Learning Domain Control Knowledge for TLPlan and Beyond

Tomás de la Rosa
Departamento. de Informática
Universidad Carlos III de Madrid
Leganés (Madrid). Spain
trosa@inf.uc3m.es

Sheila McIlraith
Computer Science Department
University of Toronto
Toronto, Ontario, Canada
sheila@cs.toronto.edu

Abstract

Domain control knowledge has been convincingly shown to improve the efficiency of planning. In particular, the forward chaining planner, TLPlan, has been shown to perform orders of magnitude faster than other planning systems when given appropriate domain-specific control information. Unfortunately, domain control knowledge must be hand coded, and appropriate domain control knowledge can elude an unskilled domain expert. In this paper we explore the problem of *learning* domain control knowledge in the form of domain control rules in a subset of linear temporal logic. Our approach is realized in two stages. Given a set of training examples, we augment the feature space by learning useful derived predicates. We use these derived predicates with the domain predicates to then learn domain control rules for use within TLPlan. Experimental results demonstrate the effectiveness of our approach.

Introduction

Hand-tailored automated planning systems augment planning systems with extra domain-specific control knowledge (DCK) and a means of processing that knowledge to help guide search. They have proven extremely effective at improving the efficiency of planning, sometimes showing orders of magnitude improvements relative to domain-independent planners and/or solving problems that evade other planners. One of the keys to their success has been their ability to drastically reduce the search space by eliminating those parts of the space that do not comply with the DCK. The best-known hand-tailored planning systems are TLPLAN (Bacchus and Kabanza 2000), TALPLANNER (Kvarnström and Doherty 2000), and SHOP2 (Nau et al. 2003). SHOP2 provides DCK in the form of Hierarchical Task Networks (HTNs) – tasks that hierarchically decompose into subtasks and ultimately into primitive actions. In contrast, both TLPLAN and TALPLANNER exploit DCK in the form of control rules specified in Linear Temporal Logic (LTL) (Pnueli 1977).

Unfortunately, the effectiveness of hand-tailored planning systems relies on the quality of the DCK provided by an expert, and while even simple DCK has proven helpful, the provision of high-quality DCK can be time consuming and challenging. In 2003, Zimmerman and Kambhampati surveyed the use of machine learning techniques within automated planning, identifying offline learning of domain knowledge and search control as promising avenues for future research (Zimmerman and Kambhampati 2003). Since then there has been renewed interest in the application of machine learning techniques to planning in order to automatically acquire control knowledge for domain-independent planners. However, there has been no work on learning LTL domain control rules.

In this paper we explore the problem of learning DCK in the form of domain control rules in a subset of LTL. The control rules are learned from a set of training examples, and are designed to serve as input to TLPLAN. TLPLAN is a highly optimized forward-chaining planner that uses domain control rules, specified in LTL, to prune partial plans that violate the rules as it performs depth-first search. Control rules are typically encoded using the predicates of the original domain together with new derived predicates that capture more complex relationships between properties of a state. As such the challenge of learning LTL control rules for input to TLPLAN requires not only learning the rules themselves but also the derived predicates that act as new features in the feature space of the learning task.

The contributions of this paper include: an algorithm for learning LTL domain control rules from training examples, exploiting techniques from inductive logic programming (ILP); and a technique for generating derived predicates in order to expand the feature space of a learning problem. The creation of a good feature space is central to effective learning. As such the ability to generate relevant derived predicates has broad applicability in a diversity of learning problems for planning. We evaluate our techniques on three benchmarks from previous IPC competitions. The results demonstrate the effectiveness of our approach, since the new learning-based TLPLAN is competitive with state-of-the-art planners. In the next section, we briefly review LTL and the planning model we consider in our work. This is followed by a more precise statement of the learning task, including the language, the generation of the training examples and the learning algorithm. Next we discuss our experimental evaluation. We conclude with a summary of our contributions and a discussion of related and future work.

Preliminaries

In this section we define the class of planning problems we consider and review LTL notation and semantics.

Planning: Our planning formalism corresponds to the non-numeric and non-temporal version of the Planning Domain Definition Language (PDDL). For simplicity, we assume that our domains are not encoded with conditional effects and that preconditions are restricted to conjunctions of literals. We use first-order predicate logic notation, but our planning problems range over finite domains and thus can be expressed as propositional planning problems.

A *planning domain* $\mathcal{D}$ has a first-order language $\mathcal{L}$ with the standard first-order symbols, variables and predicate symbols. We assume our language is function-free with the exception of 0-ary constant terms. A *planning task* Π is defined as the tuple $(\mathcal{C}, \mathcal{O}, \mathcal{X}, s_0, G)$ where:

1. $\mathcal{C}$ is a finite set of constants in Π. We refer to $\mathcal{L}(\Pi)$ as the domain language extended with task-specific constants. $\mathcal{L}(\Pi)$ defines the state space for the task. A state s of Π is a conjunction of atoms over $\mathcal{L}(\Pi)$.
2. $\mathcal{O}$ is the set of operators, where each $o \in \mathcal{O}$ is a pair (pre, eff) where pre is a first-order formula that stipulates the preconditions of an operator, and eff is the effect of o, normally expressed as a conjunction of literals. The set of operators $\mathcal{O}$ define the set of applicable actions $\mathcal{A}$ by grounding variables in o with suitable constants drawn from $\mathcal{C}$. An action $a \in \mathcal{A}$ is applicable in a state s if pre holds in s. When an action is applied, the result is a new state s' where atoms in s are modified according to eff.
3. $\mathcal{X}$ is a set of axioms where each $\delta \in \mathcal{X}$ is the pair $(head, body)$ with $head$ an atom and $body$ a first-order formula. Axioms in $\mathcal{X}$ define the set of *derived predicates*. The rest of predicates in $\mathcal{L}(\Pi)$ are called *fluent predicates*.
4. s_0 is the set of atoms representing the initial state.
5. G is the set of atoms representing the goals.

