Inference and Learning with Planning Models

by

Diego Aineto García

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

A thesis submitted for the degree of

Título de Doctor por la Universitat Politècnica de València

Under the supervision of:

Eva Onaindía de la Rivaherrera
Sergio Jiménez Celorrio

June 2022

ABSTRACT

Inference and learning are the acts of reasoning about some collected evidence in order to reach a logical conclusion regarding the process that originated it. In the context of a state-space model, inference and learning are usually concerned with explaining an agent's past behaviour, predicting its future actions or identifying its model. In this thesis, we present a framework for inference and learning in the state-space model underlying the classical planning model, and formulate a palette of inference and learning problems under this unifying umbrella. We also develop effective planning-based approaches to solve these problems using off-the-shelf, state-of-the-art planning algorithms. We will show that several core inference and learning problems that previous research has treated as disconnected can be formulated in a cohesive way and solved following homogeneous procedures using the proposed framework. Further, our work opens the way for new applications of planning technology as it highlights the features that make the state-space model of classical planning different from other models.

RESUMEN

Inferencia y aprendizaje son los actos de razonar sobre evidencia recogida con el fin de alcanzar conclusiones lógicas sobre el proceso que la originó. En el contexto de un modelo de espacio de estados, inferencia y aprendizaje se refieren normalmente a explicar el comportamiento pasado de un agente, predecir sus acciones futuras, o identificar su modelo. En esta tesis, presentamos un marco para inferencia y aprendizaje en el modelo de espacio de estados subyacente al modelo de planificación clásica, y formulamos una paleta de problemas de inferencia y aprendizaje bajo este paraguas unificador. También desarrollamos métodos efectivos basados en planificación que nos permiten resolver estos problemas utilizando algoritmos de planificación genéricos del estado del arte. Mostraremos que un gran número de problemas de inferencia y aprendizaje claves que han sido tratados como desconectados se pueden formular de forma cohesiva y resolver siguiendo procedimientos homogéneos usando nuestro marco. Además, nuestro trabajo abre las puertas a nuevas aplicaciones para tecnología de planificación ya que resalta las características que hacen que el modelo de espacio de estados de planificación clásica sea diferente a los demás modelos.

RESUM

Inferència i aprenentatge són els actes de raonar sobre evidència arreplegada a fi d'aconseguir conclusions lògiques sobre el procés que la va originar. En el context d'un model d'espai d'estats, inferència i aprenentatge es referixen normalment a explicar el comportament passat d'un agent, predir les seues accions futures, o identificar el seu model. En esta tesi, presentem un marc per a inferència i aprenentatge en el model d'espai d'estats subjacent al model de planificació clàssica, i formulem una paleta de problemes d'inferència i aprenentatge davall este paraigua unificador. També desenrotllem mètodes efectius basats en planificació que ens permeten resoldre estos problemes utilitzant algoritmes de planificació genèrics de l'estat de l'art. Mostrarem que un gran nombre de problemes d'inferència i aprenentatge claus que han sigut tractats com desconnectats es poden formular de forma cohesiva i resoldre seguint procediments homogenis usant el nostre marc. A més, el nostre treball obri les portes a noves aplicacions per a tecnologia de planificació ja que ressalta les característiques que fan que el model d'espai d'estats de planificació clàssica siga diferent dels altres models.

ACKNOWLEDGEMENTS

I have thoroughly enjoyed myself throughout my PhD studies and I consider myself extremely lucky for it. Nonetheless, it was the support of the people around me that made this such a joyous journey so I would like to use this opportunity to thank them.

Let me start by expressing my gratitude to my PhD advisors Eva and Sergio. To Eva, for her impeccable guidance during my five years under her care, and for providing me with innumerable opportunities to pave my career as a researcher. And to Sergio, the liveliest mind I have had the pleasure to meet, always coming up with new problems and challenges to stimulate my own. I will dearly remember our time together.

I would also like to extend my thanks to my coauthors Miquel Ramírez, Enrico Scala and Ivan Serina, to whom I am indebted for their scientific support. Their work and collaboration have positively impacted this thesis and me.

Of course, I owe everything to my parents, who have given their all for me. They have shed every last drop of sweat for my sake, often at the cost of their own health. I do not think I will ever be able to repay them for all the love they have poured into me, but I will strive my utmost to make myself worth of such a gift.

Last, all the work developed in this doctoral thesis has been possible thanks to the FPU16/03184 fellowship that I have enjoyed for the duration of my PhD studies. I have also been supported by my advisors' grants TIN2017-88476-C2-1-R, TIN2014-55637-C2-2-R-AR, and RYC-2015-18009.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Introduction

1. INTRODUCTION

1.1. Inference and Learning with Planning Models

Inference is the process of reaching a conclusion on the basis of evidence and reasoning. In general, inference problems are concerned with the explanation of past happenings or the prediction of future events. We can find examples of both in the research developed by the planning community, which has shown increasing interest in inference problems in recent years. For instance, planning has been used to address the problems of finding explanations for observed behavior (Davis-Mendelow et al., 2013), Diagnosis (Sohrabi et al., 2010), and Goal Recognition (Ramírez & Geffner, 2009; Sukthankar et al., 2014). The above works lead to a natural distinction between inference tasks devoted to *explicability* and *predictability* (Chakraborti et al., 2019).

An issue that grabs our attention when we explore the literature is that, despite all sharing a common underlying computational core with planning, these inference problems are formulated in very different ways and using different formalism. Finding preferred explanations for observed behavior has been formulated with Linear Temporal Logic (LTL) (Sohrabi et al., 2011). Situation calculus (Sohrabi et al., 2010) and event-based languages (Grastien et al., 2011; Haslum & Grastien, 2011) are commonly used to capture the evolution of the system. Recognition problems have also been addressed from different perspectives: with planning-based formulations (E-Martín et al., 2015; Ramírez & Geffner, 2010; Sohrabi et al., 2016), with heuristics (Pereira et al., 2020) or adopting changes in the agent model to enhance the recognition problem (Keren et al., 2019). This lack of homogeneity is in stark contrast with the formulation of inference problems that we can find in sequential models such as Markov Decision Processes (MDPs) and Hidden Markov Models (HMMs), as well as other graphical models such as Bayesian Networks (BN).

Another issue that plagues these approaches is a very constrained understanding of the observation model that defines the partial information received by the observer. In general, little attention has been paid to formalizing the observation model of the observer, and most works adopt a simplistic view that disregards how the observations are produced. The approach by Ramírez and Geffner, 2010 assumes a deterministic sensor and partial observations solely over the executed actions of the actor. The work by Keren et al., 2019 deals with noisy (non-deterministic) sensor models that emit one among several observable tokens but these are likewise only over actions. Exceptionally, extending observations to handle noise over state variables and actions is proposed in Sohrabi et al., 2016 but this formulation is also constrained to a single element per observation. Additionally, all the cited approaches ignore the distribution of the observations, thus assuming all observations are uniformly distributed.

In this doctoral dissertation, we aim at addressing these two issues. Regarding the

lack of homogeneity, we argue that it is caused by the commonly accepted notion that a planning model is simply a technique to solve a specific inference problem, rather than a state-space model over which a palette of different inference (and learning) problems can be formulated naturally. Briefly, a state-space model is a discrete-time, dynamic model consisting of a transition model that describes the evolution of the actor state, and an observation model that describes how the observer perceives the actor state at each time step. Indeed, when we look at MDPs, HMMs and BNs, the inference problems are formulated with respect to their underlying state-space models and solved with algorithms that exploit their structures, resulting in a much more cohesive view. In that regard, we believe that the model-based approach to Goal Recognition presented in (Ramírez & Geffner, 2010) set the foundations for a unifying formulation of inference with planning models, since this work showed that predicting the goals and plans of an actor agent can be formulated with respect to the state-space model underlying the classical planning model and solved using a planner.

The main contribution of this PhD thesis is a novel framework for inference and learning problems formulated over the state-space model underlying the classical planning model, which allows us to formulate several inference problems in a cohesive way and to apply straightforward the last advances developed by the latest off-the-shelf planners. The state-space model of classical planning and the inference problems formulated on it present a few particularities that make for an interesting case study when compared to the other models such as HMMs and BNs. For starters, classical planning uses as a **factored (and usually first-order) representation** of the transition model that compactly generalizes to a family of (possibly infinite) different state-space models depending on the number of objects and their identity. The size of the state space described by such representations is exponential in the number of state variables and, generally, cannot be enumerated. Another difference is that the classical planning model is **cost-driven** instead of stochastic, so the best solution is commonly associated to a lower cost rather than a higher probability. Last, but not least, is what we consider to be the most important difference, ***intermittency* of the observation model**, meaning that not every state/action traversed is observed. This gives rise to a scenario where the length of the true trajectory that generated the observations is unbounded as the number of *lost* observations is unknown. This is something that we cannot find in other state-space models where the inference algorithms require a known horizon for the trajectory. This last difference is not inherent to the state-space model of classical planning but rather enabled by its native solvers (planners) that are capable of finding paths of unbounded length. Our framework also incorporates a rich observation model that supports deterministic and non-deterministic sensors as well as a broader range of observations to fit the needs of the problem at hand. The observation model allows us to follow a model-based approach to check whether a sensor reading is possible with the given sensor model. In summary, our novel framework integrates planning (synthesis of unbounded sequences) with inference and learning where there may be gaps of unbounded length between the collected observations.

Using this framework as an unifying umbrella, and over the state-space model defined by the classical planning model and our observation model, we define three inference problems concerned with different aspects of the actor. The first one is the Observation Decoding problem which aims at finding the best explanation (understood as a hidden state trajectory) to some observed behaviour of the actor. Next, we define the Temporal Inference problem which generalizes some of the inference problems discussed above including Diagnosis and Goal Recognition. Temporal Inference is about finding the hypothesis that best fits the observations but hypotheses can refer to any point in time (past, present or future) and, consequently, this problem can be used to both explain the past and predict the future. Our last inference problem is the Model Recognition problem which is concerned with identifying the model of the actor among a set of candidate models.

The advantages of this framework also translate to the learning problem which in planning is commonly understood as learning the action model (the preconditions and effects of the operators) that describe the transitions of the observed agent. The primary motivation for learning action models is to solve model-based planning tasks afterwards, but there exists as well a large variety of planning-related tasks that rely upon the existence of an action model such as the inference problems discussed above. Action models are also used in *explainable AI planning* to form a common basis for communicating with users and facilitate the generation of transparent and explainable decisions (Fox et al., 2017) as well as explanations in terms of the differences with a human mental model (Chakraborti et al., 2018a). *Counterplanning* requires a model of the opponent agent in order to recognize its goals (Pozanco et al., 2018) and *Model Reconciliation* aims to conform the models of two agents with respect to an observation of a plan computed with one of the two models (Chakraborti et al., 2017). Additionally, there is common agreement in the planning community that the unavailability of an adequate domain model is a bottleneck in the applicability of planning technology to many real-world domains (Kambhampati, 2007)

Motivated by the difficulty and cost of crafting action models, research in action model learning has seen huge advances. Since the emergence of pioneer learning systems like ARMS (Yang et al., 2007), we have seen systems able to learn action models with quantifiers (Amir & Chang, 2008; Zhuo et al., 2010), from noisy actions or noisy states (Mourão et al., 2012; Zhuo & Kambhampati, 2013), from null state information (S. N. Cresswell et al., 2013), from incomplete domain models (Zhuo & Kambhampati, 2017; Zhuo et al., 2013) and many more. A limitation shared by all of these approaches is that they are unable to learn when the learning examples contain gaps (an unknown number of lost observations). This limitation is, however, easily addressed when the learning problem is formulated under our framework since support for gapped sequences is naturally provided by a planner.

1.2. Overview of contributions

In this thesis we develop a framework for inference and learning in the state-space model given by the classical planning model extended with a rich observation model. Our framework serves as a unifying umbrella that allows a crisp and cohesive formulation of inference and learning problems and their solutions. Moreover, as discussed above, inference in the state-space model of classical planning presents some distinguishing and challenging features that are not found in other state-space models, being the lack of a bounded horizon solution the most interesting one.

The main contributions of this thesis are then the framework itself, as well as the inference and learning problems formulated under it:

1. Observation Decoding is the problem concerned with finding the most likely explanation to some observed behaviour. Solving the Observation Decoding problem presents two challenges: i) computing the explanations, and ii) a mechanism to rank explanations that allows us to identify and compute the most likely one. We address both challenges with a planning-based approach that interprets the observations as an LTL property. This allows us to compute a Deterministic Finite Automaton (DFA) that monitors the property that embodies the observations which we compile away into a planning problem. An explanation is any solution to this planning problem, as the DFA is effectively restricting the possible trajectories of the actor to those that satisfy the observations. Ranking the explanations is done on the basis of the cost functions associated to the model of the actor and the observation model.

2. Temporal Inference is a hypothesis-based formulation that generalizes the problem of estimating the past (e.g., for finding explanations), present (e.g., for diagnosis and monitoring), or future state (e.g., for prediction and goal recognition) of an agent based on some observations of its behaviour. We use LTL as a language for expressing hypotheses, which allows us to specify expressive hypotheses and also exploit the same ideas used for Observation Decoding. The solution of a Temporal Inference problem is the most likely hypothesis, which we define as the one supported by the best explanation following the principles of Inference to the Best Explanation (Harman, 1965; Schupbach, 2017).

3. The goal of Model Recognition is to figure out the model of the actor from a set of candidate models. The solution is the candidate model more likely to generate the observation sequence. Our definition of most likely model follows a double criteria that considers 1) the capacity of the candidate model to explain the observations, and 2) the modifications required by the model in order to generate an explanation. Our planning-based approach leverages a novel encoding that allows representing any STRIPS action model using a finite set of variables. This encoding is incorporated into the compilation to planning which allows the planner to make

the necessary modifications to the candidate model and search for an explanation in a single planning episode.

4. Learning is about estimating the model of the actor from a set of learning examples so that the resulting model can explain the examples. Our planning-based approach to learning leverages the same encoding used in Model Recognition which allows a planner to modify the input model. We will also serialize the learning examples into a single sequence so that we can build a monitor DFA for all of them and compile them away. The resulting planning problem will have as solutions plans that first build the output model and then generates a serialized explanation for all the learning examples.

1.3. A Note on Software

We make fully available the source code for our planning-based approach at this repository *https://github.com/daineto/meta-planning*. The repository also includes most of the evaluation scripts and the benchmarks used to generate our experimental data.

1.4. Declaration of Previous Work

This thesis is based on the following previously published material:

- Diego Aineto, Sergio Jiménez and Eva Onaindia. *Learning STRIPS action models with classical planning*. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS-18)*, pages 399-407. AAAI Press, 2018. [Chapter 7]
- Diego Aineto, Sergio Jiménez, Eva Onaindia and Miquel Ramírez. *Model recognition as planning*. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS-19)*, pages 13-21. AAAI Press, 2019. [Chapter 6]
- Diego Aineto, Sergio Jiménez Celorrio and Eva Onaindia. *Learning action models with minimal observability*. In *Artificial Intelligence Journal (AIJ)*, volume 275, pages 104-137, 2019. [Chapter 7]
- Diego Aineto, Sergio Jiménez Celorrio and Eva Onaindia. *Explanation-Based Learning of Action Models*. In *Knowledge Engineering Tools and Techniques for AI Planning*, pages 3-20. Springer, 2020. [Chapter 7]
- Diego Aineto, Sergio Jiménez and Eva Onaindia. *Observation decoding with sensor models: Recognition tasks via classical planning*. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS-20)*, pages 11-19. AAAI Press, 2020. [Chapter 4].

- Diego Aineto, Sergio Jiménez and Eva Onaindia. *Generalized Temporal Inference via Planning*. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR-2021)*, pages 22-31. AAAI Press, 2021. [Chapter 5].

- Diego Aineto, Sergio Jiménez and Eva Onaindia. *A comprehensive framework for learning declarative action models*. In *Journal of Artificial Intelligence Research (JAIR)*, volume 74, pages 1091-1123, 2022. [Chapter 7]

Additionally, the Observation Decoding problem presented in Chapter 4 has been recently extended in the following publication:

- Diego Aineto, Eva Onaindia, Miquel Ramírez, Enrico Scala and Ivan Serina. *Explaining the Behaviour of Hybrid Systems with PDDL+ Planning*. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI-22)*, 2022 (accepted).

1.5. Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 covers the necessary background on classical planning. The chapter starts by providing a formal account of the classical planning problem and its semantics, as well as of the planning languages. We will also describe the most relevant approaches to solve classical planning problems and briefly present the main ideas of some extensions that are relevant to this thesis.

In Chapter 3 we present our framework for inference and learning. The chapter provides a thorough formalization of the model of the observer, a clear description of fundamental assumptions that shape our framework, and a brief introduction to the inference and learning problems that will be developed in later chapters.

Chapters 4, 5, and 6 are dedicated, respectively, to Observation Decoding, Temporal Inference, and Model Recognition, and all follow the same structure. These chapters start with a working example that serves to illustrate the problem and aid explanations. Following, a formal definition of the problem and its solution is provided and extensively justified. Afterwards, we present our planning-based approach to address the problem which usually entails compiling some elements of the inference problem (like the observations) away into one or more classical planning problems. Finally, we experimentally evaluate our approach and briefly discuss some related topics and works.

The learning problem is presented in Chapter 7 and its structure is not all that different from the previous chapters. The main difference is that instead of a working example, we start the chapter with a quick discussion of the state-of-the-art and motivating the use of planning for learning.

Finally, we conclude with Chapter 8 providing a summary of contributions and results, and outlining open challenges and future lines of research related to this thesis.

Part II

Background

2. CLASSICAL PLANNING

In this chapter, we will introduce the basic definitions of classical planning that will be used throughout the manuscript as well as an overview of the main approaches and topics of classical planning most closely related to this contribution.

2.1. The Classical Planning Problem

Classical planning is the problem of finding a sequence of actions that starting from an initial state reaches a goal state. In classical planning the initial state is *fully known* and the effects of actions are *deterministic*. Classical planning can be understood as a path-finding problem in the directed graph whose nodes represents states and whose edges represent actions. Following we formalize all the components of a classical planning problem and the underlying finite state model.

Let $X = \{x_1, \ldots, x_k\}$ be a finite set of state variables, where each $x \in X$ is associated with a finite domain $D(x)$. A valuation is a function $v : X \to D(X)$ from X to a particular set of values in the domain $D(X)$[1]. Additionally, given a valuation v and $Z \subseteq X$ we define $v_Z : Z \to D(X)$ by $v_Z(x) = v(x)$ for every $x \in Z$.

Definition 1 (Constraint)**.** *A constraint over X is a pair $\langle Z, R\rangle$ where $Z \subseteq X$ is a subset of k variables and R is a k-ary relation on the domains of the variables Z. We denote by* $\mathsf{Constr}(X)$ *the class of constraints over the set of variables X.*

Definition 2. *Given a valuation $v : X \to D(X)$ and a constraint $\varphi = \langle Z, R\rangle$ belonging to* $\mathsf{Constr}(X)$*, we write $v \models \varphi$ and say that v satisfies φ, if $v_Z \in R$.*

Let us illustrate Definitions 1 and 2 with a small example that assumes the set of variables $X = \{x_1, x_2, x_3\}$ with domain $D(X) = \{a, b\}$. As a constraint let us use "$x_1 \neq x_3$" expressed by $\varphi = \langle Z, R\rangle$ where $Z = \{x_1, x_3\}$ and $R = \{(a, b), (b, a)\}$. Given the valuations $v = (a, b, b)$ and $v' = (b, a, b)$, we say that v satisfies φ but v' does not. This is because $v_Z = (a, b)$ is in R while $v'_Z = (b, b)$ is not.

Definition 3 (Planning Problem)**.** *A planning problem is a tuple $\mathcal{P} = \langle X, A, \mathsf{App}, \mathsf{Eff}, s_0, G\rangle$ where:*

- *$X = \{x_1, \ldots, x_k\}$ is a finite set of multivalued variables,*
- *A is a finite set of actions,*
- *$\mathsf{App} : A \to \mathsf{Constr}(X)$ is the applicability constraint and it describes the condition in terms of variables in X that allow an action to be executed.*

[1] For ease of notation, we will sometimes assume that all state variables X share the same domain $D(X)$

- $\mathsf{Eff} : A \to \mathsf{Constr}(X \cup X^+)$ *with* $X^+ = \{x_1^+, \ldots, x_k^+\}$ *is the effects constraint. The variables in* X^+ *refer to the updated values of the state variables after the execution of an action. The effects constraint describes changes in variables in* X^+ *in terms of variables in* X*.*

- $s_0 : X \to D(X)$ *is the known initial state.*

- $G \in \mathsf{Constr}(X)$ *is the goal condition.*

We note that this definition of planning problem implicitly defines a state s as a valuation over X, that is, $s : X \to D(X)$. The value of a state variable $x \in X$ in a state s is then given by $s(x)$. Additionally, we refer by *action model* to the part of a planning problem that abstracts the initial situation and goal condition.

Definition 4 (Action model)**.** *Given a planning problem* $\mathcal{P} = \langle X, A, \mathsf{App}, \mathsf{Eff}, s_0, G\rangle$*, its action model is the tuple* $M_a = \langle X, A, \mathsf{App}, \mathsf{Eff}\rangle$*.*

The definitions above only define the components of a planning problem, but not the behaviour of the system that they describe. Following we define the semantics of a planning problem as a state-space model.

Definition 5 (Semantics of a Planning Problem)**.** *The semantics of a planning problem* $\mathcal{P} = \langle X, A, \mathsf{App}, \mathsf{Eff}, s_0, G\rangle$ *is the state-space model* $\mathcal{S}(\mathcal{P}) = \langle S, s_0, S_G, A, f\rangle$ *where:*

- $S = \{s \mid s \in D(X)^X\}$ *is the finite and discrete state space*[2]*,*

- $s_0 \in S$ *is the known initial state,*

- $S_G = \{s \in S \mid s \models G\}$ *is the set of goal states,*

- A *is the set of actions,*

- *and* $f : S \times A \to S$ *is the deterministic transition function where* $s' = f(a, s)$ *is the state that follows* s *after executing* $a \in A$ *and it holds that* $s \models \mathsf{App}(a)$ *and* $(s, s') \models \mathsf{Eff}(a)$*.*

A plan in the state model $\mathcal{S}(\mathcal{P})$ is a finite sequence of actions $\pi = (a_1, \ldots, a_n)$ that generates the state trajectory $\tau = (s_0, \ldots, s_n)$ such that for all $i \in \{1, \ldots, n\}$ $s_i = f(s_{i-1}, a_i)$. We define the state reached after the execution of a plan recursively as follows:

$$s_0[\pi] = f(f(\ldots f(f(s_0, a_1), a_2)\ldots), a_n)$$

Similarly, we define the trajectory generated by a plan as:

$$s_0[\![\pi]\!] = (s_0, s_0[\pi_{1:1}], s_0[\pi_{1:2}], \ldots, s_0[\pi])$$

[2]The notation $D(X)^X$ in set theory refers to the class of functions $s : X \to D(X)$, i.e., the set of all valuations over X.

We denote by $\Pi(M_a, s_0)$ (resp. $\mathcal{T}(M_a, s_0)$) the set of plans (resp. trajectories) that can be generated with the action model M_a starting from s_0. A solution plan is one that reaches a goal state, i.e., such that $s_n \in S_G$. We denote by $\Pi(M_a, s_0, G)$ or $\Pi(\mathcal{P})$ the set of solution plans for a planning problem $\mathcal{P}$ and by $\mathcal{T}(M_a, s_0, G)$ or $\mathcal{T}(\mathcal{P})$ the set of trajectories generated by $\Pi(\mathcal{P})$. Generally, solution plans with lower length $|\pi|$ are preferred.

A planning problem may be extended with a cost function $c : A \to \mathbb{R}^+$ that assigns non-negative costs to the actions of the problem. Plan costs can be computed by accumulating the costs of all actions in the plan.

Definition 6 (Plan cost)**.** *The cost of a plan* $\pi = (a_1, \ldots, a_n)$ *is*

$$cost(\pi) = \sum_{i=1}^{n} c(a_i)$$

When action costs are considered, plans with lower costs are preferred over plans with higher costs. We say that a solution plan π is *optimal* if it has minimum cost, i.e, $cost(\pi) \leq cost(\pi')$ for all $\pi' \in \Pi(\mathcal{P})$. Planning without costs can be considered a special case of planning with costs where all actions have unitary cost, i.e., $\forall a \in A\ c(a) = 1$, which leads to solution plans with lower length $|\pi|$ being preferred. Hereafter, we will assume that planning problems and action models always include a cost function.

2.2. The Planning Domain Definition Language

The most common representation in classical planning uses Boolean state variables that assert whether a proposition about the world holds in a given state. This representation, known as Strips (Fikes & Nilsson, 1971), is the one that has attracted most of the work in classical planning. More recently, the *Planning Domain Definition Language*, PDDL (D. McDermott et al., 1998) has become the *de facto* standard to express classical planning problems, primarily, because it is the language used in the *International Planning Competitions* (IPC) (Bacchus, 2001; Gerevini et al., 2009; Hoffmann & Edelkamp, 2005; Long & Fox, 2003; López, Celorrio, & Olaya, 2015; D. V. McDermott, 2000). PDDL is a relational language with a finite number of predicates, actions, variables and objects used to represent Strips problems as well as extensions of Strips. Classical planners *ground* predicates and actions by substituting variables for objects to achieve a propositional representation like Strips.

In the following, we formalize the Strips fragment of PDDL (D. V. McDermott, 2000) as well as the extension that considers *conditional effects*.

2.2.1. The Strips Fragment of PDDL

Let V be a set of variable symbols and P be a set of predicate symbols such that each predicate $p \in P$ has arity $ar(p) \in \mathbb{N}$. We start by defining a *schematic state variable* as a

combination of a predicate and a tuple of variable symbols of the same arity.

Definition 7 (Schematic State Variable)**.** *A schematic state variable is of the form* $\breve{x} = p(args)$ *where* $p \in P$ *and* $args \in V^{ar(p)}$*. We denote by* $\breve{X}(P, V)$ *the class of schematic variables defined over P and V.*

Schematic state variables are useful for the purpose of specifying actions in a relational way. A relational specification of an action, which we refer to as *operator*, defines all its components in terms of a subset of variables from V. A characteristic of STRIPS operators is the closed-world representation via a set of variables that implicitly represent conjunctive constraints.

Definition 8 (Operator)**.** *A* STRIPS *operator over V and P is a tuple* $\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del} \rangle$ *where*

- name *is the name of the operator,*
- $\mathsf{Pars} \subseteq V$ *is a set of parameters,*
- $\mathsf{Pre} \subseteq \breve{X}(P, \mathsf{Pars})$ *is the precondition list,*
- $\mathsf{Add} \subseteq \breve{X}(P, \mathsf{Pars})$ *is the add list,*
- $\mathsf{Del} \subseteq \breve{X}(P, \mathsf{Pars})$ *is the delete list.*

STRIPS *constraints require that* $\mathsf{Del} \subseteq \mathsf{Pre}$*,* $\mathsf{Del} \cap \mathsf{Add} = \emptyset$ *and* $\mathsf{Pre} \cap \mathsf{Add} = \emptyset$*.*

Grounding is the process by which predicates and operators are instantiated using a set of objects O. Grounding a predicate results in a propositional state variable.

Definition 9 (State Variable)**.** *A state variable is of the form* $p(args)$ *where* $p \in P$ *and* $args \in O^{ar(p)}$*. We denote by* $X(P, O)$ *the class of state variables defined over P and O.*

The grounding of an operator $\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del} \rangle$ with a set of objects $args \subseteq O$, $|args| = |\mathsf{Pars}|$, results in the *grounded operator* $\breve{a}(args) = \langle \mathsf{name}, args, \mathsf{Pre}(args), \mathsf{Add}(args), \mathsf{Del}(args) \rangle$ where all occurrences of the i-th parameter in Pars are replaced by the i-th argument in $args$.

A grounded operator $\breve{a}(args) = \langle \mathsf{name}, args, \mathsf{Pre}(args), \mathsf{Add}(args), \mathsf{Del}(args) \rangle$ can be seen as an *action with structure* that encodes action a as well as its applicability $\mathsf{App}(a)$ and effects $\mathsf{Eff}(a)$ constraint in the following way:

- Its name and arguments are used as *label* or identifier of action a, i.e., $a = \mathsf{name}(args)$.
- The precondition list $\mathsf{Pre}(args)$ defines the propositional state variables in $X(P, O)$ that must be evaluated to $\top$ in order for action a to be applicable and encodes the applicability constraint

$$\mathsf{App}(a) = \langle \mathsf{Pre}(args), \{(\top, \ldots, \top)\} \rangle \tag{2.1}$$

- Add (resp. Del) defines the propositional state variables in $X(P,O)^+$ that are evaluated to $\top$ (resp. $\bot$) after the execution of action a. The add and delete lists together encode the effects constraint

$$\mathsf{Eff}(a) = \langle \mathsf{Add}(args)^+ \cup \mathsf{Del}(args)^+, \{(\top, \ldots, \top, \bot, \ldots, \bot)\} \rangle \tag{2.2}$$

Then, the state transition function is defined as $f(a, s) = s'$ such that

$$s'(x) = \begin{cases} \top & x \in \mathsf{Add}(args) \\ \bot & x \in \mathsf{Del}(args) \\ s(x) & x \notin \mathsf{Add}(args) \wedge x \notin \mathsf{Del}(args) \end{cases} \tag{2.3}$$

so that the resulting state s' is computed by setting state variables in $\mathsf{Add}(args)$ to $\top$, variables in $\mathsf{Del}(args)$ to $\bot$, and leaving the rest of variables as they were in s.

Definition 10 (STRIPS Planning Problem)**.** *A* STRIPS *planning problem* $\breve{\mathcal{P}} = \langle P, V, O, \breve{A}, s_0, G \rangle$ *consists of*

- *a finite set* P *of predicates,*
- *a finite set* V *of schema variables,*
- *a finite set* O *of objects*
- *a finite set* $\breve{A}$ *of operators over* V *and* P
- *a initial state* $s_0 : X(P,O) \to \{\top, \bot\}$
- *a goal condition* $G \in \mathsf{Constr}_\wedge(X(P,O))$

A STRIPS planning problem $\breve{\mathcal{P}} = \langle P, V, O, \breve{A}, s_0, G \rangle$ encodes the planning problem $\mathcal{P} = \langle X(P,O), A(\breve{A}, O), \mathsf{App}, \mathsf{Eff}, s_0, G \rangle$ in the following way:

- $X(P,O)$ is the set of propositional state variables obtained by grounding predicates P with objects O.
- $A(\breve{A}, O)$ is the set of actions encoded by the grounding of the operators in $\breve{A}$ with objects O.
- App is defined for every action in $A(\breve{A}, O)$ following equation 2.1.
- Eff is defined for every action in $A(\breve{A}, O)$ following equation 2.2.
- s_0 is a valuation over the propositional state variables $X(P,O)$.
- G is a constraint over the propositional state variables $X(P,O)$.

As we did for a classical planning problem, we refer to the part of a STRIPS problem that abstracts the situation and goals as *schematic action model* and denote it by $\breve{M}_a = \langle P, V, \breve{A} \rangle$.

The PDDL specification (Fox & Long, 2003; D. V. McDermott, 2000) describes a number of extensions including those involving negated fluents, conditional effects and disjunctive and quantified (existential or universal) preconditions and effects. These extensions can be compiled into the present formulation of a STRIPS planning problem and, in fact, most of the state-of-the-art planners support such extensions either natively or through compilations.

2.2.2. Conditional Effects

Conditional effects are fundamental to the formulation of the problems described in this thesis so we will formalize them. Conditional effects extend the expressiveness of STRIPS by allowing to describe the state variables X^+ of a successor state in terms of the current state variables X. While conditional effects can be compiled away into regular STRIPS the cost of such compilation is an exponential blow up in the size of the problem or a polynomial increase in the plan length (Nebel, 2000; Rintanen, 2003). Consequently, classical planners that natively support conditional effects perform better than those that compile them away.

Definition 11 (Conditional effect)**.** *A conditional effect over predicates P and schema variables V is a tuple* $\mathsf{ce} = \langle \mathsf{CPre}, \mathsf{CAdd}, \mathsf{CDel} \rangle$ *where* CPre*,* CAdd*,* CDel *are subsets of* $\breve{X}(P, V)$.

CPre[3] defines the condition which is required to be true in order to trigger the effects described by CAdd and CDel.

Definition 12 (Operator with conditional effects)**.** *An operator with conditional effects over predicates P and variable symbols V is a tuple*

$$\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del}, \mathsf{CE} \rangle$$

where name, Pars, Pre, Add, Del *are as defined in Definition 8 and* CE *is a set of conditional effects over P and* Pars.

Let a be the action encoded by the grounded operator $\breve{a}(args) = \langle \mathsf{name}, args, \mathsf{Pre}(args), \mathsf{Add}(args), \mathsf{Del}(args), \mathsf{CE}(args) \rangle$. We denote by $Trigger(s, a)$ the subset of conditional effects triggered by the execution of a in s

$$Trigger(s, a) = \{\mathsf{ce}(args) \in \mathsf{CE}(args) \mid s \models \langle cpre(\mathsf{ce}(args)), \{\langle \top, \ldots, \top \rangle\}\rangle\}$$

[3]The conditional effects extension of PDDL also allows defining CPre as a universally quantified formula

where function *cpre* returns the component CPre of the given conditional effect. Then, the transition function of equation 2.3 is now defined as

$$s'(x) = \begin{cases} \top & x \in Adds(s,a) \\ \bot & x \in Dels(s,a) \\ s(x) & x \notin Adds(s,a) \wedge x \notin Dels(s,a) \end{cases} \tag{2.4}$$

where $Adds(s,a)$ is the subset of state variables that are evaluated to $\top$ after the execution of a in s

$$Adds(s,a) = \mathsf{Add}(args) \cup \bigcup_{\mathsf{ce}(args) \in Trigger(s,a)} cadd(\mathsf{ce}(args))$$

and $Dels(s,a)$ is the set of state variables that are evaluated to $\bot$ after the execution of a in s

$$Dels(s,a) = \mathsf{Del}(args) \cup \bigcup_{\mathsf{ce}(args) \in Trigger(s,a)} cdel(\mathsf{ce}(args))$$

where functions *cadd* and *cdel* return respectively the components CAdd and CDel of a given conditional effect.

2.3. Complexity of Strips Planning

The two decision problems related to classical planning are 1) `PlanEx` concerned with the complexity of finding a plan reaching a goal state, and 2) `PlanCost` concerned with the complexity of finding solution plans below a given cost.

Definition 13 (`PlanEx`)**.** *Given a classical planning problem P,* `PlanEx`(P) *is the decision problem defined by the question*

Does a solution plan π for P exist?

Definition 14 (`PlanCost`)**.** *Given a classical planning problem P and a constant $k \in \mathbb{R}_0^+$,* `PlanCost`(P,k) *is the decision problem defined by the question*

Does a solution plan π for P with $cost(\pi) \leq k$ exist?

Following the characterization of the complexity of classical planning problem in the Strips representation created by Bylander (1994), both decision problems are PSPACE-complete, that is, the class of problems that can be solved in polynomial space with no restrictions on running time. These complexity results, however, are at odds with the practical results of planning where problems that have been shown to be in NP and even in P

are commonly used as benchmarks (Helmert, 2003, 2006b). More recently, (Bäckström & Jonsson, 2011) narrows the gap between theory and practice by introducing new methods for complexity analysis of planning that seek to provide more precision and complexity separations than previous methods allowed.

2.4. Main Approaches to Classical Planning

This section reviews the state-of-the-art approaches to classical planning developed in recent years, namely, *heuristic search* and *propositional satisfiability*.

2.4.1. Classical Planning as Heuristic Search

A solution plan for a classical planning problem $\mathcal{P}$ can be seen as a path in the state space model $\mathcal{S}(\mathcal{P})$ whose nodes represent states and whose edges represent actions. Standard graph search algorithms, such as Dijkstra (Cormen et al., 1989), can be used to find such solutions. However, blind search algorithms require enumerating the state space, which is exponential in the number of state variables, making them ineffective.

In contrast, a more successful approach is the use of heuristic search algorithms that make use of model-based heuristics automatically extracted from the structure of STRIPS planning problems. This success is evidenced by the fact that the IPC is dominated by planners implementing the planning as heuristic search approach (Bonet et al., 1997; Helmert, 2006a; Hoffmann & Nebel, 2001; Lipovetzky & Geffner, 2017; Richter & Westphal, 2010). Heuristics are often computed by solving relaxations of the original problem. Among the possible relaxations, the delete relaxation which assumes that actions have no negative effects (empty delete lists) is very prominent in classical planning. Following we comment on some effective heuristics.

- **Max and additive heuristics**: The h_{max} and h_{add} heuristics estimate the cost of a goal condition by decomposing it into atomic formulas and estimating cost of each atom independently (Bonet & Geffner, 2001). Then, the h_{max} heuristic uses the maximum of such costs, while the h_{add} uses the sum of them all. The h_{max} heuristic is admissible and can be used to find optimal plans but its practical performance is lower than h_{add} for *satisficing planning* that only seeks to find a solution plan.

- **The FF heuristic**: The h_{FF} heuristic uses the length of a relaxed plan, extracted from the *relaxed planning graph* (Hoffmann & Nebel, 2001), as an estimation for the distance to a goal state.

- h^m **heuristics**: The h^m heuristics generalizes h_{max} by assuming that the cost of a goal G is the cost of the most expensive m-tuple of atoms in G. In practice, the computation of h^2 is too expensive but, if it is only computed in the initial state,

it can be useful for optimal planning (Haslum & Geffner, 2000) and for capturing mutexes (Blum & Furst, 1995).

- **Landmark heuristics**: Landmarks are necessary features of any solution plan (Hoffmann et al., 2004) and they can refer to both state variables and actions. Landmark heuristics compute the landmarks of a problem in the initial state and then estimate the distance from the current state to a goal state as the number of landmarks that still need to be achieved.

The downside of the heuristic approach is that the search process can get stuck in dead ends when heuristics fail to identify them. Related to this is the topic of unplannability (Muise & Lipovetzky, 2015), that aims to prove that a planning problem is unsolvable and has direct applications for the recognition of dead ends.

We conclude this section with a brief description of the two heuristic planners used in this thesis: FF (Hoffmann & Nebel, 2001) and Fast Downward (Helmert, 2006a). The FF planner uses the h_{FF} heuristic extracted from the relaxed plan as well as the relaxed plan itself. First, FF implements an incomplete but very effective *greedy search* that only considers actions of the relaxed plan. Then, if it fails to find a solution, it launches a complete Greedy Best First Search guided by the h_{FF} heuristic. The Fast Downward planner implements several advanced techniques such as multi-queue search for combining different heuristics, helpful actions (preferred operators), and anytime search for improving plan quality. Nowadays, Fast Downward has become an extensible planning framework that offers a large quantity of search methods and heuristics over which it is easy to develop new techniques and heuristics.

2.4.2. Classical Planning as Propositional Satisfiability

The planning as propositional satisfiability approach (Kautz & Selman, 1992, 1996) maps a planning problem $\mathcal{P}$ into a propositional formula $C(\mathcal{P}, N)$ that encodes all solution plans for $\mathcal{P}$ of up to N time steps. Then, the problem of finding a solution plan for $\mathcal{P}$ can be solved by progressively increasing N until a model of $C(\mathcal{P}, N)$, representing a solution plan π of N time steps, is found. In contrast, if no model is found, it can be said that the problem is unsolvable at that horizon. We note that the length of the solution plan $|\pi|$ may be greater than N as a single time step may involve the parallel execution of multiple actions.

The formula $C(\mathcal{P}, N)$ uses propositional variables annotated with a temporal index in $0, \ldots, N$ to represent both the state variables and actions of $\mathcal{P}$ at a given time step. $C(\mathcal{P}, N)$ is a propositional formula in *Conjunctive Normal Form* (CNF), represented as a set of clauses. Clauses are used to represent the initial state and goal conditions of a planning problem $\mathcal{P}$ as well as the applicability and effects constraints of the set of actions. Additional clauses are defined to preserve the value of state variables not modified by an action and to forbid the parallel execution of incompatible actions.

With the recent advances in the area of propositional satisfiability, state-of-the-art SAT solvers are now able to deal with thousands of variables and hundreds of thousands of clauses. As a result, planning as SAT has become feasible and can be considered a compelling alternative to the planning as heuristic search approach. In particular, planning as SAT is a competitive approach for optimal classical planning even though such approaches minimize the parallel length, *makespan*, instead of the number of actions in the plan. Moreover, it is well known that the SAT approach outperforms heuristic search in domains with large dead-ends (Hoffmann et al., 2007).

In this thesis we use the SAT-based planner Madagascar (Rintanen, 2014) that implement several refinements that improve the performance of planning as SAT. In particular, Madagascar uses a heuristic for variable selection in the SAT solver, an improved planning horizon search and better memory management.

As closing thoughts we highlight some limitations that affect both the heuristic search and SAT approach. The first limitation has to do with the ground representation of a planning problem as it can sometimes become a bottleneck if the grounding is too large to compute without running out of memory. To address this issue, recent investigations work on the computation of heuristics directly from the lifted representation of the problem (Corrêa et al., 2021). Another line of investigation to overcome the limitation of searching in huge state spaces is generalized planning, which proposes to search in compact solution spaces such as programs or policies (Aguas et al., 2021).

2.5. Planning with Temporally Extended Goals

Classical planning is the simplest model for automated planning that makes many assumptions. More expressive planning models, such as non-deterministic planning, temporal planning, or planning with temporally extended goals (TEGs), relax these assumptions. Among these models, planning with TEGs is of special interest in the context of this thesis because, as we will show in Chapter 4, observations of an actor agent can be assimilated to TEGs.

In planning, a goal is traditionally understood as a property that a final state must satisfy. Temporally extended goals (TEGs) extend this notion to express properties that must hold over intermediate and/or final states of the trajectory generated by a solution plan. In other words, TEGs describe the desired qualities of a solution instead of qualities of the state reached by a solution. TEGs are compelling because many real-world planning problems involve complex goals such as achieving subgoals following some order, safety and liveness constraints, and deadlines. This has led to TEGs being commonly expressed with rich logical languages such as Linear Temporal Logic (LTL) (Pnueli, 1977).

Planning with LTL goals has been addressed by using blind search on a search space that is constantly being pruned via goal progression (Bacchus & Kabanza, 1998; Kvarnström & Doherty, 2000). Another line of research advocates for a compilation-based

approach that translates a planning problem with an LTL goal into a classical planning problem (Baier & McIlraith, 2006; S. Cresswell & Coddington, 2004; Patrizi et al., 2011; Rintanen, 2000). This latter approach generally outperforms the former as it is able to leverage state-of-the-art classical planning technology.

The compilation-based approach to planning with TEGs involves a double translation step. First, the LTL formula representing the TEG is translated into an automaton that accepts state trajectories that satisfy the TEG. Then, the planning problem is augmented to include the initial location of the automaton as part of the initial state and the accepting conditions of the automaton as part of the goal condition. The transition system of the planning problem is similarly augmented to encode the transitions of the automaton.

2.6. Planning with Sensing

A sensing process is often imprecise, either because the information provided by the observations is partial or because it is incorrect. The discrepancy about the actual state of the actor and what an observer is able to gather generates *uncertainty*. One way to deal with uncertainty, and the one followed in this thesis, is to compile uncertainty away into a classical planning problem (Albore et al., 2009; Hoffmann & Brafman, 2005). There are, however, other planning models that deal with uncertainty about the current state. We dedicate this section to briefly discuss planning with sensing and its associated models.

In planning with sensing the true state of the system is not assumed to be fully known. Instead, sensors are used to obtain partial information about the states. Uncertainty is represented by sets of states, referred to as *beliefs* that contain all states consistent with our partial knowledge of the state. The solution of a planning with sensing problem is a choice of actions, often expressed as a *policy*, that ensures that all resulting trajectories reach a goal belief, that is, a belief state where all the possible states are goal states.

The logical model for planning with sensing is known as *contingent planning* (Peot & Smith, 1992; Pryor & Collins, 1996). Contingent planning extends classical planning by 1) allowing uncertainty in the initial situation and in the transition function, and 2) by the addition of a *sensing model* that maps state-action pairs into a set of possible observation. The observation provides information about the true state of the system as it rules out any state that does not map into it. A policy that solves a contingent planning problem can be computed with dynamic programming using the *Value Iteration* (VI) method (Bellman, 1957; Bertsekas, 1995). The problem with VI is that it is an exhaustive method that considers the full set of belief states which is exponential in the number of variables. An alternative solution method formulates policies as solutions of an acyclic AND/OR graph that can be solved by the heuristic method AO* (Bonet & Geffner, 2000; Bryce et al., 2006; Nilsson, 1980; Pearl, 1984).

A probabilistic alternative for the planning with sensing problem is given in the form of Partially Observable Markov Decision Processes (POMDPs) (Kaelbling et al., 1998).

POMDPs generalize MDPs by allowing partially observable states and using a probabilistic sensor model that maps the true but possibly hidden state into observations. A POMDP can be seen as an MDP over belief states where beliefs are now interpreted as probability distributions over the set of states. As a consequence, the number of belief states is infinite but policies can still be computed using the VI implementation enabled by Sondik's results (Smallwood & Sondik, 1973).

Part III

The Learning and Inference Framework

3. FRAMEWORK

This chapter presents our main contribution, a framework for inference and learning where the underlying state-space model is implicitly described by an action model. The purpose of the proposed framework is to present an unifying umbrella under which inference and learning problems can be defined and solved following homogeneous procedures. Our framework integrates a model of action (the actor) that describes the agent, system or environment being observed, and a model of observation of an agent that observes what the actor does (the observer). We assume that the behaviour of the actor agent is governed by an action model that implicitly represents all the possible behaviours that the actor may follow. Similarly, we assume that the observer perceives the world through a sensor model that determines the observations that may be captured. Since this is a framework for inference and learning, the observer is assumed to be a passive agent that cannot control the actor; the observer's role is limited to collecting data and using this data to obtain some knowledge about the actor.

In this chapter, we first introduce the fundamental notions to formalize the sensor model of the observer, establishing the connection between the sensor model and the model of action of the actor. Then, we provide a high-level unifying view of the problems that are approachable with our framework, which will be explained in depth in the subsequent chapters.

3.1. The Observer

The observer describes the entity whose role is to collect data on the actor and use this data to answer some questions or ascertain some aspects about the actor. In our framework, the observer is described through a sensor model that abstracts the actual sensing process. The sensor model describes the information that a sensor is able to provide and how this information is limited by the environment. Consequently, while the collected data stem from a state of the actor, the information of such actor's state that the sensor model actually conveys is *partial* and possibly *noisy*. Following, we formalize the sensor model and its components.

