The PELA architecture: integrating planning and learning to improve execution

Sergio Jiménez, Fernando Fernández and **Daniel Borrajo**
Departamento de Informática.
Universidad Carlos III de Madrid.
Avda. de la Universidad, 30. Leganés (Madrid). Spain.
sjimenez@inf.uc3m.es

Abstract

Building architectures for autonomous rational behavior requires the integration of several AI components, such as planning, learning and execution monitoring. In most cases, the techniques used for planning and learning are tailored to the specific integrated architecture, so they could not be replaced by other equivalent techniques. Also, in order to solve tasks that require lookahead reasoning under uncertainty, these architectures need an accurate domain model to feed the planning component. But the manual definition of these models is a difficult task. In this paper, we propose an architecture that uses off-the-shelf interchangeable planning and learning components to solve tasks that require flexible planning under uncertainty. We show how a relational learning component can be applied to automatically obtain accurate probabilistic action models from executions of plans. These models can be used by any classical planner that handles metric functions, or, alternatively, by any decision theoretic planner. We also show how these components can be integrated to solve tasks continuously, under an online relational learning scheme.

Introduction

There have been many approaches in the literature for building architectures for autonomous rational behavior. Frequently, these architectures include in their components modules for reasoning (among others based on planning, rules and/or search), and learning (among others based on acquiring the domain model and/or some heuristics). In most cases, the reasoning and learning techniques used are extensively tailored to the specific integrated architecture. Therefore, if new more powerful reasoning techniques that could solve the same tasks come up, it requires some programming effort to replace them.

Moreover, these architectures need complex action models representing all the potential execution outcomes for solving tasks that require lookahead reasoning under uncertainty (deliberative instead of reactive reasoning). The manual specification of these models is a difficult task. Therefore, a practical approach has been using classical planning without considering any information about the probability of success of actions in the plans, and re-planning when necessary (Yoon, Fern, & Givan 2007). But, since they do not consider information about the probability of success, they present the drawback of leading to non-robust plans. They might lead to dead-ends or dangerous situations to the executing agent, as well as missing opportunities for better plans.

In this paper, we propose an architecture based on the integration of off-the-shelf interchangeable planning and learning components to overcome the previous shortcomings. Specifically, this architecture follows a cycle of: (1) **planning** with an STRIPS model to generate sequences of actions; (2) **execution** of the plans and tagging of executions as `success`, `failure` or `dead-end`; and (3) **learning** patterns for predicting these outcomes and compiling them into a new action model valid for off-the-shelf planners. The learning component is a relational technique that is used to automatically obtain accurate probabilistic domain models from execution of plans that can be used by any classical planner that handles metric functions, or, alternatively, by any decision theoretic planner. We show how these components can be integrated to solve tasks continuously, under an online relational learning scheme. Experimental results show how both kinds of planners (classical and decision theoretic ones) can benefit from their integration with that relational learning component.

The first section describes in detail the architecture. The second section shows the experimental results obtained using the test bed of the probabilistic track of the International Planning Competition (IPC).[1] Finally, the fourth and fifth sections present related work and conclusions.

Integrating planning, execution and learning

This section describes the PELA (Planning, Execution and Learning Architecture) architecture (Figure 1).

Planning

We start planning with an off-the-shelf classical planner and a STRIPS domain model described in PDDL (Fox & Long 2003) that contains no information about the uncertainty of the world. Once we start executing actions PELA will

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

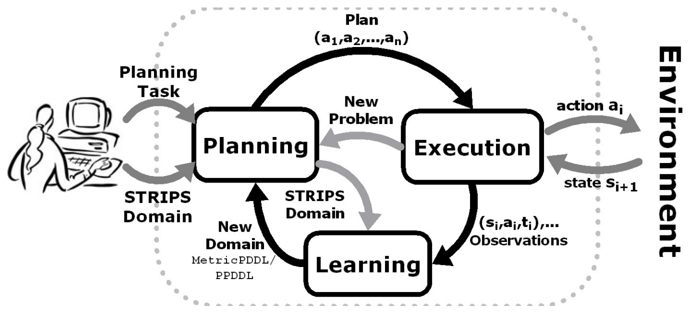

Figure 1: Overview of the PELA Architecture.

learn from its performance and upgrade the domain model to consider probabilistic information about actions outcomes. Therefore, in the following cycles, the planner can use this information to obtain more robust plans.

