Three Relational Learning Approaches for Lookahead Heuristic Planning

Tomás de la Rosa, Sergio Jiménez, Rocío García-Durán,
Fernando Fernández, Angel García-Olaya, and **Daniel Borrajo**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
{trosa,rgduran,sjimenez,ffernand,agolaya}@inf.uc3m.es, dborrajo@ia.uc3m.es

Abstract

The computation of relaxed plans provides valuable information about the solution to a planning problem in polynomial time. Moreover, in some domains relaxed plans can be directly taken as part of the solution saving expensive search episodes. Unfortunately, given that the construction of relaxed plans ignores the *delete* effects of actions, relaxed plans may present flaws. In this paper, we propose a novel technique for repairing flaws in relaxed plans, based on domain-specific rules learned from experience. The paper presents and evaluates three different relational learning approaches to automatically induce domain-specific rules from examples. The three learning approaches correspond to the planning systems ROLLER, CABALA and REPLICA that took part in the IPC-2008 learning track.

Introduction

The lookahead strategy for heuristic planning (Vidal 2004) benefits from the fact that relaxed plans usually contain many actions from the final solution plan. Specifically, the lookahead strategy exploits a given relaxed plan to generate a new state, called *lookahead state*, through consecutive application of the actions in the relaxed plan. This *lookahead state* is often closer to the goals and can be used in the search as an extra descendent of the current state.

Unfortunately, relaxed plans present flaws in many planning domains. Particularly, in domains with limited resource sharing, like the *Ferry* domain, and/or in domains with strong subgoals interactions, like the *Blocksworld* domain, relaxed plans lack essential actions, present useless actions and/or mess up the order of some. The lookahead strategy for heuristic planning designed two mechanisms to overcome these flaws: (1) including the *lookahead states* into a *Best First Search* (BFS), so the search can find solutions despite misleading *lookahead states* and (2) defining a set of rules for repairing relaxed plans when actions from the relaxed plan are non-applicable.

Since these rules for repairing relaxed plans are general rules, they miss particularities of domains and hence they are only able to address a limited set of flaws. In this paper we propose a mechanism to automatically learn domain-specific rules for repairing relaxed plans. In particular, we present three relational learning approaches for automatically acquiring these rules in the form of *generalized policies*. These three learning approaches correspond to the planning systems ROLLER, CABALA and REPLICA that took part in the IPC-2008 learning track.

The paper is organized as follows. The second section describes the concept of a relaxed plan and the common flaws of relaxed plans. The third section explains how to use relaxed plans within a lookahead strategy to generate lookahead states. The fourth section explains how to repair flaws in relaxed plans to generate good quality *lookahead states*. The fifth section describe the ROLLER, CABALA and REPLICA systems. The sixth section discusses the results obtained by these three systems at the learning track of IPC-2008. Finally, the last section presents the related work and some conclusions.

The Relaxed Plan

We follow the propositional STRIPS formalism to describe our approaches. We define a planning problem P as the tuple (L, A, s_0, G) being L the set of literals of the problem, A the set of actions, s_0 the set of literals describing the initial state and G the set of literals describing the problem goals. Each action $a \in A$ is a tuple $(pre(a), add(a), del(a))$ where $pre(a)$ represents the action preconditions, $add(a)$ represents the positive effects of the action and $del(a)$ represents the negative effects of the action. The actions *applicable* in a state s_i are those $a \in A$ such that $pre(a) \subseteq s_i$. The state *resulting* from applying the action a in the state s_i is $result(s_i, a) = \{s_i - del(a)\} \cup add(a)$. Under this definition, the solution to a planning problem P is a plan $\mathcal{P}$ consisting of the sequence of actions $(a_1, \ldots, a_n)$ that corresponds to the sequence of state transitions $(s_0, s_1, ..., s_n)$ such that s_i results from executing the action a_i in the state s_{i-1} and s_n is a goal state, i.e., $G \subseteq s_n$.

Computing the Relaxed Plan

The relaxed plan, denoted by $\mathcal{P}^+$, is a solution to the relaxed planning problem P^+; a simplification of the original problem in which the *delete* effects of actions are ignored. The relaxed plan is extracted from the relaxed planning graph, a sequence of *fact layers* and *action layers* (F_0, A_0, …, F_{t-1}, A_{t-1}, F_t). This sequence is built in polynomial time and space, and represents a reachability graph of the applicable

actions in the relaxed problem P^+. The relaxed plan is directly extracted from the planning graph by goal regression.