3.1.1. The Sensor Model

First, we define the observable variables of the observer as a set $Y = \{y_1, \ldots, y_l\}$, where each $y \in Y$ is associated with a finite domain $D(y)$. The set of observable variables is disjoint from the state variable of the actor ($X \cap Y = \emptyset$) and allows us to introduce a second level of representation. In this way, observable variables can be used to represent features and abstractions that provide partial information about the actor state. Even if in

some domains some state variables might be observable, defining X and Y as disjoint sets allows us to distinguish distinguish between the true state, given by the state variables, and the perceived state, given by the observable variables. We refer to a valuation over observable variables as *observation*.

Definition 15 (Observation)**.** *An observation $o : Y \to D(Y)$ is a valuation over the observable variables Y. We will use the notation $o = \langle y_1 = w_1, \ldots, y_m = w_m \rangle$ when we want to make explicit the values emitted for each observable variable in an observation.*

Consider, for example, the set of observable variables $Y = \{\texttt{color}, \texttt{texture}\}$ with $D(\texttt{color}) = \{\texttt{white}, \texttt{black}\}$ and $D(\texttt{texture}) = \{\texttt{rough}, \texttt{smooth}\}$. A possible observation would be $o = \langle \texttt{color} = \texttt{black}, \texttt{texture} = \texttt{rough} \rangle$. Observing the execution of the actions of the actor's plan is a common practice in planning-related inference problems. This is enabled by assuming that the set of state variables X includes a special variable `action` with domain $D(\texttt{action}) = \{v_a\}_{a \in A}$ which always takes value $\texttt{action} = v_a$ after the execution of action a.

An observation by itself is not sufficient to infer the state of the actor. It is necessary to know how state variables and observable variables connect to each other. A *sensing function* relates a subset of state variables $Z \subseteq X$ to an observable variable $y \in Y$ by defining the constraints over Z that produce an emission $w \in D(y)$ for y.

Definition 16 (Sensing Function)**.** *Let $Z \subseteq X$ be the subset of state variables that determines the value of an observable variable $y \in Y$; a sensing function for y $\mathsf{sense}_y : \mathsf{Constr}(Z) \to D(y)$ maps constraints over Z into observed values in $D(y)$.*

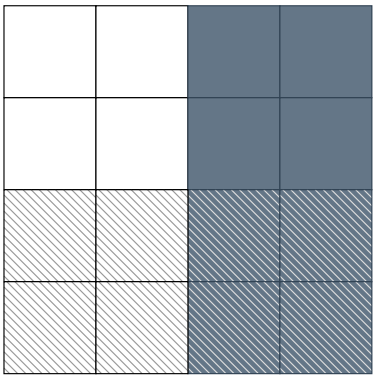

Figure 3.1: Example of a 4×4 grid featuring tiles with `white` and `black` color, and `smooth` and `rough` texture.

Continuing with our previous example assume that the state variables of the actor are given by the set $X = \{\texttt{x\_coord}, \texttt{y\_coord}\}$ with domain $D(X) = \{1, \ldots, 4\}$ representing the location of the actor in the 4×4 grid illustrated in Figure 3.1. We have a sensor for the observable variable `color` that emits the value `white` when the actor is in the left half-side of the grid and `black` when it is in the right half-side. Regarding the sensor for the observable variable `texture`, it emits the value `smooth` when the actor is in the top

half-side of the grid and `rough` when it is in the bottom half-side. The sensing function for `color` is formally defined as

$$\begin{aligned} \mathsf{sense}_{\mathtt{c}}(\{\mathtt{x\_coord}\}, \text{"}\mathtt{x\_coord} \leq 2\text{"}) &= \mathtt{white} \\ \mathsf{sense}_{\mathtt{c}}(\{\mathtt{x\_coord}\}, \text{"}\mathtt{x\_coord} \geq 3\text{"}) &= \mathtt{black} \end{aligned} \tag{3.1}$$

where "`x_coord` $\leq$ 2" succinctly represents the relation $R = \{1, 2\}$, that is, when `x_coord` takes values 1 or 2, and "`x_coord` $\geq$ 3" represents the relation $R = \{3, 4\}$, in which case the variable `x_coord` takes the values 3 or 4. The sensing function for `texture` is similarly formalized as

$$\begin{aligned} \mathsf{sense}_{\mathtt{t}}(\{\mathtt{y\_coord}\}, \text{"}\mathtt{y\_coord} \leq 2\text{"}) &= \mathtt{smooth} \\ \mathsf{sense}_{\mathtt{t}}(\{\mathtt{y\_coord}\}, \text{"}\mathtt{y\_coord} \geq 3\text{"}) &= \mathtt{rough} \end{aligned} \tag{3.2}$$

The domain of a sensing function $Dom(\mathsf{sense}_y)$ is the set of constraints in $\mathsf{Constr}(Z)$ where sense_y is defined. Each constraint $\varphi \in Dom(\mathsf{sense}_y)$ defines a subset $S_\varphi = \{s \in S \mid s \models \varphi\}$ of the state space S of the actor where an observed value of variable y can be emitted[4]. For instance, the constraint φ_{white} that makes the emission `color` = `white` possible denotes the set of states $S_{\varphi_{\mathsf{white}}} = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle\}$. Hereinafter, we will denote the constraint that generates the emission $y = w$ by φ^y_w (or simply φ_w when it is clear from the context), i.e., $\varphi^y_w \in Dom(\mathsf{sense}_y)$ such that $\mathsf{sense}_y(\varphi^y_w) = w$.

Definition 17 (Exhaustiveness)**.** *We say that a sensing function* sense_y *is* exhaustive *if it holds that*

$$S = \bigcup_{\varphi \in Dom(\mathsf{sense}_y)} S_\varphi \tag{3.3}$$

Alternatively, sense_y *is* exhaustive *if the constraints in* $Dom(\mathsf{sense}_y)$ *consider all possible tuples of values for variables Z, that is,*

$$D(X)^k = \bigcup_{(Z,R) \in Dom(\mathsf{sense}_y)} R \tag{3.4}$$

where $|Z| = k$ *and* $D(X)^k$ *is the Cartesian product of the domains of Z, that is, the set of all possible tuples of values for Z.*

Whatever characterization we follow, the exhaustiveness property guarantees that the sensing function always emits a value for the observable variable in all the states of the

[4]For ease of notation we assume that given a value $w \in D(y)$, there is a single constraint $\varphi \in Dom(\mathsf{sense})$ such that $\mathsf{sense}(\varphi) = w$.

state space S. We can easily check that, in fact, the sensing functions $\mathsf{sense}_{\mathtt{c}}$ and $\mathsf{sense}_{\mathtt{t}}$ defined in equations 3.1 and 3.2 are exhaustive. For instance, let φ_{white} and φ_{black} be the constraints that emit respectively `color = white` and `color = black` as defined in equation 3.1. Following the characterization of exhaustiveness of equation 3.3 we have

$$\begin{aligned} S_{\varphi_{\mathsf{white}}} &= \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle\} \\ S_{\varphi_{\mathsf{black}}} &= \{\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle, \langle 4,4\rangle\} \end{aligned} \tag{3.5}$$

and the union $S_{\varphi_{\mathsf{white}}} \cup S_{\varphi_{\mathsf{black}}}$ does indeed result in the complete state space S. For the characterization of equation 3.4, let $R_{\mathsf{white}} = \{1,2\}$ be the relation of φ_{white} and $R_{\mathsf{black}} = \{3,4\}$ the relation of φ_{black}. Since $Z = \{\mathtt{x\_coord}\}$ we have that

$$\{1,2,3,4\}^1 = R_{\mathsf{white}} \cup R_{\mathsf{black}}$$

which indeed holds.

Finally, we are ready to define the sensor model, which relates the state variables of the actor X with the observable variables of the observer Y by including one sensing function for each observable variable.

Definition 18 (Sensor Model)**.** *A sensor model is a tuple $M_s = \langle X, Y, \mathsf{Sense}\rangle$ where:*

- $X = \{x_1, \ldots, x_k\}$ *are the state variables of the acting agent,*
- $Y = \{y_1, \ldots, y_l\}$ *are the observable variables of the observer agent,*
- $\mathsf{Sense} = \{\mathsf{sense}_1, \ldots, \mathsf{sense}_l\}$ *is a set of exhaustive sensing functions for observable variables Y.*

Considering the idea that an emitted value of an observable variable defines a constraint that limits the number of states where the emission is feasible, we state that an observation defines a set of constraints where each constraint is associated to a different observable variable. More specifically, an observation $o = \langle y_1 = w_1, \ldots, y_l = w_l\rangle$ defines the set of constraints

$$\Phi_o = \{\varphi_{w_i}^{y_i} \mid 1 \leq i \leq l\} \tag{3.6}$$

where $|Y| = l$ and $\varphi_{w_i}^{y_i} \in Dom(\mathsf{sense}_{y_i})$ is the constraint such that $\mathsf{sense}_{y_i}(\varphi_{w_i}^{y_i}) = w_i$. A state can generate an observation o only if it satisfies all the constraints in Φ_o.

Definition 19. *We say that a state s accepts an observation o if $s \models \varphi$ for all $\varphi \in \Phi_o$.*

The set of states that satisfies all constraints in Φ_o and, therefore, accepts the observation o is given by the intersection of the sets that satisfy each constraint

$$S_o = \bigcap_{\varphi \in \Phi_o} S_\varphi$$

Let us go back to our example to illustrate this, using the sensor model $M_s = \langle X, Y, \mathsf{Sense} \rangle$ given by:

- $X = \{\texttt{x\_coord}, \texttt{y\_coord}\}$
- $Y = \{\texttt{color}, \texttt{texture}\}$
- $\mathsf{Sense} = \{\mathsf{sense}_\mathtt{c}, \mathsf{sense}_\mathtt{t}\}$

Consider next the observation $o = \langle \texttt{color} = \texttt{black}, \texttt{texture} = \texttt{rough} \rangle$ that defines the set of constraints $\Phi_o = \{\varphi_{\texttt{black}}, \varphi_{\texttt{rough}}\}$. Recall that $\varphi_{\texttt{black}} = \langle \{\texttt{x\_coord}\}, \texttt{x\_coord} \geq 3 \rangle$ constrains the emission of the value `black` to the right half-side of the grid, and $\varphi_{\texttt{rough}} = \langle \{\texttt{y\_coord}\}, \texttt{y\_coord} \geq 3 \rangle$ constrains the emission of the value `rough` to the bottom half-side of the grid, so the set of states defined by these constraints is:

$$S_{\varphi_{\mathsf{black}}} = \{\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle, \langle 4,4\rangle\}$$
$$S_{\varphi_{\mathsf{rough}}} = \{\langle 1,3\rangle, \langle 2,3\rangle, \langle 3,3\rangle, \langle 4,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle, \langle 3,4\rangle, \langle 4,4\rangle\}$$

Then, the set of states that accept the observation $o = \langle \texttt{color} = \texttt{black}, \texttt{texture} = \texttt{rough} \rangle$ is

$$S_o = S_{\varphi_{\mathsf{black}}} \cap S_{\varphi_{\mathsf{rough}}} = \{\langle 3,3\rangle, \langle 4,3\rangle, \langle 3,4\rangle, \langle 4,4\rangle\}$$

which represents the bottom-right quadrant of the grid.

3.1.2. Deterministic and Non-Deterministic Sensor Models

A deterministic sensor model is one that, given a state, always emits the same observation. In contrast, a non-deterministic sensor model may emit different observations from the same state. Our definition of sensor model (and by extension our framework) is general and allows the representation of non-deterministic sensor models. Nevertheless, it is convenient to be able to distinguish a deterministic sensor model from a non-deterministic one. Next, we define another property of interest to sensing functions, *exclusiveness*, that is useful to characterize deterministic sensor models.

Definition 20 (Exclusiveness)**.** *We say that a sensing function* sense_y *is* exclusive *if it holds that*

$$S_\varphi \cap S_{\varphi'} = \emptyset, \text{ for all } \varphi, \varphi' \in Dom(\mathsf{sense}_y) \tag{3.7}$$

Alternatively, sense *is* exclusive *if all pairs of constraints in* $Dom(\mathsf{sense}_y)$ *define disjoint relations*

$$R \cap R' = \emptyset, \text{ for all } \langle Z, R\rangle, \langle Z, R'\rangle \in Dom(\mathsf{sense}_y) \tag{3.8}$$

The exclusiveness property ensures that, for a given state, the sensing function sense_y of the observable variable y will always generate the same emission since the state will only satisfy one of the conditions in $Dom(\mathsf{sense}_y)$. Going back to our example, we can check that the sensing function sense_c of Equation 3.1 is also exclusive, since the sets $S_{\varphi_{\mathsf{white}}}$ and $S_{\varphi_{\mathsf{black}}}$ (Eq. 3.5) are disjoint and so are the relations $R_{\mathsf{white}} = \{1, 2\}$ and $R_{\mathsf{black}} = \{3, 4\}$.

A *deterministic sensor model* is one in which all its sensing functions are exclusive, and, consequently, it always generates the same observation from a given state. Deterministic sensor models are generally coupled with observable variables that represent abstractions or features of the actor state. Note that, even with a deterministic sensor, there is still uncertainty about the actor state if observations only convey partial information. Non-deterministic sensor model are those that contain one or more non-exclusive sensing function and may, therefore, generate different observations from the same state. Non-deterministic sensors are generally associated to *noise* as they can be used to represent erroneous or spurious readings of a sensor.

3.1.3. Extending a Sensor Model with Costs

Given an observable variable, modeling which value is more likely to be emitted by a sensor in a non-deterministic setting is particularly interesting. This can be quite helpful, for instance, in noisy sensors to indicate that emissions with small deviations from the correct value happen more often. In order to support this feature we introduce the concept of *sensing cost function*.

Definition 21 (Sensing Cost Function)**.** *Given a sensing function* $\mathsf{sense}_y : \mathsf{Constr}(Z) \to D(y)$*, a sensing cost function* $sc_y : \mathsf{Constr}(X) \times D(y) \to \mathbb{R}^+$ *assigns a non-negative cost to pairs of constraints and emitted values for observable variable y.*

Given a sensing function sense_y and its associated sensing cost function sc_y, let φ_w be the constraint such that $\mathsf{sense}_y(\varphi_w) = w$ and let $Dom_w(sc_y) = \{\langle \varphi, w'\rangle \mid w = w'\}$ be the subset of the domain of sc that assigns costs to the emitted value w. For the sensing cost function to be properly defined, we require that

$$S_{\varphi_w} = \bigcup_{\langle \varphi, w \rangle \in Dom_w(sc)} S_\varphi, \text{ for all } w \in D(y) \tag{3.9}$$

$$S_\varphi \cap S_{\varphi'} = \emptyset, \text{ for all } \varphi, \varphi' \in Dom_w(sc_y) \text{ and for all } w \in D(Y) \tag{3.10}$$

Equations 3.9 and 3.10 can be seen as the exhaustiveness and exclusiveness properties subject to the subsets $S_{\varphi_w} \subseteq S$. Equation 3.9 ensures that a sensing cost is assigned to every state within S_{φ_w}, while Equation 3.10 ensures that the costs are deterministically assigned. As long as these properties are respected, the sensing cost function can decompose a constraint $\varphi_w \in Dom(\mathsf{sense}_y)$ into a set of constraints $Dom_w(sc_y)$, which allows the sensing cost function to define regions of the state space that assign different costs to the same emission.

In order to illustrate this, assume that the sensing function sense_c now describes a noisy sensor that can non-deterministically emit `white` or `black` from any tile in the grid, but we have a sensing cost function sc_c that assigns a lower sensing cost to `white` from the left side than from the right side, and vice-versa for `black`. Equation 3.9 says that we can decompose the constraint "the whole board" into "left side" and "right side" as long as they define the same set of states. Equation 3.10 requires that the constraints "left side" and "right side" are disjoint so only one cost is assigned. We will show a more in-depth example of a non-deterministic sensor model with costs in the next chapter.

A sensor model with costs $M_s = \langle X, Y, \mathsf{Sense}, SC \rangle$ extends a sensor model with a set of sensing cost functions $SC = \{sc_1, \ldots, sc_l\}$, each associated to a sensing function. A sensor model with costs allows us to assign a cost to an observation by accumulating the costs of each emission.

Definition 22 (Observation Cost)**.** *Given a sensor model with costs $M_s = \langle X, Y, \mathsf{Sense}, SC \rangle$ and a state s that accepts the observation $o = \langle y_1 = w_1, \ldots, y_l = w_l \rangle$, the cost observing o from s is*

$$ocost(s, o) = \sum_{i=1}^{l} sc_{y_i}(\varphi, w_i) \tag{3.11}$$

where $|Y| = l$ and $\varphi \in Dom_{w_i}(sc_{y_i})$ is the constraint that holds in s.

The observation cost is an score of how likely it is for a state to emit an observation when observed through a sensor model. Given two observations o and o' accepted by a state s, we consider o a more likely observation than o' if $ocost(s, o) < ocost(s, o')$. In the absence of sensing cost functions this distinction cannot be made so it is equivalent to considering the observation costs of all observations accepted by a state as equal. This is known as *fair non-determinsm* and it means that all observations accepted by a state are considered equally likely.

3.2. A Theory for Acting and Sensing

Our framework for inference and learning integrates the action model of the actor and the sensor model of the observer. The result of the integration is a state-space model where the action model describes how the *hidden* state of the actor evolves in time, and the sensor model describes how observations are generated by the true hidden state at each time step. The state-space model defined by the action model of the actor can no longer be considered fully observable from the perspective of the observer. Instead, the state of the actor is considered hidden and only partially observable via the observations generated by the sensor model.

The inherently sequential data of this state-space model are collected by the observer in the form of an observation sequence $\omega = (o_1, \ldots, o_t)$. Each observation o_i of ω is defined as expressed in Definition 15, where index i is an integer that denotes the order of occurrence of o_i in a discrete-time system.

An observation sequence $\omega = (o_1, \ldots, o_t)$ is emitted by the true trajectory $\tau = (s_0, \ldots, s_n)$ which remains hidden to the observer. The sensing process that generates the observation sequence is assumed to be *intermittent*, which broadly speaking can be understood as the actor and the observer being out of sync. More formally, *intermittency* in the sensing process means that the rate of change at which the actor generates new states and the rate at which the observer samples the states may not be the same. This assumption is generally interpreted as the observer failing to observe some of the actor's states which means that the length of the observation sequence ω is lower than the length of the state trajectory τ that emitted it, i.e., $|\omega| \leq |\tau|$. An important implication of intermittency is that it is not possible to know or even set a bound to the length of the true trajectory that generated an observation sequence. This is in stark contrast with other state-space models where the length of the observation sequence denotes the length of the true state trajectory.

In the remainder of this section we first explain the fundamental assumptions that shape the state-space model of our framework. Afterwards, we discuss the scope of the acting-sensing integration, that is, the kind of questions that the observer can ask about the actor with our proposal.

3.2.1. Fundamental Assumptions

Our acting-sensing framework relies upon two fundamental assumptions:

- The *synthesis assumption* says that the possible ways in which the actor may behave starting from a known initial situation are governed by its action model.
- The *sensing assumption* says that the observations that the observer is able to capture from the behaviour of the actor agent are governed by its sensor model.

These two assumptions establish the dependencies between the elements of our framework, namely, an action model $\mathcal{M}_a$, a sensor model $\mathcal{M}_s$, an initial state s_0, a state trajectory τ of the actor, and an observation sequence ω of the observer. We represent the fundamental assumptions of our framework and the resulting dependencies with the *Bayesian network* of Figure 3.2. *Synthesis* refers to the process of generating a state trajectory τ starting from s_0 with the action model $\mathcal{M}_a$. *Sensing* refers to the process of perceiving an observation sequence ω from a state trajectory τ through a sensor model $\mathcal{M}_s$.

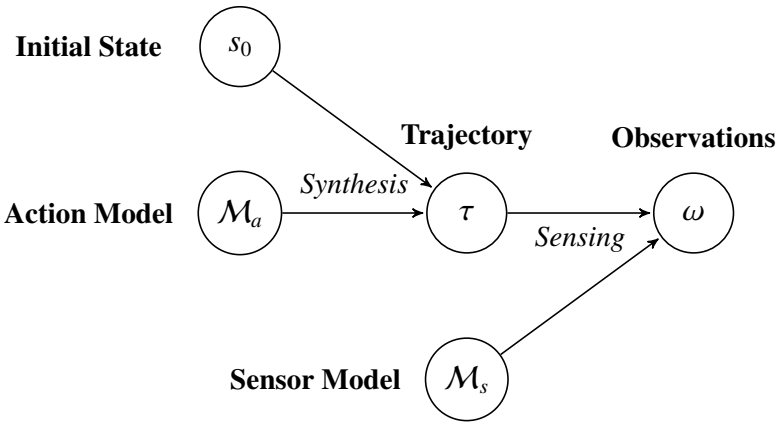

Figure 3.2: *Bayesian network* illustrating the synthesis and sensing assumptions.

Our two fundamental assumptions are analogous to those that shape any other state-space model. The general understanding of a state-space model is that of a discrete-time, stochastic model that contains one set of equations describing the evolution of the latent state, and another set describing how the observer perceives the latent state at each time step. For instance, Hidden Markov Models (HMMs), one of the most common kinds of state-space models use a transition matrix A that encodes the transition probability $P(s_i|s_{i-1})$, an emission matrix B that encodes the emission probability $P(o_i|s_i)$ and a priors vector π that encodes the probability of the initial states $P(s_0)$. The main difference with respect to the state-space model defined by our framework is that the evolution of the latent space is implicitly represented with an action model, and that ours is a cost-driven process instead of a stochastic one. Moreover, it is generally assumed that each visited state generates at least one observation so our assumption of intermittency in the sensing process can also be considered an important difference with respect to other state-space models.

Both the synthesis and sensing assumptions can be interpreted from a satisficing and an optimization perspective. The satisficing interpretation of the synthesis assumption restricts the possible situations in which an actor with action model M_a can find itself to the state space S of the action model. A trajectory τ then describes some specific behaviour of the actor, and the set of possible behaviours starting from an initial situation s_0 is limited to the set of trajectories $\mathcal{T}(s_0, M_a)$. The satisficing interpretation of the sensing assumption states that the observer knows the conditions which led to some specific observation and that the observer can use this information to narrow down the state of the actor at

the time of the observation. More specifically, the observer uses its sensor model M_s to delimit the states of the actor from which an observation o can be emitted to the subset S_o of the state space.

The optimization interpretation of the synthesis and sensing assumptions is supported by the cost function of M_a and (possibly) the sensing cost functions of M_s. This interpretation is related to the assumption of rationality that is often used to rank solutions. Rationality is identified with the maximization of expected utility. Rational agents are those that when presented with different options act in a way that maximizes their expected utility. In the field of Automated Planning rationality is commonly interpreted as choosing the actions that achieve the agent's goals in the most efficient way. Plan cost provides a score that allows us to measure the efficiency with which an agent pursues its goals. Additionally, an optimal plan describes the behaviour of a perfectly rational agent and provides an upper bound on the efficiency with which goals may be pursued.

In this work, however, the goals of the actor agent are not assumed to be known and, even if they are known, they are treated agnostically and plugged into the model of Figure 3.2 as a last observation. Therefore, we cannot measure the efficiency with which they are pursued. Efficiency in our framework takes a different meaning, which concerns both the synthesis and sensing process. We understand efficiency in the synthesis process as the actor being more likely to follow trajectories generated by lower cost plans, while efficiency in the sensing process is understood as the sensors being more likely to emit lower cost observations. Based on this, the optimization interpretation of the synthesis assumption states that a trajectory τ is more likely than another τ' if $cost(\tau) < cost(\tau')$. Similarly, the optimization interpretation of the sensing assumption states that given two states s and s' in S_o, s is more likely to be the source of the observation o if $scost(s, o) < scost(s', o)$.

3.2.2. The Scope of the Acting-Sensing Integration

This section poses a number of question regarding the actor and explains how these questions translate into inference and learning problems involving the elements of our framework. All of the problems discussed here are initiated by the observer watching the actor and willing to use its data to ascertain some aspect of the actor.

The first problem at hand follows from the uncertainty inherent to the sensing process itself. The observer collects some pieces of data on the actor in the form of observations and uses this data to guess what the actor was doing, that is, to find the true trajectory. We refer to this problem as *Observation Decoding* since the term *decoding* is commonly used to refer to the problem of finding the most likely explanation to some evidence. In the context of this dissertation, the evidence is the sequence of observations collected by the observer and the explanation is a trajectory of the actor that accounts for the observations. The Observation Decoding problem takes as input the action model M_a of the actor and its initial situation s_0 as well as a sequence of observations ω and the sensor model M_s

used to capture ω. Then, the input is used to compute the most likely state trajectory that explains the collected sequence of observations ω. In terms of the Bayesian network of Figure 3.2, the solution to the observation decoding problem is given by

$$\tau^* = \underset{\tau \in \mathcal{T}(M_a, s_0)}{\arg\max}\, P(\tau|\omega, s_0, M_a, M_s) \tag{3.12}$$

We consider the observation decoding problem as the building block of our framework because solving more involved inference and learning problems generally requires the ability to find an explanation to a sequence of observations.

Building on top of the observation decoding problem we can define other inference problems concerned with different aspects of the actor. Our next problem, *temporal inference*, allows the observer to answer a wide landscape of questions regarding the past, present and future of the actor. The Temporal Inference problem extends the observation decoding problem with a set of hypotheses $\mathcal{H}$ such that each hypothesis $\eta \in \mathcal{H}$ describes a possible answer to the question at hand. Hypotheses will be formalized in Chapter 5 but, for now, it suffices to understand that hypotheses allows the observer to make conjectures about the past and future of the actor. Hypothesizing about the past is useful to understand parts of actor's behaviour that were not clearly portrayed by the sensing process due to its intermittency or the partial information conveyed by the observations. On the other hand, hypothesizing about the future can be used to predict the behaviour of the actor based on what we already know. Solving the Temporal Inference problem is finding the best answer to the question defined by the set of hypotheses $\mathcal{H}$ so it can be formalized as finding the most likely hypothesis in the set.

$$\eta^* = \underset{\eta \in \mathcal{H}}{\arg\max}\, P(\eta|\omega, s_0, M_a, M_s) \tag{3.13}$$

The last inference problem addressed in this thesis is the *Model Recognition* problem. In contrast with the previous inference problems, model recognition works on the premise that the observer is uncertain about the model of the actor. In Model Recognition, the observer has at its disposal a finite and explicit set of candidate models for the actor and wants to figure out which one fits better with the observed behaviour. Then, the Model Recognition problem extends the observation decoding problem by considering a set of candidate action models $\mathcal{M}_a$ instead of a single one. The Model Recognition problem can be regarded as a classification problem whose solution is formulated as

$$M_a^* = \underset{M_a \in \mathbb{M}_a}{\arg\max}\, P(\omega|s_0, M_a, M_s)P(M_a) \tag{3.14}$$

The *Learning* problem, similar to the Model Recognition problem, is also concerned with the model of the actor. The difference lies in that the observer does not have now an explicit set of candidate models to choose from and instead needs to use the collected

data to learn the action model of the actor. Learning a state-space model is widely understood as estimating the free parameters that define the transitions of the model from a set of learning examples. In the context of planning, the learning problem is traditionally referred to as *Action Model Learning* or *Action Model Acquisition*, and its most common formulation assumes that the free parameters to learn are the App and Eff constraints that define the transition function of the underlying state model. Here, learning examples are the result of multiple episodes of the observer watching the actor and are represented as pairs $\langle s_0, \omega \rangle$. An Action Model Learning problem takes as input an *unspecified* action model $M_a[] = \langle X, A, c \rangle$ that leaves the App and Eff constraints undefined, a set of learning examples $\mathcal{L}$, and the sensor model M_s used to collect the learning examples. Then, the goal of Action Model Learning is to compute

$$\theta^* = \arg\max_{\theta} P(\mathcal{L}|M_a[\theta], M_s) \tag{3.15}$$

where $\theta = \langle \mathsf{App}, \mathsf{Eff} \rangle$. In other words, learning is about estimating the App and Eff constraints within the defined syntax that maximizes the likelihood of the learning examples.

As a final summary, this section has introduced the inference and learning problems that are addressed in this thesis. Each problem is accompanied with a probabilistic interpretation of its solution based on the dependencies established by our fundamental assumptions and illustrated in Figure 3.2. In following chapters, the solutions presented here will be used as starting points to develop approximations that allows us to solve these problems. The remaining chapters of this dissertation are structured as follows:

- The Observation Decoding problem is explained in detail in Chapter 4.
- The Temporal Inference problem is explained in detail in Chapter 5.
- The Model Recognition problem is explained in detail in Chapter 6.
- The Action Model Learning problem is explained in detail in Chapter 7.

Part IV

Inference

4. OBSERVATION DECODING

This chapter is dedicated to the Observation Decoding problem, the problem of finding the best explanation for the sequence of observations collected by the observer. We start the chapter presenting a working example in section 4.1 that will be used throughout the chapter to aid explanations. Section 4.2 formalizes the Observation Decoding problem, its solution and other related concepts. In section 4.3, we justify and motivate the solution of the Observation Decoding problem presented in the previous section. Next, in section 4.4 we present our proposal to address the Observation Decoding problem using planning. We finally conclude this chapter with an evaluation of the approach (section 4.5) and a discussion of topics related to the Observation Decoding problem (section 4.6).

4.1. Working Example: Blindspots

This section presents a working example named *Blindspots* that is based on a simple grid navigation domain. The main draw of this example is that it features a non-deterministic sensor model with sensing cost functions which will allow us to highlight the importance of exploiting sensing costs when this knowledge is available. The *Blindspots* domain receives its name because it simulates a scenario where the observer uses a zenith camera to locate the actor in a grid with some *covered* tiles that block the vision of the camera. Figure 4.1 shows the grid that we will be using for this working example. In this figure we have colored *open* tiles in white and *covered* tiles in grey. The figure also shows a trajectory of the actor agent represented by a solid arrow. Assuming that a tile (x, y) is described by their x and y coordinates, the trajectory shows the actor moving from tile $(3, 1)$ to tile $(3, 5)$ passing through four tiles of the covered area.

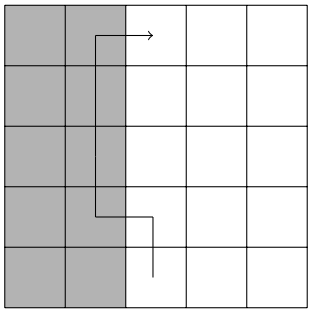

Figure 4.1: 5×5 grid used in the *Blindspots* working example. Open tiles are colored in white and covered tiles in grey. The arrow depicts a trajectory that shows the actor agent moving from tile $(3, 1)$ to tiles $(3, 5)$.

The actor can only move one tile at a time, in any of the four directions N,S,W or E, and all moves are equally likely (moving against the wall has no effect). The camera used

by the observer can pinpoint the location of the actor with 90% reliability when it is on an *open* tile, but has a 10% chance to fail meaning that the observed location is *unknown*. Additionally, when the actor is moving through the covered tiles the camera's line of sight is blocked so it is completely unable to locate the agent.

Regarding the definition of the action model of the actor, this working example considers a single state variable `loc` with domain $D(\texttt{loc}) = \{1, \ldots, 5\}^2$ representing the location of the actor in the 5×5 grid. For instance, `loc` = (1,1) would situate the actor in the bottom left tile, while `loc` = (3,3) would mean that he is in the centermost tile. The set of actions of the actor consist of actions $\texttt{move}((x1, y1), (x2, y2))$ that represent the actor moving from a tile $(x1, y1)$ to an adjacent tile $(x2, y2)$. Since moving in any direction is equally likely, we define all actions cost as $-\log 0.25$ which is the log probability of moving in one specific direction.

The sensor model also presents a single observable variable `obs_loc` that represents the observed location of the actor. Variable `obs_loc` has the same domain as state variable `loc` but extended with a special token `unknown` that represents the failure of the camera to locate the actor. Formally, $D(\texttt{obs\_loc}) = D(\texttt{loc}) \cup \{\texttt{unknown}\}$. The sensing function for `obs_loc` is defined as follows:

$$\texttt{sense}_{\texttt{obs\_loc}}(\texttt{loc}, \text{"open tile"}) = \texttt{loc}$$
$$\texttt{sense}_{\texttt{obs\_loc}}(\texttt{loc}, \text{"anywhere"}) = \texttt{unknown}$$

The first rule of the sensing function indicates that the camera can emit the correct location of the actor when it is in a clear tile. The relation represented by "open tile" encompasses all pairs in $\{3, \ldots, 5\} \times \{1, \ldots, 5\}$, that is, all tiles in the three rightmost columns of the grid which are the open tiles. The second rule says that the camera can sometimes fail to locate the actor no matter its position in the grid. This means that `loc` = `unknown` can be emitted from any tile, so the relation of this constraint is effectively the domain of state variables `loc`.

Following, we define the sensing cost function $sc_{\texttt{obs\_loc}}$ for variable `obs_loc` as

$$\texttt{sc}_{\texttt{obs\_loc}}(\langle \texttt{loc}, \text{"open tile"} \rangle, \texttt{loc}) = -\log 0.9$$
$$\texttt{sc}_{\texttt{obs\_loc}}(\langle \texttt{loc}, \text{"open tile"} \rangle, \texttt{unknown}) = -\log 0.1$$
$$\texttt{sc}_{\texttt{obs\_loc}}(\langle \texttt{loc}, \text{"covered tile"} \rangle, \texttt{unknown}) = -\log 1$$

The first and second rules are associated to the sensing process when the actor is in the open area and define the costs for emitting its correct position or `unknown` position, respectively. The last rule is the cost associated to the emission of `unknown` when the agent moves through the covered area. The cost associated to each emission are defined as the log probability of the emission.

The Blindspots working example also lends itself to a Hidden Markov Model representation. Figure 4.2 shows a fragment of the HMM representation that accounts for the top left corner of the grid. In this HMM representation tiles are represented as states (colored blue) and actions as transitions between states. Since the actor moves in any of the four directions with equal probability, each transition has probability 0.25. The orange rectangles show the emission probabilities associated to each state. For instance, the state representing tile $(3, 5)$ emits the correct position with probability 0.9 because $(3, 5)$ is an open tile, and emits unknown (representing the failure to locate the actor) with probability 0.1.

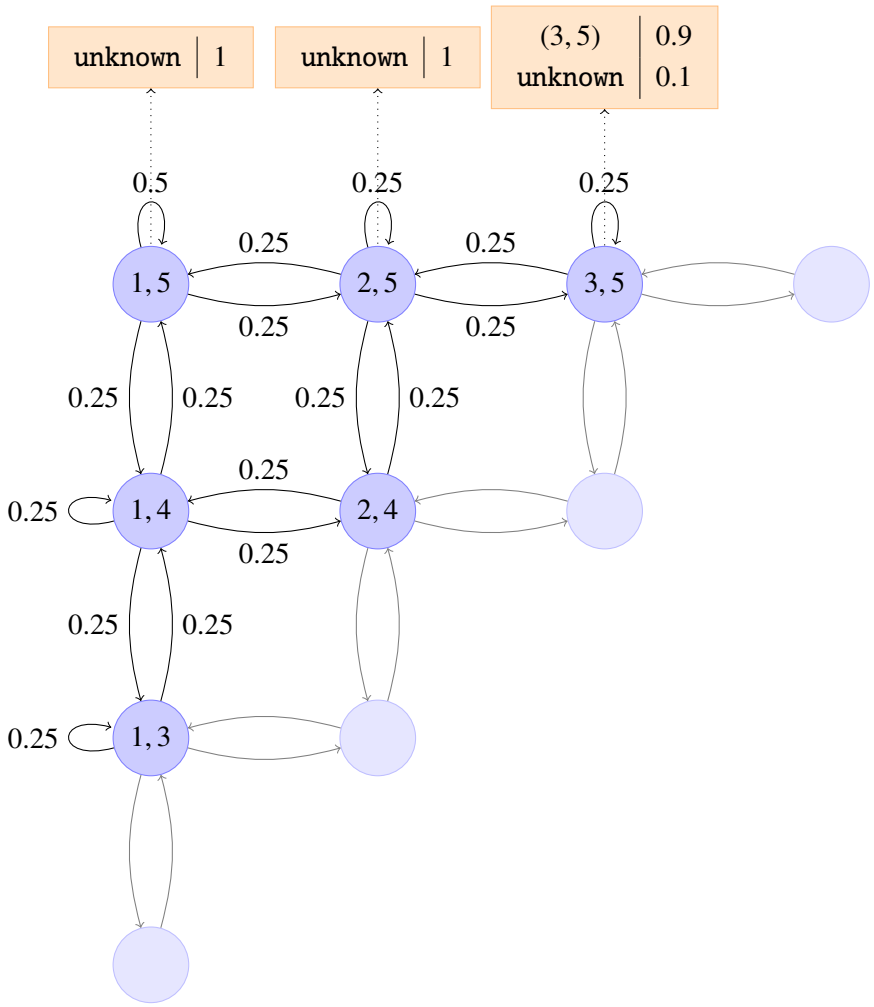

Figure 4.2: Fragment of the HMM representation of the *Blindspots* working example.

The action and sensor models defined in this working example and the HMM of Figure 4.2 are equivalent considering that transition probabilities have been translated to action costs and emission probabilities to sensing costs. The translation applies a logarithmic transformation to the action and sensing costs that transforms the probability maximization natural to HMMs into an equivalent cost minimization more suited for classical planning approaches (Jiménez et al., 2006; Little & Thiebaux, 2007). Another key difference with respect to the HMM representation is that the state-space model is not

fully enumerated but implicitly defined with an action model. The same happens with the relation between states and observations. In the next section we will see that, while both our framework and HMMs can model the same domain, our approach is more flexible to observation sequences with gaps caused by the inttermitency of the sensing process.

4.2. The Observation Decoding Problem

In this section, we formalize the Observation Decoding problem along with some other related concepts. An Observation Decoding problem takes as inputs the model and initial situation of the actor, as well as a sequence of observations and the sensor model used to capture them.

Definition 23 (Observation Decoding problem)**.** *An Observation Decoding problem is given by the tuple* $\langle M_a, M_s, s_0, \omega \rangle$ *where:*

- $M_a = \langle X, A, \mathsf{App}, \mathsf{Eff}, c \rangle$ *is the action model of the actor,*
- $M_s = \langle X, Y, \mathsf{Sense}, SC \rangle$ *is the sensor model of the observer,*
- s_0 *is the known initial situation,*
- $\omega = (o_1, \ldots, o_t)$ *is an observation sequence.*

The solution to an Observation Decoding problem is the most likely explanation for the collected observation sequence ω. Broadly speaking, we understand an explanation as a state trajectory that can be the source of the observation sequence. This means that a state trajectory $\tau = (s_0, \ldots, s_n)$ explains an observation sequence $\omega = (o_1, \ldots, o_t)$ if all the observations in ω are accepted by states in τ while also preserving the order of the sequence. Since we are working under the assumption that the sensing process is intermittent, the length of ω is always less or equal than the length of τ, i.e., $|\omega| \leq |\tau|$. Therefore, we must first decide on an alignment that determines which state of τ emitted each observation of ω. An alignment is a strictly monotonic function $\alpha : \mathbb{N} \to \mathbb{N}$ that maps observation indices $\{1, \ldots, t\}$ into trajectory indices $\{0, \ldots, n\}$.

Definition 24 (Explanation)**.** *Given a decoding problem* $\langle M_a, M_s, s_0, \omega \rangle$ *with* $\omega = (o_1, \ldots, o_t)$*, we say that a state trajectory* $\tau = (s_0, \ldots, s_n)$ *explains the observation sequence* ω *under alignment* α *if* $\tau \in \mathcal{T}(M_a, s_0)$ *and for all* $i \in \{1, \ldots, t\}$ *it holds that* $s_{\alpha(i)}$ *accepts* o_i*. We denote by* $\mathcal{T}_\omega(M_a, s_0)$ *the set of explanations.*

The definition of explanation is related to the satisficing interpretation of the synthesis and sensing assumptions. Under the synthesis assumption an explanation must describe some possible behaviour of the actor so they are limited to trajectories in $\mathcal{T}(M_a, s_0)$. The sensing assumption further reduces the possible explanations to the subset $\mathcal{T}_\omega(M_a, s_0) \subseteq \mathcal{T}(M_a, s_0)$ of trajectories that accept all the observations.

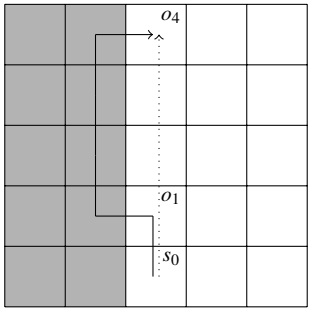

Figure 4.3: Example of an Observation Decoding problem in the *Blindspots* working example. The example considers a four-observation sequence $\omega = (o_1, o_2, o_3, o_4)$ with o_2 and o_3 being `unknown` emissions, and two explanations (solid and dotted arrows) that might generate the given observation sequence.

Figure 4.3 shows an example of a decoding problem in our *Blindspots* working example. In this decoding problem the observer has collected four observations: two of them, denoted as o_1 and o_4 in the figure, successfully locate the actor, but in between those two observations the camera produces two `obs_loc = unknown` emissions. The observation sequence is then

Time	**Observation**
1	`obs_loc` = $(3, 2)$
2	`obs_loc = unknown`
3	`obs_loc = unknown`
4	`obs_loc` = $(3, 5)$

where "Time" is not a time-stamp but an indication of how observations are sequenced. The doted and solid arrows represent two possible explanations for the collected observations assuming an initial situation $s_0 = \langle \texttt{loc} = (3, 1) \rangle$. We refer to the explanations denoted by the dotted and solid arrows as $\tau 1$ and $\tau 2$, respectively. The two explanations shown in the figure and their alignments are specified below

$\tau 1$	
Time	**State**
0	`loc` = $(3, 1)$
1	`loc` = $(3, 2)$
2	`loc` = $(3, 3)$
3	`loc` = $(3, 4)$
4	`loc` = $(3, 5)$

$\tau 2$	
Time	**State**
0	`loc` = $(3, 1)$
1	`loc` = $(3, 2)$
2	`loc` = $(2, 2)$
3	`loc` = $(2, 3)$
4	`loc` = $(2, 4)$
5	`loc` = $(2, 5)$
6	`loc` = $(3, 5)$

$$\alpha1(1) = 1 \qquad \alpha2(1) = 1$$
$$\alpha1(2) = 2 \qquad \alpha2(2) = 3$$
$$\alpha1(3) = 3 \qquad \alpha2(3) = 4$$
$$\alpha1(4) = 4 \qquad \alpha2(4) = 6$$

According to the sensor model for the *Blindspots* working example, when the emitted value for `obs_loc` is a location the actor must be in the corresponding open tile. The `unknown` emission, on the other hand, can be emitted from any tile. Both $\tau1$ and $\tau2$ are valid explanations for ω as in both cases the observation $\langle \texttt{obs\_loc} = (3,2) \rangle$ is aligned with state $\langle \texttt{loc} = (3,2) \rangle$, and $\langle \texttt{obs\_loc} = (3,5) \rangle$ is aligned with $\langle \texttt{loc} = (3,5) \rangle$, while $\langle \texttt{obs\_loc} = \texttt{unknown} \rangle$ can be aligned with any state. We also note that observations o_2 and o_3 could have been aligned with any other of the unmatched states in $\tau2$ without affecting the validity of the explanations.

This example also serves to highlight an important difference between our approach that assumes intermittency in the sensing process and other state-space models such as HMMs that do not. In the previous section we have already shown that this working example can be represented as an HMM and, therefore, we could find the most likely explanation using the appropriate HMM algorithm which is known as the Viterbi algorithm. However, HMM algorithms do not assume intermittency so our observation sequence of length 4 can only be explained by a state trajectory of length 4. This means that under the HMM assumptions the set of explanations $\mathcal{T}_\omega(M_a, s_0)$ would in fact be the singleton consisting only of $\tau1$. In our framework, however, intermittency causes the length of an explanation to be unbounded, which also means that the set of explanations $\mathcal{T}_\omega(M_a, s_0)$ is actually infinite in domains with reversible actions, like our working example.

Now that we have defined the set of explanations $\mathcal{T}_\omega(M_a, s_0)$, our next step is to rank explanations in $\mathcal{T}_\omega(M_a, s_0)$ in order to define the most likely explanation. We leverage the optimization interpretation of our fundamental assumptions in order to define two costs associated to an explanation and that will be used to define the most likely explanation. The first cost is the *synthesis cost*, named after the synthesis assumption. The synthesis cost of an explanation τ is the cost incurred by the actor when following the behaviour described by τ. We define this cost as the cost of the plan that generates τ.

Definition 25 (Synthesis cost)**.** *The synthesis cost of a trajectory $\tau \in \mathcal{T}(M_a, s_0)$ is*

$$synCost(\tau) = cost(\pi) \tag{4.1}$$

where π is the plan in $\Pi(M_a, s_0)$ such that $\tau = s_0[\![\pi]\!]$.

Similarly we define the *sensing cost* that follows from the sensing assumption. The sensing cost measures the cost incurred by the sensors during the emission of the observation sequence from the explanation. We define the sensing cost of an explanation and an observation sequence as the accumulated observation cost of each pair of aligned states and observations

Definition 26 (Sensing cost)**.** *The sensing cost of a trajectory τ that explains an observation sequence ω under alignment α is*

$$senCost_{\alpha}(\tau, \omega) = \sum_{i=1}^{t} ocost(s_{\alpha(i)}, o_i) \tag{4.2}$$

Recall that the observation cost is defined as a constant when the sensor model does not include sensing cost functions so, by extension, the sensing cost is also a constant in this case. Following, we summarize all the costs associated to the sensing process:

- The sensing cost function sc_y defines the cost of an emission $y = w$.
- The observation cost $ocost(s, o)$ accumulates the cost of the emission of each observable variable in o.
- The sensing cost $senCost_{\alpha}(\tau, \omega)$ accumulates the observation cost of aligned state-observations pairs in an explanation.

Finally, we define the most likely explanation as the explanation in $\mathcal{T}_{\omega}(M_a, s_0)$ that minimizes the sum of the synthesis and sensing costs. An in-depth justification of this solution is given in the next section.

