Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Symbolic Reinforcement Learning using Inductive Logic Programming

Author:
Kiyohito Kunii

Supervisor:
Prof. Alessandra Russo
Mark Law
Ruben Vereecken

Submitted in partial fulfillment of the requirements for the MSc degree in MSc in Computing Science of Imperial College London

September 2018

Abstract

Reinforcement Learning (RL) has been applied and proven to be successful in many domains. However, most of current RL methods face limitations, namely, low learning efficiency, inability to use abstract reasoning, and inability of transfer learning to similar environments. In order to tackle these shortcomings, we introduce a new learning approach called *ILP(RL)*. ILP(RL) is based on Inductive Logic Programming (ILP) and Answer Set Programming (ASP) and learns a state transition with ILP and navigate through an environment using ASP. ILP(RL) learns a general concept of a state transition, called a hypothesis in an environment, and generates a plan for a sequence of actions to a destination. The learnt hypotheses is highly expressive and transferable to a similar environment. While there are a number of past papers that attempt to incorporate symbolic representation to RL problems in order to achieve efficient learning, there has not been any attempt of applying ASP-based ILP into a RL problem. We evaluated ILP(RL) in a various simple maze environments, and show that ILP(RL) finds an optimal policy faster than Q-learning. We also show that transfer learning of the learnt hypothesis successfully improves learning on a new but similar environment. While this preliminary ILP(RL) framework has limitations for scalability and flexibility, the results of the evaluations are promising, and open up an avenue for future research directions.

Acknowledgements

I would like to thank Prof. Alessandra Russo for supervising my project, and for her enthusiasm for my work and invaluable guidance throughout.
I would also like to thank Mark Law for his expertise on inductive logic programming and answer set programming, as well as many fruitful discussions, and Ruben Vereecken for his expertise on reinforcement learning and for providing me with advice and assistance for technical implementation.

Contents

List of Figures and Tables

Chapter 1

Introduction

1.1 Motivation

Reinforcement learning (RL) is a subfield of machine learning concerned with agents learning how to behave in an environment by interacting with the environment in order to maximise the total rewards they receive. The strength of RL is that it can be applied to many different environments where it is unknown to an agent how to perform a task. RL has been proven to work well in a number of complex environment, such as dynamic, real environments given sufficient learning time. In particular, combined with deep learning (DL), RL has been applied successfully in a number of areas, such as video games [1], the game of Go [2] and robotics [3].

Despite these successful applications of RL, there are still a number of issues that RL needs to overcome. First, most of the RL algorithms require a large number of trial-and-error interactions with a long training time, which is also computationally expensive. Second, most of the RL algorithms have no thought process and reasoning in the decision making and do not make use of high-level abstract reasoning, such as understanding symbolic representations or causal reasoning. Third, the transfer learning (TL), where the experience that an agent gained by performing one task can be applied in a different task is limited, and the agent performs poorly even on a new but similar task.

In order to overcome these limitations of existing RL methods, we introduce a new RL approach which is based on using Inductive Logic Programming (ILP). ILP is another subfield of machine learning which is based on logic programming. It focuses on the computation of the hypothesis expressed in the form of logic programs that, together with background knowledge, entail all positive examples and none of the negative examples. ILP has several advantages over RL. First, unlike most statistical machine learning methods, ILP requires a small number of training data due to the presence of language bias, which defines what the logic program can learn. Second, the learnt hypothesis is expressed using a symbolic representation and therefore is easy for human users to interpret . Third, since the learnt hypothesis is an abstract and general concept, it can be easily applied to a different learning task, making transfer learning possible. The disadvantages of an ILP system are that examples, or training data, have to be clearly defined, which may not be possible in a complex senario. Another disadvantage is that ILP suffers from scalability. The search space for a hypothesis defined by the language bias increases with respect to the complexity of learning tasks and slows down the learning process. Despite these shortcomings, however, there have been a number of advances in ILP, especially ILP frameworks based on Answer Set Programming (ASP), Taking into consideration the progress and limitations of RL and ILP, we

developed a proof-of-concept for a new RL approach by incorporating a new ASP-based ILP framework called Learning from Answer Set (ILASP). This new symbolic RL learns a state transition within an environment, and uses the learnt hypothesis and background knowledge to generate a sequence of actions in the environment.
There is a certain amount of work introducing symbolic representation into RL in order to achieve more data-efficient learning as well as transparent learning. One of the methods is to incorporate symbolic representations into Deep Reinforcement Learning (DRL) [4]. This approach shows potential. In addition, there has been work on using ASP with RL to achieve data-efficient learning. However, none of the papers attempt to apply inductive learning in RL scenarios.
Finally, recent advances in Inductive Logic Programming (ILP) research have enabled us to apply ILP in more complex situations and there are a number of new ILP frameworks based on Answer Set Programmings (ASPs) that work well in non-monotonic scenarios.
Because of the recent advancement of ILP and RL, it is natural to consider that a combination of both approaches would be the next field to explore. Therefore, our motivation is to combine these two different subfields of machine learning and devise a new way of RL algorithm in order to overcome some of the limitations in the RL.

1.2 Objectives

The main objective of this project is to provide a proof-of-concept of a new RL framework using ILP called *ILP(RL)*, and to investigate the potential of ILP(RL) for improving the learning efficiency and transfer learning capability beyond the limitations of current RL methods.
The objective of this project is divided into the following high-level objectives:

1. **Translation of state transitions into ASP syntax.**
   In RL, an agent interacts with an environment by taking an action and observing a state and rewards (Markov Decision Process, or MDP). Since ILP(RL) requires their inputs to be specified in ASP syntax, the conversion between MDP and ASP is required.

2. **Development of learning tasks.**
   ILP frameworks are based on a search space specified by language bias for a learning task, and need to be specified by the user. Various hyper-parameters for the learning task are also considered.

3. **Development of a planning.**
   Having learnt a hypothesis using an ILP algorithm, the agent needs to choose an action based on the learnt hypothesis. We investigate how the agent can effectively plan a sequence of actions using ASP.

4. **Evaluation of the new framework in various environments.**
   In order to investigate the applicability of ILP(RL), evaluation of the new framework in various scenarios is conducted in order to gain insight into its potential, in particular how it improves the learning process and capability of transfer learning.

This project is a proof of concept for a new way of RL using ILP and ASP and therefore more complex environments such as continuous states or stochastic environments are not considered in this project. The possibilities of applying these more complex environments are discussed in Section6.2.

