Computational Intelligence, Volume 00, Number 000, 2009

Integrating Planning, Execution and Learning to Improve Plan Execution

Sergio Jiménez, Fernando Fernández **and** Daniel Borrajo

Departamento de Informáatica, Universidad Carlos III de Madrid.
Avda. de la Universidad, 30. Leganés (Madrid). Spain

Algorithms for planning under uncertainty require accurate action models that explicitly capture the uncertainty of the environment. Unfortunately, obtaining these models is usually complex. In environments with uncertainty, actions may produce countless outcomes and hence, specifying them and their probability is a hard task. As a consequence, when implementing agents with planning capabilities, practitioners frequently opt for architectures that interleave classical planning and execution monitoring following a *replanning when failure* paradigm. Though this approach is more practical, it may produce fragile plans that need continuous replanning episodes or even worse, that result in execution dead-ends. In this paper, we propose a new architecture to relieve these shortcomings. The architecture is based on the integration of a relational learning component and the traditional planning and execution monitoring components. The new component allows the architecture to learn probabilistic rules of the success of actions from the execution of plans and to automatically upgrade the planning model with these rules. The upgraded models can be used by any classical planner that handles metric functions or, alternatively, by any probabilistic planner. This architecture proposal is designed to integrate off-the-shelf interchangeable planning and learning components so it can profit from the last advances in both fields without modifying the architecture.

Key words: Cognitive architectures, Relational reinforcement learning, Symbolic planning.

1. INTRODUCTION

Symbolic planning algorithms reason about correct and complete action models to synthesize plans that attain a set of goals (Ghallab *et al.*, 2004). Specifying correct and complete action models is an arduous task. This task becomes harder in stochastic environments where actions may produce numerous outcomes with different probabilities. For example, think about simple to code actions like the `unstack` action from the classic planning domain *Blocksworld*. Unstacking the top block from a tower of blocks in a stochastic *Blocksworld* can make the tower collapse in a large variety of ways with different probabilities.

A different approach is to completely relieve humans of the burden of specifying planning action models. In this case machine learning is used to automatically discover the preconditions and effects of the actions (Pasula *et al.*, 2007). Action model learning requires dealing with effectively exploring the environment while learning in an incremental and online manner, similarly to Reinforcement Learning (RL) (Kaelbling *et al.*, 1996). This approach is difficult to follow in symbolic planning domains because random explorations of the world do not normally discover correct and complete models for all the actions. This difficulty is more evident in domains where actions may produce different effects and can lead to execution dead-ends.

As a consequence, an extended approach for implementing planning capabilities in agents consists of defining deterministic action models, obtaining plans with a classical planner, monitoring the execution of these plans and repairing them when necessary (Fox *et al.*, 2006). Though this approach is frequently more practical, it presents two shortcomings. On the one hand, classical planners miss execution trajectories. The classical planning action model only considers the nominal effects of actions. Thus, unexpected outcomes of actions may result in undesired states or even worse in execution dead-ends. On the other hand, classical planners ignore probabilistic reasoning. Classical planners reason about the length/cost/duration of plans without considering the probability of success of the diverse trajectories that reach the goals.

© 2009 The Authors. Journal Compilation © 2009 Wiley Periodicals, Inc.

In this paper we present the Planning, Execution and Learning Architecture (PELA) to overcome shortcomings of traditional integrations of deterministic planning and execution. PELA is based on introducing a learning component together with the planning and execution monitoring components. The learning component allows PELA to generate probabilistic rules about the execution of actions. PELA generates these rules from the execution of plans and compiles them to upgrade its deterministic planning model. The upgraded planning model extends the deterministic model with two kinds of information, state-dependent probabilities of action success and state-dependent predictions of execution dead-ends. PELA exploits the upgraded action models in future planning episodes using off-the-shelf classical or probabilistic planners.

The performance of PELA is evaluated experimentally in probabilistic planning domains. In these domains PELA starts planning with a deterministic model –a STRIPS action model– which encodes a simplification of the dynamics of the environment. PELA automatically upgrades this action model as it learns knowledge about the execution of actions. The upgrade consist of enriching the initial STRIPS action model with estimates of the probability of success of actions and predictions of execution dead-ends. Finally, PELA uses the upgraded models to plan in the probabilistic domains. Results show that the upgraded models allow PELA to obtain more robust plans than a traditional integration of deterministic planning and execution.

The second Section of the paper describes PELA in more detail. It shows PELA's information flow and the functionality of its three components: planning, execution and learning. The third Section explains how the learning component uses a standard relational learning tool to upgrade PELA's action models. The fourth Section shows an empirical evaluation of PELA's performance. The fifth Section describes related work and, finally, the sixth Section discusses some conclusions and future work.

2. THE PLANNING, EXECUTION AND LEARNING ARCHITECTURE

PELA displays its three components in a loop: **(1) Planning** the actions that solve a given problem. Initially, the planning component plans with an off-the-shelf classical planner and a STRIPS-like action model A. This model is described in the standard planning language PDDL (Fox and Long, 2003) and contains no information about the uncertainty of the world. **(2) Execution** of plans and classification of the execution outcomes. PELA executes plans in the environment and labels the actions executions according to their outcomes. **(3) Learning** prediction rules of the action outcomes to upgrade the action model of the planning component. PELA learns these rules from the actions performance and uses them to generate an upgraded action model A' with knowledge about the actions performance in the environment. The upgraded model A' can have two forms: A'_c a PDDL action model for deterministic cost-based planning over a metric that we call *plan fragility* or A'_p an action model for probabilistic planning in PPDDL (Younes *et al.*, 2005), the probabilistic version of PDDL. In the following cycles of PELA, the planning component uses either A'_c or A'_p, depending on the planner we use, to synthesize robust plans. Figure 1 shows the high level view of this integration proposal.

The following subsections describe each component of the integration in more detail.

2.1. Planning

The inputs to the planning component are: a planning problem denoted by P and a domain model denoted by A, in the first planning episode, and by A'_c or A'_p, in the subsequent ones. The planning problem $P = (I, G)$ is defined by I, 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 STRIPS-like action consisting of 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.

Each action $a \in A'_c$ is a tuple $(pre(a), eff(a))$. Again $pre(a)$ represents the action preconditions and $eff(a)$ is a set of conditional effects of the form $eff(a) = (and(when\ c_1(and\ o_1 f_1))(when\ c_2(and$

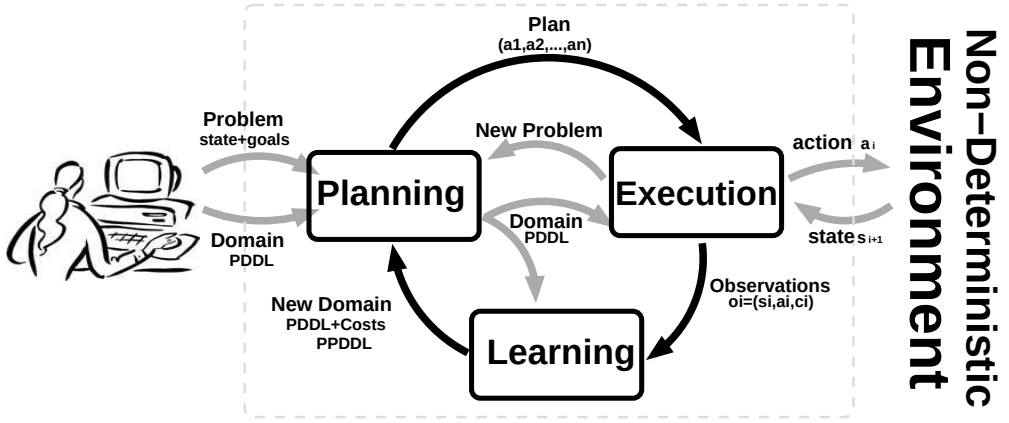

FIGURE 1. Overview of the planning, execution and learning architecture.

$o_2 f_2)) \ldots (when\ c_k (and\ o_k f_k)))$ where, o_i is the outcome of action a and f_i is a fluent that represents the fragility of the outcome under conditions c_i. We will define later the fragility of an action.

Each action $a \in A'_p$ is a tuple $(pre(a), eff(a))$, $pre(a)$ represents the action preconditions and $eff(a) = (probabilistic\ p_1\ o_1\ p_2\ o_2 \ldots p_l\ o_l)$ represents the effects of the action, where o_i is the outcome of a , i.e., a formula over positive and negative effects that occurs with probability p_i.

The planning component synthesizes a plan $p = (a_1, a_2, ..., a_n)$ consisting of a total ordered sequence of instantiated actions. When applying p to I, it would generate a 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., $G \subseteq s_n$. When the planning component reasons with the action model A, it tries to minimize the number of actions in p. When reasoning with action model A'_c, the planning component tries to minimize the value of the *fragility* metric. In the case of planning with A'_p, the planning component tries to maximize the probability of reaching the goals. In addition, the planning component can synthesize a plan p_{random} which contains applicable actions chosen randomly. Though p_{random} does not necessarily achieve the problems goals, it allows PELA to implement different *exploration/exploitation* strategies.

2.2. Execution

The inputs to the execution component are the total ordered plan $p = (a_1, a_2, ..., a_n)$ and the initial STRIPS-like action model A. Both inputs are provided by the planning component. The output of the execution component is the set of observations $O = (o_1, \ldots, o_i, \ldots, o_m)$ collected during the executions of plans.

The execution component executes the plan p one action at a time. For each executed action $a_i \in p$, this component stores an observation $o_i = (s_i, a_i, c_i)$, where:

- s_i is the conjunction of literals representing the facts holding before the action execution;
- a_i is the action executed; and
- c_i is the class of the execution. This class is inferred by the execution component from s_i and s_{i+1} (the conjunction of literals representing the facts holding after executing a_i in s_i) and the STRIPS-like action model of $a_i \in A$. Specifically, the class c_i of an action a_i executed in a state s_i is:
  - `SUCCESS`. When s_{i+1} matches the STRIPS model of a_i defined in A. That is, when it is true that $s_{i+1} = \{s_i / Del(a_i)\} \cup Add(a_i)$.
  - `FAILURE`. When s_{i+1} does not match the STRIPS model of a_i defined in A, but the problem goals can still be reached from s_{i+1}; i.e., the planning component can synthesize a plan that theoretically reaches the goals from s_{i+1}.
  - `DEAD-END`. When s_{i+1} does not match the STRIPS domain model of a_i defined in A, and the

problem goals cannot be reached from s_{i+1}; i.e., the planning component cannot synthesize a plan that theoretically reaches the goals from s_{i+1}.

Figure 2 shows three execution episodes of the action `move-car(location,location)` from the *Tireworld*. 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 movement there is a probability of getting a flat tire and flat tires can be replaced with spare ones. Unfortunately, some locations do not contain spare tires which results in execution dead-ends. In the example, the execution of action `move-car(A,B)` could result in `SUCCESS` when the car does not get a flat tire or in `DEAD-END` when the car gets a flat tire, because at location B there is no possibility of replacing flat tires. On the other hand, the execution of action `move-car(A,D)` is safer. The reason is that `move-car(A,D)` can only result in either `SUCCESS` or `FAILURE`, because at location D there is a spare tire for replacing flat tires.

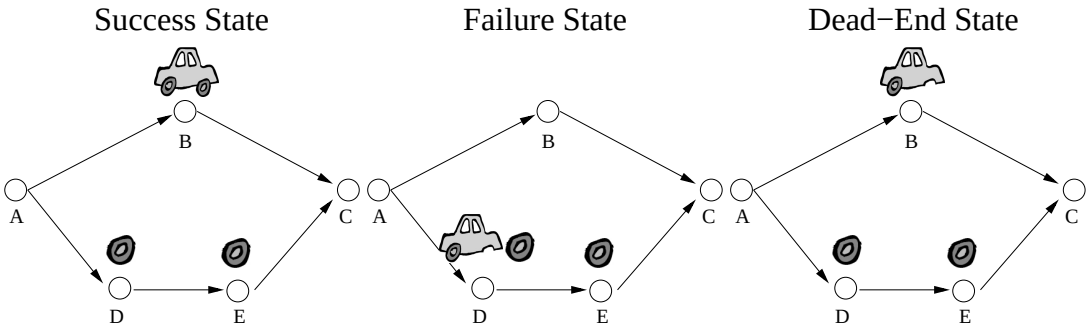

FIGURE 2. Execution episodes for the `move-car` action in the *Tireworld*.

The algorithm for the execution component of PELA is shown in Figure 3. When the execution of an action a_i is classified as `SUCCESS`, the execution component continues executing the next action in the plan a_{i+1} until there are no more actions in the plan. When the execution of an action does not match its STRIPS model, then the planning component tries to replan and provide a new plan for solving the planning problem in this new scenario. In case replanning is possible, the execution is classified as a `FAILURE` and the execution component continues executing the new plan. In case replanning is impossible, the execution is classified as a `DEAD-END` and the execution terminates.

