Automatic Generation of High-Level State Features for Generalized Planning

Damir Lotinac and **Javier Segovia-Aguas** and **Sergio Jiménez** and **Anders Jonsson**
Dept. Information and Communication Technologies, Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
{damir.lotinac,javier.segovia,sergio.jimenez,anders.jonsson}@upf.edu

Abstract

In many domains generalized plans can only be computed if certain high-level state features, i.e. features that capture key concepts to accurately distinguish between states and make good decisions, are available. In most applications of generalized planning such features are hand-coded by an expert. This paper presents a novel method to automatically generate high-level state features for solving a generalized planning problem. Our method extends a compilation of generalized planning into classical planning and integrates the computation of generalized plans with the computation of features, in the form of conjunctive queries. Experiments show that we generate features for diverse generalized planning problems and hence, compute generalized plans without providing a prior high-level representation of the states. We also bring a new landscape of challenging benchmarks to classical planning since our compilation naturally models classification tasks as classical planning problems.

1 Introduction

A generalized plan is a single solution valid for a set of planning problems. Generalized plans are usually built with branching and repetition constructs which allow them to solve arbitrarily large problems, and problems with partial observability and non-deterministic actions [Bonet *et al.*, 2010; Hu and Levesque, 2011; Srivastava *et al.*, 2011; Hu and De Giacomo, 2013]. For many problems, generalized plans can only be efficiently computed if branching (and/or repetition) is done according to key features that allow high-level reasoning and help to accurately distinguish between states.

To illustrate this, consider the problem of finding the minimum element in the following list of five integers: $(2, 5, 3, 1, 4)$. A classical plan for this problem is the 4-action sequence $\langle inc(i), inc(i), inc(i), j = i \rangle$ where i and j are list iterators that initially point to the first position in the list, $inc(i)$ increments iterator i, and $j = i$ assigns iterator i to j. The goal is for iterator j to point to the minimum element in the list. The plan fails to generalize since it is no longer valid if the order of the integers in the list is changed or if integers are added to (or deleted from) the list.

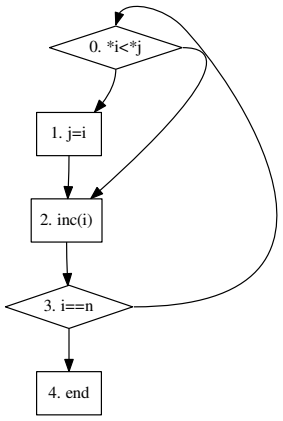

Figure 1: Planning program for finding the minimum element in a list of integers of size n.

A generalized plan can solve this problem for different list orders and sizes. In this work we focus on generalized plans in the form of *planning programs* [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016]. Figure 1 shows the planning program for finding the minimum element in a list of integers of any size. Instructions on lines 0 and 3, represented with diamonds, are conditional goto instructions that, respectively, jump to line 2 when $*i \not< *j$ and to line 0 when $i \neq n$. The outgoing left branch of a diamond indicates that the condition holds and the right branch that it does not. Instructions on lines 1 and 2 are sequential instructions and are represented with boxes. Finally, `end` marks the program termination.

In this program, conditions $*i < *j$ and $i == n$ are high-level state features necessary to compactly represent a solution that generalizes. These features abstract different list contents and sizes as well as different values for iterators i and j. Specifically, condition $*i < *j$ abstracts the set of states where the element pointed to by i is smaller than that pointed to by j. Likewise, $i == n$ abstracts the set of states

where iterator i reaches the end of the list, no matter the list size.

In generalized planning problems, high-level state features are traditionally hand-coded which requires significant human expertise. The contribution of this work is to automatically generate the high-level features required to solve a given generalized planning problem. We do so updating the notion of *planning programs* and extending a previous compilation of generalized planning into classical planning [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016] to tightly integrate the computation of generalized plans with the computation of features in the form of conjunctive queries. The second contribution of this work is bringing a new landscape of challenging benchmarks to classical planning given that our extended compilation naturally models classification tasks as classical planning problems. This allows us to solve problems that integrate planning and classification using an off-the-shelf classical planner.

2 Background

The planning model we consider is *classical planning with conditional effects*. This formalism allows a generalized plan to repeatedly refer to the same action, while the actual action effects depend on the state in which the action is applied. In this section we also define *planning programs*, the formalism we use to represent and compute generalized plans.

