Inductive Learning of Temporal Advice Formulae for Guiding Planners

MSc Thesis *(Afstudeerscriptie)*

written by

Paulius Skaisgiris

under the supervision of **Dr. Balder ten Cate** and **Dr. Daniele Meli**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
7 April 2025	Dr. Benno van den Berg (chair)
	Dr. Balder ten Cate (co-supervisor)
	Dr. Daniele Meli (co-supervisor)
	Dr. Gregor Behnke
	Dr. Ronald de Haan

Institute for Logic, Language and Computation

Abstract

Reinforcement learning (RL), one of the most successful methods for planning in stochastic environments, suffers from sample inefficiency, requiring extensive exploration of the environment to converge on good solutions. Additionally, most RL methods function as black boxes, limiting human intervention. This thesis attempts to tackle these problems and presents a method for learning temporal advice formulae to enhance the efficiency, quality, and safety of planning algorithms while maintaining transparency.

We use linear temporal logic on finite traces as a general framework for expressing advice. Inspired by previous works by Meli et al. and Ielo et al., we combine the existing research on learning time-independent advice for planners and inferring formulae from execution traces, to develop a unified method for learning temporal advice. We represent the temporal logic formulae as answer set programs and use the ILASP software for inductively learning them from execution traces. Unlike previous work, our approach tailors temporal logic formulae for guiding planning agents and accounts for partially observable and noisy domains. This integration enables automated advice generation, aiming to improve decision-making in automated planning.

We experimentally validate our approach in two environments: a simple fully observable gem pickup scenario and RockSample, which involves long planning horizons and partial observability. Our results demonstrate that generalizable temporal advice formulae can be learned from only a few examples, provided they are of high quality and clearly distinguish good from bad behavior.

Keywords: linear temporal logic on finite traces, inductive learning of answer set programs, planning in markov decision processes, guiding planners

Acknowledgements

Firstly, I would like to extend my gratitude to my patient, supportive, and insightful thesis supervisors, Balder ten Cate and Daniele Meli. I'm grateful for the opportunity to have worked with you. I've learned a great deal about how you tackle novel academic challenges, and you have inspired me to pursue further work in academia. I am grateful that you treated me almost like a colleague, and we approached this thesis as if we were collaborating as fellow academics.

I'm deeply grateful to Matthew Tait for his kindness and patience in answering all my questions about ILASP. Without his support, this thesis would have turned out very differently, and his help truly made the ILASP part of this thesis much easier.

I would like to thank Prof. Alessandro Farinelli for various discussions and overall motivation and belief in the project. I'm also very grateful to the academics who engaged in thoughtful conversations about my thesis, gave intriguing ideas to pursue further, or explained their papers and codebases. Specifically, I would like to thank Dr. Kristin Yvonne Rozier, Dr. Mark Law, Prof. Giuseppe de Giacomo, Nima Motamed, Antonio Ielo, Dr. Alessio Cecconi, and Dr. Malvin Gattinger.

There have been numerous people in my personal life who have contributed to keeping me satiated, sprightly, and sane throughout the course of writing this thesis. To each of them I am extremely grateful. This thesis is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

Artificial intelligence (AI) research aims to create systems capable of autonomously performing tasks, assisting in complex decision-making, and solving challenging problems beyond human capability. Evidently, a crucial skill for developing such systems is planning, the ability to reason about actions and their consequences over time. Automated systems are increasingly used in safety-critical environments due to their cost-effectiveness, performance, and speed. In these contexts, planning is essential for choosing actions that not only lead to desirable outcomes but also ensure a high level of safety.

This thesis presents a method for learning symbolic advice aimed for enhancing efficiency, improving solution quality, and potentially increasing the safety and reliability of planning algorithms, all while maintaining transparency. In doing so, we contribute to the expanding field of neurosymbolic AI, which aims to integrate transparent, provable symbolic approaches with generalizable yet data-intensive neural models that lack explicit reasoning capabilities.

In this thesis, we follow the common approach of representing planning problems using Markov Decision Processes (MDPs). MDPs provide a mathematical framework for decision-making in stochastic environments, where outcomes are partly random and partly under the control of a decision-maker. An MDP is defined by a set of states, actions, transition probabilities, rewards, and a discount factor. The agent does not know neitneitherher the reward function nor the transition function. The key assumption in an MDP is the Markov property, which states that transitions depend only on the current state and action. Planning in MDPs involves finding an optimal policy function, a mapping from states to actions. Agents learn to plan in this environment based on the MDP's reward function, which indicates whether their actions result in positive or negative rewards. The main algorithms for finding policies use dynamic programming techniques such as Value Iteration and Policy Iteration. These algorithms progressively refine estimates of the expected future rewards for specific actions in given states, facilitating intelligent decision-making even in uncertain environments.

Despite significant progress, current approaches to planning in MDPs have notable drawbacks. Reinforcement learning (RL), one of the most successful methods for planning in MDPs, is known to be sample inefficient, requiring extensive exploration of the environment to converge on good solutions. While this phenomenon may be tolerable in simulators, it becomes problematic when an agent uses RL to learn in real-world environments. In addition, most RL methods function as black boxes, learning in their own opaque way, limiting human intervention.

To address these issues, the fields of warm-starting RL and reward shaping propose various solutions. A common approach involves learning from offline datasets before transitioning to online learning in the real system [Hester et al., 2017, Nair et al., 2021]. However, these algorithms rely on deep reinforcement learning, which is data-intensive, slow, and lacks guarantees on the type of behavior learned from offline data.

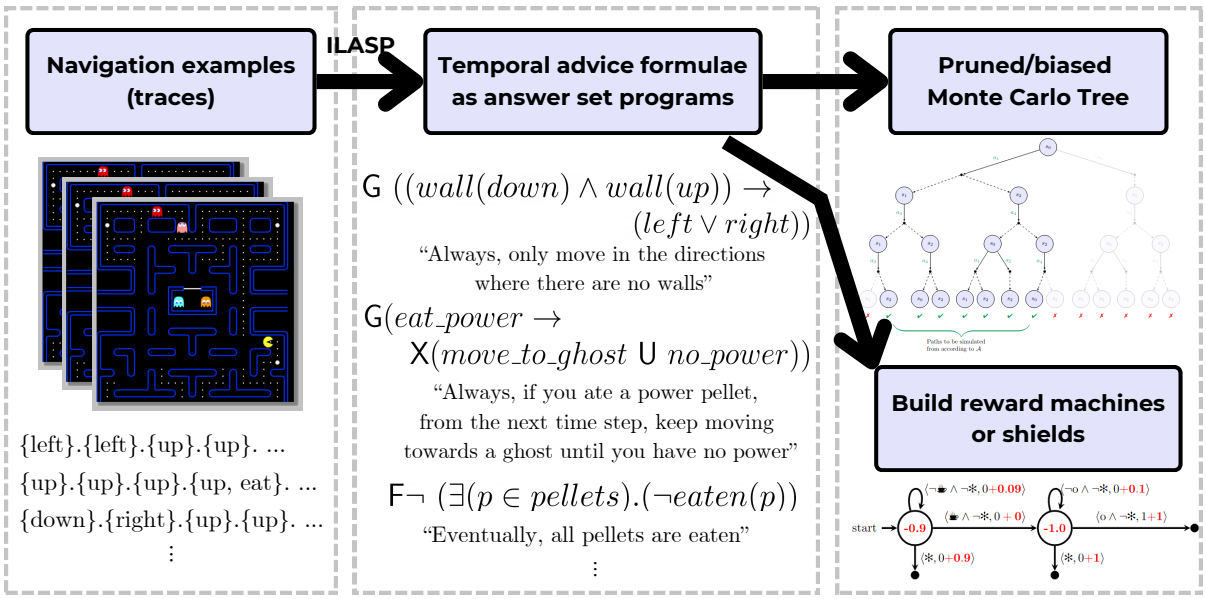


Figure 1.1: The high-level approach of this thesis. Figures on the right are taken from Chakraborty [2023] and Toro Icarte et al. [2022].

In contrast, some approaches use symbolic methods, making their guidance more transparent. For example, recent work on reward shaping leverages temporal logic to modify the MDP's reward signal, influencing the planner to achieve symbolic goals [Toro Icarte et al., 2022, De Giacomo et al., 2021]. Similarly, Toro Icarte et al. [2018] used temporal logic formulae to provide advice to the planner, integrating this advice into the R-MAX algorithm to softly guide the agent. Another symbolic approach, learning from policy sketches [Andreas et al., 2017], associates symbolic goals with subpolicies, optimizing the overall objective by maximizing rewards across all subpolicies.

However, manually designing suitable advice or heuristics, as seen in the aforementioned works, poses a significant challenge [Rozier, 2016] (see Figures 1 and 2 in the paper). This process often involves extensive trial and error and the results can be prone to reward hacking [Skalse et al., 2022], where AI systems exploit flaws in the reward function to maximize their score in unintended ways. Ideally, advice generation should be automated, based on demonstrations of good past behaviour, and resistant to human biases.

Having said this, some papers do propose methods for learning guiding mechanisms. For example, there are works outlining learning reward machines [Icarte et al., 2019, Xu et al., 2020, Dohmen et al., 2022] and subgoal automata [Furelos-Blanco et al., 2021]. However, the learned objects are highly specific and are not designed for use in other guiding approaches. For instance, reward machines cannot be directly applied to compute heuristic functions.

In contrast, we follow the approach established by Toro Icarte et al. [2018], using linear temporal logic on finite traces ($\mathsf{LTL_f}$) [De Giacomo and Vardi, 2013] as a general specification for advice. We argue that these temporal advice formulae can be applied to existing methods for guiding planners. Therefore, this thesis aims to automatically learn such temporal advice formulae and use them as a foundation for various RL guiding techniques. The high-level approach of this thesis is illustrated in Figure 1.1.

The method we use to learn these formulae is also noteworthy. Our work is largely inspired by Meli et al. [2024] who represented time-indepedent advice for planners as answer set programs (ASPs) and learned them them using the ILASP [Law et al., 2020b] system. In addition, Ielo et al. [2023] represented $\mathsf{LTL_f}$ formulae in ASP and demonstrated that ILASP can also be used to learn such formulae from traces of system execution. However, the approaches of Meli et al. [2024] and Ielo et al. [2023] have not yet been combined. Specifically, Meli et al. [2024] concentrate

on learning *time-independent* advice, whereas Ielo et al. [2023] derive generic $\mathsf{LTL_f}$ formulae that merely describe examples, rather than $\mathsf{LTL_f}$ formulae suitable for guiding planning agents. Furthermore, the latter work does not consider partially observable or noisy planning domains which we address in this thesis. In this thesis, we combine these two approaches, arriving at a solution to learn temporal advice formulae as defined by Toro Icarte et al. [2018].

To illustrate the goal of this work, consider the temporal advice formulae and their natural language interpretations for Pac-Man, as shown in the center of Figure 1.1. These examples illustrate the generality of temporal advice formulae. The first formula represents an actionable rule, specifying which actions the agent should take. The second formula outlines a complex, temporally extended maneuver for the agent. The last example defines a high-level goal for the agent to achieve without specifying how to accomplish it. Once learned, this advice can be used to guide the planner toward exploring action sequences (or futures) that satisfy the formula. The method proposed in this thesis enables the learning of both low-level and high-level advice. Nonetheless, our focus will be on low-level, actionable temporal advice formulae, as this type of advice is often overlooked in the literature.

This thesis is structured as follows. Chapter 2 offers the necessary background knowledge to understand the main results of this thesis. Chapter 3 formalizes the problem of learning $\mathsf{LTL_f}$ formulae and provides a proof of the learning task's correctness, demonstrating that the ILASP software can be effectively used for this purpose. Chapter 4 explores the motivation behind temporal advice formulae, adapts the ILASP learning task from learning $\mathsf{LTL_f}$ formulae to learning temporal advice formulae, and demonstrates the learning of advice on a toy example. Chapter 5 details experiments evaluating our proposed method in both simple and more complex environments, along with a discussion on runtime performance and generalization results. Finally, Chapter 6 concludes the thesis and provides an extensive section on future work.

1.2 Related work

Computing and using heuristics in classical planning. If one relaxes the non-determinism assumption in MDPs, we get a fully observable and fully deterministic environment which is studied by the classical planning literature. In classical planning, heuristics are estimation functions which provide an informed guess about the value of some state of the search problem [Ghallab et al., 2004, Nissim et al., 2011]. Search algorithms, such as A^* [Hart et al., 1968], use these heuristics to guide their search process. Various approaches exist to compute heuristics in classical planning environments. The well-known ones are relaxed planning [Hoffmann and Nebel, 2001], landmark heuristics [Hoffmann et al., 2004], abstraction heuristics [Edelkamp, 2002], critical path heuristics [Blum and Furst, 1997]. These approaches differ in that relaxed planning simplifies the problem by ignoring delete lists, landmark heuristics focus on mandatory subgoal sequences, abstraction heuristics rely on state space reduction, and critical path heuristics analyze action dependencies within the planning graph.

Recent works in classical planning significantly surpass the traditional work. The approaches include using neural networks as heuristic functions [Ferber Patrick et al., 2020], learning heuristics using hypergraph networks [Shen et al., 2020], learning a propositional PDDL action model from unlabeled image pairs and generates plans from initial to goal states in latent space [Asai et al., 2022].

However, a significant limitation of these techniques is their restricted applicability to real-world planning scenarios, which are often modeled as Markov decision processes with dynamic agent-environment interaction. These settings introduce complexities such as non-determinism, partial observability, and sub-goal reasoning, with objectives that go beyond simple final states. Furthermore, these methods compute heuristics based on a given model of the problem, which is an assumption we do not make.

Guiding reinforcement learning planners. Reinforcement learning is notorious for being

sample-inefficient, i.e., requiring a lot of agent-environment interactions before converging to a good policy. Warm-starting RL with offline methods has been a promising way of addressing these issues. Recent studies have shown the successful use of heuristic functions from classical planning in non-deterministic environments, particularly in reinforcement learning-based planning algorithms [Cheng et al., 2021]. Differently, the authors of [Nair et al., 2021] learn an initial policy offline and further fine-tune it online. The approaches discussed so far are not symbolic, and generate a function that maps states to their estimated values. However, these value estimates are often difficult for humans to interpret, heavily reliant on the training dataset, and unsuitable for human-in-the-loop applications.

In the RL literature, an alternative to modifying the policy function before fully online learning is augmenting the MDP's reward function. MDPs usually provide rewards in a Markovian manner, limiting their ability to capture temporally extended goals. Brafman et al. [2018] introduced a temporal logic framework for non-Markovian rewards, inspiring methods like reward machines [Camacho et al., 2019, Patel et al., 2021, Toro Icarte et al., 2022, Furelos-Blanco et al., 2023], which use automata to shape reward signals, and restraining bolts [De Giacomo et al., 2020, 2021], which enforce structured constraints. These methods have been extended to probabilistic reward machines [Dohmen et al., 2022] to model stochastic rewards.

RL taking advice dates back to the work of [Maclin and Shavlik, 1996] where they designed an imperative programming language which was used to express advice as if-else statements with loops. Their programs look similar in spirit to our actionable advice, but linear temporal logic is more expressive than their formalism. In another recent work, Andreas et al. [2017] proposed policy sketches. A policy sketch is a sequence of sub-goals given by the user to solve a task. They show how to use the sketch to compose (previously learned) policies for each sub-task to solve a novel task. Again, their advice language is quite constrained in comparison with $\mathsf{LTL_f}$ advice. Toro Icarte et al. [2018] recognized that $\mathsf{LTL_f}$ can be used as a general language to provide advice to RL agents and they demonstrated that in preliminary experiments by passing a temporal advice formulae to an R-MAX [Brafman and Tennenholtz, 2003] algorithm.

Coupling RL with symbolic planning is another way to speed up RL. Previous work attached a STRIPS planner to an RL agent, shaping its reward signal using the abstract plan knowledge which resulted in faster convergence and favourable scaling to bigger domains [Grounds and Kudenko, 2008, Grzes and Kudenko, 2008]. Lyu et al. [2019] developed symbolic deep reinforcement learning, by endowing RL with a planner-controller-meta-controller architecture, enabling hierarchical decision making. Illanes et al. [2020] treated symbolic plans as high-level instructions, enabling structured exploration while allowing agents to refine behaviors. While our approach draws partial inspiration from these methods, our work distinguishes itself by its ability to learn symbolic representations of low or high level actions. Furthermore, we don't assume we know the effects of the actions in the environment, thus not allowing us to represent the task in a planning formalism as in e.g. Illanes et al. [2020].

Some approaches prioritize safety by restricting actions rather than shaping the reward signal. Shielding [Alshiekh et al., 2017] monitors agent actions, correcting those that violate temporal logic-based safety specifications. Recent work extends this to uncertain and noisy environments with probabilistic logic shields [Yang et al., 2023]. Another approach, using pure past linear temporal logic, maintains the expressive power of shielding while reducing computational complexity [Varricchione et al., 2024]. These methods define safety properties manually rather than learning them, but they relate to our work as potential ways to guide planners, except that we strive to learn the specifications. Notably, all these works focus on high-level events, whereas we can also learn specifications at the action level.

Learning guiding mechanisms for RL planners. There has been work on learning the aforementioned guiding mechanisms such as reward machines [Rens and Raskin, 2020, Icarte et al., 2019, Neider et al., 2021] and probabilistic reward machines [Dohmen et al., 2022]. The learning approaches differ from the approach of this thesis as they focus on learning automata,

and employ tabular search or (some variant of) Angluin's L^* automata learning method [Angluin, 1987].

A close work to our appreach as they use ILASP to learn a guiding mechanism is Furelos-Blanco et al. [2021]. They introduce an interleaved automaton learning task, leveraging ILASP to infer subgoal automata from high-level event traces. While similar to reward machines (RMs), subgoal automata differ in that RM transitions incorporate both propositional formulae and a reward function. Their experiments show that the learned automaton achieves returns comparable to a hardcoded one across various environments. As mentioned earlier, temporal advice formulae are more general than subgoal automata and could be applied to a broader range of planning agent guiding approaches than these subgoal automata.

Learning low-level symbolic time-independent advice. One specific promising technique represents advice formulae as answer set programs and automatically synthesizes them using inductive learning. Meli et al. [2024] were the first to apply inductive learning from answer sets (ILASP) to learning advice formulae of a real-world partially-observable scenario. They showed that simple, interpretable, time-independent advice formulae in the form of answer set programs can be induced from executions of a partially observable markov decision process, leading to faster achievement of higher rewards. The paper thus provides a method for automatically learning domain knowledge and effectively applying it to automated planners.

Learning temporal logic formulae. In recent years, various approaches have emerged for learning temporal logic formulae from traces. Some studies have focused on the computational complexity of learning linear temporal logic formulae from traces [Fijalkow and Lagarde, 2021, Bordais et al., 2024], concluding that learning formulae of different fragments of linear, branching-time, and alternating-time temporal logic is NP-complete. Despite this, a wide range of practical approaches to learning these formulae exists. Noteworthy methods for learning $\mathsf{LTL_f}$ formulae include SAT-based techniques [Neider and Gavran, 2018], automata-based approaches [Camacho and McIlraith, 2021], and approximation-based anytime algorithms [Raha et al., 2022]. More recent methods even explore learning LTL formulae using GPUs [Valizadeh et al., 2024]. Additionally, some studies have focused on learning branching-time temporal logic [Roy and Neider, 2023, Pommellet et al., 2024].

Recently, Ielo et al. [2023] presents an approach which allows to learn specifications expressed in $\mathsf{LTL_f}$ which fit a set of finite system execution traces. The authors embedded the semantics of linear temporal logic on finite traces ($\mathsf{LTL_f}$) [De Giacomo and Vardi, 2013] in answer set programming and showed that ILASP can be used to learn $\mathsf{LTL_f}$ formulae in an efficient manner. We will use this work as a primary reference in this thesis due to its use of ILASP, making it compatible with our other main reference, Meli et al. [2024], and its open-source repository.

1.3 Contributions

This thesis offers the following contributions:

- We extend the work of Ielo et al. [2023] by offering a formal reduction proof that demonstrates $\mathsf{LTL_f}$ formulae can be learned using the ILASP software, something that was not included in their paper. To the best of our knowledge, this is the first detailed correctness proof of its kind, as other similar studies typically assert the correctness of their encoding or provide a proof without many details.

- We extend the work of Meli et al. [2024] by learning temporal advice formulae as opposed to time-independent advice formulae as in their research. In a similar vein, we extend the work of Toro Icarte et al. [2018] by automatically learning the type of formal advice they describe. To the best of our knowledge, this thesis is the first work to describe the learning of temporal advice.

- We extend Ielo et al. [2023] by applying their approach to model planning domains and learn temporal advice formulae, while also exploring noisy domains and the effects of penalties. This expands on a research direction mentioned in their paper. Additionally, unlike Ielo et al. [2023], who only assessed computational efficiency, we evaluate the quality of the learned formulae, focusing on their applicability to guiding planners.

- This thesis provides additional documentation for the relatively new ILASP software, demonstrating its usefulness through application to new domains. Due to ILASP's academic, closed-source nature and limited documentation, working with it presented various challenges. We hope this thesis, particularly the experiments in Chapter 5 that examine the impact of specific design choices modeling a problem in ILASP, will be a valuable resource for future ILASP users. Specifically, we evaluate our approach on experiments in two environments: a newly introduced simple gem pickup environment and one more ecomplex environment with a long planning horizon, RockSample.

- To facilitate further research, we provide a public code repository[1] which is extendable to new enviroments and experiments. The repository holds the reproducible experiments of this thesis and detailed documentation (complementing this thesis). We also include a detailed section on potential future extensions of this thesis (Section 6.2).

[1]The repository can be accessed at `https://gitlab.com/p-skaisgiris/temporal-advice-ilasp`. Going forward, we will omit this link and refer to the files in this code repository unless stated otherwise.

Chapter 2

Preliminaries

This chapter provides the foundational concepts necessary to understand this thesis. We introduce linear temporal logic on finite traces, answer set programming, inductive learning of answer set programs, and planning in Markov decision processes, each serving as a crucial building block for our proposed approach.

2.1 Linear temporal logic on finite traces

Linear temporal logic is used to reason about a single computation or run of a system. One can express and verify temporal properties of systems [Lamport, 1983]. Note that as well linear temporal logic is used to represent the order in which things happen, not the actual time at which it happens [Lamport, 1983].

In this thesis, among other things, we will extend an existing approach to learning time-independent advice and attempt to learn advice which may involve a time element as well. As you shall later see, we will do so by learning advice in the form of linear temporal logic on finite traces [Pnueli, 1977, De Giacomo and Vardi, 2013] formulae. We will now familiarize ourselves with linear temporal logic on finite traces ($\mathsf{LTL_f}$). First, let us start with the symbols with which one can describe a formula in this logic.

Definition 2.1.1 (Syntax of Linear Temporal Logic on Finite Traces) Let $\mathcal{P}$ be the set of propositional atoms. Then, $\mathsf{LTL_f}$ formulae are recursively defined as

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \mathsf{X}\varphi \mid \mathsf{F}\varphi \mid \varphi\mathsf{U}\psi \mid \mathsf{G}\varphi$$

where $\top$ is the logical truth symbol, $p \in \mathcal{P}$, φ and ψ are complex (not necessarily propositional) formulae. Following those, are formulae with temporal operators X (next), F (eventually), U (until), G (always).

Now that we can form $\mathsf{LTL_f}$ formulae, it is only natural to try to understand what the bundle of symbols representing an LTL formula mean.

Definition 2.1.2 (Models of $\mathsf{LTL_f}$) Let $\mathcal{P}$ be the set of propositional symbols. A *model* of $\mathsf{LTL_f}$ over variables in $\mathcal{P}$ is a sequence $\lambda = \lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ associating to each $i \in \{0, \ldots, n-1\}$ a state $\lambda_i \subseteq \mathcal{P}$, consisting of the set of all propositional variables that are assumed to hold at the instant i. Such models are called *finite traces* (often simply called traces), and they represent the sets of propositions that are true at consecutive time instants.

We will use the following notation when talking about traces:

- $|\lambda|$ is the length of the trace

- λ, i is the ith time moment on a trace (note that the first time moment is $i = 0$). We may also denote this as $\lambda[i]$.
- $last(\lambda) = |\lambda| - 1$ is the last time moment of the trace
- $time(\lambda) = \{0, \ldots, |\lambda| - 1\}$

Formulae are made true on their models. The specific way by which a formula is made true on its corresponding model is described by semantics.

Definition 2.1.3 (Semantics of $\mathsf{LTL_f}$) Let λ be a finite trace and φ an $\mathsf{LTL_f}$ formula. With $\lambda, i \models \varphi$ we denote that a formula φ holds/is made true on a trace λ at the time instant $i \in time(\lambda)$. We inductively define the semantics of the subformulae as follows:

$\lambda, i \models p$ for $p \in \mathcal{P}$, iff $p \in \lambda_i$;
$\lambda, i \models \neg\varphi$ iff $\lambda, i \not\models \varphi$;
$\lambda, i \models \varphi \wedge \psi$ iff $\lambda, i \models \varphi$ and $\lambda, i \models \psi$;
$\lambda, i \models \varphi \vee \psi$ iff $\lambda, i \models \varphi$ or $\lambda, i \models \psi$;
$\lambda, i \models \mathsf{X}(\varphi)$ iff $i < last(\lambda)$ and $\lambda, i + 1 \models \varphi$;
$\lambda, i \models \mathsf{F}(\varphi)$ iff $\exists j$ with $i \leqslant j \leqslant last(\lambda)$ such that $\lambda, j \models \varphi$;
$\lambda, i \models \mathsf{G}(\varphi)$ iff $\forall j$ with $i \leqslant j \leqslant last(\lambda)$ such that $\lambda, j \models \varphi$;
$\lambda, i \models (\varphi \mathsf{U} \psi)$ iff $\exists j$ with $i \leqslant j \leqslant last(\lambda)$ s.t. $\lambda, j \models \psi$ and $\forall k$ with $i \leqslant k < j$ s.t. $\lambda, k \models \varphi$.

When $\lambda, 0 \models \varphi$ (also written as $\lambda \models \varphi$) we say that λ is a *model* of φ and the formula φ *satisfies* the trace λ.

Example 2.1.4 Let λ be the following finite trace $\lambda = s_0, s_1, s_2, s_3$ where each s_i is represented by the set of propositions true in that state. Concretely, let

$$\lambda = \{p\}, \{q\}, \{p, q\}, \{r\}$$

This trace represents a scenario where, according to the defined $\mathsf{LTL_f}$ semantics, only p is true at the first timestep, only q is true at the second timestep, both p and q are true on the third timestep, etc. The following are true on this trace:

- $\lambda, 0 \models \mathsf{F}r$ because there exists a time instance (namely, s_4) at which r is true
- $\lambda, 0 \models \mathsf{X}q$ because $\lambda, 1 \models q$
- $\lambda, 0 \models \mathsf{G}(p \rightarrow \mathsf{F}r)$ this is true because from every state where p is true (s_0 and s_2), r eventually becomes true as well

2.2 Answer set programming

Answer set programming is a declarative form of programming. This means that, instead of telling the computer *how* to solve the problem, we simply describe *what* the problem is, and leave the search of a solution to the computer [Gebser et al., 2012]. ASP was designed to solve difficult, NP-hard search problems [Lifschitz, 2008]. It allows developers to encode a problem as a set of logical rules, where the solutions correspond to the "answer sets" or stable models that satisfy these rules [Lifschitz, 2008]. In this work, we will use `clingo` [Gebser et al., 2019] as the ASP grounder and solver. In the following parts of this section, we will introduce the basic notions of

ASP, closely following the definition formulations found in Lifschitz [1999], Law et al. [2018a,b], and provide some illustrative examples.

Definition 2.2.1 (Terms, atoms, literals)

- An integer, symbolic constant (e.g. `alice`, `cat`), or a variable (e.g. `X`, `Book`) is a *term*. If f is a symbolic constant and $(t_1, \ldots, t_n)$ is a tuple of terms, then $f(t_1, \ldots, t_n)$ is also a term.
- An *atom* is an expression of the form `p(x_1, ..., x_n)` where `p` is a predicate symbol of arity n (taking n variables) and `x_1, ..., x_n` are terms.
- A *literal* is a positive or a negative atom. For instance, if `a` is a positive atom, then its negation is `not a`, a negative atom.

Answer set programs P are made up of rules. When writing such rules, we use literals. There are three kinds of rules: normal rules, choice rules, and weak constraints. This thesis focuses exclusively on normal rules and their special cases within answer set programming, so we omit the introduction of choice rules and weak constraints. We acknowledge that this may be then called a restricted fragment of ASP, but we will nonetheless refer to these programs simply as "answer set programs" throughout this work for brevity and clarity.

Definition 2.2.2 (Normal rule) *Normal rules* (also simply called rules, definite clauses, or clauses) are statements of the form

$$a \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m$$

where $m, n \geqslant 0$, a is a literal, $b_1, \ldots, b_n$ are positive literals and $\neg c_1, \ldots, \neg c_m$ are negative literals. In a normal rule r, the set of literals appearing on the left side of the $\leftarrow$ symbol is referred to as the *head* of the rule. Conversely, the set of literals appearing on the right side of the $\leftarrow$ symbol constitutes the *body* of the rule.

Note that in ASP it is possible to have multiple literals in the head of the rule which would denote disjunction, but in this thesis we only place a single literal in the head and $head(r)$ can return a set with at most one element.

Let $body(r)$ be the body of a rule r, and let $head(r)$ be the head of the rule r. Let $body(r)^+$ be the set of positive literals in the body, and $body(r)^-$ be the set of negative literals in the body. A rule r is said to be positive if $body(r)^- = \emptyset$.

In ASP syntax, the above normal rule could be written as

```
a :- b_1, ..., b_n, not c_1, ..., not c_m.
```

Definition 2.2.3 (Hard constraint) A normal rule r is called a *hard constraint* if $head(r) = \emptyset$. Consider the following example, where $body(r) = innocent(sally) \wedge motive(sally)$, it would be written in ASP as

```
:- innocent(sally), motive(sally).
```

The above program means that it cannot be the case that both `innocent(sally)` and `motivy(sally)` are true at the same time. That is, by adding a constraint, we remove some of the program's models (see def. below).

Definition 2.2.4 (Fact) A normal rule r for which $body(r) = \emptyset$ is called a *fact*. In that case, the head is simply true, appears in the model. Let $head(r) = \{motive(sally)\}$. In ASP this is written as

```
motive(sally).
```

ASP programs often use variables to express general knowledge. *Grounding* replaces these variables with all possible constants from the program's domain, creating a set of variable-free rules. There exist software tools which perform grounding automatically, such as `clingo` [Gebser et al., 2019] which we use in this thesis.

Definition 2.2.5 (Grounding, Herbrand base, and Herbrand interpretations)

- A rule or an atom is said to be *grounded* if it does not contain any variables.
- The *Herbrand Base* of an ASP program P, denoted $HB(P)$, is the set of all ground (variable free) atoms that can be formed from predicates and constants in P.
- A set of grounded atoms $I \subseteq HB(P)$ is called a *(Herbrand) interpretation* of P.

Example 2.2.6 Consider the following program (Example from https://potassco.org/clingo/run/ called "Harry and Sally"):

```
1 motive(harry).
2 motive(sally).
3 guilty(harry).
4 innocent(Suspect) :- motive(Suspect), not guilty(Suspect).
```

Since there's only one variable `Suspect`, the grounding process would replace it with the symbolic constants `harry` and `sally` from the program to arrive at the following ground rules:

```
1 motive(harry).
2 motive(sally).
3 guilty(harry).
4 innocent(harry) :- motive(harry), not guilty(harry).
5 innocent(sally) :- motive(sally), not guilty(sally).
```

Definition 2.2.7 (Safe rule) A rule is said to be *safe* if all the variables in the rule's head also appear in a positive literal in the rule's body. This ensures that the variables can be instantiated with concrete values during the grounding process and that the grounding is finite. Each rule in an ASP program must be safe for it to be a model (see def. below).

