The Knowledge Engineering Review, Vol. 00:0, 1–24. © 2009, Cambridge University Press
DOI: 10.1017/S000000000000000 Printed in the United Kingdom

A Review of Machine Learning for Automated Planning

SERGIO JIMÉNEZ

TOMÁS DE LA ROSA

SUSANA FERNÁNDEZ

FERNANDO FERNÁNDEZ

DANIEL BORRAJO

Departamento de Informática, Universidad Carlos III de Madrid.
Avda. de la Universidad, 30. Leganés (Madrid). Spain
E-mail: sjimenez@inf.uc3m.es

Abstract

Recent discoveries in automated planning are broadening the scope of planners, from toy problems to real applications. However, applying automated planners to real-world problems is far from simple. On the one hand, the definition of accurate action models for planning is still a bottleneck. On the other hand, off-the-shelf planners fail to scale up and to provide good solutions in many domains. In these problematic domains, planners can exploit domain-specific control knowledge to improve their performance in terms of both speed and quality of the solutions. However, manual definition of control knowledge is quite difficult. This paper reviews recent techniques in machine learning for the automatic definition of planning knowledge. It has been organized according to the target of the learning process: automatic definition of planning action models and automatic definition of planning control knowledge. In addition, the paper reviews the advances in the related field of reinforcement learning.

1 Introduction

Automated Planning (AP) is the branch of Artificial Intelligence that studies the computational synthesis of ordered sets of actions that perform a given task. AP appeared in the late ’50s as the result of studies into state-space search, theorem proving and control theory to solve the practical needs of robotics and automatic deduction. The STanford Institute Problem Solver STRIPS (Fikes and Nilsson, 1971), developed to be the planning component for controlling the autonomous robot Shakey (Nilsson, 1984), perfectly illustrates the interaction of these influences. From the days of Shakey to now, AP has produced accepted standards for representing planning tasks and efficient algorithms for solving them (Ghallab et al., 2004). In the last decade AP systems have been successfully applied to real world problems such as planning space missions (Nayak et al., 1999), managing fire extinctions (Castillo et al., 2006) or controlling underwater vehicles (Bellingham and Rajan, 2007).

Despite these successful examples, the application of AP systems to real world problems is still complicated, mainly because of two knowledge definition problems:

- Automated planners require an accurate description of the planning task. These descriptions include a model of the actions that can be carried out in the environment, a specification of the state of the environment and the goals to achieve. In the real world, the execution of an action may result in numerous outcomes, the knowledge of the state of the environment may be partial and the goals may not be completely defined. Generating exact definitions of the planning tasks in advance is unfeasible for most real-world problems.
- Off-the-shelf planners often fail to scale-up or yield good quality solutions. Usually, the search for a solution plan in AP is a *PSpace-complete* problem (Bylander, 1991, 1994). Current state-of-the-art planners try to cope with this complexity through reachability analysis by (1) grounding actions and (2) performing a search process guided by domain-independent heuristics. Nevertheless, when the number of objects is large, the resulting ground search trees cannot be traversed in a reasonable time. Moreover, where heuristics are poorly informed –like in the case of some domains with strong sub-goals interactions– this kind of analysis is misleading. Domain-specific search control knowledge has been shown to improve the scalability of planners in these situations (Bacchus and Kabanza, 2000; Nau et al., 2003). Defining search control knowledge is usually more difficult than defining the planning task because it requires expertise, not only in the task to solve, but also in the planning algorithm.

Since the beginning of research in AP, Machine Learning (ML) has been a useful tool to overcome these two knowledge acquisition problems. A comprehensive survey of AP systems that benefit from ML can be found at (Zimmerman and Kambhampati, 2003). At the present time, there is renewed interest in using ML to improve planning. In 2005 the first International Competition on Knowledge Engineering for Planning Systems (ICKEPS) was held and – in 2008– a track for learning-based planners was opened within the International Planning Competition (IPC). Furthermore, workshops on planning and learning have periodically taken place within the International Conference on Automated Planning and Scheduling (ICAPS). Even if this review includes classic systems for planning and learning, mainly it focuses on covering the last approaches on ML for AP.

The review is organized according to AP's two knowledge acquisition problems: (1) learning planning **action models** and (2) learning **search control**. Figure 1 illustrates these two targets of the ML process. Traditionally, most planners separate the definition of the action models from the definition of search control knowledge to allow the planner the use of different representations. There are cases, however, where this division between models and control knowledge is unclear like, for example, macro actions that can be placed in both categories. The paper also reviews the latest techniques in Reinforcement Learning (RL) (Kaelbling et al., 1996; Sutton and Barto, 1998) which are very closely related to the ML for AP task. In RL the agent interacts with its environment to automatically learn an *action-value* function of the environment. This *action-value* function combines information from both the search control and action model to solve a particular task.

For each of the reviewed techniques, the paper provides a unified study analyzing its scope, and strengths and weaknesses, with respect to four common issues in the ML process: *knowledge representation*, *extraction of the experience*, *learning algorithm* and *exploitation of learned knowledge*. The paper is structured as follows: the second section presents the two areas of the study, Automated planning and Machine Learning; the third section reviews the diverse techniques for the automatic learning of planning action models; the fourth section reviews the different approaches for automatically learning planning search control knowledge; the fifth section describes the techniques for RL and the last section outlines promising new avenues of research in learning for planning systems.

2 Background

This section provides an overview of the area under study, *Automated Planning*, and presents our criteria for reviewing the planning and learning systems.

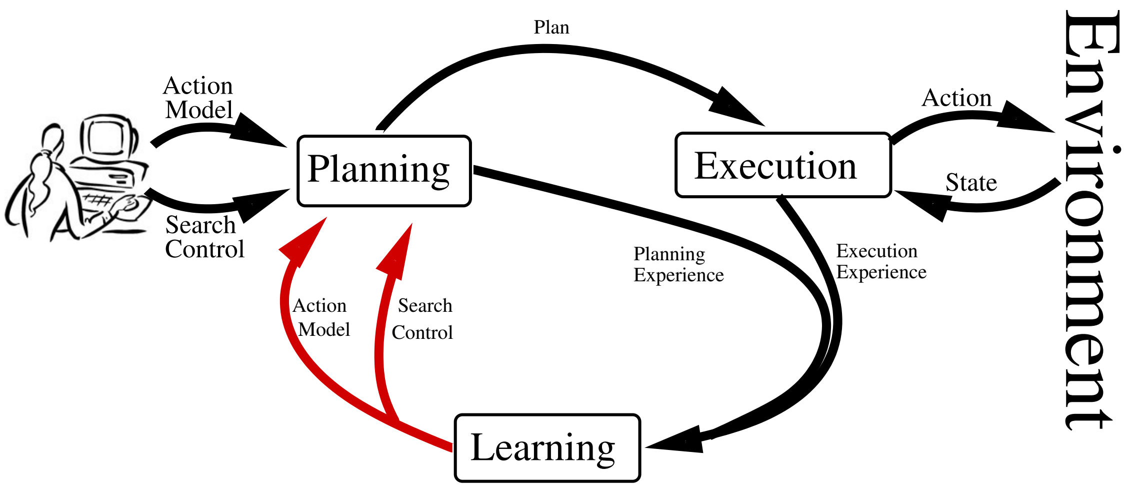

Figure 1 Use of ML to improve the knowledge of the AP processes.

2.1 Automated Planning

AP aims to produce solvers that are effective when faced with different classes of problems. AP produces solvers that exploit environment dynamics models to reason on different tasks in different environments. An AP task is defined by two elements:

- *The domain*, that consists of the set of states of the environment S together with the set of actions A that indicates the transitions between these states.
- *The problem*, that consists of the set of facts $s_0 \in S$ that indicates the initial state of the environment and the set of facts $G \subseteq S$ that indicates the goals of the AP task.

A solution to an AP task is a sequence of actions $a_1, \ldots, a_n$. This sequence of actions corresponds to a sequence of state transitions $s_0, \ldots, s_n$ such that a_i is applicable in state s_{i-1} and produces state s_i, s_n is a goal state; that is $G \subseteq s_n$. The cost of the solution is the sum of the action costs. A solution is considered optimal when it minimizes the expression $\sum_{i=1}^{n} cost(a_i)$.

Specifying accurate action models to address AP tasks in the real world is a complex exercise. Even in traditionally easy-to-code planning domains– like *Blocksworld*– it is hard to specify the actions' potential outcomes when the environment is non-deterministic. In extreme cases, like planning for the autonomous control of a Mars Rover (Bresina et al., 2005) or an underwater vehicle (McGann et al., 2008), this knowledge cannot be specified because it is as of yet unknown. In these situations, the success of the AP systems fully depends on the skills of the experts who define the action model. With the aim of assisting experts, the planning community is developing systems for the acquisition, validation and maintenance of AP models. ML seems to be a useful tool for these systems to automate the extraction and organization of knowledge from examples of plans, execution traces or user preferences.

When the AP task is accurately defined, it looks like it should be easy to tackle by searching for a path in a state-transition graph, a well-studied problem. However, in AP this state-transition graph often becomes so large that search is intractable. The complexity of the algorithms to solve this type of problems grows with the number of states, which is exponential in the number of problem variables (number of objects and predicates of the problem). Prior to the mid '90s, planners were unable to synthesize plans with more than ten actions in an acceptable amount of time.

In the late 90's, a significant scale-up in planning took place due to the appearance of the reachability planning graph (Blum and Furst, 1995). This discovery allowed for the development of powerful domain-independent heuristics (Hoffmann and Nebel, 2001a; Bonet and Geffner, 2001) that were computable in polynomial time. Recent discoveries– such as the automatic extraction of search landmarks (Porteous and Sebastia, 2004) and the automatic construction of symbolic Pattern Data Bases (Edelkamp, 2002)–

improve planners' speed and quality performance. Planners using these advances can often synthesize one-hundred action plans in seconds, yet scalability limitations are still present in AP and even well-studied domains –like the *Blocksworld*– become challenging for planners when there is a relatively large number of objects. On the one hand, current domain-independent heuristics are expensive to compute. This effect is more evident in domains where heuristics are misleading. In these domains planners spend most of their planning time computing useless node evaluations. On the other hand, given that these domain-independent techniques are based on action grounding, the planners' search trees become intractable when the number of problem objects and/or action parameters reaches a certain size. These problems make domain-independent planners' application to various real problems difficult. Logistics applications need to handle hundreds of objects as the same time as hundreds of vehicles and locations (Florez et al., 2010) which makes computing evaluation functions in every search node unfeasible. In this case, ML has a role to play in capturing useful control knowledge skipped by the domain-independent techniques.

2.2 Machine Learning

According to (Mitchell, 1997), there is a set of issues in common when using ML to improve an automatic process. The review of the planning and learning systems completed in this paper is carried out in relation to them:

1. *Knowledge representation*. First, the type of knowledge that the ML process will learn must be defined. In this paper we consider two different targets of ML for AP, **action models** to feed planners and **search control** to guide the planners' search for solutions. Second, how to represent the learned knowledge must be decided. In this case, two representation decisions must be made:
   (a) *Representation language*. The kind of notation used to encode the target concept and the experience. Because AP tasks are normally described in **predicate logic** this is the most often used representation language for encoding AP concepts. That said, others languages like **description logic** or **temporal logic** are also used, although to a lesser extent.
   (b) *Feature space*. The set of features that the ML algorithm considers to learn the target concept. In AP, these features are normally the **domain predicates** used to define the the AP task's actions, states and goals.
2. *Extraction of experience*. How learning examples are collected. In the case of AP, learning examples can be **autonomously** collected by the planning system or provided by an **external agent**, such as a human expert. Implementing a mechanism to autonomously collect learning examples is a complex process. Using a planner to collect experience is an open issue mainly because guaranteeing an AP problem's solvability with a given domain model is just as hard as the original AP task. Random explorations frequently under-sample AP tasks' state and action spaces. AP actions typically present preconditions that can only be satisfied by specific sequences of actions which have low probability of being chosen by chance.
3. *Learning algorithm*. How to capture patterns from the collected experience. Different approaches can extract these patterns. **Inductive Learning** builds patterns by generalizing from observed examples. **Analytical Learning** uses prior knowledge and deductive reasoning to build patterns that explain the information from the learning examples. **Hybrid Inductive-Analytical Learning** combines the two previous learning techniques to obtain the benefits of both: a better accuracy in generalization when prior knowledge is available and use of observed learning data to overcome the shortcomings in the prior knowledge. When designing learning algorithms for AP inductive learning is the most commonly used technique, but analytical and hybrid approaches have also used based on the domain definition of the AP task to build explanations of the collected learning examples.
4. *Exploitation of learned knowledge*. How the automatic system benefits from the learned knowledge. The decisions made on each one of the previous three issues will affect the learned knowledge. If the learned knowledge is imperfect, it must be applied via a mechanism that guarantees a robust exploitation. In the case of AP, the imperfect knowledge could be the result of several circumstances: some representation choices may not be expressive enough to capture relevant knowledge for a given

domain; the strategy for collecting the learning experience may miss significant examples of the target knowledge; or the learning algorithm may get stuck in local minima or not be able to capture patterns of the target knowledge within reasonable time and memory requirements. In these situations, a direct use of the learned knowledge may damage the planning process. Planning and learning systems need to be equipped with mechanisms that allow them to plan as robustly as possible despite flaws in the learned knowledge.