A solution to the planning task Π is a plan $\pi = (a_1, \ldots, a_n)$ where $a_i \in \mathcal{A}$ and each a_i is applied sequentially in state s_{i-1} resulting in state s_i. s_n is the final state and $G \subseteq s_n$. If there is no a plan with fewer actions than n then the plan is said to be optimal.

LTL: First-order linear temporal logic (FOLTL) is an extension of standard first-order logic with temporal modalities. The first-order language $\mathcal{L}$ is augmented with modal operators $\Box$ (always), $\Diamond$ (eventually), $\mathbf{U}$ (until), and $\bigcirc$ (next). We refer to this new language as $\mathcal{LT}$. Formulas in $\mathcal{LT}$ are constructed using logical connectives in the standard manner and if ϕ_1 and ϕ_2 are well-formed formulas (wff), then $\Box\phi_1, \Diamond\phi_1, \bigcirc\phi_1$ and $\phi_1 \mathbf{U} \phi_2$ are wff as well. Formulas are interpreted over a sequence of states $\sigma = \langle s_0, s_1, \ldots \rangle$ where each s_i shares the same universe of discourse (i.e., in a particular planning task Π). Here we briefly describe the semantics of temporal operators. For a more general description refer to (Emerson 1990) or in the planning context to (Bacchus and Kabanza 2000). If ϕ_1 and ϕ_2 are $\mathcal{LT}$ formulas, and v is a function that substitutes variables with constants of $\mathcal{LT}$, we say temporal operators are interpreted over a state sequence σ as follows:

1. $\langle \sigma, s_i, v \rangle \models \Box\phi_1$ iff for all $j \geq i$, $\langle \sigma, s_j, v \rangle \models \phi_1$. i.e., ϕ_1 is true for all states in σ.
2. $\langle \sigma, s_i, v \rangle \models \bigcirc\phi_1$ iff $\langle \sigma, s_{i+1}, v \rangle \models \phi_1$. i.e., ϕ_1 is true in next state in σ.
3. $\langle \sigma, s_i, v \rangle \models \Diamond\phi_1$ iff there exists $j \geq i$, such that $\langle \sigma, s_j, v \rangle \models \phi_1$. i.e., ϕ_i will eventually become true in some state in the future.
4. $\langle \sigma, s_i, v \rangle \models \phi_1 \mathbf{U} \phi_2$ iff there exists $j \geq i$ such that $\langle \sigma, s_j, v \rangle \models \phi_2$ and for all k, with $i \geq k < j$, $\langle \sigma, s_k, v \rangle \models \phi_1$. i.e., ϕ_1 is and remains true until ϕ_2 becomes true.

LTL and TLPLAN: As in TLPLAN, we will not use the $\Diamond$ (eventually) modal operator, since it does not result in pruning of the planning search space. (I.e. $\Diamond\phi$ will always hold in the current state, because the planner expects ϕ to become true sometime in the future.) Note that TLPLAN also exploits an additional goal modality, which is helpful for planning and which we also exploit. GOALf expresses that f is a goal of the planning problem being addressed. In order to determine whether or not a plan prefix, generated by forward chaining planner, TLPLAN, could lead to a plan that satisfies an arbitrary LTL formula, TLPLAN exploits the notion of progression over LTL with finite domain and bounded quantification. Intuitively, progression of an LTL formula breaks a formula down into a formula that must hold in the current state, conjoined with a formula that must hold in the rest of the plan being constructed. The progression algorithm is defined for each LTL modality, and is described in detail in (Bacchus and Kabanza 2000).

Learning Restricted LTL

The task we address in this paper is to automatically learn domain control rules for input to TLPLAN. Following (Bacchus and Kabanza 2000) consider the Blocksworld example

$$\Box(\forall x_{:clear(x)} goodtower(x) \rightarrow \bigcirc(clear(x) \vee \exists y_{:on(x,y)} goodtower(y))$$

where $goodtower(x)$ is a derived predicate encoding that block x and all others blocks below are in their final position. The rule in the example indicates that all good towers will remain the same in the next state or if another block is stacked on the top of a good tower, it is also well placed.

The learning paradigm we use is similar to the one used in the learning track of IPC-2008. A learning component takes as input a set of high-quality problem- or domain-specific bootstrap plans from which plan properties are learned that are used to augment the planner or planning domain so that subsequent plan generation over related planning instances is improved.

In this particular instance, we generate our own set of training examples from a set of training problems, as described in detail below. Normally training examples are plans for simpler (generally smaller) formulations of the planning problem being addressed. From here they may be transformed into a representation suitable for input to the learning component. In our problem, the input to our learning component ultimately takes the form of sequences of states that result from the execution of (optimal) plans. Given this training input, we proceed in two steps: first learning new features of our training examples through the generation of derived predicates, and then from these augmented training examples, learning our domain control rules. The first step – generation of derived predicates – is an optional step and we evaluate the quality and impact of our control rules both with and without the use of our learned derived predicates.

