Comenius University, Bratislava

Faculty of Mathematics, Physics and Informatics

Department of Applied Informatics

Action Model Learning: Challenges and Techniques

Dissertation Thesis

Michal Čertický

Bratislava, 2013

Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Applied Informatics

Action Model Learning: Challenges and Techniques

Dissertation Thesis

Author: RNDr. Michal Čertický
Supervisor: doc. PhDr. Ján Šefránek CSc.
Study Programme: 9.2.1. Computer Science

Bratislava 2013

15391687

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	RNDr. Michal Čertický
Study programme:	Computer Science (Single degree study, Ph.D. III. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Dissertation thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Action Model Learning: Challenges and Techniques

Aim: - To analyze the existing methods for automated action model learning, based on the set of properties relevant to their applicability in real-world conditions.
- Design new algorithms and analyze them.
- Conduct experiments to verify the properties of introduced algorithms.

Tutor:	doc. Ing. Ján Šefránek, CSc.
Department:	FMFI.KAI - Department of Applied Informatics
Vedúci katedry:	doc. PhDr. Ján Rybár, PhD.
Assigned:	19.10.2010

Approved: 19.10.2010 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

....

Student Tutor

15391687

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	RNDr. Michal Čertický
Študijný program:	informatika (Jednoodborové štúdium, doktorandské III. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	dizertačná
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Učenie modelu akcií: výzvy a techniky

Cieľ: - analyzovať existujúce metódy automatického učenia akcií. Kritériami hodnotenia budú vlastnosti, dôležité z hľadiska aplikovateľnosti metódy v reálnych podmienkach
- navrhnúť vlastné algoritmy a dokázať ich vlastnosti
- experimentálne overiť navrhnuté metódy.

Školiteľ:	doc. Ing. Ján Šefránek, CSc.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	doc. PhDr. Ján Rybár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 19.10.2010

Dátum schválenia: 19.10.2010 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....
študent školiteľ

Abstract

Knowledge about domain dynamics, describing how certain actions affect the world, is called an *action model* and constitutes the essential requirement for planning and goal-oriented intelligent behaviour. *Action learning*, as the automatic construction of action models based on the observations, has become a hot research topic in recent years, and various methods employing a wide variety of AI tools have been developed. Diversity of these methods renders each of them usable under different conditions and in various domains.

We have identified a common collection of important properties and used them as a basis for the comprehensive comparison and as a set of challenges that we were trying to overcome when designing our two new methods.

First of them, an *online* algorithm called $3SG$, can be used to learn the *probabilistic* action models with *conditional effects* in the presence of *incomplete observations*, *sensoric noise*, and *action failures*, while keeping the computational *complexity* low (polynomial w.r.t. the size of the input), thus managing to cope with all the appointed challenges. Our experiments, conducted in two fundamentally different domains (action computer game *Unreal Tournament 2004* and a real-world robotic platform *SyRoTek*), demonstrate its usability in computationally intensive scenarios, as well as in the presence of action failures and significant sensoric noise.

Our second method, based on a certain paradigm of logic programming called *Reactive ASP*, is not as versatile, being able to deal with only half of the challenges, but may still prove practical in some situations because of its purely declarative nature.

Keywords: action model, learning, partially observable, probabilistic, noise, action failures, conditional effects, answer set programming

Abstrakt

Znalosti o dynamike domény, popisujúce ako jednotlivé akcie ovplyvňujú svet, sa nazývajú *modelom akcií* a predstavujú nutnú podmienku pre plánovanie a cieľavedomé inteligentné správanie. *Učenie sa akcií*, teda automatická konštrukcia a spresňovanie modelov akcií na základe pozorovaní, sa v posledných rokoch stalo atraktívnou výskumnou témou a objavilo sa niekoľko zaujímavých metód, využívajúcich širokú paletu metód UI. Ich diverzita má za následok fakt, že každá z nich je použiteľná za rôznych podmienok, resp. v rozdielnych typoch domén. Identifikovali sme spoločnú množinu dôležitých vlastností týchto metód a použili ich ako základ pre ich dôkladnú analýzu a ako ciele pri tvorbe našich dvoch nových metód.

Prvá z nich, *online* algoritmus s názvom 3*SG*, produkuje *pravdepodobnostné* modely akcií s *podmienenými efektami*, a vyrovná sa so *zlyhaním akcií*, *senzorickým šumom*, a *neúplnými pozorovaniami*, pričom však udržiava svoju časovú zložitosť dostatočne nízku (polynomiálnu vzhľadom na veľkosť vstupu). Naše experimenty vykonané v dvoch fundamentálne odlišných doménach (akčnej počítačovej hre *Unreal Tournament 2004* a reálnej robotickej platforme *SyRoTek*), demonštrujú použiteľnosť tohoto algoritmu vo výpočtovo náročných prostrediach, ako aj za prítomnosti nezanedbateľného senzorického šumu a zlyhaní akcií.

Naša druhá metóda, založená na špecifickej paradigme logického programovania nazývanej *Reaktívne ASP*, nie je natoľko univerzálna, keďže spĺňa iba polovicu nami stanovených cieľov. Napriek tomu však môže byť vhodným kandidátom v istých situáciách, vďaka svojej deklaratívnej povahe.

Kľúčové slová: model akcií, učenie, plánovanie, čiastočná pozorovateľnosť, pravdepodobnosť, šum, zlyhania akcií, podmienené efekty, ASP

Contents

Chapter 1

Introduction

1.1 Background

Knowledge about domain dynamics, describing how certain actions affect the world, is essential for planning and intelligent goal-oriented behaviour of both living and artificial agents. Such knowledge is in artificial systems referred to as *action model*, and is usually manually constructed by domain experts.

Complete specification of action models in case of complex domains is, however, a difficult and time consuming task. In addition to that, it is often needed to modify this action model when confronted with new information. Research on various methods of *automatic construction of action models* has therefore become a hot topic in recent years. This inductive process of constructing and subsequent modification of action models is referred to as *action model learning* (sometimes abbreviated simply as "action learning"). Recent action learning methods take various approaches and employ the wide variety of AI tools.

Let us mention the heuristic *greedy search based* action learning introduced in [Zettlemoyer-Pasula-Kaelbling, 2003], *perceptron* training method found in [Mourao-Petrick-Steedman, 2010], learning through *inference over logic programs* described in [Balduccini, 2007], or two solutions based on the conversion of action learning into different classes of *satisfiability problems*, available in [Amir-Chang, 2008] and [Yang-Wu-Jiang, 2007]. In chapter 3 of this thesis, we provide detailed analysis of these modern methods and their properties.

1.2 Problems

The diversity of action learning methods renders each of them usable under different conditions and in different kinds of domains. Modern methods are most often evaluated and compared based on the following set of properties:

- Usability in *partially observable domains*,
- ability to learn *probabilistic* action models,
- dealing with *sensoric noise* and *action failures*,
- induction of action's *effects* and/or *preconditions*,
- learning *conditional effects*,
- *tractability* (often related to *online* vs. *offline* approach).

These properties, or rather *challenges*, can be used to determine *in which domains* each method can be used and *what exactly* it can learn.

Examination of the available solutions has shown that majority of them only possess a small subset of these desired properties (see the table in figure 3.1 for comprehensive comparison and the rest of the chapter 3 for detailed evaluation of individual methods).

Our goal was the **introduction** and **analysis** of novel **action learning methods** that would possess as many of the aforementioned properties as possible.

1.3 Proposed Solutions

In this work, we introduce **two new methods**. First of them is built upon the following two notions:

1. Compact structure for action model representation called the *effect formula* ($\mathcal{EF}$),
2. and the online algorithm called 3SG (*Simultaneous Specification, Simplification, and Generalization*) that produces and modifies an action model represented by $\mathcal{EF}$ in a polynomial time (w.r.t. the size of the input).

The $3SG$ algorithm is able to overcome all six discussed challenges: it is designed to learn the *probabilistic* action models with *preconditions* and *conditional effects* in *noisy*, *partially observable domains*, while keeping the *computational complexity* sufficiently low. Both the theoretical analysis and the empirical evidence gathered during our experiments demonstrate its usability in computationally intensive scenarios, where we can afford to trade the precision for speed.

Second method introduced in this thesis is based on a certain paradigm of logic programming, called *reactive ASP*[1]. Even though the idea to use logic programming for action learning is not new, in this method we use its newer (reactive) variant together with a simpler, more compact action model encoding.

This approach can overcome three out of six discussed challenges: it can be used in *partially observable*, and *non-deterministic* domains to induce both *preconditions* and *effects*. Despite having less appealing properties than $3SG$ algorithm, this method can still be a more favourable candidate under certain conditions, thanks to its purely declarative nature.

1.4 Outline

After first three sections of chapter 2, which provide an intuitive description of the background and motivation behind the problem of action learning, we will get to section 2.4, which explains the intuitions behind individual properties of action learning methods.

Chapter 3 then provides a comprehensive analysis of five recent methods, followed by their evaluation/comparison based on this set of properties.

Similarly, chapter 4 contains a comparison of these methods based on their compatibility with four different types of domains.

Along with establishing the formal framework and defining the necessary terminology in chapters 5 and 7, we intorduce the representation structures used by our own, as well as some of the alternative solutions (chapter 6).

[1] "ASP" stands for Answer Set Programming.

The 3SG algorithm will be described in detail in chapter 8, together with its theoretical analysis. Next chapter then presents the results of our experiments conducted in two fundamentally different domains (action computer game *Unreal Tournament 2004*, and a real-life *robotic platform SyRoTek*).

Following two chapters (10 and 11) introduce the second method based on the reactive variant of ASP, compare it extensively to the most similar alternative from chapter 3, and present the experimental results.

Chapter 12 concludes this thesis by briefly summarizing the most important findings and contributions.

Chapter 2

Preliminaries

In this chapter, we will explain the most basic notions related to the problem of action model learning on intuitive level. Important terminology will be defined more formally further in the text.

2.1 Planning: Why do we need a domain description?

Over past few decades, the problem of **planning** has become a significant part of artificial intelligence research. Among numerous approaches and methods that were developed and successfully used either in abstract, or real-world domains, we must first mention the pioneer work of Richard Fikes and Nils Nilsson, who established an extremely influential paradigm in early 70's [Fikes-Nilsson, 1971] by introducing a simple action representation formalism called STRIPS. The task of planning has since then been understood as finding a sequence of actions leading an agent from the initial world state, to a world state satisfying its goals.

Methods for solving the planning task are countless and make use of various techniques from other parts of AI and computer science. To mention just a few, we can start with graph-search based methods like Partial Order Planing [Sacredoti, 1975, Tate, 1977, Wilkins, 1984], GraphPlan [Blum-Furst, 1997, Blum-Langford, 2000, Kambhampati, 2000, Lopez-Bacchus, 2003], OBBD-Base Planning [Jensen, 1999, Jensen-Veloso, 2000, Jensen, 2003], logic-based methods like SATPlan
[Kautz-Selman, 1992, Baioletti-Marcugini-Milani, 1998, Rintanen, 2009] or Logic Programming [Subrahmanian-Zaniolo, 1995, Eiter et al., 2000], hierarchical network based planning like HTN [Kutluhan-Hendler-Nau, 1994], probabilistic planning with Markov Decision Processes (MDPs and POMDPs) [McMahan-Gordon, 2005, Theocharous-Kaelbling, 2003], or Genetic Planning [Farritor-Dubowsky, 2002]. Planning itself however, is not the central topic of this text and we will not explain these methods in detail. We merely recapitulate them in order to point out one thing they all have in common: they all need some kind of domain dynamics description - the **action model**.

2.2 Representation Languages: How to express our action model?

Action model is simply a description of dynamics within our planning domain, and can be understood as an expression of all the **actions** that can be executed in it, among with their **preconditions** and **effects** in some kind of representation language. Given an **initial** and **goal state**, the planning is simply an algorithmic process exploiting given action model expressed by some kind of representation language to find suitable plans.

Just like there are many planning algorithms, there is a large number of representation languages (often also called **action languages**) adopted by them. Many of such languages have their roots in STRIPS [Fikes-Nilsson, 1971] language, that we already mentioned, while enriching it by various features (note that STRIPS itself has been modified and revised since its first introduction). As an example of such languages, we can name PDDL (Planning Domain Definition Language) [McDermott et al., 1998, Fox-Long, 2003], providing features like existential and universal quantifiers, ADL (Action Description Language) [Pednault, 1989, Pednault, 1994], supporting the conditional effects, or our own language IK-STRIPS (STRIPS for Incomplete Knowledge) [Čertický, 2010a, Čertický, 2010b], designed for a non-monotonic representation of partial knowledge. There are also other widely-used languages, evidently deviating from the original STRIPS-like pattern. In their case, the action models are often represented using a notion of fluents (time and action-dependent features) and laws (expressing the causality within the domain). Languages like $\mathcal{K}$ [Eiter et al., 2000], $\mathcal{C}$ [Guinchiglia-Lifschitz, 1998] or C+ [Lee-Lifschitz, 2003] can serve as appropriate examples.

2.3 Action Learning: Why do we need to obtain action models automatically?

No matter what the representation language is, the action model must be known if an agent is supposed to plan his actions. Normally, it is hand-written by a programmer or domain expert. However, describing planning domains tends to be extraordinarily difficult and time-consuming task, especially in complex (possibly real-world) domains. This fact is the cause of our ambition to automatically generate the action models for our domain (no matter what kind [1]) based on agent's observations over time. This process is referred to as **action learning**.

2.4 Challenges and Properties

In this section, we will take a look at the set of **interesting properties** that any action learning method might or might not have, and try to explain the intuitions behind them (more formal definitions of the notions used here will be presented in chapter 7). These properties will then serve as the basis for the next chapter, where we will evaluate the collection of modern action learning approaches.

2.4.1 Usability in Partially Observable Domains

Every domain is either fully, or partially observable. As an example of a **fully observable domain** let us consider a game of chess. Both players (agents) have a full visibility of all the features of their domain - in this case the configuration of the pieces on the board. Such configuration is typically called a **world state**. On the other hand, by **partially observable domain** we understand any environment in which agents have limited observational capabilities - in other words, they can perceive only a small part of the state

[1]As a matter of fact, the automatic action learning is a necessary condition of an **environmental universality** as described in [Čertický, 2008] and [Čertický, 2009]. If an agent is supposed to be usable in various domains, it needs an ability to learn its dynamics from a sequence of observations.

of their environment (world states are partially observable). Real world is an excellent example of a partially observable domain. Agents of real world (for example humans) can only observe a small part of their surroundings: they can only hear sounds from their closest vicinity, see only objects that are in their direct line of sight, etc.

An action learning method is **usable in partially observable domains** only if it is capable of producing useful action models, even if world states are not fully observable.

2.4.2 Learning Probabilistic Action Models

There are two ways of modelling a domain dynamics (writing an action model) depending on whether we want the randomness to be present or not. An action model is **deterministic**, if actions it describes have all a unique set of always successful effects. Conversely, in case of a **probabilistic** (or stochastic) action models, action effects are represented by a probability distribution over the set of possible **outcomes**. Let us clarify this concept with the help of a simple "toy domain" called Blocks World (example 2.1), discussed extensively (among others) in [Nilsson, 1980], [Gupta-Nau, 1992], [Slaney-Thiebaux, 2001], or [Russel-Norvig, 2003][2].

Example 2.1
The Blocks World domain consists of a finite number of blocks stacked into towers (see figure 2.1) on a table large enough to hold them all. The positioning of towers on the table is irrelevant. Agents can manipulate this domain by moving the blocks from one position to another. Action model of the sim-

[2]Because of its simplicity, we will return to this Blocks World domain several times throughout the text, in order to explain various aspects of the action learning process.

plest Blocks World versions is composed of only one action $move(B, P_1, P_2)$. This action merely moves a block B from position P_1 to position P_2 (P_1 and P_2 being either another block, or the table).

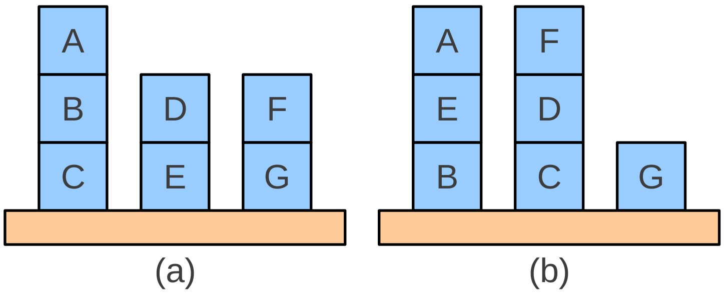


Figure 2.1: Two different world states of Blocks World.

Deterministic representation of such action would look something like this:

```
Action (
        Name & parameters:
                $move(B, P_1, P_2)$
        Preconditions:
                $\{on(B, P_1), free(P_1), free(P_2)\}$
        Effects:
                $\{\neg on(B, P_1), on(B, P_2)\}$
)
```

Our action is defined by its name, preconditions, and a unique set of effects $\{\neg on(B, P_1), on(B, P_2)\}$, all of which are applied each time the action is executed. This basically means that every time we perform an action $move(B, P_1, P_2)$, the block B will cease to be at position P_1 and will appear at P_2 instead. In a simple domain like Blocks World, this seems to be sufficient.

In the real world however, the situation is not so simple, and our attempt to move a block can have different outcomes[3]:

```
Action (
        Name & parameters:
                $move(B, P_1, P_2)$
        Preconditions:
                $\{on(B, P1), free(P_1), free(P_2)\}$
        Effects:
```

$$\begin{cases} 0.8: & \neg on(B, P_1), on(B, P_2) \\ 0.1: & \neg on(B, P_1), on(B, table) \\ 0.1: & nochange \end{cases}$$

)

This representation of our action defines the following probability distribution over three possible outcomes:

1. probability of 0.8 that block B indeed appears at P_2 instead of P_1,
2. probability of 0.1 that block B falls down on the table,
3. probability of 0.1 that we fail to pick it up and nothing happens.

We can easily see that the probabilistic action models are better suited for real-world domains, or complex simulations of non-deterministic nature, where agent's sensors and effectors are often imprecise and actions sometimes lead to unpredicted outcomes.

[3]Similarly to the variation of Blocks World featuring realistic physics in a three-dimensional simulation presented in [Pasula-Zettlemoyer-Kaelbling, 2007]. We will mention this paper further in this text, since it features an interesting method for probabilistic action learning.

2.4.3 Dealing with Action Failures and Sensoric Noise

In some cases we prefer learning the deterministic action models in stochastic domains. (Recall that action models are used for planning. Planning with probabilistic models is computationally harder [Littman et al. 1998, Domshlak-Hoffman, 2007], which makes such models unusable in some situations.) Therefore we need an alternative way of dealing with nondeterministic nature of our domain. There are two sources of problems that can arise in this setting:

Action Failures

As we noted in 2.4.2, actions in non-deterministic domains can have more than one outcome. In a typical situation though, each action has one outcome with significantly higher probability than the others. In case of action $move(B, P_1, P_2)$ from Blocks World, this **expected outcome** was actually moving a block B from position P_1 to P_2. Then, if after the execution the block was truly at position P_2, we considered the action successful. If the action had any other outcome, it was considered unsuccessful - we say that the action **failed**.

From the agent's point of view, action failures pose a serious problem, since it is difficult for him to decide whether given action really failed (due to some external influence), or the action was successfull, but his expectations about the effects were wrong (if his expectations were wrong, he needs to modify his action model accordingly).

Sensoric Noise

Another source of complications is so-called **sensoric noise**. In real-world domains, we are typically dealing with sensors that have limited precision. This means that the observations we get do not necessarily correspond to the actual state of the world.

Even when agent's action is successful, and the expected changes occur, he may observe the opposite. From the agent's point of view, this problem is similar to the problem with action failures. In this case he needs to solve the dilemma, whether his expectations were incorrect, or the observation was imprecise.

In addition to that, sensoric noise can cause one more complication of a technical nature: If the precision of the observations is not guaranteed, even a single observation can be internally inconsistent (some of the sensors returning mutually conflicting information). Logic-based action learning methods sometimes fail to deal with this fact.

2.4.4 Learning both Preconditions and Effects

Since the introduction of STRIPS [Fikes-Nilsson, 1971], a common assumption is that actions have some sort of **preconditions** and **effects**.

Preconditions[4] define what must be established in a given world state before an action can even be executed. Looking back at Blocks World (example 2.1), the preconditions of action $move(B, P_1, P_2)$ require both positions P_1 and P_2 to be free (meaning that no other block is currently on top of them).

[4]Preconditions are sometimes called *applicability conditions* - especially when we formalise actions as operators over the set of world states.

Otherwise, this action is considered inexecutable.

Effects[5] simply specify what is established after a given action is executed, or in other words, how the action modifies the world state.

Some action learning approaches either produce effects and ignore preconditions, or the other way around. They are therefore incapable of producing complete action model from the scratch, and thus are usable only in situations when some partial hand-written action model is provided. We want to avoid the necessity to have any prior action model.

2.4.5 Learning Conditional Effects

Research in the field of action description languages has shown that expressive power of early STRIPS-like representations is susceptible to be improved by addition of so-called conditional effects. This results from the fact that actions, as we usually talk about them in natural language, have different effects in different world states.

Consider a simple action of person P drinking a glass of beverage B - $drink(P, B)$. Effects of such action would be (in natural language) expressed by following sentences:

- P will cease to be thirsty.
- If B was poisonous, P will be sick.

We can see that second effect (P becoming sick) only applies *under certain conditions* (only if B was poisonous). We call an effect like this a **conditional**

[5]Effects are sometimes called *postconditions* - especially in the early publications in STRIPS-related context.

effect.

STRIPS language for instance did not support conditional effects. Of course, there was a way to express aforementioned example, but we needed two separate actions with different sets of preconditions for it:
$drink\_if\_poisonous(P, B)$ and $drink\_if\_not\_poisonous(P, B)$.

Having a support for conditional effects thus allows us to specify domain dynamics by lower number of actions, making our representation less space consuming and more elegant. Several state-of-the-art action languages provide the apparatus for defining conditional effects - see the following example:

Example 2.2
STRIPS extensions like Action Description Language (ADL) [Pednault, 1989, Pednault, 1994] or Planning Domain Definition Language (PDDL) [McDermott et al., 1998, Fox-Long, 2003] express the effects of $drink(P, B)$ action in the following LISP-resembling syntax:

```
:effect     (not (thirsty ?p))
:effect     (when (poisonous ?b) (sick ?p))
```

Definition of the same two effects in some of the fluent-based languages like $\mathcal{K}$ [Eiter et al., 2000], on the other hand, employs the notion of so-called *dynamic laws*:

```
caused -thirsty(P) after drink(P,B).
caused sick(P) after poisonous(B), drink(P,B).
```

Aside from creating more elegant and brief action models, the ability to learn conditional effects provides one important advantage: It allows for more convenient input structure from our sensors. If we were unable to work with conditional effects, our sensors would have to be able to observe and interpret a large number of actions like $drink\_if\_poisonous(P, B)$ or $drink\_if\_not\_poisonous(P, B)$. However, if our action model supports conditional effects, the sensors only need to work with a smaller number of more general actions like $drink(P, B)$.

2.4.6 Online Algorithms and Tractability

Algorithms that run fast enough for their output to be useful are called **tractable** [Hopcroft-Motwani-Ullman, 2007]. Alternative, more strict, definition requires tractable algorithms to run in polynomial time [Hromkovič, 2010].

Algorithms whose input is served one piece at a time, and upon receiving it, they have to take an irreversible action without the knowledge of future inputs, are called **online** [Borodin-El-Yaniv, 1998].

For the purposes of action learning we prefer using online algorithms, which run once at each time step - after the observation. At any point in time, only this newest observation is served as the input for the algorithm, while older observations are not processed. Algorithm uses this observation to irreversibly modify agent's action model. Since the input of such algorithm is relatively small, tractability is usually not an issue here.

If we, on the other hand, decided to use an offline algorithm for action learning, we would have to provide the whole history of observations on the input. Algorithms operating over such large data sets are prone to be intractable.

Since online algorithms are designed to run repeatedly during the "life" of an agent, he has some (increasingly accurate) knowledge at his disposal at all times.

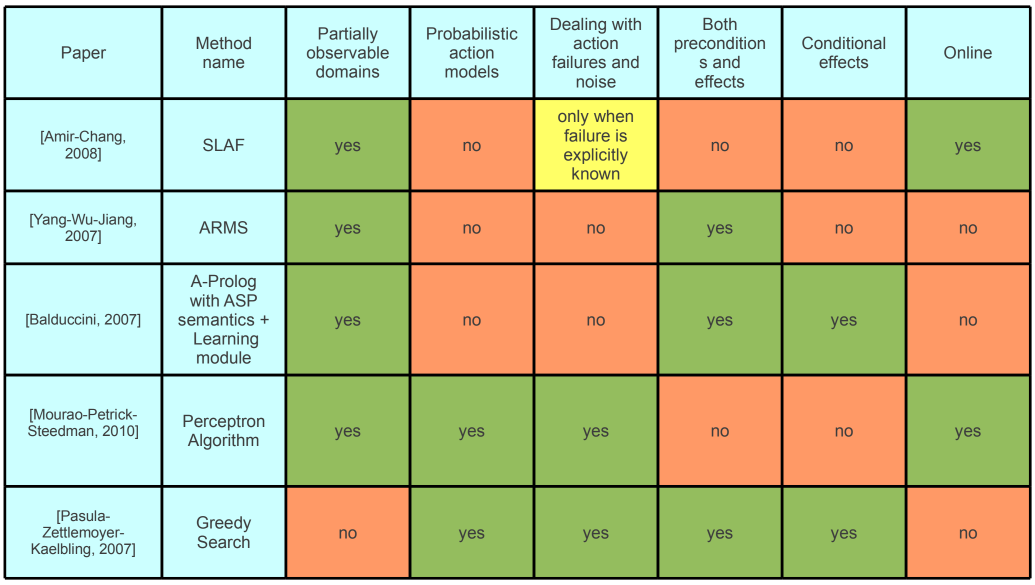
There is, however, a downside to using online algorithms for action learning. Recall that with online algorithms the complete history of observations is not at our disposal, and we make an irreversible change to our action model after each observation. This change can cause our model to become inconsistent with some of the previous observations. It also means that the precision of induced action models depends on the ordering of observations. Online algorithms are therefore potentially less precise than their offline counterparts. Lower precision is, however, often traded for tractability.

