of planning programs) and f (the mapping from input instances to class labels) as follows:

- We extend the instruction set $\mathcal{I}$ with a *choice instruction* defined as $choose(\Pi_1|\cdots|\Pi_m)$, where each Π_j, $1 \leq j \leq m$, is a planning program in $\mathcal{S}$ (possibly yet to be programmed).
- We represent *multiple* planning programs $\Pi_1, \ldots, \Pi_m$ (plus an additional program Π_0 that only contains the choice instruction, cf. Figure 2). Representing multiple planning programs is akin to using *procedures* and defining a set of new *call instructions* that allow programs to call other programs (Segovia-Aguas, Jiménez, and Jonsson 2016a). In this case however, it is not necessary to explicitly include call instructions since the choice instruction is the only instruction that chooses among the planning programs in $\mathcal{S}$.

Figure 2 shows a set of planning programs that represent an unsupervised classification model with two class labels. The first cluster corresponds to the program Π_1 shown in Figure 1(b), acting as the prototype for the cluster of planning tasks in Figure 1(a), and the second cluster corresponds to the program Π_2, acting as the prototype for the cluster of planning instances that move the agent towards the top-right corner of an $n \times n$ grid. (We remark that we can use a single planning frame to represent navigation instances with different grid sizes by including fluents x_{max} and y_{max} expressing the maximum values of x and y).

The execution model for planning programs with a choice instruction $choose(\Pi_1|\cdots|\Pi_m)$ behaves in the same way as before for all other types of instructions (including the conditions for *termination*, *success* and *failure*). The only

behaviour that has to be redefined is the execution model of the choice instruction in program state (s, i):

- If $w_i = choose(\Pi_1|\cdots|\Pi_m)$, its execution actively *chooses* a planning program among $\Pi_1, \ldots, \Pi_m$ to program and execute, setting the new program state to $(s, 0)$

Our new compilation takes as input an unsupervised classification task $\langle \Phi, E, m\rangle$ and a number of program lines n and outputs a single classical planning instance $P_{m,n} = \{F_{m,n}, A_{m,n}, I_{m,n}, G_{m,n}\}$. Since the compilation shares many similarities with the existing compilation described in the previous section, we only describe the relevant differences here:

- The fluent set $F_{m,n}$ defines $m + 1$ planning programs $\Pi_0, \Pi_1, \ldots, \Pi_m$ with n lines. This is done by introducing a *copy* f_j of each fluent $f \in F_{pc} \cup F_{ins}$ for each planning program Π_j, $0 \leq j \leq m$ (Segovia-Aguas, Jiménez, and Jonsson 2016a). As a result, a program state (s, i, j) keeps track not only of the current line i, $0 \leq i \leq n$, but also of the current planning program Π_j, $0 \leq j \leq m$.
- Likewise, for each action α that simulates an instruction, we make a copy α_j for each Π_j, $0 \leq j \leq m$, and the action set $A_{m,n}$ includes $P(\alpha_j)$ and $R(\alpha_j)$ for programming and repeating the execution of α_j.
- An end action $\mathsf{end}_{t,i,j}$, $t < T$, resets the program state to $(I_{t+1}, 0, 0)$, i.e. execution restarts for the next input instance P_{t+1} on line 0 of the "choice" program Π_0.
- $A_{m,n}$ also includes m actions choose_j, $1 \leq j \leq m$, for simulating the execution of the choice instruction $choose(\Pi_1|\cdots|\Pi_m)$, each actively choosing one of the programs among $\Pi_1, \ldots, \Pi_m$ to execute. Formally, the action choose_j is defined as

$$\mathsf{pre}(\mathsf{choose}_j) = \{\mathsf{pc}_{0,0}\},$$
$$\mathsf{cond}(\mathsf{choose}_j) = \{\neg\mathsf{pc}_{0,0}, \mathsf{pc}_{0,j}\}.$$

- The initial state $I_{m,n}$ is defined such that the program Π_0 equals that in Figure 2, except that the choice is between $\Pi_1, \ldots, \Pi_m$, while the lines of the planning programs $\Pi_1, \ldots, \Pi_m$ are initially empty. The program state is initialized to $(I_1, 0, 0)$, i.e. the initial state of the input instance P_1 and line 0 of the planning program Π_0.
- The goal condition is defined as $G_{m,n} = \{\mathsf{done}\}$ as before.