2.3. Learning

The learning component generates rules that generalize the observed performance of actions. These rules capture the conditions referred to the state of the environment that define each probability of execution success, failure and dead-end of the domain actions.

Then, it compiles these rules and the STRIPS-like action model A into an upgraded action model A' with knowledge about the performance of actions in the environment. The inputs to the learning component are the set of observations O collected by the execution component and the original action model A. The output is the upgraded action model A'_c defined in PDDL for deterministic cost-based planning or A'_p defined in PPDDL for probabilistic planning.

Learning in PELA assumes actions have nominal effects. This is influenced by the kind of actions that traditionally appear in planning tasks that typically present a unique *good* outcome.

The implementation of the learning component is described throughout next section.

3. EXPLOITATION OF EXECUTION EXPERIENCE

This section explains how PELA learns rules about the actions performance using a standard relational classifier and how PELA compiles these rules to improve the robustness of the synthesized plans. In this paper PELA focused on learning rules about the success of actions. However, the off-

Function Execution (InitialState, Plan, Domain):Observations

InitialState: initial state
Plan: list of actions (a1, a2 , ..., an)
Domain: Strips action model
Observations: Collection of Observations

$Observations = \emptyset$
$state = InitialState$
While $Plan$ is not $\emptyset$ do
    $a_i = Pop(Plan)$
    $newstate = execute(state, a_i)$
    if $match(state, newstate, a_i, Domain)$
        $Observations = collectObservation(Observations, state, a_i,$`SUCCESS`$)$
    else
        $Plan = replan(newstate)$
        If $Plan$ is $\emptyset$
            $Observations = collectObservation(Observations, state, a_i,$`DEAD-END`$)$
        else
            $Observations = collectObservation(Observations, state, a_i,$`FAILURE`$)$
    $state = newstate$
Return $Observations$;

Figure 3. Execution algorithm for domains with dead-ends.

the-shelf spirit of the architecture allows PELA to acquire other useful execution information, such as the actions durations (Lanchas *et al.*, 2007).

3.1. Learning rules about the actions performance

For each action $a \in A$, PELA learns a model of the performance of a in terms of these three classes: `SUCCESS`, `FAILURE` and `DEAD-END`. A well-known approach for multiclass classification consists of finding the smallest decision tree that fits a given data set. The common way to find these decision trees is following a *Top-Down Induction of Decision Trees* (TDIDT) algorithm (Quinlan, 1986). This approach builds the decision tree by splitting the learning examples according to the values of a selected attribute that minimize a measure of variance along the prediction variable. The leaves of the learned trees are labelled by a class value that fits the examples satisfying the conditions along the path from the root of the tree to those leaves. Relational decision trees (Blockeel and Raedt, 1998) are the first-order logic upgrade of the classical decision trees. Unlike the classical ones, relational trees work with examples described in a relational language such as predicate logic. This means that each example is not described by a single feature vector but by a set of logic facts. Thus, the nodes of the tree do not contain tests about the examples attributes, but logic queries about the facts holding in the examples.

For each action $a \in A$, PELA learns a relational decision tree t_a. Each branch of the learned decision tree t_a represents a prediction rule of the performance of the corresponding action a:

- The *internal nodes* of the branch represent the set of conditions under which the rule of performance is true.
- The *leaf nodes* contain the corresponding class; in this case, the action performance (`SUCCESS`, `FAILURE` or `DEAD-END`) and the number of examples covered by the pattern.

Figure 4 shows the decision tree learned by PELA for action `move-car(Origin,Destiny)` using 352 tagged examples. According to this tree, when there is a spare tire at `Destiny`, the action failed 97 over 226 times, while when there is no spare tire at `Destiny`, it caused an execution dead-end in 64 over 126 times.

To build a decision tree t_a for an action a, the learning component receives two inputs:

- *The language bias* specifying the restrictions in the parameters of the predicates to constrain their instantiation. This bias is automatically extracted from the STRIPS domain definition: (1) the types of the target concept are extracted from the action definition and (2) the types of the

```
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. Relational decision tree for `move-car(Origin,Destiny)`.

rest of literals are extracted from the predicates definition. Predicates are extended with an extra parameter called *example* that indicates the identifier of the observation. Besides, the parameters list of actions is also augmented with a label that describes the class of the learning example (`SUCCESS`, `FAILURE` or `DEAD-END`). Figure 5 shows the language bias specified for learning the model of performance of action `move-car(Origin,Destiny)` from the *Tireworld*.

```
% 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 5. Language bias for the *Tireworld*.

- *The knowledge base*, specifying the set of examples of the target concept, and the background knowledge. In PELA, both are automatically extracted from the observations collected by the execution component. The action execution (example of target concept) is linked with the state literals (background knowledge) through the identifier of the execution observation. Figure 6 shows a piece of the knowledge base for learning the patterns of performance of the action `move-car(Origin,Destiny)`. Particularly, this example captures the execution examples with identifier `o1`, `o2` and `o3` that resulted in success, failure and dead-end respectively, corresponding to the action executions of Figure 2.

PELA uses TILDE[1] for the tree learning but there is nothing that prevents from using any other relational decision tree learning tool.

3.2. Upgrade of the Action Model with Learned Rules

PELA compiles the STRIPS-like action model A and the learned trees into an upgraded action model. PELA implements two different upgrades of the action model: (1) *compilation to a metric representation*; and (2) *compilation to a probabilistic representation*. Next, there is a detailed description of the two compilations.

3.2.1. *Compilation to a Metric Representation.* **In this compilation, PELA transforms each action $a \in A$ and its corresponding learned tree t_a into a new action $a' \in A'_c$ which contains a metric of the fragility of a. The aim of the fragility metric is making PELA generate more robust plans that solve more problems in stochastic domains. This aim includes two tasks, avoiding execution dead-ends and avoiding replanning episodes, i.e., maximizing the probability of success of plans. Accordingly, the fragility metric expresses two types of information: it assigns infinite cost to situations that can cause**

[1]TILDE (Blockeel and Raedt, 1998) is a relational implementation of the *Top-Down Induction of Decision Trees* (TDIDT) algorithm (Quinlan, 1986).

```
% Example o1
 move-car(o1,a,b,success).
% Background knowledge
 vehicle-at(o1,a). not-flattire(o1).
 spare-in(o1,d). spare-in(o1,e).
 road(o1,a,b). road(o1,a,d). road(o1,b,c).
 road(o1,d,e). road(o1,e,c).

% Example o2
 move-car(o2,a,c,failure).
% Background knowledge
 vehicle-at(o2,a).
 spare-in(o2,d). spare-in(o2,e).
 road(o2,a,b). road(o2,a,d). road(o2,b,c).
 road(o2,d,e). road(o2,e,c).

% Example o3
 move-car(o3,a,b,deadend).
% Background knowledge
 vehicle-at(o3,a).
 spare-in(o3,d). spare-in(o3,e).
 road(o3,a,b). road(o3,a,d). road(o3,b,c).
 road(o3,d,e). road(o3,e,c).
```

FIGURE 6. Knowledge base after the executions of Figure 2.

execution dead-ends and it assigns a cost indicating the success probability of actions when they are not predicted to cause execution dead-ends.

Given $prob(a_i)$ as the probability of success of action a_i, the probability of success of a total ordered plan $p = (a_1, a_2, ..., a_n)$ can be defined as:

$$prob(p) = \prod_{i=1}^{n} prob(a_i).$$

Intuitively, taking the maximization of $prob(p)$ as a planning metric should guide planners to find robust solutions. However, planners do not efficiently deal with a product maximization. Thus, despite this metric is theoretically correct, experimentally it leads to poor results in terms of solutions quality and computational time. Instead, existing planners are better designed to minimize a sum of values (like length/cost/duration of plans). This compilation defines a metric indicating not a product maximization but a sum minimization, so off-the-shelf planners can use it to find robust plans. The definition of this metric is based on the following property of logarithms:

$$log(\prod_{i} x_i) = \sum_{i} log(x_i)$$

Specifically, we transform the probability of success of a given action into an action cost called *fragility*. The *fragility* associated to a given action a_i is computed as:

$$fragility(a_i) = -log(prob(a_i))$$

The fragility associated to a total ordered plan is computed as:

$$fragility(p) = \sum_{i=1}^{n} fragility(a_i).$$

Note that a minus sign is introduced in the *fragility* definition to transform the maximization into a minimization. In this way, the maximization of the product of success probabilities along a plan is transformed into a minimization of the sum of the fragility costs.

Formally, the compilation is carried out as follows. Each action $a \in A$ and its corresponding learned tree t_a are compiled into a new action $a' \in A'_c$ where:

(1) The parameters of a' are the parameters of a.
(2) The preconditions of a' are the preconditions of a.
(3) The effects of a' are computed as follows. Each branch b_j of the tree t_a is compiled into a conditional effect ce_j of the form ce_j=(when B_j E_j) where:

   (a) B_j=(and $b_{j1}...b_{jm}$), 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_j=(and {effects(a) ∪ (increase (fragility) f_j)});
   (c) effects(a) are the STRIPS effects of action a; and
   (d) `(increase (fragility)` f_j`)` is a new literal which increases the `fragility` metric in f_j units. The value of f_j is computed as:

   - when b_j does not cover execution examples resulting in dead-ends,

$$f_j = -log(\frac{1+s}{2+n})$$

     where s refers to the number of execution examples covered by b_j resulting in success, and n refers to the total number of examples that b_j covers. Regarding the Laplace's *rule of succession* we add 1 to the success examples and 2 to the total number of examples. Therefore, we assign a probability of success of 0.5 to actions without observed executions;

   - when b_j covers execution examples resulting in dead-ends.

$$f_j = \infty$$

     PELA **considers as execution dead-ends states where goals are unreachable from them.** PELA **focuses on capturing undesired features of the states that cause dead-ends to include them in the action model. For example, in the** *triangle tireworld* **moving to locations that do not contain spare-wheels.** PELA **assigns an infinite fragility to the selection of actions in these undesired situations so the generated plans avoid them because of their high cost.** PELA **does not capture undesired features of goals because the PDDL and PPDDL languages do not allow to include goals information in the action models.**

Figure 7 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 $-log(\frac{97+1}{97+129+2})$. The second branch covers `dead-end` examples, so it increases the fragility cost in ∞ (or a sufficiently big number in practice; 999999999 in the example).

```
(: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 7. Compilation into a metric representation.

3.2.2. *Compilation to a probabilistic representation.* In this case, PELA compiles each action $a \in A$ and its corresponding learned tree t_a into a new probabilistic action $a' \in A'_p$ where:

(1) The parameters of a' are the parameters of a.
(2) The preconditions of a' are the preconditions of a.
(3) Each branch b_j of the learned tree t_a is compiled into a probabilistic effect pe_j=(when B_j E_j) where:
   (a) B_j=(and $b_{j1}...b_{jm}$), where b_{jk} are the relational tests of the internal nodes of branch b_j;
   (b) E_j=(probabilistic p_j effects(a));
   (c) effects(a) are the STRIPS effects of action a;
   (d) p_j is the probability value and it is computed as:
      - when b_j does not cover execution examples resulting in dead-ends,
$$p_j = \frac{1+s}{2+n}$$
        where s refers to the number of success examples covered by b_j, and n refers to the total number of examples that b_j covers. The probability of success is also computed following the Laplace's *rule of succession* to assign a probability of 0.5 to actions without observed executions;
      - when b_j covers execution examples resulting in dead-ends,
$$p_j = 0.001$$
        Again, PELA **does not only try to optimize the probability of success of actions but it also tries to avoid execution dead-ends. Probabilistic planners will try to avoid selecting actions in states that can cause execution dead-ends because of their low success probability.**

Figure 8 shows the result of compiling the decision tree of Figure 4 corresponding to the action `move-car(Origin,Destiny)`. In this compilation, the two branches are coded as two probabilistic effects. The first one does not cover `dead-end` examples so it has a probability of $\frac{97+1}{97+129+2}$. The second branch covers `dead-end` examples so it has a probability of 0.001.

```
(: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 8. Compilation into a probabilistic representation.

4. EVALUATION

To evaluate PELA we use the methodology defined at the probabilistic track of the International Planning Competition (IPC). This methodology consists of:

- A common representation language. PPDDL was defined as the standard input language for probabilistic planners.