Chapter 3

Evaluation based on Properties

In this chapter we will describe five methods that are currently used to solve the task of action learning. Each of them represents a different approach and uses a different set of tools and/or representation formalisms to express and generate new action models or improve the existing ones.

Conclusion of every section contains the **evaluation** of corresponding method based on the set of **properties established in the previous chapter**. For a comprehensive comparison of these methods, see figure 3.1.



Paper	Method name	Partially observable domains	Probabilistic action models	Dealing with action failures and noise	Both precondition s and effects	Conditional effects	Online
[Amir-Chang, 2008]	SLAF	yes	no	only when failure is explicitly known	no	no	yes
[Yang-Wu-Jiang, 2007]	ARMS	yes	no	no	yes	no	no
[Balduccini, 2007]	A-Prolog with ASP semantics + Learning module	yes	no	no	yes	yes	no
[Mourao-Petrick-Steedman, 2010]	Perceptron Algorithm	yes	yes	yes	no	no	yes
[Pasula-Zettlemoyer-Kaelbling, 2007]	Greedy Search	no	yes	yes	yes	yes	no

Figure 3.1: Comparison of current action learning methods based on the properties from chapter 2.

3.1 SLAF - Simultaneous Learning and Filtering

First method we will take a look at is called Simultaneous Learning and Filtering (SLAF) and was published in a paper called *Learning partially observable deterministic action models* [Amir-Chang, 2008] in 2008.

This method produces the action models in two phases:

1. Building a propositional formula φ over time by calling the SLAF algorithm once after each observed action (at each time step t).
2. Interpreting this formula φ by using a satisfiability (SAT) solver algorithms (e.g., [Moskewitz et al., 2001]).

3.1.1 Phase 1: SLAF Algorithm

Representation language:

Let A be a set of all the actions possible in our domain. Let P be a set of all the domain's fluent literals (domain features that can change over time, typically by executing actions[1]). Knowledge about the action model and world states is then compactly encoded by logic formulas over a vocabulary $L = P \cup L_f^0$, where $L_f^0 = \bigcup_{a \in A}\{a^f, a^{f\circ}, a^{\neg f}, a^{[f]}, a^{[\neg f]}\}$ for every $f \in P$. The intuition behind propositions in the set L_f^0 is as follows:

a^f : *"a **causes** f"*, meaning that the execution of a causes literal f to be true.

[1]We will provide a formal definition of the fluent literals further in this thesis. For now, this intuition should suffice.

$a^{f\circ}$: *"a* ***keeps*** *f"*, meaning that the execution of a does not affect the value of fluent f.

$a^{[f]}$: *"a* ***causes*** *FALSE* ***if*** $\neg f$*"*, meaning that a literal f is a precondition of a, and it must hold before the execution of a.

Also, for a set of propositions P, let P' represent the same set of propositions, but with every proposition primed (i.e. each proposition f is annotated to become f'). Authors use such primed fluents to denote the value of unprimed fluents one step into the future after taking an action. Furthermore, let $\varphi_{[P'/P]}$ denote the same formula as φ, but with all primed fluents replaced by their unprimed counterparts. For example formula $(a \vee b')_{[P'/P]}$ is equal to $(a \vee b)$ when $b \in P$.

SLAF algorithm is called at each time step t with the following input and output:

Input:

- Most recently executed action a $(a \in A)$.
- Partial observation σ from current time step $(\sigma \subseteq P)$.
- Output of the previous call of SLAF algorithm in a form of propositional formula φ.

Output:
Propositional logical formula φ over the vocabulary L. φ represents all the combinations of action models that could possibly have given rise to the observations in the input and all the corresponding states in which the world may now be (after the sequence of time steps that were given in the input occurs). This formula is called a *transition belief formula.*

The output of SLAF algorithm, after the execution of action a, is defined in the recursive fashion:

Definition 1 (SLAF Output)**.**

$$SLAF[a](\varphi) \equiv Cn^{L \cup P'}(\varphi \wedge \tau_{eff}(a))_{[P'/P]}$$

where $\tau_{eff}(a)$ is a propositional axiomatization of action a and $Cn^{L \cup P'}$ is appropriate *consequence finding operator* over a vocabulary $L \cup P'$.

Axiomatization $\tau_{eff}(a)$ is a propositional expression of consequences of the fact that action a was executed:

$$\begin{aligned} \tau_{eff}(a) &\equiv \bigwedge_{f \in P} Pre_{a,f} \wedge Eff_{a,f} \\ Pre_{a,f} &\equiv \bigwedge_{l \in \{f, \neg f\}} (a^{[l]} \Rightarrow l) \\ Eff_{a,f} &\equiv \bigwedge_{l \in \{f, \neg f\}} ((a^l \vee (a^{f\circ} \wedge l)) \Rightarrow l') \wedge (l' \Rightarrow (a^l \vee (a^{f\circ} \wedge l))). \end{aligned}$$

Here $Pre_{a,f}$ states that if l is a precondition of action a, then it must have held in the world state before executing a. $Eff_{a,f}$ then states that fluents before and after execution of a must be consistent with the action models defined by propositions a^f, $a^{\neg f}$, and $a^{f\circ}$.

As an efficient consequence finding operator $Cn^{L \cup P'}(\varphi)$, it is possible to use simple *propositional resolution* [Davis-Putnam, 1960, Chang-Lee, 1973]. However, in their implementation, authors rather used the *prime implicates finder* [Marquis, 2000], which is under certain conditions both tractable, and keeps formula φ reasonably short.

3.1.2 Phase 2: Interpreting Results

When we interpret the transition belief formula, we assume the existence of three logical axioms, that disallow the *inconsistent* and *impossible* models:

1. $a^f \vee a^{\neg f} \vee a^{f\circ}$
2. $\neg(a^f \wedge a^{\neg f}) \wedge \neg(a^{\neg f} \wedge a^{f\circ}) \wedge \neg(a^f \wedge a^{f\circ})$
3. $\neg(a^{[f]} \wedge a^{[\neg f]})$

for all possible $a \in A$ and $f \in P$. First two axioms mean that action a either **causes** f, **causes** $\neg f$, or **keeps** f. Third axiom says that action cannot have both f and $\neg f$ as its preconditions.

To find out if an action model M is possible, we can use any of numerous SAT solvers. If a conjunction $\varphi \cup M \cup AXIOMS$ is satisfiable, then M is possible. Otherwise, it is impossible.

3.1.3 Evaluation

Amir and Chang's method is designed only for **deterministic domains**, where our observations are always true, and **action failures do not occur**[2]. Transition belief formula is then *exact* in the sense that it represents every possible and no impossible *action model - world state* combinations (given previous observations). In fact, the number of such *model - state* combinations is decreasing over time when new observations are added, making our

[2]Authors suggested a solution for situations with possible action failures, but it depended on the unrealistic assumption that the agent always knew, whether the given action was successful, even when he still did not know what the action was supposed to do.

knowledge more precise. However, there is **no probability distribution** over these possible *model* - *state* combinations.

Authors have shown that their solution is **tractable**, since SLAF algorithm called once in each time step runs in polynomial time when implemented correctly (for actual pseudo-code of SLAF algorithm see [Amir-Chang, 2008]). However, answering certain queries about learned action models is more time-consuming task, since deciding the satisfiability of a propositional formula is an NP-complete problem[3]. Current SAT solvers, however, deal with this problem quite efficiently.

3.2 ARMS - Action-Relation Modelling System

This next method was proposed in a paper called *Learning action models from plan examples using weighted MAX-SAT* [Yang-Wu-Jiang, 2007]. It is closely related to the SLAF method by Amir et al. (described in section 3.1) due to similar objectives (learning the action models) and usage of satisfiability solvers. However, the input needed for this technique is different. Amir et al.'s method needs each observed action to be followed by a **state observation**. In the absence of these observations, SLAF algorithm would not work. ARMS method, on the other hand, takes only a set of so-called **plan examples** as an input, and does not need any state observations between individual actions. Also, resulting action models are *approximations*, in contrast to SLAF, which aimed to find the *exact solutions*.

[3]Cook-Levin theorem [Cook, 1971].

The ARMS method runs a single recursive algorithm consisting of the following six steps:

1. Convert all the observed action instances to their **schema forms** by replacing the constants by corresponding variables (actual representation language used here is PDDL - it will be described in the following subsection).
2. For all the unexplained actions, build the set of logical clauses of three types called **action**, **information**, and **plan constraints**. Also use a *frequent-set mining algorithm* [Agrawal, 1994] to find pairs of actions (a, b), such that a frequently coexists with b in our observed plans.
3. Assign weights to all the clauses (constraints) generated in the previous step.
4. Solve the weighted MAX-SAT problem over this set of clauses with assigned weights using external MAX-SAT solver. During their experiments, the authors used MaxWalkSat solver [Kautz-Selman-Jiang, 1997, Selman-Kautz-Cohen, 1993].
5. Update our current knowledge with the set of action models with highest weights. If we still have some unexplained actions, go to step 2.
6. Otherwise terminate the algorithm.

Let us now take a closer look at the ARMS algorithm, its input, output and individual steps.

3.2.1 ARMS Algorithm

Input:

Plan example is a triple consisting of:

1. **initial state**,
2. **goal state**, both described by a list of propositions,
3. and a **sequence of actions**, represented by an *action name* and *instantiated parameters.*

Each plan on the input of ARMS algorithm must be *successful* in that it achieves the goal from the initial state. The plan examples may only contain **action instances** - expressions like $move(block01, pos01, pos02)$, where $block0$, $pos01$ and $pos01$ are constants. These plan examples do not include action's preconditions and effects. These are to be learned by the algorithm.

Representation Language and Output:

Resulting action model is expressed in a Planning Domain Definition Language - PDDL [McDermott et al., 1998, Fox-Long, 2003]. This action language provides us, among other useful features, with type definitions for variables/constants. To describe the PDDL language, we will once again use the $move(B, P_1, P_2)$ action of Blocks World (example 2.1):

```
(:action move
  :parameters (?b - block ?p1 ?p2 - position)
  :precondition (and (on ?b ?p1) (free ?p1) (free ?p2))
  :effect (and (not (on ?b ?p1) (on ?b ?p2))
)
```

Symbols **?b**, **?p1**, and **?p2** (or any lowercase letter preceded by a question mark) are used to denote variables. In the **:parameters** section of this action expression, we have the **type definitions**. Expression "?b - block" simply says that the variable **?b** can only be substituted by a constant c, if c is a block - that is, if a "(block c)" clause holds in our domain.

Converting Actions to their Schema Forms:
A plan examples consist of a sequences of *action instances* (with constant parameters). ARMS algorithm *converts* all such plans by substituting all the occurrences of an instantiated object in every action instance with the variable of the same type. That, of course, means that the algorithm needs to have access to domain specific knowledge about the types of individual constants.

Building Constraints:
Each iteration of ARMS algorithm generates the set of logical clauses (called *constraints* by the authors[4]), which later serve as an input for weighted MAX-SAT solver. There are three types of such clauses:

1. **Action Constraints** are imposed on individual actions. They are derived from the *general axioms of correct action representations.* Let pre_i, add_i, and del_i be preconditions, add list, and delete list of action a_i respectively. Generated *action constraints* are then following two clauses:

   (A.1) $pre_i \cap add_i = \emptyset$

   The intersection of precondition and add list of all actions must be empty.

[4]Note that we will use the "constraint" term with a different meaning furhter in the text.

(A.2) $del_i \subseteq pre_i$

In addition to that, for every action we require that the delete list is a subset of the precondition list.

2. **Information Constraints** are used to explain *why* the (optionally) *observed intermediate state exists in a plan.* Suppose we observe p to be true between actions a_n and a_{n+1}. Also let $\{a_{i_1} \dots a_{i_k}\}$ be the set of actions from our current plan (appearing in the plan in that order), which share the parameter type with p. Then:

(I.1) $p \in add_{i_1} \vee p \in add_{i_2} \vee \cdots \vee p \in add_{i_k}$

p must be generated by an action $a_{i_k} (a \leq i + k \leq n)$, which means that it is in the add list of a_{i_k}.

(I.2) $p \notin del_{i_k}$

Last action a_{i_k} must not delete p, which means that it must not occur in delete list of a_{i_k}.

3. **Plan Constraints** represent the relationship between actions in a plan that *ensures that the plan is correct* and that the *action's add list clauses are not redundant*:

(P.1) Every precondition p of every action b must be in the add list of earlier action a and is not deleted by any actions between a and b.

(P.2) At least one clause r in the add list of action a must be useful for achieving a precondition of some later action.

While constraints P.1 and P.2 provide the general guide principle for ensuring the correctness of a plan, in practice there are *too many instantiations* of these constraints. To ensure the efficiency, authors suggest to generate them not for every possible action pair, but only for pairs of actions that frequently coexist in the input plan examples. They employ the *frequent-set mining algorithm* [Agrawal, 1994] to find such pairs.

Assigning Weights

To solve a weighted MAX-SAT problem in Step 3 of ARMS algorithm, each constraint clause must be associated with a weight value between zero and one. The higher the weight, the higher the priority in satisfying the clause. To assign the weights, authors apply the following heuristics:

1. **Action Constraints** receive an empirically determined constant weight $W_A(a)$ for an action a.

2. **Information Constraints** also receive empirically determined constant weight $W_I(r)$ for a clause r. This assignment is generally the highest among all the constraint's weights.

3. **Plan Constraints** take the weight value $W_P(a, b)$ equal to prior probability of the occurrence of action pair (a, b) in plan examples (while taking into account predetermined *probability threshold* or *minimal support-rate*).

3.2.2 Evaluation

The method of Yang et al. is unique among current action learning approaches in a fact that it does not require the world state observations between individual actions (while still being able to use them if they are present), which can prove particularly useful in some situations. As its input however, it requires successful plan examples, which not only means that it **cannot deal with action failures**, but also that we need to have extra information about the initial and goal states.

The fact that resulting action models are represented by PDDL language with type definitions means that subsequent planning may potentially be faster. The downside of this is that domain-specific background knowledge (about the types of individual constants) is needed for successful learning. **Conditional effects** are **not produced** by ARMS algorithm, even though such construct is allowed in PDDL. As for **probabilistic representation** of actions, they are **not supported** by the basic versions of PDDL at all.

In conclusion, ARMS method is designed for learning **deterministic** action models (with both **preconditions** and **effects**) from the successful plan examples instead of *action-state* observations. Learning process is in its essence highly heuristic, which is necessary, since it's based on the NP-hard weighted MAX-SAT problem.

3.3 Learning through Logic Programming and A-Prolog

In a paper called *Learning Action Descriptions with A-Prolog: Action Language $\mathcal{C}$* [Balduccini, 2007], Marcello Balduccini demonstrated how action models can be learned in a purely declarative manner, using non-monotonic logic programming with stable model semantics.

The prior goal of his paper was to show that a representation language A-Prolog [Gelfond-Lifschitz, 1988, Gelfond-Lifschitz, 1991, Baral, 2003] is a powerful formalism for knowledge representation and reasoning, usable (besides planning and diagnosis) also for learning tasks. Action learning here is based solely on the semantics of A-Prolog, and includes both *addition* of new, and *modification* of existing knowledge about domain dynamics.

Representation language chosen for the expression of actual action models is, however, not A-Prolog itself, but $\mathcal{C}$ [Guinchiglia-Lifschitz, 1998] ($\mathcal{C}$ was originally designed as an action language).

Balduccini's method can be viewed as a simple three-step process:

1. **Translation** of current knowledge from $\mathcal{C}$ to a set of **A-Prolog facts**.
2. Using a so-called **learning module** (an A-Prolog logic program) in conjunction with the observation **history** to compute the set of stable models by an external ASP solver[5].
3. Translation of resulting stable models back to action language $\mathcal{C}$.

3.3.1 Translation to A-Prolog

Overview of $\mathcal{C}$

$\mathcal{C}$ is one of the languages that quite strongly deviate from the typical STRIPS-like pattern - it does not encapsulate individual actions. Instead, it conveniently describes the whole domain dynamics by a set of **laws**. In general, there are two types of such laws:

[5]ASP (or Answer Set Programming) solvers (for example DLV [Leone et al., 2006] or Smodels [Niemela-Simons-Syrjanen, 2000]) take a logic program as an input and return the set of *stable models* (also called *answer sets*) on the output. Stable models represent all the possible meanings of the input logic program under the *stable model semantics*.

1. **Static laws** of the form: **caused** r **if** $l_1, \ldots, l_n$

2. **Dynamic laws** of the form: **caused** r **if** $l_1, \ldots, l_n$ **after** $p_1, \ldots, p_m$

Here r (the *head* of the law) is a literal or $\perp$, $l_1, \ldots, l_n$ (the *if-preconditions*) are all literals, and $p_1, \ldots, p_m$ (the *after-preconditions*) are either literals, or actions. More detailed definition of a $\mathcal{C}$ language can be found (with slightly different terminology) in [Lifschitz-Turner, 1999].

To illustrate how this language is used in practice, take one more look at our $move(B, P_1, P_2)$ action of Blocks World (example 2.1), this time encoded in $\mathcal{C}$:

```
caused $on(B, P_2)$ if $\emptyset$ after $move(B, P_1, P_2), on(B, P_1), free(P_1), free(P_2)$
caused $\neg on(B, P_1)$ if $\emptyset$ after $move(B, P_1, P_2), on(B, P_1), free(P_1), free(P_2)$
```

Note that "if $\emptyset$" part can be omitted in this simple example, since the set of *if-preconditions* is empty.

Overview of A-Prolog

A-Prolog is a knowledge representation language that allows formalisation of various kinds of common sense knowledge and reasoning. The language is a product of research aimed at defining a formal semantics for logic programs with default negation [Gelfond-Lifschitz, 1988], and was later extended to allow also a classical (or *explicit*) negation [Gelfond-Lifschitz, 1991].

A **logic program** in A-Prolog is a set of **rules** of the following form:

$$h \leftarrow l_1, \ldots, l_m,\ not\ l_{m+1}, \ldots\ not\ l_n.$$

where h and $l_1, \ldots, l_n$ are classical literals and *not* denotes a default negation. Informally, such rule means that "if you believe $l_1, \ldots, l_m$, and have no reason to believe any of $l_{m+1}, \ldots, l_n$, then you must believe h". The part to the right of the "$\leftarrow$" symbol ($l_1, \ldots, l_m,$ *not* $l_{m+1}, \ldots$ *not* l_n) is called a *body*, while the part to the left of it (h) is called a *head* of the rule. Rules with an empty body are called *facts*. Rules with empty head are called *constraints*.

The semantics of A-Prolog is built on the concept of **stable models**. Consider a logic program Π and a consistent set of classical literals S. We can then get a subprogram called *program reduct of* Π *w.r.t. set* S (denoted Π^S) by removing each rule that contains *not* l, such that $l \in S$, in its body, and by removing every "*not* l" statement, such that $l \notin S$. We say that a set of classical literals S is *closed* under a rule a, if holds $body(a) \subseteq S \Rightarrow head(a) \in S$.

Definition 2 (Stable Model)**.** Let Π be a logic program. Any minimal set S of literals *closed* under all the rules of Π^S is called a *stable model* of Π.

Intuitively, a stable model represents one possible meaning of knowledge encoded by logic program Π and a set of classical literals S.

Translation

Even though it is possible to translate each $\mathcal{C}$ law into *one* A-Prolog rule [Lifschitz-Turner, 1999], Balduccini rather decided to translate it **into a set of facts**.

Let us now take a $\mathcal{C}$ law and denote it $w(v_1, \ldots, v_k)$, where $v_1, \ldots, v_k$ is a list of variables appearing in it (in previous example it would be B, P_1, P_2) and prefix w is just any name assigned to it. In Balduccini's paper, he calls this prefix $w^{\mathcal{P}}$, and its variable list $w^{\mathcal{V}}$.

Static law w is now encoded into following set of A-Prolog facts:

$s\_law(w^{\mathcal{P}}).$

$head(w^{\mathcal{P}}, r^w).$

$vlist(w^{\mathcal{P}}, \lambda(w^{\mathcal{V}}))$.

$if(w^{\mathcal{P}}, \langle l_1^w, \ldots, l_n^w \rangle)$.

First two facts say that the symbol $w^{\mathcal{P}}$ is reserved for a *static law* and that a head of this law is literal r^w. Third fact describes how variables of this law are *groundified* (subsequently substituted by constants[6]). Fourth fact enumerates all the literals representing the *if-conditions* of this law.

Semantic meaning of any static $\mathcal{C}$ law is preserved by addition of the following A-Prolog rule (uppercase letters are used here to denote variables):

$$\begin{aligned} h(H, T) \;\leftarrow\; & s\_law(W), vlist(W, VL), \\ & head(W, H_g), gr(H, VL, H_g), \\ & all\_if\_h(W, VL, T). \end{aligned}$$

Here $gr(H, VL, H_g)$ intuitively means that H can be groundified to H_g using the variable mapping from VL, and $all\_if\_h(W, VL, T)$ means that all the if-preconditions of a law W hold in time step T w.r.t. the same variable mapping. Intuitive meaning of literal $h(H, T)$ is that head H holds in the step T.

Dynamic law w is translated into A-Prolog in a similar manner:

$d\_law(w^{\mathcal{P}})$.

$head(w^{\mathcal{P}}, r^w)$.

$vlist(w^{\mathcal{P}}, \lambda(w^{\mathcal{V}}))$.

$if(w^{\mathcal{P}}, \langle l_1^w, \ldots, l_n^w \rangle)$.

[6]In A-Prolog, we call rules that do not contain variables *ground*. Note that non-ground rules are semantically equivalent to the set of their ground instances.

$$after(w^{\mathcal{P}}, \langle p_1^w, \ldots, p_m^w \rangle).$$

First fact means that the symbol $w^{\mathcal{P}}$ is reserved for a *dynamic law.* Second, third and fourth law are all identical to previous case. Last fact is added to enumerate all the *after-preconditions* of law w.

Semantic meaning of dynamic laws is expressed by the following rule:

$$\begin{aligned} h(H, T+1) \ \leftarrow \ & d\_law(W), vlist(W, VL), \\ & head(W, H_g), gr(H, VL, H_g), \\ & all\_if\_h(W, VL, T), all\_after\_h(W, VL, T). \end{aligned}$$

Meaning of this rule is similar to previous simpler case, except it describes the causal dependence between two time steps T and $T+1$.

Note 3.3.1. In addition to the ones that we mentioned, there are some more A-Prolog rules defining the complete semantics of language $\mathcal{C}$. In order to save space, we must omit them. You can find them in the original paper [Balduccini, 2007].

In this manner, we are able to translate all the static and dynamic $\mathcal{C}$ laws into A-Prolog. Additionally, **observations** are encoded by statements of the form $obs(l, t)$ or $hpd(a, t)$, meaning that a literal l was observed, or an action a happened at time step t (respectively). The collection of such literals over a time period is called **history**.

Deciding if the Modification is Needed

Next step of Balduccini's method is the detection of the need for learning. When given a history, an agent should check if it is consistent with its current action description. If there is at least one stable model, they are consistent and we say that a history can be explained by our description. Otherwise we need to rebuild our action description by **appending a learning module** to the semantic definitions and history, and **re-calculating the stable models**.

3.3.2 Learning Module

Learning module is simply the following set of fifteen rules:

1. $\{if(W, N, L_g)\}$
2. $\leftarrow if(W, N, L_{g_1}), if(W, N, L_{g_2}), L_{g_1} \neq L_{g_2}.$
3. $\leftarrow has\_if(W, N), N > 1, not\ has\_if(W, N-1).$
4. $\leftarrow if(W, N, L_g), not\ valid\_gr(W, N, L_g).$
5. $\{d\_law(W), s\_law(W)\} \leftarrow available(W).$
6. $\leftarrow d\_law(W), s\_law(W).$
7. $\{head(W, H_g)\} \leftarrow newly\_defined(W).$
8. $\leftarrow newly\_defined(W), not\ has\_head(W).$
9. $\leftarrow head(W, H_{g_1}), head(W, H_{g_2}).$
10. $\leftarrow head(W, H_g), not\ valid\_gr(W, N, H_g).$
11. $\{after(W, N, A_g)\} \leftarrow d\_law(W).$
12. $\leftarrow after(W, N, A_{g_1}), after(W, N, A_{g_2}), A_{g_1} \neq A_{g_2}.$
13. $\leftarrow has\_after(W, N), N > 1, not\ has\_after(W, N-1).$
14. $\leftarrow after(W, N, A_g), not\ valid\_gr(W, N, A_g).$

15. $vlist(W, VL) \leftarrow newly\_defined(W), avail\_list(W, VL).$

Space limitations prevent us from explaining all of the rules here, but we can clearly see how new laws are generated when we take a look at rules (5) and (6). Rule (5) intuitively says that any available constant can be used as prefix of a new dynamic or static law, while rule (6) says that it cannot be used for both. For the full explanation of the learning module, see the original paper.