Lemma 1. *Any plan π that solves $P_{m,n}$ corresponds to a valid model $\langle \mathcal{S}, f\rangle$ for the unsupervised classification task $\langle \Phi, E, m\rangle$.*

Proof sketch. To solve $P_{m,n}$ the plan π has to simulate the choice instruction for each input instance P_t, $1 \leq t \leq T$, actively choosing a planning program among $\Pi_1, \ldots, \Pi_m$. This active choice corresponds exactly to the mapping f from input instances to class labels. In addition, for each input P_t, π has to program the instructions (if not yet programmed) of the chosen planning program Π_j, $1 \leq j \leq m$, and then simulate the execution of Π_j on P_t. For π to solve $P_{m,n}$, this simulated execution has to successfully validate that Π_j solves P_t. Hence $\mathcal{S} = \{\Pi_1, \ldots, \Pi_m\}$ satisfies the property required for the model $\langle \mathcal{S}, f\rangle$ to be valid. □

Determining Class Labels

Once we have access to a valid model $\langle \mathcal{S}, f\rangle$ for an unsupervised classification task $\langle \Phi, E, m\rangle$, a related task is to classify planning instances not in E. Consider a planning instance $P = \langle F, A, I, G\rangle \in \Gamma(\Phi) \setminus E$, i.e. instantiated from the same planning frame Φ as the input instances in E, but not part of the input.

To classify P, for each planning program $\Pi_j \in \mathcal{S}$ such that $1 \leq j \leq m$, we execute Π_j on P to verify whether Π_j solves P. If Π_j does solve P, we classify P using class label C_j.

In general, however, there is no guarantee that there exists a planning program in $\mathcal{S}$ that solves P. In this case we have a couple of choices:

- For each $\Pi_j \in \mathcal{S}$, we can record the final program state (s, i) such that $w_i = end$, and determine how *close* s is to solving P. We can estimate the distance from s to G, e.g. counting the number of goal conditions in G satisfied by s, and classify P according to the program in $\mathcal{S}$ that minimizes this distance.
- We can compute a *new* planning program that solves P, and extend $\langle \mathcal{S}, f\rangle$ with this new planning program and the associated class label. In other words, we do not consider that the new instance belongs to any of the existing classes so we define a new class, and assign it to P.

Unsupervised Classification as Planning

Supervised classification can be modeled as a generalized planning task (Lotinac et al. 2016). Likewise a noise-free decision tree can be seen as a particular contingent plan (Albore, Palacios, and Geffner 2009) whose internal nodes contain only sensing actions and its leaf nodes contain actions that transform the state assigning a class label to it. The goal of this particular contingent planning task is assigning the desired class label to each possible initial state.

Here we show that a similar approach can be defined for unsupervised classification, as an alternative to the type of unsupervised classification task that we present in previous sections. The aim of this section is not to be competitive with state-of-the-art ML techniques when solving unsupervised classification tasks but to go deep in the connection of the automated planning and ML models.

A noise-free unsupervised classification task is the task of computing a noise-free classifier that chooses a label from a set of class labels $C = \{c_1, \ldots, c_m\}$ for a given set of unlabelled examples $\{e_1, \ldots, e_T\}$. Each input e_t, $1 \leq t \leq T$ is now an assignment of values to a finite set of features $X = \{x_1, \ldots, x_l\}$, where each variable x_k, $1 \leq k \leq l$, has an associated finite domain $D(x_k)$. This unsupervised classification task can be modeled as a generalized planning task $\mathcal{P} = \{P_1, \ldots, P_T\}$ such that each individual planning tasks $P_t = \langle F, A, I_t, G_t\rangle$, $1 \leq t \leq T$, models the classification of the t^{th} example:

- F comprises the fluents required for representing the learning examples, their labels and an extra fluent (classified) that indicates whether or not a given example has already received a class label.