- A simulator of stochastic environments. *MDPsim*[2] was developed to simulate the execution of actions in stochastic environments. Planners communicate with *MDPsim* in a high level communication protocol that follows the client-server paradigm. This protocol is based on the exchange of messages through TCP sockets. Given a planning problem, the planner sends actions to *MDPsim*, *MDPsim* executes these actions according to a given probabilistic action model described in PPDDL and sends back the resulting states.
- **A performance measure. At IPC probabilistic planners are evaluated regarding these metrics:**

(1) Number of problems solved. The more problems a planner solves, the better the planner performs. This is the main criterion to evaluate the performance of PELA **in our experiments. In stochastic domains, planners need to avoid executions dead-ends and to reduce the number of replanning episodes to succeed reaching the problem goals in the given time bound.**
(2) Time invested to solve a problem. The less time a planner needs, the better the planner performs. Our experiments also report this measure to distinguish the performance of planners when planners solve the same number of problems.
(3) Number of actions to solve a problem. The less actions a planner needs, the better the planner performs. Though this metric is computed at IPC, we do not use it to evaluate the performance of PELA**. Comparing probabilistic planners with this metric might be confusing. In some cases robust plans are the shortest ones. In other cases longer plans are the most robust because they avoid execution dead-ends or because they have a higher probability of success. Like the** *time-invested* **metric, the number of actions could also be used to distinguish the performance of planners when they solve the same number of problems.**

In our experiments both PELA and *MDPSim* share the same problem descriptions. However, they have different action models. On the one hand, PELA tries to solve the problems starting with a STRIPS-like description of the environment A which ignores the probability of success of actions. On the other hand, *MDPSim* simulates the execution of actions according to a PPDDL model of the environment $A_{perfect}$. As execution experience is available PELA will learn new action models A'_c or A'_p that approach the performance of PELA to the performance of planning with the perfect model of the environment $A_{perfect}$.

4.1. The Domains

We evaluate PELA over a set of *probabilistically interesting* domains. A given planning domain is considered *probabilistically interesting* (Little and Thiébaux, 2007) when the shortest solutions to the domain problems do not correspond to the solutions with the highest probability of success. Given that classical planners prefer short plans, a classical replanning approach fails more often than a probabilistic planner. These failures mean extra replanning episodes which usually involve more computation time. And/or when the shortest solutions to the domain problems present execution dead-ends. Given that classical planners prefer short plans, a classical replanning approach solves less problems than a probabilistic planner.

Probabilistically interesting domains can be generated from classical domains by increasing their *robustness diversity*, i.e., the number of solution plans with different probability of success. In this paper we propose to artificially increase the *robustness diversity* of a classical planning domain following any of the proposed methods:

- Cloning actions. Cloned actions of diverse robustness are added to the domain model. Particularly, a cloned action a' keeps the same parameters and preconditions of the original action a but presents (1) different probability of success and/or (2) a certain probability of producing execution dead-ends. Given that classical planners handle STRIPS-like action models, they do not reason about

[2] *MDPsim* can be freely downloaded at http://icaps-conference.org/

the probability of success of actions and they arbitrarily choose among cloned actions ignoring their robustness.

- Adding fragile macro-actions. A macro-action a' with (1) low probability of success and/or (2) with a certain probability of producing execution dead-ends is added to the domain. Given that classical planners ignore robustness and prefer short plans, they tend to select the fragile macro-actions though they are less likely to succeed.
- Transforming action preconditions into success preferences. Given an action with the set of preconditions p and effects e, a precondition $p_i \in p$ is removed and transformed into a condition for e that (1) increases the probability of success and/or (2) avoids execution dead-ends. For example, *when* p_i (*probability* 0.9 (*and* $e_1, \ldots, e_i, \ldots, e_n$)) and *when* $\neg p_i$ (*probability* 0.1 (*and* $e_1, \ldots, e_i, \ldots, e_n$)). Again, classical planners prefer short plans, so they skip the satisfaction of these actions conditions though they produce plans more likely to fail.

We test the performance of PELA over the following set of *probabilistically interesting* domains:

Blocksworld. This domain is the version of the classical four-actions *Blocksworld* introduced at the probabilistic track of IPC-2006. This version extends the original domain with three new actions that manipulate towers of blocks at once. Generally, off-the-shelf classical planners prefer manipulating towers because it involves shorter plans. However, these new actions present high probability of failing and causing no effects.

Slippery-gripper (Pasula *et al.*, 2007). This domain is a version of the four-actions *Blocksworld* which includes a nozzle to `paint` the blocks. Painting a block may wet the gripper, which makes it more likely to fail when manipulating blocks. The gripper can be dried to move blocks safer. However, off-the-shelf classical planners will generally skip the `dry` action, because it involves longer plans.

Rovers. This domain is a probabilistic version of the IPC-2002 *Rovers* domain specifically defined for the evaluation of PELA. The original IPC-2002 domain was inspired by the planetary rovers problem. This domain requires that a collection of rovers equipped with different, but possibly overlapping, sets of equipment, navigate a planet surface, find samples and communicate them back to a lander. In this new version, the navigation of rovers between two waypoints can fail. Navigation fails more often when waypoints are not visible and even more when waypoints are not marked as traversable. Off-the-shelf classical planners ignore that navigation may fail at certain waypoints, so their plans fail more often.

OpenStacks. This domain is a probabilistic version of the IPC-2006 *OpenStacks* domain. The original IPC-2006 domain is based on the *minimum maximum simultaneous open stacks* combinatorial optimization problem. In this problem a manufacturer has a number of orders. Each order requires a given combination of different products and the manufacturer can only make one product at a time. Additionally, the total quantity required for each product is made at the same time (changing from making one product to making another requires a production stop). From the time that the first product included in an order is made to the time that all products included in the order have been made, the order is said to be *open* and during this time it requires a *stack* (a temporary storage space). The problem is to plan the production of a set of orders so that the maximum number of *stacks* simultaneously used, or equivalently, the number of orders that are in simultaneous production, is minimized. This new version, specifically defined for the evaluation of PELA, extends the original one with three cloned *setup-machine* actions and with one macro-action *setup-machine-make-product* that may produce execution dead-ends. Off-the-shelf classical planners ignore the robustness of the cloned *setup-machine* actions. Besides, they tend to use the *setup-machine-make-product* macro-action because it produces shorter plans.

Triangle Tireworld (Little and Thiébaux, 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 most probable one to reach the destination, and there is always a trajectory where execution dead-ends can be avoided. Therefore, an off-the-shelf planner using a STRIPS action model will generally not take the most robust path.

Satellite. This domain is a probabilistic version of the IPC-2002 domain defined for the evaluation of PELA. The original domain comes from the satellite observation scheduling problem. This domain involves planning a collection of observation tasks between multiple satellites, each equipped with slightly different capabilities. In this new version a satellite can take images without being

calibrated. Besides, a satellite can be calibrated at any direction. The plans generated by off-the-shelf classical planners in this domain skip calibration actions because they produce longer plans. However, calibrations succeed more often at calibration targets and taking images without a calibration may cause execution dead-ends.

With the aim of making the analysis of results easier, we group the domains according to two dimensions, the determinism of the *action success* and the presence of execution *dead-ends*. Table 1 shows the topology of the domains chosen for the evaluation of PELA.

- *Action success.* This dimension values the complexity of the learning step. When probabilities are not state-dependent one can estimate their value counting the number of success and failure examples. In this regard, it is more complex to correctly capture the success of actions in domains where action success is state-dependent.
- *Execution Dead-Ends.* This dimension values the difficulty of solving a problem in the domain. When there are no execution dead-ends the number of problems solved is only affected by the combinatorial complexity of the problems. However, when there are execution dead-ends the number of problems solved depends also on the avoidance of these dead-ends.

	Probabilistic	**State-Dependent + Probabilistic**
Dead-Ends Free	*Blocksworld*	*Slippery-Gripper*, *Rovers*
Dead-Ends Presence	*OpenStacks*	*Triangle-tireworld*, *Satellite*

TABLE 1. Topology of the domains chosen for the evaluation of PELA.

4.2. Correctness of the PELA models

This experiment evaluates the correctness of the action models learned by PELA. The experiment shows how the error of the learned models varies with the number of learning examples. Note that this experiment focuses on the exploration of the environment and does not report any exploitation of the learned action models for problem solving. The off-line integration of learning and planning is described and evaluated later in the paper, at Section 4.3. Moreover this experiment does not use the learned models for collecting new examples. The on-line integration of exploration and exploitation in PELA is described and evaluated at Section 4.4.

The experiment is designed as follows: For each domain, PELA addresses a set of randomly-generated problems and learns a new action model after every twenty actions executions. Once a new model is learned it is evaluated computing the absolute error between (1) the probability of success of actions in the learned model and (2) the probability of success of actions in the *true model*, which is the PPDDL model of the *MDPsim* simulator. The probability of success of an action indicates the probability of producing the nominal effects of the action. Recall that our approach assumes actions have nominal effects. Since the probability of success may be state-dependent, each error measure is computed as the mean error over a test set of 1000 states[3]. In addition, the experiment reports the absolute deviation of the error measures from the mean error. These deviations –shown as square brackets– are computed after every one hundred actions executions and represent a confidence estimate for the obtained measures.

The experiment compares four different exploration strategies to automatically collect the execution experience:

[3]The 1000 test states are extracted from randomly generated problems. Half of the test states are generated with random walks and the other half with walks guided by LPG plans, because as shown experimentally, in some planning domains random walks provide poor states diversity given that some actions end up unexplored.

(1) *FF*: Under this strategy, PELA collects examples executing the actions proposed by the classical planner Metric-FF (Hoffmann, 2003).

This planner implements a deterministic forward-chaining search. The search is guided by a domain independent heuristic function which is derived from the solution of a relaxation of the planning problem.

In this strategy, when the execution of a plan yields an unexpected state, FF replans to find a new plan for this state.

(2) *LPG*: In this strategy examples are collected executing the actions proposed by the classical planner LPG (Gerevini *et al.*, 2003).

LPG implements a stochastic search scheme inspired by the SAT solver Walksat. The search space of LPG consists of "action graphs" representing partial plans. The search steps are stochastic graph modifications transforming an action graph into another one.

This stochastic nature of LPG is interesting for covering a wider range of the problem space. Like the previous strategy, LPG replans to overcome unexpected states.

(3) *LPG-εGreedy*: With probability ε, examples are collected executing the actions proposed by LPG. With probability $(1 - \varepsilon)$, examples are collected executing an applicable action chosen randomly. For this experiment the value of ε is 0.75.

(4) *Random*: In this strategy examples are collected executing applicable actions chosen randomly.

In the *Blocksworld* domain all actions are applicable in most of the state configurations. As a consequence, the four strategies explore well the performance of actions and achieve action models with low error rates and low deviations. Despite the set of training problems is the same for the four strategies, the *Random* strategy generates more learning examples because it is not able to solve problems. Consequently, the *Random* strategy exhausts the limit of actions per problem. The training set for this domain consisted of forty five-blocks problems. Figure 9 shows the error rates and their associated deviations obtained when learning models for the actions of the *Blocksworld* domain. **Note that the plotted error measures may not be within the deviation intervals because the intervals are slightly shifted in the X-axis for improving their readability.**

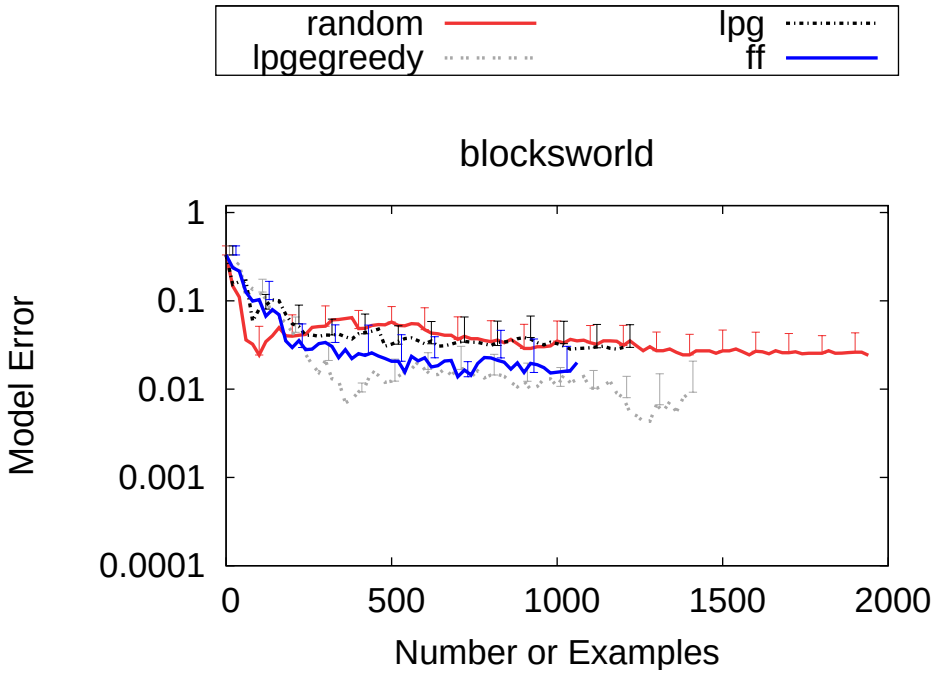

FIGURE 9. Error of the learned models in the *Blocksworld* domain.

In the *Slippery-gripper* there are differences in the speed of convergence of the different strategies.