Flaws in Relaxed Plans

One can find three types of flaws in a relaxed plan:

- *Disorganized* actions. In domains with strong subgoals interactions, e.g., in the *Blocksworld* domain, the order of subgoals is decisive for achieving a solution plan. For example, in the *Sussman's anomaly* problem (initial state $\{ontable(B)$, $clear(B)$, $ontable(A)$, $on(C,A)$, $clear(C)$, $armempty()\}$ and goals $\{on(A,B)$, $on(B,C)\}$), there are two actions in the first action layer of the relaxed planning graph: ($pickup(B)$ and $unstack(C,A)$). These actions are both members of the relaxed and the solution plans. However, action $unstack(C,A)$ must be applied first in order not to undo the goal $on(B,C)$.
- *Missing* actions. In domains that involve strong sharing of limited resources and/or navigation, relaxed plans may lack of actions that appear in solution plans. In these domains, ignoring the *delete* effects of actions produces relaxed plans assuming that limited resources are always available and/or navigation objects are located at diverse places at the same time. For example, in the *Sussman's anomaly* problem, action $putdown(C)$ does not appear in the relaxed plan because the fact $armempty()$ is never deleted in the relaxed plan.
- *Useless* actions. In domains that involve strong sharing of limited resources and/or navigation, relaxed plans frequently contain actions that are absent in solution plans for the same previous reason.

Lookahead Strategy for Heuristic Planning

Relaxed plans can be exploited to synthesize intermediate states that are closer to a goal state than the direct descendants of the current state. These intermediate states, called *lookahead states*, are then added to the list of nodes that can be chosen to be expanded so they can be used within different search algorithms. This strategy slightly increases the branching factor of the search process but generally, as shown by the YAHSP planner at IPC-2004, [1] this strategy improves the performance.

When relaxed plans present flaws, the generated *lookahead states* may not guide the search towards the goals. To relieve this problem one can include *lookahead states* into a *Weighted Best-First Search* (WBFS) algorithm. This algorithm is complete so even when *lookahead states* are misleading, the search process can find a solution. Additionally, one can develop methods for repairing the relaxed plan to generate better *lookahead states*.

Figure 1 shows the algorithm for using lookahead states during the search. This algorithm is based on a modification of a BFS where a *lookahead state* is inserted into the open list in each node expansion. This algorithm is slightly different than the one implemented in YAHSP, where the generation of *lookahead states* was implemented with a recursive function, which allows the process to concatenate several lookahead states (if possible) without inserting these nodes into the *open-list*.

[1] http://ipc04.icaps-conference.org/

Lookahead BFS (s_0,G,GP): *plan*

---

s_0: initial state
G: goals
GP: Generalized Policy

---

open-list $= \{s_0\}$
$n = \emptyset$
while *open-list* $\neq \emptyset$ **and not** solved(n, G) **do**
  n = pop-first-node(*open-list*)
  $RP =$ compute-relaxed-plan(n, G)
  $RRP =$ **repair-relaxed-plan**(RP, n, G, GP)
  $ls =$ get-lookahead-successor(RRP, n)
  $S =$ get-standard-successors(n)
  $S =$ sort(ls,S)
  open-list = merge(S,*open-list*)

Figure 1: A Generic Lookahead BFS algorithm.

Repairing Relaxed Plans: A Domain-specific Approach

The original definition of the lookahead strategy for heuristic planning used heuristic information, based on experimental knowledge, to build the lookahead state for a given state. Particularly, it heuristically searches for one plan (the repaired relaxed plan) containing as many actions as possible from the relaxed plan. Since this technique is domain-independent, it is not able to capture some singularities of the diverse planning domains. For a detailed explanation of the technique refer to (Vidal 2004).

Our approach replaces the standard mechanism for repairing relaxed plans with domain-specific rules in the form of a *generalized policy*. Specifically, our approach computes the *lookahead state* of the current state by iteratively selecting the best action to apply from the relaxed plan following a *generalized policy*. The algorithm for the generation of lookahead states following a *generalized policy* is defined in Figure 2. This algorithm is able to repair *disorganized* and *useless* actions of the relaxed plan but, by the time being, it cannot identify the actions that are *missing*.

A *generalized policy* is a set of rules that map any problem of a given domain, i.e. the diverse combinations of initial state and goals, into the preferred action to execute in order to achieve the problem goals. Consequently, a perfect *generalized policy* is able to solve any problem instance from a given domain. Learning a *generalized policy* for a given planning domain is a three-step process:

1. **Definition of the policy representation**. *Predicate Logic* seems to be the obvious option for representing a *generalized policy* for planning. Nevertheless, other kinds of lan-

repair-relaxed-plan (RP, n, G, GP): RRP

RP: Relaxed-plan
n: current node
GP: Generalized Policy

$RRP = \emptyset$
$ls = n$
while $(A = \text{applicable}(ls, RP)) \neq \emptyset$ **do**
  $action = \text{query-policy}(ls, G, GP)$
  $ls = \text{apply}(action, ls)$
  $RP = RP - \{action\}$
  $RRP = \text{push}\ (action, RRP)$
return RRP