3 Learning Planning Action Models

AP algorithms reason about correct and complete action models that indicate the state-transitions of the world. Building planning action models from scratch is difficult and time-consuming, even for AP experts. An alternative approach is to use ML so people do not have to hand-code the action models. This section reviews ML techniques for the automatic definition of action models for AP. The review classifies the techniques by the stochasticity of the actions effects and by the observability of the state of the environment.

1. *Action effects*. In many planning tasks deterministic world dynamics cannot be assumed. This is the case with planning domains that include stochastic procedures such as the toggling of a coin or the rolling of a dice, or non-deterministic outcomes, such as robot navigation in the real world.
2. *State observability*. In many planning tasks handling a complete and exact description of the state of the environment is inconceivable. Parts of the current state may be confused or missing due to sensors' failures or to their inability to completely sense the world. For example, when controlling a robot in the real world.

Accordingly, we defined four categories for AP modelling: deterministic actions in fully observable environments; deterministic actions in partially observable environments; stochastic actions in fully observable environments; and stochastic actions in partially observable environments. Despite the fact that other classifications are possible– like for instance by grouping according to the the learning target (preconditions, effects, conditions of effects, probabilities of outcomes, …)– we believe this one is useful for planning purposes because each class corresponds to a different planning paradigm. Figure 2 summarizes the classification of the planning action modelling systems by giving some example implementations. This table does not intend to be an exhaustive enumeration so the systems in the table are just a sample. The rest of the section describes the systems belonging to each of the four categories in detail.

MODEL	**FEATURES**		**IMPLEMENTATIONS**
	Strengths	**Weakness**	
Deterministic effects **Full** state observability	• Learning complexity is theoretically bounded • Efficient planning algorithms • Complete coverage of the learning examples	• Poor expressiveness	LIVE (Shen and Simon, 1989), EXPO (Gil, 1992), OBSERVER (Wang, 1994)
Deterministic effects **Partial** state observability	• Complete coverage of the learning examples	• Poor expressiveness • Inefficient planning algorithms	ARMS (Yang et al., 2007), (Amir and Chang, 2008), (Mourão et al., 2008), LOCM (Cresswell et al., 2009)
Probabilistic effects **Full** state observability	• Rich expressiveness • Efficient planning algorithms	• Non-existent online learning	(Oates and Cohen, 1996), TRAIL (Benson, 1997), LOPE (García-Martínez and Borrajo, 2000), (Pasula et al., 2007), PELA (Jiménez et al., 2008)
Probabilistic effects **Partial** state observability	• Rich Expressiveness	• High planning and learning complexity	(Yoon and Kambhampati, 2007)

Figure 2 Implementations of systems for planning action modelling.

3.1 Learning Deterministic Models in Fully Observable Environments

This is the simplest scenario and corresponds to the classical planning task, where planning actions present deterministic effects and observations of the states are correct and complete. To offer an example of the modeled actions, Figure 3.1 shows the application of the `unstack(B,A)` action from the *Blocksworld* in a fully observable and deterministic environment. The *Blocksworld* is a classic domain in AP which consists of a set of blocks, a table and a robot hand: the Blocks can be on top of other blocks or on the table; a block that has nothing on it is clear; and the robot hand can hold one block or be empty. The `unstack(B,A)` action makes the robot hand unstack block `B` from block `A`.

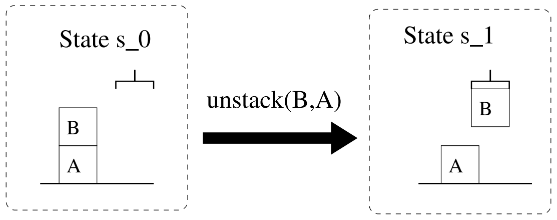

Figure 3 Example of application of the `unstack` action in a fully observable and deterministic *Blocksworld.*

3.1.1 The Learning Task

1. **Knowledge representation**. In terms of AP, the suitable languages for representing deterministic actions are the languages defined for classical planning. The Planning Domain Definition Language (PDDL) (Fox and Long, 2003; Thiébaux et al., 2005; Gerevini et al., 2009b) is the standard representation language for classical planning. It was created as the planning input language for the International Planning Competition (IPC[1]) to standardize the planning representation languages and facilitating comparative analysis of diverse planning systems. Although several of the systems reviewed in this paper use other representation languages, we have used PDDL to illustrate our examples. A PDDL action a consists of a tuple *<head(a),pre(a),eff(a)>* where,
   - *head(a)* is the head of action a containing the action name and the list of typed parameters,
   - *pre(a)* is the set of action preconditions. This set represents the facts that needs to be true for the application of action a and,
   - *eff(a)* is the set of action effects. This set includes the facts that are no longer true after the application of action a (known as *del* effects and represented by $del(a)$) and the facts made true by the application of action a (known as *add* effects and represented by $add(a)$). In PDDL, the *del* effects are preceded by the logical connective *not*.

   Figure 4 shows the PDDL representation for action `unstack` from the *Blocksworld* as an example.

```
(:action unstack
  :parameters (?top - block ?bottom - block)
  :precondition (and (emptyhand) (clear ?top) (on top? ?bottom))
  :effect (and (not (emptyhand)) (not (clear ?top)) (not (on (?top ?bottom))
               (holding ?top) (clear ?bottom)))
```

Figure 4 PDDL representation for action `unstack` from the *Blocksworld.*

2. **Learning examples**. Learning examples consist of tuples $< s_{i-1}, a_i, s_i >$ where a_i is the action applied at state s_{i-1}, i.e., the *pre-state* and s_i is the state produced by applying the action, i.e., the *post-state*.
3. **Learning algorithm**. When actions present deterministic effects and the environment states are fully observable, the dynamics of a given action are captured from the lifted literals of the *pre-state* and the *post-state* of action executions. Learning the action preconditions is accomplished by lifting the literals from a set of *pre-states*. Learning the effects of a given action is completed by lifting the set

[1]http://www.icaps-conference.org/index.php/Main/Competitions

of literals that is different in the *pre-states* and *post-states* of action executions. This set of literals is called the *delta-state* and is formally defined as: $\Delta(s_{i-1}, s_i) = (s_{i-1} \setminus s_i) \bigcup (s_i \setminus s_{i-1})$. The first term of the *delta-state* captures the *del* effects of action a_i and the second term captures its *add* effects. In both learning steps– learning the action preconditions and learning the action effects– the lifting process substitutes each grounded object with a variable. The type for each variable is given by the most specific type in the type hierarchy of the substituted object.

4. **Exploitation of the learned knowledge**. In this scenario, the learned models may present unnecessary preconditions or missing effects. These model flaws can be eliminated by observing more executions of the action but collecting a good observation sample of planning actions is still an open issue. Random applications of AP actions may leave actions unexplored. Normally AP actions present preconditions that can only be satisfied by specific sequences of actions which have a low probability of being chosen by chance. A planner may also fail to provide solutions with intermediate learned models. Considering any literal of the *pre-state* of an action as a real precondition of that action leads to very restrictive models that fail to solve problems known to be solvable. An alternative approach is to rely on an external source of experience that collects these extra observations of actions, a teacher for instance.

3.1.2 Implementations

The LIVE system (Shen and Simon, 1989) was an extension of the General Problem Solver (GPS) framework (Ernst and Newell, 1969) with a learning component. LIVE alternated problem solving with model learning to automatically define operators. The decision about when to alternate depended on *surprises*, that is situations where an action effects violated its predicted model. EXPO (Gil, 1992) generated plans with the PRODIGY system (Minton, 1988), monitored the plans execution, detected differences in the predicted and the observed states and constructed a set of specific hypotheses to fix those differences. Then the EXPO filtered the hypotheses heuristically. OBSERVER (Wang, 1994) learned operators by monitoring expert agents and applying the version spaces algorithm (Mitchell, 1997) to the observations. When the system already had an operator representation, the preconditions were updated by removing facts that were not present in the new observation's pre-state; the effects were augmented by adding facts that were in the observation's delta-state.

All of these early works were based on direct liftings of the observed states. They also benefit from experience beyond simple interaction with the environment such as *exploratory plans* or *external teachers*, but none provided a theoretical justification for this second source of knowledge. The work recently reported in (Walsh and Littman, 2008) succeeds in bounding the number of interactions the learner must complete to learn the preconditions and effects of a STRIPS action model. This work shows that learning STRIPS operators from pure interaction with the environment, can require an exponential number of samples, but that limiting the size of the precondition lists enable sample-efficient learning (polynomial in the number of actions and predicates of the domain). The work also proves that efficient learning is also possible without this limit if an agent has access to an *external teacher* that can provide solution traces on demand.

In the review we focused on learning STRIPS-like action models but others systems have tried to learn more expressive action models for deterministic planning in fully observable environments. Examples would include the learning of conditional costs for AP actions (Jess Lanchas and Borrajo, 2007) or the learning of conditional effects with quantifiers (Zhuo et al., 2008).

3.2 Learning Deterministic Models in Partially Observable Environments

In this scenario actions also present deterministic effects, but the state is not fully observable. As a consequence, the current state is described by the set of possible states called *belief state*. This scenario corresponds to the conformant planning task, when the current state is non observable, and to the contingent planning task when the current state can be partially observed. As an example of the actions modeled in this scenario, Figure 3.2 shows the application of the `unstack(B,A)` action in a deterministic and non-observable *Blocksworld*. In the example, there are two blocks, A and B. The current state is uncertain and is described by a belief state b_0 that represents the disjunction of five possible states. The result of applying action `unstack(B,A)` is a belief state b_1 which represents the disjunction of four possible states. State 1 in b_0 generates state 3 in b_1, while the action applied to the rest of states in b_0 has no effect, so they are still probable in b_1.

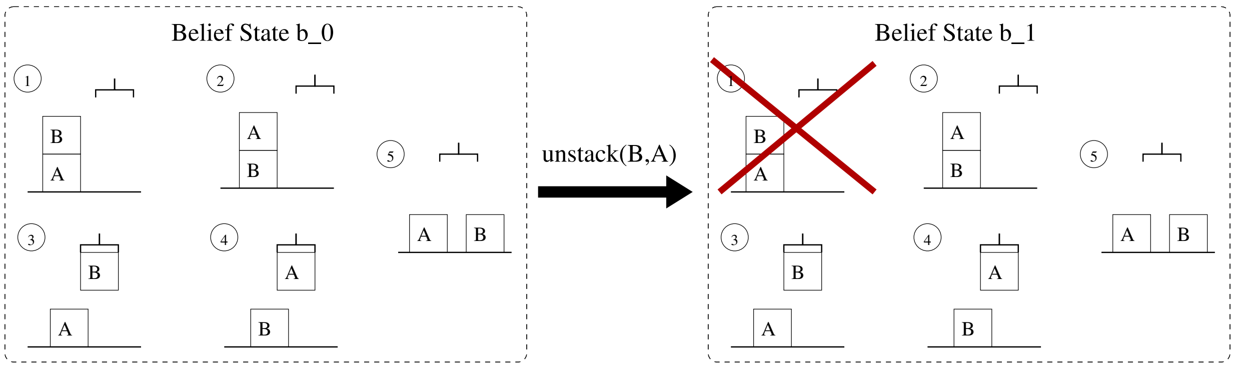

Figure 5 Example of application of the `unstack(B,A)` action in a non-observable two-blocks *Blocksworld*.