1.3 Contributions

The contributions of this project are as follows.

1. **Development of ILP(RL).**
   This project contributes to the development of a new learning framework ILP(RL), by incorporating an ASP-based ILP framework into RL. ILP(RL) is able to learn state transitions with inductive learning and execute a planning using ASP. To our knowledge, this is the first attempt of incorporating an ASP-based ILP learning framework is incorporated into an RL scenario.

2. **Implementation of ILP(RL).**
   Another contribution is the implementation of ILP(RL). We developed the initial version of fully working ILP(RL) in order to examine the properties of the current ILP(RL) framework.

3. **Evaluation of ILP(RL).**
   We conducted four evaluations in different environments and show that ILP(RL) finds an optimal path faster than an existing RL algorithm, as well as limited transfer learning. These evaluations also reveal some of the limitations of the current framework of ILP(RL). We also show that the learnt hypothesis is easy to understand for human users, and can be applied to other similar environments to optimise the agent's learning process.

1.4 Report Outline

Chapter 2. The necessary background of Answer Set Programming, Inductive Logic Programming and Reinforcement Learning for this project are described.

Chapter 3. The descriptions of the new framework, called ILP(RL), is explained in details, and we highlight each aspect of the learning steps with examples.

Chapter 4. The performance of ILP(RL) is measured in various maze game environments, and learning performance and the capability of the transfer learning are compared it against an existing RL algorithm. We evaluate the outcomes and discuss some of the issues we currently face with the current framework.

Chapter 5. We review previous research in the related fields. Since there is no research that attempts to apply ASP-based ILP to RL, we review applications of ASP in RL and the symbolic representations in RL.

Chapter 6. We summarise the work of this project and discuss avenues of further research.

Chapter 2

Background

This chapter introduces the necessary background on Answer Set Programming (ASP), Inductive Logic Programming (ILP) and Reinforcement Learning (RL), which provide the foundations of our research.

2.1 Answer Set Programming (ASP)

Answer Set Programming (ASP) is a form of declarative programming aimed at knowledge-intensive applications such as difficult search problems [5]. We first introduce a stable model which is the foundation of ASP, and describe ASP syntax.

2.1.1 Stable Model Semantics

The semantics of the logic are based on the notion of interpretation, which is defined under a *domain*. A domain contains all the objects that exist in a given problem. In logic, it is convention to use a special interpretation called a *Herbrand interpretation*.

Definition 2.1. *Definite Logic Program* is a set of definite rules, and A *definite rule* is of the form:

$$\underbrace{h}_{\text{head}} \leftarrow \underbrace{a_1, a_2, ..., a_{\text{n}}}_{\text{body}} \tag{2.1}$$

h and a_1, ..., a_{n} are all atoms. h is the *head* of the rule and a_1, ..., a_{n} are the *body* of the rule.

Definition 2.2. *Normal Logic Program* is a set of normal rules, A *normal rule* is of the form:

$$\underbrace{h}_{\text{head}} \leftarrow \underbrace{a_1, a_2, ..., a_{\text{n}}, not\ b_{\text{n+1}}, ...not\ b_{\text{n+k}}}_{\text{body}} \tag{2.2}$$

where $h, a_1, ...a_{\text{n}},$ and $b_{\text{n+1}}, ..., b_{\text{n+k}}$ are all atoms.

Definition 2.3. The *Herbrand Domain*, or *Herbrand Universe*, of a normal logic program *P*, denoted *HD(P)*, is the set of all ground terms constructed from constants and function symbols that appear in *P*.

Definition 2.4. The *Herbrand Base* of a normal logic program *P*, denoted *HB(P)*, is the set of all ground atoms that are formed by predicate symbols in *P* and terms in *HD(P)*.

Definition 2.5. The *Herbrand Interpretation* of a program *P*, denoted *HI(P)*, is a subset of the HB(P), and ground atoms in an HI(R) are assumed to be true [6].

In order to define a stable model, we need to define a minimal Herbrand model of a normal logic program *P*.

Definition 2.6. The *Herbrand Model* of a definite logic program *P* is a *HI(P)* that satisfies all the clauses in *P*. In other words, the set of clauses *P* is unsatisfiable if no Herbrand model is found.

Definition 2.7. *Least Herbrand Model*, denoted *M(P)*, is an unique Minimal Herbrand model for definite logic programs.

Example 2.1.1. (Herbrand Interpretation, Herbrand Model and Least Herbrand Model)
We use an simple example to highlight the definitions of Herbrand Interpretation, Herbrand Model and Least Herbrand Model.

$$\text{P} = \begin{cases} \textit{drive(X)} \leftarrow \textit{hasCar(X)}. \\ \textit{hasCar(john)}. \end{cases} \qquad \text{HD(P)} = \{ \text{ john } \}, \text{HB(P)} = \{ \text{ hasCar(john), drive(john) } \}$$

Given the above program, there are four Herbrand Interpretations:
HI(P) = $\langle$ {hasCar(john)}, {drive(john)}, {hasCar(john), drive(john)}, {} $\rangle$,
and one Herbrand Model (as well as M(P)) = {q(a), p(a)}.

Definition 2.8. The *Minimal Herbrand Model* of a normal logic program *P* is a Herbrand model *M* if none of the subset of *M* is a Herbrand model of *P* [7].

To solve a normal logic program *P*, the program *P* needs to be grounded. The *grounding* of *P* is the set of all clauses c $\in$ *P* and with variables replaced by all possible terms in the Herbrand Domain.

Definition 2.9. The algorithm of grounding starts with an empty program *Q* = { } and the relevant grounding is constructed by adding each rule *R* to *Q* such that:

- *R* is a ground instance of a rule in *P*.
- Their positive body literals already occurs in the in the rules in *Q*.

The algorithm terminates when no more rules can be added to *Q* [5].

Example 2.1.2. (Grounding)

$$\text{P} = \begin{cases} \textit{drive(X)} \leftarrow \textit{hasCar(X)}. \\ \textit{hasCar(john)}. \end{cases}$$

ground(P) in this example is {drive(john) $\leftarrow$ hasCar(john), hasCar(john) }, as hasCar(john) is already grounded and added to Q, and it is also a positive body literal for the first rule, resulting in drive(john) $\leftarrow$ hasCar(john).

The entire program must not only be grounded in order for an ASP solver to work, but each rule must also be *safe*. A rule *R* is safe if every variable that occurs in the head of the rule occurs at least once in body$^+$(R). In order to obtain the Stable Model of a program P, P needs to be converted using *Reduct* with respect to an interpretation X.