- A contains the actions necessary to label a given example with a class label in C. For instance, in a two-class unsupervised classification, $C = \{\mathsf{class}_1, \mathsf{class}_2\}$ and $A = \{\mathsf{setClass}_1, \mathsf{setClass}_2\}$. While the $\mathsf{setClass}_1$ action makes the fluent class_1 true, action $\mathsf{setClass}_2$ makes the fluent class_2 true, and both actions make the fluent classified true to prevent the execution of these actions more than once for the same example. Formally, action $\mathsf{setClass}_\mathsf{c}$ is defined as follows:

$$\mathsf{pre}(\mathsf{setClass}_\mathsf{c}) = \{\neg\mathsf{classified}\},$$
$$\mathsf{cond}(\mathsf{setClass}_\mathsf{c}) = \{\emptyset \rhd \{\mathsf{classified}, \mathsf{class}_\mathsf{c}\}\}.$$

- I_t contains the fluents that describe the t^{th} example while $G_t = \{\mathsf{classified}\}$ requires that the example has been labeled.

According to this formulation, the solution Π to a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ that models an unsupervised learning task of this type is a noise-free classifier for the T unlabelled learning examples if and only if Π assigns every class $c_j \in C$, $1 \leq j \leq m$, to at least one example e_t, $1 \leq t \leq T$. This generalized planning task can be compiled into a classical planning instance (Jiménez and Jonsson 2015; Segovia-Aguas, Jiménez, and Jonsson 2016a) adding a new goal condition to the resulting planning instance that forces Π to assign every class label to at least one input example. For instance, in a two-class unsupervised learning task, fluents class_1 and class_2 are also goals of the planning instance resulting from the compilation.

This modelling of traditional ML tasks as classical planning instances is straightforward when examples are described using *logic*, like in the classic *Michalski's train* classification task (Michalski, Carbonell, and Mitchell 2013). However, we can also model classification tasks for which the features used to describe learning examples have finite domains. In this case we assume that the fluents F are instantiated from a set of predicates Ψ and a set of objects Ω, and that there exists a predicate $\mathsf{assign}(v, x) \in \Psi$ and that Ω is partitioned into two sets Ω_v (the *variable objects*) and Ω_x (the *value objects*). Intuitively, a fluent $\mathsf{assign}(v, x)$, $v \in \Omega_v$ and $x \in \Omega_x$, is true if and only if x is the value currently assigned to the variable v. A given variable represents exactly one value at a time, so for a given v, fluents $\mathsf{assign}(v, x)$, $x \in \Omega_x$, are *invariants*.

We show here a simple unsupervised classification task that can be modeled in this way. The task is a two-cluster classification of examples that correspond to the different value combinations of three Boolean variables, x_1, x_2 and y. More precisely, there is a total of eight examples, four corresponding to the variable values of the formula $y = x_1 \rightarrow x_2$ (the first cluster) and the other four corresponding to the formula $y = x_2 \rightarrow x_1$ (the second cluster). Figure 3 shows a program Π that solves this unsupervised classification task.

Even though there are different ways of clustering the examples, a classical planner would prefer compact cluster models, like the one shown in Figure 3, i.e. that requires a small number of program lines. The fluent $y = \neg x_1 \wedge x_2$ that appears in the goto instruction (line 0) represents a *derived fluent* that holds in all the states for which the encoded formula is satisfied.

```
Π: 0. goto(3,!(y = ¬x1 ∧ x2))
   1. setClass1
   2. end
   3. setClass2
   4. end
```

Figure 3: Example planning program for the two-cluster classification task of learning examples that correspond to the logic formulae $y = x_1 \rightarrow x_2$ and $y = x_2 \rightarrow x_1$.

To synthesize the planning program in Figure 3 starting from scratch we need to automatically discover the *condition* $y = \neg x_1 \wedge x_2$ that determines how the input examples should be classified. In this case the unsupervised learning task is to identify the derived fluents that allow a compact classification model. This task can be addressed by extending the compilation for solving generalized planning tasks with a unification mechanism (Lotinac et al. 2016). Note that this is a more traditional ML task than the structured prediction addressed in previous sections, which classifies examples according to *behavior* rather than their features.

Evaluation

We evaluated our approach empirically on an Intel Core i5 3.10 GHz x 4 processor with a memory bound of 4GB and a time constraint of 600 seconds. We use the classical planner Fast Downward (Helmert 2006) in the LAMA-2011 setting (Richter and Westphal 2010) to solve the compiled classical planning instances. We performed two kinds of experiments[1]:

1. Compute a valid classification model $\langle \mathcal{S}, f \rangle$ given an unsupervised classification task $\langle \Phi, E, m \rangle$.
2. Classify a given planning instance P given an existing model $\langle \mathcal{S}, f \rangle$ by determining the planning program in $\mathcal{S}$ that solves P.

