ROLLER: A Lookahead Planner Guided by Relational Decision Trees

Tomás de la Rosa and **Sergio Jiménez**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
trosa@inf.uc3m.es, sjimenez@inf.uc3m.es

Abstract

In this paper we describe the version of the planner ROLLER submitted to the learning track of the International Planning Competition. This version, learns domain dependent general policies with the aim of improving a lookahead strategy for forward search planning. ROLLER performs the policy learning in a two-step classification process with the relational classifier TILDE. At the first step the classifier captures the preferred operator to be applied in the different planning contexts. At the second step the classifier captures the preferred bindings for each operator in the different planning contexts. In this version of ROLLER a planning context is specified by the helpful actions of the current state, the problem goals, the static predicates of the problem and the last action applied.

Introduction

ROLLER is a learning-based planner for solving STRIPS planning problems defined in the typed PDDL language. Specifically, ROLLER starts with a set of training problems from a given domain, learns a domain dependent policy solving these problems and finally, uses the policy for ordering the actions of the relaxed plans in a lookahead planning strategy (Vidal 2004).

As shown in IPC-2004[1], the relaxed plans built for evaluating nodes in heuristic planners can be extended to solution plans. Lookahead planners take advantage of this fact to outperform the scalability of standard heuristic planners. However, the extension of a relaxed plan to a solution plan is not direct in all planning domains. Since relaxed plans are built ignoring the actions deletes, they may lack of some essential actions or they may need an action reordering to be useful. Particularly, these effects happens in domains with a strong subgoal interaction, e.g the blocksworld. With the aim of relieving these undesired effects, ROLLER reorders the actions of the computed relaxed plans according to domain dependent knowledge captured in the form of a general policy.

The paper is organized as follows. The first section describes the basic notions of heuristic planning and the definition of *planning context* used by this version of ROLLER. The second section explains the concept of relational decision trees and how to learn general policies with them. The third section shows how to assist a lookahead planner with a domain dependent general policy. Finally the last section discusses some conclusions.

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

Helpful Context in Heuristic Planning

We follow the propositional STRIPS formalism to describe our approach. We define a planning task P as the tuple (L, A, I, G) with L the set of literals of the task, A the set of actions, where each action $a = (pre(a), add(a), del(a))$. I is the set of literals describing the initial state and G the set of literals describing the goals. Under this definition, solving a planning task implies finding a plan $\mathcal{P}$ as the sequence $(a_1, \ldots, a_n)$ that transforms the initial state into a state in which the goals have been achieved.

FF (Hoffmann & Nebel 2001) heuristic returns the number of actions in the relaxed plan denoted by $\mathcal{P}^+$, which is a solution of the relaxed planning task P^+; a simplification of the original task in which the deletes of actions are ignored. The relaxed plan is extracted from the relaxed planning graph, which is built as a sequence of fact and action layers. This sequence, represented by (F_0, A_0, ..., A_t, F_t), describes a reachability graph of the applicable actions in the relaxed task. For each search state the length of the relaxed plan extracted from this graph is used as the heuristic estimation of the corresponding state.

Moreover, the relaxed plan extraction algorithm marks a set of facts G_i in the planning graph, for each fact layer F_i, as a set of literals that are goals of the relaxed planning task or are preconditions of some actions in a subsequent layer in the graph. According to this, the set of helpful actions are defined as:

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

Traditionally, the helpful actions are used in the search as a pruning technique, because they are considered as the only candidates for being selected during the search. In this work we use helpful actions with a different purpose. Given that each state generates its own particular set of helpful actions, we argue that the helpful actions, together with the remaining goals, the static literals of the planning task and the last applied action, encode a useful context related to each state. The aim of defining a *helpful context* is to determine in later planning episodes, which action from the relaxed plan should be applied. Formally, we define the helpful context, $\mathcal{H}(s)$, of a state s as:

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

where $target(s) \subseteq G$ describes the set of goals not achieved in state s ($target(s) = G - s$), $static(s) = static(I)$ is the set of literals that remain true during the search, because they are not changed by any action in the problem and $last(s) \in A$ represents the action applied for reaching the state s.