As noted in the introduction, TLPLAN exploits LTL domain control rules by pruning partial plans (states) that violate the domain control rules. As noted by the developers of TLPLAN, only a subset of LTL formulas have the capacity to prune states. For example, if in every state the property φ must hold, then the corresponding TLPLAN LTL formula will be $\Box\varphi$, i.e., *a*lways φ. Indeed, every control rule reported in (Bacchus and Kabanza 2000) was of this form. As such, we restrict the problem to finding φ restricted to formulas over $\mathcal{L} \cup \{\bigcirc, \mathbf{U}\}$.

Definition 1 (R-LTL formula) *An R-LTL formula is a restricted LTL formula of the form $\Box\varphi$, where φ is a well-formed formula in the language $\mathcal{L} \cup \{\bigcirc, \mathbf{U}\}$ (written $\mathcal{LT}_R$).*

These R-LTL formulas will be learned from the input to our learning component which, as described above, takes the form of sequences of states obtained from simpler related planning problems that we have solved.

Definition 2 (Optimal state sequence) *Given a planning task Π, an optimal state sequence $\sigma^+ = \langle s_0, \ldots, s_m \rangle$ is a state sequence where every s_i belongs to a state sequence resulting from the execution of an optimal plan $\pi = \langle a_1, \ldots, a_n \rangle$ from the initial state, with $m \leq n$.*

Definition 3 (Suboptimal state sequence) *Given a planning task Π, a suboptimal state sequence $\sigma^- = \langle s_0, \ldots, s_m \rangle$ is a state sequence where at least one s_i, with $i > 0$, does not belong to any state sequence resulting from the execution of every optimal plan $\pi = \langle a_1, \ldots, a_n \rangle$ from the initial state.*

Although we use optimal sequences to induce DCK, we do not expect to learn control formulas that generate solely optimal plans. The inherent process of induction over a restricted set of examples, and the complex structure of the hypothesis spaces hinder this.

The Control Rule Learning Task: Given,

- the language $\mathcal{LT}_R$ for the target formula.
- the universe of discourse[1], H, for the target formula.
- background knowledge $\mathcal{B}$ on $\mathcal{LT}_R(H)$, expressed as a (possibly empty) set of axioms.
- A set of positive and negative examples, such that:
  - for each example indexed by i, $1 \leq i \leq n$ there is an associated universe of discourse, $D_1, \ldots, D_n$
  - A set of positive examples E^+ in $\mathcal{LT}_R(D_i)$, representing optimal state sequences σ^+.
  - A set of negative examples E^- in $\mathcal{LT}_R(D_i)$, representing suboptimal state sequences σ^-.

The target of learning is to find a (hypothesis) formula φ in $\mathcal{LT}_R(H)$ such that all examples in E^+ are interpretations of φ and all examples in E^- are not interpretations of φ.

Training Examples

As noted above, the input to our R-LTL learning component is a set positive and negative examples. I.e., a set of sequences of states that results from the execution of actions from optimal (respectively, suboptimal) plans. In order to generate this input, we start with a set of training problems – a set of planning problems to be solved. These problems are solved with a Best-first Branch-and-Bound algorithm (BFS-BnB) using the FF relaxed plan heuristic (Hoffmann and Nebel 2001). After finding a first plan, the algorithm tries to find shorter plans until it has explored the entire search space. The right way of achieving this is with an admissible heuristic. In practice, however, using known domain-independent admissible heuristic leads the algorithm to only solve very small problems, many times uninteresting from learning perspective. The FF heuristic works reasonably well for meaningful small problems with rare over-estimation during the exhaustive BFS-BnB search. Exceptions are treated as noise in the training data. After solving each of the problems in the original suite, the result is a set of all possible (optimal) solutions. Training problems should be small enough to allow BFS-BnB to perform the exhaustive search, but also should be big (interesting) enough to deliver some knowledge to the training base.

We have characterized the control rule learning task with respect to a set of optimal and suboptimal state sequences. However, the relevant feature of these sequences is the good and bad *transitions* that exist between consecutive states. A good transition, as characterized by a 2-sequence $\sigma_T = \langle s_i, s_{i+1} \rangle$, maintains the plan on an optimal path. In contrast, a bad transition, also characterized by a 2-sequence, denotes the point at which a heretofore optimal plan deviates from its optimal path, becoming suboptimal. Our training examples reflect this. E^+ contains all *good transitions* i.e., all 2-sequences σ_T^+, while E^- contains all *bad transitions*, $\sigma_T^- = \langle s_i, s_{i+1} \rangle$ where s_{i+1} is part of a suboptimal path. Note that those 2-sequences where both states are in a suboptimal path are not of interest for learning because they don't reflect the transition from an optimal to a suboptimal path, which we want our control rules to avoid.

Thus, given a planning task Π and the sequence $\sigma_T = \langle s_i, s_{i+1} \rangle$, an example for our learning component will comprise the following information for each 2-sequence:

class: positive or negative

current state: all atoms in s_i

next state: for every atoms in s_{i+1} such that its predicate symbol, P, appears in some effect of an operator in $\mathcal{O}$, assert a new predicate $next\_P$, with parameters corresponding to the original atom. It is irrelevant to have static atoms in both current and next state. This simplification does not confer additional meaning but does reduces the size of the example.

goal predicates: assert new predicates as follows,

- for every P in G assert a new predicate $goal\_P$.
- for every P in G that is true in s_i, assert a new predicate $achievedgoal\_P$.
- for every P in G that is false in s_i, assert a new predicate $targetgoal\_P$.