Benchmarks

The instances used in experiments come from three generic domains (i.e. planning frames). In these generic domains, all planning instances share the planning frame $\Phi = \langle F, A \rangle$. This means that we can first create clusters for the planning instances in E, compute the planning program that provides the prototype behavior of each cluster, and then test multiple planning instances for each task to see how they are classified.

In the first domain, **Grid**, the goal is to navigate to a goal position by incrementing or decrementing the x or y values. The domain also includes fluents that represent the goal position (cf. the example in Figure 1). The second domain, **List**, models lists of integers, in which actions iterate over list elements or apply an operation to the current list element. In the third domain, **Boolean Circuits**, actions consist of applying Boolean functions such as *and*, *not*, and *or* with

[1]The source code and benchmarks are available at https://github.com/aig-upf/automated-programming-framework

		Insts	Lines	Clusters	Facts	Oper	Search	Preprocess	Total
Grid	H-V	4	2	4	284	292	0.04	1.23	1.27
	Quadrant	4	4	2	356	634	5.77	1.04	6.81
List	Visit	4	4	2	362	762	0.20	0.51	0.71
Bool	Assign	8	2	2	220	194	0.17	0.38	0.55
	Nor-Nand	8	3	2	326	330	0.79	0.56	1.35

Table 1: Number of learning instances; bounds on the number of lines per cluster and clusters; number of facts and operators in the compiled classical planning task; search, preprocessing and total time (in seconds) elapsed while computing the solution.

conditional effects, using Boolean variables $\{x_1, x_2, y\}$ to create the circuits.

For the **Grid** domain we created two different classification tasks: **H-V** that comprise horizontal and vertical navigation tasks and **Quadrant** where navigation is done towards the top-right or bottom-left quadrant (cf. Figure 2).

In the **List** domain we tested a **Visit** task whose classes are to perform operations (i.e. visit) on all the list elements or only on every second element (odd positions in the list). For the **Boolean Circuits** domain we implemented two tasks: **Assign**, in which the aim is to perform either of these two operations $x_1 \leftarrow x_2$ and $x_2 \leftarrow x_1$; and **NOR-NAND**, in which the aim is to correctly create and classify *nor* and *nand* circuits.

Computing Classification Models

Table 1 summarizes the results of the first kind of experiments. In this table we provide the number of input instances, the bounded number of lines for each planning program, and the number of clusters (each corresponding to a planning program). Also we report some data of the planning compilation like the number of facts and operators the planner has to handle, and the times required for preprocessing and search.

The solutions obtained in **H-V** were four clusters decreasing or increasing the corresponding variable (x or y) in order to reach the target position along a horizontal or vertical line.

$$\Pi_1 : 0.inc(x), 1.goto(0, !(x = x_G)),$$
$$\Pi_2 : 0.dec(y), 1.goto(0, !(y = y_G)),$$
$$\Pi_3 : 0.dec(x), 1.goto(0, !(x = x_G)),$$
$$\Pi_4 : 0.inc(y), 1.goto(0, !(y = y_G)).$$

Regarding the **Quadrant** task, the two planning programs have to navigate to a specific target position in one of the two quadrants (top-right or bottom-left), so the planning program Π_1 for **Quadrant** is the combination of Π_1 and Π_4 from the **H-V** task, and program Π_2 for **Quadrant** is the combination of Π_2 and Π_3 from the **H-V** task.

In the task for the **List** domain, the first program Π_1 visits all elements of the list:

$$\Pi_1 : 0.visit(n), 1.next(n), 2.goto(0, !(n = nil)),$$

while the second program Π_2 only visits every second element by applying the `next` action twice in each iteration.