2.1 Classical Planning With Conditional Effects

We represent states in terms of propositional variables or *fluents* and assume that fluents are instantiated from predicates. Specifically, there exists a set of predicates Ψ, and each predicate $p \in \Psi$ has an argument list of arity $ar(p)$. Given a set of objects Ω, the set of fluents F is induced by assigning objects in Ω to the arguments of predicates in Ψ, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ where, given a set X, X^n is the n-th Cartesian power of X.

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 represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). 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.

A *classical planning problem with conditional effects* is a tuple $P = \langle \Psi, \Omega, A, I, G \rangle$, where Ψ is a set of predicates and Ω is a set of objects (inducing a set of fluents F), A is a set of actions, I is an initial state and G is a goal condition, i.e. a set of literals. Each action $a \in A$ has a set of literals $\mathsf{pre}(a)$ 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 sets of literals C (the condition) and E (the effect).

Action 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 $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

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.

2.2 Planning Programs

Our aim is to generate high-level state features to compute generalized plans. Our definition of generalized planning is loosely based on that of Hu and De Giacomo (2011). We define a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ as a set of multiple individual planning problems that share fluents and actions. Consequently, each individual planning problem $P_t \in \mathcal{P}$ is defined as $P_t = \langle \Psi, \Omega, A, I_t, G_t \rangle$, where Ψ, Ω and A are shared and only the initial state I_t and goal condition G_t differ from other planning problems in $\mathcal{P}$.

To represent and compute generalized plans we exploit a recent formalism called *planning programs* [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016]. Given a planning problem $P = \langle \Psi, \Omega, A, I, G \rangle$, a planning program Π is a numbered list of *instructions* with each instruction belonging to one of the following three types:

1. *Sequential instruction*, i.e. an action in A.
2. *Conditional goto instruction*, $goto(i', !f)$ where, i' is the target program line and $f \in F$ is the jump condition.
3. *Termination instruction* marking the program end.

To execute a planning program Π on P, we maintain a current state s, initialized to I, and a program counter pc, initialized to 0. Let w be the instruction on the line indicated by pc. If $w \in A$, we update s as $s = \theta(s, w)$ and increment pc. If $w = goto(i', !f)$, we set pc to i' if f does not hold in s, and increment pc otherwise (as in the original work, we jump whenever the condition f is false). Finally, if w is a termination instruction, execution ends successfully.

Since conditional goto instructions may cause infinite loops, execution fails whenever it reaches a pair of state and program counter (s, pc) already visited. A planning program Π solves P if the execution ends successfully and the goal condition holds in the resulting state, i.e. $G \subseteq s$. A planning program Π solves a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ if it solves every problem $P_t \in \mathcal{P}$.

Jiménez and Jonsson (2015) described a compilation that takes a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ as input and produces a single classical planning problem P_n, where n bounds the maximum number of program lines. Briefly, P_n extends the set of fluents of instances in $\mathcal{P}$ with new fluents for encoding the content of the n program lines and the current value of the program counter. In addition, P_n replaces the actions of instances in $\mathcal{P}$ with new actions for programming and executing the program instructions on the different program lines. A solution plan to P_n corresponds to a planning program Π that solves every problem in $\mathcal{P}$.

3 Generating High-Level State Features

This section defines high-level state features as conjunctive queries and extends planning programs [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016] such that *conditional*

$$\mathsf{equal}(a,b) \leftarrow \exists x_1.\mathsf{assign}(a,x_1), \mathsf{assign}(b,x_1).$$

$$\mathsf{lessthan\text{-}pointers}(a,b) \leftarrow \exists x_1,x_2.\mathsf{points\text{-}to}(a,x_1), \mathsf{points\text{-}to}(b,x_2), \mathsf{lessthan}(x_1,x_2).$$

Figure 2: Derived predicates in the form of conjunctive queries for finding the minimum number in a list.

goto instructions jump according to the value of a conjunctive query. The section also extends the compilation from generalized planning into classical planning to generate planning programs with conjunctive queries using a classical planner.