Definition 27 (**Most likely explanation**)**.** *The solution to an Observation Decoding problem $\langle M_a, M_s, s_0, \omega \rangle$ is the most likely explanation given by*

$$\tau^* = \underset{\tau \in \mathcal{T}_{\omega}(M_a, s_0)}{\arg\min} \; synCost(\tau) + senCost(\tau, \omega) \tag{4.3}$$

We note that, when the sensor model $M_s = \langle X, Y, \mathsf{Sense} \rangle$ does not include sensing cost functions the sensing cost is a constant for all explanations so the term can be removed from the minimization

$$\tau^* = \underset{\tau \in \mathcal{T}_{\omega}(M_a, s_0)}{\arg\min} \; synCost(\tau) \tag{4.4}$$

Let us go back to our working example and talk now about the costs associated to the explanations $\tau 1$ and $\tau 2$. Starting with the synthesis cost, we find that $synCost(\tau 1) < synCost(\tau 2)$ since in the *Blindspots* working example all actions have the same cost and the plan for $\tau 1$ has fewer actions than the plan for $\tau 2$.

$$synCost(\tau 1) = 4(-\log 0.25) = 2.41$$
$$synCost(\tau 2) = 6(-\log 0.25) = 3.61$$

The sensing costs paint a very different picture where $cost_{\alpha 1}(\tau 1, O) > cost_{\alpha 2}(\tau 2, O)$. The computations are as shown below

$$senCost_{\alpha 1}(\tau 1, \omega) = 2(-\log 0.9) + 2(-\log 0.1) = 2.09$$
$$senCost_{\alpha 2}(\tau 2, \omega) = 2(-\log 0.9) + 2(-\log 1) = 0.09$$

where $-\log 0.9$ corresponds to the emission of a location from an open tile, $-\log 0.1$ to an empty reading from an open tile, and $-\log 1$ to an empty emission from a covered tile. These costs indicate that the observation sequence ω is more likely to be emitted from the trajectory $\tau 2$ than from $\tau 1$.

From these results we can see that, if the sensing cost functions were unavailable, the most likely explanation would be $\tau 1$ as it is the one that most efficiently reproduces the observed behaviour of moving from tile $(3, 1)$ to tile $(3, 5)$. On the other hand, if we consider the sensing costs, the overall cost of $\tau 1$ is higher than the cost of $\tau 2$ due to a much higher sensing cost. This higher sensing cost is modelling the fact that it is very unlikely for the actor to go unnoticed to the camera twice while on an open tile.

4.3. The Solution of the Observation Decoding Problem

In this section we explain how we arrive at the definition of most likely explanation given in Definition 27. Our starting point is Equation (3.12) of Section 3.2.2 that provides a probabilistic interpretation of the Observation Decoding problem under the Bayesian network that defines the state-space model of our framework. We repeat this equation below for convenience.

$$\tau^* = \underset{\tau \in \mathcal{T}(M_a, s_0)}{\arg\max}\ P(\tau | \omega, s_0, M_a, M_s) \tag{4.5}$$

From the definition of conditional probability we arrive at the following equation.

$$\tau^* = \underset{\tau \in \mathcal{T}(M_a, s_0)}{\arg\max}\ P(\tau, \omega | s_0, M_a, M_s) \tag{4.6}$$

The joint likelihood $P(\tau, \omega | M_a, M_s)$ expresses the probability that τ and ω happen together given the models M_a and M_s and the initial situation s_0. From the Bayesian Network of figure 3.2 it follows that the joint likelihood can be computed as:

$$P(\tau, \omega | M_a, M_s) = P(\tau | s_0, M_a) P(\omega | \tau, M_s) \tag{4.7}$$

where $P(\tau|s_0, M_a)$ is associated to the synthesis process and $P(\omega|\tau, M_s)$ to the sensing process. We will therefore refer to these probability distributions as the synthesis and sensing probabilities.

The synthesis probability: The *synthesis probability* $P(\tau|s_0, M_a)$ is the probability of generating τ with action model $\mathcal{M}_a$ starting from the initial situation s_0. Therefore, we consider a trajectory $\tau \notin \mathcal{T}(M_a, s_0)$ has a synthesis $P(\tau|s_0, M_a) = 0$. Following the synthesis assumption, the synthesis probability assigns higher probability to lower cost trajectories:

$$P(\tau|s_0, \mathcal{M}_a) \propto -synCost(\tau) \tag{4.8}$$

The sensing probability: The *sensing probability* is the likelihood of generating an observation sequence $\omega = \langle o_1, o_2 \ldots, o_t \rangle$ from a trajectory τ through the sensor model M_s. The sensing probability assigns a probability $P(\omega|\tau, \mathcal{M}_s) = 0$ to trajectories not in $\mathcal{T}_\omega(M_a, s_0)$ since these trajectories cannot generate the observation sequence ω. Following the sensing assumption, the sensing probability assigns higher probability to lower cost observations:

$$P(\omega|\tau, \mathcal{M}_s) \propto -senCost_\alpha(\tau, \omega) \tag{4.9}$$

Substituting Equation 4.7 in Equation 4.6 we arrive at

$$\tau^* = \underset{\tau \in \mathcal{T}_\omega(M_a, s_0)}{\arg\max}\ P(\tau|s_0, M_a)P(\omega|\tau, M_s) \tag{4.10}$$

This equation defines the most likely explanation as the trajectory that maximizes the product of the synthesis and sensing probabilities. Since our framework is cost-driven, we redefine this solution as a cost minimization of the sum of the synthesis and sensing costs of a trajectory arriving at the solution given in Definition 27.

$$\tau^* = \underset{\tau \in \mathcal{T}_\omega(M_a, s_0)}{\arg\min}\ synCost(\tau) + senCost(\tau, \omega) \tag{4.11}$$

4.4. Observation Decoding as Planning

We come up with a proposal that exploits planning technology as a resolution method for discovering the true state trajectory from a sequence of observations of the actor, that is, to solve the Observation Decoding problem. Our Observation Decoding as planning approach draws on the idea that an observation sequence expresses a property about the

true trajectory. We make this observation based on the already established notion that an observation o defines a constraint Φ_o that states that accept o must satisfy. An observation sequence extends the constraint with the total order relationship that exists between the observations of an observation sequence.

Linear Temporal Logic (LTL) is a formalism for specifying properties about sequences of states where each state has a unique successor, based on a linear-time perspective. LTL formulas have been used by the planning community to represent Temporally Extended Goals (TEGs) that describe properties about the state trajectories generated by solution plans. We leverage these ideas in our proposal by interpreting an observation sequence ω as an LTL property that any explanation in $\mathcal{T}_\omega(M_a, s_0)$ must satisfy. This allows us to compile an Observation Decoding problem into a classical planning problem whose solutions are explanations.

In summary, our Observation Decoding as planning approach compiles an Observation Decoding problem $\langle M_a, M_s, s_0, \omega\rangle$ into a classical planning problem $\mathcal{P}(\omega)$ such that $\Pi(\mathcal{P}(\omega))$ is the set of solution plans that generate the set of explanations $\mathcal{T}_\omega(M_a, s_0)$. Additionally, by properly defining the action costs of $\mathcal{P}(\omega)$, the optimal solution plan will generate the most likely explanation. As is the case in the compilation-based approach to planning with TEGs, our approach also involves two translation steps:

1. translate the LTL property of the observation sequence ω into a monitor automaton $\mathcal{A}(\omega)$ that accepts only state trajectories in $\mathcal{T}_\omega(M_a, s_0)$ and
2. encode $\mathcal{A}(\omega)$ as part of the classical planning problem $\mathcal{P}(\omega)$

The rest of this section is structured as follows: sections 4.4.1 and 4.4.2 are devoted to explain the first translation step of the compilation scheme, an outline of LTL and the constructions of the monitor automaton; section 4.4.3 explains the second translation step, i.e., how to encode the automaton $\mathcal{A}(\omega)$ in a planning problem $\mathcal{P}(\omega)$; section presents two theorems regarding the soundness and completeness of the compilation; section 4.4.5 explains how to define the action costs for $\mathcal{P}(\omega)$ so that the most likely explanation can be computed; and section 4.4.6 guides the reader through the compilation with an example.

4.4.1. LTL Preliminaries

Following, we briefly introduce the syntax and semantics of LTL_f (Giacomo & Vardi, 2013), a variant of LTL interpreted over finite state trajectories which uses the same syntax as that of LTL, and suffices for our purposes. The LTL_f syntax uses negation ($\neg$), boolean connective ($\wedge$), and temporal operators X (next) and U (until). An atomic formula is true, false, or an element of a set of propositional symbols P. Non-atomic formulae are defined inductively:

- $\neg\psi$ if ψ is a formula.

- $(\psi \wedge \chi)$ if ψ and χ are formulae.
- $\mathsf{X}\,\psi$ if ψ is a formula.
- $(\psi\ \mathsf{U}\ \chi)$ if ψ and χ are formulae.

Other temporal operators such as F (eventually), G (always), and R (release) are defined in terms of the basic operators of the language. $\mathsf{F}\,\psi$ is defined as $(\mathsf{true}\ \mathsf{U}\ \psi)$, $\mathsf{G}\,\psi$ as $\neg\mathsf{F}\neg\psi$, and $(\psi\ \mathsf{R}\ \chi)$ as $\neg(\neg\psi\ \mathsf{U}\ \neg\chi)$. We also assume the standard boolean abbreviations $\vee$ and $\rightarrow$ (implies) as well as the abbreviation *Last*, which stands for $\neg\mathsf{X}\mathsf{true}$ and denotes the last instant of the finite trace (Giacomo & Vardi, 2013).

The semantics of a LTL_f formula ψ is given in terms of finite sequences of states $\tau = (s_0, s_1, \ldots, s_n)$ such that $s_i \subseteq P$ for $i \in \{0, \ldots, n\}$. We say that τ satisfies ψ, and write $\tau \models \psi$, iff τ if $\tau_{0:n} \models \psi$, where for every $i \in \{0, \ldots, n\}$:

- $\tau_{i:n} \models p$ where $p \in P$ iff $p \in s_i$.
- $\tau_{i:n} \models \neg\psi$ iff $\tau_{i:n} \not\models \psi$.
- $\tau_{i:n} \models (\psi \wedge \chi)$ iff $\tau_{i:n} \models \psi$ and $\tau_{i:n} \models \chi$.
- $\tau_{i:n} \models \mathsf{X}\,\psi$ iff $i < n$ and $\tau_{i+1:n} \models \psi$.
- $\tau_{i:n} \models (\psi\ \mathsf{U}\ \chi)$ iff there exists a $j \in \{i, \ldots, n\}$ such that $\tau_{j:n} \models \chi$ and for every $k \in \{i, \ldots, j-1\}$ it holds that $\tau_{k:n} \models \psi$

While LTL is defined over propositional variables, it is simple to draw an equivalence to the finite domain variables used in this thesis. It suffices to understand P as the set of propositions of the form $x = v$ where $x \in X$ and $v \in D(x)$, and a state as a subset of P containing one such proposition for each variable in X. Consequently, each tuple in the relation R of a constraint $\langle Z, R\rangle \in \mathsf{Constr}(X)$ defines a conjunction of propositions (or a proposition if $|Z| = 1$), and the relation itself defines a disjunction.

4.4.2. A Monitor Automaton for the Observation Sequence

An observation sequence $\omega = (o_1, \ldots, o_t)$ expresses the property that a state trajectory $\tau = (s_0, \ldots, s_n)$ must first contain a state that accepts observation o_1, then a state that accepts o_2, and so on for all observations. This kind of property is known in temporal logic as an *ordered occurrence*. An ordered occurrence is a property of the form $\mathsf{XF}(\psi_1 \wedge \mathsf{XF}(\psi_2 \wedge \ldots \wedge \mathsf{XF}\psi_n))$ where subformulas $\psi_1 \ldots \psi_n$ must be satisfied in order. In the case of an observation sequence $\omega = (o_1, \ldots, o_t)$, the subformulas that must be satisfied in order are the constraints $\Phi_{o_1} \ldots \Phi_{o_t}$ imposed by the observations. Given an observation sequence $\omega = (o_1, o_2, \ldots, o_t)$, we denote by $\mathsf{ord}(\omega) = \mathsf{XF}(\Phi_{o_1} \wedge \mathsf{XF}(\Phi_{o_2} \wedge \ldots \wedge \mathsf{XF}\Phi_{o_t}))$ the LTL_f property described by ω.

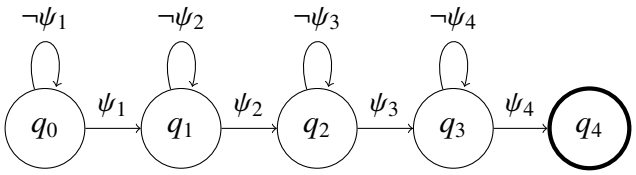

Figure 4.4: Monitor automaton for an observation sequence $\omega = (o_1, o_2, o_3, o_4)$ where $\psi_i = \Phi_{o_i}$.

Our next step is to compute a monitor automaton $\mathcal{A}(\omega)$ such that all accepted trajectories satisfy the property $\mathsf{ord}(\omega)$ and, therefore, are explanations in $\mathcal{T}_\omega(M_a, s_0)$. The ordered occurrence property described by an observation sequence translates to a Deterministic Finite Automaton (DFA) with a sequential structure and self-loops as shown in Figure 4.4. This automaton includes a location and two transitions for each observation in the observation sequence. One transition is used to move to the next location when the current state accepts the observation, and the other is used to remain in the same location when the state does not accept it. Formally, given an observation sequence $\omega = (o_1, \ldots, o_t)$, the monitor automaton $\mathcal{A}(\omega) = \langle Q, \Sigma, \delta, q_0, F \rangle$ that accepts state trajectories that satisfy the property $\mathsf{ord}(\omega)$ is specified as follows:

- $Q = \{q_0, \ldots, q_t\}$ is the set of automaton locations consisting of one location for each observation in ω plus an additional initial location q_0,
- $\Sigma = \mathsf{Constr}(X)$ is a finite set of input symbols defined here as the class of constraints over X,
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function. This automaton defines transitions $q_i = \delta(q_{i-1}, \Phi_{o_i})$ for $i \in \{1 \leq i \leq t\}$ that advance the location of the automaton, and loop transitions of the form $q_{i-1} = \delta(q_{i-1}, \neg\Phi_{o_i})$ to remain in the same location.
- $F \subseteq Q$ is the set of accepting locations. a singleton in this case. In this automaton, $F = \{q_t\}$ is a singleton consisting only of the last location.

The locations of $\mathcal{A}(\omega)$ represent the current progress in the verification of the property $\mathsf{ord}(\omega)$ by a trajectory τ. Being at location q_i means thereby that τ satisfies $\mathsf{ord}(\omega)$ up to Φ_{o_i}, which can be interpreted as having explained the first i observations of ω. When a state s of a trajectory τ is processed in the automaton location q_{i-1}, the transition $q_i = \delta(q_{i-1}, \Phi_{o_i})$ is triggered iff s accepts o_i, progressing the automaton to location q_i; otherwise the automaton remains at q_{i-1} through transition $q_{i-1} = \delta(q_{i-1}, \neg\Phi_{o_i})$. Consequently, reaching location q_t means that τ satisfies $\mathsf{ord}(\omega)$; i.e., that τ is an explanation for ω.

4.4.3. The Compilation

Before deepening into the compilations scheme, let us summarize what we know so far. We have an observation sequence of the actor, ω, and we know that ω implicitly encodes

an LTL property which determines the set of trajectories that comply with ω. This set of trajectories is named $\mathcal{T}_\omega(M_a, s_0)$ and we are interested in finding the trajectory that reports the best (most likely) explanation to ω. This is the problem known as Observation Decoding and formulated as $\langle M_a, M_s, s_0, \omega \rangle$.

In the previous section, we explained how to build the automaton $\mathcal{A}(\omega)$ that accepts the state trajectories of $\mathcal{T}_\omega(M_a, s_0)$. Since our proposal consists in formulating the Observation Decoding problem as a planning problem $\mathcal{P}(\omega)$, the next translation step is to encode the automaton $\mathcal{A}(\omega)$ as part of the planning problem. In other words, the outcome of the compilation will be a planning problem whose search space is restricted to the state trajectories that explain ω ($\mathcal{T}_\omega(M_a, s_0)$).

Therefore, our aim is to build a planning problem $\mathcal{P}(\omega)$ that incorporates the monitor automaton $\mathcal{A}(\omega)$. In order to do so, we extend the action model of the actor M_a with a new set of state variables and actions to encode $\mathcal{A}(\omega)$. The resulting $\mathcal{P}(\omega)$ can thus be regarded as the parallel composition of the state-space model of the actor and the automaton $\mathcal{A}(\omega)$.

Formally, an Observation Decoding problem $\langle M_a, M_s, s_0, \omega \rangle$ is compiled into a classical planning problem $\mathcal{P}(\omega)$ defined as follows:

$$\mathcal{P}(\omega) = \langle X \cup X^\omega, A \cup A^\omega, \mathsf{App} \wedge \mathsf{App}^\omega, \mathsf{Eff} \wedge \mathsf{Eff}^\omega, s_0^\omega, G^\omega \rangle$$

where:

- $X \cup X^\omega$ extends the sets of state variables X with $X^\omega = \{\mathsf{q}\}$ such that variable q is used to keep track of the location of $\mathcal{A}(\omega)$. Variable q has domain $D(\mathsf{q}) = \{0, \ldots, t\}$ where $|\omega| = t$.
- $A \cup A^\omega$ extends the actor actions with $A^\omega = \{a^{o_1}, \ldots, a^{o_t}\}$, which contains one action for each transition in $\mathcal{A}(\omega)$ that advances the location of the automaton. We will refer to the actions in A^ω as *sensing actions*.
- The applicability and effects constraints are extended to account for the new sensing actions and used to encode the transitions $q_i = \delta(q_{i-1}, \Phi_{o_i})$ of $A(\omega)$. More details of this encoding will be given below.
- the initial state s_0^ω is defined as $s_0^\omega(x) = s_0(x)$ for $x \in X$ and $s_0^\omega(\mathsf{q}) = 0$.
- G^ω is the condition given by $\langle \mathsf{q}, \text{"}\mathsf{q} = t\text{"} \rangle$.

The initial state s_0^ω sets the initial situation of the actor agent given by s_0 as well as the initial location q_0 of the automaton $\mathcal{A}(\omega)$. The goal condition G^ω is to reach the accepting location q_t of $\mathcal{A}(\omega)$ which is interpreted as having satisfied the LTL_f property $\mathsf{ord}(\omega)$ and ensures that all solution plans generate explanations for ω. A sensing action $a^{o_i} \in A^\omega$ encodes the automaton transition $q_i = \delta(q_{i-1}, \Phi_{o_i})$. Then, a sensing action a^{o_i} is applicable

in a state s when 1) the automaton is in location q_{i-1}, i.e., $s \models \langle \mathtt{q}, \text{"}\mathtt{q} = i - 1\text{"} \rangle$, and 2) $s \models \Phi_{o_i}$, meaning that s accepts the observation o_i. The execution of a^{o_i} updates the value of $\mathtt{q}$ to reflect that $\mathcal{A}(\omega)$ transitions to the next location q_i. Hence, the effect constraint of the sensing actions, $\mathsf{Eff}^{\omega} : A^{\omega} \rightarrow \mathsf{Constr}(X \cup X^{\omega} \cup X^{\omega+})$, does not update the value of state variables X and, therefore, sensing actions do not modify the state of the actor. We also note that loop transition $q_i = \delta(q_i, \neg\Phi_{o_i})$ of $\mathcal{A}(\omega)$ are not encoded as part of $\mathcal{P}(\omega)$ since they can be considered *noop* actions with no effect on the state of the problem.

A plan π that solves the compiled problem $\mathcal{P}(\omega)$ is a sequence composed of both actor actions and sensing actions that starting from the initial state s_0^{ω} reaches a state $s \models G^{\omega}$. Any solution plan for $\mathcal{P}(\omega)$ must contain all actions in A^{ω} exactly once so a solution plan contains t sensing actions if t is the length of the observation sequence. Additionally, the last action of π is always the sensing action a^{o_t} as this is the only action that can achieve the goal condition $G^{\omega} : \langle \mathtt{q}, \text{"}\mathtt{q} = \mathtt{t}\text{"} \rangle$.

Following, we present two different encodings for the sensing actions and discuss their efficiency as well as the scenarios where each one could be applied.

Basic encoding of A^{ω}. Let $a^{o_i} \in A^{\omega}$ be the sensing action which represents the transition $q_i = \delta(q_{i-1}, \Phi_{o_i})$ of the automaton $\mathcal{A}(\omega)$. The basic encoding simply consists in defining the applicability constraint of a^{o_i} so that it checks that the automaton is in the starting location q_{i-1} of the transition and that the actor's state accepts the observation o_i, i.e., it satisfies Φ_{o_i}. The effect of a^{o_i} will simply update the location of the automaton to the destination location q_i. Formally:

$$\mathsf{App}(a^{o_i}) : \Phi_{o_i} \cup \{\langle \mathtt{q}, \text{"}\mathtt{q} = \mathtt{i} - 1\text{"} \rangle\}$$

where the starting location of the transition q_{i-1} is encoded as $\mathtt{q} = i - 1$, and

$$\mathsf{Eff}(a^{o_i}) : \langle \mathtt{q}^{+}, \text{"}\mathtt{q}^{+} = \mathtt{i}\text{"} \rangle$$

meaning that the automaton $\mathcal{A}(\omega)$ transits to the destination location q_i which is encoded as $\mathtt{q} = i$.

Encoding of A^{ω} with dead-ends. In this encoding, the applicability constraint of the sensing action a^{o_i} only checks that the automaton $\mathcal{A}(\omega)$ is in the starting location q_{i-1} of the transition. Formally:

$$\mathsf{App}(a^{o_i}) : \{\langle \mathtt{q}, \text{"}\mathtt{q} = \mathtt{i} - 1\text{"} \rangle\}$$

This makes a^{o_i} applicable even in states that do not accept the observation o_i. We deal with this situation by making the sensing action a^{o_i} introduce an artificial dead-end that makes the problem $\mathcal{P}(\omega)$ unsolvable if it is executed in a state that does not accept the observation. In order to create the dead-end, we extend the state variables of the planning problem $\mathcal{P}(\omega)$ with a Boolean variable `valid` that must be true in the goal state.

Creating a dead-end involves defining constraints $\mathsf{dend}(\varphi)$ for every $\varphi \in \Phi_{o_i}$ that introduce the dead-end $\mathtt{valid} = \bot$ iff φ is not satisfied. This way, if any of the observed values in o_i cannot be emitted from the current state a dead-end is created invalidating the solution. Formally, a $\mathsf{dend}(\varphi)$ constraint defined over $\varphi = \langle Z, R\rangle$ is of the form:

$$\mathsf{dend}(\varphi) = \langle Z \cup \{\mathtt{valid}^+\}, \bar{R} \times \{\bot\}\rangle$$

where $|Z| = k$ and $\bar{R} = D(Z)^k \setminus R$. Since the negation of a constraint $\varphi = \langle Z, R\rangle$ is the constraint $\bar{\varphi} = \langle Z, \bar{R}\rangle$, a $\mathsf{dend}(\varphi)$ constraint can be interpreted as "if $\bar{\varphi}$, then create the dead-end $\mathtt{valid} = \bot$".

The effects of the sensing action consider the dead-end constraint imposed by each emission in the observation and updates the location of the automaton $\mathcal{A}(\omega)$. Formally, given $\Phi_{o_i} = \{\varphi^{y_1}_{w_1}, \ldots, \varphi^{y_l}_{w_l}\}$, we define the effects constraint of the sensing action a^{o_i} as

$$\mathsf{Eff}(a^{o_i}) : \langle \mathtt{q}^+, \text{"}\mathtt{q}^+ = \mathtt{i}\text{"}\rangle \bigcup_{j=1}^{k} \mathsf{dend}(\varphi^{y_j}_{w_j})$$

This way, the effect constraint states that the solution remains valid if the observation o_i is accepted, but if it is not, then the solution is invalidated through the introduction of the dead-end.

Notes on Implementation

In this section, we show how to implement each encoding in PDDL, and discuss the limitations and advantages of each one. We start with an example of a sensing action implemented in PDDL. In order to do this we are going to go back to the example of Figure 4.3, and implement the sensing action associated to the first observation $o_1 = \langle \mathtt{obs\_loc} = (3, 2)\rangle$. If we follow the basic encoding, we need to implement the following constraints

$$\mathsf{Pre}(\mathtt{sense\_1}) : \langle \mathtt{q}, \text{"}\mathtt{q} = 0\text{"}\rangle \cup \varphi_{(3,2)}$$
$$\mathsf{Eff}(\mathtt{sense\_1}) : \langle \mathtt{q}^+, \text{"}\mathtt{q}^+ = 1\text{"}\rangle$$

where $\mathtt{sense\_1}$ is the name given to this sensing action, and $\varphi_{(3,2)} = \langle \mathtt{loc}, \{(3, 2)\}\rangle$.

Figure 4.5 shows how this action is implemented in PDDL. Since PDDL uses propositional variables, assume that variable $\mathtt{loc}$ is implemented by the propositional variables resulting from the grounding of predicate $\mathtt{at}$ of arity 1 and objects $O = \{\mathtt{tile\_i\_j} \mid i, j \in \mathtt{N}^{\leq 5}\}$ representing tiles of the grid. Additionally, the propositional variables $\mathtt{q\_i}$, $i \in 0 \ldots t$, implement the variable $\mathtt{q}$ that represents the current location in the automaton.

```
(:action sense_1
     :parameters ()
     :precondition
          (and (q_0)
               (at tile_3_2)
           )
     :effect
          (and (not (q_0))
               (q_1)
           )
)
```

Figure 4.5: PDDL implementation of a sensing action following the basic encoding.

As we can see in the figure, the implementation of the basic encoding is straightforward. However, it is important to take into account that a constraint $\varphi_w^y = \langle Z, R \rangle$ with $|R| > 1$ means that there are multiple possible ways to emit $y = w$. In other words, φ_w^y defines a disjunction where each tuple in R is a disjunct. This makes the basic encoding undesirable when we have constraints with $|R| > 1$ as it leads to disjunctive preconditions which would rapidly increase the branching factor of the compiled problem.

Next, we are going to implement the same action following the encoding with dead-ends. This time the constraints to implement are

$$\mathsf{Pre}(\texttt{sense\_1}) : \langle \mathsf{q}, \text{"}\mathsf{q} = 0\text{"} \rangle$$
$$\mathsf{Eff}(\texttt{sense\_1}) : \langle \mathsf{q}^+, \text{"}\mathsf{q}^+ = 1\text{"} \rangle \cup \mathsf{dend}(\varphi_{(3,2)})$$

where

$$\mathsf{dend}(\varphi_{(3,2)}) = \langle \{\texttt{loc}, \texttt{valid}^+\}, \bar{R} \times \{\bot\} \rangle$$

and $\bar{R} = D(\texttt{loc}) \setminus (3, 2)$

The most challenging part of this encoding is the implementation of the dead-end constraints which is done with conditional effects. To this end, for each tuple in the relation $\bar{R}$ we define a conditional effect where the condition is given by the tuple and the effect is $\texttt{valid} = \bot$. Assuming that the cardinality of the relations $\bar{R}$ of the dead-end constraint is b, i.e., $|\bar{R}| = b$, the number of conditional effects in this encoding scales linearly with the number of observable variables $|Y| = l$ so that each sensing action contains $b \cdot l$ conditional effects.

Figure 4.6 shows the PDDL implementation of the `sense_1` sensing action of our example following the encoding with dead-ends. As we can see, encoding the constraint $\mathsf{dend}(\varphi_{(3,2)})$ has resulted in 24 conditional effects, one for each location that cannot emit `obs_loc` $= (3, 2)$. From this example, it is clear that the use of the basic encoding is preferable as long as the constraints have cardinality 1.

```
(:action sense_1
     :parameters ()
     :precondition
          (and (q_0))
     :effect
          (and (not (q_0))
               (q_1)
               (when (at tile_1_0) (not (valid)))
               (when (at tile_1_1) (not (valid)))
               (when (at tile_1_2) (not (valid)))
               (when (at tile_1_3) (not (valid)))
               (when (at tile_1_4) (not (valid)))
               (when (at tile_1_5) (not (valid)))
               (when (at tile_2_0) (not (valid)))
               (when (at tile_2_1) (not (valid)))
               (when (at tile_2_2) (not (valid)))
               (when (at tile_2_3) (not (valid)))
               (when (at tile_2_4) (not (valid)))
               (when (at tile_2_5) (not (valid)))
               (when (at tile_3_0) (not (valid)))
               (when (at tile_3_1) (not (valid)))
               (when (at tile_3_3) (not (valid)))
               (when (at tile_3_4) (not (valid)))
               (when (at tile_3_5) (not (valid)))
               (when (at tile_4_0) (not (valid)))
               (when (at tile_4_1) (not (valid)))
               (when (at tile_4_2) (not (valid)))
               (when (at tile_4_3) (not (valid)))
               (when (at tile_4_4) (not (valid)))
               (when (at tile_4_5) (not (valid)))
               (when (at tile_5_0) (not (valid)))
               (when (at tile_5_1) (not (valid)))
               (when (at tile_5_2) (not (valid)))
               (when (at tile_5_3) (not (valid)))
               (when (at tile_5_4) (not (valid)))
               (when (at tile_5_5) (not (valid)))
       )
)
```

Figure 4.6: PDDL implementation of a sensing action following the encoding with dead-ends.

As a final summary, on one hand we have the basic encoding which is more efficient but is limited to very restrictive sensor models whose constraints do not define disjunctions. On the other hand, the encoding with dead-ends is flexible to implement any type of sensor model but the use of conditional effects and dead-ends makes the compiled problem much harder. As a rule of thumb, we recommend using the basic encoding whenever applicable and the encoding with conditional effects in the rest of cases. It is also worth noting that the very common assumption that the values of state variables are directly observed can be cast into a sensor model that fits the criteria for the basic encoding. In order to build this type of sensor model, we only need to define an observable variable y for each state variable $x \in X$ such that $D(y) = D(x)$ and define the sensing function as $\mathsf{sense}_y(\langle x, "x = v" \rangle) = v$ so that the actual value of x is always emitted.

4.4.4. Properties of the compilation

A solution plan $\pi \in \Pi(\mathcal{P}(\omega))$ is a sequence of actor and sensing actions of the form $\pi = (a_1, \ldots, a_{b_1}, \ldots, a_{b_2}, \ldots, a_{b_t})$ with $a_{b_i} = a^{o_i}$ for $1 \leq i \leq t$. π contains all the information needed to generate an explanation τ for the observation sequence ω. Looking only at the actor actions in the solution, we can extract the actor plan $\pi^A = (a_1, \ldots, a_{b_1-1}, a_{b_1+1}, \ldots, a_{b_{t-1}-1}, a_{b_{t-1}+1}, \ldots, a_{b_t-1})$ where only the sensing actions, $a_{b_1}, \ldots, a_{b_t}$, of the solution plan π are missing in π^A.

Theorem 1. *Let π be a solution plan for $\mathcal{P}(\omega)$ and let π^A be the actor plan, the execution of π^A in the actor's initial state s_0 generates a trajectory τ that explains the observation sequence ω.*

Proof: Let $\pi_{i:i'}$ denote the subsequence of all the actions from a_i to $a_{i'}$ of plan π, and let $b_1, \ldots, b_t$ be the indexes of the t sensing actions in π. We can prove by induction that the trajectory τ, generated by the execution of π^A in the actor initial state s_0, explains ω.

Let $\pi^A_{1:b_1} = \pi_{1:b_1-1}$ be the subsequence of π containing all actor actions up to the first sensing action. The basis of the induction is to prove that the execution of $\pi^A_{1:b_1}$ in the actor's initial state s_0 reaches a state that accepts o_1, that is, $s_0[\pi^A_{1:b_1}] \models \Phi_{o_1}$. Given that $a_{b_1} = a^{o_1}$, its pre-state $s^\omega_0[\pi^A_{1:b_1}]$ necessarily satisfies Φ_{o_1} otherwise a^{o_1} would either not be applicable or $s^\omega_0[\pi] \not\models G^\omega$ (depending on the used encoding) invalidating π as a solution. Since the actions of the actor A neither have their applicability dependant on X^ω nor modify X^ω by definition of the effects constraint $\mathsf{Eff} : A \rightarrow \mathsf{Constr}(X \cup X^+)$, the trajectories generated by executing $\pi^A_{1:b_1}$ starting from s^ω_0 and from s_0 are identical with respect to variables X. Therefore, if $s^\omega_0[\pi^A_{1:b_1}] \models \Phi_{o_1}$ then $s_0[\pi^A_{1:b_1}] \models \Phi_{o_1}$ too.

Now let $\pi^A_{b_i:b_{i+1}} = \pi_{b_i+1:b_{i+1}-1}$ be the subsequence of π containing all actor actions between the sensing actions that occupy positions b_i and b_{i+1}, $1 \leq i < t$. For the induction step we need to prove that from the state $s = s_0[\pi^A_{1:b_i}]$ reached by applying all actor actions up to sensing action $a_{b_i} = a^{o_i}$ we can reach a state that accepts observation o_{i+1} by executing the subsequence of actions $\pi^A_{b_i:b_{i+1}}$, that is, $s[\pi^A_{b_i:b_{i+1}}] \models \Phi_{o_{i+1}}$. Assuming that $s' = s^\omega_0[\pi_{1:b_i}]$ and using the same reasoning as used above for the basis of the induction we can be sure that $s'[\pi^A_{b_i:b_{i+1}}] \models \Phi_{o_{i+1}}$. We also know that $\pi_{1:b_i}$ and $\pi^A_{1:b_i}$ only differ in the sensing actions and that sensing actions, by definition of the effects constraint $\mathsf{Eff}^\omega : A^\omega \rightarrow \mathsf{Constr}(X \cup X^\omega \cup X^{\omega+})$, do not modify state variables X. Therefore, $\forall x \in X$ it holds that $s(x) = s'(x)$ and if $s'[\pi^A_{b_i:b_{i+1}}] \models \Phi_{o_{i+1}}$ then $s[\pi^A_{b_i:b_{i+1}}] \models \Phi_{o_{i+1}}$. □

The position of the sensing action in the solution plan π also determines the alignment between the trajectory $\tau = s_0[\![\pi^A]\!]$ and the observation sequence ω.

Corollary 1. *Let π be a solution plan for Π and let $b_1, \ldots, b_t$ be the indexes of the t sensing actions in π. The alignment between the explanation $\tau = s_0[\![\pi^A]\!]$ and the observation sequence ω is $\alpha(i) = b_i - i$ for $i \in \{1, \ldots, t\}$*

We already know from Theorem 1 that the pre-states of the sensing actions in the

solution plan π are the states that accept the observations and therefore the points where the explanation τ and the observation sequence ω align. Additionally, it is easy to see that $|\pi^A_{1:b_i}| = |\pi_{1:b_i}| - i$ as $\pi^A_{1:b_i}$ contains the same actions as $\pi_{1:b_i}$ except for the i sensing actions in that segment of the plan. From this, we arrive at the alignment $\alpha(i) = b_i - i$ for $i \in \{1, \ldots, t\}$ by offsetting the position b_i of the sensing action a^{o_i} by i.

Theorem 2. *For every trajectory τ that explains the observation sequence ω, there exists a solution plan π to the compiled problem $\mathcal{P}(\omega)$ such that $\tau = s_0[\![\pi^A]\!]$.*

Proof: Assume that τ is an explanation for ω under alignment α and let $\pi' = (a_1, a_2, \ldots, a_n)$ be the sequence of actor actions such that $\tau = s_0[\![\pi']\!]$. Now let π be a plan built by adding sensing actions to π' such that a^{o_i} is inserted after action $a_{\alpha(i)}$ for $i \in \{1, \ldots, t\}$. We proof that π is indeed a solution to $\mathcal{P}(\omega)$ by contradiction. If π is not a solution to $\mathcal{P}(\omega)$ it means that either π is not a valid plan or it does not achieve the goal condition. For π to be invalid it is necessary that the inserted sensing actions are not applicable or their execution makes following actions not applicable. We know that sensing actions do not modify the actor state variables X so their execution cannot affect the applicability of following actions. Additionally, a sensing action a^{o_i} not being applicable in state $s = s^{\omega}_0[\pi'_{1:\alpha(i)}]$ implies that $s \not\models \Phi_{o_i}$ and consequently that $s_0[\pi'_{1:\alpha(i)}] \not\models \Phi_{o_i}$ which contradicts the initial assumption that τ is an explanation for ω under alignment α. □

4.4.5. Computing the most likely explanation

Now that we have demonstrated that valid explanations for an observation sequence can be extracted from solution plans to our compiled problem $\mathcal{P}(\omega)$, our next step is to find the most likely explanation. In order to accomplish this, we define the cost of a solution plan $\pi \in \Pi(\mathcal{P}(\omega))$ to be the same as the cost of the explanation contained in π. In this way, by optimally solving $\mathcal{P}(\omega)$ we will be able to find a plan from where the most likely explanation can be extracted.

Let π be a solution to the compiled problem $\mathcal{P}(\omega)$ and $\tau = s_0[\![\pi^A]\!]$ the explanation generated by the embedded actor plan π_A. We want the cost of the solution plan π to be

$$cost(\pi) = synCost(\tau) + senCost_{\alpha}(\tau, \omega) \tag{4.12}$$

where α is the alignment denoted by the sensing actions. By definition, $synCost(\tau) = cost(\pi^A)$ so the synthesis cost is already given by the cost of the actor actions in π. Then, our objective is to define the action costs of sensing actions in such a way that their accumulated cost is equal to the sensing cost of the explanation.

$$senCost_{\alpha}(\tau, \omega) = \sum_{i=1}^{t} ocost(s_{\alpha(i)}, o_i) = \sum_{i=1}^{t} c(a^{o_i})$$

From this equation we can establish an equivalence between the action cost of a sensing action a^{o_i} and the observation cost $ocost(s, o_i)$ where s is the pre-state of a^{o_i} and also the state aligned with observation o_i. In order to define such a cost, we need to redefine action costs as state-dependant $c_s(a)$ so that the cost of executing action a depends on the state s where it is executed. Then, the cost of executing a sensing action a^{o_i} in a state s such that $s \models \Phi_{o_i}$ is given by the observation cost of the observation o_i in state s.

$$c_s(a^{o_i}) = ocost(s, o_i)$$

With this we have achieved our objective of Equation 4.12 and the actor plan π^A embedded in the optimal solution plan π^* will render the most likely explanation $\tau^* = s_0[\![\pi^A]\!]$. We also note that, when the observation cost is a constant (because the sensor model does not include sensing cost functions), all sensing actions are assigned the same cost. Therefore, all solution plans in $\Pi(\mathcal{P}(\omega))$ will have the same accumulated cost for the sensing actions and will only differ in the cost of the actor actions. This is the same as ranking explanations only by their synthesis cost which is what happens when the sensor model does not include sensing cost functions.

4.4.6. A walk-through of the compilation

In this section we revisit the *Blindspots* example of Figure 4.3 to walk the reader through the compilation scheme. Recall that the actor is initially situated at tile $(3, 1)$, and the observation sequence contains four observations that show: first, the actor is observed at tile $(3, 2)$; the next two observations are given by the inability of the camera to capture the actor's location as it is in a covered tile; lastly, the actor is seen at tile $(3, 5)$.

Time	Observation	Sensing action
1	obs_loc = $(3, 2)$	sense_1
2	obs_loc = unknown	sense_2
3	obs_loc = unknown	sense_3
4	obs_loc = $(3, 5)$	sense_4

Following our compilation, we extend the set of state variables with a variable q with domain $D(\mathtt{q}) = \{0, \ldots, 4\}$, where $\mathtt{q} = 0$ represents the starting location of the automaton, and $\mathtt{q} = 4$ the accepting location. In order to encode the transitions of the automaton we need four sensing actions, one for each observation, as indicated in the above table. The sensing action sense_1 is defined as follows

$$\mathsf{Pre}(\mathtt{sense\_1}) : \langle \mathtt{q}, \text{"}\mathtt{q} = 0\text{"} \rangle$$
$$\mathsf{Eff}(\mathtt{sense\_1}) : \langle \mathtt{q}^+, \text{"}\mathtt{q}^+ = 1\text{"} \rangle \cup \mathsf{dend}(\varphi_{(3,2)})$$

where $\mathsf{dend}(\varphi_{(3,2)})$ is the dead-end constraint defined over $\varphi_{(3,2)} = \langle \texttt{loc}, \text{"}\texttt{loc} = (3,2)\text{"} \rangle$. This constraint is defined as

$$\mathsf{dend}(\varphi_{(3,2)}) = \langle \{\texttt{loc}, \texttt{valid}^+\}, \text{"}\texttt{loc} \neq (3,2)\text{"} \times \{\bot\} \rangle$$

where "$\texttt{loc} \neq (3,2)$" the relation $\bar{R} = D(\texttt{loc}) \setminus \{(3,2)\}$. Action `sense_1` encodes the transition that advances the location of the automaton $\mathcal{A}(\omega)$ from location q_0 to q_1 when observation o_1 is accepted. The only precondition of the action is $\texttt{q} = 0$, which is initially true. The execution of `sense_1` updates the value of `q` through effect "$\texttt{q}^+ = 1$", which encodes the change of location in the automaton. However, executing `sense_1` in a state that satisfies "$\texttt{loc} \neq (3,2)$" will introduce a dead-end $\texttt{valid}^+ = \bot$. Therefore, the only way to reach the goal is to execute `sense_1` in state $s = \{\texttt{loc} = (3,2), \texttt{q} = 0\}$.

Action `sense_2` is an interesting case as the emission $\texttt{obs\_loc} = \texttt{unknown}$ is valid no matter the location of the actor. This means that the dead-end constraint is unnecessary and the action is simply encoded as

$$\mathsf{Pre}(\texttt{sense\_2}) : \langle \texttt{q}, \text{"}\texttt{q} = 1\text{"} \rangle$$
$$\mathsf{Eff}(\texttt{sense\_2}) : \langle \texttt{q}^+, \text{"}\texttt{q}^+ = 2\text{"} \rangle$$

As we can see, `sense_2` only checks and updates variable `q` so it becomes applicable immediately after `sense_1` is executed. The two remaining sensing actions follow the same structure as `sense_2` and `sense_1`, respectively.

Regarding the cost of the sensing actions we identify the following cases:

- Actions `sense_1` and `sense_4` have a cost of $-\log 0.9$ (because the probability that the camera reliably captures the location of the agent when it is at an open tile is 0.9).
- Actions `sense_2` and `sense_3` have a cost of $-\log 0.1$ when the actions are executed in a state in which the agent is located at an open tile.
- Actions `sense_2` and `sense_3` have a cost of $-\log 1$ in states where the agent is located on a covered tile (because the probability that the camera is unable to locate the agent when it is at a covered tile is 1).

Next, we discuss the solution to the classical planning problem generated by our compilation. It is worth noting that any solution plan will contain the four sensing actions executed in order. Solving the planning problem optimally renders the following plan.

#	Action	Cost
1	move$((3,1),(3,2))$	$-\log 0.25$
2	sense_1	$-\log 0.9$
3	move$((3,2),(2,2))$	$-\log 0.25$
4	move$((2,2),(2,3))$	$-\log 0.25$
5	sense_2	$-\log 1$
6	move$((2,3),(2,4))$	$-\log 0.25$
7	sense_3	$-\log 1$
8	move$((2,4),(2,5))$	$-\log 0.25$
9	move$((2,5),(3,5))$	$-\log 0.25$
10	sense_4	$-\log 0.9$

This plan contains all the information needed to build the solution to our Observation Decoding problem. First, the execution of the actor's actions (numbered 1,3,4,6,8, and 9 in the plan) in the initial situation of the actor generates the following trajectory τ that explains the observation sequence.

Time	State
0	loc $= (3,1)$
1	loc $= (3,2)$
2	loc $= (2,2)$
3	loc $= (2,3)$
4	loc $= (2,4)$
5	loc $= (2,5)$
6	loc $= (3,5)$

Next, the sensing actions (numbered 2,5,7, and 10 in the plan) indicate the alignment between the explanation and the observation sequence. More specifically, a sensing action indicates an alignment between the state where it is executed and its associated observation. For instance, the action sense_1 indicates and alignment between the post-state of move$((3,1),(3,2))$, $\langle$loc $= (3,2)\rangle$, and the observation $\langle$obs_loc $= (3,2)\rangle$. Considering this, the solution plan shows the following alignment α:

$$\alpha(1) = 1$$
$$\alpha(2) = 3$$
$$\alpha(3) = 4$$
$$\alpha(4) = 6$$

meaning that the first observation is emitted by state 1 of the trajectory τ, the second observation obs_loc = unknown is emitted by state 3 of the trajectory, the third obser-

vation `obs_loc = unknown` is emitted by state 4 and finally the fourth observation is emitted by the last state (6) of τ.

Lastly, the cost of the solution plan is exactly

$$synCost(\tau) + senCost_{\alpha}(\tau, \omega) = 6(-\log 0.25) + 2(-\log 0.9) + 2(-\log 1)$$

where:

- The term $synCost(\tau)$ is the accumulated cost of the actor's actions; in this case, the resulting value is given by the number of transitions of the agent across the cells of the grid (6 transitions) and the probability of each transition, which is 0.25.
- The term $senCost_{\alpha}(\tau, \omega)$ is the accumulated cost of the sensing actions (see the table of the solution plan).

4.5. Experimental evaluation

In this experimental evaluation we analyse the performance of our approach to infer the real trajectory followed by the actor agent. We also want to evaluate the benefit of exploiting sensing cost functions when this information is available so we will compare our approach when the sensor model includes sensing costs, OD_S, to when it does not, OD_N.

The evaluation consists in providing the same observation sequence ω to both OD_S and OD_N and measuring the similarity of the most likely explanation found by each approach with the actual trajectory that generated ω. In order to compare the similarity between two trajectories we use the *plan diversity* metric presented in (Srivastava et al., 2007) to instead measure the diversity (or similarity) of the plans that generated each trajectory. This is an appropriate approach when actions have deterministic effects and provides some benefits such as being insensitive to symmetries. More specifically, we selected the δ_{α} variant of the diversity metric that interprets a plan as a bag of actions and computes the similarity between two plans as the set-difference between the two. Formally:

$$\delta_{\alpha}(\pi_i, \pi_j) = \frac{|B_i - B_j|}{|B_i| + |B_j|} + \frac{|B_j - B_i|}{|B_i| + |B_j|}$$

where B_i and B_j are the bag of actions of plans π_i and π_j, respectively. A value of 0 for this metric represents complete similarity of plans while a value of 1 represents complete diversity. For this experiment we computed $\delta_{\alpha}(\pi, \pi_S)$ and $\delta_{\alpha}(\pi, \pi_N)$ where π is the original plan followed by the actor, and π_S and π_N are the plans corresponding to the most likely explanations found by OD_S and OD_N, respectively.