In the **Boolean Circuits** domain, the input is always the whole set of Boolean variable assignments corresponding to a given Boolean circuit. In this case, instead of obtaining the expected Boolean functions, the planner finds sequences of actions that set the outcome to true or false, thus classifying input instances depending on the value of the variable in the goal condition (true or false). The prototype planning programs for clusters in **Assign** are

$$\Pi_1 : 0.not(x_1), 1.and(x_1, x_2),$$
$$\Pi_2 : 0.not(x_2), 1.or(x_2, x_1),$$

while in **NOR-NAND** they are

$$\Pi_1 : 0.not(y), 1.or(x_1, y),$$
$$\Pi_2 : 0.or(x_1, x_2), 1.not(x_1).$$

The *not(var)* function directly modifies the var value while *or(var_1,var_2)* and *and(var_1,var_2)* assign the result to the left variable var_1. The Boolean functions computed for the **Assign** task appear in Table 2, and those for **NOR-NAND** in Table 3. Their input instances are the four possible combinations of two boolean variables x_1 and x_2 for two program classifiers (eight instances in total), and the goals are to assign specific values to x_1, x_2 and/or y. Those goals and planner classifications are in bold in the tables (e.g. **Assign** input $x_1 = 0$ and $x_2 = 1$ for task $x_2 \leftarrow x_1$, is classified into Π_2 such that x_2 becomes 0). In order to avoid planners to just assign a value to a variable, like **NOR-NAND** domain that requires a more complex strategy, the resulting values have to be assigned to x_1 and y.

Assign					
Input		Π_1		Π_2	
x_1	x_2	x_1	x_2	x_1	x_2
0	0	**0**	**0**	0	1
0	1	**1**	1	0	**0**
1	0	**0**	0	1	**1**
1	1	0	1	**1**	**1**

Table 2: Boolean results for **Assign** task

NOR-NAND								
Input			Π_1			Π_2		
x_1	x_2	y	x_1	x_2	y	x_1	x_2	y
0	0	0	**1**	0	**1**	1	0	0
0	1	0	**1**	1	**1**	**0**	1	**0**
1	0	0	**1**	0	**1**	**0**	0	**0**
1	1	0	1	1	1	**0**	1	**0**

Table 3: Boolean results for **NOR-NAND** task

Domain	Task	Tests	Lines	Clusters	Facts	Oper	Search	Preprocess	Total
Grid	H-V	8/8	2	4	180	52	0.06	10.31	10.37
	Quadrant	6/6	4	2	102	30	0.02	0.98	1.00
List	Visit	8/8	4	2	122	30	0.04	1.60	1.64
Pointers	FRS	7/7	4	3	175	46	0.43	23.42	23.85

Table 4: Number of tests (Correctly Classified/Total); bounds on lines per cluster and clusters; number of facts and operators; search, preprocessing and total time (in seconds) elapsed while computing the solution.

Determining Class Labels

For the second kind of experiments we create additional instances for the tasks **H-V**, **Quadrant** and **Visit**. We were unable to test the **Boolean Circuits** domain because the previous inputs already included all possible variable assignments corresponding to a given Boolean circuit. In addition, we test a new generic domain called **Pointers** in which the three possible clusters have to perform find, select and reverse tasks on lists. The *find* task counts the number of occurrences of a given element in a list, the *select* task searches for the minimum element in the list, and the *reverse* task reverses the order of the elements in the list.

Table 4 displays the same features as in Table 1, but the planning programs are already programmed for each cluster, so the instructions are included as fluents in the initial state of the compiled classical planning instance and the set of instances are included as tests in the domain (correctly classified / total number of instances). The idea is to check the outcome of noise-free classification for more complex problems, instead of dedicating resources to the search of the planning programs themselves, a costly operation due to the exponential complexity in the bound on the number of lines.

All the tests in the second table have been classified correctly using the provided knowledge in the form of existing planning programs. Nevertheless, we can find some extreme cases where planning programs with different structures can come up to the same result given an instance. In these situations, the classical planner can classify the instance in an arbitrary way, using by default the plan length, since its behavior is consistent with any of the clusters.

Related work

Previous work on learning *generalized policies* already relaxed traditional ML assumptions in the representation of learning examples and in the output of the learning process. In particular these works computed, from a set of planning tasks, a function mapping state and goals, into the preferred action to execute in the state to achieve the goals. The followed approach was extending the classic algorithm for learning decision lists to the planning setting (Khardon 1999; Martín and Geffner 2004).

Inductive Logic Programming (ILP) (Muggleton 1999) also deals with ML tasks where the examples and the learned models are described in logic. A relevant example is the ILP algorithm that extends to the logic setting the classic ML algorithm for learning decision trees (Blockeel and De Raedt 1998) and that has was also used to learn generalized policies (De la Rosa et al. 2011).