This thesis focuses solely on safe ASP rules. While considering unsafe rules may be relevant for theoretical investigations of ASP, they are of no relevance here in our thesis which focuses on the applications of ASP.

Example 2.2.8 The following is an example of a safe normal rule as the only variable Y appearing in the head also appear in the body in a positive literal $edge(X, Y)$.

```
reachable(Y) :- edge(X, Y), reachable(X).
```

The following is an example of an unsafe rule.

```
reachable(Y) :- not visited(Y).
```

The variable Y appears only in the negative literal `not visited(Y)`. This makes the rule unsafe,

as there is no way to determine the possible values of Y.

Let us now turn from syntactic representation of ASP programs to their evaluation.

Definition 2.2.9 (Derivation, model of a program) Let r be a rule and a Herbrand interpretation $I \subseteq HB(P)$. We say that $head(r)$ is *derived* from I (we drop "by I" when it's clear from context) whenever $body(r)^+ \subseteq I$ and $body(r)^- \cap I = \emptyset$.

A Herbrand interpretation $I \subseteq HB(P)$ is a *model* of a program P if and only if for every rule $r \in P$ which is not a hard constraint, $head(r) \in I$ whenever $head(r)$ is derived by I.

Referring back to the ASP example in Definition 2.2.2, `a` will be in a model only when all of `b_1` through `b_n` are in the model and `c_1` through `c_n` are not in the model.

Definition 2.2.10 (Reduct of a program) Given a program P and a Herbrand interpretation $I \subseteq HB(P)$, the *reduct* P^I is constructed from the set of ground instances of rules in P in two steps:

1. Remove rules whose bodies contain the negation of an atom in I
2. Remove all negative literals from the remaining rules

The semantics of ASP programs P are defined in terms of answer sets.

Definition 2.2.11 (Answer sets) Any $I \subseteq HB(P)$ is an *answer set* of P if it is the minimal model of the reduct P^I, that is, there is no strict subset of I that is a model of P^I. The set of all answer sets of a program P will be denoted as $AS(P)$.

Example 2.2.12 Recall the Example 2.2.6 and its grounded program P. Here, we provide the steps how to work out the answer set for this program. In this case, the Herbrand base is as follows:

$$HB(P) = \{\texttt{motive(harry)}, \texttt{motive(sally)}, \texttt{guilty(harry)}, \texttt{innocent(harry)}, \texttt{innocent(sally)}\}$$

Now, we must pick $I \subseteq HB(P)$ to be the candidate answer set. Note that the first three lines of the program are provided as facts, so they will definitely be in the interpretation. Let us then set $I_1 = \{\texttt{motive(harry)}, \texttt{motive(sally)}, \texttt{guilty(harry)}\}$ and compute the reduct P^I:

1. Since `guilty(harry)` $\in I$, we remove line 4.
2. We remove `not guilty(sally)` from line 5, resulting in a rule r:
   `innocent(sally) :- motive(sally).`

As per Definition 2.2.9, I_1 is not a model of P^{I_1} since $body(r)^+ = \{\texttt{motive(sally)}\} \subseteq I_1$ but $head(r) = \texttt{innocent(sally)} \notin I_1$. If we set $I_2 = I_1 \cup \{\texttt{innocent(sally)}\}$, then I_2 is a model for P^{I_2} and since we added only the necessary grounded atoms to it, it is also an answer set of P.

If we added the hard constraint from Definition 2.2.3 to this program, we would rule out the only answer set and be left with no available answer sets for this program.

2.3 Inductive learning of answer set programs

Inductive Logic Programming (ILP) is a machine learning subfield that constructs first-order logic theories from examples and background knowledge [Muggleton, 1993, Cropper and Dumančić, 2022]. Unlike empirical learning frameworks, ILP uses first-order relational logic combined with domain knowledge to induce a set of logical rules (a hypothesis) that generalizes training examples. The goal is to induce a hypothesis that, along with the provided background knowledge, logically entails as many positive examples and as few negative examples as possible [Cropper and Dumančić, 2022]. ILP represents data as logic programs rather than, e.g. tables.

Traditional ILP approaches primarily focus on learning programs of Prolog [Clocksin and Mellish, 2003], the classic procedural logic programming language. Prolog is query-oriented and procedural, whereas answer set programming, introduced in Section 2.2, takes a declarative approach that generates and explores stable models to solve problems. This makes ASP more suitable for addressing a wide range of search problems, including searching for a hypothesis of a learning problem. Based on ILP and ASP, we now consider learning from answer sets (LAS) and introduce ILASP (Inductive Learning of Answer Set Programs) [Law et al., 2020b].

ILASP is a collection of algorithms for solving LAS tasks [Law et al., 2018b]. Unlike many other ILP systems, ILASP guarantees optimal solutions by transforming the task into a meta-level ASP program, iteratively solved until optimal answer sets correspond to solutions [Law et al., 2018b]. Furthermore, ILASP can generate programs with features such as choice rules as well as hard and weak constraints, which are not supported in Prolog. Each iteration of the ILASP system has incorporated new features, with the concept of partial interpretations–used to represent examples within the framework–being a core element since its initial version [Law et al., 2014].

Definition 2.3.1 (Partial interpretation) A *partial interpretation* example, e_{pi}, is a pair of sets of ground atoms $\left\langle e_{pi}^{inc}, e_{pi}^{exc} \right\rangle$. The first element in the tuple is called the inclusion set and the second is called the exclusion set. An interpretation (Definition 2.2.5) I of a program P is said to *extend* e_{pi} iff $e_{pi}^{inc} \subseteq I$ and $e_{pi}^{exc} \cap I = \emptyset$.

A partial interpretation e_{pi} is *bravely accepted* by a program P if and only if there exists an answer set $A \in AS(P)$ such that A extends e_{pi}. A partial interpretation e_{pi} is *cautiously accepted* by a program P if and only if every $A \in AS(P)$ extends e_{pi} [Law et al., 2014].

ILASP2i addresses the limitation of the first ILASP versions by allowing background knowledge to vary based on the context of each example, enabling more nuanced explanations. This context-dependent learning, implemented through an iterative algorithm, significantly improves scalability, achieving up to two orders of magnitude faster processing and reduced memory usage compared to ILASP2 [Law et al., 2016].

Definition 2.3.2 (Context-dependent partial interpretation (CDPI)) A *context-dependent partial interpretation* example e_{cdpi} is a pair $\langle e_{pi}, e_{ctx} \rangle$, where e_{pi} is a partial interpretation and e_{ctx} is a program called the context of e.

A program P is said to *bravely accept* e if there is at least one answer set A of $P \cup e_{ctx}$ that extends e_{pi}, such an A is called an accepting answer set of P w.r.t. e. In essence, a CDPI requires the learned program, when combined with the context of an example e, to bravely entail all inclusion atoms e_{pi}^{inc} and no exclusion atoms e_{pi}^{exc} [Law et al., 2018b]. A program P is said to cautiously accept e if all answer sets $A \in AS(P \cup e_{ctx})$ extend e_{pi}.

Having formalized examples as CDPIs, we now present the core learning task solved by ILASP.

Definition 2.3.3 (Learning from context-dependent answer set programs) An $\text{ILP}^{context}_{LAS}$ task is a tuple of the form $T = \langle B, \mathcal{H}, \langle E^+, E^- \rangle \rangle$, where B is an ASP program representing the background knowledge, $\mathcal{H}$ is a set of ASP rules called the hypothesis space, E^+ and E^- are sets of CDPIs, called positive and negative examples respectively. A hypothesis $H \subseteq \mathcal{H}$ is an inductive solution of T (written $H \in \text{ILP}^{context}_{LAS}(T)$) if and only if:

1. $\forall \langle e^+, C \rangle \in E^+, \exists A \in AS(B \cup C \cup H)$ s.t. A extends e^+, and
2. $\forall \langle e^-, C \rangle \in E^-, \nexists A \in AS(B \cup C \cup H)$ s.t. A extends e^-.

A hypothesis H is an *optimal inductive solution* of T if and only if there is no inductive solution H' of T such that $|H'| < |H|$.

Note that the positive examples for an inductive solution must each be bravely accepted, while the negative examples must each be cautiously accepted.

So far we assumed perfectly labeled examples requiring full coverage by the learned program. This is often unrealistic in real-world applications where noisy or mislabeled examples may necessitate partial coverage to avoid overfitting. ILASP3 introduced a noise-tolerant extension of the previous frameworks, enabling the learning of complex knowledge from noisy data (represented by weighted context-dependent partial interpretations) through a balance of coverage and complexity to mimic robust human-like inductive reasoning [Law et al., 2018b].

Definition 2.3.4 (Weighted context-dependent partial interpretation) A *weighted context-dependent partial interpretation* e is a tuple $\langle e_{id}, e_{pen}, e_{cdpi} \rangle$, where e_{id} is a constant, called the identifier of e (unique to each example), e_{pen} is the penalty of e and e_{cdpi} is a CDPI. The penalty e_{pen} is either a positive integer, or ∞. A program P bravely/cautiously accepts e iff it bravely/cautiously accepts e_{cdpi}.

Definition 2.3.5 (Learning from noisy and context-dependent answer set programs) An ILP^{noise}_{LAS} task is a tuple of the form $T = \langle B, \mathcal{H}, \langle E^+, E^- \rangle \rangle$, where B is an ASP program representing the background knowledge, $\mathcal{H}$ is a set of ASP rules called the hypothesis space, E^+ and E^- are sets of weighted CDPIs, called positive and negative examples respectively.

Let $uncov(H, T)$ be the set consisting of all examples $e \in E^+$ (resp. E^-) such that $B \cup H$ does not bravely accept (resp. bravely accepts) e. The penalty of H is denoted as $pen(H, T) = \sum_{e \in uncov(H,T)} e_{pen}$. The score of H is denoted as $S(H, T) = |H| + pen(H, T)$.

A hypothesis $H \subseteq \mathcal{H}$ is an *inductive solution* of T (written $H \in \text{ILP}^{noise}_{LAS}(T)$) if and only if $S(H, T)$ is finite. H is an *optimal inductive solution* of T if and only if $S(H, T)$ is finite and $\nexists H' \subseteq \mathcal{H}$ such that $S(H', T) < S(H, T)$.

Examples with infinite penalties must be covered by any inductive solution, as any hypothesis failing to do so receives an infinite score. An ILP^{noise}_{LAS} task T is considered satisfiable if $\text{ILP}^{noise}_{LAS}(T)$ is non-empty (has at least one hypothesis which fits all the above criteria); otherwise, if $\text{ILP}^{noise}_{LAS}(T)$ is empty, T is considered unsatisfiable.

Note that, differently than in Law et al. [2018b], the definitions for the inductive learning tasks omit the orderings of context-dependent examples. This is done simply because in this thesis we did not make use of ordering the examples.

2.4 Planning in Markov decision processes

Due to the inherent chaos of the real world, planning is inherently difficult. A formal model serves as a crucial starting point for managing this unpredictability. Markov decision processes provide

a mathematical framework for modeling sequential decision-making in uncertain environments [Ghallab et al., 2004]. In this framework, an agent interacts with its environment over discrete time steps by receiving a state representation, selecting an action, and subsequently obtaining a reward along with a transition to a new state [Sutton and Barto, 2020]. This process is visually depicted in Figure 2.1. MDPs represent interactions with the environment as a series of probabilistic state transitions, where actions lead to non-deterministic outcomes. The core goal of this approach is to find the best actions to take given a state in order to maximize the total expected reward. This usually involves learning how to balance immediate reward with delayed rewards.

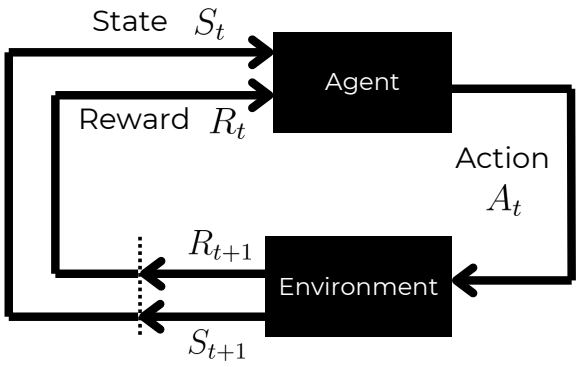


Figure 2.1: The agent-environment interaction in a Markov decision process.

Definition 2.4.1 (Markov Decision Process (MDP)) A *Markov decision process* is a 5-tuple (S, A, P, R, γ) where:

- S is the set of states,
- A is the set of actions,
- P is the state transition model $P : S \times A \to \Pi(S)$ where $\Pi(S)$ is the probability distribution defined over states,
- R is the reward function $R : S \times A \to \mathbb{R}$,
- γ is the discount factor $\gamma \in [0, 1]$.

The key unmentioned assumptions of an MDP, as outlined in Sutton and Barto [2020], are as follows:

1. Markov Property: Transitions depend solely on the current state and action.
2. Stationarity: Transition and reward functions are time-invariant (unless time is a state variable).

A complete sequence of interactions between an agent and its environment is called an episode, the sum of all rewards in an episode is called a return. The objective of learning to plan in an MDP is to use some algorithm to derive the policy function $\pi : S \to A$ that enables the agent to maximize the expected total discounted return, given by $E\left[\sum_{t=0}^{K} \gamma^t \cdot R(s_t, a_t)\right]$, where K represents the episode length.

In standard decision-making, MDPs assume an agent always knows the environment's exact state. However, many real-world scenarios involve uncertainty. Therefore, partially observable MDPs (POMDPs) are formal models which handle incomplete environment knowledge. Agents in POMDPs use imperfect sensors, receiving partial observations and maintaining a belief state

(probability distribution) over possible states [Ghallab et al., 2004]. This replaces direct state observation and state-to-action policies with observation history/belief state-to-action policies. This change allows to model information gathering. While MDPs are simpler, POMDPs are more realistic but computationally complex.

Definition 2.4.2 (Partially observable MDP (POMDP)) A *partially observable Markov decision process* is a 7-tuple $(S, A, P, R, Z, O, \gamma)$ where:

- S is the set of states
- A is the set of actions
- P is the state transition model $P : S \times A \to \Pi(S)$ where $\Pi(S)$ is the probability distribution defined over states.
- R is the reward function $R : S \times A \to \mathbb{R}$
- Z is a set of observations
- $O : S \times A \to \Pi(Z)$ is the observation model
- γ is the discount factor $\gamma \in [0, 1]$

Following Toro Icarte et al. [2018], we define versions of MDPs and POMDPs that incorporate a signature. Doing so allows us to establish a formal model for describing these decision processes using symbolic predicates. These versions play a key role in defining our learning task for temporal advice formulae.

Definition 2.4.3 (Signature) Let $\mathcal{S} = \langle \Omega, C, arity \rangle$ be a *signature* where Ω is a finite set of predicate symbols, C is a finite set of constant symbols, and $arity : \Omega \to \mathbb{N}$ assigns an arity to each predicate.

Definition 2.4.4 (Ground atoms, literals, and the truth assignment of a signature) Let $\mathcal{S}$ be a signature.

- $GA(\mathcal{S}) = \{P(c_1, \ldots, c_{arity(P)}) \mid P \in \Omega, c_i \in C\}$ is the set of all *ground atoms* of $\mathcal{S}$.
- $\mathsf{lit}(\mathcal{S}) = GA(\mathcal{S}) \cup \{\neg p \mid p \in GA(\mathcal{S})\}$ is the set of *ground literals* of $\mathcal{S}$.
- The *truth assignment* of $\mathcal{S}$ is $\tau \subseteq \mathsf{lit}(\mathcal{S})$ s.t. for every $a \in GA(\mathcal{S})$ exactly one of a or $\neg a$ is in τ. Let $T(\mathcal{S})$ denote the set of all truth assignments of $\mathcal{S}$.

Definition 2.4.5 (MDP with a signature) An *MDP with a signature* is a 7-tuple $(S, A, P, R, \gamma, \mathcal{S}, L)$ where:

- (S, A, P, R, γ) are the standard MDP components (Definition 2.4.1).
- $\mathcal{S} = \langle \Omega, C, arity \rangle$ is a signature where $\Omega = A \cup E$ is the set of predicates, where A are the agent's actions in the MDP and E are environment predicates describing the state.
- $L : S \to T(\mathcal{S})$ is a labeling function that assigns each state a truth assignment of $\mathcal{S}$.

Definition 2.4.6 (POMDP with a signature) An *MDP with a signature* is a 9-tuple $(S, A, P, R, Z, O, \gamma, \mathcal{S}, L)$ where:

- $(S, A, P, R, Z, O, \gamma)$ are the standard POMDP components (Definition 2.4.2).
- $\mathcal{S} = \langle \Omega, C, arity \rangle$ is a signature where $\Omega = A \cup E$ is the set of predicates, where A are the agent's actions in the POMDP and E are environment predicates describing the state.
- $L : S \to T(\mathcal{S})$ is a labeling function that assigns each state a truth assignment of $\mathcal{S}$.

Chapter 3

Passive learning of $\mathsf{LTL_f}$ formulae using ILASP

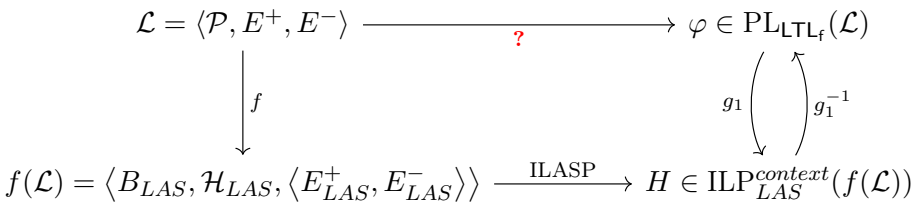


Figure 3.1: The high-level approach of this chapter.

The main goal of this thesis is to learn temporal advice formulae from positive and negative examples. Linear temporal logic on finite traces is an effective formalism for expressing such formulae, as noted by Toro Icarte et al. [2018]. To learn $\mathsf{LTL_f}$ formulae in practice, we need an algorithm. Rather than developing a new one, we reduce the $\mathrm{PL}_{\mathsf{LTL_f}}$ problem to an $\mathrm{ILP}^{context}_{LAS}$problem, enabling us to use the existing ILASP software [Law et al., 2020b] as such algorithm.

This chapter provides a rigorous proof of the ILASP learning task's correctness for $\mathsf{LTL_f}$ which is missing in Ielo et al. [2023]. The proof formally demonstrates that the $\mathsf{LTL_f}$ learning setup can be correctly converted into an $\mathrm{ILP}^{context}_{LAS}$ instance, ensuring that the solutions to the original problem align with those of the transformed setup. The high-level idea is illustrated in Figure 3.1.

This chapter is structured as follows:

- Section 3.1 formally introduces the $\mathsf{LTL_f}$ formula learning task.
- Section 3.2 demonstrates how Ielo et al. [2023] implemented $\mathsf{LTL_f}$ model checking in ASP which is central to the proofs in the next section.
- Section 3.3 formally shows that we can reduce the formal $\mathsf{LTL_f}$ learning problem to a formal ILASP learning problem, meaning that we can indeed use the ILASP algorithm to learn $\mathsf{LTL_f}$ formulae correctly in practice.
- Section 3.4 introduces a formal $\mathsf{LTL_f}$ framework that relaxes the requirement that all examples must be covered. It also discusses optimal solutions for both the formal $\mathsf{LTL_f}$ learning task and the corresponding ILASP task.

This chapter focuses on fitting linear traces and learning $\mathsf{LTL_f}$ formulae, and does not address temporal advice, which will be discussed in the next chapter. We emphasize learning $\mathsf{LTL_f}$ formulae because temporal advice formulae, as shown in the next chapter, are an extension of

$\mathsf{LTL_f}$. Therefore, it is crucial to ensure that we begin with a formally correct encoding before extending it.

3.1 Passive learning of $\mathsf{LTL_f}$ formulae

We present the formalization the task of passive learning of $\mathsf{LTL_f}$ formulae, denoted $\mathrm{PL}_{\mathsf{LTL_f}}$, as originally mentioned in Neider and Gavran [2018] and explicitly formalized in Ielo et al. [2023]. The term "passive" distinguishes this approach from "active" learning, where the learning system can query an oracle for additional data or guidance [Neider and Gavran, 2018].

Definition 3.1.1 (Passive learning task $\mathrm{PL}_{\mathsf{LTL_f}}$) Let $\mathcal{P}$ be a set of propositional symbols. A $\mathrm{PL}_{\mathsf{LTL_f}}$ passive learning task is a tuple $\mathcal{L} = \langle \mathcal{P}, E^+, E^- \rangle$ where the set of positive traces E^+ and the set of negative traces E^- are defined over $\mathcal{P}$ and are such that $E^+ \cap E^- = \emptyset$. An $\mathsf{LTL_f}$ formula φ written in $\mathcal{P}$ is an *inductive solution* of a $\mathrm{PL}_{\mathsf{LTL_f}}$ task (denoted $\varphi \in \mathrm{PL}_{\mathsf{LTL_f}}(\mathcal{L})$) if and only if the following hold:

1. $\forall \lambda \in E^+, \lambda \models \varphi$, and
2. $\forall \lambda \in E^-, \lambda \not\models \varphi$.

3.2 Encoding $\mathsf{LTL_f}$ model checking in ASP

Since $\mathrm{ILP}^{context}_{LAS}$ operates on ASPs, a reduction from $\mathrm{PL}_{\mathsf{LTL_f}}$ to $\mathrm{ILP}^{context}_{LAS}$ requires a way to evaluate $\mathsf{LTL_f}$ formulae on linear traces in ASP. The necessary components for any model checking tool are the models, the formulae, and the rules of the semantics. We present these components along with some extra constraints on the used objects, encoded in ASP as Ielo et al. [2023] did.

3.2.1 Encoding traces

Any linear trace can be encoded in ASP semantics using the predicates `trace/1` and `trace/2`. The first predicate is used for "connecting" the trace, and the second predicate is used for specifying which proposition is true at that timestep.

Example 3.2.1 Example 1 from Ielo et al. [2023]. Consider the trace $\lambda = \{a\}, \{a, b\}, \{\}$. It can be encoded in ASP as follows:

```
1 trace(0).
2 trace(0, a).
3 trace(1).
4 trace(1, a).
5 trace(1, b).
6 trace(2).
```

3.2.2 Encoding formulae

Ielo et al. [2023] represent $\mathsf{LTL_f}$ formulae as syntax trees (STs), similarly as Neider and Gavran [2018] have done. The ST is encoded in ASP using the predicates `label/2` and `edge/2`. Each syntax tree is made up of labeled nodes, and they are connected in a specific way using edges. Take note to not mistake the syntax tree nodes with the timesteps of the traces as they represent completely different objects. Each node is annotated with a label, drawn from the set $\mathcal{O} \cup \mathcal{P}$ thereby ensuring the syntax tree represents a meaningful expression. Specifically, $\mathcal{O}$ is a set of propositional logic operators or temporal operators, i.e. $\mathcal{O} = \{\texttt{neg}, \texttt{and}, \texttt{or}, \texttt{implies}, \texttt{next}, \texttt{eventually}, \texttt{always}, \texttt{until}\}$.

Example 3.2.2 The following temporal logic specification $\mathsf{G}(a \rightarrow (\mathsf{F}b))$ which denotes that it is always the case that if event a takes place, then eventually b will also take place, can be encoded in ASP as follows:

```
label(1, always).
edge(1,2).
label(2, implies).
edge(2, 3).
edge(2, 4).
label(3, a).
label(4, eventually).
label(5, b).
```

Now, we present a function which converts any $\mathsf{LTL_f}$ formula its corresponding ST expressed as an answer set program.

Definition 3.2.3 (Encoding $\mathsf{LTL_f}$ formulae in ASP) Given an arbitrary $\mathsf{LTL_f}$ formula φ, we define a recursive transformation function g_1 that converts such formulae into their ASP syntax tree representation. In the following, let $n, m_1, m_2 \in \mathbb{N}^+$. For each recursive step, we choose a fresh value for m_1 (and m_2) that has not been used previously in the recursion, ensuring that it is incremented by exactly one and satisfies $m_1 < m_2$.

$$
\begin{aligned}
g_n(p) &= \{\texttt{label}(n,\texttt{p})\} \text{ where } p \in \mathcal{P} \\
g_n(\neg\psi) &= \{\texttt{label}(n,\texttt{neg}), \texttt{edge}(n,m_1)\} \cup g_{m_1}(\psi) \\
g_n(\psi \wedge \sigma) &= \{\texttt{label}(n,\texttt{and}), \texttt{edge}(n,m_1), \texttt{edge}(n,m_2)\} \cup g_{m_1}(\psi) \cup g_{m_2}(\sigma) \\
g_n(\psi \vee \sigma) &= \{\texttt{label}(n,\texttt{or}), \texttt{edge}(n,m_1), \texttt{edge}(n,m_2)\} \cup g_{m_1}(\psi) \cup g_{m_2}(\sigma) \\
g_n(\psi \rightarrow \sigma) &= \{\texttt{label}(n,\texttt{implies}), \texttt{edge}(n,m_1), \texttt{edge}(n,m_2)\} \cup g_{m_1}(\psi) \cup g_{m_2}(\sigma) \\
g_n(\mathsf{X}\psi) &= \{\texttt{label}(n,\texttt{next}), \texttt{edge}(n,m_1)\} \cup g_{m_1}(\psi) \\
g_n(\mathsf{F}\psi) &= \{\texttt{label}(n,\texttt{eventually}), \texttt{edge}(n,m_1)\} \cup g_{m_1}(\psi) \\
g_n(\mathsf{G}\psi) &= \{\texttt{label}(n,\texttt{always}), \texttt{edge}(n,m_1)\} \cup g_{m_1}(\psi) \\
g_n(\psi \mathsf{U} \sigma) &= \{\texttt{label}(n,\texttt{until}), \texttt{edge}(n,m_1), \texttt{edge}(n,m_2)\} \cup g_{m_1}(\psi) \cup g_{m_2}(\sigma)
\end{aligned}
$$

Thus, to transform an arbitrary $\mathsf{LTL_f}$ formula φ to ASP, we would apply $g_1(\varphi)$ where 1 is fixed and refers to the root node index of the syntax tree.

3.2.3 Encoding the structural validity of a syntax tree

In the previous subsection, we outlined an approach to encoding $\mathsf{LTL_f}$ formulae in the ASP formalism. To avoid expressing or learning ill-formed syntax trees which would be ambiguous when translated back into $\mathsf{LTL_f}$ formulae, we must enforce constraints.

Appendix A outlines a running example and explains the necessity of each constraint in detail. Here, we present the $\Sigma^{\mathcal{P}}_{constr}$ program as introduced in Ielo et al. [2023], which lists all constraints in full. The comments clarify the purpose of each code block, and these constraints are used in the reduction proof in the next section.

Listing 3.1: Program $\Sigma^{\mathcal{P}}_{constr}$ showing the constraints of a structural validity of a syntax tree.

```
% For every p in propositions
proposition(p).
% A node is a term in edge/2 or something that is labeled via label/2
node(X) :- label(X,_).
node(X) :- edge(_,X).
node(X) :- edge(X,_).
% Remove redundant solutions (e.g. 1,2,5)
```

```
% by ensuring that if node X+1 is defined, then so is X
node(X) :- node(X+1), X >= 1.
% The DAG must be connected
reach(1).
reach(T) :- edge(R,T), reach(R).
:- node(X), not reach(X).
:- node(X), not edge(_,X), X > 1.
% At most two edges from the same node
:- node(X), 3 <= #count { Z: edge(X,Z) }.
% Each node must be labeled
:- node(X), not label(X,_).
% Each node cannot have multiple labels
:- label(X,A), label(X,B), A != B.
% Labels must match arity of the nodes
arity(X,0) :- node(X), not edge(X,_).
arity(X,2) :- node(X), edge(X,Y), edge(X,Y1), Y < Y1.
arity(X,1) :- node(X), not arity(X,0), not arity(X,2).
symbol(A,0) :- proposition(A).
symbol(next,1). symbol(until,2). symbol(eventually,1). symbol(always,1).
symbol(neg,1). symbol(and,2). symbol(or,2). symbol(implies,2).
:- arity(X,N), label(X,Y), not symbol(Y,N).
% Syntax tree admits a BFS-indexing
id(1,(0,0)).
id(V,(U,V*V+U)) :- edge(U,V).
:- id(I,RI), id(I+1, RJ), RI >= RJ.
```

3.2.4 Encoding semantics

The semantics of $\mathsf{LTL_f}$ formulae are evaluated on traces using syntax trees. As the classic approach to traversing a syntax tree is recursion, the semantics presented by Ielo et al. [2023] are also implemented recursively. We first present the semantics and then discuss the key components. The complete ASP program, Σ_{sem}, implementing these semantics is shown below:

Listing 3.2: The ASP program Σ_{sem} encoding recursive $\mathsf{LTL_f}$ semantics.

```
1  order(X, LHS, RHS) :- edge(X,LHS), edge(X,RHS), LHS < RHS.
2  holds(T, X) :- label(X, A), proposition(A), trace(T, A).
3  holds(T, X) :- label(X, next), edge(X, Y), holds(T+1, Y),
4                 not last(T), trace(T).
5  holds(T, X) :- label(X, until), order(X,LHS,RHS), holds(T, RHS), trace(T).
6  holds(T, X) :- label(X, until), order(X,LHS,RHS), holds(T, LHS),
7                 holds(T+1, X), trace(T).
8  holds(T, X) :- label(X, and), order(X,A,B), holds(T, A),
9                 holds(T, B), trace(T).
10 holds(T, X) :- label(X, or), edge(X, A), holds(T, A), trace(T).
11 holds(T, X) :- label(X, neg), edge(X, Y), not holds(T, Y), trace(T).
12 holds(T, X) :- label(X,implies), order(X,LHS,RHS),
13                holds(T,RHS), holds(T,LHS).
14 holds(T, X) :- label(X,implies), order(X,LHS,RHS),
15                not holds(T,LHS), trace(T).
16 holds(T, X) :- label(X, eventually), edge(X,Y), holds(T,Y).
17 holds(T, X) :- label(X, eventually), holds(T+1, X), trace(T).
18 holds(T, X) :- label(X, always), edge(X, Y), holds(T, Y), last(T).
19 holds(T, X) :- label(X, always), edge(X, Y), holds(T, Y),
20                holds(T+1, X), trace(T).
21 last(T) :- trace(T), not trace(T+1).
22 sat :- holds(0,1).
23 unsat :- not sat.
24 :- sat, unsat.
```

- Line 1: defines the `order/3` predicate is used to distinguish between the left and right side

of labels of a node i and is used for evaluation of non-commutative operators $\rightarrow$ and U (it is also used for evaluation of `and` but solely for convenience reasons).

- Line 21: predicate `last/1` denotes whether a timestep is the end of a trace.
- Lines 2-20: encodes the main semantics of each propositional logic and temporal operator. The predicate `holds/2` is used for showing what subformula of the syntax tree holds at what time. For instance, `holds(4, 2)` would say that the 2nd label of the syntax tree holds at timestep 4. If `label(2, a)`, then, as per the semantics of a proposition in line 2, `a` should appear in the trace at timestep 4.

  Formally, let φ be a formula (syntax tree) and φ_x a subformula rooted in the node x. Let $\lambda \in \mathcal{E}$. The `holds(T, X)` predicate models that $\lambda \models \varphi_x$.
- Line 22-23: denote whether the syntax tree (and thus, the $\mathsf{LTL_f}$ formula) holds. Specifically, whether the formula holds at timestep 0 (start of the trace) as evaluated from the 1st node of the syntax tree.
- Line 24: it cannot be the case that both `sat` and `unsat` are true.