Individual rules of this learning module, together with definitions of $\mathcal{C}$ language semantics, cause ASP solver to produce stable models representing all the reasonable models of domain dynamics. Models that are, in addition to that, consistent with observation history, represent our valid action models. Naturally, when the history grows over time, the number of such models is decreasing, thus making our knowledge more precise.

3.3.3 Evaluation

The main advantage of Balduccini's method is its declarative character, meaning that basically nothing new needs to be implemented in order to use it. The problem is that proposed A-Prolog encoding of knowledge is too robust, and using ASP solvers to compute stable models for non-monotonic logic programs of this length is **intractable**. In addition to that, the **whole observation history** needs to be checked at each time step, making the computational time grow exponentially.

Interestingly for us, the non-monotonic character of A-Prolog allows the method to be used with **partial observations**. The action language $\mathcal{C}$, chosen to represent the output, implicitly allows and encourages the use and generation of **conditional effects**.

On the other hand, **deterministic character** of this representation **disallows** the production of **probabilistic action models** and makes dealing with **action failures impossible**.

In conclusion, we need to say that this method was never intended to be used in complex practical applications. It was rather designed to analyse and demonstrate certain properties of A-Prolog formalism. However, like many other AI techniques that bring into play ASP solvers, it is still usable in domains that are simple enough and computation process does not need to be quick.

3.4 Kernelised Voted Perceptrons with Deictic Coding

This next method, described by Mourao, Petrick and Steedman in a paper called simply *Learning action effects in partially observable domains*[7] [Mourao-Petrick-Steedman, 2010], is based on a well known **Perceptron Algorithm** introduced by F. Rosenblatt in 1958 [Rosenblatt, 1958].

It aims at learning only the effects of STRIPS-like actions (not their preconditions) and treats this problem as a **set of binary classification problems** which are solved using the **kernelised voted perceptron** [Freund-Schapire, 1999, Aizerman-Braverman-Rozoner, 1964].

Since all the effects here are in the form of conjuntions of literals, it is suf-

[7]This paper extends the earlier method proposed in [Mourao-Petrick-Steedman, 2008] by providing a better support for incomplete observations and noisy domains.

ficient to learn the rule for each effect fluent separately (one binary classification problem per fluent). Authors have chosen to use the **perceptron** [Rosenblatt, 1958] to address this particular learning problem, since it is a simple, yet fast binary classifier.

Specifically, having a strictly given bank of kernel perceptrons P, where $\forall p \in P : p$ corresponds to one possible effect fluent, we perform the following routine at each time step:

1. Choosing a relevant subset of fluents using the **deictic coding** approach.
2. Representing currently observed world state and the executed action as a **vector over** $\{-1, 0, 1\}$.
3. Using this vector as an input for each $p \in P$ and **adjust the internal weight** w of p accordingly (training).

This way, after a sufficient number of observations, each $p \in P$ can be used to predict whether its corresponding fluent will change, given a *state-action* pair on the input.

3.4.1 Deictic Coding

Before doing anything else, authors compute a reduced form of the input *state-action* pair using an approach called deictic coding [Agre-Chapman, 1987]. For a given action instance a, they construct the set of **objects of interest** $\mathcal{O}^a$ by combining a *primary set of objects* (given by parameters of a) and *secondary set of objects*, which are directly *connected* to any of the objects in the primary set. We say that two objects c_i and c_j are connected in a state s, if $\exists\ l(c_1, \ldots, c_n) \in s$ such that $c_i, c_j \in \{c_1, \ldots, c_n\}$.

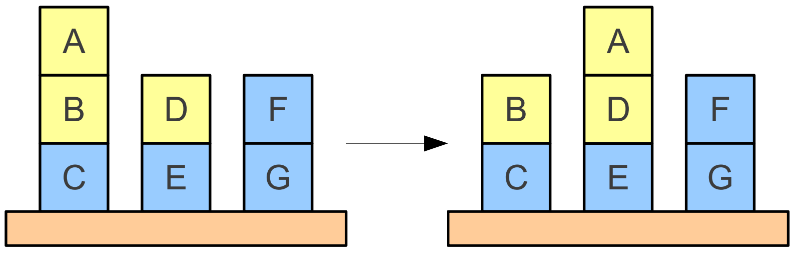


Figure 3.2: Action instance $move(A, B, D)$ in the Blocks World domain.

Let us take a look at the example configuration of the Blocks World (figure 3.2). Given an action $move(A, B, D)$ (moving a block A from block B to D), our primary set of objects would be $\{A, B, D\}$, secondary set would be $\{C, E\}$ (since we have literals $on(B, C)$ and $on(D, E)$ in our knowledge base), and we would ignore all the other objects ($\{F, G, table\}$). $\mathcal{O}^{\text{move(A,B,D)}}$ is then a set $\{A, B, D, C, E\}$.

Intuitively, deictic coding is used to heuristically filter out all the irrelevant (w.r.t. current action) objects from the input world state. Authors have shown that it not only speeds up the learning process significantly, but also allows this method to be scaled up to large domains. On the other hand, it prevents learning such effects that are seemingly not connected to our action at the moment (just because of the lack of knowledge about this connection). Consequently, it makes us more dependent on prior background knowledge.

3.4.2 Input/Output Representation

As we mentioned before, the **input** state-action pairs are translated into the vectors over $\{-1, 0, 1\}$. This translation is done as follows:

Each possible action and each 0-ary (object-independent) fluent is represented by one digit. Then for each object $o \in \mathcal{O}^a$, all the possible relations between o and all other objects in $\mathcal{O}^a$ must be represented by an additional digit. This digit equals to 1 if the corresponding fluent is observed *true*, or if the corresponding action is the *current* action. It equals to -1 if the fluent is observed *false*, or if the action is *not* the *current* one. If a digit corresponds to a fluent which was *not currently observed*, its value is set to 0.

Vector representation of a state-action pair from figure 3.2 follows. Note however, that in this simplistic case we have only one action, no object-independent fluents, and we can observe the whole world state:

	Actions:
1	move(A,B,D)
	Properties of A:
1	free
-1	on-A
1	on-B
-1	on-C
-1	on-D
-1	on-E
	Properties of B:
-1	free
-1	on-A
-1	on-B
1	on-C
-1	on-D
-1	on-E
	. . . *and similarly for objects D,C and E.*

The form of the expected **output** representing the action's effect on a state is similar to input vectors. The digits are set to 1 if the corresponding fluent changes, -1 if it stays the same and 0 if it was not observed either before, or after the action.

The **ordering** of the digits is of course constrained, so that two objects with the same role in two instances of the same action must share the same position in the vector.

3.4.3 Training the Perceptron

The perceptron **maintains a weight vector** (or matrix in general) w which is adjusted at each time step. The i-th input vector x_i is usually[8] classified by the perceptron using the decision function $f(x_i) = sign(\langle w \cdot x_i \rangle)$. Then if $f(x_i)$ is not a correct class, weight w is set either to $w + x_i$ or to $w - x_i$ (depending on whether the mistake was negative or positive). If $f(x_i)$ is correct, w stays unchanged.

This *perceptron algorithm* is guaranteed to converge on a solution in a finite number of steps only if the data is *linearly separable* [Minsky-Papert, 1969]. Unfortunately, this is in general not our case, and we need to deal with this problem.

One solution for non-linearly separable data is to **map the input feature space into a higher-dimensional space**, where the data is linearly separable. Since the explicit mapping is problematic, due to a massive expansion in the number of features, authors decided to apply the **kernel-trick** [Freund-Schapire, 1999] that allows the *implicit mapping*.

Note that the decision function can be written in terms of the dot product of the input vectors:

$$f(x_i) = sign(\langle w \cdot x_i \rangle) = sign(\sum_{j=1}^{n} \alpha_j y_j \langle x_j \cdot x_i \rangle)$$

where α_j is the number of times the j-th example has been misclassified by the perceptron, while $y_j \in \{-1, 1\}$ says whether the mistake was positive or negative. By replacing the typical dot product with a *kernel function* $k(x_i, x_j)$ which calculates $\langle \phi(x_i) \cdot \phi(x_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be applied in higher dimensional spaces without ever requiring

[8]In the original Perceptron Algorithm.

the mapping to be explicitly calculated. In their implementation, authors used the polynomial kernel of degree 3: $k(x, y) = (x \cdot y + 1)^3$

3.4.4 Evaluation

Mourao et al.'s method aims to learn the action effects of **partially observable** STRIPS domains. For this purpose, the authors decided to use a *voted kernel perceptron learning model*, while encoding the knowledge about the domain into binary vectors. The choice of perceptrons as the basis for learning allows for an elegant handling of **noise** and **action failures**, while keeping the computational complexity relatively low. This learning method is in principle stochastic and might be used to learn **probabilistic** action models.

Unfortunately, this solution lacks a certain degree of versatility, since it can not generate **neither preconditions, nor the conditional effects** of the actions.

Usability of this solution in **large domains** is achieved by applying the deictic coding to filter out irrelevant parts of world states. On the downside, using this approach intensifies our dependency on prior background knowledge, and prevents finding the effects affecting seemingly unrelated parts of the world.

3.5 Greedy Search for Learning Noisy Deictic Rule Sets

The fifth section of this chapter describes the action learning method introduced by Pasula, Zettlemoyer, and Kaelbling, which falls into the category of algorithmic techniques (in contrast to Balduccini's purely declarative approach from section 3.3). The paper was published in 2007 and is called *Learning Symbolic Models of Stochastic Domains* [Pasula-Zettlemoyer-Kaelbling, 2007].

Authors apply the **greedy search algorithms** to the problem of modelling **probabilistic actions** in **noisy** domains. For the purpose of evaluation, they designed a simulation of a three-dimensional Blocks World with realistic physics.

On the input, their algorithm takes a training set E consisting of examples of a form (s, a, s'), where a denotes an action instance, and s, s' are preceding and following (fully observed) world states. It then searches for an action model A that maximizes the likelihood of the action effects seen in E, while penalizing the candidate models based on their complexity.

For representation of the generated action models, the notion of *Noisy Deictic Rules (NDRs)* was introduced. Since, as authors claim, learning the sets of these rules is in general NP-hard [Zettlemoyer-Pasula-Kaelbling, 2003], the choice of greedy approach is adequate. Their search algorithm is hierarchically structured into three levels of greedy search:

1. **LearnRules** method represents the outermost level. It searches the space of possible rule sets, often by constructing new rules, or altering existing ones.

2. **InduceOutcomes** method is repeatedly called by the *LearnRules* method, thus representing the middle level. It generates the set of possible outcomes for a given rule.

3. **LearnParameters** method, called by the *InduceOutcomes*, then finds a probability distribution over these outcomes, while optimizing the likelihood of examples covered by this rule.

After the description of used representation language, we will discuss these three levels starting from the inside out, so that each subroutine is described before the one that depends on it.

3.5.1 Representation by NDRs

Each *Noisy Deictic Rule* specifies a small number of action **outcomes**, representing individual ways how the action can affect the world, along with the probability distribution over them. It consists of four components:

1. **Name** of the action with **parameters**,

2. a list of **deictic references** introducing additional variables and defining their types,

3. description of **context** in which the rule applies,

4. and a set of **outcomes** with assigned **probabilities** summing up to 1.0 (including a compulsory *noise* outcome).

Our traditional action $move(B, P_1, P_2)$ of Blocks World (example 2.1) can be represented as a set of two NDR rules quite easily, as depicted in figure

3.3. The deictic reference part ($\{T : \mathit{table}(T)\}$) of these rules serves simply to introduce additional variable T and define its type. We can see that two rules are applicable in different *contexts* - first rule applies if a position P_1 is not too high[9], while the second rule describes the situation when we are trying to put B on top of a tall pile of blocks.

$$
\begin{array}{l}
\mathit{move}(B, P_1, P_2) : \{T : \mathit{table}(T)\} \\
\mathit{free}(P_1), \mathit{free}(P_2), \mathit{on}(B, P_1), \mathit{height}(P_1) < 10 \\
\left\{ \begin{array}{ll}
0.7: & \neg \mathit{on}(B, P1), \mathit{on}(B, P2) \\
0.1: & \neg \mathit{on}(B, P1), \mathit{on}(B, T) \\
0.1: & \mathit{no\ change} \\
0.1: & \mathit{noise}
\end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{move}(B, P_1, P_2) : \{T : \mathit{table}(T)\} \\
\mathit{free}(P_1), \mathit{free}(P_2), \mathit{on}(B, P_1), \mathit{height}(P_1) \geq 10 \\
\left\{ \begin{array}{ll}
0.3: & \neg \mathit{on}(B, P1), \mathit{on}(B, P2) \\
0.2: & \neg \mathit{on}(B, P1), \mathit{on}(B, T) \\
0.1: & \mathit{no\ change} \\
0.4: & \mathit{noise}
\end{array} \right\}
\end{array}
$$

Figure 3.3: Action $\mathit{move}(B, P_1, P_2)$ of the Blocks World domain represented as a *noisy deictic rule*.

Intuitively, outcomes of the first rule mean that with probability of 0.7 we succeed in moving the block B to P_2, while with probability of 0.1 it either falls on the table, or nothing changes (for example if we failed to pick it up).

[9]Suppose that $\mathit{height}(P_1)$ is a function counting how many blocks are stacked under the position P_1.

Last "*noise*" outcome sums up all the other things that could happen but are so unlikely, that we do not want to model them individually.

Situations covered by the second rule are less stable. If our block is on top of a big stack of blocks, the chance that we successfully move it is only 0.3, while the chance that it falls on the table is 0.2. There is a high probability here that the big stack of blocks topples. All the possible configurations of blocks after that happening are included in the "*noise*" outcome.

Probability Calculation

Since the effects of the noise are not explicitly modelled, we cannot precisely calculate **the probability** $P(s'|s,a,r)$ that a noisy deictic rule r assigns to moving from state s to state s' when action a is taken. We can, however, bound the probability as:

$$\begin{aligned}\hat{P}(s'|s,a,r) &= p_{min}P(\Psi'_{noise}|s,a,r) + \sum_{\Psi'_i \in r} P(s'|\Psi'_i,s,a,r)P(\Psi'_i|s,a,r) \\ &\leq P(s'|s,a,r)\end{aligned} \tag{3.1}$$

where $P(\Psi'_i|s,a,r)$ is a probability assigned to outcome Ψ'_i and the outcome distribution $P(s'|\Psi'_i,s,a,r)$ is a deterministic distribution that assigns all of its mass to the relevant s'. The p_{min} constant assigns a small amount of probability mass to every possible next state s'. To ensure that the probability model remains well-defined, p_{min} multiplied by the number of possible states should not exceed 1.0.

Note that, in this representation, it is possible for a resulting state s' to be covered by more than one outcome. The probability of s' occurring after the execution of a given action is the sum of probabilities associated with relevant outcomes.

Compulsory Default Rule

In this formalism, any valid **rule set** representing an action model needs to contain one additional rule - a *default rule* which has an **empty context** and two possible outcomes: **no change** and **noise**. This rule expresses the probability of change, if no other rule applies.

3.5.2 Scoring Metric

A greedy search algorithm needs some kind of **scoring metric** to **judge** which parts of the search space are more desirable.

Definition 3 (Scoring metric for *rules* and *rule sets*)**.** Let E_r denote the set of input examples covered by rule r. Scoring metric $S(r)$ over a rule r is then computed as

$$S(r) = \sum_{(s,a,s') \in E_r} log(\hat{P}(s'|s,a,r)) - \alpha PEN(r)$$

where $PEN(r)$ is a complexity penalty applied to rule r and α is the scaling parameter[10]. Scoring metric $S(R)$ over a rule set R is then simply a sum of scores of all the rules $r \in R$:

$$S(R) = \sum_{r \in R} S(r)$$

We can see that $S(R)$ favours the rule sets that maximize the probability bound, while penalizing those that are overly complex.

[10]Authors used the scaling parameter $\alpha = 0.5$ in their experiments.

3.5.3 Learning Parameters

The *LearnParameters* method takes an incomplete rule consisting of an *action name with parameter list*, a set of *deictic references*, a *context*, and a set of *outcomes*, and **learns the distribution** P that maximizes the score of r on the examples E_r covered by it. The optimal distribution is simply the one that maximizes the log likelihood of E_r. In our case this will be:

$$L = \sum_{(s,a,s')\in E_r} log(\hat{P}(s'|s,a,r))$$

Fortunately, even though estimating the maximum-likelihood parameters is a nonlinear programming problem, it is an instance of a well-studied problem of maximizing a convex function over a probability simplex. Several gradient ascent algorithms with guaranteed convergence are known for this problem [Bertsekas, 1999]. The *LearnParameters* algorithm uses the **conditional gradient method**, which works by, at each iteration, moving along the axis with the maximal partial derivative. The step sizes are chosen using the **Armijo rule** discussed in [Armijo, 1966].

3.5.4 Inducing Outcomes

Input for the *InduceOutcomes* algorithm is an incomplete rule lacking the whole outcomes part. Its purpose is to find the optimal way to **fill the set of outcomes** and its associated probability distribution that maximizes the score according to metric from definition 3.

InduceOutcomes performs a greedy search over a restricted subset of possible outcome sets which cover at least one training example.

It searches this space until there are no more immediate moves that improve the rule score. To generate outcome sets, two operators are used:

- *Add* operator picks a pair of two non-contradictory outcomes and creates a new one that is their conjunction.
- *Remove* operator drops an outcome from the set if it is overlapping with another outcome on all the covered examples.

For each set of outcomes it considers, *InduceOutcomes* calls the *LearnParameters* function to supply the best probability distribution it can.

3.5.5 Learning Rules

Now that we know how to fill in incomplete rules, we will describe *LearnRules*, the outermost level of our learning algorithm, which takes a set of examples E and performs a greedy search over the space of *proper* rule sets. We say that a rule set is *proper* w.r.t. E, if it includes at most one rule that is applicable to every example $e \in E$ (in which a change occurs) and if it does not include inapplicable rules.

Search starts with a rule set containing only the *default rule* and applies some of the following set of **operators** to obtain new rule sets:

1. *ExplainExamples*
2. *DropRules*
3. *DropLits*[11]
4. *DropRefs*

[11]Here *"Lits"* stands for *literals*.

5. *GeneralizeEquality*

6. *ChangeRanges*

7. *SplitOnLits*

8. *AddLits*

9. *AddRefs*

10. *RaiseConstants*

11. *SplitVariables*

Due to relative complexity of these operators and limited space, we will not describe them here. See [Pasula-Zettlemoyer-Kaelbling, 2007] for their detailed specification.

To assign the score to newly produced rule sets, we use the scoring metric from definition 3 again.

3.5.6 Evaluation

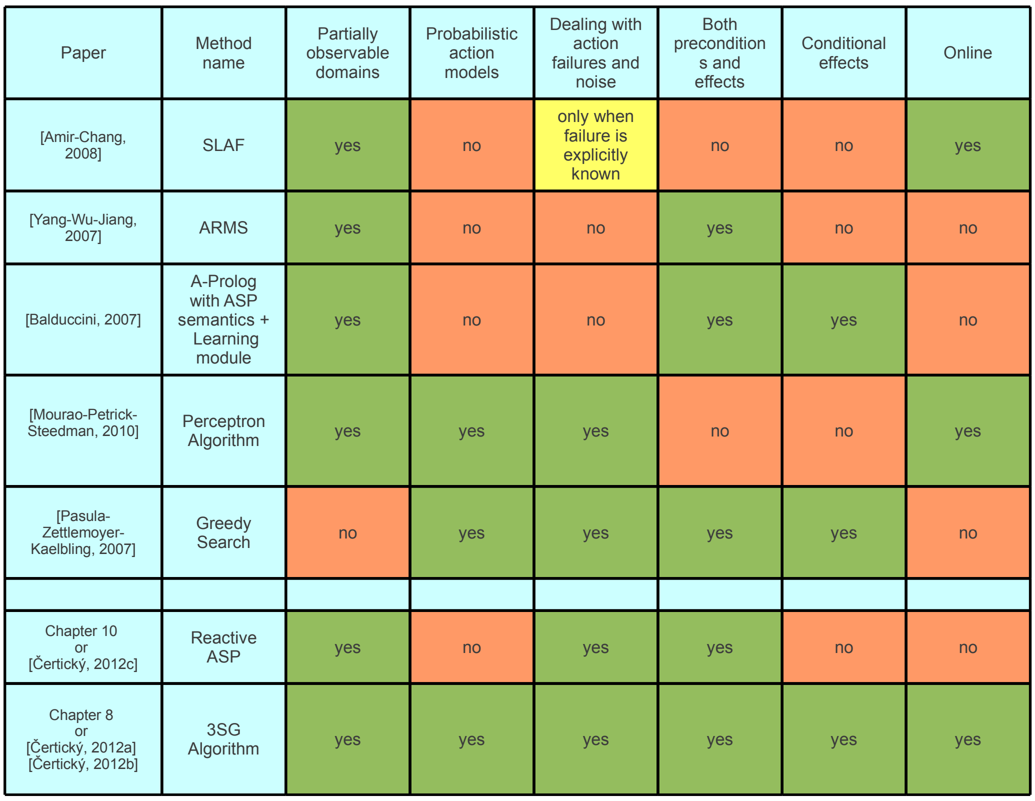
This method, proposed by Pasula et al., is a typical example of algorithmic approach to the problem of action learning. Authors experimented with relatively simplistic *greedy search* algorithm in conjunction with an appropriate representation language for the purpose of learning **probabilistic action models** including both **preconditions** (in a form of *contexts* in this case) and **effects**.

Even though the method effectively deals with **action failures** and sensoric **noise**, there are some significant downsides that we need to mention. The most restricting disadvantage of this technique is the **lack of support for partially observable domains**.

Usability in real-world domains (even if we overlooked the *full-observation* requirement) is diminished also by the fact that the algorithm used is a three-level search over the increasingly large dataset - the set of all previously observed **examples** is **needed** on the input, making the computational complexity a serious issue.

3.6 New Methods

In chapters 8 and 10, we will introduce two new methods for action model learning. Figure 3.4 contains a property-based comparison of these new methods to alternatives that we described in this chapter.



Paper	Method name	Partially observable domains	Probabilistic action models	Dealing with action failures and noise	Both precondition s and effects	Conditional effects	Online
[Amir-Chang, 2008]	SLAF	yes	no	only when failure is explicitly known	no	no	yes
[Yang-Wu-Jiang, 2007]	ARMS	yes	no	no	yes	no	no
[Balduccini, 2007]	A-Prolog with ASP semantics + Learning module	yes	no	no	yes	yes	no
[Mourao-Petrick-Steedman, 2010]	Perceptron Algorithm	yes	yes	yes	no	no	yes
[Pasula-Zettlemoyer-Kaelbling, 2007]	Greedy Search	no	yes	yes	yes	yes	no
Chapter 10 or [Čertický, 2012c]	Reactive ASP	yes	no	yes	yes	no	no
Chapter 8 or [Čertický, 2012a] [Čertický, 2012b]	3SG Algorithm	yes	yes	yes	yes	yes	yes

Figure 3.4: Comparison of properties of current action learning methods to new methods that will be introduced in this thesis.

Chapter 4

Evaluation based on Domain Compatibility

The analysis of alternative action learning methods in previous chapter is quite extensive. However, it shows us how they deal with individual challenges, helps us understand how it all relates to application domains, and allows us to select a suitable collection of domains for our own experiments.

Before we move on, it makes sense to compare all the action learning methods based on their usability in different kinds of domains, just as we did based on their properties.

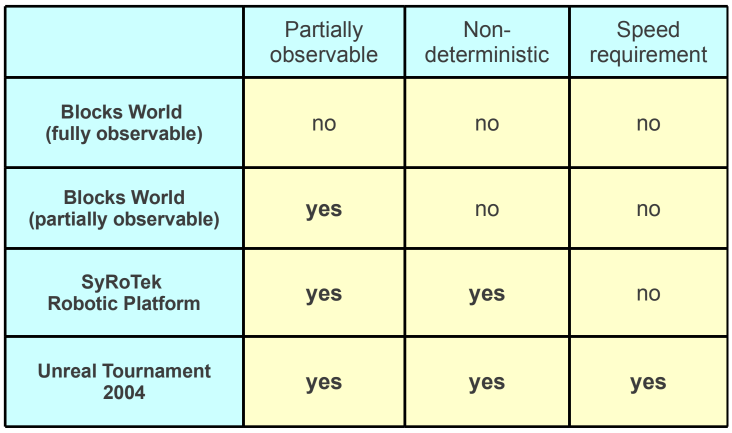
Any domain comes with a distinct set of *requirements* that action learning algorithms need to meet, if they are to be used there. The most commonly discussed are:

Partial observability: As we mentioned before, most of the domains are only partially observable. It means that, at any given moment, agents can only perceive a part of their surroundings, thus having incomplete information about the world state.

Non-determinism: By non-determinism of a domain we understand the possibility of action failures, sensoric noise, or any other kind of random events.

Speed requirement: Suppose that we want our agents to learn the action models in real time. Some domains are more demanding in terms of computation times than others. In such domains, agents often perceive elaborate observations in quick succession. Therefore, the learning algorithms need to be tractable in order to be usable there.

We will discuss and experiment with four different domains in this thesis, each having a different set of requirements (figure 4.1).



	Partially observable	Non-deterministic	Speed requirement
Blocks World (fully observable)	no	no	no
Blocks World (partially observable)	**yes**	no	no
SyRoTek Robotic Platform	**yes**	**yes**	no
Unreal Tournament 2004	**yes**	**yes**	**yes**

Figure 4.1: Four domains with different requirements.

The *Blocks World (fully observable)* is a simple toy domain, already explained in Example 2.1. Its modification, *partially observable Blocks World*, has the same rules, but the agent only gets a randomly selected half of the observations (he does not know the position of some of the blocks, or whether they are free or blocked).