Figure 2: Algorithm for repairing the Relaxed plan with a learned generalized policy.

guages make learning easier for some domains. For example, *Description Logics* (Martin & Geffner 2004) is more suitable for capturing recursive information or *Temporal Logic* (Bacchus & Kabanza 2000) for capturing time-line restrictions over actions in the plan. Additionally, one can define extra predicates to improve learning. These predicates, also known as meta-predicates (Veloso *et al.* 1995), can capture diverse extra information of the planning process such as previously executed actions (Minton 1988), pending goals in the case of backward search planners (Borrajo & Veloso 1996), hierarchical levels in the case of hybrid POP-hierarchical planners (Fernández, Aler, & Borrajo 2005), or deletes of the relaxed plan graph (Yoon, Fern, & Givan 2007).

2. **Generation of learning examples**. Learning examples are extracted from the experience collected solving training problems. Therefore, the quality of the learning examples depend on the quality of the problems used for training. At IPC-2008, the training problems were provided by the organizers. Different approaches consists of incrementally generating larger problems through random walks (Fern *et al.* 2004) or actively generating the problems that can lead to interesting learning (Fuentetaja & Borrajo 2006).
3. **Generalization from the learning examples**. At this step, a relational learning algorithm is used to generalize the decisions made when solving the training problems.

Next, we instantiate this three-step general scheme in the learning systems: ROLLER, CABALA and REPLICA.

Roller: Learning *Helpful Context*-action Policies

Definition of the Policy Representation ROLLER learns *helpful context*-action policies. This kind of *generalized policy* encodes the state using the concept of *helpful actions*, introduced in the FF planner (Hoffmann & Nebel 2001) for pruning the search space. Formally, the set of *helpful actions* of a given state s is defined as:

$$helpful(s) = \{a \in A_0 \mid add(a) \cap G_1 \neq \emptyset\}$$

where A_0 is the first action layer in the relaxed planning graph and G_1 is set set of (sub)goals that need to be achieved in the proposition layer F_1, which are determined during the relaxed plan extraction. Given that each state generates its own particular set of *helpful actions*, ROLLER assumes that the *helpful actions*, together with the remaining goals and the static literals of the planning task, encode a useful context related to each state. Particularly, ROLLER defines the helpful context, $\mathcal{H}(s)$, of a state s as:

$$\mathcal{H}(s) =< helpful(s), target(s), static(s) >$$

where $target(s) = G - s$ describes the set of goals not achieved in state s, and $static(s) = static(s_0)$ is the set of literals that remain true during the search, since they are not changed by any action in the problem.

Generation of Learning Examples ROLLER solves the training problems, first using the Enforced Hill Climbing algorithm (EHC) (Hoffmann & Nebel 2001), and then, refining the found solution with a Depth-first Branch-and-Bound algorithm (DFBnB). DFBnB increasingly generates better solutions according to a given metric (the plan length in this ROLLER version) and a time bound. The final search tree is traversed and all nodes belonging to one of the solutions with the best cost are tagged for generating learning instances for the generalization step. Specifically, for each tagged node, ROLLER generates learning examples consisting of:

- The helpful context of the node, i.e., the current state encoded by its helpful actions plus the set of remaining goals and the static predicates of the planning problem.
- The class of the node, i.e., the action selected in the best cost solution at this search node. The fact that each action may have different arguments (in terms of type and number of arguments) makes the definition of this class concept unfeasible for many classifiers. Accordingly, ROLLER separates examples into two classification steps in order to build generalized policies with off-the-shelf classifiers.
  - Action class. ROLLER labels the best operator to choose in the different helpful contexts of the search.
  - Bindings class. For each operator in the domain, ROLLER labels the best bindings (instantiated arguments) to choose in the different helpful contexts of the search. In this case, the class indicates whether the node is part (selected class) or not (rejected class) of one of the best cost solutions.

Besides, we believe that this two-step decision process is also clearer from the decision-making point of view, and helps users to understand the generated policy better by focusing on either the decision on which action to apply, or which bindings to use given a selected action.