3.3 Reduction from $\text{PL}_{\mathsf{LTL_f}}$ to $\text{ILP}^{context}_{LAS}$

We can now start working on the main result of this chapter which would serve as formal proof that ILASP, a practical algorithm for solving $\text{ILP}^{context}_{LAS}$ is usable for $\text{PL}_{\mathsf{LTL_f}}$ as well.

Let us formally define what it means to reduce one learning framework to another. We follow Law et al. [2018a] in defining a general learning framework: "A learning framework $\mathcal{F}$ defines what a learning task of $\mathcal{F}$ is and what an inductive solution is for a given learning task of $\mathcal{F}$". Furthermore, we use their notion of reduction.

Definition 3.3.1 (Reduction between learning frameworks) Let $ILP_{\mathcal{F}}(T_{\mathcal{F}})$ denote the set of all inductive solutions to the learning task instance $T_{\mathcal{F}}$ for learning framework $\mathcal{F}$. A framework $\mathcal{F}_1$ *reduces* to $\mathcal{F}_2$ if for every $\mathcal{F}_1$ task $T_{\mathcal{F}_1}$ there is an $\mathcal{F}_2$ task $T_{\mathcal{F}_2}$ such that $ILP_{\mathcal{F}_1}(T_{\mathcal{F}_1}) = ILP_{\mathcal{F}_2}(T_{\mathcal{F}_2})$.

Specifically, in our case this would mean that $\text{PL}_{\mathsf{LTL_f}}$ reduces to $\text{ILP}^{context}_{LAS}$ if for every $\text{PL}_{\mathsf{LTL_f}}$ task T_1 there is an $\text{ILP}^{context}_{LAS}$ task T_2 such that $\text{PL}_{\mathsf{LTL_f}}(T_1) = \text{ILP}^{context}_{LAS}(T_2)$.

With the necessary definitions in place, we can proceed to prove that any $\text{PL}_{\mathsf{LTL_f}}$ instance can be framed as an $\text{ILP}^{context}_{LAS}$ instance. Additionally, we will show that all solutions to the original problem are also solutions to the transformed instance, and vice versa. Thus, our starting point is defining how a $\text{PL}_{\mathsf{LTL_f}}$ instance can be framed as an $\text{ILP}^{context}_{LAS}$ instance.

Definition 3.3.2 (Transformation f from $\text{PL}_{\mathsf{LTL_f}}$ to $\text{ILP}^{context}_{LAS}$) We will denote an arbitrary instance of a $\text{PL}_{\mathsf{LTL_f}}$ task as $\mathcal{L} = \langle \mathcal{P}, E^+, E^- \rangle$ and we will denote its corresponding $\text{ILP}^{context}_{LAS}$ instance as $f(\mathcal{L}) = \langle B_{LAS}, \mathcal{H}_{LAS}, \langle E^+_{LAS}, E^-_{LAS} \rangle \rangle$ where the individual components are defined as follows.

- $B_{LAS} = \Sigma^{\mathcal{P}}_{constr} \cup \Sigma_{sem}$. The respective programs are presented in full in Listing 3.1 and Listing 3.2.
- Let $\mathcal{O} = \{\texttt{neg}, \texttt{and}, \texttt{or}, \texttt{implies}, \texttt{next}, \texttt{eventually}, \texttt{always}, \texttt{until}\}$. Then:

$$\begin{aligned} H_{LAS} = &\{\texttt{label(}n\texttt{,}o\texttt{).} \mid n \in \mathbb{N}^+, o \in \mathcal{O}\} \\ &\cup \{\texttt{label(}n\texttt{,}p\texttt{).} \mid n \in \mathbb{N}^+, p \in \mathcal{P}\} \\ &\cup \{\texttt{edge(}n_1\texttt{,}n_2\texttt{).} \mid n_1, n_2 \in \mathbb{N}^+\} \end{aligned}$$

As this section focuses on demonstrating the correspondence between the solution spaces of $\mathcal{L}$ and $f(\mathcal{L})$, an infinite hypothesis space poses no issue. In practice, however, we impose a finite bound to limit the maximum size of the syntax tree.

- Let $\lambda \in E^+$ be an arbitrary positive linear trace. Then, we can represent the trace as a CDPI $e^{\lambda}_{cdpi} = \left\langle \left\langle e^{inc}_{pi}, e^{exc}_{pi} \right\rangle, e_{ctx} \right\rangle$ as follows:
  - $e^{inc}_{pi} = \{\texttt{sat}\}$, $e^{exc}_{pi} = \{\}$
  - $e_{ctx} = \{\texttt{trace(}i\texttt{).} \mid i \in time(\lambda)\} \cup \{\texttt{trace(}i\texttt{,}p\texttt{).} \mid i \in time(\lambda), p \in \lambda[i]\}$

  Then, $E^+_{LAS} = \{e^{\lambda}_{cdpi} \mid \lambda \in E^+\}$.
- E^-_{LAS} is defined the same as E^+_{LAS} for each negative linear trace $\lambda \in E^-$.

Now, we demonstrate that if a solution to $\mathcal{L}$ exists, then its transformed ASP syntax tree is a solution to $f(\mathcal{L})$.

Lemma 3.3.3 If $\varphi \in \text{PL}_{\text{LTL}_\text{f}}(\mathcal{L})$, then $g_1(\varphi) \in \text{ILP}^{context}_{LAS}(f(\mathcal{L}))$.

Proof. Assume that $\varphi \in \text{PL}_{\text{LTL}_\text{f}}(\mathcal{L})$. As per Definition 2.3.3, $g_1(\varphi) \in \text{ILP}^{context}_{LAS}(f(\mathcal{L}))$, if and only if:

1. $\forall \langle e, C\rangle \in E^+_{LAS}, \exists A \in AS(B_{LAS} \cup C \cup g_1(\varphi))$ s.t. A extends e, and
2. $\forall \langle e, C\rangle \in E^-_{LAS}, \nexists A \in AS(B_{LAS} \cup C \cup g_1(\varphi))$ s.t. A extends e.

We can reformulate the above to make it clearer. For brevity and future reference, set $\Sigma = B_{LAS} \cup C \cup g_1(\varphi)$. We may omit the e^{exc} requirements in the definition of "A extends e" because the transformation f sets the exclusion set to an empty set for each CDPI. So, $g_1(\varphi) \in \text{ILP}^{context}_{LAS}(f(\mathcal{L}))$ if and only if:

Z1. $\forall \left\langle \left\langle e^{inc}, e^{exc} \right\rangle, e_{ctx} \right\rangle \in E^+_{LAS}, \exists A \in AS(\Sigma)$ s.t. $\texttt{sat.} \in A$, and

Z2. $\forall \left\langle \left\langle e^{inc}, e^{exc} \right\rangle, e_{ctx} \right\rangle \in E^-_{LAS}, \forall A \in AS(\Sigma)$ s.t. $\texttt{sat.} \notin A$.

Firstly, we prove **Z1**. Assume an arbitrary $\left\langle \left\langle e^{inc}, e^{exc} \right\rangle, e_{ctx} \right\rangle \in E^+_{LAS}$. We need to show that there exists an answer set of Σ s.t. $\texttt{sat} \in A$. To prove the existence of an answer set with a specific property, one can simply construct a Herbrand interpretation, demonstrate that it satisfies the criteria of being an answer set, and verify that it possesses the desired property. In Definition 3.3.4 we have constructed an interpretation I_φ by examining Σ and adding all the basic facts presented there, as well as all the facts which are derivable (Definition 2.2.9).

Now, we must prove that I_φ in Definition 3.3.4 is indeed a minimal model of the reduct Σ^{I_φ}, i.e., an answer set for Σ. We first have to compute the reduct Σ^{I_φ} which is done in Observation 3.3.5. In Lemma 3.3.7 we show that I_φ is a minimal model for Σ^{I_φ}.

So, the interpretation I_φ is an answer set for Σ and, due to its construction, one that has $\texttt{sat}$. Since the positive trace was chosen arbitrarily, we have proven that Z1 holds.

□

Now, let us prove **Z2**. Assume an arbitrary negative CDPI $e^{\lambda}_{cdpi} = \left\langle \left\langle e^{inc}, e^{exc} \right\rangle, e_{ctx} \right\rangle \in E^-_{LAS}$ and let A be an arbitrary answer set of Σ. Suppose, for the sake of contradiction, that $\texttt{sat} \in A$. The transformation f directly encodes all time steps and their corresponding truth assignments from the negative trace λ into the negative CDPI e^{λ}_{cdpi}. Recall that $g_1(\varphi) \subseteq \Sigma$ where φ is a solution to $\text{PL}_{\text{LTL}_\text{f}}(\mathcal{L})$, so $\lambda, 0 \not\models \varphi$. If A is an answer set, it will look almost exactly like the interpretation in Definition 3.3.4 except for the CDPI and the derived facts due to the CDPI. Nevertheless, if an answer set to such a program has $\texttt{sat}$ in it, then, due to the rule of

`sat`, it must have `holds(0,1)` in it, otherwise, A is not minimal and thus not an answer set. Observe that, due to Definition 3.3.4 and Lemma 3.3.6, `holds(0,1)` $\in A$ if and only if $\lambda, 0 \models \varphi$. However, we have assumed that λ is a negative example, and λ cannot satisfy φ. Contradiction. This means that for any negative CDPI and for any answer set A of Σ, `sat` $\notin A$. □

■

Definition 3.3.4 (Candidate answer set I_φ for Σ) Let $arity : subf(\varphi) \to \{0, 1, 2\}$ be a function that maps subformulae ψ of φ to the arity of its main connective, note that then $arity(\psi) = 0$ if ψ is an atom. Let the following program all of the grounded facts in Σ:

$$\begin{aligned}\text{Facts}(\Sigma) =& \{\texttt{proposition}(p) \mid p \in \mathcal{P}\} \cup \{\texttt{reach(1)}\} \cup \{\texttt{id(1,(0,0))}, \texttt{ last}(last(\lambda))\} \\ &\cup \{\texttt{symbol(next,1)}, \texttt{symbol(until,2)}, \texttt{symbol(eventually,1)}, \texttt{symbol(or,2)} \\ &\quad \texttt{symbol(always,1)}, \texttt{symbol(neg,1)}, \texttt{symbol(and,2)}, \texttt{symbol(implies,2)}\} \\ &\cup \{\texttt{symbol}(p,0) \mid p \in \mathcal{P}\} \cup g_1(\varphi) \cup e_{ctx} \cup e^{inc}\end{aligned}$$

In the following, any n_ψ refers the node index n which appears as the first term in the `label` predicate of $g_n(\psi)$. Consider the following interpretation $I_\varphi \subseteq HB(\Sigma)$:

$$\begin{aligned}I_\varphi = \text{Facts}(\Sigma) &\cup \{\texttt{node}(n_\psi) \mid \psi \text{ is a subformula of } \varphi\} \\ &\cup \{\texttt{reach}(n_\psi) \mid \psi \text{ is a subformula of } \varphi\} \\ &\cup \{\texttt{arity}(n_\psi, arity(\psi)) \mid \psi \text{ is a subformula of } \varphi\} \\ &\cup \{\texttt{id}(m_\alpha, (n_\psi, m_\alpha \cdot m_\alpha + n_\psi)) \mid \psi \text{ is a subf. of } \varphi, \text{ and } \alpha \text{ is the immediate subf. of } \psi\} \\ &\cup \{\texttt{order}(n_\psi, m_\alpha, m_\beta) \mid \psi \text{ is a subf. of } \varphi, \text{ and } \alpha, \beta \text{ are immediate subf. of } \psi\} \\ &\cup \{\texttt{holds}(i, n_\psi) \mid \lambda, i \models \psi, \text{ where } \psi \text{ is a subformula of } \varphi\}\end{aligned}$$

Observation 3.3.5 (The reduct Σ^{I_φ}) The reduct is computed by taking out rules with negative atoms in I_φ and removing negative literals from remaining rules. Note that most of Σ has positive rules, which means that little has to change. The parts worth discussing are the `arity` predicates, the hard constraints, and the `holds` predicates since they are the only ones with `not` in them. In the following, we argue how the reduct will look like and why I_φ would satisfy its rules.

Recall the arity rules from $\Sigma^{\mathcal{P}}_{constr}$:

```
1 arity(X,0) :- node(X), not edge(X,_).
2 arity(X,2) :- node(X), edge(X,Y), edge(X,Y1), Y < Y1.
3 arity(X,1) :- node(X), not arity(X,0), not arity(X,2).
```

Consider a subformula ψ of φ. We argue that the `arity` predicates in Σ correspond exactly to the arity of ψ:

- If ψ is an atom, it has no edges in $g_n(\psi)$. Consequently, we can derive `arity(`n_ψ`, 0)` from its ASP rule which exactly corresponds to $arity(\psi) = 0$. Then, the rule for `arity(`n_ψ`,1)` will be taken out of the reduct, and the body of the rule for `arity(`n_ψ`,2)` will not be satisfied.
- If the most complex operator in ψ is one of $\{\wedge, \vee, \rightarrow, \mathsf{U}\}$, then $arity(\psi) = 2$. By $g_n(\psi)$, there will be two edges from the starting node n_ψ to nodes m_1 and m_2, where $m_1 < m_2$. Which is exactly the body of the `arity(`n_ψ`, 2)` rule, allowing us to derive it and add it to the interpretation. Then, the rules for `arity(`n_ψ`,0)` and `arity(`n_ψ`,2)` will be taken out of the reduct.

- If the most complex operator in ψ is one of $\{\neg, \mathsf{X}, \mathsf{F}, \mathsf{G}\}$, then $arity(\psi) = 1$. Note that if this is the case, ψ will not have satisfied the conditions for deriving `arity(`n_ψ`, 0)` or `arity(`n_ψ`, 2)`, and so we can derive `arity(`n_ψ`, 1)` as per its ASP rule. Then, the rule for `arity(`n_ψ`,0)` will be taken out of the reduct, and the body of the rule for `arity(`n_ψ`,2)` will not be satisfied.

Importantly, the formula transformation ensures that no extra arity predicates are derived in any of these cases. The above is both an argument for how `arity` will appear in the reduct and for I_φ satisfying the `arity` rules of the reduct Σ^{I_φ}.

Next, we demonstrate that the hard constraints are either removed in the reduct or their bodies are not satisfied. For reference, we write out all of the hard constraints from $\Sigma^{\mathcal{P}}_{constr}$ and Σ_{sem}.

Listing 3.3: Hard constraints from $\Sigma^{\mathcal{P}}_{constr}$ and Σ_{sem}.

```
1  :- node(X), not reach(X).
2  :- node(X), not edge(_, X), X > 1.
3  :- node(X), 3 #count {Z:edge(X,Z)}.
4  :- node(X), not label(X, _).
5  :- label(X,A), label(X,B), A < B.
6  :- arity(X,N), label(X,Y), not symbol(Y,N).
7  :- id(I,RI), id(I+1,RJ), RI >= RJ.
8  :- sat, unsat.
```

Lines 1, 2, 4, and 6 will not be present in the reduct Σ^{I_φ} because all of the atoms which are negated in these rules will be in I_φ. In the case of line 2, the constraint will be remain for $node(1)$ only, but will not be satisfied because $1 \not> 1$. The bodies of line 3 and 5 will not be satisfied because g_n never adds to the resulting set more than two edges starting from the same node and exactly one `label` predicate is added per subformula. The body of line 8 cannot be satisfied because `unsat :- not sat.` and $e^{inc} =$ `sat` $\in I_\varphi$. Line 7 ensures that the tree admits a breadth-first-search traversal indexing [Ielo et al., 2023, Furelos-Blanco et al., 2021] which prevents cycles. W.l.o.g. let `id(`m_1`, (`n`, `$m_1 \cdot m_1$` + `n`))`, `id(`m_2`, (`n`, `$m_2 \cdot m_2$` + `n`))` $\in I_\varphi$ where $n, m_1, m_2 \in \mathbb{N}$ and $m_2 = m_1 + 1$. In order to satisfy the body of line 7, the first tuple inside the `id` predicate would have to be lexicographically greater or equal to the tuple in the second `id` predicate. This cannot be the case because the g_n function adds `id` predicates with unique node identifiers for each subformula and $m_1 < m_2$.

Lastly, by construction of I_φ, the grounded `holds` atoms will be in the interpretation only if $\lambda, i \models \psi$, meaning that the rest of the `holds` facts for the same n_ψ will either have been removed in the reduct (because an atom $\in I_\varphi$ will be in their body) or they will not be satisfied due to the fact that $\lambda, i \models \psi$ iff `holds(`i`, `n_ψ`)`.

We have now detailed how Σ^{I_φ} would look like.

In the following lemma we demonstrate that the predicates `holds` are correctly derived in the interpretation and thus correctly encode $\mathsf{LTL_f}$ semantics in ASP.

Lemma 3.3.6 Let ψ be any subformula of φ, and c the main connective of ψ. Let r be the ASP rule in Σ_{sem} for which $head(r) =$ {`holds(`i`, `n_ψ`)`} and {`label(`n_ψ`, `c`)`} $\subseteq body(r)$. For all $i \in time(\lambda)$,

$$\text{if } body(r)^+ \subseteq I_\varphi \text{ and } body(r)^- \cap I_\varphi = \emptyset \text{ then } \texttt{holds(}i\texttt{, } n_\psi\texttt{)} \in I_\varphi.$$

Proof. We prove this by case distinction. For all of the cases recall that $g_n(\psi) \subseteq g_1(\varphi) \subseteq I_\varphi$ for some node index n recursively generated in $g_1(\varphi)$. Assume an arbitrary $i \in time(\lambda)$.

Case $\psi = p$ for $p \in \mathcal{P}$. Consider the following relevant grounded rule r from Σ_{sem} for

atomic propositions:

$$\texttt{holds(}i\texttt{,}n_\psi\texttt{) :- label(}n_\psi\texttt{,}p\texttt{), proposition(}p\texttt{), trace(}i\texttt{,}p\texttt{).}$$

Assume $body(r)^+ \subseteq I_\varphi$ and $body(r)^- \cap I_\varphi = \emptyset$. Since $\texttt{trace(}i\texttt{, }p\texttt{)} \in I_\varphi$, it must be that $p \in \lambda[i]$ due to f, in other words, $\lambda, i \models p$. By construction of I_φ, then, $\texttt{holds(}i\texttt{, }n_p\texttt{)} \in I_\varphi$, i.e. $\texttt{holds(}i\texttt{, }n_\psi\texttt{)} \in I_\varphi$. □

Case $\psi = \neg\alpha$. Consider the $\mathsf{LTL_f}$ semantics rule r for negation:

$$\texttt{holds(}i\texttt{,}n_\psi\texttt{) :- label(}n_\psi\texttt{,neg), edge(}n_\psi\texttt{,}m_\alpha\texttt{), not holds(}i\texttt{,}m_\alpha\texttt{), trace(}i\texttt{).}$$

Assume, $body(r)^+ \subseteq I_\varphi$ and $body(r)^- \cap I_\varphi = \emptyset$. Due to the latter, $\texttt{holds(}i\texttt{, }m_\alpha\texttt{)} \notin I_\varphi$ which, by the construction of I_φ, means that $\lambda, i \not\models \alpha$. By the semantics of negation, $\lambda, i \models \neg\alpha$, and by the construction of I_φ, $\texttt{holds(}i\texttt{, }n_{\neg\alpha}\texttt{)} \in I_\varphi$, and thus $\texttt{holds(}i\texttt{, }n_\psi\texttt{)} \in I_\varphi$ as required. □

Case $\varphi = \alpha \mathsf{U} \beta$. Consider the $\mathsf{LTL_f}$ semantics rule r_1 and r_2 for until:

$$\begin{aligned} r_1 = \texttt{holds(}i\texttt{, }n_\psi\texttt{) :- label(}n_\psi\texttt{, until), order(}n_\psi\texttt{,}m_\alpha\texttt{,}m_\beta\texttt{),}\\ \texttt{trace(}i\texttt{), not holds(}i\texttt{, }m_\alpha\texttt{), holds(}i\texttt{, }m_\beta\texttt{).}\\ r_2 = \texttt{holds(}i\texttt{, }n_\psi\texttt{) :- label(}n_\psi\texttt{, until), order(}n_\psi\texttt{,}m_\alpha\texttt{,}m_\beta\texttt{),}\\ \texttt{trace(}i\texttt{), holds(}i\texttt{, }m_\alpha\texttt{), holds(}i+1\texttt{, }n_\psi\texttt{).} \end{aligned}$$

Also, recall that

$$\begin{aligned} \lambda, i \models (\alpha \mathsf{U} \beta) \text{ iff } \exists j \text{ with } i \leqslant j \leqslant last(\lambda) \text{ s.t. } \lambda, j \models \beta \text{ and}\\ \forall k \text{ with } i \leqslant k < j \text{ s.t. } \lambda, k \models \alpha. \end{aligned}$$

Note that if $i = j$, $\lambda, i \models (\alpha \mathsf{U} \beta)$ for one time instance only and so we have to show $\texttt{holds(}i\texttt{, }n_\psi\texttt{)} \in I_\varphi$ only. However, if $j > i$, then $\lambda, j \models (\alpha \mathsf{U} \beta)$ for all time instances $i \leqslant j$ and so we have to show $\texttt{holds(}j\texttt{, }n_\psi\texttt{)} \in I_\varphi$ for all such j.

First, let's prove that the statement holds for r_1. Assume, $body(r_1)^+ \subseteq I_\varphi$ and $body(r_1)^- \cap I_\varphi = \emptyset$. Due to construction of I_φ, we have $\lambda, i \not\models \alpha$ and $\lambda, i \models \beta$. This is exactly the $i = j$ case, which means that $\lambda, i \models \alpha \mathsf{U} \beta$ and then indeed $\texttt{holds(}i\texttt{, }n_\psi\texttt{)}$ by construction of I_φ.

Now, let's prove the statement for r_2. Assume, $body(r_2)^+ \subseteq I_\varphi$ and $body(r_2)^- \cap I_\varphi = \emptyset$. By the construction of I_φ, we have $\lambda, i \models \alpha$ and $\lambda, i+1 \models \psi$. Due to this, and that $i < i+1$ which is necessary in the second conjunct of the U semantics, we can conclude that $\lambda, i \models \alpha \mathsf{U} \beta$. And, by the construction of I_φ, we have $\texttt{holds(}i\texttt{, }n_\psi\texttt{)} \in I_\varphi$.

□

Cases $\alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \rightarrow \beta$, $\mathsf{X}\alpha$, $\mathsf{F}\alpha$, $\mathsf{G}\alpha$. The rest of the proofs follow an anologous pattern and we omit them. □

■

In the following lemma we demonstrate that the candidate answer set I_φ is indeed an answer set for Σ.

Lemma 3.3.7 The Herbrand interpretation I_φ is a minimal model of the reduct Σ^{I_φ}.

Proof. We have to demonstrate the following:

1. I_φ is a model for Σ^{I_φ}.
2. I_φ is a minimal model for Σ^{I_φ}. That is, no strict subset of I_φ is a model for Σ^{I_φ}.

Item 1. To show that a Herbrand interpretation is a model for Σ^{I_φ}, we have to show that for all rules $r \in \Sigma^{I_\varphi}$, if $body^+(r) \subseteq I_\varphi$ and $body^-(r) \cap I_\varphi = \emptyset$, then $head(r) \in I_\varphi$. Previously,

we have already argued why heads of the hard constraints in Σ^{I_φ} will not be derived from I_φ. Since the bodies of all facts are empty, all of the heads are trivially derivable and must be in I_φ. Observe that indeed it is the case for `reach(1)`, `id(1, (0,0))` and all `proposition`, `symbol` facts. The encoded formula $g_1(\varphi) \in I_\varphi$ is comprised of a set of `label`, `edges`, and `id` literals. By adding these literals into I_φ, the ASP rules allow to derive further `node`, `reach`, `arity`, `id`, and `order` literals. Note that their heads will be already added in I_φ in a way that is tightly coupled with $g_1(\varphi)$, i.e. each n_ψ corresponds to the node from the formula transformation which is exactly how the ASP rules would allow us to derive them. This is partly due to the fact that all node indices in g_1 follow the rules in the bodies of the ASP rules.

The interpretation I_φ also has e_{ctx} which does not contradict any other rule, but enables to derive `last` and `holds` literals which are already in I_φ. `last`$(last(\lambda))$ indeed has the last index of the trace λ as its term, and this corresponds to the body of its ASP rules.

Of the utmost importance are the `holds` literals, which encode the LTL_f semantics in ASP. This is, however, not straightforward to see, due to the recursive nature of the rules. In Lemma 3.3.6 we formally demonstrate that the `holds` literals are in I_φ justifiably so, i.e. the bodies of the `holds` rules are satisfied. This would mean that the presence of bodies of the `holds` rules in Σ_{sem} correspond exactly to the satisfiability of a formula at a time instant on a trace. Since every subformula ψ of φ which satisfies the trace at a specific time instance i leads to `holds(`i`,` n_ψ`)` being in the interpretation, we can further conclude that since $g_1(\varphi) \in I_\varphi$, `holds(0, 1)` $\in I_\varphi$. The rule `sat :- holds(0,1).` would thus allow us to derive `sat` which is already in I_φ by construction.

We have shown that the heads of all rules in Σ^{I_φ} which are derived by I_φ are indeed in I_φ, thus, we can conclude that I_φ is a model for Σ^{I_φ}. □

Item 2. Suppose, towards contradiction, that $J_\varphi \subset I_\varphi$ and J_φ is a model for Σ^{I_φ}. If J_φ is a model for Σ^{I_φ}, clearly, all of the facts Facts(Σ) present in Σ^{I_φ} should be in J_φ. Since $g_1(\varphi) \subseteq J_\varphi$, then all of the grounded predicates `node`, `reach`, `arity`, `order` will be in J_φ as well because the bodies of their rules involve elements of $g_1(\varphi)$ or what can be derived from it. The only non-obvious case is the `holds` predicate due to its recursive nature. Nevertheless, in Lemma 3.3.8 we have shown that the `holds` predicates in J_φ and I_φ correspond.

We have shown that if J_φ is a model for Σ^{I_φ}, then it comprises of all the facts already in I_φ, i.e. $J_\varphi = I_\varphi$. Since we have initially assumed that J_φ is a strict subset of I_φ, we have arrived at a contradiction. Then, I_φ is a minimal model for Σ^{I_φ}. □ ■

Lemma 3.3.8 Let $J_\varphi \subset I_\varphi$ be the Herbrand interpretation defined in Part 2 of Lemma 3.3.7. For all $i \in time(\lambda)$,

$$\texttt{holds(}i\texttt{,}\ n_\psi\texttt{)} \in J_\varphi \iff \texttt{holds(}i\texttt{,}\ n_\psi\texttt{)} \in I_\varphi$$

Proof. We immediately get the ($\Rightarrow$) direction simply because $J_\varphi \subset I_\varphi$. Now, we prove the ($\Leftarrow$) direction by induction on the complexity of ψ. The general strategy here will be to show that the `holds` rules for each formula are derived by J_φ, thus the corresponding `holds` must be in J_φ since we assumed it is a model of Σ^{I_φ}. Assume `holds(`i`,` n_ψ`)` $\in I_\varphi$.

Base case. Let $\psi = p$ where $p \in \mathcal{P}$. Consider the following relevant grounded rule r from Σ_{sem} for atomic propositions:

$$\texttt{holds(}i\texttt{,}n_\psi\texttt{) :- label(}n_\psi\texttt{,}p\texttt{), proposition(}p\texttt{), trace(}i\texttt{,}p\texttt{).}$$

We have to show that $body(r)^+ \subseteq J_\varphi$. Since `holds(`$i$`,` n_ψ`)` $\in I_\varphi$, $body(r)^+ \subseteq I_\varphi$. Observe that $body(r)^+ \subseteq$ Facts(Σ) and that Facts$(\Sigma) \subseteq J_\varphi$ because J_φ is a model for Σ^{I_φ} as detailed in Item 2 of Lemma 3.3.7. □

Inductive step. Assume the following inductive hypothesis.

- Let α be an $\mathsf{LTL_f}$ formula written in $\mathcal{P}$ and $\texttt{holds}(i,m_\alpha) \in J_\varphi \Leftrightarrow \texttt{holds}(i,m_\alpha) \in I_\varphi$.
- Let β be an $\mathsf{LTL_f}$ formula written in $\mathcal{P}$ and $\texttt{holds}(i,m_\beta) \in J_\varphi \Leftrightarrow \texttt{holds}(i,m_\beta) \in I_\varphi$.

We prove that the main statement holds for complex $\mathsf{LTL_f}$ formulae.

- Case $\psi = \neg\alpha$. Consider the $\mathsf{LTL_f}$ semantics rule r for negation:

  $$\texttt{holds}(i,n_\psi) \texttt{ :- } \texttt{label}(n_\psi,\texttt{neg}), \texttt{edge}(n_\psi,m_\alpha), \texttt{not holds}(i,m_\alpha), \texttt{trace}(i).$$

  We have to show $body(r)^+ \subseteq J_\varphi$ and $body(r)^- \cap J_\varphi = \emptyset$. In particular, $body(r)^+ \subseteq \text{Facts}(\Sigma) \subseteq J_\varphi$. Suffices to show $\texttt{holds}(i,\ m_\alpha) \notin J_\varphi$. Observe:

  $$\begin{aligned} \texttt{holds}(i,\ n_\psi) \in I_\varphi &\iff \lambda, i \models \neg\alpha \\ &\iff \lambda, i \not\models \alpha \\ &\iff \texttt{holds}(i,\ m_\alpha) \notin I_\varphi \\ &\overset{IH}{\iff} \texttt{holds}(i,\ m_\alpha) \notin J_\varphi \end{aligned}$$

  Since $head(r)$ is derivable by J_φ, it must be in J_φ, since we have assumed that J_φ is a model of Σ^{I_φ}. □

- Case $\psi = \alpha \mathsf{U} \beta$. Consider the $\mathsf{LTL_f}$ semantics rule r_1 and r_2 for until:

  $$\begin{aligned} r_1 = \texttt{holds}(i,\ n_\psi) \texttt{ :- } &\texttt{label}(n_\psi,\ \texttt{until}), \texttt{order}(n_\psi,m_\alpha,m_\beta), \\ &\texttt{trace}(i), \texttt{not holds}(i,\ m_\alpha), \texttt{holds}(i,\ m_\beta). \\ r_2 = \texttt{holds}(i,\ n_\psi) \texttt{ :- } &\texttt{label}(n_\psi,\ \texttt{until}), \texttt{order}(n_\psi,m_\alpha,m_\beta), \\ &\texttt{trace}(i), \texttt{holds}(i,\ m_\alpha), \texttt{holds}(i+1,\ n_\psi). \end{aligned}$$

  Also, recall that

  $$\begin{aligned} \lambda, i \models (\alpha \mathsf{U} \beta) \text{ iff } &\exists j \text{ with } i \leqslant j \leqslant last(\lambda) \text{ s.t. } \lambda, j \models \beta \text{ and} \\ &\forall k \text{ with } i \leqslant k < j \text{ s.t. } \lambda, k \models \alpha. \end{aligned}$$

  First, assume $\texttt{holds}(i,\ n_\psi) \in I_\varphi$ due to r_1. The `label`, `order`, and `trace` grounded literals are in J_φ because they are in $\text{Facts}(\Sigma)$. Since $\texttt{holds}(i,\ m_\beta) \notin I_\varphi$, by the inductive hypothesis $\texttt{holds}(i,\ m_\alpha) \notin J_\varphi$. Similarly, since $\texttt{holds}(i,\ m_\alpha) \in I_\varphi$, by the inductive hypothesis $\texttt{holds}(i,\ m_\alpha) \in J_\varphi$. So, $head(r_1)$ is derivable by J_φ, and since it is a model of Σ^{I_φ}, $\texttt{holds}(i,\ n_\psi) \in J_\varphi$.