Robotic Platform SyRoTek [Kulich et al., 2010] is a real-world domain containing a set of remotely controlled robots with a variety of sensors navigating through a customizable maze. There is a considerable amount of sensoric noise and action failures, but the agents have enough time between individual observations.

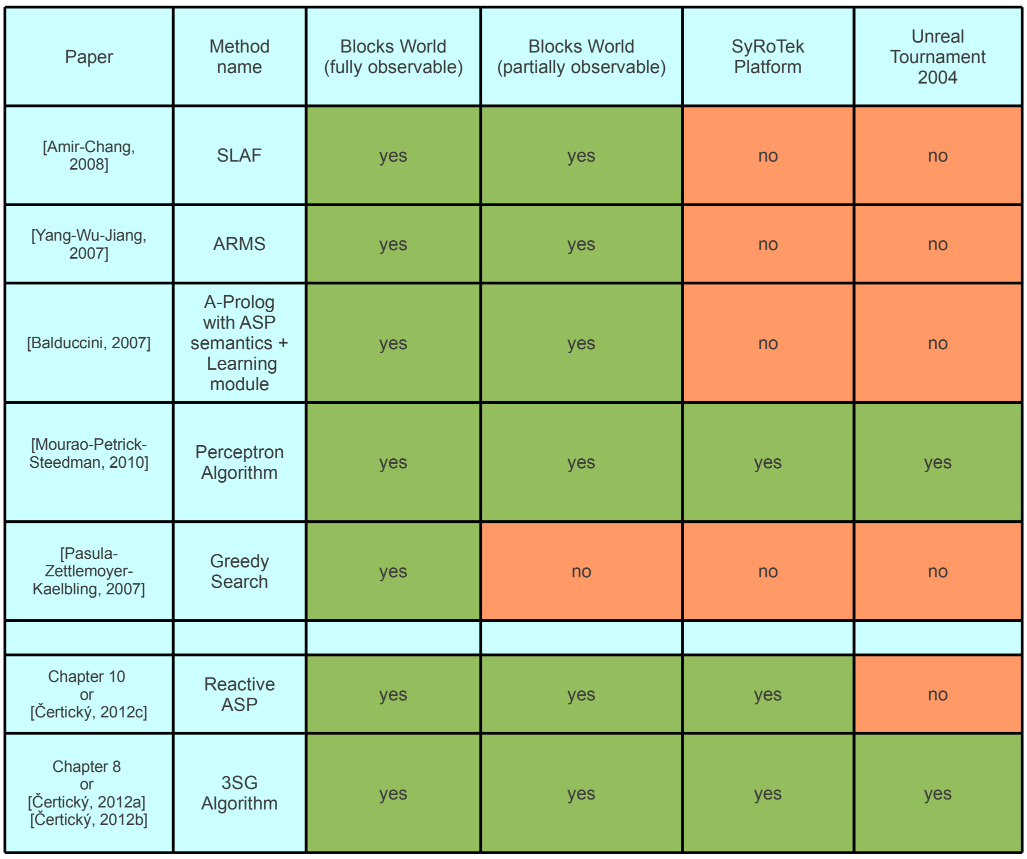
Unreal Tournament 2004 [Gemrot et al., 2009] is an action computer game. In addition to being non-deterministic, this domain requires agents to process their observations very quickly. There are many actions happening every second, and each observation contains lots of information.

Table in figure 4.2 provides a comprehensive overview of the compatibility of all discussed action learning methods with these four domains.

We can see that, while all the methods can be used in the fully observable Blocks World, the greedy-search method by Pasula et al. is unusable in its partially observable version (since, like we mentioned in section 3.5, it requires complete observations).

Frequent action failures in robotic platform SyRoTek render the majority of discussed methods unusable (according to corresponding sections in chapter 3, most of them are unable to deal with this problem). Only the perceptron-based approach by Mourao et al. and two methods of our own (chapters 8 and 10) are able to successfully deal with action failures in this partially observable domain.

Unreal Tournament 2004 seems to be the most demanding of all the domains discussed in this thesis. In addition to dealing with partial observability and action failures, it forces the agents to process elaborate observations relatively quickly. This renders one of our own methods ("reactive ASP" method from



Paper	Method name	Blocks World (fully observable)	Blocks World (partially observable)	SyRoTek Platform	Unreal Tournament 2004
[Amir-Chang, 2008]	SLAF	yes	yes	no	no
[Yang-Wu-Jiang, 2007]	ARMS	yes	yes	no	no
[Balduccini, 2007]	A-Prolog with ASP semantics + Learning module	yes	yes	no	no
[Mourao-Petrick-Steedman, 2010]	Perceptron Algorithm	yes	yes	yes	yes
[Pasula-Zettlemoyer-Kaelbling, 2007]	Greedy Search	yes	no	no	no
Chapter 10 or [Čertický, 2012c]	Reactive ASP	yes	yes	yes	no
Chapter 8 or [Čertický, 2012a] [Čertický, 2012b]	3SG Algorithm	yes	yes	yes	yes

Figure 4.2: Comparison of domain compatibility of action learning methods.

chapter 10) unusable. However, the approach by Mourao et al. and our 3SG algorithm (chapter 8) can both be used in this domain.

Chapter 5

Basic Notions

Action Model

From now on, we will be using the notion of **action model** defined as a double $(\mathcal{D}, \mathcal{P})$, where $\mathcal{D}$ is a *description of domain dynamics* in some kind of representation structure or language, and $\mathcal{P}$ is a *probability function* over $\mathcal{D}$. **Action learning** will then be understood as an algorithmic process of improving $\mathcal{D}$, while using $\mathcal{P}$ to do so. Before we can define the notion of action learning formally, we need to have a precise description of $\mathcal{D}$ and $\mathcal{P}$ (presented in chapters 6 and 7).

Fluents, World States, Observations

Description of domain dynamics $\mathcal{D}$ explains how the execution of individual **actions** leads from one discrete **world state** to another. To describe a world state, we use the collection of **fluent literals** (often called simply *fluents*): a fluent is either an atom f (positive fluent) or its negation $\neg f$ (negative fluent). Every fluent represents one individual feature of our domain. The set of all fluents of our domain is denoted by $\mathcal{F}$. Complement of a positive fluent f (denoted by $\overline{f}$) is defined to be $\neg f$, while complement of a negative fluent $\neg f$ (denoted by $\overline{\neg f}$) is a positive fluent f.

World state s is a non-empty set of fluents that contains exactly one out of every pair of mutually complementary fluents (either f or $\neg f$, but never both of them). We will denote the set of all the possible world states by $\mathcal{S}$. Formally:

$$\mathcal{S} = \{s \mid \forall (f, \neg f) \in \mathcal{F} : (f \in s \wedge \neg f \notin s) \vee (f \notin s \wedge \neg f \in s)\}$$

Observation is defined as any non-empty set of fluents. Unlike world states, observations may be incomplete $(\exists (f, \neg f) \in \mathcal{F} : f \notin s \wedge \neg f \notin s)$ in partially observable domains and even inconsistent $(\exists (f, \neg f) \in \mathcal{F} : f \in s \wedge \neg f \in s)$ if the sensoric noise is present.

Actions and Parameters

Action instance is an expression consisting of an **action name** and a (possibly empty) set of constant **parameters** reffering to objects of our domain. Typical example of **action instance** is "$move(block01, block02, table)$" expression of Blocks World (example 2.1). The set of all the action instances possible in our domain is called $\mathcal{A}$. Throughout the rest of this text, we will use the notion of action instance to express observed actions. We will mostly refer to them simply as *"actions"*.

Chapter 6

Representation Structures

Having established the meaning of $\mathcal{F}, \mathcal{S}$ and $\mathcal{A}$, we can move on to formal definitions of domain dynamics description $\mathcal{D}$. Throughout this chapter, we will describe **four different structures** that can be used to **represent** domain dynamics $\mathcal{D}$ (one of them was presented in related literature and other three are our own modifications). Then we will define a probability function $\mathcal{P}$ over each of those structures.

6.1 Transition Relation $\mathcal{TR}$

Transition relation ($\mathcal{TR}$) is quite common way of describing the domain dynamics. The term has been used for example in [Amir-Russel, 2003] or [Amir-Chang, 2008] and is similar to the representation structures used for instance in [Eiter et al., 2005].

Definition 4 (Transition Relation)**.** Having the sets $\mathcal{F}, \mathcal{S}$ and $\mathcal{A}$ defined as above, a **transition relation** is any relation $\mathcal{TR} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$.

Intuitive meaning of every $(s, a, s') \in \mathcal{TR}$ is that *"the execution of action **a** in a world state **s** causes a world state **s'** to hold in the next time step"*.

Note that the space complexity of a transition relation $\mathcal{TR}$ is $O(|\mathcal{A}| \cdot |\mathcal{S}|^2)$ which is equal to $O(|\mathcal{A}| \cdot (2^{|\mathcal{F}|/2})^2)$.

6.2 Effect Relation $\mathcal{ER}$

It is evident that representing the domain dynamics by a transition relation $\mathcal{TR}$ is relatively robust in terms of space requirements. In order to allow for faster, tractable solutions, we would like to come up with a more compact representation.

First alternative to $\mathcal{TR}$ introduced in this work is called an **effect relation** $\mathcal{ER}$. Its structure is quite similar to that of $\mathcal{TR}$:

Definition 5 (Effect Relation)**.** Having the sets $\mathcal{F}, \mathcal{S}$ and $\mathcal{A}$ defined as above, an **effect relation** is any relation $\mathcal{ER} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{F}$.

Here the meaning of each triple $(s, a, f) \in \mathcal{ER}$ is that *"the execution of action **a** in a world state **s** causes a fluent **f** to become true in the next time step"*. Let us now take a look at the relationship between the elements of $\mathcal{TR}$ and $\mathcal{ER}$.

Transformation 1. *Any $(s, a, f) \in \mathcal{ER}$ can be expressed by a transition relation $\mathcal{TR}$.*

We can simply translate (s, a, f) into a set $\{(s, a, s') \mid s' \in \mathcal{S} \wedge f \in s' \wedge \overline{f} \in s\}$. In other words, we translate (s, a, f) into a set of all the (s, a, s') triples where f *began to hold* after a was executed.

Transformation 2. *Any $(s, a, s') \in \mathcal{TR}$ can be expressed by an effect relation $\mathcal{ER}$.*

If we want to represent (s, a, s') by an effect relation $\mathcal{ER}$, we need to consider two distinct cases:

$s \neq s'$: This means that there exists some $f \in s'$ such that $\overline{f} \in s$. In order to represent (s, a, s') by $\mathcal{ER}$, we need to have (s, a, f) included in this $\mathcal{ER}$ for every such f:

$$\forall f \in \mathcal{F} : (\overline{f} \in s \land f \in s') \Rightarrow (s, a, f) \in \mathcal{ER}$$

$s = s'$: This means that action a has no effect in world state s. This fact is expressed implicitly by the **absence of** (s, a, f) elements in $\mathcal{ER}$. Any $\mathcal{ER}$, where the following holds, implicitly represents (s, a, s') if $s = s'$:

$$\nexists f \in \mathcal{F} : (s, a, f) \in \mathcal{ER}$$

Notice that the **space complexity** of $\mathcal{ER}$ is only $O(|\mathcal{A}| \cdot |\mathcal{F}| \cdot 2^{|\mathcal{F}|/2})$ which is lower than that of $\mathcal{TR}$ (considering $|\mathcal{F}| > 4$). We managed to save some space by expressing part of the action model implicitly.

6.3 Effect Formula $\mathcal{EF}$

Another alternative representation structure is called the **effect formula** $\mathcal{EF}$, and will be used in one of the methods presented in this work ($3SG$ algorithm). Unlike in previous two cases, the effect formula is not structured as a relation. It is rather a finite set of **propositional atoms** over a vocabulary $\mathcal{L}_{\mathcal{EF}} = \{a^f \mid a \in \mathcal{A} \land f \in \mathcal{F}\} \cup \{a^f_c \mid a \in \mathcal{A} \land f, c \in \mathcal{F}\}$. The intuitive meaning of atoms from $\mathcal{EF}$ follows:

- a^f: *"action **a** causes **f**"*

  Example: $move(b,c,a)^{on(b,a)}$

- a^f_c: *"**c** must hold in order for **a** to cause **f**"* (c is a condition of a^f)

  Example: $move(b,c,a)^{on(b,a)}_{free(a)}$

For example, $\mathcal{EF} = \{drink^{\neg thirsty}, drink^{sick}, drink^{sick}_{poisoned}\}$ means that *"if we drink a beverage, we will not be thirsty"* and that *"if the beverage was poisoned, we will get sick"*.

Notice that every a^f-type element (sometimes called an "effect") can have a set of its own "conditions" $a^f_{c_1} \dots a^f_{c_n}$. Also note that even though $\mathcal{EF}$ is formally defined as a *set*, it is basically understood as a *conjunction* of all its elements. This means that all the conditions $c_1 \dots c_n$ must hold in a world state in order for their effect a^f to be applicable.

A careful reader may already see that this causes a loss of expressivity. If the whole $\mathcal{EF}$ is a conjunction, we cannot use it to express the effects with disjunctive conditions (for example *"if the beverage was poisoned* **or** *if we are allergic, we will get sick"*).

However, expressing disjunctive effect conditions is not needed. Neither our own action learning algorithms, nor the majority of alternative methods are able to actually induce them. Therefore, there is no need for our representation structure to support them.

Thanks to this simplification, this new candidate for action model representation is considerably more compact. Upper bound of its space complexity is only $O(|\mathcal{A}| \cdot (|\mathcal{F}| + |\mathcal{F}|^2))$, which is exponentially lower than in previous cases (considering $|\mathcal{F}| > 4$). This makes it a suitable candidate for our action learning algorithm $3SG$.

6.4 A-Prolog Literals $\mathcal{AL}$

After two relations and a set of specific propositional atoms, we move on to the fourth possible way of representing domain dynamics - as a set of *A-Prolog literals.* Action's effects and preconditions encoded in this way will be used in our second action learning method, described in chapter 10.

The *A-Prolog* language, based on the semantics proposed by Gelfond and Lifschitz in [Gelfond-Lifschitz, 1988], [Gelfond-Lifschitz, 1991] and described extensively in [Baral, 2003], is a formalism used in a certain paradigm of logic programming called *Answer Set Programming.* As we already mentioned in section 3.3, this language can be conveniently used for various kinds of commonsense reasoning, including the action model representation and learning. One example of A-Prolog based encoding of actions was already described in [Balduccini, 2007]. We propose an alternative which is more compact [Čertický, 2012c], but on the other hand completely lacks the ability to represent any kind of conditional effects.

The main idea behind our encoding lies in the simplifying assumption that for any pair of mutually complementary fluents $(f, \neg f)$, every possible action a must either:

1. **cause** a positive fluent f to hold
   (we encode this by the A-Prolog literal "$causes(a, f)$")

2. **cause** a **complementary** (negative) fluent $\neg f$ to hold
   (encoded by "$causes(a, \neg f)$")

3. or **keep the value** of those fluents unchanged
   (encoded by "$keeps(a, f)$")

In addition to that, we want our action models to contain the information about action's executability. In that respect, for every single fluent f (positive or negative), an action a can either:

1. **have** a fluent f as its **precondition** (encoded by "$pre(a, f)$")

2. or **not have** it as its **precondition** ("$\neg pre(a, f)$")

Action's (unconditional) effects and preconditions can then be expressed by a set of A-Prolog literals $\mathcal{AL}$.

Note that for every positive fluent, the set $\mathcal{F}$ must contain also its negative counterpart (and vice versa). We can therefore divide $\mathcal{F}$ into a set of all the positive fluents $\mathcal{F}^+$ and a set of all the negative fluents $\mathcal{F}^-$ with the same cardinality ($|\mathcal{F}^+| = |\mathcal{F}^-| = |\mathcal{F}|/2$). For every $a \in \mathcal{A}$ and every $f \in \mathcal{F}^+$, the $\mathcal{AL}$ set contains exactly one out of following three literals $\{causes(a, f), causes(a, \neg f), keeps(a, f)\}$. In addition to that, $\mathcal{AL}$ also contains exactly one out of the following two literals $\{pre(a, f), \neg pre(a, f)\}$ for every single fluent $f \in \mathcal{F}$ (positive or negative). This means that, for any action model, the space complexity of $\mathcal{AL}$ is exactly $|\mathcal{A}| \cdot |\mathcal{F}|/2 \, + \, |\mathcal{A}| \cdot |\mathcal{F}|$.

6.5 Probability Function

Having defined various set-based structures for description of domain dynamics, we can now understand the process of action learning as an **iterative modification** of $\mathcal{D}$ by **adding** and **deleting** some of its elements **based on the observations**.

However, remember that we want our methods to be usable in **probabilistic domains** with the possibility of **action failures** and **sensoric noise**. This prevents us from deleting elements from $\mathcal{D}$ based on just one observation.

Example 6.1
Imagine, for example, that our action model $\mathcal{D}$ says that *"if we move block b_1 from block b_3 to a free block b_2, it will be on top of it"*. Such knowledge can be represented by a simple effect formula $\mathcal{EF} = \{move(b_1, b_3, b_2)^{on(b_1,b_2)}$, $move(b_1, b_3, b_2)^{on(b_1,b_2)}_{free(b_2)}\}$. Now imagine that we try to move it, but our hand slips and nothing happens. If this happens rarely, deleting corresponding elements from $\mathcal{EF}$ is not a good idea (because they are correct most of the time).

To decide which elements of $\mathcal{D}$ are correct most of the time, and which are mostly false (and therefore should be deleted), we need a **probability function** $\mathcal{P}$.

Regardless of how $\mathcal{D}$ is represented, function $\mathcal{P}$ assigns every element $i \in \mathcal{D}$ a real number $p \in [0, 1]$ representing its probability:

$$\mathcal{P}(i) = \frac{pos(i)}{pos(i) + neg(i)}$$

where $pos(i)$ is a number of positive examples and $neg(i)$ is a number of negative examples w.r.t. i.

The definition of $pos(i)$ and $neg(i)$ differs based on the representation of $\mathcal{D}$ (see the following chapter).

Chapter 7

Examples, Action Learning and Performance Measures

7.1 Positive and Negative Examples

Recall that the observation o (as defined in chapter 5) is a non-empty set of fluents ($o \neq \emptyset \ \wedge \ o \subseteq \mathcal{F}$).

Triple (o, a, o') where o, o' are observations from two consecutive time steps, and $a \in \mathcal{A}$ is an action that was executed in the first of them, is called an **example**. From agent's perspective, we can define the example more formally:

Definition 6 (Example)**.** An example from time step t is a triple $(o, a, o')_t$ if our sensors provided the following information:

1. $\forall f \in o : f$ was true in time $t-1$,

2. $\forall f \in o' : f$ was true in time t,

3. a was an action executed in time $t-1$.

Throughout the rest of this text, if we do not need to specify a certain time step, we will omit the subscript t and use the simplified notation "(o, a, o')".

Collection of all the examples that are used by an agent to learn the action model is called a **training set**. Another collection of examples, called **test set**, is commonly used for the evaluation of previously learned action model. Test set is usually smaller than training set, and they should always be disjoint.

Let us now define a set of all the fluents that **changed their value** within an example (o, a, o') as $\Delta(o, o') = \{f \mid \overline{f} \in o \wedge f \in o'\}$.

Intuitively, we say that (o, a, o') is a **positive example** w.r.t. $i \in \mathcal{D}$ if it *"confirms"* the meaning of i. Conversely, (o, a, o') is a **negative example** w.r.t. i, if it *"denies"* the meaning of i. Since the semantics of individual elements $i \in \mathcal{D}$ depends on our choice of representation structure, we need to separately define the positive and negative examples for all four of $\mathcal{TR}, \mathcal{ER}, \mathcal{EF}$ and $\mathcal{AL}$.

Recall that the notation $pos(i)$ and $neg(i)$ denotes the **number of** positive and negative **examples** w.r.t. element $i \in \mathcal{D}$.

$\mathcal{TR}$:

In case of $\mathcal{D}$ represented by a **transition relation** $\mathcal{TR}$, we define the number of *positive* and *negative* examples w.r.t. an element (s, a, s') in a following manner:

$$pos((s, a, s')) = |\{(o, a, o') \mid o = s \wedge o' = s'\}|$$

$$neg((s,a,s')) = |\{(o,a,o') \mid o = s \land \exists f \in o' : \overline{f} \in s'\}|$$

$\mathcal{ER}$:

If our action model $\mathcal{D}$ is represented by an **effect relation** $\mathcal{ER}$, we say that an example (o,a,o') is *positive* w.r.t. (s,a,f) only if we are sure that f changed its value from $\overline{f}$ after a was executed. *Negative* examples are those, where $\overline{f}$ still holds after the execution of a.

$$pos((s,a,f)) = |\{(o,a,o') \mid o = s \land f \in \Delta(o,o')\}|$$

$$neg((s,a,f)) = |\{(o,a,o') \mid o = s \land \overline{f} \in o'\}|$$

$\mathcal{EF}$:

When we use an **effect formula** $\mathcal{EF}$ to represent $\mathcal{D}$, we need to define the positive and negative examples for both kinds of elements: a^f and a^f_c. Definitions for a^f-type elements are quite similar to previous case.

$$pos(a^f) = |\{(o,a,o') \mid f \in \Delta(o,o')\}|$$

$$neg(a^f) = |\{(o,a,o') \mid \overline{f} \in o'\}|$$

In compliance with the meaning of a^f_c, that treats c as a *condition* of a causing f, we define the *positive* examples as those where c was observed when f started to hold. Conversely, any example where $\overline{c}$ was observed when a caused f is considered *negative*.

$$pos(a^f_c) = |\{(o,a,o') \mid c \in o \land f \in \Delta(o,o')\}|$$

$$neg(a^f_c) = |\{(o,a,o') \mid \overline{c} \in o \land f \in \Delta(o,o')\}|$$

$\mathcal{AL}$:

If our action model is represented by a set of **A-Prolog literals** $\mathcal{AL}$, we need to define the positive and negative examples for four separate kinds of literals: $causes(a, f)$, $keeps(a, f)$, $pre(a, f)$ and $\neg pre(a, f)$.

$$pos(causes(a, f)) = |\{(o, a, o') \mid f \in \Delta(o, o')\}|$$

$$neg(causes(a, f)) = |\{(o, a, o') \mid \overline{f} \in o'\}|$$

Any example where f began to hold after the execution of action a is considered *positive* w.r.t. the literal $causes(a, f)$. Examples where f did not hold after a are considered *negative* w.r.t. this literal.

$$pos(keeps(a, f)) = |\{(o, a, o') \mid (f \in o \wedge f \in o') \wedge (\overline{f} \in o \wedge \overline{f} \in o')\}|$$

$$neg(keeps(a, f)) = |\{(o, a, o') \mid (f \in o \wedge \overline{f} \in o') \vee (\overline{f} \in o \wedge f \in o')\}|$$

Examples, in which the fluent stays unchanged after the execution of action a, are considered *positive* w.r.t. $keeps(a, f)$. Examples, where f changed to $\overline{f}$ (or vice versa), are *negative* w.r.t. $keeps(a, f)$. Note that the intuition behind this notion is that if an action keeps the fluent f unchanged, it also keeps the complementary fluent $\overline{f}$ unchanged.

$$pos(pre(a, f)) = |\{(o, a, o') \mid f \in o\}|$$

$$neg(pre(a, f)) = |\{(o, a, o') \mid \overline{f} \in o\}|$$

$$pos(\neg pre(a, f)) = |\{(o, a, o') \mid (o, a, o') \textit{ is negative w.r.t. } pre(a, f)\}|$$

$$neg(\neg pre(a, f)) = |\{(o, a, o') \mid (o, a, o') \textit{ is positive w.r.t. } pre(a, f)\}|$$

We consider those examples, where f was true when the action a was executed, *positive* w.r.t. the literal $pre(a, f)$, since this literal says that f is a precondition of action a. Examples, where f was not true when a was executed, are *negative* w.r.t. this literal. Literal $\neg pre(a, f)$ has a meaning opposite to $pre(a, f)$ and therefore the examples that are positive w.r.t. $pre(a, f)$ are negative w.r.t. $\neg pre(a, f)$ and vice versa.

Note 7.1.1. From now on, we will not discuss the first two representation structures ($\mathcal{TR}$ and $\mathcal{ER}$), since our algorithms do not use them. To represent the domain dynamics $\mathcal{D}$, we will use the $\mathcal{EF}$ in chapters 8 and 9 and the $\mathcal{AL}$ in chapters 10 and 11.

7.2 Action Learning

We have already mentioned that the action learning is an iterative process where we add or delete some elements of $\mathcal{D}$.

In other words, this modification either **specifies** or **generalizes** our domain description $\mathcal{D}$, depending on the set of examples covered by it. To understand this, we first need to define following two notions:

We say that an action model $\mathcal{D}$ **covers** an example (o, a, o'), if it *correctly describes all the changes* that occur in this example. In other words (considering that $\mathcal{D}$ is represented by $\mathcal{EF}$) $\forall f \in \Delta(o, o') : a^f \in \mathcal{EF}$, or (if $\mathcal{D}$ is represented by $\mathcal{AL}$) $\forall f \in \Delta(o, o') : causes(a, f) \in \mathcal{AL}$.

