Universidad Carlos III de Madrid

TESIS DOCTORAL

Planning and Learning under Uncertainty

Autor

Sergio Jiménez Celorrio

Directores

Dr. D. Daniel Borrajo Millán y Dr. D.Fernando Fernández Rebollo

Departamento

Departamento de Informática. Escuela Politécnica Superior

Leganés, Mayo 2011

TESIS DOCTORAL

Departamento de Informática. Escuela Politécnica Superior
Universidad Carlos III de Madrid

Planning and Learning under Uncertainty

Presentada por: Sergio Jiménez Celorrio
Dirigida por: Dr. D. Daniel Borrajo Millán y Dr. D.Fernando Fernández Rebollo

Para la obtención del grado de DOCTOR por la
UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Leganés, España
2011

Autor: Sergio Jiménez Celorrio

Título: Planning and Learning under Uncertainty

Grado: Doctor

Departamento: Informática

Universidad: Escuela Politécnica Superior, Universidad Carlos III de Madrid

Firma del autor

En Leganés a ..... de 2011

PLANNING AND LEARNING UNDER UNCERTAINTY

Autor: Sergio Jiménez Celorrio

Directores: Dr. D. Daniel Borrajo Millán y Dr. D.Fernando Fernández Rebollo

Tribunal Calificador		Firma
Presidente:	.....	.....
Vocal:	.....	.....
Vocal:	.....	.....
Vocal:	.....	.....
Secretario:	.....	.....

Calificación: .....

Leganés, ... de ..... de 2011

Contents

List of Figures

Acknowledgements

First, I would like to thank Daniel Borrajo and Fernando Fernández for supervising my PhD studies and for guaranteeing funds during the entire thesis course. Thanks also to James Cussens, Maria Fox and Derek Long –visiting your research groups was a turning point in my PhD.

Special thanks to all the people that contributed to this work. Thanks to Jesus Lanchas, Amanda and Andrew Coles for their collaborations and thanks to Emil Keyder for making his planner available.

Thanks to all the PLG members (Mr.Bedia and Mr.Manzano included), working with you has been fantastic. Thanks Carlos, for our talks in the corridors, Raquel and Susana for our AI meals and Tomas. Tomas I want to be like you.

Of course, thanks to Lamarta, Eljose and the sunset crew for sharing real moments with me throughout these years of work.

Last but not least, I want to thank my parents, B.O.R.J.A., the Arganda-Carreras Bunch, ojos-verdes and Acheipe Acheope for their unconditional support.

I dedicate this thesis to the colleagues that stopped their scientific careers because of the lack of funds.

Resumen

La Planificación Automática es la rama de la Inteligencia Artificial que estudia los procesos computacionales para la síntesis de conjuntos de acciones cuya ejecución permita alcanzar unos objetivos dados. Históricamente, la investigación en esta rama ha tratado de resolver problemas teóricos en entornos controlados en los que conocía tanto el estado actual del entorno como el resultado de ejecutar acciones en él. En la última década, el desarrollo de aplicaciones de planificación (gestión de las tareas de extinción de incendios forestales (Castillo et al., 2006), control de las actividades de la nave espacial *Deep Space 1* (Nayak et al., 1999), planificación de evacuaciones de emergencia (Muñoz-Avila et al., 1999)) ha evidenciado que tales supuestos no son ciertos en muchos problemas reales.

Consciente de ello, la comunidad investigadora ha multiplicado sus esfuerzos para encontrar nuevos paradigmas de planificación que se ajusten mejor a este tipo de problemas. Estos esfuerzos han llevado al nacimiento de una nueva área dentro de la Planificación Automática, llamada planificación con incertidumbre. Sin embargo, los nuevos planificadores para dominios con incertidumbre aún presentan dos importantes limitaciones: (1) Necesitan modelos de acciones detallados que contemplen los posibles resultados de ejecutar cada acción. En la mayoría de problemas reales es difícil obtener modelos de este tipo. (2) Presentan fuertes problemas de escalabilidad debido a la explosión combinatoria que provoca la complejidad de los modelos de acciones que manejan.

En esta Tesis se define un paradigma de planificación capaz de generar, de forma eficiente y escalable, planes robustos en dominios con incertidumbre aunque no se disponga de modelos de acciones completamente detallados. La Tesis que se defiende es que la integración de técnicas de aprendizaje automático relacional con los procesos de decisión y ejecución permite desarrollar sistemas de planificación capaces de enriquecer automáticamente su modelo de acciones con información adicional que les ayuda a encontrar planes más robustos. Los beneficios de esta integración son evaluados experimentalmente mediante una comparación con planificadores probabilísticos del estado del arte los cuales no modifican su modelo de acciones.

Abstract

Automated Planning is the component of Artificial Intelligence that studies the computational process of synthesizing sets of actions whose execution achieves some given objectives. Research on Automated Planning has traditionally focused on solving theoretical problems in controlled environments. In such environments both, the current state of the environment and the outcome of actions, are completely known. The development of real planning applications during the last decade (planning fire extinction operations (Castillo et al., 2006), planning spacecraft activities (Nayak et al., 1999), planning emergency evacuation actions (Muñoz-Avila et al., 1999)) has evidenced that these two assumptions are not true in many real-world problems.

The planning research community is aware of this issue and during the last years, it has multiplied its efforts to find new planning systems able to address these kinds of problems. All these efforts have created a new field in Automated Planning called planning under uncertainty. Nevertheless, the new systems suffer from two limitations. (1) They precise accurate action models, though the definition by hand of accurate action models is frequently very complex. (2) They present scalability problems due to the combinatorial explosion implied by the expressiveness of its action models.

This thesis defines a new planning paradigm for building, in an efficient and scalable way, robust plans in domains with uncertainty though the action model is incomplete. The thesis is that, the integration of relational machine learning techniques with the planning and execution processes, allows to develop planning systems that automatically enrich their initial knowledge about the environment and therefore find more robust plans. An empirical evaluation illustrates these benefits in comparison with state-of-the-art probabilistic planners which use static actions models.

Chapter 1

Introduction

This thesis work belongs to the area of Automated Planning in Artificial Intelligence. More specifically, it approaches the task of synthesizing plans in environments with uncertainty and how machine learning can assist in this task.

1.1 Overview

Planning is the capability of human-intelligence that concerns with the choice and organization of actions by anticipating their outcomes. As an example imagine you arrange a weekend trip to Paris. In this situation you plan, constrained by the money, the time and the knowledge you have, how to get there, which places to visit, where to sleep, etc. This is really a complex intellectual capability: the world where we live changes continuously and the vision we have of it is limited.

Automated Planning (AP) is the branch of Artificial Intelligence (AI) that studies the computational synthesis of sets of actions to carry out a given task. AP arose in the late '50s from converging studies into state-space search, theorem proving and control theory to solve the practical needs of robotics. The STanford Research Institute Problem Solver (STRIPS) (Fikes and Nilsson, 1971), developed for controlling the autonomous robot SHAKEY (Figure 1.1), perfectly illustrates the interaction of these influences. Since the SHAKEY'S days, an AP task is defined as: (1) the state-transition function of a dynamic system, (2) the initial state of the system and (3) the goals to be achieved.

AP is a PSpace-complete problem (Bylander, 1991) and solving complex AP problems with a general domain independent planner is still intractable. According to the previous definition, AP tasks seem to be easily tackled by searching for a path in a state-transition graph, which is a well-studied problem. However, in AP this state-transition graph can easily be so large that specifying it explicitly is not feasible (Bylander, 1994). Furthermore, when planning to solve real-world problems, the execution of actions can result in different outcomes making the AP processes even more complex. Regarding the *trip to Paris* example, our flight could be overbooked, a museum closed, our hotel full, etc.

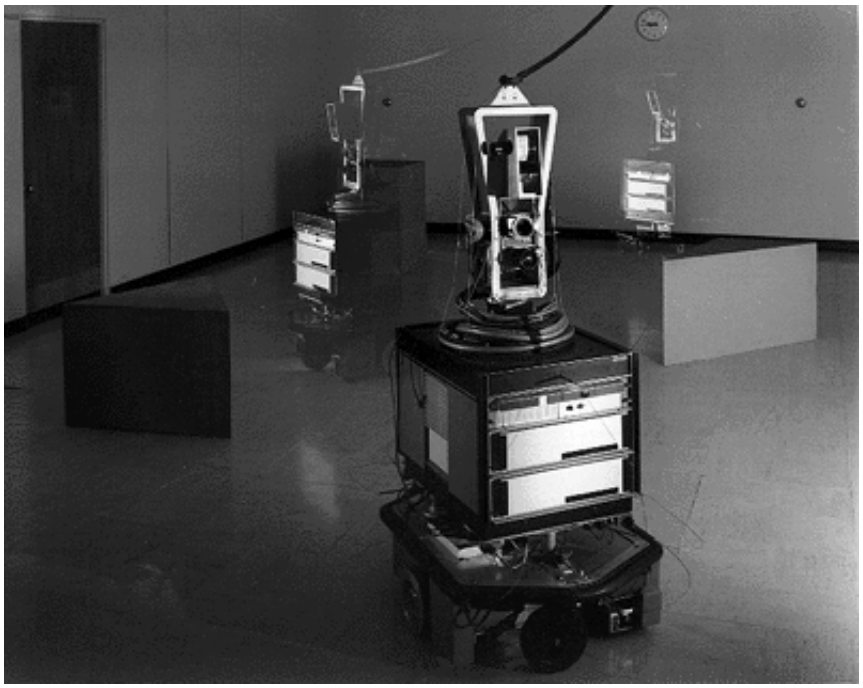

Figure 1.1: The *Shakey* robot (Nilsson, 1984) at the Stanford Research Institute.

Planning Under Uncertainty (PUU) is the subfield of AP that studies how to address planning problems in non-deterministic environments. In the early nineties, extensions to existing planners started to be developed to allow reasoning with partial perception of the environment and stochastic actions. At this moment, more elaborated PUU planners based on algorithms for solving MDPs and POMDPs have been developed. But these planners still present two important drawbacks: (1) Specifying accurate action models for non-deterministic environments is very complex and sometimes impossible, like when planning for the Mars Rovers (Bresina et al., 2005). (2) They do not scale well. Their complexity grows polynomially with the size of the state-space. In the AP problems this state-space grows exponentially in the number of predicates defined in the domain, making the current MDP-based algorithms not feasible for large problems.

But, how humans cope with the real-world planning-inherent difficulties? Basically we prioritize the possible situations and plan to cover only the most likely ones (leaving others to be completed as more information is available). Besides, we are able to use experience. We are endowed with learning skills that help us in the decision-taking. Back to the *trip to Paris* example, if we had previously visited Paris, we would have useful knowledge about how to get there, which places are worthy, how crowded they are, etc.

Machine Learning (ML) is the branch of AI which studies the process of chang-

ing the structure of computer programs in response to input information to improve future performances. Since the beginning, research in AI has been concerned with ML (as early as 1959 Arthur Samuel developed a program (Samuel, 1959) that learned to play better the game of checkers) so, very often, ML has been referred to changes in systems that perform tasks associated with AI, such as artificial perception, robot control or AP. Particularly, in the first '90s, ML was massively used within AP. At this time, classical planners were only able to synthesize no more than 10 action plans in many domains and ML allowed them to significantly speed and scale up. Nowadays, classical planners are able to achieve impressive performance in numerous domains, but the interest in intersecting ML with AP is alive with other aims like learning expert control knowledge for the new planning paradigms or learning planning action models.

In this thesis we claim that **ML can efficiently assist AP to build robust plans in stochastic environments by upgrading the planning action model with knowledge learned from execution**.

1.2 Objectives

Current PUU planners, based on heuristic search or dynamic programming, only obtain robust plans when they have complete and correct action models. Nevertheless, planning applications need to address PUU tasks in domains where complete and correct action models are not always available. Examples of these are the control of underwater vehicles, Mars planetary rovers, the management of emergency evacuations, fire extinctions, etc. Besides, current PUU planners suffer from scalability problems. These planners search for optimal or near-optimal plans reasoning about complex action models that explicitly represent the potential outcomes of actions. However, current techniques for finding optimal or near-optimal plans are computationally expensive and achieve limited success even for classical planning.

In this thesis, I aim to provide an innovative solution to these two open issues in PUU. This solution is based on complementing the planning and execution processes of PUU with learning abilities. **The overall objective of the thesis is the definition of an integration of the processes of planning, execution and learning that helps to, efficiently and scalably, synthesize robust plans in stochastic environments**.

The integration of techniques for PUU and ML is poorly studied. There are only few works on learning search control for MDPs (Yoon et al., 2002; Gretton and Thiébaux, 2004) and action modelling for non-deterministic domains (Benson, 1997; Pasula et al., 2007a). **The first specific objective of the thesis is to review the state-of-the-art of PUU and to analyze how ML can improve it**.

On the other hand, the use of ML within classical planning has been exhaustively studied (Zimmerman and Kambhampati, 2003) but there are still open issues that limit the practical use of current *planning and learning* systems: (1) traditionally planning and learning are integrated *ad-hoc*. That is, a specific learning

algorithm is designed to assist a particular planning algorithm; (2) the learning of planning knowledge is generally done off-line; that is, new knowledge is not assimilated during the planning process; and (3) the learning examples are provided by an external agent. **The second, third and fourth specific objectives of the thesis are to propose an architecture design for PUU that (1) integrates interchangeable and off-the-shelf planning and learning components, (2) implements both, off-line and on-line learning and (3) autonomously collects significant learning examples**.

1.3 Reader's guide to the thesis

This document is organized in three parts: **Part I** describes the state of the art in AP. Specifically, it reviews the literature about classical planning (Chapter 2), ML for assisting classical planning (Chapter 3), planning under uncertainty (Chapter 4) and ML for assisting planning under uncertainty (Chapter 5). **Part II** explains the work done along the thesis research. Particularly, it describes preliminary work in learning for planning under uncertainty (Chapter 6); second, it analyzes an integration proposal of planning, execution and learning for robust planning in domains with uncertainty (Chapter 7) and third, it illustrates how this integration proposal can be applied for learning planning models of the action durations (Chapter 8). Finally, **Part III** summarizes the contributions of this thesis and discusses some conclusions and future work.

Part I

State of the art

Chapter 2

Classical planning

This chapter is a review of existing classical planning techniques. The review is posed following a general problem solving approach.

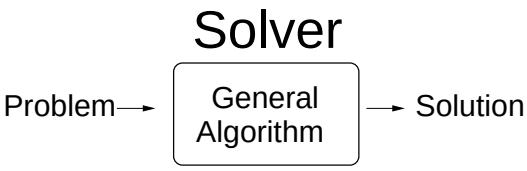

Figure 2.1: Overview of a general problem solver.

2.1 Introduction

In 1957 Herbert Simon and Allen Newell created the *General Problem Solver (GPS)* (Ernst and Newell, 1969). The aim of this work was to establish a universal mechanism that solves any problem that can be described in a high level representation using a general algorithm (Figure 2.1). Still nowadays, AI researchers continue exploring diverse forms in which computers might carry out the task of the general problem solving. Normally they all fit within these three components:

1. *A Conceptual Model*. The formal definition of the problems to solve and the shape of their solutions.

2. *A Representation Language*. The language used to describe the problem.

3. *An Algorithm*. The mechanism used to solve the problem.

Particularly, AP is a form of general problem solving concerned with the selection of actions in a dynamic system. Therefore AP requires a conceptual model able to describe dynamic systems. Most of the AP approaches take the *state-transitions*

model as their conceptual model. But they introduce several assumptions to this model that make it more operational:

1. *Finite World*. The dynamic system has a finite set of states.
2. *Static World*. The dynamic system stays in the same state until a new action is executed.
3. *Deterministic World*. The execution of an action brings the dynamic system to a single other state.
4. *Fully Observable World*. There is complete knowledge about the current state of the dynamic system.
5. *Implicit Time*. Actions have no duration. As a consequence, the state transitions are instantaneous.
6. *Restrictive Goals*. The only objective of the planning task is to find a set of state transitions that links a given initial state to another state satisfying the goals.

According to these assumptions, **Classical Planning** is the AP subfield that studies how to tackle the planning problems that can be described within this conceptual model. Otherwise, with the aim of addressing more realistic planning problems, some of these assumptions have been relaxed: **Temporal Planning** is the subfield that studies how to tackle planning problems when actions effects may not be instantaneous. **Planning Under Uncertainty** is the subfield that studies the relaxation of the *Full Observability* and *Deterministic World* assumptions. That is, planning problems with incomplete knowledge of the current state and where the outcome of actions is stochastic. **Planning with Continuous Actions** studies how to address the planning task when the effects of the actions are continuous, so the *Finite world* assumption is not true anymore. And **Planning with Extended Goals** studies how to find plans in domains where goals need to express requirements of different strengths such as constraints and user preferences.

2.2 The classical planning task

Classical Planning is the task of finding a sequence of actions that reaches a desired state in a deterministic and fully observable environment. Most of the research done in AP has focused on solving classical planning tasks. Thanks to that, classical planning is now a well formalized and well characterized problem with algorithms and techniques that scale-up reasonably well. Figure 2.2 shows an example of a classical planning task consisting of a robot navigation in a grid. In this example an autonomous robot can move one cell in four different ways: *up*, *down*, *left* or *right*. The robot is on cell `A5` and must reach cell `D2` avoiding the obstacles (cells `B2`, `D3`, and `E4`).

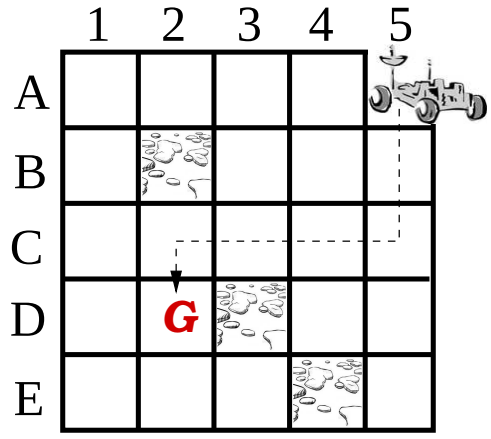

Figure 2.2: Example of a classical planning task.

2.3 The conceptual model

The conceptual model for classical planning is a deterministic, finite and fully observable state-transition system denoted by $\sum = (S, A, C)$, where:

- S is a finite set of states.
- A is a finite set of actions.
- $C(s, a)$ is a function representing the cost of applying the action $a \in A$ in the state $s \in S$.

According to this conceptual model, a classical planning problem is defined as a triple $P = (\sum, s_0, G)$ where $s_0 \in S$ is the initial state and $G \subseteq S$ is the set of goal states. Finding a solution to a classical planning problem P consists on generating a sequence of actions $(a_1, a_2, ..., a_n)$ corresponding to 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 $s_n \in G$ is a goal state. The optimal solution for a classical planning problem is the one that minimizes the expression $\sum_{i=1}^{n} c(s_{i-1}, a_i)$.

2.4 The representation languages

A planning representation language is a notation for the syntax and the semantic of planning tasks. Particularly this representation capability must support the description of: the initial state of the environment, the environment state transitions (actions), the goals of the planning task and the solution plans. Next there is a review of the main languages used for representing the planning task.

STRIPS

Though the STRIPS (Fikes and Nilsson, 1971) notation is one of the oldest planning representation languages, it is still very popular. STRIPS handled the *Frame Problem*[1] assuming that actions only change a small part of the world. That is, if situation s_n results from applying action a_n in situation s_{n-1}, then STRIPS assumes that s_n and s_{n-1} are very much alike. According to this assumption any formula non explicitly asserted in a state is taken to be false, which allows STRIPS to avoid the explicit specification of negated literals.

The STRIPS representation of a planning problem is a tuple $P =< L, A, I, G >$ where:

- L: represents the literals set.
- A: represents the actions set.
- $I \subseteq L$: represents the subset of literals holding in the given initial state.
- $G \subseteq L$: represents the subset of literals holding in a goal state.

STRIPS represents state s_i as a conjunction of literals from L indicating the facts holding in instant i and action $a \in A$ as the following three lists of literals from the set L:

1. *The preconditions list*, $Pre(a) \subseteq L$: subset of literals that need to be true for the application of action a.
2. *The delete list*, $Del(a) \subseteq L$: subset of literals no longer true after the application of action a.
3. *The add list*, $Add(a) \subseteq L$: subset of literals made true by the application of action a.

The STRIPS representation of a solution plan is a sequence P of *applicable* actions $P = a_0, ..., a_n$ that *results* in some goal state $s_n \in G$ starting from the given initial state s_0. Formally, the concepts of *applicable* action and *resulting* state are defined as:

1. The actions *applicable* in a state s_i are those $a \in A$ such that $Pre(a) \subseteq s_i$
2. The state *resulting* from applying the action a in the state s_i is $result(s_i, a) = \{s_i / Del(a)\} \cup Add(a)$

Figure 2.3 shows an example of a planning action represented in the STRIPS notation. Specifically, the example shows the STRIPS representation for the action `unstack(block,block)` from the *Blocksworld*. The *Blocksworld* is a classic

[1] The *Frame Problem* (McCarthy and Hayes, 1969) is the problem of expressing a dynamic system using logic without explicitly specifying which conditions are not affected by an action.

domain in AP which consists of a set of blocks, a table and a gripper: the Blocks can be on top of other blocks or on the table, a block that has nothing on it is clear and the gripper can hold one block or be empty. Given its clarity and complexity, it is by far the most frequently domain used in the AI planning literature.

```
unstack(TOP,BOTTOM)
 Pre: [emptyhand, clear(TOP), on(TOP,BOTTOM)]
 Del: [emptyhand, clear(TOP), on(TOP,BOTTOM)]
 Add: [holding(TOP), clear(BOTTOM)]
```

Figure 2.3: STRIPS representation for the action `unstack` from the *Blocksworld*.

ADL

The Action Description Language (ADL) (Pednault, 1994) augmented the representation capability of the STRIPS notation with the expressive power of the situation calculus. The main contributions of ADL were:

1. The introduction of negations, disjunctions and quantified formulas in the action preconditions and problem goals.

2. The introduction of the equality predicate for the comparison of variables.

3. The definition of conditional effects. Now, actions may have different outcomes according to the current state.

Figure 2.4 shows an example of a planning action represented in ADL. This is the ADL representation implemented in the UCPOP planner for the action `unstack(block,block)` in a version of the *Blocksworld*. In this version the action `unstack(block,block)` only succeed when the gripper is not blocked.

```
Unstack(top,bottom)
PRECOND: top≠bottom∧On(top,bottom)∧∀¬Holding(b)∧∀¬On(b,top)
EFFECTS: ¬Blockedhand|Holding(top)∧¬On(top,bottom)
```

Figure 2.4: ADL representation for the action `unstack` from the *Blocksworld*.

PDDL

The Planning Domain Definition Language (PDDL) (Fox and Long, 2003) was created with the aim of standardising the planning representation languages and facilitate comparative analyses of the diverse planning systems. PDDL was developed as the planning input language for the International Planning Competition

(IPC).[2] Along the different IPCs, PDDL has evolved to cover the representation needs of the new AP challenges:

1. PDDL1.2 (used in IPC-1998 and IPC-2000) contained the STRIPS and the ADL functionality plus the use of typed variables.

2. PDDL2.1 (IPC-2002) augmented the original PDDL version with:

   - Numeric variables and the ability to test and update their values.
   - Durative actions with both discrete and continuous effects.

3. PDDL2.2 (IPC-2004) extended the previous versions with:

   - Derived predicates, which allow the planner to reason about higher-level concepts in the domain. Additionally, these higher-level concepts can be recursively defined.
   - Timed initial literals, which are literals that will become true at a predictable time independent of what the planning agent does.

4. PDDL3.0 (IPC-2006) enriched the expresivity of the language to define:

   - State trajectory constraints that the solution plan must satisfy.
   - Goal and State trajectory preferences that the solution plan should satisfy.

5. PDDL3.1 (IPC-2008) supports *functional* STRIPS (Geffner, 2001). *Functional* STRIPS is a different encoding for the planning domain. Instead of mapping the literals of the planning problem to $true$ or $false$, *functional* STRIPS maps the objects of the planning problem to their properties. This encoding provides a more natural modelling for many planning domains and makes easier the extraction of some heuristics, such as the causal graph heuristic (Helmert and Geffner, 2008) or *pattern database* heuristics (Edelkamp, 2002).

Thought PDDL3.1 covers all these functionalities most of the existing planners do not implement them; in fact, the majority only support the STRIPS subset besides typing and the equality predicate. Figure 2.5 shows the action `unstack(block,block)` from a version of the *Blocksworld* domain represented in PDDL. In this version the execution of the action `unstack` takes 10 units of time and 5 units of the gripper's battery level.

[2] The International Planning Competition takes place every two years since 1998 with the aim of evaluating the performance of state-of-the-art planning techniques (http://ipc.icaps-conference.org/)

```
(:action unstack
  :parameters (?top - block ?bottom)
  :duration (= ?duration 10)
  :precondition (and (not (= ?top ?bottom))
                     (on-top-of ?top ?bottom)
                     (forall (?b - block)
                             (not (holding ?b)))
                     (forall (?b - block)
                             (not (on-top-of ?b ?top)))
                     (> (battery) 5))
  :effect (and (holding ?top)
               (not (on-top-of ?top ?bottom))
               (decrease (battery) 5)))
```

Figure 2.5: PDDL representation for the action `unstack` from the *Blocksworld*.

2.5 The algorithms

There are two main approaches to solve classical planning problems: (1) explicit **search** for a solution plan or (2) **compiling** the classical planning problem into another form of problem solving with effective algorithms.

Search for planning

Search algorithms systematically explore a graph trying to find a path that reaches one of the goal nodes n_g starting from a given initial node n_0. The different search algorithms are characterised by the following features:

- *The search space*. In AP the state-space, the plan-space and the planning-graph are the most used search spaces.

  1. The state-space. In this search space each graph node corresponds to a state of the dynamic system and each arc corresponds to a state transition resulting from an action execution.
  2. The plan-space search. In this case the graph nodes are partially specified plans and arcs correspond to plan refinement operations.
  3. The planning-graph it is a middleground search space. It is a directed layered graph where arcs are only permited from one layer to the next one. There are two kinds of layer: (1) *action layer*, the set of actions whose preconditions are holding in the previous *proposition layer* and (2) *proposition layer*, the union of the effects added by the actions of the previous *action layer* plus the literals of the previous *proposition layer* (at first the literals of initial state). In this search space, the nodes represent the literals holding in a given instant i. The arcs from a *proposition layer* to an *action layer* represent the preconditions of the actions

and the arcs from the *action layer* to the *proposition layer* represent the positive effects of the actions.

- *The search direction*. From the initial node to the goal nodes (forward), from the goal nodes to the initial node (backward) or bidirectional, in this case a forward and a backward search are run simultaneously until they meet in the middle.

- *The search algorithms*. Once the search space and the search direction are defined one can systematically search for a solution. The simplest search algorithms are the brute-force algorithms that exhaustively visit the nodes of the search space according to their depth to the root. Instances of these algorithms are Breadth-First and Depth-First search. In AP the search space is frequently extremely large, so, rather than brute-force search algorithms, one needs algorithms that focus search on the nodes that seems more promising. In order to focus the search, one can use a heuristic function $h(n)$ that estimates how close node n is to a goal node. Of course the performance of these algorithms is determined by the accuracy of the defined heuristic function. There are a whole bunch of different heuristic search algorithms (Russell and Norvig, 1995): Hill-Climbing, Best-First Search, Iterative Deepening Best-First Search, etc.

- *The heuristic function*. Heuristics can be directly derived as the cost of the solutions to a relaxed task. To derive domain-independent heuristics, this relaxed task should be automaticaly extracted from the encoding of the original one. The more common relaxations of the planning task are:

  1. Ignoring the actions' delete lists. Using the planning-graph one can compute a solution to this relaxed task in linear time. Most of the current heuristic planners rely on this idea to implement their heuristic function.
  2. Ignoring certain atoms. There is a new trend in building heuristics for AP following the ideas of *Pattern DataBases*.[3] However the applications of this technique to classical planning (Edelkamp, 2002) is still not straightforward given that the quality of this kind of heuristics depends crucially on the selection of a collection of abstractions (patterns) that are appropriate to the domain.
  3. Reducing the goal sets to subsets. One can partition the planning problem into subproblems each with its own subset of goals, solve these subproblems and combine the cost of these solutions to compute heuristics.

[3] *Pattern DataBases* are dictionaries of heuristics that have been originally applied to the Fiffteen Puzzle (Culberson and Schaeffer, 1998) and have achieved significant success in traditional heuristic search benchmarks (Felner et al., 2004).

Generally, the heuristics implemented in AP are non-admissible. This fact is due to (1) the existing admissible heuristics are poorly informed and (2) the application of optimal search algorithms to planning domains is frequently too expensive in terms of computation time.

- *The pruning method*. The search algorithm can be aided with pruning mechanisms that eliminate the nodes considered non-promising. More precisely, the heuristic search planner FF (Hoffmann and Nebel, 2001) introduced the concept of *helpful actions* for restricting the successors of any state to those produced by members of the respective relaxed solution.

Compilation for planning

The most popular compilations of the classical planning task are:

- Planning as boolean satisfiability (SAT).[4] This compilation (Kautz and Selman, 1992a) uses the good scalability performance of the SAT solvers for addressing the classical planning problem. The compilation lies in encoding all the potential solution plans of length N as a boolean formula. Thus any assignment of truth values that makes the formula satisfiable represents a valid plan for the planning problem. The planning problem is compiled as follows:

  1. Propositions and actions are instantiated and time-tagged.
     (a) The initial state is coded as a set of propositions holding at time 0.
     (b) The goals are coded as propositions holding at time at time N.
  2. Different actions at the same time t $(0 < t < N)$ are mutually exclusive.
  3. If the preconditions of an action are holding at time t the effects of this action must hold at time $t + 1$.
  4. The frame problem is handled by stating that any proposition p is *TRUE* at time t if: (1) it was *TRUE* at time $t - 1$ and the action taken at time t does not make it *FALSE*, or (2) the action taken at time t makes p *TRUE*.

  Given that *Planning as SAT* correctly finds the plans of a given length N, it can be directly used for finding optimal solutions in terms of plan length by systematically increasing the value of N.

[4] The SAT problem is a NP-Complete problem (Garey and Johnson, 1979) that lies on determining if truth values can be assigned to the variables of a given boolean formula in such a way that the formula evaluates to *TRUE*.

- Planning as a constraint satisfaction problem (CSP).[5] Similar to the SAT compilation, the CSP compilation (van Beek and Chen, 1999) encodes the potential solution plans of length N as a CSP problem. In this case any assignment of values that makes the resulting CSP problem satisfiable represents a valid plan for the planning problem. For this compilation the planning problem is encoded into a state-variable representation as follows:

  1. Let D be the set of objects of a planning problem and $D_c \subseteq D$ the subset of objects of the class C. And let A be the set of all instantiated actions.
  2. There are two types of variables in the compiled CSP problem:
     (a) State variables of the form $state_i(t)$, with value v ranging over D_c represent that the fact $i(v)$ is true at time t.
     (b) An extra variable $action(t)$, ranging over A, whose value represents the action taken in state t.
  3. There are four types of constraints in the compiled CSP problem:
     (a) Every state variable $state_i(t)$ whose value in the initial state is v is encoded into the unary constraint $state_i(0) = v$
     (b) Every state variable $state_i(t)$ whose value in the goals is v is encoded into the unary constraint $state_i(N) = v$
     (c) Every precondition v of the action a is encoded as the binary constraint:
     $$(action(t) = a, state_i(t) = v)$$
     (d) Every action effect v of the action a is encoded as the binary constraint:
     $$(action(t) = a, state_i(t + 1) = v)$$
     (e) In order to handle the frame problem, for every action $action(t)$ and every state variable value $state_i(t)$ non affected by the action $action(t)$, there is a constraint:
     $$(action(t) = a, state_i(t) = v, state_i(t + 1) = v)$$

  Unlike the SAT compilation, the CSP compilation is more compact: (1) the encoding of states using variables is smaller and (2) the variable $action(t)$ allows one to avoid the need of the mutual exclusion constraints. Furthermore, this compilation can also cover planning problems with numeric variables.

[5] The CSP is a NP-Complete problem that lies on, given a set of variables, the domain of values they can take and a set of constraints defining the incompatibilities of the variable values, determining if there is a compatible assignment of values to the variables that does not violates the constraints.

- Planning as model checking.[6] In this compilation (Cimatti et al., 1997), the planning domain is formalized as a specification of the possible models for plans; the initial state plus the problem goals, usually described in temporal logic, are formalized as requirements about the desired behavior for the plans. Finally, the planning problem is solved by searching through the possible plans, checking that there exists one that satisfies the specified requirements. This compilation allows one to naturally define state trajectories or soft goals as extra requirements over desired plans. Besides it supports the definition of expressive search control knowledge also coding it as new plan requirements.

2.6 The implementations

This is an overview of planners that have introduced relevant advances in the subfield of classical Planning.

STRIPS

The STRIPS planner (Fikes and Nilsson, 1971) based on simple backward chaining in the state-space. STRIPS worked with a subgoals stack and tried to always satisfy first the subgoal in the top of the stack to produce an ordered sequence of actions that solves the planning task.

UCPOP

UCPOP (Penberthy and Weld, 1992) was based on searching in the plans space so it could work on any subgoal non linearly. The UCPOP algorithm started with a dummy plan that consists of a *start* action whose effects are the initial state and a *goal* action whose preconditions are the goals. UCPOP attempted to complete this dummy plan by adding new actions until all preconditions are guaranteed to be satisfied. If UCPOP has not yet satisfied a precondition, then all action effects that could satisfy the desired proposition are considered. UCPOP chooses one effect and then adds a causal link to the plan to record this choice. If a third action might interfere with the precondition being supported by the causal link, then UCPOP tries to resolve the interference by: (1) reordering steps in the plan, (2) creating additional subgoals, or (3) adding new equality constraints.

[6] The Model Checking problem (J.R. Burch et al., 1990) consists of determining whether a given property, usually described as a temporal logic formula, holds in a given model of a system. Traditionally, this problem has been studied for the automatic verification of hardware circuits and network protocols

SATPLAN

SATPLAN (Kautz and Selman, 1992b) constructs a planning-graph up to some length N; it compiles the constraints implied by the planning-graph into a conjunctive normal form (CNF) formula and solve it using a state-of-the-art SAT solver. If no solution was found at this length, SATPLAN increases N and starts again. The output of this planner is an optimal plan in terms of solution length.

PRODIGY

PRODIGY (Veloso et al., 1995) is a non-linear planner provided with a depth-first backward chaining means-ends analysis search. The search implementation allowed goals interleaving to solve the non-linear problems. The PRODIGY representation language was very expressive supporting conditional effects, numeric predicates, invocations of external functions and the definition of domain dependent control knowledge for pruning and guiding the search process.

GRAPHPLAN

GRAPHPLAN (Blum and Furst, 1995) starts with an initial planning-graph which only contains the *proposition layer* with the literals of the initial state. Then GRAPHPLAN builds a planning-graph expanding sequentially *action* and *proposition layers* until it reaches a *propositional layer* that satisfies the goals of the planning task. This planning-graph contains all potential parallel plans up to a certain length N, where N is the number of action layers; after that, GRAPHPLAN tries to extract a plan by searching the graph backwards from the goals. The GRAPHPLAN search either finds a solution plan or determines that the goals are not all achievable by a plan of length N. In this case it extends the planning graph one time step (the next action layer and the next propositional layer), and then it searches again for a solution plan.

FF

FF (Hoffmann and Nebel, 2001) is a forward-chaining heuristic state-space planner. FF uses a domain independent heuristic function derived from the cost of the solution to a relaxation of the planning problem. This relaxation consists of ignoring the delete lists of all actions and extracting an explicit solution using a GRAPHPLAN-style algorithm. The number of actions in the relaxed solutions is taken as a goal distance estimation. These estimations guide the heuristic algorithm enforced hill-climbing (EHC) that performs a variation of breadth-first search. Specifically EHC performs a breadth-first search which is interrupted each time a node successor s' of node s satisfies $h(s') < h(s)$ retaking up the breadth-first search again from s'. In case EHC search fails, FF automatically switches to weighted best-first search. Moreover, the relaxed plans are used to prune the

search space. Usually, the actions that are really useful in a state are contained in the relaxed plan.

SHOP

SHOP (Nau et al., 2003) is based on ordered task decomposition, which is a type of Hierarchical Task Network (HTN) planning. HTN planners decompose tasks into subtasks until they reach tasks that can be performed directly by the planning actions. Thus, they need to have a set of methods indicating how to decompose tasks into subtasks. For each of these tasks they may have more than one applicable method, i.e., more than one way to decompose the tasks. At this point, SHOP performs a forward search to decide the alternative decomposition that is solvable at a lower level.