3.2.1 The learning task

1. **Knowledge representation**. Because in this scenario actions are deterministic, a classical planning representation language, such as PDDL, provides a suitable representation. Figure 4 shows the PDDL representation for action `unstack` from the *Blocksworld*.
2. **Learning examples**. Learning examples consist of tuples $< b_{i-1}, a_i, b_i >$ where b_{i-1} is a belief state corresponding to the partial observation of the *pre-state*, a_i is the action applied at the *pre-state* and b_i is the belief state corresponding to the partial observation of the *post-state*.
3. **Learning algorithm**. In this scenario, lifting the *pre-state* and *delta-state* of observations does not define the preconditions and effects of an action. Given that the states of the environment are not completely known, a given observation of a *pre-state* cannot be associated with an action's preconditions. Likewise, changes of a given set of literals cannot be directly associated to the application of a specific action. In this scenario two approaches can be used:
   - *Inference*. Inference can be used to obtain action models that satisfy the collected learning examples. This task is closely related to the Boolean satisfiability (SAT) problem which determines if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate as true. The action modeling task in this scenario can be defined as the satisfaction of a logic formula for every action in the domain. This formula is the disjunction of the possible conjunctions of *pre-states* and *post-states* observed for the action. Figure 6 shows an example of the formula for obtaining models for the action `unstack(X,Y)` that satisfy the observation in Figure 3.2.
   - *Induction*. One can search the space of possible action models to find the best fit for the observed learning examples. This task is closely related to the *Inductive Logic Programming* (ILP) task that searches the space of the *Logic Programs*[2] to find the best fit for a set of observed facts. Action modeling in this scenario can be defined as the induction of two *logic programs* for every action in the domain. Figure 7 shows the two logic programs that model action `unstack` from the *Blocksworld*.

     The first logic program captures the preconditions of the action. The head of this one-clause logic program consists of the action name and parameters, and the body is the subset of literals from the observed *pre-states* that best fits the learning examples. This learning process can be

[2]Here, the concept of *Logic program* is used in a narrower sense generally applied, i.e., as a conjunction of disjunctions of literals where each disjunction presents at most one positive literal. This type of disjunction is called a *Horn clause*.

```
(or ;;; pre-state 1 post-state 1
    ;;; pre-state 1 post-state 2
    (and (Unstack B A)
         (DelClear B) (DelOn B A) (DelOntable A)
         (AddClear A) (AddOn A B) (AddOntable B))
    ;;; pre-state 1 post-state 3
    (and (Unstack B A)
         (DelEmptyhand) (DelClear B) (DelOn B A)
         (AddHolding B) (AddClear A))

     ...

    ;;; pre-state 5 post-state 5
    (and (Unstack B A))
)
```

Figure 6 Example of a logic formulae for modeling action `unstack` from a two-blocks *Blocksworld.*

```
preconditions_unstack(TOP,BOTTOM):- object(TOP),object(BOTTOM),
                                    emptyhand(), clear(TOP), on(TOP,BOTTOM).

effects_unstack(TOP,BOTTOM):- object(TOP),object(BOTTOM),
                              del_emptyhand(),del_clear(TOP), del_on(TOP,BOTTOM),
                              add_holding(TOP),add_clear(BOTTOM).
```

Figure 7 Representation of action `unstack` from the *Blocksworld* as two logic programs.

implemented as a heuristic search guided by the number of observed *pre-states* of the action covered and not covered by the different potential programs. The second program captures the effects of the actions. The head of this logic program is comprised of the action name and parameters, and the body consists of the subsets of literals from the observed *delta-states* that best fits the learning examples. This process can also be implemented as a heuristic search guided in this case by the coverage of the observed *delta-states*.

4. **Exploitation of the learned knowledge**. An action model that is built satisfying/covering a given set of examples does not guarantee that it will present the exact set of preconditions and effects. Like in the previous scenario, the learned models may present unnecessary preconditions. In this scenario the learned effects may also capture invalid changes in the state caused by the partial-observability of the *pre* and *post* states. As a consequence, planning with this kind of models may fail to address AP problems known to be solvable.

3.2.2 Implementations

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 proceeds in two phases. In the first phase, ARMS extracts frequent action sets from plans that share a common set of parameters. ARMS also finds some frequent literal-action pairs with the help of the initial state and the goal state that provide an initial guess on the actions preconditions, and effects. In the second phase, ARMS uses the frequent action sets and literal-action pairs to define a set of weighted constraints that must hold in order to make the plans correct. Then, ARMS solves the resulting weighted MAX-SAT problem and produces action models from the solution of the SAT problem. This process iterates until all actions are modelled. For a complex planning domain that involves hundreds of literals and actions, the corresponding weighted MAX-SAT representation is likely to be too large to be solved efficiently as the number of clauses can reach up to tens of thousands. For that reason ARMS implements a hill-climbing method that models the actions approximately. Consequently, the ARMS output is a model which may be inconsistent with the examples.

(Amir and Chang, 2008) introduced an algorithm that tractably generates all the STRIPS-like models that could have lead to a set of observations. Given a formula representing the initial belief state, a sequence of executed actions $(a_1, a_2, ..., a_n)$ and the corresponding observed states $(s_1, ..., s_n)$ (where partial observations of states are given), it builds a complete explanation of observations by models of actions through a Conjunctive Normal Form (CNF) formula. By linking the possible states of fluents to the effect propositions in the action models, the complexity of the CNF encoding can be controlled to find exact solutions efficiently in some circumstances. 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 all the transition relations consistent with the actions and observations. The formula returned at the end includes all consistent models, which can then be retrieved with additional processing.

Unlike the previous approaches, the one described in (Mourão et al., 2008) deals with both missing and noisy predicates in the observations. For each action in a given domain, they use kernel perceptrons to learn predictions of the domain properties that change because of the action execution. LOCM (Cresswell et al., 2009) induces action schemas without being provided with any information about initial, goal or intermediate state descriptions for the example action sequences. LOCM receives descriptions of plans or plan fragments, uses them to create states machines for the different domain objects and extracts the action schemas from these state machines.

3.3 Learning Stochastic Models in Fully Observable Environments

In this scenario the observations of the state of the environment are perfect, but actions present stochastic effects. This scenario corresponds to the probabilistic planning task. Figure 3.3 shows an example of the `unstack(B,A)` action being applied in a fully observable and stochastic version of the *Blocksworld.* In the example, the action produces three different alternative outcomes with probabilities of 0.5, 0.3 and 0.2 respectively.

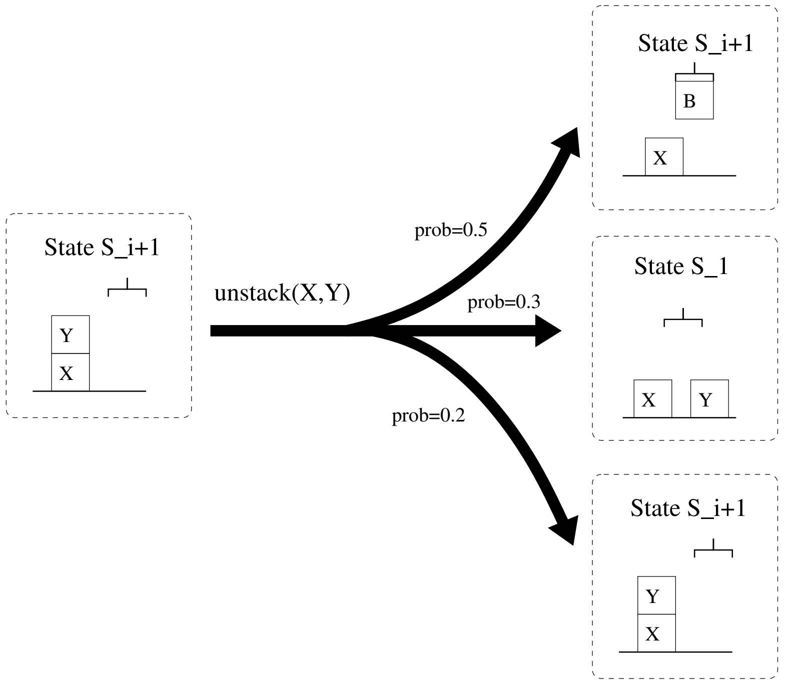

Figure 8 Example of application of the `unstack(B,A)` action in a fully observable and stochastic *Blocksworld.*

3.3.1 The Learning Task

1. **Knowledge representation**. Languages for probabilistic planning support the representation of AP models with stochastic state-transitions. The Probabilistic Planning Domain Definition Language (PPDDL) (Younes et al., 2005) is the current standard representation language defined for the IPC's probabilistic planning track. PPDDL is essentially an extension of PDDL2.1 (Fox and Long, 2003) that supports actions with *probabilistic effects*. The syntax for *probabilistic effects* in PPDDL is formally defined as follows:

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

where outcome o_i of the action occurs with probability p_i. It is required that $1 \geq p_i \geq 0$ and $\sum_{i=1}^{k} p_i = 1$. Figure 9 shows the PPDDL representation for the action `unstack` from the probabilistic version of the *Blocksworld* defined at IPC-2004. The *probabilistic effects* of this action indicate that with probability 0.5 the robot hand will successfully unstack block `?top` from block `?bottom`, with probability 0.3 the block `?top` will fall down on the table and that with probability 0.2 the blocks will not be moved.

```
(:action unstack
  :parameters (?top - block ?bottom - block)
  :precondition (and (emptyhand) (clear ?top) (on top? ?bottom))
  :effect
    (and
     (probabilistic 0.5 (and (not (emptyhand)) (not (clear ?top))
                             (not (on ?top ?bottom))
                             (holding ?top) (clear ?bottom))
                    0.3 (and (not (on ?top ?bottom))
                             (on-table ?top) (clear ?bottom)))))
```

Figure 9 PPDDL representation for the action `unstack` from the *Blocksworld*.

2. **Learning examples**. Learning examples consist of tuples $< s_{i-1}, a_i, s_i >$ where a_i is the action applied at state s_{i-1}, the *pre-state* and s_i is the *post-state*. Given that action effects are stochastic, pairs *(pre-state,action)* may have different associated *post-states*.
3. **Learning algorithm**. Because the environment is fully observable, action preconditions are captured by lifting the *pre-state* of the learning examples. Nevertheless, learning stochastic action effects is more complex because of the overlapping *post-states*; i.e., given an action transition defined by the tuple *(pre-state, action, post-state)*, several action effects could describe the transition. This aspect of overlapping examples is also present in the induction of stochastic logic models such as *Stochastic Logic Programs* (Muggleton, 1995; Cussens, 2001), *Bayesian Logic Programs* (Jaeger, 1997; Kersting and Raedt, 2001) or *Markov Logic Networks* (Richardson and Domingos, 2006). These techniques share a two-steps strategy for implementing the model learning task:
   (a) *Structural Learning*. At this step, ILP algorithms are used to induce first-order logic formulas that generalize the relations from the ground facts of the learning examples.
   (b) *Parameter Estimation*. This step estimates the probabilities associated with the induced logic formulas. The output of the *Structural Learning* step is frequently 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 computing the conditional probability $prob(P|C)$. When the induced rules are disjunctive, this probability is directly extracted from the frequency of the learning examples but when rules overlap, estimating this conditional probability requires more complex approaches:
   i. *Bayesian Estimation* (BE). This approach estimates the conditional probability following the Bayes Theorem,

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

ii. *Maximum Likelihood Estimation* (MLE). This approach assumes that the examples follow a known probability distribution and iteratively tunes the distribution's parameters to achieve a good fit.

Accordingly, modelling an action's stochastic effects can be defined as learning a *Stochastic Logic Program* where each clause in the program represents one probabilistic effect of the action. The heads of the clauses of the logic program consist of the action name and action parameters, and the bodies consist of the subsets of literals from the observed *delta-states* that best fit the learning examples for this effect. Figure 10 shows an example of stochastic logic program for the effects of action `unstack` from the *Blocksworld*.

```
0.5:effects_unstack(TOP,BOTTOM):- object(TOP),object(BOTTOM),
                                  del_emptyhand(),del_clear(TOP),del_on(TOP,BOTTOM),
                                  add_holding(TOP),add_clear(BOTTOM).

0.3:effects_unstack(TOP,BOTTOM):- object(TOP),object(BOTTOM),
                                  del_on(TOP,BOTTOM),
                                  add_ontable(TOP),add_clear(BOTTOM).

0.2:effects_unstack(TOP,BOTTOM):- object(TOP),object(BOTTOM).
```

Figure 10 Representation of the effects of action `unstack` from the *Blocksworld* as a Stochastic Logic Program.

4. **Exploitation of the learned knowledge**. The on-line modelling of actions with stochastic effects is an open issue. Actions may present outcomes that result in dead-end states which complicates the environment's exploration. It is also not enough to detect causal flaws in the learned model. It is necessary to detect flaws in the probabilities associated with the models. There are no mechanisms to refine stochastic actions when states that do not fit the model are observed.