Generalization from Learning Examples ROLLER uses TILDE (Blockeel & De Raedt 1998) for both the action and bindings classification. This tool implements a relational version of the TDIDT algorithm, though we could have used any other off-the-shelf tool for relational classification. Figure 3 shows the actions tree learned for the *Blocksworld* domain. Regarding this tree, the first branch states that when there is a `stack(X,Y)` action in the set of helpful/candidate actions and there is a pending `goal_on(X,Y)`, the action `stack` was selected in the learning examples 69 over 71 times, independently of the rest of helpful/candidate actions.

```
selected(-A,-B)
candidate_stack(A,-C,-D) ?
+--yes: target_goal_on(A,C,D) ?
|     +--yes: [stack] 71.0 [[pick_up:0.0,put_down:2.0,
|     |                     stack:69.0,unstack:0.0]]
|     +--no:  target_goal_on(A,C,-E) ?
|           +--yes: target_goal_on(A,E,D) ?
|           |     +--yes: [put_down] 2.0 [[pick_up:0.0,put_down:2.0,
|           |     |                        stack:0.0,unstack:0.0]]
|           |     +--no:  [stack] 6.0 [[pick_up:0.0,put_down:2.0,
|           |                           stack:4.0,unstack:0.0]]
|           +--no:  [put_down] 33.0 [[pick_up:0.0,put_down:29.0,
|                                     stack:4.0,unstack:0.0]]
+--no:  candidate_unstack(A,-F,-G) ?
     +--yes: candidate_pick_up(A,-H) ?
     |     +--yes: target_goal_on(A,F,H) ?
     |     |     +--yes: [pick_up] 11.0 [[pick_up:7.0,put_down:0.0,
     |     |     |                        stack:0.0,unstack:4.0]]
     |     |     +--no:  target_goal_on(A,-K,G) ?
     |     |           +--yes: [unstack] 22.0
     |     |           |       [[pick_up:0.0,put_down:0.0,
     |     |           |         stack:0.0,unstack:22.0]]
     |     |           +--no:  target_goal_on(A,G,F) ?
     |     |                 +--yes: [pick_up] 2.0
     |     |                 |       [[pick_up:2.0,put_down:0.0,
     |     |                 |         stack:0.0,unstack:0.0]]
     |     |                 +--no:  [unstack] 14.0
     |     |                         [[pick_up:2.0,put_down:0.0,
     |     |                           stack:0.0,unstack:12.0]]
     |     +--no:  [unstack] 36.0 [[pick_up:0.0,put_down:0.0,
     |                             stack:0.0,unstack:36.0]]
     +--no:  [pick_up] 27.0 [[pick_up:27.0,put_down:0.0,
                             stack:0.0,unstack:0.0]]
```

Figure 3: Relational decision tree learned for the action selection in the *Blocksworld* domain.

A more detailed explanation of ROLLER can be found at (De la Rosa, Jiménez, & Borrajo 2008). This work used the *helpful-context* policy for guiding search algorithms instead of for repairing relaxed plans.

Cabala: Learning Generalized Policies as Planning Cases

Definition of the policy representation

CABALA learns the generalized policy in the form of planning cases called *typed sequences*. A *typed sequence* is an abstracted sub-state transition relative to an object type, which partially captures a plan from an object perspective. Formally, a *typed sequence* of a given type is an ordered list of pairs

$$Q = (\mathcal{T}_0, \emptyset), (\mathcal{T}_1, a_1) \ldots, (\mathcal{T}_n, a_{\mathrm{n}})$$

where $\mathcal{T}_i$ is a typed sub-state generated from the state s_i and a_{i} is the applied action in the state state $s_{\mathrm{i}-1}$. A typed sub-state $\mathcal{T}_i$ is a collection of properties. Referred to a particular object o, it represents all properties that object o can define for a particular state s_i. A property, first introduced by the domain analysis in TIM (Fox & Long 1998), is defined as a predicate subscripted with the object position of a literal, e.g., on_1 is a property of object A in the literal $on(A, B)$. In addition, an object sub-state is the set of the state literals in which the object is present. Then, the set of object properties that forms the typed sub-state is computed from the object sub-state. For instance, suppose we have the following state,

$$s_i = \{on(A, B), ontable(B), clear(A), armempty()\}$$

Then, the object sub-state for *block A* would be:

$$s_{A,i} = \{on(A, B), clear(A)\},$$

which is generalized to the typed sub-state of type *block*

$$\mathcal{T}_{block,i} = (on_1\ clear_1).$$

If action $pickup(A)$ is applied in the state s_i, the new object sub-state for block A would be $s_{A,i+1} = \{holding(A)\}$, and consequently, the typed sub-state would become $\mathcal{T}_{block,i+1} = \{holding_1\}$.

Generation of learning examples

CABALA obtains the learning examples from solved problems as follows: For each object instance in the problem, a typed sequence is generated. This process is straightforward, since each step in the sequence is created with the corresponding object sub-state from the solution path. Sequences are grouped in a case base by domain types, so a new case is inserted in the type of object from which it was generated. If the object sub-state does not change when an action is applied, a `no-op` is saved to represent a void action from the object perspective. A merge process verifies that equivalent sequences, only varying in the number of `no-op`, do not repeat in the case base. Figure 4 shows a complete example of a typed sequence generated from a solution plan in the *Blocksworld* domain. The typed sequence has one step more because the typed sub-state of the initial state is included without any action.