2.7 Discussion

The first classical planners, from the 70's such as STRIPS (Fikes and Nilsson, 1971) performed a simple goal regression in the state-space. These planners presented two important drawbacks: They were not scalable because their search process was non-informed, e.g., STRIPS implemented an exhaustive Depth-First Search algorithm. And they were not able to solve non-linear problems like the Sussman Anomaly.[7]

In the early 90's two new planning paradigms attempted to address these limitations. Partial order planners like UCPOP (Penberthy and Weld, 1992) solved non-linear problems by searching in the plan space. However, because the search of partial order planers was still not informed it still suffered from scalability problems. The second approach consisted on directly improving the STRIPS paradigm (Veloso et al., 1995): (1) allowing the planner to interleave subgoals during the search so non-linear problems can be solved and (2) providing the planner with domain-dependent control knowledge for making the search more efficient.

In the mid 90's two important contributions tackled the scalability deficiencies of AP from a domain-independent point of view. On the one hand, the planning-graph provided a framework to significantly reduce the AP search space. And on the other, the first powerful domain independent heuristics were developed. The Heuristic Search Planner (HSP) (Bonet and Geffner, 2001) implemented a *hill-climbing* search algorithm with a heuristic function built by relaxing the planning task into a simpler one that ignored the delete effects of all the actions. Later, the FF (Hoffmann and Nebel, 2001) planner introduced a new heuristic computed as the length of the *relaxed plan* automatically extracted from the *relaxed planning*

[7] The Sussman Anomaly (Sussman, 1975) is a *Blocksworld* problem with three blocks labeled A, B, and C. The problem starts with B on the table, A on the table and C on top of A, and consisted on stacking the blocks such that A is on top of B and B is on top of C. Typically, non-linear planners separate the goals into two subgoals `on(A,B)` and `on(B,C)` and tried to satisfy them independently.

graph, a new variation of the hill-climbing algorithm, called *enforced hill-climbing*, for reducing the number of node evaluations and a novel pruning mechanism called *helpful actions* for eliminating from the search the actions that are not promising according to the relaxed planning graph. In the last years, the researchers working on heuristic planning are making big efforts in defining domain-independent methods for extracting heuristics through the construction of pattern databases (Haslum et al., 2007).

In the late 90's the organization of the IPC caused a revolution in the AP research community. The first competition (IPC-1998) allowed to compare the performance of all these AP paradigms and established standard test benches and representation languages to easily test and evaluate the new developed planning techniques. At the next competition (IPC-2000), planners based on domain dependent control knowledge participated for the first time. These planners were hand tuned to guide their search processes according to the structure of the domain. They significantly outperformed the rest of participants and achieved performance values that even nowadays are not reachable by the state-of-the-art domain independent planners. IPC-2002 augmented the scope of the planning tasks. Particularly, the expressiveness of the standard planning language PDDL was extended to cope with durative actions and continuous resource consumption. IPC-2004 established a separated competition for optimal planning. Since optimal planners prove a guarantee on the quality of the found solution their performance on computation time and plan quality is not comparable with the satisfying planners. IPC-2006 gave emphasis to plan quality as defined by a external user in terms of trajectory constraints and/or soft goals. IPC-2008 enriched the planning representation language to support functional STRIPS (Geffner, 2001) which provides a more natural encoding for new promising AP heuristics based on the causal-graph or in the construction of PDBs.

At the present time there are still important open issues in classical planning that need to be tackled:

- **Domain-dependent planners significantly outperform the existing domain-independent planners**. Contrary to the domain independent techniques they are able to capture the regularities of the domains, meaning impressive speed and scalability performance. However, the expert knowledge used by these planners may not be completely available; this is partly because it is very hard to encode such knowledge, and it is partly because there is no expert to provide it, for example, in the space missions. An important open issue is how this expert knowledge can be extracted and compiled automatically and independently of the domain.

- **Off-the-shelf classical planners cannot deal with requirements of real-world problems**. Though the standard planning representation language PDDL allows one to represent complex user requirements, such as user preferences and restrictions, quality metrics, durative and concurrent actions,

... most of the state-of-the-art classical planners do not support them or are not able to manage them efficiently.

- **Classical planning models are hard to design, validate and maintain**. Knowledge acquisition is also a bottleneck in the field of classical planning. The process of encoding a planning domain model is laborious and at present, most of the domain model designers use planners as their only tool to develop and debug domain models.

Chapter 3

Learning for classical planning

This chapter is a review of the major approaches for classical planning that profit from ML. The review is organized according to the target of the ML process: learning **search control** or learning planning **action models**.

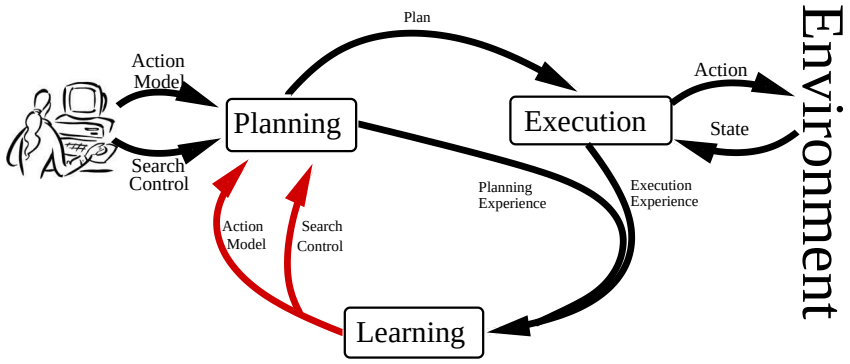

Figure 3.1: Integration of ML within classical planning.

3.1 Introduction

As Mitchell points out in his ML textbook (Mitchell, 1997), there is a set of common issues when using ML to improve an automatic process:

1. *The target concept*. The type of knowledge that the ML process will learn. When talking about planning this knowledge can be: **search control** to guide the planners search or **domain models** to feed the planners (Figure 3.1).

2. *The extraction of the experience*. The experience can be self-extracted by the planning agent or provided by an external agent, such as a human expert. In the first case there are three different opportunities where knowledge can be automatically extracted: before the planning starts, i.e., **learning from the**

problem representation, during the planning process, i.e., **learning from the search tree** or after the planning process, i.e., **learning from the execution of plans**.

3. *The knowledge representation*. When learning for planning one must take two representation decisions:

   - *The features space*. The set of features that the ML algorithm considers for learning the target concept. In the case of planning these features may be the **state predicates**, the **goal predicates**, the **previous selected action**, etc.
   - *The representation language*. The notation used for encoding the target concept and the experience. The most used representation languages for encoding planning features are **predicate logic** and **description logic**.

4. *The learning paradigm*: **inductive**, **analytical** or **hybrid** algorithms.

   - **Inductive Learning**. These learning techniques induce general theories based on observations. In this approach, the input to the learning process is a set of observed instances and the output is a classifier consistent with them used to classify the subsequent instances. Inductive Learning can be broken into Supervised, Unsupervised and Reinforcement Learning. In **Supervised Learning** inputs and correct classification outputs are *'a priori'* known. In **Unsupervised Learning** just the inputs are specified. Finally, in **Reinforcement Learning** the learning agent collects its own inputs and correct outputs by trial and error.
   - **Analytical Learning**. These techniques use prior knowledge and deductive reasoning to explain the information provided by the learning examples. This fact makes analytical techniques not to be so constrained to the learning examples to achieve good generalizations.
   - **Inductive-Analytical Learning**. Purely inductive learning techniques formulate general hypotheses by finding empirical regularities over the learning examples. Purely analytical techniques use prior knowledge to derive general hypothesis deductively. This learning approach combines the two techniques to obtain the benefits of both: a better generalization accuracy when prior knowledge is available and reliance on observed learning data to overcome shortcomings of prior knowledge.

5. *The learning environment*. The environment determines the final learning technique. For example, learning in **stochastic** environments means coping with noise, learning in **partially observable** environments means coping with incomplete examples, etc.

3.2 Learning techniques

Since the STRIPS days, AP defines models of the planning tasks using dialects of predicate logic because this language allows planners to represent problems (1) compactly, it uses variables, and (2) relationally, it captures relations between objects. Therefore, the learning techniques for improving AP normally support these two features. Nevertheless, other ML techniques like genetic algorithms have also been used in AP. In these situations, a compilation of the planning knowledge to other representations, such as attribute-value representation, has to be performed. Next, there is description of the relational learning techniques that best fit to the AP tasks.

3.2.1 Inductive Logic Programming

Inductive Logic Programming (ILP) arises from the intersection of ML and logic programming and deals with the development of inductive techniques to learn a given target concept from examples and background knowledge. Formally, the inputs to an ILP system are:

1. A representation language L, specifying the syntactic restrictions of the domain predicates.

2. A set of ground learning examples E of the target concept, consisting of true examples (positive examples) E^+ and false examples (negative examples) E^- of the concept to learn.

3. Background knowledge B, which provides additional information to argument the classification of the learning examples.

Figure 3.2 shows an example of language bias, set of learning examples and background knowledge for learning the concept of `grandfather(X,Y)`.

```
 L={ grandfather(male,person),father(male,person),
     mother(female,person)}
E+={ grandfather(pepe,juan),grandfather(pepe,ana),
     grandfather(pepe,carlos)}
E-={ grandfather(pepe,luis),grandfather(ana,carlos)}
 B={ father(pepe,luis),father(pepe,ana),father(luis,ana),
     father(luis,juan),mother(ana,carlos)}
```

Figure 3.2: Inputs for learning the `grandfather(X,Y)` concept with ILP.

With these three inputs, ILP systems try to find a hypothesis H, expressed in L which agrees with the examples E and the background knowledge B. In terms of

logical inference, this task is defined as the induction of a *complete* and *consistent logic program*[1] H:

- H is *complete* when for all $e \in E^+$ it is true that $H \vdash e$.
- H is *consistent* when for none $e \in E^-$ it is true that $H \vdash e$.

Figure 3.3 shows an example of *complete* and *consistent logic program* found for the concept of `grandfather(X,Y)` with the inputs of Figure 3.2.

```
H={ grandfather(X,Y) :- father(X,Z), father(Z,Y).
    grandfather(X,Y) :- father(X,Z), mother(Z,Y).}
```

Figure 3.3: Hypothesis learned for the grandfather concept with ILP.

The ILP task can be viewed as a search process among all the hypothesis that can be produced. The search process of an ILP system is characterised by:

- *The search space*. The search space in ILP, called the *hypothesis space*, is determined by the language of the possible logic programs L. The selected language L, syntactically restricts the form of the possible induced logic programs. Additionally, since this *hypothesis space* is normally huge, ILP systems structure it sorting hypothesis by their generability to make the search process more efficient.
- *The search goals*. The search goals consist of finding a hypothesis satisfying some quality criterion, usually based on these metrics:
  - *The classification accuracy* that measures the ratio of learning examples correctly classified by the induced logic programs. An example is the function $f_{accuracy}(h) = \frac{p(h)}{p(h)+n(h)}$, where $p(h)$ and $n(h)$ are the number of positive and negative examples covered by the evaluated hypothesis.
  - *The classification coverage* which measures the amount of learning examples covered by the induced logic programs. An example is the function $f_{coverage}(h) = p(h) - n(h)$, where $p(h)$ and $n(h)$ are the number of positive and negative examples covered by the evaluated hypothesis.
  - *The information compression* which denotes the understandability of the learned hypothesis. A common measure is the length of the hypothesis. An example is the function $f_{compression}(h) = p(h) - n(l) - l(h)$,

[1] *A logic program* is a set of disjunctions of literals with maximum one positive literal each. This class of disjunction is called *Horn clause*

where $p(h)$ and $n(h)$ are the number of positive and negative examples covered by the hypothesis, and $l(h)$ is the number of literals in the hypothesis.

- *The search direction*. It can be either bottom-up or top-down. A bottom-up direction means a generalization process of the most specific hypotheses allowed by the language bias. This approach includes ILP strategies like the *least general generalization* implemented in the GOLEM system (Muggleton and Feng, 1990) or *inverse entailment* implemented in the PROGOL system (Muggleton, 1995a). A top-down direction means a specification process starting from the most general logic program. This last approach is basically an extension of the classical decision tree learning algorithms to the first-order logic and it is better suited for learning with noisy learning examples since the top-down search can be more easily guided by heuristics. One of the first programs to follow this approach was FOIL (Quinlan and Cameron-Jones, 1995).

- *The search algorithms*. Best-First search is the desired search strategy, but due to the large size of the search spaces, greedy strategies such as hill-climbing or beam-search algorithms have to be implemented.

- *The heuristics*. In the case of a top-down search, an accuracy measure can be directly used to guide the search. In the case of the bottom-up search, heuristics needs to also measure the information compression offered by the hypothesis.

- *The pruning methods*. An structured hypothesis space allows one to prune unpromising hypothesis by rejecting the hypothesis that inherit some undesired properties of their ancestors such as covering a negative example or not covering a given positive one. For example, if we know that a given hypothesis covers an undesired negative examples we can reject all the generalizations of the hypothesis. And in the same way, if we know that a given hypothesis does not cover a desired positive example, we can reject all the specializations of this hypothesis.

ILP has traditionally being considered as a binary classification task for the given target concept. However, in the recent years, ILP techniques have broadened to cover the whole spectrum of ML task such as regression, clustering or association analysis. Particularly there is a new trend in ILP that lies on extending the classical propositional ML algorithms to the relational framework as for instance:

Learning relational trees

A very 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 value for the target concept that fits the examples that satisfy 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. In a similar way, the algorithms for learning regression trees (Karalic, 1992) have also been extended to the relational setting (Kramer, 1996). Figure 3.4 shows the relational decision tree for the concept of `grandfather(X,Y)`.

```
grandfather(X,Y,Class).
father(X,Z)?
+--yes: father(Z,Y)?
|       +--yes: [Class=true]
|       +--no: mother(Z,Y)?
|              +--yes: [Class=true]
|              +--no: [Class=false]
+--no: [Class=false]
```

Figure 3.4: Relational decision tree for the concept of `grandfather(X,Y)`.

Relational instance based learning

Unlike tree learning methods that explicitly construct a classifier from the learning examples, these techniques perform what is called *lazy learning*, i.e., they simply store the learning examples and delay the generalization until the moment a new example has to be classified. One disadvantage of these methods is that the cost of classifying an instance can be high because it implies all the generalization computations. On the other hand the generalization of these methods is local to the neighborhood of the new example which allows one to achieve better performance when the classification function is very complex.

Relational instance based learning methods include the relational versions of propositional instance based methods such as nearest neighbors techniques or locally weighted regressions and also the Case-Based Reasoning (CBR) techniques that make use of relational representations. The generalization in all these methods is done by analyzing instances similar to the new query instance ignoring the ones that are very different. Thereby a critical issue is the definition of an accurate

similarity measure among the instances described in predicate logic (Sebag, 1997; Ramon and Bruynooghe, 2001).

Relational kernels methods

Kernel methods require a positive definite covariance function to be defined between the learning examples. Given that learning examples are described in predicate logic, kernels for structured data have to be used, such as the convolution kernel (Haussler, 1999) or kernels defined on graphs (Gartner et al., 2003b). Because Gaussian processes are a Bayesian technique they provides more information that just a classification prediction. They also give an indication of the expected accuracy of this prediction.

3.2.2 Explanation Based Learning

Explanation Based Learning (EBL) deals with building explanations for justifying why a subset of learning examples merit their assigned target concept. Unlike ILP, EBL systems build these explanations making use of these two elements: A domain theory that validates the correctness of the conjectured explanations and a set of learning examples that statistically evaluate the coverage of the conjectured explanations. Formally the inputs to an EBL system are:

- A representation language L specifying the syntactic constraints on the predicate definitions.
- A set E of ground learning examples of the target concept.
- The current domain theory D consisting of background knowledge that can be used to explain the set of learning examples. This knowledge can be incomplete or should be redefined in other terms.

The EBL task is formally defined as finding a hypothesis H such that H is complete and consistent with the learning examples E and the current domain theory D. Unlike ILP, the EBL task reformulates the current domain theory D with the learned hypothesis H to improve the effectiveness of D in subsequent explanations.

3.2.3 Case Based Reasoning

Case Based Reasoning (CBR) memorizes cases (past experiences) in order to assist in the solution of future problems. Examples of CBR are: a mechanic who fixes a car by recalling another car that exhibited similar symptoms, a lawyer who defends a particular client based on legal precedents, a cook who prepares chocolate muffins for the first time recalling when he made plain muffins.

In general terms, a case can be described as the tuple *C=(Problem, Solution, Effects)* where *problem* is the problem solved in a past episode, *solution* is a description of the way the *problem* was solved and *effects* is a description of the

result of applying the *solution* to the *problem*. The CBR process is formalized as a four-step reasoning process:

1. *Retrieve*. Given a target problem, first CBR retrieves from memory the cases that are relevant for solving it. In the previous *muffins* example, this step retrieves the significant experience of *making plain muffins*.

2. *Reuse*. At this step CBR maps the solution from the previous case to the target problem. This may involve adapting the solution as needed to fit the new situation. In the *muffins* example, the retrieved solution must be adapted to include the addition of chocolate.

3. *Revise*. After mapping the previous solution to the target situation, this step tests the new solution in the real world and, if necessary, revises it. In the *muffins* example, if the cook fails with the adapted recipe, for instance, he added too few chocolate, he has to revise it.

4. *Retain*. Finally, once the solution has been successfully adapted to the target problem, the resulting experience is stored as a new case in memory for new demands.

CBR accepts anecdotal evidence as its main operating principle: it is not based on statistical data, like ILP and it is not based on a domain theory, like EBL. Therefore an important drawback of CBR is that there is no guarantee that the CBR generalization is correct.

3.3 Learning planning search control

Results of the IPC-2002 evidenced that planners using hand-coded search control performed orders of magnitude faster than the rest of planning systems. But hand-coding this knowledge is frequently very tricky because it implies expertise on of both the planning domain and the planning system. This section revises the different systems that automatically learn search control knowledge to improve the speed and/or the quality of the planning processes. On the one hand, the speed-up techniques are based on making the planner avoid unpromising portions of the search space without exploring it. On the other hand, the quality improvement methods are based on biasing the search process towards the kind of solutions preferred by the user.

3.3.1 Learning macro-actions

Macro-actions are the first attempt to speed-up the planning process. They are extra actions added to a given domain theory resulting from combining the actions that

are frequently used together. Figure 3.5 shows an example of the macro-action *UNLOAD-DROP* induced by the MacroFF system for the *Depots* domain.[2]

```
(:action UNLOAD
  :parameters (?x - hoist ?y - crate ?t - truck ?p - place)
  :precondition (and (in ?y ?t) (available ?x) (at ?t ?p)
                     (at ?x ?p))
  :effect (and (not (in ?y ?t)) (not (available ?x))
               (lifting ?x ?y)))

(:action DROP
  :parameters (?x - hoist ?y - crate ?s - surface ?p - place)
  :precondition (and (lifting ?x ?y) (clear ?s) (at ?s ?p)
                     (at ?x ?p))
  :effect (and (available ?x) (not (lifting ?x ?y)) (at ?y ?p)
               (not (clear ?s)) (clear ?y) (on ?y ?s)))

(:action UNLOAD-DROP
  :parameters (?h - hoist ?c - crate ?t - truck ?p - place
               ?s - surface)
  :precondition (and (at ?h ?p) (in ?c ?t) (available ?h)
                     (at ?t ?p) (clear ?s) (at ?s ?p))
  :effect (and (not (in ?c ?t)) (not (clear ?s))
               (at ?c ?p) (clear ?c) (on ?c ?s)))
```

Figure 3.5: Macro-action induced by MacroFF for the *Depots* domain.

The use of macro-actions reduces the depth of the search tree. However, this benefit decreases with the number of new macro-actions as they enlarge the branching factor of the search tree causing the *utility problem*.[3] To decrease this negative effect filters deciding the applicability of the macro-actions should be defined.

Since the beginning of AP the use of macro-actions has been widely explored. Traditionally most of the techniques use an off-line approach to generate and filter macro-actions before using them in search. Early works on macro-actions began with a version of the STRIPS planner (Fikes et al., 1972). It used previous solution plans and segments of the plans as macro-actions to solve subsequent problems. Later, MORRIS (Korf, 1985) extended this approach by adding some filtering heuristics to prune the generated set of macro-actions. This approach distinguished between two types of macro-actions: S-macros that occur frequently

[2]The *Depots* domain was created for IPC-2002 combining two classic domains of AP, the *Logistics* and the *Blocksworld*. In this domain trucks can transport crates around locations. Once crates are at their destination locations, hoists stack them on pallets. So the stacking problem is like a *Blocksworld* problem with multiple hands. Besides, trucks can also behave like tables, since the pallets on which crates are stacked are limited.

[3]The utility problem (Minton, 1988) comes out when the cost of using learned knowledge overtakes its benefit. This problem is due to the difficulty of storing the learned knowledge and the difficulty of determining which knowledge is useful for a particular problem.

during search and T-macros, those that occur less often, but model some weakness in the heuristic. The REFLECT system (Dawson and Silklossly, 1977) took the alternative approach of forming macro-actions based on pre-processing of the domain. All sound pairwise combinations of actions were considered as macro-actions and filtered through some basic pruning rules. Due to the small size of the domains with which the planner was reasoning, the number of macro-actions remaining following this process was sufficiently small to use in planning.

More recent works on macro-actions include the IPC-2004 competitor MacroFF (Botea et al., 2005). In this work macro-actions are extracted in two ways: from solution plans and by the identification of statically connected abstract components. Besides, an off-line filtering technique is used to prune the list of macro-actions. Marvin (Coles and Smith, 2007) also competed in IPC-2004 using action-sequence-memorization techniques to generate macro-actions on-line that allow the planner escape from plateaus without any exploration. Recently, (Newton et al., 2007) uses a genetic algorithm to generate a collection of macro-actions independent of the source planner.

3.3.2 Learning planning cases

Planning cases are stored and indexed as cases for later retrieval. When a new problem is presented, a case-based reasoner searches through its plans library for similar problems. If an exact match is found, the previous plan may be returned with no changes. Otherwise, the reasoner must either try to modify a previous case to solve the new problem or to plan from scratch. Figure 3.6 shows planning cases learned by the CABALA (de la Rosa et al., 2007) system for the *Depots* domain.

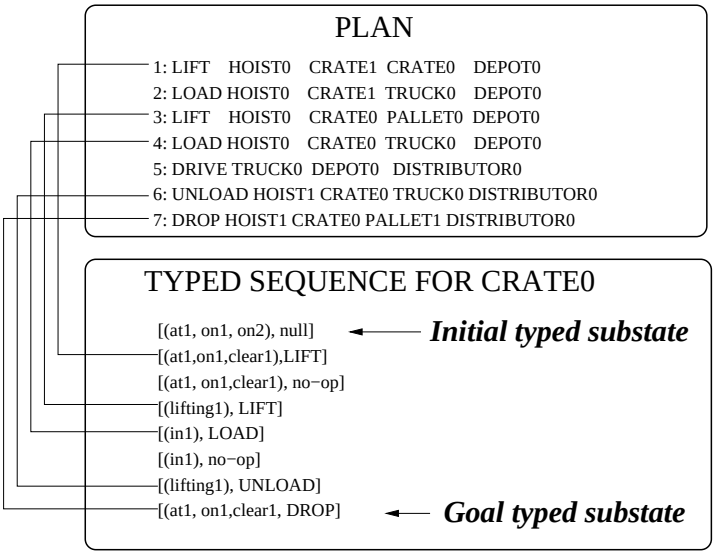

Figure 3.6: Cases learned by the CABALA system for the *Depots* domain.

The main drawback of this approach is finding an appropriate similarity metric between the planning problems, determining how to modify an existing plan to solve a new problem, and determining when it would be faster simply to plan for the new problem from scratch. As with macro-actions and control rules, utility is also a problem when handling libraries of plan cases. As the libraries get larger, the search times for relevant cases can exceed the time required to plan from scratch for a new case. But, unlike macro-actions, cases can memorize goals information.

By developing the first case-based planner (CHEF), Hammond helped to define the case-based approach to problem solving and to explanation (Hammond, 1990). Given a set of goals and a current situation, the first task for CHEF was to find an old plan that solved a past problem that is similar to the current problem. The next tasks were to adapt the old solution to fit the new circumstances and to store the new solution so that it can be reused in the future. However in addition to old plans, Hammond illustrated the use of memory for plan adaptation, plan repair, and failure anticipation. The modification strategies of CHEF do not consider the internal causal dependency structure of the plan, and thus can lead to incorrect plans even relative to the domain knowledge contained in the case base and the modifier. The PRIAR system (Kambhampati and Hendler, 1992) proposed to integrate the modification of plans with generative planning, so the modifications of plans are guided by the causal dependencies of the plan being modified, rather than by execution-time failures or by the results of external simulation.

PRODIGY/ANALOGY (Veloso and Carbonell, 1993) introduced the application of derivational analogy and abstraction to planning. This system stored planning traces to avoid failure paths in future exploration of the search tree. To retrieve similar planning traces, PRODIGY/ANALOGY indexed them using the minimum preconditions to reach a set of goals. The cased-based planning system PARIS (Bergmann and Wilke, 1996) proposed the introduction of abstraction techniques to store the cases organized in a hierarchical memory. This technique improves the flexibility of the cases adaptation, thus increasing the coverage of a single case.

The DsPlanners (Winner and Veloso, 2003) were introduced by Winner and Veloso with the aim of avoiding the cost of maintaining exhaustive plans cases databases. A DsPlanner is an automatically generated domain-specific planning program. This domain-specific planners are generated in a two-step process: (1) converting example plans into DsPlanners and (2) merging DsPlanners. The first step is performed by choosing the first parameterization that allows part of the solution plan to match that of a previously-saved DsPlanner. The second step is performed by searching if it is a sub-plan of the previously stored, or they overlap. If such a match is found, the two DsPlanners are combined. If no match is found, the plan is simply added to the end of the DsPlanner.

Recently, the CABALA (de la Rosa et al., 2007) system used typed sequences cases in heuristic planning for node ordering during the plan search.

3.3.3 Learning control rules

A control rule is an IF-THEN rule for guiding the exploration of the planning search tree. Control rules can guide the exploration in two different ways: proposing node pruning or proposing node ordering during the tree exploration. Figure 3.7 shows an example of a control rule for node pruning learned by the PRODIGY planner in the *Depots* domain.

```
(control-rule INDUCED-SELECT-UNLOAD
  (if (and (current-goal (lifting <hoist1> <crate1>))
           (true-in-state (clear <crate1>))
           (true-in-state (on <crate1> <surface1>))
           (true-in-state (available <hoist1>))
           (true-in-state (available <hoist2>))
           (true-in-state (at <truck1> <depot1>))
           (some-candidate-goals nil)
           (type-of-object <hoist1> hoist)
           (type-of-object <hoist2> hoist)
           (type-of-object <crate1> crate)
           (type-of-object <truck1> truck)
           (type-of-object <depot1> depot)
           (type-of-object <surface1> surface)))
  (then select operators unload))
```

Figure 3.7: Control rule for the *Depots* domain.

Control-rules usually enrich the planning language with extra predicates, called metapredicates,[4] that allow the planner to capture specific knowledge of the given domain. However, the knowledge captured strongly depends on the quality of the learning examples used. When the learning examples are not significant enough, the induced control rules may mislead the search process of the planner. Besides control-rules also suffer from the utility problem.

On the one hand, there are systems that inductively learn control rules. Among these systems Inductive Learning Programming (ILP) is the most popular learning technique. The GRASSHOPPER system (Leckie and Zukerman, 1991) used FOIL (Quinlan and Cameron-Jones, 1995) to learn control rules that guide the PRODIGY planner when selecting goals, operators and binding operators. Purely inductive systems need a big number of learning examples to acquire efficient control knowledge. On the other hand, there are analytical systems: PRODIGY's EBL module (Minton, 1988) learned search control rules for the PRODIGY planner from a few examples of correct and wrong decisions. STATIC (Etzioni, 1993) obtains control rules without any learning example just using Explanation Based Learning (EBL) to analyse the relations between actions' preconditions and effects.

[4] Metapredicates are extra predicates used to reason about the metastate of the planner (e.g., the goals that the planner is currently working on, the operators being considered, etc.).

The main disadvantage of these methods is that, given that there are no learning examples, there is no measure of the learned knowledge utility.

With the aim of solving the problems of the purely inductive and purely analytical approaches some researchers have tried to combine them: the pioneering systems based on this principle are LEX-2 (Mitchell et al., 1982) and META-LEX (Keller, 1987). AXA-EBL (Cohen, 1990) combined EBL + induction. It first learns control rules with EBG and then refines them with learning examples. DOLPHIN (Zelle and Mooney, 1993) was an extension of AXA-EBL which used FOIL as the inductive learning module. The HAMLET (Borrajo and Veloso, 1997) system combines deduction and induction incrementally. First it learns control rules with EBL, that usually are too specific and then uses induction to generalize and correct the rules. EVOCK (Aler et al., 2002) applies the genetic programming paradigm to solve the learning problems caused by the hypothesis representation language.

3.3.4 Learning generalized policies

A *policy* is a mapping between the world states and the preferred action to be executed in order to achieve a certain set of goals. A *generalized policy* is a mapping from the problems of a given domain, i.e., the diverse combinations of initial state and goals, into the preferred action to be executed in order to achieve the goals. Thereby, a good generalized policy is able to solve all the possible problems instances of a given domain. For example, a generalized policy for the *Blocksworld* can be given as follows:

```
(1) put all blocks on the table, and then
(2) move block x on block y when x should be on y,
    both blocks are clear, and y is well placed
```

Figure 3.8: A generalized policy for the *Blocksworld* domain.

The problem of learning generalized policies was first studied by Roni Khardon. Khardon's L2ACT (Khardon, 1999), induced generalized policies for both the *Blocksworld* and the *Logistics* domain by extending the decision list learning algorithm (Rivest, 1987) to the relational setting. This first approach presented two important weaknesses: (1) it relied on human-defined background knowledge that expressed key features of the domain, e.g. the predicates `above(block1,block2)` or `in_place(block)` for the Blocksworld, and (2) the learned policies do not generalize well when the size of the problems is increased. Martin and Geffner solved these weaknesses of the Khardon's approach in the *Blocksworld* domain by changing the representation language of the generalized policies (Martin and Geffner, 2000). Instead of representing the current state and the problems goals using predicate logics they used a concept language.[5] This representation allows

[5] *Concept Languages* also known as *Description Logics* (Brachman and Levesque, 1984; Geffner,

them to learn rules of the form: *apply action type a to any object in class C*.

In the last years the scope of generalized policy learning has been augmented over a diversity of domains making this approach competitive with the state-of-the-art planners. This achievement is due to the assimilation of two new ideas: (1) The policy representation language is enriched with extra predicates. Like control-rule based planners, these systems introduce metapredicates to capture more effective domain specific knowledge. (2) The learned policies are not longer greedily applied but combined with heuristic planning algorithms. Specifically (Yoon et al., 2007b) complements the information of the current state with the relaxed planning graph and the learned policies are used during node expansions in a best-first search heuristic algorithm. At each node expansion of the best-first search, they add to the search queue the successors of the current best node as usual, but also they add the states encountered by following the policy from the current best node for a given horizon.

Recently the Roller system (de la Rosa et al., 2008, 2011) defined the problem of learning generalized policies as a two-step standard classification process. At the first step the classifier captures the preferred operator to be applied in the different planning contexts. At the second step the classifier captures the preferred bindings for each operator in the different planning contexts of a given domain. These contexts are defined by the set of helpful actions extracted from the relaxed planning graph of a given state, the goals remaining to be achieved, and the static predicates of the planning task. Additionally, Roller implemented two methods for guiding the search of a heuristic planner with the learned policies. The first one consists of using the resulting policy in a Depth First Search algorithm. The second one consists of ordering the node evaluation of the Enforced Hill Climbing algorithm with the learned policy.

3.3.5 Learning hierarchical knowledge

Hierarchical planning combines hierarchical domain-specific representation of the planning models together with domain-independent search for problem solving. One of the best-known approaches for modelling hierarchical knowledge about a planning domain is Hierarchical Task Network (HTN). Current HTN planners can outperform the state-of-the-art ones and provide a natural modelling framework in many real-world applications like fire extinctions (Castillo et al., 2006), evacuation planning (Muñoz-Avila et al., 1999), manufacturing, or autonomous vehicles management. In HTN planning, complex tasks are decomposed into simpler tasks until a sequence of primitive actions is generated. Therefore, the input to an HTN planner includes a set of operators similar to the ones used in classical planning (primitive actions) and a set of methods describing how tasks should be decomposed into subtasks in this particular domain. Figure 3.9 shows the method for the

1999; Donini et al., 1995) have the expressive power of fragments of standard first-order logic but with a syntax that is suited for representing and reasoning with classes of objects.

task `transport-crate(crate,destination)` from the HTN description of the *Depots* domain.

```
(:method (transport-crate ?crate ?destination)
  ; precondition
  (and (truck ?truck) (at ?truck ?place1)
       (at ?crate ?place2) (different ?place1 ?place2))
  ; subtasks
  (:ordered (drive ?truck ?place1)
            (load ?crate ?truck)
            (drive ?truck ?place2)
            (unload ?crate ?truck)))
```

Figure 3.9: Method for hierarchical planning in the *Depots* domain.

The specification of these hierarchical methods by hand is a hard task that requires expert knowledge. And defining a general algorithm for automatically learning them for any domain is still an open issue. However there is existing work about learning task decompositions of related tasks and subtasks.

The ALPINE system (Knoblock, 1990) completely automates the generation of abstraction hierarchies from the definition of a problem space. Each abstraction space in a hierarchy is formed by dropping literals from the original problem space; thus it abstracts the preconditions and effects of operators as well as the states and goals of a problem. Concerning abstraction in planning, the system PARIS stores each case in different levels of abstraction (Bergmann and Wilke, 1996). To solve new problems, the problem will be compared with cases of the hierarchy of abstractions and the planner is used to refine the abstract case and to adapt it to the new problem. X-LEARN (Reddy and Tadepalli, 1997) uses a generalize-and-test algorithm based on ILP to learn goal-decomposition rules. These (potentially recursive) rules are 3-tuples that tell the planner how to decompose a goal into a sequence of subgoals in a given world state, and therefore are functionally similar to methods in HTN domains. X-learns training data consists of solutions to the planning problems ordered in an increasing order of difficulty (authors refer to this training set as an exercise set, as opposed to an example set which is a set of random training samples without any particular order). This simple-to-hard order in the training set is based on the observation that simple planning problems are often subproblems of harder problems and therefore learning how to solve simpler problems will potentially be useful in solving. HICAP (*Hierarchical Interactive Case-based Architecture for Planning*) (Muñoz-Avila et al., 1999) integrates the SHOP hierarchical planner together with a case-based reasoning (CBR) system named NACODAE to assist with the authoring of plans for noncombatant evacuation operations. It interacts with users to extract a stored similar case that allows one to solve the current problem. The system CASEADVISOR also integrates CBR and hierarchical planning (Carrick et al., 1999). It uses plans previously stored to

obtain information of how to solve a task instead of choosing the refining method. The KNOMIC (Knowledge Mimic) (van Lent and Laird, 2001) is a machine learning system which extracts hierarchical performance knowledge by observation to develop automated agents that intelligently perform complex real-world tasks. The knowledge representation learned by KNOMIC is a specialized form of Soar's production rule format (Rosenbloom et al., 1993). The rules implement a hierarchy of operators with higher level operators having multiple sub-operators representing steps in the completion of the high level operator. These rules are then used by an agent to perform the same task. Langley and Rogers describes how ICARUS, a cognitive architecture that stores its knowledge of the world in two hierarchical categories of concept memory and skill memory, can learn these hierarchies by observing problem solving in sample domains (Langley and Choi, 2006).

There is a recent trend of work that attempts to solve the learning of HTN domain descriptions. CAMEL (Ilghami et al., 2005) uses an extended version of the Candidate Elimination incremental algorithm to learn expansions methods in a HTN planner by observing plan traces. It is designed for domains in which the planner is given multiple methods per task, but not their preconditions. That is, the structure of the hierarchy is known in advance and the learning task is to identify under what conditions different hierarchies are applicable. The problem with this work is that it required very many plan traces to converge (completely determine the preconditions of all methods). CAMEL++ is also an algorithm for learning preconditions for HTN methods that enables the planner to start planning before the method preconditions are fully learned. By doing so, the planner can start solving planning problems with a smaller number of training examples than is required to learn the preconditions completely with insignificant cost of few incorrect plans. Camel required all information about the methods except for their preconditions to be given to the learner in advance, so that the only information for the learner to learn was the methods preconditions. Besides, each input plan trace for Camel needed to contain a lot of information so that the learner could learn from it. At each decomposition point in a plan trace, the learner needed to have all the applicable method instances, rather than just the one that was actually used. The HDL algorithm (Ilghami et al., 2006) starts with no prior information about the methods. HDL does not need most of that information. At each decomposition point, it only needs to know about one or at most two methods: The method that was actually used to decompose the corresponding task; and one (if there are any) of the methods that matched that task but whose preconditions failed (to serve as a negative training sample). The system HTN-Maker (Hogg et al., 2008) receives even less input information, HTN-Maker is able to produces an HTN domain model from a STRIPS domain model, a collection of STRIPS plans, and a collection of task definitions.