Execution

Actions are executed in sequence as they appear in the plan. We assume that the outcome of executing actions in the environment is correctly observed. When the execution of an action does not correspond to its STRIPS model we re-plan to solve the planning task in this new scenario. After an action a_i is executed in a state s_i, the execution is tagged with a tag t_i as:

1. `success` when the outcome of executing the action matches its STRIPS model;
2. `failure` when the outcome of executing the action does not match the model, but the task goals can still be achieved by re-planning; or
3. `dead-end` when the outcome of executing the action does not match the domain model, and re-planning can no longer achieve the planning goals.

Learning

Once a set of tagged actions executions has been gathered, we look for patterns of the performance of the actions. A very well-known approach to find patterns of a concept consists of building the smallest decision tree that fits a set of examples of the target concept. Given that off-the-shelf planners represent knowledge relationally, we need to find relational patterns of the actions performance. In our experiments we are using TILDE (Blockeel & Raedt 1998), which implements a relational version of the (TDIDT) algorithm. But, there is nothing that prevents one from using any other off-the-shelf relational learning tool that generates a relational model of the actions performance (in terms of the three classes `success`, `failure` or `dead-end`).

Relational tree learning Relational decision trees contain logic queries about the relational facts holding in them. In PELA, a relational decision tree is learned for every action in the planning domain. When building each tree, the learning component receives two inputs:

- *The language bias* specifying the restrictions in the variables of the predicates to constrain their instantiation. Figure 2 shows the language bias specified for learning the patterns of the performance of the action `move-car(Origin,Destiny)` from the *tireworld* domain.[2] This bias is automatically extracted from the STRIPS domain definition: (1) the types of the target concept are extracted from action definition and (2) the types of the rest of the literals are extracted from the predicate definitions.

```
% The target concept
  type(move-car(example,location,location,class)).
  classes([success,failure,deadend]).
% The domain predicates
  type(vehicle-at(example,location)).
  type(spare-in(example,location)).
  type(road(example,location,location)).
  type(not-flattire(example)).
```

Figure 2: Example of language bias.

- *The knowledge base*, specifying the set of examples of the target concept, and the background knowledge. In PELA, both are extracted from the tagged executions collected in the previous step. Figure 3 shows a piece of the knowledge base for learning the patterns of performance of the action `move-car(Origin,Destiny)`. Particularly, this example captures an execution example with id `e0` that resulted in failure going from location `n00` to `n11`. The background knowledge captures the set of literals holding in the state before the action was executed (literals of the predicates not-flattire, vehicle-at, road, and spare-in). The action execution (example of target concept) is linked with the state literals (background knowledge) through an identifier that represents the execution instance, `e0`. The instantiated action is also augmented with the label that describes the class of the learning example (`success`, `failure` or `dead-end`).

```
% The example
  move-car(e0,n00,n11,failure).
% The background knowledge
  spare-in(e0,n11).   spare-in(e0,n21).   spare-in(e0,n22).
  not-flattire(e0).   vehicle-at(e0,n00). road(e0,n22,n21).
  road(e0,n00,n11).   road(e0,n11,n10).   road(e0,n10,n20).
  road(e0,n10,n21).   road(e0,n11,n22).   road(e0,n21,n20).
  road(e0,n00,n10).
```

Figure 3: Knowledge base corresponding to example `e0`.

Each branch of the learned decision tree will represent a pattern of performance of the corresponding action:

- *the internal nodes* of the branch contain the set of conditions under which the pattern of performance is true.
- *the leaf nodes* contain the corresponding class; in this case, the performance of the action (`success`, `failure` or `dead-end`) and the number of examples covered by the pattern.