4.5.1. Evaluation Benchmark

We created a benchmark of Observation Decoding problems defined over planning domains. The action and sensor models for these domains are defined in similar fashion to our *Blindspots* working example. This means that we make the state partially observable by defining counterpat observable variables for some of the state variables (as in the `obs_loc` and `loc` variables), and extending their domain with the `unknown` emission that represents a sensor failure. Additionally, we use the sensing functions and sensing cost functions to split the state space into regions of high (H) and low (L) observability. For instance, in the Blindspots working example, open tiles define the H region and covered tiles the L region. Following we briefly describe the domains used and how they were modified for this evaluation:

- ***Blindspots*:** This is the domain of our working example. Briefly, the only observable variable is the location of the actor which is more visible from the open tiles than from covered tiles. For the creation of the benchmark, the grids were randomly generated as well as the initial and final positions of the actor.
- ***Intrusion*:** This domain emulates attacks on host computers (Pereira et al., 2017; Ramírez & Geffner, 2010). The attacker needs to perform a number of steps in order to steal data or vandalize a host. We added a few modifications to allow more ways to accomplish the goal and made it so that some attack paths are more observable than others. In this way the attacker can choose between a faster attack approach that is easier to detect (by the observer) or a slower but safer one.
- ***2-handed Blocksworld*:** The classic *blocksworld* domain with four operators but using an additional hand. We define several variables to emit the status of a block, and one variable for each hand, one of them highly observable.
- ***Office*:** This is a transportation domain used in (Alford et al., 2009), where a robot needs to pick up and deliver packages in a building. The building is represented as rooms connected by doors, and the robot may need to go through many rooms to deliver a package. In this domain, we define an observable variable for each package and make it so packages are easily traceable (high observability) in some rooms while in others not (low observability).

For each of these domains, we defined three sensor models with observability 100-0, 80-20 and 60-40 for the H and L regions by tuning the sensing cost functions. For instance, the Blindspots sensor model with observability 100-0 means that the actor can always be located in an open tile but never in a covered tile. The observation sequences used as input of the decoding problems were generated by applying each sensor model to a dataset consisting of 50 trajectories of the corresponding action model. We also simulated an intermittent sensing process by removing observations consisting only of `unknown` emissions, thus creating gaps in the observation sequence.

4.5.2. Results

Following, we present the experimental results obtained by solving our evaluation benchmark with the OD_S and OD_N configurations of our approach. All experiments were run on an Intel Core i5 3.10GHz x 4 16GB of RAM. The compiled planning problems were optimally solved with an A* search guided by the blind heuristic (h=0), using the search code of the METRIC-FF (Hoffmann, 2003) planner.

Table 4.1 compiles the results for the four domains. The values of column H (high observability) and column L (low observability) denote the observability degree of each region. For both OD_S and OD_N we report the quality of the solutions (measured as plan diversity) and run-time, averaged over 50 Observation Decoding problems.

	M_s		OD_S		OD_N	
Domain	H	L	$\delta_\alpha(\pi, \pi_S)$	t	$\delta_\alpha(\pi, \pi_N)$	t
BLINDSPOTS	100	0	0.04	0.32	0.22	0.02
	80	20	0.08	0.10	0.20	0.02
	60	40	0.11	0.17	0.17	0.03
INTRUSION	100	0	0	0.02	0.58	0.78
	80	20	0.07	1.45	0.18	0.81
	60	40	0.13	3.57	0.14	0.76
BLOCKS 2H	100	0	0	0.20	0.34	78.748
	80	20	0.05	0.150	0.27	74.440
	60	40	0.07	0.122	0.26	76.020
OFFICE	100	0	0	0.20	0.58	0.20
	80	20	0.23	0.57	0.38	0.19
	60	40	0.16	4.88	0.23	0.19

Table 4.1: Comparison between OD_S and OD_N

We can observe that the average diversity of OD_N is always higher than OD_S, meaning that OD_S is able to find explanations that are closer to the original trajectory. Interestingly, the difference between OD_S and OD_S narrows down as the difference between the observability of the H and L regions decreases. This results seems to indicate that as we get closer to a uniform distribution (which would be a 50-50 sensor model) the information provided by the sensing cost function decreases. This is perfectly reasonable since not considering the sensing functions is equivalent to assuming a uniform distribution of observations; in other words, once we reach the uniform distribution (50-50 sensor model) both OD_S and OD_S become equivalent. This is also the reason why the quality of the solutions found by OD_S (resp. OD_S) increases (resp. decreases) as we go from the 100-0 to the 60-40 sensor model. This phenomena is manifested in a fairly gently way, with the exception of the significant diversity drop of OD_N from sensor model 100-0 to 80-20 in the INTRUSION domain. This happens because only two solution paths exist for these problems and increasing the low observability degree from 0 to 20 uncovers some

variables that identify almost unambiguously the solution path.

Regarding run-time, it is difficult to draw conclusions as results vary from domain to domain. We see domains like *office* and *blindspots* where OD_N is faster, and domains like *blocks 2h* where the opposite is true. However, it is important to point out that the state-dependant costs needed to compile the sensing cost functions in OD_S forces as to use an A* search guided by a blind heuristic, which in turn forces us to keep the problems at a small size. This is not a problem for OD_N, as it does not use state-dependant costs, and can benefit from planning heuristics.

In conclusion, exploiting the information provided by sensing cost functions does indeed give us an edge and improve the quality of the explanations found to the Observation Decoding problems. However, the planning community has so far ignored the research of heuristics that support state-dependant costs, a feature needed to exploit this information, which limits its applicability to small problems. In light of these results, in the following chapters we will assume that the sensor model does not include sensing costs in order to tackle more interesting problems.

4.6. Discussion

While in this thesis we restrict our attention to the classical planning model, we believe that the proposed approach naturally extends to more expressive planning models such as hybrid planning, temporal planning or non-deterministic planning. The hybrid setting, in particular, looks very appealing in our view as it allows the representation of dynamical systems that exhibit a discrete and continuous behaviour, and captures the control of continuously evolving physical activities typical in automated manufacturing, chemical engineering and robotics systems. We can find preliminary results of what this extension looks like in (Aineto, Onaindia, et al., 2020, 2022), where the problem of finding explanations in a hybrid setting is formalized using the Hybrid Automata formalism and then translated to PDDL+ planning.

On the topic of the sensor model, extending observations to handle noise over state variables is proposed in (Sohrabi et al., 2016) but this formulation is constrained to a single fluent per observation and noisy observations are handled by skipping them. The idea of skipping observations is better suited for atomic observations (such as actions) that are either right or wrong. In our formulation we have several observable variables, and a single observation may contain a noisy reading for one variable and a correct one for another. Our model-based approach to noise, or non-determinism in general, is much more flexible as it checks whether that a reading is possible with the given sensor model. It is also worth pointing out that our proposal of compiling the monitor $\mathcal{A}(\omega)$ away in order to restrict solutions to explanations for ω is compatible with existing inference approaches. Our compilation can be plugged directly into Ramírez and Geffner's Goal Recognition problem (Ramírez & Geffner, 2010) and other derived approaches with the advantage of

being able to exploit richer sensor models which has the added benefit of making away with the existing limitation of single element observations.

An important topic for discussion is the scalability of the proposed approach and how it can be improved. It is well known that compilations to planning tend to sacrifice scalability for the ability to be solved with off-the-shelf planners. It is our belief that a more integrated approach that interprets observations as landmarks (and the monitor automaton as a landmark graph) and incorporates them inside a search scheme would improve scalability considerably. Similar ideas have been proposed for Goal Recognition (Pereira et al., 2017, 2020; Vered et al., 2018) where the scores of landmark-based heuristics are used to predict the most likely goal.

The last topic we want to touch upon is related to the information conveyed by an observation. It is important to understand that the gain of information is not the same for all observations. This is something that we can easily see in our working example. Consider, for instance, the observations $\langle \texttt{obs\_loc} = \texttt{unknown} \rangle$ and $\langle \texttt{obs\_loc} = (3,2) \rangle$. It is clear that the latter provides more information since it can only be emitted by state $\langle \texttt{loc} = (3,2) \rangle$ while the former can be emitted from any state. In that regard, being able to observe the actions executed by the actor, as is the case in many inference approaches, is very informative as it provides information about both the pre-state and post-state of the action.

5. TEMPORAL INFERENCE

This chapter revolves around exploiting the action model of the actor and the sensing model of the observer to do Temporal Inference, that is, for uncovering the (past) behaviour of the agent prior to some observation, the (present) behaviour at the time of the last captured observation, or the (future) behaviour after the last observation. In other words, Temporal Inference is about seeking answers to questions regarding unknown segments of the actor's behaviour. Inspecting the actor's behaviour from a sequence of observations is approached as providing an answer to the question posed by a set of alternative *hypotheses*. A question may be related to a single state of the agent or to several states visited by the agent.

We start off by introducing a working example in Section 5.1 that we will use throughout the chapter. Section 5.2 formalizes the concept of hypothesis. The Temporal Inference problem and three special classes are formulated in sections 5.3 and 5.4, respectively, and followed in Section 5.5 with a justification of their solutions. Then, we present our planning-based approach to solve the Temporal Inference problem in Section 5.6 and evaluate it in Section 5.7. We conclude this chapter in Section 5.8 with a discussion of related topics and works.

5.1. Working Example: Driverlog

The working example used in this chapter is based on the *Driverlog* domain introduced at the 3^{rd}*International Planning Competition* (IPC) (Fox & Long, 2003). This is a transportation domain where trucks deliver packages between locations. The actor in this working example refers to a system composed of trucks, drivers and packages, and their interactions with the environment, which is defined by the graph of locations illustrated in Figure 5.1:

- A single truck named `trk`.
- Two drivers named `drv1` and `drv2`.
- Two packages named `pkg1` and `pkg2`.
- A set of 17 locations $Loc = \{\texttt{d1}, \ldots, \texttt{d4}, \texttt{s1}, \ldots, \texttt{s4}, \texttt{t1}, \ldots, \texttt{t3}, \texttt{h1}, \ldots, \texttt{h6}\}$ grouped in four cities such that $\{\texttt{d1}, \ldots, \texttt{d4}\}$ are the locations of the *Diamond city* (`Dc`), $\{\texttt{s1}, \ldots, \texttt{s4}\}$ are the locations of the *Square city* (`Sc`), $\{\texttt{t1}, \ldots, \texttt{t3}\}$ are the locations of the *Triangle city* (`Tc`), and $\{\texttt{h1}, \ldots, \texttt{h6}\}$ are the locations of the *Hexagon city* (`Hc`).

The action model of the actor contemplates a set of actions for drivers and trucks

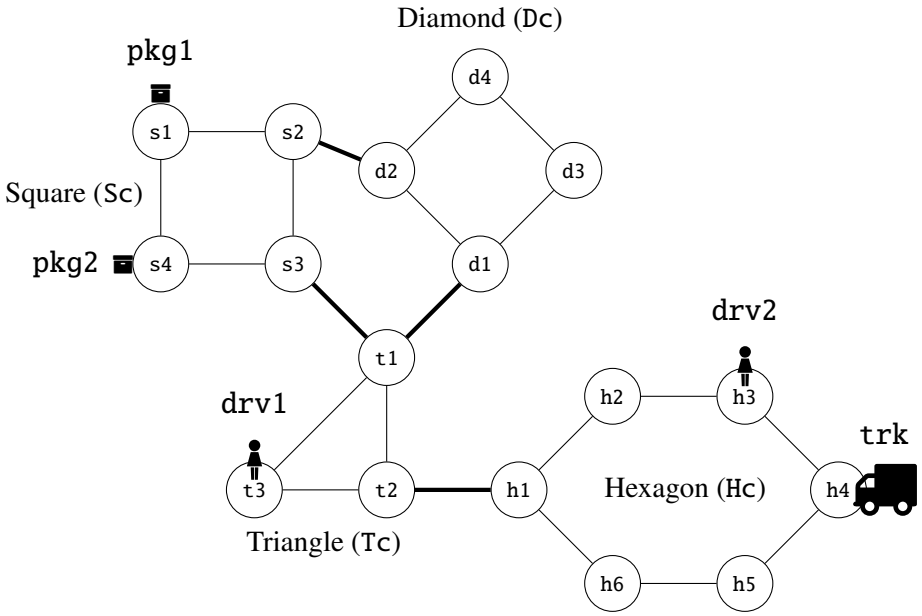

Figure 5.1: Working example based on the *Driverlog* domain.

to move between locations, drivers to board and debark from trucks and packages to be loaded and unloaded from trucks. More specifically:

- A driver can `board` onto a truck when both are in the same location. Alternatively, the driver can `debark` from the truck he is driving.
- A driver boarded in a truck can `drive` the truck between connected locations. Drivers can also `walk` between two locations but only when the locations are within the same city.
- A truck can `load` a package if both are in the same location. Alternatively, the truck can `unload` packages in its current location.

The state of the actor can be described in terms of the locations of the trucks, the drivers and the packages involved. With that regard, the set of state variables of the actor are:

- Variables `loc_pkg1` and `loc_pkg2` with domain $Loc \cup \{\texttt{trk}\}$ that represent the location of the two packages. For instance, $\langle \texttt{loc\_pkg1} = \texttt{s1}, \texttt{loc\_pkg2} = \texttt{s4} \rangle$ means that `pkg1` is at location 1 of Square city (`Sc`) and that `pkg2` is at location 4 of the same city. These variables also indicate whether a package is loaded into the truck, e.g. $\texttt{loc\_pkg1} = \texttt{trk}$.
- Variables `loc_drv1` and `loc_drv2` with domain $Loc \cup \{\texttt{trk}\}$ that represent the location of the two drivers. For instance, $\langle \texttt{loc\_drv1} = \texttt{t3}, \texttt{loc\_drv2} = \texttt{h3} \rangle$ indicates

that drv1 is at location 3 of the Triangle (Tc) city and drv2 is at location 3 of the Hexagon city (Hc). These variables also represent that a driver is inside the truck, e.g. loc_drv1 = trk.

- A variable loc_trk with domain *Loc* representing the location of the truck. For instance, loc_trk = h4 means that trk is at location 4 of Hexagon city.

Besides these variables there are also other variables used to represent the static state knowledge of the environment, the locations graph, which we will skip for clarity. The set of state variables considerd in this working example is thus X = {loc_pkg1, loc_pkg2, loc_drv1, loc_drv2, loc_trk}, which contains the variables that describe the location of the packages, the drivers and the truck of our working example. Figure 5.2 shows the representation of the state depicted in Figure 5.1.

```
loc_pkg1 = s1
loc_pkg2 = s4
loc_drv1 = t3
loc_drv2 = h3
loc_trk = h6
```

Figure 5.2: State representation of the situation shown in Figure 5.1

In the deterministic sensor model used in this working example, observable variables are considered features that abstract the state of the actor. In particular, we assume that the truck is equipped with a low resolution GPS that reports the city where the truck is located. Additionally, cities track down the number of available drivers (those who are not driving), and the number of undelivered packages (e.g. the delivery company has an office at each city for tracking down this information). The exact location of a truck, a package, or a driver is, however, unknown. The sensor model for this example is thus defined with the following set Y of observable variables:

- Variable city_trk with domain D(city_trk) = {Dc, Sc, Tc, Hc} indicates the city where a truck is located; the precise location within the city is, however, unknown. For example, the emission city_trk = Hc means that trk is at some location of the Hexagon city.

- Variables free_Dc, free_Sc, free_Tc and free_Hc represent the number of available drivers in a city,i.e., those who are not already driving a truck. For instance, free_Tc = 1 means that there is one available driver at the Triangular city.

- Variables undel_Dc, undel_Sc, undel_Tc and undel_Hc represent the number of undelivered packages in a city (packages that already loaded into trucks are not counted). For instance, undel_Sc = 2 means that there are two packages pending delivery at the Square city.

Next, we show how the sensing functions must be defined in order to achieve the desired behaviour for our observable variables. The simplest case is the sensing function of variable `city_trk` since its value is only dependent on the state variable `loc_trk`:

$$\begin{aligned}
&\mathsf{sense}_{\texttt{city_trk}}(\texttt{loc_trk}, Loc_{\texttt{Dc}}) = \texttt{Dc}\\
&\mathsf{sense}_{\texttt{city_trk}}(\texttt{loc_trk}, Loc_{\texttt{Sc}}) = \texttt{Sc}\\
&\mathsf{sense}_{\texttt{city_trk}}(\texttt{loc_trk}, Loc_{\texttt{Tc}}) = \texttt{Tc}\\
&\mathsf{sense}_{\texttt{city_trk}}(\texttt{loc_trk}, Loc_{\texttt{Hc}}) = \texttt{Hc}
\end{aligned}$$

where $Loc_c \subseteq Loc$ is the subset of locations of city c. In this way, the sensing function $\mathsf{sense}_{\texttt{city\_trk}}$ will always emit the correct city where the truck is located.

The observable variables for undelivered packages and available drivers count the number of packages or drivers in a city. In both cases, their values depend on two state variables because there are two packages and two drivers. Since their sensing functions are similar we will only use `free_Dc` as an example. Assuming $Loc_{\texttt{Dc}} \subseteq Loc$ is the set of locations of the Diamond city and $\overline{Loc_{\texttt{Dc}}} = Loc \setminus Loc_{\texttt{Dc}}$, the sensing function for `free_Dc` is defined as follows:

$$\begin{aligned}
&\mathsf{sense}_{\texttt{free_Dc}}(\{\texttt{loc_drv1}, \texttt{loc_drv2}\}, Loc_{\texttt{Dc}} \times Loc_{\texttt{Dc}}) = 2\\
&\mathsf{sense}_{\texttt{free_Dc}}(\{\texttt{loc_drv1}, \texttt{loc_drv2}\}, Loc_{\texttt{Dc}} \times \overline{Loc_{\texttt{Dc}}} \cup \overline{Loc_{\texttt{Dc}}} \times Loc_{\texttt{Dc}}) = 1\\
&\mathsf{sense}_{\texttt{free_Dc}}(\{\texttt{loc_drv1}, \texttt{loc_drv2}\}, \overline{Loc_{\texttt{Dc}}} \times \overline{Loc_{\texttt{Dc}}}) = 0
\end{aligned}$$

The first rule says that the sensor will emit `free_Dc` = 2 when both `drv1` and `drv2` are in the Diamond city; the second rule produces the emission `free_Dc` = 1 when either of the drivers is in a location of Diamond city; and the third rule emits `free_Dc` = 0 when none of them are in the city.

```
city_trk = Hs

free_Sc = 0     free_Dc = 0
free_Tc = 1     free_Hc = 1

undel_Sc = 2    undel_Dc = 0
undel_Tc = 0    undel_Hc = 0
```

Figure 5.3: Representation of the observation of the state shown in Figure 5.1

The observation of the state illustrated in Figure 5.1, with the sensor model described above, is shown in Figure 5.3. For clarity, we will often omit observable variables whose value is 0 as follows $o = \langle$`city_trk` = `Hs`, `undel_Sc` = 2, `free_Tc` = 1, `free_Hc` = 1$\rangle$.

The observations generated by the sensor model used in this working example are abstractions that summarize the world state. As a consequence, there may exist multiple states that emit the same observations. For instance, the observation of Figure 5.3 would also be emitted from the state $s = \langle$`loc_pkg1` = `s2`, `loc_pkg2` = `s3`, `loc_drv1` = `h6`, `loc_drv2` = `t1`, `loc_trk` = `h1`$\rangle$ among others.

5.2. Hypotheses

Hypotheses describe assumptions regarding the past, present or future of the actor which are used to pose questions concerning the explicability or predicatability of the actor's behaviour. Moreover, a hypothesis may refer to a single actor state, such as hypothesizing about the goals of the actor, or to several states, for instance, if we wish to know the path followed by the actor or the order in which it will pursue its goals.

First, we need to clarify what is considered as the "past" and "future" of the actor. In our framework, we regard the last observation o_t of an observation sequence $\omega = (o_1, \ldots, o_t)$ as an observation of the present state of the actor as it contains the most recent information. Hence, if we want to find an answer about something that happened before o_t then it means we are looking into the past of the actor. If we ask for something that will happen after o_t then we are speculating about the future.

We start by defining the concept of a *conjecture*, which is used to describe an assumption over a single state of the actor.

Definition 28 (Conjecture)**.** *A conjecture $\tilde{s} : Z \to D(Z)$, $Z \subseteq X$, is a valuation of a subset of state variables.*

A conjecture is similar to the traditional concept of *partial state*, where only some variables have known values and the rest of variables remain undefined. As an example,

$$\tilde{s} = \langle \texttt{loc\_trk} = \texttt{t1}, \texttt{loc\_drv1} = \texttt{trk} \rangle$$

is a conjecture over $Z = \{\texttt{loc\_trk}, \texttt{loc\_drv1}\}$ that specifies that `trk` is being driven by `drv1` and that `trk` is at location `t1` of the Triangle city. Moreover, much like observations, conjectures can also be interpreted as constraints. Specifically, a conjecture $\tilde{s}$ over the subset of state variables $Z = \{x_1, \ldots, x_k\}$, describes a constraint $\varphi_{\tilde{s}} = \langle Z, R \rangle$ where R is a singleton consisting of a tuple $\langle \tilde{s}(x_1), \ldots, \tilde{s}(x_k) \rangle$. In that regard, the conjecture $\tilde{s} = \langle$`loc_trk` = `t1`, `loc_drv1` = `trk`$\rangle$ specifies the constraint

$$\varphi_{\tilde{s}} = \langle \{\texttt{loc\_trk}, \texttt{loc\_drv1}\}, \{\langle \texttt{t1}, \texttt{trk} \rangle\} \rangle$$

A hypothesis combines the collected data, in the form of observations, with conjectures about parts of interest in the actor's trajectory. Depending on how hypotheses are

formulated, particularly in the temporal relation between conjectures and observations, they can be used to indistinctly pose queries about the past (e.g., for finding explanations), present (e.g., for diagnosis and monitoring) or future (e.g., for prediction and goal recognition). A hypothesis then expresses state constraints as well as temporal constraints between observations and conjectures. Hypotheses can only be interpreted in the context of an observation sequence.

Definition 29 (Hypothesis). *A hypothesis is a sequence* $\eta = (h_1, h_2, \ldots, h_m)$ *where*

$$h = \begin{cases} o & o \in D(Y)^Y \\ \tilde{s} & \tilde{s} \in D(Z)^Z \\ (o, \tilde{s}) & o \in D(Y)^Y \wedge \tilde{s} \in D(Z)^Z \end{cases} \tag{5.1}$$

and at least one of the elements of the sequence is a conjecture or a pair observation-conjecture.

This definition is sufficiently expressive to write hypotheses that interleave observations and conjectures in many different ways. Figure 5.4 shows a visual representation of a hypothesis and the interleaving between observations (in blue) and conjectures (in red).

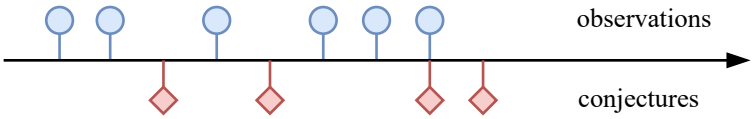

Figure 5.4: A hypothesis interleaving observations (blue) and conjectures (red).

Figure 5.4 illustrates all three cases of Equation (5.1) where observations and conjectures can occur in isolation or coincide in time. Table 5.1 presents the hypothesis of Figure 5.4 in more detail. As we can see in the table, this hypothesis interleaves a sequence of six observations with four conjectures. Considering that the last observation, o_8 in this example, is an observation of the present state of the actor, conjectures $\tilde{s}_3$ and $\tilde{s}_5$ refer to the past of the agent, $\tilde{s}_8$ refers to its present, and $\tilde{s}_9$ to a future state.

Following we present two examples of hypothesis using our *driverlog* working example. Both hypotheses are based on the same observation sequence shown in Table 5.2, which assumes `city_trk` as the only observable variable. The observation sequence of Table 5.2 stands for a piece of information where the GPS has located first `trk` at the Hexagon city (`Hc`), and later at the Diamond city (`Dc`).

Hypothesis example 1. We know by the observation sequence of Table 5.2 that `trk` has visited first `Hc` and `Dc` afterwards. We conjecture that `trk` was driven by `drv2` and that it traversed location `t2` and then `t1` of the Triangle city after visiting `Hc` and before visiting `Dc`. The hypothesis that describes this situation is shown in Table 5.3

Time	Observation	Conjecture
1	o_1	-
2	o_2	-
3	-	$\tilde{s}_3$
4	o_4	-
5	-	$\tilde{s}_5$
6	o_6	-
7	o_7	-
8	o_8	$\tilde{s}_8$
9	-	$\tilde{s}_9$

Table 5.1: Hypothesis illustrated in Figure 5.4

Time	Observation
1	`city_trk = Hc`
2	`city_trk = Dc`

Table 5.2: Example of a 2-observation sequence in the driverlog working example that assumes a single observable variable `city_trk`.

Hypothesis example 2. In this hypothesis we make the conjecture that, at some point after observing `trk1` in the Diamond city, `pkg1` will be delivered at location `d4`. Table 5.4 shows the hypothesis of this example.

5.3. The Temporal Inference problem

Temporal Inference is about seeking answers to questions regarding unknown segments of the actor's behaviour. The observer, having collected some observations on the actor, poses a number of alternative hypotheses each embodying a different answer to the question at hand. A Temporal Inference Problem (TIP) extends the observation decoding problem with a set of hypotheses, with the goal of finding the hypothesis that best fits the observed behaviour of the acting agent. Following, we formally define the Temporal inference problem.

Definition 30 (Temporal inference problem)**.** *We formally define a Temporal inference problem as a tuple* $\langle M_a, M_s, s_0, \omega, \mathcal{H} \rangle$ *where:*

- $M_a = \langle X, A, \mathsf{App}, \mathsf{Eff}, c \rangle$ *is the action model of the actor.*
- $M_s = \langle X, Y, \mathsf{Sense} \rangle$ *is the sensor model of the observer.*
- s_0 *is the initial situation of the actor.*
- $\omega = (o_1, \ldots, o_t)$ *is an observation sequence.*

Time	Observation	Conjecture
1	city_trk = Hc	-
2	-	loc_trk = t2, loc_drv2 = trk
3	-	loc_trk = t1, loc_drv2 = trk
4	city_trk = Dc	-

Table 5.3: Hypothesis of Example 1.

Time	Observation	Conjecture
1	city_trk = Hc	-
2	city_trk = Dc	-
3	-	loc_pkg1 = d4

Table 5.4: Hypothesis of Example 2.

- $\mathcal{H}$ *is a finite and non-empty set of hypotheses over* ω.

The solution to the Temporal Inference problem is the *most likely hypothesis*, which we define as the hypothesis satisfied by the lowest cost trajectory. Since a hypothesis $\eta \in \mathcal{H}$ contains the observation sequence ω alongside some conjectures, a trajectory that satisfies η can be interpreted as an explanation for ω that complies with the conjectures in η. In light of this, we can also interpret the most likely hypothesis as the one supported by the best explanation.

Formally, we say that a state trajectory $\tau = (s_0, \ldots, s_n)$ satisfies a hypothesis $\eta = (h_1, \ldots, h_m)$ when there exists an alignment between τ and η such that all conjectures and observations of η are accepted by states of τ.

Definition 31. *A state trajectory* $\tau = (s_0, \ldots, s_n)$ *satisfies a hypothesis* $\eta = (h_1, \ldots, h_m)$ *under alignment* α *if, for* $i \in \{1, \ldots, m\}$*, it holds that* $s_{\alpha(i)} \models \Phi_{h_i}$ *where*

$$\Phi_{h_i} = \begin{cases} \Phi_{o_i} & \text{when } h_i = o_i \\ \{\varphi_{\tilde{s}_i}\} & \text{when } h_i = \tilde{s}_i \\ \Phi_{o_i} \cup \{\varphi_{\tilde{s}_i}\} & \text{when } h_i = (o_i, \tilde{s}_i) \end{cases} \tag{5.2}$$

As we can see, Definition 31 simply extends the definition of explanation (Def. 24) to account for the constraints imposed by the conjectures. Later, when we present our proposal to solve the TIP with planning, we will see that interpreting conjectures as constraints will allow us to use the same ideas presented in the previous chapter.

Now we are ready to define the solution to a TIP as the most likely hypothesis.

Definition 32 (Most likely hypothesis)**.** *The solution to a Temporal Inference problem* $\langle M_a, M_s, s_0, \omega, \mathcal{H} \rangle$ *is the most likely hypothesis* $\eta^* \in H$ *computed as*

$$\eta^* = \arg\min_{\eta \in \mathcal{H}} \left\{ \min_{\tau \in \mathcal{T}_\eta(M_a, s_0)} synCost(\tau) \right\} \tag{5.3}$$

where $\mathcal{T}_\eta(M_a, s_0)$ is the set of trajectories that satisfy the hypothesis η.

This definition follows the principles of Inference to the Best Explanation (Harman, 1965; Schupbach, 2017). Inference to the Best Explanation is a form of inference that reasons to a hypothesis based upon the premise that it provides a better explanation than any other available, competing hypothesis. Hence, one reaches the conclusion that a hypothesis is true from the premise that it provides a better explanation for the evidence than would any other hypothesis.

5.4. Special Classes of Temporal Inference Problems

A wide palette of inference problems can be expressed as particular instances of a TIP. This section describes three special classes of Temporal inference problems in which we will put the focus for the remainder of the chapter. The three selected classes of TIP are based on the crisp semantics of problems that refer to the present, the past or the future of the actor.

Monitoring. The goal of monitoring is to ascertain the present state of the actor. In a monitoring problem the conjectures refer exclusively to the current state of the actor, that is, to the state associated to the last observation. Hypotheses in a monitoring TIP only contain a single conjecture that corresponds in time with the last observation. Formally, a set of hypotheses $\mathcal{H}$ represents a monitoring problem if, for every single hypothesis $\eta = (h_1, h_2, \ldots, h_m) \in \mathcal{H}$, it holds that

- $\forall i, 1 \leq i < m, h_i \in D(Y)^Y$, and
- $h_m \in D(Y)^Y \times D(Z)^Z$

Hindsight. Hindsight is about looking into the past of the acting agent. A hindsight problem poses a question about unknown segments of the actor's behaviour so far and, therefore, all conjectures refer to past states. Formally, a set of hypotheses $\mathcal{H}$ represents a hindsight problem if, for every single hypothesis $\eta = (h_1, h_2, \ldots, h_m) \in \mathcal{H}$, it holds that

- $\exists i, i < m$, such that $h_i \in D(Z)^Z$ or $h_i \in D(Y)^Y \times D(Z)^Z$, and
- $\forall j, i < j \leq m, h_m \in D(Y)^Y$

Prediction. Prediction is concerned with the future of the actor. A prediction problem speculates about the actor's future by making conjectures about its future states. Formally, a set of hypotheses $\mathcal{H}$ represents a prediction problem if, for every single hypothesis $\eta = (h_1, h_2, \ldots, h_m) \in \mathcal{H}$, it holds that

- $\exists j, j < m$ such that $\forall i, 1 \leq i \leq j, h_i \in D(Y)^Y$, and

- $\forall i, j < i \leq m, h_i \in D(Z)^Z$

Figure 5.5 provides a graphical guide on how hypotheses must be defined in each class of problems. Summarizing, in monitoring we have a single conjecture about the present state of the actor, in hindsight all conjectures are about past states, and in prediction all conjectures are about future states.

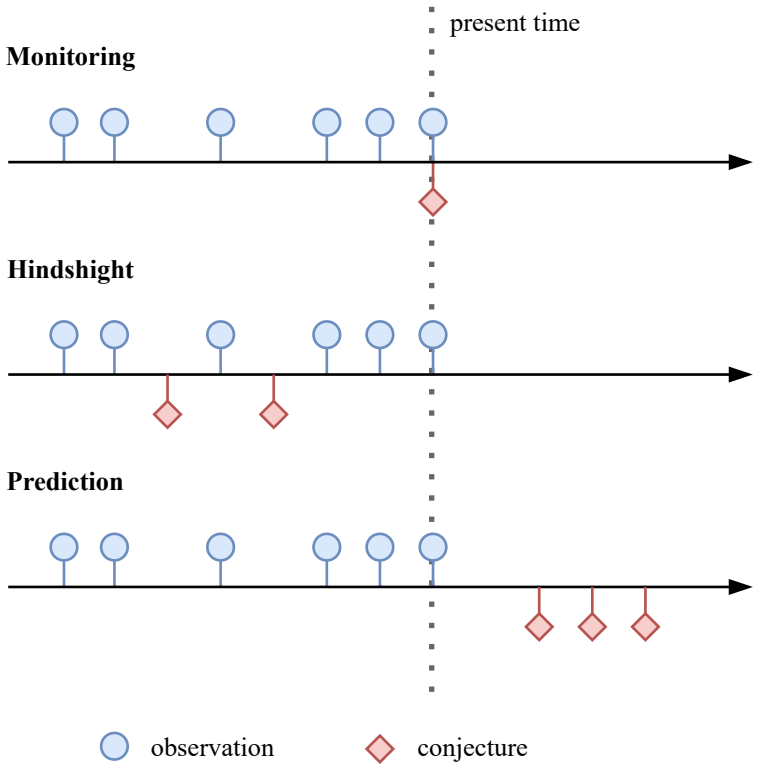

Figure 5.5: Special classes of Temporal inference problems

Following, we illustrate the three types of Temporal inference problems using the *driverlog* working example. In all examples, we assume that the hypothesis are defined over the two-observation sequence shown in Table 5.5. This sequence shows that we first observed the two packages in the Square city, one free driver in the Triangle city and `trk` in the Hexagon city; afterwards we observed one less package in the Square city and `trk` in the Diamond city.

Time	Observation
1	`undel_Sc=2, free_Tc=1, city_trk = Hc`
2	`undel_Sc=1, free_Tc=1, city_trk = Dc`

Table 5.5: Example of a two-observation sequence in the driverlog working example.

Monitoring example. An example of monitoring is identifying the driver who is currently driving `trk`, for which we generate two alternative hypotheses $\mathcal{H} = \{\eta, \eta'\}$, one where `trk` is driven by `drv1` and another where it is driven by `drv2`. These two hy-

potheses are shown in Table 5.6, where o_1 and o_2 refer to the two observations previously described.

	Time	Observation	Conjecture
η	1	o_1	-
	2	o_2	`loc_drv1 = trk`
η'	1	o_1	-
	2	o_2	`loc_drv2 = trk`

Table 5.6: Example of hypotheses in a monitoring TIP.

Hindsight example. The observation sequence of Table 5.5 used for these examples show that a package is picked up between the first and the second observation. In light of this, we can formulate a hindsight problem in which we ask about the package which was picked up. To this end we create two alternative hypotheses $\mathcal{H} = \{\eta, \eta'\}$ (Table 5.7), one with `pkg1` in `trk` and another with `pkg2`. Note that inserting the conjecture between the two observations pushes the second observation to time index 3.

	Time	Observation	Conjecture
η	1	o_1	-
	2	-	`loc_pkg1 = trk`
	3	o_3	-
η'	1	o_1	-
	2	-	`loc_pkg2 = trk`
	3	o_3	-

Table 5.7: Example of hypotheses in a hindsight TIP.

Prediction example. For the prediction example, let us assume that the delivery point of the two packages is location `d1` of the Diamond city. An example of a prediction problem is speculating about the order in which packages `pkg1` and `pkg2` will be delivered after having observed that one package was already picked up and that the truck traveled to the Diamond city. This problem is formulated with the two hypotheses shown in Table 5.8 where η conjectures that `pkg1` will be delivered before `pkg2` and η' conjectures the reverse order. In order to speculate about the order in which goals are achieved, the hypotheses must include a sequence of conjectures such that goals not satisfied in previous conjectures are satisfied in later ones. For this particular example, in both hypotheses the conjecture $\tilde{s}_3$ says that one package has already been delivered to location `d1` while the other is still in its initial location (`s4` for `pkg2` and `s1` for `pkg1`), while conjecture $\tilde{s}_4$ shows both packages at the delivery point.

This prediction example shows that writing hypotheses that contain multiple conjectures enriches the expressiveness of the Temporal inference problem as it enables expressing properties concerning several states of the actor like, for instance, the order in which

	Time	Observation	Conjecture
η	1	o_1	-
	2	o_2	-
	3	-	`loc_pkg1 = d1, loc_pkg2 = s4`
	4	-	`loc_pkg1 = d1, loc_pkg2 = d1`
η'	1	o_1	-
	2	o_2	-
	3	-	`loc_pkg1 = s1, loc_pkg2 = d1`
	4	-	`loc_pkg1 = d1, loc_pkg2 = d1`

Table 5.8: Example of hypotheses in a prediction TIP.

goals will be satisfied. The same is applicable to the hindsight problem to query about the specific order of two past occurrences.

5.5. The Solution to the Temporal Inference Problem

This section explains how we arrive at the solution to the Temporal Inference problem shown in equation 5.3. The goal of Temporal Inference is to identify the hypothesis that better adjusts the evidence provided by the observations, all while complying with the models of the actor and the observer. Equation (5.3) of Section 3.2.2 (replicated below for convenience) provides a probabilistic interpretation of the exact solution to this problem.

$$\arg\max_{\eta \in \mathcal{H}} P(\eta|\omega, s_0, M_a, M_s) \tag{5.4}$$

where $P(\eta|\omega, s_0, M_a, M_s)$ is the likelihood that the actor behaves as described in η given its action model M_a and the sequence ω observed with the sensor model M_s. From the definition of conditional probability we arrive at the following equality

$$\arg\max_{\eta \in \mathcal{H}} P(\eta|\omega, s_0, M_a, M_s) = \arg\max_{\eta \in \mathcal{H}} P(\eta, \omega|s_0, M_a, M_s) \tag{5.5}$$

where $P(\eta, \omega|s_0, M_a, M_s)$ is the joint likelihood of η and ω happening at the same time. Since the likelihood $P(\omega|s_0, M_a, M_s)$ can be computed by marginalizing over all trajectories, a natural way to define $P(\eta, \omega|s_0, M_a, M_s)$ is to marginalize only over trajectories that satisfy η, that is, the set $\mathcal{T}_\eta(M_a, s_0)$. Consequently, we define this joint likelihood as

$$P(\eta, \omega|s_0, M_a, M_s) = \sum_{\tau \in \mathcal{T}_\eta(M_a, s_0)} P(\omega, \tau|s_0, M_a, M_s) \tag{5.6}$$

where $P(\omega, \tau|s_0, M_a, M_s)$ is the joint likelihood of the observation sequence and the trajectory that we are already familiar with. The computation of equation 5.6 is still intractable as the inference setting is unbounded and, therefore, the set $\mathcal{T}_\eta(M_a, s_0)$ is potentially infinite. We articulate the following common approximation that assumes that the sum is dominated by its largest term:

$$P(\eta, \omega|s_0, M_a, M_s) \approx \max_{\tau \in \mathcal{T}_\eta(M_a, s_0)} P(\omega, \tau|s_0, M_a, M_s) \tag{5.7}$$

Ultimately, this approximation leads us to the following solution to the Temporal Inference problem, which defines the most likely hypothesis as the one supported by the best explanation.

$$\arg\max_{\eta \in \mathcal{H}} P(\eta|\omega, s_0, M_a, M_s) \approx \arg\max_{\eta \in \mathcal{H}} \max_{\tau \in \mathcal{T}_\eta(M_a, s_0)} P(\omega, \tau|s_0, M_a, M_s) \tag{5.8}$$

As we can see, Equation (5.8) is the probabilistic interpretation of our cost-based solution presented in Equation (5.3).

5.6. Temporal Inference as Planning

This section presents our proposal to address the TIP using planning. From Definition 32, the most likely hypothesis is the one satisfied by the lowest cost trajectory. This means that, for each hypothesis $\eta \in \mathcal{H}$, we need to compute the lowest cost trajectory in the set $\mathcal{T}_\eta(M_a, s_0)$. This problem is closely related to an observation decoding problem with the exception that:

1. a trajectory needs to satisfy a hypothesis that now includes not only observations but both observations and conjectures
2. the calculation needs to be done for every hypothesis

Our proposal addresses these differences in the following way:

1. We extend our observation decoding as planning approach in order to find trajectories that satisfy a hypothesis. This implies compiling a planning problem $\mathcal{P}(\eta)$ such that its solutions generate trajectories $\mathcal{T}_\eta(M_a, s_0)$.
2. We compile a TIP $\langle M_a, M_s, s_0, \omega, \mathcal{H} \rangle$ into $|\mathcal{H}|$ planning problems $\mathcal{P}(\eta)$, one for each hypothesis in $\mathcal{H}$. After solving each problem $\mathcal{P}(\eta)$ optimally, the solutions are ranked by cost to determine the most likely hypothesis.

We leverage the ordered occurrence of the components of a hypothesis to extend our observation decoding as planning approach to solve a TIP. More specifically, a hypothesis $\eta = (h_1, h_2, \ldots, h_m)$ expresses an ordered occurrence property $\mathsf{ord}(h) = \mathsf{XF}(\Phi_{h_1} \wedge \mathsf{XF}(\Phi_{h_2} \wedge \ldots \wedge \mathsf{XF}\Phi_{h_m}))$. Thus, the same ideas used for finding explanations to observation sequences can be exploited to finding trajectories that satisfy a hypothesis. In this case, we translate the property $\mathsf{ord}(\eta)$ to its corresponding monitor automaton $\mathcal{A}(\eta)$ and build a classical planning problem $\mathcal{P}(\eta)$ following the compilation proposed in Section 4.4.3.

Regarding the implementation of the sensing actions, we note that the constraint defined by a conjecture always has cardinality 1 and, therefore, meets the criteria for the basic encoding of the sensing actions. Recall from Section 4.4.3 that constraints with cardinality 1 can be directly encoded in the applicability constraint of a sensing action, while for higher cardinality constraints (as is generally the case for observations) we recommend the encoding with dead-ends. The same rules applies here for a sensing action a^{h_i} encoding the transition $q_i = \delta(q_{i-1}, \Phi_{h_i})$ of the monitor automaton $\mathcal{A}(\eta)$. Consequently, assuming $\Phi_{h_i} = \Phi_{o_i} \cup \varphi_{\tilde{s}_i}$, the constraints Φ_{o_i} defined by the observation will be encoded following the criteria specified in Section 4.4.3, and the constraint $\varphi_{\tilde{s}_i}$ defined by the conjecture will be encoded following the basic encoding.

5.7. Experimental evaluation

This section evaluates the ability of an off-the-shelf planner to solve the palette of Temporal inference problems presented in this work. We tested our approach on a benchmark of **monitoring**, **hindsight**, and **prediction** problems, in five well-known classical planning domains from the IPC: *grid*, *miconic*, *driverlog*, *floortile*, and *openstacks*.

A Temporal inference problem $\langle M_a, M_s, s_0, \omega, \mathcal{H} \rangle$ is constructed as follows:

- We use the action model M_a of the original domain as available in http://api.planning.domains.
- The sensor models are defined in the same way as the one used in our *driverlog* working example. This means that the values of the observable variables Y are high-level observations of the world states that convey how many objects meet some particular property, but not the actual identity of the objects. For instance, in the *miconic* domain, we have "the number of passengers inside a lift"; in the *openstacks* domain, we have "the number of waiting/started/shipped orders". In both cases, the actual identity of the passengers/orders remains unknown to the observer.
- $\mathcal{H}$ is a set of hypotheses regarding the current state in monitoring TIPs, a sequence of past states in hindsight TIPs, and a sequence of future states in prediction TIPs. Hypotheses allow us to answer queries such as "which are the currently opened locks?" (*grid*), "what was the order of the first four painted tiles?" (*floortile*), and

Domain	O%	Monitoring				Hindsight				Prediction			
		$\|\mathcal{H}\|$	$\|\mathcal{H}^*\|$	Q	T	$\|\mathcal{H}\|$	$\|\mathcal{H}^*\|$	Q	T	$\|\mathcal{H}\|$	$\|\mathcal{H}^*\|$	Q	T
Grid	30	5.40	1.00	1.00	7.97	2.00	1.00	1.00	135.52	2.00	1.00	1.00	32.17
	50		1.00	1.00	27.85		1.00	1.00	146.05		1.00	1.00	13.73
	70		1.00	1.00	65.78		1.00	1.00	138.33		1.00	1.00	17.79
Miconic	30	6.00	1.10	1.00	53.22	5.00	1.90	0.90	149.04	5.00	3.00	1.00	140.57
	50		1.10	1.00	180.29		2.00	1.00	140.24		2.60	1.00	255.56
	70		1.20	1.00	120.13		1.80	1.00	121.02		2.60	1.00	320.65
Driverlog	30	5.40	1.40	0.90	162.61	5.30	1.00	1.00	238.16	5.30	1.00	1.00	169.24
	50		1.40	0.90	284.90		1.00	1.00	399.23		1.00	1.00	250.70
	70		1.40	0.90	390.91		1.00	1.00	560.77		1.00	1.00	349.16
Floortile	30	5.90	1.30	0.40	37.31	6.00	4.70	0.80	443.10	6.00	3.00	1.00	676.76
	50		1.30	0.50	56.97		4.00	0.80	141.56		3.80	0.90	713.32
	70		1.50	0.60	44.53		3.50	0.80	128.48		4.70	1.00	734.02
Openstacks	30	5.60	2.30	0.80	5.93	5.80	2.00	1.00	11.19	5.90	2.50	1.00	15.70
	50		2.30	0.80	6.81		1.70	1.00	11.32		2.50	1.00	20.30
	70		2.30	0.80	9.60		1.60	1.00	14.28		2.50	1.00	18.77

Table 5.9: Results for Temporal inference problems of *monitoring*, *hindsight*, and *prediction* in five different domains and for three different levels of observability.

"what will be the order in which the last three packages will be delivered?" (*driverlog*). For each TIP, we created a set $\mathcal{H}$ in which only one of the hypothesis is correct with respect to the original trajectory of the actor. Observations were collected by observing some trajectory τ of the agent with the sensor model. We also define an observability parameter that determines the intermittency of the sensing process by specifying the percentage of states of τ that emit an observation.

All experiments were run on an Intel Core i5 3.10GHz x 4 16GB of RAM. The planning problems were optimally solved with the FAST-DOWNWARD planning system, following the configuration for the *merge and shrink* heuristic (Helmert et al., 2014) that uses bisimulation based shrinking, the merge strategy SCC-DFP, and the appropriate label reduction setting (Sievers et al., 2016), as recommended at the http://www.fast-downward.org/ website. Planning problems are solved with a timeout value of 120 secs of CPU-time, and 8GB of memory. The source code and test benchmark for the evaluation is available at https://github.com/anonsubs/planning_inference.

5.7.1. Empirical Results

Table 5.9 summarizes the results of evaluating our approach on monitoring, hindsight and prediction TIPs. Column "O%" is the observability parameter (30%, 50%, or 70%). For each type of TIP, we report the size of the hypotheses set ($|\mathcal{H}|$), the number of most likely hypotheses as computed by our approach ($|\mathcal{H}^*|$), the ratio of TIPs whose correct hypothesis is among $\mathcal{H}^*$ (Q), and the accumulated time (T) in secs to solve the TIP. We note that solving a TIP implies solving $|\mathcal{H}|$ planning instances $P(\eta)$, one instance for each η in $\mathcal{H}$.