Specifically, pure planning strategies FF and LPG converge slower. In this domain, the success of actions depends on the state of the gripper (wet or dry). Capturing this knowledge requires examples of action executions under both types of contexts, i.e., actions executions with a wet gripper and with a dry gripper. However, pure planning strategies FF and LPG present poor diversity of contexts because they skip the action `dry` as it means longer plans.

In the *Rovers* domain the *random* strategy does not achieve good error rates because this strategy does not explore the actions for data communication. The explanation of this effect is that these actions only satisfy their preconditions with a previous execution of actions `navigate` and `take-sample`. Unfortunately, randomly selecting this sequence of actions with the right parameters is very unlikely. Figure 10 shows error rates obtained when learning the models for the *Slippery-gripper* and the *Rovers* domain. The training set for the *Slippery-gripper* consisted of forty five-blocks problems. The training set for the *Rovers* domain consisted of sixty problems of ten locations and three objectives.

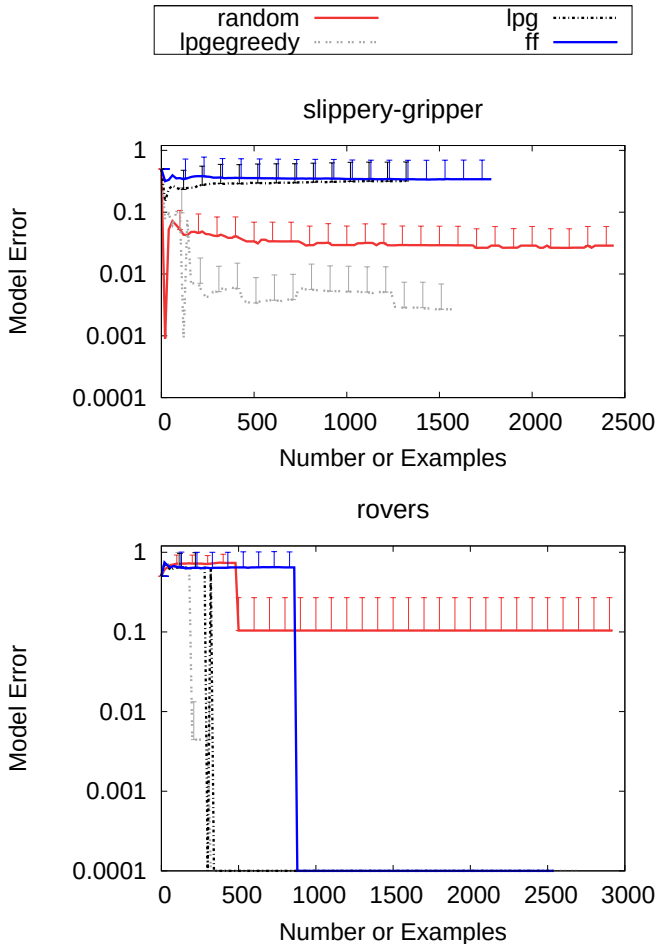

FIGURE 10. Model error in the *Slippery-gripper* and *Rovers* domains.

In the *Openstacks* domain pure planning strategies (FF and LPG) prefer the macro-action for making products despite it produces dead-ends. As a consequence, the original action for making products ends up being unexplored. As shown by the *LPG-ε Greedy* strategy, this negative effect is relieved including extra stochastic behavior in the planner. On the other hand, a full random strategy ends up with some actions unexplored as happened in the rovers domain.

In the *Triangle-tireworld* domain, error rates fluctuate roughly because the action model consists

only of two actions. In this domain the FF strategy does not reach good error rates because the shortest path to the goals always lack of `spare-tires`. The performance of the FF strategy could be improved by initially placing the car in diverse locations of the triangle. Figure 11 shows error rates obtained for the *Openstacks* and the *Triangle-tireworld* domain. The training set for the *Openstacks* consisted of one hundred problems of four orders, four products and six stacks. The training set for the *Triangle-tireworld* consisted of one hundred problems of size five.

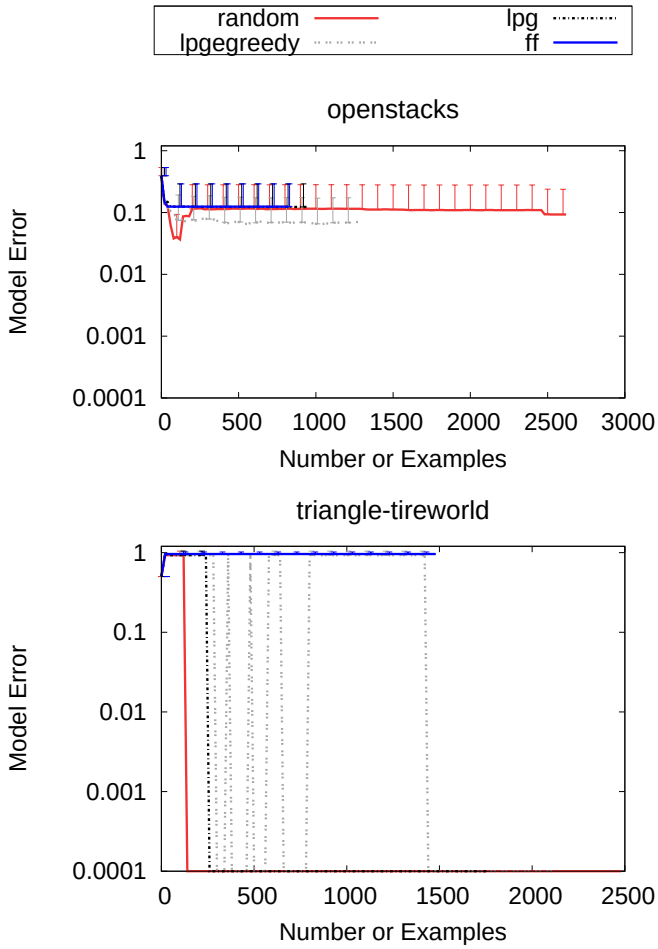

FIGURE 11. Model error in the *Openstacks* and the *Triangle-tireworld* domains.

For the *Satellite* domain we used two sets of training problems. The first one was generated with the standard problem generator provided by IPC. Accordingly, the goals of these problems always are either `have-image` or `pointing`. Given that in this version of the *satellite* domain can have images without calibrating, the action `calibrate` was only explored by the *random* strategy. However, the *random* strategy cannot explore action `take-image` because it implies a previous execution of actions `switch-on` and `turn-to` with the right parameters. To avoid these effects and guaranteeing the exploration of all actions, we built a new problem generator that includes as goals any dynamic predicate of the domain. Figure 12 shows the results obtained when learning the models with the two different training sets. As shown in the graph titled *satellite2*, the second set of training problems improves the exploration of the configurations guided by planners and achieves models of a higher quality. The training set for the *satellite* domain consisted of sixty problems with one satellite and four objectives.

Overall, the *random* strategy does not explore actions that involve strong causal dependencies.

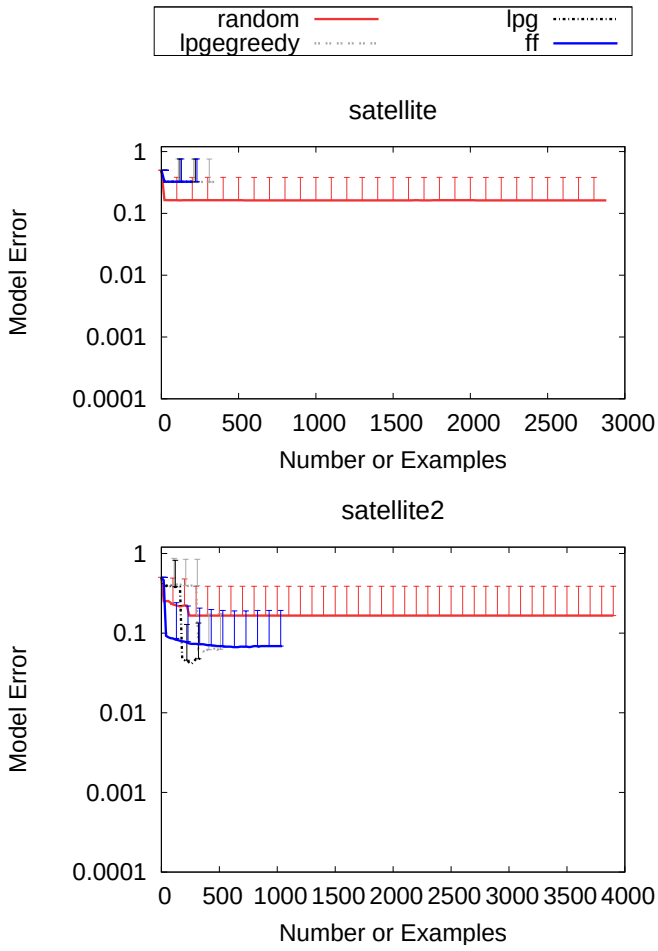

FIGURE 12. Error of the learned models in the *Satellite* domain.

These actions present preconditions that can only be satisfied by specific sequences of actions which have low probability to be chosen by chance.

Besides, random strategies generate a greater number of learning examples because random selection of actions is not a valid strategy for solving planning problems. Hence, the *random* strategy (and sometimes also the *LPG*ε*greedy*) exhausts the limit of actions for addressing the training problems. This effect is more visible in domains with dead-ends. In these domains *FF* and *LPG* generate fewer examples from the training problems because they usually produce execution dead-ends. On the other hand one can use a planner for exploring domains with strong causality dependencies. However, as shown experimentally by the FF strategy, deterministic planners present a strong bias and in many domains the bias keeps execution contexts unexplored. Even more, in domains with presence of execution dead-ends in the shortest plans, this strategy may not be able to explore some actions, though they are considered in the plans.

4.3. PELA off-line performance

This experiment evaluates the planning performance of the action models learned *off-line* by PELA. In the *Off-line* setup of PELA the collection of examples and the action modelling are separated from the problem solving process. This means that the updated action models are not used for collecting new observations.

The experiment is designed as follows: for each domain, PELA solves fifty small training problems

and learns a set of decision trees that capture the actions performance. Then PELA compiles the learned trees into a new action model and uses the new action model to address a test set of fifteen planning problems of increasing difficulty. Given that the used domains are stochastic, each planning problem from the test set is addressed thirty times. The experiment compares the performance of four planning configurations:

(1) FF + STRIPS *model*. This configuration represents the *classical re-planning* approach in which no learning is performed and serves as the baseline for comparison. In more detail, FF plans with the PDDL STRIPS-like action model and re-plans to overcome unexpected states. This configuration (Yoon *et al.*, 2007) corresponds to the best overall performer at the probabilistic tracks of IPC-2004 and IPC-2006.
(2) FF + PELA *metric model*. In this configuration Metric-FF plans with the model learned and compiled by PELA. Model learning is performed after the collection of 1000 execution episodes by the *LPGεGREEDY* strategy. The learned model is compiled into a metric representation (Section 3.2.1).
(3) GPT + PELA *probabilistic model*. GPT is a probabilistic planner (Bonet and Geffner, 2004) for solving MDPs specified in the high-level planning language PPDDL. GPT implements a deterministic heuristic search over the state space. In this configuration GPT plans with the action model learned and compiled by PELA. This configuration uses the same models than the previous configuration but, in this case, the learned models are compiled into a probabilistic representation (Section 3.2.2).
(4) GPT + *Perfect model*. This configuration is hypothetical given that in many planning domains, the perfect probabilistic action model is unavailable. Thus, this configuration only serves as a reference to show how far is PELA from the solutions found with a perfect model. In this configuration the probabilistic planner GPT plans with the exact PPDDL probabilistic domain model.

Even if PELA learned perfect action models, the optimality of the solutions generated by PELA depends on the planner used for problem solving. PELA addresses problem solving with suboptimal planners because its aim is solving problems. Solutions provided by suboptimal planners cannot be proven to be optimal so we have no measure of how far PELA solutions are from the optimal ones. Nevertheless, as it is shown at IPC, suboptimal planners success to address large planning tasks achieving good quality solutions.

In the *Blocksworld* domain the configurations based on the deterministic planning (*FF + STRIPS model* and *FF + PELA metric model*) solve all the problems in the time limit (1000 seconds). On the contrary, configurations based on probabilistic planning do not solve problems 10, 14 and 15 because considering the diverse probabilistic effects of actions boosts planning complexity. In terms of planning time, planning with the actions models learned by PELA generate plans that fail less often and require less replanning episodes. In problems where replanning is expensive, i.e., in large problems (problems 9 to 15), this effect means less planning time. Figure 13 shows the results obtained by the four planning configurations in the *Blocksworld* domain. The training set consisted of fifty five-blocks problems. The test set consisted of five eight-blocks problems, five twelve-blocks problems and five sixteen-blocks problems.