Apart from learning decomposition task methods, learning has also been applied to hierarchical planning for other purposes. Lotem and Nau build a method for extracting knowledge on HTN planning problems for speeding up the search (Lotem and Nau, 2000). This knowledge is gathered by propagating properties through

an AND/OR tree that represents disjunctively all possible decompositions of an HTN planning problem. This knowledge is used during the search process of the GraphHTN planner, to split the current refined planning problem into independent subproblems.

3.3.6 Learning heuristic functions

A heuristic function $H(s; A; g)$ is a function of a state s, an action set A, and a goals set g that estimates the cost of achieving the goals g starting from s and using the actions of A. In case the estimation provided by the heuristic function is accurate, a greedy application of the actions recommended by the heuristic will achieve the goals without search. However, when a heuristic is imperfect, it must be used in the context of a heuristic search algorithm, where the accuracy of the heuristic impacts the search efficiency.

Most of the state-of-the-art planners are based on heuristic search over the state-space (12 over the 20 IPC-2006 participants). The performance of these planners depend strongly on defining good domain-independent heuristic functions that provide good guidance across the wide range of different domains. Currently these heuristics are built by approximating the reachability of the goals by ignoring some types of interactions among actions (usually, the delete effects of actions). Though they are effective in general they are expensive to compute and there are many domains where these functions misestimate the distance to the goal leading to poor guidance. As a consequence, heuristic planners suffer from scalability problems. This effect becomes more prominent in domains where the heuristic function is less accurate because, in these domains heuristic planners spend most of the planning time computing useless node evaluations, e.g. *Blocksworld*. With the aim of avoiding this undesirable effect one can try to directly learn the heuristic function for a given domain: Regarding this approach, (Yoon et al., 2006; Xu et al., 2007) build a state generalized heuristic function through a regression process. The regression examples consists of observations of the true distance to the goal from diverse states, together with extra information from the relaxed planning graph. The obtained heuristic function provide more accurate estimations that capture domain specific regularities expressed as a weighted linear combination of features, that is,

$$H(s; A; g) = \sum_i w_i * f_i(s; A; g)$$

where the w_i are the weights and the f_i represent the different features of the planning context. The main disadvantage of this leaning approach is that the result of the regression is poorly understandable by humans making the verification of the correctness of the acquired knowledge difficult.

3.4 Learning planning domain models

AP is a form of problem solving that requires accurate models of the dynamics of the environment to reason about problems. However, building these models from scratch is difficult and time-consuming, even for AP experts. This section revises techniques for the automatic definition of planning action models in *deterministic and fully observable environments*.

The LIVE system (Shen and Simon, 1989) is an extension of the GPS framework (Ernst and Newell, 1969) with a learning component. LIVE alternates 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. Once new rules are learned, problem solving is resumed and a solution plan may be constructed through means-ends analysis.

The EXPO system (Gil, 1992) refines *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. The diverse differences are correlated with a typical cause for the expectation failure. EXPO constructs a set of specific hypotheses to fix the detected difference. After being heuristically filtered, each hypothesis is tested in turn with an experiment. After the experiment is designed, a plan is constructed to achieve the situation required to carry out the experiment. The experiment plan must meet constraints such as minimizing plan length and negative interference with the main goals.

Unlike the previous works that refined planning operators by an active exploration of the environment, OBSERVER (Wang, 1994) learns operators by observing expert agents. The observations of the expert agent consists of: (1) the sequence of actions being executed, (2) the state in which each action is executed, and (3) the state resulting from the execution of each action. Planning operators are learned 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) learns 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 receives a goal input given that LOPE creates its own goals from the situations

with the highest utility.

Despite the previous techniques were based on inductive learning, analytical learning has also been used for action modelling. Specifically, EBL has been used for learning low-level operators as encapsulated control loops that are specialized to best fit a particular distribution of observed learning problems (Levine and DeJong, 2006). Finally, the techniques previously revised for the automatic definition of HTN decomposition methods (Section 3.3.5) are also considered as action modelling techniques.

3.5 Discussion

Before the mid 90's ML was exhaustively used in AP to learn search control knowledge that improved the scalability of planners. During this period planners implemented uninformed search algorithms, so ML made planners achieve better performance in many domains.

In the late 90's two factors made the planning community decrease its interest in ML. On the one hand, the appearance of powerful domain independent heuristics boosted the performance of planners. Suddenly the baseline for evaluating the performance of the learning-based planners shifted dramatically. On the other, the existing relational learning algorithms were inefficient and performed poorly over a diversity of domains.

At the present time, encouraged by the applications of AP to real-world problems and the maturity of relational learning, there is a renewed interest in learning for planning. In fact, in 2005 the International Competition on Knowledge Engineering for Planning Systems (ICKEPS) was created and in 2008, a learning track took place for the first time at IPC. ML seems again to be the solution to current challenges of AP ranging from learning control knowledge for large planning problems (Yoon et al., 2008) to learning action models for environments with unknown dynamics (Yang et al., 2007). Nevertheless, deeper studies have still to be done in different issues for providing planning systems with complete learning skills:

- **There is no general method for generating good quality learning experience for planning**. The effectiveness of the learning-based planners strongly depends on the training examples used. In most systems these learning examples are collected solving small problems from the same domain. Therefore, the quality of the learning examples will depend on the selection of the problems used for training. Traditionally, these training problems are obtained by a random generator but this approach presents two limitations. First, guaranteeing that a given AP problem is solvable is as hard as solving it. Second, training problems are usually created by a random generator with a set of parameters that define the problems difficulty. Adjusting these parameters for generating significant learning examples implies domain expertise. Recent works discusses several active learning schemes to overcome

these limitations (Fern et al., 2004; Fuentetaja and Borrajo, 2006) however, further studies have to be done to come up with a general solution.

- **There is no evidence to extract the key features for a given planning domain**. The effectiveness of the learned knowledge strongly depends on the set of features chosen for describing the learning examples and the target concept. If the chosen features are not able to capture the relevant knowledge of a given domain, the learning algorithms will produce useless knowledge.

- **The on-line learning of planning knowledge has not been studied in depth**. To completely integrate the planning and learning processes farther studies in on-line learning search control and action models have to be done. These studies, must cover autonomous generation of experience and planning with incomplete and/or incorrect models. These issues have already been extensively studied in other related AI areas such as Reinforcement Learning.

- **There is no effective ML for the most successful domain-dependent planners**. The planner TLPlan (Bacchus and Kabanza, 2000) guided a forward state-space search with hand-written control rules represented in linear temporal logic. The planner SHOP2 (Nau et al., 2001) used human-coded task-decomposition rules for HTN planning. Given sufficient control knowledge, these planners generate plans orders of magnitude faster than domain independent planners. Learning this kind of control knowledge for a variety of domains is still an open issue.

- **There is no standard methodology to evaluate the performance of the planning and learning systems**. At IPC-2008, an evaluation of learning-based planners over a diversity of domains has been carried out for the first time. However, there are now neither standard metrics nor representation languages that facilitates the comparison of the diverse learning-based techniques.

- **Most of learning-based planners do not improve the overall performance of the best non-learning planners**. As shown in IPC-2008 the state-of-the-art domain independent planners outperform learning-based planners over a diversity of domains. The main reason is that learning can often degrade performance. Current learning-based planners need to improve their robustness to poor or wrong learning in order to be more competitive.

- **In practice, the model learning proccess may not achieve perfect action models**. Learning examples may be faulty or insufficient, ML algorithms can get stuck in local minima, ... Regarding this situations, Kambhampati recently introduced the concept of *model-lite planning* (Kambhampati, 2007). This concept refers to a new research line for developping planning techniques that (1) cope with incompletely specified domain models and pro-

vides plans according to the knowledge available and/or (2) automatically improves the domain models with time and experience.

Chapter 4

Planning under uncertainty

This chapter reviews the different PUU paradigms. The review is organized according to two dimensions: *observability of the state* and *determinism of the action models*.

4.1 Introduction

The usefulness of classical planning for real-world problems is limited given that almost never the whole truth about the environment is known. More specifically, there are typically two different sources of uncertainty:

1. *Action model*. In many planning problems one cannot assume deterministic world dynamics. For instance, planning domains that include stochastic procedures such as the toggling of a coin or the rolling of a dice.

2. *Observability*. In many planning problems handling a complete and perfect description of the state of the environment is unconceivable. For example when planning for an underwater robot or a Mars rover.

PUU studies the extensions of the classical planning framework to develop planning systems able to deal with *non-deterministic action models*, i.e., able to generate plans reasoning about actions with diverse potential outcomes and able to deal with *partial observability of the environment*, i.e., able to generate plans in spite of the limited information of the current state available. Figure 4.1 shows the diverse PUU paradigms according to how they extend the classical planning framework.

4.2 The conformant planning task

Conformant planning is the task of solving a planning problem in a non-observable environment. Conformant planners synthesize sequences of actions that safely achieve the goals, starting from an undetermined initial state with no sensing during

	ACTION MODEL	
OBSERVABILITY	**Deterministic**	**Stochastic**
Full	*Classical*	*Probabilistic*
Partial	*Contingent*	*Contingent Probabilistic*
None	*Conformant*	*Conformant Probabilistic*

Figure 4.1: Planning under uncertainty paradigms.

the course of the action execution. Figure 4.2 shows an example of a conformant planning problem consisting of a robot navigation in a grid. In this example the robot has to reach cell `D2` starting from cell `A5` and avoiding the obstacles. In this case, as the robot is not able to perceive the contents of cells C2,C3, E3 and E4 it has to avoid them to generate a conformant solution.

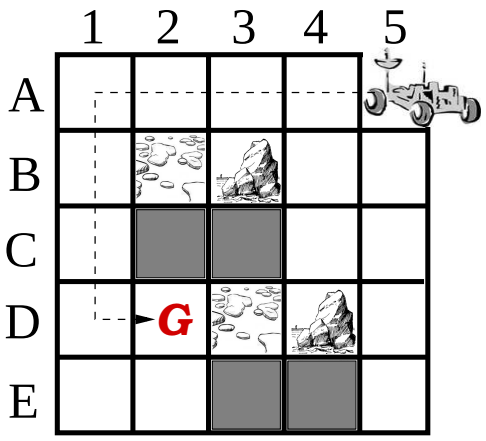

Figure 4.2: Example of a conformant planning problem.

4.2.1 The conceptual model

The conformant planning task is modelled as a deterministic *belief state* transition system $\sum = (B, A, C)$ where:

- B is the set of belief states. Conformant planning algorithms do not handle single states but belief states. A belief state is the set of all the states the system could possibly be at a given instant.

- A is the set of actions. Each action $a \in A$ is a function that maps a belief state $b \in B$ into another belief state $b' \in B$. The set of actions $A(b) \subseteq A$ that can be safely applied in a belief state $b \in B$ are the actions $a \in A$ that

can be applied in any state s that is possible according to b

$$A(b) = \{a \mid Pre(a) \subseteq s, \forall s \in b\}$$

The belief state $b' \in B$ resulting of applying action $a \in A$ in the belief state $b \in B$ is described as:

$$b' = \{s' \mid s' \in result(a, s), \forall s \in b\}$$

- $C(a)$ represents the cost of applying the action $a \in A$.

Regarding this conceptual model, a conformant planning problem is defined as the tuple $P = (\sum, b_0, G)$ where $b_0 \in B$ is the initial belief state, i.e., the set of possible initial states. And $G \subseteq B$ is the set of belief states that contain only goal states. Finding a solution to a conformant planning problem P consists of synthesizing a sequence of actions $(a_1, a_2, ..., a_n)$ corresponding to a sequence of belief-states transitions $(b_0, b_1, ..., b_n)$ such that b_i results from executing the action a_i in the belief-state b_{i-1} and b_n is a belief-state that contains only goal states. The optimal solution to a conformant planning problem is the one that minimizes the expression $\sum_{i=0}^{n} c(a_i)$.

4.2.2 The representation language

Conformant planning problems are described using a standard classical planning representation language augmented for specifying the set of possible initial states of the problem. At IPC, the classical planning representation language PDDL is augmented with the predicate $(oneof\ l_1\ l_2 \ldots\ l_n)$ for expressing that exactly one of the l_i literals is true in the initial state and the predicate $(or\ (not\ l_i)\ (not\ l_j))$ for expressing the mutual exclusion of the couple of literals l_i and l_j. Figure 4.3 shows the representation of a conformant problem from the *Blocksworld* domain of the conformant track of IPC.

4.2.3 The algorithms

There are three main approaches to tackle the conformant planning problem:

- Extending classical planning to handle belief states. The pioneer conformant planners followed this approach; particularly they tried to extend the planning-graph techniques to the conformant planning setting.

- Explicit search in the conceptual model. The problem of conformant planning can be addressed by directly searching in the belief-states space. To guide this search process, one can use heuristic functions based on computing a distance measure between belief states.

- Compiling the conformant planning problem into another form of problem solving for which there are efficient algorithms. The main compilations of the conformant planning task are:

```
(define (problem prob01)
 (:domain blocks)
 (:objects A B - block)
 (:init
  (and (oneof (handempty) (holding A) (holding B)) ;(holding ?x)
       (oneof (holding A) (clear A) (on B A))      ;(above A ?x)
       (oneof (holding A) (ontable A) (on A B))    ;(on A ?x)
       (oneof (holding B) (clear B) (on A B))      ;(above B ?x)
       (oneof (holding B) (ontable B) (on B A))    ;(on B ?x)

       (or (not (handempty)) (not (holding A)))
       (or (not (handempty)) (not (holding B)))
       (or (not (holding A)) (not (holding B)))

       (or (not (holding A)) (not (clear A)))
       (or (not (holding A)) (not (on B A)))
       (or (not (clear A)) (not (on B A)))

       (or (not (holding A)) (not (ontable A)))
       (or (not (holding A)) (not (on A B)))
       (or (not (ontable A)) (not (on A B)))

       (or (not (holding B)) (not (clear B)))
       (or (not (holding B)) (not (on A B)))
       (or (not (clear B)) (not (on A B)))

       (or (not (holding B)) (not (clear B)))
       (or (not (holding B)) (not (on A B)))
       (or (not (clear B)) (not (on A B)))

       (or (not (on A B)) (not (on B A)))))
 (:goal (and (ontable A) (on B A))))
```

Figure 4.3: Problem from the *Blocksworld* of the conformant track of IPC.

- SAT Problem. The conformant planning can be compiled into SAT following two different approaches:
  - Generate-and-test. Given a plan length, a SAT solver is used to generate candidate conformant plans. Then, each candidate is tested for conformant validity. This strategy only works well when there are few candidate plans; otherwise it is too inefficient.
  - Compiling the planning theory into a *deterministic-Decomposable Negated Normal Form* (d-DNNF). The compiled theory is transformed into a new theory over the action variables only, and finally the conformant plan, if there is one, is obtained from this theory by a single invocation of a SAT engine.
- Model Checking problem. In this compilation of the planning prob-

lem (Cimatti and Roveri, 2000), the planning domain is formalized as a specification of the possible models for plans; the conformant planning problem is solved by searching through plans, checking that there exists one plan p that satisfies two extra requirements: (1) p must be still applicable after executing any prefix of p in any of the possible initial states and (2) all the states resulting from the execution of p in the initial belief state must be goal states.

– Classical planning. Given KL to denote literals L that are *known to be true* and L/X to denote literals L that are true if a given literal X is true. This compilation (Palacios and Geffner, 2007) generates a classic planning problem by transforming conditions of the form $C \wedge X \rightarrow L$ into the form $KC \rightarrow L/X$ and merging these conditions according to the rule $L/X_1, L/X_2 \rightarrow KL$ provided that $X1, X2$ are known to hold.

4.2.4 The implementations

This is a enumeration of some of the most relevant conformant planners.

Conformant-Graphplan

CONFORMANT-GRAPHPLAN (CGP) (Smith and Weld, 1998). It expands separate planning graphs for each different possible state of the environment and one for each possible nondeterministic outcome. It keeps track of mutual exclusion relations among the different graphs and then it searches backwards for a plan that is valid in all the possible worlds.

Conformant-MBP

The Conformant Model Based Planner (CMBP) (Cimatti and Roveri, 2000) is based on the planning via model checking paradigm. It relies on Binary Decision Diagrams (BDDs) (Bryant, 1986) to compactly represent and efficiently search for a conformant plan in the space of possible plans.

Conformant-FF

CONFORMANT-FF (Brafman and Hoffmann, 2004) implements a forward heuristic search in the space of belief states guided by an extension of the FF's heuristic. The belief states are coded by the intersections of propositions contained in all the possible states of the belief state. The extension to the FF's heuristic uses SAT techniques to efficiently solve relaxed conformant planning tasks in the search states. The relaxed planning tasks are built by ignoring (1) all the deletes of the action effects (2) propositions of the conditional effects (all except one).

T0

The T0 planner (Palacios and Geffner, 2006) is the winner of the IPC-2006 conformant planning track. This planner maps conformant planning problems into deterministic problems which are then solved by the classical planner FF.

Approximation-Based Conformant Planner

The APPROXIMATION-BASED CONFORMANT PLANNER (CPA) (Tran et al., 2008) is the winner of the IPC-2008 conformant planning track. CPA implements a Best-First Search in the space of *partial states* instead of the space of belief states. The planner relies on the observation that a belief state can (sometimes) be replaced by the intersection of its members, thereby reducing the size of the search space. The search is guided by a combination of the following heuristics: the cardinality of the partial state, the number of satisfied subgoals and the *total sum heuristic* (a heuristic computed adding the heuristic values of completions of the *partial state*).

4.3 The contingent planning task

Contingent planning is the task of solving a planning problem in a environment where the current state is undetermined, but it is possible to observe some aspects of it during the execution of the plan. Contingent planning is also referred in literature as conditional planning. Figure 4.4 shows an example of a contingent planning problem. In this example a mobile robot has to plan its actions to navigate from cell `A5` to the goal cell `C2` avoiding obstacles. In this example the robot cannot initially determine the content of cells `C2`, `C3`, `D3` and `D4` but when it gets close to them, it can sense their contents and decide whether to cross `C2` and `C3`, if there is no obstacle, or to follow the path through `A1,A2,A3,A4` which is free of obstacles.

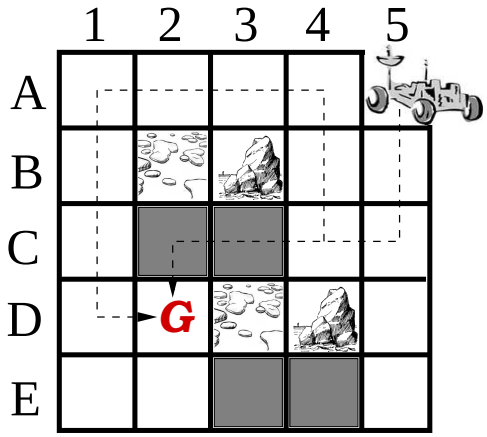

Figure 4.4: Example of a contingent planning problem.

4.3.1 The conceptual model

In contingent planning, part of the initial state of the environment is unknown. As a consequence, contingent planning does not reason about states, but about belief states. Additionally, the outcomes of the observations cannot be predicted thus, the contingent planning task is modelled as a non-deterministic belief-states-transition system $\sum = (B, A, C, O)$ where:

- B is the set of belief states. A belief state is the set of all the states the system could possibly be at a given instant.

- A is the set of actions. Each action $a \in A$ is a function that maps a given belief state $b \in B$ into a set of belief states. The set $A(b) \subseteq A$ of actions that can be safely applied in a belief state $b \in B$ are the actions $a \in A$ that can be applied in any state s that is possible according to b

$$A(b) = \{a \mid Pre(a) \subseteq s, \forall s \in b\}$$

- $C(a)$ is the cost of applying the action $a \in A$.

- $O(a, b)$ is the noise-free sensor model function. This function returns the possible observed states after applying the action a in the belief state b.

$$O(a, b) = \{o(a, s) \mid \forall s \in b\}$$

  where $o(a, s)$ is the observed state after applying the action a in the state s. After the execution of action $a \in A$, an observation o is performed to exclude from the resulting belief-state b^o the states inconsistent with o. Thus, the resulting belief state $b^o \in B$ is given by:

$$b^o = \{s' \in S \mid s' = o(a, s), s \in b\}$$

Regarding this conceptual model, a contingent planning problem is defined as the tuple $P = (\sum, b_0, G)$ where $b_0 \in B$ is the initial belief state, i.e., the set of possible initial states. And $G \subseteq B$ is the set of belief states that contain only goal states. The solution to a contingent planning problem is a conditional plan normally expressed as a *belief state*-action policy $\pi(b_{i-1}) = a_i$ whose execution starting from the initial belief-state b_0 results in a belief-state $b_n \in G$ which contains only goal states. The optimal solution to a contingent planning problems is the policy $\pi^*(b_{i-1}) = a_i$ that minimizes the sum of costs to reach a belief-state $b_n \in G$ from all the possible reachable states. A solution to a contingent planning problem can also be expressed as a decision tree where the internal nodes of the tree correspond to observations of the current state of the environment that is not available in plan-synthesis time.

4.3.2 The representation languages

Contingent planners extend the classical planning representation languages to capture two new features: (1) like in conformant planning, there is no initial state but a set of possible initial states and (2) the sensor model indicating the observations available at execution time. Normally these observations available are modelled in the planning domain as *sensing actions*. *Sensing actions* are planning actions with effects representing some information observed from the current state. Therefore, a planner should choose *sensing actions* when the lack of information can prevent it to achieve the problem goals. Currently there is no standard representation language for the sensing actions and normally each contingent planner defines its own representation. Figure 4.5 shows an example of a sensing action for the CNLP planner (Peot and Smith, 1992). In this action, `effect1` and `effect2` are mutually exclusive and cover all the possible outcomes for the observation.

```
(:observation test-robot-handempty
  pre: (unknown(handempty))
  effect1: (handempty)
  effect2: (not (handempty)))
```

Figure 4.5: *Sensing action* to check the state of the robot hand in the *Blocksworld*.

4.3.3 The algorithms

There are three main approaches for contingent planning:

- Extending classical planning algorithms for dealing with contingencies. The first contingent planners followed this approach.

- Explicit search in the conceptual model. Contingent planning can solved by AND/OR search algorithms able to deal with disjunctive transitions such as LAO* or the family of dynamic programming algorithms.

- Compiling the contingent planning problem into another form of problem solving. The size of a valid contingent plan constructed by searching in the conceptual model is exponential in the number of observations. Another approach to deal with this drawback is compiling the contingent planning problem in another form of problem solving for which there are effective algorithms. An example is the Conformant Planning compilation (Albore et al., 2007). This compilation transform the sensings in the contingent problem into a set of non-deterministic actions for a conformant planning problem.

4.3.4 The implementations

CNLP

The Conditional Non-Linear Planner CNLP (Peot and Smith, 1992) was one of the first planners that included sensing actions. CNLP is a conditional version of the Systematic Nonlinear Planner (SNLP) (Mcallester and Rosenblitt, 1991) that allowed to represent uncertain information in planning time using the predicate function `unknown`. CNLP used sensing actions to know if some unknown literal is true or false during execution time.

Cassandra

CASSANDRA (Pryor and Collins, 1996) does not attempt to construct a contingency plan until it encounters an uncertainty. In fact, if it encounters no uncertainty it constructs a solution plan in the same way as the classical planner UCPOP. Cassandra notices uncertainty when its current plan becomes dependent upon a particular outcome of an action. At this moment, the plan constructed by CASSANDRA becomes a plan branch for that outcome and CASSANDRA plans again to reach the problem goals assuming that the action produced a different outcome.

Sensory GraphPlan

Sensory GraphPlan SGP (Weld et al., 1998) is an extension of GRAPHPLAN to handle sensing actions. Like CONFORMANT-GRAPHPLAN (Smith and Weld, 1998), SGP expands separate planning graphs for each different possible state of the environment and one for each possible nondeterministic outcome. The difference is that the SGP expansion considers the set of possible sensor values and takes the cross product to determine the set of partitions which could be induced by the sensor. These partitions lead to the introduction of new propositions to the subsequent layers.

Contingent-FF

CONTINGENT-FF (Hoffmann and Brafman, 2005) extends the FF PLANNER with the ability to handle initial state uncertainty expressed in the form of a CNF formula. Contingent-FF implements a heuristic AO* search. The output of Contingent-FF are tree-shaped plans with branches.

MBP

MBP (Bertoli et al., 2006) is implemented on the top of the NuSMV (Cimatti et al., 1999) model checker. As CMBP, it relies on BDDs (Bryant, 1986) to compactly represent and efficiently search for a solution plan in the space of possible plans.

4.4 The probabilistic planning task

Probabilistic Planning is the task of finding a plan that reaches a set of goals in a stochastic environment. In stochastic environments, frequently, there is no plan that guarantees reaching the goals. So, probabilistic planners must reason about the likelihood of the actions' outcomes to obtain plans that (1) maximize the probability of reaching the goals and (2) balance the risk of producing undesirable states. Figure 4.6 shows an example of a probabilistic planning problem consisting of a robot navigation in a grid. In this problem the robot must reach the cell `D2` starting from the cell `A5`, avoiding the obstacles (cells `B2`, `D3` and `E4`) and considering that every movement has a known probability of reaching a wrong cell. For instance, moving the robot from cell `A5` to `B5` might result in the robot at wrong cells `A4` or `B4`.

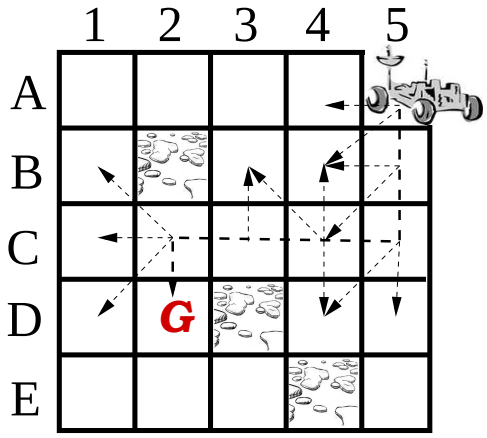

Figure 4.6: Example of a probabilistic planning problem.

4.4.1 The conceptual model

Probabilistic Planning tasks are modelled as a stochastic fully observable state-transition system with probability distributions associated to each state transition. This model, also known as MDP, is denoted by $\sum = (S, A, P, C, R)$, where:

- S is the finite set of states.
- A is the finite set of actions with stochastic effects.
- $P_a(s_i|s)$, is the probability that action $a \in A$ executed in state $s \in S$, leads to state $s_i \in S$. So, for each $s \in S$, if there exists $a \in A$ and $s_i \in S$ such that $P_a(s_i|s) \neq 0$, it is true that $\sum_i P_a(s_i|s) = 1$.
- $C(s, a)$ is the cost of applying action $a \in A$ in state $s \in S$.

- $R(s_i)$ is the reward achieved when reaching state $s_i \in S$.

According to this conceptual model, a probabilistic planning problem is defined as a tuple $P = (\sum, s_0, G)$ where $s_0 \in S$ is the initial state and $G \subseteq S$ is the set of goal states. Finding a solution to a probabilistic planning problem P consists of generating a sequence of actions $(a_1, a_2, ..., a_n)$ corresponding to a sequence of state transitions $(s_0, s_1, ..., s_n)$ such that s_i results from executing action a_i in state s_{i-1} and $s_n \in G$ is a goal state. The quality of a solution to a probabilistic planning problem depends on three factors: (1) the probability of achieving the goals $\prod_{i=0}^{n} P_a(s_i|s_{i-1})$, (2) the cost of the solution $\sum_{i=0}^{n} c(s_i, a_i)$ and (3) the achieved reward $\sum_{i=0}^{n} r(s_i)$. When the cost and the reward functions are expressed in the same units, the optimal solution to the probabilistic planning problem is the one that maximizes the expression $\prod_{i=0}^{n} P_a(s_i|s_{i-1}) * \sum_{i=0}^{n} r(s_i) - c(s_{i-1}, a_i)$

4.4.2 The representation languages

The Probabilistic Planning Domain Definition Language (PPDDL) (Younes et al., 2005) is the representation language for the probabilistic planning track of IPC. PPDDL is essentially an extension of PDDL2.1 (Fox and Long, 2003) to represent actions with *probabilistic effects* and *state rewards*.

- *Probabilistic effects* are an explicit declaration of the possible outcomes of an action execution. Their PPDDL syntax is as follows:

  $$(probabilistic\ p_1\ o_1\ p_2\ o_2 \ldots p_k\ o_k)$$

  where the outcome o_i of the action occurs with probability p_i. It is required that $p_i >= 0$ and $\sum_{i=1}^{k} p_i = 1$. However, PPDDL1.0 allows a probability-outcome pair to be left out if the effect is empty. In other words, $q = 1 - \sum_{i=1}^{k} p_i$ with $q > 0$. PPDDL1.0 allows arbitrary nesting of conditional and probabilistic effects, contrary to popular propositional encodings, such as probabilistic STRIPS operators (PSO's) (Kushmerick et al., 1995), which do not allow conditional effects nested inside probabilistic effects. While arbitrary nesting does not increase the expressiveness of the language, it can allow for exponentially more compact representations of certain effects given the same set of state variables and actions (Rintanen, 2003). However, any PPDDL action can be translated into a set of PSOs with at most a polynomial increase in the size of the representation. Consequently, it follows from the results of Littman (Littman, 1997) that PPDDL is representationally equivalent to dynamic Bayesian networks, which is also another popular representation of probabilistic planning problems.

- State rewards, are associated with state transitions and are encoded using fluents. PPDDL reserves the fluent `(reward)` to represent the total accumulated reward since the start of execution. Action preconditions and effect

conditions are not allowed to refer to the reward fluent, which means that the accumulated reward does not have to be considered part of the state-space. Besides, the initial value of reward is always zero. These two restrictions on the use of the `reward` fluent allow a probabilistic planner to handle domains with rewards, without implementing full support for fluents.

As an example, Figure 4.7 shows the action `unstack(block,block)` from the probabilistic version of the *Blocksworld* domain at the IPC-2004. This action indicates that with probability 0.7 a robot arm will unstack successfully the block `?top` from the block `?bot` and that with probability 0.3 the block `?top` will fall down.

```
(:action unstack
  :parameters (?top - block ?bot)
  :precondition
   (and (not (= ?top ?bot))
             (forall (?b - block)
                     (not (holding ?b)))
             (on-top-of ?top ?bot)
             (forall (?b - block)
                     (not (on-top-of ?b ?top))))
  :effect
   (and (probabilistic 0.7 (and(holding ?top)
                               (not (on-top-of ?top ?bot)))
                       0.3 (when(not (= ?bot table))
                                (and (decrease (reward) 10)
                                     (not (on-top-of ?top ?bot))
                                     (on-top-of ?top table))))))
```

Figure 4.7: PPDDL representation for the action `unstack` from *Blocksworld*.

4.4.3 The algorithms

There are three main approaches to tackle the probabilistic planning task:

- Extending a classical planner to handle probabilistic effects. The first attempts to face probabilistic planning problems followed this approach. Particularly, they extended partial order planners (Onder and Pollack, 1999) or graphplan planners (Blum and Langford, 1999). The solution plans provided by this approach consists on simple sequential plans, so they had to be complemented with replanning and plan repairing techniques (Fox et al., 2006a) to overcome unexpected outcomes of actions.

- Explicit search in the conceptual model. This group of algorithms model the dynamics of the environment as a Markov Decision Process (MDP) and

search for state-action policies optimizing a given utility function.[1] Examples of this approach are the classic *dynamic programming algorithms* for solving MDPs (Bellman, 1957; Bertsekas, 1995) also used in model-based Reinforcement Learning (Kaelbling et al., 1996). The drawback of this group of algorithms is that they rely on complete state enumeration so their time complexity is polynomial in the size of the state-space. In the AP problems, the size of the state-space grows exponentially with the number of features describing the problem. This state-space explosion problem limits the use of the MDP framework, and overcoming it has become an important topic of research. Over the last years, three different solutions have been posed:

1. *Heuristic Search*. Heuristic search algorithms limit computation to the states that are reachable from the initial state of the problem. These solutions include the LAO* algorithm (Hansen and Zilberstein, 2001), a generalization of the A* algorithm for MDPs, or the Learning Depth-First Search[2] (LDFS) algorithm (Bonet and Geffner, 2006), a generalization of the IDA* for MDPs.
2. *Symbolic Dynamic Programming*. Symbolic Dynamic Programming (SDP) exploits relational representation in the construction of the value function. This approach includes algorithms for:
   (a) First-Order Approximate Linear Programming (Boutilier et al., 2001) which uses EBL to implement a symbolic version of the value function regression. In this case the learning examples correspond to a sequence of actions that achieves a goal and EBL is used to generalize the regression of the value function over this actions sequence.
   (b) Updating the Bellman's backup operator to the relational setting (Kersting et al., 2004).
   (c) Using First-Order Binary Decision Diagrams (FODD) to represent value functions in a compact way (Wang et al., 2007).
3. *Factored Planning* (Guestrin et al., 2002). This solution exploits independence within a planning problem to decompose it, and then work on each subproblem (factor) separately while trying to piece the factor's solutions into a valid global solution.

- Compiling the planning problem into another problem solving paradigm for which there are effective algorithms. The main compilations of the probabilistic planning problem are:

[1] The *utility function* is a numeric function combination of the *cost* and *reward* functions which gives preference to the different states and transitions of the MDP.

[2] Learning in LDFS is understood in the same sense of in *Real-Time Heuristic search* (Korf, 1990; Bulitko and Lee, 2006; Hernández and Meseguer, 2007), i.e., as local updates of the value/heuristic function with information obtained from simulation.

1. The classical planning problem. This compilation (Jiménez et al., 2006a; Little and Thiébaux, 2007; Yoon et al., 2007a) builds exactly one deterministic action per outcome of a probabilistic action with an associated cost value indicating the probabilities associated to the outcome.
2. The E-MAJSAT problem.[3] This compilation (Majercik and Littman, 1998) is similar to the SAT compilation of the classical planning task except the encoding of the action effects. In this case, each action effect generates a clause consisting of *random propositions*, i.e., propositions that are true with a given probability value. Once the planning problem is encoded, a E-MAJSAT solver determines all possible satisfying assignments. For each satisfying assignment, computes the product of probabilities associated to the satisfied clauses. Finally, it returns the satisfying assignment the with highest product.
3. The CSP. Like the classical planning compilation, it fixes the length of the solution plan and converts it into a CSP. For this compilation the planning problem is encoded into a state-variable representation as follows:
   (a) For each step of the n-length plan there are three types of variables in the compiled CSP problem:
      i. i state variables whose values specify the state predicates holding at that step of the plan.
      ii. One action variable whose value specifies the action taken at that step of the plan.
      iii. r random variables whose values specify the particular stochastic outcome of the action taken at that step. The probability of each random value in these variables is indicated by the probability of the corresponding stochastic outcome of the action.
   (b) There are four types of constraints in the compiled CSP problem:
      i. Every state variable $state_i(t)$ whose value in the initial state is v is encoded into the unary constraint $state_i(0) = v$
      ii. Every state variable $state_i(t)$ whose value in the goals is v is encoded into the unary constraint $state_i(N) = v$
      iii. For every possible action outcome there is a constraint over: (1) the state variables at step t representing the action preconditions, (2) the state variables at step $t + 1$ representing the action outcome, (3) the random variables at step t representing the outcomes of the action, and (4) the action variable at step t.

[3] E-MAJSAT is a NP-complete problem consisting of, given a set of clauses with associated probabilities, finding the assignment of truth values that produces the highest product of the probabilities of satisfied clauses.

4.4.4 The implementations

This is an enumeration of some of the most referenced probabilistic planners:

mGPT

MGPT (Bonet and Geffner, 2004) is based on heuristic search for solving MDP models. It uses the algorithm *Labelled Real-Time Dynamic Programming* (LRTDP) which is a heuristic-search algorithm that implements a labelling scheme on top of the RTDP algorithm to get a faster convergence time together with a heuristic automatically extracted from the problem representation.

Foalp

FOALP (Sanner and Boutilier, 2006) translates the probabilistic planning problem into a First-Order Markov Decision Process (FOMDP) and uses approximate solution techniques for FOMDPs to derive a utility function using *First-Order Approximate Linear Programming*.

FPG

FPG (Buffet and Aberdeen, 2006) is the winner of the probabilistic track of IPC-2006. FPG uses gradient ascent for direct policy search and factors the parameterized policy by using a function approximation for each action.