Definition 2.10. The *Reduct* of P with respect to any set X of ground atoms of a normal logic program can be constructed such that:

- If the body of any rule in P contains an atom which is not in X, those rules need to be removed.
- All default negation atoms in the remaining rules in P need to be removed.

Definition 2.11. X is a *stable model* of P if it is the least Herbrand Model of P^X [7].

Example 2.1.3. (Reduct)

$$P = \begin{cases} male(X) \leftarrow not\ female(X). \\ female(X) \leftarrow not\ male(X). \end{cases} \quad X = \{male(john), female(anna)\}$$

Where X is a set of atoms. ground(P) is
male(john) ← not female(john).
male(anna) ← not female(anna).
female(john) ← not male(john).
female(anna) ← not male(anna).

The first step removes male(anna) ← not female(anna). and female(john) ← not male(john).

male(john) ← not female(john).
female(anna) ← not male(anna).

The second step removes negation atoms from the body.
Thus reduct P^X is $(ground(P))^X$ = {male(john), female(anna).}

For normal logic program, there may be more than one or no stable models. In stable model semantics, there are two different notion of entailments.

Definition 2.12. An atom a is *bravely* entailed by a normal logic program P, denoted $P \models_b a$, if it is true in at least one stable model of P. An atom a is *cautiously* entailed by a normal logic program P, denoted $P \models_c a$ if it is true in every stable model of P [8].

Example 2.1.4. (Brave and Cautious Entailment)

$$P = \begin{cases} male \leftarrow not\ female,\ tall. \\ female \leftarrow not\ male,\ tall. \\ tall. \end{cases}$$

In this case, $P \models_b$ {male, female, tall} and $P \models_c$ {tall}

2.1.2 Answer Set Programming (ASP) Syntax

An answer set of normal logic program P is a stable model defined by a set of rules, where each rule consists of literals, which are made up with an atom p or its default negation *not p* (*negation as failure*). Answer Set Programming (ASP) is a normal logic program with extensions: constraints, choice rules and optimisation statements.
A *constraint* of the program P is of the form:

$$\leftarrow a_1, ..., a_n, not\ b_n, ..., not\ b_{n+k} \tag{2.3}$$

where the normal rule has an empty head, and a normal rule with empty body is a *fact*. The constraint filters any irrelevant answer sets. When computing ground(P)$_x$, the empty head becomes $\bot$, which cannot be in the answer sets. So any answer set that includes ground body of a constraint is eliminated [5].
A *choice rule* can express possible outcomes given an action choice, which is of the form:

$$\underbrace{l\{h_1, ..., h_m\}u}_{\text{head}} \leftarrow \underbrace{a_1, ..., a_n, not\ b_{n+1}, ..., not\ b_{n+k}}_{\text{body}} \tag{2.4}$$

where l and u are integers and h_i for $1 \leq i \leq m$. The head in the choice rule is called *aggregates*. Informally, if the body of a ground choice rule is satisfied by an interpretation X, given a ground choice rule, the choice rule generates all answer sets in which $l \leq |X \cap \{h_1, ..., h_m\}| \leq u$ [8].
Optimisation statement is useful to sort the answer sets in terms of preference, which is of the form:

$$\begin{aligned} &\#minimize[a_1{=}w_1,...a_n{=}w_n]\ or \\ &\#maximize[a_1{=}w_1,...a_n{=}w_n] \end{aligned} \tag{2.5}$$

where w_1,..., w_n are integer weights and a_1,...,a_n are ground atoms. ASP solvers compute the scores of the weighted sum of the sets of ground atoms based on the true answer sets, and find optimal answer sets which either maximise or minimise the score.
Clingo is one of the modern ASP solvers that executes the ASP program and returns answer sets of the program [9].

2.2 Inductive Logic Programming (ILP)

Inductive Logic Programming (ILP) is a subfield of machine learning aimed at supervised inductive concept learning, and is the intersection between machine learning and logic programming [10]. The purpose of ILP is to inductively derive a hypothesis H that is a solution of a learning task, which covers all of the positive examples and none of the negative examples, given a hypothesis language for search space and cover relation [11]. ILP is based on learning from entailment, as shown in Equation 2.6.

$$B \wedge H \models E \tag{2.6}$$

where E contains all of the positive examples, denoted E^+, and none of the negative examples, denoted E^-.
An advantage of ILP over statistical machine learning is that the hypothesis that an agent learns can be easily understood by a human, as it is expressed in first-order logic, making the learning process explainable. By contrast, a limitation of ILP is scalability. There are usually thousands or more examples in many real-world examples. Scaling ILP tasks to cope with large examples is a challenging task [12].

2.2.1 ILP under Answer Set Semantics

There are several ILP non-monotonic learning frameworks under the answer set semantics. We first introduce two of them: *Cautious Induction* and *Brave Induction* ([13]), which are the foundations of *Learning from Answer Sets* discussed in Section 2.2.2, a state-of-the-art ILP framework that we will use for our new framework (for other non-monotonic ILP frameworks, see [14], [15], and [16]).

Cautious Induction

Definition 2.13. *Cautious Induction* task[1] is of the form $\langle B, E^+, E^- \rangle$, where B is the background knowledge, E^+ is a set of positive examples and E^- is a set of negative examples. $H \in ILP_{\text{cautious}}\langle B, E^+, E^- \rangle$ if and only if there is at least one answer set A of $B \cup H$ ($B \cup H$ is satisfiable) such that for every answer set A of B ∪ H:

- $\forall e \in E^+ : e \in A$
- $\forall e \in E^- : e \notin A$

The limitation of Cautious Induction is that positive examples must be true for all answer sets and negative examples must not be included in any of the answer sets. These conditions may be too strict in some cases, and Cautious Induction is not able to accept a case where positive examples are true in some of the answer sets but not all answer sets of the program.

Brave Induction

Definition 2.14. *Brave Induction* task is of the form $\langle B, E^+, E^- \rangle$ where, B is the background knowledge, E^+ is a set of positive examples and E^- is a set of negative examples. $H \in ILP_{\text{brave}}\langle B, E^+, E^- \rangle$ if and only if there is at least one answer set A of $B \cup H$ such that:

- $\forall e \in E^+ : e \in A$
- $\forall e \in E^- : e \notin A$

The limitation of Brave Induction is that it is not possible to learn constraints, since the conditions of the Definition 2.14 only apply to at least one answer set A, whereas constrains must rule out all answer sets that meet the conditions of the Brave Induction.