In the *Slippery-gripper* domain the *FF + STRIPS model* configuration is not able to solve all problems in the time limit. Since this configuration prefers short plans, it tends to skip the `dry` action. As a consequence, planning with the STRIPS model fails more often and requires more replanning episodes. In problems where replanning is expensive, this configuration exceeds the time limit. Alternatively, the configurations that plan with the models learned by PELA include the `dry` action because this action reduces the fragility of plans. Consequently, plans fail less often, less replanning episodes take place and less planning time is required.

In the *Rovers* domain the probabilistic planning configurations are not able to solve all the problems because they handle more complex action models and consequently they scale worse. In terms of planning time, planning with the learned models is not always better (problems 7, 12, 13, 15). In this domain, replanning without the fragility metric is very cheap and it is worthy even if it generates fragile plans that fail very often. Figure 14 shows the results obtained by the four planning

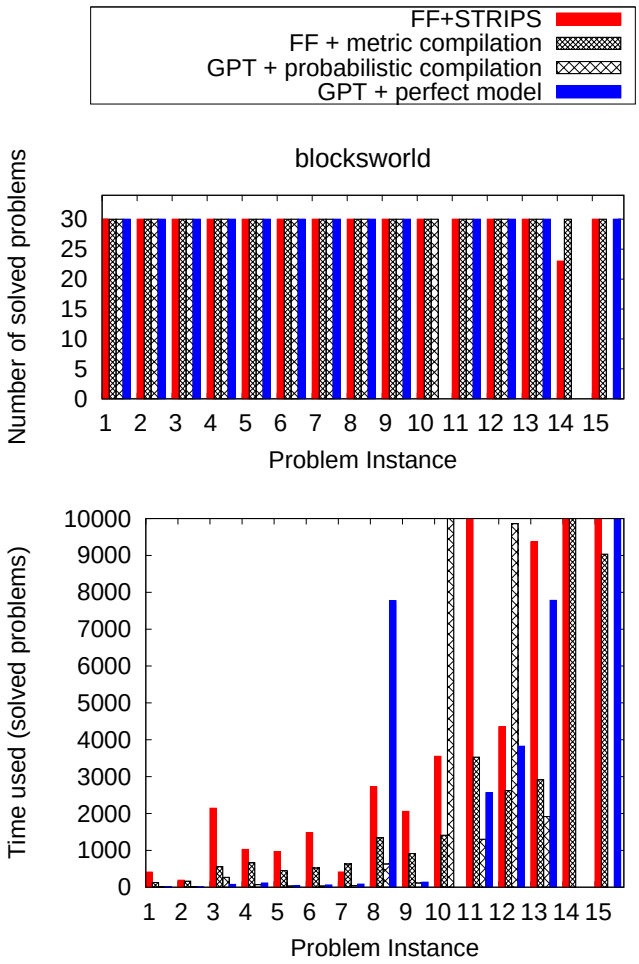

FIGURE 13. Off-line performance of PELA in the *Blocksworld*.

configurations in the *Slippery-gripper* and the *Rovers* domain. The training set for the *Slippery-gripper* consisted of fifty five-blocks problems. The test set consisted of five eight-blocks problems, five twelve-blocks problems and five sixteen-blocks problems. The training set for the *Rovers* domain consisted of sixty problems of ten locations and three objectives. The test set consisted of five problems of five objectives and fifteen locations, five problems of six objectives and twenty locations, and five problems of eight objectives and fifteen locations.

In the *Openstacks* domain planning with the STRIPS model solves no problem. In this domain the added macro-action for making products may produce execution dead-ends. Given that the deterministic planner FF prefers short plans, it tends to systematically select this macro-action and consequently, it produces execution dead-ends. On the contrary, models learned by PELA capture this knowledge about the performance of this macro-action so it is able to solve problems. However, they are not able to reach the performance of planning with the perfect model. Though the models learned by PELA correctly capture the performance of actions, they are less compact than the perfect model so they produce longer planning times. Figure 15 shows the results obtained in the *Openstacks* domain.

In the *Triangle-tireworld* robust plans move the car only between locations with spare tires available despite these movements mean longer plans. The STRIPS action model ignores this knowledge because it assumes the success of actions. On the contrary, PELA correctly captures this knowledge learning from plans execution and consequently, PELA solves more problems than the classical

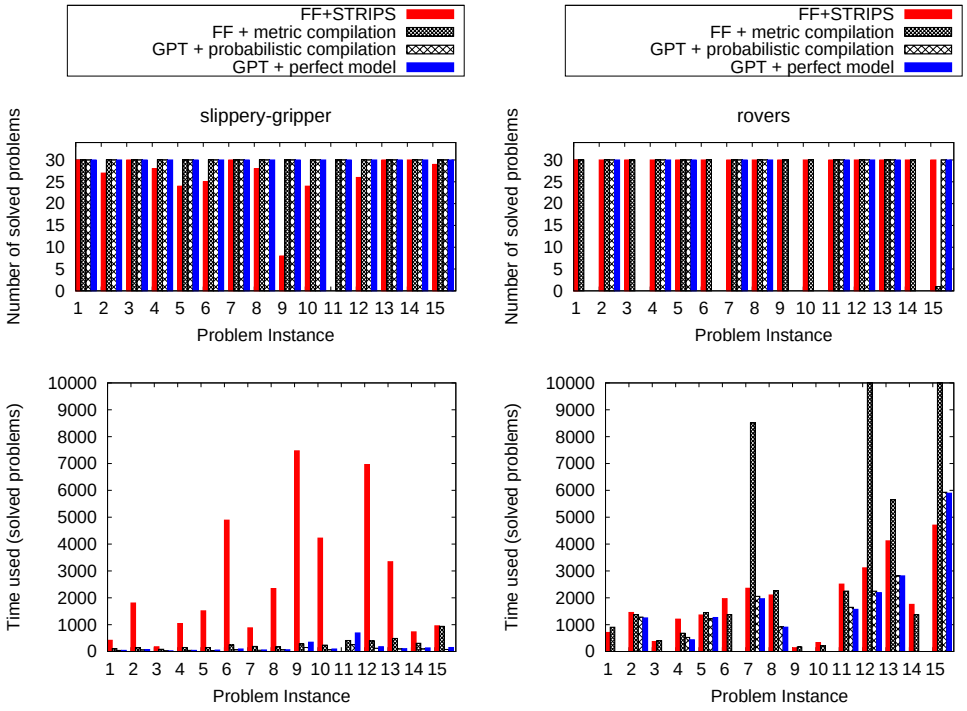

FIGURE 14. Off-line performance of PELA in the *Slippery-gripper* and the *Rovers* domains.

replanning approach. In terms of time, planning with the models learned by PELA means longer planning times than planning with the perfect models because the learned models are less compact.

In the *Satellite* domain planning with the STRIPS model solves no problem. In this domain the application of action `take-image` without calibrating the instrument of the satellite may produce an execution dead-end. However, this model assumes that actions always succeed and as FF tends to prefer short plans, it skips the action `calibrate`. Therefore, it generates fragile plans that can lead to execution dead-ends. Figure 16 shows the results obtained in the *Triangle-tireworld* and *Satellite* domain. The training set for the *Openstacks* consisted of one hundred problems of four orders, four products and six stacks. The test set consisted of five problems of ten orders, ten products and fifteen stacks; five problems of twenty orders, twenty products and twenty-five stacks and five problems of twenty-five orders, twenty-five products and thirty stacks. The training set for the *Triangle-tireworld* consisted of one hundred problems of size five. The test set consisted of fifteen problems of increasing size ranging from size two to size sixteen.

To sum up, in dead-ends free domains planning with the models learned by PELA takes less time to solve a given problem when replanning is expensive, i.e., in large problems or in hard problems (problems with strong goals interactions). In domains with presence of execution dead-ends, planning with the models learned by PELA solves more problems because dead-ends are skipped when possible. Otherwise, probabilistic planning usually yields shorter planning times than a replanning approach. Once a probabilistic planner finds a good policy, then it uses the policy for all the attempts of a given problem. However, probabilistic planners scale poorly because they handle more complex action models that produce greater branching factors during search. On the other hand a classical planner needs to plan from scratch to deal with the unexpected states in each attempt. However, since they ignore diverse action effects and probabilities, they generally scale better. Table 2 summarizes the number of problems solved by the four planning configurations in the different domains. For each domain, each configuration attempted thirty times fifteen problems of increasing difficulty (450 problems per domain). Table 3 summarizes the results obtained in terms of computation time in the solved problems by the four planning configurations in the different domains. Both tables show

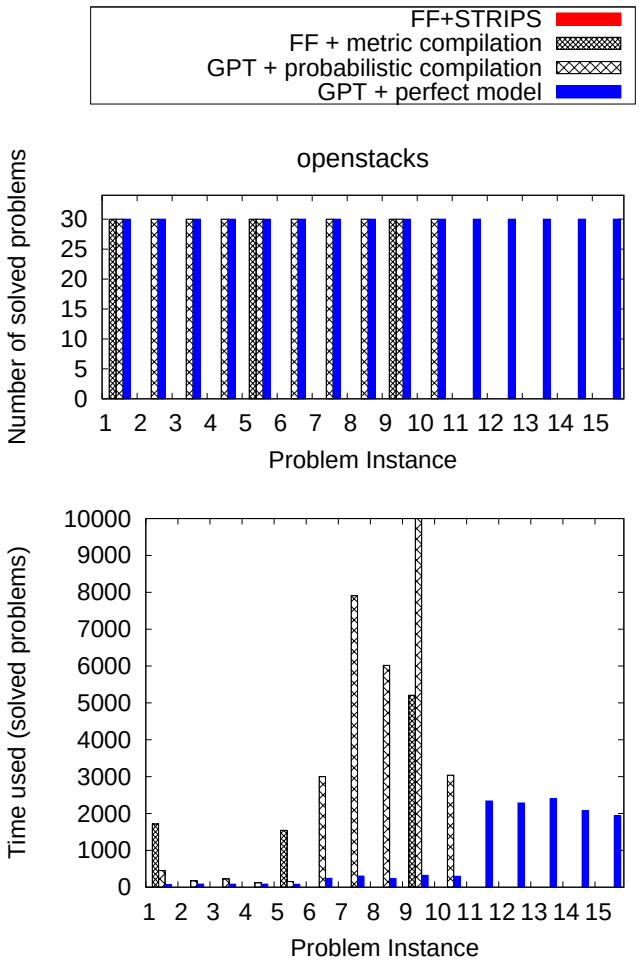

FIGURE 15. Off-line performance of PELA in the *Openstacks* domain.

results split in two groups: domains without execution dead-ends (*Blocksworld*, *Slippery-Gripper* and *Rovers*) and domains with execution dead-ends (*OpenStacks*, *Triangle-tireworld*, *Satellite*).

	Number of Problems Solved			
	FF	*FF+metric model*	*GPT+probabilistic model*	*GPT+perfect model*
Blocksworld (450)	443	450	390	390
Slippery-Gripper (450)	369	450	450	450
Rovers (450)	450	421	270	270
OpenStacks (450)	0	90	300	450
Triangle-tireworld (450)	5	50	373	304
Satellite (450)	0	300	300	420

TABLE 2. Summary of the number of problems solved by the off-line configurations of PELA.

The performance of the different planning configurations is also evaluated in terms of actions used to solve the problems. Figure 17 shows the results obtained according to this metric for all the domains. Though this metric is computed at the probabilistic track of IPC, comparing the

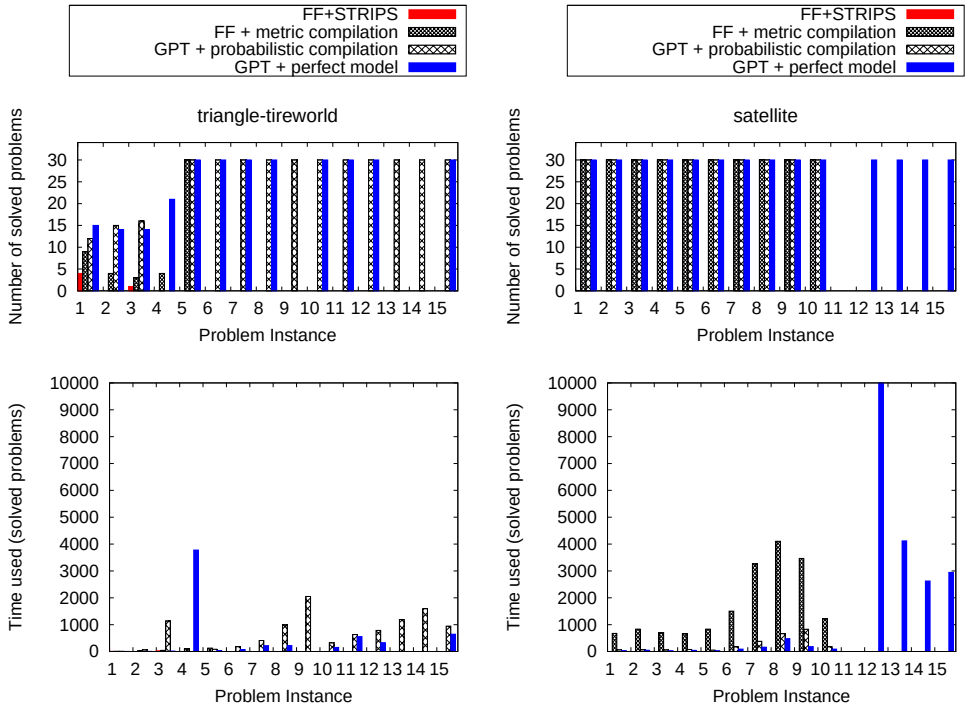

FIGURE 16. Off-line performance of PELA in the *Triangle-tireworld* and *Satellite* domains.