Learning General Policies with Decision Trees

ROLLER follows a three-step process for learning the general policies building relational decision trees:

1. Generation of learning examples. ROLLER solves a set of training problems and records the decisions made by the planner when solving them.
2. Actions Classification. ROLLER obtains a classification of the best operator to choose in the different helpful contexts of the search according to the learning examples.
3. Bindings Classification. For each schematic operator in the domain, ROLLER obtains a classification of the best bindings (instantiated arguments) to choose in the different helpful contexts of the search according to the learning examples.

The process for the generation of the learning examples is shared by both classification steps. The learning is separated into two classification steps in order to build general policies with off-the-shelf classifiers. The fact that each planning action may have different arguments (in terms of arguments type and arguments number) makes unfeasible, for many classifiers, the definition of only one learning target concept. 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.

Generation of Learning Examples

With the aim of providing the classifier with learning examples corresponding to good quality planning decisions, 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 of 60 seconds. 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 classifier. Specifically, for each tagged node, ROLLER generates learning examples consisting of:

- the helpful context of the node, i.e., the helpful actions extracted from the node siblings plus the set of remaining goals, the static predicates of the planning problem and the action applied to reach the current state.
- the class of the node. For the *Actions Classification* this class indicates the operator of the node applied action. In the *Bindings Classification* the node siblings with the same operator than the tagged node are also added as learning examples. In the case, the class indicates whether the node is part (selected class) or not (rejected class) of one of the best cost solutions.

Because DFBnB is an exhaustive search algorithm, solving the training problems of large size can take too much time. With the aim of keeping acceptable learning times, ROLLER rejects the training problems that EHC cannot solve in a given time-bound. Besides, ROLLER ensures a set of small learning examples generating extra training problems by splitting the original ones into problems with less goals. Particularly, the number of goals for the extra problems is computed according to this formula.

$$num\_new\_goals = (split\_k * |G|)/h_{\mathrm{FF}}(I) \qquad (1)$$

where $split\_k$ is an input parameter of ROLLER that represent a plan length estimation for solving a problem of one goal. Once ROLLER has solved the training problems it performs an incremental learning from the smallest training problem to the biggest one. The size of a problem is estimated as the length of the plan found by EHC. At each learning step roller takes a given number of training problems, generates the corresponding learning examples and use them to build the general policy. The newly generated policy is compared with the best one generated so far, and if the new policy is better it is considered as the best one for the next learning step. For the comparison between policies, ROLLER uses a set of test problems T and the quality metric defined in the learning track of the competition. According to this metric:

1. Given a test problem $t \in T$, let N* be the minimum number of actions for solving t with any of the two policies.
2. The policy that produce a solution with N actions will get a score of N*/N for the problem. If the policy does not solve the problem get a score of 0.
3. The final policy score is simply the sum of scores received over all the test problems $t \in T$.

The Classification Algorithm

A classical approach to assist decision making consists of gathering a significant set of previous decisions and building a decision tree that generalizes them. The leaves of the resulting tree contain the decision to make and the internal nodes contain the conditions over the examples features that lead to those decisions. The common way to build these trees is following the *Top-Down Induction of Decision Trees* (TDIDT) algorithm (Quinlan 1986). This algorithm builds the decision tree repeatedly splitting the set of learning examples by the conditions that maximize the examples entropy. Traditionally, the learning examples are described in an attribute-value representation. Therefore, the conditions of the decision trees represent tests over the value of a given attribute of the examples. On the contrary, decisions in AI planning are described relationally: a given action is chosen to reach some goals in a given context all described in predicate logic.

Recently, new algorithms for building relational decision trees from examples described as logic facts have been developed. This new relational learning algorithms are similar to the propositional ones except that the conditions in the tree consist of logic queries about relational facts holding in the learning examples. Since the space of potential relational decision trees is normally huge, these relational learning algorithms are biased according to a specification of syntactic restrictions called *language bias*. This specification contains the predicates that can appear on the examples, the target concept, and some learning-specific knowledge as type information, or input and output variables of predicates. In our approach all this language bias is automatically extracted from the PDDL definition of the planning domain.