3.1 High-Level State Features

The notion of a high-level state feature is very general and has been used in different areas of AI and for different purposes. If we restrict ourselves to planning, a high-level state feature can broadly be viewed as a state abstraction to compactly represent planning problems and/or solutions to planning problems. Diverse formalisms have been used to represent high-level state features in planning ranging from first order clauses [Veloso *et al.*, 1995] to description logic formulae [Martín and Geffner, 2004], LTL formulae [Cresswell and Coddington, 2004], PDDL derived predicates [Hoffmann and Edelkamp, 2005] and, more recently, observation formulae [Bonet *et al.*, 2010].

In our formalism, states are represented in terms of fluents that are instantiated from the set of predicates Ψ. We consider high-level state features that are arbitrary formulae over the predicates in Ψ. A high-level state feature is also known as a *derived predicate* if it produces a new predicate whose truth value is determined by the corresponding formula. Derived predicates have proven useful for concisely representing planning problems with complex conditions and effects [Thiébaux *et al.*, 2005] and for more efficiently solving optimal planning problems [Ivankovic and Haslum, 2015].

In this paper we restrict ourselves to formulae in the form of *conjunctive queries* from database theory [Chandra and Merlin, 1977]. Conjunctive queries are a fragment of first-order logic in which formulae are constructed from atoms using conjunction and existential quantification (disallowing all other logical symbols). A conjunctive query can be written as

$$\varphi = (x_1,\ldots,x_k).\exists x_{k+1},\ldots,x_m.\phi_1 \wedge \cdots \wedge \phi_q,$$

where $x_1,\ldots,x_k$ are *free variables*, $x_{k+1},\ldots,x_m$ are *bound variables*, and $\phi_1,\ldots,\phi_q$ are *atoms*.

Figure 2 shows two derived predicates, in the form of conjunctive queries, that correspond to features $a == b$ and $*a < *b$. In both predicates, a and b act as free variables while x_1 and x_2 are bound variables. The first derived predicate models whether two given iterators point to the same memory address, while the second models whether the value pointed to by an iterator is less than the value pointed to by another iterator. The example assumes that predicate assign models the assignment of a memory address to a pointer variable, points-to models the content of the associated memory location, and lessthan models that a value is less than another.

Given a planning problem $P = \langle \Psi, \Omega, A, I, G \rangle$, we define a derived predicate on a set of variables $X = \{x_1,\ldots,x_m\}$, each with domain Ω. Each atom $p(v)$ in the derived predicate consists of a predicate $p \in \Psi$ and a tuple $v \in X^{ar(p)}$ that assigns variables in X to arguments of p. In addition, we make the following assumptions regarding predicates and objects:

1. We partition Ω into a set of *variable objects* Ω_v (not to be confused with the variables of a conjunctive query) and a set of remaining objects Ω_o.
2. For each predicate $p \in \Psi$ we designate at most one argument as a *variable argument* to be filled by a variable object (WLOG the variable argument always goes first). In the example, the first argument of predicates assign and points-to is a variable argument. For predicates without variable arguments, such as lessthan, we simply remove the variable argument and associated variable object.
3. In a conjunctive query, free variables can only be assigned to variable arguments of predicates and have domain Ω_v, while bound variables can only be assigned to non-variable arguments and have domain Ω_o.

Note that we could get rid of variable objects and variable arguments by redefining predicates (e.g. assign could become assign-i, assign-j and assign-n). As a consequence, all variables of a conjunctive query would be bound. However, variable objects offer flexibility, by allowing the variable objects to vary, and by generating derived predicates that are valid for a range of variable objects. For instance, the conditions $i == n$ and $*i < *j$ in the program of Figure 1 are generated by assigning objects to a and b accordingly (this requires i, j and n to be *variable objects* of the planning problem).

3.2 Planning Programs with Conjunctive Queries

To incorporate conjunctive queries into planning programs we simply replace the fluent f of conditional goto instructions $goto(i', !f)$ with a conjunctive query φ. The execution of a planning program with conjunctive queries proceeds as explained in Section 2, except when the instruction associated with the current program counter is a conditional goto instruction $goto(i', !\varphi)$. In that case, pc is set to i' if φ does not unify with the current state s, else pc is incremented.