Similarly, an action model is **in conflict** with an example (o, a, o'), if some of the effects or preconditions it describes are *inconsistent with* this example. In case of $\mathcal{D}$ represented by $\mathcal{EF}$: $\exists\, a, f : a^f \in \mathcal{EF} \wedge (\forall c : a^f_c \in \mathcal{EF} \Rightarrow c \in o) \wedge (\overline{f} \in$

$o')$. In case of $\mathcal{D}$ represented by $\mathcal{AL}$: $\exists\, a, f : \big(causes(a,f) \in \mathcal{AL} \wedge \overline{f} \in o'\big) \vee \big(keeps(a,f) \in \mathcal{AL} \wedge (\overline{f} \in \Delta(o,o') \vee f \in \Delta(o,o'))\big) \vee \big(pre(a,f) \in \mathcal{AL} \wedge \overline{f} \in o\big)$.

We say that $\mathcal{D}'$ is a **generalization** of $\mathcal{D}$ ($\mathcal{D}$ is a **specification** of $\mathcal{D}'$), if $\mathcal{D}'$ **covers** every example (o, a, o') covered by $\mathcal{D}$ and at least one more example. Using this terminology, we can define the action learning in the following fashion:

Definition 7 (Action Learning)**. Action learning** is a process of iterative *specification* and *generalization* of $\mathcal{D}$ in order to:

1. **Maximize** the set of examples **covered** by $\mathcal{D}$.

2. **Minimize** the set of examples that are **in conflict** with $\mathcal{D}$.

Note 7.2.1. When dealing with probabilistic domains, we will often also need to maintain a probability function $\mathcal{P}$ and use it throughout this process.

Note 7.2.2. Action learning can also be seen as a multi-objective optimization problem [Deb, 2005], where one objective is the number of *conflicts*, and the other one is the number of *covered examples*. This problem is sometimes referred to as *precision-recall* trade-off [Gordon-Kochen, 2007, Bishop, 2006]. We will take a look at action learning from this perspective further in the text.

7.3 Miscellaneous Notions

Before we conclude this chapter, let us take one more look at the set of *desired properties* of action learning methods, that were first enumerated in section 2.4. We should try to briefly recapitulate and clarify some of these notions, using our newly established terminology.

Notion 1. Algorithms with their input served *one piece at a time* which, upon receiving it, have to take an irreversible action without the knowledge of the future inputs, are called **online** [Borodin-El-Yaniv, 1998]. In case of online action learning algorithms, this piece of input is the most recent *example.* They run repeatedly during an agent's existence, while improving his action model. In contrast, offline action learning algorithms need to have the whole training set at their disposal. Therefore, they are not suitable for gradual learning during agent's possibly long life.

Notion 2. A *domain* is called **probabilistic**, if the execution of the same action in the same world state can have different outcomes. Some of the unexpected outcomes are referred to as **action failures** or **noise**. An *action model* $(\mathcal{D}, \mathcal{P})$ is called probabilistic, if $\mathcal{P}$ assigns a probability value $0 < \mathcal{P}(i) < 1$ to at least one element $i \in D$.

Notion 3. A domain is **fully observable**, if $\forall(o, a, o')$: both o and o' meet the definition of a *world state* (in that they are complete and internally consistent). Otherwise, we say that a domain is **partially observable**.

Notion 4. We say that an action model contains the **conditional effects**, if (according to this model) at least one action has a different sets of applied effects when executed in different world states.

Chapter 8

$3SG$ Algorithm

To address the shortcomings of current action learning methods discussed in chapter 3, we introduce and analyse our own method - an online algorithm called $3SG$ [Čertický, 2012a, Čertický, 2012b].

8.1 The Algorithm

Throughout this description, we will be referring to the pseudocode from figure 8.3. Acronym "$3SG$" stands for *"Simultaneous Specification, Simplification and Generalization"* and it says a lot about the algorithm's structure[1]. $3SG$ is composed of three corresponding parts:

- By **generalization**, we understand the addition of a^f type atoms into $\mathcal{EF}$. That is because their addition enlarges the set of examples covered by $\mathcal{EF}$. Generalization is in our algorithm taken care of by the first *Foreach* cycle (lines 0-10), which adds a hypothesis (atom a^f) about

[1]The name "$3SG$" is inspired by algorithm $SLAF$ from [Amir-Chang, 2008].

a causing f, after we observe that $\overline{f}$ changed to f. If such element is already in $\mathcal{EF}$, then the algorithm modifies its probability, along with all its conditions.

Imagine, for example, that our $\mathcal{EF}$ was empty when we observed the world state from figure 8.1 (Blocks World) followed by the action $a =$ "$move(b, c, a)$". In other words, $o = \{on(b, c),\ \neg on(b, a),\ free(a),$ $\neg free(c), \dots\}$, and the following observation $o' = \{\neg on(b, c), on(b, a),$ $\neg free(a),\ free(c), \dots\}$. Fluents that changed their value are therefore $\Delta = \{\neg on(b, c),\ on(b, a),\ \neg free(a),\ free(c)\}$. We start with the first of them and add the "$move(b, c, a)^{\neg on(b,c)}$" hypothesis into $\mathcal{EF}$.

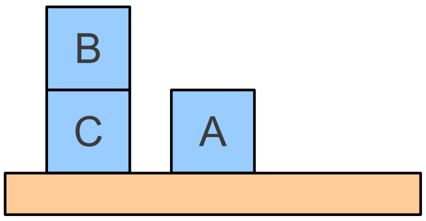


Figure 8.1: Simple instance of the Blocks World.

- **Specification** is on the other hand an addition of a^f_c type atoms, which is taken care of by the second *Foreach* cycle (lines 12-18). We call it a "specification" because new a^f_c elements restrict the set of examples covered by $\mathcal{EF}$ by imposing new conditions on the applicability of previously learned a^f effects. This cycle iterates over all our a^f type atoms, with respect to which our current example is negative, lowers their probability, and generates several hypotheses about effect a^f failing because of the existence of some condition a^f_c such that c does not hold right now.

Now imagine that we are in the world state from figure 8.2, and we try to execute the same action "$move(b, c, a)$". Since this is not possible ("a" is not free), nothing will happen ($o' = o = \{\neg free(a), on(b, c), \neg on(c, table), \ldots\}$). Therefore, this is a negative example w.r.t. our previous hypothesis "$move(b, c, a)^{\neg on(b,c)}$". To explain this, we assume that our hypothesis has some conditions that are not met in this particular example. Therefore, we take everything that for sure does not hold in o, and use it to create conditions:

$move(b, c, a)^{\neg on(b,c)}_{free(a)}$, $move(b, c, a)^{\neg on(b,c)}_{\neg on(b,c)}$, $move(b, c, a)^{\neg on(b,c)}_{on(c,table)}$ ...

We can see that we have added the correct condition ("a must be free"), and a couple of wrong ones. The wrong ones will, however, be eventually deleted in the "Simplification" part.

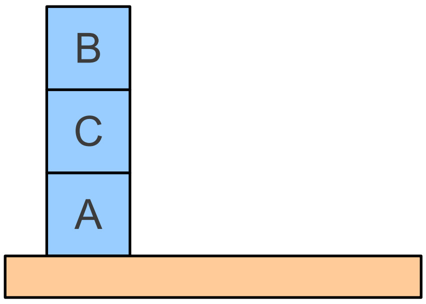


Figure 8.2: Another instance of the Blocks World.

- **Simplification** of $\mathcal{EF}$ is simply *forgetting* of those elements, which have not been validated enough (their probability is too low) during some limited time period. This is taken care of by the last two cycles (lines 21-28). Notice that this part of the algorithm makes use of three constant parameters, which need to be set in advance: Positive integer *memoryLength* represents the number of time steps after which we may forget the improbable elements, and $minP \in (0, 1]$ is the minimal probability threshold used to decide which of them are probable enough. Another positive integer, $minEx$, specifies the minimal number of relevant (positive or negative) examples that needs to be met in order to keep the element in $\mathcal{EF}$.

```
INPUT:  Newest example (o,a,o'), where a is executed action, and o,o' are observations.
        Current action model ⟨EF,P⟩, which we will modify (possibly empty).
OUTPUT: Modified action model ⟨EF,P⟩.

Foreach $f \in \Delta(o, o')$ do {
    // generalization of EF
    If $a^f \notin EF$ then Add $a^f$ to EF.
    Else {
        Modify $P(a^f)$ by incrementing $\mathrm{pos}(a^f)$ .
        Foreach $c \in o$ do {
            if $a_c^f \in EF$ then Modify $P(a_c^f)$ by incrementing $\mathrm{pos}(a_c^f)$ .
            if $a_{\overline{c}}^f \in EF$ then Modify $P(a_{\overline{c}}^f)$ by incrementing $\mathrm{neg}(a_{\overline{c}}^f)$ .
        }
    }
}

Foreach $f$ such that $\overline{f} \in o'$ do {
    If $a^f \in EF$ {
        Modify $P(a^f)$ by incrementing $\mathrm{neg}(a^f)$ .
        // specification of EF
        Foreach $c$ such that $\overline{c} \in o$ do: if $a_c^f \notin EF$ then Add $a_c^f$ to $EF$ .
    }
}

// simplification of EF
Foreach $a_c^f \in EF$ older than memoryLength do {
    If $P(a_c^f) < minP$ then Delete $a_c^f$ from EF.
}
Foreach $a^f \in EF$ older than memoryLength do {
    If ( $P(a^f) < minP$ and there is no $a_c^f$ in EF) or ( $\mathrm{pos}(a^f) + \mathrm{neg}(a^f) < minEx$ ) {
        Delete $a^f$ from EF.
    }
}
```

Figure 8.3: Pseudocode of $3SG$ algorithm.

Note 8.1.1. Even though it happens quite rarely, sensoric noise can cause our observations to be internally inconsistent (containing f and $\overline{f}$ at the same time). It therefore makes sense to check for such inconsistency before calling the $3SG$ algorithm, and if we have two mutually complementary fluents in our observation, remove both of them.

8.2 Correctness and Complexity

Before we can discuss the properties of $3SG$, we need to understand that it is an *online* algorithm. Like we mentioned in previous chapter, in the context of action learning, this means that instead of the whole training set, we always get only one (newest) example on the input. This example is used for irreversible modification of our action model. One drawback of online algorithms, compared to their offline counterparts, is that their final result depends on the ordering of received inputs, which makes them potentially less precise. On the other hand, online algorithms process considerably smaller inputs, which makes them significantly faster, and therefore more suitable for real-time learning.

Dependency on the ordering of inputs leads us to the question of algorithm's correctness. In general, we declare an algorithm correct, if it returns a correct result in finite time for any input. The definition of a *"correct result"* will in our case have to be quite loose for the following reasons:

1. Online algorithms like $3SG$ do not have an access to specific future or past inputs.

2. Even if they did, we could not guarantee the consistency of new input with the older ones, because of sensoric noise and possible action failures.

As a correct result, we will therefore consider such action model that *covers the new example* and at the same time holds that *if some new conflict with an old example occurred*, it has to be caused either by the *elimination of conflict* with new example, or by *covering* it.

Theorem 1 (Correctness)**.** *3SG algorithm without forgetting (memoryLength = ∞) is correct according to above-mentioned definition of correct result.*

Proof. In order to declare correctness, we must prove the following three statements:

i. *3SG algorithm will terminate for any input.* This is a direct result of algorithm's structure - first two cycles (lines 0-18 in Fig. 8.3) always iterate over a finite set of observed fluents (either o, or o'), while not modifying this set at all. Remaining two cycles (lines 21-28) then iterate over the $\mathcal{EF}$ set, while they can only delete elements from it (they do not add anything new). Note that this condition would be satisfied even if the forgetting was allowed.

ii. *Resulting action model always covers the new example.* According to aforementioned definition of covering, for every $f \in \Delta(o, o')$ we need to have an a^f atom in $\mathcal{EF}$. This is taken care of by the first part of the algorithm that adds a^f into $\mathcal{EF}$ (line number 2). As long as forgetting is forbidden, no other part of the algorithm can delete this atom.

iii. *If a new conflict with older example occurred, it is caused either by the elimination of conflict with new example, or by covering it.* This condition is a trivial result of the fact that (without forgetting) the $\mathcal{EF}$ is modified only by addition of a^f on line 2 (in order to cover new example), or by addition of new conditions a^f_c on line 16 if the new example is negative w.r.t. corresponding a^f. According to definition of conflict (above), addition of such $a^f_{\overline{c}}$ conditions that $\overline{c} \in o$, removes the conflict with our example (because $\overline{c}$ and c cannot be in o at the same time - see note 8.1.1).

$\square$

Theorem 2 (Complexity)**.** *Let n denote the cardinality of input $\mathcal{EF}$, and m the cardinality of the bigger of two input observations o, o'. Worst-case time complexity of 3SG algorithm is polynomial in the size of the input. More precisely, if the $\mathcal{EF}$ is implemented by a B-Tree, the complexity is $\mathcal{O}\left((m^2 + n^2)\log n\right)$.*

Proof. First of all, we will calculate the complexity of all four *Foreach* cycles within our algorithm. Once again, we will be referring to the pseudocode from figure 8.3. Operations, that are not mentioned in this proof, run in constant time and can be ignored.

i. In the first cycle, we iterate over the set $\Delta(o, o')$ (line 0) which has m elements in the worst case. In every iteration, we need to find out if $a^f \in \mathcal{EF}$ (lines 2 and 3). Finding an element in a B-tree has a time complexity of $\mathcal{O}(\log n)$ [Bayer-McCreight, 1972]. This *if*-condition gives us two possible outcomes, while computationally more complex is the case where $a^f \in \mathcal{EF}$. There we have a nested *Foreach* loop (line 5), which again iterates over at most m elements, and for each of them, it needs to check the existence of 2 different elements in $\mathcal{EF}$ (lines 6 and 7). These checks give us the worst-case complexity of $\mathcal{O}(2 \cdot \log n)$. Overall complexity of the first cycle is therefore in the worst case: $\mathcal{O}\big(m \cdot m \cdot (2 \cdot \log n)\big) = \mathcal{O}(m^2 \cdot \log n)$.

ii. In the second cycle, we iterate over the set of at most m elements (line 12) and in every iteration we check for the existence of a certain element in $\mathcal{EF}$ (line 13) with the complexity of $\mathcal{O}(\log n)$. The nested loop on line 16 then, again at most m times, checks for the existence of element in $\mathcal{EF}$ and adds it, if it is not already there. Since the complexity of insertion into a B-Tree is also $\mathcal{O}(\log n)$ [Bayer-McCreight, 1972], this whole cycle gives us the worst-case complexity of: $\mathcal{O}\big(m \cdot ((\log n) + m \cdot 2(\log n))\big) = \mathcal{O}(m^2 \cdot \log n)$

iii. In the third cycle, our algorithm deletes (forgets) some of the old atoms from $\mathcal{EF}$. In order to know which atoms to delete, we need to test all n of them and, in the worst case, delete all of them (line 22). Since the deletion from B-Tree runs also in $\mathcal{O}(\log n)$ [Bayer-McCreight, 1972], this cycle has overall complexity of $\mathcal{O}(n \cdot \log n)$.

iv. Last cycle iterates once again over all n elements of $\mathcal{EF}$ (line 24), but here we need to check the non-existence of relevant a^f_c atoms (line 25) for potentially all of them. This requires another iteration over all the n elements of $\mathcal{EF}$. Together with the deletion of a^f we get: $\mathcal{O}(n \cdot n \cdot \log n) = \mathcal{O}(n^2 \cdot \log n)$

Now, if we combine the complexities of all four cycles (running one after another), we get the overall complexity of $3SG$ algorithm:

$$\begin{aligned} & \mathcal{O}(m^2 \cdot \log n) + \mathcal{O}(m^2 \cdot \log n) + \mathcal{O}(n \cdot \log n) + \mathcal{O}(n^2 \cdot \log n) \\ = \ & \mathcal{O}\left((m^2 + m^2 + n + n^2)\log n\right) \\ = \ & \mathcal{O}\left((m^2 + n^2)\log n\right) \end{aligned}$$

□

Note 8.2.1. Implementing the $\mathcal{EF}$ as a B-Tree is not entirely optimal for our problem. Notice that in the last cycle we needed to iterate over the whole $\mathcal{EF}$ set in order to find out what needs to be deleted. If we had our $\mathcal{EF}$ structured better, we could accelerate this process. The first thing that comes to mind is remembering the pointers between a^f atoms and all of their a^f_c conditions.

In the appendix A, we take a closer look at some of the miscellaneous properties of $3SG$ algorithm, which may be less important, but can provide a deeper insight into its behaviour.

8.3 Performance Metrics

As we mentioned before, the action model serves as a representation of domain dynamics. In other words, it should accurately describe the changes happening within our domain and, consequently, we should be able to use this model to *predict* those changes. To evaluate the quality of induced action model, we can therefore try to analyse its ability to predict the changes happening in examples from our test set.

Definition 8. We say that $\mathcal{EF}$ **predicts** a fluent f in an example (o, a, o') if $a^f \in \mathcal{EF} \wedge (\nexists c : a^f_c \in \mathcal{EF} \wedge c \notin o)$.

The quality evaluation in learning problems like this is typically done by computing the *precision* and *recall* of the model, and their weighted harmonic mean called the **F-measure** [Lewis, 1994, Makhoul et al., 1999, Rijsbergen, 1979].

In our case, the **precision** of the action model is defined as the ratio of *predicted* fluent changes that were *consistent with the observations* (correct predictions / all predictions). **Recall**, on the other hand, is the ratio of observed fluent *changes* that we were *able to predict* (predicted changes / all changes). In other words:

- Let $A(f)$ be the number of observed examples (o, a, o') from the test set, where $f \in \Delta(o, o') \wedge$ $\mathcal{EF}$ *predicts* f (correct predictions).
- Let $B(f)$ be the number of observed examples from the test set, where $f \in \Delta(o, o')$, but $\mathcal{EF}$ *does not predict* f (unpredicted changes).
- And finally, let $C(f)$ be the number of observed examples from the test set, where $\mathcal{EF}$ *predicts* f, but $\overline{f} \in o'$ (incorrect predictions).

The precision and recall values with respect to a certain fluent f are then defined as:

$$P(f) = \frac{A(f)}{A(f) + C(f)} \qquad R(f) = \frac{A(f)}{A(f) + B(f)}$$

Now, if we compute the arithmetic mean of precision and recall w.r.t. every $f \in \mathcal{F}$ and denote them by P and R, the F-measure of our action model $\mathcal{EF}$ can be defined as:

$$F_\beta = (1 + \beta) \cdot \frac{P \cdot R}{\beta^2 \cdot P + R}$$

The non-negative weight constant β here allows us to assign higher significance either to precision, or to recall. In our case, we considered the precision to be slightly more important[2], so we assigned β the value of 0.5.

Notice that this performance metrics combines both precision and recall in such a way, that it strongly penalizes the models with either of them too

[2]Most commonly used values for β are 0.5 if the precision is more important, 2 if the recall is more important, or 1 if they are considered even.

low. This ensures that the models need to be both accurate and descriptive enough, in order to be considered good.

In chapter 9, we will compute and present the F-measure of our action model over the course of the learning process in various experimental settings.

8.3.1 Worst Case Analysis

As outlined in section 2.4, the fact that $3SG$ is an online algorithm means that its performance depends on the ordering of the input examples. In other words, different orderings of the training set may cause the F-measure of induced action model to yield different values.

Online algorithms are sometimes evaluated by comparing their performance to that of the *optimal offline algorithm* (which is an *unrealizable* algorithm with full knowledge of the future). This comparison, called *competitive analysis* [Borodin-El-Yaniv, 1998], is considered a worst-case analysis, since we use the worst possible ordering of the inputs for performance comparison.

Despite being an interesting framework for mathematical algorithmic analysis, competitive analysis is not uniformly worthwhile over all possible application areas simply because it is too pessimistic, assuming a malicious adversary that chooses the worst input ordering by which to measure algorithm's performance [Borodin-El-Yaniv, 1998].

In appendix B, we define the worst case for $3SG$ algorithm and discuss the likelihood of this case.

8.4 Additional Ideas

In the *simplification* part, the algorithm uses three constant parameters *memoryLength*, *minP* and *minEx* to determine what should be *forgotten*.

It might be practical to set some of those parameters dynamically. In general, they are used to reduce the computation time by shrinking the algorithm's input (by deleting the elements from $\mathcal{EF}$).

The *memoryLength* parameter alone may be used to adjust the trade-off between the computation speed and the learning rate. If the *memoryLength* is higher, we have a potential to induce the high-quality action model using fewer examples. On the other hand, if the *memoryLength* is too small, we may even end up being completely unable to learn some of the effects.

Therefore, it seems reasonable to have our *memoryLength* as high as our hardware allows in a given domain. An effective implementation of $3SG$ algorithm should adjust this parameter dynamically by:

- computing the *current average time* available between two consecutive examples (denoted by *avg*)
- and *increasing* the *memoryLength* by a small value while the actual running time of $3SG$ is safely below *avg*,
- or *decreasing* it whenever it gets too close to *avg*.

However, during the experiments discussed in the next chapter, we do not adjust the parameters at all. Our experimental implementation is quite simplistic, and only uses static parameters.

Chapter 9

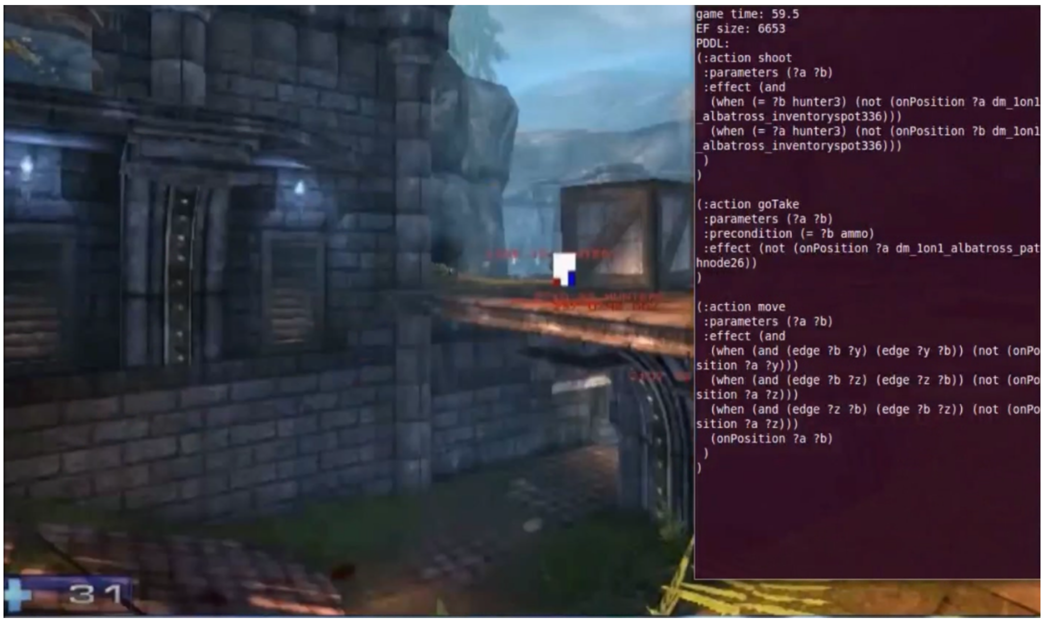
Experiments with $3SG$ Algorithm

According to the comparison table in figure 4.2, $3SG$ algorithm is compatible with all four discussed domains. In order to test it in practice, we have conducted two experiments in the most difficult of them. The first domain, Unreal Tournament 2004, was quite demanding in terms of computation times and was able to examine the performance of $3SG$ in complex environments with many actions per minute. Second experiment was conducted in a real-world environment with considerable amount of sensoric noise and action failures - robotic platform SyRoTek.

9.1 Unreal Tournament 2004

The complexity of our algorithm (theorem 2) suggests that the *size* of our *action model representation* will play an important role in the overall performance. Therefore, in the first part of this experiment, we will measure the size of $\mathcal{EF}$. In addition to that, we will be interested in actual running *speed* of $3SG$ in practice, and of course also the *quality* of induced *models.*

As a domain for our first experiment, we have chosen the action FPS[1] computer game *Unreal Tournament 2004*, which was accessed via Java framework Pogamut [Gemrot et al., 2009]. During the game, our agent was learning the *effects* and *preconditions* of all the observed actions like *shoot, move, changeWeapon, takeItem, etc.* These were executed either by him, or by other agents (opponents).



```
game time: 59.5
EF size: 6653
PDDL:
(:action shoot
 :parameters (?a ?b)
 :effect (and
  (when (= ?b hunter3) (not (onPosition ?a dm_1on1
_albatross_inventoryspot336)))
  (when (= ?a hunter3) (not (onPosition ?b dm_1on1
_albatross_inventoryspot336)))
 )
)

(:action goTake
 :parameters (?a ?b)
 :precondition (= ?b ammo)
 :effect (not (onPosition ?a dm_1on1_albatross_pat
hnode26))
)

(:action move
 :parameters (?a ?b)
 :effect (and
  (when (and (edge ?b ?y) (edge ?y ?b)) (not (onPo
sition ?a ?y)))
  (when (and (edge ?b ?z) (edge ?z ?b)) (not (onPo
sition ?a ?z)))
  (when (and (edge ?z ?b) (edge ?b ?z)) (not (onPo
sition ?a ?z)))
  (onPosition ?a ?b)
 )
)
```

Figure 9.1: In-game screenshot from the UT2004 experiment. In the right part of the screen, we can see a small part of the learned action model (converted to PDDL for better readability).

[1]FPS stands for *"First Person Shooter"* game genre. Players of the FPS game perceive their three-dimensional surroundings from their own viewpoint, while moving around, collecting items, and engaging in combat.

The agent called $3SG$ algorithm once after every observed action, which was approximately 7.432 times every second. The average size of his observations (o and o') was 70 fluents, which means that he processed approximately 140 fluents on every call. Following results were gathered during a 48.73 minute long game.