Paragraph

PARAGRAPH (Little and Thiébaux, 2006) extends the Graphplan framework to probabilistic planning by introducing a node for each of an action's possible outcomes, so that there are three different types of layers in the graph: proposition, action, and outcome.

FF-Replan

FF-REPLAN (Yoon et al., 2007a) compiles the input probabilistic domain into a deterministic domain. There is two alternative compilations: (1) keeping only the most probable outcome or (2) creating a new deterministic domain with exactly one deterministic action per outcome of a probabilistic action. Then it generates a plan using the classical planner FF (Hoffmann and Nebel, 2001). If the execution of the plan reaches an unexpected state, FF-REPLAN replans with the same compilation of the problem to find a plan for this state.

RFF

RFF (Teichteil-Konigsbuch et al., 2008) is the winner of the probabilistic track of IPC-2006. This planner is based on computing an off-line policy combining

classical planning and simulation. RFF compiles the probabilistic problem into a deterministic problem with exactly one deterministic action per outcome of a probabilistic action. Then it computes a solution plan with the classical planner FF (Hoffmann and Nebel, 2001). Later, RFF uses Monte-Carlo simulation to estimate the probability of failure of the plan steps. If this probability exceeds a threshold at a given step, RFF computes a new plan for overcoming the failures of this step and starts a new Monte-Carlo simulation for the new plan.

LPFF

LPFF (Kalyanam and Givan, 2008) follows a *divide and conquer* strategy. First, it creates a new deterministic problem with exactly one deterministic action per outcome of a probabilistic action. Second, it uses a classical planner to get a plan $(a_1, a_2, ..., a_n)$ corresponding to a sequence of state transitions $(s_0, s_1, ..., s_n)$ such that s_n is a goal state. Finally, LPFF uses a probabilistic planner to extract policies for the subproblems consisting of reaching state s_{i+1} from state s_i. The sequence of these policies forms the policy for the original problem.

SEH

This planner implements the Stochastic Enforced Hill Climbing algorithm (SEH) (Wu et al., 2008). This algorithm extends the heuristic search algorithm *EHC* for classical planning to the probabilistic planning framework. Particularly, SEH uses a heuristic function based on estimating the expected cost to the goals together with a best-first search (instead of the breath first search of classic EHC) for escaping *plateaus*.

HMDPP

HMDPP (Keyder and Geffner, 2008) implements a heuristic search guided by two heuristics. The first one is h_{add} computed with the domain resulting from the compilation of probabilistic actions into cost actions. The second one, used for breaking ties, is a PDB heuristic.

4.5 The conformant probabilistic planning task

Conformant probabilistic planning is a generalization of conformant planning. Particularly, conformant probabilistic planning is the task of synthesizing plans that maximize the probability of reaching a given set of goals in a non-observable stochastic environment.

4.5.1 The conceptual model

The conceptual model for the conformant probabilistic planning is a non-observable stochastic transition system with probability distributions associated to each tran-

sition. This model, also known as Non-Observable MDP (NOMDP), is defined as a tuple $\sum = (B, A, P, C)$, where:

- B is the set of belief states. In this case, a belief state $b \in B$ is a probability distribution over the states. The probability assigned by b to each state $s \in S$ is denoted by $b(s)$.
- A is a finite set of actions with stochastic effects. Each action $a \in A$ maps a given belief state $b \in B$ in a new belief state b_a.
- $P_a(s'|s)$, is the probability that action $a \in A$ executed in state $s \in S$, leads to state $s' \in S$. Regarding this, the probability of a yielding s' when applied in $b \in B$, can be computed as the sum of the probability distribution determined by b weighted by the probability that action a leads from $s \in S$ to s':

$$b_a(s') = \sum_{s \in S} P_a(s'|s)b(s)$$

- $C(a)$ represents the cost of applying action $a \in A$.

According to this conceptual model, a conformant probabilistic problem is a tuple $P = (\sum, b0, G)$ where $b0 \in B$ is the initial belief state, i.e., the set of possible initial states. And $G \subseteq B$ is the set of belief states that contain only goal states. Finding a solution to a conformant probabilistic planning problem consists of synthesizing the policy that maximizes the probability $\prod_{i=0}^{n} b_{a_i}(G)$ for all the reachable belief states.

4.5.2 The representation language

Conformant probabilistic planning problems can be described in a probabilistic planning representation language (such as PPDDL) specifying the initial state through a disjunction of literals.

4.5.3 The algorithms

- Explicit search in the conceptual model. Heuristic search algorithms able to deal with disjunctive transitions such as LAO* or dynamic programming algorithms such as value iteration search for a solution plan in a NOMDP.
- Compilation into another problem solving paradigm
  - E-MAJSAT Problem. This compilation is similar to the E-MAJSAT compilation for probabilistic planning. The only difference is that this compilation codes the possible initial states with *random propositions*.
  - CSP Problem. This compilation is similar to the CSP compilation for probabilistic planning. The only difference is the encoding of the initial state. In this case, the initial belief state b_0 is encoded using an extra

stochastic action. This action is always executed at time step zero, it has no preconditions and it presents probabilistic effects corresponding to the possible states of b_0.

4.5.4 The implementations

Buridan

BURIDAN (Kushmerick et al., 1995) takes as input a probability distribution over states and produces a plan such that the probability to reach the goals after the plan execution is greater than a given threshold.

MAXPlan

MAXPLAN (Majercik and Littman, 1998) is based on compiling a planning instance into an E-Majsat problem (a probabilistic version of SAT), and then draws on techniques from Boolean satisfiability and dynamic programming to solve the E-Majsat problem.

CPPlan

The CPP planner (Hyafil and Bacchus, 2003) implements the CSP compilation for the conformant probabilistic planning task and then it uses standard CSP backtracking algorithms to compute the solution plans. Finally, among all the solution plans, CPP chooses the one with the highest probability of success: the probability that a given execution path is traversed by a given solution plan.

Probapop

PROBAPOP (Onder et al., 2004) is based on generating base plans with the deterministic partial-order planner VHPOP and then refine the base plans until it finds a solution plan that meets or exceeds a given probability threshold.

ComPlan

COMPLAN (Huang, 2006) performs depth-first branch-and-bound search in the plan space. For each potential search node, an upper bound is computed on the success probability of the best plans under the node, and the node is pruned if this upper bound is not greater than the success probability of the best plan already found. A major source of efficiency for this algorithm is the efficient computation of these upper bounds, which is possible by encoding the original planning problem as a propositional formula and compiling the formula into deterministic decomposable negation normal form.

4.6 The contingent probabilistic planning task

Contingent probabilistic planning is the task of synthesising a contingent plan in stochastic and partially observable environments.

4.6.1 The conceptual model

The conceptual model for contingent probabilistic planning is a partially observable stochastic transition system with probability distributions associated to each state transition. This model, also known as POMDP, can be represented as a tuple $\sum = (B, A, P, O, C)$, where:

- B is the set of belief states. In this case, a belief state $b \in B$ is a probability distribution over the states. The probability assigned by b to each state $s \in S$ is denoted by $b(s)$.
- A is a finite set of stochastic actions.
- $P_a(s'|s)$, is the probability that action $a \in A$ executed in state $s \in S$, leads to state $s' \in S$. Regarding this, the probability of a yielding s' when applied in $b \in B$, can be computed as the sum of the probability distribution determined by b weighted by the probability that action a leads from $s \in S$ to s':

$$b_a(s') = \sum_{s \in S} P_a(s'|s)b(s)$$

- O is the set of observations. For each $o \in O$, $a \in A$ and $s \in S$ there is a known probability $P_a(o|s)$ that represents the probability of observing o in the state s after executing action a. These probabilities are defined for each state $s \in S$ and action $a \in A$ thus, $\sum_{o \in O} P_a(o|s) = 1$.
- $C(s, a)$ represents the cost of applying action $a \in A$ in state $s \in S$.

With all these definitions, the probability of observing $o \in O$ after executing $a \in A$ in belief state b is given by:

$$b_a(o) = \sum_{s \in S} P_a(o|s)b(s)$$

Finally, the probability of reaching state s after executing action a in belief state b, observing o is given by:

$$b_a^o(s) = P_a(o|s)\frac{b_a(s)}{b_a(o)}$$

According to this conceptual model, a contingent probabilistic problem is a tuple $P = (\sum, b0, G)$ where $b0 \in B$ is the initial belief state, i.e., the set of possible initial states. And $G \subseteq B$ is the set of belief states that contain only goal states. Finding a solution to a contingent probabilistic planning problem consists of synthesizing the policy that maximizes the probability $\prod_{i=0}^{n} b_{a_i}^o(G)$ for all the reachable belief states.

4.6.2 The representation language

Contingent probabilistic planning needs a representation language that is able to describe probabilistic actions and the acquisition of information about the current state at execution time. This can be captured with a probabilistic planning domain description language –like PPDDL– extended with sensing actions. Nevertheless, there is no standard representation.

4.6.3 The algorithms

- Explicit search in the conceptual model. Heuristic search algorithms able to deal with disjunctive transitions such as LAO* or dynamic programming algorithms such as value iteration search for a solution plan in a POMDP.

4.6.4 The implementations

GPT

GPT (Bonet and Geffner, 2004) is based on heuristic search for solving MDP models. It uses the algorithm *Labelled Real-Time Dynamic Programming*(LRTDP) which is a heuristic-search algorithm that implements a labelling scheme on top of the RTDP algorithm to get a faster convergence time together with heuristics automatically extracted from the problem representation.

Pond

POND (Bryce, 2006), like GPT, searches forward in the space of belief states. As GPT, it implements various search algorithms (A*, AO*, LAO*, Enforced Hill-Climbing) and relaxed plan heuristics that are applied depending on the user preferences. One of these heuristics is a distance estimation between the current belief state and the goal state. This distance value is computed as an aggregate measure of the underlying distances between states in the belief states.

4.7 Interleaving planning and execution

The PUU algorithms seen in this chapter *off-line* reason about complete and correct action and sensor models. These models must capture the uncertainty of the world, so defining them by hand, is a very hard task. Even in traditionally easy-to-code planning domains like the *Blocksworld*, it is very complex (and sometimes impossible) to *'a priori'* know all potential action outcomes and their associated probabilities. Besides, many of the reviewed PUU algorithms concentrate on optimizing plan quality in terms of the expected utility of solutions. Nevertheless, the synthesis of optimal or near-optimal plans is computationally very expensive and current AP techniques only achieve limited success when optimizing a numerical function.

Because of these factors, using a PUU algorithm in the real-world does not always pay off. As a matter of fact, practitioners frequently prefer to interleave simple planning algorithms with execution monitoring and to adapt plans when conflicts appear after a state update. Figure 4.8 shows an overview of a simple architecture that interleaves planning and execution. According to this architecture model, a classical planner synthesizes a plan regarding the description of the initial state, the problem goals and a deterministic action model (note that this action model is incomplete since it does not consider any contingency). The plan is executed step by step and each execution is monitored for reacting to unexpected events.

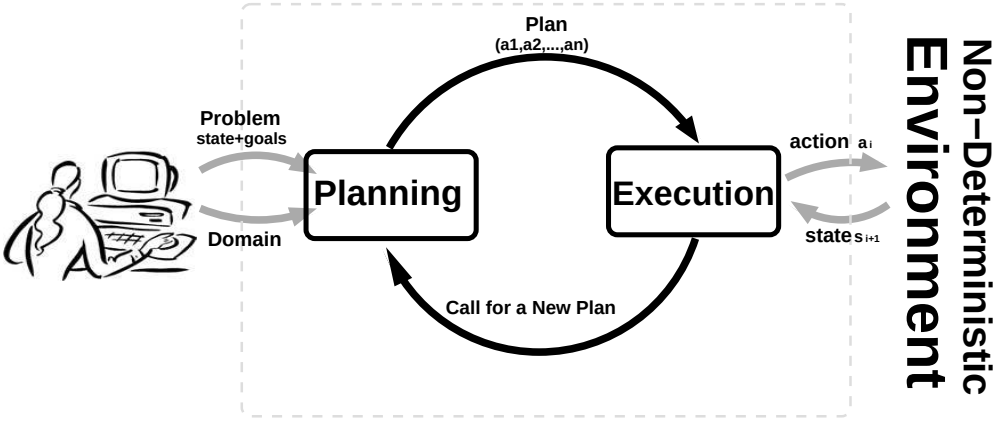

Figure 4.8: Overview of an architecture for *interleaving planning and execution*.

Next, there is a detailed description of the most common techniques for implementing an architecture that interleaves planning and execution:

4.7.1 Planning

When interleaving planning and execution there is a trade-off between the quality and the urgency of the planning response. On the one hand, the planning time must be long enough for generating plans that reach the problem goals and satisfy the quality requirements. On the other hand, the planning time must be short enough for guaranteeing a smooth execution of the plan. This trade-off can be addressed in different ways:

- Hard-coding reactive behavior. The designer specifies a library of plans that define the actions to follow for solving a given set of tasks. However, hard-coding a plan library is an arduous task. Otherwise, one can automatically generate the plan library with AP (Kelly et al., 2008). In this case, HTN is a powerful tool for specifying highly detailed alternative plans that fulfil a set of tasks. In addition, CBR can be used to recover and adapt the most suitable plan according to the current situation (Aha et al., 2005).

- Implementing reactive behavior with time-bounded deliberative techniques.
  - *Anytime search algorithms* (Hansen and Zhou, 2007). These algorithms search greedily for a solution that can be quickly computed and monotonically improve the solution as long as time is available.
  - *Real-Time search algorithms* (Korf, 1990; Bulitko and Lee, 2006; Hernández and Meseguer, 2007). These search algorithms fast provide steps towards a goal state though the complete solution is not constructed. These steps will eventually converge to a solution when enough time is available. This family of algorithms is convenient for search problems unapproachable from traditional search algorithms.

4.7.2 Execution

The execution of plans in the real-world is monitored to guarantee the achievement of the planning goals. The monitoring process consist of checking that:

- The goals of the planning task keep the same. If planning goals are added or removed, the current plan may become useless.
- The execution of an action produces the action nominal effects. This includes checking that the action execution consumed/produced the expected resources and checking that the beginning and end of the execution happened at the expected time.

Plan monitoring was first implemented in the Shakey robot for checking whether a remaining subplan was still executable. A given subplan was considered executable when the preconditions of the subplan actions, that were not established by actions of the subplan, were currently holding in the environment. Figure 4.9 shows an example of a *Triangle Table* (Fikes et al., 1972) for checking the executability of the subplan `unstack(A,C), put-down(A), pick-up(B)` in a two-blocks *Blocksworld* problem. The literals of the first column correspond to the set of literals that have to hold for the subplan to be executable.

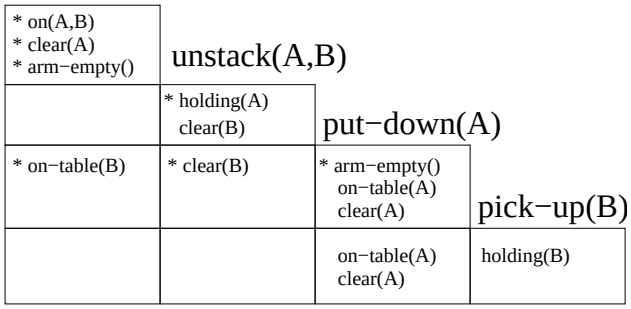

Figure 4.9: Example of *Triangle Table* for the *Blocksworld*.

Action monitoring, opposite to *plan monitoring*, only checks the preconditions of an action when it is executed so it does not keep track of the whole plan. *Action monitoring* is less effective because it does not lookahead to see that an unexpected event will cause an action failure in the future but it is easier to implement and do not require annotations.

One cannot always rely on agents to communicate their state to the monitoring process. For this reason, new approaches for *monitoring through deduction* are emerging. Among these are the *report-based monitoring* (Kaminka et al., 2002), where the monitored state is inferred (via plan-recognition) from routine communications, or the *knowledge-based execution monitoring*, where the actual state is derived from the available perceptual information. This last approach is useful for those domains in which it is only possible to monitor a subset of the action preconditions.

When execution monitoring detects that the current plan does not longer attain the goals, there are two alternative strategies to modify it:

- *Replanning*, this strategy computes a new plan from scratch for the new state of the environment and/or the new planning goals.

- *Repairing*, this strategy adapts the existing plan to the new state of the environment and/or the new planning goals. Because planning is time consuming, *repairing* takes advantage of old plans to guide the search for new plans (Gerevini and Serina, 2000; Koenig et al., 2002). However, as shown in the recent work for *repairing* based on *plan stability* (Fox et al., 2006a) the benefits of *repairing* decrease when the environment is highly dynamic.

4.7.3 Planning and execution in autonomous systems

An autonomous system is an entity (either software or hardware) which can perform desired tasks in unstructured environments without continuous human guidance. Traditionally, architectures for controlling autonomous systems share the following four layers.

1. *Sensory layer*: reads the sensor of the system and extracts a symbolic representation of the state of the environment. Raw sensory data are too detailed and sometimes noisy. This layer implements mechanisms for the fusion and filtering of the sensory data to obtain a reliable representation of the current state of the environment (Fox et al., 2006b).

2. *High-level planning layer*: builds mission plans consisting of high level actions. Figure 4.10 shows an example of a mission plan for the Mars Rovers (Bresina et al., 2005). At this layer, off-the-shelf planners are not widely used. Instead, autonomous architectures implement specific planners designed for fitting the performance of certain tasks. Frequently, this layer is implemented by a CSP-based domain dependent planner (Nayak et al., 1999;

Lemai and Ingrand, 2004; Bresina et al., 2005; McGann et al., 2008). The CSP framework provides expressive representations of the planning tasks such as handling time, resources and preferences. In addition, a CSP-planner can synthesize partially ordered plans which are convenient for dynamic environments given that one can delay instantiation commitments to the execution. Finally, a CSP search in the plan space can be easily adapted to incremental planning and plan repair.

```
 Time:   (ACTION) [Duration; Cost]
-------- -------- ----------------------------------------
 00.000: (NAVIGATE WAYPOINT2 WAYPOINT1) [D:5.000; C:1.000]
 05.000: (CALIBRATE CAMERA0 OBJECTIVE0 WAYPOINT1) [D:5.000;
          C:1.000]
 10.000: (SAMPLE_ROCK ROVER1STORE WAYPOINT1) [D:8.000; C:1.000]
 18.000: (NAVIGATE WAYPOINT1 WAYPOINT2) [D:5.000; C:1.000]
 23.000: (TAKE_IMAGE WAYPOINT2 OBJECTIVE0 CAMERA0 HIGH_RES)
         [D:7.000; C:1.000]
 30.000: (NAVIGATE WAYPOINT2 WAYPOINT3) [D:5.000; C:1.000]
 35.000: (COMMUNICATE_ROCK_DATA GENERAL WAYPOINT1 WAYPOINT3
          WAYPOINT2) [D:10.000; C:1.000]
 45.000: (COMMUNICATE_IMAGE_DATA GENERAL OBJECTIVE0 HIGH_RES
          WAYPOINT3 WAYPOINT2) [D:15.000; C:1.000]
```

Figure 4.10: Example of a mission plan for the Mars Rovers domain.

3. *Low-level planning layer*: controls the completion of the actions from the mission plan. Figure 4.11 shows an example of a path-planning problem corresponding to the control of the high-level action `(NAVIGATE WAYPOINT2 WAYPOINT1)`. This layer is frequently implemented as a set of controllers

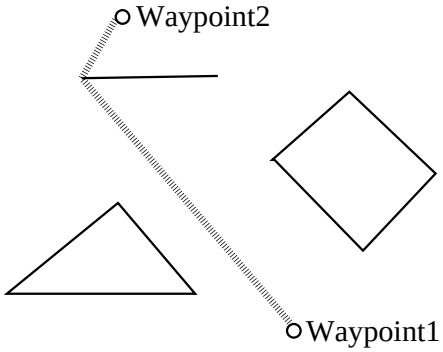

Figure 4.11: Path-planning for `(NAVIGATE WAYPOINT2 WAYPOINT1)`.

for the diverse specific actions. An example of these low-level controllers are the path-planning modules developed for controlling the movement of

Non-Player Characters (NPCs) in commercial computer games such as UNREAL[4] or DRAGON AGE[5]. These modules use informed search algorithms such as A*,D* (Stentz, 1994), RTA* (Korf, 1990; Bulitko and Lee, 2006; Hernández and Meseguer, 2007) or RRT (Lavalle, 2000) guided by the Euclidean distance to synthesize the sequence of steps for transversing the distance between two given points.

4. *Actuators layer*: interacts with the environment through diverse mechanisms: speakers, motors, robotic arms, etc.

Figure 4.12 shows an overview of a generic architecture for the control of an autonomous system. According to this generic architecture the *sensory layer* extracts a symbolic representation of the current state. With this information the *high-level planning layer* chooses a task and synthesizes a mission plan to fulfill the task. Each action in the mission plan is communicated to the *low level planning layer* which tries to execute them in the environment controlling the *actuators layer* and the *sensory layer*. If the execution of the action eventually fails, the *low level planning* demands *the high-level planning layer* a new plan for overcoming the failure. Example of architectures that deal with the integration of sensory/motor functions, low-level control and deliberative capabilities are (Simmons, 1994), (Alami et al., 1998), (Hertzberg et al., 1998), (Beetz, 1999) or (Lenser et al., 2002).

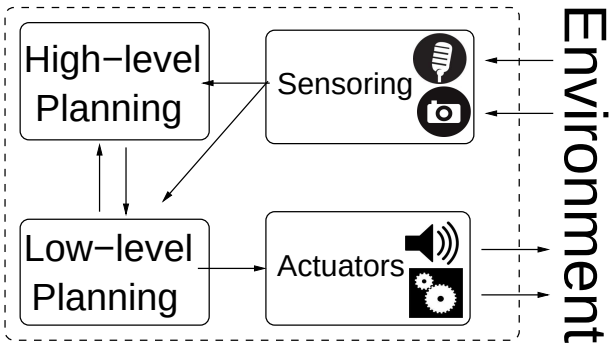

Figure 4.12: Generic Architecture for an autonomous system.

4.8 Discussion

In the late 50's Richard Bellman initiated the modern approach to *Dynamic Programming* (Bellman, 1957). Since this work, more elaborated algorithms have

[4]http://www.unreal.com/
[5]http://dragonage.bioware.com/

been develop to solve optimization problems over MDPs and POMDPs (Lovejoy, 1991; Bertsekas, 1995). However, the application of these algorithms to PUU was limited given that their complexity is polynomial in the size of the state-space and in AP, this size grows exponentially with the number of features describing the problem.

In the last decade, there is a strong interest in boosting the scalability of *Dynamic Programming*. On the one hand, this interest has resulted in *Symbolic Dynamic Programming* that handles relational representation of the value function and the state-space (Boutilier et al., 2001; Groote and Tveretina, 2003; Kersting et al., 2004; Wang et al., 2007). On the other hand, heuristic planning, which is based on propositional representation of the state-space, has developed efficient domain independent heuristic and reachability tools for PUU (Brafman and Hoffmann, 2004; Hoffmann and Brafman, 2005; Bonet and Geffner, 2006). The models, algorithms and representation languages of both approaches are getting closer and ideas and benchmarks developed in one approach are fast applied and tested in the other.

At the present time, PUU is still a young research area with many challenges to prove the practical utility of this new planning paradigm:

- **Current PUU planners have strong scalability limitations**. Currently, conformant planners are far from solving interesting problems like the *home-sequencing problem*.[6] To illustrate this drawback, the most difficult problem of the IPC-2008 conformant track in the *Blocksworld* domain only had 4 blocks. Additionally, the size of the current contingent plans is exponential with the number of observations. Because of that, building or verifying complete contingent plans for a realistic problem with the current contingent planners is in general not feasible.

- **Current PUU planners handle action models with limited expressiveness**. Most of the reviewed PUU planners are only able to deal with STRIPS actions and cannot express preconditions or problem goals with disjunctions or quantified formulas, handle time, action cost or state rewards.

- **PUU planning models are hard to design, validate and maintain**. Like classical planning, PUU needs complete and correct action models but the design, validation and maintenance of PUU models is harder. Even in simple planning domains like the *Blocksworld*, when actions have stochastic behavior they may result in innumerable different outcomes. Specifying by hand all of them and/or their associated probabilities is hard, and if it is viable at a given instant, they could vary over time.

As a consequence of the current limitations of the PUU techniques, practitioners prefer interleaving deterministic planning with execution monitoring and

[6] The *home-sequencing problem* is the problem of executing a sequence of instructions that makes a processor move to a desired state. In this problem, no observation is available because memory variables are not readable.

on-line *replanning* or *plan repair*: the current classical planners scale better (FF-Replan achieved the best overall performance at the probabilistic competition of IPC-2004 and IPC-2006), they are more expressive (last developed planners give support to durative actions and user preferences) and their actions models are simpler (deterministic planners just consider the nominal effects of actions). However, this approach is not a *panacea*. Since classical planning assumes no action failures, it can produce too *fragile* plans that have to be continuously repaired or even worse, plans that cannot be fixed when broken.

Chapter 5

Learning for planning under uncertainty

This chapter analyzes how AP can benefit from ML when planning for domains with uncertainty. The analysis is structured regarding the target of the ML process: learning **search control** or learning **action models**.

5.1 Introduction

Like classical planning, PUU can benefit from ML for:

- *Learning planning search control*. **The combinational explosion problem in PUU is even greater than in classical planning**. The uncertainty caused by partial observability and non-deterministic actions multiplies the amount of possible reachable states and therefore, boosts the complexity of the search processes. Search control knowledge can help PUU to reduce this complexity but, again, hand-coding this knowledge is a hard task that requires expertise on the domain and the planner.

- *Learning planning action models*. PUU requires correct and complete action models but **the task of defining correct and complete action models is harder in environments with uncertainty**. Even for traditionally simple planning domains like the *Blocksworld*, it is very complex to *'a priori'* know all potential action outcomes and their associated probabilities.

ML can be a solution to automatically define *search control* and *action models* for PUU, but the integration of planning and learning in environments with uncertainty presents new difficulties: (1) partial observations of the current state can produce learning examples with missing or wrong literals. Besides, (2) the non-determinism of the planning actions introduce extra noise to the learning examples.

5.2 Learning techniques

AI has considered diverse alternatives for representing uncertainty like *certainty factors* (Shortliffe, 1976), the *Dempster-Shafer theory* (Shafer, 1976), *default logic* (Reiter, 1980), *non-monotonic logic* (McDermott and Doyle, 1980) or *fuzzy logic* (Zimmerman, 1990). This chapter focuses on learning techniques that represent uncertainty combining first-order logic and probability because they naturally suit current representations of the PUU tasks. ML techniques for learning representations combining first-order logic and probability share a two-steps strategy:

1. *Structural Learning*. At this step, relational learning algorithms are used for inducing first-order logic formulas that generalize the relations over the ground facts of the learning examples.

2. *Parameter Estimation*. This step estimates the probabilities associated to the induced logic formulas. Frequently, the output of the *Structural Learning* step is a set of rules of the form $C \rightarrow P$ where P is the prediction of the rule and C is the set of conditions under which the prediction is true. In this case, the *Parameter Estimation* step consists of a computation of the conditional probability $prob(P|C)$. When the induced rules are disjunctive, this probability is directly extracted from the frequency of the learning examples. However, when rules are overlapping, the estimation of this conditional probability requires more elaborated approaches:

   (a) *Bayesian Estimation* (BE). The Bayes Theorem estimates the conditional probability as follows

   $$prob(P|C) = prob(C|P)\frac{prob(P)}{prob(C)}$$

   (b) *Maximum Likelihood Estimation* (MLE). MLE is an statistical method for fitting a mathematical model to some observed data. This method assumes that data follow a known probability distribution and iteratively tunes the parameters of the distribution to achieve a good fit.

According to the type of structure of the induced logic formulas and the technique used for the parameter estimation there are different ML techniques. Next there is a review of the most relevant ones for PUU.

5.2.1 Learning Stochastic Logic Programs

Stochastic Logic Programs (SLPs) (Muggleton, 1995b; Cussens, 2001) represent a distribution probability over a Prolog proof tree. The probability over predicates has to be obtained by marginalization. Specifically, a SLP consists of a set of parametrised logic clauses $p : C$ where p is the parameter (a real number that

belongs to the interval [0,1]) and C is a *Horn clause*. Figure 5.1 shows an example of a SLP for *the blood type inheritance model*.[1]

```
0.97: bloodtype(X,ab) :- mother(M,X), mothercopy(M,a),
                         father(F,X), fathercopy(F,b).
0.01: bloodtype(X,a) :- mother(M,X), mothercopy(M,a),
                        father(F,X), fathercopy(F,b).
0.01: bloodtype(X,b) :- mother(M,X), mothercopy(M,a),
                        father(F,X), fathercopy(F,b).
0.01: bloodtype(X,0) :- mother(M,X), mothercopy(M,a),
                        father(F,X), fathercopy(F,b).

0.97: bloodtype(X,ab) :- mother(M,X), mothercopy(M,b),
                         father(F,X), fathercopy(F,a).
0.01: bloodtype(X,a) :- mother(M,X), mothercopy(M,b),
                        father(F,X), fathercopy(F,a).
0.01: bloodtype(X,b) :- mother(M,X), mothercopy(M,b),
                        father(F,X), fathercopy(F,a).
0.01: bloodtype(X,0) :- mother(M,X), mothercopy(M,b),
                        father(F,X), fathercopy(F,a).

0.97: bloodtype(X,T) :- mother(M,X), mothercopy(M,T),
                        father(F,X), fathercopy(F,0).
0.03: bloodtype(X,T2) :- mother(M,X), mothercopy(M,T),
                         father(F,X), fathercopy(F,0),
                         T2\=T.

0.97: bloodtype(X,T) :- mother(M,X), mothercopy(M,0),
                        father(F,X), fathercopy(F,T).
0.03: bloodtype(X,T2) :- mother(M,X), mothercopy(M,0),
                         father(F,X), fathercopy(F,T),
                         T2\=T.

0.97: bloodtype(X,T) :- mother(M,X), mothercopy(M,T),
                        father(F,X), fathercopy(F,T).
0.03: bloodtype(X,T2) :- mother(M,X), mothercopy(M,T),
                         father(F,X), fathercopy(F,T),
                         T2\=T.
```

Figure 5.1: Stochastic Logic Program for *the blood type inheritance model*.

The *structural learning* of SLPs is implemented by standard ILP algorithms for inducing Logic Programs, like the algorithms developed in PROGOL (Muggleton, 1995a). The *parameter estimation* of SLPs requires an approach able to deal with overlapping rules such as BE or MLE. Given a learning example e and a SLP s, it

[1] *The blood type inheritance model* (Friedman et al., 1999) is a genetic model of the inheritance of a single gene that determines a person's X blood type $[a|b|ab|0]$. Each person X has two copies of the chromosome containing this gene, one inherited from the mother and one inherited from the father.

is unknown which clause of s explains e. As an example, in the SLP of Figure 5.1, two different rules could generate the example $bloodtype(John, a)$.

5.2.2 Learning Bayesian Logic Programs

Bayesian Logic Programs (BLPs) (Wellman et al., 1992; Poole, 1993; Jaeger, 1997; Ngo and Haddawy, 1997; Kersting and Raedt, 2001) represent a first-order Bayesian Network. Specifically, a BLP consists of two elements: (1) A set of Bayesian clauses $A \mid A_1, ..., A_n$ where $A, A_1, .., A_n$ are logic atoms implicitly quantified. (2) A set of conditional probability distributions corresponding to the Bayesian clauses. Figure 5.2 shows an example of a BLP for *the blood type inheritance model*.

(1) Set of Bayesian clauses

```
bloodytype(X)| mothercopy(X), fathercopy(X).
mothercopy(X)| mother(Y,X), mothercopy(Y), fathercopy(Y).
fathercopy(X)| father(Y,X), mothercopy(Y), fathercopy(Y).
```

(2) Conditional probability distribution of clause `bloodytype(X).`

`mothercopy(X)`	`fathercopy(X)`	Probability(`bloodtype(X)`) a	b	ab	0
a	a	0.97	0.01	0.01	0.01
a	b	0.1	0.1	0.97	0.01
...			...		
0	0	0.01	0.01	0.01	0.97

Figure 5.2: Bayesian Logic Program for *the blood type inheritance model*.

Unlike SLPs, BLPs explain each logical concept with exactly one rule. As a consequence, the *structural learning* step must induce only one clause for each concept. At this step, one can use ILP algorithms for *learning from interpretation* (Muggleton et al., 1994; De Raedt and Dehaspe, 1997). These algorithms consider all learning examples as positive examples and induce a hypothesis that is logically true for them. The *parameter estimation* is implemented using the joint probability distributions and the Bayes Theorem. As an example, the conditional probability of having `bloodytype(a)` given `mothercopy(a)` and `fathercopy(a)` is expressed as:

$$P(A \mid B, C) = P(A)P(B \mid A)\frac{P(C \mid A, B)}{P(B)P(C \mid B)}$$

where `A` is `bloodytype(X)`, `B` is `mothercopy(X)` and `B` is `fathercopy(X)`.

5.2.3 Learning Markov Logic Networks

The two previous ML techniques combined probability with restricted subsets of first-order logic: *Horn clauses* in case of SLPs and *Bayesian clauses* in the case of BLPs. Markov Logic Networks (MLNs) (Richardson and Domingos, 2006) is a step further in generality from those because MLNs combine probability and first-order logic with no restrictions other than finiteness of the domain. Specifically, a MLNs is a set of first-order logic formulas with a weight attached to each formula. These formulas can be viewed as soft constraints over the examples and their weights as measures of the strength of the constraints. The higher the weight, the greater the difference in log probability between an example that satisfies the formula and one that does not. Figure 5.3 shows an example of a MLN.

English	First-Order	Weight
Friends of friends are friends	$\forall x \forall y \forall z Friends(x,y) \wedge Friends(y,z) \Rightarrow Friends(x,z)$	0.7
Friendless people smoke	$\forall x(\neg(\exists y Friends(x,y)) \Rightarrow Smoke(x))$	2.3
Smoking causes cancer	$\forall x Smoke(x) \Rightarrow Cancer(x)$	1.5
If 2 people are friends, either both smoke or neither does	$\forall x \forall y Friends(x,y) \Rightarrow (Smoke(x) \Leftrightarrow Smoke(y))$	1.1

Figure 5.3: Example of a MLN.

The *structural learning* of MLNs requires specific ILP algorithms because they must induce not just *Horn clauses* but arbitrary formulas (Kok and Domingos, 2005; Mihalkova and Mooney, 2007). These algorithms implement a heuristic search in the space of formulas guided by an evaluation function that measures the pseudo-likelihood (Besag, 1975) of a formula over the data. Candidate formulas are created by adding each predicate (negated or otherwise) to the current formula, with all possible combinations of variables, subject to the constraint that at least one variable in the new predicate must appear in the current formula. The *parameter estimation* step is done by maximizing the pseudo-likelihood of the data (Besag, 1975). Combined with the L-BFGS optimizer, pseudo-likelihood can yield efficient learning of MLN weights even in domains with millions of ground atoms.

5.2.4 Reinforcement Learning

The revised ML techniques share a common feature: they depend on external expertise to (1) specify the learning target and (2) collect significant examples of the learning target. Unfortunately, such external expertise is not always available.

Reinforcement Learning (RL) reduces the required external expertise to a reward signal guidance, learning to take decisions by trial and error. RL agents interact with the environment to collect experience that– with the appropriate algorithms– is processed to generate an optimal policy (Kaelbling et al., 1996; Sutton and Barto, 1998). Since RL discovers the best decisions by trying them, a key issue in RL is

determining when to try new actions and when to use known ones. RL studies this issue as the *exploration-exploitation* dilemma, where *exploration* is defined as trying new actions and *exploitation* is defined as applying actions that succeeded in the past. In general terms, good answers to the *exploration-exploitation* dilemma consider the number of trials allowed; the larger the number of trials is, the worse is to prematurely converge to known actions because they may be suboptimal. For surveys on efficient *exploration-exploitation* strategies see (Wiering, 1999; Reynolds, 2002).

Model-Based and Model-Free Reinforcement Learning

RL was primarily concerned with obtaining optimal policies when the model of the environment is unknown but actually, there are two ways to proceed. *Model-Based* RL requires a environment's model consisting of a transition and a reward model. This approach learns a transition model of the environment and applies standard *dynamic programming* algorithms to find a good policy. *Model-Free* RL does not require a model of the environment. In domains with large uncertainty learning to achieve goals is easier than learning a model of the environment.