2.2.2 Inductive Learning of Answer Set Programs (ILASP)

Learning from Answer Sets (LAS)

Learning from Answer Sets (LAS) was developed in to facilitate more complex learning tasks that neither Cautious Induction nor Brave Induction could learn. We define examples used in LAS are a set of atoms called a *Partial Interpretation*.

Definition 2.15. A *Partial Interpretation E* is of the form:

$$E = \langle e^{\text{inc}}, e^{\text{exc}} \rangle. \tag{2.7}$$

where e^{inc} and e^{exc} are called *inclusions* and *exclusions* of *E* respectively. An answer set *A extends E* if and only if $(e^{\text{inc}} \subseteq A) \wedge (e^{\text{exc}} \cap A = \emptyset)$.

A *Learning from Answer Sets* task is of the form:

$$T = \langle B, S_{\text{M}}, E^+, E^- \rangle \tag{2.8}$$

where B is background knowledge, S_{M} is hypothesis space, and E^+ and E^- are examples of positive and negative partial interpretations. S_{M} consists of a set of normal rules, choice

[1] This is more general definition of Cautious Induction than the one defined in [13], as the concept of negative examples was not included in the original definition.

rules and constraints. S_{M} is the hypothesis space constructed by *mode declaration M*, which is specified by *language bias* of the learning task. Mode declaration specifies what can occur in a hypothesis by specifying the predicates, and consists of two parts: *modeh* and *modeb*. *modeh* and *modeb* are the predicates that can occur in the head of the rule and body of the rule respectively.

Definition 2.16. Learning from Answer Sets (LAS)
Given a learning task T, the set of all possible inductive solutions of T, denoted $ILP_{\text{LAS}}(T)$, and a hypothesis H is an inductive solution of $ILP_{\text{LAS}}(T)\langle B, S_{\text{M}}, E^+, E^-\rangle$ such that:

1. $H \subseteq S_{\text{M}}$
2. $\forall e^+ \in E^+ : \exists A \in AS(B \cup H) | A$ extends e^+
3. $\forall e^- \in E^- : \nexists A \in AS(B \cup H) | A$ extends e^-

Inductive Learning of Answer Set Programs (ILASP)

Inductive Learning of Answer Set Programs (ILASP) is an algorithm that is capable of solving a LAS task $ILP_{\text{LAS}}(T)\langle B, S_{\text{M}}, E^+, E^-\rangle$. It is based on two fundamental concepts: *positive solutions* and *violating solutions*.
A hypothesis H is a positive solution if and only if

1. $H \subseteq S_{\text{M}}$
2. $\forall e^+ \in E^+ \exists A \in AS(B \cup H) | A$ extends e^+

A hypothesis H is a violating solution if and only if

1. $H \subseteq S_{\text{M}}$
2. $\forall e^+ \in E^+ \exists A \in AS(B \cup H) | A$ extends e^+
3. $\exists e^- \in E^- \exists A \in AS(B \cup H) | A$ extends e^-

Given both definitions of positive and violating solutions, $ILP_{\text{LAS}}\langle B, S_{\text{M}}, E^+, E^-\rangle$is positive solutions that are not violating solutions [8].

Context-dependent Learning from Answer Sets

Context-dependent learning from answer sets ($ILP^{context}_{LAS}$) is a further generalisation of ILP_{LAS} with *context dependent examples*[17][2]. *Context-dependent examples* are examples allow each example to have it's own extra background knowledge, which applies only to each specific example called *context*. This way the background knowledge is more structured rather than one fixed background knowledge that is applied to all examples.
Formally, *context-dependent partial interpretation (CDPI)* is of the form:

$$\langle e, C\rangle \tag{2.9}$$

where *e* is a partial interpretation and C is a context.

[2]The original paper developed the framework called *learning from ordered answer sets ($ILP^{context}_{LOAS}$)*. In this project, we do not use ordered answer sets and therefore simplify it to $ILP^{context}_{LAS}$.

Definition 2.17. $ILP^{context}_{LAS}$ task is of the form $T = \langle B, S_{\text{M}}, E^{+}, E^{-} \rangle$. A hypothesis H is an inductive solution of T if and only if:

1. $H \subseteq S_{\text{M}}$
2. $\forall \langle e, C \rangle \in E^{+}, \exists A \in AS(B \cup C \cup H) | A$ extends e
3. $\forall \langle e, C \rangle \in E^{-}, \nexists A \in AS(B \cup C \cup H) | A$ extends e

Context dependent examples provide more structured organisation of the background knowledge in order to improve efficiency of the learning algorithm.

2.3 Reinforcement Learning (RL)

Reinforcement learning (RL) is a subfield of machine learning regarding how an agent behaves in an environment in order to maximise its total reward. As shown in Figure 2.1, the agent interacts with an environment, and at each time step t the agent takes an action and receives observation, which affects the environment state and the reward (or penalty) it receives as the action outcome. In this section, we briefly introduce the background of RL necessary for our new framework as well as a benchmark of our evaluation discussed in Chapter 4.

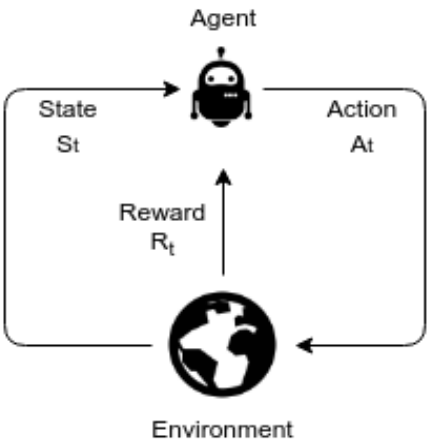

Figure 2.1: The interaction between an agent and an environment

2.3.1 Markov Decision Process (MDP)

An agent interacts with an environment in a sequence of discrete time steps, which is part of the sequential history of observations, actions and rewards. The sequential history is formalised as

$$H_{\text{t}} = S_1, R_1, A_1, ..., A_{\text{t-1}}, S_{\text{t}}, R_{\text{t}}. \tag{2.10}$$

where S is *states*, R is *rewards* and A is *actions*. A state S_{t} determines the next environment and has a *Markov property* if and only if

$$\mathbb{P}[S_{\text{t+1}} | S_{\text{t}}] = \mathbb{P}[S_{\text{t+1}} | S_1, ..., S_{\text{t}}]. \tag{2.11}$$

In other words, the probability of reaching S_{t+1} depends only on S_t, which captures all the relevant information from the earlier history [18]. When an agent must make a sequence of decision, the sequential decision problem can be formalised using the *Markov decision process (MDP)*. MDP formally represents a fully observable environment of an agent for RL.