9.1.1 Size of the Action Model

During this experiment, the agent has observed and processed 21733 examples, which corresponded to 7632 time steps (multiple actions could be observed in a single time step). The size of $\mathcal{EF}$ determines the size of the entire action model, since the probability function $\mathcal{P}$ is defined implicitly by a single equation. Our constants were set to following values: $minP = 0.9$, $minEx = 3$, $memoryLength = 50$. This means that we were deleting (forgetting) all those elements from $\mathcal{EF}$ that were not supported by at least 3 examples, or if their probability was lower than 0.9 after 50 time steps (approximately 19.54 seconds). The chart in figure 9.2 depicts the size of $\mathcal{EF}$ during the whole 48.73 minute game. We can see that, thanks to forgetting, the size of our action model stays more or less on the constant level. If the forgetting was disabled, this size would grow uncontrollably.

9.1.2 Running Speed

In order to measure the actual running speed, we have created a log of consequent examples observed by an agent during the game, and only after that we repeatedly called $3SG$ with each of these recorded examples on input. The algorithm was implemented in Python and our action model was stored

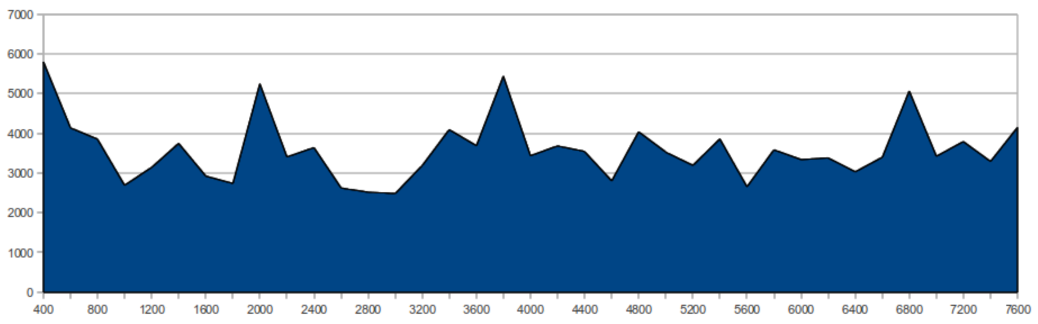


Figure 9.2: Development of $\mathcal{EF}$ size (y-axis) over time.

in MySQL database ($\mathcal{EF}$ implemented as a B-Tree). Experiment was conducted under a Linux OS, on a system with Intel(R) Core(TM) i5 3.33GHz processor and 4GB RAM. Processing all 21733 examples recorded during 48.73 minute game took only 12 minutes and 10.138 seconds. This means that $3SG$ algorithm can be used for action learning in real time in domains at least as complex as UT2004 game.

9.1.3 Quality of Induced Models & Learning Curve

Let us now take a closer look at the learning process and the quality of induced action models, using the F-measure (defined in chapter 8) as our performance metrics. During the quality analysis, our 21733 examples were divided into disjoint *training set* and *test set*, while the later contained approximately 15% of all the examples (chosen randomly, using a uniform distribution). In the learning process, we only used the examples from the training set. After every time step, we computed the F-measure of our model, based on its ability to predict the changes occurring in the test set.

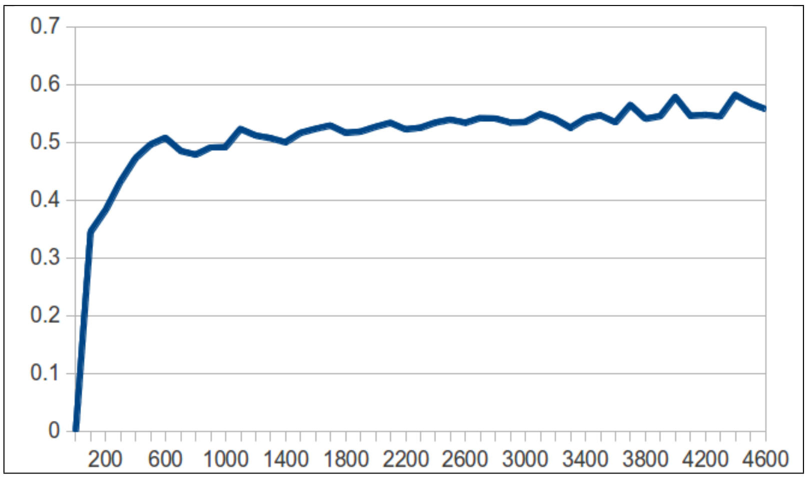


Figure 9.3: Development of the F-measure (y-axis) over time.

Learning curve in figure 9.3 depicts the development of the quality of our action model (expressed by the F-measure) during the experiment. Notice the occasional slight decreases of the F-measure value. They can be caused by the online nature of $3SG$ algorithm. However, it is apparent that the quality is improving in the long term. Another thing to note is the rapid quality increase during the first minutes of the experiment. See figure 9.4 for more detailed picture of the first 500 time steps. It looks like the learning process is fastest during the first 80 time steps (less than a minute), but continues during the whole experiment.

9.1.4 Resulting Models

Let us now take a closer look at our resulting action model. To enhance the readability, we have translated the $\mathcal{EF}$ into planning language PDDL. This translation itself is quite straightforward and uninteresting. The important thing about it is that, during the translation, all the atoms from $\mathcal{EF}$ whose probability was lower than $minP$ were ignored. Also, to simplify our PDDL

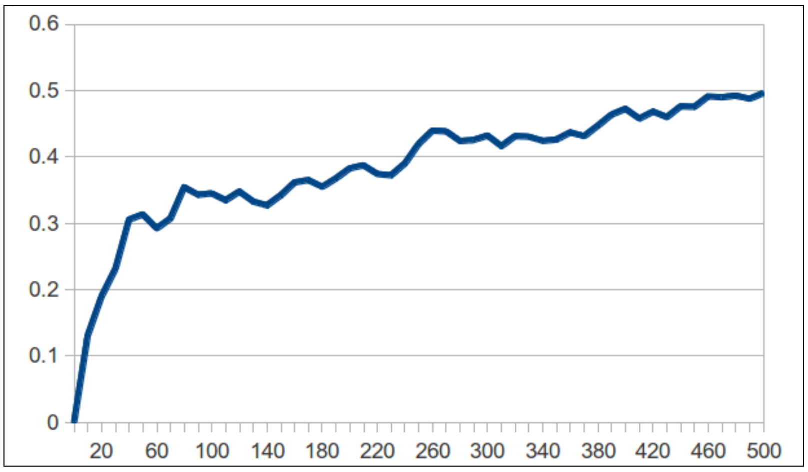


Figure 9.4: Development of the F-measure (y-axis) during the first 500 time steps.

representation even further, we have:

- merged the effect groups, where it was possible by introduction of variables,
- and if all the effects of an action had one common condition, we converted this condition into a precondition of the whole action.

This way we acquired a simplified, human-readable PDDL representation of our action model.

First learned action depicted in figure 9.5 is called $changeWeapon(A, B, C)$. This one has been learned exactly according to our intuitions: *An agent A can change the equipped weapon from B to C only if $B \neq C$. When he does this, C will be equipped, and B will no longer be equipped.* Second action, $move(A, B)$, contains more learned effects (to keep things brief, we present four of them). First two effects are not surprising: After the $move(A, B)$ action, *agent A will be on position B, and will not be on any other adjacent*

```
(:action changeWeapon
        :parameters (?a ?b ?c)
        :precondition (and (differentWeapon ?b ?c) (differentWeapon ?c ?b))
        :effect (and
                (equippedWeapon ?a ?c)
                (not (equippedWeapon ?a ?b))
        )
)
(:action move
        :parameters (?a ?b)
        :effect (and
                (onPosition ?a ?b)
                (when (and (edge ?b ?z) (edge ?z ?b))
                                (not (onPosition ?a ?z)))
                (when (= ?b playerstart25) (dead ?a))
                (when (= ?b playerstart20) (health ?a maximum))
                ... etc.
        )
```

Figure 9.5: Example of learned actions (converted to PDDL action language).

position Z. Last two effects are less obvious but they have been learned correctly, based on observed examples. Specifically, third effect of *move* action says that *if agent A moves to a position called playerstart*25*, he will die.* The agent learned this, because almost every time he arrived there, something killed him. Similarly, he learned that moving to a position *playerstart*20 causes his health to get to maximum value. This was probably learned because there was a Medkit item at that position. Interesting fact is that first four (intuitive) effects were learned very quickly (they were in $\mathcal{EF}$ during the first minute of gameplay). The less obvious effects took our agent more time to learn (2.5 and 16 minutes respectively).

9.2 Robotic Platform SyRoTek

Our second experiment was conducted using the real-world robots of the SyRoTek platform [Kulich et al., 2010]. SyRoTek system allows its users to remotely control multiple mobile robots with a variety of sensors in the environment with customizable obstacles.

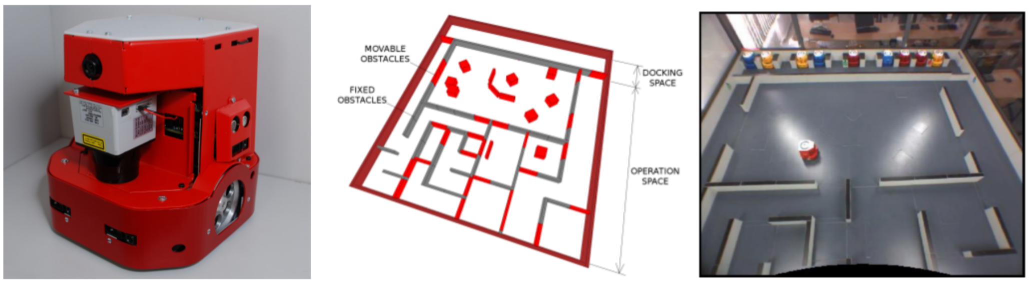


Figure 9.6: SyRoTek robot, map of the arena, and a photo taken during our experiment. The size of the arena is 350x380cm.

During this experiment we only used a single robot with a laser rangefinder sensor and a camera-based localization system. All the other sensors were ignored. Also, the agent could only execute three kinds of actions: *move forward*, *turn left*, or *turn right*. He chose his actions randomly and always executed only one action at a time. Since the agent acted in a real world environment, we experienced a considerable amount of sensoric noise and action failures. On the other hand, this experiment was less computationally intensive than the previous one.

Continuous data extracted from the sensors were discretized into fluents. Agent perceived exactly 56 fluents per observation, and called $3SG$ algorithm after every action - approximately once every 8 seconds. The experiment was terminated after 1250 examples (which took approximately 166 minutes)[2]. Our constants were set to following values: $minP = 0.65, minEx = 3, memoryLength = 150$. In other words, we deleted an element of $\mathcal{EF}$ if it was not supported by at least 3 examples, or if its probability was lower than 0.65 after 150 time steps (approximately 20 minutes).

9.2.1 Quality of Induced Models & Learning Curve

Due to relatively small number of examples and a long time between individual actions, the algorithm's running speed and the size of $\mathcal{EF}$ were not an issue. More interesting was the actual development of quality of our learned model. 1250 examples were first divided into a *training set* consisting of 1000 examples and a *test set* of 250 randomly chosen examples (with a uniform distribution).

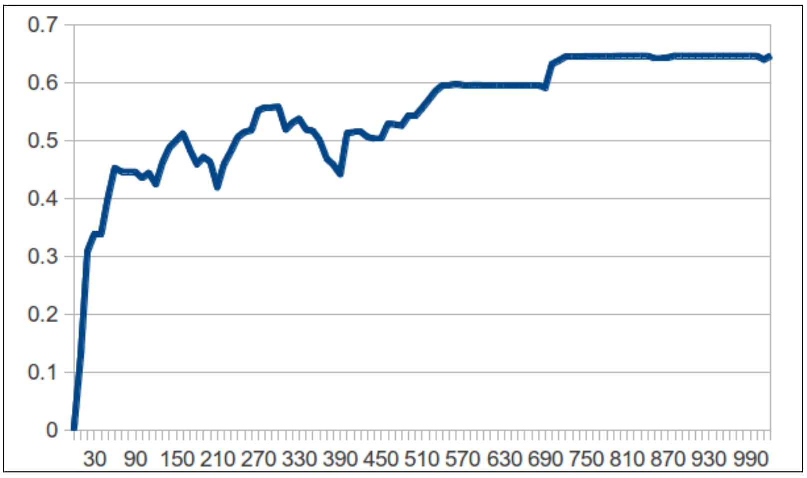


Figure 9.7: Development of the F-measure (y-axis) over time.

As our quality metrics, we have once again chosen the F-measure (as defined

[2]Unlike in previous experiment, our agent here could only observe one action at a time. This means that, in this case, the number of time steps is the same as the number of examples.

in section 8.3). Learning curve (figure 9.7) is similar to the one from previous experiment. We can see the rapid quality increase during the first 80 time steps. There are also several slight decreases (caused by the online nature of $3SG$ algorithm), but overall long-term trend is increasing. An interesting thing to notice here are the periods when the quality of our model was almost constant (for example, during time steps 550-690). During these periods, the agent was stuck between the obstacles (recall that his actions were random, so this could happen quite easily).

9.2.2 Resulting Models

Once again, we have translated our induced action model into PDDL for better readability.

What you see in figure 9.8, is a fragment of the action model from the middle of learning process. First depicted action, $move(A, B)$, simply tries to move the agent forward (from position A to B). We can see that our agent was able to learn its first two effects just as expected: *Moving from A to B causes us to be on position B, and no longer on A.* Third and fourth effects were temporary. They were added by $3SG$ at a certain time step and disproved and deleted later. Notice that fourth effect ($orientation(west)$) is a result of the sensoric noise or the action failure (since the *move* action should not typically change agent's orientation).

Second action, called $turnLeftFrom(A)$, also had two intuitive and a couple of temporary effects. First effect simply says that *after turning left from direction A, we will no longer be facing A.* Second effect means that *turning left from A causes us to face Y, if A is in clockwise direction from Y.*

```
(:action move
        :parameters (?a ?b)
        :effect (and
                (position ?b)
                (not (position ?a))
                (blocked 27x41)
                (orientation west)
        )
)
(:action turnLeftFrom
        :parameters (?a)
        :effect (and
                (not (orientation ?a))
                (when (clockwise ?y ?a) (orientation ?y))
                (when (= ?a west) (front me 27x41))
                (when (= ?a west) (not (back me 28x38)))
                ... etc.
        )
)
```

Figure 9.8: Example of learned actions from the middle of learning process (converted to PDDL action language).

We should point out the fact that we used quite a long *memoryLength* (150 time steps is relatively long period within our 1250-step experiment). This means that temporary effects are kept in the action model longer (they are deleted later, even if they have low probability).

9.3 Summary

For our experiments, we have selected the most demanding domains (according to figure 4.2): Unreal Tournament 2004 and SyRoTek platform.

Only one of the alternative action learning methods discussed in this thesis is usable in both of them - the *perceptron* based method by Mourao et al. (section 3.4). However, according to a comparison table from figure 3.4, it has a certain disadvantages. Unlike $3SG$ algorithm, the perceptron method can not be used to learn the action's preconditions or the conditional effects.

Our experiment in a computationally intensive domain of Unreal Tournament 2004 has shown that, even despite considerable short-term fluctuations, the size of the effect formula $\mathcal{EF}$ remains approximately the same in the long run. This means that, thanks to *forgetting*, the size of the algorithm's input (and therefore its running time) does not grow uncontrollably.

The actual running time appears to be sufficient for relatively complex domains like Unreal Tournament 2004, even with quite inefficient implementation (as we mentioned before, we used the sub-optimal structure of B-Tree to represent $\mathcal{EF}$, and did not adjust the *memoryLength* parameter dynamically).

The sensoric noise and action failures in a real-world domain SyRoTek did not pose a significant problem. As we have seen, the learning rate in this domain was satisfactory. An interesting thing to notice in both domains are slight short-term decreases in action model quality, caused by the online nature of $3SG$.

Chapter 10

Reactive ASP Method

The second action learning technique introduced in this work is not as versatile in terms of domain compatibility as $3SG$ algorithm (comparison in chapter 4), but it has one extra advantage - it is purely declarative.

The method is based on the logic programming paradigm called ASP (Answer Set Programming) [Gelfond-Lifschitz, 1988, Gelfond-Lifschitz, 1991, Baral, 2003]. The idea to use the ASP for the purposes of action learning has already been published in [Balduccini, 2007] and analysed in section 3.3 of this thesis, together with the overview of the *A-Prolog* representation language.

However, in reaction to recent introduction of so-called *Reactive ASP* and implementation of effective tools [Gebser et al., 2011], we propose a different approach, and show how using the Reactive ASP together with more compact knowledge encoding can provide some advantages in certain situations [Čertický, 2012c].

10.1 Reactive ASP

Answer Set Programming has lately become a popular declarative problem solving paradigm, with growing number of applications, among others also in the field of reasoning about actions.

Semantics of ASP enables us to elegantly deal with incompleteness of knowledge and makes the representation of action's properties easy, due to non-monotonic character of default negation operator [Lifschitz, 2002]. The representation language associated with ASP, called A-Prolog, is described in detail in section 3.3. Please recall that A-Prolog deals with *logic programs*, which are simply sets of so-called *rules*. There are two special kinds of rules: *facts* and *constraints*. Every logic program has a corresponding (possibly empty) finite set of *stable models* (sets of A-Prolog literals).

Since we are dealing with dynamic systems, we can take advantage of a more advanced structure called *incremental logic program*. An incremental logic program is a triple $(B, P[t], Q[t])$ of logic programs, with a single parameter t ranging over natural numbers [Gebser et al., 2008]. While B only contains **static** knowledge, parametrized $P[t]$ constitutes a **cumulative** part of our knowledge. $Q[t]$ is so-called **volatile** part, but it is not a part of our method. In this technique, t will always correspond to the most recent time step, and $P[t]$ will describe how the newest example affects our current action model.

Furthermore, the authors of [Gebser et al., 2011] augment the concept of incremental logic programming with asynchronous information, refining the statically available knowledge, and indroducing the concept of *reactive ASP*. They also released the first **reactive ASP solver** *oClingo* which we use in this method.

The *oClingo* solver is a server application, which listens on a specified port and waits for new knowledge. This knowledge is sent to *oClingo* by a *controller* application in a form of cumulative logic program $P[t]$ which, in our case, always represents the latest example.

10.2 Learning with Reactive ASP

In chapter 6, we have explained how an action model can be expressed by a set of A-Prolog literals $\mathcal{AL}$. We also know that every ASP logic program has a set of corresponding *stable models*.

The whole method is based on the fact that we can write an ASP logic program that would have a number of stable models, each of which contains the $\mathcal{AL}$-representation of exactly one action model consistent with all our previous examples (or at least with most of them). Throughout this chapter, we will refer to this logic program as *learner*.

Note 10.2.1. At any given time t, there might exist several action models that are consistent with all the previous examples $(o, a, o')_1$, $(o, a, o')_2$ $\ldots$, $(o, a, o')_t$. It is easy to see that with increasing t, the number of such action models will decrease. First, we will describe how to learn models like this, but further in this chapter, we will also deal with action models that are only consistent with the majority of previous examples (not all of them). This will help us deal with non-determinism.

Note 10.2.2. Also note that any action model represented by $\mathcal{AL}$ is *"complete"* in a sense that it either *covers* an example (o, a, o'), or is *in conflict* with it (see the definitions in 7.2).

Let us now describe our *learner*, which is basically a short incremental logic program $(B, P[t], \emptyset)$ with initially large (but gradually decreasing) number of *stable models*, each *corresponding to* a single action model consistent with previous examples.

At time step 1, the online ASP solver *oClingo* computes the first stable model and stores it in memory (along with all the partial computations it has done so far). At every successive time step, we confront this model with a new example by encoding this example as a cumulative logic program $P[t]$ and sending it to *oClingo* server.

10.2.1 Static Knowledge B: Generating Stable Models

Stable models corresponding to all the possible action models are **generated** by the **static** part of our learner program (logic program B). It consists of the following collection of rules:

```
% Effect generator and axioms:
causes(A,F) ← not causes(A,¬F), not keeps(A,F).
causes(A,¬F) ← not causes(A,F), not keeps(A,F).
keeps(A,F) ← not causes(A,F), not causes(A,¬F).
← causes(A,F), causes(A,¬F).
← causes(A,F), keeps(A,F).
keeps(A,F) ← keeps(A,¬F).
keeps(A,¬F) ← keeps(A,F).

% Precondition generator and axioms:
pre(A,F) ← fluent(F), not ¬pre(A,F).
¬pre(A,F) ← fluent(F), not pre(A,F).
← pre(A,F), ¬pre(A,F).
← pre(A,F), pre(A,¬F).
```

Figure 10.1: Static (time-independent) part B of our *learner* $(B, P[t], \emptyset)$.

Note 10.2.3. The syntax of *oClingo* input allows us to use the *variables.* Variables in ASP are usually denoted by uppercase letters (in our case A and F). Also keep in mind that the logic programs in this chapter are simplified to improve the readability and save space. The exact ready-to-use ASP solver compatible encodings are downloadable from [Link 1, 2012].

In the first part of B, we have three **choice rules** that **generate stable models** where action A either causes a fluent F, causes $\neg F$, or keeps F. Next two **constraints filter out** the answer sets corresponding to **impossible models** - where A causes F and $\neg F$ at the same time, or where A both keeps and changes the value of F. Last two rules merely express the equivalence between two possible notations of A keeping F.

The second part is very similar. Here we have two choice rules generating stable models where A either has, or does not have a precondition F. Constraints here eliminate those models, where A *has* and does *not have* a precondition F at the same time, or where it has both F and $\neg F$ as its preconditions.

10.2.2 Cumulative $P[t]$: Stable Model Elimination

Second part of our learner, a time-aware cumulative program $P[t]$, is sent to *oClingo* after every example $(o, a, o')_t$. $P[t]$ consists of two parts with two different functions:

1. it ensures that we **eliminate** all the stable models representing action models inconsistent with the new example $(o, a, o')_t$ (figure 10.2),
2. and it **encodes** the actual example $(o, a, o')_t$ containing the newest

observation and executed action (figure 10.3).

```
#external obs/2.
#external exe/2.
← obs(F,t), exe(A,t), causes(A,¬F).
← obs(¬F,t), obs(F,t−1), exe(A,t), keeps(A,F).
← exe(A,t), obs(¬F,t−1), pre(A,F).
```

Figure 10.2: First part of the cumulative (time-aware) component $P[t]$ of our learner logic program $(B, P[t], \emptyset)$.

In figure 10.2, first two statements (begining with #) are just a technicality. They merely instruct *oClingo* that it should accept two kinds of binary literals from the controller application[1]: fluent observations `obs` and executed actions `exe`.

Remaining **three constraints take care of actual learning**, by **eliminating stable models** that are **in conflict with** observations from the latest example. First constraint says that if F was observed after executing A, then A cannot cause $\neg F$. The second constraint tells us that if F changed value after executing A, then A does not keep the value of F. And the last one means that if the action A was executed when $\neg F$ held, F cannot be a precondition of A.

At every time step, these constraints are added to our knowledge with parameter t substituted by a current time step number. Next, we need to add the actual latest observation. These observations are sets of `obs` and `exe` literals. See the example of observation that is sent to *oClingo* in figure 10.3.

[1]Telling *oClingo* what to accept is not necessary if we use the new `--import=all` parameter. This option was implemented only after we designed our learner logic program.

```
#step 9.
exe(move(b1,b2,table),9).
obs(on(b1,table),9).
obs(¬on(b1,b2),9).
obs(on(b2,table),9).
obs(¬on(b2,b1),9).
#endstep.
```

Figure 10.3: An observation from Blocks World sent to *oClingo* at time step number 9. It describes the configuration of two blocks $b1$ and $b2$ on the table after we moved $b1$ from $b2$ to the *table*.

We say that a constraint *"fires"* in a stable model, if its body holds there. In that case, this stable model becomes *"illegal"*, and is thrown away. Now recall that in time step 1, *oClingo* generated the first possible model and stored it in memory. An observation like the one above can cause some of our constraints to fire in it and eliminate it. For example, if our stable model (action model) contained literal "`causes(`$move(b1, b2, table)$`,`$on(b1, b2)$`)`", the first constraint would fire.

If that happens, *oClingo* simply **computes** the **new stable model** that is not in conflict with any of our previously added constraints. This is how we update our knowledge, so that we always have an action model consistent with previous observations at our disposal. Note that each observation potentially reduces the number of possible stable models of our logic program $(B, P[t], \emptyset)$, thus making our knowledge more precise. After a sufficient number of examples, $(B, P[t], \emptyset)$ will have only one possible stable model remaining, which will represent the correct action model.

10.3 Noise and Non-determinism

The problem arises in the presence of sensoric noise or action failures, since the noisy observations or failed actions could eliminate the correct action model. This could eventually leave us with an empty set of possible models. However, we propose a workaround that might overcome this issue.

The problem with non-determinism is that we cannot afford to eliminate the action model after the first negative example. We need to have some kind of error tolerance. For that reason, we should modify the cumulative part of our program $P[t]$, so that our constraints fire only after a certain number of negative examples (figure 10.4).

```
negExCauses(A,F,C+1) ←
          negExCauses(A,F,C), obs(¬F,t), exe(A,t).


negExKeeps(A,F,C+1) ←
          negExKeeps(A,F,C), obs(¬F,t),
          obs(F,t-1), exe(A,t).


negExPre(A,F,C+1) ←
          negExPre(A,F,C), exe(A,t), obs(¬F,t-1).


← causes(A,F), negExCauses(A,F,C), C > 5.
← keeps(A,F), negExKeeps(A,F,C), C > 5.
← pre(A,F), negExPre(A,F,C), C > 5.
```

Figure 10.4: Modified $P[t]$ deals with sensoric noise, by introducing error tolerance.