  Now, assume $\texttt{holds}(i,\ n_\psi) \in I_\varphi$ due to r_2. Again, the `label`, `order`, and `trace` grounded literals are in J_φ because they are in $\text{Facts}(\Sigma)$. Since $\texttt{holds}(i,\ m_\alpha) \in I_\varphi$, by the inductive hypothesis $\texttt{holds}(i,\ m_\alpha) \in J_\varphi$. Observe:

  $$\begin{aligned} \texttt{holds}(i,\ n_\psi) \in I_\varphi &\iff \lambda, i \models \alpha \mathsf{U} \beta \\ &\iff \lambda, k \models \alpha \text{ and } \lambda, j \models \alpha \mathsf{U} \beta \\ &\iff \texttt{holds}(k,\ m_\alpha), \texttt{holds}(j,\ n_\psi) \in I_\varphi \end{aligned}$$

  In particular, fix $j = last(\lambda)$, then $\texttt{holds}(last(\lambda),\ m_\beta) \in I_\varphi \overset{IH}{\Leftrightarrow} \texttt{holds}(last(\lambda),\ m_\beta) \in J_\varphi$. And we also know that $\texttt{holds}(k,\ m_\alpha) \in I_\varphi \overset{IH}{\Leftrightarrow} \texttt{holds}(k,\ m_\alpha) \in J_\varphi$ for all $k < j$.

And so, $\texttt{holds(}i\texttt{, }m_\alpha\texttt{)} \in J_\varphi$ but we also have $\texttt{holds(}i+1\texttt{, }n_\psi\texttt{)} \in J$ by trivial reverse induction on $k = last(\lambda)$. So, $head(r_2)$ is derivable by J, and since it is a model of Σ^{I_φ}, $\texttt{holds(}i\texttt{, }n_\psi\texttt{)} \in J_\varphi$.

□

- The rest of the cases are similar. □

■

The final step is to demonstrate that if we have a solution to $f(\mathcal{L})$ (for example, using the ILASP algorithm), we can transform it back into a well-formed $\mathsf{LTL_f}$ formula that is a solution to $\mathcal{L}$. This would confirm that the solutions found by algorithms like ILASP actually address the underlying theoretical problem we are concerned with.

Lemma 3.3.9 If $H \in \text{ILP}^{context}_{LAS}(f(\mathcal{L}))$, then $H^\varphi \in \text{PL}_{\mathsf{LTL_f}}(\mathcal{L})$.

Proof. Assume $H \in \text{ILP}^{context}_{LAS}(f(\mathcal{L}))$. Equivalently, we must show that from H, we can construct a well-formed $\mathsf{LTL_f}$ formula H^φ satisfying

1. For all $\lambda \in E^+$, $\lambda \models H^\varphi$, and
2. For all $\lambda \in E^-$, $\lambda \models H^\varphi$

Recall that $H \subseteq \mathcal{H}_{LAS}$. This is a set of ASP facts representing a syntax tree, which is essentially a labeled binary tree, as the maximum arity of operators in $\mathsf{LTL_f}$ is 2. If that is indeed the case, we can use some classic in-order graph traversal algorithm to convert this tree to an $\mathsf{LTL_f}$ formula. We would treat this algorithm as the function g_1^{-1}. Let $T = (V, E)$ where $V = \{n \mid \texttt{label(}n\texttt{, \_)} \in H\}$ and $E = \{(e_1, e_2) \mid \texttt{edge(}e_1\texttt{, }e_2\texttt{)} \in H\}$. Due to the constraints in $\Sigma^{\mathcal{P}}_{constr}$, we assume that the root of the tree is $n = 1$, and observe that the edges are directed, thus making this (potential) tree, directed and rooted. Recall that H is a solution to $\text{ILP}^{context}_{LAS}(f(\mathcal{L}))$ if and only if:

1. $\forall \langle e, C\rangle \in E^+_{LAS}, \exists A \in AS(B_{LAS} \cup C \cup H)$ s.t. A extends e, and
2. $\forall \langle e, C\rangle \in E^-_{LAS}, \nexists A \in AS(B_{LAS} \cup C \cup H)$ s.t. A extends e.

Because the answer sets are generated from the combined program of B_{LAS} and H, any solution H must satisfy the constraints within the background knowledge B_{LAS}. In our subsequent proofs, we will make use of these constraints. To employ the inorder tree traversal algorithms, we have to demonstrate that T is a binary tree. Specifically, we have to show the following:

1. **Each node is labeled by exactly one label.** If this were not the case, it would be ambiguous how to read the tree and convert it to an $\mathsf{LTL_f}$ formula. Since elements of H only involve labels already, we don't have to address the case when a node is unlabeled, and suffices to show that there cannot be more than 1 label for a node. Suppose, towards contradiction, that some node n is labeled by more than 1 label. Take two such literals $\texttt{label(}n\texttt{, }l_1\texttt{)}, \texttt{label(}n\texttt{, }l_2\texttt{)} \in H$ where $l_1 \neq l_2$. Recall the rule in B_{LAS}:

   ```
   :- label(X,A), label(X,B), A < B.
   ```

   The body of this hard constraint would then be satisfied since $l_1 \neq l_2$ and so either l_1 is lexicographically less than l_2 or the other way around. This would contradict the fact that A is an answer set for the program with both B_{LAS} and H in it, and H would thus not be a solution to $\text{ILP}^{context}_{LAS}(f(\mathcal{L}))$. So, the first term for every `label` literal in H must be unique. □

2. T **is connected.** Suppose, towards contradiction, that T is disconnected. If that is the case, it is a standard result in graph theory, that if $|V| = n$, then for a disconnected graph $|E|$ is less than $n-1$. Without loss of generality, fix $|E| = n-2$. We shall now show that if this is the case, the answer set A would violate one of the hard constraints, contradicting that it's a model.

   The `reach` literals encode connectedness in the ASP program. Note that, `reach(1)` $\in A$ due to B_{LAS}. Let $N = \{n \mid \texttt{node(n)} \in A\}$ and $R = \{\texttt{reach(1)}\} \cup \{\texttt{reach}(v) \mid \texttt{reach}(v) \in A\}$. Since there are $n-2$ edges, $|R| \leqslant 1 + n - 2$ (there can be less if the `reach` condition is not satisfied). Recall the following hard constraint in B_{LAS}:

   ```
   :- node(X), not reach(X).
   ```

   Now, $|N| = n$ and $|R| \leqslant n-1$, so there will be at least one node which will not be reachable and thus the body of this hard constraint would be satisfied. This would mean that A is not a model of a program with B_{LAS} and H together, so H is not a solution to $\mathrm{ILP}^{context}_{LAS}(f(\mathcal{L}))$. Contradiction. It must be the case that T is connected.

   Observe also that due to the rule `reach(T) :- edge(R,T), reach(R).`, the tree must be constructed in ascending order, and not descending, otherwise no other `reach` literals will be derived and it will contradict the afore-mentioned hard constraint. □

3. T **is acyclic.** Suppose, towards contradiction, that T is cyclic, i.e. there must be at least one cycle in T. If T is cyclic, it is a standard result in graph theory, that if $|V| = n$, for a cyclic graph, $|E| > n-1$. Without loss of generality, fix $|E| = n$.

   Note that due to the rule `node(X) :- node(X+1), X >= 1`, the nodes used to describe the tree start from 1 and are sequential. Due to this and the fact that the graph is cyclic, there must exist an edge $\langle e_1, e_2\rangle \in E$ s.t. $e_1 > e_2$. Central to this proof will is the rule `id(V,(U,V*V+U)) :- edge(U,V)`, and so for the mentioned edge, we derive `id(`e_2`,(`$e_1, e_2 \cdot e_2 + e_1$`))` $\in A$. Given that edge $\langle e_1, e_2\rangle$ is one which creates a cycle, and considering the T's connectivity and the sequential ordering of edges shown in the previous point, there must be some edge $\langle e_3, e_2 + 1\rangle \in E$ s.t. $e_3 < e_2 + 1$ and then `id(`$e_2 + 1, (e_3, (e_2+1)^2 + e_3)$`)` $\in A$.

   Note that the body of the hard constraint `:- id(I,RI), id(I+1, RJ), RI >= RJ` would be satisfied. Both `id(`e_2`,(`$e_1, e_2 \cdot e_2 + e_1$`))`, `id(`$e_2 + 1, (e_3, (e_2+1)^2 + e_3)$`)` $\in A$. According to the hard constraint, $(e_1, e_2 \cdot e_2 + e_1)$ cannot be lexicographically greater than or equal to $(e_3, (e_2+1)^2 + e_3)$. However, it is, because $e_1 > e_2$ and $e_3 < e_2 + 1$. So, the body of this hard constraint is satisfied, and it contradicts that A is a model of the program B_{LAS} with H, which contradicts the fact that H is a solution to $\mathrm{ILP}^{context}_{LAS}(f(\mathcal{L}))$. Thus, T must be acyclic. □

4. T **is a binary tree.** We have concluded that due to the constraints in $\Sigma^{\mathcal{P}}_{constr}$, T is directed and rooted. In the previous points we have shown that T is connected and acyclic, thus making T a tree by definition. Lastly, we must show that it is binary. The sufficient rule to demonstrate this fact is `:- node(X), 3 <= # count { Z: edge(X, Z) }`. If there are 3 or more edges for the same node, the body of the hard constraint will be satisfied, and there will be no answer set, which means that there will be no solution to $\mathrm{ILP}^{context}_{LAS}(f(\mathcal{L}))$ which would contradict our initial assumption. So, every node has at most 2 edges from it, which means that T is a binary tree. □

We have shown that T is a labeled binary tree. This enables us to apply an inorder tree traversal technique as the function g_1^{-1} to construct a well-formed $\mathsf{LTL_f}$ formula H^{φ}.

Claim 3.3.10 $\forall \lambda \in E^+, \lambda \models H^\varphi$.

Proof. Since H is a solution for $f(\mathcal{L})$, we have that $\forall \langle e, C\rangle \in E^+_{LAS}, \exists A \in AS(B_{LAS} \cup C \cup H)$ s.t. A extends e. Due to the way examples are transformed to CDPIs in Definition 3.3.2, `sat` $\in A$ for every such A. Suppose, towards contradiction, that there exists $\lambda \in E^+$ s.t. $\lambda \not\models H^\varphi \iff \lambda, 0 \not\models H^\varphi$. Note that A will be identical to the interpretation described in Definition 3.3.4, except for the specifics of the examples and the labeled binary tree which fits them. Due to this, Lemma 3.3.6, and Lemma 3.3.7, we can conclude that `holds(0, 1)` $\notin A$. Since the literal `sat` can only be derived by applying a rule whose body involves `holds(0, 1)`, and this is not satisfied, `sat` cannot be in A. If `sat` were to remain in A despite the rule's body being unsatisfied, this would violate the condition that A is a minimal model for Σ. Since we initially assumed that `sat` $\in A$, contradiction. In particular, recall that there exists an answer set for the program $B_{LAS} \cup C \cup H$, i.e. B_{LAS} and H are fixed in this program, so H adheres to all the constraints present in B_{LAS} and since the same program H fits all positive examples one by one, it also fits all of them at the same time. Thus, H^φ must fit all positive traces. ■

Claim 3.3.11 $\forall \lambda \in E^-, \lambda \not\models H^\varphi$.

Proof. The proof is by contradiction and is analogous to the proof of Claim 3.3.10. ■

■

And so, we can conclude with a theorem in which we put all the aforementioned results together.

Theorem 3.3.12 $\text{PL}_{\mathsf{LTL_f}}$ reduces to $\text{ILP}^{context}_{LAS}$.

Proof. We must show that for an arbitrary $\text{PL}_{\mathsf{LTL_f}}$ task, there is an $\text{ILP}^{context}_{LAS}$ task such that the sets of their inductive solutions are the same. Due to our previous results, we can immediately conclude that $\text{PL}_{\mathsf{LTL_f}}(\mathcal{L}) = \text{ILP}^{context}_{LAS}(f(\mathcal{L}))$ where $\mathcal{L}$ is an arbitrary $\text{PL}_{\mathsf{LTL_f}}$ task and $f(\mathcal{L})$ is its transformed version. The $\subseteq$ direction has been proved in Lemma 3.3.3 and the $\supseteq$ direction has been proved in Lemma 3.3.9. ■

3.4 Optimal solutions and weighted examples

This section explores optimal solutions for the learning problems discussed, explains their significance, and introduces a learning problem where fitting all examples is not required.

In Section 3.3 we demonstrated that the sets of solutions of $\text{PL}_{\mathsf{LTL_f}}$ and its $\text{ILP}^{context}_{LAS}$ instance correspond. However, the $\text{PL}_{\mathsf{LTL_f}}$ learning task, as defined in Definition 3.1.1 without any size constraints, admits a trivial solution, as shown below.

Fact 3.4.1 (Trivial solution of $\text{PL}_{\mathsf{LTL_f}}$) Let $\mathcal{L} = \langle \mathcal{P}, E^+, E^-\rangle$ be an arbitrary passive learning task of $\mathsf{LTL_f}$ formulae. Because the positive and negative examples are disjoint, i.e. $E^+ \cap E^- = \emptyset$, we always have an $\mathsf{LTL_f}$ formula φ which solves the learning task. Specifically, this formula is:

$$\varphi = \bigvee_{\lambda \in E^+} \bigwedge_{i \in time(\lambda)} \left[\mathsf{X}^i \left(\bigwedge_{p \in \lambda_i} p \wedge \bigwedge_{q \notin \lambda_i} \neg q \right) \wedge \neg \mathsf{X}^{|\lambda|} \top \right]$$

The X^i means that the temporal next operator is applied i times. The formula φ intuitively

means that we explicitly enumerate all the propositions which are true and the negations of the propositions which are not true in the order that they appear for each positive trace. Note that $\neg \mathsf{X}^{|\lambda|}\top$ means that there is no subsequent time step, so the formula captures exactly the length of each trace.

Such a formula is uninteresting because it simply enumerates each model as an $\mathsf{LTL_f}$ formula. This is a clear case of overfitting, rendering the solution practically useless. It fails to generalize, which defeats the purpose of learning general solutions applicable to unseen data. As noted by Neider and Gavran [2018], adding an optimality criterion, such as the formula length, to the original task yields smaller, more general, and more human-comprehensible, formulae. We thus introduce a modified version of the learning task.

Definition 3.4.2 (Optimal solution of a $\mathrm{PL}_{\mathsf{LTL_f}}$ passive learning task) Let $\mathcal{L} = \langle \mathcal{P}, E^+, E^- \rangle$ be a $\mathrm{PL}_{\mathsf{LTL_f}}$ task. Let $|\cdot|$ be the function computing the length of a logical formula. An $\mathsf{LTL_f}$ formula φ, written in $\mathcal{P}$, is an optimal solution of $\mathrm{PL}_{\mathsf{LTL_f}}$ if and only if φ is a solution of $\mathrm{PL}_{\mathsf{LTL_f}}$ and there is no $\mathsf{LTL_f}$ formula φ' written in $\mathcal{P}$ that is a solution of $\mathrm{PL}_{\mathsf{LTL_f}}$ and $|\varphi'| < |\varphi|$.

Note that, intuitively, the optimal solutions of $\mathrm{PL}_{\mathsf{LTL_f}}$ and $\mathrm{ILP}^{context}_{LAS}$ (Definition 2.3.3) correspond, as their optimality criteria are both defined in terms of solution length. Because ILASP searches for optimal solutions when executed, we will obtain minimal-length $\mathsf{LTL_f}$ formulae that fit the data. This would offer a generalizable solution relevant to new data and various applications. A formal proof of this fact is out of the scope for this thesis and left for future work.

Until now, we've implicitly assumed error-free data. This is often unrealistic, as real-world data acquisition, whether through sensors or demonstrations of tasks by humans, is prone to errors. Learning from flawed data can lead to overly complex solutions that, while fitting the training set well, generalize poorly to real-world scenarios. Therefore, to account for varying data quality, we assign each trace a weight representing its importance or score and modify the learning task.

Definition 3.4.3 (Passive learning task $\mathrm{PL}^{weight}_{\mathsf{LTL_f}}$) Let $\mathcal{P}$ be a set of propositional symbols. A $\mathrm{PL}^{weight}_{\mathsf{LTL_f}}$ passive learning task is a tuple $\mathcal{L} = \langle \mathcal{P}, E^+_w, E^-_w \rangle$ where:

- $E^+_w = \{\langle \lambda, w \rangle \mid \lambda \text{ is a trace over } \mathcal{P},\ w \in \mathbb{R}\}$ is a set of positive traces where w is the weight associated with the positive trace λ.
- $E^-_w = \{\langle \lambda, w \rangle \mid \lambda \text{ is a trace over } \mathcal{P},\ w \in \mathbb{R}\}$ is a set of negative traces where w is the weight associated with the negative trace λ.
- $\{\lambda \mid \langle \lambda, w \rangle \in E^+\} \cap \{\lambda \mid \langle \lambda, w \rangle \in E^-\} = \emptyset$.

Let $|\cdot|$ be the function computing the length of a logical formula. Let $uncov(\varphi, \mathcal{L})$ be the following set:

$$uncov(\varphi, \mathcal{L}) = \{\langle \lambda, w \rangle \mid \langle \lambda, w \rangle \in E^+ \text{ and } \lambda \not\models \varphi\} \cup \{\langle \lambda, w \rangle \mid \langle \lambda, w \rangle \in E^- \text{ and } \lambda \models \varphi\}.$$

The score of φ is defined as $S(\varphi, \mathcal{L}) = |\varphi| + \sum_{\langle \lambda, w \rangle \in uncov(\varphi, \mathcal{L})} w$. An $\mathsf{LTL_f}$ formula φ written in $\mathcal{P}$ is a *solution* of a $\mathrm{PL}^{weight}_{\mathsf{LTL_f}}$ task (denoted $\varphi \in \mathrm{PL}^{weight}_{\mathsf{LTL_f}}(\mathcal{L})$) if and only if $S(\varphi, \mathcal{L})$ is finite.

Given this new framework, we can quite easily cast it as an $\mathrm{ILP}^{noise}_{LAS}$ instance using a similar transformation function we have defined before.

Definition 3.4.4 (Transformation f from $\text{PL}^{weight}_{\mathsf{LTL_f}}$ to ILP^{noise}_{LAS}) Let $\mathcal{L} = \langle \mathcal{P}, E^+_w, E^-_w \rangle$ be an arbitrary instance of a $\text{PL}^{weight}_{\mathsf{LTL_f}}$ task. Let $f(\mathcal{L}) = \langle B_{LAS}, \mathcal{H}_{LAS}, \langle E^+_{LAS}, E^-_{LAS} \rangle \rangle$ be the ILP^{noise}_{LAS} instance where the individual components are defined as follows.

- B_{LAS} and H_{LAS} are defined the same as in Definition 3.3.2.
- Let $\langle \lambda, w \rangle \in E^+_w$ be an arbitrary positive weighted linear trace. Then, we can represent the trace as a CDPI $e^\lambda_{wcdpi} = \left\langle e_{pen}, \left\langle e^{inc}_{pi}, e^{exc}_{pi} \right\rangle, e_{ctx} \right\rangle$ as follows:
  - $e_{pen} = w$, $e^{inc}_{pi} = \{\texttt{sat}\}$, $e^{exc}_{pi} = \{\}$
  - $e_{ctx} = \{\texttt{trace(}i\texttt{).} \mid i \in time(\lambda)\} \cup \{\texttt{trace(}i\texttt{, }p\texttt{).} \mid i \in time(\lambda), p \in \lambda[i]\}$

  Then, $E^+_{LAS} = \{e^\lambda_{wcdpi} \mid \lambda \in E^+_w\}$.
- E^-_{LAS} is defined the same as E^+_{LAS} for each negative weighted linear trace $\langle \lambda, w \rangle \in E^-_w$.

However, a result similar to Theorem 3.3.12 showing that the solution spaces correspond is out of the scope for this thesis. If such a result were to be established, one could leverage ILASP3 which deals with weighted examples to learn $\text{PL}^{weight}_{\mathsf{LTL_f}}$ formulae in practice.

Note that $\text{PL}^{weight}_{\mathsf{LTL_f}}$ also has a trivial solution which corresponds to Fact 3.4.1 or one that disregards minimizing the (finite) penalties altogether. To avoid such trivial outcomes and properly account for the importance of each trace, we define an optimal solution.

Definition 3.4.5 (Optimal solution of a $\text{PL}^{weight}_{\mathsf{LTL_f}}$ passive learning task) Let $\mathcal{L} = \langle \mathcal{P}, E^+_w, E^-_w \rangle$ be an arbitrary passive learning task. An $\mathsf{LTL_f}$ formula φ, written in $\mathcal{P}$, is an *optimal solution* of $\text{PL}^{weight}_{\mathsf{LTL_f}}$ if and only if:

- φ is a solution of $\text{PL}^{weight}_{\mathsf{LTL_f}}$, and
- there is no $\mathsf{LTL_f}$ formula φ' written in $\mathcal{P}$ that is a solution of $\text{PL}^{weight}_{\mathsf{LTL_f}}$ and $score(\varphi', \mathcal{L}) < score(\varphi, \mathcal{L})$.

Optimal solutions are particularly important in this learning framework, making software that finds them for $\text{PL}^{weight}_{\mathsf{LTL_f}}$ highly desirable. Intuitively, the optimal solutions of $\text{PL}^{weight}_{\mathsf{LTL_f}}$ and ILP^{noise}_{LAS} (Definition 2.3.5) seem to correspond, as both use solution length and the sum of the weights of the uncovered examples in their optimality criteria. A formal proof of this result is left for future work.

Chapter 4

Learning temporal advice formulae

In this chapter we set out to investigate what exactly are temporal advice formulae, write down some desiderata, formalize them, all this to guide our further empirical efforts to learn them. This chapter is structured as follows:

- Section 4.1 introduces a simple scenario to illustrate our ideas, provides the motivation for temporal advice formulae, and formally defines the $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ logic used to express them.
- Section 4.2 adapts the learning task from Ielo et al. [2023] to learn temporal advice formulae.
- Section 4.3 demonstrates how ILASP can be used with the adapted learning task to learn temporal advice formulae on an illustrative dataset for the simple scenario described earlier, and discusses the quality of the results.

4.1 Defining temporal advice formulae

First, let us consider a simple running example that will be used throughout this chapter to illustrate the methodology and the initial experiments in the next chapter, thanks to its straightforward representation.

The setup consists of N gems randomly placed without overlap along a line of length L, with an agent starting at a random position on this line. The goal is to collect all the gems to win the game using the least amount of actions possible. The agent has perfect knowledge of the environment, including the total number of gems. The user can command the agent to move left, move right, or pick up a gem. At each time step, the agent performs exactly one action. The environment is visually depicted in Figure 4.1.

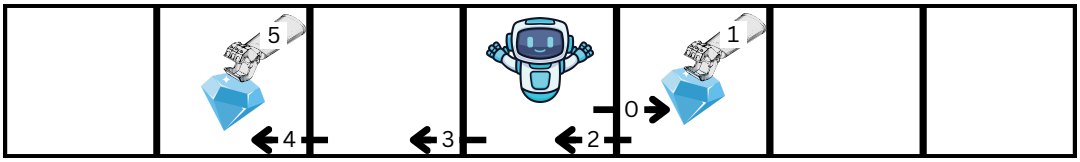


Figure 4.1: An optimal action sequence to solve this particular instance of the gem pickup game. The robot icon shows where the agent starts, and each numbered action indicates which action was taken at which timestep.

Imagine your very young sibling, possibly experiencing video games for the first time, playing gem pickup and controlling the agent. They might feel frustrated, not knowing what to do to progress. As their older sibling, you can help by giving them hints or advice, telling them which actions or goals will help them win. The easiest to understand advice could be ones that tell

exactly what action to take depending on the state of the game, without any notion of memory or future planning. For instance, "if the distance to the target gem is strictly positive, go right".

Given an arbitrary environment, it is difficult to say what kind of actions or goals are desirable for the agent to take in order to win. Rather than manually analyzing the environment, formalizing our observations, and then supplying them to planning agents, as done in prior work, e.g. [Alshiekh et al., 2017, Camacho et al., 2019, Illanes et al., 2020, De Giacomo et al., 2021], we aim to learn these advice automatically from data, for example, from demonstrations of good and bad gameplays in the gem pickup scenario. The ultimate use of these advice is that they should serve as an interpretable symbolic specification to accelerate planning or learning, reducing the amount of data needed to learn good planning policies, runtime, and ideally improving result quality.

Learning time-independent advice formulae has already been investigated in Meli et al. [2024]. The language that the authors use to describe these time-independent advice is answer set programming. Their learned advice formulae take the form $action \leftarrow prec_1 \wedge \ldots \wedge prec_n \wedge \neg prec_{n+1} \wedge \ldots \wedge \neg prec_m$. Intuitively, this rule means that if all the positive conditions specified in the rule's body are true, and none of the negative conditions are true, then the current state of the environment makes it advantageous for the planner to carry out the action defined in the rule's head. Within their formalism, the aforementioned time-independent advice for the gem pickup task could be represented as follows

```
right :- target(G), distance(G, D), D >= 1.
```

The authors showed that even such time-independent advice can help online planners gain better results faster.

However, time-independent advice have limitations. For example, navigating a maze requires following a sequence of steps rather than just reacting to the current position. Temporal advice address the need for planning by incorporating time, which allows them to express action sequences and long-term goals. Such temporal advice can describe either low-level action sequences, such as "move right, then right again," or high-level goals, like "at some point, all gems are collected." This analogy motivates our approach to guiding artificial planning agents through advice, thereby injecting domain knowledge into the learning process and reducing the required amount of trial-and-error for effective performance.

A high-level temporal specification, as in the latter example, defines a desired world state without detailing how to achieve it. Thus, a planning algorithm is still required to figure out the exact steps, and the temporal specification would guide the planner to satisfy this specification. However, even actionable temporal specifications are not intended to be followed blindly. All of these formulae are meant to softly guide the planner. The situation changes when we can provably show that the temporal specification must be fully adhered to, such as in cases where safety behavior is learned from verified data. In that case, the planner could be guided in a hard manner, some of its actions should not be performed at all if they do not adhere to the temporal specification.

It is also important to recognize that high-level goals are not necessarily expressed as liveness specifications. For instance, for the Pac-Man game, "always avoid ghosts" is a valuable temporal property, even though it does not directly specify the actions necessary to satisfy it.

Linear temporal logic [Pnueli, 1977] is a natural formalism for reasoning about future actions. This is illustrated by Toro Icarte et al. [2018], who used LTL_f to provide temporal advice to reinforcement learning agents. In fact, it is due to their work that we use the term "advice" and not e.g. "heuristics" (as Meli et al. [2024] do) or "instructions". We adapted their definition of MDPs with a signature (Definition 2.4.5) by emphasizing action predicates in addition to environment predicates, allowing us to provide and learn actionable temporal advice. We now define the central concept that we will strive to learn using ILASP.

Definition 4.1.1 (Temporal advice formula for an MDP) Let $\mathcal{M}$ be a MDP for which we want to provide advice. Specifically, let $\mathcal{M} = \langle S, A, P, R, \gamma, \mathcal{S}, L\rangle$ where signature $\mathcal{S} = \langle \Omega, C, arity\rangle$ where $\Omega = A \cup E$ is comprised of the finite set of action predicates A and environment description predicates E. A *temporal advice formula* is an $\mathsf{LTL_f}$ with an existential quantifier formula written in $GA(\mathcal{S})$. We refer to this logic as $\mathsf{LTL_f^a}$.

The formulas of this logic are evaluated on finite linear traces as $\mathsf{LTL_f}$ is, with the addition of V, a function which assigns variables to domain objects. The semantics of the formulas of $\mathsf{LTL_f^a}$ are the same as $\mathsf{LTL_f}$ with the additions of the existential quantifier $\exists$ whose semantics are:

$$\lambda, i, V \models \exists(x \in D).\varphi \text{ iff for some } a \in D, \text{ it is the case that } \lambda, i, V(x/a) \models \varphi$$

Clearly, temporal advice formulae can also be defined analogously for a POMDP.

As Toro Icarte et al. [2018] do, we shall use the first-order existential quantifier to abbreviate disjunctions. Specifically, if $T = \{t_1, \ldots, t_k\} \subseteq C$ is a set of constant symbols, then $\exists(x \in T).\varphi(x) \stackrel{\text{def}}{=} \varphi(t_1) \vee \ldots \vee \varphi(t_k)$. Furthermore, we use comparison operators to abbreviate filtering. Let $\star$ be one of the comparison operators $<, \leqslant, >, \geqslant, =,$ or $\neq$. Then,

$$\exists(x \in \{t \mid t \star c, t \in T\}).\varphi(x) \stackrel{def}{=} \bigvee_{x \in \{t \mid t \star c, t \in T\}} \varphi(x).$$

We write $\varphi(x) \star c$ if the domains of x and c are clear from context.

Indeed, we can express the aforementioned statements using this language. For instance, the low-level advice "move right, then right again" can be formalized as $right \wedge \mathsf{X} right$. We can restate the high-level advice "at some point, all gems are collected" with an existential quantifier which $\mathsf{LTL_f^a}$ permits as "at some point, it is not the case that there is a gem that is not picked up", or, formally: $\mathsf{F}\neg(\exists(G \in gems).(\neg picked(G)))$.

Time-independent advice, as in Meli et al. [2024], is a subset of temporal advice formulae. Specifically, if r is a learned ASP advice which is an normal rule in ASP, then we can represent it as a temporal advice formula $\mathsf{G}(\bigwedge_{p \in body(r)} \rightarrow head(r))$. That is, at each time step, we check if the environment satisfies the preconditions (body) of the rule, and if so, then the action (head) should be true.

One could reasonably ask why we use $\mathsf{LTL_f}$, a language for describing temporal properties of traces as opposed to a conventional planning language such as PDDL [Ghallab et al., 1998]? The main reason is that we build on the work of [Ielo et al., 2023] and use their existing code, avoiding the need to implement new semantics ourselves. Additionally, many RL-guidance techniques discussed in Section 1.2 use $\mathsf{LTL_f}$ as their specification language which suggests that adopting this formalism could enable us to effectively apply and build upon these methodologies. Finally, PDDL can be expressed in $\mathsf{LTL_f}$ [De Giacomo and Vardi, 2013].

4.1.1 Actionable temporal advice

We have stressed that $\mathsf{LTL_f^a}$ can be used to describe both action sequences, but also high-level goals. In the remainder of the thesis, we will focus on learning actionable temporal advice. This is partly motivated due to the approach of Meli et al. [2024] but also because most existing work which guided RL used high-level temporal specifications, and the low-level, actionable advice are not explored. We note that learning high-level goals is a matter of slightly changing the learning task and encoding of the problem, introducing more environmental predicates with which to describe these high-level states. The empirical investigation of this research direction is left for future work.

We also acknowledge that removing action predicates entirely, and relying solely on low-level environment descriptors, could potentially enable us to learn the environment's dynamics. For

example, for the scenario of chess, if we only observe predicates like `piece_position(P, X, Y)`, `piece_type(P, Type)`, and `board_state(Z)`, without explicitly having actions in the dataset like `move_piece(P, X, Y, X', Y')`, we could potentially infer how those low-level state predicates evolve over time based on the observed patterns. We could, for instance, learn the legal moves of each piece by observing how their positions change across different board states. Again, while this approach presents an intriguing avenue for future research, it falls outside the scope of this work. This is because, in our current setting, we assume we possess a pre-existing model of how the environment changes in response to actions. Our focus is on utilizing the given environment dynamics to learn actionable advice, rather than learning those dynamics from scratch.

Actionable temporal advice formulae should express the temporal dependencies of actions and their preconditions. To ensure this in the learning task, we require every hypothesis to include at least one action predicate. This is done by encoding the action predicates as propositions because propositions are needed for any well-formed $\mathsf{LTL_f}$ formula.