3.3.2 Implementations

In (Oates and Cohen, 1996), the authors induced propositional rules corresponding to conditional probabilistic effects of actions. Their approach explores a given AP domain by taking random actions and observing their execution. For each execution, they register an observation in the form of $o_i =< s_i, a_i, s_{i+1} >$, where a_i is the action executed in s_i, and s_{i+1} is the resulting state of executing a_i in s_i. Then they induce the effects of a given action by performing a general-to-specific *Best-First Search* in the space of the action's possible conditional effects. For each action, this search is guided by the frequency of observations covered by the action's possible conditional effects and terminates when a user-specified number of search nodes are explored.

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 problem with the current action model. When this model is too bare to solve the problem, it asks an external expert a solution plan, it observes the plan's execution and it learns a new model from these observations. If the current action model makes TRAIL generate 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 straight off uses standard ILP tools for learning.

The LOPE system (García-Martínez and Borrajo, 2000) learns planning operators with an associated probability of success. At the beginning, the system has no knowledge. It perceives the initial state, and selects a random action to execute in the environment. Then it loops by (1) executing an action, (2) perceiving the resulting state and its utility, (3) learning a model from the perception and (4) planning for further interaction with the environment (if the execution of the plan is completed, or the system has observed a disparity between the predicted situation and the situation perceived). The LOPE's planning component does not explicitly receive a goal input given that LOPE creates its own goals from the highest utility situations.

In (Pasula et al., 2007), the authors learn probabilistic action models from externally provided observations which again consist of tuples $o_i =< s_i, a_i, s_{i+1} >$. The induced action models are probabilistic STRIPS operators extended in two ways: (1) they can refer to objects not mentioned in the action parameters, and (2) for each action, they add a *noise* outcome that groups the set of possible outcomes that are the least probable for the action. The action modelling algorithm has three learning layers:

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 fit the learning examples. An evaluation function guides the search favoring rule sets that assign high likelihood to the learning examples and penalizing model complexity (number of preconditions plus number of outcomes).
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. This work estimates the outcomes parameters implementing a MLE with the conditional gradient method. The MLE is performed for each set of outcomes considered in the previous layer.

The PELA system (Jiménez et al., 2008) automatically upgrades a STRIPS action model with situation-dependent probabilities learned from plan executions. These probabilities are captured using the TILDE tool for the automatic induction of first order decision trees (Blockeel and De Raedt, 1998).

3.4 Learning Stochastic Models in 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.

3.5 Discussion

While learning of propositional action models has been widely studied from a theoretical approach (via the notion of sample complexity (Strehl and Littman, 2005)), results for learning AP action models are largely empirical. Works on model learning for AP typically evaluate their correctness by computing the learned model's error divergence from a given reference model. When developing planning applications for solving problems in the real-world however, reference models are nonexistent. Some systems exploit other kinds of AP knowledge available. The ARMS system (Yang et al., 2007) exploits a collection of plans to evaluate the learned model's quality. However, as previously explained, there is no guarantee that a model that satisfies a given collection of examples succeeds in addressing a given problem.

A different approach consists of learning the model in an incremental and online manner, similar to Reinforcement Learning. This approach does not require previous knowledge about the planning task. In this case, we need a mechanism that guarantees convergence to a valid model. AP systems can iteratively detect and refine defective action models after an action execution. Defective models are detected though their differences with the observed states. When a defective action model is observed, it is refined by collecting more experience about the action. We can solve the action inconsistency and resume the action learning process. As previously explained, however, the autonomous collection of AP experience is still an open problem.

When the planning system fails to detect a defective action model, a standard planning algorithm may not be able to find a solution plan (though it exists). Kambhampati introduced the concept of model-lite planning (Kambhampati, 2007). This new planning paradigm advocates the development of planning techniques that do not only search for solution plans that perfectly satisfy the current action model but search for the most plausible solution plan according to the current action model. The core idea is viewing a planning problem as an MPE (most plausible explanation) problem, given an incomplete domain definition and the evidence of initial state and goals, which enables one to use publicly available approximate solvers that scale well, e.g., MAXWALKSAT.

4 Learning Planning Search Control Knowledge

Planners' performance can be improved by exploiting knowledge about the structure of the AP tasks that is not explicitly encoded in the domain model. This fact was evidenced at IPC-2002 where planners exploiting domain-specific control knowledge performed orders of magnitude faster than the state-of-the-art planners. Hand-coding search control knowledge is complex because it implies expertise in the AP domain and in the planner's search algorithm. This section reviews the different systems that benefit from ML in defining control knowledge to improve the speed and/or the quality of the planning processes. The review is organized around the target of the learning process. We review four different approaches in learning search control knowledge for AP (macro-actions, generalized policies, generalized heuristic functions and hierarchical decomposition methods). The table in Figure 11 shows systems that follow each approach. This table is not exhaustive. The systems presented are just a representative sample of existing implementations. The following subsections explain each of the four approaches in detail.

MODEL	FEATURES		IMPLEMENTATIONS
	Strengths	**Weaknesses**	
Macro-actions	• Robust to wrong learned knowledge • Valid for different planners	• Utility problem	REFLECT (Dawson and Silklossly, 1977), MORRIS (Korf, 1985), MacroFF (Botea et al., 2005a), Marvin (Coles and Smith, 2007) , (Newton et al., 2007)
Generalized Policies	• Standard relational classification algorithms	• Engineering effort to integrate with different search algorithms and domain-independent heuristics	(Minton, 1988), PRIAR (Kambhampati and Hendler, 1992), HAMLET (Borrajo and Veloso, 1997), (Khardon, 1999), (Martin and Geffner, 2000), DISTILL (Winner and Veloso, 2003), OBTUSEWEDGE (Yoon et al., 2007), CABALA (de la Rosa et al., 2007), ROLLER (de la Rosa et al., 2008)
Generalized Heuristics	• Standard relational regression algorithms • Easy integration with different search algorithms and heuristics	• Poor readability	(Yoon et al., 2006), (Xu et al., 2007)
Decomposition Methods	• Very expressive	• No completely automatic learning	CAMEL (Ilghami et al., 2002), HDL (Ilghami et al., 2006), HTNMAKER (Hogg et al., 2008)

Figure 11 Overview of AP systems that benefit from ML for the extraction of domain-specific search control.

4.1 Macro-actions

Macro-actions were among the first attempts to speed-up planning processes. They are extra actions resulting from assembling actions that are frequently used together. Macro-actions reduce the depth of the planning algorithm's search tree. Their benefit decreases with the number of new macro-actions as they enlarge the search tree's branching factor causing the *utility problem*. Addressing this trade-off is the key issue at stake when attempting to make macro-actions work. Figure 12 shows an example of the macro-action `unstack-putdown` for the *Blocksworld* domain. This macro-action is the result of assembling actions `unstack` and `putdown`.

```
;;; Primitive actions
  (:action unstack
    :parameters (?x - block ?y - block)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect (and (holding ?x)
                 (clear ?y)
                 (not (clear ?x))
                 (not (handempty))
                 (not (on ?x ?y))))

  (:action putdown
    :parameters (?x - block)
    :precondition (holding ?x)
    :effect (and (not (holding ?x))
                 (clear ?x)
                 (handempty)
                 (ontable ?x)))

;;; Macro-action
  (:action unstack-putdown
    :parameters (?x - block ?y - block)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect (and (clear ?y)
                 (not (on ?x ?y))
                 (ontable ?x)))
```

Figure 12 Macro-action `unstack-putdown` for the *Blocksworld* domain.

4.1.1 Learning Macro-Actions

1. **Knowledge representation**. Macro-actions are represented as new actions of the action model so they follow the *predicate logic* representation of AP actions. Actions a_i and a_j are assembled into a macro-action $a_{i,j}$ in the following way:
   - The parameters $par(a_{i,j}) \subseteq (par(a_i) \bigcup par(a_j))$.
   - The preconditions $pre(a_{i,j}) = (pre(a_i) \bigcup pre(a_j)) \setminus add(a_i)$.
   - The positive effects $add(a_{i,j}) = (add(a_i) \bigcup add(a_j)) \setminus del(a_j)$.
   - The negative effects $del(a_{i,j}) = (del(a_i) \bigcup del(a_j)) \setminus add(a_j)$.
2. **Learning examples**. Learning examples are solution plans p that consist of sequences of instantiated actions $p = (a_1, a_2, ..., a_n)$ that correspond to the sequences of state transitions $(s_0, s_1, ..., s_n)$ such that s_0 is the initial state of the problem, s_i results from executing action a_i at state s_{i-1} and s_n is a goal state, i.e., $G \subseteq s_n$, where G are the problem goals.
3. **Learning algorithm**. Algorithms for learning macro-actions extract action sub-sequences from the solution plans and count their occurrences to capture the most useful ones. Typically, the process to extract action sub-sequences defines two parameters:
   - l, the length of the macro-actions. The minimum value for l is $l = 2$ because a macro-action should have at least two actions. The maximum value for l is the length of the longest solution plan in the learning examples. In practice this value has to be smaller to be useful.
   - k, the number of actions that can be skipped from a solution plan. This parameter allows the learning system to extract macro-actions that ignore, at most, k irrelevant intermediate actions from a solution plan. In practice, small values for k reduce the number of macro-actions but k values that are too small may skip useful macros-actions occurrences.

   Regarding these parameters, the complexity of extracting macro-actions of length l from a plan with n actions is $\binom{l+k}{l}n$. The first factor is the cost to enumerate macro-actions of length l within a $l + k$

sized window. The second factor is the cost of sliding the window along the solution plan. A similar approach can be used to extract macro-actions from partially-ordered plans (Botea et al., 2005b).

4. **Exploitation of the learned knowledge**. Macro-actions are usable straight away by any off-the-shelf planner because they can be included in the action model as standard actions. The inclusion of macro-actions can damage the time and quality performance of the original action model. This happens when macro-actions produce a reduction in the depth of the search that does not payoff the increase in the branching factor (the *utility problem*). Current systems for learning macro-actions evaluate this problem experimentally. For example, they define a set of AP problems similar to the target ones and if a learned macro-action improves the planner's performance in these problems, then it is considered a good candidate to populate the action model. This approach for dealing with the *utility problem* was first introduced in systems that learn control rules (Minton, 1988).

4.1.2 The Implementations

Since the beginning of AP the use of macro-actions has been widely explored. The first macro-action learning system was STRIPS (Fikes et al., 1972). It used previous solution plans as macro-actions to solve subsequent problems as well as to monitor the execution of plans in the real world. Later, MORRIS (Korf, 1985) extended this approach by adding filtering heuristics to prune the generated set of macro-actions. This approach distinguished two types of macro-actions: S-macros that occur frequently during search and T-macros, that occur less often, but model some weakness in the heuristic. The REFLECT system (Dawson and Silklossly, 1977) took the alternative approach of generating macro-actions based on domain pre-processing. All sound combinations of actions were considered to be macro-actions and filtered through basic pruning rules. Due to the small size of the domains with which the planner was reasoning, the remaining number of macro-actions was small enough to use in planning.

Macro-actions systems traditionally use an off-line approach to generate and filter macro-actions before using them but a few systems have tried ML to filter macro actions during the search. In (McCluskey, 1987) the author used chunks and in (García-Durán et al., 2006) they used control rules to decide when to apply the macro-actions.

Recent works successfully integrate macros with state-of-the-art heuristic search planners. These works include the IPC-2004 competitor MACROFF (Botea et al., 2005a, 2007). Here partially ordered macro-actions are extracted by identifying statically connected abstract components and then, an off-line filtering technique is used to prune the list of macro-actions. *Marvin* (Coles and Smith, 2007), also a participant at IPC-2004, used action-sequence memorization techniques to generate macro-actions on-line that allow the planner to escape from plateaus without any exploration. Wizard (Newton et al., 2007) used a genetic algorithm to generate and filter a collection of macro-actions independent from the baseline planner. Algorithms for *n-grams* analysis have also been recently applied to learning macro-actions for the heuristic planner FF (Muise et al., 2009).

4.2 Generalized Policies

A generalized policy is a mapping of planning contexts (sometimes also called meta-states) into the preferred actions to apply in the context. A planning context typically consists of the current state as well as with the set of goals. An accurate generalized policy is able to solve any problem from a given domain without any search having to be done, by repeatedly applying the preferred action for each planning context. Figure 13 shows an example of a generalized policy in the form of a decision list (ordered set of decision rules) for the *Blocksworld* domain. This generalized policy `stacks` block `TOP` on block `BOTTOM` when `TOP` should be on `BOTTOM`, `BOTTOM` is clear and `BOTTOM` is already *well placed*[3]. In all other cases the policy `unstacks` all blocks and `puts` them `down` on the table so they can be placed in their final position.