Definition 2.18. Markov Decision Process (MDP)
Markov decision process (MDP) is defined in the form of a tuple $\langle S, A, T, R \rangle$ where:

- S is the set of finite states that is observable in the environment.
- A is the set of finite actions taken by the agent.
- T is *a state transition function*: $S \times A \times S \rightarrow [0, 1]$, which is a probability of reaching a future state $s' \in S$ by taking an action $a \in A$ in the current state $s \in S$.
- R is a reward function $R_a(s, s') = \mathbb{P}[R_{t+1}|S_{t+1} = s', A_t = a, S_t = s]$, the expected immediate reward that action *a* in state *s* at time *t* will return.

The objective of an agent is to solve an MDP by taking a sequence of actions to maximise the total cumulative reward it receives. A method of finding the optimal solution of an MDP is Reinforcement Learning (RL). Formally, the objective of a RL agent is to maximise *expected discounted return* over time steps *k*,which is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.12}$$

where γ is a discount rate $\gamma \in$ [0,1], a parameter which represents the preference of the agent for the present reward over future rewards. If γ is low, the agent is more interested in maximising immediate rewards.

2.3.2 Policies and Value Functions

Most RL algorithms are concerned with estimating *Value functions*. Value functions estimate the expected return, or discounted cumulative future reward, for a given action in a given state. The expected reward for an agent is dependent on the agent's action. The state value function $v_\pi(s)$ of an MDP under a *policy* π is the expected return starting from state *s*, which is of the form:

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] \text{ for all } s \in \mathcal{S} \tag{2.13}$$

The optimal state-value function $v^*(s)$ maximises the value function over all policies in the MDP, which is of the form:

$$v^*(s) = \max_\pi v_\pi(s) \tag{2.14}$$

Value functions are defined by *policies*, which map from states to probabilities of choosing each action. A policy π is defined as follows:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{2.15}$$

While the value functions estimate the value of a particular state, the value of taking an action a given a particular state s is similarly estimated by an *state-action value function*, or *Q-function*, under a policy π. State-action value function is defined as follows:

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \text{ for all } s \in \mathcal{S} \tag{2.16}$$

An optimal policy achieves the optimal state-action value functions, and it can be computed by maximising over the optimal state-action value functions. The optimal state-action-value function $q^*(s,a)$ maximises the state-action value functions over all policies in the MDP, which is of the form:

$$q^*(s,a) = \max_{\pi} q_{\pi}(s,a) \tag{2.17}$$

2.3.3 Temporal-Difference (TD) Learning

One of the RL approaches for solving a MDP is called *Temporal-Difference (TD) Learning*. TD learning is an online model-free RL which learns directly from episodes of incomplete experiences without a model of the environment. TD learning updates the estimates of value function as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.18}$$

where $R_{t+1} + \gamma\ V(S_{t+1})$ is the target for TD update, which is a biased estimate of $v_{\pi}\ (S_t)$, and $\delta = R_{t+1} + \gamma\ V(S_{t+1})$ - $V(S_t)$ is called *TD error*, which is the error in $V(S_t)$ available at time t+1. Since the TD learning only needs to know the estimate of one step ahead and does not need the final outcome of the episodes (also called TD(0)), it can learn online after every time step. TD learning also works without the terminal state, which is the goal for an agent. TD(0) is proved to converge to v_{π} in the table-based case [19].
Q-learning is off-policy TD learning introduced in [20], where the agent only knows about the possible states and actions. The transition states and reward probability functions are unknown to the agent. The update rule for the state-action value function, called *Q function*, is of the form:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(R_{t+1} + \gamma\ max(a+t)\ Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{2.19}$$

where α is a constant step-size parameter, or learning rate, $\alpha \in [0,1]$, and γ is a discount rate. The function $Q(S,A)$ predicts the best action A in state S to maximise the total cumulative rewards. Q-learning is off-policy because a policy being evaluated and improved is different from the policy being used to take an action.

Algorithm 1 Q-learning for leaning Q

1: **procedure**
2: Initialise $Q(s,a)$ arbitrarily
3: **repeat** for each episode
4: Choose a from s using policy derived from Q
5: Take action a, observe r, s'
6: $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(R_{t+1} + \gamma\ max(a+t)\ Q(s_{t+1},a_{t+1}) - Q(s_t,a_t))$
7: $s \leftarrow s'$
8: **until** s is terminal **return** Q

Q-learning algorithm is defined in Algorithm 1. Since the state-action value function Q learnt by Q-learning directly approximates q*, Q-learning is guaranteed to converge to an optimal policy in a finite tabular representation [19]. Since Q-learning is one of the most widely used RL methods and our experiments are conducted in a tabular representation, we use it as one of the benchmarks for our new framework.

2.3.4 Model-based and Model-free RL

A *model* is a representation of an environment that an agent can used to understand how the environment should look like, which is a state transition function T and a reward function R in MDP. There are two different types of RL methods: *model-based* and *model-free* RL method. Model-based RL is where, when a model of the environment is known, the agent uses the model to plan for a series of actions to achieve the agent's goal. The model itself can be learnt by interacting with the environment by taking actions, which return states and rewards, and the parameters of the action models can be estimated with maximum likelihood methods [21]. Using a model, an agent can do planning to generate or improve a policy for the modelled environment.
By contrast, model-free RL is where a model is unknown and the agent learns to achieve the goal solely by interacting with the environment without using the model. Thus the agent knows only possible states and actions, and the transition state and reward probability functions are unknown. When the model of the environment is correct, the model-based RL agent can reach an optimal policy without trial-and-error interactions with the environment. When the model is not a true representation of the environment, or the true MDP, however, the planning algorithms will not lead to the optimal policy, but a sub-optimal policy.
The model-based learning is a core foundation of our framework, and our framework attempts to learn a state transition function using a ILP framework.