The best score is obtained when $|\mathcal{H}^*| = 1$ and $Q = 1$, meaning that only one hypoth-

esis is recognized as the most likely and that such hypothesis is the correct one. For each combination of TIP (monitoring, hindsight and prediction), domain and observability percentage, we report in Table 5.9 the average score across 10 problems. Thus, our approach was evaluated over $5 \times 3 \times 3 \times 10 = 450$ TIPs.

As expected, the best results, i.e. larger values for Q and/or lower values for $|\mathcal{H}^*|$, are obtained for high degrees of observability since more data is available. Yet, we also find quite a few cases of 30% observability that achieve the top performance. Increasing observability comes however at the cost of longer planning solutions (more sensing actions), which require longer computation times. The prediction results for the *floortile* in Table 5.9 show that increasing the observability may actually hinder the performance of the system as computation times are pushed over the set timeout of 120s.

Interestingly, we can observe in Table 5.9 that the worst quality results are found in monitoring TIPs and to a lesser extent also in hindsight TIPs. This is particularly remarkable in the *floortile* domain which features highly interconnected goals. In general, a drop in the quality of the solutions is expected when the observation sequence only conveys a small subset of the final goals that the actor pursues. This is because our approach does not assume that the goals of the actor are known so we find explanations that are optimal with respect to what is revealed by the observation sequence. However, the prefix of an optimal plan is not necessarily optimal with respect to the subset of goals achieved by that prefix. This is what is causing the low performance in the floortile domain, since the goal of the actor is to have the whole board painted but the last observation only shows that a handful of tiles are painted. As a result, this mismatch between the information conveyed by the observations and the goals pursued by actor causes a drop of the quality. This limitation can easily be overcome if the actor's goals are known, in which case they will be added as the last conjecture of the hypothesis.

Summarizing the results of our evaluation, we have shown that our approach offers an homogeneous and versatile solution to the problem of uncovering the past, present and future of an actor. Our approach is generally accurate and unambiguous in selecting the most likely hypothesis. The exception to this is found in domains with highly interconnected goals, a limitation that is primarily caused by the goals being unknown in our framework.

5.8. Discussion

The Temporal Inference problem blurs the line between *explicability* and *predictability*, cementing the view that inference oriented towards explaining the past or to predicting the future are two sides of the same coin and as such, they can be formulated within a single unified framework (Chakraborti et al., 2019). In doing so, our approach generalizes a number of inference problems found in the literature like Diagnosis (Sohrabi et al., 2010), Plan Monitoring (Fritz & McIlraith, 2007) and Goal Recognition (Ramírez & Geffner,

2010).

Another relevant aspect is that existing formulations of inference problems typically account for hypotheses on the unobserved behaviour at a single point of the actor's trajectory. Assumptions regarding the initial state are included in approaches to the Diagnosis problem (Sohrabi et al., 2010), estimations about the actual state are considered in Plan Monitoring (Fritz & McIlraith, 2007), and the set of hypotheses in Goal Recognition contains goal states. The ability to hypothesize that two variables are achieved in some particular order or conjecturing about whether a variable is achieved before or after some observation is a powerful inference ability.

The trade-off between information and scalability that we can find in existing approaches to inference is also an important topic of discussion. On the more informative but less scalable side, we have (Ramírez & Geffner, 2010) that solves two planning problems for each hypothesis. One problem is used to compute the most likely explanation given the observations and the other to compute a "null hypothesis" (a solution inconsistent with the observations) to determine the degree in which the explanation is necessary for a given goal. Then, if the cost of the null hypothesis is much larger than the cost of the most likely explanation, they can conclude that the explanation is not only sufficient but necessary. Our Temporal Inference problem makes no assumption regarding the goals of the actor so we cannot follow this counter-factual approach. Instead, we follow the principles of Inference to the Best Explanation that requires a single planning problem per hypothesis and allows for a more general formulation. On the opposite side of the balance we find the heuristic-based approach (E-Martín et al., 2015; Pereira et al., 2017, 2020; Vered et al., 2018) that use heuristics as fast but less informed estimates to predict the goal of the actor.

6. MODEL RECOGNITION

In this chapter we present the *Model Recognition* problem, a problem that shifts the focus to the action model of the actor. In contrast with observation decoding and temporal inference, in Model Recognition it is assumed that the action model of the actor is unknown and, instead, we are presented with a finite set of candidate models from which to pick the one that best fits the observed behaviour of the actor.

As usual, we start the chapter by presenting a working example in Section 6.1. The Model Recognition problem is formalized in Section 6.2 and its solution motivated in Section 6.3. While we provide a formalization of Model Recognition that is general to any action model, we restrict our attention to the specific case of action models encoded by STRIPS schematic representations to which we dedicate Section 6.4. In Section 6.5 we present our approach to solve the Model Recognition problem, an approach that is later experimentally evaluated in Section 6.6. Finally, we conclude the chapter with a discussion (Section 6.7) of related topics.

6.1. Working Example: Navigation Strategies

The Model Recognition problem, as well as the learning problem that we will see in the next chapter, are concerned with the action model of the actor. Therefore, hereinafter we will put the focus of the working example on the action model and we will assume a fairly simple sensor model for the observer. This working example includes two different action models that serve as the set of candidate action models. In particular, the two action models used here describe different strategies to visit all the tiles of the 6×6 grid. In both models the actor can choose to move in any of the four directions (up, down, left, right) but the applicability of some actions is partly restricted by a controller.

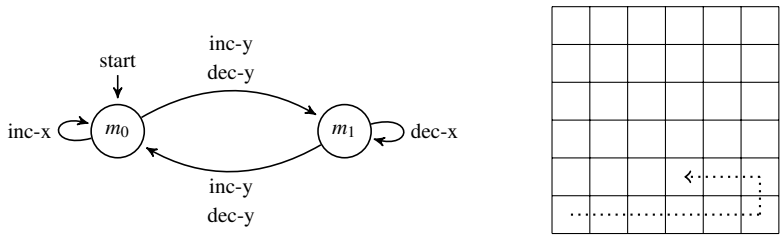

Figure 6.1: Example of navigation strategy. (Left) Automata to control that the actor is allowed to increments its x-coordinate in mode m_0 and decrease it in mode m_1. (Right) Actor navigating a 6×6 grid.

Figure 6.1 shows one of the navigation strategies considered in this working example as well as the automaton that acts as its controller. In this example, the controller has two

modes, m_0 and m_1 (m_0 is the initial mode), and the navigation strategy it implements can be described by the following three rules:

- The actor can move right (inc-x) but not left (dec-x) when in mode m_0.
- The actor can move left (dec-x) but not right (inc-x) when in mode m_1.
- The actor can move up (inc-y) or down (dec-y) at any time but by doing so it switches the controller mode.

This navigation strategy contemplates several different ways to visit all the tiles of the grid, including the horizontal zig-zag pattern shown in Figure 6.1. Our second navigation strategy, illustrated in Figure 6.2, is very similar to the first one but swaps the controller modes for the left and right moves of the actor.

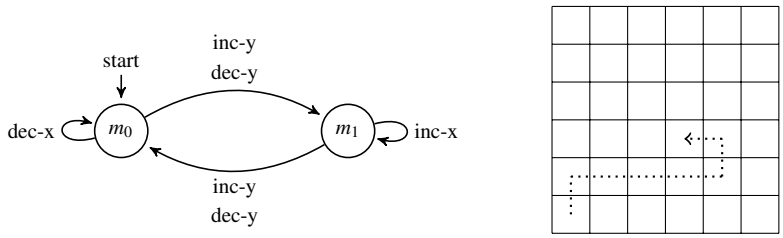

Figure 6.2: Another example of navigation strategy. (Left) Automata to control that the actor is allowed to decrease its x-coordinate in mode m_0 and increment it in mode m_1. (Right) Actor navigating a 6×6 grid.

The two action models that make up the candidates set of this working example are M_a^R and M_a^L, corresponding to the navigation strategies of Figure 6.1 and Figure 6.2, respectively. We use the superscript "R" to identify the navigation strategy that allows the actor to move right when in mode m_0, and similarly, the superscript "L" to identify the one that allows to move left when in mode m_0. In both action models, the set of state variables X consists of two coordinate variables `xcoord` and `ycoord` with domain $\{1, \ldots, 6\}$ and a controller mode variable `mode` with domain $\{0, 1\}$. The set of actions A is the same in both cases, with A consisting of 8 actions, one for each combination of direction and mode. The two action models differ in the applicability and effects constraints that determine when an action can be executed and its outcome. In summary, both action models use the same set of variables to describe the state of the actor and have the same actions available, but the behaviour of the actions may be different. Tables 6.1 and 6.2 specify the applicability (App) and effects (Eff) constraints of both action models. In both models, moving up and down can be done freely as long as it is within the bounds of the grid, and doing so switches the controller mode. The M_a^R can move right if it is in mode m_0 and left if it is in mode m_1, and vice-versa for the M_a^L action model.

Regarding the sensor model of the observer, in this working example we assume that the observer can correctly locate the actor through the observable variables xobs and yobs

Action	App	Eff
m0-inc-x	"mode = 0 ∧ xcoord < 6"	"$\texttt{xcoord}^+ = \texttt{xcoord} + 1$"
m0-dec-x	"mode = 0 ∧ xcoord > 1"	-
m0-inc-y	"mode = 0 ∧ ycoord < 6"	"$\texttt{mode}^+ = 1 \wedge \texttt{ycoord}^+ = \texttt{ycoord} + 1$"
m0-dec-y	"mode = 0 ∧ ycoord > 1"	"$\texttt{mode}^+ = 1 \wedge \texttt{ycoord}^+ = \texttt{ycoord} - 1$"
m1-inc-x	"mode = 1 ∧ xcoord < 6"	-
m1-dec-x	"mode = 1 ∧ xcoord > 1"	"$\texttt{xcoord}^+ = \texttt{xcoord} - 1$"
m1-inc-y	"mode = 1 ∧ ycoord < 6"	"$\texttt{mode}^+ = 0 \wedge \texttt{ycoord}^+ = \texttt{ycoord} + 1$"
m1-dec-y	"mode = 1 ∧ ycoord > 1"	"$\texttt{mode}^+ = 0 \wedge \texttt{ycoord}^+ = \texttt{ycoord} - 1$"

Table 6.1: Applicability (App) and effects (Eff) constraints of the M_a^R action model.

Action	App	Eff
m0-inc-x	"mode = 0 ∧ xcoord < 6"	-
m0-dec-x	"mode = 0 ∧ xcoord > 1"	"$\texttt{xcoord}^+ = \texttt{xcoord} - 1$"
m0-inc-y	"mode = 0 ∧ ycoord < 6"	"$\texttt{mode}^+ = 1 \wedge \texttt{ycoord}^+ = \texttt{ycoord} + 1$"
m0-dec-y	"mode = 0 ∧ ycoord > 1"	"$\texttt{mode}^+ = 1 \wedge \texttt{ycoord}^+ = \texttt{ycoord} - 1$"
m1-inc-x	"mode = 1 ∧ xcoord < 6"	"$\texttt{xcoord}^+ = \texttt{xcoord} + 1$"
m1-dec-x	"mode = 1 ∧ xcoord > 1"	-
m1-inc-y	"mode = 1 ∧ ycoord < 6"	"$\texttt{mode}^+ = 0 \wedge \texttt{ycoord}^+ = \texttt{ycoord} + 1$"
m1-dec-y	"mode = 1 ∧ ycoord > 1"	"$\texttt{mode}^+ = 0 \wedge \texttt{ycoord}^+ = \texttt{ycoord} - 1$"

Table 6.2: Applicability (App) and effects (Eff) constraints of the M_a^L action model.

that always take on the values of their counterparts xcoord and ycoord. The controller mode that dictates the navigation strategy is however completely hidden to the observer, as there is no observable variable that depends on the state variable mode.

6.2. The Model Recognition Problem

Model recognition is the problem of identifying the action model of the actor among a set of candidate models on the basis of an observation sequence. We require that all the candidate action models share the same state variables and actions. The motivation behind this requirement is to ensure that all candidate models are consistent with the sensor model so that a priori they can generate explanations for the observation sequence. We say that two models that share the same state variables and actions are *comparable*.

Definition 33 (Comparable action models)**.** *Two action models* $M_a = \langle X, A, \mathsf{App}, \mathsf{Eff}, c\rangle$ *and* $M'_a = \langle X', A', \mathsf{App}', \mathsf{Eff}', c'\rangle$ *are* **comparable** *iff both share the same state variables and actions, i.e,* $X = X'$ *and* $A = A'$*.*

We will use the term *space* to refer to the set of all action models defined over the same set of state variables. An important observation to be made is that, as the state

variables and actions are the same for all models within the space, all models in the space are comparable to one another.

Definition 34 (Space of action models)**.** *The space of action models $\mathcal{M}(X, A)$ is the set of all models defined over state variables X and actions A.*

Proposition 1. *All action models belonging to the same space are comparable to one another.*

Comparable models can differ in the App and Eff constraints and in the cost function c. This leads to different scenarios for two given comparable models $M_a = \langle X, A, \mathsf{App}, \mathsf{Eff}, c\rangle$ and $M'_a = \langle X, A, \mathsf{App}', \mathsf{Eff}', c'\rangle$:

- If $\mathsf{App} = \mathsf{App}'$ and $\mathsf{Eff} = \mathsf{Eff}'$ but $c \neq c'$, then the sets of trajectories $\mathcal{T}(M_a, s_0)$ and $\mathcal{T}(M'_a, s_0)$ are the same but the cost of the trajectories is different, which may lead to different preferred explanations for each model.
- If $\mathsf{App} \neq \mathsf{App}'$ or $\mathsf{Eff} \neq \mathsf{Eff}'$, then the set of trajectories $\mathcal{T}(M_a, s_0)$ and $\mathcal{T}(M'_a, s_0)$ are different and, therefore, the explanations each model can generate are also different.

This latter case is exactly what happens in our navigation policies working example, where M_a^R and M_a^L are comparable action models that differ only in the effects constraints. We will see later how this difference results in M_a^R and M_a^L generating different explanations for the same observation sequence.

A Model Recognition problem takes as input the initial situation and a sequence of observations of the actor, the sensor model used to capture them, and a set of candidate action models.

Definition 35 (Model Recognition Problem)**.** *A* Model Recognition *problem is a tuple $\langle \mathbb{M}_a, M_s, s_0, \omega\rangle$ where:*

- $\mathbb{M}_a$ *is a set of candidate comparable action models.*
- $M_s = \langle X, Y, \mathsf{Sense}\rangle$ *is the sensor model of the observer.*
- s_0 *is the known initial situation of the actor.*
- $\omega = (o_1, \ldots, o_t)$ *is an observation sequence.*

In Model Recognition the goal is to find the model among the set of candidates that best fits the observed behaviour. More formally, the solution to the Model Recognition problem is the most likely action model $M_a^* \in \mathbb{M}_a$, that is, the action model in the candidates set that best explains the observation sequence ω.

Definition 36 (Most likely action model)**.** *The solution to a Model Recognition problem* $\langle \mathbb{M}_a, M_s, s_0, \omega \rangle$ *is the most likely action model* $M_a^* \in \mathbb{M}_a$

$$M_a^* = \arg\min_{M_a \in \mathbb{M}_a} \left\{ \min_{\tau \in \mathcal{T}_\omega(s_0, M_a)} synCost_{M_a}(\tau) \right\} \tag{6.1}$$

where $synCost_{M_a}(\tau)$ *explicitly denotes the action model used to compute the synthesis cost since in this problem we are dealing with multiple models.*

Let us illustrate a Model Recognition problem and its solution using our working example. Recall that in this working example we have two action models M_a^R and M_a^L both describing different strategies to visit all of the tiles in a 6×6 grid. M_a^R cannot move left when the controller is in mode m_0 and right when it is in mode m_1; vice-versa for M_a^L. Figure 6.3 presents an example of Model Recognition problem that assumes $\mathbb{M}_a = \{M_a^R, M_a^L\}$ as the set of candidate models. On the left of the figure we have the initial situation of the actor and 5 observations distributed on the grid and annotated with their order in the observation sequence. On the right we have the best explanation computed with action model M_a^R. While M_a^L can also explain the observations, it is forced to revisit tiles to do so. This means that the synthesis cost of the explanation by M_a^R is lower than the one by M_a^L and, consequently, M_a^R is the most likely action model that solves the Model Recognition problem illustrated by this figure.

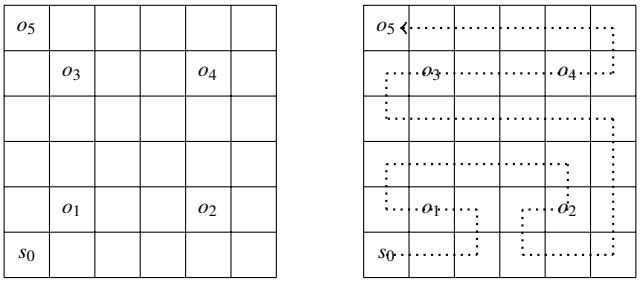

Figure 6.3: (Left) Example of initial situation and 5-observations sequence in the navigation strategies working example. (Right) Best explanation computed with M_a^R.

Figure 6.4 presents a different example where some of the observations have changed. In this case, the situation is reversed and M_a^R cannot explain the observations without revisiting tiles, while M_a^L can as shown in Figure 6.4 Right. This makes M_a^L the most likely action model for the Model Recognition problem illustrated in this figure.

When the candidates set $\mathbb{M}_a$ does not include any model that can explain ω, Definition 36 fails to discriminate between models and there is no solution to the Model Recognition problem. Considering that the premise of Model Recognition is that the observer does not know the action model of the actor and wants to ascertain it, the possibility that none of the candidate models can explain its behaviour should be accounted for. This situation arises frequently when candidate models are the output of some machine learning algorithm.

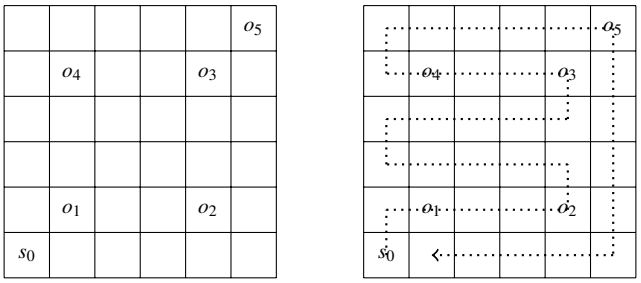

Figure 6.4: (Left) Another example of initial situation and 5-observations sequence in the navigation strategies working example. (Right) Best explanation computed with action model M_a^L.

Our proposal to tackle this problem is to allow the modification of candidate models so that the modified model can generate an explanation for the observation sequence. More specifically, we allow a candidate model M_a to be edited into a comparable model M'_a and redefine the synthesis cost to consider the cost of the explanation by M'_a as well as the cost of editing M_a into M'_a. With that regard we redefine the synthesis cost as

$$synCost_{M_a}(\tau) = \min_{M'_a \in \mathcal{M}(X,A)} \alpha\ cost_{M'_a}(\tau) + (1-\alpha)\ ecost(M_a, M'_a) \tag{6.2}$$

where M'_a is the edited candidate model, $cost_{M'_a}(\tau)$ is the cost of τ with the edited model, $ecost(M_a, M'_a)$ is the cost of editing the candidate model M_a into M'_a and $\alpha \in [0, 1)$ weighs both costs. This technique, known as the *weighted-sum* method (Miettinen, 1998), results in a single-objective optimization problem, whose optimal solution is global Pareto optimal, if it is unique. We note that using $\alpha = 0$ means that models that require less modifications (or no modifications) are preferred, but it will not discriminate between models with the same editing cost. On the other hand, α should be kept smaller than 1, otherwise it will not be possible to discriminate between candidate models that explain the observations, since any model could be transformed into the one that generated the lowest cost explanation at no cost.

The editing cost $ecost(M_a, M'_a)$ measures the cost of editing an action model M_a to transform it into a comparable model M'_a. This cost needs to be defined for any pair of models within the same space $\mathcal{M}(X, A)$. Defining this cost requires us to characterize the space of action models and to define the allowed edit operations. We focus our attention on the case of study where the set of candidate models are induced by STRIPS schematic action models. As we will see later in Section 6.4, working with STRIPS schematic action models makes it possible to define the space of action models as a finite set of action models and to use simple edit operations to modify them.

6.3. The solution to the Model Recognition Problem

This section explains how we arrive at the solution to the Model Recognition problem shown in equation 6.1. A Model Recognition problem $\langle \mathbb{M}_a, M_s, s_0, \omega \rangle$ is a classification problem where each class is represented by a different action model in $\mathbb{M}_a$ and the observation sequence ω is the example to classify. The action model associated to each class acts as the corresponding *class prototype* and it summarizes the set of trajectories $\mathcal{T}(M_a, s_0)$ that can be synthesized with such model. The solution to the Model Recognition problem is then the model $M_a^* \in \mathbb{M}_a$ that maximizes this expression.

$$M_a^* = \arg\max_{M_a \in \mathbb{M}_a} P(\omega | s_0, M_a, M_s) P(M_a) \tag{6.3}$$

where $P(\omega | M_a, M_s)$ is the likelihood of the observation sequence ω and $P(M_a)$ is the prior probability of the action model M_a. The $P(M_a)$ probability expresses whether one model is known to be a priori more likely than the others, for example, because the machine learning algorithm that generates the candidate models assigns different confidence values to each model. When this probability is not given as input, we can reasonable assume that, a priori, all models are equiprobable and remove the term from equation 6.3. The $P(\omega | s_0, M_a, M_s)$ likelihood can be computed by marginalizing over all trajectories that can be generated with M_a.

$$P(\omega | s_0, M_a, M_s) = \sum_{\tau \in \mathcal{T}(M_a, s_0)} P(\omega, \tau | s_0, M_a, M_s) \tag{6.4}$$

where $P(\omega, \tau | s_0, M_a, M_s)$ is the joint likelihood of the observation sequence and the trajectory that we are already familiar with. The computation of equation 6.4 is intractable as the inference setting is unbounded and, therefore, the set $\mathcal{T}(M_a, s_0)$ is potentially infinite. In order to deal with this, we articulate the following approximation:

$$P(\omega | M_a, M_s) \approx \max_{\tau \in \mathcal{T}(M_a, s_0)} P(\omega, \tau | M_a, M_s) \tag{6.5}$$

The difference between Equation (6.4) and Equation (6.5) is that the former favors the model that maximizes the probability of all ways of explaining ω, while the latter favors the model that maximizes the probability of the most likely explanation. We argue that Equation (6.5) aligns better with the principle of rationality as it prefers models that can produce one good explanation rather than several worse explanations.

Finally, recall from previous chapters that the joint likelihood $P(\omega, \tau | M_a, M_s)$ is computed as the product of the synthesis probability $P(\tau | s_0, M_a)$ and the sensing probability $P(\omega | \tau, M_a)$ associated, respectively, to the synthesis and sensing costs. We can then rewrite equation 6.3 into

$$M_a^* = \arg\max_{M_a \in \mathbb{M}_a} \max_{\tau \in \mathcal{T}(M_a, s_0)} (\omega|\tau, M_a) P(\tau|s_0, M_a) P(M_a) \tag{6.6}$$

which simplifies to

$$M_a^* = \arg\max_{M_a \in \mathbb{M}_a} \max_{\tau \in \mathcal{T}(M_a, s_0)} P(\tau|s_0, M_a) \tag{6.7}$$

under the assumptions that 1) all candidate models are a priori equally likely, and 2) all observations emitted from a state are equally likely (which is the case when no sensing costs are considered). As we can see, Equation (6.7) is the probabilistic version of our cost-based solution presented in Equation (6.1).

6.4. Recognition of STRIPS Action Models

While our formulation of the Model Recognition problem is general for any action model, the definition of an edit cost in such an unconstrained setting is unfeasible, so going forward we will focus on a specific type of action models. In particular, we restrict our attention to the recognition of STRIPS and STRIPS-compilable action models. It is well-known that HTNs as well as diverse automata representations like finite state controllers, GOLOG programs or reactive policies can be encoded as STRIPS classical planning models (Alford et al., 2009; Baier et al., 2007; Bonet et al., 2010; Segovia-Aguas et al., 2019; Segovia-Aguas et al., 2018).

The edit cost $ecost(M_a, M'_a)$ that will be defined later in this section measures the cost of transforming M_a into M'_a as the edit distance between their schematic representation. STRIPS schematic action models provide a compact and general representation for specifying classical planning models since it applies to different instances with different sets of objects. For instance, the action models M_a^R and M_a^L of our navigation strategies working example can be represented by schematic action models $\breve{M}_a^R = \langle P, V, \breve{A} \rangle$ and $\breve{M}_a^L = \langle P, V, \breve{A}' \rangle$ where:

- The set of predicates P consists of predicates `xcoord` and `ycoord` of arity 1, predicates `m0` and `m1` of arity 0, and predicate `next` of arity 2.
- $V = \{\mathtt{v}_1, \mathtt{v}_2\}$ is the set of schema symbols.
- Operators $\breve{A}$ are as shown in Figure 6.5 (Left) and operators $\breve{A}'$ in Figure 6.5 (Right), represented in the STRIPS fragment of PDDL

In this schematic representation the set of object O represents coordinates, so for the 6×6 grid of our working example we would have $O = \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_6\}$. Then, state

variables $\texttt{xcoord}(\texttt{c}_\texttt{i})$ and $\texttt{ycoord}(\texttt{c}_\texttt{i})$ represent the current coordinates of the actor in a grid that is described by static state variables $\texttt{next}(\texttt{c}_\texttt{i}, \texttt{c}_{\texttt{i+1}})$, and $\texttt{m0}()$ and $\texttt{m1}()$ represent the controller mode. It is also worth noting that the same schematic action model can be used to define the navigation strategy for $N \times N$ grids of any size, just by changing the set of objects.

We start by adapting our definitions of comparable action models and space of action models for the specific case of schematic action models. Two action models are comparable if they share the same state variables and actions. Similarly, we want two *comparable schematic action models* to induce the same state variables and actions when grounded with the same set of objects. Recall from Section 2.2.1 that the set of state variables induced by the grounding of predicates P with objects O is $X(P, O)$ where $x \in X(P, O)$ is of the form $p(args)$ such that $p \in P$ and $args \in O^{ar(p)}$. Therefore, two comparable schematic action models should share the same predicates to induce the same state variables. On the other hand, the induced set of actions $A(\breve{A}, O)$ is composed of actions of the form $a = \mathsf{name}(args)$ such that a is encoded by the grounded operator $\breve{a}(args) = \langle \mathsf{name}, \mathsf{Pre}(args), \mathsf{Add}(args), \mathsf{Del}(args) \rangle$. In other words, the name of the operator and the objects used to ground it serve as the identifier of the action, while the $\mathsf{Pre}(args)$, $\mathsf{Add}(args)$, $\mathsf{Del}(args)$ lists define its applicability and effects constraints. Then, two comparable schematic action models will induce the same set of actions if their operators have the same *headers*, understanding a header as the pair of name and parameters of an operator ($head(\breve{a}) = \langle \mathsf{name}, \mathsf{Pars} \rangle$).

Definition 37 (Comparable schematic action model)**.** *Two schematic action models $\breve{M}_a = \langle P, V, \breve{A} \rangle$ and $\breve{M}'_a = \langle P', V', \breve{A}' \rangle$ are comparable iff $P = P'$ and $Head(\breve{A}) = Head(\breve{A}')$ where $Head(\breve{A}) = \{head(\breve{a}) \mid \breve{a} \in \breve{A}\}$ is the set of headers of a set of operators $\breve{A}$.*

Definition 38 (Space of schematic action models)**.** *The space of schematic action models $\breve{\mathcal{M}}(P, Head)$ is the set of all models defined over predicates P and headers $Head$.*

For example, we claim that schematic action models $\breve{M}^R_a$ and $\breve{M}^L_a$ are comparable since they share the same set of predicates and the headers of their operators are also the same as seen in Figure 6.5. Additionally, it is implicit by this definition that two comparable schematic action models also share the same set of variable symbols V as this set is given by the parameters of the operators ($\texttt{v}_1$ and $\texttt{v}_2$ in our working example).

The notion of "comparable" also applies to operators, in which case we want the two operators to induce the same set of actions when grounded with the same set of objects. More formally, two operators $\breve{a} \in \breve{A}$ and $\breve{a}' \in \breve{A}'$ are comparable if $head(\breve{a}) = head(\breve{a}')$. Think, for instance, of two operators $\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del} \rangle$ and $\breve{a}' = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}', \mathsf{Add}', \mathsf{Del}' \rangle$ with the same header but different precondition, add and delete lists. Both $\breve{a}$ and $\breve{a}'$ induce the same set of actions but the semantics of the actions (their applicability and effects) are different. Figure 6.5 presents several examples of comparable operators, since $\texttt{m0-inc-x}(\texttt{v}_1, \texttt{v}_2)$, $\texttt{m0-dec-x}(\texttt{v}_1, \texttt{v}_2)$, $\texttt{m1-inc-x}(\texttt{v}_1, \texttt{v}_2)$ and

```
(:action m0-inc-x                                  |   (:action m0-inc-x
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m0)             |     :precondition (and (xcoord ?v1) (m0)
                     (next ?v1 ?v2))               |                        (next ?v1 ?v2))
  :effect (and (not (xcoord ?v1)) (xcoord ?v2)))   |     :effect ())
                                                   |
(:action m0-dec-x                                  |   (:action m0-dec-x
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m0)             |     :precondition (and (xcoord ?v1) (m0)
                     (next ?v2 ?v1))               |                        (next ?v2 ?v1))
  :effect ())                                      |     :effect (and (not (xcoord ?v1)) (xcoord ?v2)))
                                                   |
(:action m0-inc-y                                  |   (:action m0-inc-y
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (ycoord ?v1) (m0)             |     :precondition (and (ycoord ?v1) (m0)
                     (next ?v1 ?v2))               |                        (next ?v1 ?v2))
  :effect (and (not (ycoord ?v1)) (ycoord ?v2)     |     :effect (and (not (ycoord ?v1)) (ycoord ?v2)
               (not (m0)) (m1)))                   |                  (not (m0)) (m1)))
                                                   |
(:action m0-dec-y                                  |   (:action m0-dec-y
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m0)             |     :precondition (and (xcoord ?v1) (m0)
                     (next ?v2 ?v1))               |                        (next ?v2 ?v1))
  :effect (and (not (ycoord ?v1)) (ycoord ?v2)     |     :effect (and (not (ycoord ?v1)) (ycoord ?v2)
            (not (m0)) (m1)))                      |                (not (m0)) (m1)))
                                                   |
(:action m1-inc-x                                  |   (:action m1-inc-x
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m1)             |     :precondition (and (xcoord ?v1) (m1)
                     (next ?v1 ?v2))               |                        (next ?v1 ?v2))
  :effect ())                                      |     :effect ((not (xcoord ?v1)) (xcoord ?v2)))
                                                   |
(:action m1-dec-x                                  |   (:action m1-dec-x
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m1)             |     :precondition (and (xcoord ?v1) (m1)
                     (next ?v2 ?v1))               |                        (next ?v2 ?v1))
  :effect ((not (xcoord ?v1)) (xcoord ?v2)))       |     :effect ())
                                                   |
(:action m0-inc-y                                  |   (:action m0-inc-y
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m1)             |     :precondition (and (xcoord ?v1) (m1)
                     (next ?v1 ?v2))               |                        (next ?v1 ?v2))
  :effect (and (not (ycoord ?v1)) (ycoord ?v2)     |     :effect (and (not (ycoord ?v1)) (ycoord ?v2)
               (not (m1)) (m0)))                   |                  (not (m1)) (m0)))
                                                   |
(:action m0-dec-y                                  |   (:action m0-dec-y
  :parameters (?v1 ?v2)                            |     :parameters (?v1 ?v2)
  :precondition (and (xcoord ?v1) (m1)             |     :precondition (and (xcoord ?v1) (m1)
                     (next ?v2 ?v1))               |                        (next ?v2 ?v1))
  :effect (and (not (ycoord ?v1)) (ycoord ?v2)     |     :effect (and (not (ycoord ?v1)) (ycoord ?v2)
               (not (m1)) (m0)))                   |                  (not (m1)) (m0)))
```

Figure 6.5: Representation in the STRIPS fragment of PDDL of the operators of $\breve{M}_a^R$ (left) and $\breve{M}_a^L$ (right) from the navigation strategy working example.

`m1-dec-x`$(\mathtt{v}_1, \mathtt{v}_2)$ in $\breve{M}_a^R$ are comparable to their counterparts in $\breve{M}_a^L$. In conclusion, we can think of comparable operators as alternative definitions for the same actions.

6.4.1. Edit costs for STRIPS schematic action models

The edit cost is a way of quantifying how dissimilar two comparable models are to one another. In this work, we understand this cost as the distance between the schematic representation of the two models that is computed by counting the number of operations required to transform one model into the other. We start this section by describing the two *edit operations* that can be performed on an operator $\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del} \rangle$ defined over predicates P:

- *Insertion*: An schematic variable $\breve{x} \in \breve{X}(P, \mathsf{Pars})$ is added to Pre or to $\mathsf{Add} \cup \mathsf{Del}$.
- *Deletion*: An schematic variable $\breve{x} \in \breve{X}(P, \mathsf{Pars})$ is removed from Pre or from $\mathsf{Add} \cup \mathsf{Del}$.

We can now formalize an *edit cost* that quantifies the number of edit operations required to transform a STRIPS schematic action model into a comparable one. The distance is symmetric and meets the *metric axioms* provided that the two edit operations, *deletion* and *insertion*, have the same positive cost.

Definition 39 (Edit cost)**.** *Let M_a and M'_a be two comparable action models with schematic representation given by $\breve{M}_a = \langle P, V, \breve{A} \rangle$ and $\breve{M}'_a = \langle P, V, \breve{A}' \rangle$, respectively. The* **edit cost** *$ecost(M_a, M'_a)$ is the minimum number of* edit operations *that is required to transform $\breve{M}_a$ into $\breve{M}'_a$:*

$$ecost(M_a, M'_a) = \sum_{\substack{\breve{a},\breve{a}' \in \breve{A} \times \breve{A}' \\ head(\breve{a}) = head(\breve{a}')}} |pre(\breve{a}) \triangle pre(\breve{a}')| + |(add(\breve{a}) \cup del(\breve{a})) \triangle (add(\breve{a}') \cup del(\breve{a}'))| \tag{6.8}$$

where $pre(\breve{a})$, $add(\breve{a})$, $del(\breve{a})$ return the precondition, add or delete list of operator $\breve{a}$, and $A \triangle B$ denotes the symmetric difference between sets A and B, i.e., $A \triangle B = (A \setminus B) \cup (B \setminus A)$.

As an example, the edit cost between M_a^R and M_a^L is $ecost(M_a^R, M_a^L) = 8$ because we need to perform 8 edit operations on $\breve{M}_a^R$ to transform it into $\breve{M}_a^L$. We can break down this cost into the following operations:

- Delete `xcoord`$(\mathtt{v}_1)$ and `xcoord`$(\mathtt{v}_2)$ from $\mathsf{Add} \cup \mathsf{Del}$ of the `m0-inc-x`$(\mathtt{v}_1, \mathtt{v}_2)$ operator.

- Insert $\texttt{xcoord}(\texttt{v}_1)$ and $\texttt{xcoord}(\texttt{v}_2)$ into $\mathsf{Add} \cup \mathsf{Del}$ of the $\texttt{m0-dec-x}(\texttt{v}_1, \texttt{v}_2)$ operator.
- Insert $\texttt{xcoord}(\texttt{v}_1)$ and $\texttt{xcoord}(\texttt{v}_2)$ into $\mathsf{Add} \cup \mathsf{Del}$ of the $\texttt{m1-inc-x}(\texttt{v}_1, \texttt{v}_2)$ operator.
- Delete $\texttt{xcoord}(\texttt{v}_1)$ and $\texttt{xcoord}(\texttt{v}_2)$ from $\mathsf{Add} \cup \mathsf{Del}$ of the $\texttt{m1-dec-x}(\texttt{v}_1, \texttt{v}_2)$ operator.

6.5. Model Recognition as planning

This section presents our planning-based approach to compute solutions to Model Recognition problems. Following Definition 36, the solution to a Model Recognition problem $\langle \mathbb{M}_a, M_s, s_0, \omega \rangle$ is the most likely action model $M_a \in \mathbb{M}_a$ that generates the explanation τ with minimal synthesis cost $synCost_{M_a}(\tau)$. Our Model Recognition as planning approach compiles the Model Recognition problem into $|\mathbb{M}_a|$ classical planning problems $P_{M_a}(\omega)$ (one for each $M_a \in \mathbb{M}_a$) where $P_{M_a}(\omega)$ is defined in such a way that the cost of a solution plan coincides with the synthesis cost induced by that solution. Then, the most likely action model is identified by solving the compiled planning problems and ranking candidate models according to the cost of the best solution found with them.

If we do not consider the redefinition of the synthesis cost (Eq. (6.2)) that contemplates the editing of candidate models, the Model Recognition problem can be interpreted as finding the candidate model that generates the best explanation for the observation sequence. Under this assumption, planning problems $P_{M_a}(\omega)$ can be built following the compilation presented in our observation decoding as planning approach (Section 4.4.3). On the other hand, if we do consider the editing of candidate models, this compilation cannot be used as it is and it needs to be extended. The main challenge here, and the focus of this section, is in how to extend the compilation presented for our observation decoding as planning approach so as to enable the editing of action models.

We address this challenge by leveraging an encoding that allows representing any STRIPS schematic action model using a finite set of variables, which we will refer to as *alphabet*. The alphabet of a schematic action model will be formally presented in the next section, but for now it suffices to understand that it is composed of Boolean variables that indicate whether an schematic variable belongs to the precondition (Pre) or effects ($\mathsf{Add} \cup \mathsf{Del}$) of an operator. As an example, Figure 6.6 shows a simple move operator and its alphabet.

Without delving into the details of the compilation, let us give an intuitive explanation of the main changes with respect to the original compilation for Observation Decoding. The first change consists in augmenting the state variables of the original compilation with the alphabet of the candidate action model M_a. This change is accompanied with a redefinition of the applicability and effects constraints of the actor actions which are now defined with respect to the values of the alphabet variables. As a consequence, the semantics of the actor actions are now encoded as part of the state variables of the compiled

```
(:action move                              |
  :parameters (?v1 ?v2)                    |
  :precondition (and (at ?v1)              |   <move, pre, at(v1)>
                     (connected ?v1 ?v2))  |   <move, pre, connected(v1,v2)>
  :effect (and (not (at ?v1))              |   <move, eff, at(v1)>
                    (at ?v2)))             |   <move, eff, at(v2)>
```

Figure 6.6: $\texttt{move}(\texttt{v}_1, \texttt{v}_2)$ operator represented in the STRIPS fragment of PDDL (Left) and its alphabet (Right).

problem using the alphabet variables, and any change in the values of these variable will effectively edit the candidate action model. Hence, the last extension that we need to contemplate is a way to modify the value of the alphabet variables. In our compilation, this is done through a new set of *edit actions* that implement the two edit operations, insertion and deletion, defined in Section 6.4.1. Summarizing, in order to make candidate models editable we need to make the following three extensions with respect to the compilation for observation decoding as planning:

- Augment the state variables of the compiled problem with the alphabet of the candidate action model.
- Redefine the applicability and effects constraints of the candidate action model so that the behaviour of the actor actions depends on its alphabet.
- Add edit actions implementing the edit operations that modify the values of the alphabet variables.

A solution plan to the compiled problem $P_{M_a}(\omega)$ presents a distinct structure as illustrated in Figure 6.7. In particular, a solution plan is split in two distinct parts:

1. An (optional) prefix consisting of edit actions that edit the candidate action model M_a into a comparable model M'_a.
2. A suffix consisting of actor and sensing actions that denote the explanation for the observation sequence ω with the edited candidate model M'_a.

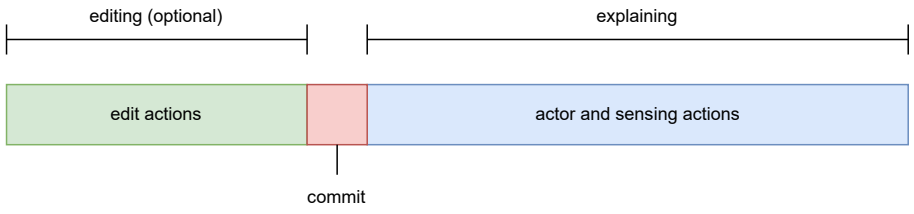

Figure 6.7: Structure of a solution plan for the compiled problem $\mathcal{P}_{M_a}(\omega)$.

It is important that the editing process and the explanation process are isolated. Otherwise, we could run into situations where the action model is edited after part of the observation sequence has already been explained. We avoid these undesired situations by introducing a `commit` action whose sole purpose is to freeze the values of the alphabet variables thus committing to the current model. Therefore, this action marks the end of the prefix, where the editing takes place, and the beginning of the suffix, where an explanation is computed.

6.5.1. The Alphabet of a STRIPS Schematic Action Model

This section formally presents the alphabet of a STRIPS schematic action model. An alphabet is a finite set Boolean variables that can be used to represent the precondition, add and delete lists of all the operators in an action model. We will also see that all models within the same space can be represented using the same set of variables, and that a full valuation over these variables encodes an specific action model. Consequently, an alphabet allows the representation of all schematic action models belonging to the same space. Furthermore, it opens a way to systematically enumerate the space of models and to quantify the distance between any two models with respect to their alphabets.

Let us start by recalling the elements that can appear in the precondition, add and delete lists of an operator. As defined in Section 2.2.1, the Pre, Add, and Del lists of an operator with header $\langle \mathsf{name}, \mathsf{Pars} \rangle$ defined over predicates P are subsets of the set $\breve{X}(P, \mathsf{Pars})$ consisting of schematic state variables defined over P and Pars. As an example, the operators of Figure 6.5 all share the same parameters $\mathsf{Pars} = \{\mathtt{v}_1, \mathtt{v}_2\}$. In this case, $\breve{X}(P, \mathsf{Pars}) = \{\mathtt{xcoord}(\mathtt{v}_1), \mathtt{xcoord}(\mathtt{v}_2), \mathtt{ycoord}(\mathtt{v}_1), \mathtt{ycoord}(\mathtt{v}_2), \mathtt{m0}(), \mathtt{m1}(), \mathtt{next}(\mathtt{v}_1, \mathtt{v}_1), \mathtt{next}(\mathtt{v}_1, \mathtt{v}_2), \mathtt{next}(\mathtt{v}_2, \mathtt{v}_1), \mathtt{next}(\mathtt{v}_2, \mathtt{v}_2)\}$ for the eight operators in each model.

However, not every combination of $\mathsf{Pre} \subseteq \breve{X}(P, \mathsf{Pars})$, $\mathsf{Add} \subseteq \breve{X}(P, \mathsf{Pars})$, and $\mathsf{Del} \subseteq \breve{X}(P, \mathsf{Pars})$ is valid, since STRIPS operators need to satisfy the STRIPS *syntactic constraints*. STRIPS constraints require $\mathsf{Del} \subseteq \mathsf{Pre}$, $\mathsf{Del} \cap \mathsf{Add} = \emptyset$ and $\mathsf{Pre} \cap \mathsf{Add} = \emptyset$. Considering these syntactic constraints, there are four possibilities for a schematic state variable $\breve{x} \in \breve{X}(P, \mathsf{Pars})$:

- $\breve{x}$ only belongs to the precondition list,
- $\breve{x}$ belongs to the precondition and delete lists,
- $\breve{x}$ only belongs to the add list, or
- $\breve{x}$ does not belong to either of the precondition, add or delete lists.

Then, given the set of schematic state variables $\breve{X}(P, \mathsf{Pars})$ for a STRIPS operator $\breve{a}$ with header $\langle \mathsf{name}, \mathsf{Pars} \rangle$ defined over predicates P, there is a total of $4^{|\breve{X}(P,\mathsf{Pars})|}$ different ways

in which $\breve{a}$ can be specified. Furthermore, the $4^{|\breve{X}(P,\mathsf{Pars})|}$ possible operators are comparable since Pars is common to all of them and they are defined over the same set of predicates P. For the operators of Figure 6.5, this means that every operator has a total of $4^{10} = 1,048,576$ comparable operators or, in other words, that every operator can be specified in 4^{10} different ways by changing their semantics. The important takeaway here is that given an operator and the set of predicates over which it is defined, it is possible to define a bounded set of all comparable operators. This also means that the space of schematic action models $\breve{\mathcal{M}}(P, Head)$ is finite as long as we assume the same cost function for all of them (which we will do in the following).

The four ways in which a schematic variable can appear in the precondition, add and delete lists of an operator are stored using two bits and, therefore, lend itself naturally to a representation using two Boolean variables. Let $\breve{x} \in \breve{X}(P, \mathsf{Pars})$ be a schematic state variable for a STRIPS schematic action $\breve{a}$ with header $\langle \mathsf{name}, \mathsf{Pars} \rangle$, we define the two Boolean variables $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$ that represent how $\breve{x}$ appears in $\breve{a}$. Table 6.3 summarizes how the truth values of variables $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$ are interpreted.

$\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$	$\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$	**Meaning**
$\bot$	$\bot$	$\breve{x} \notin \mathsf{Pre}$ and $\breve{x} \notin \mathsf{Add}$ and $\breve{x} \notin \mathsf{Del}$
$\bot$	$\top$	$\breve{x} \notin \mathsf{Pre}$ and $\breve{x} \in \mathsf{Add}$ and $\breve{x} \notin \mathsf{Del}$
$\top$	$\bot$	$\breve{x} \in \mathsf{Pre}$ and $\breve{x} \notin \mathsf{Add}$ and $\breve{x} \notin \mathsf{Del}$
$\top$	$\top$	$\breve{x} \in \mathsf{Pre}$ and $\breve{x} \notin \mathsf{Add}$ and $\breve{x} \in \mathsf{Del}$

Table 6.3: Interpretation of the truth values of Boolean variables $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$

Definition 40 (The alphabet of a STRIPS operator). *Let $\breve{X}(P, \mathsf{Pars})$ be the set of schematic state variables over P for a* STRIPS *operator $\breve{a}$ with header $\langle \mathsf{name}, \mathsf{Pars} \rangle$. The alphabet of operator $\breve{a}$ is the set $\Lambda_{\breve{a}} = \{\langle \mathsf{name}, pe, \breve{x} \rangle \mid pe, \breve{x} \in \{\mathtt{pre}, \mathtt{eff}\} \times \breve{X}(P, \mathsf{Pars})\}$. We will refer to the variables $\lambda \in \Lambda_{\breve{a}}$ as the alphabet variables.*

From this definition we can also conclude that two comparable operators will have the same alphabet, since the set of schematic variables $\breve{X}(P, \mathsf{Pars})$ as well as the name of the operators are the same.