	Planning Time of Problems Solved (seconds)			
	FF	*FF+metric model*	*GPT+probabilistic model*	*GPT+perfect model*
Blocksworld	78,454.6	35,267.1	26,389.4	38,416.7
Slippery-Gripper	36,771.1	4,302.7	1,238.3	2,167.1
Rovers	28,220.0	349,670.0	18,635.0	18,308.9
OpenStacks	0.0	8,465.3	33,794.6	12,848.7
Triangle-tireworld	34.0	306.0	10,390.1	6,034.1
Satellite	0.0	17,244.1	2,541.3	21,525.9

TABLE 3. Summary of the planning time (accumulated) used by the four off-line configurations of PELA.

performance of probabilistic planners regarding the number of actions is tricky. In *probabilistically interesting* problems, robust plans are longer than fragile plans. When this is not the case, i.e., robust plans correspond to short plans, then a classical replanning approach that ignores probabilities will find robust solutions in less time than a standard probabilistic planner because it handles simpler action models.

4.4. PELA on-line performance

This experiment evaluates the planning performance of the models learned by PELA within an *on-line* setup.

The *on-line* setup of PELA consists of a closed loop that **incrementally** upgrades the planning action model of the architecture as more execution experience is available. In this setup PELA uses the updated action models for collecting new observations.

The experiment is designed as follows: PELA starts planning with an initial STRIPS-like action

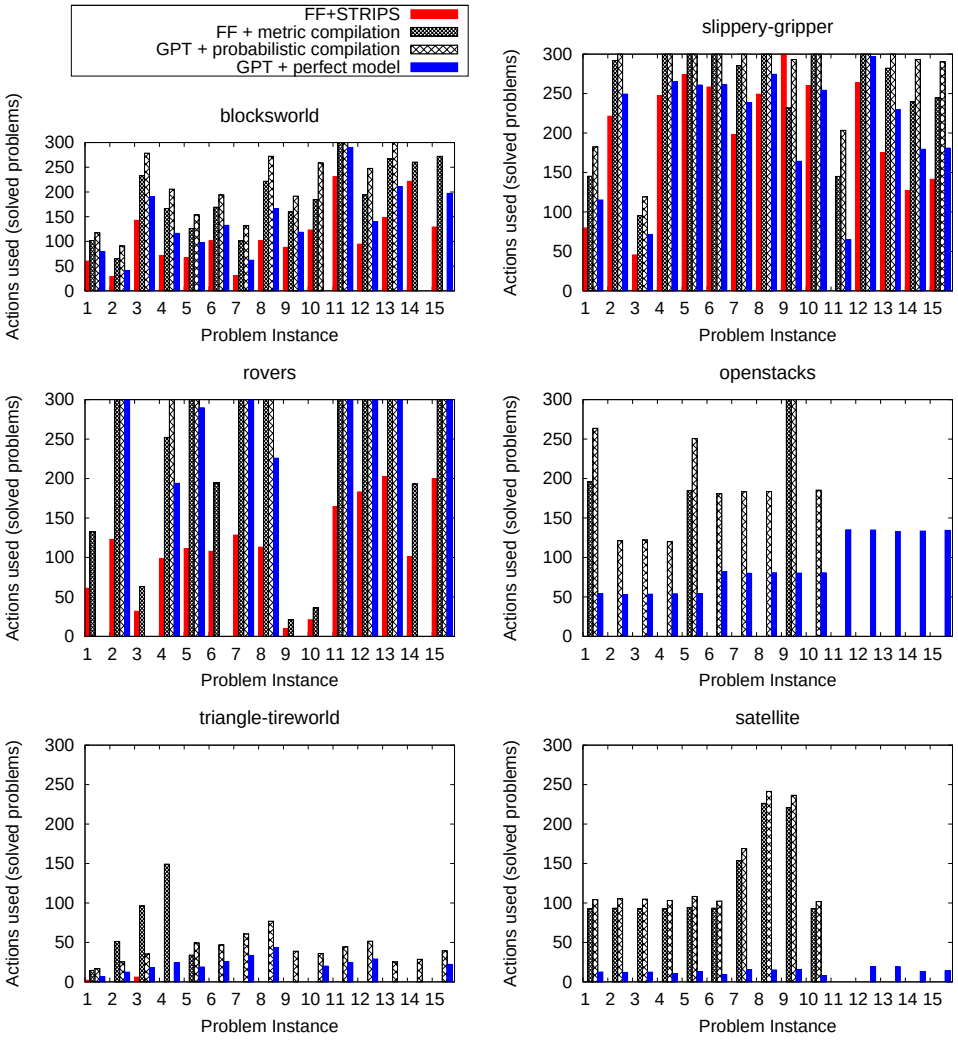

FIGURE 17. Actions used for solving the problems by the off-line configurations of PELA.

model and every fifty action executions, PELA upgrades its current action model. At each upgrade step, the experiment evaluates the resulting action model over a test set of thirty problems.

The experiment compares the performance of the two baselines described in the previous experiment (*FF + STRIPS* and *GPT + Perfect model*) against five configurations of the PELA on-line setup. Given that the baselines implement no learning, their performance is constant in time. On the contrary, the on-line configurations of PELA vary their performance as more execution experience is available. These five on-line configurations of PELA are named *FF-εGreedy* and present ε values of 0, 0.25, 0.5, 0.75 and 1.0 respectively. Accordingly, actions are selected by the planner FF using the current action model with probability ε and actions are selected randomly among the applicable ones with probability $1-\varepsilon$. These configurations range from *FF-εGreedy0.0*, a fully random selection of actions among the applicable ones, to *FF-εGreedy1.0*, an exploration fully guided by FF with the current action model. The *FF-εGreedy1.0* configuration is different from the *FF+STRIPS* off-line configuration because it modifies its action model with experience.

These configurations are an adaptation of the most basic exploration/exploitation strategy in RL to the use of off-the-shelf planners. RL presents more complex ways of exploring in which selection probabilities are weighted by their relative value functions. For an updated survey see (Wiering, 1999; Reynolds, 2002).

In the *Blocksworld* the five on-line configurations of PELA achieve action models able to solve the test problems faster than the classical replanning approach. In particular, except for the pure random configuration (*FF-ε Greedy0.0*), all PELA configurations achieve this performance after one learning iteration. This effect is due to two factors: (1) in this domain the knowledge about the success of actions is easy to capture because it is not state-dependent; and (2) in this domain it is not necessary to capture the exact probability of success of actions for robust planning; it is enough to capture the differences between the probability of success of actions that handle blocks and actions that handle towers of blocks.

In the *Slippery-gripper* domain the convergence of the PELA configurations is slower. In fact, the *FF-ε Greedy1.0* PELA configuration is not able to solve the test problems in the time limit until completing the fourth learning step. In this domain, the probability of success of actions is more difficult to capture because it is state-dependent. However, when the PELA configurations properly capture this knowledge, they need less time than the classical replanning approach to solve the test problems because they require less replanning episodes.

In the *Rovers* domain the performances of planning with STRIPS-like and planning with perfect models are very close because in this domain there is no execution dead-ends and replanning is cheap in terms of computation time. Accordingly, there is not much benefit on upgrading the initial STRIPS-like action model. Figure 18 shows the results obtained by the on-line configurations of PELA in the *Rovers* domain.

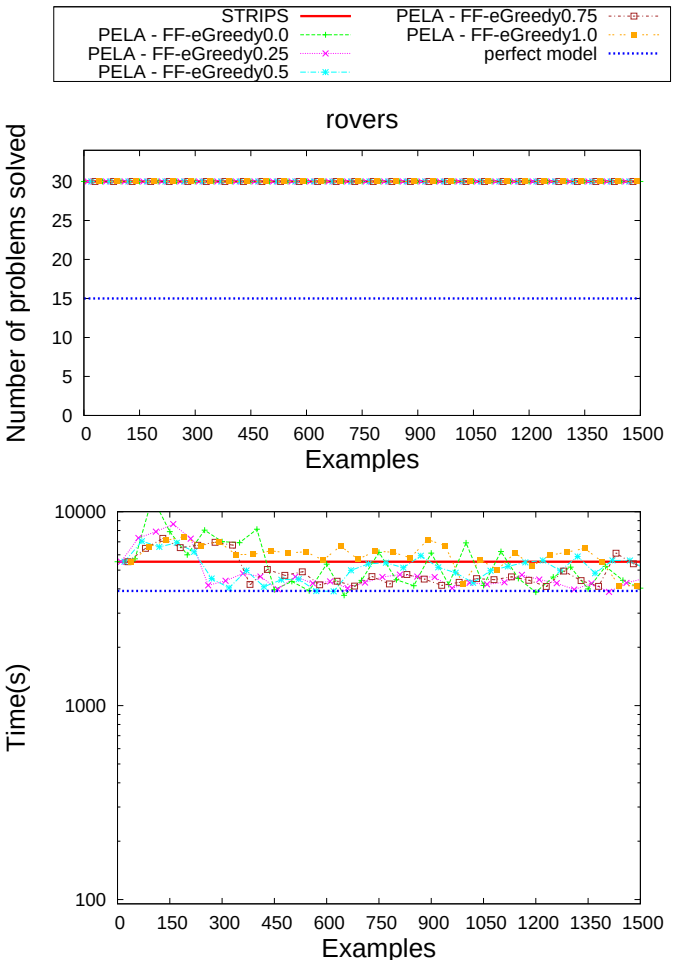

FIGURE 18. On-line performance of PELA in the *Rovers* domain.

In the *Openstacks* domain the *FF+Strips* baseline does not solve any problem because it generates plans that do not skip the execution dead-ends. On the contrary, all the on-line configurations of PELA achieve action models able to solve the test problems after one learning step. In terms of planning time, planning with the PELA models spends more time than planning with the perfect models because they are used in a replanning approach.

In the *Triangle-tireworld* the *FF-εGreedy1.0* configuration is not able to solve more problems than a classical replanning approach because it provides learning examples that always correspond to the shortest paths in the triangle. Though the PELA configurations solve more problems than a classical replanning approach, it is far from planning with the perfect model because FF does not guarantee optimal plans.

In the *Satellite* domain only the *FF-εGreedy1.0* configuration is able to solve the test problems because the strong causal dependencies of actions of the domain. These configurations are the only ones capable of capturing the fact that `take-image` may produce execution dead-ends when instruments are not calibrated. Figure 19 shows the results obtained in the *Satellite* domain.

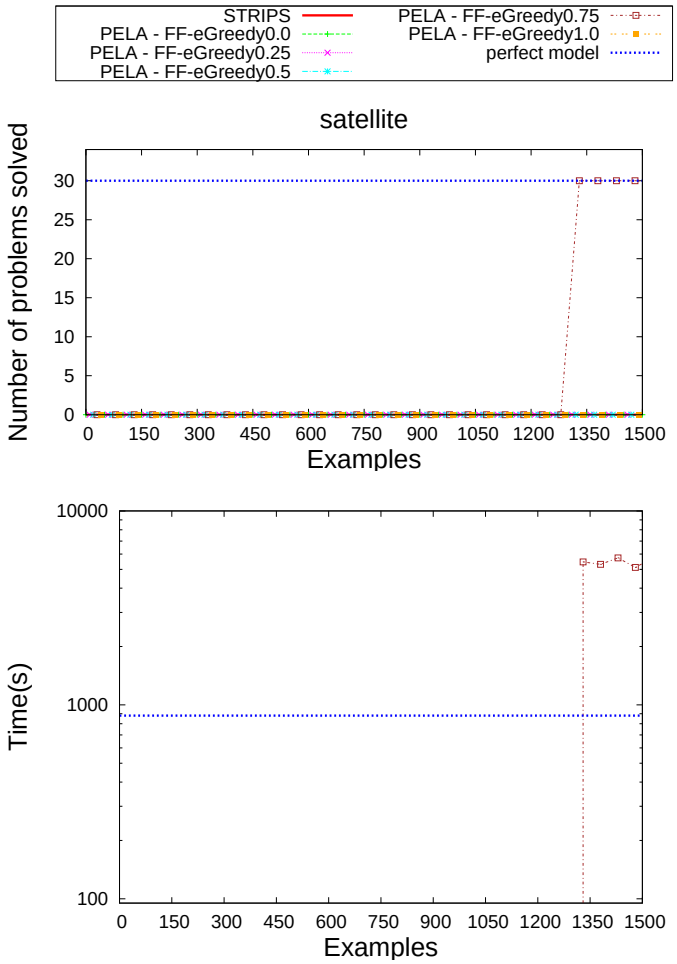

FIGURE 19. On-line performance of PELA in the *Satellite* domain.