Pure *Model-free RL* algorithms do not model the decision-making as a function of the state, like *value/heuristic functions*, but with a function of pairs $< state, action >$ called *action-value functions*. The *Q-function*, which provides a measure of the expected reward for taking action a at state s, is an example of an *action-value function*. *Q-learning* (Watkins, 1989), a well-known *Pure Model-free* RL algorithm, updates the *Q-function* with every observed tuple $< s, a, s', r >$ (s' represents the new state and r represents the obtained reward). *Q-learning* completes the update of the *Q-function* using the following formula,

$$Q_{t+1}(s,a) := Q_t(s,a) + \alpha(r(s,a) + \gamma \max_{a'} Q(s',a') - Q_t(s,a))$$

where α is the learning rate that determines to what extent the newly acquired information overrides the old information. When $\alpha = 0$ the agent does not learn anything, while when $\alpha = 1$ the agent only considers the most recent information. γ is the discount factor which determines the importance of future rewards. A factor of $\gamma = 0$ makes the agent *greedy* (the agent only considers current rewards), while a factor approaching 1 makes the agent strive for a long-term high reward. *Pure model-free* RL also includes *Monte Carlo* methods (Barto and Duff, 1994). Most of *pure model-free* RL methods guarantee to find optimal policies and use very little computation time per observation. However, they typically make inefficient use of the collected observations and require extensive experience to achieve good performance.

Relational Reinforcement Learning

Most of the work on RL has focused on propositional representations for the states and the actions. Thus, RL is not applicable to tasks with relational structure (like

the classical AP tasks) without extensive engineering effort. Recently, there are coming up numerous attempts to bring together RL algorithms and the expressiveness of relational representations for encoding states and actions. They are grouped in a new field called Relational Reinforcement Learning (RRL). This section only considers the model-free RRL techniques because the model-based ones (*Symbolic Dynamic Programming* techniques) were revised in Section 4.4.3 as algorithms for probabilistic planning. The different approaches for model-free RRL are:

- *Relational learning of the q-value function.* This approach consists of using relational regression tools to generalize the value function. So far, three different relational regression algorithms have been used:
  - Relational regression trees (Dzeroski et al., 2001). For each pair *(action, relational-goal)*, a regression tree is built from a set of examples of the form `(state,q-value)`. The leaf nodes of the induced tree represent the predictions of the *q-value*. The test nodes represent the set of facts that have to hold for the predictions to be true. Figure 5.4 shows an example of relational regression tree that captures the *q-values* of the action *move(Block,Block)* for solving the set of tasks on(Block,Block) in the *Blocksworld* domain.

```
goal_on(A,B), numberofblocks(C),action_move(D,E).
on(A,B)?
+ yes: [0]
+--no: clear(A)?
       +--yes:[1]
       +--no: clear(E)?
              +--yes:[0.9]
              +--no:[0.81]
```

Figure 5.4: Relational regression tree for the goals *on(X,Y)* in the *Blocksworkd*.

  - Instance based algorithms with relational distance (Driessens and Ramon, 2003). In this case, a k-nearest neighbor prediction is done. It computes a weighted average of the *q-values* of the examples stored in memory where the weight is inversely proportional to the distance between the examples. The used distance requires to cope with the relational representations of states and actions in the examples.
  - Relational Kernels methods (Gartner et al., 2003a). They use the incrementally learnable 'Bayesian' regression algorithm *Gaussian processes* to approximate the mappings between *q-values* and *(state,action)* pairs. In order to employ *Gaussian processes* in a relational setting they use graph kernels as the covariance function between state-action pairs.
- *Relational learning of the optimal policy.* An important drawback of the previous methods is that the value function can be very hard to predict in

stochastic tasks. An alternative approach consist of directly learning the policies and only implicitly represent the value function. Note that this approach requires relational classifiers (like relational decision trees (Dzeroski et al., 2001)) rather than the regression learners used in the previous methods. The advantage of this alternative approach is that, usually, it is easier to represent and learn suitable policies for structured domains than to represent and learn accurate value functions. Figure 5.5 shows an example of relational decision tree that captures when the selection of the action *move(Block,Block)* is optimal for solving the set of tasks on(Block,Block) in the *Blocksworld* domain.

```
goal_on(A,B), numberofblocks(C),action_move(D,E).
above(D,A)?
+--yes: optimal
+--no: action_move(A,B)?
       +--yes: optimal
       +--no: nonoptimal
```

Figure 5.5: Relational decision tree for the goals *on(X,Y)* in the *Blocksworkd*.

Reinforcement Learning and learning for planning

The aims of RL are closely related to the aims of learning for AP. In fact, building a RL agent involves similar decisions to the ones in learning for AP such as, how to represent the agent's environment and actions; the agent's strategy for collecting the learning examples; the learning algorithm; and the exploitation of the learned knowledge.

Unlike most of the learning for AP approaches, RL present tightly integrated solutions for knowledge acquisition and knowledge exploitation. However, RL typically presents two kinds of shortcomings when addressing AP problems:

- *Scalability*. The space state of symbolic planning problems is normally huge. This space grows exponentially with the number of objects and predicates in the problem. Many RL algorithms, like *Q-learning*, require a table with one entry for each state in the state space limiting their applicability to AP problems. A solution to this limitation is using relational models which adopt the same state representation as symbolic planning, like done in RRL.

- *Generalization*. RL focuses learning on the achievement of particular goals. Each time goals change RL agents need to learn from scratch, or a *transfer learning* process at least. This is not exactly the case of RRL. Given that RRL represent goals using first order predicates, RRL agents can act in tasks with additional objects without reformulating their learned policies,

although additional training may be needed to achieve optimal (or even acceptable) performance levels. Even more, in the case of *model-based* RL, the agent learns a model of the environment and exploits it to learn policies more efficiently. The learned model is typically used to generate advise on how to explore or to plan trajectories so that the agent can obtain higher rewards. When *model-based* techniques are applied to RRL (Croonenborghs et al., 2007b) they produce symbolic action models similar to the ones learned in AP. In this case, the main difference comes from the fact that AP learns action models in standard planning representation languages and that the problem solving is carried out by off-the-shelf planners.

Despite the advances in RRL, applying RRL to planning problems is still an open issue. In complex planning tasks like the *Blocksworld* ones, RRL agents spend long time exploring actions without learning anything because no rewards (goal states) are encountered. By the time being the limitations of random exploration in RRL are relieved by two approaches. The first one uses traces of human-defined *reasonable policies* to provide the learning with some positive rewards (Driessens and Matwin, 2004). The second one applies transfer learning to the relational setting (Croonenborghs et al., 2007a).

To sum up, RRL focuses on learning policies for particular goals, like *on(X,Y)* for the *Blocksworld*, and fail to solve problems in which interacting goals have to be achieved, for instance building a tower of specific blocks *on(X,Y), on(Y,Z), on(Z,W)*. In this kind of problems, traditionally addressed in AP, the achievement of a particular goal may undo previously satisfied goals. This kind of goals must be attained in a specific order (as happens in the *Sussman's Anomaly*). As far as we know, none of the reported RRL approaches have mechanisms to automatically capture this knowledge about the interaction of goals.

5.3 Learning planning search control

Present algorithms for PUU fail to scale up: On the one hand, *dynamic programming* algorithms scale polynomially with the size of the state-space. In AP the state-space grows exponentially with the number of predicates defined in the domain. In addition, the state-space becomes larger with the presence of partial observability and/or non-determinism. On the other hand, heuristic search algorithms should scale better because they only process the more promising states of the state-space according to the heuristic function. Nevertheless, current heuristic functions for PUU are both misleading and expensive to compute.

Next, there is a review of the PUU techniques that improve their scalability through ML. These techniques adapt search control learning techniques for classical planning (discussed in Chapter 3) to the PUU framework. Basically, they adapt the mechanism for generating learning examples to the PUU framework:

- *Learning a generalized evaluation function* (Gretton and Thiébaux, 2004). Given a set of small probabilistic planning problems, this approach computes

the optimal policies for the problems with a MDP solver. Then, it uses a first-order regression to generalize a first-order value function over the values of optimal policies.

- *Learning a generalized policy*. Given a set of small probabilistic planning problems, this approach (Yoon et al., 2002) uses the probabilistic planner PGRAPHPLAN to synthesise plans $p = a_1, a_2, \ldots, a_n$ corresponding to 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. Next, plans p are simulated action by action. If the state resulting from simulating a_i is different from s_i, then a new plan is computed for reaching the goals in the new state. This process is repeated until a goal state is reached (or a limit of simulations is exceeded). Finally, a generalized policy is learned from the pairs $(state, action)$ traversed in the simulation. In many domains, solutions to small problems do not capture the singularities of larger problems. As an alternative for those domains, (Fern et al., 2006) learns from the pairs $(state, action)$ generated following the Approximate Policy Iteration (API) algorithm (Bertsekas and Tsitsiklis, 1996).

In contrast to RRL that captures how to reach a specific set of goals, the revised learning techniques capture how to solve any problem from a given domain. RRL requires learning from scratch, or at least a *transfer learning*,[2] each time the set of goals changes.

5.4 Learning planning domain models

This section revises the current ML techniques for the automatic definition of action models in environments with uncertainty. The review is organised regarding two dimensions: *determinism* and *observability*.

Action modelling in deterministic and fully observable environments

The techniques for learning deterministic action models in fully observable environments are discussed in Section3.4.

Action modelling in stochastic and fully observable environments

The first work in this category (Oates and Cohen, 1996) induces propositional rules corresponding to probabilistic effects of actions. Specifically, it explores a given AP domain by taking random actions and observing their execution. For each execution, it registers an observation $o_i = (s_i, a_i, s_{i+1})$; where s_i is a state, a_i is the action executed in s_i, and s_{i+1} is the resulting state of executing a_i in s_i. Finally, it

[2] *Transfer learning* is the use of data from one or more source tasks in order to learn a target task with less data or to achieve a higher performance level.

induces the effects of a given action by performing a general to specific Best-First search on the space of possible conditional effects of the action. This search is guided by the frequency of observations covered by the possible conditional effect.

The TRAIL system (Benson, 1997) learns relational models of probabilistic actions. TRAIL explores a given AP domain as follows: first it tries to solve a certain problem with the current action model. When this model is too scarce to solve the problem, it demands a solution plan to an external expert, it observes the execution of the plan and it learns a new model from these observations. In case the current action model allows TRAIL to synthesize a plan for the problem, TRAIL executes that plan and observes its execution to refine the action model. TRAIL action models are an extended version of Horn clauses so TRAIL makes use of standard ILP tools for the learning.

(Pasula et al., 2007a) learns probabilistic action models from externally provided observations consisting of tuples (s_i, a_i, s_{i+1}). The induced action models are probabilistic STRIPS operators extended in two ways: (1) they can refer to objects not mentioned in the action arguments, and (2) for each action, they add a *noise* outcome that groups the set of possible outcomes that are poorly probable for the action. The action modelling algorithm involves three layers of learning:

1. Learning the action structure. At this layer a greedy search process is performed in the space of action sets to select the action preconditions and outcomes that best fits the learning examples. An evaluation function which favors rule sets that assign high likelihood to the learning examples and penalizes model complexity (number of preconditions plus number of outcomes) guides the search.

2. Learning the action outcomes. Given an action and a set of learning examples, greedy search is used to decide the best set of outcomes and their corresponding parameters for the action. This learning is performed every time a new rule is constructed in the previous layer.

3. Parameter estimation. Given an action, the learning examples are used to estimate a distribution over action outcomes. In general, an action may have overlapping outcomes. This work estimates the outcomes parameters using the conditional gradient method. Note that this estimation is performed for each set of outcomes considered in the previous layer.

Action modelling in deterministic and partially observable environments

The ARMS system (Yang et al., 2007) learns PDDL planning operators from examples consisting of tuples (s_0, s_n, p); where s_0 is an initial state, s_n is a goal state and $p = (a_1, a_2, ..., a_n)$ is a plan corresponding to a sequence of state transitions $(s_0, s_1, ..., s_n)$. ARMS 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.

Alternatively, Eyal Amir 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.

Action modelling in stochastic and partially observable environments

In this scenario actions present stochastic effects and the observations of the world state may be incomplete and/or incorrect. Action modelling in stochastic and partially observable environments has been poorly studied and currently, there is no general strategy for addressing this modelling task. Preliminary work (Yoon and Kambhampati, 2007) addresses this problem using *Weighted Maximum Satisfiability* techniques for finding the action model that best explains the collected observations. Specifically, in order to learn the preconditions of every action a, this approach enumerates all the possible axioms of the form $a(x, y) : -p(x, y)$, with all the available predicates and variable matchings. Then, it updates the weights associated to the axioms according to a set of learning examples. Axioms corresponding to wrong predicates or unnecessary predicates would get close to zero weight after enough learning examples, and axioms corresponding to necessary predicates would get close to one. Likewise, for learning the effects of every action a, this approach enumerates all the possible effects of the form $a(x, y) : -e(x, y)$ and then updates the corresponding associated weights in a similar way.

This modelling task is related to a new area of AI called *Plan recognition*. *Plan recognition* is the task of identifying the plan performed by an actor from observing the actor's actions and their effects on the environment. (Bui and Venkatesh, 2002)

5.5 Discussion

While the first learning based planning systems are from the early 70's (Fikes et al., 1972) learning for PUU is a new field with interesting research opportunities:

- **There is little work on learning control knowledge for PUU**. Despite PUU algorithms present strong scalability problems, there is little work on learning search control for probabilistic planning and no work for contingent or conformant planning.

- **There is little work on action modelling for PUU**. There is a lot of research work done in learning actions models from observations in deterministic en-

vironments. But when the environment is stochastic, there are just few works that only can learn rough models assuming perfect observations of the environment. Besides contingent planning is poorly studied and therefore, there is no work on learning sensor models.

- **Model-Learning for PUU is off-line**. The existing techniques for learning model actions for PUU are not designed for planning and naturally acquiring knowledge as more experience is available. Indeed, *model-free RL* is a classic approach for the integration of learning and PUU. However, it has strong deliberative limitations since it does not deal with resources, duration, preferences. Besides, *RL* do not generalize the acquired knowledge. As a consequence, every time a new goal has to be achieved, new policies have to be learnt again, even if the dynamics of the environment did not change.

Moreover, the traditional problems of the integration of planning and learning are also present when learning for PUU: Both the selection of the right training problems and the features of the domains is also vital for learning significant knowledge in environments with uncertainty.

Part II

Integrating planning, execution and learning for planning under uncertainty

Chapter 6

Learning instances success for robust planning

This chapter describes an integration of planning, execution and learning to automatically capture the performance of instantiated actions (Jiménez et al., 2005a,b).

6.1 Introduction

The execution of a plan that theoretically solves a problem, can fail because special features of objects were not captured in the initial representation. These situations are found in many real-world domains, such as project management, workflow control, robotics and, generally, any domain where some agents perform some actions better than the rest. In this kind of domains, the success of a plan execution depends on how the plan actions are instantiated. However, AP action models assume that all the objects from one type behave exactly the same (unless explicitly specified in the domain theory with particular predicates).

This chapter presents an integration proposal for planning, execution and learning to automatically capture the performance of instantiated actions. To capture this knowledge, the system observes whether an action execution is successful or not and stores this knowledge in a table, called *robustness table*. To exploit the knowledge registered at the *robustness table*, the system defines control knowledge that decides the instantiation of the actions looking up in the *robustness table*. Figure 6.1 shows a high level view of this integration for planning, execution and learning to capture the success of instances. The following subsections describe each component of the integration in more detail.

6.2 Planning

The nonlinear backward chaining planner PRODIGY (Veloso et al., 1995) implements the planning component of this integration proposal. The inputs to PRODIGY

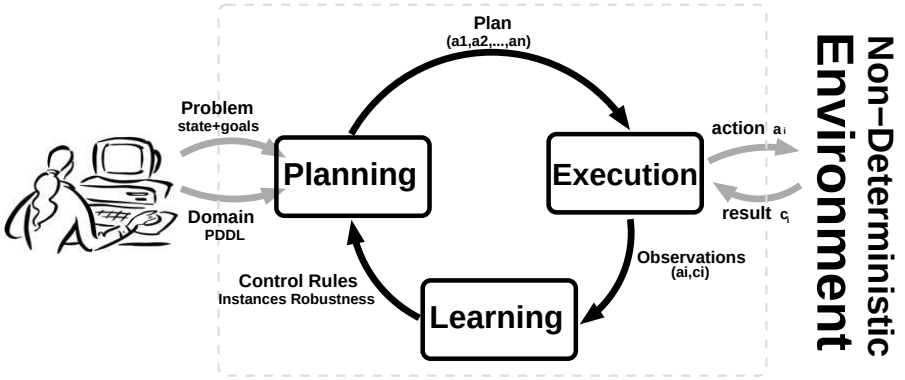

Figure 6.1: Architecture for learning instances success.

are the usual ones for a classical planner: a domain model and a problem definition. Additionally, PRODIGY accepts declarative control knowledge described as a set of control rules. These control rules act as domain dependent heuristics, and they are the main reason for using this planner, given that they provide an easy method for exploiting automatically acquired knowledge. As shown in Figure 6.2, the reasoning cycle of PRODIGY involves four decision points: (1) choosing a goal from the set of pending goals and subgoals; (2) choosing an operator to achieve a particular goal; (3) choosing the bindings to instantiate the operator and (4) choosing whether to apply an instantiated operator whose preconditions are satisfied or to continue subgoaling on another unsolved goal.

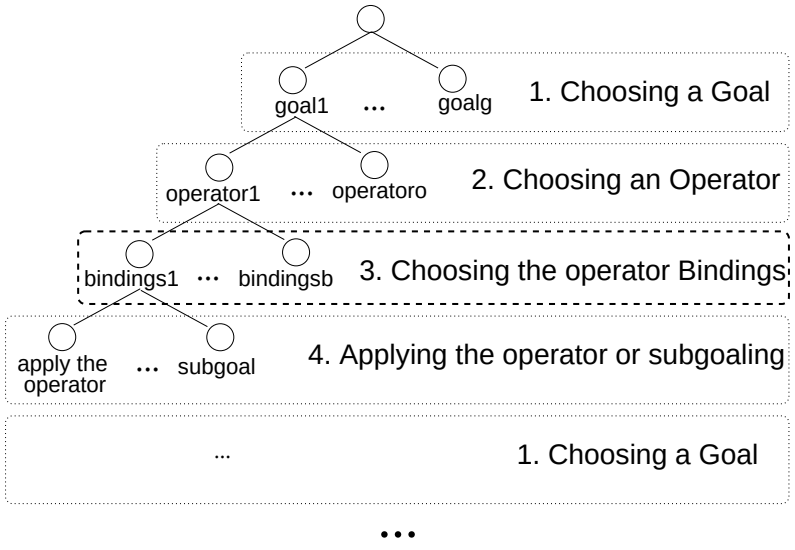

Figure 6.2: The reasoning cycle of the PRODIGY planner.

In this integration proposal, PRODIGY is **executed with control rules that make the planner prefer the most robust bindings for a given action** in order to guide the planner towards robust solutions. The output of PRODIGY, as it is used in this approach, is a total-ordered plan $p = (a_1, a_2, ..., a_n)$.

6.3 Execution

The execution component executes the plan $p = (a_1, a_2, ..., a_n)$ provided by the planning component for solving a certain problem. After the execution of an action a_i, the execution component invokes the learning component for updating the *robustness value* of action a_i. When the execution of an action a_i is a `FAILURE`, the execution component aborts the plan execution. The execution algorithm is shown in Figure 6.3.

Function Execution (Plan, RobTable):RobTable

Plan: list of actions (a_1, a_2, ..., a_n)
RobTable: Table with the robustness of actions

For all actions a_i in $Plan$ do
    $class = execution(ai)$
    $RobTable = Learning(ai, class, RobTable)$
    If $class = FAILURE$ Then break;
Return $RobTable$;

Figure 6.3: Algorithm for executing plans and updating the *robustness table*.

6.4 Learning

The learning component updates the *robustness table* which stores the estimations of the performance of actions. Specifically, the *robustness table* stores tuples of the form `(op-name, op-params, r-value)`, where `op-name` is the action name, `op-params` is the list of instantiated parameters of the action, and `r-value`, is the *robustness value* which indicates a measure of the probability of success of the instantiated action. Table 6.1 shows an example of *robustness table* for planning in the *Tourist* domain (Castillo et al., 2008). In this domain a tourist needs a plan for organizing his visits to the different places of a given city. This *robustness-table* captures which is the best day for a tourist to visit a fixed place. According to the table, the best day for visiting the `Prado` museum would be on Wednesday because this instantiation presents the highest associated *robustness value*.

The learning component updates the *robustness value* of a given action instantiation according to the following algorithm: when the action execution is successful, the *robustness value* of the action is increased. Otherwise, when the execution

Action	Parameters	Robustness
prepare-visit	(PRADO MONDAY)	5.0
prepare-visit	(PRADO TUESDAY)	6.0
prepare-visit	(PRADO WEDNESDAY)	**8.0**
prepare-visit	(PRADO THURSDAY)	4.0
prepare-visit	(PRADO FRIDAY)	2.0
prepare-visit	(PRADO SATURDAY)	1.0
prepare-visit	(PRADO SUNDAY)	1.0
prepare-visit	(ROYAL-PALACE MONDAY)	2.0
prepare-visit	(ROYAL-PALACE TUESDAY)	2.0
	...	

Table 6.1: Example of a *robustness-table* for the *Tourist* domain.

results in a FAILURE, the new *robustness value* is the square root of the old *robustness value*. In this way, we penalize actions with recent failures because we assume they will be more likely to fail in the next future. The learning algorithm for updating the *robustness table* is shown in Figure 6.4.

Function Learning (ai, r,RobTable):RobTable

ai: executed action
r: execution outcome (FAILURE or SUCCESS)
RobTable: Table with the robustness of actions

if $r = success$
  Then
    $robustness(ai, RobTable) = robustness(ai, RobTable) + 1$
  Else
    $robustness(ai, RobTable) = \sqrt{robustness(ai, RobTable)}$
Return $RobTable$;

Figure 6.4: Algorithm for updating the *robustness table* with a new execution.

6.5 Exploitation of the learned knowledge

We guide the search of the *PRODIGY* planner with control rules. Among all the possible action instantiations, these rules choose the instantiation with the greatest *robustness value* at the *robustness table*. An example of these control rules for the *Tourist* domain is shown in Figure 6.5. This control rule makes PRODIGY prefer the most *robust* day to prepare the visit of tourist <user-1> to the place <place-1>. Specifically, the most *robust* day is the day when visiting the <place-1> is more probable to please <user-1> according to the learned *robustness table*.

Suppose that a tourist called Mike wants to obtain a plan to visit the *Prado* museum and the *Royal Palace* in Madrid, Figure 6.6 shows the search tree of PRODIGY for this example, and how the control-rule of Figure 6.5 prefers to

```
(control-rule prefer-bindings-prepare-visit
 (IF
  (and (current-goal (prepared-visit <user-1> <place-1>))
       (current-operator prepare-visit)
       (true-in-state (current-time <user-1> <day-1> <time-1>))
       (true-in-state (current-time <user-1> <day-2> <time-2>))
       (diff <day-1> <day-2>)
       (more-robust-than (prepare-visit <user-1> <place-1>
                                        <day-1>)
                         (prepare-visit <user-1> <place-1>
                                        <day-2>))))
 (THEN prefer bindings ((<day> . <day-1>))((<day> . <day-2>))))
```

Figure 6.5: Control rule for preferring the best day to visit a museum.

prepare the visit to the *Prado* museum on Wednesday, among all the possible instantiations, because `PREPARE-VISIT PRADO WEDNESDAY 8.0` is the instantiation with the greatest robustness value in the *Robustness Table* for the *Prado* museum.

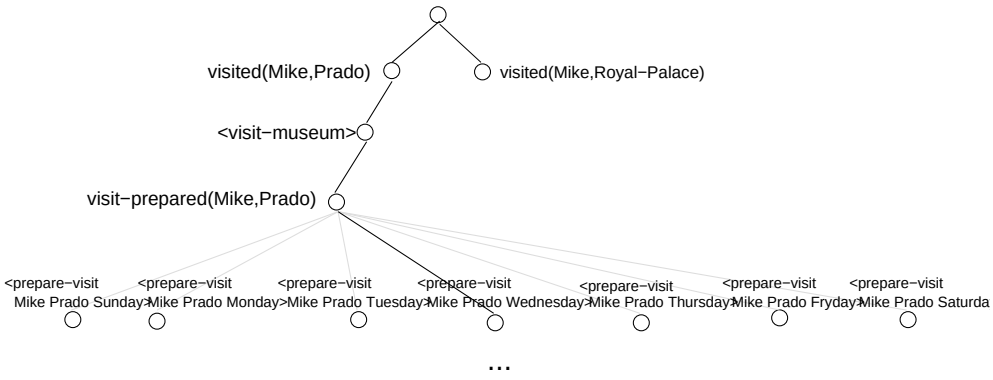

Figure 6.6: Example of PRODIGY planning guided by control rules.

If the system greedily exploits these control rules, the behavior of the different instantiations of actions are not explored. To avoid this effect we follow an *e-greedy* strategy: control rules for preferring the more robust bindings are only followed 80% of times. In the remaining 20% of times, the bindings are chosen randomly from among all the possible choices.

6.6 Evaluation

In this preliminary work, the execution of actions is simulated without considering the current state of the world. Specifically, the simulator only takes into account

the values of the action parameters for determining the class of an action execution (*success* or *failure*). Thereby, the input to the simulator is the action to execute and the output is whether the execution was a *success* or a *failure*. For every instantiated action a_i in the domain, the simulator has a *Bernoulli* distribution b_i that models the performance of a_i. To simulate the execution of a given instantiated action a_i, a random value is generated (*success* or *failure*) following the associated probability distribution b_i.

The domain

The architecture proposal is evaluated in the *Tourist* domain from the SAMAP project (Castillo et al., 2008). This planning domain consists of the following operators:

- `MOVE`, this operator computes the duration and cost of the tourist movements and updates the predicates current time, money-available and current location of the tourist according to these computations. For these experiments the path-planning problem of the domain is not considered. Specifically, it is a assumed that a tourist is always able to move between any two places in a city in one hour.
- There is a `VISIT-<PLACE>` operator for each possible type of place considered in the ontology. From visiting a museum, to eating in a restaurant or watching a film. The only difference between these operators is the type of place (`VISIT-MUSEUM`, `VISIT-RESTAURANT`, ...) and the partially instantiated goal added by the action. For example, action `VISIT-MUSEUM` adds `visited-museum`, action `VISIT-RESTAURANT` adds `visited-restaurant`, ...
- `PREPARE-VISIT`, this operator computes the preconditions and effects shared by all operators VISIT-PLACE, such as the time of the beginning of a visit, the price, the duration and the cost.

The problems

The test problem set consists of 100 random generated problems with increasing complexity. The initial state of the random problems represents the free time of the tourist for each day in the week, its available money and its initial location. The problem goals describe the places the tourist wants to visit. The complexity of the problems is defined in terms of the available time the user has to visit all the goals by the following ratio:

$$complexity = \frac{time_{goals}}{time_{available}}$$

Where $time_{available}$ represent the sum of the tourist free time and $time_{goals}$ represents the time needed to visit all the goals. So, when the complexity ratio of

a problem is over 1.0 the planner will not find a solution. Figure 6.7 shows how problems complexity affects the number of plan steps successfully executed.

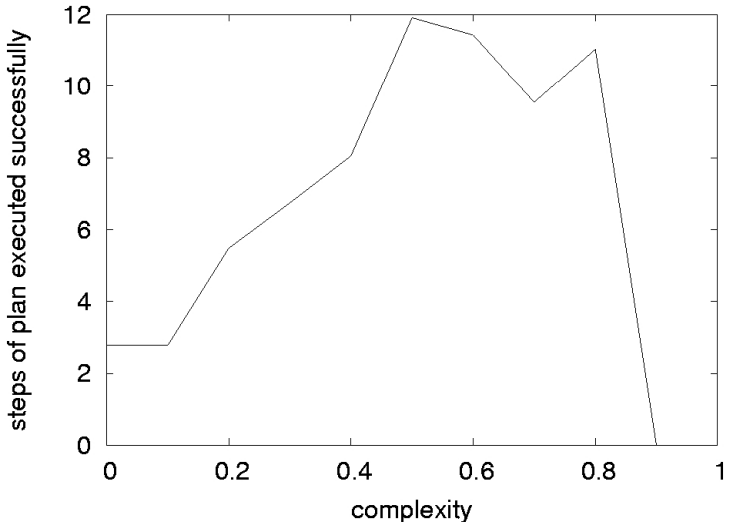

Figure 6.7: Analysis of the problems complexity in the *Tourist* domain.

Correctness of the learned knowledge

The first experiment aims to evaluate the correctness of the knowledge captured in the *robustness table*. In this experiment we measure the number of successfully executed actions as we solve 25 training problems. Results are shown in Figure 6.8. This number converges quickly to approximately 13-14 steps. The average length of the plans that solve the problems from the test set is 19,5. So, in terms of percentage, 13-14 steps executed successfully represents approximately 66-72% percentage of plan executed successfully. The speed of the convergence is due to the fact that failure probabilities do not change along the time.

Performance of the learned knowledge

The second experiment aims to evaluate the benefit of planning with the learned knowledge. In this experiment we solve 25 training problems and then we measure the usefulness of guiding the planning with the resulting *robustness table* and the control-rules for binding preference. Figure 6.9 shows a comparison of the behavior of two different planning configurations: (1) the planning system without control rules (*Default Behavoiur*) and (2) the planning system making use of the captured control knowledge (*With Control Rules*).

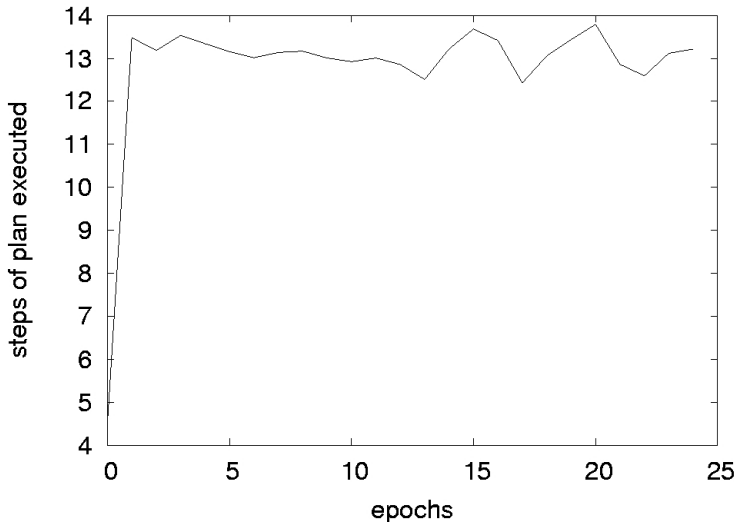

Figure 6.8: Evolution of the number of plan steps successfully executed.

6.7 Discussion

This chapter presented an architecture model to learn the success of instantiated actions. The applicability of this approach is limited for the following reasons:

1. Planning. The PRODIGY planner does not implement domain-independent heuristics for guiding the synthesis of plans. Consequently, PRODIGY is not competitive with the current state-of-the-art planners over a variety of domains.

2. Execution. The execution process is very simplified. The outcome of an action simulation is directly the class (*success* or *failure*). Normally, the execution module infers this class from the state of the environment.

3. Learning. The success of actions may depend on the state. However, this learning approach claims that the success of actions only depends on how actions are instantiated. Besides, the size of the *robustness-table* grows exponentially with the number of parameters of actions.

4. Exploitation of the learned knowledge. PRODIGY control rules guide locally the planning process. However, locally choosing the most robust action instantiation does not guarantee to find the most robust plan. In addition, the proposed definition of control rules is domain-dependent. Domain knowledge is necessary for determining which are the action parameters to select regarding the *robustness-table*, i.e. which is the `Then` part of the control rules. For example, in the *Tourist* domain, the parameter to select is the `<day>` to visit a given place.

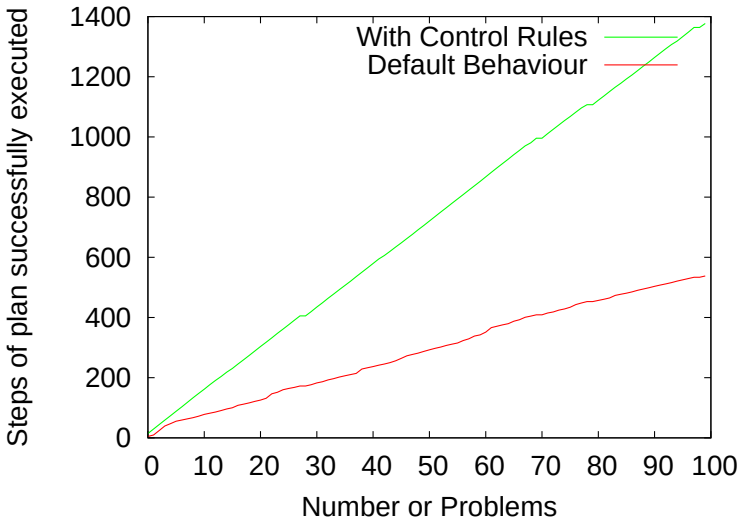

Figure 6.9: Plan steps successfully executed by the two planning configurations.

5. Integration. The proposed planning and learning integration is *'ad hoc'*. The *robustness* knowledge is captured in the form of control rules that are only useful for planning with PRODIGY.

Despite these limitations, the approach is able to capture execution knowledge when the performace of actions depends on the parameter instantiation. As shown experimentally, the proposed inclusion of execution knowledge makes PRODIGY prefer robust intantation of actions reducing failures in plan executions.

Chapter 7

PELA: Planning, Execution and Learning Architecture

This chapter describes the Planning, Execution and Learning Architecture (PELA) for robustly planning in stochastic environments (Jiménez et al., 2008, 2011b).

7.1 Introduction

AP algorithms reason about correct and complete action models for synthesising plans that attain a given set of goals. However, the specification of correct and complete action models is a complex task. When planning for environments with uncertainty, like the real-world, the task of action modelling becomes harder: even actions traditionally simple to code, like the classic four actions from *Blocksworld*, may produce countless outcomes.

As a consequence, an extended approach for addressing planning problems in domains with uncertainty consists of defining deterministic action models, obtaining plans with a classical planner and repairing these plans when necessary. Though this approach is frequently more practical, it presents two shortcomings:

- Classical planners miss execution trajectories. The classical planning action model only considers the nominal effects of actions. Thus, unexpected outcomes of actions, that may result in execution dead-ends or new planning opportunities, cannot be foreseen.

- 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.

A different approach for addressing deliberative tasks in domains with uncertainty consists of integrating planning, execution and learning processes for capturing the uncertainty of the environment interacting with it. At present, there are two areas in AI which study this issue with different views:

- Reinforcement Learning. RL (Kaelbling et al., 1996) provides a framework for integrating planning, execution and learning based on the strong theoretical foundations of MDPs. However, RL presents significant shortcomings for solving traditional AP problems. First, the scalability of the RL algorithms depends on the size of the state-space; in AP this size is easily huge. Second, RL suffers from generalization limitations. Given that RL focuses learning on the achievement of particular goals, each time goals change, RL requires learning from scratch, or at least a *transfer learning* process.

- Cognitive architectures. Architectures like PRODIGY (Veloso et al., 1995), SOAR (Rosenbloom et al., 1993) or more recently ICARUS (Langley and Choi, 2006), implement the integration of planning, execution and learning skills combining automated planners and rule-learning mechanisms for guiding them. These architectures scale better in the size of the state-space and are able to learn general knowledge that is valid for any planning problem within a given domain. Nevertheless, they are based on classical planning so they do not solve problems robustly under uncertainty. Besides, these architectures usually integrate specific tailored components that could not be replaced by other equivalent ones. Finally, unlike RL, they do no implement on-line learning mechanisms to incorporate knowledge as planning tasks are solved.

The work presented in this chapter belongs to the AI area of cognitive architectures. Specifically, the chapter describes PELA, an integration proposal of planning, execution and learning for stochastic environments.

7.2 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 7.1 shows the high level view of this integration proposal.

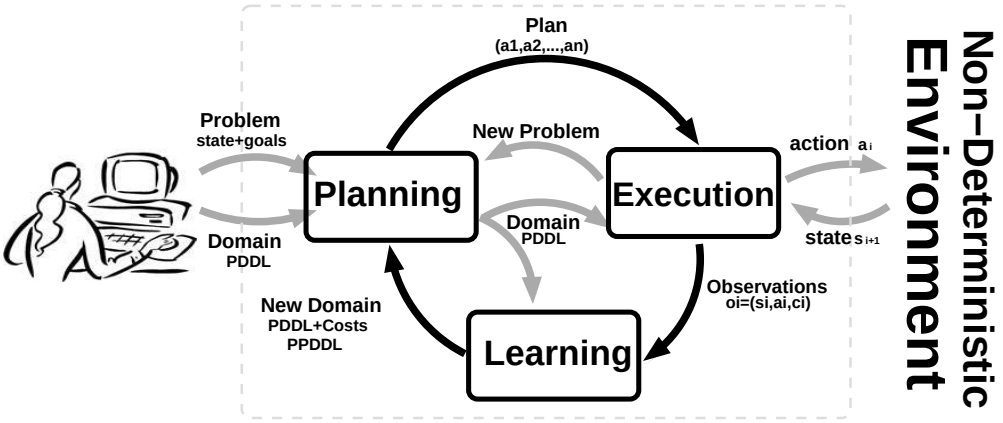

Figure 7.1: Overview of the planning, execution and learning architecture.