[3]The concept of *well-placed* should be given in the domain theory. A block is considered *well-placed* when it is on the right block (or table) with respect to the goal, and all blocks beneath it are *well-placed* too.

```
stack(Top,Bottom) :- state_holding(TOP), goal_on(TOP,BOTTOM), state_clear(BOTTOM),
                     wellplaced(BOTTOM).
putdown(BLOCK) :- state_holding(BLOCK).
unstack(TOP,BOTTOM) :- state_clear(TOP), state_armempty().
pickup(BLOCK) :- state_clear(BLOCK), state_armempty().
```

Figure 13 A generalized policy in the form of a *decision list* for the *Blocksworld* domain.

4.2.1 Learning Generalized Policies

- **Knowledge representation**. A key issue when representing planning contexts is selecting the *feature space*. The feature space is the set of predicates considered for the learning process. This set must be general enough to capture the domain's relevant knowledge and specific enough to make the learning tractable. In AP the feature space has usually consisted of predicates for describing the current state and the goals of the planning task. The PRODIGY system enriched the feature space with extra predicates, called *metapredicates* (Minton, 1988). *Metapredicates* capture useful knowledge about the planning context, like the current applicable operators or the goals that still need to be achieved. Recent works on learning generalized policies have extended the definition of metapredicates to include concepts from heuristic planning, like metapredicates for capturing the *actions of the relaxed plan* in the current state (Yoon et al., 2008) or for capturing the set of *helpful actions* in the current state (de la Rosa et al., 2008).
  There have been two main approaches for representing generalized policies. A generalized policy can be represented as a set of rules that capture the preferred action to apply in a given planning context. In this case, the generalized policy is formally defined as a tuple $\pi =< L, R >$, where

  - L the set of literals used to describe the different planning contexts. It defines the feature space used during learning.
  - R is a set of rules, where the rule's `Head` is the action to apply and the `Body` is the set of predicates that describes the planning context under which the action should be applied.

  A generalized policy can also be represented as a set of decision instances with a distance metric to retrieve similar instances. This is the representation approach that CBR (Case-Based Reasoning) planners follow. Even if this representation is able to capture more specific domain regularities, it has the serious drawback of requiring an appropriate similarity metric. Utility is also a problem when handling large collections of decision instances, because, as a collection gets larger, the time it takes to search for similar instances also increases. This second representation for generalized policies is formally defined as a tuple $\pi =< L, I, D >$ where:

  - L the set of literals used to describe the planning contexts. It defines the *feature space*.
  - I is a set of tuples, $i =< c_i, a_i >$, where c_i is an instantiated planning context and a_i is the instantiated action applied in c_i.
  - D is a distance metric that computes the distance between two different planning contexts. Given a new planning context c, the policy decides the action a_i to execute in c by computing the closest tuple in I and returning its associated action a_i, as shown in,

  $$\pi(c) = \arg_{a_i} \min_{(<c_i,a_i>\in I)} D(c, c_i)$$

  Both approaches usually represent planning contexts in predicate logic because it is a natural encoding for the AP tasks. Predicate logic provides mechanisms to define extra predicates that enrich planning contexts. As an example of these extra predicates, Figure 14 shows the definition of the `wellplaced(Block)` predicate for the *Blocksworld* (Khardon, 1999). This concept is not in the *Blocksworld* domain's original coding, but it is useful to define a compact generalized policy as is shown in the example of Figure 13.

```
ingoal(Block) :- goal_ontable(Block).
ingoal(Top) :- goal_on(Top,Bottom).

wellplaced(Block) :- state_ontable(Block), goal_ontable(Block).
wellplaced(Block) :- state_on(Block,Bottom), not(ingoal(Bottom)).
wellplaced(Block) :- state_on(Block,Bottom),goal_on(Block,Bottom),wellplaced(Bottom).
```

Figure 14 Representation of the `wellplaced(Block)` concept for the *Blocksworld* domain in *predicate logic*.

Learning these kinds of predicates in predicate logic is still an open issue so they have to be hand-coded. Planning systems have used other representation languages that succeed in learning these concepts. Languages for describing object classes have been shown to provide a useful bias for learning these concepts. These languages include *concept language* (Martin and Geffner, 2000) and *taxonomic Syntax* (Mcallester and Givan, 1989) which provide operators to define recursive concepts over predicates, like the $*$ *operator*. In the *Blocksworld* domain, for example, the useful concept of *well-placed* block can be defined in a very compact way using these languages (Figure 15).

$$(goal\_ontable = state\_ontable) : wellplaced$$
$$(\forall goal\_on^{*}.\,(goal\_on = state\_on)) : wellplaced$$

Figure 15 *Well-placed* concept for the *Blocksworld* domain expressed in *concept language*.

Expressing planning knowledge in *temporal logic* has also been shown to be useful. This is the case of TLPLAN (Bacchus and Kabanza, 2000). Unfortunately, learning planning knowledge in *temporal logic* has not been resolved.

- **Learning examples**. Learning examples are extracted from solutions to training problems. The learning examples consist of tuples $< c_i, a_{i+1} >$ where c_i is a planning context and a_{i+1} is the action applied at context c_i.
- **Learning algorithm**. When the policy consists of a set of rules, the learning task is very much related to the ILP task. Learning a generalized policy can be defined as the induction of a logic program for every action in the domain. This logic program captures when to apply the action. The head of the rules in this logic program consists of the action name and parameters, and the body is the subset of literals from the planning context that best covers the learning examples. This learning task can be implemented as a heuristic search guided by the coverage of learning examples. When the policy consists of a set of relevant instances, the learning task stores and manages the set of instances. In this case, learning a large number of instances can actually be counterproductive because it is hard to store and manage them and because of the difficulty involved in determining which instance to use to solve a particular problem. A solution to this problem is to post-process the stored instances to only manage the most relevant ones. Examples are REPLICA (García-Durán et al., ress) which extracts a set of prototypes from the instances or DISTILL (Winner and Veloso, 2003) which builds a highly compressed instances library by generalizing and merging solution plans.
- **Exploitation of the learned knowledge**. A generalized policy can be used to directly select which action to apply for a given planning context. When a learned generalized policy is imperfect, however, its application may fail to solve problems like in the case of the early systems in this category. These systems learned control-rules to guide the exploration of planning search trees. A control-rule is an `IF-THEN` rule that proposes node selection, pruning or ordering during tree exploration. A set of control rules can be viewed as a *partial* generalized policy since they do not provide action recommendation for all possible planning contexts. When a given control-rule recommendation was imperfect, it often prevented the planner from finding a solution plan. To exploit generalized policies

more robustly, recent works have exploited the learned policy as an evaluation function within search algorithms like Beam-Search or Limited Discrepancy Search. These search algorithms allow planning to combine the guidance provided by the learned knowledge with other sources of advice like domain-independent heuristics.

4.2.2 Implementations

There are a group of systems that inductively learn control rules. Among them 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. There are also analytical systems: PRODIGY/EBL module (Minton, 1988) learned search control rules for the PRODIGY planner from a few examples of right and wrong decisions. STATIC (Etzioni, 1993) obtains control rules without solving any problems. It simply uses Explanation Based Learning (EBL) to analyze the relations between actions preconditions and effects. With the aim of overcoming the limitations of purely inductive and purely analytical approaches some researchers 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 and induction. It first learns control rules with EBG and then refines them with learning examples. DOLPHIN (Zelle and Mooney, 1993; Estlin and Mooney, 1996) 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 or general and then uses induction to generalize and specialize the rules. EVOCK (Aler et al., 2002) uses *Genetic Programming* to evolve the rules learned by HAMLET and yield more effective search control.

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 did not generalize well when the size of the problems increases. Martin and Geffner solved these limitations in the *Blocksworld* domain by changing the representation language of the generalized policies from predicate logic to *concept language* to learn recursive concepts (Martin and Geffner, 2000).

Recently the scope of generalized policy learning has increased over a variety of domains making this approach competitive with state-of-the-art planners. This achievement is due to two new ideas: (1) the policy representation language is enriched with extra predicates that capture more effective domain-specific knowledge; and (2) the learned policies are applied not greedily but within the framework of heuristic search algorithms. A prominent example is the OBTUSEWEDGE system (Yoon et al., 2007), the best learner of the IPC-2008 learning track. This system enriches the knowledge of the current state with the relaxed planning graph and uses the learned policies to generate look-ahead states within a Best-First Search. ROLLER (de la Rosa et al., 2008) defined the problem of learning generalized policies as a two-step standard classification process. In the first step a classifier captures the preferred operator to be applied in the different planning contexts. In the second one, another classifier captures the preferred bindings for each operator in a given domain's different planning contexts. These contexts are defined by the set of helpful actions extracted from a given state's relaxed planning graph, the goals still unachieved, and the the planning task's static predicates.

Systems also exist that represent generalized policies through collections of planning instances, as CBR systems for AP. Instance-based AP systems are not usually competitive in a variety of domains because they present the drawback of having to define an appropriate similarity metric that works well in different kinds of domains. The PRIAR system (Kambhampati and Hendler, 1992) proposed integrating the modification of plans with generative planning. PRODIGY/ANALOGY (Veloso and Carbonell, 1993) introduced the application of derivational analogy to planning. It stored planning traces to avoid failure paths in future problem solving episodes. 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. DISTILL (Winner and Veloso, 2003) incorporates example plans into a structure called *dsPlanner*. DISTILL converts the plan into a parameterized if-statement and searches through each of the if-statements already stored in the *dsPlanner* to merge them. If the learned *dsPlanner* is accurate it can be used directly to solve any problem from the domain without search. CABALA (de la Rosa et al., 2007) uses object centered solutions plans, called typed sequences, for node ordering during the plan search in heuristic planning. Another instance-based learner, REPLICA, implemented a *Nearest Prototype Learning* (García-Durán et al., ress) using a relational distance inspired by the metrics used in relational data-mining. OAKPlan (Serina, 2010) uses a compact graph structure to encode planning problems. This structure provides a detailed description of the planning problem's topology and allows the learner to define an effective retrieval procedure based on kernel functions.

In (de la Rosa et al., 2009) a planning system was recently described that was able to benefit from generalized policies represented in any of the two approaches (rules or instances). This system queries the policies for fixing the flaws of *relaxed plans*, and then it uses the resulting relaxed plans as grounded macro-actions in a *Best-First Search*.

4.3 Generalized Heuristic Functions

Heuristic functions are used in AP to focus the search of a solution plan on the search nodes that seem the most promising. Heuristic functions for AP compute an estimate of the distance from a given search node to a node in which goals are satisfied. Domain-independent heuristic functions for AP can be directly derived from the cost of the solutions to a relaxed task. The most common relaxation of the AP task consists of ignoring the delete effects of actions. Most of today's heuristic planners rely on this idea to implement their heuristics. Since this approach is domain-independent, it fails to capture planning domains singularities. This section describes how to obtain AP heuristics that capture domain-specific knowledge with ML. It focuses on heuristics for the most popular search approach in AP, the forward state-space search.

4.3.1 Learning Generalized Heuristic Functions

- **Knowledge representation**. They are functions $H(s; A; G)$ of a state s, the action model A and a goals set G, that estimate the cost of achieving the goals G starting from s and using actions from A. *Predicate logic* is a natural encoding for AP heuristic functions because they express knowledge about the current state, goals and actions of the AP task. Representation languages, like *taxonomic Syntax*, that focused on the properties of objects have also been used. Figure 16 shows an example of a simple domain-specific heuristic for the *Blocksworld*. This heuristic function estimates a cost value of 0, 1, 4 or 6 steps to achieve the goal `on(A,B)` with regard to different configurations of the current state.

```
goal_on(A,B).
on(A,B)?
+ yes: [0]
+--no: on(C,A) AND on(D,B)?
       +--yes: [6]
       +--no: on(C,A) OR on(D,B)?
              +--yes: [4]
              +--no: [1]
```

Figure 16 Domain-specific heuristic for the *Blocksworld*.

- **Learning examples**. Learning examples consist of tuples $< s_i, c_i, g_i >$ where s_i is the current state and c_i is the actual cost of achieving the goals g_i from the state s_i.

- **Learning algorithm**. The aim of the learning algorithm is generalizing the cost values captured in the learning examples. Since the target concept to be generalized from the learning examples is numeric, this learning task corresponds to a regression task. When learning examples are represented in predicate logic, standard relational regression algorithms can be used, such as learning relational regression trees (Blockeel et al., 1998).
- **Exploitation of the learned knowledge**. The main advantage of this approach is that the learned knowledge can be directly combined with other standard sources of guidance for AP such as domain-independent heuristics. The learned knowledge is difficult for humans to understand.