Overall, the upgrade of the action model performed by PELA does not affect to actions causality. Therefore, the on-line configuration of PELA can assimilate execution knowledge without degrading the coverage performance of a classical replanning approach. In particular, experiments show that even at the first steps of the on-line learning process (when the learned knowledge is imperfect),

the introduction of the learned knowledge does not prevent PELA to solve problems. On the other hand, approaches that learn probabilistic action models from scratch by generalizing human provided observations (Pasula *et al.*, 2007) cannot guarantee the soundness of their intermediate action models. In particular, these approaches cannot guarantee that a given set of learning examples will produce an action model able to solve problems with a fixed planning strategy.

Besides, once PELA is presented with enough execution experience, the PELA on-line configurations address probabilistic planning problems more robustly than the classical replanning approach. Nevertheless, the action models learned within the on-line setup may not properly capture the performance of all actions in a given domain. Execution experience may be insufficient (generally at the first learning steps) or too biased (the training problems may provide learning examples of the same kind). As shown experimentally, these problems are more noticeable in domains with execution dead-ends. In these domains, the performance of the PELA on-line configurations depend on capturing some *key* actions, i.e., the actions that produce execution dead-ends. When a given configuration does not capture the success of the *key* actions it will perform poorly. On the other hand, this effect is less noticeable in domains free from execution dead-ends. In this kind of domains, configurations can outperform a classical replanning approach though the success of actions is not exactly captured. Table 4 summarizes the number of problems solved by the seven planning configurations in the different domains at the end of the on-line learning process. For each domain, each configuration attempted thirty problems of increasing difficulty. Table 5 summarizes the results obtained in terms of computation time in the solved problems by the seven planning configurations in the different domains. Both tables show results split in two groups: domains without execution dead-ends (*Blocksworld*, *Slippery-Gripper* and *Rovers*) and domains with execution dead-ends (*OpenStacks*, *Triangle-tireworld*, *Satellite*). The number of problems solved is not revealing in domains without dead-ends, because the seven configurations solve all the problems. In this kind of domains one must analyze the planning time, given that fragile plans imply more replanning episodes are needed, and consequently longer planning times. On the contrary, the number of problems solved is a reliable performance measure in domains with execution dead ends.

Number of Problems Solved at the end of the online process

	Strips	*ε Greedy0.0*	*ε Greedy0.25*	*ε Greedy0.5*	*ε Greedy0.75*	*ε Greedy1.0*	*Perfect model*
Blocksworld (30)	30	30	30	30	30	30	30
Slippery-Gripper (30)	30	30	30	30	30	30	30
Rovers (30)	30	30	30	30	30	30	15
OpenStacks (30)	0	30	30	30	30	30	30
Triangle-tireworld (30	0	2	0	1	0	0	15
Satellite (30)	0	0	0	0	30	0	30

Table 4. Summary of the number of problems solved by the on-line configurations of PELA.

Planning Time in the solved problems at the end of the online process

	Strips	*ε Greedy0.0*	*ε Greedy0.25*	*ε Greedy0.5*	*ε Greedy0.75*	*ε Greedy1.0*	*Perfect model*
Blocksworld	2,094.4	1,183.8	1,372.6	989.4	1,137.6	1,056.0	308.0
Slippery-Gripper	497.6	968.2	424.6	436.6	423.0	415.0	102.2
Rovers	5,522.2	4,037.4	4,526.0	5,003.4	4,992.0	4,233.8	3,906.2
OpenStacks	0	13,527.4	12,221.4	12,808.4	13,399.6	12,936.0	1,323.4
Triangle-tireworld	0	258.0	0	50.0	0	0	1,976.0
Satellite	0	0	0	0	5,730.4	0	881.0

Table 5. Summary of the computation time (accumulated) used by the four on-line configurations of PELA.

5. RELATED WORK

There is extensive prior work on general architectures for reasoning, execution and learning, ranging from execution-oriented, as in robotics applications (Peterson and Cook, 2003), to more cognitive-oriented (Rosenbloom *et al.*, 1993). The most relevant example to our work is ROGUE (Haigh and Veloso, 1999) which learned propositional decision trees and used them as control rules for the PRODIGY planning architecture (Veloso *et al.*, 1995). These architectures are not based on standard languages like PDDL or PPDDL for reasoning and learning, and different planning or learning techniques cannot be easily plugged-in and tested over a variety of domains.

The first approach for the Planning, Execution and Learning Architecture (Jiménez *et al.*, 2005) captured the performance of instantiated actions as control rules for the Prodigy planner (Veloso *et al.*, 1995). A second approach (Jiménez *et al.*, 2008), closer to the current architecture also learned relational trees about the performance of actions. This approach did not implement εGreedy strategies for the on-line integration of planning and learning and lacked of exhaustive evaluation.

There are recent works that also study how to automatically enrich planning action models. In many applications it is difficult to code PDDL actions for modeling motions: the geometry of the environment may be complex to analyze, the duration of these actions may be difficult to be defined, ... Instead, these works automatically verify and build planning actions of this kind using knowledge from motion plans (Choi and Amir, 2009; Wolfe *et al.*, 2010).

5.1. Action modelling

Focusing on the action modelling, we classified the existing learning approaches according to two features (1) the kind of learned models (*deterministic* versus *probabilistic*) and (2) the observability of the state of the environment (*partial* versus *full*).

Despite other classifications are possible, for instance the target of the learning (preconditions, effects, conditions of effects, probabilities of outcomes, ...) we believe this one is useful for planning purposes because each class corresponds to a different planning paradigm

5.1.1. *Learning deterministic actions in fully observable environments.* This problem is closely related to the Inductive Logic Programming (ILP) problem. In this regard, the *hot spots* for planning are how to generate significant learning examples (how to explore the performance of planning actions) and how to handle defective learned knowledge (how to plan when the learned action models are incomplete and incorrect). The LIVE system (Shen and Simon, 1989) alternated problem solving with rule learning for the automatic definition of STRIPS-like operators. The decision for alternation mainly depends on *surprises*, i.e., situations where an action's consequences violate its predicted models. When no rule can be found for solving the problem, LIVE will generate and execute an exploration plan, or a sequence of actions, seeking for surprises to extend the rule set. The EXPO system (Gil, 1992) refined *incomplete* planning operators, i.e., operators with some missing preconditions and effects. EXPO generates plans, monitors their execution and detects differences between the state predicted according to the internal action model and the observed state. EXPO constructs a set of specific hypotheses to fix the detected differences. After being heuristically filtered, each hypothesis is tested in turn with an experiment and a plan is constructed to achieve the situation required to carry out the experiment.

OBSERVER (Wang, 1994), unlike previous works that refined planning operators by an active exploration of the environment, learned operators by observing expert agents. The observations of the expert agent consists of: (1) the sequence of actions being executed, (2) the pre-state and the post-state resulting from the execution of each action. OBSERVER learned planning operators from these observation sequences in an incremental fashion utilizing a conservative specific-to-general inductive generalization process. Eventually, the system solves practice problems with the new operators to refine them from execution traces. The LOPE system (Garcia-Martinez and Borrajo, 2000) learned planning operators by observing the consequences of executing planned actions in the environment. At the beginning, the system has no knowledge, it perceives the initial situation, and selects a random action to execute in the environment. Then it loops by (1) executing an action, (2) perceiving the resulting situation of the action execution and its utility, (3) learning a model from the perception

and (4) planning for further interaction with the environment (in case the execution of the plan is finished, or the system has observed a mismatch between the predicted situation and the situation perceived). The planning component of LOPE does not explicitly receive a goal input given that LOPE creates its own goals from the situations with the highest utility.

Because in this category the effects of actions are deterministic and fully observable, they can be acquired by lifting the delta-state (the set of literals that differ between the pre-state and the post-state) of an action execution. In this regard, the main difficulty is to extract the actual preconditions of actions, because the direct lifting of a pre-state may include unnecessary preconditions. Recently, the work reported in (Walsh and Littman, 2008) succeeds to bound the number of interactions the learner must have with the environment to learn the preconditions (and effects) of a STRIPS action model.

5.1.2. *Learning deterministic actions in partially observable environments.* This category, given that observations of the current state are incomplete, requires ILP techniques able to deal with noise in the learning examples. In this category one can find two different approaches. On the one hand, the ARMS system (Yang *et al.*, 2007) which encodes example plan traces as a weighted maximum satisfiability problem, from which a candidate STRIPS-like action model is extracted. The output of ARMS is a single model, which is built heuristically in a hill-climbing fashion. Consequently, the resulting model is sometimes inconsistent with the input.

On the other hand, Amir and Chang introduced an algorithm that exactly learns all the STRIPS-like models that could have lead to a historical of observations (Amir and Chang, 2008). Given a formula representing the initial belief state, a sequence of executed actions $(a_1, a_2, ..., a_n)$ and the corresponding observed states $(s_1, ..., s_n)$, the learning algorithm updates the formula of the belief state with every action and observation in the sequence. This update makes sure that the new formula represents exactly all the transition relations that are consistent with the actions and observations. The formula returned at the end includes all consistent models, which can be retrieved then with additional processing. Both techniques do not consider the exploration process needed to extract the learning examples and assume they are provided by an external expert.

5.1.3. *Learning probabilistic actions in fully observable environments.* The PELA architecture fits in this category.

The task addressed in this category does not need to handle sets of possible states because they are fully observable. On the contrary, actions present stochastic effects, so they can not be learned by just lifting the delta-state. This task is very much related with the induction of stochastic logic models such as *Stochastic Logic Programs* (Muggleton, 2001; Cussens, 2001), *Bayesian Logic Programs* (Jaeger, 1997; Kersting and Raedt, 2001) or *Markov Logic Networks* (Richardson and Domingos, 2006).

One of the earliest works of this kind was the TRAIL system (Benson, 1997) that used Inductive Logic Programming to learn operators for reactive behavior. Besides preconditions and postconditions, these operators contained a success rate that indicated the percentage of times when the operator successfully achieved its postcondition.

The most relevant work in this category (Pasula *et al.*, 2007) is able to learn from scratch more expressive action models than PELA including preconditions and different outcomes of actions. However, this approach does not generate its own learning examples and requires specific planning and learning algorithms. Instead, PELA explores the world to generate its own learning examples, 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 kinds of off-the-shelf planners (like cost-based or probabilistic planners). Thus, PELA can directly profit from the last advances in both fields without modifying the source of the architecture. Even more, the off-the-shelf spirit of the architecture allows PELA to change the learning component to acquire other useful planning information, such as actions duration (Lanchas *et al.*, 2007).

5.1.4. *Learning probabilistic actions in partially observable environments.* Further studies are needed for action modelling in stochastic and partially observable environments. Preliminary work (Yoon and Kambhampati, 2007) addresses this problem using techniques for weighted maximum satisfiability in order to find the action model that better explains the collected observations.