Figure 4 shows the decision tree learned for the action `move-car(Origin,Destiny)` using 352 tagged examples. This tree says that when there is a spare wheel at `Destiny`, the action `move-car(Origin,Destiny)`,

[2]In the *tireworld* a car needs to move from one location to another. The car can move between different locations via directional roads. For each of the movements there is a probability of getting a flat tire. A flat tire can be replaced with a spare one; however, some locations do not contain spare tires.

failed 97 over 226 times, while when there is no spare-wheel at `Destiny`, it caused an execution dead-end in 62 over 126 times.

```
move-car(-A,-B,-C,-D)
spare-in(A,C) ?
+--yes: [failure] [[success:97.0,failure:129.0,deadend:0.0]]
+--no:  [deadend] [[success:62.0,failure:0.0,deadend:64.0]]
```

Figure 4: Example of relational decision tree.

Once a tree t_i is built for every action a_i in the STRIPS domain model, the patterns of performance of the actions are compiled into a new action model that allows off-the-shelf planners to (1) avoid execution dead-ends and (2) reason about the robustness of actions. In this work we present and evaluate two different compilations of these patterns:

Compilation to a metric representation Each t_i is compiled together with the STRIPS model of a_i into a new action model with conditional effects:

1. The parameters and preconditions of the STRIPS model remain the same.
2. The effects: each branch b_j of the tree t_i is compiled into a conditional effect ce_{ji}=(when B_j E_{ji}) where:
   (a) B_j=(and $b_{j1}...b_{jn}$), where b_{jk} are the relational tests of the internal nodes of branch b_j (in the tree of Figure 4 there is only one test, referring to spare-in(A,C));
   (b) E_{ji}=(and $\{$effects(a_i) $\cup$ (increase (fragility) f_j)$\}$);
   (c) effects(a_i) are the STRIPS effects of action a_i; and
   (d) (increase (fragility) f_j) is a new literal which increases the cost metric `fragility` in f_j units. The value of f_j is computed as:
   - when b_j does not cover execution examples resulting in dead-ends,
   $$f_j = -log(s/t)$$
   where s refers to the number of execution examples, covered by b_j, resulting in success, and t refers to the total amount of examples that b_j covers;
   - when b_j covers execution examples resulting in dead-ends
   $$f_j = \infty$$
   So, planners will try to avoid these situations because of their high cost.

Intuitively, maximizing a metric indicating the product of the probability of success of actions should allow an off-the-shelf planner to find robust plans. However, off-the-shelf planners do not deal very efficiently with minimizing products of values. Instead, existing planners are better designed to minimize the sum of cost values. Thus, we define the fragility cost metric regarding the following property of logarithms: $log(a) + log(b) = log(a * b)$. Therefore, the minimization of the sum of the negative logarithms of the success probabilities accumulated throughout the plan is equivalent to maximizing the product of action success probabilities.

Figure 5 shows the result of compiling the decision tree of Figure 4. In this case the tree is compiled into two conditional effects. Given that there is only one test on each branch, each new conditional effect will only have one condition (spare-in or not(spare-in)). As it does not cover `dead-end` examples, the first branch increases the fragility cost in $-log(97/(97 + 129))$. The second branch covers `dead-end` examples, so it increases the fragility cost in ∞ (or a sufficiently big number; in this case 999999999).

```
(:action move-car
  :parameters ( ?v1 - location ?v2 - location)
  :precondition (and (vehicle-at ?v1) (road ?v1 ?v2)
                     (not-flattire))
  :effect
   (and
    (when (and (spare-in ?v2))
          (and (increase (fragility) 0.845)
               (vehicle-at ?v2) (not (vehicle-at ?v1))))
    (when (and (not (spare-in ?v2)))
          (and (increase (fragility) 999999999)
               (vehicle-at ?v2) (not (vehicle-at ?v1))))))
```

Figure 5: Compilation into a metric representation.