These new meta-style predicates are used within the learning algorithm to induce particular LTL formulas. For example, the $next$ meta-predicate serves to induce $\bigcirc\varphi$ sub-

[1]Recall that a universe of discourse is the set of objects about which knowledge is being expressed.

formulas, and $goal$ serves to induce the $goal$ modality used in TLPLAN. $achieved$ and $targetgoal$ are pre-defined predicates for all domains with the appropriate semantics.

LTL Formula Learning Algorithm

In the previous subsection, we described the control rule learning task and the form of our example input. In order to realize this algorithm, we use the ICL Tool *(Inductive Classification Logic)* which is part of the ACE Toolkit (Raedt et al. 2001). ICL learns first-order formulas (in DNF or CNF format) with respect to a specific class of examples. For our work, we learn the $positive$ class described in the learning examples. The ICL algorithm performs a beam search within the hypothesis space, using as successor the possible refinement of rules. Since handling all formula refinements is in general intractable, ICL and many others ILP algorithms imposes declarative constraints to the inductive hypothesis search called *language bias*. The language bias in ICL is defined in a language called $\mathcal{D}lab$ (Nedellec et al. 1996). In $\mathcal{D}lab$ one can indicate how many and which predicates of the language can be introduced to the refinement of a single formula. In our work we automatically construct the language bias in $\mathcal{D}lab$ syntax using the types and predicate definition of the domain. The only restriction we impose is to have one or zero literals in the head of a clause. In the latter case the head is the *false* constant. After learning first-order formulas with ICL, we translate them into R-LTL formulas (i.e., changing $next\_P$ and $goal\_P$ meta-predicates as explained previously).

To this point, we have not addressed the issue of learning formulas containing the $\mathbf{U}$ modal operator. We argue that it is not necessary as a result of the following theorem.

Theorem 1 *An R-LTL formula $\Box\varphi_1$, where φ_1 only has temporal modalities over first-order expressions (non-nested $\bigcirc$and $\mathbf{U}$), has an equivalent formula $\Box\varphi_2$ where φ_2 is over the language $\mathcal{L} \cup \{\bigcirc\}$.*

Proof Sketch:If we treat any first-order expression with a modal operator as an atomic formula, the formula φ_1 could be written in DNF or CNF format. $\Box$ operator is distributive over $\wedge$ and $\vee$, so we can for any sub-formula containing $\mathbf{U}$, get the sub-formula in the form $\Box(\phi_1\mathbf{U}\phi_2)$. Following the rules for progression of LTL (Bacchus and Kabanza 2000):

$$\Box(\phi_1\mathbf{U}\phi_2) \equiv \phi_1\mathbf{U}\phi_2 \wedge \bigcirc\Box(\phi_1\mathbf{U}\phi_2)$$
$$\phi_1\mathbf{U}\phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \bigcirc(\phi_1\mathbf{U}\phi_2))$$

Additionally, the sub-formula $\bigcirc(\phi_1\mathbf{U}\phi_2)$ cannot be false in $\Box(\phi_1\mathbf{U}\phi_2)$ because it is actually true in every state of the sequence being progressed, thus $\Box(\phi_1\mathbf{U}\phi 2) \equiv \Box(\phi_2 \vee \phi_1)$.

Learning Derived Predicates

In the previous section we described our approach to learning R-LTL formulas for input to TLPLAN. In the introduction, we noted that LTL control rules are often specified in terms of additional derived predicates that capture additional properties of the state of the world. In this section we discuss how to generate such derived predicates with a view to improving the quality of the R-LTL formulas we learn.

Derived predicates serve two purposes within planning. They serve as a parsimonious way of encoding certain preconditions, and they capture more complex properties of the state of the world. Given a set of predicates that are sufficient for planning, our goal is to learn derived predicates that will be of utility in the expression of control knowledge. We have identified three types of derived predicates that are found in DCK reported by (Bacchus and Kabanza 2000) as well as in other learning approaches detailed in the related work section. They are:

Compound Predicates: The union of two predicates. For instance, in the Blocksworld domain, predicates (*clear A*) and (*on A B*) produce the new predicate (*clear_on A B*).

Abstracted Predicates: A new predicate that is created by ignoring a variables. For instance, predicate (*on A B*) produces the new predicate (*abs_on A*), representing that block *A* is on another block.

Recursive Predicate: A predicate with at least two arguments of the same type begets a predicate that encodes transitive relations between constants. For instance

$$above(A,B) \equiv on(A,B) \vee (on(A,X) \wedge above(X,B))$$

These types of derived predicates are primitive schemas, analogous to relational clichés (Silverstein and Pazzani 1991), which are patterns for building conjunctions of predicates, together with constraints on the combination of predicates and variables.

The above mentioned predicates can also be combined in different ways to produce more complex predicates. For instance, in the Parking domain, a new predicate that relates the curb where a car is parked can be derived by abstracting a compound predicate as follows:

$$\begin{aligned} &behind\_car\_at\_curb\_num(Car, FrontCar, Curb) \equiv \\ &\quad behind\_car(Car, FrontCar) \wedge \\ &\quad at\_curb\_num(FrontCar, Curb) \end{aligned}$$

$$\begin{aligned} &abs\_behind\_car\_at\_curb\_num(Car, Curb) \equiv \\ &\quad \exists x(behind\_car(Car, x) \wedge at\_curb\_num(x, Curb)) \end{aligned}$$

In this work, we learn derived predicates to augment the background knowledge used to learn the R-LTL formulas. The syntactic form of the above three derived predicate types is described in terms of the head and body of axioms.