Let $\varphi = (x_1,\ldots,x_k).\exists x_{k+1},\ldots,x_m.\phi_1 \wedge \cdots \wedge \phi_q$ be a conjunctive query, and let $u \in \Omega_v^k$ be an assignment of variable objects to the free variables $x_1,\ldots,x_k$. We describe a strategy for unifying φ with the current state s. The idea is to maintain a subset $\Phi \subseteq \Omega_o^{m-k}$ of possible joint assignments of objects to the bound variables $x_{k+1},\ldots,x_m$. We then unify the atoms of φ with s one at a time, starting with ϕ_1, and update the set Φ as we go along. After processing all atoms, φ unifies with s if and only if Φ is non-empty, i.e. if there remains at least one possible joint assignment to the bound variables.

To illustrate this idea, consider again the example problem introduced in Section 1 for finding the minimum element in

the integer list $(2, 5, 3, 1, 4)$. Consider the derived predicate $\mathsf{lessthan\text{-}pointers}(a, b)$ from Figure 2, and let (i, j) be the assignment of variable objects to the free variables a, b. Assume that in the current state s, iterator i points to the third position of the list while iterator j points to the first position. Unification proceeds one atom at a time, and initially $\Phi = \Omega_o^2$, i.e. all joint assignments to the bound variables x_1, x_2 are possible.

The first atom $\mathsf{points\text{-}to}(i, x_1)$ unifies with $x_1 = 3$, the element in the third position of the list. Consequently, joint assignments in Φ that do not assign the value 3 to x_1 are no longer possible and thus removed. Likewise, the second atom $\mathsf{points\text{-}to}(j, x_2)$ unifies with $x_2 = 2$, and joint assignments that do not assign the value 2 to x_2 are removed from Φ. As a result, Φ contains a single possible joint assignment $(3, 2)$ to x_1, x_2. Since 3 is not less than 2, no joint assignment in Φ unifies with the third atom $\mathsf{lessthan}(x_1, x_2)$. As a result, Φ becomes empty, and φ is considered *non-unifiable* with s.

3.3 Computing Planning Programs with Conjunctive Queries

In this section we extend the compilation of Jiménez and Jonsson (2015) to compute planning programs with conjunctive queries. The extended compilation takes as input a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ and constants n, q and m (that bound the number of program lines, atoms and bound variables, respectively) and outputs a single planning problem $P^m_{n,q}$. A solution to $P^m_{n,q}$ corresponds to a planning program Π with conjunctive queries such that Π solves $\mathcal{P}$.

Since programming and executing sequential and termination instructions is identical to the original compilation, we only describe here the part of $P^m_{n,q}$ that corresponds to programming and evaluating conjunctive queries. We define a set of bound variables $X = \{x_1, \ldots, x_m\}$ and a set of *slots* $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$. Each slot is a placeholder for an atom of a conjunctive query, and we also define a dummy slot σ_0. The compilation is extended with the following novel fluents:

- For each pair of program lines i, i' such that $i' \neq i + 1$, a fluent $\mathsf{ins}_{i,\mathsf{goto}(i')}$ indicating that the instruction on line i is a goto instruction $goto(i', !\varphi)$.
- For each slot $\sigma_k \in \Sigma \cup \{\sigma_0\}$, a fluent slot^k indicating that σ_k is the current slot.
- For each line i and slot $\sigma_k \in \Sigma$, a fluent eslot^k_i indicating that slot σ_k on line i is empty.
- For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \Psi$, variable object $v \in \Omega_v$ and variable tuple $(y_2, \ldots, y_{ar(p)}) \in X^{ar(p)-1}$, a fluent $\mathsf{atom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})$ indicating that $p(v, y_2, \ldots, y_{ar(p)})$ is the atom in slot σ_k of line i.
- For each slot $\sigma_k \in \Sigma$ and object tuple $(o_1, \ldots, o_m) \in \Omega_o^m$, a fluent $\mathsf{poss}^k(o_1, \ldots, o_m)$ indicating that at σ_k, $(o_1, \ldots, o_m)$ is a possible joint assignment of objects to the bound variables $x_1, \ldots, x_m$.
- A fluent eval indicating that we are done evaluating a conjunctive query and a fluent acc representing the outcome of the evaluation (true or false).

In the initial state, all fluents above appear as false except slot^0, indicating that we are ready to program and unify the atoms of any conjunctive query. The initial state on other fluents is identical to the original compilation, as is the goal condition. We next describe the set of actions that have to be added to the original compilation to implement the mechanism for programming and evaluating conjunctive queries.