Along this work we used the tool TILDE (Blockeel & De Raedt 1998) for both the action and bindings classification. This tool implements a relational version of the TDIDT algorithm.

Learning the Actions Tree

The inputs to the actions classification are:

- *The language bias*, that specifies restrictions in the values of arguments of the learning examples. In our case, this bias is automatically extracted from the PDDL domain definition and consists of the predicates for representing the target concept, i.e., the action to select, and the background knowledge, i.e., the helpful context.
- *The learning examples*. They are described by the set of examples of the target concept, and the background knowledge associated to these examples. As previously explained, the background knowledge describes the *helpful-context* of the action selection.

The resulting relational decision tree represents a set of disjoint patterns of action selection that can be used to provide advice to the planner: *the internal nodes* of the tree contain the set of conditions under which the decision can be made. The *leaf nodes* contain the corresponding class; in this case, the decision to be made and the number of examples covered by the pattern. Figure 1 shows the actions tree learned for the satellite domain. Regarding this tree, the first branch states that when there is a `calibrate` action in the set of helpful/candidate actions, it was correctly selected in 44 over 44 times, independently of the rest of helpful/candidate actions. The second branch says that if there is no `calibrate` candidate but there is a `take_image` one, the planner has finally chosen correctly to `take_image` in 110 over 110 times and so on for all the four branches.

```
selected(-A,-B,-C)
candidate_calibrate(A,B,-D,-E,-F) ?
+--yes:[calibrate] 44.0 [[turn_to:0.0,switch_on:0.0,
|                         switch_off:0.0,calibrate:44.0,
|                         take_image:0.0]]
+--no: candidate_take_image(A,B,-G,-H,-I,-J) ?
       +--yes:[take_image] 110.0 [[turn_to:0.0,switch_on:0.0,
       |                            switch_off:0.0,calibrate:0.0,
       |                            take_image:110.0]]
       +--no: candidate_switch_on(A,B,-K,-L) ?
              +--yes:[switch_on] 59.0 [[turn_to:15.0,switch_on:44.0,
              |                         switch_off:0.0,calibrate:0.0,
              |                         take_image:0.0]]
              +--no: [turn_to] 149.0 [[turn_to:149.0,switch_on:0.0,
                                      switch_off:0.0,calibrate:0.0,
                                      take_image:0.0]]
```

Figure 1: Relational decision tree learned for the action selection in the satellite domain.

Learning the Bindings Tree

At this step, a relational decision tree is built for each action in the planning domain. These trees, called bindings trees, indicate the bindings to select for the action in the different planning contexts. The *language bias* for learning a bindings tree is also automatically extracted from the PDDL domain definition. But in this case, the target concept represents the action of the planning domain and its arguments, plus an extra argument indicating whether the set of bindings was selected or rejected by the planner in a given context;

The *learning examples* for learning a bindings tree consist of examples of the target concept, and their associated background knowledge.

Figure 2 shows a bindings tree built for the action `turn-to` from the satellite domain. According to this tree, the first branch says that when there is a sibling node that is a `turn-to` of a satellite C from a location E to a location D with the goal of `goal_pointing` D, another goal is having an image of E and we have to calibrate C in E, the `turn-to` has been selected by the planner in 12 over 12 times.

```
turn_to(-A,-B,-C,-D,-E,-F)
candidate_turn_to(A,B,C,D,E),target_goal_pointing(A,B,C,D)?
+-yes:target_goal_have_image(A,B,E,-G)?
|     +--yes: static_fact_calibration_target(B,-H,D)?
|     |       +--yes:[selected] 12.0 [[selected:12.0,rejected:0.0]]
|     |       +--no: [rejected] 8.0 [[selected:0.0,rejected:8.0]]
|     +--no:  [rejected] 40.0 [[selected:0.0,rejected:40.0]]
+-no: candidate_turn_to(A,B,C,D,E),target_goal_have_image(A,B,E,-I)?
      +--yes: target_goal_have_image(A,B,D,-J) ?
      |       +--yes:[rejected] 48.0 [[selected:0.0,rejected:48.0]]
      |       +--no: [selected] 18.0 [[selected:18.0,rejected:0.0]]
      +--no:  [selected] 222.0 [[selected:220.0,rejected:2.0]]
```

Figure 2: Relational decision tree learned for the bindings selection of the `turn_to` action from the satellite domain.