Here we can see that our observations do not directly appear in the bodies of constraints. Instead, they are capable of increasing the negative example count (which is kept in the variable `C` of `negExCauses`, `negExKeeps` and `negExPre` literals). Constraints then fire when the number of negative examples is higher than some constant value of *error tolerance threshold* (in this case set to 5).

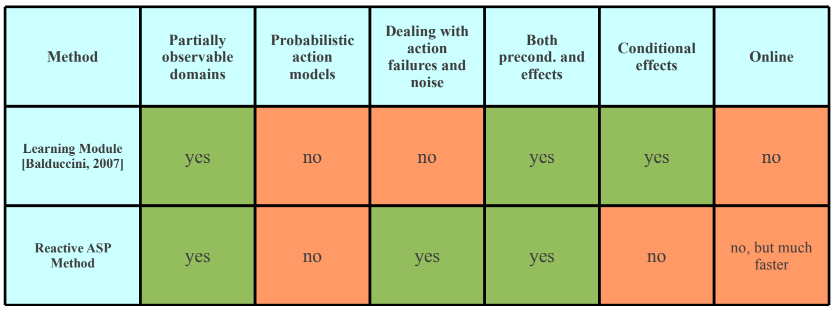
Note 10.3.1. The *error tolerance threshold* is a numeric constant 5 here to keep things simple, but we can easily imagine better, dynamically computed threshold values. We could, for example, set this threshold to a value of 30% of the *positive+negative* example count. This way, our constraint would fire only if more than 30% of all the relevant examples were negative. See the appendix C for the $P[t]$ program with dynamically computes the error tolerance threshold.

10.4 Similarities and Differences

Since the Answer Set Programming has already been used for action learning by Balduccini (see section 3.3 or [Balduccini, 2007]), we are obliged to provide an extensive comparison of these two methods. In this section, we will enumerate and discuss their similarities and differences.

10.4.1 Dealing with Incomplete Knowledge

From the viewpoint of domain compatibility, both methods share the ability to deal with the incompleteness of knowledge. Naturally, the absence of complete observations may slow down the learning process in both cases.



Method	Partially observable domains	Probabilistic action models	Dealing with action failures and noise	Both precond. and effects	Conditional effects	Online
Learning Module [Balduccini, 2007]	yes	no	no	yes	yes	no
Reactive ASP Method	yes	no	yes	yes	no	no, but much faster

Figure 10.5: Comparison of Balduccini's learning module and Reactive ASP method based on the properties from section 2.4.

By *slowing down* we understand that we might require more time steps to induce precise action models. However, the incompleteness of observations *does not* affect the computation times at individual time steps.

10.4.2 Action's Preconditions

Our method learns not only the effects, but also the preconditions of actions. Similarly, Balduccini's learning module supports the induction of preconditions. However, they take form of so-called *impossibility conditions* of action language $\mathcal{C}$.

10.4.3 Conditional Effects

Balduccini's learning module allows for direct induction of *conditional effects*, which is its greatest advantage over our method. Since we use the $\mathcal{AL}$ (chapter 6) as our action model representation structure, we are unable to express conditional effects at all. Having the conditional effects allows for more elegant representation of resulting action models. On the other hand, we must keep in mind that learning conditional effects is in general harder and more time-consuming problem.

10.4.4 Encoding of Action Models

An action model is, in case of Balduccini's learning module, encoded by a set of A-Prolog literals of the following types: $d\_law(L)$, $s\_law(L)$, $head(L, F)$, $action(L, A)$, and $prec(L, F)$, with L substituted by a name (unique constant identifier) of a $\mathcal{C}$-language [Guinchiglia-Lifschitz, 1998] law, A by an action instance, and F by a fluent. Notice that this way, we directly encode the syntactic form of individual $\mathcal{C}$-language laws into logic programs. See figure 10.6 for a simplistic example of this encoding.

Following $\mathcal{C}$-language law:
"*caused on(b1,table) after move(b1,b2,table), on(b1,b2), free(b1), free(table).*"
is in Balduccini's learning module translated into:

```
d_law(dynamicLaw25).
head(dynamicLaw25,on(b1,table)).
action(dynamicLaw25,move(b1,b2,table)).
prec(dynamicLaw25,on(b1,b2)).
prec(dynamicLaw25,free(b1)).
prec(dynamicLaw25,free(table)).
```

Figure 10.6: Example of $\mathcal{C}$-language $\rightarrow$ A-Prolog translation used in Balduccini's learning module.

Every time an example is added, the whole history is confronted with such a logic program. If the observation is not explained by it (see section 3.3), we add more facts to it (either creating new laws, or adding effect conditions to existing ones).

In our case, the action model encoding is more compact[2]. We do not translate an action model from any action language like $\mathcal{C}$, which allows us to omit the *d_law* and *s_law* predicates and L parameter. Instead we have chosen an abstract, semantics based, direct encoding of a domain dynamics, where every effect or precondition is represented by a single literal.

10.4.5 Extending the Techniques

The bottom line here is that our representation structure is simpler, but it has lower expressive power. However, our simple semantic based encoding makes it fairly easy to extend the learning by the ability to deal with noise and action failures. It is probable that Balduccini's learning module could also be similarly extended, but it would be far less straightforward process. His learning module consists of 14 rules describing the syntactic structure of $\mathcal{C}$-language representation of action model, rather than focusing directly on the semantics.

10.4.6 Speed

When processing a new example (o, a, o') using this method, we sometimes need to compute a new stable model of our learner logic program. When we do this, we need to make sure that this new model is not in conflict with any of the previous examples. Therefore we need to keep the record of all

[2]Note that the main reason we can afford more compact encoding is the fact that we do not support conditional effects.

the previous observations. According to the definition from chapter 7, this means that our method is *not online* (similarly to Balduccini's method).

However, thanks to the nature of *Reactive ASP* and *oClingo* solver, we do not need to search for new stable models after every observation. In addition to that, we can save a lot of time during the actual search for stable models because we are able to skip a big portion or computations related to older observations:

At each time step, the *oClingo* solver keeps everything that it has computed at previous steps in memory, and only adds new observation. If the current model is disproved, some revisions might be needed, but significant part of the computation has already been performed and results are stored in memory. This, together with relatively compact encodings allows us to learn action models relatively quickly.

Such optimization is quite important, since the computational complexity of some of the algorithms involved in the search for stable models is exponential in the size of the input [Simons et al., 2002]. Moreover, deciding whether an A-Prolog logic program has a stable model is an NP-complete problem.

10.5 Performance Metrics

Just like in case of $3SG$ algorithm, we will use the F-measure to evaluate the performance of the "Reactive ASP" method. However, since we are using a different structure for action model representation, we need to define the notion of *predicting* for action models represented by $\mathcal{AL}$.

Definition 9. We say that $\mathcal{AL}$ **predicts** a fluent f in an example (o, a, o')

if exactly one of the following statements holds:

- $causes(a, f) \in \mathcal{AL} \land f \in \Delta(o, o')$
- $keeps(a, f) \in \mathcal{AL} \land f \in o \land f \in o'$

Once again, the F-measure is defined using the notions of *precision* P and *recall* R, with the β parameter set to 0.5 (thus assigning a higher significance to precision over recall):

$$F_\beta = (1 + \beta) \cdot \frac{P \cdot R}{\beta^2 \cdot P + R}$$

Recall that P and R are computed as an arithmetic mean of $P(f)$ and $R(f)$ for every $f \in \mathcal{F}$, where:

$$P(f) = \frac{A(f)}{A(f) + C(f)} \qquad R(f) = \frac{A(f)}{A(f) + B(f)}$$

The values of $A(f)$, $B(f)$ and $C(f)$ represent the numbers of *correct change predictions*, *unpredicted changes* and *incorrect predictions* within the test set, and are defined similarly as in section 8.3:

- $A(f)$ is the number of observed examples (o, a, o') from the test set, where $f \in \Delta(o, o') \land$ $\mathcal{AL}$ *predicts* f (correct predictions).
- $B(f)$ is the number of observed examples from the test set, where $f \in \Delta(o, o')$, but $\mathcal{AL}$ *does not predict* f (unpredicted changes).
- $C(f)$ is the number of observed examples from the test set, where $\mathcal{AL}$ *predicts* f, but $\overline{f} \in o'$ (incorrect predictions).

In chapter 11, we will use the F-measure to evaluate the action model induced by the "Reactive ASP" method, and to plot the learning curves.

Chapter 11

Experiments with Reactive ASP Method

According to the comparison table in figure 4.2, this method is not compatible with the most difficult domain (Unreal Tournament 2004).

To support some of our claims from section 10.4, we will be studying the behaviour of our method in two variants of Blocks World, since they have also been used in Balduccini's experiments [Balduccini, 2007]. This allows us to responsibly compare these two approaches.

Specifically, the experiments will analyse the quality of induced action models and the running speed. There is no need to discuss the size of the action models during the learning process, since it stays unchanged as long as we represent them by $\mathcal{AL}$.

11.1 Fully Observable Blocks World

The most basic description of Blocks World domain, as described in example 2.1, only contains one kind of action: $move(B, P_1, P_2)$. In order to make the domain little more complex, and to be able to induce the effects and preconditions of more than one action type, we have split $move(B, P_1, P_2)$ into two different action types:

- $pickUp(B, P_1)$
- $putOn(B, P_2)$

An agent is able to observe the position of individual blocks (fluents like $on(b_1, b_2)$, $\neg on(b_1, table)$) and he can also see whether individual positions are free ($free(b_1)$) and what he is holding in his hand ($inHand(b_3)$, $\neg inHand(b_1)$).

In this variant, the agent can fully observe every world state. He has the information about the position of each block, he always knows which positions are free, and what he holds in his hand.

There is no sensoric noise, which means that the agent's observations are always precise. Also, the domain is deterministic, meaning that, under the same conditions, the actions will always have the same outcome.

Our instance of the Blocks World consisted of four blocks (b_1, b_2, b_3, b_4) and a *table*. Total number of action instances was therefore 50 ($|\mathcal{A}| = 50$) and a number of fluents was 70 ($|\mathcal{F}| = 70$).

Our training sets consisted of randomly generated examples, but were always *valid* in a sense that the generated sequence of actions/states never violated any of the rules of the domain. The actions were selected randomly (with uniform probability distribution), but only from the set of actions executable in a given state. We generated exactly one action per time step.

11.1.1 Running Speed

The experiments were conducted using the *oClingo* solver, version 3.0.92b, under a 64bit Linux OS with Intel(R) Core(TM) i5 3.33GHz CPU and 4GB RAM.

Processing the training set of *30 examples* with full observations took 2.538 seconds. In [Balduccini, 2007], we can find an experimental comparison of Balduccini's learning module and *Iaction* learning system [Otero-Varela, 2006]. Their experiment was also conducted with 4 blocks on a training set of 30 examples. *Iaction* system found a solution in 36 seconds on Pentium 4, 2.4GHz, while the results of Balduccini's module was fairly comparable with 14 seconds on somewhat faster computer (Pentium 4, 3.2Ghz).

Another two experiments, with a training set of *150* and *1000 examples* with full observations, took 17.9 and 299.4 seconds to finish.

11.1.2 Quality of Induced Models & Learning Curve

For the purpose of quality evaluation, we generated a valid *training set* of 900 examples and an independent valid *test set* of another 100 examples.

In the learning process, we only used the examples from the training set. To plot the learning curve (figure 11.1), we computed the F-measure of the current action model after every time step. As expected, the quality improves fastest during the first few time steps.

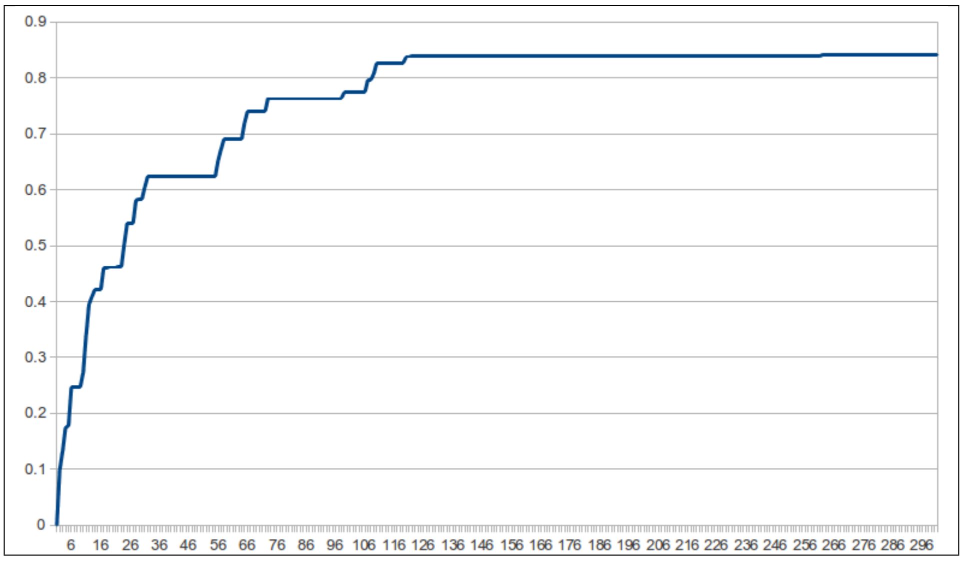


Figure 11.1: Development of the F-measure (y-axis) over first 300 time steps.

11.2 Partially Observable Blocks World

Partially observable Blocks World domain has basically the same rules as its fully observable variant, except the agent can only observe a subset of the fluents describing the world state.

The instance we used is the same as in previous case (consisting of four blocks and a table). We used the same randomly generated valid training sets with one action per time step, only this time, we deleted a randomly selected half of the fluents from every observation.

11.2.1 Running Speed

The running speed was analysed on the same machine as in case of previous experiment (section 11.1). Processing the training set of *30 examples* with partial observations took 2.741 seconds. Larger training sets, with *150* and

1000 examples, took 14.7 and 380.7 seconds. We can see that the running times in both variants of the Blocks World domain are quite similar.

11.2.2 Quality of Induced Models & Learning Curve

To measure the quality of induced action models, we once again generated a valid *training set* of 900 examples and an independent valid *test set* of another 100 examples. Only this time, all the examples in both sets contained partial observations (with randomly selected half of the fluents deleted).

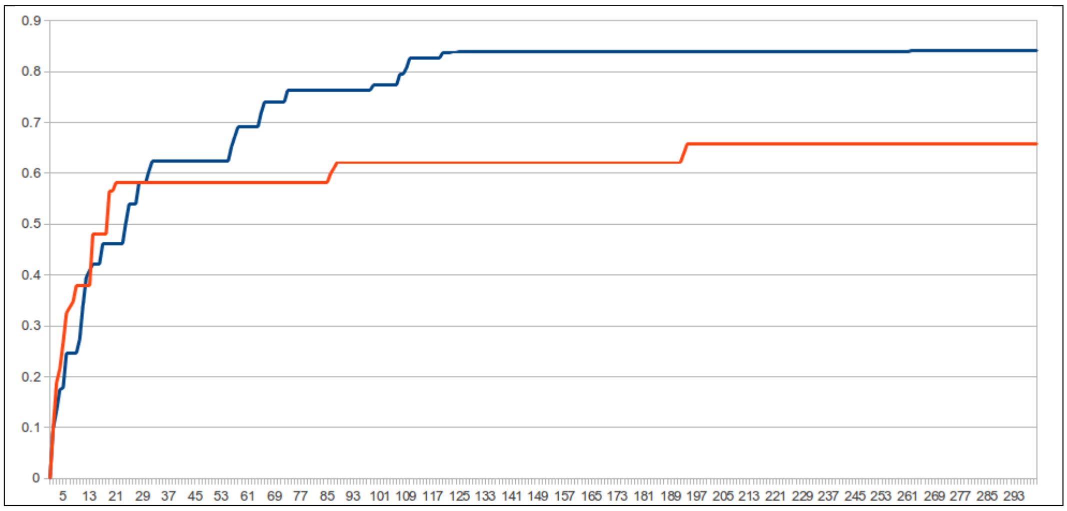


Figure 11.2: Development of the F-measure (y-axis) over first 300 time steps. Blue line represents a learning curve with full observations, and orange line with partial observations.

In figure 11.2, we compare the learning curves of the "Reactive ASP" method in both variants of the Blocks World. We can see that the partial observations make the learning process slower (orange curve). There are generally longer time periods between the quality increases.

11.3 Summary

Out of four domains discussed in this thesis, we have selected two variants of Blocks World for our experiments with *reactive ASP* method for action model learning. This allowed us to compare our approach to the most similar alternative by Balduccini (examined in section 3.3).

According to experiment results, even though both methods are offline and use an exponential algorithms, our approach seemed to be noticeably faster.

Our expectation about the learning rate being somewhat lower in the presence of incomplete observations has also been confirmed. Anyway, as expected, the actual running speed appears to be similar in both cases.

From a broader perspective, our other method ($3SG$ algorithm) appears to be a more suitable candidate for real-time learning in complex domains, due to its online nature.

However, the declarative character of "Reactive ASP" method can sometimes prove practical, and is probably the only reason to consider using it instead of $3SG$.

Chapter 12

Conclusion and Future Work

Based on relevant literature, we have identified a common collection of important properties or challenges that modern action learning methods try to overcome. Each of these methods is able to deal with a different subset of these subproblems, which makes it applicable in other situations and domains.

These properties have served not only as a basis for the evaluation and comparison of current techniques, but also as challenges that we were trying to overcome when designing our own methods.

The *online* algorithm $3SG$ is the first of two novel methods introduced in this thesis. It can be used to learn the *probabilistic* action models with *preconditions* and *conditional effects* in the presence of *incomplete observations*, *sensoric noise*, and *action failures*. In other words, it successfully deals with all the challenges discussed before.

Theoretical analysis shows that the time *complexity* of $3SG$ is polynomial in the size of the input, and also that it is formally *correct* under certain conditions.

Our experiments, conducted in two fundamentally different domains (action computer game *Unreal Tournament 2004* and a real-world robotic platform *SyRoTek*), demonstrate its usability in the computationally intensive scenarios, as well as in the presence of action failures and significant sensoric noise.

The second introduced method, based on the notion of *Reactive ASP*, represents a new way of using the logic programming paradigm for action model learning.

Unlike the alternative approach, introduced by Balduccini in 2007, it can be used in the presence of *sensoric noise* and *action failures.* Empirical evidence gathered during our experiments suggests that, thanks to a more compact encoding of action models and the usage of a "reactive" variant of ASP, our method is noticeably faster, and thus can be used in more complex domains. On the other hand, the simplistic action model encoding means that we are unable to learn the *conditional effects.*

Even though it is not as versatile as $3SG$ algorithm (it can only overcome half of the discussed challenges), this method may be suitable in some situations because of its purely declarative nature.

There are some open issues put forward by the research conducted in this thesis. One of the most promising topics for future investigation is the influence of actual agent's behaviour on the speed of the learning process. During our experiments, the agent was acting either randomly or reflexively. However, we believe that the "explorative" behaviour, by which the agent would proactively try to confirm or disprove his current beliefs, might positively influence the learning rate and improve the overall quality of learned action models.

Another interesting issue, that attracts further attention, is the problem of "discretization" of continuous data from agent's sensors into the form of fluents. Numerous machine learning algorithms are known to produce better models by discretizing continuous domain features. Moreover, all the action learning methods discussed here strictly require the data to be discrete. It would be useful to inspect how various practices from the field of discretization research can be employed in the field of action model learning.

Even though these issues are beyond the scope of this thesis, they can be seen as a direct continuation of the research conducted in it.

Appendices

Appendix A

Additional Properties of Effects (learned by $3SG$)

Following collection of theorems will describe the conditions under which we learn action's effects using $3SG$ algorithm. Let us therefore start by formal definition of this concept.

Definition 10. *Learned effect* will be understood as a double $E = \langle a^f, C = \{a^f_{c0}, a^f_{c1}, \ldots, a^f_{ci}\}\rangle$ where C is a set of conditions of a^f. We say that *we have learned an effect* E if the following holds w.r.t. our action model $\langle \mathcal{EF}, \mathcal{P}\rangle$:

i. $a^f \in \mathcal{EF}$

ii. $\forall a^f_c \in C : (a^f_c \in EF) \wedge \big(\mathcal{P}(a^f_c) \geq minP\big)$

iii. $(C = \emptyset) \Rightarrow \big(\mathcal{P}(a^f) \geq minP\big)$

In other words, a^f and all its conditions must be contained in $\mathcal{EF}$, and these conditions need to have high enough[1] probability value $\mathcal{P}$. If E is a *non-conditional* effect ($C = \emptyset$), we also require a^f itself to be probable enough.

Note A.1. Note that if we have at least one probable condition a^f_c, we allow a^f to be improbable itself. That is because effect a^f can contain useful information even if it does not apply most of the time, but we know under which conditions it does. Imagine, for example, that a^f means *"eating food causes you to be sick"*. In 99% of examples it may not happen. However, if in addition to that, we have a strong condition a^f_c meaning that *"eating food causes you to be sick only if the food is poisoned"*, then these two atoms together represent a useful piece of information.

For the purposes of following theorems, we will establish a notation, where $pos(a^f)_c$ means the *"number of positive examples w.r.t. a^f, where fluent c was in observation o"* (and similarly for negative examples: $neg(a^f)_c$).

Theorem 3. Consider $3SG$ algorithm without forgetting (memoryLength = ∞). If we observe a sufficient number of positive and no negative examples w.r.t. a^f (precisely $pos(a^f) \geq minEx \wedge neg(a^f) = 0$), then we can be sure that we have *learned the non-conditional effect* $E = \langle a^f, \emptyset \rangle$. We can also say that we *have not learned* any other *conditional effect* $E' = \langle a^f, C \neq \emptyset \rangle$.

Theorem 4. If we have observed at least one negative example w.r.t. a^f, then we need forgetting ($memoryLength \in \mathbb{N}$) in order to learn the *non-conditional* effect $E = \langle a^f, C = \emptyset \rangle$.

Theorem 5. Consider a $3SG$ algorithm without forgetting (memoryLength = ∞). If we observe a sufficient number of positive examples w.r.t. a^f

[1]Minimal required probability value is specified by a constant $minP$ from interval $(0, 1)$. For example: $minP = 0.9$.

(i.e. $pos(a^f) \geq minEx$), while the number of those, where c was observed is sufficiently higher than those with $\overline{c}$ observed (more precisely $\forall a^f_c \in C$: $pos(a^f)_c \geq minP \cdot \big(pos(a^f)_c + pos(a^f)_{\overline{c}}\big)$), and at the same time every $a^f_c \in C$ is supported by at least $minEx$ examples, and a^f has at least one negative example where $\overline{c}$ held, then we can safely say that we will learn the *conditional* effect $E = \langle a^f, C \neq \emptyset \rangle$.

Proof. Definition 10 enumerates three conditions that need to be met before we can say that we have *learned* the effect E. One after another, we will show how these conditions result from the premises of our theorem.

i. $a^f \in \mathcal{EF}$ because $3SG$ algorithm adds a^f to $\mathcal{EF}$ when confronted with the first positive example (line 2 in Fig. 8.3), and we assumed that $pos(a^f) \geq minEx$, and $minEx > 0$.

ii. In order for any a^f_c to appear in $\mathcal{EF}$, we need (according to line 16) the $neg(a^f)_{\overline{c}}$ to be greater than 0, which is directly our assumption. Now realize that, according to semantics of $\mathcal{EF}$, positive examples for a^f_c are exactly those examples that are positive w.r.t. a^f and $c \in o$. Similarly, negative examples for a^f_c are exactly those which are positive w.r.t. a^f, but with $\overline{c} \in o$. Our assumption "$pos(a^f)_c \geq minP \cdot \big(pos(a^f)_c + pos(a^f)_{\overline{c}}\big)$" can therefore be rewritten as "$pos(a^f_c) \geq minP \cdot \big(pos(a^f_c) + neg(a^f_c)\big)$". This can be further modified to "$pos(a^f_c) / \big(pos(a^f_c) + neg(a^f_c)\big) \geq minP$", which is equivalent with the "$\mathcal{P}(a^f_c) \geq minP$" condition of definition 10, since we also assume that a^f_c is sufficiently supported ($pos(a^f_c) + neg(a^f_c) \geq minEx$).

iii. Finally, we will prove the last condition $((C = \emptyset) \Rightarrow \big(P(a^f) \geq minP\big))$ by contradiction. Let us assume its negation: $(C = \emptyset) \wedge \big(P(a^f) < minP\big)$

The fact that C is empty means (assuming we do not allow empty observations) that there was no negative example w.r.t. a^f (algorithm would otherwise add some a^f_c into C on line 16). Since we assume that $pos(a^f) \geq minEx$, and we do not have any negative examples, the probability $\mathcal{P}(a^f)$ must equal 1. Therefore it cannot be lower than $minP$.

$\square$

Appendix B

Worst Case for $3SG$ Algorithm

First of all, let us denote our *training set* by E and *test set* by T. Recall that online algorithms may perform differently when given the same set of inputs in different order.

When talking about online *action learning* algorithms, our inputs are all the examples from the training set E. A permutation of E, denoted by $p(E)$, then corresponds to a single ordering of inputs. We will use the following notation: $p(E) = \Big((o_1, a_1, o'_1),\ \ldots,\ (o_n, a_n, o'_n)\Big)$

Also, let $F\Big(p(E)\Big)$ denote the F-measure of the action model that we get after gradually processing all the examples $(o_i, a_i, o'_i) \in p(E)$ by our learning algorithm. Permutation $p(E)$ is called a *worst case* if there does not exist another permutation $q(E)$ such that $F\Big(q(E)\Big) < F\Big(p(E)\Big)$.