4.1.2 Beyond propositional $\mathsf{LTL_f}$

While Meli et al. [2024] represented advice in the form of *action* ← *preconditions* and allowed n-ary predicates in both *action* and *preconditions*, the paper Ielo et al. [2023] focused on learning standard $\mathsf{LTL_f}$ formulae, which do not allow predicates with variables.

We argue that to effectively express temporal advice in $\mathsf{LTL_f}$, we require predicates with variables, not just propositions. Specifically, the existential quantifier $\exists$ will be used to bind the objects that are common to both the action and its environmental preconditions. These variable-based predicates offer a more compact and general representation of temporal patterns.

As a running example, consider the predicate $obstacle(x, y)$ which describes that there is some obstacle at coordinates $x \in X$ and $y \in Y$. The sets X and Y are finite sets of numbers. An example of the grounded version of this predicate (and thus, a proposition) would look like $obstacle(1, 2)$.

Increase in hypothesis space size. Expressing advice with propositions essentially requires grounding the *obstacle* predicate for every possible combination of x and y values. So, in order to learn advice as propositional $\mathsf{LTL_f}$ formula, the predicate *obstacle* would have to be grounded, with each possible combination of $x \in X$ and $y \in Y$ values multiplying the hypothesis space by a factor of $|X| \cdot |Y|$. In contrast, allowing to learn the predicate $obstacle(x, y)$ with variables allows us to represent the desired concept of an obstacle as a single entity, resulting in a much more compact hypothesis space.

Limitations in expressing bounds. Predicates with variables enable us to efficiently represent conditions involving comparisons. For instance, $\exists(x, y \in coords, x < 2).(obstacle(x, y))$ concisely expresses that there's an obstacle with an x-coordinate less than 2. Assuming $coords = \{0, 1, 2, 3\}$, achieving the same with propositions would require enumerating all possible cases:

$$(obstacle(0, y) \vee obstacle(1, y)) \wedge \neg obstacle(2, y) \wedge \neg obstacle(3, y).$$

This propositional representation is not only cumbersome but also less intuitive for people to understand. For brevity, our example does not enumerate the available values for y, doing so would increase its size by a factor of $|coords|$. While encoding bounds or comparisons using grounded propositions in ASP and ILASP is technically possible, it is significantly more complex and time-consuming.

For more details on the design decisions of encoding temporal advice in ILASP, refer to Appendix B.

4.2 Specifying the learning problem in ILASP

In this section, we adapt the ILASP encoding for learning $\mathsf{LTL_f}$ formulae from Ielo et al. [2023] (partly described in Section 3.2)and incorporate ideas from Meli et al. [2024] and Section 4.1. This adapted approach allows us to learn $\mathsf{LTL_f^a}$ using ILASP.

First, let us see how the hypothesis space is originally declared in ILASP by Ielo et al. [2023] for learning $\mathsf{LTL_f}$ formulae. We will later modify this to be able to learn temporal advice formulae. The hypothesis space in inductive logic programming is declared using a mode bias. In ILASP, we can specify this using the `#modeh` (to declare the available heads of rules) and `#modeb` (to declare the available bodies of rules) keywords. Here, Ielo et al. [2023] simply use `#modeh` to specify the hypothesis space, meaning that the hypotheses will be made up of basic facts, without any bodies. Given that the hypotheses here are $\mathsf{LTL_f}$ formulae which are represented in ASP as syntax trees (described in Section 3.2.2), we will strive to learn parts of the syntax tree, i.e. the labeled nodes and edges connecting them. Consider the following code:

```
1  #constant(node_id, 1..n).
2  #constant(op, next).
3  #constant(op, until).
4  #constant(op, eventually).
5  #constant(op, always).
6  #constant(op, and).
7  #constant(op, neg).
8  #constant(op, or).
9  #constant(op, implies).
10 % For every p in propositions
11 #constant(atom, p).
12 #modeh(edge(const(node_id), const(node_id))).
13 #modeh(label(const(node_id), const(op))).
14 #modeh(label(const(node_id), const(atom))).
```

- Line 1: enforce a finite `n` amount of nodes in the syntax tree (using the type `node_id`) to prevent running out of computational resources.
- Lines 2-9: define the type `op` to represent the operators from $\mathcal{O}$ (defined in Section 3.2.2).
- Line 11: include each proposition $p \in \mathcal{P}$ to ensure that any proposition present in the example traces could be part of the hypothesis.
- Lines 12-14: define the learnable elements of the syntax tree. The predicate `edge/2` defines connections between node pairs, while `label/2` assigns meaning to each node by representing it with either logical operators or atomic propositions.

The hypothesis space encompasses all potential edges and node labels, including logical operators and atoms, and their combinations. However, not every combination results in a valid $\mathsf{LTL_f}$ formula. The structural validity criteria detailed in Section 3.2.3 eliminate ill-formed syntax trees as we have argued in Section 3.3 and illustrate practically in Appendix A.

Now, we introduce the temporal advice encoding in a step-by-step manner and provide concise explanations and side-by-side code comparisons illustrating each design decision.

First, we modify the hypothesis space and the kind of facts we can learn. Listing 4.1 presents the code for learning propositional atoms, while Listing 4.2 shows the code for learning actions with preconditions, including 0-ary actions. Instead of allowing any atom encountered in the traces to be learned as is done above, we restrict to learning only actions in the head of the rule which will allow us to learn actionable temporal advice, adhering to Section 4.1.1. We observed in our initial experiments that a time variable must be included in the label of n-ary predicates for proper functionality. We defer the explanation for this until the mode bias.

Listing 4.1: LTL_f atoms

```
1 #modeh(label(const(node_id), const(atom))).
2 % for every proposition p
3 #constant(atom, p).
4 proposition(p).
```

Listing 4.2: $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ atoms

```
1 % for every action act
2 % 0-ary
3 #modeh(label(const(node_id), act1)).
4 proposition(act1).
5 % n-ary
6 #modeh(label(const(node_id), act2(var(object)), var(time))).
7 proposition(act2(Obj)) :- trace(_, act2(Obj)).
```

The introduction of `var(time)` in line 6 in Listing 4.2 adds a ternary `label` predicate, necessitating adjustments to our syntax tree constraints. The key issue of this change is that the rules which have `label/3` in its head will result in multiple occurrences of the grounded predicate within the Herbrand interpretation. In general, we want to prevent the same node being labeled with multiple labels. We can no longer naively only allow a single grounded version to be labeled per syntax tree node as we have variables, and, once grounded, we will have multiple occurrences of this same action, e.g. `label(1,pickup(1),1)`, `label(1,pickup(2),1)`, `label(1,pickup(3),1)`. We must permit such multiple `label` occurrences for the same syntax tree node because ultimately we want to learn the rule's general version, e.g. `label(1,pickup(X),1)`. However, it's crucial to prevent scenarios where the same node is annotated with multiple action labels e.g. `label(1,pickup(X),1)`, `label(1,left,1)`. Listings 4.3 and 4.4 compare the original code with our modified solution, which addresses the issue of multiple labels.

Listing 4.3: Syntax tree constraints for LTL_f

```
1 % Exactly one label per node
2 :- node(X), not label(X,_).
3 % Only one label per node
4 :- label(X,A), label(X,B), A != B.
5 ...
6 :- arity(X,N), label(X,Y), not symbol(Y,N).
```

Listing 4.4: Syntax tree constraints for $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$

```
1  node(X) :- label(X,_,_).
2  % It cannot be the case that a node does not have a label
3  :- node(X), not label(X,_), not label(X,_,_).
4  % Define constraints for labels with arity = 0:
5  % No node can have two different arity 0 labels.
6  :- label(X, A), label(X, B), A != B.
7  :- label(X, A, _), label(X, B), A != B.
8  :- label(X, A), label(X, B, _), A != B.
9  % For every action with arity > 0:
10 operation_type(act1, act1(V1)) :- label(_, act1(V1), _).
11 % A node cannot have labels of different types if arity > 0.
12 :- label(X, A, _), label(X, B, _), A != B,
13    operation_type(T1, A), operation_type(T2, B), T1 != T2.
14 ...
15 :- arity(X,N), label(X,Y), not symbol(Y,N).
16 :- arity(X,N), label(X,Y,_), not symbol(Y,N).
```

Given our current constraints, we still do not rule out one undesirable scenario. Namely, when the same node is labeled with the same action, but different bodies, e.g.

```
label(1,pickup(X),T) :- gem(X,T).
label(1,pickup(X),T) :- distance(X,D,T), D < 2.
```

Disallowing such rules is not straightforward, and some advanced ILASP meta rules are needed as shown in Listing 4.5. Note that ILASP version $\geqslant$ 4.4.1 is necessary to support such ASP rule injection during the solving stage.

Listing 4.5: Disallowing ternary label predicates

```
#bias("attribute(ternary) :- head(label(_,_,_)).").
#inject("all_active.").
#inject("rule_ids_to_constrain(X) :- attribute(X,ternary).").
#inject("rule_ids_to_constrain_selected(X) :- nge_HYP(X),
    rule_ids_to_constrain(X).").
#inject(":- rule_ids_to_constrain_selected(X),
           rule_ids_to_constrain_selected(Y), X!=Y.").
```

Since we now allow `label/3`, we need to modify the model checking to support it as done in Listing 4.6

Listing 4.6: Semantics for `label/2` and `label/3`

```
holds(T, X) :- label(X, A), proposition(A), trace(T, A).
holds(T, X) :- label(X, A, T), proposition(A), trace(T, A).
```

Having introduced variables into `label/3`, it requires us to ensure the safety of our resulting ASP rules. Remember that an ASP rule is considered safe (Definition 2.2.7) only if every variable present in the rule's head also appears within a positive literal in its body. To achieve this, we leverage the mode bias to enable the learning of positive predicates that utilize the same variables found in `label/3`. This approach also naturally allows us to model environment preconditions for agent actions. These preconditions can be empty, serving solely to ensure rule safety, or they can represent concrete conditions, such as "pick up a rock only if the distance to it is at most 0".

To specify when these preconditions are satisfied in ASP, we derive the necessary conditions directly from the provided trace, which can be seen in lines 4 and 6 of Listing 4.7. It's important to emphasize the use of `var(time)` in `label/3`. There is a purely technical reason for this: if the n-ary action predicates have any preconditions (literals in the rule's body), these preconditions should have happened at the same time as the action. If not for this, we risk expressing the preconditions for an action that have happened *at some point* in the trace, but not *at the same time* as the action did. In general, our base assumption here is that certain states of the environment prompts the agent to take a certain action, and we want to capture that in a normal ASP rule.

Listing 4.7: Define action preconditions using environmental descriptors

```
% This could serve as an empty precondition for action act1
% Because precondition1 is made true when act1 is performed
#modeb(1, precondition1(var(obj), var(time)), (positive)).
precondition1(ObjID, T) :- trace(T, act1(ObjID)).
% Example of a precondition that is true due to the environment
#modeb(1, precondition2(var(object1), var(object2), var(time)), (positive)).
precondition2(ObjID1, ObjID2, T) :- trace(T, prec2(ObjID1, ObjID2)).
```

We assume 0-ary actions do not have environment preconditions. As 0-ary action predicates are independent of objects, the state of environment objects is not important. Consequently, we've designed these actions to have no rule body, which also allows us to narrow the hypothesis space as shown in Listing 4.8.

Listing 4.8: Reduce search space by removing some bodies

```
% Edges shouldn't have a body
#bias(":- head(edge(X,Y)), body(_).").
% Every logical operator o should have no body
#bias(":- head(label(_, o)), body(_).").
% Every 0-ary action act should have no body
#bias(":- head(label(_, act)), body(_).").
```

Further reduction of the search space can be done based on the specific environment. For instance, take a look at our encoding of gem pickup in the code repository[1].

Discouraging trivial solutions. Working within the $\mathsf{LTL_f}$ formalism requires careful consideration to avoid learning overly general and ultimately useless advice. Specifically, we risk learning logical tautologies or the trivial $\mathsf{LTL_f}$ solution (Fact 3.4.1). While ILASP's preference for shorter solutions (see "optimal solution" in Definition 2.3.3) inherently discourages the trivial solution, tautologies (e.g., $right \rightarrow right$) can fit all $\mathsf{LTL_f}$ models. The issue of tautologies can be resolved by integrating negative examples into the ILASP task. This prevents tautologies from being learned, as they would incorrectly satisfy negative examples as well. We also acknowledge that it may not be possible to come up with negative examples in certain situations, in which case one could introduce an artificial negative example with infinite penalty to discourage tautologies. However, as demonstrated in Section 5.2.2, our specific context allows to straightforwardly define negative examples.

4.3 Learning temporal advice formulae for gem pickup

This section explains a method for learning temporal advice formulae for the gem pickup scenario using a specific dataset. It also covers the process of converting syntax trees to $\mathsf{LTL_f^a}$ and briefly discusses the quality of the results obtained from the dataset.

Definition 4.3.1 We define the signature for the gem pickup scenario as follows. This signature enables the specification of concrete temporal advice formulae and, importantly, is used to encode the scenario in ASP and define the ILASP learning task. Let $\mathcal{S}_{gp} = \langle \Omega, C, arity \rangle$ be a signature where

- $\Omega = A \cup E = \{left, right, pickup\} \cup \{dist, picked\}$.
- $C = gems \cup distances = \{G_1, \ldots, G_k\} \cup \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$ for k gems in the scenario.
- $arity(left) = arity(right) = 0$
  $arity(pickup) = arity(picked) = arity(gem) = 1$,
  and $arity(dist) = 2$.

Here, A represents the agent's available actions: left and right for movement, and $pickup(G)$ to collect a gem $G \in gems$. E is the set of environment description predicates, which are used to describe the state as the agent acts in the environment. Specifically, $picked(G)$ indicates whether a gem $G \in gems$ was successfully picked up, and $dist(G, D)$ describes the distance $D \in distances$ to each gem $G \in gems$. A positive distance implies the gem is D steps to the right of the agent, while a negative distance means it is D steps to the left.

With the help of Definition 4.3.1 and the general template described in Section 4.2, we can now define the temporal advice formulae learning task for the gem pickup scenario. For its full implementation, please refer to the gem pickup `.las` script.

In Figure 4.1 you can see a visually depicted trace of the gem pickup environment. We treat this as a positive example trace because all of the gems were picked up successfully. You can see in which order the actions were taken to pick up the gems. To see how the environmental predicates are evolving with each step, take a look at the encoding of this trace in ILASP:

```
1 #pos(ex1, {sat}, {}, {
2 trace(0).
```

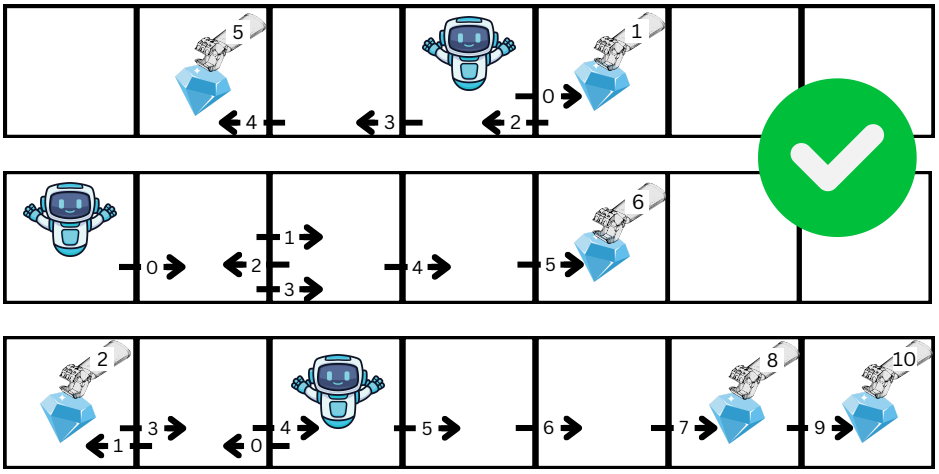
[1] ilasp_asp_tasks/gem_pickup.las

```
3  trace(0,right).
4  trace(0,dist(0,1)).
5  trace(0,dist(1,-2)).
6  trace(1).
7  trace(1,pickup(0)).
8  trace(1,dist(0,0)).
9  trace(1,dist(1,-3)).
10 trace(2).
11 trace(2,left).
12 trace(2,picked(0)).
13 trace(2,dist(0,0)).
14 trace(2,dist(1,-3)).
15 trace(3).
16 trace(3,left).
17 trace(3,picked(0)).
18 trace(3,dist(0,1)).
19 trace(3,dist(1,-2)).
20 trace(4).
21 trace(4,left).
22 trace(4,picked(0)).
23 trace(4,dist(0,2)).
24 trace(4,dist(1,-1)).
25 trace(5).
26 trace(5,pickup(1)).
27 trace(5,picked(0)).
28 trace(5,dist(0,3)).
29 trace(5,dist(1,0)).
30 }).
```

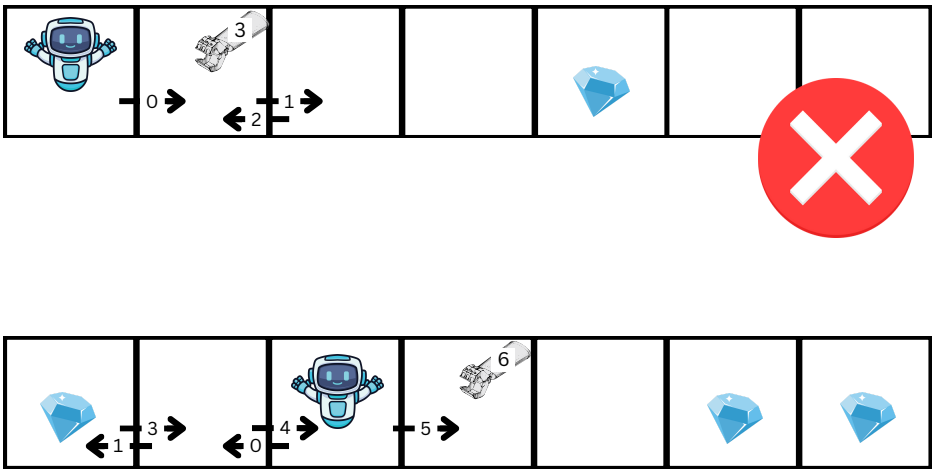
In order to learn more generalizable and better hypotheses, we should use more data. Thus, we manually prepare a dataset of five traces in total (including the aforementioned one). Figure 4.2 shows the three positive traces and two negative ones. In the positive traces, the robot always picks up all the games, in the negative traces, the trace ends when the robot attempts to pick up a gem without being on it.

To get a sense of how well our method works without running it yet, we may speculate about the potential outputs. That is, what temporal behaviour is exhibited in Figure 4.2 by theas agent? How would this behaviour be expressed in $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ using the signature Definition 4.3.1? Let us investigate a few examples.

- $\varphi_1 = \mathsf{X}\ right$ is not going to be a good hypothesis as it does not fit the second and third positive traces and fits the first negative trace.
- $\varphi_2 = right \rightarrow right$ is a logical tautology, it is true in every trace. So, while it fits the positive traces, it also fits the negative traces which we do not want. Also, it is simply not informative of our traces at all as any logical tautology does not bear additional information about its model.
- $\varphi_3 = \mathsf{GF}(\exists(G \in gems).pickup(G))$ means that at every time step, it will eventually be the case that the *pickup* action on any gem will be performed. This is a reasonably good hypothesis since all positive traces conclude with a *pickup* action. However, the same applies to all the negative traces as well, with the key distinction being that in positive traces, the agent stands on the gems it intends to collect. So, this hypothesis is too general as it fits the negative traces as well.
- $\varphi_4 = \mathsf{G}(\exists(G \in gems, D \in distances, -1 < D < 1).(pickup(G) \wedge dist(G, D)))$ means that it is always the case that the agent will perform the `pickup` action on a gem G if the distance to G is 0 (because we use integers as distance objects in the signature). This hypothesis is excellent because it fits the positive examples and does not satisfy the negative ones. We aim to learn such hypotheses using ILASP.



(a) A set of three positive traces. The episodes end when all of the gems are collected.



(b) A set of two negative traces. The episodes end when the *pickup* action is performed on a cell without a gem.

Figure 4.2: The training set for the gem pickup environment. The image of the robot indicates the starting position of the agent, the arrows show the movement and at what timestep it was done. The robot hand indicates the *pickup* action being performed at that cell.

The $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ representation of φ_4 is rather cumbersome, but its corresponding ASP formulation, as would be returned by ILASP, is more readable:

```
1  label(1,always).
2  edge(1,2).
3  label(2,pickup(G),T) :- dist(G,D,T), -1 < D < 1.
```

Let us first address the process of converting ASP syntax trees to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ (e.g. the above code to φ_4). When we convert the ASP formula to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$, we get rid of the time variable. As mentioned in Section 4.2, that variable has a purely technical role in ILASP. Observe that the body of the `label/3` rules defines over which sets we are quantifying in the $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ formula. Also, the action and environment predicates after the quantification are joined together by conjunctions. This type of transformation captures the kind of semantics we aim to express with the ASP syntax tree. Keep in mind that all of the learned formulae satisfy the trace at timestep 0, but are not necessarily expected to do so beyond that.

And thus finally, we execute ILASP version 2i on the gem pickup task and the five described traces. We use a custom pylasp script (see Section 5.2.4 for more information) to retrieve multiple near-optimal hypotheses, not only the optimal one. Here, we present a few results as returned by

ILASP, interpret their meaning, and show how they would be converted to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$.

Consider H_1:

```
1 label(1,eventually).
2 edge(1,2).
3 label(2,pickup(G),T) :- dist(G,D,T), -1 <= D <= 1.
```

H_1 converted to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ would be:

$$\mathsf{F}(\exists(G \in gems, D \in distances, -1 \leqslant D \leqslant 1).(pickup(G) \wedge dist(G, D)))$$

This formula serves as an actionable heuristic because it defines the conditions of the environment needed in order to perform the pickup action. The only issue is the eventually modality, which does not specify exactly when the action must be performed. However, we do not need to address this detail here, as its handling will depend on the specific planning algorithms and how they utilize these $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ formulae.

It is encouraging that we were able to learn this formula. It closely matches our manually derived prediction φ_4. The key distinction lies in the use of "eventually" versus "always," with "eventually" being a less restrictive modality. One might also question the equality appearing in both comparisons. Note that in the positive examples, the pickup action was performed when the distance to the picked gem was 0, and in the negative examples, the pickup was performed when the distance to the picked (or any) gem was more than 1 and/or less than -1. Therefore, this hypothesis remains consistent with the data.

Consider H_2:

```
1 label(1,always).
2 edge(1,2).
3 label(2,neg).
4 edge(2,3).
5 label(3,pickup(G),T) :- dist(G,D,T); D >= 1.
```

H_2 converted to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ would be:

$$\mathsf{G}\neg(\exists(G \in gems, D \in distances, D \geqslant 1).(pickup(G) \wedge dist(G, D)))$$

This temporal advice formula defines a safety condition similar to H_2, but due to the always modality, it instructs the agent to *never* pick up a gem when the distance is 1 or more. This is again encouraging, as the formula is logically equivalent to φ_4 and aligns with the behavior we aimed to capture from the dataset.

Consider H_3:

```
1 label(1,eventually).
2 edge(1,2).
3 label(2,and).
4 edge(2,3).
5 edge(2,4).
6 label(3,next).
7 edge(3,5).
8 label(5,pickup(G),T) :- gem(G,T).
9 label(4,right).
```

H_3 converted to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ would be:

$$\mathsf{F}(\mathsf{X}\,[\exists(G \in gems).(pickup(G))] \wedge right)$$

This temporal advice formula describes a compound action: at some point, move right and then pick up a gem. Since in this case the pickup action has no specific preconditions (aside from the gem precondition, which serves as a placeholder to enable the use of the existential quantifier), the exact timing of the action is unclear. Once again, it would be up to the planners to determine how to utilize this advice.

Consider H_4:

```
1  label(1,until).
2  edge(1,2).
3  edge(1,3).
4  label(2,eventually).
5  edge(2,4).
6  label(4,right).
7  label(3,pickup(G),T) :- gem(G,T).
```

H_4 converted to $\mathsf{LTL}_\mathsf{f}^\mathsf{a}$ would be:

$$[\mathsf{F}\ right]\,\mathsf{U}\,[\exists(G \in gems).(pickup(G))]$$

This formula describes that eventually *right* should be true at least until the time when a *pickup* on any gem is performed. While again we defer the usage of these formulae when talking about specific planning algorithms, we selected this solution to demonstrate that some solutions may be very ambiguous to put into practice. To show this, consider the following four traces:

$$\lambda_1 = \{\}, \{\}, \{pickup(2)\}, \{right\}$$
$$\lambda_2 = \{\}, \{right\}, \{pickup(3)\}, \{\}$$
$$\lambda_3 = \{pickup(2)\}, \{\}, \{\}, \{\}$$
$$\lambda_4 = \{\}, \{\}, \{pickup(3), right\}, \{\}$$

All four of these rather different traces satisfy H_4, highlighting that some of the learned advice formulae can be quite ambiguous when recommending actions to satisfy the specification. Note that in our setup, we assume that the agent can only perform one action at a time, so λ_4 is not realistic in this context. However, it still demonstrates another case of ambiguity if this assumption were relaxed.

Chapter 5

Experiments for learning temporal advice formulae

In this chapter, we empirically apply ILASP to learn temporal advice formulae. Since no approaches to learning temporal advice currently exist, our empirical evaluation primarily aimed to assess the effectiveness of the method we propose. We explore two environments. First, we use the simplified gem pickup environment whose simplicity allows for greater experimentation. Second, we examine RockSample, a more complex environment featuring a large action space and requiring long action sequences to complete the task. The outline of the sections in this chapter are as follows:

- Section 5.1 introduces the research questions with which we aim to understand the effect of various learning setup design decisions on learning temporal advice using ILASP.
- Section 5.2 presents the experimental setup including experiment design, environment descriptions, dataset preparation, the ILASP version used, methodology for handling multiple results, and the procedures for hypothesis evaluation.
- Section 5.3 presents the experimental results and discusses the outcomes.

We hope that our findings provide practical guidelines to adapt our approach to novel environments. For instance, our results may motivate practitioners to critically examine their datasets and prioritize acquiring missing agent behavior examples to learn more effective advice formulae. Additionally, our work offers a practical motivation for theoreticians exploring sample complexity and informativeness of examples for learning temporal logic formulae.

5.1 Research questions

This section outlines the research questions we explore for learning temporal advice formulae. Our focus is on identifying the design choices that minimize runtime while maximizing the generalizability to unseen data of the learned formulae. We also examine the data requirements for successful learning, including the quality and quantity of positive and negative examples, and whether fitting every training example is essential.

Amount of training data. As with any machine learning task, data is crucial. Both the quantity and quality of the data are key factors. Acquiring data can sometimes be difficult or costly, so it is important to have a strong rationale for obtaining it. In the first experiment, we aim to investigate the significance of data quantity in learning temporal advice formulae. Specifically, we aim to answer the following two research questions:

RQ1. How does the amount of data affect the results?

RQ2. How does the imbalance between positive and negative examples impact the results?

Allowed formula size. In Section 4.2, the maximum number of nodes in the syntax tree is limited by n. Modifying this parameter greatly affects the hypothesis space size, as each additional node introduces all possible rule combinations at that node index. In addition, if the problem is unsatisfiable, it may be due to the fact that the syntax tree was too restrictive to express a complex rule which would be a solution to the learning problem. We raise the following question:

RQ3. How does the allowed size of the syntax tree affect the results?

Poorly labeled training data. In the toy example from Section 4.3, the distinction between good behavior in positive data points and bad behavior in negative data points is clear. But what if it is not? Real-world data acquisition is often noisy, leading to situations where positive and negative data points are not significantly different. Hence, we ask the question:

RQ4. How does having similar examples in both the positive and negative sets impact the results?

Absence of environment predicates. In Section 4.1.1, we emphasized the importance of environmental predicates in describing the state of an environment, which leads the agent to take certain actions. However, this was based primarily on intuition. Concretely, this leads us to the following question:

RQ5. How does the absence of environmental predicates impact the results?

Propositional vs. first-order temporal advice. In Section 4.1.2, we argued that incorporating first-order elements into our logic is necessary to obtain small and generalizable formulae. However, Ielo et al. [2023] did not take this approach, which raises the possibility that our intuition may not have been entirely accurate. Therefore, in this experiment, we aim to challenge our intuition and ask the following question:

RQ6. How does changing the first-order predicates to propositions affect the results?

Penalties for uncovered examples. Fitting every example can be challenging, and in some cases, the problem may even be unsatisfiable. Allowing some examples to be uncovered in exchange for a penalty offers a practical solution. This experiment explores learning temporal advice formulae from datasets with examples equipped with penalties. The formal problem for this setup is defined in Definition 2.3.5. This experiment investigated a future work direction identified by Ielo et al. [2023]. We ask the following questions:

RQ7. Do different penalty settings yield noticeable result differences?

RQ8. Do adaptive penalties computed based on the trace score yield better generalizability results?

Using property specifications patterns as sketches. To accelerate the learning task and shape the structure of the learned formulae, we introduce background knowledge in the form of $\mathsf{LTL_f}$ sketches. An $\mathsf{LTL_f}$ sketch is a partially defined $\mathsf{LTL_f}$ formula with missing components represented by placeholders, denoted as ? (question marks) [Roy, 2024]. The specific sketches that we investigate are based on the property specification patterns [Dwyer et al., 1999]. This is driven by the observation that the patterns identified by Dwyer et al. [1999] capture interesting temporal behaviors in systems, and we are curious to explore their applicability to planning scenarios like ours. We raise the following question:

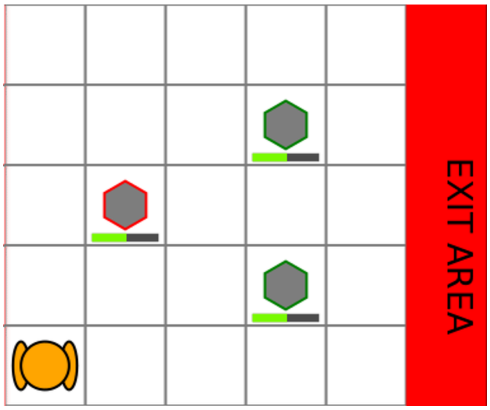


Figure 5.1: Instance of RockSample[5,3]. Green-outlined rocks are valuable, and the red-outlined rock is worthless. Bars below indicate the probability of each rock being valuable, at the start of the episode they are all 50%. Figure from https://github.com/JuliaPOMDP/RockSample.jl/tree/master.

RQ9. How does learning temporal advice formulae using a specification pattern sketch affect the results?

5.2 Experimental setup

Here, we present the details how we empirically attempted to answer the research questions outlined in the previous section. All experiments were run on a computer with 8 GB RAM and Intel Core i7-4720HQ CPU @ 2.60GHz.

5.2.1 Environments

In order to setup the learning tasks in ILASP, we follow the general template described in Section 4.2 and adjust it as needed for each environment. Unless specified, we limited the maximum amount of syntax tree nodes to 10 for gem pickup and 7 for RockSample.

Gem pickup. This environment was already introduced in Section 4.1 and its signature was introduced in Definition 4.3.1. The sets of actions and environment descriptors are the same except that we further simplified the learning task and reduced the hypothesis space by excluding the *picked* predicate as an environment description predicate[1]. The size of the hypothesis space (generated with the ILASP flag `-s`) for this environment is 615.