4.3.2 Implementations

In (Yoon et al., 2006), the authors build a generalized heuristic function through linear regression. They learn domain-specific corrections $\Delta(s; A; G) = \sum_i w_i * f_i(s; A; G)$ to the *relaxed plan heuristic* $RPH(s; A; G)$ introduced by the FF planner (Hoffmann and Nebel, 2001b). These corrections are expressed as a weighted linear combination of features, where w_i are the weights and f_i represent the different features of the planning context. The regression examples consist of observations of the true distance to the goals from different states obtained with the FF planner. The resulting heuristic function,

$$H(s; A; G) = RPH(s; A; G) + \Delta(s; A; G)$$

provides more accurate estimates that capture domain-specific regularities.

The previous approach ignores the actual performance of the heuristic when used in a search algorithm. It may attempt to learn corrections for the heuristic even if it provides good guidance during search. In (Xu et al., 2007), they present an alternative approach that only considers learning when the heuristic misleads in a given search strategy. The heuristic learned by this approach,

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

only attempts to discriminate between good and bad states well enough to find the goals within a *Beam-Search* procedure, rather than attempting to model the distance to the goals precisely. This approach implements the following weight learning strategy. For each state s_j in the solution to a learning problem, if s_j is not contained in the search's $beam_j$, there is a search error. In this case, weights are updated like the *Perceptron*'s weights, so that state s_j will be preferred by the heuristic, remaining in the $beam_j$ in future search episodes.

4.4 Hierarchical Decomposition Methods

An extended strategy for coping with problems complexity is to decompose problems into simpler sub-problems. When an effective decomposition is found, the sum of the costs of addressing the sub-problems is smaller than the cost of directly addressing the original problem. Hierarchical Task Networks (HTN) is one of the best studied approaches for modelling AP task decomposition. The HTN approach combines hierarchical domain-specific representations of the planning task and domain-independent search strategies for problem solving.

The input to a HTN planner includes an action model that encodes a set of *primitive actions* similar to the STRIPS actions used in classical planning and a *task model*. The task model defines a set of *methods* which describes how tasks should be decomposed into subtasks in a particular domain. The job of the HTN planner consists of exploiting the task model to decompose a given planning task into simpler subtasks until a sequence of primitive actions is generated. Figure 17 shows the primitive action `unstack` and the method for the task `move-block` from a HTN description of the *Blocksworld* domain. This method moves a block to its final position by defining two subtasks: `moving-x-from-y-to-z` that unstacks block x from y, in order to stack x on z, and `moving-x-from-table-to-z` that picks-up block x from the table and stacks it on z. In both subtasks, the method *checks* that the block x can be directly moved to its final position (primitive actions *check*, *check2* and *check3*) and makes sure that x will not be moved in the future.

```
(:operator (!unstack ?top ?bottom)
             ;preconds
             ((clear ?top) (on ?top ?bottom))
             ;effects
             ((holding ?top) (clear ?bottom)))

(:method (move-block ?x ?z) ;head
           method-for-moving-x-from-y-to-z
           ;preconds
           (:first (on ?x ?y))
           ; Decomposition
           ((!unstack ?x ?y) (!stack ?x ?z)
            (!assert ((dont-move ?x)))
            (!remove ((stack-on-block ?x ?z)))
            (check ?x) (check2 ?y) (check3 ?z))

           method-for-moving-x-from-table-to-z
           ;preconds
           nil
           ; Decomposition
           ((!pickup ?x) (!stack ?x ?z)
           (!assert ((dont-move ?x)))
           (!remove ((stack-on-block ?x ?z)))
           (check ?x)))
```

Figure 17 Examples of operator and method for HTN planning in the *Blocksworld* domain.

Current HTN planners, such as SHOP2 (Nau et al., 2003), can outperform state-of-the-art domain-independent planners and provide a natural modelling framework for many real-world applications like fire extinction (Castillo et al., 2006), evacuation planning (Muñoz-Avila et al., 1999) or game playing (Nau et al., 1998). Defining effective *decomposition methods* is still complex because these methods reflect deep knowledge of the domain. The operator and method shown in Figure 17 belong to a HTN representation of the block-stacking algorithm (Erol et al., 1992). Below we review some learning approaches to automatically define decomposition methods within the HTN planning paradigm.

4.4.1 Learning Hierarchical Decomposition Methods

1. **Knowledge representation**. A HTN method is a procedure that describes how to decompose a *non-primitive* task into *primitive* tasks or simpler *non-primitive* tasks. A method m is formally defined by a triple m =<*head, preconds, subtasks*>, where *head* represents the name and parameters of the task to decompose, *preconds* is a logical formula denoting the method preconditions, and *subtasks* is a partially ordered sequence of subtasks. A method m is applicable to a state s and task t if *head*(m) matches t and the *preconds*(m) are satisfied in s. The result of applying a method m to a state s and task t is the sequence of subtasks *subtasks*(m).
2. **Learning examples**. Learning examples consist of a set of planning problems, that is a set of pairs $< s_0, G >$ where s_0 is an initial state and G is a set of goals, together with the corresponding solution plans $p = (a_1, a_2, \ldots, a_n)$. These solution plans can be human provided or generated by a classical planner.
3. **Learning algorithm**. The algorithms reviewed in Section 3 to learn the action model preconditions can also be applied to learn the HTN method preconditions. A reduced version of the HTN method decompositions can be induced by using hierarchical layers of macros. Decompositions induced following this approach do not exploit the HTNs full expressive power that includes definitions of alternative decompositions, recursive concepts or loops. At this point, there is no completely automatic algorithm for learning subtask decompositions that benefits from HTNs full expressiveness. Some algorithms need to start from *decomposition methods* partially specified by humans, others need hierarchical plans (which implies previously specified methods) and some others need certain tasks called *annotated tasks* specified that define the equivalence of the HTN tasks and the set of goals G in a classical version of the AP problem.

4. **Exploitation of the learned knowledge**. The generality level of the learned knowledge is the key issue to make a given HTN description effective. Learning decomposition methods that are too general may produce infinite recursion when problem solving, decreasing the advantages of HTN planning over classical planning. On the other hand, learning too specific HTN methods is less likely to be effective in decomposing new problems. Current methods to address this trade-off are based on measuring the performance of the learned model against a set of test problems.

4.4.2 Implementations

CAMEL (Ilghami et al., 2002) learns the preconditions of HTN *decomposition methods* from observations of plan traces, using the *version space* algorithm (Mitchell, 1997). CAMEL is designed for domains where the planner receives multiple methods per task, but not their preconditions. The structure of the hierarchy here is known in advance and the learning task tries to identify under which conditions the different hierarchies are applicable. This approach requires many plan traces to completely converge (completely determine the preconditions of all methods). CAMEL++ (Ilghami et al., 2005) 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. Both CAMEL and CAMEL++ require each input plan trace needs to contain extra information so that the learner could acquire the models. At each decomposition point in a plan trace, the learner needs to have all the applicable method instances, not just the one that was actually used.

The HTN Domain Learning (HDL) algorithm (Ilghami et al., 2006) starts with no prior information about the methods. It examines hierarchical plan traces produced by an expert problem-solver. For each decomposition point in the traces, HDL checks to see if the responsible method is already present. If not, HDL creates a new method and initializes a new *version space* to capture its preconditions. The method that was actually used to decompose the corresponding task serves as a positive example and the methods that matched that task but whose preconditions failed, serve as negative examples for the corresponding version space.

HTN-Maker (Hogg et al., 2008) generates an HTN domain model from a STRIPS domain model, a collection of plans p generated by a STRIPS planner, and a collection of *annotated tasks*. An *annotated task* is a triple *(n, Pre, Effects)* where n is a task, *Pre* is a set of atoms known as the preconditions, and *Eff* is a set of atoms known as the effects. HTN-Maker first generates the list of states $S = (s_0, \ldots, s_n)$ by applying the actions in the plan p starting from the initial state s_0. Then, it traverses these states, and if there is an annotated task whose effects match the state s_{i+n} and whose preconditions match the state s_i, it regresses the effects of the annotated task through (1) previously learned methods or (2) a primitive task if there are no previously learned methods.

4.5 Learning Planning Search Control in Domains with Uncertainty

When planning in domains with uncertainty, the combinatorial explosion problem becomes even more acute. Partial observability of the environment and non-deterministic outcomes of actions multiply the number of states that can be reached, and increase the complexity of the search processes. Search control knowledge can help reduce this complexity but as mentioned before, hand-coding this knowledge is difficult and requires expertise on the domain and the planner. There is a review of techniques for *Planning Under Uncertainty* (PUU) that improve their own scalability through ML. These techniques preserve the representation formalisms and learning algorithms reviewed in the section but adapt the mechanisms to generate the learning examples and apply the learned knowledge to the PUU framework:

- *Macro-actions*. In (Theocharous and Kaelbling, 2003), the authors solve large Partially Observable Markov Decision Processes (POMDP) by using a dynamically-created finite-grid approximation of the belief space and then applying *dynamic programming* to compute a value function at the grid points. This work exploits macro-actions to visit a significantly smaller part of the belief space of the POMDP than with simple primitive actions. Macro-actions also make the convergence of the *dynamic programming* faster because the agent can look further into the future.

- *Generalized policies*. Given a set of small probabilistic planning problems, the work presented in (Yoon et al., 2002) uses the probabilistic planner PGRAPHPLAN to synthesize solution 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 action a_i in the state s_{i-1} and s_n is a goal state. The execution of plans p are simulated one action at a time. If the state resulting from simulating a_i is different from s_i, then a new plan is computed to attain the goals in the new state. This process is repeated until a goal state is reached or a limit of simulations is exceeded. A generalized policy is then learned from the pairs *(state,action)* traversed in the simulations. In many domains, solutions to small problems do not capture the singularities of larger problems. As an alternative, in (Fern et al., 2006) the authors learn the generalized policy from pairs *(state, action)* generated following the Approximate Policy Iteration algorithm (Bertsekas and Tsitsiklis, 1996).
- *Generalized evaluation functions*. Given a set of small probabilistic planning problems, the approach presented in (Gretton and Thiébaux, 2004) computes the optimal policies for the problems with a MDP solver. Then, it uses first-order regression to generalize a first-order value function over the values of optimal policies. In (Sanner and Boutilier, 2005, 2006) the authors solve First Order MDPs (FOMDPs) by representing the value function linearly with respect to a set of first-order basis functions and computing suitable weights by casting the corresponding optimization as a first-order linear program.
- *Decomposition methods*. In (Hogg et al., 2009), the HTN-MAKER algorithm has been adapted to non-deterministic planning domains, where actions may have multiple possible outcomes.

4.6 Discussion

IPC-2008 created a specific track for AP systems that benefit from learning search control. This competition was the first forum to compare the performance of different planning and learning systems in a common test-bench. Most of the participants based their proposals on heuristic planning and presented relevant advances in the field like extensions to the planning context with information from the relaxed planning graph or techniques to combine learned knowledge with other sources of guidance like domain-independent heuristics. One of the most important conclusions drawn from the competition was that none of the participant systems is successful in capturing useful knowledge in all the proposed domains. In some domains, the representation language's bias prevented the systems from learning this knowledge, in others the cause was the learning examples' poor quality or the learning algorithms' limitations. It became clear that the best approach for learning useful knowledge in a given domain need to be ascertain. This task involves better understanding:

1. What the best representation for a given domain is. IPC-2008 showed that action policies are inaccurate in domains like *Sokoban* and *N-puzzle* because they look over knowledge regarding the trajectory on the way to the goals. Numeric functions, like the Manhattan distance that provides a lower-bound for the solution's length, produce useful search control for these domains. Further studies are needed to establish theoretical considerations about the expressive power and/or limitations of the various forms of learned knowledge. These studies should also cover other kinds of planning paradigms able to exploit more expressive representations than off-the-shelf planners like planners using temporal logic (Bacchus and Kabanza, 2000), or AP systems able to reason about loops (Srivastava et al., 2008; Winner and Veloso, 2003; Shavlik, 1989; Larkin et al., 1986).
2. What the best learning examples for a given domain are. In most of the systems reviewed, the learning examples are extracted from experience collected solving problems from a training set. These examples can be collected in different ways to suit the learning task's singularities:

   (a) *Quality*. In order to learn search control knowledge that provides accurate guidance towards the goals, learning examples should correspond to good quality solutions. Since optimal algorithms for AP are expensive in terms of time, a more practical approach for generating good solutions consists of using *Branch and Bound* strategies (de la Rosa et al., 2009).