Compilation to a probabilistic representation In this case the t_i are compiled together with the STRIPS model of a_i into a new action model with probabilistic effects. This compilation is defined as before, except that:

1. E_{ji}=(probabilistic p_j effects(a_i));
2. p_j is the probability value and it is computed as:
   - when b_j does not cover execution examples resulting in dead-ends,
   $$p_j = s/t$$
   where s refers to the number of success examples covered by b_j, and t refers to the total amount of examples that b_j covers;
   - when b_j covers execution examples resulting in dead-ends,
   $$p_j = 0.001$$
   therefore, probabilistic planners try to avoid these situations because of their low probability of success.

Figure 6 shows the result of compiling the decision tree of Figure 4 of the action `move-car(Origin,Destiny)`. In this compilation, the two branches are coded as two probabilistic effects. As the first one does not cover `dead-end` examples, it has a probability of $(97/(97 + 129))$. As the second branch covers `dead-end` examples, it has a probability of 0.001.

Both compilations augment the size of the initial STRIPS model, meaning longer planning times. Specifically, the increase in size is proportional to the number of leaf nodes of the learned trees. One can control the size of the resulting trees by using declarative biases as the amount of tree pruning desired. However, extensive pruning may result in a less robust behavior as leaf nodes would not be so fine grained.

Experimental evaluation

This section presents the experimental results obtained by PELA in two different domains:

- *Triangle Tireworld* (Little & Thibaux 2007). In this version of the tireworld both the origin and the destination locations are at the vertex of an equilateral triangle, the shortest path is never the

```
(:action move-car
  :parameters ( ?v1 - location ?v2 - location)
  :precondition (and (vehicle-at ?v1) (road ?v1 ?v2)
                     (not-flattire))
  :effect
    (and
     (when (and (spare-in ?v2))
           (probabilistic 0.43
                          (and (vehicle-at ?v2)
                               (not (vehicle-at ?v1)))))
     (when (and (not(spare-in ?v2)))
           (probabilistic 0.001
                          (and (vehicle-at ?v2)
                               (not (vehicle-at ?v1))))))
```

Figure 6: Compilation into a probabilistic representation.

most probable one to reach the destination, and there is always a longer trajectory that avoids execution dead-ends. Therefore, an off-the-shelf planner using a STRIPS action model will generally not take the most robust path.

- *Slippery-gripper* (Hanna M. Pasula & Kaelbling 2007). This domain is a version of the four-actions blocksworld (`pick-up`, `put-down`, `stack`, `unstack`) with a nozzle to `paint` the blocks. Painting a block may wet the gripper, which makes it more likely to fail when manipulating the blocks. The gripper can be dried to move blocks safer. Generally, an off-the-shelf classical planner will not choose the `dry` action, because it involves longer plans.

Correctness of the learned models

This experiment evaluates the correctness of the learned models. Particularly, we compared the accuracy of the models obtained by three different strategies for collecting experience. For each strategy we planned for 25 randomly-generated planning problems and executed no more than 50 actions for solving each problem.

1. *FF*: the examples are collected executing the actions proposed by the deterministic planner Metric-FF (Hoffmann 2003);
2. *LPG*: the stochastic planner LPG (Gerevini & Serina 2002) is used instead; and
3. *Random*: the actions to execute are chosen randomly.

For all the strategies, the execution of actions is performed in the simulator provided by the probabilistic track of the IPC and after every 50 executions a model is learned, compiled and evaluated. The learned models are evaluated computing the absolute difference between the probabilities of success and dead-end of each action in the learned model and their true value in the perfect model (the one used in the simulator). Each measure is the average of the differences in a test set of 500 random situations.

Figure 7 shows the results obtained by PELA when learning the models for the actions `move-car(location,location)` and `pickup(block,block)` from the *tireworld* and the *slippery-gripper* domains respectively. These actions are taken as examples because their probabilities of success or dead-ends depend on the current state of the environment. In both cases the convergence values show that the experienced gathered with the random strategy is richer, given that classical planners avoid selecting some actions because they make the plans longer. This effect is stronger in the Metric-FF planner, because it is deterministic. So, as expected, strategies that generate a wider diversity of plans improve the convergence rate, with the penalty of potentially exploring dangerous parts of the problem space.