RockSample. Introduced in Smith and Simmons [2004], RockSample is a large partially observable environment which models rover science exploration (Figure 5.1). A rover is placed on a $n \times n$ grid with k rocks on it, we will refer to such problem instances as RockSample$[n, k]$. The rover is aware of its own position as well as the positions of the rocks. However, the value of each rock is unknown. This is the source of partial observability: the rover only possesses a probabilistic estimate of each rock's worth. Sampling a valuable rock results in a positive reward, and the rock becomes worthless. Sampling a worthless rock leads to a negative reward. Therefore, avoiding worthless rocks is essential. The rover can use a noisy long-range sensor to update its beliefs about rock values. An episode of the problem concludes when the rover exits the map through any tile in the rightmost column, which also grants a positive reward. All other moves have neither cost nor reward.

[1]The specific encoding for this environment is presented in ilasp_asp_tasks/gem_pickup.las

The $check(R)$ action probes rock R with the rover's noisy sensor, telling it whether it's valuable or not. The noise level is dictated by efficiency η, decreasing exponentially with Euclidean distance from the target. When $\eta = 1$, the sensor is accurate, when $\eta = 0$, the output has a 50/50 chance of being correct. For other η values, the output is a linear combination of these behaviours. At the start of each episode, each rock has a 50% probability of being valuable.

Definition 5.2.1 The following signature $\mathcal{S}_{rs} = \langle \Omega, C, arity \rangle$ for RockSample was used to encode the environment in ASP, where:

- $\Omega = A \cup E$, where $A = \{east, west, north, south, check, target\_sample\}$ and $E = \{guess, dist, delta\_x, delta\_y, num\_sampled\}$
- $C = rocks \cup guess\_val \cup dist\_thr \cup perc\_rocks$ where
  - $rocks = \{r_1, \ldots, r_k\}$ where k is the total amount of rocks in the environment.
  - $guess\_val = \{30, \ldots, 90\}$ which represents the probabilities of some rock being valuable.
  - $dist\_thr = \{0, 1, 2, 3, 4\}$ which represents the distance thresholds for the predicates describing distances.
  - $perc\_rocks = \{0, 25, 50, 75\}$ representing the percentages of rocks sampled.
- $arity(east) = arity(west) = arity(north) = arity(south) = 0$, $arity(check) = arity(target\_sample) = 1$, $arity(guess) = arity(dist) = arity(delta\_x) = arity(delta\_y) = 2$.

Here, A represents the agent's available actions: east, west, north, south for movement, and $check(R)$ to update the rover's belief about the value of $R \in rocks$. E is the set of environment description predicates, which are used to describe the state as the agent acts in the environment. Specifically, $check(R)$ indicates the usage of the sensor on a rock $R \in rocks$, and $target\_sample(R)$ indicates sampling a rock $R \in rocks$. The predicate $guess(R, G)$ denotes the probability $G \in guess\_val$ of a rock $R \in rocks$ being valuable. The predicate $dist(R, D)$ describes the Manhattan distance $D \in dist\_thr$ to a rock $R \in rocks$, and $delta\_x(R, D)$ as well as $delta\_y(R, D)$ describe the x and y coordinate distance $D \in dist\_thr$ to the rock $R \in rocks$.

The size of the hypothesis space[2] (generated with the ILASP flag `-s`) for this environment is 1267.

5.2.2 Preparing training and testing datasets

Each dataset was made of a split of p positive and n negative examples which are henceforth denoted as (p, n) datasets. In ILASP, we followed Ielo et al. [2023] and used the `#pos` keyword to specify only positive examples in the datasets. However, given our setup, we can specify negative examples by placing `sat` in the exclusion set for each trace. By specifying only the positive examples, we take advantage of ILASP's brave induction, which is more efficient since we only need to find one answer set, rather than ensuring that no answer set exists with `sat`. We supply the generated datasets in the code repository for reproducibility of results.

Gem pickup. We implemented the gem pickup environment and generated the datasets using a script[3] in our repository. A reward of +10 is given for picking up the gem successfully, whereas a reward of -100 is given if the pickup action is done not on a gem, all movement actions give a reward of -1.

[2]The specific encoding for this environment is presented in ilasp_asp_tasks/rocksample.las
[3]scripts/generate_gem_pickup.py

The positive examples are intended to demonstrate good behaviour, so we implemented a greedy algorithm which solved the gem pickup task optimally in the least amount of steps needed. The agent simply went to the nearest gem, picked it up, and continued to the next nearest gem until all gems were picked up. For the negative examples, we chose to use monte carlo tree search with iterations of the search tree set to 1000. We chose to use episodes generated by MCTS as negative examples because they clearly exhibit sub-optimal behaviour. The agent's tendency to get trapped in local optima is largely due to the initial positive reward from gem pickups. This encourages it to repeatedly try the same area, even if further rewards are unlikely.

The training set was generated for a 1-dimensional line of size 10 and 3 gems. The amount examples in the training sets varied due to the research question, ranging from 3 positive examples and 3 negative examples, to 100 positive examples and 100 negative examples. The test sets are comprised of 50 positive and 50 negative examples for various size and number of gem configurations to test the generalization of the learned hypotheses.

RockSample. The datasets were generated using the Partially Observable Monte Carlo Planner[4] (POMCP) [Silver and Veness, 2010]. Both positive and negative traces were created using this planner. To generate training data, we produced 100 traces with 2^{15} particles and both tree and rollout knowledge set to preferred (high planning accuracy) and 100 traces with 2^7 particles and tree and rollout knowledge set to only legal or random (low planning accuracy). We then selected the top n and bottom n traces from each set to control for randomness and attempt to get more representative good or bad traces. We created training sets for $n \in \{3, 10, 25, 50\}$. All training sets used a 12×12 grid with 4 rocks. Test sets were generated similarly, always selecting the top 50 and bottom 50 traces, but varied grid sizes $\in \{6, 12, 18\}$ and rock counts $\in \{4, 8\}$ to evaluate the generalization of the hypotheses.

5.2.3 Experiment design

Having defined our environments and data, we present the specific experiments designed to answer each research question.

RQ1. We run the learning setup with a balanced dataset of n positive and n negative examples, for $n \in \{3, 10, 25, 50\}$ for both environments. As is commonly expected in machine learning scenarios, we anticipate that more data will result in more generalizable formulae, though it will take longer to process in order to fit all the examples.

RQ2. We evaluate the learning task by varying the ratio of positive to negative data for both environments. We expect that more positive data will lead to more generalizable formulae, and having fewer negative examples will not significantly impact the quality.

RQ3. Using the (3, 3) gem pickup dataset we experiment with varying the size of the allowed syntax tree by restricting it to n nodes where $n \in \{3, 5, 7, 10, 20\}$ to investigate how this affects the runtime and the quality of the learned formulae. For gem pickup, this corresponds to the hypothesis space sizes of 174, 295, 420, 615, 1330, respectively.

We expect that, due to increase in the hypothesis space, the runtime will be increased. However, we do not expect a significant impact on generalization results, as ILASP prioritizes the shortest solutions. Allowing longer solutions should not change the final returned rules for the same dataset, except in cases where the problem is otherwise unsatisfiable.

RQ4. We attempt to learn formulae using datasets where some negative examples are included in the positive example set. For the (3,3) gem pickup dataset, we introduced one, two, and three negative examples into the positive example set, respectively.

[4]Specifically, we used this script https://gitlab.com/dan11694/ilasp_pomdp/-/blob/master/pomcp/scripts/run_tests.sh

We conjecture that if the same or highly similar behaviours appear in both positive and negative sets, the system may struggle to differentiate between good and bad behavior, leading to inconsistent or less meaningful learned temporal advice. In addition, since trying to find formulae which differentiate the positive and negative examples is more difficult, we expect an increase in learning runtime.

RQ5. For each environment, we remove the environmental predicates from the dataset, effectively modifying the set of predicates in the MDP's signature to the actions only, $\Omega = A$.

We hypothesize that we can still learn generalizable formulae, although they may be less actionable without environmental predicates. However, this could be achieved more quickly due to the reduced size of the hypothesis space.

RQ6. For each environment, we convert each first-order predicates in the traces into propositions. For example, $pickup(3)$ becomes $pickup\_3$, and $dist(0, -2)$ becomes $dist\_0\_\_2$, and so on. For this experiment, we use the original specification of the learning task from Ielo et al. [2023]. At the end of each dataset, we add `#constant(p).` and `proposition(p).` for every proposition encountered in the trace.

We believe the results will align with our intuition, as we do not see how propositions could generalize to unseen data. Additionally, this approach causes an explosion in the size of the hypothesis space because instead of variables, we have specific values, so we also predict that the runtime will not be reduced.

RQ7 and RQ8. Since there is no predefined best approach, we explored several methods for setting penalties. First, we applied constant penalties to all examples, testing values of $pen \in \{2, 5, 10, 20, 50\}$. Next, we set penalties based on the length of each trace. Finally, we experimented with a piecewise linear function to assign penalties according to the returns of the trace:

$$pen(return) = \begin{cases} low & \text{if } return < 1Q \\ mid & \text{if } 1Q \leqslant return \leqslant 3Q \\ high & \text{if } return > 3Q \end{cases}$$

where 1Q is the first quartile of all returns in that dataset, and 3Q is the third quartile of all returns in that dataset. Specifically, we investigated $(5, 2, 5), (7, 4, 7), (15, 5, 15)$ where the first element refers to *low*, the second element refers to *mid*, and the third to *high*. This experiment was conducted using (3,3) datasets from both environments and (10,10) dataset only for gem pickup.

We anticipate that in some setups, penalties will significantly improve fitting time by allowing problematic examples to be sacrificed at a cost. However, the application of this method may lead to increased complexity, transforming the task into an optimization problem that requires the careful balancing of example fitting, formula length, and incurred penalties.

We expect the approach assigning penalties based on return of traces to yield better results. High-return positive examples likely highlight crucial patterns that should be reflected in the learned formulae, while low-return examples may show undesirable behaviors that are equally important to account for. Examples with intermediate returns, however, may not be as critical to fit.

RQ9. This experiment was conducted using the (10,10) gem pickup dataset and the (3,3) RockSample dataset. Specifically, we investigate the following patterns from Dwyer et al. [1999], where ILASP has to fill in the indexed ? parts in each formula:

1. $\mathsf{G}(?_1 \rightarrow \mathsf{G}(\neg ?_2))$ which we call absence after event. Once $?_1$ happens, then always $?_2$ does not happen.

2. $\mathsf{F}?_1 \rightarrow (\neg?_2\mathsf{U}?_3)$. which we call absence before event. If eventually $?_1$ happens, then $?_2$ does not happen until $?_3$ happens.
3. $\mathsf{G}\neg?$ which we call absence global. It is always the case that ? does not happen.
4. $\mathsf{G}(?_1 \rightarrow \mathsf{GF}?_2)$ which we call recurrence after event. It is always the case that if $?_1$ happens, it triggers that $?_2$ keeps forever repeating.
5. $\mathsf{GF}?$ which we call recurrence global. It is always the case that ? keeps eventually happening.
6. $\mathsf{G}(?_1 \rightarrow \mathsf{F}?_2)$ which we call response global. It is always the case that once $?_1$ happens, eventually $?_2$ will happen.
7. $\mathsf{F}?_1 \rightarrow (?_2\mathsf{U}?_3)$ which we call universality before event. If eventually $?_1$ happens, then $?_2$ happens until $?_3$ happens.
8. $\mathsf{G}?$ which we call universality global. Always ? happens.
9. $?_1\mathsf{U}?_2$ which we call until. $?_1$ happens at least until $?_2$ happens.

We anticipate that the more placeholders there are, the longer it would take to arrive at a result, but we also anticipate that the generalizability of the formula will largely depend on the sketch itself. That is, we do not expect much variation in terms of accuracy or F1 scores between the multiple retrieved solutions for the same pattern.

5.2.4 ILASP version used and retrieving multiple hypotheses

Due to its significantly faster runtime, we selected ILASP2i for running our experiments. Although we experimented with using ILASP4 (following Meli et al. [2024]) and ILASP3, their runtime was much too long due to the way these algorithms deal with counterexamples. However, our use of ILASP2i aligns with Ielo et al. [2023].

Because we are learning advice, which are intended to guide planners, we are primarily concerned with their semantic meaning and practical applicability. We are less concerned with the formula length or whether they represent the absolute optimal solution. Furthermore, Instead of using ILASP's standard `-version=2i` flag, we used a custom PyLASP script[5] for the 2i version. This script replicates ILASP2i's solution cycles but additionally outputs the top 10 best (in terms of score) solutions to a file. The generation of these solutions involved the following process: upon finding an optimal solution using ILASP2i's method, that solution was added as a hard constraint to the underlying ASP program, preventing it from being returned again, and the solving stage was invoked again.

Because we are interested in syntax trees which are comprised of facts and normal rules, we ran ILASP with the `-nc` flag which does not consider hard constraints in the search space, and thus makes the learning process slightly faster. We also specified `-max-rule-length=6` and `-ml=5` which allowed maximum of 5 literals in the bodies of the rules in the hypothesis space.

Given our focus on semantic meaning, why not select solutions based on a metric other than length? The choice of shortest solutions was largely driven by their straightforward accessibility within the default ILASP framework. We acknowledge the potential value of alternative selection metrics and defer their theoretical and empirical investigation to future work.

5.2.5 Evaluation metrics

How can we assess the quality of a learned symbolic formula? Judging the results qualitatively, based on the formula's perceived applicability within the environment, is one approach. However,

[5] pylasp2i_multiple.py

this method is flawed. While we present some initial results using qualitative judgement, we place greater emphasis on a quantitative evaluation, which we describe below.

All of our research questions were framed in terms of total runtime and generalization capabilities. Total runtime was calculated per experimental configuration, with a 10 hour timeout. The generalization capability was evaluated as follows. Let Λ be a test set of example traces. The learned hypothesis H for each experimental configuration by checking, for each example trace $\lambda \in \Lambda$, if the program $H \cup B \cup \lambda$ yielded `sat` (for positive examples) or `unsat` (for negative examples) in its answer set. We thus computed a classification array which was then compared to the true positive/negative values to calculate accuracy and F1 scores. These scores indicate the generalization performance of learned theories on unseen data and different environment configurations.

Each environment included multiple test sets, varying in size and number of objects, but each test set had 50 positive and 50 negative traces. For each experimental configuration, we evaluated the top 10 hypotheses generated by ILASP, as described in Section 5.2.4. Execution time is measured for the complete run that produces all 10 hypotheses. Accuracy and F1 scores are reported as the mean and standard deviation across these top 10 solutions.

5.3 Results and discussion

This section presents the result tables from our experiments and our interpretation of them. Our code repository contains the solutions that generated these outputs, and the `README.md` file provides instructions for reproducing the results.

In all tables, "TO" indicates a 10-hour timeout. The goal is to minimize execution time and maximize accuracy and F1 scores. Bold values indicate the best results per column. For values with standard deviation, the best result is determined using the values of standard deviation subtracted from the mean.

In the tables with gem pickup results, L indicates the length of the line one which the agent navigates, and G denotes the amount of gems. In the tables with RockSample results, N indicates the size of the grid, and R denotes the amount of rocks.

5.3.1 Amount of training data

Table 5.1: Results of learning the 10 best advice formulae in the gem pickup environment with varying dataset sizes. The train setup indicates the number of positive and negative examples.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3)	**48**	0.88 ±.04	0.90 ±.03	0.63 ±.04	0.73 ±.02	**0.93 ±.02**	**0.93 ±.02**	0.95 ±.04	0.95 ±.03	0.97 ±.05	0.97 ±.05
(10,10)	496	**0.93 ±.00**	**0.93 ±.00**	**0.66 ±.00**	**0.75 ±.00**	0.93 ±.03	**0.93 ±.02**	**0.98 ±.01**	**0.98 ±.01**	**1.00 ±.01**	**1.00 ±.01**
(25,25)	TO	-	-	-	-	-	-	-	-	-	-
(50,50)	TO	-	-	-	-	-	-	-	-	-	-

Gem pickup. Table 5.1 presents the results of learning temporal advice formulae across different dataset sizes. The execution time was influenced by dataset size, with ILASP timing out after 10 hours for both the (25,25) and (50,50) datasets. This is surprising, given that these datasets remain extremely small compared to modern machine learning setups, where dataset sizes are orders of magnitude larger. As expected, the larger dataset yielded better generalization results

Table 5.2: Results of learning the 10 best advice formulae in the gem pickup environment with imbalanced datasets. The train setup indicates the number of positive and negative examples.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(1,3)	16	0.58 ±.04	0.48 ±.01	0.57 ±.04	0.45 ±.03	0.53 ±.03	0.41 ±.10	0.52 ±.07	0.38 ±.04	0.59 ±.03	0.47 ±.07
(3,1)	14	0.48 ±.06	0.52 ±.16	0.47 ±.04	0.49 ±.14	0.49 ±.04	0.54 ±.13	0.55 ±.02	0.60 ±.09	0.51 ±.10	0.56 ±.15
(3,10)	TO	-	-	-	-	-	-	-	-	-	-
(10,0)	**12**	0.47 ±.07	0.63 ±.07	0.47 ±.05	0.64 ±.06	0.48 ±.03	0.65 ±.03	0.47 ±.06	0.64 ±.06	0.48 ±.05	0.64 ±.05
(10,3)	69	**0.87 ±.03**	**0.89 ±.02**	**0.62 ±.05**	**0.73 ±.02**	**0.92 ±.03**	**0.92 ±.02**	**0.94 ±.04**	**0.94 ±.03**	**0.97 ±.05**	**0.97 ±.04**

than the smaller one. However, the difference is not substantial, and it is particularly encouraging that the tiny (3,3) dataset already demonstrates strong generalization even outperforming the larger (10,10) dataset on the L=10, G=5 test set.

Notably, the solutions for both datasets did not generalize as well to the smaller environment. This is likely because the reduced environment size makes it easier for MCTS to find an optimal solution. As a result, more examples in this environment exhibit optimal behavior but are labeled as negative (because they were computed with MCTS), meaning the actual results should be viewed more optimistically. In summary, regarding **RQ1 gem pickup**: larger datasets require more time but can lead to more generalizable solutions that better balance accuracy and F1 score.

The next experiment explores the impact of imbalance between positive and negative examples in the training set. Table 5.2 presents the results. Generally, when easy solutions exist, fewer data points in the training set result in shorter execution times. However, an excess of negative examples over positive ones can cause ILASP to time out, and the solutions retrieved from such small, imbalanced datasets tend to be of poor quality. In summary, **RQ2 gem pickup**: having more positive examples is crucial, but a certain number of negative examples is still necessary for achieving good results.

Table 5.3: Results of learning the 10 best advice formulae in the RockSample with varying dataset sizes. The train setup indicates the number of positive and negative examples.

Train	**Run**	N=12,R=4		N=6,R=4		N=12,R=8		N=18,R=4		N=18,R=8	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3)	**145**	0.62 ±.05	0.69 ±.02	0.67 ±.06	0.70 ±.01	0.65 ±.13	0.71 ±.08	0.54 ±.09	0.63 ±.08	0.53 ±.04	0.65 ±.02
(10,10)	TO	-	-	-	-	-	-	-	-	-	-
(25,25)	TO	-	-	-	-	-	-	-	-	-	-
(50,50)	TO	-	-	-	-	-	-	-	-	-	-

RockSample. Table 5.3 presents the results for RockSample across different dataset sizes. Unfortunately, only the (3,3) dataset finished without timing out. Given that the hypothesis

sizes for gem pickup (615) and RockSample (1267) are not drastically different, these timeouts were unexpected. Additionally, the hypotheses retrieved for the (3,3) dataset do not generalize as well to unseen data compared to those from gem pickup. Unlike gem pickup, RockSample does not show a drop in performance for the smaller dataset, further supporting the idea that the difference between positive and negative examples in gem pickup is not as clear in the smaller scenario. In summary, **RQ1 RockSample**: larger datasets lead to timeouts, suggesting that the learning task for RockSample is significantly more challenging, while the impact of dataset size on generalization scores remains unclear.

Table 5.4: Results of learning the 10 best advice formulae in the RockSample with imbalanced datasets. The train setup indicates the number of positive and negative examples.

Train	**Run**	N=12,R=4		N=6,R=4		N=12,R=8		N=18,R=4		N=18,R=8	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(1,3)	87	**0.62 ±.04**	**0.69 ±.02**	**0.66 ±.05**	**0.70 ±.01**	**0.62 ±.12**	0.70 ±.07	0.55 ±.08	0.64 ±.07	**0.52 ±.03**	0.64 ±.02
(3,1)	**81**	0.62 ±.13	0.70 ±.08	0.69 ±.17	0.75 ±.11	0.56 ±.07	0.59 ±.11	0.54 ±.05	0.62 ±.08	0.51 ±.04	0.55 ±.14
(3,10)	TO	-	-	-	-	-	-	-	-	-	-
(10,0)	84	0.63 ±.14	0.74 ±.08	0.68 ±.20	0.78 ±.12	0.64 ±.16	**0.74 ±.09**	**0.56 ±.05**	**0.70 ±.03**	0.59 ±.11	**0.71 ±.06**
(10,3)	261	0.54 ±.01	0.66 ±.00	0.58 ±.06	0.70 ±.03	0.49 ±.04	0.65 ±.02	**0.52 ±.01**	0.66 ±.01	**0.50 ±.01**	**0.66 ±.01**

Table 5.4 presents the results of training on imbalanced data. Similar to the gem pickup scenario, ILASP times out for the (3,10) dataset. However, unlike gem pickup, increasing the dataset size does not necessarily improve generalization.

In summary, **RQ2 RockSample**: when positive and negative examples exhibit similar behavior, dataset imbalance has a minimal impact on performance.

5.3.2 Allowed formula size

Gem pickup. Table 5.5 presents the results for varying the limits on the number of allowed nodes in the syntax tree. We see that the execution time increases almost exponentially given the size of the hypothesis space. Interestingly, the smallest hypothesis space produced slightly better results overall. This may be due to the way ILASP explores solutions, which is somewhat random and influenced by the hypothesis space.

In summary, **RQ3 gem pickup**: as expected, runtime increases with a larger hypothesis space, while the differences in generalization results remain relatively small. This also helps explain why the RockSample scenario times out, as its hypothesis space is twice the size of gem pickup's.

5.3.3 Poorly labeled training data

Gem pickup. Table 5.6 presents the results for a (3,3) dataset. We also tested scenarios (10+1,9) and (10+2,8), that is, 10 original positives plus 1 or 2 negative examples in the positive set, respectively, but both resulted in timeouts.

In summary, **RQ4 gem pickup**: as expected, combining good and bad behavior makes it more challenging to derive generalizable solutions that clearly distinguish between positive and

Table 5.5: Results of learning the 10 best advice formulae for gem pickup varying allowed syntax tree sizes. The train setup indicates the maximum amount of nodes in the syntax tree.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
3	**5**	**0.89 ±.02**	**0.90 ±.02**	0.63 ±.04	0.73 ±.02	0.92 ±.03	**0.93 ±.02**	**0.96 ±.03**	**0.96 ±.02**	**0.99 ±.01**	**0.99 ±.01**
5	12	0.87 ±.13	0.87 ±.13	**0.66 ±.0**	0.74 ±.01	0.89 ±.10	0.91 ±.08	0.88 ±.22	0.87 ±.23	0.93 ±.16	0.93 ±.15
7	26	0.87 ±.05	0.89 ±.03	0.66 ±.01	**0.75 ±.01**	0.93 ±.04	0.93 ±.03	0.95 ±.04	0.96 ±.03	0.98 ±.04	0.98 ±.03
10	48	0.88 ±.04	0.90 ±.03	0.63 ±.04	0.73 ±.02	**0.93 ±.02**	**0.93 ±.02**	0.95 ±.04	0.95 ±.03	0.97 ±.05	0.97 ±.05
20	552	0.84 ±.12	0.85 ±.12	0.66 ±.01	0.74 ±.01	0.89 ±.1	0.91 ±.08	0.86 ±.21	0.86 ±.22	0.91 ±.15	0.91 ±.15

Table 5.6: Results of learning the 10 best advice formulae for gem pickup with poorly labeled data. The train setup indicates the number of positive + negative and negative examples.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3)	48	**0.88 ±.04**	**0.90 ±.03**	**0.63 ±.04**	**0.73 ±.02**	**0.93 ±.02**	**0.93 ±.02**	**0.95 ±.04**	**0.95 ±.03**	**0.97 ±.05**	**0.97 ±.05**
(3+1,2)	63	0.78 ±.12	0.80 ±.12	0.61 ±.04	0.70 ±.04	0.83 ±.09	0.86 ±.07	0.80 ±.20	0.79 ±.23	0.89 ±.14	0.89 ±.15
(3+2,1)	37	0.54 ±.02	0.68 ±.01	0.53 ±.01	0.66 ±.04	0.51 ±.00	0.67 ±.00	0.55 ±.03	0.69 ±.01	0.59 ±.04	0.71 ±.02
(3+3,0)	**12**	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00

negative unseen examples. The runtime may also be affected significantly, as exemplified by the fact that (10+1,9) and (10+2,8) scenarios timed out.

Appendix C provides a detailed analysis of the similarity between positive and negative traces as well as an analysis of the distribution of their return scores. These findings, along with those from RQ4, suggest that the positive and negative data in RockSample are not sufficiently different. While positive examples for RockSample are generated by MCTS with a much higher number of iterations than negative ones, the resulting action sequences appear too similar. This makes it difficult for ILASP to identify fitting formulae or ones that generalize well beyond the training data.

5.3.4 Absence of environment predicates

Gem pickup. Results are shown in Table 5.7. The results are in contrast to what we expected, even leading the (10,10) no environmental predicates scenario to timeout. This suggests that environmental predicates play a crucial role in describing example behavior.

In summary, **RQ5 gem pickup**: the absence of environmental predicates increases execution time and negatively impacts generalization results.

Table 5.7: Results of learning the 10 best advice formulae in the gem pickup environment with no environmental predicates. The train setup specifies the number of positive and negative examples, "Full" included environmental predicates, while "NEN" excluded them.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3) Full	**48**	**0.88 ±.04**	**0.90 ±.03**	**0.63 ±.04**	**0.73 ±.02**	**0.93 ±.02**	**0.93 ±.02**	**0.95 ±.04**	**0.95 ±.03**	**0.97 ±.05**	**0.97 ±.05**
(3,3) NEN	123	0.63 ±.14	0.63 ±.14	0.63 ±.13	0.66 ±.09	0.60 ±.12	0.61 ±.12	0.61 ±.19	0.62 ±.21	0.64 ±.14	0.63 ±.16
(10,3) Full	**69**	**0.87 ±.03**	**0.89 ±.02**	0.62 ±.05	**0.73 ±.02**	**0.92 ±.03**	**0.92 ±.02**	**0.94 ±.04**	**0.94 ±.03**	**0.97 ±.05**	**0.97 ±.04**
(10,3) NEN	127	0.84 ±.17	0.87 ±.14	**0.73 ±.11**	0.78 ±.08	0.88 ±.17	0.90 ±.13	0.87 ±.18	0.89 ±.14	0.89 ±.17	0.92 ±.13
(10,10) Full	**496**	0.93 ±.00	0.93 ±.00	0.66 ±.00	0.75 ±.00	0.93 ±.03	0.93 ±.02	0.98 ±.01	0.98 ±.01	1.00 ±.01	1.00 ±.01
(10,10) NEN	TO	-	-	-	-	-	-	-	-	-	-

RockSample. The results are shown in Table 5.8. In this case, removing the environmental predicates did not result in any significant differences. The slight variations in generalizability and execution times are likely due to the randomness in the ILASP solving stage which is caused by the altered hypothesis space. The lack of environmental predicates did not prevent timeouts in the (10,10) dataset case. In summary, **RQ5 RockSample**: for a dataset with mixed good and bad behavior, the absence of environmental predicates has no noticeable effect.

5.3.5 Propositional vs. first-order temporal advice

Gem pickup. The results are shown in Table 5.9. In summary, **RQ6 gem pickup**: as expected, the learned solutions do not generalize well, but they do slightly reduce execution time. This confirms that our choice to use general first-order predicates was reasonable.

RockSample. The results are shown in Table 5.10. Contrary to our expectation, the execution time is reduced, especially for the (10,10) case which changed from a 10-hour time out to just 15 minutes. In summary, **RQ6 RockSample**: learning $\mathsf{LTL_f}$ instead of temporal advice formulae does not significantly affect generalizability in scenarios where the behavior is not clearly defined by the examples. However, it may simplify the learning problem and reduce execution time.

5.3.6 Penalties for uncovered examples

Gem pickup. The results for (3,3) are shown in Table 5.11, while the results for (10,10) are presented in the Appendix, Table 6.2. For the (10,10) dataset, the formulae obtained under different penalty configurations show no significant difference from the scenario without penalties. However, in the (3,3) dataset, lower penalty values can worsen the results, as many examples are sacrificed, leading to non-generalizable solutions. This effect is evident in scenarios with constant penalties of 2 and 5.

In summary, **RQ7 gem pickup**: when solutions are found without penalties, introducing large penalties has little impact on performance. However, applying low penalties can significantly

Table 5.8: Results of learning the 10 best advice formulae for the RockSample environment with no environmental predicates. The train setup indicates the number of positive and negative examples and "Full" denotes the setup with environmental predicates, and "NEN" denotes the setup without.

Train	**Run**	N=12,R=4		N=6,R=4		N=12,R=8		N=18,R=4		N=18,R=8	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3) Full	86	**0.62 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	0.65 ±.13	0.71 ±.08	0.54 ±.09	0.63 ±.08	**0.53 ±.04**	**0.65 ±.02**
(3,3) NEN	**83**	0.60 ±.04	**0.69 ±.02**	**0.65 ±.04**	**0.71 ±.02**	**0.56 ±.03**	**0.66 ±.01**	**0.62 ±.04**	**0.70 ±.02**	0.51 ±.03	**0.64 ±.01**
(10,3) Full	134	0.54 ±.01	0.66 ±.0	0.58 ±.06	0.70 ±.03	0.49 ±.04	0.65 ±.02	0.52 ±.01	0.66 ±.01	0.50 ±.01	0.66 ±.01
(10,3) NEN	**93**	0.54 ±.01	0.66 ±.0	0.58 ±.06	0.70 ±.03	0.49 ±.04	0.65 ±.02	0.52 ±.01	0.66 ±.01	0.50 ±.01	0.66 ±.01
(10,10) Full	TO	-	-	-	-	-	-	-	-	-	-
(10,10) NEN	TO	-	-	-	-	-	-	-	-	-	-

reduce the generalizability of the solutions. **RQ8 gem pickup**: contrary to our expectations, the dynamic penalty based on trace returns does not lead to better generalization than trace length penalties.

RockSample. The results for the (3,3) dataset are shown in Table 5.12. In summary, **RQ7 RockSample**: since ILASP finds solutions for scenarios without penalties, adding penalties does not lead to a significant improvement. **RQ8 RockSample**: the dynamic penalty based on trace returns does not lead to a significant difference in results compared to other approaches.

These findings suggest that using penalties for small datasets such as this one is not worthwhile, especially when solutions can already be found without them.

5.3.7 Using property specifications patterns as sketches

Gem pickup. The results are shown in Table 5.13. Contrary to our expectations, more placeholders do not necessarily result in significantly longer execution times. In fact, $\mathsf{G}\neg?$ (train setup 3) had the longest runtime despite having only a single placeholder. Care must be taken when selecting the specification pattern, as it can lead to poor performance, an unsatisfiable learning problem, or a timeout. In summary, **RQ9 gem pickup**: as anticipated, the choice of the specification pattern plays a crucial role in determining generalization results and, more broadly, the feasibility of the learning problem. Nevertheless, the chosen specification pattern may increase the generalization scores and sometimes even reduce the runtime.