2.3.5 Transfer Learning

Transfer learning is where knowledge learnt in one or more tasks can be used to learn a new task better than if without the knowledge in the first task [22]. Since training a RL algorithm tends to be time consuming and computationally expensive, and transfer learning mitigates the problems by applying the trained agent in a different setting. While transfer learning is an active research area in machine learning, and is particularly important in RL, since most of the RL research has been done in a simulation or game scenarios, and training RL models in a real physical environment is more expensive. Even in virtual environments like games, the transfer learning between different tasks will greatly impact potential applications. With transfer learning, the agent requires less time to learn a new task with the help of what was learnt in previous tasks.

Chapter 3

Framework

This chapter introduces a new proof-of-concept framework called *ILP(RL)*, a new framework for learning a state transition function using inductive learning with ILASP and planning with ASP. The development of this framework is one of the main objectives of this project and we explain the framework in details in this chapter.

3.1 Overview

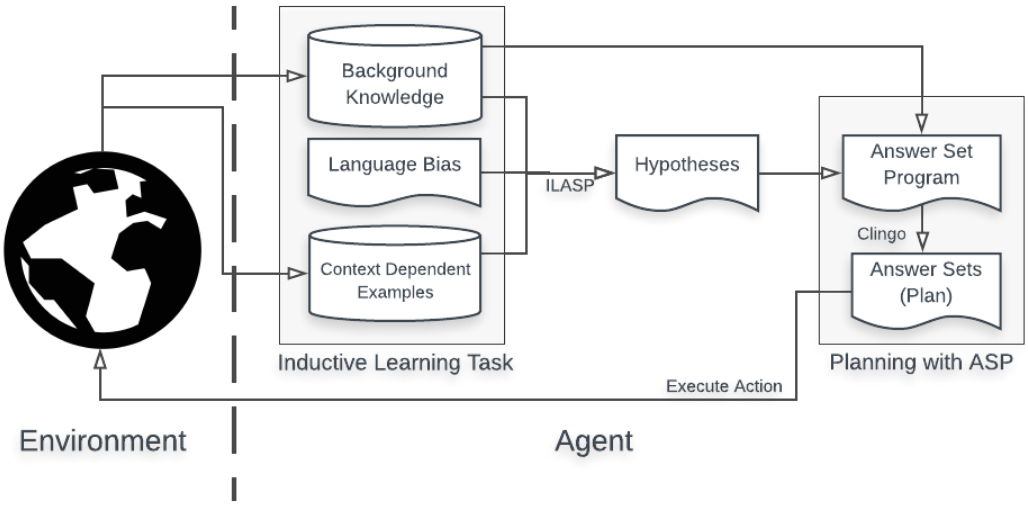

Figure 3.1: ILP(RL) overview

The overall architecture of ILP(RL), is shown in Figure 3.1. ILP(RL) mainly consists of two components: inductive learning with ILASP and planning with ASP.
The first step is inductive learning with ILASP. An agent interacts with an unknown environment and receives state transition experiences as context dependent examples. Together with pre-defined background knowledge and language bias, these examples are used to inductively learn and improve hypotheses iteratively, which are state transition functions in the environment.
The second step is planning with ASP. The interaction with the environment gives the agent information about the environment, such as locations of walls or a terminal state. The agent

remembers these information as background knowledge or context of the context dependent examples, and, together with the learnt hypotheses, uses them to make an action plan by solving an ASP program. The plan is a sequence of actions and it navigates the agent in the environment.
The agent iteratively executes this cycle in order to learn and improve the hypotheses as well as an action planning. The mechanisms of each step are explained in detail in the following sections.

3.2 Environment

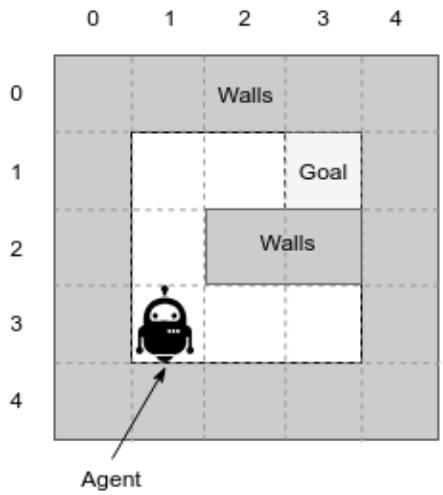

Figure 3.2: 5×5 grid maze example (environment)

Since there are no existing base frameworks for ILP(RL), our focus in this project is to develop a preliminary version of ILP(RL). Therefore, we use a very simple environment that allows us to see the potential of our proposed framework. The base environment is a simple grid maze, and we assume that the environment is a discrete deterministic environment.
We use a simple example to explain the environment as shown in Figure 3.2. States are expressed as X and Y coordinates. In Figure 3.2, for example, the agent is located at {X=1, Y=3}. The agent can take one of four possible actions each time: up, down right or left. Every time the agent takes an action, the agent receives the following experiences: a reward R_t, the next state S_{t+1} and surrounding information of the current state. The base environment mainly consists of three different elements: a goal, walls and paths. The goal cell called the *terminal state*. The agent receives a negative reward in any states except the terminal state. When the agent reaches the terminal state, the agent receives a positive reward and the current learning episode is complete. Since the agent's goal is to maximise the total estimated rewards over time at each episode, this goal is equivalent to finding the shortest path from a starting state to a terminal state.
We also assume that the agent can see adjacent vertical and horizontal, but not diagonal, states around the agent. This assumption allows a agent to learn a state transition in the environment using inductive learning. For example, the agent at {X=1, Y=3} can see that there are walls at {X=0, Y=3} and {X=1, Y=4}. More details of how to use this surrounding information are described in 3.3.

The applicability for a more complex environment is not considered in this project, and is discussed in Further Research in Section 6.2.

3.3 Inductive Learning Task

The first step is to construct a learning task using ILASP. The objective of the inductive learning is to learn state transitions in a given environment, which is used to generate an action plan at later steps. The target hypotheses are specified as follows:

```
state_after(V1):-adjacent(right, V0, V1), state_before(V1), action(right), wall(V0).
state_after(V0):-adjacent(right, V0, V1), state_before(V1), action(right), not wall(V0).
state_after(V1):-adjacent(left, V0, V1), state_before(V1), action(left), wall(V0).
state_after(V0):-adjacent(left, V0, V1), state_before(V1), action(left), not wall(V0).
state_after(V1):-adjacent(down, V0, V1), state_before(V1), action(down), wall(V0).
state_after(V0):-adjacent(down, V0, V1), state_before(V1), action(down), not wall(V0).
state_after(V1):-adjacent(up, V0, V1), state_before(V1), action(up), wall(V0).
state_after(V0):-adjacent(up, V0, V1), state_before(V1), action(up), not wall(V0).
```
(3.1)

where state_after is the next state S_{t+1}, and state_before is the current state S_t. action is an action A_t taken by the agent. adjacent(D, V0, V1) specifies that V0 is next to V1 in the direction of D. For example, adjacent(right, V0, V1) means V0 is immediately adjacent to V1. wall(V) and not wall(V) specify whether a state is a wall or not wall in a state V respectively. We describe the details of how to construct the background knowledge, context dependent examples and the language bias in the following sections, and these are the necessary components for the agent to learn the target hypotheses.