Definition 4 (Compound predicate) *Given two predicates, $pred1(a_1,\ldots,a_n)$ and $pred2(b_1,\ldots,b_m)$ a compound predicate is an axiom $\delta = \langle head, body\rangle$ where head is a new predicate $pred1\_pred2(c_1,\ldots,c_o)$ and body $= pred1(a_1,\ldots,a_n) \wedge pred2(b_1,\ldots,b_m)$. The predicate symbol in head is the string concatenation of $pred1$ and $pred1$ predicate symbols with an "_" in between, and the arguments $c_1,\ldots,c_o$ are the free variables in body.*

Definition 5 (Abstracted predicate) *Given a predicate $pred(a_1,\ldots,a_n)$, an abstracted predicate is an axiom $\delta = \langle head, body\rangle$ where head is a new predicate $abs\_pred(c_1,\ldots,c_o)$ and body is $pred(a_1,\ldots,a_n)$ with one a_i existentially bound, called typed variable. Arguments $c_1,\ldots,c_o$ are the free variables in body.*

Quantifications in TLPLAN is bounded, so to build abstracted predicates, the typed variable is bounded to the unary predicate encoding the argument type. Therefore, the semantics of an abstracted predicate is that the typed variable may be replaced by any constant of the variable type.

Definition 6 (Link-recursive predicate) *Given a 2-ary predicate $pred(a_1, a_2)$, a link-recursive predicate is an axiom $\delta = \langle head, body\rangle$ where $head = link\_rec\_pred(a_1, a_2)$ and body has the form:*

$$pred(a_1, a_2) \vee \exists a_x.(pred(a_1, a_x) \wedge link\_rec\_pred(a_x, a_2))$$

Definition 7 (End-recursive predicate) *Given a 2-ary predicate $pred1(a_1, a_2)$ and a n-ary predicate $pred2(a_x, b_1, \ldots, b_{n-1})$ with $n \geq 1$, an end-recursive predicate is an axiom $\delta = \langle head, body\rangle$ where $head = end\_rec\_pred(a_x)$ and body has the form:*

$$pred2(a_x, b_1, \ldots, b_{n-1}) \vee \exists a_y.(pred(a_1, a_2) \wedge end\_rec\_pred(a_y))$$

with substitutions $[a_x/a_1], [a_y/a_2]$ for right recursion or $[a_x/a_2], [a_y/a_1]$ for left recursion.

The task of learning derived predicates consists of searching through the space of new derived predicates until reaching certain criteria for improving the task of learning the R-LTL formula. We elaborate on the form of this criteria below. The space of new derived predicates is defined by the possible compound, abstracted, and recursive predicates that can be constructed from the original predicates of the domain model. Formally, we say that given the inputs (with empty background knowledge) for the learning task of a formula φ in $LT_R(H)$, the task of acquiring derived predicates for the learning task consists of selecting background knowledge $\mathcal{B}_i$ in the form of logic programs, such that φ improves a given evaluation criterion.

Derived Predicates Learning Algorithm

The learning algorithm performs a beam search in the space of possible new derived predicates. As with most ILP algorithms a best-first search is not feasible. Figure 1 shows the pseudo-code for the algorithm. Derived predicate functions (i.e., *compound*, *abtracted* and *recursive*) compute all possible combinations of each type of derived predicate, given a list of current predicates in node n. Each successor adds a single new derived predicate to the current predicates from the combinations computed by these functions. The algorithm stops when at depth $d + 1$ any node improved the best evaluation at depth d. It returns the best set of predicates found so far. The argument $eval\_set$ is a set of problems (different from the training set) used for evaluation purposes. At each node, a set of rules is induced with the current predicates and function f_{eval} evaluates how good these rules are with regard to problems in $eval\_set$. The function f_{eval} could obviously be replaced by the accuracy of the rule learner. However, we found in empirical evaluations that guiding the search with this accuracy does not guarantee an increase in the number of solved problems.

The translation of derived predicate into TLPLAN DCK is straightforward. Each derived predicate returned by LEARNDERIVEDPREDS is defined as a new predicate using

```
LEARNDERIVEDPREDS ($preds$, $f_{eval}$, $eval\_set$, $k$): $preds$

$beam \leftarrow \{preds\}$; $irrelevant \leftarrow \emptyset$
repeat
  $best\_fn \leftarrow f_{acc}$(first($beam$)); $successors \leftarrow \emptyset$
  for all $n$ in $beam$ do
    Add $\{compound(n) \cup abstracted(n) \cup recursive(n)\} - irrelevant$
        to $successors$
  for all $n'$ in $successors$ do
    evaluate $f_{acc}(n', eval\_set)$
    if $f_{acc}(n', eval\_set) < best\_fn$ then
      Add $n'$ to $irrelevant$
  $beam \leftarrow$ first $k$ $n'$s in sorted($successors, f_{acc}$)
until $f_{acc}$(first($beam$))$> best\_fn$
return first($beam$)
```

Figure 1: Algorithm for learning derived predicates.

```
(def-defined-predicate (abs_2_targetgoal_on ?block1)
   (exists (?block2) (block ?block2)
           (targetgoal_on ?block1 ?block2)))
```

Figure 2: TLPLAN abstracted predicate (Blocksworld)

the formula that is the *body* of the axiom definition. Figure 2 shows the definition of an abstracted predicate for the Blocksworld domain, using the TLPLAN DCK language. R-LTL formulas learned with the best set of derived predicates are translated as described in the previous section, but may now refer to one or more derived predicates.