RockSample. The results for the (3,3) dataset are presented in Table 5.14, showing only the results for the setups which had solutions. We also tested the (10,10) dataset, but all results were unsatisfiable. Similar to the gem pickup case, we observe that adding more placeholders does not lead to an increase in runtime. In summary, **RQ9 RockSample**: similarly as for gem pickup, the choice of specification pattern is critical for both generalization and the overall feasibility of

Table 5.9: Results of learning the 10 best $\mathsf{LTL_f}$ formulae in the gem pickup environment. The train setup indicates the number of positive and negative examples. Bold values per dataset size.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3) FO	48	**0.88 ±.04**	**0.90 ±.03**	**0.63 ±.04**	**0.73 ±.02**	**0.93 ±.02**	**0.93 ±.02**	**0.95 ±.04**	**0.95 ±.03**	**0.97 ±.05**	**0.97 ±.05**
(3,3) prop	**21**	0.52 ±.05	0.29 ±.13	0.51 ±.02	0.14 ±.22	0.53 ±.03	0.31 ±.11	0.53 ±.02	0.17 ±.10	0.53 ±.05	0.24 ±.12
(10,10) FO	496	**0.93 ±.00**	**0.93 ±.00**	**0.66 ±.00**	**0.75 ±.00**	**0.93 ±.03**	**0.93 ±.02**	**0.98 ±.01**	**0.98 ±.01**	**1.00 ±.01**	**1.00 ±.01**
(10,10) prop	**472**	0.80 ±.19	0.72 ±.38	0.63 ±.07	0.59 ±.31	0.75 ±.17	0.69 ±.36	0.87 ±.20	0.77 ±.41	0.85 ±.20	0.76 ±.40

Table 5.10: Results of learning the 10 best $\mathsf{LTL_f}$ formulae for the RockSample environment. The train setup indicates the number of positive and negative examples. Bold values per dataset size.

Train	**Run**	N=12,R=4		N=6,R=4		N=12,R=8		N=18,R=4		N=18,R=8	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
(3,3) FO	145	**0.62 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	0.65 ±.13	**0.71 ±.08**	0.54 ±.09	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
(3,3) prop	**62**	**0.64 ±.07**	0.68 ±.08	0.67 ±.11	0.67 ±.24	**0.64 ±.08**	0.68 ±.07	**0.67 ±.11**	**0.72 ±.09**	**0.61 ±.06**	0.64 ±.07
(10,10) FO	TO	-	-	-	-	-	-	-	-	-	-
(10,10) prop	**876**	0.79 ±.10	0.83 ±.08	0.72 ±.14	0.78 ±.10	0.78 ±.11	0.79 ±.11	0.77 ±.08	0.81 ±.06	0.76 ±.11	0.76 ±.11

the learning problem. However, the right pattern can also enhance generalization scores and, in some cases, reduce runtime.

5.4 Key takeaways

The previous sections presented experimental results and their interpretation addressing nine research questions. We now present specific examples of the learned temporal advice formulae for the best and worst configurations in both gem pickup and RockSample environments[6]. For the following formulae, we omit the $\exists$ quantifier and simplify the expressions, as at this point, the domains of variables are clear from context.

Recall that we reported the average accuracies and standard deviations of the top 10 solutions in the tables of Section 5.3. To compute a worst-case average accuracy score, we subtracted the standard deviation of accuracy from the mean accuracy, for each test set, and computed the average of these values. We report this value for the best and worst configurations below.

First, we list a few examples of learned formulae for the best generalizing configurations for the gem pickup environment.

[6]To access all learned formulae for each experimental configuration, please refer to our code repository.

Table 5.11: Results of learning the 10 best advice formulae in the gem pickup environment with different penalties for a (3,3) dataset. A single number in the train setup column means a constant penalty, Tr. len. means the trace length of the example, and three numbers indicate the different *low*, *mid*, and *high* penalties.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
None	48	0.88 ±.04	**0.90 ±.03**	0.63 ±.04	0.73 ±.02	**0.93 ±.02**	**0.93 ±.02**	0.95 ±.04	0.95 ±.03	0.97 ±.05	0.97 ±.05
2	**22**	0.50 ±.04	0.27 ±.24	0.51 ±.03	0.24 ±.23	0.48 ±.03	0.23 ±.23	0.52 ±.09	0.30 ±.29	0.54 ±.04	0.31 ±.27
5	151	0.64 ±.02	0.64 ±.02	0.63 ±.03	0.68 ±.05	0.69 ±.06	0.72 ±.07	0.60 ±.03	0.55 ±.07	0.72 ±.04	0.71 ±.02
10	122	0.85 ±.12	0.86 ±.13	0.66 ±.02	0.73 ±.05	0.88 ±.14	0.88 ±.13	0.91 ±.14	0.91 ±.14	0.93 ±.13	0.94 ±.13
20	68	0.86 ±.09	0.87 ±.09	0.62 ±.04	0.73 ±.02	0.90 ±.07	0.91 ±.06	0.90 ±.15	0.90 ±.16	0.93 ±.12	0.94 ±.11
50	81	**0.88 ±.02**	**0.89 ±.02**	0.63 ±.05	0.73 ±.02	0.92 ±.02	0.92 ±.02	**0.95 ±.02**	**0.95 ±.02**	**0.99 ±.01**	**0.99 ±.01**
Tr. len.	159	0.65 ±.03	0.65 ±.03	0.64 ±.03	0.68 ±.05	0.69 ±.06	0.72 ±.07	0.60 ±.03	0.56 ±.07	0.72 ±.04	0.71 ±.02
(5,2,5)	89	0.69 ±.19	0.63 ±.26	0.62 ±.07	0.60 ±.19	0.71 ±.19	0.64 ±.31	0.72 ±.21	0.68 ±.24	0.74 ±.20	0.69 ±.26
(7,4,7)	132	0.87 ±.05	0.89 ±.03	**0.67 ±.00**	**0.75 ±.00**	0.93 ±.04	0.93 ±.03	0.96 ±.04	**0.96 ±.03**	0.98 ±.04	0.98 ±.03
(15,5,15)	118	0.79 ±.15	0.80 ±.14	0.65 ±.01	0.73 ±.01	0.84 ±.11	0.86 ±.09	0.78 ±.23	0.76 ±.26	0.86 ±.16	0.86 ±.16

Gem pickup, the (10,10) dataset (avg. worst-case accuracy = 0.89).

$$\mathsf{G}\neg(\mathit{pickup}(G) \wedge \mathit{dist}(G, D) \wedge D \leqslant -1) \tag{5.1}$$

In words: "A gem is never picked up when its distance is less than or equal to -1." Since the agent should only pick up a gem when it is exactly on it, this is a reasonable and expected piece of advice for the gem pickup environment.

$$\mathsf{G}((\mathit{pickup}(G) \wedge \mathit{dist}(G, D) \wedge D \leqslant -1) \rightarrow \mathit{left}) \tag{5.2}$$

This formula generalizes well but lacks informativeness. The antecedent is never satisfied in positive examples (as established by the previous formula), but it may be in negative examples due to suboptimal moves by MCTS, which can sometimes attempt a *pickup* action without being on a gem. However, since exactly one action is performed per time step in our setups, the consequent is never true in such cases. As a result, this formula holds for many positive examples but rarely for negative ones.

Gem pickup, the (10,10) dataset with specification pattern $\mathsf{G}\,[\mathsf{F}?_1 \rightarrow \mathsf{G}?_2]$ (avg. worst-case accuracy = 0.95).

$$\mathsf{G}\,[\mathsf{F}\psi \rightarrow \mathsf{G}\psi] \tag{5.3}$$

$$\text{where } \psi = \neg\,(\mathit{left}\mathsf{U}(\mathit{right}\mathsf{U}((\mathit{pickup}(G) \wedge \mathit{dist}(G, D) \wedge D \geqslant 0)))) \tag{5.4}$$

Table 5.12: Results of learning the 10 best advice formulae for the RockSample environment with different penalties for a (3,3) dataset. A single number in the train setup column means a constant penalty, Tr. len. means the trace length of the example, and three numbers indicate the different *low*, *mid*, and *high* penalties.

Train	**Run**	N=12,R=4		N=6,R=4		N=12,R=8		N=18,R=4		N=18,R=8	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
None	**86**	0.62 ±.05	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	0.71 ±.08	0.54 ±.09	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
2	125	0.60 ±.05	0.68 ±.02	**0.66 ±.05**	0.70 ±.02	0.61 ±.10	**0.70 ±.06**	0.56 ±.07	0.65 ±.06	0.52 ±.03	**0.65 ±.02**
5	110	**0.63 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	**0.72 ±.08**	0.53 ±.08	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
10	102	**0.63 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	**0.72 ±.08**	0.54 ±.08	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
20	109	**0.63 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	**0.72 ±.08**	0.54 ±.08	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
50	106	**0.63 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	**0.72 ±.08**	0.54 ±.08	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
Tr. len.	108	**0.63 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	**0.72 ±.08**	0.53 ±.08	0.63 ±.08	0.53 ±.04	**0.65 ±.02**
(5,2,5)	109	0.59 ±.04	0.68 ±.02	0.64 ±.04	0.70 ±.02	0.58 ±.08	0.68 ±.05	**0.57 ±.06**	**0.67 ±.05**	0.52 ±.03	**0.65 ±.02**
(7,4,7)	111	0.61 ±.05	**0.69 ±.02**	**0.66 ±.05**	**0.70 ±.01**	0.62 ±.12	0.70 ±.07	0.55 ±.08	0.64 ±.07	0.52 ±.03	0.64 ±.02
(15,5,15)	103	**0.63 ±.05**	**0.69 ±.02**	**0.67 ±.06**	**0.70 ±.01**	**0.65 ±.13**	**0.72 ±.08**	0.53 ±.08	0.63 ±.08	0.53 ±.04	**0.65 ±.02**

This formula effectively fits the positive examples while distinguishing them from the negative ones, as long as ψ does not hold throughout all negative traces. ψ in words: "It is not the case that $left$ remains true until $right$ takes over and continues to be true until a gem is eventually picked up at a nonnegative distance."

This also highlights a potential fault in supplying specification patterns - we did not enforce that the syntax tree cannot refer to its different branch. As a result, the syntax tree learned an edge that connected both the F and G subformulae to the same node, leading to the same formula.

Gem pickup, the (10,10) dataset with specification pattern G¬? (avg. worst-case accuracy = 0.99).

$$\mathsf{G}\neg\left[(left \wedge \mathsf{X}left) \vee (pickup(G) \wedge dist(G, D) \wedge D \leqslant -1)\right] \tag{5.5}$$

In words: "It is always the case that two consecutive $left$ actions are not performed, nor is a pickup executed on a gem at a negative distance." This provides actionable guidance that can be consistently followed during execution.

$$\mathsf{G}\neg\neg(right\mathsf{U}(left\mathsf{U}(pickup(G) \wedge dist(G, D) \wedge D \geqslant 0))) \tag{5.6}$$

Simplified in words: "Always, either $right$ remains true until a gem is picked up at a nonnegative distance, or $left$ is true until a gem is picked up at a nonnegative distance." This aligns with the

Table 5.13: Results of learning the 10 best advice formulae in the gem pickup environment with different specification patterns for a (10,10) dataset. Each number in the train setup corresponds to a specification in Section 5.3.7. Rows without results and TO are unsatisfiable.

Train	**Run**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
None	496	0.93 ±.0	0.93 ±.0	0.66 ±.0	0.75 ±.0	0.93 ±.03	0.93 ±.02	0.98 ±.01	0.98 ±.01	1.00 ±.01	1.00 ±.01
1	**129**	**0.98 ±.00**	**0.98 ±.00**	0.82 ±.02	**0.85 ±.01**	**0.99 ±.00**	**0.99 ±.00**	**0.99 ±.00**	**0.99 ±.00**	**1.00 ±.00**	**1.00 ±.00**
2	7	-	-	-	-	-	-	-	-	-	-
3	25899	**0.98 ±.00**	**0.98 ±.00**	**0.83 ±.02**	0.85 ±.02	**0.99 ±.00**	**0.99 ±.00**	**0.99 ±.00**	**0.99 ±.00**	**1.00 ±.00**	**1.00 ±.00**
4	14	-	-	-	-	-	-	-	-	-	-
5	7	-	-	-	-	-	-	-	-	-	-
6	15	-	-	-	-	-	-	-	-	-	-
7	7908	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00	0.50 ±.00	0.67 ±.00
8	TO	-	-	-	-	-	-	-	-	-	-
9	135	0.88 ±.13	0.84 ±.29	0.64 ±.05	0.67 ±.24	0.87 ±.13	0.82 ±.29	0.92 ±.15	0.87 ±.31	0.94 ±.16	0.89 ±.31

behavior of positive examples in gem pickup, as the greedy algorithm commits to a target and moves toward it without changing direction.

We present some examples of the worst solutions. **Gem pickup, the (3,1) dataset** (avg. worst-case accuracy = 0.45).

$$\mathsf{X}right \tag{5.7}$$

Clearly, this is poor advice, as taking right in the next move is might appear in the negative examples as well but did not in this training set's.

$$\neg(pickup(G) \wedge dist(G, D) \wedge D \geqslant 2) \tag{5.8}$$

At the start, do not *pickup* a gem which has a distance greater than or equal to 2. This is, incidentally, a good advice overall, but badly generalizes in terms of a formula, as the negative examples generated by MCTS sometimes performs a *pickup* action randomly.

Now, we list a few examples of learned formulae for the best generalizing configurations for RockSample. **RockSample, the (3,3) dataset** (avg. worst-case accuracy = 0.53).

$$\mathsf{F}(\exists(R \in rocks, P \in guess\_val, P \leqslant 30).(check(R) \wedge guess(R, P))) \tag{5.9}$$

In words: eventually, check a rock whose probability of being good is less than or equal to 30%. This is, overall, reasonable advice, but quite vague and too general, it will be true on negative traces as well.

$$\neg\mathsf{X}east \text{ and } \neg(east \rightarrow west) \tag{5.10}$$

It is easy to see why both of these would be too general and fit the negative traces as well.

Table 5.14: Results of learning the 10 best advice formulae in the RockSample environment with different specification patterns for a (3,3) dataset. Each number in the train setup corresponds to a specification in Section 5.3.7.

Train	**Run**	N=12,R=4		N=6,R=4		N=12,R=8		N=18,R=4		N=18,R=8	
setup	**time[s]**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
None	86	0.62 ±.05	0.69 ±.02	0.67 ±.06	0.70 ±.01	0.65 ±.13	0.71 ±.08	0.54 ±.09	0.63 ±.08	0.53 ±.04	0.65 ±.02
7	**77**	0.59 ±.01	**0.68 ±.00**	0.63 ±.00	**0.70 ±.00**	0.55 ±.01	0.66 ±.01	**0.60 ±.00**	**0.69 ±.00**	0.50 ±.00	0.63 ±.00
9	89	**0.65 ±.05**	**0.70 ±.02**	**0.71 ±.05**	0.69 ±.01	**0.72 ±.13**	**0.76 ±.07**	0.49 ±.08	0.58 ±.08	**0.55 ±.04**	**0.66 ±.02**

RockSample, the (3,3) dataset with specification pattern $?_1\mathsf{U}?_2$ (avg. worse-case accuracy = 0.55).

$$north\mathsf{U}\left[\mathsf{F}(check(R) \wedge guess(R,P) \wedge P \leqslant 30)\right] \text{ and } north\mathsf{U}(\mathsf{X}\neg east) \tag{5.11}$$

Both of these temporal advice formulae describe concrete behaviour at the start of the trace, but these are poor in quality - because both positive and negative examples are generated stochastically, it is likely that these formulae will not fit either at the start of the trace.

RockSample, the (10,10) propositional dataset (avg. worse-case accuracy = 0.65).

$$\mathsf{F}(dist\_0\_0) \vee (num\_sampled\_50) \tag{5.12}$$

In words: eventually, it will be the case that the distance to rock 0 will be 0 or the percent of sampled rocks will be 50%. We can start to see why the propositional variant bore the best results overall for RockSample. Both of these propositions describe desirable conditions for good RockSample behaviour which are less likely to have been achieved in the negative examples.

$$(\mathsf{F}(delta\_x\_3\_\_2)) \rightarrow dist\_0\_0 \tag{5.13}$$

If, at the start of the trace, the x-axis distance to rock 3 is ever -2, then the distance to rock 0 is 0. This seems like it there was a positive example with a randomly favourable position for the rover at the start of the trace, as it was generated on a rock 0.

We now present the least generalizable temporal advice formulae for RockSample. **RockSample, the (10,3) dataset** (avg. worst-case accuracy = 0.50):

$$\mathsf{X}(east \rightarrow \mathsf{F}west) \text{ and } \mathsf{X}\neg\mathsf{X}east \vee \neg\mathsf{X}east \tag{5.14}$$

Both of these formulae describe behaviour at the start of the trace that seems too general which are likely to fit the negative examples as well.

Finally, we summarize the quantitative findings from the tables in a structured manner to efficiently convey the outcomes of the experiments. In addition, it may serve as a practical guide for practicioners attempting to learn temporal advice across various environments and datasets.

1. Creating the dataset.

   (a) **The quality of positive and negative examples.** The most crucial consideration is making sure that positive examples reflect desirable agent behavior and negative examples reflect undesirable behavior. Mixing desirable and undesirable behaviors in examples will increase runtimes and harm generalization.

(b) **Dataset imbalance.** Ensure you have more positive examples than negative ones, but include enough negative examples to maintain distinction in behaviours and generalizability. When the behaviors in both positive and negative sets are similar, imbalance issues may be less critical.

(c) **Dataset size.** Larger datasets may yield better generalization but could lead to longer runtimes or timeouts in complex settings. Ideally, every additional positive/negative data point demonstrates a desirable/undesirable behaviour.

2. Specifying the learning setup.

(a) **Preconditions for first-order actions.** Including environmental predicates as action preconditions may lead to more generalizable solutions but at the cost of an increased hypothesis space and execution time.

(b) **Propositions instead of first-order predicates.** When the learning task is difficult or instances where positive and negative examples exhibit similar behavior, converting predicates to propositional form can be advantageous. This may result in improved tractability and enhanced generalizability. Conversely, in alternative scenarios, such a transformation may substantially diminish the generalizability of the resulting solutions.

(c) **Penalties for examples.** For satisfiable learning problems, particularly those with already favorable results, introducing penalties is unlikely to yield significant improvements. However, if a dataset's learning task is unsatisfiable without penalties, exploring varying penalty configurations presented in this thesis may be beneficial.

(d) **Specification pattern.** If you desire a specific kind of temporal advice (e.g. liveness or safety properties), consider using a pattern for ILASP to fill in the placeholders. However, choosing a pattern that aligns well with the problem's nuances is crucial, as it can significantly impact both the feasibility of the learning problem and the generalization results.

Applying the approach of learning temporal advice formulae with ILASP to a new environment should be relatively straightforward, as this thesis provides open-source code with extendable Python classes along with the key considerations mentioned above.

Chapter 6

Conclusion and future work

6.1 Conclusion

This thesis presented an approach to automatically learn temporal advice specifications intended to guide various planning algorithms. We make several key contributions. We provide a formal reduction proof demonstrating the correctness of the Ielo et al. [2023] ILASP setup, ensuring that $\mathsf{LTL_f}$ formulae can be accurately learned and establishing the first detailed correctness proof of its kind.

Building on the work of Toro Icarte et al. [2018], we used linear temporal logic on finite traces to express advice and combined the ILASP-based approaches of Meli et al. [2024] and Ielo et al. [2023] to learn temporal advice formulae. Unlike previous studies, which focused on time-independent advice, this thesis is the first to learn temporal advice formulae. Additionally, we apply ILASP-based learning to model planning domains, exploring noisy environments and penalty effects while evaluating both computational efficiency and the quality of learned formulae.

We empirically demonstrated our methodology on the simplified gem pickup environment, which allows for extensive experimentation due to its simplicity. In addition, we used our approach to learn temporal advice formulae for RockSample, a more complex environment with a large action space and long action sequences required to complete the task. We explored various changes to the learning task to assess their impact on the runtime of the learning algorithm and the generalization scores of the retrieved advice formulae.

We have demonstrated that $\mathsf{LTL_f}$ equipped with the $\exists$ quantifier leads to advice formulae which generalizes better to unseen data which is more suitable for guiding planners. Most importantly, the experimental results point to the fact that the quality of the data in this learning setup is paramount. If the positive examples exhibit too many behaviors similar to the negative ones, even in cases where sacrificing fit incurs a penalty, achieving highly generalizable results is unlikely.

This thesis further contributes to the practical application of ILASP. By documenting challenges and necessary design decisions in a novel application, we complement the tool's limited documentation. Finally, we provide a public, extensible code repository containing reproducible experiments and comprehensive documentation, fostering future research and advancements in learning temporal advice formulae.

6.2 Future work

The methodology that we presented bears limitations. This section describes the many avenues of research the present thesis could be a stepping stone for. The scope of these projects are beyond this thesis, but we recognize their value and encourage the research community to investigate them.

6.2.1 Improvements in the learning setup

Based on our observations in Section 5.4, we offer the following improvements to our setup:

1. Discourage logically equivalent formulae in the top k retrieved soltuions. Some of the top k best solutions often have redundant, logically equivalent hypotheses. For instance, $\mathsf{G}\neg p$ logically equivalent to $\neg\mathsf{F}p$ or $\mathsf{FF}p$ can be simplified to $\mathsf{F}p$.
2. When the solution involves ranges, only keep the most restrictive one as the other are redundant. For example, among the top k solutions, the variable might be constrained as $D \leqslant 1$ and $-2 \leqslant D \leqslant 1$, but the second constraint is tighter and the first one is thus redundant.
3. Provide stronger temporal advice results for RockSample and other environments. Specifically, choose datasets that have clearly good and clearly bad behaviour.

6.2.2 Correctness proofs for other learning setups

In Section 3.4 we have mentioned that a proof of the result that *optimal* solution spaces correspond between between $\text{PL}_{\mathsf{LTL_f}}$ and $\text{ILP}^{context}_{LAS}$. Furthermore, in that section we have identified a novel learning setup for temporal logic, $\text{PL}^{weight}_{\mathsf{LTL_f}}$. Two results are still open problems, namely that solution spaces and optimal solution spaces correspond between $\text{PL}^{weight}_{\mathsf{LTL_f}}$ and ILP^{noise}_{LAS}. In Section 4.2 we have modified the learning task as was originally presented by Ielo et al. [2023] to be able to learn temporal advice formulae. There, we have provided first strides in arguing that the learning setup is correct and we can utilize ILASP to learn the formulae that we want, but a full formal proof of this result is still missing.

6.2.3 Applying the learned temporal advice formulae to planners

The following future directions for research illustrate the generality and potential of applying automatically learned advice expressed in temporal logics.

Classical planners Existing research has explored encoding domain knowledge using first-order temporal logic [Bacchus and Kabanza, 2000, Baier and Mcilraith, 2006], demonstrating its usefulness in representing domain-specific information for search problems. However, to the best of our knowledge, no prior work has focused on learning this knowledge from scratch. We thus think immediate future work could investigate applying automatically learned temporal advice formulae to enhance planning algorithms.

Model-based reinforcement learning Planning algorithms can easily incorporate numerical heuristic state value estimates in an approach-agnostic manner. A practical extension of our work is implementing learned temporal advice formulae by defining background knowledge functions for the given environment, as proposed by Toro Icarte et al. [2018]. This enables the learned temporal advice to be integrated with background knowledge to produce specific state value estimates. Toro Icarte et al. [2018] further show that these estimates can enhance reinforcement learning by modifying the R-MAX [Brafman and Tennenholtz, 2003] algorithm to incorporate them into the planning process.

A more restrictive approach to guiding planners is the use of shields, which prevent certain actions from being taken [Alshiekh et al., 2017]. Safety behaviors can be expressed in temporal logic, converted into shields, and used to constrain the actions of a reinforcement learning planner. A promising research direction would be exploring the conditions under which temporal safety specifications can be effectively learned and applied to RL planners. We strongly believe that carefully curating datasets would be crucial to maximizing the likelihood of learning a useful safety specification.

Model-free reinforcement learning Chakraborty [2023] introduced a broad definition of symbolic advice: "A *symbolic advice* $\mathcal{A}$ is a logical formula over finite paths whose truth value can be tested with an operator $\models$." This definition is very general, encompassing any symbolic formalism that can be evaluated on paths, which naturally includes temporal logics. In addition, they have a ready to use code base. Consequently, our approach to learning temporal advice formulae can be directly integrated with the method proposed by Chakraborty [2023] to prune Monte Carlo Tree Search.

The main drawback of this approach is the potential loss of optimality, as it may exclude optimal action sequences. To maintain optimality while still guiding MCTS, a softer approach could be used by biasing the exploration process rather than imposing hard constraints. One way to achieve this is by computing a macro-action, which is an action sequence that adheres to the temporal logic specification, and using it to influence the probability distribution of actions during the exploration step. This is similar to the method proposed by Meli et al. [2024], but their approach was limited to time-independent advice. Further research is needed to determine how to effectively extract useful macro-actions from a temporal logic formula, as there may exist multiple equally valid action sequences which satisfy the specification as illustrated in the last example of Section 4.3.

A widely used method for influencing reinforcement learning planners is through reward machines [Toro Icarte et al., 2022] or restraining bolts [De Giacomo et al., 2020, 2021]. In these appraoches, a temporal specification is provided and then converted into an automaton, which modifies the reward function of the underlying MDP. This adjustment reinforces the agent positively or negatively based on whether its actions contribute to satisfying the temporal specification on top of the existing reward function. An interesting research direction would be to explore the use of automatically learned advice formulae for generating reward machines and empirically investigating their usefulness in guiding planners. Additionally, comparing the efficiency of learning reward machines versus learning temporal advice formulae could provide valuable insights.

6.2.4 Learning advice expressed in other logics

Although we have argued that linear temporal logic is a general specification language, we recognize the value of exploring other formalisms. Some of these may be more easily applied to guide planners. For example, computation tree logic [Clarke and Emerson, 1981] incorporates quantification over execution paths, making it a potentially more natural formalism for reasoning about alternating action sequences and more suitable for guiding Monte Carlo tree search-based approaches. Alternating-time temporal logic [Alur et al., 2002] is a further generalization of computation tree logic allowing quantification over sets of agents. It could serve as a general formalism for temporal advice in multi-agent settings, something altogether not explored in this thesis. Mission-time temporal logic [Reinbacher et al., 2014] extends LTL by allowing the specification of time intervals during which a temporal logic formula must hold. This formalism could help define precisely when temporal advice should be followed within an episode. Recently, temporal logics for reasoning about actions in MDPs have been introduced, such as r-PLBP, which addresses safety and reward considerations for bounded policies under uncertainty [Lutz, 2023]. Adapting our approach to learn formulae in this logic and leveraging them to guide RL agents could be a significant step in improving the effectiveness of reinforcement learning agents.

6.2.5 Learning safety and norm specifications

In safety-critical situations, we want to prevent agents from ever taking actions that may lead to catastrophic outcomes. In those scenarios, softly guiding the agent with advice as our approach does is not good enough, as it does not guarantee that only safe actions will be taken. Shielding [Alshiekh et al., 2017] is an approach to prevent the agent from taking unsafe actions by making

sure that the agent adheres to the safety specification expressed in temporal logic. We think that our approach could be adapted to learn such safety specifications, but one must be extra careful with the data and guarantee that it shows safe behvaiour only.

In a related vein, one could learn norm specifications for agents. Alechina et al. [2018] demonstrate that norms can be expressed using temporal logic, meaning our setup would require only minor adjustments to learn them. The primary challenge would then be determining how to effectively apply these norms to agents. Recent work provides an approach to do just that by expressing norms in defeasible deontic logic and enforcing them on RL agents [Neufeld et al., 2022]. Additionally, a related approach uses ILASP to learn interpretable ASP specifications aiming to capture the ethical decisions made by the RL agent [Veronese et al., 2023]. Building on these methods to implement the learning of diverse norm specifications presents a concrete opportunity to advance RL agents' alignment with human values.

6.2.6 Porting to FastLAS and using custom scoring functions

FastLAS [Law et al., 2020a] presents a similar inductive logic programming approach to ILASP, offering attractive advantages of enhanced scalability, speed, and support for custom scoring functions. However, our preliminary experiments revealed that FastLAS was not readily applicable to our specific scenario. Following Law et al. [2020a], our learning task is classified as non-observational. This is because we try to learn a syntax tree composed of `edge` and `label` predicates. To ensure the syntax tree's validity, we must define constraints using these predicates. Additionally, the `label` predicate encodes the $\mathsf{LTL_f}$ semantics. And so, the primary issue encountered with FastLAS was its significantly slower runtime compared to ILASP, even in our smallest gem pickup scenarios. Attempt to mitigate this using the multiprocessing option proved unsuccessful. Therefore, refining our setup to better integrate with FastLAS to benefit from its computational efficiency remains a direction for future research.

In our work we put the most emphasis on learning advice formulae for guiding planners and we don't care that much about the length of the formula, provided that it is more useful for advising the planner. However, the most common scoring function for learned formulae is based on Occam's razor, that is, $score(\varphi) = |\varphi|$ where $|\cdot|$ is the length of the formula. This scoring function is used in ILASP as well to find the shortest fitting hypothesis. In the weighted example case, this score is augmented with the penalties incurred by every uncovered example. A promising future work direction would be to figure out alternative scoring functions which are more suited for learning symbolic advice for guiding planners.

The formalization of this problem for $\mathsf{LTL_f}$ formulae could look something like the following. Let $\mathrm{PL}_{\mathsf{LTL_f}} = \langle \mathcal{P}, E^+, E^- \rangle$ be a passive learning task. Let $score : \Phi \to \mathbb{R}$ be the scoring function of logical formulae. An $\mathsf{LTL_f}$ formula φ, written in $\mathcal{P}$, is an optimal solution of $\mathrm{PL}_{\mathsf{LTL_f}}$ if and only if φ is a solution of $\mathrm{PL}_{\mathsf{LTL_f}}$ and there is no $\mathsf{LTL_f}$ formula φ' written in $\mathcal{P}$ that is a solution of $\mathrm{PL}_{\mathsf{LTL_f}}$and $score(\varphi') < score(\varphi)$. This setup is more general than what we have explored in this thesis and would open up new research avenues beyond merely fitting formulae to traces and prioritizing length.

There exist works that may serve as useful references for this research direction. For instance, Cecconi et al. [2018, 2022] provide a software framework to evaluate reactive constraints which are closely related to temporal logics to compute various custom measures on test datasets. They provide a plethora of concrete measurements which go beyond accuracy as we have investigated mostly in our work. These measurements could be implemented in FastLAS and thus prefer advice formulae which adhere to the measurement more.

For example, Cecconi et al. [2018, 2022] could serve as a valuable references for this research direction. They offer a software framework for evaluating reactive constraints which are closely related to temporal logics to compute various custom measures on test datasets. They provide a wide range of concrete functions that go beyond the accuracy-focused approach we primarily explored. These measurements could be integrated into FastLAS, enabling the system to prefer

advice formulae that align better with the desired criteria.

6.2.7 Predicate invention

Our approach assumes that we, as learning task designers, define the hypothesis space, requiring a deep understanding of the environment. The choice of predicates and background knowledge supplied to the ILP task is crucial, as these elements enable the ILP system to describe traces and later allow the agent to reason effectively. In essence, these predicates serve as the agent's way of conceptualizing the problem. For example, we predefined $left$ and $right$ as predicates for constructing the syntax tree hypothesis.

However, a more flexible approach would be to automate predicate learning, allowing the system to discover useful concepts for describing the environment or guiding the actions an agent takes. This could help define high-level goals and even generate new possible actions, provided that abstract predicates can be mapped to the agent's capabilities. Recent practical works exploring this approach include Hocquette and Muggleton [2020] and Cropper and Morel [2021].

Bibliography

Natasha Alechina, Mehdi Dastani, and Brian Logan. Norm specification and verification in multi-agent systems. *Journal of Applied Logics*, 5(2):457, 2018.