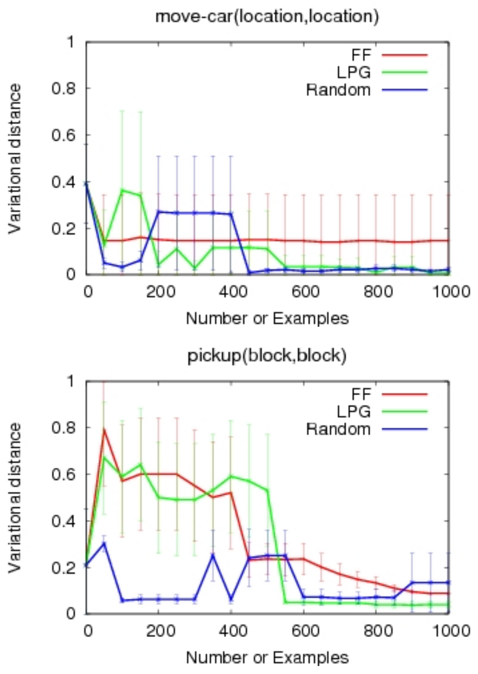

Figure 7: Error of the learned and compiled models.

PELA offline performance

This experiment evaluates the performance of PELA when there are no constraints for collecting the learning examples. In this case, PELA also starts from a PDDL STRIPS version of the domains, and tries to solve test problems using the action model obtained by the *Random* strategy with 500 learning examples. The set of test problems consists of 15 problems for each domain of increasing difficulty. Each problem is attempted 30 times, and we measured the average of the obtained results by four different planning configurations:

1. STRIPS. Metric-FF plans with the PDDL STRIPS action model and re-plans when necessary. This configuration was the unofficial winner of the last probabilistic planning competition. It corresponds to the *classical re-planning* approach, that uses a STRIPS action model without any learning. It serves as the baseline for comparison;
2. PELA + *Metric Compilation*. The PELA architecture using Metric-FF as the planner, and the action model resulting from compiling into a metric (classical) representation;
3. PELA + *Probabilistic Compilation*. PELA using the probabilistic planner GPT (Bonet & Geffner 2005) and the PPDDL action model resulting from the probabilistic compilation;[3]

[3]PPDDL is the probabilistic version of PDDL defined for the probabilistic track of the IPC.

4. *Perfect model*. The probabilistic planner GPT plans with the exact PPDDL probabilistic domain model. This configuration is hypothetical, given that in many planning domains the perfect probabilistic action model is unavailable. Thus, it only serves as a reference to show how far we are from near to optimal solutions with respect to robustness.

In the *triangle tireworld* (Figure 8), PELA solves more problems than *classical planning and re-planning* with both compilations. We are still behind the use of a perfect model. In the *slippery-gripper* there are no execution dead ends so all configurations solve all problems. In this domain, we measure the number of actions needed to solve a problem, where fewer actions means more robust plans. Figure 9, shows that overall PELA solves problems with fewer actions than *classical re-planning*. This is always true for the probabilistic compilation, and true except for three cases for the metric compilation. This might be caused by the fact that the planner we use after learning, Metric-FF, is non-optimal. But, even in this situation, PELA saves a 9% of the accumulated number of actions and 15% of the accumulated time.

In general terms, PELA handles more complex action models, which may increase the planning time, as happens in the tireworld. However, this increase in complexity can be widely worthy in domains where re-planning (like the *slippery-gripper*) or reaching dead-end states (like the *tireworld*) is expensive and a perfect domain model is "a priori" unavailable.