As already mentioned, our work is also related to existing work that synthesizes programs in order to output structured information about inputs (Lake, Salakhutdinov, and Tenenbaum 2015; Ellis, Solar-Lezama, and Tenenbaum 2015).

Conclusions

In this paper we formalize unsupervised classification of planning instances. We follow a prototype-based clustering approach where generalized plans act as prototype clusters that capture patterns in the solutions of different planning instances. To do so we extend the planning program formalism for representing generalized plans and introduce a compilation that allows us to perform unsupervised classification of planning instances using an off-the-shelf classical planner.

Our compilation assumes that the number of clusters m is a priori known. If lower values of m are used the classical planner will not be able to find a solution. Larger values of m do not formally affect to our compilation, but in practice classical planners are sensitive to the size of their inputs so they affect performance and may result in overfitting. By overfitting we mean that a larger m finds more structures with the same set of instances, thus it will be easier to fail in the classification of a new instance.

Generalized planning algorithms can also be understood as unsupervised learning methods where the examples to classify are reachable states and class labels are the actions to execute in any state. In this regard a generalized plan can be seen as the learned cluster model because it assigns an action to every reachable state. If the generalized plan is induced from classical plans we can consider them as *labels* and in this case the generalized planning algorithm would correspond to supervised learning (De la Rosa et al. 2011).

In its current state our work cannot compete with state-of-the-art ML techniques on standard unsupervised data-sets (e.g. 2D clustering). Our models are noise-free by definition, and error over learning examples is not allowed. Still, it does not mean we cannot misclassify an instance due to an ambiguous structure. We believe however that connecting the models of automated planning and ML is valuable since it brings new benchmarks to planning and pushes the scope of ML techniques.

An interesting direction for future work is to further restrict planning programs according to some criterion, which would help make the approach scale better. Previous work on synthesizing programs rely on program sketching (Solar-Lezama 2008) or generate programs from acyclic grammars (Ellis, Solar-Lezama, and Tenenbaum 2015). In both cases, this severely restricts the search space for synthesizing programs. Quite likely, a similar approach could be used

in the compilation to classical planning to restrict the search for planning programs (Baier, Fritz, and McIlraith 2007).

Acknowledgments This work is partially supported by grant TIN2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain.

References

Albore, A.; Palacios, H.; and Geffner, H. 2009. A translation-based approach to contingent planning. In *International Joint Conference on Artificial Intelligence*.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, 26–33.

Bakir, G.; Taskar, B.; Hofmann, T.; Schölkopf, B.; Smola, A.; and Vishwanathan, S. 2007. *Predicting Structured Data*. MIT Press.

Blockeel, H., and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artif. Intell.* 101(1-2):285–297.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*.

De la Rosa, T.; Jiménez, S.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research* 40:767–813.

Ellis, K.; Solar-Lezama, A.; and Tenenbaum, J. 2015. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28*.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Mach. Learn.* 84(1-2):81–107.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *International Joint Conference on Artificial Intelligence*, 918–923.

Jiménez, S., and Jonsson, A. 2015. Computing Plans with Control Flow and Procedures Using a Classical Planner. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, 62–69.

Jiménez, S.; De La Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review* 27(04):433–467.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113(1):125–148.

Lake, B.; Salakhutdinov, R.; and Tenenbaum, J. 2015. Human-level concept learning through probabilistic program induction. *Science* 350:1332–1338.

Liu, M.; Jiang, X.; and Kot, A. C. 2009. A multi-prototype clustering algorithm. *Pattern Recognition* 42(5):689–698.

Lotinac, D.; Segovia, J.; Jiménez, S.; and Jonsson, A. 2016. Automatic generation of high-level state features for generalized planning. In *International Joint Conference on Artificial Intelligence*.

Martín, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Appl. Intell* 20:9–19.

Michalski, R. S.; Carbonell, J. G.; and Mitchell, T. M. 2013. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

Muggleton, S. 1999. Inductive logic programming: issues, results and the challenge of learning language in logic. *Artificial Intelligence* 114(1):283–296.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research* 35:623–675.

Ramırez, M., and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010)*, 1121–1126.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016a. Generalized planning with procedural domain control knowledge. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016b. Hierarchical finite state controllers for generalized planning. In *International Joint Conference on Artificial Intelligence*.

Solar-Lezama, A. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation.

Winner, E., and Veloso, M. 2003. Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*, 800–807.