(b) *Diversity*. Some domains, like *logistics*, present many solutions for a given planning problem that correspond to swaps in the order of certain actions. These solutions express different decisions for the same planning context, which makes learning more complex. In order to avoid this effect, learning can be completed from partially ordered plans (Veloso et al., 1990). Partially-ordered plans contain the same set of actions as the totally-ordered plans, but partially-ordered plans only include the necessary constraints on the action-ordering.

(c) *Eagerness*. Learning examples can correspond to every planning decisions (*eager approach*) or just to decisions where the planner failed to make the desired choice (*lazy approach*), i.e., the planner choice was not part of the solution or was not optimal. A *lazy approach* is planner-dependent, but it generates more compact search control knowledge.

3. What the best learning algorithm for a given domain is. Different learning algorithms present different biases that affect the learning process like where decision lists learning algorithms specify the maximum number of literals per rule. Different learning algorithms can also present different memory and learning time performance and, depending on the algorithm, the resulting model can present a different readability and applicability. Lazy learning algorithms present low learning times but large times when exploiting that learned knowledge. Recently, works on meta-learning (Brazdil et al., 2009) try to compute estimates that characterize these features of the learning algorithms but they normally focus on propositional versions of the ML algorithms.
4. When to exploit the learned knowledge. Most of the participants in the learning track of IPC-2008 suffered from performance problems caused by the use of imperfect learned knowledge. In practice, there is no sign of a given piece of search control knowledge's quality and the only possible evaluation is to compare the performance of the planning system with and without the learned knowledge. This *evaluation by testing* strategy is the underlying motivation of AP systems based on portfolio of planners (Howe et al., 1999; Vrakas et al., 2005). These systems evaluate different planning configurations in a set of training problems to obtain the planning setup that allows them to address the target problems more robustly. This approach was also followed by the winner of the IPC-2008 learning track (Gerevini et al., 2009a).

5 Reinforcement Learning

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). Building a RL agent involves similar decisions to the ones in learning for AP: the representation (how to encode the agent's environment and actions); the learning examples (the agent's strategy for collecting experience); the learning algorithm (what kind of algorithm performs the best for the specified task); and the exploitation of the learned knowledge (how can the agent benefit from the obtained knowledge). Unlike most of the learning for AP approaches, RL present tightly integrated solutions for knowledge acquisition and knowledge exploitation. A key issue in these integrated solutions 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).

5.1 Model-Based and Model-Free RL

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. *Model-Based* RL typically relies on standard *dynamic programming* algorithms (Bellman and Kalaba, 1965; Bertsekas, 1995) to find a value/heuristic function that provides optimal policies. Learning in *Model-Based* RL is understood as 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.

Model-Free RL does not require a model of the environment. Below, there are two different approaches for the *Model-Free* RL task:

- *Combining learning action models with Model-Based RL.* This approach learns a transition model of the environment and applies standard *dynamic programming* algorithms to find a good policy.
- *Pure Model-free RL.* 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.

5.2 Relational Reinforcement Learning

This section reviews Relational Reinforcement Learning (RRL) (Dzeroski et al., 2001; Otterlo, 2009) given that RRL uses a knowledge representation similar to one used in learning for AP. The section focuses on *pure model-free* RRL since techniques for learning relational action models were already discussed in Section 3. Further explanations on model-based RRL can be found in (Boutilier et al., 2001; Wang et al., 2007; Sanner and Kersting, 2010) also referenced as Relational, Symbolic or First Order Dynamic Programming.

1. **Knowledge representation**. The learned knowledge is represented in the form of a *First Order policy* $\pi : S \rightarrow A$ that maps relational states– coded in predicate logic– into the preferred action to apply next for achieving a set of *First Order goals*. Unlike generalized policies for AP (Section 4), RRL policies do not include information on different goals. RRL policies only capture how to solve a particular task or a particular set of tasks, like the same task with different number of objects. RRL requires consequently learning from scratch or at least, some kind of *transfer learning* (Taylor and Stone, 2007; Mehta et al., 2008) when the nature of goals change. *Transfer learning* is defined as exploiting data from one or more source tasks to improve learning performance in a different target task.
   A *First Order policy* can be implicitly represented as a *First Order Q-function* $Q(< S, A >) = R$ that maps pairs state-action $S - A$ into their expected reward R. This representation is closely related to the concept of heuristic function used in AP that maps states into an estimate of the cost to pay (negative reward) for achieving the goals. In AP this cost estimate can be automatically extracted from the problem representation by ignoring the *delete* effects of actions.
2. **Learning examples**. Learning examples consist of tuples $< s_i, a_i, s_{i+1}, r_i >$ where a_i is the action applied at state s_i, s_{i+1} is the resulting state and r_i is the obtained reward. Learning examples in RRL are typically generated from random exploration.

3. **Learning algorithm**. There are different approaches for model-free RRL:

   - *Relational learning of the q-value function.* This approach uses relational regression to generalize the value function. Figure 18 shows a relational regression tree that captures the *q-values* of action *move(Block,Block)* when solving the set of tasks *on(Block,Block)* in a three blocks *Blocksworld.*

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

Figure 18 Relational regression tree for the goals *on(X,Y)* in a three blocks *Blocksworkd.*

   - *Relational learning of the optimal policy.* An alternative approach is to directly learn the policy. This approach requires relational classification instead of relational regression. The advantage of this approach is that, usually, it is easier to understand policies than to understand value functions for structured domains. Figure 19 shows a RRL policy that captures when the selection of action *move(Block,Block)* is optimal for solving the set of tasks *on(Block,Block)* in a three blocks *Blocksworld* domain.

```
goal_on(A,B),move(D,E).
above(D,A)?
+--yes: equal(E,B)?
|       +--yes: nonoptimal
|       +--no: optimal
+--no: above(D,B)?
       +--yes: optimal
       +--no: move(A,B)?
              +--yes: optimal
              +--no: nonoptimal
```

Figure 19 RRL policy for the goals *on(X,Y)* in a three blocks *Blocksworkd.*

5.3 Implementations

With regard to learning a relational *Q-function*, different relational regression algorithms have been tried:

- Relational regression trees (Dzeroski et al., 2001). For each pair *(action, relational-goal)*, a regression tree is built from a set of examples with the form *(state,q-value)*. Leaf nodes of the tree represent the predictions of the *q-value*. Test nodes of the tree represent the facts that have to hold for the predictions to come true.
- Instance based algorithms with relational distance (Driessens and Ramon, 2003). This work implements a k-nearest neighbor prediction. The prediction computes a weighted average for the *q-values* of the collected examples. Weights used for the prediction are inversely proportional to the distance to the examples. The used distance copes with relational representations for states and actions.
- Relational Kernels methods (Gartner et al., 2003). These methods use the incrementally learnable *Gaussian processes* for Bayesian-regression to build mappings of *(state,action)* pairs into *q-values*. Graph kernels are used as the covariance function between state-action pairs to employ *Gaussian processes* in the relational setting.

With regard to directly learning a relational policy, (Dzeroski et al., 2001) uses relational decision trees to determine which actions are optimal in the different environment's states when achieving a particular task.

5.4 Discussion

The aims of RL are closely related to the aims of learning for AP. 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.

6 Conclusions

This section summarizes the contents of the paper and discusses some issues identified in the review that are still open.

6.1 Summary

Before the mid 90's ML was used exhaustively in AP to learn search control knowledge that improved the performance of planners. During this period planners implemented uninformed search algorithms, so ML helped planners achieve better performance in many domains. In the late 90's the planning community's interest in ML waned for two reasons. The appearance of powerful domain-independent heuristics boosted the performance of planners. Suddenly, the baseline for evaluating the benefit of ML shifted dramatically. That said, existing relational learning algorithms were inefficient and performed poorly in a diversity of domains.

Today 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 2005 the International Competition on Knowledge Engineering for Planning Systems (ICKEPS) began and in 2008 a learning track was opened at IPC. Workshops on planning and learning have taken place periodically at the International Conference on Automated Planning and Scheduling. ML seems to be once again one of the solutions to the challenges of AP.

In the paper we focused on two of AP's challenges: defining effective AP action models and defining search control knowledge to improve the planners performance. For the first challenge, we reviewed systems that learned action models with diverse forms of preconditions and effects. These systems generally present learning algorithms that are effective when the appropriate learning examples are collected although it is still not clear how to automatically collect good sets of learning examples in AP. When learning results in imperfect action models, there are few mechanisms for diagnosing the flaws of the models or algorithms to robustly plan with them.

For the second challenge, we reviewed different forms of search control knowledge to improve off-the-shelf planners. This kind of knowledge has been shown to boost the performance of planners in particular domains. Learning effective search control knowledge over a collection of domains is still challenging since different planning domains may present very different structures. We also reviewed learning search control for HTN planners which is a more expressive form of control knowledge that uses a different planning paradigm based on domain-specific problem decomposition. In this planning paradigm, search control knowledge and action models are not separated in the domain theory's definition.

The paper also reviewed techniques for RRL, a form of RL that, like AP, uses predicate logic to represent states and actions. Though current RRL techniques can solve relational tasks, they are focused on achieving a particular set of goals and present generalization limitations in comparison with learning for AP techniques. Furthermore, they have problems collecting significant experience for complex tasks like the ones traditionally addressed in AP, where achieving a particular goal may undo previously satisfied goals.

6.2 Open Issues

Below, we provide with a list of what we consider to be open issues or future avenues in learning for AP. We divide these issues based on the paper's main four topics: knowledge representation, learning examples, learning algorithms, and exploitation of the learned knowledge:

- **Knowledge representation**. Finding an effective knowledge representation for AP over a collection of domains is still an open issue. If the chosen representation is not able to express key knowledge in a given domain, the learning algorithms will not generate effective AP knowledge for this domain. Examples of this key knowledge are the concepts of *well-placed* block in the *Blocksworld* domain or the *in-final-city* concept in the *Logistics* domain (Khardon, 1999). Further studies need to induce what the key knowledge for a given domain would be and what the suitable representation for expressing it would be. In this regard, the work presented in (Yoon et al., 2008) used a set of testing problems to evaluate the utility of the different features of the planning context when learning a generalized policy. Another potentially relevant aspect is how the representation language affects the ability to learn effective knowledge. In (Martin and Geffner, 2000; Cresswell et al., 2009) the authors showed the impact of learning using object centered representations for AP but the use of new standard representations such as SAS+ (Helmert, 2009) instead of PDDL has not yet been studied in detail.
  More expressive representation formalisms like programs, temporal formulae or hierarchical representations have been shown to describe abstract structures such as loops or hierarchies that represent effective knowledge for many AP domains. Most of the methods for learning these representations however need extra labelling such as annotations to mark abstract tasks, the beginning and end of loops, end-conditions, etc. This kind of labelling cannot easily be obtained automatically from observed executions. This knowledge cannot be directly used by off-the-shelf planners and requires

"ad-hoc" planning systems able to exploit this kind of knowledge. Further studies in learning and in compiling it for off-the-shelf planners are needed.