3.3.1 Background Knowledge

First we define the necessary background knowledge for the inductive learning. In order to learn the state transition for each direction as shown in 3.1, we need to define the meaning of "being next to" a state. These rules are defined as *adjacent*, which are of the form:

```
adjacent(right, (X+1,Y),(X,Y)):-cell((X,Y)), cell((X+1,Y)).
adjacent(left,(X,Y), (X+1,Y)):-cell((X,Y)), cell((X+1,Y)).
adjacent(down, (X,Y+1),(X,Y)):-cell((X,Y)), cell((X,Y+1)).
adjacent(up, (X,Y), (X,Y+1)):-cell((X,Y)), cell((X,Y+1)).
```
(3.2)

where cell corresponds to a state, and X and Y represent x-coordinate and y-coordinate. The rules 3.2 are given as background knowledge and allow the agent to understand the relation between two adjacent states. *cell((X,Y))* is defined as follows:

```
cell((0..X, 0..Y)).
```
(3.3)

where X and Y are the width and height of an environment respectively, they specify the size of an environment. For example, the grid maze shown in Figure3.2 has both height and width of 4, thus the type cell is defined in the background knowledge as cell((0..4, 0..4)).

3.3.2 Context Dependent Examples

Context dependent examples contain actual state transition that the agent gains by interacting with an environment. Since all the interactions with the environment are examples of valid moves, they are used as positive examples. A positive example is expressed in the following ASP form:

$$\#\mathsf{pos}(\{e^{\text{inc}}_{\text{ILP(RL)}}\}, \{e^{\text{exc}}_{\text{ILP(RL)}}\}, \{C_{\text{ILP(RL)}}\}) \tag{3.4}$$

It is equivalent to context dependent partial interpretation (CDPI) in $ILP^{context}_{LAS}$. As defined in the Equation 2.9, CDPI is of the form $\langle E, C\rangle$ where $E = \langle e^{\text{inc}}, e^{\text{exc}}\rangle$. Each of the component in CDPI in ILP(RL) is defined as follows:

Definition 3.1. $C_{\text{ILP(RL)}}$ contains an action a_t, the current state s_t, and adjacent walls of s_t.

Definition 3.2. $e^{\text{inc}}_{\text{ILP(RL)}}$ includes $s_{\text{t+1}} \in S$ such that:

- $\text{s}_{\text{t+1}} = \overset{*}{\text{s}}_{\text{t+1}}$
- $\forall A \in AS(B \cup H_\text{t} \cup C_{\text{ILP(RL)}})|\ \text{s}_{\text{t+1}} \notin A$

Definition 3.3. $e^{\text{exc}}_{\text{ILP(RL)}}$ includes $s_{\text{t+1}} \in S$ such that:

- $s_{\text{t+1}} \neq \overset{*}{s}_{\text{t+1}}$
- $\exists A \in AS(B \cup H_\text{t} \cup C_{\text{ILP(RL)}})|\ \text{s}_{\text{t+1}} \in A$

where $\overset{*}{s}_{\text{t+1}}$ is the true next state where the agent is at $t+1$, B is the background knowledge, H_t is the hypotheses at t and S is all the states in the environment.

Example 3.3.1. (Context dependent examples).

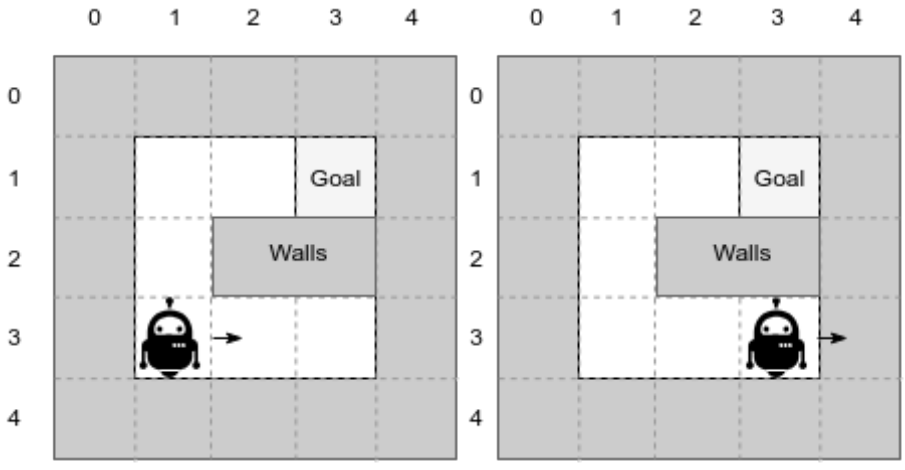

Figure 3.3: 5×5 grid maze example (context dependent example)

We use a simple 5×5 grid maze environment to highlight how an agent gains a positive example. Suppose $H_\text{t} = \emptyset$ and an agent takes an action "right" to move from state $(1,3)$ to $(2,3)$, as shown on the left in Figure 3.3. a_t is "right", s_t is $(1,3)$ and $\overset{*}{s}_\text{t}$ is $(2,3)$. According to Definition 3.2, $A = \emptyset$ since $H_\text{t} = \emptyset$, thus $\mathsf{state\_after}((2,3))$ is in $E^{\text{exc}}_{\text{ILP(RL)}}$, $A = \emptyset$ is there is no

exclusions. $C_{\text{ILP(RL)}}$ includes a_{t}, s_{t} and adjacent walls of s_{t}. The following positive example is generated:

$$\begin{aligned}&\#\text{pos}(\{\text{state\_after}((2,3))\},\\&\quad\{\},\\&\quad\{\text{state\_before}((1,3)).\ \text{action}(\text{right}).\ \text{wall}((0,\ 3)).\ \text{wall}((1,\ 4)).\})\end{aligned} \tag{3.5}$$