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

7.3 Planning

The inputs to the planning component are: a planning problem denoted by P together with 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 $eff(a) = (and(when\ c_1(and\ o_1 f_1)) \ldots (when\ c_k(and\ o_k f_k)))$ 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 $eff(a) = (probabilistic\ p_1\ o_1\ \ldots p_l\ o_l)$ 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 num-

ber 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.

7.4 Execution

The inputs to the execution component are provided by the planning component. These inputs are the total ordered plan $p = (a_1, a_2, ..., a_n)$ and the initial STRIPS-like action model A. 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 a plan p one action at a time. For each executed action a_i, 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 (`SUCCESS`, `FAILURE` or `DEAD-END`) 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) together with the STRIPS-like action model of $a_i \in A$.

We have devised two approaches for the execution component: a preliminary one and the current one.

7.4.1 Preliminary approach

The preliminary approach for the execution component (Jiménez, 2007) was only valid for domains free from execution dead-ends, e.g. the *Slippery-Gripper*.[1] According to this approach, the class c_i of an action a_i executed in a state s_i is:

1. `SUCCESS`. When s_{i+1} matches the STRIPS model of a_i defined in A. That is, when it is true that $s_{i+1} = \{s_i / Del(a_i)\} \cup Add(a_i)$.
2. `FAILURE`. When s_{i+1} does not match the STRIPS model of a_i defined in A. That is, when it is not true that $s_{i+1} = \{s_i / Del(a_i)\} \cup Add(a_i)$.

[1] The *Slippery-gripper* domain is a non-deterministic *Blocksworld* with a gripper to manipulate the blocks and a nozzle to `paint` them. Painting a block may wet the gripper, which makes it more likely to fail. The gripper can be dried to move blocks more safely.

Figure 7.2 shows two execution episodes in the *Slippery-Gripper* domain with the same s_i and a_i, but with different s_{i+1}. The first execution corresponds to an execution classified as a SUCCESS and the second one to an execution classified as a FAILURE.

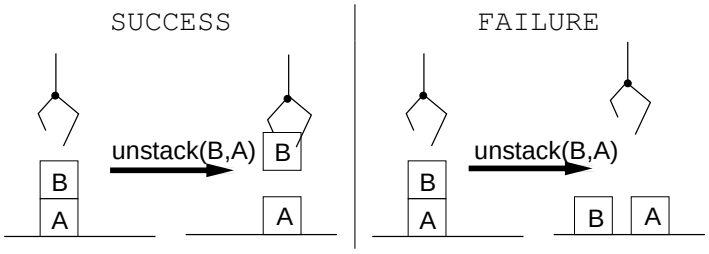

Figure 7.2: Two execution episodes in the *Slippery-Gripper* domain.

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 is classified as a FAILURE, the execution module invokes the planning component with s_{i+1} as the new initial state of the problem so the planning component can replan for solving the planning problem in this new state. This execution algorithm is shown in Figure 7.3.

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
    $a_i = Pop(Plan)$
    $newstate = executes(state, a_i)$
    if $matches(state, newstate, a_i, Domain)$
        $Observations = collectObservation(Observations, state, a_i, SUCCESS)$
    else
        $Observations = collectObservation(Observations, state, a_i, FAILURE)$
        $Plan = replan(newstate)$
    $state = newstate$
Return $Observations$;

Figure 7.3: Executes a plan and classifies actions as SUCCESS or FAILURE.

7.4.2 Current approach

This implementation of the execution component extends the previous one for handling domains with execution dead-ends, e.g., the *tireworld*.[2] In this new implementation of the execution component, the class c_i of an action a_i executed in a state s_i is:

1. `SUCCESS`. When s_{i+1} matches the STRIPS model of a_i defined in A. That is, when it is true that $s_{i+1} = \{s_i / Del(a_i)\} \cup Add(a_i)$.

2. `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}.

3. `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 7.4 shows three execution episodes of the action `move-car` from the *tireworld*: 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.

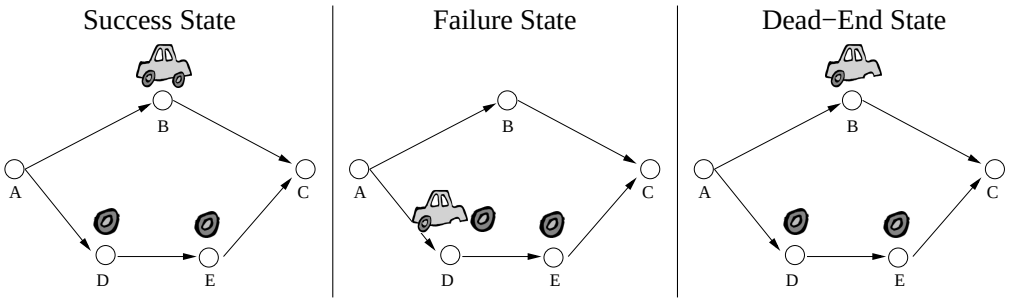

Figure 7.4: Execution episodes for the `move-car` action in the tireworld.

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

[2] In the *tireworld* a car needs to move from one location to another. The car can move between different locations via directional roads. For each 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 leads to execution dead-ends.

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. The extended algorithm for the execution component is shown in Figure 7.5.

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
    $a_i = Pop(Plan)$
    $newstate = executes(state, a_i)$
    if $matches(state, newstate, a_i, Domain)$
        $Observations = collectObservation(Observations, state, a_i, SUCCESS)$
    else
        $Plan = replan(newstate)$
        If $Plan$ is not $\emptyset$
            $Observations = collectObservation(Observations, state, a_i, FAILURE)$
        else
            $Observations = collectObservation(Observations, state, a_i, DEADEND)$
    $state = newstate$
Return $Observations$;

Figure 7.5: Extended execution algorithm for domains with dead-ends.

7.5 Learning

The learning component searches for rules that generalize the observed performance of actions. Then, it compiles these rules together with 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'. We have also devised two approaches for implementing the learning component.

7.5.1 Preliminary approach

Again, the preliminary approach (Jiménez and Cussens, 2006) is only valid for domains free from execution dead-ends. For each action $a \in A$, the learning component induces a Stochastic Logic Program (SLP) that represent its state-dependent

probability of success. The induction of the SLP is performed in a two-step process: A first step for inducing state-dependent rules about the success of a (structural learning) and a second step for estimating the probabilities associated to the induced rules (parameter estimation).

Structural learning of action success rules

The structural learning is performed by the ALEPH[3] system. ALEPH heuristically searches for *Horn clauses* that explain as many *positive examples* of the target concept as possible, covering the least possible amount of *negative examples*. The heuristic search implemented in ALEPH is robust to noisy learning examples, and supports declarative *background knowledge* in the form of PROLOG programs to improve its efficiency.

For each action $a \in A$, ALEPH induces rules that capture the context in which the action succeeds. The induced rules are *Horn clauses*, where the head of the clause is the target concept *"success-actionName(actionParameters)"* and the body of the clause is the set of predicates that describe the context in which the action succeeds. Additionally, ALEPH indicates the number of positive and negative examples covered by a rule. As an example, Figure 7.6 shows a rule induced by ALEPH for the action `unstack(block,block)` of the *Slippery-Gripper* domain.

```
[Rule 1]
[Pos cover = 22 Neg cover = 27]
success_unstack(A,B,C) :- drygripper(A).
```

Figure 7.6: Rule induced by ALEPH for action `unstack(block,block)`.

To induce the success rules of a given planning action, the inputs to the ILP system ALEPH are:

- *The language bias*. This bias is automatically extracted from the domain model and consists of PROLOG clauses that define the object types, the domain predicates and the learning target concept. Domain predicates are extended with an extra parameter called *example* for indicating the identifier of the observation in the learning examples. Figure 7.7 shows the language bias for learning the target concept of `success-unstack(block,block)` in a two-blocks *Slippery-Gripper* domain.

- *The knowledge base*. This knowledge is a set of PROLOG ground clauses encoding the observations collected by the execution component. Each ob-

[3] ALEPH is an ILP system based on Stephen Muggleton's ideas of inverse entailment (Muggleton, 1995a) developed by Ashwin Srinivasan and Rui Camacho. ALEPH can be freely downloaded at http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph

```
% Types
 type_of_object(blockA,block).
 type_of_object(blockB,block).
% Predicates
 modeb(*,drygripper(+example)).
 modeb(*,wetgripper(+example)).
 modeb(*,emptygripper(+example)).
 modeb(*,ontable(+example,-block)).
 modeb(*,clear(+example,-block)).
 modeb(*,holdinggripper(+example,-block)).
 modeb(*,on(+example,-block,-block)).
% Target concept
 modeh(*,success_unstack(+example,+block,+block)).
```

Figure 7.7: Language bias for the operator `unstack(block,block)`.

servation is encoded by: one clause indicating the action executed together with a set of clauses capturing the state in which the action was executed. Figure 7.8 shows a piece of knowledge base that encodes the observations corresponding to the two execution episodes depicted at Figure 7.2. The first observation (with identifier `o1`) corresponds to an execution classified as `SUCCESS` and hence, it will be used as a positive example for learning the success rules of action `unstack(block,block)`. On the contrary, the second observation (with identifier `o2`) corresponds to an execution classified as `FAILURE` and hence, it will be used as a negative example for learning the success rules of action `unstack(block,block)`.

```
% Observation o1
 success_unstack(o1,blockB,blockA).
% State o1
 drygripper(o1).
 emptygripper(o1).
 on(o1,blockB,blockA).
 ontable(o1,blockA).
 clear(o1,blockB).
```

```
% Observation o2
 success_unstack(o2,blockB,blockA).
% State o2
 wetgripper(o2).
 emptygripper(o2).
 on(o2,blockB,blockA).
 ontable(o2,blockA).
 clear(o2,blockB).
```

Figure 7.8: Learning examples for the action `unstack(block,block)`.

Parameter estimation of action success rules

At this step, the probability associated to the induced rules is estimated through *Maximum Likelihood Estimation* (MLE). In this approach, MLE is implemented

with a PRISM[4] program that models the classification of the observations as positive or negative examples of the induced success rules. Particularly, the PRISM program assumes the following generative model for the classification of the observations:

1. An unclassified observation is selected with a probability equal to its relative frequency in the set of observations.
2. A rule is chosen according to an unknown probability over the induced rules.
3. When the chosen rule does not cover the selected observation, we have a failure and the process returns to step 2. Otherwise, it continues.
4. The selected observation is finally classified as positive or negative according to an unknown probability associated with the chosen rule.

The application of MLE over this PRISM program simultaneously estimates the probabilities of choosing a given induced rule (step 2) and the classification probability of the rules (step 4). This estimation is a missing data problem, since we do not know which rule classified each observation. In such situations, PRISM handles MLE using the *Expectation-Maximization* algorithm (EM). The missing data includes an unknown number of failures at step 3 so *Failure-Adjusted Maximization* (FAM) (Cussens, 2001), a particular version of EM, is used.

Figure 7.9 shows the final SLP induced for the `unstack` action from the slippery-gripper domain consisting of one rule. The head of the rule represents the target concept (in the example the success of the action `unstack`) and the body of the rule represents the set of predicates describing the conditions that make the target concept true. The parameter of the rule represent the probability of success of the action given that the body of the rule is true in the current state. This SLP states that when executing the action `unstack(block,block)` with the gripper `dry`, the action is going to succeed 80% of times.

```
0.8 : unstack(A,B,C) :- drygripper(A).
```

Figure 7.9: SLP induced for action `unstack` from the *Slippery-gripper* domain.

7.5.2 Current approach

The following rule learning approach is valid for capturing the performance of actions in domains with presence of execution dead-ends, like the *tireworld*.

For each action $a \in A$, PELA learns a relational decision tree t_a that models the performance of a in terms of these three classes: *success*, *failure* and *dead-end*.

[4]PRISM (Sato and Kameya, 2001) is a general programming language intended for symbolic-statistical modelling. PRISM can be freely downloaded at http://mi.cs.titech.ac.jp/prism/

PELA uses TILDE[5] for the tree learning but there is nothing that prevents from using any other relational decision tree learning tool. Given that rules captured in a decision tree are disjunctive, this approach does not require the parameter estimation step. Each branch of the learned decision tree will represent a rule of performance of the corresponding action:

- *the internal nodes* of the branch contain 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. The estimation of the probability associated to the rules is directly extracted from the frequency of the learning examples.

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

Figure 7.10 shows the decision tree learned 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 action definition and (2) the types of the rest of literals are extracted from the predicates definition. As in the previous rule learning approach, 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 7.11 shows the language bias specified for learning the model of performance of action `move-car(Origin,Destiny)` from the *tireworld*.
- *The knowledge base*, specifying the set of examples of the target concept, and the background knowledge. In PELA, both are automatically extracted

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

```
% 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 7.11: Language bias for the *tireworld*.

from the classified executions 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 7.12 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 7.4.

```
% 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 7.12: Knowledge base after the executions of Figure 7.4.

7.6 Exploitation of the learned knowledge

PELA compiles the STRIPS-like action model A together with the learned knowledge into an upgraded action model A'. 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 compilations.

7.6.1 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 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 7.10 there is only one test, referring to `spare-in(A,C)`);
   (b) E_j=(and {effects(a) $\cup$ (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 7.13 shows the result of compiling the decision tree of Figure 7.10. 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 7.13: Compilation into a metric representation.

7.6.2 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 7.14 shows the result of compiling the decision tree of Figure 7.10 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 7.14: Compilation into a probabilistic representation.

7.7 Evaluation

In 2004, Michael Littman and Hakan Younes created the probabilistic track of IPC with the aim of evaluating the existing techniques for probabilistic planning. This track defined an evaluation methodology for probabilistic planners consisting of:

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

- A simulator of stochastic environments. *MDPsim* [6] 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 interchange 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. Figure 7.15 illustrates the interchange of messages between a probabilistic planner and the simulator *MDPSim*.

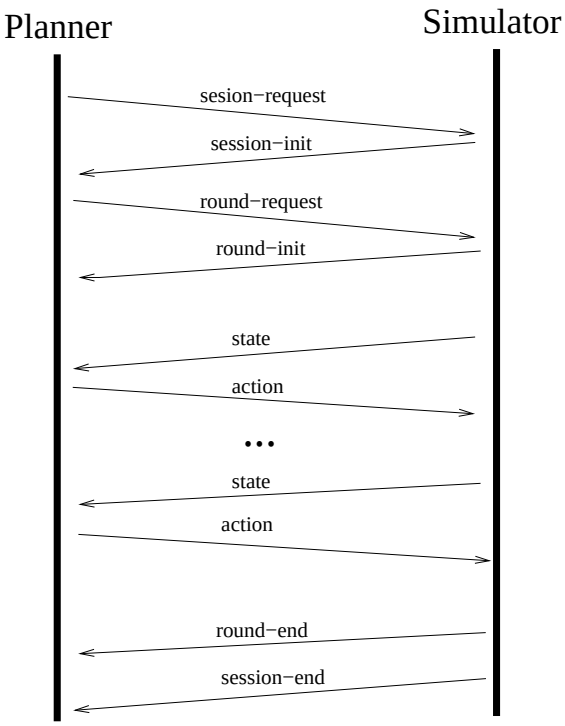

Figure 7.15: Interchange of messages between a planner and *MDPSim*.

- A performance measure. Probabilistic planners are evaluated regarding these metrics:

  1. Number of problems solved. The more problems a planner solves the better the planner performs.
  2. Time invested to solve a problem. The less time a planner needs the better the planner performs.

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

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, comparing probabilistic planners with this metric is tricky because in some domains, e.g., *triangle tireworld*, robust plans are longer than fragile plans.

We use the methodology defined at the probabilistic track of IPC to evaluate PELA. Specifically, PELA is integrated with *MDPSim* as follows: Both PELA and *MDPSim* share the same problem description. 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. On the other hand, *MDPSim* simulates the execution of actions according to a PPDDL model of the environment. Figure 7.16 illustrates the integration of PELA with the *MDPSim* simulator.

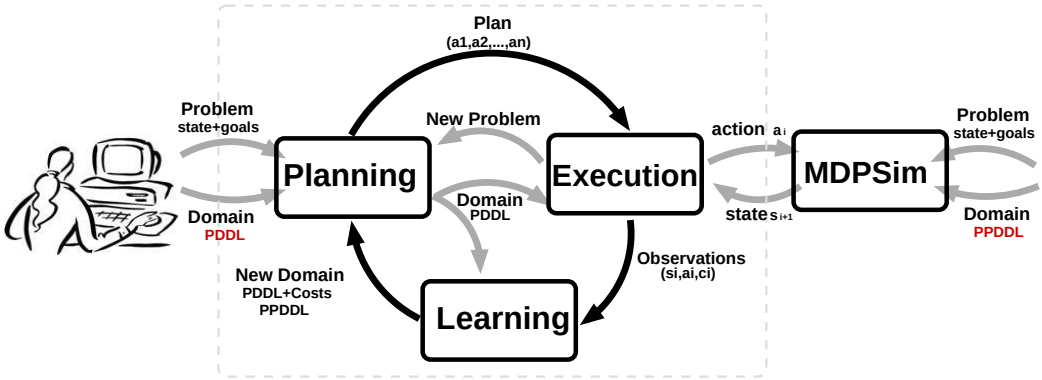

Figure 7.16: Integration of PELA with the *MDPSim* simulator.

7.7.1 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 thesis we propose to artificially increase the *robustness diversity* of a classical planning domain following any of the posed 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 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., 2007b). 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 7.17 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 sense, 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.

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

Figure 7.17: Topology of the domains chosen for the evaluation of PELA.

7.7.2 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 7.7.3. 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 7.7.4.

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[7]. 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.

[7]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.

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

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 7.18 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.

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.

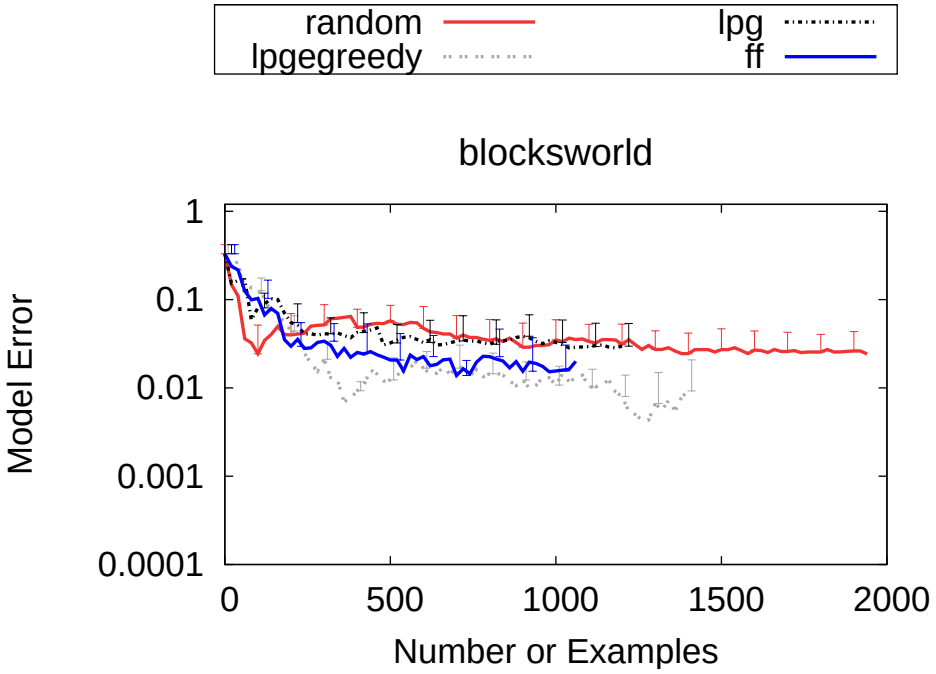

Figure 7.18: Error of the learned models in the *Blocksworld* domain.

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 7.19 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.

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-εGreedy* 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 7.20 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

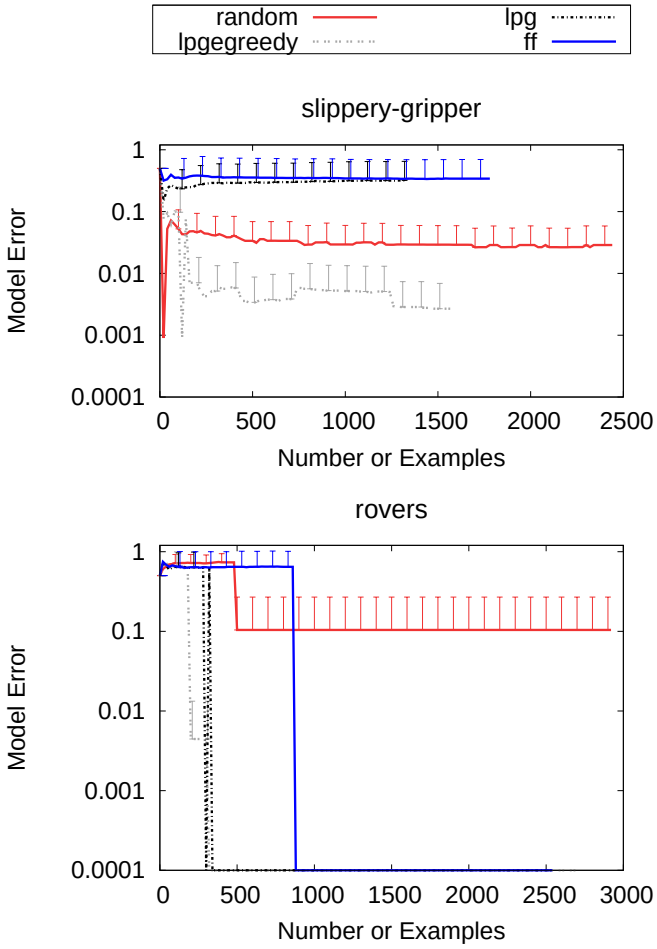

Figure 7.19: Model error in the *Slippery-gripper* and *Rovers* domains.

products and six stacks. The training set for the *Triangle-tireworld* consisted of one hundred problems of size five.

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 7.21 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 train-

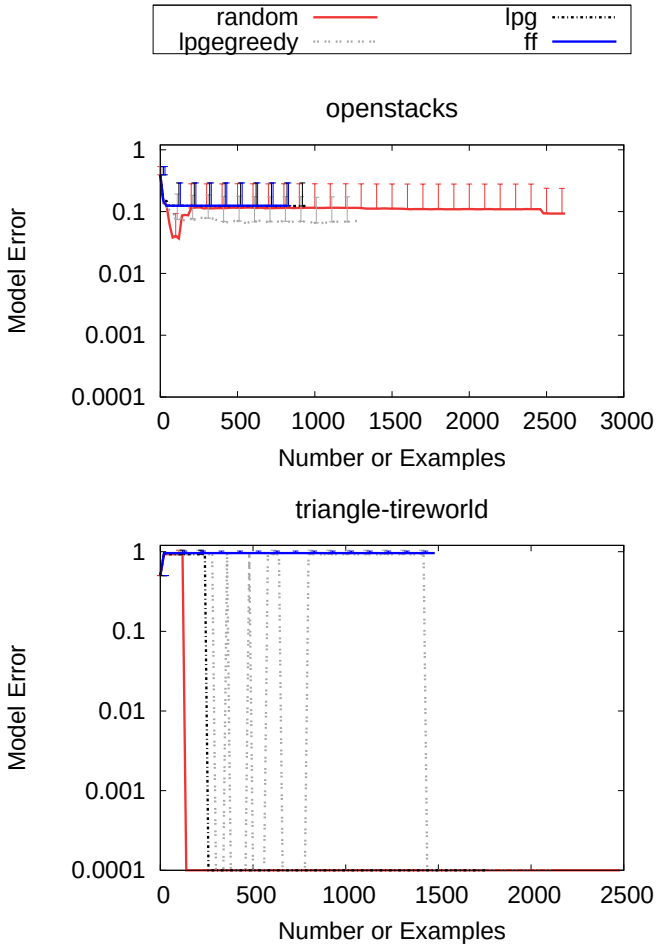

Figure 7.20: Model error in the *Openstacks* and the *Triangle-tireworld* domains.

ing 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. 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 *LPGεgreedy*) 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

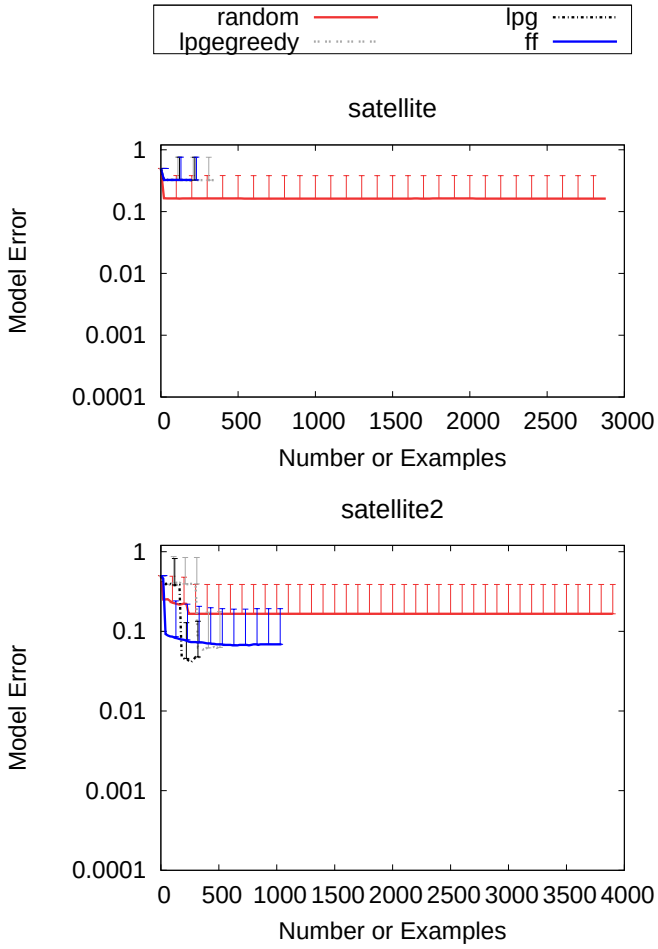

Figure 7.21: Error of the learned models in the *Satellite* domain.

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.

7.7.3 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., 2007a) 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 7.6.1).

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 7.6.2).

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. 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 7.22 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.

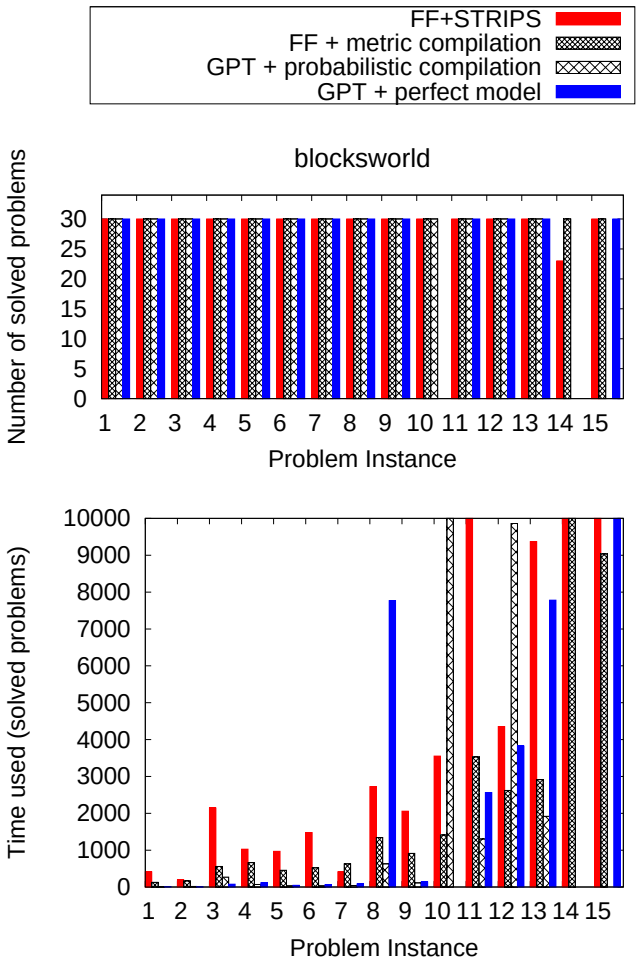

Figure 7.22: Off-line performance of PELA in the *Blocksworld*.

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 7.23 shows the results obtained by the four planning 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.

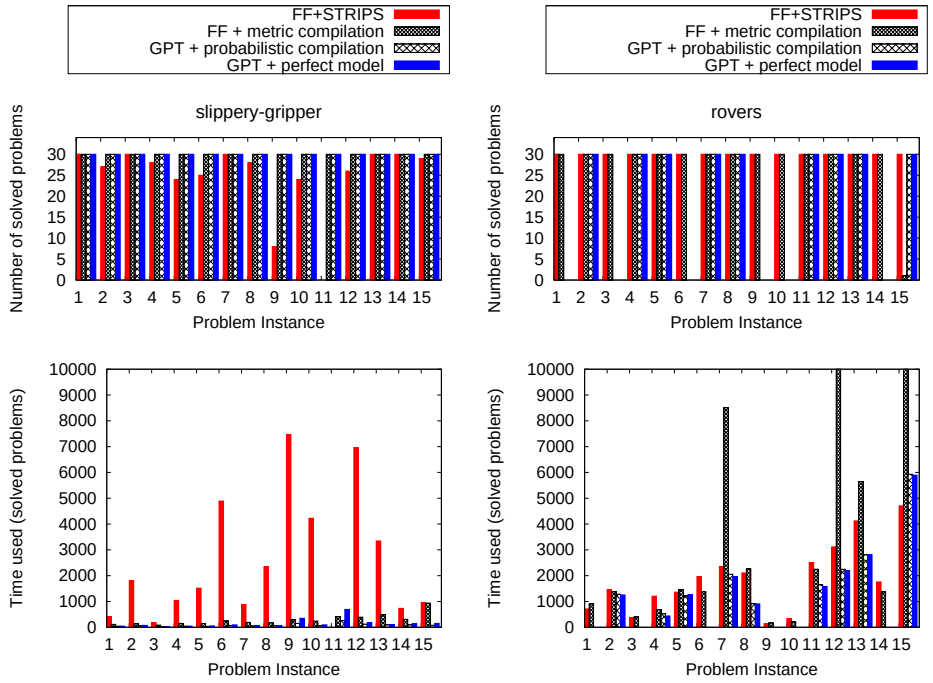

Figure 7.23: Off-line performance of PELA in the *Slippery-gripper* and the *Rovers* domains.

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 7.24 shows the results obtained in the *Openstacks* domain.

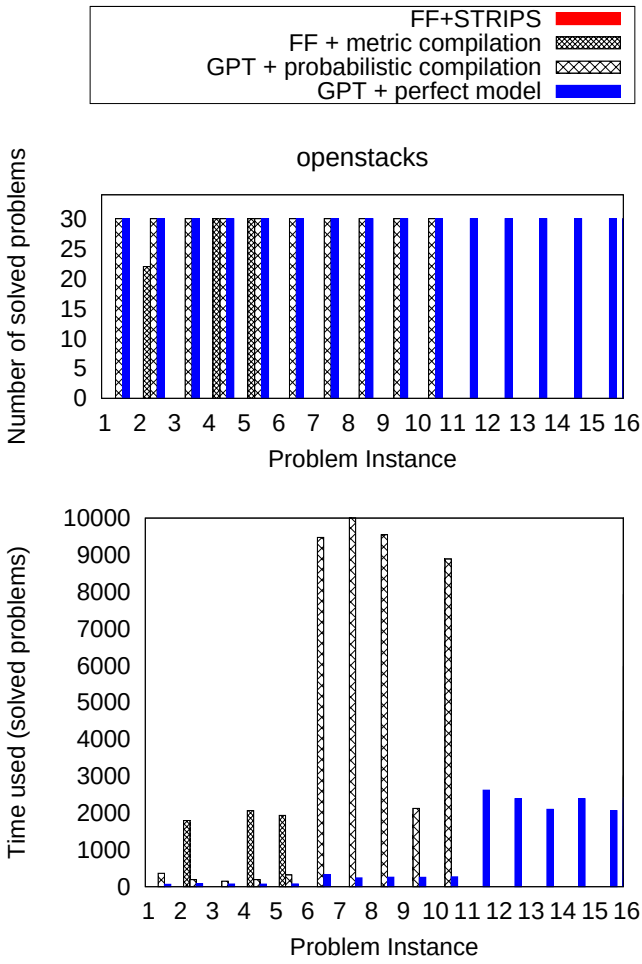

Figure 7.24: Off-line performance of PELA in the *Openstacks* domain.

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 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 7.25 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.

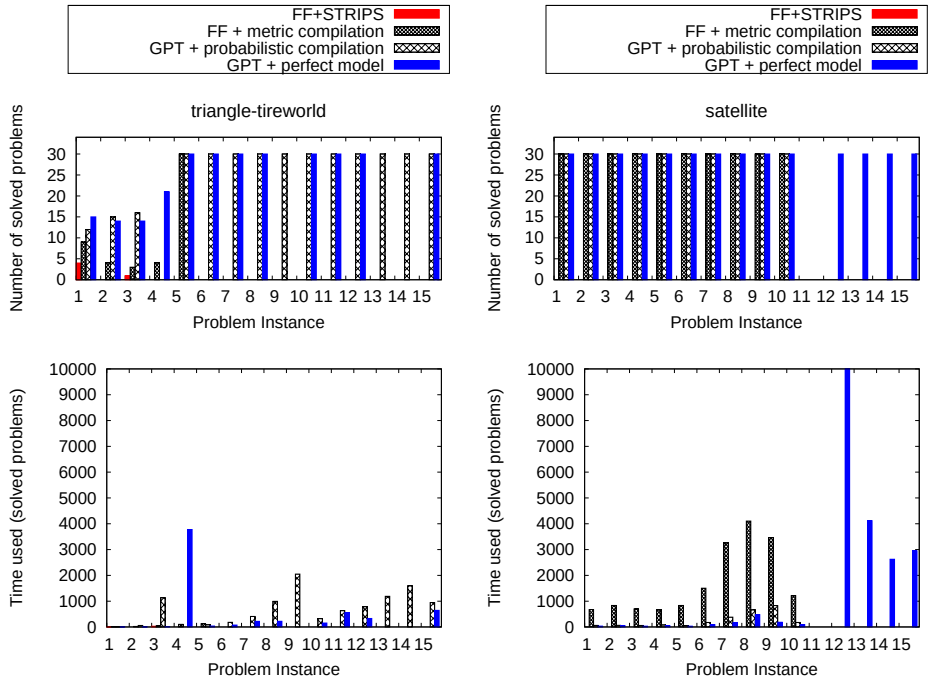

Figure 7.25: Off-line performance of PELA in the *Triangle-tireworld* and *Satellite* domains.

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 7.26 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 7.27 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 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*	*FF+ prob. model*	*GPT*
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

Figure 7.26: Summary of the problems solved by the off-line configurations of PELA.

	Planning Time of Problems Solved (seconds)			
	FF	*FF+ metric model*	*FF+ prob. model*	*GPT*
Blocksworld	78454.6	35267.1	26389.4	38416.7
Slippery-Gripper	36771.1	4302.7	1238.3	2167.1
Rovers	28220.0	349670.0	18635.0	18308.9
OpenStacks	0.0	8465.3	33794.6	12848.7
Triangle-tireworld	34.0	306.0	10390.1	6034.1
Satellite	0.0	17244.1	2541.3	21525.9

Figure 7.27: Summary of the planning time of the four 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 7.28 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 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.