Initial State:	(ontable A) (clear A)
	(ontable B) (clear B)
	(ontable C) (clear C)
Goals:	(on A B) (on C A)

Plan	Typed Sequence (Block A)
initial state	$[(clear_1\ ontable_1), \emptyset]$
0: (PICKUP A)	$[(holding_1), pickup]$
1: (STACK A B)	$[(clear_1\ on_1), stack]$
2: (PICKUP C)	$[(clear_1\ on_1)$,`no-op`$]$
3: (STACK C A)	$[(on_1\ on_2), stack]$

Figure 4: A typed sequence example in the *Blocksworld* domain

Generalization of learning examples

CABALA implements a lazy generalization of the learning examples. CABALA stores learning examples in a case base, and when a new problem needs to be solved, the most similar case is retrieved. For each object instance in the new

problem, a typed sequence is retrieved. CABALA implements a simple retrieval scheme that only considers the first step of the sequence referred to the initial state, and the last step of the sequence referring to the goal state. CABALA performs two matches to retrieve a sequence. The first one matches the typed sub-state generated from the goals against the last step of all sequences of the corresponding type. For the second match, the typed sequence generated from the initial state is matched against the first step of the sequences resulting from the first match.

A more detailed explanation of the CABALA retrieval scheme is in (De la Rosa, García-Olaya, & Borrajo 2007). This work used typed sequences to order node evaluations in the Enforced Hill-Climbing algorithm.

Replica

Definition of the policy representation

REPLICA learns Relational Instance-Based Policies (RIBP). A RIBP is defined by a tuple $\pi =< L, R, d >$ where:

- L defines the set of literals used to describe the state space.
- R is a set of tuples, $t_i =< m_i, a_i >$, where m_i is a *meta-state*, i.e., an instant in the search process containing relevant information about the search state. In this case, each m_i is composed of the current state s_i and the pending goals g_i. Otherwise, a_i is the instantiated action applied in s_i.
- d is a distance metric that can compute the relational distance between two different meta-states.

Given a meta-state m, the policy decides the action to execute in m by computing the closest tuple in R and returning its associated action, as shown in Equation 1.

$$\pi(m) = \arg_{a_i} \min_{(<m_i,a_i>\in R)} dist(m, m_i) \quad (1)$$

To compute the distance between two generic meta-states, $d(m_1, m_2)$, we follow a simplification of the RIBL distance metric (Kirsten, Wrobel, & Horváth 2001), which has been adapted to our approach. Let us assume that there are K predicates in a given domain, $p_1, \ldots, p_K$. Then, the distance between the meta-states is a function of the distance between the same predicates in both meta-states:

$$d(m_1, m_2) = \sqrt{\frac{\sum_{k=1}^{K} w_k d_k(m_1, m_2)^2}{\sum_{k=1}^{K} w_k}} \quad (2)$$

Equation 2 includes a weight factor, w_k, for each predicate. These weights modify the contribution of each predicate to the distance metric. And $d_k(m1, m2)$ computes the distance contributed by predicate p_k to the distance metric. For instance, in the *Zenotravel* domain, there are five different predicates ($K = 5$) that define the regular predicates of the domain, plus one referring to the goal $L_{zenotravel} = \{at, in, fuel\_level, next, goal\_at\}$. In each state, different instantiations of the same predicate may coexist. For instance, two literals of predicate at: $(at\ p0\ c0)$ and $(at\ pl0\ c0)$. Then, when computing $d_k(m1, m2)$ we are, in fact, computing the distance between two sets of literals. Equation 3 shows how to compute such distance:

$$d_k(m_1, m_2) = \frac{1}{N} \sum_{i=1}^{N} \min_{y \in Y_k(m_2)} d'_k(Y_k^i(m_1), y) \quad (3)$$

where $Y_k(m_i)$ is the set of literals of predicate y_k in m_i, N is the size of the set $Y_k(m_1)$, $Y_k^i(m_i)$ returns the ith literal from the set $Y_k(m_i)$, and $d'_k(y_k^1, y_k^2)$ is the distance between two literals, y_k^1 and y_k^2 of predicate y_k. Basically, this equation computes, for each literal y in $Y_k(m_1)$, the minimal distance to every literal of predicate y_k in m_2. Then, the distance returns the average of all those distances. Finally, we only need to define the function $d'_k(y_k^1, y_k^2)$. Let us assume the predicate y_k has M arguments. Then,

$$d'_k(y_k^1, y_k^2) = \sqrt{\frac{1}{M} \sum_{l=1}^{M} \delta(y_k^1(l), y_k^2(l))} \quad (4)$$

where $y_k^i(l)$ is the lth argument of literal y_k^i, and $\delta(y_k^1(l), y_k^2(l))$ returns 0 if both values are the same, and 1 if they are different.