A conjunctive query φ is activated by programming a goto instruction $goto(i', !\varphi)$ on a given line i. As a result of programming the goto instruction, all slots on line i are marked as empty. For each pair of program lines i, i', the action $\mathsf{pgoto}_{i,i'}$ for programming $goto(i', !\varphi)$ on line i is defined as

$$\begin{aligned}\mathsf{pre}(\mathsf{pgoto}_{i,i'}) &= \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{nil}}\},\\ \mathsf{cond}(\mathsf{pgoto}_{i,i'}) &= \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,\mathsf{goto}(i')}\},\\ &\qquad \emptyset \rhd \{\mathsf{eslot}^1_i, \ldots, \mathsf{eslot}^q_i\}\}.\end{aligned}$$

The precondition contains two fluents from the original compilation: pc_i, modeling that the program counter equals i, and $\mathsf{ins}_{i,\mathsf{nil}}$, modeling that the instruction on line i is empty.

Once activated, we have to program the individual atoms in the slots of the conjunctive query φ. After programming an atom, the slot is no longer empty. For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \Psi$, variable object $v \in \Omega_v$ and tuple of bound variables $(y_2, \ldots, y_{ar(p)}) \in X^{ar(p)-1}$, the action $\mathsf{patom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})$ is defined as

$$\begin{aligned}&\mathsf{pre}(\mathsf{patom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})) = \{\mathsf{pc}_i, \mathsf{slot}^{k-1}, \mathsf{eslot}^k_i\},\\ &\mathsf{cond}(\mathsf{patom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})) =\\ &\qquad \{\emptyset \rhd \{\neg\mathsf{eslot}^k_i, \mathsf{atom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})\}\}.\end{aligned}$$

Note that the action $\mathsf{patom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})$ assigns a concrete variable object v to its only free variable, i.e. the conjunctive queries that we generate already assign variable objects to free variables.

The key ingredient of the compilation are step actions that iterate over the atoms in each slot while propagating the remaining possible values of the bound variables. For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \Psi$, variable object $v \in \Omega_v$ and tuple of bound variables $(y_2, \ldots, y_{ar(p)}) \in X^{ar(p)-1}$, step action $\mathsf{step\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})$ is defined as

$$\begin{aligned}&\mathsf{pre}(\mathsf{step\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})) =\\ &\quad \{\mathsf{pc}_i, \mathsf{slot}^{k-1}, \mathsf{atom\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})\},\\ &\mathsf{cond}(\mathsf{step\text{-}}p^k_i(v, y_2, \ldots, y_{ar(p)})) = \{\emptyset \rhd \{\neg\mathsf{slot}^{k-1}, \mathsf{slot}^k\}\}\\ &\cup \{\{\mathsf{poss}^{k-1}(o_1, \ldots, o_m), p(v, o(y_2), \ldots, o(y_{ar(p)}))\} \rhd\\ &\qquad \{\mathsf{poss}^k(o_1, \ldots, o_m)\} : (o_1, \ldots, o_m) \in \Omega_o^m\}.\end{aligned}$$

To apply a step action, an atom has to be programmed first. The unconditional effect is moving from slot σ_{k-1} to slot σ_k. In addition, the step action updates the possible assignments to the bound variables $x_1, \ldots, x_m$.

For an assignment $(o_1, \ldots, o_m)$ to be possible at slot σ^k, it has to be possible at σ^{k-1}, and the atom $p(v, y_2, \ldots, y_{ar(p)})$ programmed at slot k has to induce a fluent that is currently true. Let $o(y)$ denote the object among $o_1, \ldots, o_m$ that is associated with the bound variable y. For example, if $y = x_2$, then $o(y) = o(x_2) = o_2$. Then the induced fluent is

given by $p(v, o(y_2), \ldots, o(y_{ar(p)}))$. Note that there is one conditional effect for each possible assignment $(o_1, \ldots, o_m)$. If $k = 1$, the condition $\mathsf{poss}^{k-1}(o_1, \ldots, o_m)$ is removed since all assignments are possible prior to evaluating the first atom.