Experimental Evaluation

We have implemented the ideas presented in previous sections within an extension to TLPLAN that we call LELTL, which stands for LEarning Linear Temporal Logic. In this section we present the experimental evaluation we have performed with LELTL. The aim of this evaluation is two-fold: we want to evaluate the strength of the LELTL planner compared to a state-of-the-art planner; and we want to compare the control rules we learned to DCK that a planning expert has encoded for the domains we consider. As an additional outcome of our evaluation, we will provide a comparison of the TLPLAN approach to current state-of-the-art planners, since previous comparisons were done 8 years ago and the state of the art has advanced significantly in that period. We used the following configurations for our experiments:

LAMA: IPC-2008 winner. Serves as a baseline for comparison (Richter, Helmert, and Westphal 2008). It was configured to stop at the first solution to make comparison with depth-first algorithms fair.

TLPLAN: TLPLAN using hand-coded control rules created by a planning expert.

LeLTL (Basic): TLPLAN using R-LTL formulas containing only domain predicates and additional goal predicates.

LeLTL-DP (Fully automated): TLPLAN with DCK comprising learned derived predicates and R-LTL formulas over domain and derived predicates.

We evaluated these configuration over three benchmarks: *Blocksworld*, *Parking*, and *Gold Miner*. The Blocksworld domain is known to be a good example of a domain requiring extra predicates in order to learn useful DCK, and it's also one of the domains in which TLPLAN had the most significant gain in performance. The Parking and Gold Miner domains were part of the IPC-2008 Learning Track.

The evaluation involved several steps. The first step was to learn the control rules with and without the derived predicates. To do so, training examples were produced from a set of 10 small problems generated by random problem generators freely available to the planning community. A validation set of 20 problems was generated for the LEARN-DERIVEDPRED algorithm, which was run with $k = 3$. Exploratory tests revealed that increasing k beyond 3 did not yield significant changes to the quality of the rules. The rule learner took up to 57 seconds to induce a set of rules.

Once the rules had been learned, they were evaluated in the context of the various planners. For each domain we tested 30 instances. The Parking and Gold Miner problem sets have 30 problems. For Blocksworld we selected the 30 larger typed instances from IPC-2000. To evaluate TLPLAN, we used the Blocksworld DCK reported in (Bacchus and Kabanza 2000), simplified to only handle goals with on and $ontable$ predicates. (i.e., goals are rarely specified with $holding$ or $clear$ predicates). No TLPLAN control rules existed for several of the domains. In these cases, we hand-coded the control rules. Each planner was run with a time bound of 900 seconds.

We appeal to the scoring mechanism used in the IPC-2008 learning track to present our results. With respect to the plan length, for each problem the planner receives N_i^*/N_i points, where N_i^* is the minimum number of actions in any solution returned by a participant for the problem i (i.e., the shortest plan returned by any of the planners), and N_i is the number of actions returned by the planner in question, for the problem i. If the planner did not solve the problem it receives a score of 0 for that problem. The metric for measuring the performance of a given planner in terms of CPU time is computed with the same scheme but replacing N by T, where T is the measure of computation time. Since we used 30 problems for the test sets, any planner can get at most 30 points for each metric. In both cases, high scores are good.

Table 1 shows the number of problems solved by each configuration in the evaluated domains. Table 2 shows the scores for the plan length metric and Table 3 shows the scores for the CPU time metric. We comment on the results for each domain separately.

Domain	LAMA	TLPlan	LeLTL	LeLTL-DP
Blocksworld	17	30	0	28
Parking	29	12	16	30
Gold-miner	29	27	30	27

Table 1: No. problems solved in test sets of 30 problems

Blocksworld: TLPLAN is clearly the best planner in this domain with LELTL-DP a close 2nd with respect to problems solved, topping TLPLAN on quality. LAMA did not solve 13 problems because it did not scale well. LELTL-DP learned the recursive derived predicate depicted in Figure 3. The control formula contains ten clauses, seven of them exploiting this derived predicate. As we explained before, predicate $(abs\_2\_targetgoal\_on\ A)$ encodes that block A needs to be placed somewhere else. Therefore, the semantic of the recursive predicate is that block A is not well placed or it is on another block (B) that recursively follows the same condition (i.e., B or a block beneath is in the wrong place). Interestingly, this actually represents the concept of "bad tower", which was originally defined as the negation of the well-known "good tower" concept in the original TLPLAN control formula for Blocksworld.

Domain	LAMA	TLPlan	LeLTL	LeLTL-DP
Blocksworld	7.21	29.78	0.00	28.00
Parking	28.13	4.24	1.43	18.57
Gold-miner	28.63	13.27	16.96	15.44

Table 2: Plan length scores. High score is good.

Domain	LAMA	TLPlan	LeLTL	LeLTL-DP
Blocksworld	0.24	30.00	0.00	0.16
Parking	10.50	9.49	1.87	25.00
Gold-miner	29.00	1.89	4.10	1.45

Table 3: CPU Time scores obtained by different planners.