Proposition 2. *Two operators have the same alphabets iff they are comparable.*

Let us exemplify the alphabet of an operator using our navigation strategies working example. In particular, we are going to look at operator $\mathtt{m0\text{-}inc\text{-}x}(\mathtt{v}_1, \mathtt{v}_2)$ shown in Figure 6.5 (Left). The alphabet of this schematic action is the finite set of Boolean variables $\langle \mathtt{m0\text{-}inc\text{-}x}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathtt{m0\text{-}inc\text{-}x}, \mathtt{eff}, \breve{x} \rangle$, where $\breve{x}$ belongs to the set $\breve{X}(P, \mathsf{Pars}) = \{\mathtt{xcoord}(\mathtt{v}_1), \mathtt{xcoord}(\mathtt{v}_2), \mathtt{ycoord}(\mathtt{v}_1), \mathtt{ycoord}(\mathtt{v}_2), \mathtt{m0}(), \mathtt{m1}(), \mathtt{next}(\mathtt{v}_1, \mathtt{v}_1), \mathtt{next}(\mathtt{v}_1, \mathtt{v}_2), \mathtt{next}(\mathtt{v}_2, \mathtt{v}_1), \mathtt{next}(\mathtt{v}_2, \mathtt{v}_2)\}$. We show the complete alphabet in Figure 6.8.

$\langle$m0-inc-x, pre, xcoord(v_1)$\rangle$	$\langle$m0-inc-x, eff, xcoord(v_1)$\rangle$
$\langle$m0-inc-x, pre, xcoord(v_2)$\rangle$	$\langle$m0-inc-x, eff, xcoord(v_2)$\rangle$
$\langle$m0-inc-x, pre, ycoord(v_1)$\rangle$	$\langle$m0-inc-x, eff, ycoord(v_1)$\rangle$
$\langle$m0-inc-x, pre, ycoord(v_2)$\rangle$	$\langle$m0-inc-x, eff, ycoord(v_2)$\rangle$
$\langle$m0-inc-x, pre, m0()$\rangle$	$\langle$m0-inc-x, eff, m0()$\rangle$
$\langle$m0-inc-x, pre, m1()$\rangle$	$\langle$m0-inc-x, eff, m1()$\rangle$
$\langle$m0-inc-x, pre, next(v_1, v_1)$\rangle$	$\langle$m0-inc-x, eff, next(v_1, v_1)$\rangle$
$\langle$m0-inc-x, pre, next(v_1, v_2)$\rangle$	$\langle$m0-inc-x, eff, next(v_1, v_2)$\rangle$
$\langle$m0-inc-x, pre, next(v_2, v_1)$\rangle$	$\langle$m0-inc-x, eff, next(v_2, v_1)$\rangle$
$\langle$m0-inc-x, pre, next(v_2, v_2)$\rangle$	$\langle$m0-inc-x, eff, next(v_2, v_2)$\rangle$

Figure 6.8: Alphabet of STRIPS operator m0-inc-x(v_1, v_2) from schematic action model $\breve{M}_a^R$ of the navigation strategies working example.

The alphabet of an operator is used to represent any possible specification of the operator by simply assigning different values to the alphabet variables. Let $\Lambda_{\breve{a}}$ be the alphabet of an operator $\breve{a}$, a valuation $m : \Lambda_{\breve{a}} \rightarrow \{\top, \bot\}$ induces the operator $\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del} \rangle$ such that:

- $\mathsf{Pre} = \{\breve{x} \in \breve{X}(P, \mathsf{Pars}) \mid m(\langle \mathsf{name}, \texttt{pre}, \breve{x} \rangle) = \top\}$
- $\mathsf{Add} = \{\breve{x} \in \breve{X}(P, \mathsf{Pars}) \mid m(\langle \mathsf{name}, \texttt{pre}, \breve{x} \rangle) = \bot \wedge m(\langle \mathsf{name}, \texttt{eff}, \breve{x} \rangle) = \top\}$
- $\mathsf{Del} = \{\breve{x} \in \breve{X}(P, \mathsf{Pars}) \mid m(\langle \mathsf{name}, \texttt{pre}, \breve{x} \rangle) = \top \wedge m(\langle \mathsf{name}, \texttt{eff}, \breve{x} \rangle) = \top\}$

As an example, Table 6.9 shows the alphabet valuation that induces the operator m0-inc-x(v_1, v_2) from both $\breve{M}_a^R$ and $\breve{M}_a^L$. Note that to improve readability in the table, we are using the notation $\langle$m0-inc-x, pe, x$\rangle$ where pe can take a value in {pre, eff}. As we can see in this table, both operators can be represented using the same set of variables. This example perfectly illustrates Proposition 2 as it shows that comparable operators share the same alphabet. The propositional encoding of these operators only differ in the valuation of some alphabet variables, in particular, the valuation of $\langle$m0-inc-x, eff, xcoord(v_1)$\rangle$ and $\langle$m0-inc-x, eff, xcoord(v_2)$\rangle$.

The proposed propositional encoding results in the most compact representation as any valuation of the alphabet defines a valid operator that complies with the syntactic constraints of STRIPS. Note that there are a total of $2^{|\Lambda_{\breve{a}}|}$ possible valuations over $\Lambda_{\breve{a}}$ which coincides with the $4^{|\breve{X}(P,\mathsf{Pars})|}$ ways in which an operator can be specified (since $|\Lambda_{\breve{a}}| = 2|\breve{X}(P, \mathsf{Pars})|$).

The notion of alphabet can be extended to an schematic action model. In this case, the alphabet needs to contain all the alphabet variables of the operators in the schematic action model.

Definition 41 (The alphabet of a schematic action model). *Let $\breve{M}_a = \langle P, V, \breve{A} \rangle$ be an schematic action model such that each operator $\breve{a} \in \breve{A}$ has alphabet $\Lambda_{\breve{a}}$. The alphabet of*

Alphabet variable	$\breve{M}_a^R$		$\breve{M}_a^L$	
	pe = pre	pe = eff	pe = pre	pe = eff
$\langle$m0-inc-x, pe, xcoord(v_1)$\rangle$	$\top$	$\top$	$\top$	$\bot$
$\langle$m0-inc-x, pe, xcoord(v_2)$\rangle$	$\bot$	$\top$	$\bot$	$\bot$
$\langle$m0-inc-x, pe, ycoord(v_1)$\rangle$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle$m0-inc-x, pe, ycoord(v_2)$\rangle$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle$m0-inc-x, pe, m0()$\rangle$	$\top$	$\bot$	$\top$	$\bot$
$\langle$m0-inc-x, pe, m1()$\rangle$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle$m0-inc-x, pe, next(v_1, v_1)$\rangle$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle$m0-inc-x, pe, next(v_1, v_2)$\rangle$	$\top$	$\bot$	$\top$	$\bot$
$\langle$m0-inc-x, pe, next(v_2, v_1)$\rangle$	$\bot$	$\bot$	$\bot$	$\bot$
$\langle$m0-inc-x, pe, next(v_2, v_2)$\rangle$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 6.9: Valuation of the alphabet variables that induces the operators m0-inc-x(v_1, v_2) from $\breve{M}_a^R$ and $\breve{M}_a^L$.

the schematic action model $\breve{M}_a$ is the set $\Lambda = \bigcup_{\breve{a}\in\breve{A}} \Lambda_{\breve{a}}$ given by the union of the alphabets of its operators.

In the same way that two comparable operators have the same alphabet, two comparable schematic action models also have the same alphabet.

Proposition 3. *Two schematic action models have the same alphabet iff they are comparable.*

With that regard, inserting an schematic variable $\breve{x} \in \breve{X}(P, \mathsf{Pars})$ into Pre (resp. Add $\cup$ Del) is represented by toggling the value of the alphabet variable $\langle \mathsf{name}, \mathtt{pre}, \breve{x}\rangle$ (resp. $\langle \mathsf{name}, \mathtt{eff}, \breve{x}\rangle$) from "$\bot$" to "$\top$". The deletion operation is similarly represented by toggling the value of an alphabet variable from "$\top$" to "$\bot$". This understanding of the edit operations makes it easy to see that our definition of edit cost is equivalent to a Hamming distance that allows only substitution as edit operation and only applies to strings of the same length. This is because both source and target models are comparable and thus have the same alphabet, and the two edit operations can be encoded with a single toggle operation. The maximum edit cost is achieved when it is necessary to toggle every value in the propositional encoding in order to transform one comparable model into another and is given by the size of the alphabet.

Consider, for instance, the comparable operators m0-inc-x(v_1, v_2) from $\breve{M}_a^R$ and $\breve{M}_a^L$ as shown in Table 6.9. The propositional encoding of the two operators only differ in the values of alphabet variables $\langle$m0-inc-x, eff, xcoord(v_1)$\rangle$ and $\langle$m0-inc-x, eff, xcoord(v_2)$\rangle$, both evaluated to "$\top$" in $\breve{M}_a^R$ and to "$\bot$" in $\breve{M}_a^L$. Transforming m0-inc-x(v_1, v_2) ($\breve{M}_a^R$) into m0-inc-x(v_1, v_2) ($\breve{M}_a^L$) requires thus two deletion operations to set the value of these to variables to "$\bot$".

6.5.2. The Compilation

This section is dedicated to explaining how to build the $|\mathbb{M}_a|$ planning problems $\mathcal{P}_{M_a}(\omega)$ resulting from the compilation of a Model Recognition problem $\langle \mathbb{M}_a, M_s, s_0, \omega \rangle$ into planning. The compilation presented here builds on top of the one presented for observation decoding that compiled an observation decoding problem $\langle M_a, M_s, s_0, \omega \rangle$ into a classical planning problem $\mathcal{P}(\omega)$:

$$\mathcal{P}(\omega) = \langle X \cup X^\omega, A \cup A^\omega, \mathsf{App} \wedge \mathsf{App}^\omega, \mathsf{Eff} \wedge \mathsf{Eff}^\omega, s_0^\omega, G^\omega \rangle$$

Particularly, the elements related to the encoding of the monitor automaton $\mathcal{A}(\omega)$ (X^ω, A^ω, App^ω, Eff^ω, and G^ω) are, for the most part, preserved as they were in the original compilation. The main changes that $\mathcal{P}_{M_a}(\omega)$ brings with respect to $\mathcal{P}(\omega)$ is the incorporation of the propositional encoding and edit actions which, together, make the candidate model M_a editable.

Following, we formally define each of the elements of the compiled planning problem $\mathcal{P}_{M_a}(\omega) = \langle X', A', \mathsf{App}', \mathsf{Eff}', s_0', G' \rangle$ associated to the candidate action model $M_a = \langle X, A, \mathsf{App}, \mathsf{Eff}, c \rangle$.

State variables: The main change to the state variables of $\mathcal{P}_{M_a}(\omega)$ with respect to the compilation of observation decoding as planning is that X' is augmented with the alphabet Λ of the schematic representation of M_a. This change implements the first step towards making actor actions A editable, which consists in encoding their applicability (App) and effects (Eff) constraints as part of the state of the compiled problem $\mathcal{P}_{M_a}(\omega)$ through the variables in Λ. We also include in X' a variable `edit_mode` used to split the editing process (`edit_mode` $= \top$) and the explanation process (`edit_mode` $= \bot$). Then, the set of state variables X' consists of:

- the state variables of the actor X,
- $X^\omega = \{\mathtt{q}\}$ used to represent the current location of the monitor automaton $\mathcal{A}(\omega)$,
- the alphabet Λ of the action model M_a, and
- a state variable `edit_mode` that indicates whether the editing process is over or not and is used to isolate the editing process and the explanation process.

Actions: As in the compilation for observation decoding as planning, the compiled problem $\mathcal{P}_{M_a}(\omega)$ includes all the actor actions A as well as the actions A^ω that encode the transitions of the monitor automaton $\mathcal{A}(\omega)$. Additionally, A' now includes a new set of edit actions A^Λ that implement the edit operations over the alphabet Λ. We also need to include here the `commit` action that marks the end of the editing process and the beginning of the explanation process. In summary, A' includes:

- the actor actions A,
- actions A^{ω} encoding the transitions of $\mathcal{A}(\omega)$,
- the set of edit actions A^{Λ} consisting of actions `insert_`λ and `delete_`λ for every $\lambda \in \Lambda$, and
- a `commit` action.

Initial situation: The initial state of $\mathcal{P}_{M_a}(\omega)$ specifies the valuation over Λ that encodes the applicability and effects constraints of the actor actions, alongside the initial situation of the actor s_0, and the initial location q_0 of the automaton $\mathcal{A}(\omega)$. With this regard, the initial state s'_0 is defined as:

- $s'_0(x) = s_0(x)$ for $x \in X$ to set up the initial situation of the actor.
- $s'_0(\mathtt{q}) = 0$ sets the initial location of $\mathcal{A}(\omega)$ as q_0.
- $s'_0(\lambda) = m(\lambda)$ for $\lambda \in \Lambda$ where m is the valuation of Λ that encodes the action model M_a. With this
- $s'_0(\mathtt{edit\_mode}) = \top$ denoting that we start in editing mode.

Goal condition: The goal condition G' is given by the constraint $\langle \mathtt{q}, \text{"}\mathtt{q} = t\text{"} \rangle$ where t is the length of the observation sequence ω. This goal condition, which is the same as in the compilation for observation decoding as planning, symbolizes reaching the accepting location of the monitor automaton $\mathcal{A}(\omega)$ and, therefore, finding an explanation for ω.

Applicability and effects constraints: We left the formalization of the App' and Eff' constraints for last as this is the part that requires more attention. This is due mainly to the need to redefine the semantics of the sets A as well as the introduction of the new edit actions A^{Λ} and the commit action. Following, we briefly summarize all the changes that the App' and Eff' constraints need to contemplate:

- A redefinition of the semantics of actions in A so that actor actions behave according to the values of the alphabet variables Λ.
- A small modification to the applicability of actions in A^{ω} to forbid their execution during the editing process.
- The definition of the newly introduced edit actions A^{Λ}.
- The definition of the `commit` action.

We start off with the redefinition of the semantics of the actor actions. Let $a \in A$ be the actor action $\mathsf{name}(args)$ whose schematic representation is given by the operator $\breve{a} = \langle \mathsf{name}, \mathsf{Pars}, \mathsf{Pre}, \mathsf{Add}, \mathsf{Del} \rangle$. We make action a editable by making its semantics dependant on the alphabet of $\breve{a}$. Say that $\breve{x}(args) \in X$ is the state variable associated to an schematic variable $\breve{x}$ in either Pre, Add, or Del, we redefine the semantics of a as implications whose antecedent are conditions over the alphabet variables $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$, and whose consequent is the state variable $\breve{x}(args)$. For example, in the applicability constraint we use implications of the form:

"If $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$, then $\breve{x}(args)$"

which makes it so that $\breve{x}(args)$ needs to hold if $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ holds. In other words, $\breve{x}(args)$ is a precondition if $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ says it is. The effects constraint is similarly specified using implications, with the difference that this time the antecedent contains both $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$. This is because we need $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$ to determine whether $\breve{x}(args)$ is an effect or not, and $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ to determine if it is a positive or negative effect.

"If not $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$, then $\breve{x}(args)^+$"

"If $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$, then not $\breve{x}(args)^+$"

These three implications encode three out of the four ways in which an schematic variable can appear in the precondition, add, and delete lists of an operator (see Table 6.3). The last remaining case is when the schematic variable belongs to neither of these lists, which can be ignored for the purpose of this encoding. Hereafter, we denote by $\mathsf{c\_pre}(\breve{x}, args)$, $\mathsf{c\_add}(\breve{x}, args)$, $\mathsf{c\_del}(\breve{x}, args)$ the constraints that define, respectively, the precondition, positive, and negative effect $\breve{x}(args)$ conditioned on the values of alphabet variables $\langle \mathsf{name}, \mathtt{pre}, \breve{x} \rangle$ and $\langle \mathsf{name}, \mathtt{eff}, \breve{x} \rangle$.

The redefined applicability and effects constraints that allow action a to be editable contemplate these kind of constraints for every schematic variable that can appear in the operator $\breve{a}$, that is, for every $\breve{x} \in \breve{X}(P, \mathsf{Pars})$. With that regard, we define $\mathsf{App}'(a)$ and $\mathsf{Eff}'(a)$ as follows:

$$\mathsf{App}'(a) = \langle \mathtt{edit\_mode}, \bot \rangle \bigcup_{\breve{x} \in \breve{X}(P, \mathsf{Pars})} \mathsf{c\_pre}(\breve{x}, args) \tag{6.9}$$

$$\mathsf{Eff}'(a) = \bigcup_{\breve{x} \in \breve{X}(P, \mathsf{Pars})} \mathsf{c\_add}(\breve{x}, args) \cup \mathsf{c\_del}(\breve{x}, args) \tag{6.10}$$

As shown in Equation 6.9, the applicability constraint also checks that the editing process is over through constraint $\langle \mathtt{edit\_mode}, \bot \rangle$. This same constraint is also added

to the applicability constraint of actions in A^{ω} to forbid their execution during the editing process. An interesting feature of these redefined actions is that they can assume any semantics definable by their operator just by changing the valuation over their alphabets. By extension, this means that an action model whose actions have been redefined as shown above, can be considered as a "template" that can take on the semantics of any model in its space just by changing the valuation over the alphabet.

Next, we move on to the edit actions A^{Λ}. For the definition of their semantics we distinguish between `insert_`λ actions, which set the value of λ to $\top$, and `delete_`λ actions, which set the value of λ to $\bot$. With that regard, the semantics of `insert_`λ for all $\lambda \in \Lambda$ are defined as follows:

$$\mathsf{App}'(\texttt{insert\_}\lambda) = \langle \texttt{edit\_mode}, \top \rangle$$

$$\mathsf{Eff}'(\texttt{insert\_}\lambda) = \langle \lambda^{+}, \top \rangle$$

and the semantics of `delete_`λ for all $\lambda \in \Lambda$ as follows:

$$\mathsf{App}'(\texttt{delete\_}\lambda) = \langle \texttt{edit\_mode}, \top \rangle \cup \langle \lambda, \top \rangle$$

$$\mathsf{Eff}'(\texttt{delete\_}\lambda) = \langle \lambda^{+}, \bot \rangle$$

The applicability constraint of both insert and delete actions restricts their execution to states where `edit_mode` $= \top$, that is, to the part of the plan where the editing process is still undergoing.

The last remaining action whose semantics we need to define is the `commit` action. This is a fairly simple action whose sole purpose is to change the value of state variable `edit_mode` from $\top$ to $\bot$, effectively marking the end of the editing process and the beginning of the explanation process. The semantics of `commit` are thus defined as follows:

$$\mathsf{App}'(\texttt{commit}) = \langle \texttt{edit\_mode}, \top \rangle$$

$$\mathsf{Eff}'(\texttt{commit}) = \langle \texttt{edit\_mode}^{+}, \bot \rangle$$

Solution plan: A solution plan for $\mathcal{P}_{M_a}(\omega)$ starts with an optional prefix consisting of edit actions. We say that this prefix is optional because no modification might be needed for the candidate model M_a to explain the observation sequence ω. After the prefix (or as first action if there is no prefix), the first action will always be the `commit` action since this is the action that makes it possible for actions in A and A^{ω} to be applicable. The last

```
01 :  (insert_m0-inc-x_eff_xcoord-v1)  06 :  (m0-inc-x 1 2)
02 :  (insert_m0-inc-x_eff_xcoord-v2)  07 :  (m0-inc-x 2 3)
03 :  (insert_m1-dec-x_eff_xcoord-v1)  08 :  (m0-inc-y 1 2)
04 :  (insert_m1-dec-x_eff_xcoord-v2)  09 :  (m1-dec-x 3 2)
05 :  (commit)                         10 :  (sense_1)
```

Figure 6.10: Excerpt of a solution plan for the compiled problem $\mathcal{P}_{M_a}(\omega)$.

part of the solution plan is a suffix consisting of actions in A and A^ω which follows the same structure as a solution plan for observation decoding.

We now show an example of a solution plan using our navigation strategies working example. Assume the compiled problem $\mathcal{P}_{M_a}(\omega)$ where ω is the observation sequence illustrated in Figure 6.1 and as candidate model we use M_a^L, that is, the navigation strategy that can only move left when the controller is in mode m_0 and right when in mode m_1. Recall also that M_a^L cannot explain ω without revisiting tiles and that the schematic representation of M_a^L is shown in Figure 6.5 (Right). Figure 6.10 shows the solution plan for $\mathcal{P}_{M_a}(\omega)$ up to the first sensing action, but assume that the full plan follows the path illustrated in Figure 6.3 (Right). In order for M_a^L to walk this path, it is necessary to edit the operators $\texttt{m0-inc-x}(\texttt{v}_1, \texttt{v}_2)$ and $\texttt{m1-dec-x}(\texttt{v}_1, \texttt{v}_2)$ so they behave as in M_a^R. The solution plan shows the four edit actions (actions 01 to 04) that are necessary to make these changes and which also constitute the prefix of the plan. Afterwards, comes the `commit` action (position 05) marking the start of the suffix consisting of actor and sensing actions (actions 06 onward).

Notes on Implementation

In this section, we show how to implement in PDDL the actor actions with redefined applicability and effects constraints so that their behaviour depends on the alphabet variables. We use as an example action `m0-inc-x` whose semantics are defined by the operator $\texttt{m0-inc-x}(\texttt{v}_1, \texttt{v}_2)$ shown in Figure 6.5. In this case, the constraints that we need to implement are:

$$\mathsf{App}'(\texttt{m0-inc-x}) = \langle \texttt{edit\_mode}, \bot \rangle \bigcup_{\check{x} \in \check{X}(P, \mathsf{Pars})} \mathsf{c\_pre}(\check{x}, args)$$

$$\mathsf{Eff}'(\texttt{m0-inc-x}) = \bigcup_{\check{x} \in \check{X}(P, \mathsf{Pars})} \mathsf{c\_add}(\check{x}, args) \cup \mathsf{c\_del}(\check{x}, args)$$

where $\check{X}(P, \mathsf{Pars}) = \{\texttt{xcoord}(\texttt{v}_1), \texttt{xcoord}(\texttt{v}_2), \texttt{ycoord}(\texttt{v}_1), \texttt{ycoord}(\texttt{v}_2), \texttt{m0}(), \texttt{m1}(), \texttt{next}(\texttt{v}_1, \texttt{v}_1), \texttt{next}(\texttt{v}_1, \texttt{v}_2), \texttt{next}(\texttt{v}_2, \texttt{v}_1), \texttt{next}(\texttt{v}_2, \texttt{v}_2)\}$

The challenge here is in implementing the constraints $\mathsf{c\_pre}(\check{x}, args)$, $\mathsf{c\_add}(\check{x}, args)$, $\mathsf{c\_del}(\check{x}, args)$. Recall that these constraints are implications "$lhs \rightarrow rhs$" where the

left-hand side *lhs* is a condition over the alphabet variables and the right-hand side *rhs* is the state variable associated to this alphabet variables. We take a different approach to implement these constraints depending on whether they appear in the applicability constraint ($\mathsf{c\_pre}(\breve{x}, args)$) or effects constraint ($\mathsf{c\_add}(\breve{x}, args)$ and $\mathsf{c\_del}(\breve{x}, args)$). In the case of $\mathsf{c\_pre}(\breve{x}, args)$ we implement the implication "$lhs \rightarrow rhs$" as an equivalent disjunction "$\neg lhs \vee rhs$". For $\mathsf{c\_add}(\breve{x}, args)$ and $\mathsf{c\_del}(\breve{x}, args)$, on the other hand, we implement them as conditional effects where *lhs* is the condition and *rhs* the effect.

```
(:action m0-inc-x
      :parameters (?v1 ?v2)
      :precondition
           (and (not (edit_mode))
                (or (not (m0-inc-x_pre_xcoord-v1)) (xcoord ?v1))
                (or (not (m0-inc-x_pre_xcoord-v2)) (xcoord ?v2))
                (or (not (m0-inc-x_pre_ycoord-v1)) (ycoord ?v1))
                (or (not (m0-inc-x_pre_ycoord-v2)) (ycoord ?v2))
                (or (not (m0-inc-x_pre_m0)) (m0))
                (or (not (m0-inc-x_pre_m1)) (m1))
                (or (not (m0-inc-x_pre_next-v1-v1)) (next ?v1 ?v1))
                (or (not (m0-inc-x_pre_next-v1-v2)) (next ?v1 ?v2))
                (or (not (m0-inc-x_pre_next-v2-v1)) (next ?v2 ?v1))
                (or (not (m0-inc-x_pre_next-v2-v2)) (next ?v2 ?v2))
                )
      :effect
           (and (when (and (not (m0-inc-x_pre_xcoord-v1)) (m0-inc-x_eff_xcoord-v1)) (xcoord ?v1))
                (when (and (not (m0-inc-x_pre_xcoord-v2)) (m0-inc-x_eff_xcoord-v2)) (xcoord ?v2))
                (when (and (not (m0-inc-x_pre_ycoord-v1)) (m0-inc-x_eff_ycoord-v1)) (ycoord ?v1))
                (when (and (not (m0-inc-x_pre_ycoord-v2)) (m0-inc-x_eff_ycoord-v2)) (ycoord ?v2))
                (when (and (not (m0-inc-x_pre_m0)) (m0-inc-x_eff_m0)) (m0))
                (when (and (not (m0-inc-x_pre_m1)) (m0-inc-x_eff_m1)) (m1))
                (when (and (not (m0-inc-x_pre_next-v1-v1)) (m0-inc-x_eff_next-v1-v1)) (next ?v1 ?v1))
                (when (and (not (m0-inc-x_pre_next-v1-v2)) (m0-inc-x_eff_next-v1-v2)) (next ?v1 ?v2))
                (when (and (not (m0-inc-x_pre_next-v2-v1)) (m0-inc-x_eff_next-v2-v1)) (next ?v2 ?v1))
                (when (and (not (m0-inc-x_pre_next-v2-v2)) (m0-inc-x_eff_next-v2-v2)) (next ?v2 ?v2))
                (when (and (m0-inc-x_pre_xcoord-v1) (m0-inc-x_eff_xcoord-v1)) (not (xcoord ?v1)))
                (when (and (m0-inc-x_pre_xcoord-v2) (m0-inc-x_eff_xcoord-v2)) (not (xcoord ?v2)))
                (when (and (m0-inc-x_pre_ycoord-v1) (m0-inc-x_eff_ycoord-v1)) (not (ycoord ?v1)))
                (when (and (m0-inc-x_pre_ycoord-v2) (m0-inc-x_eff_ycoord-v2)) (not (ycoord ?v2)))
                (when (and (m0-inc-x_pre_m0) (m0-inc-x_eff_m0)) (not (m0)))
                (when (and (m0-inc-x_pre_m1) (m0-inc-x_eff_m1)) (not (m1)))
                (when (and (m0-inc-x_pre_next-v1-v1) (m0-inc-x_eff_next-v1-v1)) (not (next ?v1 ?v1)))
                (when (and (m0-inc-x_pre_next-v1-v2) (m0-inc-x_eff_next-v1-v2)) (not (next ?v1 ?v2)))
                (when (and (m0-inc-x_pre_next-v2-v1) (m0-inc-x_eff_next-v2-v1)) (not (next ?v2 ?v1)))
                (when (and (m0-inc-x_pre_next-v2-v2) (m0-inc-x_eff_next-v2-v2)) (not (next ?v2 ?v2)))
                )
)
```

Figure 6.11: PDDL implementation of an actor action with redefined semantics.

Figure 6.11 shows the action with redefined semantics implemented in PDDL. Since the set $\breve{X}(P, \mathsf{Pars})$ contains 10 elements, the PDDL implementation results in 10 disjunctions in the preconditions representing the $\mathsf{c\_pre}(\breve{x}, args)$ constraints, 10 conditional effects representing the $\mathsf{c\_add}(\breve{x}, args)$ constraints, and another 10 conditional effects representing the $\mathsf{c\_del}(\breve{x}, args)$ constraints. There are a few optimization that can be done to reduce these numbers. For example, if we know that a variable is static, like the `next` variables, there is no need to implement the $\mathsf{c\_add}(\breve{x}, args)$ and $\mathsf{c\_del}(\breve{x}, args)$ constraints

associated to it (since static variables are not modified). Another optimization that often makes sense is to not consider variables with repeated parameters, such as `next(v1, v1)`.

6.6. Experimental Evaluation

In this section, we evaluate the empirical performance of Model Recognition as planning in three different use cases. Each use case presents a meaningful scenario where Model Recognition can be exploited. As in our working example, we assume a simple sensor model that assumes that part of the actor state is observable and part is hidden.

All experiments were run in an Intel Core i5 3.10GHz x 4 16GB of RAM and the classical planner we used to solve the instances that result from the compilation was Madagascar (Rintanen, 2014) due to its ability to deal with classical planning problems with dead-ends (López, Celorrio, & Olaya, 2015). Other planners, such as FastDownward were also tested but provided worse experimental results. Planning problems were solved with a timeout value of 1000 secs of CPU-time, and 8GB of memory.

6.6.1. Use Case 1: Recognition of *regular automata*

The first experiment, which doubles as a proof of concept, exploits Model Recognition for a classical *string classification* problem. In this experiment, the observation sequence represents the string to classify and each candidate model represents a different regular automaton that accepts strings that belong to its language.

Figure 6.12 illustrates a 4-symbol and 5-state *regular automaton* for recognizing the $(abcd)^+$ language. The *input alphabet* is $\Sigma = \{a, b, c, d\}$, and the machine states are $Q = \{q_0, q_1, q_2, q_3, \underline{q_4}\}$, where $\underline{q_4}$ is the only accepting state. For instance, executing the planning model that encodes the *regular automaton* of Figure 6.12 with the input string *abcdabcd* produces the following 8-action plan $(\langle a, q_0\rangle \rightarrow q_1)$, $(\langle b, q_1\rangle \rightarrow q_2)$, $(\langle c, q_2\rangle \rightarrow q_3)$, $(\langle d, q_3\rangle \rightarrow \underline{q_4})$, $(\langle a, \underline{q_4}\rangle \rightarrow q_1)$, $(\langle b, q_1\rangle \rightarrow q_2)$, $(\langle c, q_2\rangle \rightarrow \underline{q_3})$, $(\langle d, q_3\rangle \rightarrow \underline{q_4})$.

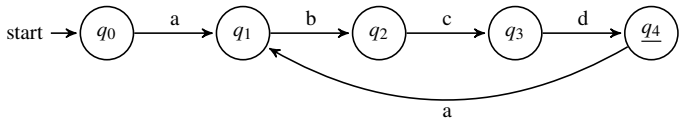

Figure 6.12: A 4-symbol and 5-state regular automata for recognizing the $(abcd)^+$ language ($\underline{q_4}$ is the accepting state).

In this experiment, $\mathbb{M}_a$ comprises five candidate action models, each representing a different 5-state and 4-symbol regular automaton. The five regular languages defined by these automata are the following:

- $\mathcal{L}1$: $a^+(b|c)d(dd)^*a$

	$\mathcal{L}1$	$\mathcal{L}2$	$\mathcal{L}3$	$\mathcal{L}4$	$\mathcal{L}5$
$\mathcal{L}1$	20	0	0	0	0
$\mathcal{L}2$	0	20	0	0	0
$\mathcal{L}3$	0	0	20	0	0
$\mathcal{L}4$	0	0	0	20	0
$\mathcal{L}5$	0	0	0	0	20

Table 6.4: Confusion matrix for regular automata recognition.

- $\mathcal{L}2$: $bd(abd)^*cd^*c^+$
- $\mathcal{L}3$: $d^*c(ac)^*db^*ac^*b^+$
- $\mathcal{L}4$: $(cc)^+bd(abd)^*$
- $\mathcal{L}5$: $(d|a)a(ba)^*c^+d(dc^*d)^*$

For each regular language, we generated 20 random strings (observation sequences), whose lengths range from 20 to 30 symbols, thus generating a total of 100 strings. Observations convey only the symbols; the internal state of the automata as well as the transitions taken (actions) remain hidden to the observer. Consequently, observation sequences only convey the string to classify.

Table 6.4 shows the *confusion matrix* resulting from classifying the 100 strings with our method. In this matrix, rows represent the actual class and columns represent the class predicted by our *Model Recognition as planning* approach. Despite using a suboptimal classical planner, we can observe that the class for all 100 input strings was correctly recognized, which proves the feasibility of our method for classification tasks. The outstanding results reveal that our approach is very suitable for the classification of generative planning models that are more restrictive than STRIPS models, as it is the case of regular automata.

6.6.2. Use Case 2: Recognizing Failures in a Non-deterministic *Blocksworld*

In this second use case, we tackle the problem of recognizing failures in a non-deterministic *blocksworld* whose actions can fail in the following ways:

- **Failure 1**: the execution of a `stack` action fails and causes no effect
- **Failure 2**: the execution of `unstack` fails and causes no effect
- **Failure 3**: `unstack` fails and drops the block on the table

The problem of recognizing the failure is posed as a Model Recognition problem $\langle \mathbb{M}_a, M_s, s_0, \omega \rangle$ such that 1) ω stems from a trajectory where a single failure took place,

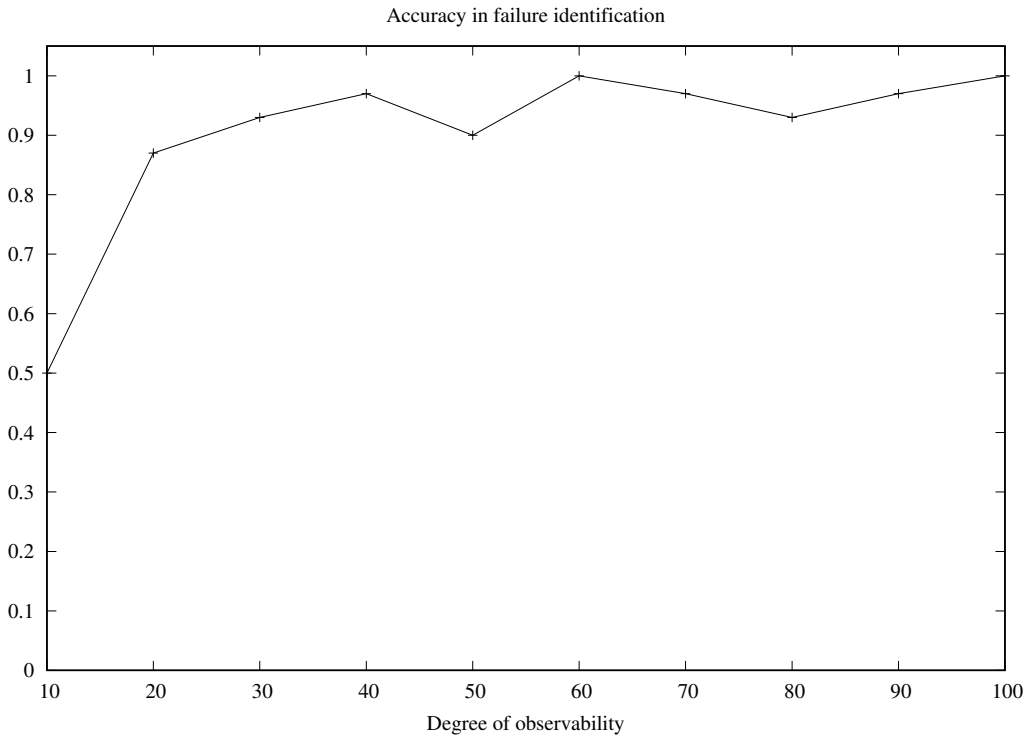

Figure 6.13: Accuracy for the recognition of failures in a non-deterministic blocksworld.

and 2) the $\mathbb{M}_a$ comprises three different 4-operator *blocksworld* action models, each one extended with an additional operator that encodes one of the three above failures. In this way, identifying a failure can be done by solving the Model Recognition problem and finding the candidate model that best explains the observation sequence. For instance, if we have two consecutive observations where the same block is held by the gripper, the candidate model that encodes failure 1 will provide the best explanation as it requires no editing.

We measure the *accuracy* of our approach across different degrees of observability. The accuracy metric is defined as the number of correct predictions about the type of failure over all predictions made, and the degree of observability indicates the probability of observing a state variable (through the corresponding observable variable). In this experiment, we also assume that the sensing process is not intermittent and, therefore, the length of the explanation is given by the length of the observation sequence.

Figure 6.13 shows the *accuracy* of our approach when identifying failures over 30 different observation sequences across different degrees of observability. The results show that accuracy stabilizes in the range of 90% to 100% once we reach a threshold of 20% observability. These are very positive results as it proves the feasibility of the approach and its robustness to partial observability.

6.6.3. Use Case 3: Recognition of Navigation Strategies

The last use case is based on our navigation strategies working example. Here, the set of candidates $\mathbb{M}_a$ consists of 8 different navigation strategies for $N \times N$ grids. As in our working example, all navigation strategies have an underlying hidden 2-mode controller that restrict the applicability of the actions. We generated a set of trajectories depicting the paths followed by each navigation strategy to solve the planning problem of *visiting all* cells in a grid. After that, observation sequences were extracted using a sensor model that, as described in our working example, only observes the position of the agent.

An interesting aspect of this experiment is that all navigation policies are at a maximum edit cost of 4 from the base *classical navigation model* that can move the robot in any direction at any given state. In other words, a maximum of 4 edit actions are needed to remove the restrictions imposed by the controller and, thus, letting the actor move freely. This aspect heavily constrains the discriminating power of our approach, since only 4 edit actions are needed for any candidate model to explain any observation sequence.

Figure 6.14 shows the classification *accuracy* achieved by our approach with respect to a range of degrees of observability, from 0% to 100% with 10% increments. In this experiment we included the 0% case which corresponds to observations sequences where only the initial and final states are observed. The figure shows that for 0% observability we were unable to unmistakably identify the navigation policies, but from 10% of observability onwards, we start to correctly classify half of the observations. Accuracy stabilizes after 40% observability in the range 0.875 to 1 which means that at most only one out of the 8 observation sequences was not correctly classified.

6.7. Discussion

We start this discussion commenting on a few interesting aspects of our approach. The first one is that our proposal to address the Model Recognition problem is strongly connected to the Temporal Inference problem. In order to draw this connection it suffices to understand that an alphabet is a set of *static* variables (at least with respect to the explanation process) that represents a candidate model. This means that the Model Recognition problem can be interpretted as a Temporal Inference problem, in the augmented state-space model that includes the alphabet of the candidate models, where the hypotheses represent different initial states and each one encodes a different candidate model.

Related to the first topic, an alternative idea to our proposal to discriminate between models that cannot explain the observation sequence is to enable edit operations over the actor state variables rather than the alphabet (Sohrabi et al., 2016). However, editing the model is more scalable since a single edit operation can see its effects manifested several times in the trajectory, which otherwise would need to be edited one by one. And on the topic of alternative formulations, our proposal leverages the Hamming distance that

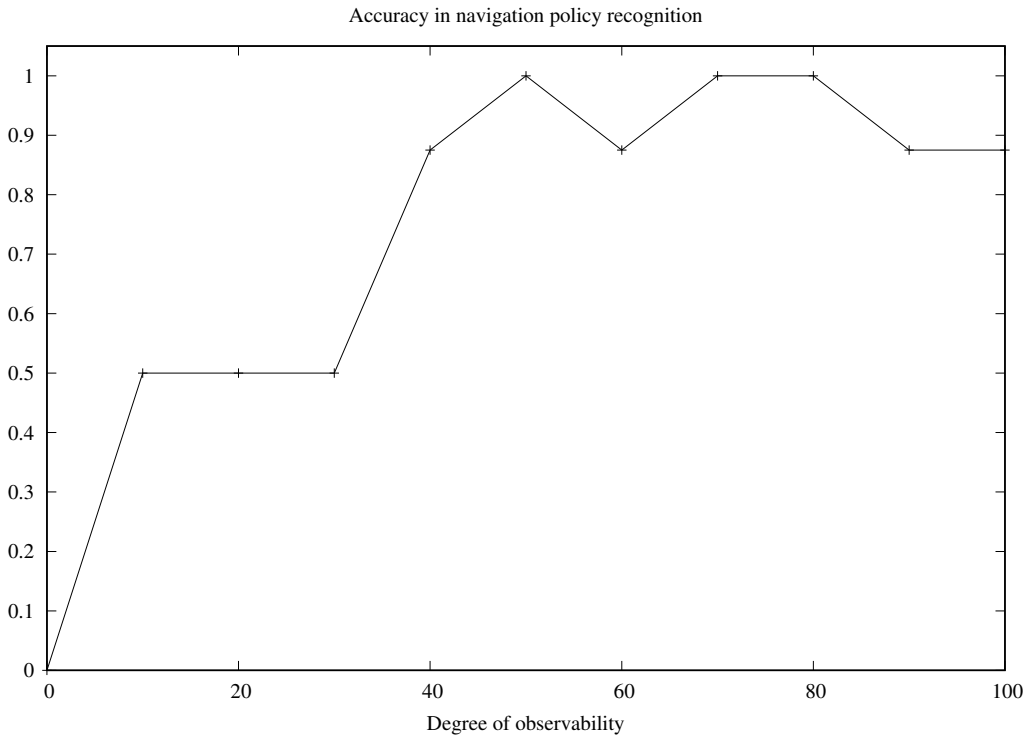

Figure 6.14: Classification accuracy for the recognition of navigation strategies.

allows only substitution, and hence, it only applies to strings of the same length. There are other edit distances, like the Levenshtein distance which allows deletion, insertion and substitution, and work for strings of different lengths. We believe this could open the way for more expressive planning languages that cannot be represented with a finite set of variables and require, instead, a set of rules or maybe a grammar to describe their syntax.

Another interesting point is that each compiled problem $\mathcal{P}_{M_a}(\omega)$ can be seen as an online learning problem where M_a is updated according to the learning example embodied by ω. With this regard parameter α can be considered a *learning rate* that weights how much we are willing to modify the input model in order to explain the new example.

Model Recognition resembles other tasks like Model Reconciliation, which uses model editing to conform two PDDL models, where one of them can be an annotated model or an empty model, with respect to a fully observed optimal plan computed with one of the two models (Chakraborti et al., 2018b; Chakraborti et al., 2017; Sreedharan et al., 2018). Also, the Model Drift (Bryce et al., 2016) approach estimates the error between a computational model, and a ground-truth model which changes as the domain evolves, as the normalized symmetric difference of the propositions that appear in the preconditions and effects of both models. In this case, plan observations are used to hypothesize variations of the model. Unlike Model Reconciliation and Model Drift, our proposal supports observations that contain no actions and incomplete – even empty– intermediate states. Moreover, we leverage our framework to posit the Model Recognition problem as a cost-driven classification problem with as many classes as candidate models.

Part V

Learning

7. ACTION MODEL LEARNING

This chapter addresses the learning problem in the context of planning models, which is usually referred to as Action Model Learning or Action Model Acquisition in the literature. Broadly speaking, the goal of Action Model Learning is to build an action model (usually its schematic representation) from a set of *learning examples*. In that sense, the learning problem is not that different from a Model Recognition problem, since they both pursue to identify the action model of the actor. However, in contrast with Model Recognition, no set of candidates is provided and the action model needs to be estimated from the learning examples.

We start this chapter in Section 7.1 discussing state-of-the-art learning approaches and motivating the use of a planning-based approach. Section 7.2 formalizes the Action Model Learning problem and Section 7.3 motivates our proposed solution to this problem. We continue in Section 7.4 with a presentation of our planning-based approach to Action Model Learning. In Section 7.5 we propose novel metrics to measure the quality of the learned models after discussing the limitations of the ones commonly used in the literature. Then, in Section 7.6 we use these new metrics to evaluate the solutions found by our approach. Finally, we conclude this chapter in Section 7.7 with a discussion of related topics.

7.1. On the Use of Planning for Action Model Learning

The purpose of this section is to highlight the benefits of using planning for Action Model Learning. In our view, the main advantage of a planning-based approach is that it is able to learn from learning examples that contain gaps, missing actions and states lost due to the intermittency of the sensing process. While intermittency is commonly assumed in the planning-based approaches to inference, no Action Model Learning approach makes this assumption. This leads to a very constrained understanding of learning example, which limits the applicability of Action Model Learning approaches. In order to stress this matter, we are going to briefly review the state-of-the-art approaches to Action Model Learning, giving special attention to the type of learning examples supported by the approach.

Before jumping into the review, let us talk about the kind of learning examples used in Action Model Learning. Generally speaking, learning examples contain data about the trajectory followed by some agent walking in the state-space model that we want to learn and are typically presented in the form of sequences. Since it is not always realistic to assume that trajectories are fully available, observations of trajectories are more commonly used in the literature as learning examples. In theory, observations of trajectories may be noisy and/or intermittent, that is, not every state/action of a trajectory is always observable. In practice, however, most approaches to Action Model Learning

adopt the following two assumptions: (i) the initial state of the trajectory is fully observed, and (ii) the sequence of actions followed by the actor is fully observed. With that regard, a learning example can be understood as a pair $\langle s_0, \omega \rangle$ such that ω is the observation sequence that stems from the trajectory $\tau = s_0[\![\pi]\!]$ traversed by some actor agent following plan π. In Action Model Learning little attention is put on the sensor model used to capture the observation sequence ω and it is usually assumed that state variables and actions are directly observable although partially.

An important factor to consider when talking about learning examples is the degree to which states in τ and actions in π are observed so we put forth a broad characterization of learning examples based on this criteria. The purpose of this characterization is to give an intuitive idea of how much information the observation sequence ω of a learning example $\langle s_0, \omega \rangle$ provides, without being too formal. With regards to the state trajectory τ we identify the following cases:

1. We say that τ is fully-observable (FO) if every state is observed and the observations are complete, meaning that all state variables are known.
2. We say that τ is partially-observable (PO) if some states are missing in the learning example or the observations are partial, meaning that the values of some state variables are unknown.
3. We say that τ is non-observable (NO) if all intermediate states are missing. This is a special case of partial observability.

Similarly, we identify the following cases with regards to the plan π:

1. We say that π is fully-observable (FO) if every action is observed.
2. We say that π is partially-observable (PO) if some actions are missing in the learning example.
3. We say that π is non-observable (NO) if all actions are missing. This is a special case of partial observability.

Now we are ready to examine the most recent and relevant approaches to Action Model Learning found in the literature. As said above, approaches will be examined according to the type of learning examples accepted by the system. We also include in this analysis our planning-based approach, named Flexible Action Model Acquisition (FAMA). Table 7.1 summarizes the types of learning examples accepted by each approach, including also the expresiviness of the learned models and the principal technique used by each approach. The first column of Table 7.1 shows the learning examples accepted by the approach according to the observability (FO, PO or NO) of the state trajectory and action sequence. With that regard, we note that the task of learning from less constrained learning examples subsumes learning from more constrained ones. Consequently, approaches

to learning from, say learning examples with NO states, will also be able to learn from learning examples with PO states. All the approaches analyzed here accept learning examples with FO actions and PO states, with some of them being able to learn also from NO states. Exceptionally, LOCM is the only approach capable of learning without any state information (the state variables are also unknown to the system), and AMAN and NOISTA admit some form of noise (denoted with an asterisk in the table). The expressiveness of the learned action models (second column of Table 7.1) varies across approaches. All the presented systems are able to learn action models in a STRIPS representation (Fikes & Nilsson, 1971) and some propose algorithms to learn more expressive action models that include quantifiers, logical implications or the type hierarchy of a PDDL domain.