Lookahead Planning with General Policies

The relaxed plan of given state may contain many actions of the solution plan, so one can use this relaxed plan to generate a new state, called lookahead state, built with the subsequent application of the actions in the relaxed plan until no more actions are applicable. This lookahead state is often closer to goal and it can be used in the search as new descendent of the current state. Figure 3 shows the algorithm implemented for using lookahead states during the search: For each state s to be expanded, ROLLER computes its helpful actions and its lookahead state. The helpful actions are placed at the end of the *open list*, but the lookahead state is placed at the beginning, assuming that this node should be expanded before trying any other in the *open list*.

```
Lookahead BFS Policy (I, G, T): plan

I: initial state
G: goals
T: (Policy) Decision Trees

open-list = {I}
while open-list ≠ ∅ and not solved(s, G) do
    s = pop-first-node(open-list)
    RP = compute-relaxed-plan(s)
    H' = helpful-successors(RP, s)
    L' = get-lookahead-successors(RP, s, T)
    open-list = concatenate(L', open-list, H')
```

Figure 3: A Lookahead BFS algorithm.

ROLLER computes the lookahead state of the current state by iteratively selecting the best action to apply from the relaxed plan following the general policy learned. Specifically, our algorithm for the generation of the lookahead state is defined in Figure 4. The function `applicable` computes the applicable actions of the relaxed plan using the current lookahead state. Functions `solve-op-tree` and `solve-binding-tree` match the context of the lookahead state with the action tree and the bindings tree respectively. The function `instances-of-operator` select a subset of the applicable actions for just considering the actions of the selected operator in by the action tree.

```
get-lookahead-successor (RP, s, T): ls

RP: relaxed-plan
s: current node
T: (Policy) Decision Trees
ls: lookahead successor

ls = s
while (A = applicable(ls, RP)) ≠ ∅ do
    action = solve-op-tree(ls, T)
    C = instances-of-operator(A, action)
    for each c in C do
        ratio(c) = solve-binding-tree(c, ls, T)
    c' = argmax(ratio(c)), c ∈ C
    ls = apply(c', ls)
    RP = RP − {c'}
return ls
```

Figure 4: Algorithm for extracting the lookahead node sorting the Relaxed plan with the learned helpful-context policy.

Roller Parameters

The roller planning system is split in two modules: the learner and the planner. The learner receives a set of training problems, generates the extra problems and learns a general policy in form of relational decision trees. The learner is parametrised by:

- timeout indicates the time-bound for the DFBnB algorithm
- prob-per-iteration the number of training problems for each step of the incremental learning
- The split constant for deciding the size of the extra training problems

The planner receives a domain file, a problem file, the directory with all the trees corresponding to the domain dependent policy learned for this domain and a time-bound. The output of ROLLER is a total-order plan if its able to find a solution plan before a specified time-bound.

Discussion

The benefits of the learning assistance in the ROLLER system are more evident in domains where the current lookahead strategies for heuristic planners do not perform well because there is a strong interaction between the goals. In this kind of domains our approach will improve the quality of the plans found and avoid node evaluation boosting the scalability. However this approach is still weak for domains where the relaxed plan actions are misleading like the driverlog. Further research have to be done in this kind domains.

The version of ROLLER submitted to the first learning track of the International Planning Competition implements a different definition of the helpful context from the one originally explained in (de la Rosa, Jiménez, & Borrajo 2008). Specifically this definition also includes the last executed action to reduce the ambiguity of similar contexts.

Acknowledgments

This work has been partially supported by the Spanish MEC project TIN2005-08945-C06-05 and regional CAM-UC3M project CCG06-UC3M/TIC-0831.

References

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

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

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

Quinlan, J. 1986. Induction of decision trees. *Machine Learning* 1:81–106.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, 150–160.