Once we have iterated over all atoms, we have to check whether there remains at least one possible assignment, thereby evaluating the entire conjunctive query. For each line i, let eval_i be an action defined as

$$\begin{aligned}\mathsf{pre}(\mathsf{eval}_i) &= \{\mathsf{pc}_i, \mathsf{slot}^q\},\\ \mathsf{cond}(\mathsf{eval}_i) &= \{\emptyset \rhd \{\mathsf{eval}\}\}\\ &\cup \{\{\mathsf{poss}^q(o_1, \ldots, o_m)\} \rhd \{\mathsf{acc}\} : (o_1, \ldots, o_m) \in \Omega_o^m\}.\end{aligned}$$

Action eval_i is only applicable once we are at the last slot σ_q. The conditional effects add the fluent acc if and only if there remains a possible assignment to $x_1, \ldots, x_m$ at σ_q.

Finally, we can now use the result of the evaluation to determine the program line that we jump to. For each pair of lines i, i', let $\mathsf{jmp}_{i,i'}$ be an action defined as

$$\begin{aligned}\mathsf{pre}(\mathsf{jmp}_{i,i'}) &= \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{goto}(i')}, \mathsf{slot}^q, \mathsf{eval}\},\\ \mathsf{cond}(\mathsf{jmp}_{i,i'}) &= \{\emptyset \rhd \{\neg\mathsf{pc}_i, \neg\mathsf{eval}, \neg\mathsf{acc}, \neg\mathsf{slot}^q, \mathsf{slot}^0\}\}\\ &\cup \{\{\neg\mathsf{acc}\} \rhd \{\mathsf{pc}_{i'}\}, \{\mathsf{acc}\} \rhd \{\mathsf{pc}_{i+1}\}\}\\ &\cup \{\emptyset \rhd \{\neg\mathsf{poss}^k(o_1, \ldots, o_m) : 1 \leq k \leq q, \forall j.o_j \in \Omega_o\}\}.\end{aligned}$$

The effect is to jump to line i' if acc is false, else continue execution on line $i + 1$. We also delete fluents eval and acc, as well as all instances of $\mathsf{poss}^k(o_1, \ldots, o_m)$ in order to reset the evaluation mechanism prior to the next evaluation of a conjunctive feature. The current slot is also reset to σ_0.

4 Classification with Planning Programs

Our extension of planning programs with conjunctive queries allows us to model supervised classification tasks as if they were generalized planning problems. Formally, the learning of a noise-free classifier from a set of labeled examples $\{e_1, \ldots, e_T\}$, where each example e_t, $1 \leq t \leq T$, is labeled with a class in $\{c_1, \ldots, c_Z\}$, can be viewed as a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ such that each individual planning problem $P_t = \langle \Psi, \Omega, A, I_i, G_i \rangle$, $1 \leq t \leq T$, models the classification of the t^{th} example:

- Ψ, Ω induces the set of fluents F representing the learning examples and their labels.
- A contains the actions necessary to associate a given example with a class. For instance, in a binary classification task, $A = \{\mathsf{setPositive}, \mathsf{setNegative}\}$.
- I_t contains the fluents that describe the t^{th} example and G_t the fluent that describes the label of the t^{th} example.

The solution Π to a generalized planning problem $\mathcal{P}$ that models a classification task is a noise-free classifier that covers all learning examples.

This model is particularly natural for classification tasks in which both the examples and the classifier are described using logic. *Michalski's train* [Michalski *et al.*, 2013] is a good example of such tasks. It defines 10 different trains (5 traveling east and 5 traveling west) and the classification target is finding rules that cause a train to travel east or west. Trains are defined using the following relations: which wagon is in a given train, which wagon is in front, the wagon's shape, its number of wheels, whether it has a roof or not (closed or open), whether it is long or short, the shape of the objects the wagon is loaded with and the class of the train.

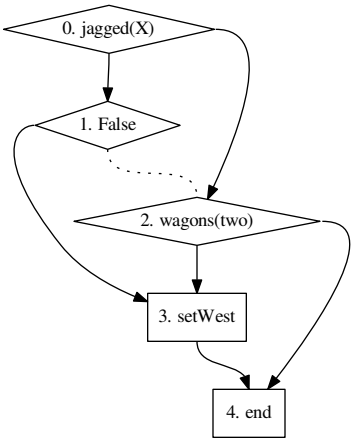

Figure 3: Planning program that encodes a noise-free classifier for the Michalski's train problem.