	Learning examples	**Learned action model**	**Technique**
ARMS	NO states FO actions	STRIPS	MAX-SAT
SLAF	PO states FO actions	quantifiers in effects	logical inference SAT solver
LAMP	PO states FO actions	quantifiers logical implications	Markov logic networks
AMAN	NO states FO* actions	STRIPS	graphical model estimation
NOISTA	PO* states FO actions	STRIPS	classification STRIPS rules derivation
CAMA	PO states FO actions	STRIPS	crowdsourcing annotation MAX-SAT
LOUGA	NO states FO actions	STRIPS negative preconditions	Genetic algorithm
LOCM2	— FO actions	predicates and types	Finite State Machines
FAMA	NO states NO actions	STRIPS	compilation to planning

Table 7.1: Characteristics of action-model learning approaches

We want to bring attention to the fact that all learning approaches with the exception of FAMA assume that a learning example encompasses a fully observed sequence of the actions; i.e, they assume all the actions performed by the agent are observable. In contrast, FAMA is able to learn from an incomplete or empty sequence of actions, and so the minimum observability case acceptable by FAMA is when the algorithm is only fed with an initial and final state of a trajectory (NO states and NO actions).

Let us now discuss why having FO actions is such a common assumption in Action Model Learning. The quick answer is that knowing all the actions means that the length of the underlying trajectory is also known. On the other hand, when the learning examples contain PO states and actions, the length of the trajectory is unknown and unbounded due to gaps in the observation sequence. Next, we have a closer look at these two scenarios

and their impact in the learning task:

1. **The learning example determines the length of the underlying trajectory**. This happens when the learning examples contain: (1) a FO action sequence; or (2) a FO state trajectory; or (3) a PO state trajectory where every state is at least partially observed.

    - The common assumption of having FO action sequences in the learning examples, as is the case of all the learning approaches of Table 7.1 except for FAMA, is unrealistic in many domains as it commonly implies the existence of human observers that annotate the observed action sequences. In some real-world applications, the observed and collected data are sensory data (e.g., home automation, robotics) or images (e.g. traffic) and one cannot rely on human intervention for labeling actions.
    - The assumption of having FO state trajectories means that the sensors are able to capture every state change at every instant, which is also typically unrealistic. Normally, the process of obtaining state feedback from sensors (or the processing of the sensor readings) is associated with a given sampling frequency that misses intermediate data between two subsequent sensor readings.
    - The assumption of having PO state trajectories where every state is at least partially observed (no missing states) seems more appropriate to reflect a real-world sensor reading but still requires that every state traversed by the actor is captured by the sensors.

    In this scenario the learning problem is **SAT compilable**, and it is known that a Boolean satisfiability problem is a NP-complete task (Cook, 1971). This is the reason why SAT solvers are commonly used in the approaches presented in Table 7.1. Furthermore, if the initial state of the trajectory is known and actions are fully-observable, the underlying trajectory can be computed in linear time by executing the given sequence of actions, starting from the given initial state. This is the reason why many learning systems in the literature impose such restrictive assumptions on the learning examples. We also note that, when the learning example contains a FO action sequence and a FO state trajectory, learning STRIPS action models is straightforward (Jiménez et al., 2012). In this case the *pre-* and *post-states* of every action are available and so action effects are derived lifting the literals that change between the pre and post-state of the corresponding action executions. Likewise preconditions are derived lifting the minimal set of literals that appears in all the pre-states of the corresponding action. The challenge in this case comes from computing the least number of examples that are necessary to learn models within a given error rate (Juba et al., 2021; Stern & Juba, 2017).

2. **The learning example does not identify the length of underlying trajectory**. This happens when learning examples contain PO or NO actions and states. In this

case, we are unaware of the number of actions of π and the number of states in τ. This gives rise to a completely different scenario and a more challenging learning task that brings one key difference: the transition between two given observed states may now involve more than one action and so **the length of τ is no longer known**. This justifies the use of **planning techniques** for solving the learning task, which now requires finding a model and an explanation to the learning example with that model. SAT approaches, on the other hand, are no longer straightforwardly applicable given the lack of a length bound for the underlying trajectory. In this particular scenario, the number of trajectories consistent with a learning example is also unbounded and grows exponentially with the actual length of the trajectory (that is now unknown). Therefore, when we assume partial observability in both actions and states, a learning approach must consider that a learning example is not an indication of the actual length of the plan. This motivates and justifies the use of planning since, as we have seen throughout this dissertation, it is able to find explanations for observations that do not convey the length of the underlying trajectory.

7.2. The Action Model Learning Problem

Action Model Learning is similar to Model Recognition in the sense that both problems pursue to identify the action model of the actor. The main difference lies in that in Model Recognition a set of candidate models to choose from is explicitly available, while in learning the model has to be estimated from a set of learning examples. The usual formulation of Action Model Learning, and the one we follow, assumes that the state variables and actions of the actor are known, and the free parameters to estimate are the applicability and effects constraints of the actions. Talking at the schematic level, this is equivalent to knowing the predicates of the model as well as the headers of the operators, and having to learn the precondition, add, and delete lists of the operators.

Action model learning uses several learning examples collected from different episodes of the actor walking through its state space in order to estimate the action model of the actor. Following this notion, a learning example can be formalized as a pair $\langle s_0, \omega \rangle$ such that ω is the observation sequence that stems from the trajectory $\tau = s_0[\![\pi]\!]$ traversed by some actor agent following plan π. We have already talked at length in the previous section that state-of-the-art approaches commonly assume that ω contains full information about the actions in π and partial information about the intermediate states of τ. We, however, impose no such assumptions and our understanding of observation sequence is the same as followed throughout this dissertation.

An Action Model Learning problem takes as input an *unspecified* action model $M_a[] = \langle X, A, c \rangle$ that conveys the state variables and actions of the actor but not their applicability or effects, a set of learning examples $\mathcal{L}$, and the sensor model M_s used to capture them.

Definition 42 (Action Model Learning Problem)**.** *An* Action Model Learning *problem is*

a tuple $\langle M_a[], M_s, \mathcal{L}\rangle$ *where:*

- $M_a[] = \langle X, A, c\rangle$ *is an* empty *action model,*
- $M_s = \langle X, Y, \mathsf{Sense}\rangle$ *is the sensor model of the observer.*
- $\mathcal{L}$ *is a set of learning examples.*

The goal of Action Model Learning is to estimate the App and Eff constraints which are left undefined in the empty action model $M_a[] = \langle X, A, c\rangle$. We define the solution of the Action Model Learning problem as the pair $\theta = \langle \mathsf{App}, \mathsf{Eff}\rangle$ such that $M_a[\theta] = \langle X, A, \mathsf{App}, \mathsf{Eff}, c\rangle$ is the action model that best explains the learning examples in $\mathcal{L}$.

Definition 43. *The solution to an Action Model Learning problem problem* $\langle M_A[], M_s, \mathcal{L}\rangle$ *is the pair* $\theta = \langle \mathsf{App}, \mathsf{Eff}\rangle$ *minimizes the accumulated cost of the best explanations generated by* $M_a[\theta]$ *for the learning examples in* $\mathcal{L}$*:*

$$\theta^* = \arg\min_{\theta} \left\{ \sum_{\langle s_0, \omega\rangle \in \mathcal{L}} \min_{\tau \in \mathcal{T}_\omega(s_0, M_a[\theta])} synCost_{M_a[\theta]}(\tau) \right\} \tag{7.1}$$

where App *and* Eff *belong to* Constr(X).

It is worth noting that, unlike the Model Recognition problem, the space of solutions in Action Model Learning is not given explicitly, but knowing the state variables X and actions A means that we know that $M_a[\theta]$ belongs to the space of action models $\mathcal{M}(X, A)$ for any θ that solves the problem. In other words, the empty action model $M_a[]$ implicitly defines the solution space of the learning problem. This is even more important if we restrict our attention to STRIPS-like action models since in that setting the space of action models, and therefore of solutions, is finite and enumerable (although exponential in the number of variables).

7.3. The Solution to the Action Model Learning Problem

This section motivates our definition of solution to an Action Model Learning problem (Definition 43). An Action Model Learning problem $\langle M_a[], M_s, \mathcal{L}\rangle$ is a learning problem over the state-space model described by the action model M_a and the sensor model M_s, where the free parameters θ to estimate are the applicability (App) and effects (Eff) constraints of the actor. The set $\mathcal{L}$ consists of learning examples of the form $\langle s_0, \omega\rangle$ used for the estimation.

The most common criterion in learning is Maximum-likelihood (ML) which computes

$$\theta^* = \arg\max_{\theta} P(\mathcal{L} \mid M_a[\theta], M_s) \tag{7.2}$$

where the likelihood of the set of learning examples is

$$P(\mathcal{L} \mid M_a[\theta], M_s) = \prod_{\langle s_0, \omega \rangle \in \mathcal{L}} P(\omega \mid s_0, M_a[\theta], M_s) \tag{7.3}$$

Alternatively, there is a different approach that instead of maximizing the likelihood of the learning examples, maximizes the probability of the most likely hidden state sequence (explanation).

$$\theta^* = \arg\max_{\theta} \prod_{\langle s_0, \omega \rangle \in \mathcal{L}} \max_{\tau \in \mathcal{T}(s_0, \omega)} P(\tau \mid \omega, s_0, M_a[\theta], M_s) \tag{7.4}$$

The differences between both approaches have been studied extensively (Allahverdyan & Galstyan, 2011), specially in the context of HMMs where the latter approach is known as Viterbi Training (VT). In general, VT converges faster but is less accurate than the ML approach. Still, both approaches are widely used and each one has scenarios where they have the edge over the other. In our formulation of the Action Model Learning problem we follow the VT approach as it has been shown to produce sparser (simpler) models, i.e., models with fewer transitions, resulting in an Occam's razor effect for learning.

Going back to Equation (7.4), recall from Section 4.3 that

$$\max_{\tau \in \mathcal{T}(s_0, \omega)} P(\tau \mid \omega, s_0, M_a[\theta], M_s) = \max_{\tau \in \mathcal{T}(s_0, \omega)} P(\tau \mid s_0, M_a[\theta]) P(\omega \mid \tau, M_s) \tag{7.5}$$

where $P(\tau \mid s_0, M_a[\theta])$ is the synthesis probability and $P(\omega \mid \tau, M_s)$ the sensing probability. Then, assuming a uniform distribution for observations, we arrive at the following estimator

$$\theta^* = \arg\max_{\theta} \prod_{\langle s_0, \omega \rangle \in \mathcal{L}} \max_{\tau \in \mathcal{T}(s_0, \omega)} P(\tau \mid s_0, M_a[\theta]) \tag{7.6}$$

which is analogous to the cost-driven interpretation of solution that we provide in Definition 43.

7.4. Action Model Learning as Planning

In this section we present FAMA, a planning-based approach for Action Model Learning focused on the learning of Strips-like action models in schematic form. The main idea behind FAMA is to interpret the learning problem as an editing problem, by exploiting the alphabet for Strips action models presented in Section 6.5.1. This allows FAMA to find a solution to the Action Model Learning problem $\langle M_a[], M_s, \mathcal{L} \rangle$ by editing any model in $\mathcal{M}(X, A)$ at no cost so that it generates the best explanations for $\mathcal{L}$.

FAMA compiles an Action Model Learning problem $\langle M_a[], M_s, \mathcal{L}\rangle$ into a planning problem $P_{M_a}(\mathcal{L})$ following a special case of the compilation presented in Section 6.5.2 that was used to build the compiled problem $P_{M_a}(\omega)$. The main characteristics of this special case are:

- M_a is the action model whose schematic representation contain empty precondition, add and delete lists.
- The set of edit actions A^Λ only contain actions encoding the insert edit operation.
- The learning examples $\mathcal{L}$ are serialized by concatenating them.

We choose as starting model M_a the *empty action model*, which has no preconditions nor effects. At the schematic level, this means that the precondition, add and delete lists of all its operators are empty. More formally, this is expressed as follows: assuming Λ is the alphabet for the space of action models $\mathcal{M}(X, A)$, M_a is the model induced by the valuation m such that $m(\lambda) = \bot$ for all $\lambda \in \Lambda$. There are two reasons for this choice, one theoretical and one practical:

- The theoretical reason is that we favor simple models that feature sparse operators with few preconditions and effects, as was already discussed when we justified our solution to the Action Model Learning problem (Section 7.3). Therefore, starting from a model with no precondition nor effects is a closer starting point to a desired solution.
- The practical reason is that if the operators of the starting model are empty, the insertion operation is the only edit operation that is needed to reach any other action model in the solution space. This means that we can remove the delete actions from the set of edit actions A^Λ thus reducing the branching factor of the problem. Arguably, the same benefit can be gained by starting from a "full" action model that has every schematic variable as precondition and effect, but action models are generally closer to the empty model than to the full one.

Serializing the learning examples makes it so that a solution plan needs to explain all learning examples one after another. This requires that, after a learning example has been explained, the state variables X are reset to the values of the initial state of the next learning example. In order to implement this, a small modification is required in the sensing actions associated to the last observation of each learning example. Let $\mathcal{L} = \{\langle s_0^1, \omega^1\rangle, \ldots, \langle s_0^k, \omega^k\rangle\}$ and assume, for notational simplicity, that all observation sequences are of the same length, i.e, $\omega^j = (o_1^j, \ldots, o_t^j)$ where $1 \leq j \leq k$. The effects constraint of the sensing actions $a^{o_t^j}$ with $1 \leq j < k$ are now defined as follows

$$\mathsf{Eff}(a^{o_t^j}) : \langle \mathtt{q}^+, \text{"}\mathtt{q}^+ = \mathtt{t}\text{"}\rangle \cup s_0^{j+1}$$

so that the execution of $a^{o_i^j}$ sets the initial state for the next learning example.

Interestingly, FAMA supports a grey-scale of prior information regarding the input model $M_a[]$, from an unspecified action model where nothing is known about its preconditions and effects to a fully-specified one where everything is known. This gray-scale can be generalized by looking at the subset $\Lambda' \subseteq \Lambda$ of alphabet variables whose values are actually known. Then, when $\Lambda' = \emptyset$ the input model is unspecified, when $\Lambda' = \Lambda$ we have a fully-specified input model, and everything in-between is a partially specified model. Supporting this grey-scale is straightforward, and only requires disabling all edit actions associated to the subset Λ' of alphabet variables whose value is already known.

As an example, consider an Action Model Learning problem $\langle M_a[], M_s, \mathcal{L} \rangle$ such that $M_a[]$ is a partially specified action model of the 4-operator *blocksworld* domain, such that the semantics of `pickup`, `putdown` and `unstack` are already known. The set $\mathcal{L}$ contains a single learning example corresponding to a partial observation of the execution of the four-action plan $\pi = \langle \texttt{unstack(B, A)}, \texttt{putdown(B)}, \texttt{pickup(A)}, \texttt{stack(A, B)} \rangle$ for inverting a two-block tower. Specifically, the learning example only conveys:

- The initial state of the trajectory, $s_0 = \{\texttt{on(B, A)}, \texttt{clear(B)}, \texttt{ontable(A)}\}$.
- The second and fourth actions of the plan, $(\texttt{putdown(B)}, \texttt{stack(A, B)})$
- The final state of the trajectory, $s_4 = \{\texttt{on(A, B)}, \texttt{clear(A)}, \texttt{ontable(B)}\}$

Therefore, this learning example falls under the category of NO state trajectory (since no intermediate states are observed), and PO plan (since half the actions are unknown). Figure 7.1 shows a solution plan to the planning problem $P_{M_a}(\mathcal{L})$ compiled from this example. Plan steps $01 - 02$ insert the preconditions of the `stack` operator, steps $03 - 07$ insert the effects, step 08 freezes the alphabet variables with the `commit` action, and steps $09 - 13$ explain the learning example. As we can see in this example, extracting the (schematic) action model from the solution plan is a straightforward process that only requires updating the alphabet Λ following the insert action in the solution plan.

The logical inference process our approach is based on has trouble learning preconditions that do not appear as negative effects since in this case no change is observed between the pre-state and post-state of an action. This is specially relevant for static state variables that never change and, hence, only appear as preconditions in the actions. In order to address this shortcoming and complete the list of learned preconditions, we apply a post-process based on the one proposed in (Kucera & Barták, 2018). The idea lies in going through every action and counting the number of cases where a state variable is present before the action is executed and the number of cases where it is not present. If a state variable is present in all the cases before the action, it is considered to be a precondition. In order to obtain a complete state trajectory, the proposal in (Kucera & Barták, 2018) applies the sequence of actions of the learning example and infers the preconditions from this FO action sequence. In our case, since the sequence of actions of the learning example might not be fully observable, we use the explanation found by the solution plan.

```
01 :  (insert_stack_pre_holding-v1)
02 :  (insert_stack_pre_clear-v2)
03 :  (insert_stack_eff_clear_v1)
04 :  (insert_stack_eff_clear-v2)
05 :  (insert_stack_eff_handempty)
06 :  (insert_stack_eff_holding-v1)
07 :  (insert_stack_eff_on-v1-v2)
08 :  (commit)
09 :  (unstack blockB blockA)
10 :  (putdown blockB)
11 :  (pickup blockA)
12 :  (stack blockA blockB)
13 :  (sense_1)
```

Figure 7.1: Example of a solution plan for the compiled problem $\mathcal{P}_{M_a}(\mathcal{L})$

7.5. Evaluation of action models

An important topic in Action Model Learning is the evaluation method used to assess the quality of the solutions found by the learning approach. In this section we discuss the metrics commonly used in the literature and motivate the use of two standard syntactic metrics (*precision* and *recall*). Then, Section 7.5.1 will define these metrics for action models, and Section 7.5.2 will introduce a semantic evaluation measure that builds upon *precision* and *recall*.

Table 7.2 summarizes the main characteristics of the evaluation of the learned action models based on the type of evaluation method (first column of Table 7.2), the metrics used in the evaluation (second column of Table 7.2) and the number of tested domains alongside the size of the training dataset (third column of Table 7.2).

As we can see in Table 7.2, the most common method is to use a syntax-based evaluation that compares the learned model against the Ground-Truth Model (GTM), that is, the model that generated the learning examples. The large majority of approaches use a similar syntax-based metric that consists in (1) counting the missing and extra schematic variables that appear in the learned model wrt the GTM and (2) normalizing this error by the total number of all the possible preconditions and effects of an action model. This is an *optimistic* metric since error rates are not normalized by the size of the actual GTM. The set of preconditions and effects of the GTM is usually smaller than the set of all possible preconditions and effects and thereby it turns out that these syntax-based metrics may output error rates below 100% for totally wrong learned models.

In order to overcome this limitation we propose the use of two standard metrics in Machine Learning, *precision* and *recall*, that are frequently used in pattern recognition, information retrieval and binary classification (Davis & Goadrich, 2006). These two syn-

	Evaluation method	Metrics	#tested domains/ training data size
ARMS	cross-validation with a test set of plan traces	error counting of #pre satisfaction and redundancy	6 1,600-4,320 actions (160 plan traces)
SLAF	manual checking wrt GTM	—	4 1,000 actions
LAMP	checking wrt GTM	error counting of extra and missing #pre and #eff	4 1,300-6,100 actions (100-200 plan traces)
AMAN	checking wrt GTM	error counting of extra and missing #pre and #eff	3 40-200 plan traces
NOISTA	checking wrt GTM	error counting of extra and missing #pre and #eff	5 5,000-20,000 actions
CAMA	checking wrt GTM	error counting of extra and missing #pre and #eff	3 15-75 plan traces
LOUGA	cross-validation with a test set of plan traces	redundant effects differences wrt the test set	5 800 - 3200 actions (160 traces)
LOCM2	manual checking wrt GTM	—	—
FAMA	checking wrt GTM validation with a test set	precision and recall	15 20-50 actions

Table 7.2: Evaluation of action models (GTM: ground-truth model)

tactic metrics are generally more informative than counting the number of errors between the learned action models and the GTM:

- $Precision = \frac{tp}{tp+fp}$, where tp is the number of *true positives* and fp is the number of *false positives*. In our particular case, true positives are schematic state variables that correctly appear in the (schematic) action model, and true negatives are schematic state variables of the learned model that should not appear.
- $Recall = \frac{tp}{tp+fn}$, where fn is the number of *false negatives* (schematic state variables that should appear in the learned model but are missing).

7.5.1. Syntactic-based precision and recall for action models

The rationale behind the adaptation of precision and recall for action models lies in counting the *edit operations* that needs to be applied to the learned action model in order to transform it into the GTM.

We now provide formal definitions of $INS(\breve{M}_a, \breve{M}'_a)$ and $DEL(\breve{M}_a, \breve{M}'_a)$, the sets of insertions and deletions, respectively, that are needed to transform an action model $\breve{M}_a$ into the reference model $\breve{M}'_a$.

Definition 44. *Let pre(ă), add(ă), and del(ă) be the precondition, add, and delete lists of an operator ă. We define:*

$$INS(\breve{M}_a, \breve{M}'_a) = \bigcup_{\substack{\breve{a},\breve{a}' \in \breve{A} \times \breve{A}' \\ head(\breve{a}) = head(\breve{a}')}} pre(\breve{a}') \backslash pre(\breve{a}) \cup add(\breve{a}') \backslash add(\breve{a}) \cup del(\breve{a}') \backslash del(\breve{a})$$

$$Del(\breve{M}_a, \breve{M}'_a) = \bigcup_{\substack{\breve{a},\breve{a}' \in \breve{A} \times \breve{A}' \\ head(\breve{a}) = head(\breve{a}')}} pre(\breve{a}) \backslash pre(\breve{a}') \cup add(\breve{a}) \backslash add(\breve{a}') \cup del(\breve{a}) \backslash del(\breve{a}')$$

With these ingredients in mind, we adapt the definitions of syntactic precision and recall to schematic action models. Let $\breve{M}_a = \langle P, V, \breve{A} \rangle$ be a schematic action model and let $\breve{M}'_a$ be the GTM. We know that $size(\breve{M}_a) = \sum_{\breve{a} \in \breve{A}} |pre(\breve{a})| + |add(\breve{a})| + |del(\breve{a})|$ and by definition the number of preconditions (elements in the Pre list) and effects (elements in the Add and Del lists) of the operators in $\breve{M}_a$ is equal to the sum of *true positives* and *false positives*; that is, $size(\breve{M}_a) = tp + fp$.

The number of *deletions* required to transform $\breve{M}_a$ into $\breve{M}'_a$ ($\left|DEL(\breve{M}_a, \breve{M}'_a)\right|$) matches our previous definition of the number of *false positives*; and $\left|INS(\breve{M}_a, \breve{M}'_a)\right|$, the number of *insertions* required to transform $\breve{M}_a$ into $\breve{M}'_a$, corresponds to the number of *false negatives* of $\mathcal{M}$. Then we can affirm that $size(\breve{M}'_a) = size(\breve{M}'_a) - \left|DEL(\breve{M}_a, \breve{M}'_a)\right| + \left|INS(\breve{M}_a, \breve{M}'_a)\right|$.

Definition 45. *The precision of $\breve{M}_a$ relative to the GTM is defined as the fraction of the common preconditions and effects between $\breve{M}_a$ and the GTM among all preconditions and effects of $\breve{M}'_a$.*

$$Precision = \frac{tp}{tp + fp} = \frac{size(\breve{M}_a) - \left|DEL(\breve{M}_a, GTM)\right|}{size(\breve{M}_a)}$$

Definition 46. *The recall of $\breve{M}_a$ relative to the GTM is defined as the fraction of the common preconditions and effects between $\breve{M}_a$ and the GTM among all preconditions and effects of the GTM.*

$$Recall = \frac{tp}{tp + fn} = \frac{size(\breve{M}_a) - \left|DEL(\breve{M}_a, GTM)\right|}{size(\breve{M}_a) - \left|DEL(\breve{M}_a, GTM)\right| + \left|INS(\breve{M}_a, GTM)\right|}$$

Intuitively, precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models. We interpret a sound learned model as one in which all preconditions and effects are correct with respect to the GTM, and so there is no need to remove anything. A complete model is one in which no precondition or effect is missing. As an example, a precision of 0.5 means that only half of the predicates that make up the learned domain model are present in the GTM, while a recall of 0.5 means that only half of the predicates that make up the GTM are present in the learned domain model.

7.5.2. Semantic-based precision and recall for action models

Pure syntax-based evaluation metrics can report low scores for learned models that are actually *sound* and *complete* but syntactically different from the GTM. Semantic evalua-

tion metrics add a distinctive value over the syntactic ones, which is that they evaluate the learned model with respect to a set of *validation* examples. These metrics measure how well a model can explain a given example, so the use of the word "semantic" here is in reference to the degree to which the learned model is able to capture the physics of the domain.

Semantic metrics alleviate two important limitations of a purely syntax-based assessment: (a) that the learned model is syntactically different from the reference model but semantically correct and (b) that the learned model comprises correct though unnecessary preconditions in regards to the reference model. This latter issue is concerned with the qualification problem, which is defined as the actual impossibility of listing all the preconditions required for a real world action to have its intended effects (Ginsberg & Smith, 1988). The use of semantic metrics is appropriate in scenarios where:

1. The GTM is unknown. This is the most common scenario in Machine Learning, where models are both learned and evaluated with respect to datasets.

2. We are interested in measuring the ability of a model to explain a given example, which is a good indicator of how the model will perform in actual planning tasks. As a rule of thumb, it is preferable to evaluate the learned models wrt a dataset because a learned model can be semantically correct though syntactically incorrect (different from the GTM). We refer to this phenomenon as *model reformulation*.

An example of *model reformulation* is the swapping of the roles of two *comparable* operators. Consider, for example, a situation where the *blocksworld* operator `stack` could be *learned* with the preconditions and effects of the `unstack` operator, and vice-versa, because they are comparable. On the contrary, this reformulation will not happen between the `stack` and `pickup` because they are not comparable. In the same way, the roles of two parameters can also be swapped (e.g., interchanging the roles of the two parameters of the operator `stack` or the operator `unstack`) and yet the learned models would be semantically correct with respect to the given input observations. A more complex kind of reformulation occurs when two or more operators are learned in a single *macro-action*. These semantic alterations typically appear in the learned models when the observability is low and the learning examples provide little data.

The ARMS system was the first to show that a semantic evaluation can be done via validation of a set of examples with the learned model (Yang et al., 2007). The underlying idea is that an error indication of the learned action models is obtained by counting the number of preconditions that are not satisfied during the execution of the learned example with the learned models, similarly to the functionality provided by the automatic validation tool VAL (Howey et al., 2004) used in the IPCs. This approach can be understood as modifying the plan trace (by adding the necessary preconditions to the intermediate states) so as to allow the execution of the observed actions using the learned models. In other words, modifying the example to fit the model.

Inspired by this approach, we present novel semantic-based metrics that builds upon the *precision* and *recall* metrics. The intuition behind these metrics is to *semantically* assess how well the learned model $\breve{M}_a$ explains a set examples according to the amount of *editing* required by $\breve{M}_a$ to explain the examples.

Definition 47. *Given a schematic action model $\breve{M}_a$, and a set of validation examples $\mathcal{V}$, the* **closest consistent schematic action model***, $\breve{M}^*_a$, is the comparable schematic action model closest to $\breve{M}_a$ (in terms of edit operations) that is able to explain the examples in $\mathcal{V}$;*

$$\breve{M}^*_a = \underset{\forall \breve{M}'_a \rightarrow \mathcal{V}}{\arg\min} \left| INS(\breve{M}_a, \breve{M}'_a) \cup DEL(\breve{M}_a, \breve{M}'_a) \right|$$

The closest consistent schematic action model $\breve{M}^*_a$ allows us to define a semantic version of *precision* and *recall* following definitions 45 and 46.

$$\textit{sem-Precision} = \frac{size(\breve{M}_a)) - \left|DEL(\breve{M}_a, \breve{M}^*_a)\right|}{size(\breve{M}_a)}$$

$$\textit{sem-Recall} = \frac{size(\breve{M}_a) - \left|DEL(\breve{M}_a, \breve{M}^*_a)\right|}{size(\breve{M}_a) - \left|DEL(\breve{M}_a, \breve{M}^*_a)\right| + \left|INS(\breve{M}_a, \breve{M}^*_a)\right|}$$

Proposition 4. *When the closest consistent schematic action model $\breve{M}^*_a$ is the GTM, the syntactic and semantic evaluation of a learned model $\breve{M}_a$) return the same values; that is, Precision = sem-Precision and Recall = sem-Recall.*

The interpretation of a sound and complete model in the semantic perspective is slightly different from the syntactic one. In this case, a sound model is one were there is no need to remove any precondition or effect in order to be consistent with some given plan traces. A complete model is one that can explain the validation examples without adding any new preconditions or effects. Unlike the semantic metric defined by ARMS, our novel semantic definitions of precision and recall are not sensitive to flaws in the action model that manifest more than once in the validation examples since the flaws are corrected only once in the learned models instead of at every intermediate state of the examples.

Computing Semantic Metrics

As we saw when we presented FAMA, our compilation accepts a grey-scale of input models and can be tweaked to disabled some edit actions, blurring the line between learning and validation. For example, if the starting model is fully-specified and edit actions are disabled our compilation can be used to validate a learning example. On the other hand, if the edit actions are enabled, our compilation performs a *soft validation* that counts the number of edit operations required by the model to explain the example.

We can use this flexibility of FAMA to pose the problem of computing the closest consistent (schematic) action model as a learning problem $\langle M_a, M_s, \mathcal{V} \rangle$ where M_a is a previously learned action model, and $\mathcal{V}$ is a set of validation examples. If all edit actions are enabled in the compilation, an optimal solution to this problem will perform the least amount of edit operations M_a required to explain $\mathcal{V}$, thus returning M_a^*. Then, the schematic representation can be extracted from its alphabet and used to compute *sem-Precision* and *sem-Recall*.

7.6. Experimental Evaluation

This section presents several experiments to evaluate the performance of FAMA and the quality of the learned models. Whenever applicable, we will consider scenarios with both known and unknown plan horizon to draw conclusions at both levels of complexity.

After presenting the setup of the experiments in section 7.6.1, we introduce three experiments that measure different aspects like the minimal number of learning examples necessary to obtain good quality models (section 7.6.2), a comparison of the quality of models obtained by FAMA compared to ARMS (section 7.6.3) and an analysis of the quality of models when using a very small number of learning examples in section 7.6.4. In these three experiments, the quality of the models is measured using the syntactic-based precision and recall metrics presented in section 7.5.1, that is, we measure the precision and recall of the learned models with respect to the GTM for each domain. Finally, the syntactic evaluation of section 7.6.4 is compared against a semantic evaluation, i.e. how well the learned models are able to reproduce the input plan traces, in section 7.6.5.

7.6.1. Setup

We evaluate FAMA on 15 IPC domains that satisfy the STRIPS requirement (Fox & Long, 2003), all taken from the PLANNING.DOMAINS repository (Muise, 2016). Table 7.3 presents the features of the tested domains that affect the size of the planning problem $P_{M_a}(\mathcal{L})$ that results from the compilation. For each domain, the columns report, from left to right, the number of operators, the number of predicates, the maximum arity of the operators, and the maximum arity of the predicates.

The details of our experimental setup are the following:

- **Learning examples**. For each domain, we generated 10 trajectories, each with 10 actions and 10 intermediate states, using random walks. Depending on the experiment, these trajectories are used to generate examples used for training or testing purposes (more details on this issue are provided at the particular experiment).
- **Planner**. The classical planner we used to solve the instances of $P_{M_a}(\mathcal{L})$ that result from our compilations is MADAGASCAR (Rintanen, 2014). We used MADAGASCAR for

several reasons:

1. Other planners such as FastDownward were also tested but provided worse experimental results
2. The ability of Madagascar to deal with instances populated with dead-ends, such as our compiled problem $P_{M_a}(\mathcal{L})$, is very helpful (López, Celorrio, & Olaya, 2015).
3. A SAT-based planner like Madagascar is particularly suitable for tasks where the learning examples determine the horizon of the solution plan. In this case, a SAT-based planner solves the prefix of the solution plan in two time steps because the actions for inserting preconditions can be applied in parallel in a single time step and the same for the actions inserting the effects.

- **Hardware**. All experiments were run on an Intel Core i5 3.50 GHz x 4 with 16 GB of RAM.

	Domain features			
	# actions	**# predicates**	**max action arity**	**max predicate arity**
Blocks	4	5	2	2
Driverlog	6	5	4	2
Ferry	3	5	2	2
Floortile	7	10	4	2
Grid	5	9	4	2
Gripper	3	4	3	2
Hanoi	1	3	3	2
Miconic	4	6	2	2
Npuzzle	1	3	3	2
Parking	4	5	3	2
Rovers	9	25	6	3
Satellite	5	8	4	2
Transport	3	5	5	2
Visitall	1	3	2	2
Zenotravel	5	4	5	2

Table 7.3: Feature description of the domains used in the experiments.

7.6.2. Impact of the size of the set of learning examples

This experiment evaluates the impact of $|\mathcal{L}|$, the size of the set of learning examples, on the performance of FAMA in order to:

1. Identify the minimal amount of input knowledge required by FAMA to learn sound and complete models,

2. Evaluate the scalability of FAMA with respect to the number of learning examples.

The experiment analyzes the evolution of the CPU-time and the precision and recall of the learned models wrt the GTM as $|\mathcal{L}|$ increases from 1 to 10 learning examples. To keep the experiment practicable, we introduced a 1000s timeout, after which the learning process is killed and a score of 0 is given to both the precision and recall of the learned model. We defined two case studies:

- **FO action sequence and PO state trajectory (known plan horizon)**: This is the common case addressed by most of the state-of-the-art learning approaches, which corresponds to a scenario where the plan horizon is given by the action sequence. In this experiment we assume a degree of observability of only 10% for the state trajectory, meaning that each state variables of a state has a 10% chance of being observed.

- **NO action sequence and NO state trajectory (unknown plan horizon)**: In this case study, both the input action sequence and state trajectory are *empty* and the length of the plan is unknown. A learning example only contains the initial state and an observation of the final state.

Figures 7.2 and 7.3 show the quality of the models and computation time, respectively, for the case FO/PO. The values plotted in these figures are averages over the 15 domains. In Figure 7.2 we see that, after three learning examples, precision stabilizes at 0.84 whereas recall stabilizes at 0.95. These results show that FAMA does, in fact, not need of learning examples to learn sound and complete models as opposite to other approaches in the literature where models are learned using around 100 traces (see Table 7.2).

Figure 7.3 displays the scalability of FAMA. Interestingly, we can observe an exponential increase in computation time for learning sets beyond five learning examples. Up to four learning examples the computation time is below 1 sec but it reaches 166 secs when the input is composed of 10 learning examples. These results match the expected performance of MADAGASCAR since this planner is known to struggle with plan horizons beyond 150-200 steps (in our case 160 steps corresponds to 8 traces since each trace has 10 actions and 10 intermediate states).

Figure 7.4 displays the average precision and recall of the 15 learned models in the scenario with unknown plan horizon. As expected, the quality of the learned models is lower than when the horizon of the plan is known. The higher complexity of this setting is also reflected in the appearance of some timeouts when solving the learning task. The first timeout is found in the *grid* domain at $|\mathcal{L}| = 3$; the *floortile* times out with $|\mathcal{L}| = 4$; and by the time $|\mathcal{L}|$ reaches 10 the number of domains where no solution is found is 6, adding the *npuzzle, parking, rovers and zenotravel* domains. We can observe in Figure 7.4 the opposite behaviour to Figure 7.2; that is, we find a drop of the quality as the number of learning examples increases. The drop in the score is caused by the increasing

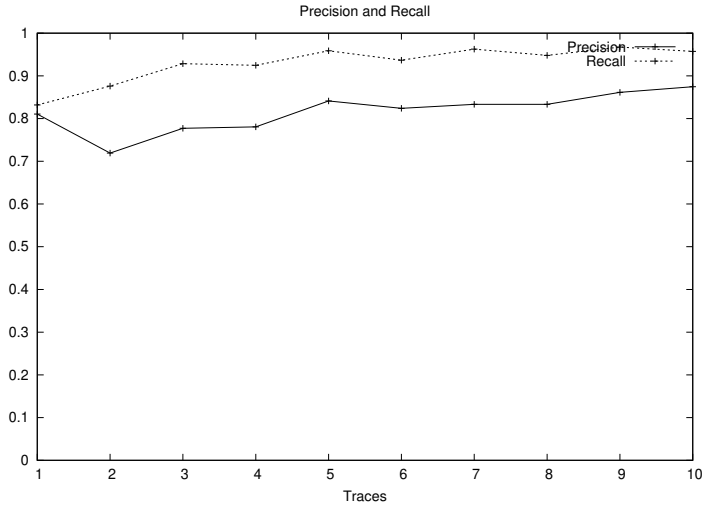

Figure 7.2: Precision and recall when learning from [1-10] learning examples with FO action sequences and PO state trajectories with 10% observability.

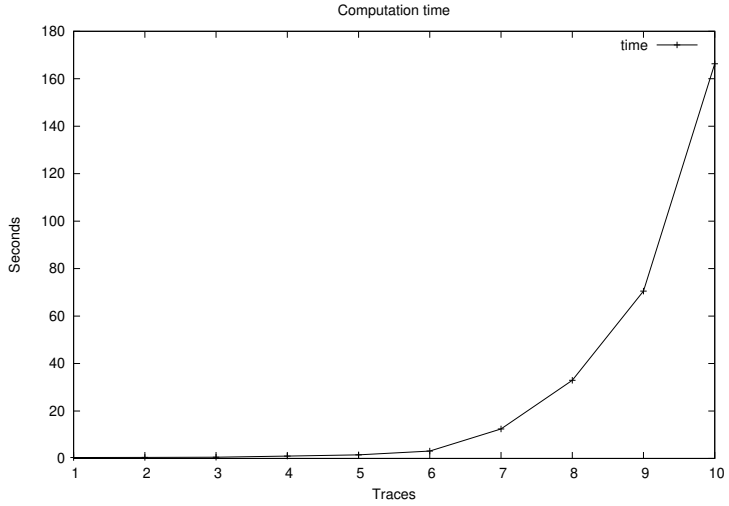

Figure 7.3: Computation time when learning from [1-10] learning examples with FO action sequences and PO state trajectories with 10% observability.

number of timeouts, meaning that no solution is found within the given time-bound, and consequently a value of 0 for precision and recall is assigned to these experiments. Figure 7.5, on the other hand, reflects that the computation time of the second case study is also higher than in the first case, which is explained by both the higher complexity and the large number of timeouts.

The conclusions we draw from these experiments is that, when the plan horizon is known, learning with few input samples yield action models whose operators contain 95% of the preconditions and effects of the GTM plus some extra ones as indicated by the values of precision and recall. With unknown plan horizons, on the other hand, the learned models are generally more different from the GTM; while timeouts are the main cause of the drop in the score, we must point out that pure syntax-based metrics are not adequate

to evaluate such under-constrained learning problems since the phenomenon of *reformulation* occurs and this largely impacts the results (we will provide experimental evidence of this in section 7.6.4). These results emphasize a relevant feature our approach: the small size of the training set required by FAMA in comparison with other approaches (see Table 7.2). Unlike extensive-data approaches, our work explores an alternative research direction to learn action models from a small number of learning examples. This is an important advantage, particularly in domains where it is costly or impossible to obtain a significant number of learning examples.

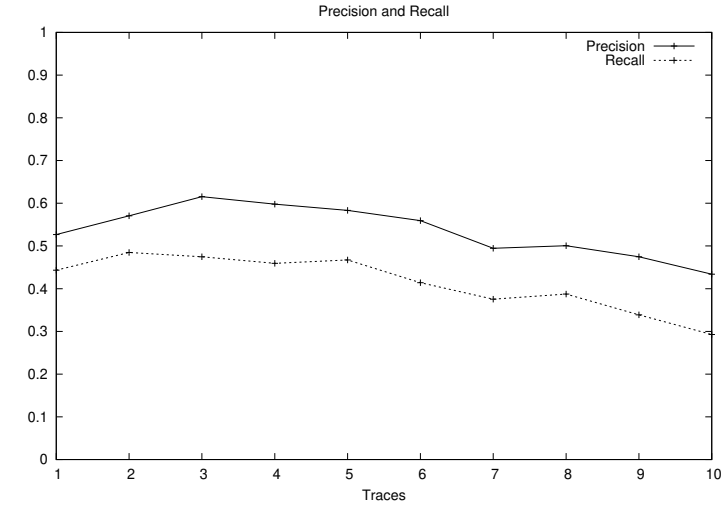

Figure 7.4: Precision and recall when learning from [1-10] learning examples with NO action sequences and NO state trajectories.

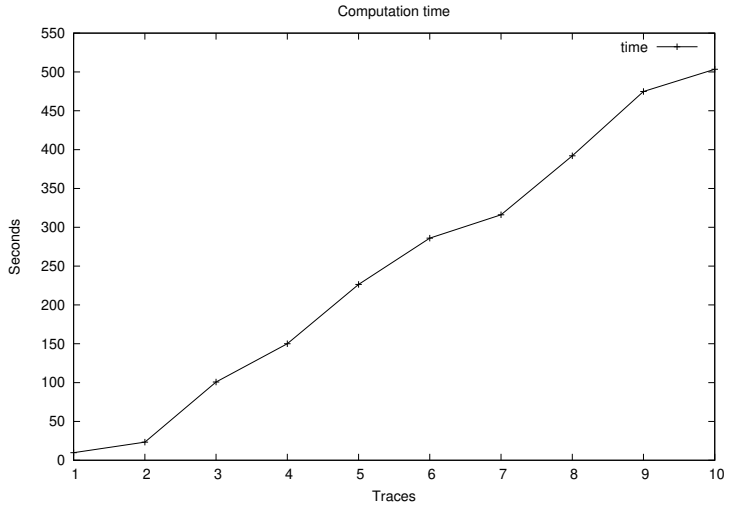

Figure 7.5: Computation time when learning from [1-10] learning examples with NO action sequences and NO state trajectories.

7.6.3. Comparison with ARMS

In this section we analyze the performance of FAMA compared to ARMS, one of the most well-known approaches to Action Model Learning. ARMS, as well as most of the existing current learning systems, works under the assumption of learning examples with FO action sequences and NO state trajectories and therefore is not able to handle the scenarios where the plan horizon is unknown. We will thereby restrict the experimentation to the cases manageable by ARMS.

In this experiment, we defined a *degree of observability* σ for the state trajectory, ranging from 0% to 100%, that measures the probability of observing a state variable, and evaluated both FAMA and ARMS for increasing values of σ using five learning examples. When $\sigma = 0$ we have a NO state trajectory, when $\sigma = 100$ we have a FO state trajectory and all cases in-between correspond to the PO scenario.

Figures 7.6 and 7.7 compare FAMA and ARMS in terms of precision and recall. The horizontal axes represent the degree of observability and vertical axes show the average precision (Figure 7.6) and recall (Figure 7.7) computed over the 15 tested domains. Remarkably, FAMA dominates in terms of precision in all cases except for the FO state trajectories. Particularly, the models learned by FAMA are between 13% to 34% more precise than those learned by ARMS. A similar trend is observed for recall (Figure 7.7), where the difference is even larger, meaning that our learned models are more complete.

The results highlight that FAMA outperforms ARMS when very few plan traces are available. This by no means is conclusive that FAMA is overall better in NP-complete scenarios but only that it is able to learn better with very limited input knowledge (actually, Figure 7.3 reflects the exponential behaviour of FAMA with more than five traces).

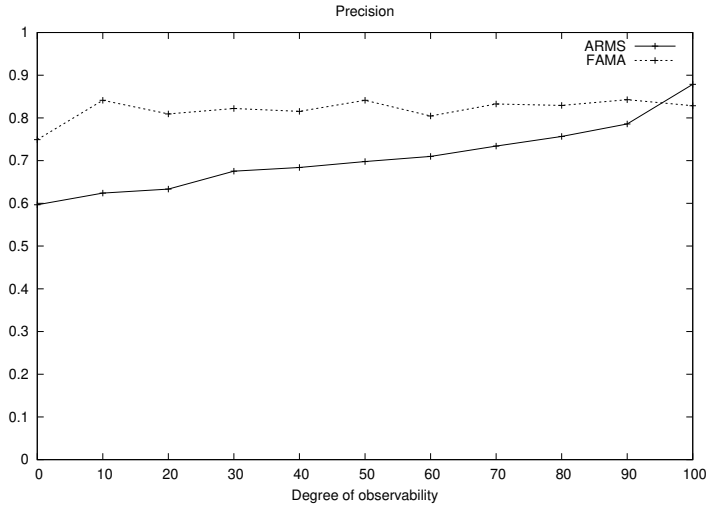

Figure 7.6: Precision comparison between FAMA and ARMS for different *degrees of observability*.

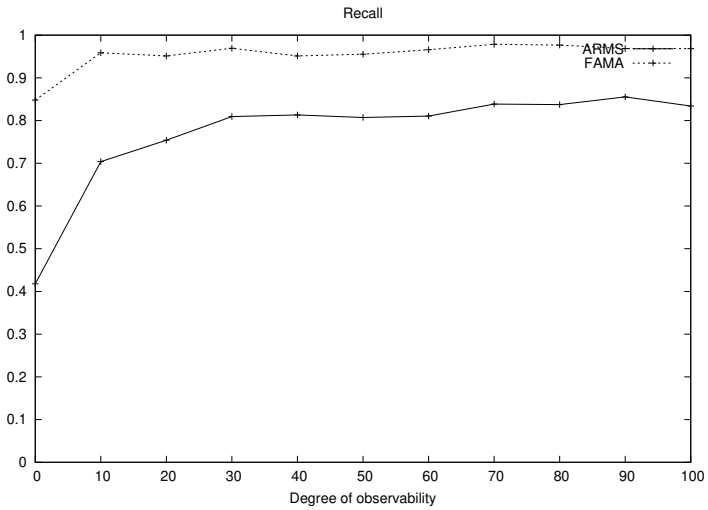

Figure 7.7: Recall comparison between FAMA and ARMS for different *degrees of observability*.

7.6.4. Learning with minimal input knowledge

In this section, we will take a closer look at the action models learned from a very small set of learning examples. To that end, we will limit the input to only two learning examples and analyze the results under different degrees of observability. We evaluate three case studies:

- **FO action sequence and PO state trajectory**: We are, once again, assuming a degree of observability of 10% for the state trajectory. Results of this case study are detailed in Table 7.4.
- **PO action sequence and PO state trajectory**: In this case study we are assuming a degree of observability of 30% for both the action sequence and state trajectory. Results are shown in Table 7.5.
- **NO action sequence and NO state trajectory**: Both the action sequence and state trajectory are completely empty so only the initial and final states are observed. Results of this case study are reported in Table 7.6.