- **Learning examples**. Implementing mechanisms to autonomously collect learning examples for AP is still an open issue. Random explorations typically undersample the AP state and action spaces. AP actions present preconditions that are only satisfied by specific sequences of actions which have a low probability of being chosen by chance. Learning examples can be extracted from solutions to AP training problems. Traditionally, these training problems are obtained by random generators with some parameters to tune the problems difficulty. This approach has two limitations. (1) It is not trivial to guarantee AP problems' solvability. Showing that from a given initial state, one can reach the goals is as hard as solving the original AP task. (2) The AP problem generators' parameters are domain-specific. The number and type of the world objects are different for different AP domains and tuning these parameters to generate good quality learning examples implies domain expertise. In (Fern et al., 2004) Random Walks were introduced to not have to have domain expertise to obtain good learning examples. Random Walks is an automatic and domain-independent strategy for automatically generating problems of gradually increasing walk length. In domains where complexity depends on the number of world objects, however, this approach is not enough. An alternative approach consists of generating training problems using an active learning approach starting from a previous state (Fuentetaja and Borrajo, 2006). In this case the learning process has the bias imposed by the above mentioned problem, but it can generate a random exploration of problems of increasing difficulty in domains where complexity depends on the number of world objects. This question is still open given that when the domain is asymmetrical, the latter approach could generate unsolvable problems. Further studies have to be undertaken to come up with more general solutions.
- **Learning algorithms**. The application and combination of new ML algorithms for AP is still an open issue. ML is constantly producing new learning algorithms with potential applications for AP. In the case of relational learning AP has only benefited from algorithms for classification and regression but existing relational learning algorithms cover almost all machine learning tasks. Algorithms for relational clustering or for inferring association rules (Raedt, 2008) have still not been explored in AP. ML algorithms for propositional data can be adapted to the relational setting used in AP (Mourão et al., 2010). Another example of applying new ML algorithms to AP is the use of kernel functions to effectively match planning instances (Serina, 2010). Despite all this new ML technology, learning algorithms present biases that affect the learning process. Research in ML for AP can take up new research directions to study the combination of different learning algorithms to reduce the negative effect of a given algorithm bias (Kittler, 1998).
- **Exploitation of the learned knowledge**. These open issues in learning for AP frequently result in imperfect learned knowledge. As the IPC-2008 learning track showed, directly applying defective learned knowledge may damage the AP process performance. The greedier the application of the learned knowledge, the more sensitive the AP algorithms are to the flaws of the learned knowledge. There is a recent line of work on the development of planning and learning systems that aims to build mechanisms to guarantee robust planning even when harmful knowledge is learned. These kind of systems do not only focus on the learning process but also on a robust application of the learned knowledge. Integrating learned knowledge with domain-independent heuristics generated a planning system that was competitive with the state-of-the-art planners in a large variety of domains (Yoon et al., 2007, 2008). Another interesting line of research is into the development of mechanisms for evaluating the quality of the learned domain control knowledge. Such mechanisms could be useful for defining different exploitation strategies. The only evaluation mechanisms currently available for the learned knowledge are empirical and consist of using the learned knowledge to solve a set of training examples to estimate its performance. Apart from the competition, another way of exploiting learned knowledge in AP is through interaction with the final user. Mixed initiative planning systems (Ferguson et al., 1996; Bresina et al., 2005; Cortellessa and Cesta, 2006) propose knowledge that can be verified and modified by the user before being applied.

This review focused on learning two inputs of the AP process, the action model and knowledge for search control. There is a full set of techniques that aim to capture information about the third input to the AP process, the problem. Plan recognition techniques (Charniak and Goldman, 1993; Bui et al., 2002) try to infer an agent's goals and plans by observing the agent's actions. The plan recognition task traditionally assumes there is a library with the space of possible plans to be recognized but recent works do not require the use of this library. These works (Ram'ırez and Geffner, 2009; Ramírez and Geffner, 2010) can also compute the goals that account for the observed examples using slightly modified planning algorithms. Further studies are needed to analyze the relationship of plan recognition techniques with the learning techniques for AP reviewed in this paper.

References

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. 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.

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

Bellingham, J. and Rajan, K. (2007). Robotics in remote and hostile environments. *Science*, 318(5853):1098–1102.

Bellman, R. and Kalaba, R. (1965). *Dynamic Programming and Modern Control Theory*. Academic Press.

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*.

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.

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

Blockeel, H., Raedt, L. D., and Ramong, J. (1998). Top-down induction of clustering trees. In *In Proceedings of the 15th International Conference on Machine Learning*.

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

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

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., Mller, M., and Schaeffer, J. (2005a). Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, 24:581–621.

Botea, A., Müller, M., and Schaeffer, J. (2007). Fast Planning with Iterative Macros. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-07*, pages 1828–1833.

Botea, A., Mller, M., and Schaeffer, J. (2005b). Learning partial-order macros from solutions. In *ICAPS05*, pages 231–240.

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

Brazdil, P., Giraud-Carrier, C., Soares, C., and Vilalta, R. (2009). *Metalearning: Applications to Data Mining*. Cognitive Technologies. Springer.

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

Bui, H. H., Venkatesh, S., and West, G. (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.

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)*.

Charniak, E. and Goldman, R. P. (1993). A bayesian model of plan recognition. *Artif. Intell.*, 64(1):53–79.

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.

Cortellessa, G. and Cesta, A. (2006). Evaluating Mixed-Initiative Systems: An Experimental Approach. In *Proceedings of the 16th International Conference on Automated Planning & Scheduling, ICAPS-06*.

Cresswell, S., McCluskey, T. L., and West, M. (2009). Acquisition of object-centred domain models from planning examples. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*.

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

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

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

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., García-Durán, R., Fernández, F., García-Olaya, A., and Borrajo, D. (2009). Three relational learning approaches for lookahead heuristic planning. In *Working notes of the ICAPS'09 Workshop on Planning and Learning*.

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 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 Automated Planning and Scheduling*.

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

Erol, K., Nau, D. S., and Subrahmanian, V. S. (1992). On the complexity of domain-independent planning. *Artificial Intelligence*, 56:223 – 254.

Estlin, T. A. and Mooney, R. J. (1996). Hybrid learning of search control for partial-order planning. In *In New Directions in AI Planning*, pages 115–128. IOS Press.

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

Ferguson, G., Allen, J. F., and Miller, B. (1996). Trains-95: Towards a mixed-initiative planning assistant. In *International Conference on Artificial Intelligence Planning Systems, AIPS96*, pages 70–77. AAAI Press.

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.

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.

Florez, J. E., Garca, J., Torralba, A., Linares, C., Garca-Olaya, A., and Borrajo, D. (2010). Timiplan: An application to solve multimodal transportation problems. In *Proceedings of SPARK, Scheduling and Planning Applications woRKshop, ICAPS'10*.

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.

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

García-Durán, R., Fernández, F., and Borrajo, D. (2006). Combining macro-operators with control knowledge. In *ILP*.

García-Durán, R., Fernández, F., and Borrajo, D. (In Press). A prototype-based method for classification with time constraints: A case study on automated planning. *Pattern Analysis and Applications Journal*.

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

Gartner, T., Driessens, K., and Ramon, J. (2003). Graph kernels and gaussian processes for relational reinforcement learning. In *International Conference on Inductive Logic Programming, ILP 2003*.

Gerevini, A., Saetti, A., and Vallati, M. (2009a). An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009b). Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668.

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

Gil, Y. (1992). *Acquiring Domain Knowledge for Planning by Experimentation*. 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*.

Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*.

Hernández, C. and Meseguer, P. (2007). Improving LRTA*(k). In *International Joint Conference on Artificial Intelligence, IJCAI-07*, pages 2312–2317.

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

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

Hogg, C., Kuter, U., and Muñoz-Avila, H. (2009). Learning hierarchical task networks for nondeterministic planning domains. In *International Joint Conference on Artificial Intelligence. IJCAI-09*.

Hogg, C., Muñoz-Avila, H., and Kuter, U. (2008). HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *National Conference on Artificial Intelligence (AAAI'2008)*.

Howe, A. E., Dahlman, E., Hansen, C., Scheetz, M., and Mayrhauser, A. V. (1999). Exploiting competitive planner performance. In *In Proceedings of the Fifth European Conference on Planning*.

Ilghami, O., Mu noz-Avila, H., Nau, D. S., and Aha, D. W. (2005). Learning approximate preconditions for methods in hierarchical plans. In *International Conference on Machine learning*.

Ilghami, O., Nau, D. S., and Muñoz-Avila, H. (2002). CaMeL: Learning Method Preconditions for HTN Planning. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling*, pages 131–141. AAAI Press.

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*.

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

Jess Lanchas, Sergio Jimnez, F. F. and Borrajo, D. (2007). Learning action durations from executions. In *Working notes of the ICAPS'07 Workshop on AI Planning and Learning*.

Jiménez, S., Fernández, F., and Borrajo, D. (2008). The PELA architecture: integrating planning and learning to improve execution. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08). Chicago, Illinois, USA*.

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

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

Keller, R. (1987). *The Role of Explicit Contextual Knowledge in Learning Concepts to Improve Performance*. PhD thesis, Rutgers University.

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

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

Kittler, J. (1998). *Combining classifiers: A theoretical framework*. Number 1.

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.

Larkin, J., Reif, F., and Carbonell, J. (1986). Fermi: A flexible expert reasoner with multi-domain inference. *Cognitive Science*, 9.

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.

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 Givan, R. (1989). Taxonomic syntax for first order inference. *Journal of the ACM*, 40:289–300.

McCluskey, T. L. (1987). Combining weak learning heuristics in general problem solvers. In *IJCAI'87: Proceedings of the 10th international joint conference on Artificial intelligence*, pages 331–333, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

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)*.

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

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.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2008). Using kernel perceptrons to learn action effects for planning. In *Proceedings of the International Conference on Cognitive Systems (CogSys 2008)*.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2010). Learning action effects in partially observable domains. In *European Conference on Artificial Intelligence*.

Muggleton, S. (1995). Stochastic logic programs. In *International Workshop on Inductive Logic Programming*.

Muise, C., McIlraith, S., Baier, J. A., and Reimer, M. (2009). Exploiting N-gram analysis to predict operator sequences. In *19th International Conference on Automated Planning and Scheduling (ICAPS)*.

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., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

Nau, D. S., Smith, S. J., and Erol, K. (1998). Control strategies in htn planning: Theory versus practice. In *In AAAI-98/IAAI-98 Proceedings*.

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*.

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*.

Otterlo, M. V. (2009). *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for Adaptive Sequential Decision Making under Uncertainty in First-Order and Relational Domains*. IOS Press.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352.

Porteous, J. and Sebastia, L. (2004). Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278.

Quinlan, J. and Cameron-Jones, R. (1995). Introduction of logic programs: FOIL and related systems. *New Generation Computing, Special issue on Inductive Logic Programming*.

Raedt, L. D. (2008). *Logical and Relational Learning*. Springer, Berlin Heidelberg.

Ram'ırez, M. and Geffner, H. (2009). Plan recognition as planning. In *IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence*.

Ramírez, M. and Geffner, H. (2010). Probabilistic plan recognition using off-the-shelf classical planners. In *National Conference on Artificial Intelligence (AAAI'2010)*.

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.

Rivest, R. L. (1987). Learning decision lists. *Machine Learning*, 2(3):229–246.

Sanner, S. and Boutilier, C. (2005). Approximate linear programming for first-order mdps. In *In Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence*, pages 509–517.

Sanner, S. and Boutilier, C. (2006). Practical linear value-approximation techniques for first-order mdps. In *Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence*.

Sanner, S. and Kersting, K. (2010). Symbolic dynamic programming for first–order pomdps. In M. Fox, D. P., editor, *Twenty–Fourth AAAI Conference on Artificial Intelligence (AAAI–10)*, Atlanta, USA. AAAI Press.

Serina, I. (2010). Kernel functions for case-based planning. *Artif. Intell.*, 174(16-17):1369–1406.

Shavlik, J. W. (1989). Acquiring recursive and iterative concepts with explanation-based learning. In *Machine Learning*.

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.

Srivastava, S., Immerman, N., and Zilberstein, S. (2008). Learning generalized plans using abstract counting. In *National Conference on Artificial Intelligence (AAAI'2008)*.

Strehl, A. L. and Littman, M. L. (2005). A theoretical analysis of model-based interval estimation. In *Proceedings of the Twenty-second International Conference on Machine Learning (ICML-05*, pages 857–864.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

Taylor, M. E. and Stone, P. (2007). Cross-domain transfer for reinforcement learning. In *International Conference on Machine Learning. ICML*.

Theocharous, G. and Kaelbling, L. P. (2003). Approximate planning in POMDPs with macro-actions. In *In Proceedings of Advances in Neural Information Processing Systems 16*.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of pddl axioms. *Artificial Intelligence*, 168(1-2):38–69.

Veloso, M. M. and Carbonell, J. G. (1993). Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278.

Veloso, M. M., Pérez, M. A., and Carbonell, J. G. (1990). Nonlinear planning with parallel resource allocation. In *In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pages 207–212. Morgan Kaufmann.

Vrakas, D., Tsoumakas, G., Bassiliades, N., and Vlahavas, I. P. (2005). HAPRC: an automatically configurable planning system. *AI Commun.*, 18(1):41–60.

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

Wang, 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.

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.*

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., 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. (2007). Using learned policies in heuristic-search planning. In *International Joint Conference on Artificial Intelligence.*

Yoon, S., Fern, A., and Givan, R. (2008). Learning control knowledge for forward search planning. *J. Mach. Learn. Res.*, 9:683–718.

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.*

Zhuo, H., Li, L., Yang, Q., and Bian, R. (2008). Learning action models with quantified conditional effects for software requirement specification. In *ICIC '08: Proceedings of the 4th international conference on Intelligent Computing*, pages 874–881, Berlin, Heidelberg. Springer-Verlag.

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