In more detail, the generalized planning problem encoding the *Michalski's train* task would be:

- Fluents F induced from $\Psi = \{$(wagons ?Number), (hasCar ?Car), (infront ?Car ?Car), (shape ?Car ?Shape), (wheels ?Car ?Number), (closed ?Car), (open ?Car), (long ?Car), (short ?Car), (double ?Car), (jagged ?Car), (load ?Car ?Shape ?Number), (eastbound), (westbound)$\}$.
- Actions $A = \{setWest\}$. We assume that any example has initial class eastbound, causing the resulting planning programs to be more compact.

$$\begin{aligned}\mathsf{pre}(\mathsf{setWest}) &= \{\emptyset\},\\ \mathsf{cond}(\mathsf{setWest}) &= \{\emptyset \rhd \neg\mathsf{eastbound}\},\\ &\quad\ \{\emptyset \rhd \mathsf{westbound}\}.\end{aligned}$$

- Each initial state I_t defines the t^{th} train and G_t defines its associated class (east or west).

Figure 3 shows a planning program encoding a noise-free classifier for the Michalski's train problem. As explained, the program assumes that examples initially have class eastbound. Line 1 of the program is an unconditional jump to line 3.

5 Evaluation

In all experiments, we run the classical planner Fast Downward [Helmert, 2006] with the LAMA-2011 setting [Richter and Westphal, 2010] on a Intel Core i5 3.10GHz x 4 with a 4GB memory bound and time limit of 3600s.

	Lines	Slots	Vars	Time	Len
List	3	2	1	1.0	135
Summatory	3	2	1	13.0	40
Trains	5	(1,1,1)	(1,1,1)	61.0	101
And	4	(2,2)	(1,1)	0.7	49
Or	4	(2,2)	(1,1)	1.4	49
Xor	4	(2,2)	(2,1)	1.5	44

Table 1: Program lines, slots and variables of the features, time (in seconds) elapsed while computing the solution, and plan length required to generate and verify the solution.

We evaluate our method in two kinds of benchmarks. We first consider benchmarks from generalized planning where the target is generating a plan that generalizes without providing any prior high-level representation of the states. This set of benchmarks include iterating over a list and computing the n^{th} term of the summatory series. On the other hand, we consider binary classification tasks which include Michalski's train (cf. Section 4) as well as generating the classifiers corresponding to the logic functions $and(X_1, X_2)$, $or(X_1, X_2)$ and $xor(X_1, X_2)$. Table 1 summarizes the obtained results. We report the number of program lines used to solve the generalized planning problem, the number of slots and bound variables required to learn the features (a list means that more than one feature was learned), the time taken to generate the program, and the plan length.

We briefly describe the features and the programs learned for the different domains. In the list domain we learn the feature $i == n$ for the program (0. $visit$, 1. $inc(i)$, 2. $goto(0, i!{=}n)$). In the summatory domain we learn the feature $b = 0$ for the program (0. $sum(a, b)$, 1. $dec(b)$, 2. $goto(0, b!{=}0)$). For Michalski's train we learn the program and features shown in Figure 3. The programs for $and(X_1, X_2)$, $or(X_1, X_2)$ and $xor(X_1, X_2)$ have the same structure: they learn a first feature that captures if a variable is false (true for the *or* function, and one true and one false for *xor*) and a second feature to capture that the class of the example was set to negative. This is the 4-line program for the $and(X_1, X_2)$ function: (0. $goto(3, !X = False)$, 1. $setFalse$, 2. $goto(4, class = False)$, 3. $setTrue$, 4. end).

6 Related Work

Our approach for learning high-level state features (and classifiers) is inspired by *version space learning* [Mitchell, 1982]. The hypothesis to learn consists of logic clauses and examples are logic facts that restrict the hypothesis forcing it to be consistent with the examples. Inductive Logic Programming (ILP) [Muggleton, 1999] also intersects Machine Learning (ML) and Logic Programming to generate hypotheses from examples. ILP has traditionally been considered a binary classification task but, in recent years, it covers the whole spectrum of ML such as regression, clustering and association analysis. The main contribution of our approach with respect to version space learning and ILP is the use of a classical planner to build and validate the learned hypotheses.

Previous work on computing generalized plans in the form of generalized policies already attempted to automatically generate higher-level state representations [Khardon, 1999]. Two examples are learning generalized policies from solved instances using description logic [Martín and Geffner, 2004] and taxonomic syntax [Yoon *et al.*, 2008] to represent and reason about classes of objects. In these works, planning and learning were clearly separate phases producing noisy learning examples in many cases. In contrast, our approach tightly integrates planning and learning.