Given these definitions, the distance between two instances depends on the similarity between the names of both sets of objects. For instance, the distance between two meta-states that are exactly the same, but with different object names, is judged as maximal distance. To partially avoid this problem, in our approach, the object names of every meta-state are renamed. Each object is renamed by its type name and an appearance index. The first renamed objects are the ones that appear as parameters of the action, followed by the objects that appear in the goals. Finally, we rename the objects appearing in literals of the state. Thus, we try to keep some kind of relevance level of the objects to find a better similarity between two instances.

Generation of learning examples

REPLICA solves the training problems, first using the Enforced Hill Climbing algorithm (EHC) (Hoffmann & Nebel 2001), and then, refining the found solution using a DFBnB algorithm. Once we have the solution plans, we extract the examples to define the policy. The best cost solution in the form $< a_1, \ldots, a_n >$ is used to generate state transitions that can be seen as tuples $t_i =< m_i, a_i >$, where $a_i \in A$ is an instantiated action of the plan, and $m_i \in M$ is a *meta-state*.

Generalization of learning examples

The higher the number of tuples t_i, the longer time will be needed to reuse the policy. After solving the training problems, the set of training tuples t_i is reduced. To reduce the number of tuples, we use the Relational Nearest Prototype Classification algorithm (RNPC) (García-Durán, Fernández, & Borrajo 2008), which is a relational version of the original algorithm ENPC (Fernández & Isasi 2004). There are three main differences with the ENPC algorithm: RNPC uses a

relational representation; the prototypes are extracted by selection as in (Kuncheva & Bezdek 1998); and we can reduce the number of final prototypes by using an optional parameter, the *reduction factor*, which refers to percentage of reduction in the number of instances between the original data set, and the one that RNPC returns. The goal is to obtain a reduced set of prototypes P which generalizes the data set, such that it can predict the class of a new instance faster than using the complete data set, and with equivalent accuracy. The RNPC algorithm is independent of the distance measure and different distance metrics could, in principle, be defined for different domains.

Because the RNPC algorithm is stochastic, it is executed ten times, generating ten different RIBPr. In order to use only one of them, we select the best RIBPr using a validation set of problems (in the case of the competition, the *target* problems). This step will return the RIBPr that solves the most problems (in the validation time-bound), following the metric used in the competition.

A more detailed explanation of REPLICA is in (García-Durán, Fernández, & Borrajo 2008). This work showed how to use this system for guiding a heuristic planner.

Experimental results of the IPC-2008

This section analyzes the results obtained by the systems ROLLER, CABALA and REPLICA at the learning track of IPC-2008.

Domains

The selected set of domains for the learning track of IPC-2008 presented a common feature: the relaxed plan heuristic implemented in the FF planner is not accurate in these domains. Heuristic planning is one of the most extended planning paradigms and FF is one of the most representative planner of this paradigm. Therefore, performing well in domains where this heuristic is misleading is very significant for the planning and learning community. Besides, we want to point out that there still are interesting challenges for this community even in domains where the FF heuristic has been shown to be accurate. For example, heuristic planners present strong scalability limitations in problems with large number of objects. These limitations restricts the application of heuristic planners to real world problems like *Logistics* applications.

Next, we discuss the flaws of relaxed plans in the competition domains:

Gold-Miner In this domain relaxed plans present flaws because there is strong sharing of limited resources. In the relaxed plan the robot can handle a *bomb* and a *laser* at the same time for destroying rocks, which is unfeasible in practice. Besides, in the relaxed plans, the robot can always destroy *rocks* with the *laser* because it ignores the fact that the *laser* destroys the *gold*. Finally, navigation is required. The propagation of the `move` action in the relaxed plan allows the robot to be in many places at the same time causing relaxed plans with useless and missing actions.
The key knowledge for this domain is recognizing that *the robot must use a bomb to destroy rocks bordering the gold*. This key knowledge can be captured as an `IF-THEN` rule that selects the right action to be executed in terms of the current state and the goals.

Matching Blocksworld In this domain relaxed plans present *useless* and *missing* actions because they ignore that actions *put-down* and *stack* make the handled *block* not *solid* when the polarity of the block and the robot arm are different.
The key knowledge for this domain is recognizing that *robot arms should unstack or pick-up blocks of the same polarity*. Nevertheless, in this domain there are solutions that are exceptions to this statement. Specifically, when the robot is handling a *top block*, i.e., a block with no other blocks above at a goal state, the polarity of the robot arm is meaningless. These exceptions include noise in the learning examples and make inductive generalization very complex in this domain.