Figure 4 shows the planner execution times in the Blocksworld domain. The x-axis is the instance number and the y-axis is the CPU time in logarithmic scale. TLPLAN performs two orders of magnitude faster than LAMA or LELTL-DP. LAMA solved 12 problems faster than LELTL-DP. However, LELTL-DP performance shows that it is more reliable for solving problems. Moreover, LELTL-DP found the best-cost solution in the 28 problems it solved. Both TLPLAN and LELTL-DP scale as a function of the problem size. LAMA's performance seems to be influenced by other properties, such as the problem difficulty.

Parking: The objective of this domain is to arrange a set of cars in a specified parking configuration where cars can be single or serially parked at a set of curbs. In this domain we were unable to manually code an effective control formula for TLPLAN. LAMA performs quite well in this domain and got the top score for the quality metric. LELTL solved 16 problems, performing reasonably well using a 15-clauses control formula without the use of derived predicates. LELTL-DP improves the basic configuration and got the top score for the time metric. It used two new compound predicates and 14 clauses in its control formula. Some of these clauses can be understood intuitively, giving us feedback to our hand-coded formulas. An example of a clause in the Parking domain is presented in Figure 5. The compound predicate specifies that Car S should be placed at Curb U, which is now clear. The clause imposes the restriction of

```
(def-defined-predicate
  (rec_1r_on_abs_2_targetgoal_on ?a)
    (or (abs_2_targetgoal_on ?a)
        (exists (?b) (block ?b)
        (and (on ?a ?b)
             (rec_1r_on_abs_2_targetgoal_on ?b)))))
```

Figure 3: Learned recursive derived predicate (Blocksworld)

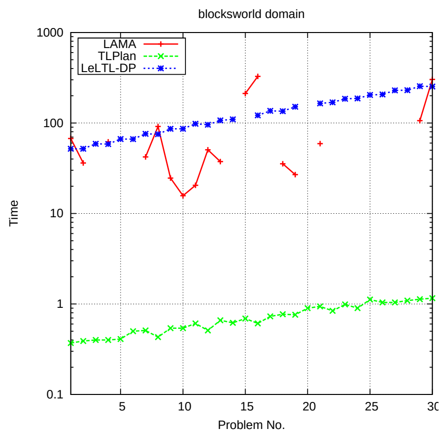

Figure 4: Comparative execution time (Blocksworld)

```
(forall (?s) (car ?s)
 (forall (?t) (car ?t)
  (forall (?u) (curb ?u)
   (implies (and (targetgoal_behind_car ?t ?s)
                 (curb_clear_targetgoal_at_curb_num ?s ?u)
                 (next (behind-car ?s ?t)))
            (false)))))
```

Figure 5: A clause with a learned compound predicate (Parking)

not parking cars in the opposite way whenever the curb at which they should be parked is available in the current state.

Gold-miner: The objective of this domain is to navigate in a grid of cells, destroying rocks with a laser and bombs until reaching a cell containing the gold. All configurations easily found plans, but only LAMA could find plans of good quality. Even using derived predicates, it is difficult to represent concepts for a grid navigation, such as ”good path”. Recursive predicates can represent paths connecting cells recursively, but there is no way to indicate if a path is better than another (i.e., related cells in a link-recursive predicate can say that two cell are connected, but do not express anything about the length). LeLTL-DP learned a compound predicate to represent a cell next to the gold, but it slightly degrades the performance of the basic configuration.

Performance Note: We tried to learn DCK for other domains such as *Satellite*, *Rovers* and *Depots*, but we did not achieve good results. Even optimized hand-tailored DCK does not lead TLPlan to good performance. TLPlan does not perform action grounding as most of state-of-the-art planners do. Thus, it is overwhelmed by continuous variable unifications when progressing control formulas in problems with large numbers of objects. On the other hand, TLPlan is still reported as competitive. This is achieved by precondition control rather than control formulas. I.e., action preconditions are augmented to preclude execution of actions leading to bad states. Consequently, there is no formula to progress and the instantiation of actions is restricted to those leading to a plan.

Discussion and Related Work

While the objective of our work was to learn R-LTL control rules, it is interesting to contrast what we have done to the general use of DCK in TLPlan. TLPlan supports the expression of DCK in a diversity of forms as well as performing various preprocessing procedures. In the work reported here, we did not attempt to replicate TLPlan's *initialization sequence*. This is a procedure that modifies the initial state for domain knowledge enrichment. It includes the use of techniques such as propagating type hierarchy as unary types, and removing useless facts or pre-computing some derived predicate. We also did not attempt to learn derived predicates for use within domain operators for the purpose of precondition control. This would necessitate a modification to the domain representation (changing operator preconditions and effects) to permit the planner to only generate the successors satisfying a control formula, rather than representing the control formula as an explicit rule. The version of TLPlan submitted to IPC-2002 exploited significant precondition control as an alternative to control formulas. The main benefit of this approach is that it eliminates the need to progress LTL formulas, which in some circumstances can be computationally expensive. By way of illustration, consider a Blocksworld problem where all “good towers” are pre-computed by the initialization sequence and the *Unstack* operator is re-written as:

```
(def-adl-operator (stack ?x ?y)
  (pre (?x) (block ?x) (?y) (block ?y)
  (and (holding ?x) (clear ?y)
       (goodtowerbelow ?y)))
  (and (del (holding ?x)) (del (clear ?y))
       (add (clear ?x)) (add (handempty))
       (add (on ?x ?y))
       (add (goodtowerbelow ?x))))
```

While neither learning precondition control nor modifying the domain representation were objectives of our work, precondition control from our control rules could be achieved via regression.