5.2. Reinforcement Learning

RL agents interact with the environment to collect experience which, by means of appropriate algorithms, is processed to generate an optimal policy (Kaelbling *et al.*, 1996). RL includes two different approaches:

- *Model-Based* **RL uses a model of the environment to generate advise on how to explore it, so that the agent can find better policies. When** *model-based* **techniques use a relational representation (Croonenborghs *et al.*, 2007b), they produce action models similar to the ones learned by** PELA**. In this case, the main difference comes from the fact that** PELA **handles action models that follow the standard planning representation languages PDDL/PPDDL and that** PELA **delegates problem solving to off-the-shelf classical/probabilistic planners.**
- *Model-Free* **RL does not benefit from a model of the environment. In some domains there is so much uncertainty that learning to achieve goals is easier than learning accurate action models. Accordingly,** *model-free* **RL algorithms do not model the decision-making as a function of the state, like** *value/heuristic* **functions, but as a function of pairs $< state, action >$ called** *action-value* **functions. The** *q-function* **is an example of an** *action-value* **function which provides a measure of the expected reward for taking action a in state s. Relational representations of the** *q-function* **(Dzeroski *et al.*, 2001) adopt the same representation as symbolic planning to efficiently code the function in relational space states.**

The aims of RL are closely related to the aims of PELA**. In fact** *model-Based* **Relational Reinforcement Learning (RRL) techniques succeed to address some planning problems. Normally, these techniques rely on complete state enumeration and their time complexity is polynomial in the size of the state-space. In planning problems, the size of the state-space grows exponentially with the number of features describing the problem (objects properties and objects relations). Overcoming the state-space explosion of** *model-Based* **RL in planning problems is an interesting research line. A promising research direction is using** *heuristic search* **algorithms to limit computation to the states reachable from the initial state. Besides, these algorithms can benefit from domain-independent heuristics extracted from the planning problem representation, like the** FF **heuristic (Hoffmann and Nebel, 2001). This research line includes the** LAO* **algorithm (Hansen and Zilberstein, 2001), a generalization of the** A* **algorithm for MDPs, or the Learning Depth-First Search (LDFS) algorithm (Bonet and Geffner, 2006), a generalization of the IDA* for MDPs.**

Model-Free **RRL was also able to learn good policies for some type of planning problems like building a tower of blocks. However, more research is needed to efficiently address problems with highly interacting goals. For instance, when building a tower of specific blocks** *on(V,W), on(W,X), on(X,Y), on(Y,Z)***. This is the kind of problems traditionally addressed in automated planning where the achievement of a particular goal may undo previously satisfied goals. In these problems,** *model-Free* **RRL often spends a long time exploring regions of the state-action space without learning anything because no rewards (goal states) are encountered. To provide** *model-Free* **RRL with some positive rewards and relieve the limitations of random exploration, recent works exploit** *human-defined policies* **(Driessens and Matwin, 2004) or transfer learning (Croonenborghs *et al.*, 2007a).**

Besides, learning techniques for planning (Zimmerman and Kambhampati, 2003) try to learn general knowledge useful when solving any problem of a given domain. Learning of *generalized policies* **(Khardon, 1999; Martin and Geffner, 2000; Winner and Veloso, 2003) is an example of capturing this general knowledge.** *Model-Free* **RL focuses learning on addressing particular problems so each time the type of the problem changes** *model-Free* **RL agents need learning from scratch, or at least a** *transfer learning* **process (Fernández and Veloso, 2006; Torrey *et al.*, 2007; Mehta *et al.*, 2008; Taylor and Stone, 2009). When using a relational representation** *model-Free* **RRL can solve problems of the same type with additional objects without reformulating their learned policies,**

although additional training may be necessary to achieve optimal (or even acceptable) performance levels.

Works on RL study the convergence of the evaluation function as more experience is available. This function combines action model and problem solving information. PELA **follows greedy strategies for both, learning and problem solving, trading off optimality in exchange for scalability. In terms of model learning,** PELA **uses techniques for learning relational decision trees. Because of the huge size of the search space handled by these techniques** PELA **uses greedy search and heuristic pruning techniques that succeed to induce compact rule sets in a reasonable time. These rule sets are not proved to be optimal, because the exhaustive search of this search space is intractable. In terms of problem solving, there are two approaches in automated planning for guaranteeing optimality:**

- **Exploring the whole search space. Unfortunately, state spaces in planning are normally huge producing combinatorial explosion.**
- **Following an A* algorithm that explores the search space guided by an admissible heuristic. Unfortunately, admissible heuristics for planning are poorly informed, so optimal planners are only able to solve small problem instances. In any case, since we compile into two different planning schemes, advances in any of those can automatically improve planning under uncertainty as understood in this paper.**

6. CONCLUSION AND FUTURE WORK

This paper describes the PELA architecture for robust planning in non-deterministic domains. In order to achieve robust plans within these domains, PELA automatically upgrades an initial STRIPS like planning model with execution knowledge of two kinds: (1) probabilistic knowledge about the success of actions and (2) predictions of execution dead-ends. Moreover, the upgrade of the action models performed by PELA does not affect the actions causality and hence, it is suitable for on-line integration of planning and learning. The PELA architecture is based on off-the-shelf planning and learning components and uses standard representation languages like PDDL or PPDDL. Therefore, different planning and/or learning techniques can be directly plugged-in without modifying the architecture.

The performance of the architecture has been experimentally evaluated over a diversity of probabilistic planning domains:

- The *model correctness* experiments revealed that random explorations of planning domains improve the accuracy of the learned models because they explore the performance of actions in diverse contexts. However, pure random explorations are inappropriate for planning domains with strong causal dependencies. Random explorations do not explore actions that require the execution of a fixed sequence of steps to be applicable. In these domains, the use of planners with stochastic behavior (such as LPG or ε-greedy strategies) provide diversity to the learning examples, as well as considering causal dependencies of actions.
- The *off-line performance* experiments showed that the action models learned by PELA make both, metric-based and probabilistic planners, generate more robust plans than a classical replanning approach. In domains with execution dead-ends, planning with the models learned by PELA increases the number of solved problems. In domains without execution dead-ends, planning with the models learned by PELA is beneficial when replanning is expensive. On the other hand, the action models learned by PELA increase the size of the initial STRIPS model, meaning generally 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.
- The *on-line performance* experiments showed that the upgrade of the action models proposed by PELA does not affect to the actions causality and consequently, it is suitable for an on-line integration of planning and learning. Even at the first learning steps, in which the gathered experience is frequently scarce and biased, the performance of PELA is not worse than a classical

replanning approach. When the gathered experience achieves enough quality, PELA addresses probabilistic planning problems more robustly than the classical re-planning approach.

Currently we are working on using PELA to acquire more execution information useful for planning. An example is our previous work on learning actions durations (Lanchas *et al.*, 2007). Other interesting direction is learning dead-deads information with knowledge about the goals and including it in the planning models. With this regard a possible direction is using the *plan constraints and preferences* defined at PDDL3.0 (Gerevini *et al.*, 2009).

Our current version of PELA assumes there is an initial action model of the environment which correctly and completely captures the nominal effects of actions. However, in complex or changing environments, even defining simple STRIPS-like action models may become a hard task. We have an evident interest in relaxing this assumption. Recently, Kambhampati introduced the concept of model-lite planning (Kambhampati, 2007) for encouraging the development of planning techniques that do not search for a solution plan but for the most plausible solution plan that respects the current action model. New work on approximate inference (Yoon and Kambhampati, 2007; Thon *et al.*, 2008; Lang and Toussaint, 2009) seems to be a way of approaching the development of these new planning techniques. In addition the evaluation of PELA assumed full observability of the environment states. However, observations of the real-world may provide state representations of the environment that are incomplete or incorrect. Given that decision trees deal with noisy learning examples, a natural extension of this work is the study of the PELA performance in environments where observations provide states with noisy information.

Contrary to intuition, we have found that it is not always worthy to generate a very accurate and complete action model, because its expressiveness sometimes does not pay off the complexity of planning with it. As an example, FF-Replan was the best performer using a STRIPS model of actions for solving problems with probabilistic encodings at IPC-2004 and IPC-2006. In this way, further research is needed to determine in which situations learning a complex model and planning with it is better than learning a simple model and planning with it.

References

Amir, E. and Chang, A. (2008). Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, **33**, 349–402.

Benson, S. S. (1997). *Learning Action Models for Reactive Autonomous Agents*. Ph.D. thesis, Stanford University, California.

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. (2004). mgpt: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, **24**, 933–944.

Bonet, B. and Geffner, H. (2006). Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *International Conference on Automated Planning and Scheduling, ICAPS06*.

Choi, J. and Amir, E. (2009). Combining planning and motion planning. In *ICRA'09: Proceedings of the 2009 IEEE international conference on Robotics and Automation*, pages 4374–4380, Piscataway, NJ, USA. IEEE Press.

Croonenborghs, T., Driessens, K., and Bruynooghe, M. (2007a). Learning relational options for inductive transfer in relational reinforcement learning. In *Proceedings of the Seventeenth Conference on Inductive Logic Programming*.

Croonenborghs, T., Ramon, J., Blockeel, H., and Bruynooghe, M. (2007b). Online learning and exploiting relational models in reinforcement learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 726–731. AAAI press.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, **44**(3), 245–271.

Driessens, K. and Matwin, S. (2004). Integrating guidance into relational reinforcement learning. *Machine Learning*, **57**, 271–304.

Dzeroski, S., Raedt, L. D., and Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, **43**, 7–52.

Fernández, F. and Veloso, M. (2006). Probabilistic policy reuse in a reinforcement learning. In *International conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Fox, M. and Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, **20**, 61–124.

Fox, M., Gerevini, A., Long, D., and Serina, I. (2006). Plan stability: Replanning versus plan repair. *International Conference on Automated Planning and Scheduling (ICAPS'06)*.

Garcia-Martinez, R. and Borrajo, D. (2000). An integrated approach of learning, planning, and execution. *Journal of Intelligent and Robotics Systems*, **29**, 47–78.

Gerevini, A., Saetti, A., and Serina, I. (2003). Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, **20**, 239–290.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, **173**(5-6), 619–668.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning Theory and Practice*. Morgan Kaufmann.

Gil, Y. (1992). *Acquiring Domain Knowledge for Planning by Experimentation*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Haigh, K. Z. and Veloso, M. M. (1999). Learning situation-dependent rules. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information.*

Hansen, E. A. and Zilberstein, S. (2001). LAO * : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, **129**(1-2), 35–62.

Hoffmann, J. (2003). The metric-FF planning system: Translating ignoring delete lists to numerical state variables. *Journal of Artificial Intelligence Research*, **20**, 291–341.

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

Jaeger, M. (1997). Relational bayesian networks. In *Conference on Uncertainty in Artificial Intelligence*.

Jiménez, S., Fernández, F., and Borrajo, D. (2005). Machine learning of plan robustness knowledge about instances. In *European Conference on Machine Learning*.

Jiménez, S., Fernández, F., and Borrajo, D. (2008). The PELA architecture: integrating planning and learning to improve execution. In *National Conference on Artificial Intelligence (AAAI'2008)*.

Kaelbling, L. P., Littman, M. L., and Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, **4**, 237–285.

Kambhampati, S. (2007). Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Senior Member track of the AAAI*, Seattle, Washington, USA.

Kersting, K. and Raedt, L. D. (2001). Towards combining inductive logic programming with Bayesian networks. In *International Conference on Inductive Logic Programming*, pages 118–131.

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

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

Lang, T. and Toussaint, M. (2009). Approximate inference for planning in stochastic relational worlds. In *International Conference on Machine Learning. ICML*.

Little, I. and Thiébaux, S. (2007). Probabilistic planning vs replanning. In *Workshop on International Planning Competition: Past, Present and Future. ICAPS07*.

Martin, M. and Geffner, H. (2000). Learning generalized policies in planning using concept languages. In *International Conference on Artificial Intelligence Planning Systems, AIPS00*.

Mehta, N., Natarajan, S., Tadepalli, P., and Fern, A. (2008). Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, **73**(3), 289–312.

Muggleton, S. (2001). Stochastic logic programs. *Journal of Logic Programming*.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic

domains. *Journal of Artificial Intelligence Research*, **29**, 309–352.

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

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

Reynolds, S. I. (2002). *Reinforcement Learning with Exploration*. Ph.D. thesis, The University of Birmingham, UK.

Richardson, M. and Domingos, P. (2006). Markov logic networks. *Machine Learning*, **62**, 107–136.

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*. MIT Press.

Shen, W. and Simon (1989). Rule creation and rule learning through environmental exploration. In *International Joint Conference on Artificial Intelligence, IJCAI-89*.

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, **10**(1), 1633–1685.

Thon, I., Landwehr, N., and Raedt, L. D. (2008). A Simple Model for Sequences of Relational State Descriptions. In *European Conference on Machine Learning*.

Torrey, L., Shavlik, J. W., Walker, T., and Maclin, R. (2007). Relational macros for transfer in reinforcement learning. In *Conference on Inductive Logic Programming*, pages 254–268.

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.

Walsh, T. J. and Littman, M. L. (2008). Efficient learning of action schemas and web-service descriptions. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 714–719. AAAI Press.

Wang, X. (1994). Learning planning operators by observation and practice. In *International Conference on AI Planning Systems, AIPS-94*.

Wiering, M. (1999). *Explorations in efficient reinforcement learning*. Ph.D. thesis, University of Amsterdam IDSIA, The Netherlands.

Winner, E. and Veloso, M. (2003). DISTILL: Towards learning domain-specific planners by example. In *International Conference on Machine Learning, ICML'03*.

Wolfe, J., Marthi, B., and Russell, S. (2010). Combined task and motion planning for mobile manipulation. In *International Conference on Automated Planning and Scheduling*, Toronto, Canada.

Yang, Q., Wu, K., and Jiang, Y. (2007). Learning action models from plan traces using weighted max-sat. *Artificial Intelligence Journal*, **171**, 107–143.

Yoon, S. and Kambhampati, S. (2007). Towards model-lite planning: A proposal for learning and planning with incomplete domain models. In *ICAPS2007 Workshop on Artificial Intelligence Planning and Learning*.

Yoon, S., Fern, A., and Givan, B. (2007). Ff-replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling (ICAPS '07)*.

Younes, H., Littman, M. L., Weissman, D., and Asmuth, J. (2005). The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research*, **24**, 851–887.

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