N-Puzzle In this domain relaxed plans present useless actions because ignoring the *delete* effect of the action `move` (the only action of the domain) makes tiles be at different places at the same time. Traditionally, the control knowledge for this domain has been represented in the form of numeric functions, such as the Manhattan distance, that provide a lower-bound for the solution length. Besides, macro actions can also be useful to find solutions faster because they can group series of movements that typically go together.

Parking Similar to the *Blocksworld*, relaxed plans in this domain assume that all the *cars* and all the *curbs* always stay *clear* which means *missing* actions from the final plan and *useless* actions in the relaxed plan. Besides, there are strong interactions between subgoals thus partial ordered actions must be well ordered to be useful.
The domain definition is flawed given that it was possible to find applicable actions, instantiated from action `move-curb-to-car` which can produce semantically incorrect states. This issue should not directly affect the planners performance because the relaxed plan heuristic recognizes these flawed states as *dead-ends*.

Sokoban Relaxed plans in this domain present useless and missing actions because either the robot and/or the blocks may be at different places at the same time. As it happens in the *N-Puzzle*, control knowledge for this domain has been traditionally expressed as numeric functions that estimate the solution length (Junghanns 1999). And again, macro-actions can find solutions faster in this domain because they can group series of movements that typically go together.

Thoughtful In this domain it is difficult to recognize the key knowledge that should be learned. The large number of predicates (18) and actions (21) and the enormous possibilities of instantiations make the generalization of any kind of knowledge difficult. At this point, we can not give an explanation of the key knowledge needed for this domain.

Problems

For each planning domain there are two distinct distributions over problem instances: the *bootstrap* distribution and the

target distribution. Problems from both distributions can be used for learning and the performance of planners is finally evaluated over problems from the *target* distribution.

Learning problems from the *bootstrap* distribution are fixed. Accordingly, systems that are not able to extract significant knowledge from this problem set will perform poorly. In this sense, participants that explore a whole search tree for generating the learning examples, are more sensitive to this effect, because exploring the whole search tree for several problems from the bootstrap distribution was frequently unfeasible. This problem could be relieved providing participants with a problem generator so they can try to generate the suitable set of learning examples. In this way the competition would also serve as a platform to evaluate active learning techniques.

In addition, the *target* distribution of problems is 'a priori' known. It means that a system can try to learn to plan in this distribution directly without performing deep generalizations on the number of problem objects, goals, ...

Metrics

The IPC-2008 learning track has evaluated the performance of planners analyzing two different dimensions: computation time and plan length. The metrics for measuring these dimensions were the same as those used for the deterministic part. The performance of a given planner in terms of plan length is computed as follows: 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, and N_i is the number of actions returned by the evaluated planner for the problem i. If the planner does not solve the problem it receives 0 points for this problem.

Likewise, the metric for measuring the performance of a given planner in terms of computational time is computed giving a planner T_i^*/T_i points for each problem, where T_i^* is the minimum time used by any planner for solving the problem i, and T_i is the time used by the evaluated planner for solving the problem i. The planner receives 0 point if it does not solve the problem.

Results

Tables 5 and 6 show an extract of the results obtained at the learning part of IPC-2008. Specifically, Table 5 shows the results using the computational time evaluation metric for the described planning and learning systems ROLLER, CABALA and REPLICA together with the overall competition winner PBP (Gerevini, Saetti, & Vallati 2004) and the winner of the best learner award OBTUSE WEDGE. Table 6 shows the results for the plan length evaluation metric.

Performance Notes

The three systems presented in this paper use the SAYPHI planner as the base planner. SAYPHI is a collection of search algorithms implemented in LISP, together with a re-implementation of the relaxed plan heuristic. Having these systems in a common implementation, facilitates testing numerous ideas and helps cleaning-up implementation tricks. However, LISP is not a good choice for implementing a competing planner. Results presented in previous section are fair in the sense that all participants have the same competition rules, including the decision of the implementation language. Obviously, an optimized implementation using C would perform far better than the obtained results (mainly in terms of planning time).

	Roller	**Cabala**	**Replica**	**O. Wedge**	**PbP**
Gold-Miner	6.37	0.0	5.14	9.38	4.42
M. Blocksworld	0.0	0.0	0.0	2.03	25.85
N-Puzzle	0.26	0.0	0.0	29.33	7.10
Parking	2.27	0.0	2.42	28.08	8.96
Sokoban	0.0	0.0	0.0	4.42	10.82
Thoughtful	0.0	0.0	0.0	3.42	23.02

Figure 5: Time metric results.