All tables in this section (Tables 7.4, 7.5 and 7.6) follow the same structure. precision (**P**) and recall (**R**) scores are computed separately for the precondition (**Pre**), add (**Add**) and delete (**Del**) lists, and also globally (**Global**). The last column reports the computation time (in seconds) needed to obtain the learned models. Missing values in the tables (reported as -) correspond to domains where no solution was found within a 1800s timeout.

Table 7.4 shows the results of the case study FO/PO. Recall scores are generally higher than the precision ones, and, in fact, the models learned for six out of the 15 domains were

perfectly complete. Although precision is overall lower, it is interesting to notice that the learned negative effects are mostly flawless. With regards to the computation time, we can observe times are below one second in most cases except for some of the more complex domains.

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
Blocks	0.86	0.67	1.0	0.67	0.8	0.44	0.89	0.59	0.24
Driverlog	0.6	0.86	0.36	0.57	0.67	0.29	0.53	0.64	0.58
Ferry	0.7	1.0	0.36	1.0	1.0	1.0	0.6	1.0	0.37
Floortile	0.69	1.0	0.55	1.0	1.0	0.82	0.69	0.95	1.38
Grid	0.68	0.88	0.5	0.86	0.88	1.0	0.67	0.9	0.65
Gripper	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.18
Hanoi	0.67	1.0	1.0	1.0	1.0	1.0	0.8	1.0	0.36
Miconic	1.0	1.0	0.57	1.0	1.0	1.0	0.84	1.0	0.3
Npuzzle	0.75	1.0	1.0	1.0	1.0	1.0	0.88	1.0	0.26
Parking	0.78	1.0	0.69	1.0	1.0	1.0	0.8	1.0	0.24
Rovers	0.54	1.0	0.3	0.76	1.0	0.46	0.48	0.85	2.14
Satellite	0.93	1.0	0.56	1.0	1.0	0.75	0.81	0.96	0.4
Transport	0.83	1.0	0.5	1.0	0.6	0.6	0.67	0.9	0.19
Visitall	1.0	1.0	0.25	0.5	1.0	1.0	0.57	0.8	1.31
Zenotravel	0.9	0.64	0.5	0.71	0.83	0.71	0.73	0.68	0.25
	0.8	0.94	0.61	0.87	0.92	0.8	0.73	0.88	0.59

Table 7.4: Precision and recall scores for learning tasks with FO action sequences and PO state trajectories with 10% observability.

Table 7.5 gathers the results of the case study PO/PO with 30% observability. We can see in the table that the scores of some domains are missing. This is the case of *floor-tile* and *grid*, which not only are fairly complex domains, but also categorized as *puzzle-like* domains, a feature that is known for putting a strain in the planners. Interestingly enough, we note the high computation time of *hanoi* and *parking*, which also qualify as a *puzzle-like* domains. Regarding quality, we find that the learned models retain a level of soundness similar to Table 7.4 but the completeness is lower than in the previous case study. This is specially noticeable in the preconditions, where recall values drop from 0.88 to 0.64. This is because the actions observed in the learning examples act as strong constraints playing a key role on the closeness of the learned model to the GTM. The more actions that are missing in the input knowledge, the more likely the occurrence of reformulations.

We now analyze the case study with NO action sequences and NO state trajectories (Table 7.6). A first outstanding observation is that, contrary to what might be expected by looking at the previous table, we are able in this case to find solutions for all the domains. This happens because the search is less constrained and consequently there are far more possible solutions for this learning problem. This broader space of solutions is also stressed in a diminished quality of the learned models. Thus, despite the learned

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
Blocks	0.89	0.89	0.8	0.89	0.83	0.56	0.84	0.78	0.77
Driverlog	0.57	0.29	0.31	0.57	0.4	0.29	0.4	0.36	7.35
Ferry	0.83	0.71	0.36	1.0	1.0	1.0	0.62	0.87	3.38
Floortile	-	-	-	-	-	-	-	-	-
Grid	-	-	-	-	-	-	-	-	-
Gripper	1.0	1.0	0.8	1.0	1.0	1.0	0.93	1.0	0.17
Hanoi	0.67	0.5	1.0	1.0	1.0	1.0	0.86	0.75	132.69
Miconic	1.0	0.33	1.0	1.0	1.0	0.67	1.0	0.56	0.71
Npuzzle	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.86	13.48
Parking	0.83	0.36	1.0	0.89	0.83	0.56	0.9	0.56	160.64
Rovers	0.43	0.73	0.24	0.47	0.56	0.38	0.39	0.61	31.13
Satellite	0.6	0.21	0.56	1.0	0.5	0.5	0.56	0.43	11.55
Transport	1.0	0.2	0.57	0.8	1.0	0.4	0.73	0.4	23.39
Visitall	0.67	1.0	0.5	0.5	1.0	1.0	0.67	0.8	3.13
Zenotravel	1.0	0.36	0.4	0.29	1.0	0.43	0.77	0.36	226.27
	0.81	0.56	0.66	0.8	0.86	0.68	0.74	0.64	47.28

Table 7.5: Precision and recall when learning with PO action sequences and PO state trajectories, 30% observability in both cases.

models being able to explain the learning examples, they are further from the original GTM. In Table 7.6 we can observe the global values of precision and recall drop to 0.57 and 0.48, respectively.

We argue, however, that syntax-based metrics are not appropriate for scenarios with minimal observability as they cannot cope with the reformulations that frequently occur in these circumstances. To illustrate this, Figure 7.8 shows the PDDL encoding of the action model of the `stack` operator learned from learning examples with NO action sequences and NO state trajectories. This learned operator removes a block from on top of another block and puts it down on the table in a single step. There are two main differences with respect to the model of the `stack` operator of the GTM: (1) the learned action is actually *unstacking* a block instead of stacking it and (2) the block on the top ends on the table, not held by the robot arm. We refer to the first difference as *role swapping* and it happens when there are missing actions in the learning examples. If no actions are present in the input traces, the names of operators become meaningless, in which case the effectively anonymous actions can interchange their behaviour with any other comparable operator. The second difference indeed reveals that the learned action model is working as an `unstack+put-down` *macro-action*. This happens when there are missing states in the learning examples since a *macro-action* can be seen as the application of more than one action in a single step, thus skipping some intermediate states.

Reformulated action models, like the one in Figure 7.8, are indeed sound models that can be used to solve planning tasks. For instance, any *blocks-world* problem can be solved unstacking all the blocks to the table (`unstack+put-down`) and then stacking them

to meet the goal conditions (`pick-up+stack`). Hence, the NO/NO case study features all the conditions for reformulation to happen, and this is the reason why scenarios such as this one are better evaluated using *semantic-based metrics*.

	Pre		Add		Del		Global		
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**Time**
Blocks	0.6	0.67	0.33	0.22	0.67	0.44	0.55	0.44	0.26
Driverlog	0.5	0.29	0.33	0.57	0.0	0.0	0.38	0.29	0.88
Ferry	0.5	0.43	0.5	0.25	0.67	0.5	0.55	0.4	0.45
Floortile	0.48	0.45	0.27	0.36	0.46	0.55	0.41	0.45	58.38
Grid	0.25	0.24	0.33	0.43	0.14	0.14	0.25	0.26	234.63
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.86	0.16
Hanoi	0.6	0.75	1.0	1.0	1.0	1.0	0.78	0.88	6.32
Miconic	0.63	0.56	0.6	0.75	0.25	0.33	0.53	0.56	0.25
Npuzzle	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.52
Parking	0.57	0.29	0.2	0.11	0.8	0.44	0.53	0.28	17.43
Rovers	0.38	0.64	0.07	0.24	0.13	0.54	0.22	0.53	1.74
Satellite	0.86	0.43	0.43	0.6	0.75	0.75	0.67	0.52	3.15
Transport	0.29	0.2	0.38	0.6	0.67	0.4	0.39	0.35	1.06
Visitall	0.0	0.0	1.0	0.5	0.0	0.0	1.0	0.2	1.21
Zenotravel	0.4	0.29	0.25	0.29	0.2	0.14	0.3	0.25	17.48
	0.54	0.46	0.51	0.53	0.52	0.48	0.57	0.48	22.99

Table 7.6: Precision and recall scores for learning tasks with NO action sequences and NO state trajectories.

```
(:action stack
 :parameters (?o1 - object ?o2 - object)
 :precondition (and (on ?o1 ?o2)(handempty ))
 :effect (and (not (on ?o1 ?o2))(clear ?o1)(clear ?o2)(ontable ?o1)))
```

Figure 7.8: PDDL encoding of the learned *stack* operator from the four-operator *blocksworld* domain.

7.6.5. Syntactic versus semantic evaluation

Our last experiment is devoted to compare the scores provided by the syntactic and semantic versions of precision and recall. For that purpose, we will evaluate the models learned in Section 7.6.4 both syntactically, using the GTM, and semantically, using a validation set of five examples. Our goal with this experiment is to gauge the suitability of the semantic metrics proposed in section 7.5.2 with respect to their well-known counterparts. With that in mind, we define two case studies:

- **FO action sequence and PO state trajectory**: In this case study the full sequence of actions is known and no states are missing, which makes it practically impossible

for reformulated models to appear. In fact, in all our experimentation with FAMA and other approaches we never observed reformulations when the full sequence of actions is known.

- **NO action sequence and NO state trajectory**: This is a case study that favors reformulations in the learned models, as previously discussed.

	Precision	**Recall**	**sem-Precision**	**sem-Recall**
Blocks	0.89	0.59	0.89	0.64
Driverlog	0.53	0.64	0.71	0.83
Ferry	0.6	1.0	0.96	1.0
Floortile	0.69	0.95	0.97	0.95
Grid	0.67	0.9	0.95	0.98
Gripper	1.0	1.0	1.0	1.0
Hanoi	0.8	1.0	0.9	1.0
Miconic	0.84	1.0	1.0	1.0
Npuzzle	0.88	1.0	1.0	1.0
Parking	0.8	1.0	0.98	1.0
Rovers	0.48	0.85	0.94	0.99
Satellite	0.81	0.96	0.93	1.0
Transport	0.67	0.9	1.0	1.0
Visitall	0.57	0.8	1.0	1.0
Zenotravel	0.73	0.68	0.85	0.88
	0.73	0.88	0.94	0.95

Table 7.7: Syntactic and semantic scores when learning with FO action sequences and PO state trajectories with 10% observability.

Table 7.7 shows the results of the case study FO/PO. Looking at the high scores of the syntactic metrics, specially the value of recall, we can conclude that the learned models are, in fact, fairly similar to the GTM. This supports our conclusion that no reformulation occurs in this case study, which also means that the space of possible solutions is restricted to models close to the GTM. The values of sem-Precision and sem-Recall are also very high across the table, which is exactly the desired behavior for these metrics given that solutions are very close to the GTM. In comparison, recall and sem-Recall show similar scores, while sem-Precision is significantly higher than precision, thus showing that the sem-Precision is more lenient towards extra preconditions or effects. This is in line with the results of the previous experiments, where the common appearance of redundant or implicit preconditions in the learned models is penalized by the precision metric. We can interpret this phenomenon as a manifestation of the qualification problem Ginsberg and Smith, 1988. For instance, the model learned for the `move` action of the *hanoi* domain specifies that both the origin and destination disks must be bigger than the one moving, but the GTM contains only one of these preconditions. This learned model is semantically correct but syntactically different from the GTM and hence penalized by the precision metric.

Table 7.8 details the results of the case study NO/NO. One first observation is the impossibility of applying a semantic evaluation in some of the most complex domains with five learning examples. Contrary to the previous case study, the difference between the syntactic and semantic metrics is larger in this scenario with unknown plan horizon. Comparing the scores of both versions, we find that learned models that achieved mediocre scores when using the GTM as reference (syntactic metrics), are in fact reasonably sound and complete, reaching overall scores of 0.92 and 0.89 in sem-Precision and sem-Recall. This is an indication that the models learned by our approach, despite syntactically different from the GTM, require very few editions to explain the validation examples.

	Precision	**Recall**	**sem-Precision**	**sem-Recall**
Blocks	0.55	0.44	0.77	0.77
Driverlog	0.38	0.29	0.86	0.86
Ferry	0.55	0.4	0.82	0.53
Floortile	0.41	0.45	-	-
Grid	0.25	0.26	-	-
Gripper	1.0	0.86	1.0	1.0
Hanoi	0.78	0.88	0.89	1.0
Miconic	0.53	0.56	1.0	0.89
Npuzzle	1.0	1.0	1.0	1.0
Parking	0.53	0.28	-	-
Rovers	0.22	0.53	-	-
Satellite	0.67	0.52	-	-
Transport	0.39	0.35	0.94	1.0
Visitall	1.0	0.2	1.0	1.0
Zenotravel	0.3	0.25	-	-
	0.57	0.48	0.92	0.89

Table 7.8: Syntactic and semantic metric scores for learning tasks with NO action sequences and NO state trajectories.

Looking at the results of both case studies we can draw two conclusions with regards to the semantic metrics proposed in this paper. The first one is that, when no reformulation occurs, these metrics behave similarly to their syntactic counterparts, which means they are a good substitute when the GTM is not available. The second conclusion is that sem-Precision and sem-Recall are better suited to evaluate reformulated models than the original syntactic metrics since they contemplate valid solutions outside the GTM that successfully explain the given input data.

7.7. Discussion

We start this discussion talking about some interesting aspects of our approach and then move the discussion to broader topics concerning the learning of action models. First, we would like to point out that our Action Model Learning approach has a lot in common with the Bayesian approach to learning which treats the parameters as random variables that

are added to the state-space. Under this approach learning just amounts to inference in the augmented model. In our case, the parameters are the alphabet variables and inference process used to estimate them consists in finding explanations for the learning examples.

Next, we would like to talk about an aspect of our approach that we have not developed formally in this dissertation. So far we have said that predicates and operator headers are inputs to the learning problem, but where do these come from? The best way to look at it is that these are the predicates and headers of the target model, which must contain all the ones conveyed by the learning examples (so it is possible to explain them) and can optionally include more. These extra predicates and headers can lead to some exciting results, since they are not needed to explain the learning examples. In fact, they can be considered as latent variables to which the planner must attribute some meaning. For instance, think of what would happen if we tried to learn the blocksworld domain from NO/NO learning examples (no observed actions, only initial and final states) and asking FAMA to use only two operators with two parameters. In such a situation FAMA would use these two operators as `unstack` + `putdown` and `stack` + `pickup` macro-actions. In the extreme, the input predicates and headers can be seen as an upper bound on how many variables and actions FAMA can use to explain the learning examples. This makes our approach also suitable for scenarios where the predicates and operator headers of the actor are unknown.

Interestingly, this feature might have application to learn policies, understanding a policy as an action model with extra preconditions that capture when actions are applied to achieve a given goal. In fact, the navigation strategies used in working example are not too far off from this notion of policy, since they are essentially a basic navigation model extended with an automaton that restricts the applicability of some of their actions. On the flip side, learning such a policy could be done by starting the learning problem with a fully-specified action model and giving FAMA some additional variables to build more restrictive preconditions.

Related to the topic of latent variables is the work in (Asai & Fukunaga, 2018; Konidaris et al., 2018) which falls outside the usual formulation of the action model learning problem. This recent work tackles the problem of learning action models from low-level sensing information and unstructured data by first extracting a latent symbolic representation. Another interesting formulation of the action model learning problem is proposed in (Bonet & Geffner, 2019, 2020). This approach uses as learning examples graphs that encode a fragment of the state-space model, which can be understood as comprising several learned examples with some shared segments.

An important topic when talking about learning is how representative are the learning examples (that is, how much information they convey) and how to collect representative examples. Ideally, representative examples are obtained by collecting all the different state trajectories that are generated when traversing the entire state space (Bonet & Geffner, 2020). Unfortunately, the size of a factored state space grows exponentially with the

number of state variables and size of their domain. As the number of world objects grows, generating the entire set of trajectories becomes intractable. To illustrate this, the state space of a 3-block *blocksworld* problem contains thirteen states, and grows to above five hundreds states for the 5-blocks case (Slaney & Thiébaux, 2001). Besides the size, the topology of the state space also affects the gathering of representative learning examples. It is easier to obtain representative examples in strongly connected state spaces than in spaces with non-reversible actions and dead-ends. While the theoretical efficiency of Machine Learning and Reinforcement Learning algorithms has been widely studied in terms of *sample complexity*, most of the work in the action model learning report only empirical results on this issue. Nevertheless, the sample complexity, for learning STRIPS action operators in fully observable environments has been successfully characterized in terms of *plan-prediction mistakes* (Walsh & Littman, 2008) and the number of examples that are necessary to safely learn models (Juba et al., 2021; Stern & Juba, 2017). Still, further research is needed for bounding sample complexity when examples are not fully observable, are intermittent, or noisy.

Classical blind search algorithms can be used for the systematic exploration of a given state-space model but they present limitations. *Breadth First Search* requires exponential memory and so it is limited to small state spaces; *Depth First Search* can handle larger state spaces but it cannot guarantee exploration diversity; and *Iterative Deepening* provides exploration diversity with low memory consume but it still demands exponential running time. *Novelty-based* algorithms are an appealing family of systematic exploration algorithms since they determine the states to be explored according to their newness. What is more, polynomial running time algorithms, like the IW(1), gracefully scale-up with the number of objects (Lipovetzky & Geffner, 2012). Interestingly, research on *online learning* seems to have taken the baton in regards to collecting representative learning examples. We can see this in works such as (Lamanna et al., 2021; Verma et al., 2021), where the focus is put on keeping track of the part of the action model that we are certain and uncertain, and guiding the exploration so as to reduce the uncertainty.

Part VI

Conclusions and Future Work

8. CONCLUSIONS

This chapter summarizes the contributions of this dissertation, and discusses open lines of research based on the work developed so far.

8.1. Summary of Contributions

This section outlines the main contributions of this PhD Thesis, referring to the chapters that presents them and the publications where they originally appeared.

1. The main contribution of this thesis is a framework for inference and learning in the state-space model given by the classical planning model extended with a rich observation model. This framework allows us to formulate a palette of different inference and learning problems in a cohesive way and to leverage the latest advances developed by state-of-the-art, off-the-shelf planners in order to solve them. Moreover, the state-space model of classical planning presents interesting and distinguishing features that are not found in other state-space models, such as a cost-driven transition model with an implicit schematic representation, and an intermittent observation model. Our framework was drafted in Aineto, Jimenez, et al., 2020 and Aineto, Jiménez, Onaindia, and Ramírez, 2019 and is presented in full in Chapter 3.

2. An Observation Decoding problem $\langle M_a, M_s, s_0, \omega \rangle$ aims at finding the trajectory in $\mathcal{T}(s_0, M_a)$ that best explains the observation sequence ω. Our planning-based approach to address this problem leverages the fact that an observation sequence specifies an ordered occurrence property that can be represented in LTL. This allows as to compute a monitor automaton $\mathcal{A}(\omega)$ that only accepts trajectories in $\mathcal{T}_\omega(s_0, M_a)$ (explanations). We present two different encodings, depending on the type of constraints of the sensor model, to compile away $\mathcal{A}(\omega)$ into a planning problem $\mathcal{P}(\omega)$ whose solutions are restricted to explanations. Our compilation also accounts for sensing cost functions which improve the accuracy of the solutions but severely impact its scalability due to a lack of research in heuristics that support state-dependant costs. The Observation Decoding problem is presented in Chapter 4 and in Aineto, Jimenez, et al., 2020.

3. The Temporal Inference problem $\langle M_a, M_s, s_0, \omega, \mathcal{H} \rangle$ generalizes the task of explaining the actor's past behaviour or predicting its future intentions. We use the LTL formalism as a language to represent hypotheses in $\mathcal{H}$, which allows us to conjecture about the trajectory rather than a single state. We identify three special classes of Temporal Inference problems, namely, Hindsight, Monitor and Prediction, according to whether the hypotheses refer to the past, present or future of the actor.

The solution of a Temporal Inference problem is the hypothesis supported by the best explanation following the principles of Inference to the Best Explanation. In order to find the explanations we apply a straight-forward extension of the compilation presented for Observation Decoding, since both observation sequences and hypotheses are interpreted as LTL properties. This allows us build a planning problem $\mathcal{P}(\eta)$ for each $\eta \in \mathcal{H}$ whose optimal solution would be the best explanations for η. Temporal Inference is presented in Chapter 5 and in Aineto et al., 2021.

4. A Model Recognition problem $\langle \mathbb{M}_a, M_s, s_0, \omega \rangle$ is about figuring out the model of the actor from a set of candidate models $\mathbb{M}_a$. The solution that we seek is the action model $M_a \in \mathbb{M}_a$ that best explains the observation sequence ω. Our formulation of this problem accounts for the possibility that none of the candidate models can explain the observation sequence by weighting the modifications required by the model in order to generate an explanation. We do so by leveraging a novel encoding that allows representing any STRIPS action model using a finite set of variables that we call *alphabet*. Our planning-based approach incorporates this alphabet alongside a new set of edit actions that implement the insertion and deletion operations. This allows us to compile a problem $\mathcal{P}_{M_a}(\omega)$ for every $M_a \in \mathbb{M}_a$ whose solutions consider the necessary modifications to the candidate model and search for an explanation in a single planning episode. The Model Recognition problem is presented in Chapter 6 and in Aineto, Jiménez, Onaindia, and Ramírez, 2019.

5. The aim of an Action Model Learning problem $\langle M_a[], M_s, \mathcal{L} \rangle$ is to estimate the applicability App and effects Eff constraints of the actor so that they complete the input model $M_a[] = \langle X, A, c \rangle$ in a way that can explain the learning examples $\mathcal{L}$. Our planning-based formulation of this problem, FAMA, poses the learning problem as an editing problem leveraging the alphabet of an action model and edit actions. Then, FAMA compiles the learning problem into a planning problem $\mathcal{P}_{M_a}(\mathcal{L})$ whose solution first edits model M_a (which initially has no preconditions nor effects) and then explains the learning examples $\mathcal{L}$. Our approach takes full advantage of our framework, which allows it to learn from learning examples with gaps (only initial and final states in the extreme), a novel feature in the area of Action Model Learning. We also propose new novel metrics for evaluating the quality of the learned action models with respect to a set of validation examples. The Action Model Learning problem is presented in Chapter 7 and in Aineto et al., 2018, 2019.

8.2. Future work

The work developed in this thesis and most of the cited works on inference and learning focus on the classical planning model or small extensions which are still purely discrete. Real-world applications such as robotics or autonomous vehicles demand hybrid representations, where state variables are discrete or continuous (Xu & Laird, 2011), and evolve

with respect to a time function which can also be continuous (Ramirez et al., 2017).

Inference in hybrid settings is a task that we have already started tackling in a recent work with very promising results (Aineto, Onaindia, et al., 2020, 2022). In this work we present an extension of the Observation Decoding problem that aims at explaining the observed behaviour of a *hybrid system* (HS). We formulate the problem using the formalism of *hybrid automata* (HA), and characterize the explanations as the language of a network of HA that comprises one automaton for the HS and another one that serves as monitor. We observe that this problem corresponds to a *Falsification* problem in model-checking, which consists in finding a hybrid trajectory that satisfies (or violates) a given property. However, state-of-the-art model checkers struggle to find concrete trajectories, so we propose a formal mapping from HA to PDDL+ and rely on off-the-shelf automated planners. An experimental analysis over domains with piece-wise constant, linear and nonlinear dynamics reveals that the proposed PDDL+ approach is much more efficient than solving directly the explanation problem with model-checking solvers. Regarding learning in hybrid settings, defining expressive arithmetic-logic preconditions and effects that constrain the value of hybrid state spaces can easily yield intractable hypothesis spaces. The computation of candidates of tractable subsets of the hypothesis space, e.g. by leveraging input observations as in *regression tree* learning, seems a promising approach to effectively handle these hypothesis spaces.

On the topic of extensions to other planning models, we believe that the conformant planning model has a lot of potential, specially for Action Model Learning. If we think of FAMA as a *learning as satisfiability* approach whose solution is a concrete model, we can think of an extension of FAMA to conformant planning as a *learning as belief propagation* approach. One advantage of such an extension is that it would be able to find all the action models consistent with the learning examples. Another important benefit is that the set of solutions could be incrementally updated by individually processing the learning examples and hence, its computational complexity is not exponentially dependent on the size of the set of learning examples.

As we have already touched upon in our discussions, scalability is a concerning aspect of the proposed approach. We envision two ways in which this can be improved. Our first idea is to discard the compilation of the monitor automaton and, instead, integrate it into a complete search scheme as a landmark graph. In theory, such approach would take better advantage of landmark-based heuristics without impacting the quality of the solutions. However, we would lose the flexibility of being able to easily leverage new advances in state-of-the-art planners. Our second idea is to follow a heuristic-based approach as is done in some works for Goal Recognition (Pereira et al., 2017, 2020; Vered et al., 2018). This approach uses heuristic scores for its estimations rather than explanations, so it sacrifices accuracy in favor of much faster solution times. It remains to be seen if such relaxations will be applicable for problems that require actual explanations, such as Observation Decoding and Action Model Learning, however the relaxed plan seems like a plausible proxy here.

Finally, we want to end this section talking about a feature of our framework that we think can lead to interesting applications when it comes to learning. The feature in question is support for intermittency, which so far we have understood as being able to explain gapped observation sequences thanks to the use of planners. We believe that this feature can also be exploited intentionally, in order to learn procedural action models. For many problems, preconditions and effects of actions are more naturally defined as procedures. For instance, in the chess game, the possible moves of a knight are naturally defined as follows: *the knight moves two squares horizontally and then one square vertically*, or *it moves one square horizontally and then two squares vertically*. In order to learn such procedures, we can leverage the capabilities of our Action Model Learning approach, FAMA. The idea is that to explain a procedural action we can use a sequence of actions. FAMA supports intermittency and therefore will use as many actions as needed to explain a learning example that comes from a procedural model.

BIBLIOGRAPHY

Aguas, J. S., Jiménez, S., & Jonsson, A. (2021). Generalized planning as heuristic search. *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 569–577.

Aineto, D., Jimenez, S., & Onaindia, E. (2020). Observation decoding with sensor models: Recognition tasks via classical planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, *30*, 11–19.

Aineto, D., Jiménez, S., & Onaindia, E. (2018). Learning STRIPS action models with classical planning. *International Conference on Automated Planning and Scheduling, (ICAPS-18)*, 399–407.

Aineto, D., Jiménez, S., & Onaindia, E. (2019). Learning action models with minimal observability. *Artificial Intelligence Journal*, *275*, 104–137.

Aineto, D., Jiménez, S., & Onaindia, E. (2021). Generalized temporal inference via planning. In M. Bienvenu, G. Lakemeyer, & E. Erdem (Eds.), *Proceedings of the 18th international conference on principles of knowledge representation and reasoning, KR 2021, online event, november 3-12, 2021* (pp. 22–31).

Aineto, D., Jiménez, S., Onaindia, E., & Ramírez, M. (2019). Model Recognition as Planning. *International Conference on Automated Planning and Scheduling, (ICAPS-19)*, 13–21.

Aineto, D., Onaindia, E., Ramírez, M., Scala, E., & Serina, I. (2020). Towards plan recognition in hybrid systems. *Joint Proceedings of the 8th Italian Workshop on Planning and Scheduling and the 27th International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion co-located with AIxIA 2020, Online Event, November 25-27, 2020*, *2745*.

Aineto, D., Onaindia, E., Ramírez, M., Scala, E., & Serina, I. (2022). Explaining the behaviour of hybrid systems with pddl+ planning. *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI-22*.

Albore, A., Palacios, H., & Geffner, H. (2009). A translation-based approach to contingent planning. *Twenty-First International Joint Conference on Artificial Intelligence*.

Alford, R. W., Kuter, U., & Nau, D. (2009). Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. *21st International Joint Conference on Artificial Intelligence, (IJCAI-09)*, 1629–1634.

Allahverdyan, A., & Galstyan, A. (2011). Comparative analysis of viterbi training and maximum likelihood estimation for hmms. *Advances in Neural Information Processing Systems*, *24*.

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

Asai, M., & Fukunaga, A. (2018). Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *National Conference on Artificial Intelligence, AAAI-18*.

Bacchus, F. (2001). The AIPS '00 planning competition. *AI Mag.*, *22*(3), 47–56. http://www.aaai.org/ojs/index.php/aimagazine/article/view/1571

Bacchus, F., & Kabanza, F. (1998). Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, *22*(1-2), 5–27.

Bäckström, C., & Jonsson, P. (2011). All pspace-complete planning problems are equal but some are more equal than others. In D. Borrajo, M. Likhachev, & C. L. López (Eds.), *Proceedings of the fourth annual symposium on combinatorial search, SOCS 2011, castell de cardona, barcelona, spain, july 15.16, 2011*. AAAI Press. http://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/view/4009

Baier, J. A., Fritz, C., & McIlraith, S. A. (2007). Exploiting procedural domain control knowledge in state-of-the-art planners. *International Conference on Automated Planning and Scheduling, (ICAPS-07)*, 26–33.

Baier, J. A., & McIlraith, S. A. (2006). Planning with temporally extended goals using heuristic search. *ICAPS*, 342–345.

Bellman, R. (1957). *Dynamic programming* (1st ed.). Princeton University Press. http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false

Bertsekas, D. P. (1995). *Dynamic programming and optimal control* (1st). Athena Scientific.

Blum, A., & Furst, M. L. (1995). Fast planning through planning graph analysis. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, 1636–1642. http://ijcai.org/Proceedings/95-2/Papers/080.pdf

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

Bonet, B., & Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. *International Conference on Artificial Intelligence Planning Systems, (ICAPS-05)*.

Bonet, B., & Geffner, H. (2019). Learning first-order symbolic planning representations from plain graphs. *ECAI*.

Bonet, B., & Geffner, H. (2020). Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. *European Conference on Artificial Intelligence*.

Bonet, B., Loerincs, G., & Geffner, H. (1997). A robust and fast action selection mechanism for planning. In B. Kuipers & B. L. Webber (Eds.), *Proceedings of the fourteenth national conference on artificial intelligence and ninth innovative applications of artificial intelligence conference, AAAI 97, IAAI 97, july 27-31, 1997,*

providence, rhode island, USA (pp. 714–719). AAAI Press / The MIT Press. http://www.aaai.org/Library/AAAI/1997/aaai97-111.php

Bonet, B., Palacios, H., & Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. *National Conference on Artificial Intelligence, (AAAI-10)*.

Bryce, D., Benton, J., & Boldt, M. W. (2016). Maintaining evolving domain models. *International Joint Conference on Artificial Intelligence, (IJCAI-2016)*, 3053–3059.

Bryce, D., Kambhampati, S., & Smith, D. E. (2006). Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*, *26*, 35–99.

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

Chakraborti, T., Kulkarni, A., Sreedharan, S., Smith, D. E., & Kambhampati, S. (2019). Explicability? legibility? predictability? transparency? privacy? security? the emerging landscape of interpretable agent behavior. *29th International Conference on Automated Planning and Scheduling, ICAPS 2019*, 86–96.

Chakraborti, T., Sreedharan, S., & Kambhampati, S. (2018a). Explicability versus explanations in human-aware planning. *International Conference on Autonomous Agents and MultiAgent Systems, AAMAS-18*, 2180–2182.

Chakraborti, T., Sreedharan, S., & Kambhampati, S. (2018b). Human-aware planning revisited: A tale of three models. *IJCAI-ECAI XAI/ICAPS XAIP Workshops*.

Chakraborti, T., Sreedharan, S., Zhang, Y., & Kambhampati, S. (2017). Plan explanations as model reconciliation: Moving beyond explanation as soliloquy. *International Joint Conference on Artificial Intelligence, (IJCAI-17)*, 156–163.

Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 151–158.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1989). *Introduction to algorithms*. The MIT Press; McGraw-Hill Book Company.

Corrêa, A. B., Francès, G., Pommerening, F., & Helmert, M. (2021). Delete-relaxation heuristics for lifted classical planning. *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 94–102.

Cresswell, S., & Coddington, A. M. (2004). Compilation of LTL goal formulas into PDDL. In R. L. de Mántaras & L. Saitta (Eds.), *Proceedings of the 16th eureopean conference on artificial intelligence, ecai'2004, including prestigious applicants of intelligent systems, PAIS 2004, valencia, spain, august 22-27, 2004* (pp. 985–986). IOS Press.

Cresswell, S. N., McCluskey, T. L., & West, M. M. (2013). Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, *28*(02), 195–213.

Davis, J., & Goadrich, M. (2006). The relationship between precision-recall and ROC curves. *International Conference on Machine learning*, 233–240.

Davis-Mendelow, S., Baier, J. A., & McIlraith, S. A. (2013). Assumption-based planning: Generating plans and explanations under incomplete knowledge. *27th AAAI Conference on Artificial Intelligence*.

E-Martín, Y., R.-Moreno, M. D., & Smith, D. E. (2015). A fast goal recognition technique based on interaction estimates. *Proc. 24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, 761–768.

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3-4), 189–208.

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

Fox, M., Long, D., & Magazzeni, D. (2017). Explainable planning. *First Workshop on Explainable AI (XAI), IJCAI-13*.

Fritz, C., & McIlraith, S. A. (2007). Monitoring plan optimality during execution. *7th International Conference on Automated Planning and Scheduling, ICAPS-2007 2007*, 144–151.

Gerevini, A., Haslum, P., Long, D., Saetti, A., & Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, *173*(5-6), 619–668. https://doi.org/10.1016/j.artint.2008.10.012

Giacomo, G. D., & Vardi, M. Y. (2013). Linear temporal logic and linear dynamic logic on finite traces. In F. Rossi (Ed.), *Proceedings of the 23rd international joint conference on artificial intelligence* (pp. 854–860). IJCAI/AAAI.

Ginsberg, M. L., & Smith, D. E. (1988). Reasoning about action II: the qualification problem. *Artificial Intelligence*, *35*(3), 311–342.

Grastien, A., Haslum, P., Thiébaux, S., et al. (2011). Exhaustive diagnosis of discrete event systems through exploration of the hypothesis space. *22nd International Workshop on Principles of Diagnosis (DX-11)*, 60–67.

Harman, G. H. (1965). The inference to the best explanation. *The philosophical review*, *74*(1), 88–95.

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In S. A. Chien, S. Kambhampati, & C. A. Knoblock (Eds.), *Proceedings of the fifth international conference on artificial intelligence planning systems, breckenridge, co, usa, april 14-17, 2000* (pp. 140–149). AAAI. http://www.aaai.org/Library/AIPS/2000/aips00-015.php

Haslum, P., & Grastien, A. (2011). Diagnosis as planning: Two case studies. *Proc. of the International Scheduling and Planning Applications workshop (SPARK)*.

Helmert, M. (2003). Complexity results for standard benchmark domains in planning. *Artif. Intell.*, *143*(2), 219–262. https://doi.org/10.1016/S0004-3702(02)00364-8

Helmert, M. (2006a). The fast downward planning system. *J. Artif. Intell. Res.*, *26*, 191–246. https://doi.org/10.1613/jair.1705

Helmert, M. (2006b). New complexity results for classical planning benchmarks. In D. Long, S. F. Smith, D. Borrajo, & L. McCluskey (Eds.), *Proceedings of the six-*

teenth international conference on automated planning and scheduling, ICAPS 2006, cumbria, uk, june 6-10, 2006 (pp. 52–62). AAAI. http://www.aaai.org/Library/ICAPS/2006/icaps06-006.php

Helmert, M., Haslum, P., Hoffmann, J., & Nissim, R. (2014). Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM (JACM)*, *61*(3), 1–63.

Hoffmann, J. (2003). The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *J. Artif. Intell. Res.*, *20*, 291–341.

Hoffmann, J., & Brafman, R. (2005). Contingent planning via heuristic forward search with implicit belief states. *Proc. ICAPS, 2005*.

Hoffmann, J., & Edelkamp, S. (2005). The deterministic part of IPC-4: an overview. *J. Artif. Intell. Res.*, *24*, 519–579. https://doi.org/10.1613/jair.1677

Hoffmann, J., Gomes, C. P., Selman, B., & Kautz, H. A. (2007). SAT encodings of state-space reachability problems in numeric domains. In M. M. Veloso (Ed.), *IJCAI 2007, proceedings of the 20th international joint conference on artificial intelligence, hyderabad, india, january 6-12, 2007* (pp. 1918–1923). http://ijcai.org/Proceedings/07/Papers/309.pdf

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, *14*, 253–302. https://doi.org/10.1613/jair.855

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

Howey, R., Long, D., & Fox, M. (2004). VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, 294–301.

Jiménez, S., Coles, A., & Smith, A. (2006). Planning in probabilistic domains using a deterministic numeric planner. *25th Workshop of the UK Planning and Scheduling Special Interest Group*.

Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., & Borrajo, D. (2012). A review of machine learning for automated planning. *The Knowledge Engineering Review*, *27*(04), 433–467.

Juba, B., Le, H. S., & Stern, R. (2021). Safe learning of lifted action models. *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, 379–389.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, *101*(1-2), 99–134.

Kambhampati, S. (2007). Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. *National Conference on Artificial Intelligence, (AAAI-07)*.

Kautz, H. A., & Selman, B. (1992). Planning as satisfiability. *ECAI*, *92*, 359–363.

Kautz, H. A., & Selman, B. (1996). Pushing the envelope: Planning, propositional logic and stochastic search. In W. J. Clancey & D. S. Weld (Eds.), *Proceedings of the thirteenth national conference on artificial intelligence and eighth innovative ap-*

plications of artificial intelligence conference, AAAI 96, IAAI 96, portland, oregon, usa, august 4-8, 1996, volume 2 (pp. 1194–1201). AAAI Press / The MIT Press. http://www.aaai.org/Library/AAAI/1996/aaai96-177.php

Keren, S., Gal, A., & Karpas, E. (2019). Goal Recognition Design in Deterministic Environments. *Journal of Artificial Intelligence Research*, *65*, 209–269.

Konidaris, G., Kaelbling, L. P., & Lozano-Pérez, T. (2018). From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, *61*, 215–289.

Kucera, J., & Barták, R. (2018). LOUGA: learning planning operators using genetic algorithms. *Pacific Rim Knowledge Acquisition Workshop, PKAW-18*, 124–138.

Kvarnström, J., & Doherty, P. (2000). Talplanner: A temporal logic based forward chaining planner. *Ann. Math. Artif. Intell.*, *30*(1-4), 119–169. https://doi.org/10.1023/A:1016619613658

Lamanna, L., Saetti, A., Serafini, L., Gerevini, A., & Traverso, P. (2021). Online learning of action models for PDDL planning. *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 4112–4118.

Lipovetzky, N., & Geffner, H. (2012). Width and serialization of classical planning problems. *European Conference on Artificial Intelligence*, 540–545.

Lipovetzky, N., & Geffner, H. (2017). Best-first width search: Exploration and exploitation in classical planning. *Thirty-First AAAI Conference on Artificial Intelligence*.

Little, I., & Thiebaux, S. (2007). Probabilistic planning vs. replanning. *ICAPS Workshop on IPC: Past, Present and Future*.

Long, D., & Fox, M. (2003). The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res.*, *20*, 1–59. https://doi.org/10.1613/jair.1240

López, C. L., Celorrio, S. J., & Olaya, Á. G. (2015). The deterministic part of the seventh international planning competition. *Artificial Intelligence*, *223*, 82–119.

López, C. L., Celorrio, S. J., & Olaya, A. G. (2015). The deterministic part of the seventh international planning competition. *Artif. Intell.*, *223*, 82–119. https://doi.org/10.1016/j.artint.2015.01.004

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). PDDL – The Planning Domain Definition Language.

McDermott, D. V. (2000). The 1998 AI planning systems competition. *AI Mag.*, *21*(2), 35–55. https://doi.org/10.1609/aimag.v21i2.1506

Miettinen, K. M. (1998). *Nonlinear multiobjective optimization*. Springer.

Mourão, K., Zettlemoyer, L. S., Petrick, R. P. A., & Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. *Conference on Uncertainty in Artificial Intelligence, UAI-12*, 614–623.

Muise, C. (2016). Planning.domains. *ICAPS system demonstration*.

Muise, C., & Lipovetzky, N. (2015). Unplannability ipc track. *The International Planning Competition (WIPC-15)*, 14. https://people.eng.unimelb.edu.au/nlipovetzky/papers/WIPC15_unplannability.pdf

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *J. Artif. Intell. Res.*, *12*, 271–315. https://doi.org/10.1613/jair.735

Nilsson, N. J. (1980). *Principles of artificial intelligence*. Morgan Kaufmann Publishers Inc.

Patrizi, F., Lipovetzky, N., De Giacomo, G., & Geffner, H. (2011). Computing infinite plans for ltl goals using a classical planner. *International Joint Conference on Artificial Intelligence, (IJCAI-11)*, 2003–2008.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

Peot, M. A., & Smith, D. E. (1992). Conditional nonlinear planning. *Artificial Intelligence Planning Systems*, 189–197.

Pereira, R. F., Oren, N., & Meneguzzi, F. (2017). Landmark-based heuristics for goal recognition. *31st AAAI Conference on Artificial Intelligence (AAAI-17)*.

Pereira, R. F., Oren, N., & Meneguzzi, F. (2020). Landmark-based approaches for goal recognition as planning. *Artif. Intell.*, *279*.

Pnueli, A. (1977). The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 46–57.

Pozanco, A., E.-Martín, Y., Fernández, S., & Borrajo, D. (2018). Counterplanning using goal recognition and landmarks. *International Joint Conference on Artificial Intelligence, (IJCAI-18)*, 4808–4814.

Pryor, L., & Collins, G. (1996). Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, *4*, 287–339.

Ramirez, M., Papasimeon, M., Benke, L., Lipovetzky, N., Miller, T., & Pearce, A. R. (2017). Real-time uav maneuvering via automated planning in simulations. *International Joint Conference on Artificial Intelligence*, 5243–5245.

Ramírez, M., & Geffner, H. (2009). Plan Recognition as Planning. *International Joint conference on Artifical Intelligence, (IJCAI-09)*, 1778–1783.

Ramírez, M., & Geffner, H. (2010). Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners. *24th AAAI National Conference on Artificial Intelligence, (AAAI-10)*.

Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, *39*, 127–177. https://doi.org/10.1613/jair.2972

Rintanen, J. (2000). Incorporation of temporal logic control into plan operators. In W. Horn (Ed.), *ECAI 2000, proceedings of the 14th european conference on artificial intelligence, berlin, germany, august 20-25, 2000* (pp. 526–530). IOS Press.

Rintanen, J. (2003). Expressive equivalence of formalisms for planning with sensing. In E. Giunchiglia, N. Muscettola, & D. S. Nau (Eds.), *Proceedings of the thirteenth international conference on automated planning and scheduling (ICAPS 2003), june 9-13, 2003, trento, italy* (pp. 185–194). AAAI. http://www.aaai.org/Library/ICAPS/2003/icaps03-019.php

Rintanen, J. (2014). Madagascar: Scalable planning with SAT. *International Planning Competition, (IPC-2014)*.

Schupbach, J. N. (2017). Inference to the best explanation, cleaned up and made respectable. *Best explanations: New essays on inference to the best explanation*, 39–61.

Segovia-Aguas, J., Celorrio, S. J., & Jonsson, A. (2019). Computing programs for generalized planning using a classical planner. *Artificial Intelligence*.

Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2018). Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research*, *62*, 755–797.

Sievers, S., Wehrle, M., & Helmert, M. (2016). An analysis of merge strategies for merge-and-shrink heuristics. *Proceedings of the 26th International Conference on Automated Planning and Scheduling*.

Slaney, J., & Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, *125*(1-2), 119–153.

Smallwood, R. D., & Sondik, E. J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations research*, *21*(5), 1071–1088.

Sohrabi, S., Baier, J. A., & McIlraith, S. A. (2010). Diagnosis as planning revisited. *12th International Conference on the Principles of Knowledge Representation and Reasoning*.

Sohrabi, S., Baier, J. A., & McIlraith, S. A. (2011). Preferred explanations: Theory and generation via planning. *Proc. of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011*.

Sohrabi, S., Riabov, A. V., & Udrea, O. (2016). Plan recognition as planning revisited. *International Joint Conference on Artificial Intelligence, (IJCAI-16)*, 3258–3264.

Sreedharan, S., Chakraborti, T., & Kambhampati, S. (2018). Handling model uncertainty and multiplicity in explanations via model reconciliation. *International Conference on Automated Planning and Scheduling, (ICAPS-18)*, 518–526.

Srivastava, B., Nguyen, T. A., Gerevini, A., Kambhampati, S., Do, M. B., & Serina, I. (2007). Domain independent approaches for finding diverse plans. *IJCAI*, 2016–2022.

Stern, R., & Juba, B. (2017). Efficient, safe, and probably approximately complete learning of action models. *International Joint Conference on Artificial Intelligence, (IJCAI-17)*, 4405–4411.

Sukthankar, G., Goldman, R. P., Geib, C., Pynadath, D. V., & Bui, H. (2014). *Plan, Activity, and Intent Recognition: Theory and Practice*. Morgan Kaufmann.

Vered, M., Pereira, R. F., Kaminka, G., & Meneguzzi, F. R. (2018). Towards online goal recognition combining goal mirroring and landmarks. *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, 2018, Suécia.*

Verma, P., Marpally, S. R., & Srivastava, S. (2021). Asking the right questions: Learning interpretable action models through query answering. *Thirty-Fifth AAAI Confer-*

ence on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021, 12024–12033.

Walsh, T. J., & Littman, M. L. (2008). Efficient learning of action schemas and web-service descriptions. *AAAI Conference on Artificial Intelligence*, *8*, 714–719.

Xu, J. Z., & Laird, J. E. (2011). Combining Learned Discrete and Continuous Action Models. In W. Burgard & D. Roth (Eds.), *AAAI conference on artificial intelligence, AAAI 2011*. AAAI Press.

Yang, Q., Wu, K., & Jiang, Y. (2007). Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, *171*(2-3), 107–143.

Zhuo, H. H., & Kambhampati, S. (2013). Action-model acquisition from noisy plan traces. *International Joint Conference on Artificial Intelligence, IJCAI-13*, 2444–2450.

Zhuo, H. H., & Kambhampati, S. (2017). Model-lite planning: Case-based vs. model-based approaches. *Artificial Intelligence*, *246*, 1–21.

Zhuo, H. H., Nguyen, T. A., & Kambhampati, S. (2013). Refining incomplete planning domain models through plan traces. *International Joint Conference on Artificial Intelligence, IJCAI-13*, 2451–2458.

Zhuo, H. H., Yang, Q., Hu, D. H., & Li, L. (2010). Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, *174*(18), 1540–1569.