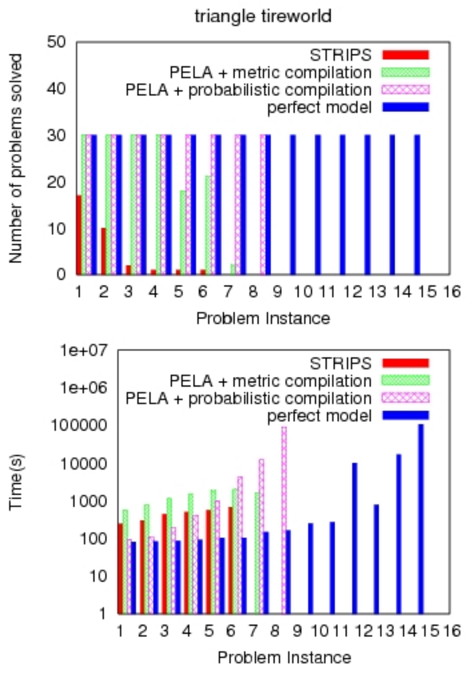

Figure 8: Test problems from triangles of size 3 (problem1) to size 17 (problem 15).

PELA online performance

This experiment evaluates the performance of PELA when the action models are learned, updated and used online.

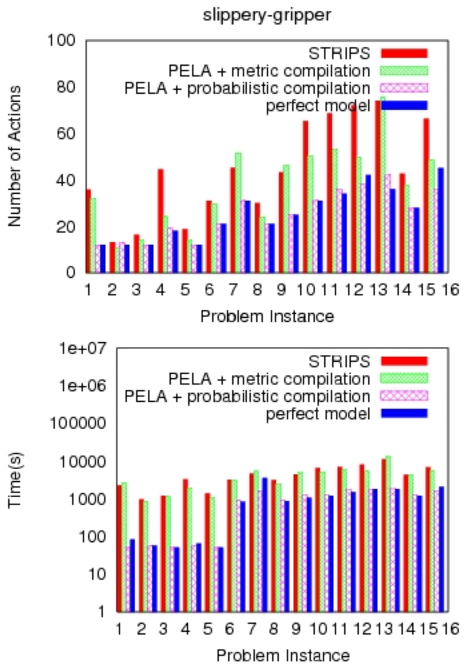

Figure 9: Test problems from 5 blocks (problem1) to 15 blocks (problem 15).

In the online configuration, the *Random* strategy for collecting experience is inappropriate because it ignores the action model and makes PELA not solving the planning tasks. Instead, for the online configuration we used the planner LPG because (1) it is able to solve planning tasks, and (2) according to the learning correctness experiments (Figure 7), it provides a potential richer exploration than planners with deterministic behavior such as FF or GPT. The online configuration of PELA performs a learning and compilation step every 50 executed actions (50 examples). Each learning and compilation step is tested against five problems of medium difficulty that are attempted 30 times each.

Figure 10 shows the experimental results and again, since in the *slippery-gripper* domain, there are no execution dead ends we measure the total number of actions needed to solve the test problems. In both domains the first iterations generate action models that are strongly biased towards the few examples available. These models initially mislead the planning process towards bad quality solutions. But, eventually the quality of the plans improves as more experience is available: more problems solved in the *Triangle Tireworld* and less actions used in the *slippery-gripper*.

Related work

There is extensive prior work on general architectures for reasoning, execution and learning, ranging from execution-oriented, as in robotics applications (Peterson & Cook 2003), to more cognitive-oriented (Rosenbloom, Newell, & Laird 1993). Among them, the more relevant example to our work is ROGUE (Haigh & Veloso 1998) which learned propositional decision trees and used them as control rules

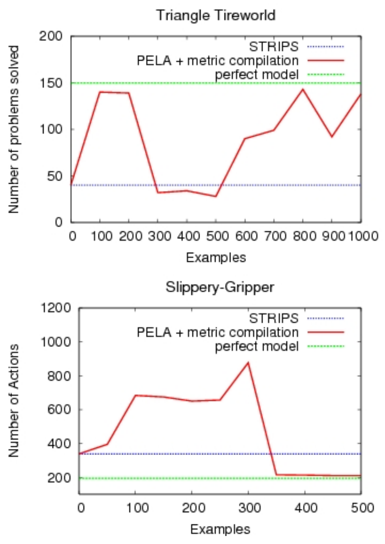

Figure 10: Planning with patterns learned online.

for the PRODIGY architecture (Veloso *et al.* 1995). However, very few architectures have learned relational representations and have used standard relational languages as PDDL or PPDDL for reasoning and learning, so that different planning and/or learning techniques can be directly plugged-in.