In this appendix, we are going to take a look at a worst case for the algorithm $3SG$. However, as you may have noticed, there may exist more than one permutation of E that can be characterized as a worst case (since more of them may have the same, worst, value of F-measure).

The case that we describe here assumes that in every time step we observe exactly one example. Our action model $\mathcal{EF}$ before the $i-$th iteration of $3SG$ algorithm will be denoted by $\mathcal{EF}_i$ (in other words, $\mathcal{EF}_i$ is the action model on the input of $i-$th iteration of $3SG$). Also, let us abbreviate our *memoryLength* parameter with letter m. Before we can define the worst case for $3SG$ algorithm, we need to establish one more concept:

Definition 11 (Fluent changes)**.** The set of doubles $\big(f,(o,a,o')\big)$, defined as

$$C = \Big\{\big(f,(o,a,o')\big) \;\Big|\; (o,a,o') \in T \;\wedge\; f \in \Delta(o,o')\Big\},$$

is called the set of *fluent changes* occurring in our test set T.

Intuitively, our worst case is such an ordering of the training set, in which last few elements contain a specific kind of negative example for every single fluent change from C. More precisely:

Theorem 6. *We say that a permutation $p(E) = \big((o_1,a_1,o'_1),\dots\ (o_n,a_n,o'_n)\big)$ of our training set E is a worst case for $3SG$ algorithm if the following condition holds for every pair of an example $(o_i,a_i,o'_i) \in \{(o_{n-m},a_{n-m},o'_{n-m}),$ $(o_{n-m+1},a_{n-m+1},o'_{n-m+1}),\dots,(o_n,a_n,o'_n)\}$ and a fluent change $\big(f,(o,a,o')\big) \in C$:*

$$\Big[\big(a^f \in \mathcal{EF}_i\big) \wedge \big(\nexists c : a^f_c \in \mathcal{EF}_i \wedge \overline{c} \in o\big)\Big] \Rightarrow \exists (o_j,a_j,o'_j) \in p(E) : i \leq j \leq n, \textit{such that:}$$

$$\Big[\big((o_j,a_j,o'_j) \textit{ is a neg. example w.r.t. } a^f\big) \wedge \big(\exists c : c \in o_j \wedge \overline{c} \in o\big)\Big]$$

In other words, for every possible fluent change $\big(f,(o,a,o')\big)$ we want to add to $\mathcal{EF}$ at least one such condition a^f_c that would render our previously learned effect a^f inapplicable (which means that $\overline{c}$ must be in o). However, most of these conditions would normally be forgotten by $3SG$ over time.

Therefore, we need to add them during the last few time steps (less than *memoryLength*), so that there is not enough time for the algorithm to delete them.

Proof. According to definition from section 8.3, the F-measure is defined as:

$$F_\beta = (1+\beta) \cdot \frac{P \cdot R}{\beta^2 \cdot P + R}$$

Precision (P), recall (R) and weight (β) are all either positive or equal to 0. Therefore, the lowest possible value of F-measure is 0. This means that any permutation $p(E)$, after which the F-measure of our action model is 0, can be considered a worst case. We will show that this holds for the permutation $p(E)$ from theorem 6.

First of all, realize that any element added to $\mathcal{EF}$ cannot be removed if it is not in $\mathcal{EF}$ longer than m time steps (two *Foreach* loops on lines 21-28 in figure 8.3 are the only places where elements are deleted from $\mathcal{EF}$). Therefore any element added to $\mathcal{EF}$ during the last m timesteps will remain there.

Since any implication $p \Rightarrow q$ can be converted into a disjunction $\neg p \lor q$, the statement from theorem 6 can also be rephrased into a disjunctive form. The following statement holds for every $\big(f, (o, a, o')\big) \in C$:

Upon processing every example (o_i, a_i, o'_i) from last m examples, at least one of following statements must be true:

i. $\neg\Big[\big(\exists a^f \in \mathcal{EF}_i\big) \land \big(\nexists c : a^f_c \in \mathcal{EF}_i \land \overline{c} \in o\big)\Big]$

ii. $\exists$ example $(o_j, a_j, o'_j) \in p(E) : i \leq j$, such that:
$\big((o_j, a_j, o'_j)$ is a neg. example w.r.t. $a^f\big) \land \big(\exists c : c \in o_j \land \overline{c} \in o\big)$

Let us show that, if this holds for a given fluent change $\big(f, (o, a, o')\big)$, then our learned action model *will not be able to predict* f in (o, a, o'). The definition of prediction from section 8.3 says that $\mathcal{EF}$ predicts f in (o, a, o') if $a^f \in \mathcal{EF} \wedge (\nexists c : a^f_c \in \mathcal{EF} \wedge c \notin o)$.

i. Statement (i.) can be rewritten as $\big(\nexists a^f \in \mathcal{EF}_i\big) \vee \big(\exists c : a^f_c \in \mathcal{EF}_i \wedge \overline{c} \in o\big)$. Both these disjuncts contradict the conditions from the definition of prediction from section 8.3 (assuming that an observation does not contain two mutually complementary fluents). Therefore $\mathcal{EF}_i$ does not predict f in (o, a, o').

ii. Second statement basically says that (either the current example or) some other example (o_j, a_j, o'_j) further in $p(E)$ is negative w.r.t. a^f and at the same time, there is some $c \in o_j$, such that $\overline{c} \in o$. When we serve this example as the input to $3SG$ algorithm, it will add a^f_c into $\mathcal{EF}$ (see lines 12-18 in figure 8.3). This automatically makes the statement (i.) true after the example (o_j, a_j, o'_j) was processed by $3SG$.

Therefore, the statement (i.) was either true for all the last m examples, or it became true after processing one of them. Since nothing that was added during last m steps cannot be deleted from $\mathcal{EF}$, we can safely say that (i.) will be true after the last example. This means that our final action model will not predict f in (o, a, o'). This should hold for every fluent change $\big(f, (o, a, o')\big)$ from our test set, making the total number of correct predictions about f, denoted by "$A(f)$" in section 8.3, equal to 0. Definition of the precision (P) and recall (R) from the same section implies that if $A(f) = 0$ for every f that changed within the test set, then $P = R = 0$ and so does the F-measure.

$\square$

Let us now take a look at the likelihood that our training set will indeed be ordered in a way that fits the definition of this worst case. We will first formulate four statements S_1, S_2, S_3, S_4 and try to express their respective probabilities $P(S_1), P(S_2), P(S_3), P(S_4)$:

1. S_1: "A given example (o_j, a_j, o'_j) is negative w.r.t. a certain a^f."
   In other words, $P(S_1)$ is a probability that, $a_j = a$ and *at the same time* $\overline{f} \in o'_j$ in a given pair $\big((o_j, a_j, o'_j), a^f\big)$.

2. S_2: "Given two examples, (o, a, o') and (o_j, a_j, o'_j), o_j contains a fluent c, such that $\overline{c} \in o$."
   In other words, $P(S_2)$ is a probability that two examples contain some mutually complementary fluents in their respective first observations.

3. S_3: "For a given fluent change $\big(f, (o, a, o')\big)$, there is at least one (o_j, a_j, o'_j) in the collection of m examples $\{(o_{n-m}, a_{n-m}, o'_{n-m}), \ldots, (o_n, a_n, o'_n)\}$, for which both previous statements hold (S_1 and S_2 at the same time)."

$$P(S_3) = 1 - \Big(1 - \big(P(S_1) \cdot P(S_2)\big)\Big)^m$$

4. S_4: "The previous statement (S_3) holds for every single fluent change $\big(f, (o, a, o')\big) \in C$ at the same time."

$$P(S_4) = P(S_3)^{|C|} = \Big[1 - \Big(1 - \big(P(S_1) \cdot P(S_2)\big)\Big)^m\Big]^{|C|}$$

Statement S_4 is practically an alternative description of the worst case from theorem 6. Notice that we need to use the following four parameters to express the probability of our worst case (statement S_4):

- m: Our *memoryLength* parameter. It is usually set to approximately 50-150.

- $|C|$: The number of all the fluent changes occurring in our test set. In a reasonable setting with big enough test set, this number is at least several thousand.
- $P(S_1)$ and $P(S_2)$: Probabilities of statements S_1 and S_2 depend heavily on our domain. However, in a typical domain, they tend to be significantly lower than 0.5.

To illustrate the actual probability of the worst case in a typical setting, quite similar to our experiments (see chapter 9), let us compute $P(S_4)$ with $m = 100, |C| = 3000, P(S_1) = 0.2, P(S_2) = 0.2$.

$$\begin{aligned} P(S_4) &= \left[1 - \left(1 - \left(0.2 \cdot 0.2\right)\right)^{100}\right]^{3000} \\ &= 6.7986533 \cdot 10^{-23} \end{aligned}$$

This probability seems to be insignificant, which leads us to question the usefulness of the competitive analysis, as a kind of worst-case analysis, in this particular example.

Appendix C

Dynamic Error Tolerance for Reactive ASP Method

Figure C.1 depicts a modified cumulative $P[t]$ of *learner* logic program for the “Reactive ASP” action learning method from chapter 10.

It counts not only the negative, but also all the positive examples w.r.t. relevant literal from $\mathcal{AL}$. Error tolerance threshold is then computed as 30% of the sum of all the positive and negative examples.

```
negExCauses(A,F,N+1) ←
          negExCauses(A,F,N), obs(¬F,t), exe(A,t).
posExCauses(A,F,N+1) ←
          posExCauses(A,F,N), obs(F,t), obs(¬F,t−1), exe(A,t).

negExKeeps(A,F,N+1) ←
          negExKeeps(A,F,N), obs(¬F,t),
          obs(F,t−1), exe(A,t).
posExKeeps(A,F,N+1) ←
          posExKeeps(A,F,N), obs(F,t),
          obs(F,t−1), exe(A,t).

negExPre(A,F,N+1) ←
          negExPre(A,F,N), exe(A,t), obs(¬F,t−1).
posExPre(A,F,N+1) ←
          posExPre(A,F,N), exe(A,t), obs(F,t−1).

← causes(A,F), negExCauses(A,F,N), posExCauses(A,F,P), N > (N+P)∗0.3.
← keeps(A,F), negExKeeps(A,F,N), posExKeeps(A,F,P), N > (N+P)∗0.3.
← pre(A,F), negExPre(A,F,N), posExPre(A,F,P), N > (N+P)∗0.3.
```

Figure C.1: Modified $P[t]$ should be able to deal with sensoric noise, by introducing error tolerance. In this case, the error tolerance is computed dynamically, based on the total number of all the relevant examples.

Bibliography

[Agrawal, 1994] R. Agrawal, R. Srikant. *Fast algorithms for mining association rules.* In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB): 487-499. 1994.

[Agre-Chapman, 1987] P. E. Agre, D. Chapman. *Pengi: an implementation of a theory of activity.* In Proceedings of the Sixth National Conference in Artificial Intelligence (AAAI-87): 268-272. 1987.

[Aizerman-Braverman-Rozoner, 1964] M. A. Aizerman, E. M. Braverman, L. I. Rozoner. *Theoretical foundations of the potential function method in pattern recognition learning.* Automation and Remote Control 25: 821-837. 1964.

[Amir-Chang, 2008] E. Amir, A. Chang. *Learning partially observable deterministic action models.* Journal of Artificial Intelligence Research, Volume 33 Issue 1: 349-402. 2008.

[Amir-Russel, 2003] E. Amir, S. Russel. *Logical Filtering.* In Proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI'03). 2003.

[Armijo, 1966] L. Armijo. *Minimization of functions having Lipschitz continuous first partial derivatives.* Pacific Journal of Mathematics, volume 16. 1966.

[Baioletti-Marcugini-Milani, 1998] M. Baioletti, S. Marcugini, A. Milani. *C-SATPlan: A SATPlan-based Tool for Planning with Constraints.* In Proceedings of AIPS-98 Workshop on Planning as Combinatorial Search. 1998.

[Balduccini, 2007] M. Balduccini. *Learning Action Descriptions with A-Prolog: Action Language $\mathcal{C}$.* In Proceedings of Logical Formalizations of Commonsense Reasoning, 2007 AAAI Spring Symposium. 2007.

[Baral, 2003] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press. 2003.

[Bayer-McCreight, 1972] R. Bayer, E. M. McCreight. *Organization and maintenance of large ordered indexes.* Acta Informatica 18. II., Volume 1, Issue 3: 173-189. 1972.

[Bertsekas, 1999] D. P. Bertsekas. *Nonlinear Programming.* Athena Scientific. 1999.

[Bishop, 2006] C. M. Bishop. *Pattern Recognition and Machine Learning, 1st edition.* New York, NY : Springer. 2006.

[Blum-Furst, 1997] A. Blum, M. Furst. *Fast Planning Through Planning Graph Analysis.* Artificial Intelligence - Volume 90:281-300. 1997.

[Blum-Langford, 2000] A. Blum, J. Langford. *Probabilistic Planning in the Graphplan Framework.* Lecture Notes in Computer Science. Volume 1809. 2000.

[Borodin-El-Yaniv, 1998] A. Borodin, R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, New York. 1998.

[Buro-Furtak, 2004] M. Buro, T. M. Furtak. *RTS games and real-time AI research.* In Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS). 2004.

[Chang-Lee, 1973] . C. Chang, R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press. 1973.

[Cook, 1971] S. A. Cook. *The complexity of theorem-proving procedures.* In proceedings of the third annual ACM symposium on Theory of computing (STOC '71). 1971.

[Čertický, 2008] M. Čertický. *An Architecture for Universal Knowledge-based Agent.* MEi:CogSci 2008 Proceedings. 2008.

[Čertický, 2009] M. Čertický. *An Architecture for Universal Knowledge-based Agent.* Master Thesis, Faculty of Mathematics and Physics, Comenius University, Bratislava. 2009.

[Čertický, 2010a] M. Čertický. *IK-STRIPS Formalism for Fluent-free Planning with Incomplete Knowledge.* Technical Reports, Comenius University, Bratislava, 2010.

[Čertický, 2010b] M. Čertický. *Fluent-free Action Representation with IK-STRIPS Planning Formalism.* Student Research Conference 2010 Proceedings. 2010.

[Čertický, 2012a] M. Čertický. *3SG algoritmus a učenie sa akčných modelov v reálnom čase.* In Proceedings of Cognition and Artificial Life XII. 2012.

[Čertický, 2012b] M. Čertický. *Učenie sa akčných modelov v reálnom čase pomocou algoritmu 3SG: vlastnosti a experimenty.* In Proceedings of Student Research Conference 2012. 2012.

[Čertický, 2012c] M. Čertický. *Action Learning with Reactive Answer Set Programming: Preliminary Report.* In Proceedings of The Eighth International Conference on Autonomic and Autonomous Systems (ICAS 2012). 2012.

[Davis-Putnam, 1960] M. Davis, H. Putnam. *A computing procedure for quantification theory.* Journal of the ACM, 7: 201-215. 1960.

[Deb, 2005] K. Deb. *Multi-objective Optimization.* Search Metodologies (Springer). 2005.

[Domshlak-Hoffman, 2007] C. Domshlak, J. Hoffmann. *Probabilistic planning via heuristic forward search and weighted model counting.* Journal of Artificial Intelligence Research 30:565-620. 2007.

[Eiter et al., 2000] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. *Planning Under Incomplete Knowledge.* Lecture Notes in Computer Science, Volume 1861: 807-821. 2000.

[Eiter et al., 2005] T. Eiter, E. Erdem, M. Fink, and J. Senko. *Updating action domain descriptions.* Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05): 418–423. 2005.

[Farritor-Dubowsky, 2002] S. Farritor, S. Dubowsky. *A Genetic Planning Method and its Application to Planetary Exploration.* ASME Journal of Dynamic Systems, Measurement and Control, 124(4): 698-701. 2002.

[Fikes-Nilsson, 1971] R. E. Fikes, N. Nilsson. *STRIPS: A new approach to the application of theorem proving to problem solving.* Artificial Intelligence 5(2): 189-208. 1971.

[Fox-Long, 2003] M. Fox, D. Long. *PDLL2.1: An Extension to PDLL for Expressing Temporal Planning Domains.* Journal of Artificial Intelligence Research, Volume 20. 2003.

[Freund-Schapire, 1999] Y. Freund, R. Schapire. *Large margin classification using the perceptron algorithm.* Machine learning 37: 277-296. 1999.

[Gebser et al., 2008] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, S. Thiele. *Engineering an Incremental ASP Solver.* In Proceedings of the 24th International Conference on Logic Programming (ICLP'08). 2008.

[Gebser et al., 2011] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. *Reactive Answer Set Programming.* In Proceedings of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2011): 54-66. 2011.

[Gelfond-Lifschitz, 1988] M. Gelfond, V. Lifschitz. *The stable model semantics for logic programming.* In Proceedings of ICLP-88: 1070-1080. 1988.

[Gelfond-Lifschitz, 1991] M. Gelfond, V. Lifschitz. *Classical negation in logic programs and disjunctive databases.* New Generation Computing: 365-385. 1991.

[Gemrot et al., 2009] J. Gemrot, R. Kadlec, M. Bída, O. Burkert, R. Píbil, J. Havlíček, L. Zemčák, J. Simlovič, R. Vansa, M. Stolba, T. Plch, C. Brom. *Pogamut 3 Can Assist Developers in Building AI (Not Only)* *for Their Videogame Agents.* Agents for games and simulations: 1–15, Springer-Verlag, Berlin, Heidelberg. 2009.

[Gordon-Kochen, 2007] M. Gordon, M. Kochen. *Recall-precision trade-off: A derivation.* Journal of the American Society for Information Science: 145-151. 2007.

[Guinchiglia-Lifschitz, 1998] E. Guinchiglia, V. Lifschitz. *An Action Language Based on Causal Explanation: Preliminary Report.* In Proceedings of 15th National Conference of Artificial Intelligence (AAAI'98). 1998.

[Gupta-Nau, 1992] N. Gupta, D. S. Nau. *On the Complexity of Blocks-World Planning.* Artificial Intelligence 56(2-3): 223-254. 1992.

[Hopcroft-Motwani-Ullman, 2007] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Boston/San Francisco/New York: 368. 2007.

[Hromkovič, 2010] J. Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics.* Springer-Verlag. 2010.

[Jensen, 1999] R. M. Jensen. *OBDD-based Universal Planning in Multi-Agent, Non-Deterministic Domains.* Master's Thesis, Technical University of Denmark Lyngby, DK-2800, Department of Automation. 1999.

[Jensen, 2003] R. M. Jensen. *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains.* School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213. 2003.

[Jensen-Veloso, 2000] R. M. Jensen, M. M. Veloso. *OBDD-based deterministic planning using the UMOP planning framework.* In Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning: 26-31. 2000.

[Kambhampati, 2000] S. Kambhampati. *Planning Graph as a (Dynamic) CSP: Exploiting EBL, DDB and other CSP Search Techniques in Graphplan.* Journal of Artificial Intelligence Research. 2000.

[Kautz-Selman, 1992] H. A. Kautz, B. Selman. *Planning as Satisfiability.* In Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92): 359-363. 1992.

[Kautz-Selman-Jiang, 1997] H. Kautz, B. Selman, Y. Jiang. *A general stochastic approach to solving problems with hard and soft constraints.* The Satisfiability Problem: Theory and Applications. 1997.

[Kulich et al., 2010] M. Kulich, K. Košnar, J. Chudoba, J. Faigl, L. Přeučil. *On a mobile robotics e-learning system.* In Twentieth European Meeting on Cybernetics and Systems Research: 597–602. 2010.

[Kutluhan-Hendler-Nau, 1994] E. Kutluhan, J. Hendler, D. S. Nau. *HTN planning: Complexity and expressivity.* In AAAI-94 Proceedings. 1994.

[Lee-Lifschitz, 2003] J. Lee, V. Lifschitz. *Describing Additive Fluents in Action Language C+.* In Proceedings of IJCAI-03. 2003.

[Leone et al., 2006] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello. *The DLV system for knowledge representation and reasoning.* ACM Trans. Comput. Log. 7: 499–562. 2006.

[Lewis, 1994] D. D. Lewis, J. Catlett. *Heterogeneous uncertainty sampling for supervised learning.* In Proceedings of the Eleventh International Conference on Machine Learning:148–156. Morgan Kaufmann. 1994.

[Lifschitz-Turner, 1999] V. Lifschitz, H. Turner. *Representing transition systems by logic programs.* In Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99). 1999.

[Lifschitz, 2002] V. Lifschitz. *Answer Set Programming and Plan Generation.* Artificial Intelligence, vol. 138: 39-54. 2002.

[Link 1, 2012] www.dai.fmph.uniba.sk/upload/9/9d/Oclingo-learning.zip

[Littman et al. 1998] M. L. Littman, J. Goldsmith, M. Mundhenk. *The computational complexity of probabilistic planning.* Journal of Artificial Intelligence Research 9:1-36. 1998.

[Lopez-Bacchus, 2003] A. Lopez, F. Bacchus. *Generalizing GraphPlan by Formulating Planning as a CSP.* In Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03). 2003.

[Makhoul et al., 1999] J. Makhoul, F. Kubala, R. Schwartz, R. Weischedel. *Performance measures for information extraction.* 1999.

[Marquis, 2000] P. Marquis. *Consequence finding algorithms.* In D. Gabbay, P. Smets (Eds.), Handbook of Defeasible Reasoning and Uncertainty Management Systems, Volume 5: Algorithms for defeasible and uncertain reasoning. Kluwer. 2000.

[McDermott et al., 1998] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins. *PDDL*

- *The Planning Domain Definition Language.* Draft. Available at http://www.cs.yale.edu/ dvm. 1998.

[McMahan-Gordon, 2005] H. B. McMahan, G. J. Gordon. *Fast exactplanning in Markov decision processes.* In Proceedings of ICAPS. 2005.

[Minsky-Papert, 1969] . M. L. Minsky, S. A. Papert. *Perceptrons.* The MIT Press. 1969.

[Moskewitz et al., 2001] M. W. Moskewitz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. *Chaff: engineering an Efficient SAT Solver.* In Proceedings of the 38th Design Automation Conference (DAC'01). 2001.

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

[Mourao-Petrick-Steedman, 2010] K. Mourao, R. P. A. Petrick, M. Steedman. *Learning action effects in partially observable domains.* In Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence. 2010.

[Niemela-Simons-Syrjanen, 2000] I. Niemela, P. Simons, T. Syrjanen. *Smodels: A System for Answer Set Programming.* In Proceedings of the 8th International Workshop on Non-Monotonic Reasoning. 2000.

[Nilsson, 1980] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto. 1980.

[Otero-Varela, 2006] R. Otero, M. Varela. *Iaction, a System for Learning Action Descriptions for Planning.* In Proceedings of 16th International Conference on Inductive Logic Programming, ILP 06. 2006.

[Pasula-Zettlemoyer-Kaelbling, 2007] H. M. Pasula, L. S. Zettlemoyer, L. P. Kaelbling. *Learning Symbolic Models of Stochastic Domains.* Journal of Artificial Inteligence Research, Volume 29: 309-352. 2007.

[Pednault, 1989] E. P. D. Pednault. *ADL: exploring the middle ground between STRIPS and the situation calculus.* Proceedings of the first international conference on Principles of knowledge representation and reasoning. 1989.

[Pednault, 1994] E. P. D. Pednault. *ADL and the state-transition model of action.* Journal of Logic and Computation 4:467–512. 1994.

[Rintanen, 2009] J. Rintanen. *Planning and SAT.* Handbook of Satisfiability: 483-504, IOS Press. 2009.

[Rijsbergen, 1979] C. J. V. Rijsbergen. *Information Retrieval.* Butterworth-Heinemann, Newton, MA, USA, 2nd edition. 1979.

[Rosenblatt, 1958] F. Rosenblatt. *The perceptron: a probabilistic model for information storage and organisation in the brain.* Psychological Review 65(6): 386-408. 1958.

[Russel-Norvig, 2003] S. Russel, P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall. 2003.

[Sacredoti, 1975] E. D. Sacredoti. *The Nonliner Nature of Plans.* IJCAI'75 Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1. 1975.

[Selman-Kautz-Cohen, 1993] B. Selman, H. Kautz, B. Cohen. *Local search strategies for satisfiability testing.* Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge 26. 1993.

[Simons et al., 2002] P. Simons, I. Niemela, and T. Soininen. *Extending and Implementing the Stable Model Semantics.* Artificial Intelligence, 138(1-2):181-234. 2002.

[Slaney-Thiebaux, 2001] J. Slaney, S. Thiebaux. *Blocks World revisited.* Artificial Intelligence 125: 119-153. 2001.

[Subrahmanian-Zaniolo, 1995] V. S. Subrahmanian, C. Zaniolo. *Relating Stable Models and AI Planning Domains.* Logic Programming, Proceedings of 12th ICLP: 233-246. 1995.

[Tate, 1977] A. Tate. *Generating Project Networks.* IJCAI'77 Proceedings of the 5th international joint conference on Artificial intelligence - Volume 2. 1977.

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

[Wilkins, 1984] D. E. Wilkins. *Domain-Independent Planning Representation and Plan Generation.* Artificial Intelligence - Volume 22, Issue 3. 1984.

[Yang-Wu-Jiang, 2007] Q. Yang, K. Wu, Y. Jiang. *Learning action models from plan examples using weighted MAX-SAT.* Artificial Intelligence, Volume 171: 107-143. 2007.

[Zettlemoyer-Pasula-Kaelbling, 2003] L. S. Zettlemoyer, H. M. Pasula, L. P. Kaelbling. *Learning probabilistic relational planning rules.* MIT TR. 2003.