Mohammed Alshiekh, Roderick Bloem, Ruediger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe Reinforcement Learning via Shielding, September 2017.

Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of The Acm*, 49(5):672–713, September 2002. ISSN 0004-5411. doi: 10.1145/585265.585270.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular Multitask Reinforcement Learning with Policy Sketches, June 2017.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987. ISSN 08905401. doi: 10.1016/0890-5401(87)90052-6.

Masataro Asai, Hiroshi Kajino, Alex Fukunaga, and Christian Muise. Classical Planning in Deep Latent Space. *Journal of Artificial Intelligence Research*, 74:1599–1686, August 2022. ISSN 1076-9757. doi: 10.1613/jair.1.13768.

Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1):123–191, 2000. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(99)00071-5. URL https://www.sciencedirect.com/science/article/pii/S0004370299000715.

Jorge A. Baier and Sheila A. Mcilraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 788–795, Boston, Massachusetts, 2006. AAAI Press. ISBN 978-1-57735-281-5.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):281–300, 1997. ISSN 0004-3702. doi: 10.1016/S0004-3702(96)00047-1.

Benjamin Bordais, Daniel Neider, and Rajarshi Roy. The Complexity of Learning Temporal Properties, August 2024.

Ronen Brafman and Moshe Tennenholtz. R-MAX – a general polynomial time algorithm for near-optimal ReinforcementLearning. *Journal of Machine Learning Research - JMLR*, 3, January 2003.

Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. LTLf/LDLf Non-Markovian Rewards. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v32i1.11572.

Alberto Camacho and Sheila A. McIlraith. Learning Interpretable Models Expressed in Linear Temporal Logic. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29:621–630, May 2021. ISSN 2334-0843, 2334-0835. doi: 10.1609/icaps.v29i1.3529.

Alberto Camacho, Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 6065–6073, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1. doi: 10.24963/ijcai.2019/840.

Alessio Cecconi, Claudio Di Ciccio, Giuseppe De Giacomo, and Jan Mendling. Interestingness of Traces in Declarative Process Mining: The Janus LTLp$$_f$$ Approach. In Mathias Weske, Marco Montali, Ingo Weber, and Jan Vom Brocke, editors, *Business Process Management*, volume 11080, pages 121–138. Springer International Publishing, Cham, 2018. ISBN 978-3-319-98647-0 978-3-319-98648-7. doi: 10.1007/978-3-319-98648-7_8.

Alessio Cecconi, Giuseppe De Giacomo, Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. Measuring the interestingness of temporal logic behavioral specifications in process mining. *Information Systems*, 107:101920, July 2022. ISSN 03064379. doi: 10.1016/j.is.2021.101920.

Chakraborty. *Monte Carlo Tree Search with Advice*. PhD thesis, 2023.

Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. Heuristic-Guided Reinforcement Learning. 2021.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1981. Springer-Verlag. ISBN 3-540-11212-X.

William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer-Verlag, Berlin, Heidelberg, 5 edition, 2003. ISBN 978-3-540-00678-8.

Andrew Cropper and Sebastijan Dumančić. Inductive Logic Programming At 30: A New Introduction. *Journal of Artificial Intelligence Research*, 74:765–850, June 2022. ISSN 1076-9757. doi: 10.1613/jair.1.13507.

Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 110(4):801–856, April 2021. ISSN 0885-6125, 1573-0565. doi: 10.1007/s10994-020-05934-z.

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, Ijcai '13, pages 854–860, Beijing, China, 2013. AAAI Press. ISBN 978-1-57735-633-2.

Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. Restraining Bolts for Reinforcement Learning Agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(09):13659–13662, April 2020. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v34i09.7114.

Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29:128–136, May 2021. ISSN 2334-0843, 2334-0835. doi: 10.1609/icaps.v29i1.3549.

Taylor Dohmen, Noah Topper, George Atia, Andre Beckus, Ashutosh Trivedi, and Alvaro Velasquez. Inferring Probabilistic Reward Machines from Non-Markovian Reward Processes for Reinforcement Learning, March 2022.

M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99cb37002)*, pages 411–420, 1999. doi: 10.1145/302405.302672.

Stefan Edelkamp. Planning with pattern databases. *Proceedings of the 6th European Conference on Planning (ECP-01)*, January 2002.

Ferber Patrick, Helmert Malte, and Hoffmann Jörg. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2020. doi: 10.3233/FAIA200364.

Nathanaël Fijalkow and Guillaume Lagarde. The complexity of learning linear temporal formulas from examples. In Jane Chandlee, Rémi Eyraud, Jeff Heinz, Adam Jardine, and Menno van Zaanen, editors, *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 237–250. PMLR, 2021.

Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo. Induction and Exploitation of Subgoal Automata for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 70:1031–1116, March 2021. ISSN 1076-9757. doi: 10.1613/jair.1.12372.

Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo. Hierarchies of Reward Machines. 2023.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019. doi: 10.1017/S1471068418000054.

Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. 1998.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier/Morgan Kaufmann, Amsterdam ; Boston, 2004. ISBN 978-1-55860-856-6.

Matthew Grounds and Daniel Kudenko. Combining Reinforcement Learning with Symbolic Planning. In Karl Tuyls, Ann Nowe, Zahia Guessoum, and Daniel Kudenko, editors, *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, volume 4865, pages 75–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-77947-6 978-3-540-77949-0. doi: 10.1007/978-3-540-77949-0_6.

Marek Grzes and Daniel Kudenko. Plan-based reward shaping for reinforcement learning. In *2008 4th International IEEE Conference Intelligent Systems*, pages 10–22–10–29, Varna, Bulgaria, September 2008. IEEE. ISBN 978-1-4244-1739-1. doi: 10.1109/IS.2008.4670492.

Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep Q-learning from Demonstrations, November 2017.

Céline Hocquette and Stephen H. Muggleton. Complete Bottom-Up Predicate Invention in Meta-Interpretive Learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 2312–2318, Yokohama, Japan, July 2020. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-6-5. doi: 10.24963/ijcai.2020/320.

J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, May 2001. ISSN 1076-9757. doi: 10.1613/jair.855.

J. Hoffmann, J. Porteous, and L. Sebastia. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, 22:215–278, November 2004. ISSN 1076-9757. doi: 10.1613/jair.1492.

Rodrigo Toro Icarte, Ethan Waldie, Richard Valenzano, Element Ai, and Margarita P Castro. Learning Reward Machines for Partially Observable Reinforcement Learning. 2019.

Antonio Ielo, Mark Law, Francesco Ricca, De Giacomo, and Alessandra Russo. Towards ILP-based LTLf passive learning. 2023.

León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila A. McIlraith. Symbolic Plans as High-Level Instructions for Reinforcement Learning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30:540–550, June 2020. ISSN 2334-0843, 2334-0835. doi: 10.1609/icaps.v30i1.6750.

Leslie Lamport. What good is temporal logic? In *IFIP Congress*, 1983.

Mark Law, Alessandra Russo, and Krysia Broda. Inductive Learning of Answer Set Programs. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence*, volume 8761, pages 311–325. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11557-3 978-3-319-11558-0. doi: 10.1007/978-3-319-11558-0_22.

Mark Law, Alessandra Russo, and Krysia Broda. Iterative learning of answer set programs from context dependent examples. *Theory and Practice of Logic Programming*, 16(5–6):834–848, 2016. doi: 10.1017/S1471068416000351.

Mark Law, Alessandra Russo, and Krysia Broda. The complexity and generality of learning answer set programs. *Artificial Intelligence*, 259:110–146, June 2018a. ISSN 00043702. doi: 10.1016/j.artint.2018.03.005.

Mark Law, Alessandra Russo, and Krysia Broda. Inductive Learning of Answer Set Programs from Noisy Examples, August 2018b.

Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2877–2885, April 2020a. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v34i03.5678.

Mark Law, Alessandra Russo, and Krysia Broda. The ILASP System for Inductive Learning of Answer Set Programs. 2020b.

Vladimir Lifschitz. Answer set planning. In *Proceedings ICLP-99*, pages 23–37, 1999.

Vladimir Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.

Sterre Lutz. R-PLBP: Temporal Logic for Reasoning about Safety and Rewards of Bounded Policies under Uncertainty. 2023.

Daoming Lyu, Fangkai Yang, Bo Liu, and Steven Gustafson. SDRL: Interpretable and Data-efficient Deep Reinforcement Learning Leveraging Symbolic Planning, February 2019.

Richard Maclin and Jude W. Shavlik. Creating advice-taking reinforcement learners. In Leslie Pack Kaelbling, editor, *Recent Advances in Reinforcement Learning*, pages 251–281. Springer US, Boston, MA, 1996. ISBN 978-0-585-33656-5. doi: 10.1007/978-0-585-33656-5_11.

Daniele Meli, Alberto Castellini, and Alessandro Farinelli. Learning Logic Specifications for Policy Guidance in POMDPs: An Inductive Logic Programming Approach. *Journal of Artificial Intelligence Research*, 79:725–776, February 2024. ISSN 1076-9757. doi: 10.1613/jair.1.15826.

Stephen Muggleton. Inductive logic programming: Derivations, successes and shortcomings. In Pavel B. Brazdil, editor, *Machine Learning: ECML-93*, pages 21–37, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47597-2.

Ashvin Nair, Abhishek Gupta, Murtaza Dalal, and Sergey Levine. AWAC: Accelerating Online Reinforcement Learning with Offline Datasets, April 2021.

Daniel Neider and Ivan Gavran. Learning Linear Temporal Properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, Austin, TX, October 2018. IEEE. ISBN 978-0-9835678-8-2. doi: 10.23919/FMCAD.2018.8603016.

Daniel Neider, Jean-Raphael Gaglione, Ivan Gavran, Ufuk Topcu, Bo Wu, and Zhe Xu. Advice-Guided Reinforcement Learning in a non-Markovian Environment. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10):9073–9080, May 2021. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v35i10.17096.

Emery A. Neufeld, Ezio Bartocci, Agata Ciabattoni, and Guido Governatori. Enforcing ethical goals over reinforcement-learning policies. *Ethics and Information Technology*, 24(4):43, December 2022. ISSN 1388-1957, 1572-8439. doi: 10.1007/s10676-022-09665-8.

Raz Nissim, Jorg Hoffmann, and Malte Helmert. Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning. 2011.

Pater Patel, Sunil Issar, J Scott Penberthy, George Ferguson, Hans Guesgen, Francisco Cruz, and Marc Pujol-Gonzalez. Domain-independent reward machines formodular integration of planning and learning. 2021.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, pages 46–57, Providence, RI, USA, September 1977. IEEE. doi: 10.1109/SFCS.1977.32.

Adrien Pommellet, Daniel Stan, and Simon Scatton. SAT-based learning of computation tree logic. In *Automated Reasoning: 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3–6, 2024, Proceedings, Part I*, pages 366–385, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-63497-0. doi: 10.1007/978-3-031-63498-7_22.

Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. Scalable anytime algorithms for learning fragments of linear temporal logic. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–280, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.

Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan,

Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Erika Ábrahám, and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, pages 357–372. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-54861-1 978-3-642-54862-8. doi: 10.1007/978-3-642-54862-8_24.

Gavin Rens and Jean-François Raskin. Learning Non-Markovian Reward Models in MDPs, January 2020.

Rajarshi Roy. *Learning Temporal Properties for Explainability and Verification.* Doctoralthesis, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, 2024.

Rajarshi Roy and Daniel Neider. Inferring Properties in Computation Tree Logic, October 2023.

Kristin Yvonne Rozier. Specification: The Biggest Bottleneck in Formal Methods and Autonomy. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, volume 9971, pages 8–26. Springer International Publishing, Cham, 2016. ISBN 978-3-319-48868-4 978-3-319-48869-1. doi: 10.1007/978-3-319-48869-1_2.

William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30:574–584, June 2020. ISSN 2334-0843, 2334-0835. doi: 10.1609/icaps.v30i1.6754.

David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In *Proceedings of the 24th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'10, pages 2164–2172, Red Hook, NY, USA, 2010. Curran Associates Inc.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and Characterizing Reward Hacking, September 2022.

Trey Smith and Reid Simmons. Heuristic search value iteration for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, Uai '04, pages 520–527, Arlington, Virginia, USA, 2004. AUAI Press. ISBN 0-9749039-0-6.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2020. ISBN 978-0-262-03924-6.

Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Advice-Based Exploration in Model-Based Reinforcement Learning. In Ebrahim Bagheri and Jackie C.K. Cheung, editors, *Advances in Artificial Intelligence*, volume 10832, pages 72–83. Springer International Publishing, Cham, 2018. ISBN 978-3-319-89655-7 978-3-319-89656-4. doi: 10.1007/978-3-319-89656-4_6.

Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning. *Journal of Artificial Intelligence Research*, 73:173–208, January 2022. ISSN 1076-9757. doi: 10.1613/jair.1.12440.

Mojtaba Valizadeh, Nathanaël Fijalkow, and Martin Berger. LTL learning on gpus. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 209–231, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-65633-0.

Giovanni Varricchione, Natasha Alechina, Mehdi Dastani, Giuseppe De Giacomo, Brian Logan, and Giuseppe Perelli. Pure-Past Action Masking. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(19):21646–21655, March 2024. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v38i19.30163.

Celeste Veronese, Daniele Meli, Filippo Bistaffa, Manel Rodríguez-Soto, Alessandro Farinelli, and Juan A. Rodríguez-Aguilar. Inductive Logic Programming For Transparent Alignment With Multiple Moral Values. In *CEUR WORKSHOP PROCEEDINGS*, pages 84–88, 2023.

Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. Joint inference of reward machines and policies for reinforcement learning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):590–598, June 2020. doi: 10.1609/icaps.v30i1.6756.

Wen-Chi Yang, Giuseppe Marra, Gavin Rens, and Luc De Raedt. Safe Reinforcement Learning via Probabilistic Logic Shields, March 2023.

Appendix

A Details on the structural validity of the syntax tree in ASP

To illustrate the constraints and their necessity, we employ a running example of an ill-formed tree exhibiting multiple violations, each of which will be addressed by a specific constraint. Consider the following ASP program attempting to represent a syntax tree representing an LTL_f formula:

```
1  label(1, a).
2  label(1, eventually).
3  edge(1, 2).
4  edge(1, 4).
5  edge(1, 5).
6  edge(2, 1).
7  label(3, c).
```

Problem: cyclic graphs. Lines 3 ad 6 show that this hypothesis has a cycle. A tree data structure must not have cycles. The following constraint addresses this issue:

```
1  \#bias(":- head(edge(X,Y)), Y <= X.").
```

In ILASP, this line means that we prune the hypothesis space in a way that only leaves edges where the second argument is larger than the first one. So, lines like line 6 would be disallowed:

```
1  label(1, a).
2  label(1, eventually).
3  edge(1, 2).
4  edge(1, 4).
5  edge(1, 5).
6  label(3, c).
```

Problem: DAG is not connected. Line 6 shows that a node is labeled but there is no edge going to it, so this node 3 is not connected to anything. This is not a valid tree. The following code ensures that a tree must be fully connected.

```
1  reach(1).
2  reach(T) :- edge(R,T), reach(R).
3  % It cannot be the case that X is a node and is not reachable
4  :- node(X), not reach(X).
5  % It cannot be the case that X is a node and does not have an edge to it
6  :- node(X), not edge(_,X), X > 1.
```

This constraint would disallow line 6:

```
1  label(1, a).
2  label(1, eventually).
3  edge(1, 2).
4  edge(1, 4).
5  edge(1, 5).
```

Problem: non-sequential nodes. In order to reduce the hypothesis space, we employ some symmetry breaking and make sure that the valid syntax trees only use sequential nodes.

```
1  node(X) :- node(X+1), X >= 1.
```

So, trees made up of nodes e.g. 1, 4, 5 would be disallowed and only 1, 2, 3 would be allowed. So, Lines 4 and 5 would be changed resulting in:

```
1 label(1, a).
2 label(1, eventually).
3 edge(1, 2).
4 edge(1, 3).
5 edge(1, 4).
```

Problem: too many neighbours. In LTL_f, each operator has an arity of at most 2, meaning that each node cannot have more than 2 neighbours (each of which denote the left and right side of the operator). This can be achieved with the following ASP constraint

```
1 :- node(X), 3 #count { Z: edge(X,Z) }.
```

This would disallow line 5 from our example above:

```
1 label(1, a).
2 label(1, eventually).
3 edge(1, 2).
4 edge(1, 3).
```

Problem: node does not have a label. Each node must have a label, otherwise it's not a valid syntax tree.

```
1 :- node(X), not label(X,_).
```

This would necessitate to add label predicates to the nodes 2 and 3 from the example above:

```
1 label(1, a).
2 label(1, eventually).
3 edge(1, 2).
4 edge(1, 3).
5 label(2, c).
6 label(3, b).
```

Problem: more than 1 label per node. A node must have exactly one label in a valid syntax tree to make it unambiguous. The constraint is as follows:

```
1 :- label(X, A), label(X, B), A != B.
```

Leading to the running example result:

```
1 label(1, eventually).
2 edge(1, 2).
3 edge(1, 3).
4 label(2, c).
5 label(3, b).
```

Problem: labels do not match the expected arities of logical operators.

```
1  proposition(b).
2  proposition(c).
3
4  arity(X,0) :- node(X), not edge(X,_).
5  arity(X,2) :- node(X), edge(X,Y), edge(X,Y1), Y < Y1.
6  arity(X,1) :- node(X), not arity(X,0), not arity(X,2).
7
8  symbol(A,0) :- proposition(A).
9  symbol(next,1).
10 symbol(until,2).
11 symbol(eventually,1).
12 symbol(always,1).
13 symbol(neg,1).
14 symbol(and,2).
15 symbol(or,2).
16 symbol(implies,2).
17
18 :- arity(X,N), label(X,Y), not symbol(Y,N).
```

The last line here says that it cannot be the case that a node `X` has an arity of `N` and the node `X` is labeled as `Y` but `Y` is not a predefined symbol of that same arity `N`.

This last change turns our running example to a valid syntax tree representing the formula F*c*:

```
label(1, eventually).
edge(1, 2).
label(2, c).
```

Note that Ielo et al. [2023] represented these constraints not as background knowledge (applying to all examples simultaneously) but as a special positive example. This approach was effective solely because their setup assigned infinite penalties to all examples, necessitating their coverage. Consequently, the special positive example also had to be satisfied, thus forcing adherence to the syntactic constraints of a syntax tree. However, since we allow penalties on examples, we incorporate these constraints as background knowledge to prevent the potential non-coverage of this special example.

B Design decisions for encoding temporal advice in ILASP

The way that temporal advice formulae appear in our work is a product of many iterations. One may reasonably ask, however, why did we settle on Definition 4.1.1 and its encoding as outlined in Section 4.2. It was primarily driven by the ASP technicalities as we demonstrate below.

At first, we thought to perhaps encode full first-order linear temporal logic and thus add first-order quantifiers in the syntax tree. However, how to do this in ASP did not seem straightforward at all. On top of it, we feared that the learning task would become very difficult, leading to long computational times and timeouts. So, we decided to stick to linear temporal logic.

Then, our idea was to allow environmental descriptors and actions to appear more freely into the formula, so that environmental variables wouldn't be limited to just appearing in the body of an action predicate as preconditions. For instance:

```
label(1, implies).
edge(1, 2).
edge(1, 3).
label(2, pickup(G)) :- gem(G).
label(3, dist(G, D)) :- distance(G, D), D < 1.
```

In this case, the variable `G` quantifies over gems. Notice that, however, `label(2, pickup(G))` and `label(3, dist(G, D))` may be derived in the interpretation but are not guaranteed to be derived for the same variable `G`. So, this approach, while more flexible, it does not ensure that statements will be done on the same objects.

Additionally, we found that the system is more inclined to learn the dynamics of the environment, or formulas that rely solely on environmental predicates, rather than those involving actions. However, our primary focus in this work is on information related to actions. Thus, we thought it reasonable to necessarily link actions with environmental variables. Note that we may allow "dummy" environmental variables to show that an action can simply be performed on an object, e.g.:

```
label(1, pickup(V1)) :- gem(V1).
```

Consider another program:

```
label(1, pickup(V1)) :- distance(V1, V2), V2 < 1.
```

This logical statement may be made true even when the agent performed a pickup action on a gem which was *at some point in the trace* a distance of < 1 away from the agent. We need to ensure that the *time* when the pickup action was executed aligns with the moment when the distance from the agent to the gem was measured. To achieve this, we introduced a time variable, `T`, which ensures that all actions and environmental variables correspond to the same point in

time. And so, instead, we have this which is the final format of the temporal advice encoding in ILASP:

```
label(1, pickup(V1), T) :- distance(V1, V2, T), V2 < 1.
```

C Analysis of generated datasets

First, we explain our notation used in Figure 6.1, Figure 6.2, Table 6.1. "Train" datasets were used for learning, while "test" datasets were for evaluation in our main experiments, we analyze both. The setups are described with x, y_z_w where x and y denote number of positive and negative examples, z is the gem pickup line length, and w is the number of gems. The normalized Levenshtein distance in Table 6.1 is calculated as the minimum number of insertions, deletions, or substitutions required to transform one trace into another. It is then normalized to the range [0,1] by dividing by the length of the longer trace. Our goal for using the Levenshtein distance is to provide an estimate of similarity between the generated traces. This measure was applied solely to computed action sequences, ignoring environmental predicates entirely. Additionally, each action was converted into a proposition without its object (e.g., *pickup*(2) was reduced to *pickup*).

In Table 6.1, we observe that the gem pickup datasets exhibit greater dissimilarity between positive and negative examples (N. Lev. Dist. $\sim$ 0.7) compared to RockSample (N. Lev. Dist. $\sim$ 0.56). However, an exception for gem pickup is test_50,50_5_3, where negative examples achieve returns that overlap with the positive examples'. Overall, gem pickup shows smaller differences between the minimum positive return and the maximum negative return than RockSample. This is likely because gem pickup is easier to solve with MCTS than RockSample.

In Figure 6.1, we see that positive examples consistently have a normalized return close to 1, while negative examples (generated by MCTS) exhibit more variation in their returns. Notably, in the case of gp_test_50,50_5_3, MCTS even finds an optimal solution, achieving a high reward. A similar pattern is observed in Figure 6.2, with the key difference being that positive returns are more widely distributed, as they, like the negative examples, are also generated by MCTS.

In summary, these results help explain why gp_test_50,50_5_3 has consistently shown lower generalization scores in our experiments. Additionally, the lower Levenshtein distances in the RockSample datasets indicate greater similarity between positive and negative traces, despite their differing returns. This, combined with some negative examples achieving high returns, further explains why generalization performance for RockSample has also remained consistently lower.

Table 6.1: Analysis of similarity between positive and negative traces for various datasets. N. Lev. Dist denotes the mean and std. deviation of normalized levenshtein distance which was computed between each positive trace and every negative trace. min(pos. ret.) refers to the minimum value of the normalized returns for the positive example, similarly for max(neg. ret.).

Environment	**Setup**	**N. Lev. Dist.**	**min(pos. ret.) - max(neg. ret.)**
Gem pickup	train_3,3_10_3	0.71 ± 0.14	0.11
	train_10,10_10_3	0.72 ± 0.15	0.08
	test_50,50_10_3	0.72 ± 0.12	0.04
	test_50,50_5_3	0.58 ± 0.14	0
	test_50,50_20_3	0.8 ± 0.10	0.05
RockSample	train_3,3_12_4	0.56 ± 0.08	0.89
	train_10,10_12_4	0.55 ± 0.07	0.62
	test_50,50_12_4	0.54 ± 0.08	0.07
	test_50,50_6_4	0.6 ± 0.09	-0.07
	test_50,50_18_4	0.57 ± 0.06	0.30

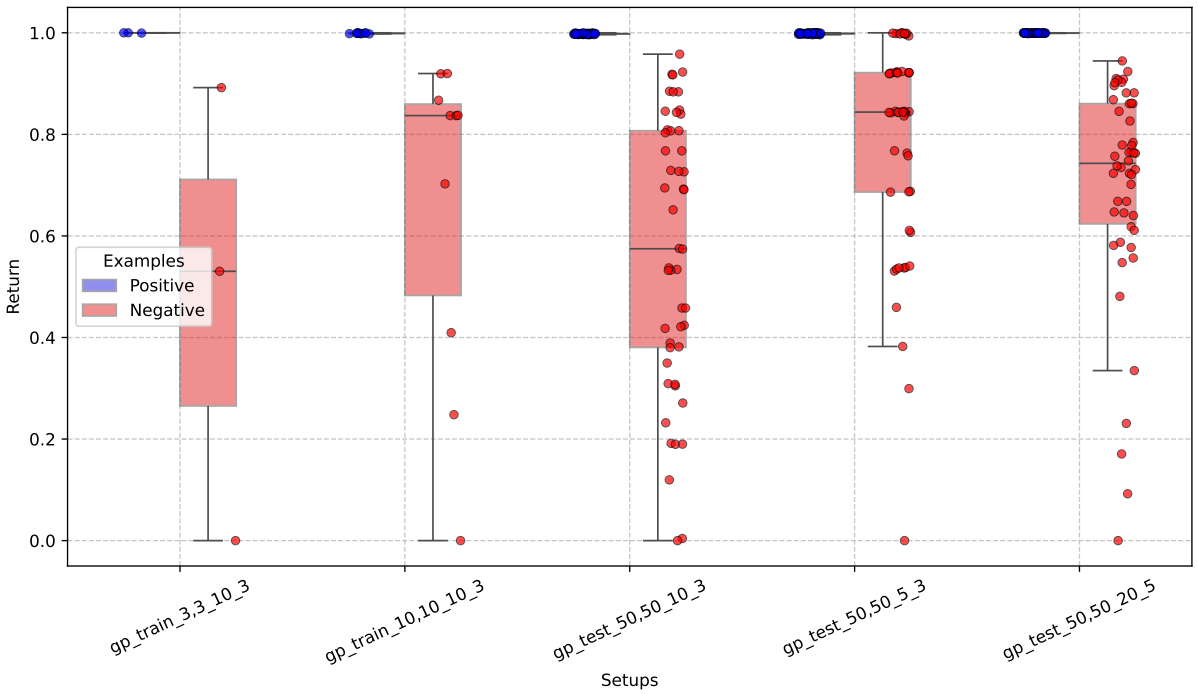


Figure 6.1: The distributions of normalized returns for positive and negative gem pickup examples are shown across setups. Points show the specific return values, and boxplots statistically summarize them.

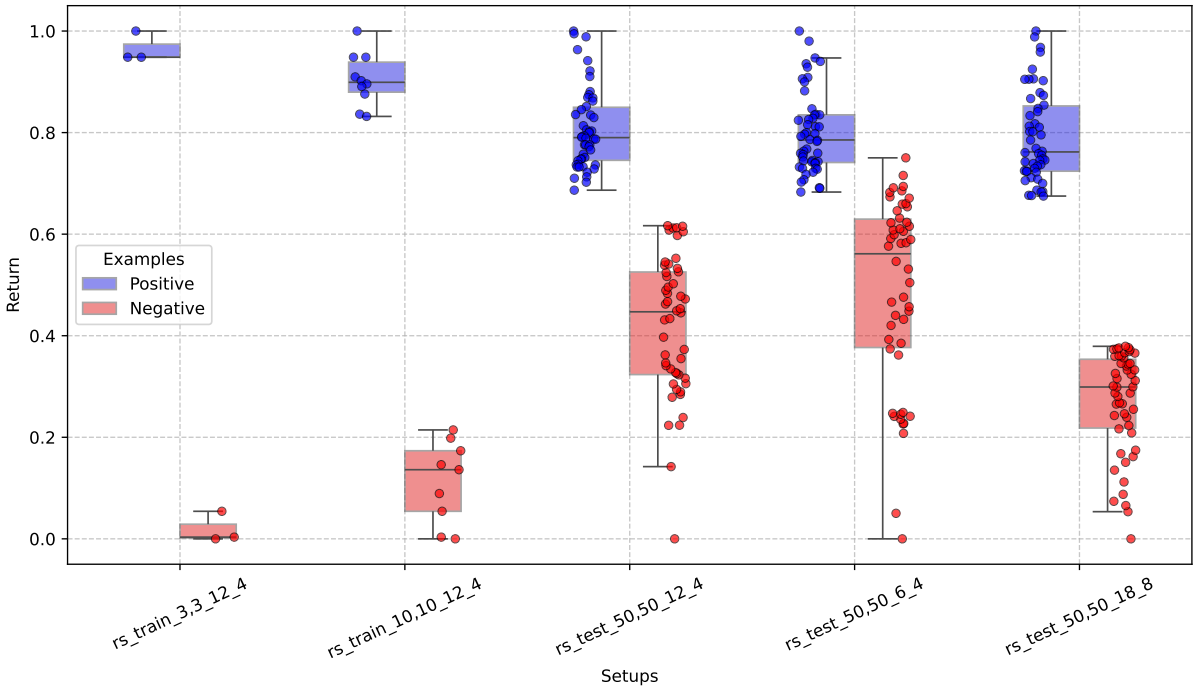


Figure 6.2: The distributions of normalized returns for positive and negative RockSample examples are shown across setups. Points show the specific return values, and boxplots statistically summarize them.

D Additional experimental results

Table 6.2: Results of learning the 10 best advice formulae in the gem pickup environment with different penalties for a 10,10 dataset. A single number in the train setup column means a constant penalty, Tr. len. means the trace length of the example, and three numbers indicate the different *low*, *mid*, and *high* penalties.

Train	**Exec.**	L=10,G=3		L=5,G=3		L=10,G=5		L=20,G=3		L=20,G=5	
setup	**time(s)**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**	**Acc.**	**F1**
None	496	0.93 ±.00	0.93 ±.00	0.66 ±.00	0.75 ±.00	0.93 ±.03	0.93 ±.02	0.98 ±.01	0.98 ±.01	1.00 ±.01	1.00 ±.01
2	7962	0.93 ±.01	0.93 ±.01	0.66 ±.00	0.75 ±.00	0.91 ±.04	0.92 ±.03	0.96 ±.02	0.97 ±.02	0.99 ±.01	0.99 ±.01
5	19658	0.95 ±.05	0.95 ±.04	0.75 ±.10	0.80 ±.06	0.96 ±.04	0.97 ±.03	0.98 ±.01	0.98 ±.01	1.00 ±.01	1.00 ±.01
10	**244**	0.93 ±.01	0.93 ±.01	0.66 ±.00	0.75 ±.00	0.91 ±.04	0.92 ±.03	0.96 ±.02	0.97 ±.02	0.99 ±.01	0.99 ±.01
20	18837	0.97 ±.01	0.98 ±.01	0.80 ±.07	0.84 ±.05	0.99 ±.01	0.99 ±.01	0.99 ±.01	0.99 ±.01	**1.00 ±.00**	**1.00 ±.00**
50	TO	-	-	-	-	-	-	-	-	-	-
Tr. len.	17147	**0.98 ±.00**	**0.98 ±.00**	**0.82 ±.02**	**0.85 ±.02**	**0.99 ±.00**	**0.99 ±.00**	**0.99 ±.00**	**0.99 ±.00**	**1.00 ±.00**	**1.00 ±.00**
5,2,5	578	0.93 ±.00	0.93 ±.00	0.66 ±.00	0.75 ±.00	0.94 ±.00	0.94 ±.00	0.98 ±.00	0.98 ±.00	**1.00 ±.00**	**1.00 ±.00**
7,4,7	394	0.93 ±.01	0.93 ±.01	0.66 ±.00	0.75 ±.00	0.91 ±.04	0.92 ±.03	0.96 ±.02	0.97 ±.02	0.99 ±.01	0.99 ±.01
15,5,15	331	0.92 ±.01	0.93 ±.01	0.65 ±.01	0.74 ±.01	0.89 ±.04	0.90 ±.04	0.95 ±.03	0.95 ±.03	0.99 ±.01	0.99 ±.01