Additionally, this work is related to two other research lines also concerned with decision-making, execution and learning:

- Model-free relational reinforcement learning(RRL) (Dzeroski, Raedt, & Blockeel 1998). Unlike our approach, the knowledge learned solving a given task with RRL techniques can not be immediately transferred for similar tasks within the same domain (though currently several RL transfer learning techniques work on this direction). Moreover as our approach delegates the decision making to off-the-shelf planners it allows us to solve more complex problems requiring reasoning about time or resources.
- Model learning of probabilistic actions (Hanna M. Pasula & Kaelbling 2007). These models are more expressive because they capture all possible outcomes of actions. However, they are more expensive to be learned and require specific planning and learning algorithms. Instead, our approach captures uncertainty of the environment using existing standard machine learning techniques and compiles it into standard planning models that can be directly fed into different off-the-shelf planners.

Conclusions

We have presented the PELA architecture which integrates planning, execution and learning in a continuous cycle: PELA plans with a STRIPS model of the environment, and automatically updates it when more execution experience is available. The update is performed by learning: (1) situation-dependent probabilities of the nominal effects; and (2) predictions of execution dead-ends. This update focuses on the nominal effects of actions, because our approach is oriented towards using off-the-shelf classical planners.

Our integration of planning and learning allows PELA to address planning tasks under uncertainty more robustly than the traditional *classical re-planning* approach. Moreover, since PELA is based on interchangeable off-the-shelf planning and learning components, it can profit from the last advances in both fields without modifying the architecture. And even more, the off-the-shelf spirit of the architecture allows PELA to use diverse planning paradigms or change the learning component to acquire other useful planning information, such as the actions duration (Lanchas *et al.* 2007).

The experiments revealed that a random exploration of the environment improves the models accuracy, though this could be unavailable for many domains. In these cases, the use of a planner with stochastic behavior (such as LPG) is preferred given that it provides diversity to the learning examples, as well as less dangerous explorations before capturing the probabilistic knowledge.

Acknowledgments

This work has been partially supported by the Spanish MEC project TIN2005-08945-C06-05 and regional CAM project CCG06-UC3M/TIC-0831, as well as a grant from MEC for sabbaticals.

References

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

Bonet, B., and Geffner, H. 2005. mGPT: A probabilistic planner based on heuristic search. *JAIR* 24:933–944.

Dzeroski, S.; Raedt, L. D.; and Blockeel, H. 1998. Relational reinforcement learning. In *International Workshop on Inductive Logic Programming*, 11–22.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Gerevini, A., and Serina, I. 2002. LPG: a planner based on local search for planning graphs with action costs. *AIPS* 13–22.

Haigh, K. Z., and Veloso, M. M. 1998. Planning, execution and learning in a robotic agent. In *AIPS*, 120–127.

Hanna M. Pasula, L. S. Z., and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence* 29.

Hoffmann, J. 2003. The metric-FF planning system: Translating ignoring delete lists to numerical state variables. *JAIR* 20:291–341.

Lanchas, J.; Jiménez, S.; Fernández, F.; and Borrajo, D. 2007. Learning action durations from executions. In *ICAPS'07 Workshop on AI Planning and Learning*.

Little, I., and Thibaux, S. 2007. Probabilistic planning vs replanning. In *ICAPS'07 Workshop on IPC: Past, Present and Future.*

Peterson, G., and Cook, D. 2003. Incorporating decision-theoretic planning in a robot architecture. *Robotics and Autonomous Systems* 42(2):89–106.

Rosenbloom, P. S.; Newell, A.; and Laird, J. E. 1993. Towards the knowledge level in soar: the role of the architecture in the use of knowledge. 897–933.

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.

Yoon, S.; Fern, A.; and Givan, B. 2007. FF-replan: A baseline for probabilistic planning. In *ICAPS*, 352–360.