Generating high-level state features for generalized planning is also related to previous work on First Order MDPs [Boutilier *et al.*, 2001; Gretton and Thiébaux, 2004]. These works adapt traditional dynamic programming algorithms to the symbolic setting and automatically generate first-order representations of the value function with first-order regression. The main contribution of our approach with respect to this research line is that we follow a compilation approach to generate useful state abstractions with off-the-shelf planners.

7 Conclusion

In generalized planning problems, high-level state features are traditionally hand-coded which requires significant human expertise. We have proposed a novel approach to automatically generating these features by tightly integrating the computation of planning programs with the computation of the features. This integration is achieved incorporating conjunctive queries into planning programs and extending an existing compilation from generalized planning to classical planning [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016] such that it can be exploited by an off-the-shelf planner.

Currently we are only able to generate high-level state features in the form of conjunctive queries, and hence we cannot model features with unbounded transitive or recursive closures. This kind of features are known to be useful for some planning domains, e.g. the *above* feature, the transitive closure of *on*, for the Blocksworld domain. In the near future we would like to extend our approach to generating more expressive features.

In addition, our approach naturally models classification tasks in which both examples and classifiers are represented using logic. The aim of this research direction is not competing with existing ML algorithms; indeed, we cannot deal with noisy examples. Instead, our aim is to provide a new formalism capable of representing tasks that integrate classification and planning. Moreover, we bring a new landscape of challenging benchmarks to classical planning.

Acknowledgment

This work is partially supported by grant TIN2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain. Sergio Jiménez is partially supported by the *Juan de la Cierva* program funded by the Spanish government.

References

[Bonet *et al.*, 2010] Blai Bonet, Hector Palacios, and Hector Geffner. Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*, 2010.

[Boutilier *et al.*, 2001] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order mdps. In *IJCAI*, volume 1, pages 690–700, 2001.

[Chandra and Merlin, 1977] Ashok Chandra and Philip Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM Symposium on Theory of Computing*, 1977.

[Cresswell and Coddington, 2004] Stephen Cresswell and Alexandra M. Coddington. Compilation of ltl goal formulas into pddl. In *ECAI*, pages 985–986, 2004.

[Gretton and Thiébaux, 2004] Charles Gretton and Sylvie Thiébaux. Exploiting first-order regression in inductive policy selection. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 217–225. AUAI Press, 2004.

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

[Hoffmann and Edelkamp, 2005] Jörg Hoffmann and Stefan Edelkamp. The deterministic part of ipc-4: An overview. *Journal of Artificial Intelligence Research*, pages 519–579, 2005.

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

[Hu and De Giacomo, 2013] Yuxiao Hu and Giuseppe De Giacomo. A generic technique for synthesizing bounded finite-state controllers. In *International Conference on Automated Planning and Scheduling*, 2013.

[Hu and Levesque, 2011] Yuxiao Hu and Hector J. Levesque. A correctness result for reasoning about one-dimensional planning problems. In *International Joint Conference on Artificial Intelligence*, pages 2638–2643, 2011.

[Ivankovic and Haslum, 2015] Franc Ivankovic and Patrik Haslum. Optimal planning with axioms. In *International Joint Conference on Artificial Intelligence*, pages 1580–1586. AAAI Press, 2015.

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

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

[Martín and Geffner, 2004] Mario Martín and Hector Geffner. Learning generalized policies from planning examples using concept languages. *Appl. Intell*, 20:9–19, 2004.

[Michalski *et al.*, 2013] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[Mitchell, 1982] Tom M Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

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

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

[Segovia-Aguas *et al.*, 2016] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning with procedural domain control knowledge. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2016.

[Srivastava *et al.*, 2011] Siddharth Srivastava, Neil Immerman, Shlomo Zilberstein, and Tianjiao Zhang. Directed search for generalized plans using classical planners. In *International Conference on Automated Planning and Scheduling*, pages 226–233, 2011.

[Thiébaux *et al.*, 2005] Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In defense of pddl axioms. *Artificial Intelligence*, 168(1):38–69, 2005.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The prodigy architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[Yoon *et al.*, 2008] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *The Journal of Machine Learning Research*, 9:683–718, 2008.