On a different topic, Theorem 1 implies that when LTL is used to express pruning constraints, the formula inside $\square$ need only exploit the $\bigcirc$ modality. This suggests that the two-stage approach presented here could be exploited within a diversity of planning systems without explicit LTL. Such a system would need a representation language able to encode information regarding the current state and changes realized when an action is applied (reaching the next state) and also would need an inference engine that computes axiom rules. An example of such a language is the one used in Prodigy (Veloso et al. 1995), which uses `IF-THEN` control rules for pruning purposes, though they were used combined with preferred and direct selection rules.

Regarding our rule learner, the use of the rule learner ICL was not a limitation for learning the necessary set of LTL. In this work we focused on only using the $\bigcirc$ modality because this was consistent with expert DCK examples we analyzed and because of Theorem 1.

This work is closely related to work that tries to learn generalized policies for guiding search. For example, L2Act (Khardon 1999) induces a set of rules from training examples, but a set of axioms must be given to the system as background knowledge. Some equivalence can be found between axioms in LeLTL-DP and L2Act. LeLTL-DP gets its axioms by deriving them from the domain definition (e.g., $(achieved\_goal\_on\ x\ y)$) or by learning them via the derived predicate learning process. (Martin and Geffner 2004) presented a technique for learning a set of rules for the Blocksworld domain without the definition of background knowledge. In this case DCK was represented in a concept language which was suitable for encoding different object (block) situations. A grammar specified how to combine different concepts in order to produce complex concepts, including the recursive ones needed for Blocksworld. This approach has the limitation that actions are restricted to one parameter, therefore both the domain model and DCK need different representations.

Recently, (Yoon, Fern, and Givan 2008) and (De la Rosa, Jiménez, and Borrajo 2008) achieved more significant results from learning in a variety of domains because they use learned DCK in combination with heuristic search techniques. The former learns a set of rules in taxonomic syntax and the latter learns a set of relational decision trees. These systems attempt to reproduce the search path observed in the training data. Their result is often referred to as a generalized policy, since given a state, problem goals, and in some cases extra features, the policy can return the action to be applied. In contrast, LeLTL is focused on pruning and as such tries to characterize the whole space of possible "correct" paths (positive class), rather than a particular path. Learning-assisted planning is not restricted to systems that learn generalized policies or similar DCK.

Conclusions and Future Work

In this paper we presented an approach to learning domain control knowledge for TLPlan in the form of control rules in a subset of LTL. We achieved this in two stages. 1) We used an off-the-shelf rule learner to induce control formulas in a subset of LTL, designed to prune suboptimal partial plans. Our technique is easily extendable to learning LTL for other purposes, such as recognizing user preferences or temporally extended goals. 2) We also proposed a greedy search technique to discover new concepts in the form of derived predicates that, when used as background knowledge, improve the generation of more complex but useful rules. To do so, we used a general ILP approach that can be applied in other learning-based planners that use a predicate logic representation. The following were some of the key insights and contributions that made our approach work:

- Adaptation of the planning problem representation so that LTL formulas could be learned with a standard first-order logic rule learner.
- Characterization of the state space in terms of two classes of state sequences, allowing the learning component to learn from both positive and negative examples.
- Identification of a core subset of LTL that yielded pruning in TLPlan control rules.
- Development of a technique for automated generation of derived predicates that enhance the feature space in which rules are hand-coded or induced by a learning algorithm.
- Evaluation of the proposed approach, illustrating that with appropriate training examples it is feasible to generate DCK for TLPlan and that with effective DCK, TLPlan remains a state-of-the-art planning system.

In future work we wish to extend our approach in order to integrate the learned DCK into the domain model via precondition control. Experimental results revealed that control formulas represented in CNF adversely affect the scalability of the planner. We further wish to investigate the use of learned derived predicates as background konwledge for learning-based planners such as ROLLER (De la Rosa, Jiménez, and Borrajo 2008).

Acknowledgments This work has been partially supported by the spanish projects TIN2008-06701-C03-03 & CCG08-UC3M/TIC-4141, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) .

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

De la Rosa, T.; Jiménez, S.; and Borrajo, D. 2008. Learning relational decision trees for guiding heuristic planning. In *Proceedings of the 18th ICAPS*.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*. MIT Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

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

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30:119–169.

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

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and F.Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Nedellec, C.; Rouveirol, C.; Bergadano, F.; and Tausend, B. 1996. Declarative bias in ilp. In Raedt, L. D., ed., *Advances in ILP, vol 32 of Frontiers in AI and Applications*. IOS Press.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS-77)*, 46–57.

Raedt, L. D.; Blockeel, H.; Dehaspe, L.; and Laer, W. V. 2001. Three companions for data mining in first order logic. In Dzeroski, S., and Lavrac, N., eds., *Relational Data Mining*. Springer-Verlag. 105–139.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd AAAI Conference (AAAI-08)*, 975–982. AAAI Press.

Silverstein, G., and Pazzani, M. J. 1991. Relational clichés: Constraining induction during relational learning. In *Proceedings of the 18th International Workshop on Machine Learning*, 203–207.

Veloso, M. M.; Carbonell, J.; Pérez, M. A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *JETAI* 7(1):81–120.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *JMLR* 9:683–718.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Magazine* 24:73 – 96.