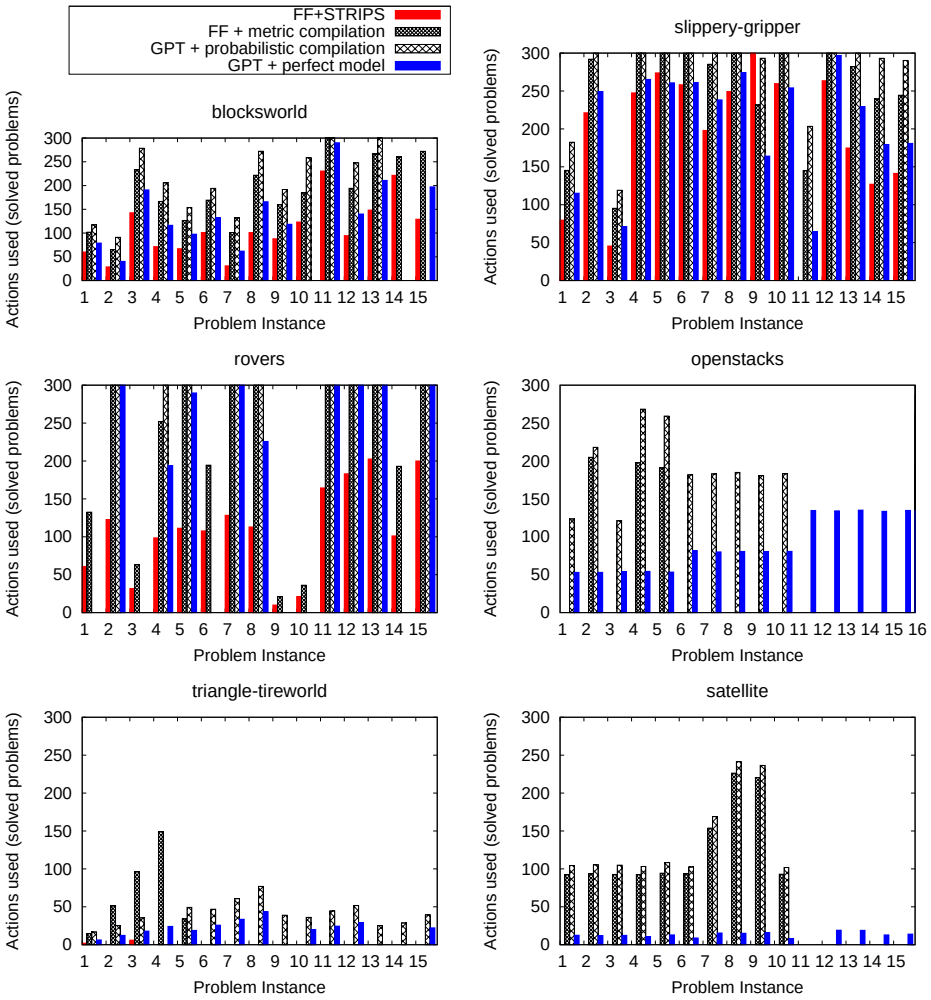

Figure 7.28: Actions used for solving the problems by the off-line configurations of PELA.

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

In the *blocksworld* the five online configurations of PELA achieve action models able to solve the test problems faster than the classical replanning approach. In particular, except the pure random configuration (*FF-εGreedy0.0*), 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, is not situation-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. Figure 7.29 shows the results obtained by the online configurations of PELA in the *Blocksworld*.

In the *slippery-gripper* domain the convergence of the PELA configurations is slower. In fact, the *FF-εGreedy1.0* 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 situation 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. Figure 7.30 shows the results obtained by the online configurations of PELA in the *Slippery-gripper* domain.

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 7.31 shows the results obtained by the online configurations 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 con-

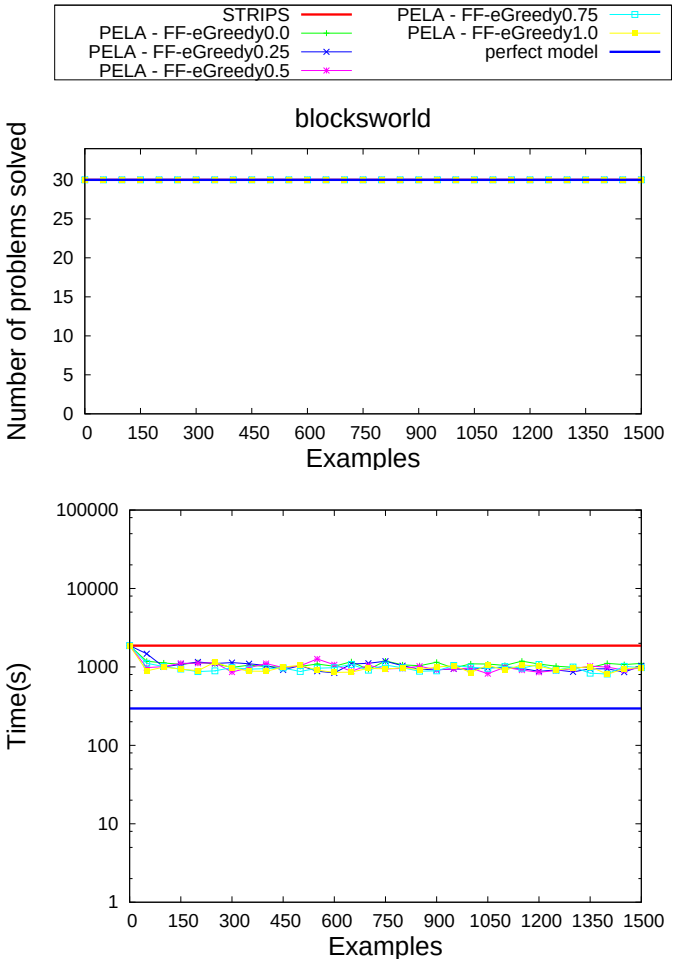

Figure 7.29: Planning in the *Blocksworld* with models learned on-line by PELA.

trary, all the online 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 spend more time than planning with the perfect models because they are used in a replanning approach. Figure 7.32 shows the results obtained by the online configurations of PELA in the *Openstacks* domain.

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. Figure 7.33 shows the results obtained by the online configurations of PELA in the *Triangle-tireworld* domain.

In the *satellite* domain only the *FF-εGreedy1.0* and *FF-εGreedy0.75* configu-

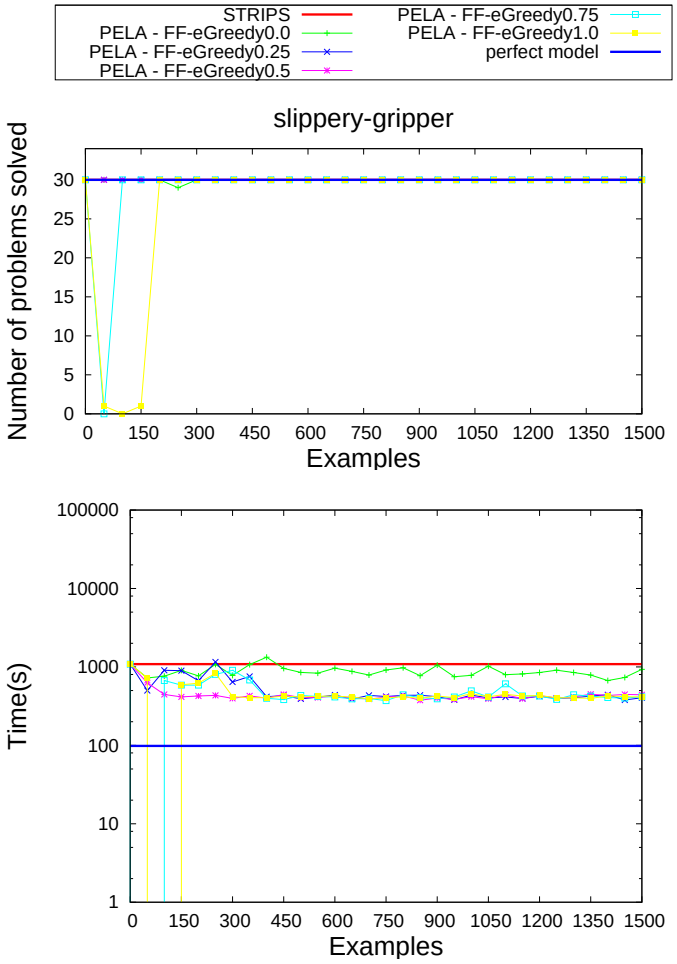

Figure 7.30: Planning in the *Slippery-gripper* with models learned on-line by PELA.

ration are 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 7.34 shows the results obtained in the *Satellite* domain.

Overall, given that the upgrade of the action model performed by PELA does not affect to actions causality, the PELA on-line configurations assimilate execution knowledge without degrading the performance of a classical replanning approach. 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

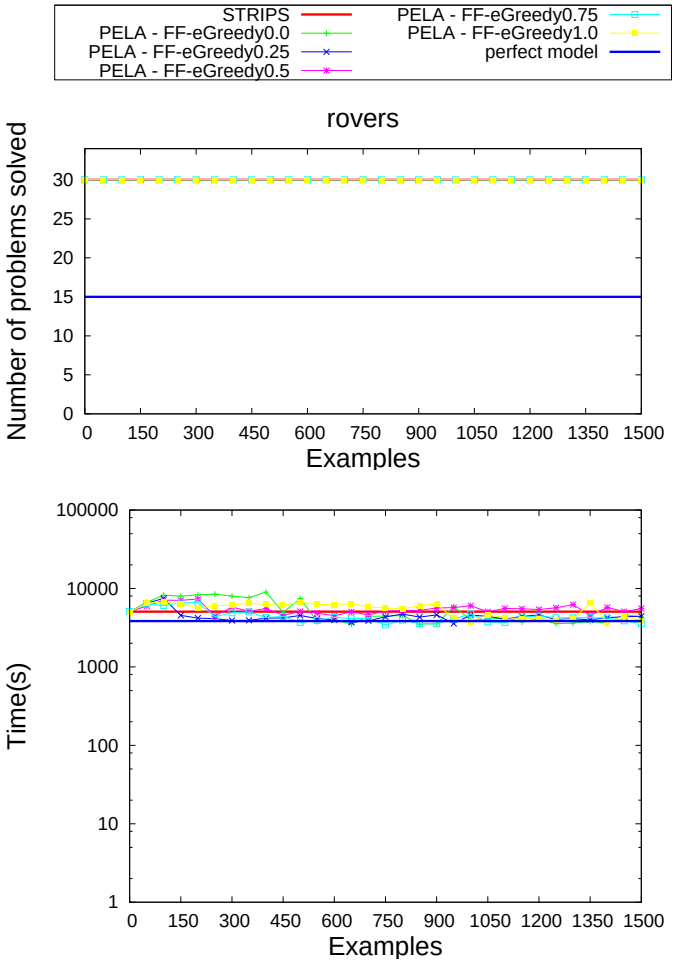

Figure 7.31: Planning in the *Rovers* with models learned on-line by PELA.

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 7.35 summarizes the number of problems solved by the four 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 7.36 summarizes the results obtained in terms of computation time in the solved problems by the four plan-

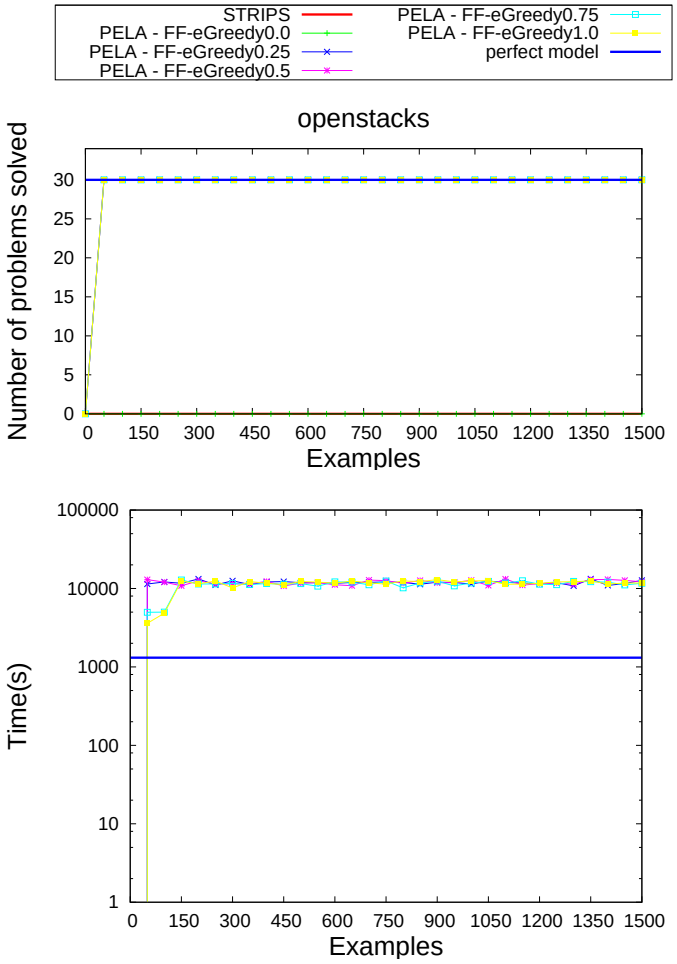

Figure 7.32: Planning in the *Openstacks* with models learned on-line by PELA.

ning 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.

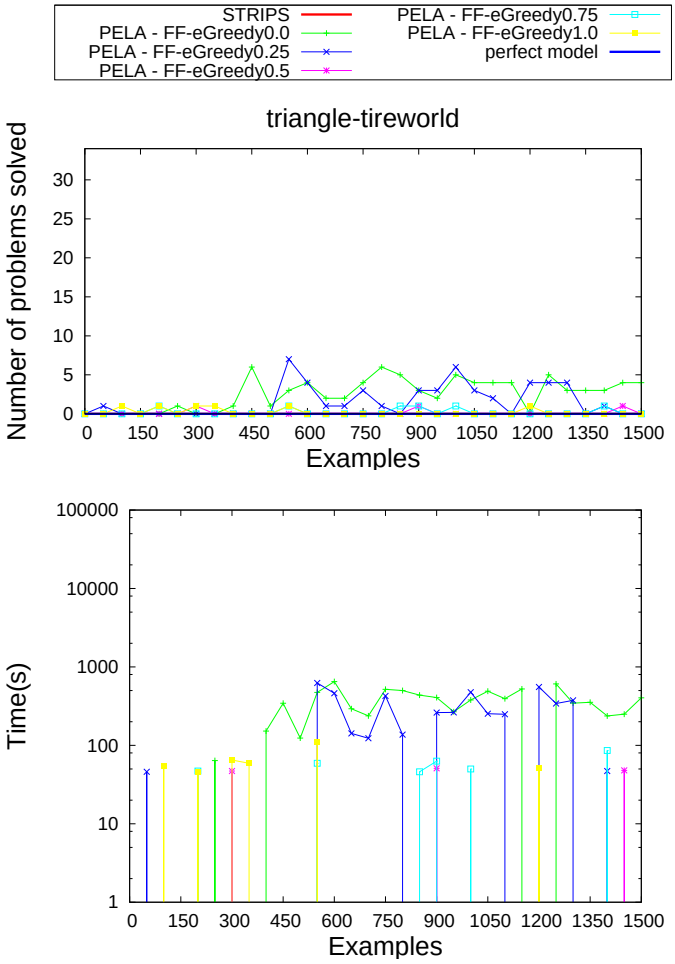

Figure 7.33: Planning in the *Triangle-tireworld* with models learned on-line by PELA.

7.8 Discussion

This chapter described the PELA architecture for robust probabilistic planning. In order to achieve robust plans within stochastic 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 to the actions causality and hence, it is suitable for on-line integrations 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 sources.

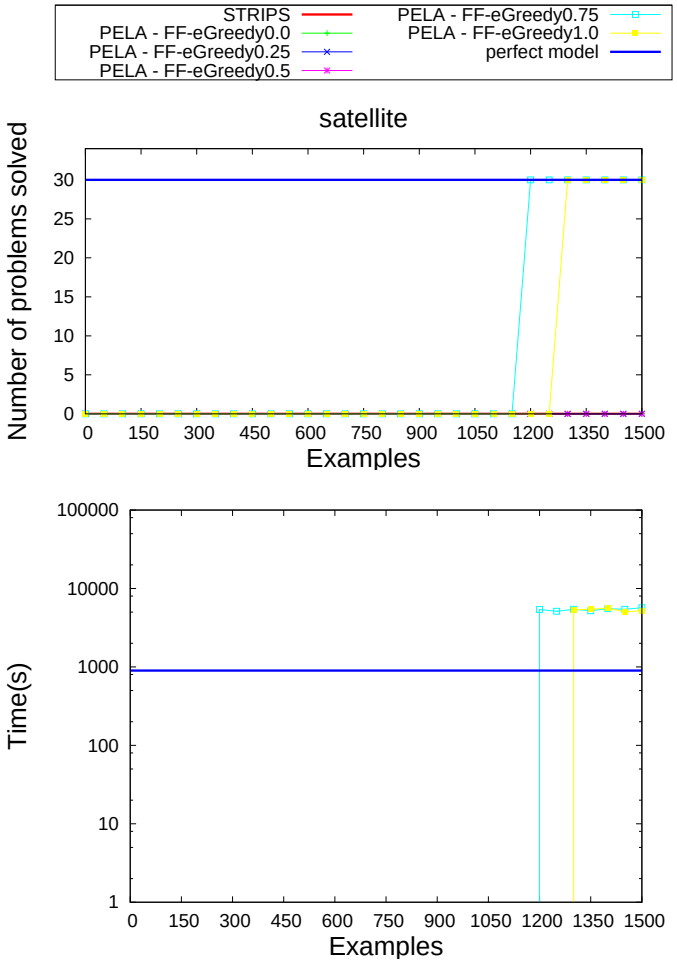

Figure 7.34: Planning in the *Satellite* with models learned on-line by PELA.

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 AP domains improve the accuracy of the learned models because they explore the performance of actions under diverse contexts. However, pure random explorations are unappropiate for AP 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 εgreedy strategies) can provide diversity to the learning examples, as well as they consider causal dependencies of actions.

- The *off-line performance* experiments showed that the action models learned

	Number of Problems Solved at the end of the online process						
	Strips	*εG0.0*	*εG0.25*	*εG0.5*	*εG0.75*	*εG1.0*	*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

Figure 7.35: 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	*εG0.0*	*εG0.25*	*εG0.5*	*εG0.75*	*εG1.0*	*Perfect model*
Blocksworld	2094.4	1183.8	1372.6	989.4	1137.6	1056.0	308.0
Slippery-Gripper	497.6	968.2	424.6	436.6	423.0	415.0	102.2
Rovers	5522.2	4037.4	4526.0	5003.4	4992.0	4233.8	3906.2
OpenStacks	0	13527.4	12221.4	12808.4	13399.6	12936.0	1323.4
Triangle-tireworld	0	258.0	0	50.0	0	0	1976.0
Satellite	0	0	0	0	5730.4	0	881.0

Figure 7.36: Summary of the computation time used by the four on-line configurations of PELA.

by PELA make both, metric-based and probabilistic planners, generate more robust plans than a classical re-planning 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 be not 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 online integrations 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 and when the gathered experience achieves enough quality, PELA addresses probabilistic planning problems more robustly than the classical re-planning approach.

The ROGUE (Haigh and Veloso, 1999) system was a previous integrated architecture for the tasks of planning, execution and learning. This system learned propositional decision trees that were used as control rules by the planner of the PRODIGY architecture (Veloso et al., 1995). However, it was not based on using standard languages as PDDL or PPDDL for reasoning and learning, so that different planning and/or learning techniques can be plugged-in. And, also, very few have learned relational representations that could be used by any planner without modifying its sources.

As presented in the introduction the aims of PELA are very related to the aims of Model-free relational reinforcement learning(RRL) (Dzeroski et al., 2001; Kersting et al., 2004). Unlike our approach, the knowledge learned solving a given task with RRL techniques cannot be immediately transferred for similar tasks within the same domain (though currently several RL transfer learning techniques work on this direction). Moreover as our approach delegates the decision making to off-the-shelf planners it allows us to solve more complex problems requiring reasoning about time or resources.

Focusing on learning probabilistic action models for AP there are previous relevant works (Benson, 1997; Pasula et al., 2007a). These works learn more expressive action models than PELA because they capture diverse outcomes of actions. However, they are more expensive to be learned and require specific planning and learning algorithms. Instead, PELA captures uncertainty of the environment using existing standard machine learning techniques and compiles it into standard planning models that can be directly fed into different off-the-shelf planners. As a consequence, PELA can directly profit from the last advances in both fields without modifying the source of the architecture. And even more, the off-the-shelf spirit of the architecture allows PELA to use diverse planning paradigms or change the learning component to acquire other useful planning information, such as the actions duration (Lanchas et al., 2007).

Chapter 8

Learning actions durations with PELA

This chapter illustrates how PELA is applied for learning different features of the planning models such as action durations (Lanchas et al., 2007). This work is a joint work with Jesus Lanchas.

8.1 Introduction

With the aim of bringing AP closer to real-world problems, AP action models are becoming more expressive. Nowadays, PDDL includes action costs, state preferences, action durations, etc. Unfortunately, in numerous planning tasks, the specification of these features is not feasible because they are *'a priori'* unknown. The following sections describe how to instantiate PELA for automatically learning duration of actions from observations of plan executions. Though this PELA instantiation focuses on modelling action duration, the same approach is valid for modelling any fluent defined in the domain model, like action cost, state-rewards, etc.

8.2 Learning actions durations with PELA

This instantiation of PELA implements the learning component with a tool for the induction of relational **regression** trees. Particularly, this PELA instantiation induces state-dependent duration models with the regression tool and compiles them into a new action model which contains state-dependent estimations of the execution duration. Planning with the compiled action model allows PELA to obtain better solutions in terms of plan duration. Figure 8.1 shows an overview of the instantiation of PELA for learning action durations from execution.

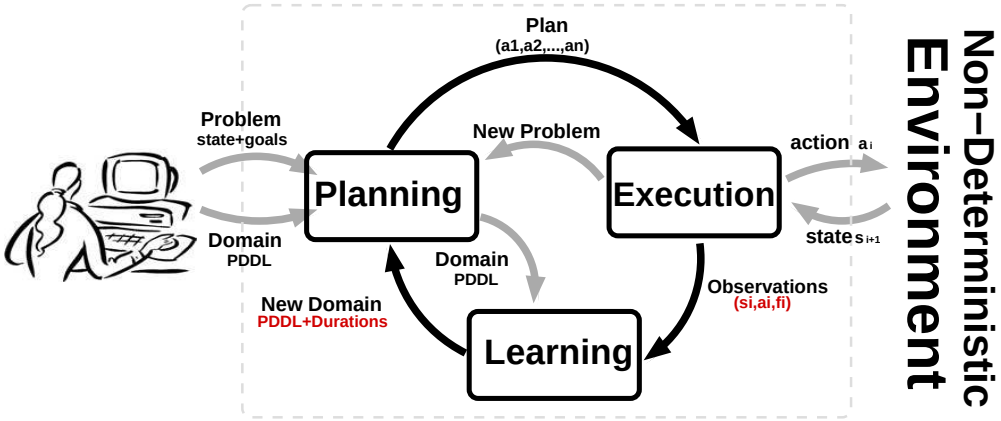

Figure 8.1: Overview of the action duration modelling with PELA.

8.2.1 Planning

The initial action model handled by the planning component is a STRIPS-like action model A described in PDDL (Fox and Long, 2003). Given that A ignores execution duration, initially, the planning component only reasons about plans length. However, when PELA upgrades the action model A to a new model A' with duration knowledge, the planning component plans minimizing a metric that indicates the execution duration of plans. Hence, the planning component requires a planner able to handle metric minimization. The output to this component is a total ordered plan $p = (a_1, a_2, ..., a_n)$.

8.2.2 Execution

Like the original PELA, the execution component executes plans $p = (a_1, a_2, ..., a_n)$ provided by the planning component and replans when action executions fail. Nevertheless, the execution component of this PELA instantiation collects a different kind of observations. After the execution of a given action a_i, this component collects an observation of the form $o_i = (s_i, a_i, f_i)$, where:

- s_i is the conjunction of literals representing the facts holding before the execution of the action a_i.
- a_i is the executed action.
- f_i is the variation of the fluent to model caused by the execution of a_i. This variation is computed subtracting the value of the fluent in state s_i from the value of the fluent in state s_{i+1}. In this work, the modeled fluent is the execution duration but there is nothing that prevents PELA from modelling other fluents like the action cost or the state-reward.

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

8.2.3 Learning

For every action $a \in A$, the learning component builds a relational regression tree t_a that models the execution duration of a. The test nodes of t_a contain the conditions under which the estimation of the duration of a is true. The leaf nodes of t_a contain situation-dependent estimations of the duration of a. Consequently, the deeper a leaf is, the more specifically a leaf predicts. As an example, Figure 8.2 shows the tree induced for action `unstack` from a version of *Blocksworld* in which the robot arm can get blocked causing longer execution times. The data of the leaf nodes describe: (1) the estimation of the fluent `duration`, in square brackets; (2) the number of examples covered by the leaf and (3) the error committed when estimating the duration of the examples covered by the leaf, in square brackets too.

```
unstack_duration(-A,-B,-C,-D)
arm_blocked(A) ?
    +--yes: [10.0] 27.0 [0.0]
    +--no:  [5.0] 48.0 [0.0]
```

Figure 8.2: Regression tree induced for the *Blocksworld* action `unstack`.

This instantiation of PELA builds the regression trees t_a using the relational learning tool TILDE (Blockeel and Raedt, 1998) with the following inputs:

- *The language bias*, specifying the restrictions about the instantiation of the domain predicates. This knowledge is automatically extracted from the STRIPS-like action model A that initially feeds the planning component. Figure 8.3 shows the language bias used for modelling the duration of action `unstack` from the *Blocksworld*.

- *The knowledge base*, containing the examples of the target concept and the associated background knowledge. In this case, the target concept is the variation of the fluent (f_i in the collected observations), and the associated background knowledge consists of the literals holding in the state (s_i in the collected observations). All literals in the knowledge base are tagged with an identifier o_i which indicates the observation they belong to. Figure 8.4 shows two observations collected from the execution of action `unstack` in a 3-blocks *Blocksworld* problem. Each observation contains the duration of the execution (10 and 5 units of time respectively) and the literals representing the state in which the action was executed.

```
% The target concept
 type(unstack_duration(example,block,block,number)).
% The domain predicates
 type(holding(example, block)).
 type(arm_empty(example)).
 type(holding_tower(example)).
 type(on_table(example, block)).
 type(on(example, block, block)).
 type(clear(example, block)).
 type(is_heavy(example, block)).
 type(arm_blocked(example)).
```

Figure 8.3: Example of language bias for action `unstack` from the *Blocksworld*.

```
% Example o1
 unstack_duration(o1,b1,b2,10).
% Background knowledge
 on_table(o1,b0). on(o1,b2,b0). on(o1,b1,b2).
 clear(o1,b1). arm_empty(o1). arm_blocked(o1).

% Example o2
 unstack_duration(o2,b2,b0,5).
% Background knowledge
 on-table(o2,b0). on(o2,b2,b0). clear(o2,b2).
 on-table(o2,b1). clear(o2,b1). arm_empty(o2).
```

Figure 8.4: Knowledge base corresponding to two executions of action `unstack`.

8.2.4 Exploitation of the learned knowledge

The induced regression trees are incorporated to the action model of the planning component. Specifically, a regression tree t_a induced for action $a \in A$ is compiled into a new action a' with conditional effects 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_i of the tree t_a is compiled into a conditional effect ce_i=(when B_i E_i) where:
   (a) B_i=(and $b_{i1}, \ldots, b_{in}$), where b_{ik} are the relational tests of the internal nodes of branch b_i (in the tree of Figure 8.2 there is only one test, referring to `arm-blocked(A)`);
   (b) E_i=(and {effects(a) $\cup$ (increase (duration) f_i)});
   (c) effects(a) are the STRIPS effects of action a; and

(d) `(increase (duration)` f_i`)` is a new literal which increases the `duration` metric in f_i units. Where f_i is the prediction value of the branch b_i

As an example, Figure 8.5 shows the PDDL conditional action resulting from the compilation of the regression tree of Figure 8.2. Since the outcome of this phase is standard PDDL code, it is suitable for any off-the-shelf planner able to handle metric minimization.

```
(:action  unstack
  :parameters (?b1 ?b2 - block)
  :precondition (and (not (= ?b1 ?b2)) (arm-empty) (clear ?b1)
                     (on ?b1 ?b2))
  :effect
   (and (when (arm-blocked)
              (and (holding ?b1) (clear ?b2) (not (arm-empty))
                   (not (clear ?b1)) (not (on ?b1 ?b2))
                   (increase (duration) 10)))
        (when (not (arm-blocked))
              (and (holding ?b1) (clear ?b2) (not (arm-empty))
                   (not (clear ?b1)) (not (on ?b1 ?b2))
                   (increase (duration) 5))))
```

Figure 8.5: The resulting PDDL action with conditional effects.

8.3 Evaluation

To evaluate the performance of this PELA instantiation, we defined a set of experiments in which PELA learns diverse duration models from observations of plans executions. In all these experiments, PELA starts with a STRIPS-like action model with no knowledge about the actions durations (Figure 8.6) and executes plans in the *MDPsim* simulator that maintains a PPDDL domain description with the true duration model of the actions.

```
(:action unstack
  :parameters (?b1 ?b2 - block)
  :precondition (and (not (= ?b1 ?b2))(arm-empty)(clear ?b1)
                     (on ?b1 ?b2))
  :effect (and (holding ?b1)(clear ?b2)(not (arm-empty))
               (not (clear ?b1))(not (on ?b1 ?b2))))
```

Figure 8.6: Example of STRIPS-like action initially considered by PELA.

The test domain is a version of the *Blocksworld* domain in which the robot arm can get blocked and there are blocks of different weights. In this domain, we tested the performance of PELA in three different configurations of the execution simulator. Each configuration covers a different kind of durations model:

1. *Deterministic durations*. This is the simplest configuration of the simulator, in which action duration is a fixed constant. In this case, actions deterministically increase the `duration` fluent. As an example, the Figure 8.7 shows action `unstack` as it is defined in the simulator. According to this configuration, the execution of action `unstack` always takes three units of time.

```
(:action unstack
  :parameters (?b1 ?b2 - block)
  :precondition (and (not (= ?b1 ?b2))(arm-empty)(clear ?b1)
                     (on ?b1 ?b2))
  :effect (and (holding ?b1)(clear ?b2)(not (arm-empty))
               (not (clear ?b1))(not (on ?b1 ?b2))
               (increase (duration) 3)))
```

Figure 8.7: Deterministic configuration of the simulator for action `unstack`.

2. *Situation-dependent duration*. In this configuration, action duration depends on the state of the environment. Now, the action model of the simulator incorporates conditional effects to increase the `(duration)` fluent depending on the state of the robot-arm `(arm-blocked)` or the type of the handled blocks `(is-heavy ?block)`. Figure 8.8 shows the definition of action `unstack` in the simulator. In this definition, when the robot arm is blocked or it is handling heavy blocks, it takes more time to complete the execution of the actions than usual. Additionally, a probabilistic effect has been introduced to randomly modify the state of the robot arm after the action execution.

3. *Stochastic duration*. In this configuration, actions present situation-dependent and probabilistic duration. This configuration, represents domains where action duration depends also on circumstances of the environment that are not captured within the domain predicates. Figure 8.9 shows the definition of action `unstack` in the simulator. According to this schema, one out of three times the execution takes less time without any observable reason.

8.3.1 Correctness of the duration models

Next, there is an analysis of the duration models obtained by PELA for the three configurations of the simulator. These models are induced from the observations collected solving fifty random five-blocks problems with the LPG planner (Gerevini et al., 2003):

```
(:action unstack
  :parameters (?b1 ?b2 - block)
  :precondition (and (not (= ?b1 ?b2))(arm-empty)(clear ?b1)
                     (on ?b1 ?b2))
  :effect
   (and (holding ?b1)(clear ?b2)(not (arm-empty))
        (not (clear ?b1))(not (on ?b1 ?b2))
        (when (and (not (is-heavy ?b1))(not (arm-blocked)))
              (increase (duration) 3))
        (when (and (not (arm-blocked))(is-heavy ?b1))
              (increase (duration) 20))
        (when (and (not (is-heavy ?b1))(arm-blocked))
              (increase (duration) 8))
        (when (and (is-heavy ?b1)(arm-blocked))
              (increase (duration) 30))
        (probabilistic
             1/2 (arm-blocked)
             1/2 (not (arm-blocked)))))
```

Figure 8.8: Situation-dependent configuration of the simulator.

1. In the deterministic configuration, our models exactly capture the execution duration of each action. As shown in Figure 8.10 for action `unstack`, the induced trees consist of a single leaf node with the exact value (the relative error is 0.0) for the `duration` fluent.

2. In the situation-dependent configuration, our models exactly captured the conditions of the action durations so the relative error of leaf nodes is 0.0 too. Figure 8.11 shows the relational regression tree induced for action `unstack`. The first query tests if the arm is blocked. The following query checks whether the block to unstack is heavy or not.

3. In the stochastic configuration, our models also successfully captured the conditions of action durations. However, the relative error of leaf nodes is not 0.0 because in this configuration of the simulator actions do not have deterministic duration. In this case, leaf nodes do not indicate an exact duration but an average of the durations observed in the examples covered by the leaf node. Figure 8.12 shows the logical decision tree induced for action `unstack` in this configuration.

8.3.2 Performance of the duration models

We compared four planning configurations to evaluate the performance of the induced duration models:

1. *LPG without experience*. LPG (Gerevini et al., 2003) is run with the STRIPS-like action model. This configuration ignores actions duration and serves as

```
(:action unstack
  :parameters (?b1 ?b2 - block)
  :precondition (and (not (= ?b1 ?b2))(arm-empty)(clear ?b1)
                     (on ?b1 ?b2))
  :effect
   (and (holding ?b1)(clear ?b2)(not (arm-empty))
        (not (clear ?b1))(not (on ?b1 ?b2))
        (when (and (not (is-heavy ?b1))(not (arm-blocked)))
              (probabilistic
                   2/3 (increase (duration) 3)
                   1/3 (increase (duration) 2)))
        (when (and (not (arm-blocked))(is-heavy ?b1))
              (probabilistic
                   2/3 (increase (duration) 20)
                   1/3 (increase (duration) 14)))
        (when (and (not (is-heavy ?b1))(arm-blocked))
              (probabilistic
                   2/3 (increase (duration) 8)
                   1/3 (increase (duration) 5)))
        (when (and (is-heavy ?b1)(arm-blocked))
              (probabilistic
                   2/3 (increase (duration) 30)
                   1/3 (increase (duration) 20)))
        (probabilistic
             1/2 (arm-blocked)
             1/2 (not (arm-blocked)))))
```

Figure 8.9: Stochastic configuration of the simulator for action `unstack`.

a baseline for comparison.

2. *LPG with experience*. In this configuration, LPG is run with the upgraded action model (which considers action durations), the optimization option to minimize the metric `(duration)` and a limit for planning of *'3 solutions'*.

3. *Metric-FF without experience*. The planner Metric-FF (Hoffmann, 2003) is run with the STRIPS-like action model. This configuration also serves as a baseline.

4. *Metric-FF with experience*. Metric-FF is run with the upgraded action model and the optimization option to minimize the metric `(duration)`.

```
unstack_duration(-A,-B,-C,-D)
[3.0] 2280.0 [0.0]
```

Figure 8.10: Duration model for the deterministic configuration of the simulator.

```
unstack_duration(-A,-B,-C,-D)
arm-blocked(A) ?
+--yes: is-heavy(A,-C) ?
|       +--yes: [30] 152.0 [0.0]
|       +--no:  [8.0] 9.0 [0.0]
+--no:  is-heavy(A,-E) ?
        +--yes: [20.0] 183.0 [0.0]
        +--no:  [3.0] 20.0 [0.0]
```

Figure 8.11: Duration model for the situation-dependent configuration of the simulator.

```
unstack_duration(-A,-B,-C,-D)
arm-blocked(A) ?
+--yes: is-heavy(A,-C) ?
|       +--yes: [27.2174] 115.0 [0.4197]
|       +--no: [7.1645] 79.0 [0.1523]
+--no: is-heavy(A,-E) ?
        +--yes:[18.0714] 168.0 [0.2168]
        +--no: [2.5882] 102.0 [0.0489]
```

Figure 8.12: Duration model for the stochastic configuration of the simulator.

Figure 8.13 shows the performance of the four planning configurations in the different configurations of the simulator: deterministic, situation-dependent and stochastic. Each planning configuration is tested in a test set of thirty problems of increasing difficulty. The performance of a planning configuration is measured in terms of the `duration` metric. The graphs show the average value of this metric after solving fifteen times each problem from the test set.

Regarding the obtained experimental results, planning with the learned duration models (*LPG with experience* and *Metric-FF with experience* configurations) achieves plans with less execution time. In the case of LPG, results obtained by *LPG with experience* are overall better. However, this effect it is not universally true (note problems 3, 25 and 29 of the deterministic configuration) because of the stochastic behavior of this planner. In the case of Metric-FF, results obtained with the induced models are also overall better. Nevertheless, in problems 1, 16 and 33 of the stochastic configuration, the induced model adds such complexity to the domain theory that Metric-FF optimizing the `duration` metric does not find a solution in the time limit.