	Roller	**Cabala**	**Replica**	**O. Wedge**	**PbP**
Gold-Miner	7.85	0.0	7.97	17.46	23.96
M. Blocksworld	1.10	1.89	1.59	5.69	20.22
N-Puzzle	0.45	0.0	0.38	24.50	17.77
Parking	15.61	0.74	14.22	25.54	19.48
Sokoban	0.0	0.0	0.88	15.27	27.24
Thoughtful	0.0	0.0	0.0	6.62	18.01

Figure 6: Number of actions metric results.

Related work

There is previous work using relational learning to correct the flaws of relaxed plans (Yoon, Fern, & Givan 2006). In this work, relational regression is used to learn corrections of the heuristic evaluations computed with the relaxed plan. These corrections consist of a generalization of the differences between the length of relaxed plans and solution plans over a set of training problems. Given that these corrections followed a numerical representation, they were not applicable within a lookahead strategy for constructing partial solution plans avoiding node evaluations. There has also been plenty of previous work on learning either domain-dependent control knowledge or generalized policies. See (Zimmerman & Kambhampati 2003) for a survey.

Conclusions and future work

None of the proposed approaches capture the key knowledge over all the competitions domains. In some domains, the bias of the representation language prevents the systems from learning the key knowledge, in others the poor quality of the learning examples or the limitations of the learning algorithms. Accordingly, it is clear that there is a need for understanding which is the best configuration for capturing correct and complete knowledge for a given domain. This task involves: (1) Deciding which is the best set of features for capturing the key knowledge for a given domain. (2) Deciding which is the best training problems for the diverse kind of planning domains. And (3) deciding which is the most suitable learning algorithm for capturing this knowledge. On the other hand, there is also a need for developing methods for robust planning when the captured control

knowledge is incorrect and incomplete. Thereby the performance of the planner is not penalized very much by the learned knowledge. A good example of this work is the integration of the learned policies within a Best First Search strategy (Yoon, Fern, & Givan 2007).

Acknowledgments

This work has been partially supported by the Spanish MICIIN project TIN2008-06701-C03-03 and the regional CAM-UC3M project CCG08-UC3M/TIC-4141.

References

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

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

Borrajo, D., and Veloso, M. 1996. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning* 11:371–405.

De la Rosa, T.; García-Olaya, A.; and Borrajo, D. 2007. Using cases utility for heuristic planning improvement. In *Proceedings of the 7th International Conference on CBR*, 137–148.

De la Rosa, T.; Jiménez, S.; and Borrajo, D. 2008. Learning relational decision trees for guiding heuristic planning. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 08)*.

Fern, A.; ; Fern, A.; Yoon, S.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *International Conference on Automated Planning and Scheduling*, 191–199.

Fernández, F., and Isasi, P. 2004. Evolutionary design of nearest prototype classifiers. *Journal of Heuristics* 10(4):431–454.

Fernández, S.; Aler, R.; and Borrajo, D. 2005. Machine learning in hybrid hierarchical and partial-order planners for manufacturing domains. *Applied Artificial Intelligence* 19(8):783–809.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research* 9:367–421.

Fuentetaja, R., and Borrajo, D. 2006. Improving control-knowledge acquisition for planning by active learning. In *European Conference on Machine Learning*, 138–149.

García-Durán, R.; Fernández, F.; and Borrajo, D. 2008. Learning and transferring relational instance-based policies. In *Transfer Learning for Complex Task. Technical Report WS-08-13*, 19–24.

García-Durán, R.; Fernández, F.; and Borrajo, D. 2008. Prototypes based relational learning. In *The 13th International Conference on Artificial Intelligence: Methodology, Systems, Applications*.

Gerevini, A.; Saetti, A.; and Vallati, M. 2004. An automatically configurable portfolio-based planner with macro-actions: Pbp. In *International Conference on Automated Planning and Scheduling*, 191–199.

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

Junghanns, A. 1999. *Pushing the Limits: New Developments in Single-Agent Search*. Ph.D. Dissertation, University of Alberta.

Kirsten, M.; Wrobel, S.; and Horváth, T. 2001. *Relational Data Mining*. Springer. chapter Distance Based Approaches to Relational Learning and Clustering, 213–232.

Kuncheva, L., and Bezdek, J. 1998. Nearest prototype classfication: Clustering, genetic algorithms, or random search? *IEEE Transactions on Systems, Man, and Cybernetics*.

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

Minton, S. 1988. *Learning effective search control knowledge: an explanation-based approach*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA, USA. Major Professor-Carbonell, Jaime.

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

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004), Whistler, British Columbia, Canada*, 150–160.

Yoon, S.; Fern, A.; and Givan, R. 2006. Learning heuristic functions from relaxed plans. In *International Conference on Automated Planning and Scheduling*.

Yoon, S.; Fern, A.; and Givan, R. 2007. Using learned policies in heuristic-search planning. In *Proceedings of the 20th IJCAI*.

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