8.4 Discussion

This chapter presented and evaluated an instantiation of PELA for modelling the duration of AP actions. This instantiation generates accurate duration models of action executions when duration is deterministic, situation-dependent or stochastic. The generated models are represented in standard PDDL so they are suitable for off-the-self planners. Though this PELA instantiation focused on learning duration models, the same approach can be directly applied to learn models for any fluent of the domain theory. In general, this approach is useful for AP tasks in which goals are achievable by different plans, the quality of these plans matters but the quality of actions is *'a priori'* unknown.

In some domains, duration of action execution is a function over others fluents. In this case, the estimations of the tree node leaves should not be a numeric value but a mathematical formula. Additionally, our approach assumes full observability of the environment so observations of action executions are always perfect. Further work has to be done to acquire more complex fluent models and to model fluents in environments where observations may be wrong or incomplete.

Relational regression trees has not been previously applied to AP action modelling. Nevertheless, classical versions of both regression and decision trees have being used for action modelling in autonomous robots: the robot ROGUE (Haigh and Veloso, 1999) used decision tree learning to acquire rules that prioritize its activities according to the values of its sensors. (Balac et al., 2000) learned regression trees that proposed the next action for a mobile robot according to the sensed state of the environment. Since the learning techniques used in both works are propositional, the internal nodes of the learned trees consist only of tests over numerical values. Consequently, these works were not able to predict according to relational representations of the state, like the ones used in AP. Otherwise, relational regression trees have been used to generalize the *q-function* of a reactive agent (Dzeroski et al., 1998). In this case predictions depend on conditions of the state described relationally. However, the target of the learning process is not an action model but a goal-oriented policy. Hence, every time different goals have to be achieved, new relational trees have to be learnt from scratch, even if dynamics of the environment do not change.

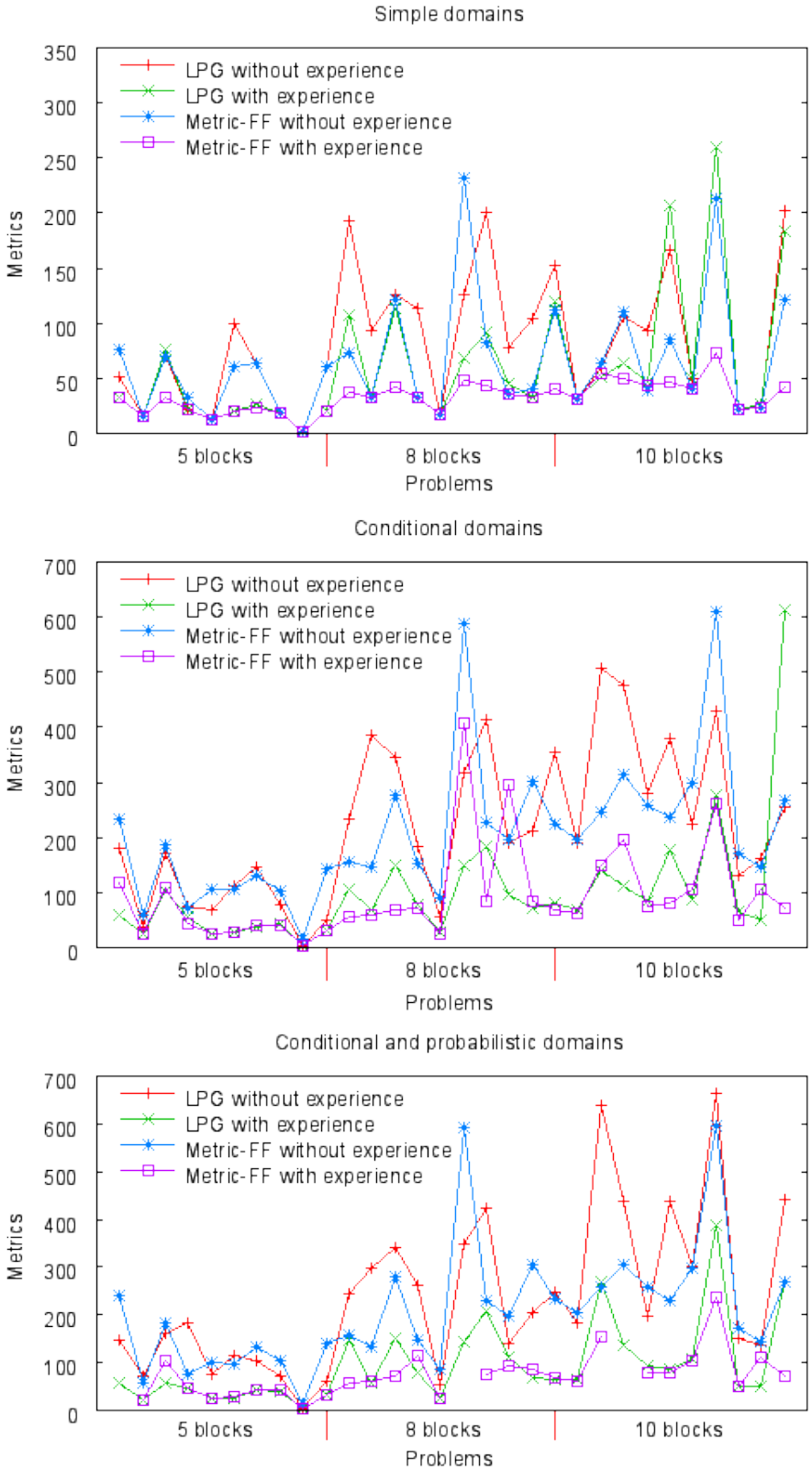

Figure 8.13: Execution duration of plans for the 3 configurations of the simulator.

Part III

Conclusions and Future Work

Chapter 9

Conclusions

This chapter presents the conclusions from the research carried out in the thesis work.

9.1 Summary

In the last years AP has experimented important advances. On the one hand, the planning-graph (Blum and Furst, 1995) provided a framework to significantly reduce the AP search space. On the other hand, domain independent heuristics (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Helmert and Geffner, 2008) became a reliable guide for the AP search algorithms. Nowadays, automated planners are able to synthesize plans of hundreds of actions in a variety of domains and the scope of AP is being extended from toy problems to real-world applications. Nevertheless, real-world applications demand action models more expressive than traditional STRIPS-like ones. They need to support execution cost, duration, probability of success, etc. Specifying action models supporting all these features is complex because frequently these features are *'a priori'* unknown.

In this thesis I argue that ML can help to overcome this bottleneck. As a matter of fact, RL has already been successfully applied to real-world tasks with unknown action models. However, the kind of solutions found by RL does not generalize. RL requires transfer learning or learning from scratch every time the task is modified, even if the environment is the same. Alternatively, cognitive architectures are able to learn general knowledge that can be directly applied to different tasks within the same environment. However, very few of these architectures have focused on solving AP problems in environments with uncertainty.

This thesis presents PELA, an architecture proposal for integrating planning, execution and learning in environments with uncertainty. PELA starts solving planning tasks with a STRIPS-like model of the environment and automatically upgrades it as more experience is available. In this way, PELA reduces the uncertainty of the environment and improves the quality of the subsequent solutions. Particularly, the learning component of PELA upgrades the initial STRIPS-like model

with: (1) probabilistic knowledge about the success of actions and (2) prediction of execution dead-ends. Moreover, given that PELA is based on off-the-shelf planning and learning components it can profit from the last advances in both fields without modifying the architecture.

Our integration of planning, execution and learning has been experimentally evaluated. Experimental results show that PELA addresses AP tasks under uncertainty more robustly than the classical *re-planning* approach. Besides, since the action model upgrade proposed by PELA does not affect to actions causality, our integration is also suitable for on-line learning. Finally, we have shown how to extend our architecture proposal for capturing other interesting features of the planning action models such as the execution duration of actions.

9.2 Contributions

The main contribution of the thesis is the definition of a general architecture for integrating processes of planning, execution and learning in domains with uncertainty. This architecture is based on off-the-shelf components and automatically captures knowledge from the execution of the plans. The architecture can learn, the robustness of instances (Jiménez et al., 2005a,b), situation dependent probabilities (Jiménez and Cussens, 2006; Jiménez et al., 2006b; Jiménez, 2007; Jiménez et al., 2008, 2011b) or actions duration (Lanchas et al., 2007) and this learning can be off-line and on-line. Additionally, the thesis work resulted in the following set of contributions:

1. An in-depth review of the state of the art in learning for classical planning (Jiménez and de la Rosa, 2008; Fernández et al., 2009; Jiménez et al., 2011a). Chapter 3 of the thesis describes the main works in learning for classical planning from a unifying approach. The diverse techniques are revised according to the target of the learning process (search control or action model). For each technique this review analyses the scope of the technique, benefits, drawbacks and the main implementations. In addition, this review motivated a study about the use of relational decision trees to learn search control for heuristic planning (de la Rosa et al., 2008, 2011).

2. An in-depth review of the state of the art in PUU from a general problem solving approach. Chapter 4 of the thesis revises the diverse PUU paradigms according to two dimensions: determinism of the action effects and observability of the environment. For each PUU paradigm, the review analyses the main advances structured by representation languages, algorithms and implementations of the paradigm.

3. A mechanism for defining *probabilistically interesting* domains from classical AP domains. As shown at IPC-2004 and IPC-2006 in which FF-REPLAN outperformed the rest of participants planners, probabilistic planning is only

better than classical replanning in *probabilistic interesting* problems. The evaluation section of Chapter 7 of the thesis describes three techniques for building *probabilistic interesting* domains from classical domains. These techniques have been used in the thesis to generate *probabilistic interesting* versions of the classical domains: *openstacks*, *rovers* and *satellite*.

4. A review of the state of the art in learning for PUU. Traditionally, learning for PUU has been studied within the MDP framework. Chapter 5 of the thesis revises the art in learning for PUU from a symbolic planning approach, in the same way that is done with learning for classical planning. That is, analysing the main works in learning for PUU according to the target of their learning process: search control or action model.

5. The analysis of domain exploration strategies in standard AP domains. Unlike traditional RL domains, random strategies do not guarantee good exploration for traditional AP domains. This is because, in AP domains, certain actions are only applicable under very specific circumstances. The evaluation section of Chapter 7 of the thesis compares the behavior of different exploration strategies for AP domains.

6. An online integration for the proccess of planning, execution and learning. PELA is able to acquire and upgrade action models in the middle of a problem solving proccess.

Chapter 10

Future Work

The architecture developed in this thesis is a suitable framework for studying different open issues in PUU:

1. *Real-time planning*: Given that AP is time consuming, it is desirable to provide PELA with reactive behavior for guaranteeing quick responses in the presence of time constraints. In this sense, we can follow two different approaches to define reactive behavior in PELA:

   (a) Planning with a *real-time search* algorithm. We can replace the search algorithm of the planner used in the planning component by a *real-time search* algorithm (Korf, 1990; Bulitko and Lee, 2006; Hernández and Meseguer, 2007). These search algorithms fast provide steps towards a goal state though the complete solution is not constructed. These steps will eventually converge to a solution when enough time is available.

   (b) Simplifying the planning task. We can simplify the planning task to quickly obtain rough solutions. For example, we can consider the relaxed planning task in which deletes of actions are ignored. In this case the solution to this simplification of the planning task, the relaxed plan, can be computed in polinomial time (Hoffmann and Nebel, 2001). Besides, the planning task can also be simplified by reducing the number of goals (Sapena and Onandía, 2008) or the number of considered objects (Edelkamp, 2002) in the planning process.

2. *Specific action execution*: We can enrich the execution component of PELA with specific execution controllers for actions that frequently appear in the planning domains. For example, a navigation module for controlling the generic action `move(origin, destiny)`.

3. *Resource and time execution monitorization*. PELA only monitors the actions causality ignoring resource and duration constraints of plans. We can improve the execution component of PELA with a CSP solver that checks

the feasibility of the pending plan in the current state of the environment regarding the current resources allocation and the current execution duration of actions (Rodríguez-Moreno et al., 2004; Bresina et al., 2005; Coles et al., 2009).

4. *Plan repairing and plan reuse*. We can provide the execution component of PELA with repairing techniques for domains in which planning from scratch is expensive. In this sense, we can use previous plans to seed the search (Gerevini and Serina, 2000), to generate lookahead states (Vidal, 2004) or to avoid node evaluation (de la Rosa et al., 2007).

5. *Learning full action models*. We can improve the learning component of PELA with deeper action modelling capabilities. For instance, algorithms for learning action preconditions or diverse action outcomes (Pasula et al., 2007a) or algorithms for learning action models in environments with partial observability (Yang et al., 2007; Amir, 2006).

6. *Model-Lite planning*. Current off-the-shelf planners assume that the action model is complete and correct. Recently, Kambhampati introduced the concept of *model-lite planning* (Kambhampati, 2007) for encouraging the development of AP techniques able to relax this assumption. Particularly, *model-lite planners* should search for a solution plan but for the most plausible solution plan that respects the current domain model. The PELA architecture is suitable framework for the development and evaluation of these ambitious techniques.

7. *Concurrent integration of components*. The information flow of PELA is sequential. A concurrent integration of the architecture components (Simmons, 1990) would reduce the impact of the computation time of each component.

Bibliography

Aha, D. W., Molineaux, M., and Ponsen, M. J. V. (2005). Learning to win: Case-based plan selection in a real-time strategy game. In *International Conference on Case-Based Reasoning, ICCBR*, pages 5–20.

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *International Journal of Robotics Research*, 17:315–337.

Albore, A., Palacios, H., and Geffner, H. (2007). Fast and informed action selection for planning with sensing. In *Conference of the Spanish Association for Artificial Intelligence (CAEPIA-07)*.

Aler, R., Borrajo, D., and Isasi, P. (2002). Using genetic programming to learn and improve control knowledge. *Artificial Intelligence*, 141(1-2):29–56.

Amir, E. (2006). Learning partially observable action schemas. In *National Conference on Artificial Intelligence (AAAI'06)*.

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

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

Balac, N., Gaines, D. M., and Fisher, D. (2000). Using regression trees to learn action models. In *IEEE Systems, Man and Cybernetics Conference, Nashville. USA*.

Barto, A. and Duff, M. (1994). Monte carlo matrix inversion and reinforcement learning. In *Advances in Neural Information Processing Systems 6*, pages 687–694.

Beetz, M. (1999). Structured reactive controllers: controlling robots that perform everyday activity. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

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

Bergmann, R. and Wilke, W. (1996). Paris: Flexible plan adaptation by abstraction and refinement. In *Workshop on Adaptation in Case-Based Reasoning. ECAI-96*.

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2006). Strong planning under partial observability. *Artificial Intelligence*, 170(4):337–384.

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming (Optimization and Neural Computation Series, 3)*. Athena Scientific.

Besag, J. (1975). Statistical analysis of non-lattice data. *The Statistician*, 24(3):179–195.

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

Blum, A. and Furst, M. (1995). Fast planning through planning graph analysis. In *International Joint Conference on Artificial Intelligence, IJCAI-95*.

Blum, A. and Langford, J. (1999). Probabilistic planning in the graphplan framework. In *European Conference on Planning*, pages 319–332.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33.

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*.

Borrajo, D. and Veloso, M. (1997). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 11(1-5):371–405.

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621.

Boutilier, C., Reiter, R., and Price, B. (2001). Symbolic dynamic programming for first-order MDPs. In *International Joint Conference on Artificial Intelligence*.

Brachman, R. J. and Levesque, H. J. (1984). The tractability of subsumption in frame-based description languages. In *National Conference on Artificial Intelligence, AAAI84*.

Brafman, R. and Hoffmann, J. (2004). Conformant planning via heuristic forward search: A new approach. In Koenig, S., Zilberstein, S., and Koehler, J., editors, *International Conference on Automated Planning and Scheduling (ICAPS-04)*.

Bresina, J. L., Jónsson, A. K., Morris, P. H., and Rajan, K. (2005). Mixed-initiative activity planning for mars rovers. In *IJCAI*, pages 1709–1710.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691.

Bryce, D. (2006). The partially-observable and non-deterministic planner. In *International Planning Competition. International Conference on Automated Planning and Scheduling.*

Buffet, O. and Aberdeen, D. (2006). The factored policy gradient planner. In *International Planning Competition. International Conference on Automated Planning and Scheduling.*

Bui, H. H. and Venkatesh, S. (2002). Policy recognition in the abstract hidden markov model. *Journal of Artificial Intelligence Research*, 17:2002.

Bulitko, V. and Lee, G. (2006). Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*, 25:119–157.

Bylander, T. (1991). Complexity results for planning. In *International Joint Conference on Artificial Intelligence. IJCAI-91*, Sydney, Australia.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

Carrick, C., Yang, Q., Abi-Zeid, I., and Lamontagne, L. (1999). Activating CBR systems through autonomous information gathering. *Lecture Notes in Computer Science*, 1650.

Castillo, L., Armengol, E., Onaindía, E., Sebastiá, L., Boticario, J., Rodríguez, A., Fernández, S., Arias, J., and Borrajo, D. (2008). Samap. a user-oriented adaptive system for planning tourist visits. *International Journal of Expert Systems With Applications*, 34.

Castillo, L., Fdez.-Olivares, J., García-Pérez, O., and Palao, F. (2006). Bringing users and planning technology together. experiences in SIADEX. In *International Conference on Automated Planning and Scheduling (ICAPS 2006)*.

Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (1999). NUSMV: a new Symbolic Model Verifier. In *Conference on Computer-Aided Verification (CAV'99)*.

Cimatti, A., Giunchiglia, F., Giunchiglia, E., and Traverso, P. (1997). Planning via model checking: A decision procedure for AR. In *European Conference on Planning*.

Cimatti, A. and Roveri, M. (2000). Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13.

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *International Conference on Machine Learning*.

Coles, A. and Smith, A. (2007). Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156.

Coles, A. I., Fox, M., Halsey, K., Long, D., and Smith, A. J. (2009). Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1):1–44. Avaliable online August 2008.

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.

Culberson, J. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14:318–334.

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

Dawson, C. and Silklossly, L. (1977). The role of preprocessing in problem solving system. In *International Joint Conference on Artificial Intelligence, IJCAI-77*, pages 465–471.

de la Rosa, T., García-Olaya, A., and Borrajo, D. (2007). Using cases utility for heuristic planning improvement. In *International Conference on Case-Based Reasoning*.

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

de la Rosa, T., Jiménez, S., Fuentetaja, R., and Borrajo, D. (2011). Scaling-up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research (JAIR)*.

De Raedt, L. and Dehaspe, L. (1997). Clausal discovery. *Machine Learning*, 26(2-3):99–146.

Donini, F. M., Lenzerini, M., Nardi, D., and Nutt, W. (1995). The complexity of concept languages. Technical Report RR-95-07, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH. Erwin-Schrödinger Strasse. Kaiserslautern. Germany.

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

Driessens, K. and Ramon, J. (2003). Relational instance based regression for relational reinforcement learning. International Conference on Machine Learning.

Dzeroski, S., Raedt, L. D., and Blockeel, H. (1998). Relational reinforcement learning. In *International Workshop on Inductive Logic Programming*.

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

Edelkamp, S. (2002). Symbolic pattern databases in heuristic search planning. In *International Conference on AI Planning and Scheduling (AIPS).*

Ernst, G. W. and Newell, A. (1969). *GPS: A Case Study in Generality and Problem Solving*. ACM Monograph Series. Academic Press, New York, NY.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–301.

Felner, A., Korf, R. E., and Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318.

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

Fern, A., Yoon, S. W., and Givan, R. (2006). Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence Research*, 25:75–118.

Fernández, S., Jiménez, S., and de la Rosa, T. (2009). *Handbook of Research on Machine Learning Applications and Trends*, chapter Improving Automated Planning with Machine Learning. Information Science Reference.

Fikes, R., Hart, P., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288.

Fikes, R. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

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

Fox, M., Ghallab, M., Infantes, G., and Long, D. (2006b). Robot introspection using learned hidden markov models. *Artificial Intelligence*, 170(2):59–113.

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

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning probabilistic relational models. In *Sixteenth International Joint Conference on Artificial Intelligence, IJCAI*, pages 1300–1309.

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

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

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, USA.

Gartner, T., Driessens, K., and Ramon, J. (2003a). Graph kernels and gaussian processes for relational reinforcement learning. In *International Conference on Inductive Logic Programming, ILP 2003*.

Gartner, T., Flach, P., and Wrobel, S. (2003b). On graph kernels: Hardness results and efficient alternatives. In *Computational Learning Theory*.

Geffner, H. (1999). Functional strips: a more general language for planning and problem solving. In *Presented at the Logic-based AI Workshop*.

Geffner, H. (2001). Functional strips: a more flexible language for planning and problem solving. *Logic-based Artificial Intelligence*, pages 188–209.

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. and Serina, I. (2000). Fast plan adaptation through planning graphs: Local and systematic search techniques. In *International Conference on Artificial Intelligence Planning Systems*.

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

Gretton, C. and Thiébaux, S. (2004). Exploiting first-order regression in inductive policy selection. In *Conference on Uncertainty in Artificial Intelligence*.

Groote, F. and Tveretina, O. (2003). Binary decision diagrams for first-order predicate logic. *Journal of Logic and Algebraic Programming*, 57(1-2):1–22.

Guestrin, C., Koller, D., Parr, R., and Venktaraman, S. (2002). Efficient solution methods for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468.

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.*

Hammond, K. J. (1990). Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228.

Hansen, E. A. and Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297.

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

Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *National Conference on Artificial Intelligence (AAAI 2007)*.

Haussler, D. (1999). Convolution kernels on discrete structures. Technical Report UCS-CRL-99-10, UC Santa Cruz.

Helmert, M. and Geffner, H. (2008). Unifying the causal graph and additive heuristics. In *International Conference on Automated Planning and Scheduling (ICAPS-2008)*.

Hernández, C. and Meseguer, P. (2007). Improving LRTA*(k). In *International Joint Conference on Artificial Intelligence, IJCAI-07*, pages 2312–2317.

Hertzberg, J., Jaeger, H., and Zimmer, U. R. (1998). A framework for plan execution in behaviour-based robots. In *IEEE International Symposium on Intelligent Control*.

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

Hoffmann, J. and Brafman, R. (2005). Contingent planning via heuristic forward search with implicit belief states. In *International Conference on Automated Planning and Scheduling (ICAPS-05)*.

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

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *National Conference on Artificial Intelligence (AAAI'2008)*.

Huang, J. (2006). Complan: A conformant probabilistic planner. In *International Planning Competition. International Conference on Automated Planning and Scheduling*.

Hyafil, N. and Bacchus, F. (2003). Conformant probabilistic planning via CSPs. In *International Conference on Automated Planning and Scheduling*.

Ilghami, O., Nau, D. S., and Muñoz-Avila, H. (2006). Learning to do HTN planning. In *International Conference on Automated Planning and Scheduling, ICAPS 2006*.

Ilghami, O., Nau, D. S., Muñoz-avila, H., and Aha, D. W. (2005). Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence*, 21:413.

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

Jiménez, S. (2007). Learning actions success patterns from execution. In *Doctoral Consoritum ICAPS'07*.

Jiménez, S., Coles, A., and Smith, A. (2006a). Planning in probabilistic domains using a deterministic numeric planner. In *PLANSIG-06 Nottingham, UK*.

Jiménez, S. and Cussens, J. (2006). Combining ILP and parameter estimation to plan robustly in probabilistic domains. In *International Conference on Inductive Logic Programming. Santiago de Compostela. Spain*.

Jiménez, S., De la Rosa, T., Fernández, S., Fernández, F., and Borrajo, D. (2011a). A review of machine learning for automated planning. *The Knowledge Engineering Review*.

Jiménez, S., Fernández, F., and Borrajo, D. (2005a). Capturing knowledge about the instances behavior in probabilistic domains. In Tuson, A., editor, *PLANSIG-05 London, UK*, pages 44–51.

Jiménez, S., Fernández, F., and Borrajo, D. (2005b). Machine learning of plan robustness knowledge about instances. In *16th European Conference on Machine Learning*.

Jiménez, S., Fernández, F., and Borrajo, D. (2006b). Inducing non-deterministic actions behaviour to plan robustly in probabilistic domains. In *Workshop on Planning under Uncertainty and Execution Control for Autonomous Systems. ICAPS'06*.

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)*.

Jiménez, S., Fernández, F., and Borrajo, D. (2011b). Integrating planning, execution and learning to improve plan execution. *Computational Intelligence*.

Jiménez, S. and de la Rosa, T. (2008). *Encyclopedia of Artificial Intelligence*, chapter Learning based planning. Information Science Reference.

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang (1990). Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science*.

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

Kalyanam, R. and Givan, R. (2008). Ldfs with deterministic plan based subgoals. In *International Planning Competition. International Conference on Automated Planning and Scheduling*.

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. AAAI Press/MIT Press.

Kambhampati, S. and Hendler, J. A. (1992). A validation structure-based theory of plan modification and reuse. *Artificial Intelligence Journal*, 55:193–258.

Kaminka, G. A., Pynadath, D. V., and Tambe, M. (2002). Monitoring teams by overhearing: A multi-agent plan recognition approach. *Journal of Artificial Intelligence Research*, 17:83–135.

Karalic, A. (1992). Employing linear regression in regression tree leaves. In *European Conference on Artificial Intelligence, ECAI-92*, pages 440–441.

Kautz, H. A. and Selman, B. (1992a). Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363.

Kautz, H. A. and Selman, B. (1992b). Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363.

Keller, R. (1987). *The Role of Explicit Contextual Knowledge in Learning Concepts to Improve Performance*. PhD thesis, Rutgers University.

Kelly, J. P., Botea, A., and Koenig, S. (2008). Offline planning with hierarchical task networks in video games. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conferenc*.

Kersting, K., Otterlo, M. V., and Raedt, L. D. (2004). Bellman goes relational. In *International Conference on Machine Learning, ICML-04*.

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.

Keyder, E. and Geffner, H. (2008). The hmdp planner for planning with probabilities. In *International Planning Competition. International Conference on Automated Planning and Scheduling*.

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

Knoblock, C. (1990). Learning abstraction hierarchies for problem solving. In *International Workshop on Machine Learning*.

Koenig, S., Furcy, D., and Bauer, C. (2002). Heuristic search-based replanning. In *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 310–317.

Kok, S. and Domingos, P. (2005). Learning the structure of markov logic networks. In *International Conference on Machine Learning (ICML-05)*.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence, 1985*, 26:35–77.

Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211.

Kramer, S. (1996). Structural regression trees. *National Conference on Artificial Intelligence (AAAI-96)*.

Kushmerick, N., Hanks, S., and Weld, D. S. (1995). An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286.

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

Langley, P. and Choi, D. (2006). A unified cognitive architecture for physical agents. In *Proceedings, The Twenty-First AAAI Conference on Artificial Intelligence, July 16-20, 2006, Boston, Massachusetts, USA*.

Lavalle, S. M. (2000). Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308.

Leckie, C. and Zukerman, I. (1991). Learning search control rules for planning: An inductive approach. In *International Workshop on Machine Learning*, pages 422–426, Evanston, IL. Morgan Kaufmann.

Lemai, S. and Ingrand, F. (2004). Interleaving temporal planning and execution in robotics domains. In *Nineteenth National Conference on Artificial Intelligence AAAI*, pages 617–622.

Lenser, S., Bruce, J., and Veloso, M. (2002). A modular hierarchical behavior-based architecture. In Birk, A., Coradeschi, S., and Tadokoro, S., editors, *RoboCup-2001: The Fifth RoboCup Competitions and Conferences*. Springer Verlag, Berlin.

Levine, G. and DeJong, G. (2006). Explanation-based acquisition of planning operators. In *International Conference on Automated Planning and Scheduling (ICAPS '06)*.

Little, I. and Thiébaux, S. (2006). Concurrent probabilistic planner in the Graph-Plan framework. In *International Conference on Automated Planning and Scheduling (ICAPS '06), The English Lake District, Cumbria, UK*.

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

Littman, M. L. (1997). Probabilistic propositional planning: Representations and complexity. In *National Conference on Artificial Intelligence (AAAI-97)*.

Lotem, A. and Nau, D. S. (2000). New advances in GraphHTN: Identifying independent subproblems in large HTN domains. In *AIPS*, pages 206–215.

Lovejoy, W. S. (1991). Computationally feasible bounds for partially observed markov decision processes. *Operational Research*, 39(1):162–175.

Majercik, S. M. and Littman, M. L. (1998). MAXPLAN: A new approach to probabilistic planning. In *Artificial Intelligence Planning Systems*, pages 86–93.

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

Mcallester, D. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *National Conference on Artificial Intelligence, AAAI91*, pages 634–639.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Webber, B. L. and Nilsson, N. J., editors, *Readings in Artificial Intelligence*, pages 431–450. Kaufmann, Los Altos, CA.

McDermott, D. V. and Doyle, J. (1980). Non-monotonic logic. *Artificial Intelligence*, 13:41–72.

McGann, C., Py, F., Rajan, K., Ryan, J., and Henthorn, R. (2008). Adaptive control for autonomous underwater vehicles. In *National Conference on Artificial Intelligence (AAAI'2008)*.

Mihalkova, L. and Mooney, R. J. (2007). Bottom-up learning of markov logic network structure. In *ICML '07: Proceedings of the 24th international conference on Machine learning*.

Minton, S. (1988). *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston, MA.

Mitchell, T., Utgoff, T., and Banerji, R. (1982). *Machine Learning: An Artificial Intelligence Approach*, chapter Learning problem solving heuristics by experimentation. Morgan Kaufmann.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.

Muggleton, S. (1995a). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286.

Muggleton, S. (1995b). Stochastic logic programs. In *International Workshop on Inductive Logic Programming*.

Muggleton, S. and Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan.

Muggleton, S., Luc, and Raedt, D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679.

Muñoz-Avila, Aha, H., Breslow, D., and Nau, L. (1999). Hicap: An interactive case based planning architecture and its application to noncombatant evacuation operations. In *Conference on Innovative Applications of Artificial Intelligence. IAAI-99.*

Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, W., Wu, D., and F.Yaman. (2003). Shop: An HTN planning system. *Journal of Artificial Intelligence Research*, 20.

Nau, D., Muñoz-avila, H., Cao, Y., Lotem, A., and Mitchell, S. (2001). Total-order planning with partially ordered subtasks. In *International Joint Conference on Artificial Intelligence*, pages 425–430.

Nayak, P., Kurien, J., Dorais, G., Millar, W., Rajan, K., and Kanefsky, R. (1999). Validating the ds-1 remote agent experiment. In *Artificial Intelligence, Robotics and Automation in Space*.

Newton, M. A. H., Levine, J., Fox, M., and Long, D. (2007). Learning macro-actions for arbitrary planners and domains. In *International Conference on Automated Planning and Scheduling*.

Ngo, L. and Haddawy, P. (1997). Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171:147–177.

Nilsson, N. J. (1984). Shakey the robot. Technical Report 323, AI Center, SRI International, Menlo Park, CA.

Oates, T. and Cohen, P. R. (1996). Searching for planning operators with context-dependent and probabilistic effects. In *National Conference on Artificial Intelligence*.

Onder, N. and Pollack, M. E. (1999). Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *National Conference on Artificial Intelligence (AAAI'99)*.

Onder, N., Whelan, G. C., and Li, L. (2004). Probapop: Probabilistic partial-order planning. In *International Planning Competition. International Conference on Automated Planning and Scheduling.*

Palacios, H. and Geffner, H. (2006). Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes). In *National Conference on Artificial Intelligence (AAAI-06)*.

Palacios, H. and Geffner, H. (2007). From conformant into classical planning: Efficient translations that may be complete too. In *International Conference on Automated Planning and Scheduling (ICAPS 2007)*.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007a). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007b). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29.

Pednault, E. P. D. (1994). ADL and the state-transition model of action. *Journal of Logic and Computation*, 4(5):467–512.

Penberthy, J. and Weld, D. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Conference on Principles of Knowledge Representation and Reasoning. KR-92*.

Peot, M. and Smith, D. (1992). Conditional nonlinear planning. In *International Conference on AI Planning Systems, AIPS-92*.

Poole, D. (1993). Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64:81–129.

Pryor, L. and Collins, G. (1996). Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339.

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

Quinlan, J. and Cameron-Jones, R. (1995). Introduction of logic programs: FOIL and related systems. *New Generation Computing, Special issue on Inductive Logic Programming*.

Ramon, J. and Bruynooghe, M. (2001). A polynomial time computable metric between point sets. *Acta Informatica*, 37(10):765–780.

Reddy, R. and Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. In *International Conference on Machine Learning*.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13:81–132.

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

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

Rintanen, J. (2003). Expressive equivalence of formalisms for planning with sensing. In *International Conference on Planning and Scheduling Systems. ICAPS03*.

Rivest, R. L. (1987). Learning decision lists. *Machine Learning*, 2(3):229–246.

Rodríguez-Moreno, M. D., Borrajo, D., Oddi, A., Cesta, A., and Meziat, D. (2004). Ipss: A problem solver that integrates planning and schedulin g. *Third Italian Workshop on Planning and Scheduling*.

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, Cambridge, MA, USA.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*, chapter 3, pages 59–94. Prentice Hall, NJ.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229.

Sanner, S. and Boutilier, C. (2006). Probabilistic planning via linear value-approximation of first-order MDPs. In *International Planning Competition. International Conference on Automated Planning and Scheduling.*

Sapena, O. and Onandía, E. (2008). Planning in highly dynamic environments: an anytime approach for planning under time constraints. *Applied Intelligence*, 29:90–109.

Sato, T. and Kameya, Y. (2001). Parameter learning of logic programs for symbolicstatistical modeling. *Journal of Artificial Intelligence Research*, pages 391–454.

Sebag, M. (1997). Distance induction in first order logic. In Džeroski, S. and Lavrač, N., editors, *International Workshop on Inductive Logic Programming*, volume 1297, pages 264–272. Springer-Verlag.

Shafer, G. (1976). *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, New Jersey.

Shen, W. and Simon (1989). Rule creation and rule learning through environmental exploration. In *International Joint Conference on Artificial Intelligence, IJCAI-89*, pages 675–680.

Shortliffe, E. H. (1976). *Computer Based Medical Consultations: MYCIN*. Elsevier, New York, USA.

Simmons, R. (1990). Concurrent planning and execution for a walking robot. Technical Report CMU-RI-TR-90-16, Robotics Institute, Pittsburgh, PA.

Simmons, R. (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1).

Smith, D. E. and Weld, D. S. (1998). Conformant graphplan. In *National conference on Artificial intelligence, AAAI-98*.

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In *IEEE International Conference on Robotics and Automation*.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. Elsevier Science Inc., New York, NY, USA.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

Teichteil-Konigsbuch, F., Infantes, G., and Kuter, U. (2008). Rff: A robust, ff-based mdp planning algorithm for generating policies with low probability of failure. In *International Planning Competition. International Conference on Automated Planning and Scheduling*.

Tran, D.-V., Nguyen, H.-K., Pontelli, E., and Son, T. C. (2008). Cpa(c)/(h): Two approximation-based conformant planners. In *International Planning Competition. Fourteenth International Conference on Automated Planning and Scheduling.*

van Beek, P. and Chen, X. (1999). CPlan: A constraint programming approach to planning. In *National Conference on Artificial Intelligence, AAAI99*, pages 585–590.

van Lent, M. and Laird, J. (2001). Learning procedural knowledge through observation. In *International conference on Knowledge capture.*

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.

Veloso, M. M. and Carbonell, J. G. (1993). Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278.

Vidal, V. (2004). A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling (ICAPS 2004)*.

Wang, C., Joshi, S., and Khardon, R. (2007). First order decision diagrams for relational mdps. In *International Joint Conference on Artificial Intelligence, IJCAI-07*.

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

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *National Conference on Artificial Intelligence, AAAI-98*.

Wellman, M. P., Breese, J. S., and Goldman, R. P. (1992). From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(1):35–53.

Wiering, M. (1999). *Explorations in efficient reinforcement learning*. PhD 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*.

Wu, J.-H., Kalyanam, R., and Givan, R. (2008). Stochastic enforced hill-climbing. In *International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia*.

Xu, Y., Fern, A., and Yoon, S. W. (2007). Discriminative learning of beam-search heuristics for planning. In *International Joint Conference on Artificial Intelligence*.

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., Benton, J., and Kambhampati, S. (2008). An online learning method for improving over-subscription planning. In *International Conference on Automated Planning and Scheduling (ICAPS-2008)*.

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

Yoon, S., Fern, A., and Givan, R. (2002). Inductive policy selection for first-order MDPs. In *Conference on Uncertainty in Artificial Intelligence, UAI02*.

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

Yoon, S., Fern, A., and Givan, R. (2007b). Using learned policies in heuristic-search planning. In *International Joint Conference on Artificial Intelligence*.

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*.

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.

Zelle, J. and Mooney, R. (1993). Combining FOIL and EBG to speed-up logic programs. In *International Joint Conference on Artificial Intelligence. IJCAI-93*.

Zimmerman, H. (1990). *Fuzzy sets, decision making, and expert systems*. Kluwer Academic Publishers, Boston, USA.

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