The next example illustrates a case where the agent tries to move to a state where a wall is. As shown on the right in Figure 3.3, the agent is at $(3, 3)$ and tries to move to a state $(3, 4)$ by taking an action "right", as shown on the left in Figure 3.3. In this case, however, there is a wall at $(4, 3)$ and therefore the agent cannot go to that state. Because of the blocking wall, $s_{\text{t}} = {s^*}_{\text{t+1}} = (3, 3)$. As in the previous case, suppose $H = \emptyset$ and therefore $A = \emptyset$.
From this example, the following positive example is generated:

$$\begin{aligned}&\#\text{pos}(\{\text{state\_after}((3,3))\},\\&\quad\{\}\\&\quad\{\text{state\_before}((3,3)).\ \text{action}(\text{right}).\ \text{wall}((3,2)).\text{wall}((4,3)).\ \text{wall}((3,4)).\}).\end{aligned} \tag{3.6}$$

3.3.3 Language Bias

We now define a search space S_M using a language bias specified by *mode declaration M*. Recall that $H \subseteq S_\text{M}$ for $ILP^{context}_{LAS}$, thus in order to learn the target hypotheses, S_M is specified as follows:

$$\begin{aligned}&\#\text{modeh}(\text{state\_after}(\text{var}(\text{cell}))).\\&\#\text{modeb}(1,\ \text{adjacent}(\text{const}(\text{action}),\ \text{var}(\text{cell}),\ \text{var}(\text{cell})),\ (\text{positive})).\\&\#\text{modeb}(1,\ \text{state\_before}(\text{var}(\text{cell})),\ (\text{positive})).\\&\#\text{modeb}(1,\ \text{action}(\text{const}(\text{action})),(\text{positive})).\\&\#\text{modeb}(1,\ \text{wall}(\text{var}(\text{cell}))).\end{aligned} \tag{3.7}$$

where #modeh and #modeb are the *normal head declarations* and the *body declarations*. The first argument of each #modeb specifies the maximum number of times that #modeb can be used in each rule (also called *recall*) [23], which we specify as 1 for #modeb in ILP(RL). var(t) is a placeholder for a variable of *type* t. In Equation 3.7, we use cell for the variable type, which is grounded using cell specified in Equation 3.3 in the background knowledge. const(t) is a placeholder for a constant term of type t, and type t must be specified as #constant(t, c),where c is a constant term, const(t) is specified as follows:

$$\begin{aligned}&\#\text{constant}(\text{action},\ \text{right}).\\&\#\text{constant}(\text{action},\ \text{left}).\\&\#\text{constant}(\text{action},\ \text{down}).\\&\#\text{constant}(\text{action},\ \text{up}).\end{aligned} \tag{3.8}$$

action type is specified as a constant since ILP(RL) needs to learn a different hypothesis of state transition function for each direction based on the context dependent examples that the agent collects.

(positive) in #modeb specifies that the body predicates only appear as positive and not negation as failure, which reduces the search space. We use (positive) for all #modeb except wall. Thus wall(var(cell)) could appear as not wall in a hypothesis, and all other body predicates should only be positive.
Finally, we define #max_penalty to specify the total number of atoms that appear in the whole hypothesis. By default it is 15, however the size of our target hypotheses defined in Equation 3.1 is larger than 15, and thus #max_penalty is increased to 50. Increasing #max_penalty allows ILASP to learn longer hypotheses at the expense of longer computation.

3.3.4 Hypothesis

Having defined the $ILP^{context}_{LAS}$ task $T = \langle B, S_{\mathrm{M}}, E^{+}, E^{-} \rangle$, ILP(RL) is able to learn hypotheses H. Since B and S_{M} are fixed, the hypotheses vary based on context dependent examples that the agent accumulates by interacting with an environment. For example, in the early phase of learning, the agent does not have many positive examples, and learns hypotheses that is a subset of the full target hypotheses that the agent could learn in the environment. Thus ILP(RL) needs to improve the hypotheses iteratively as illustrated in Figure 3.1. Inductive learning is executed when the current hypotheses do not cover the new context dependent example that the agent gains. If the current hypotheses cover the new positive example, there is no need to re-execute ILASP.
The learnt hypotheses will be used for ASP planning, which we describe in the next section.

Example 3.3.2. (Hypothesis).

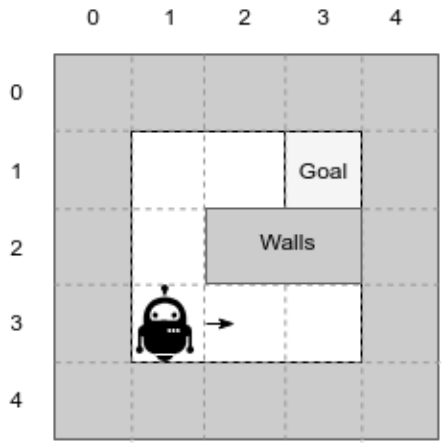

Figure 3.4: 5×5 grid maze example (hypothesis)

Using the context dependent examples illustrated in Example 3.3, we explain how the agent learns hypotheses. Suppose that the agent takes an action ”right” and gains one positive example, as shown on the right in Example 3.3. The full learning task for this simple case is shown as follows:

Listing 3.1: Learning task example

```
% Background knowledge
cell((0..4, 0..4)).
adjacent(right,(X+1,Y),(X,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(left,(X,Y),(X+1,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(down,(X,Y+1),(X,Y)):-cell((X,Y)),cell((X,Y+1)).
```

```
adjacent(up,(X,Y),(X,Y+1)):- cell((X,Y)),cell((X,Y+1)).
% Context dependent examples
#pos({state_after((2,3))},
     {},
     {state_before((1,3)). action(right).
     wall((0,3)). wall((1, 4)).}).
% Language bias
#modeh(state_after(var(cell))).
#modeb(1, adjacent(const(action),
          var(cell),var(cell)),(positive)).
#modeb(1, state_before(var(cell)), (positive)).
#modeb(1, action(const(action)), (positive)).
#modeb(1, wall(var(cell))).

#max_penalty(50).

#constant(action, right).
#constant(action, left).
#constant(action, down).
#constant(action, up).
```

In the above learning task, ILASP learns the following hypothesis.

state_after(V0) :- adjacent(right, V0,V1).

The agent learnt a state transition based on the right adjacent. This hypothesis is subset of the target hypothesis. Because there is only one positive example, the learnt hypothesis is not likely to be useful to know a state transition regarding the right direction. ILP(RL) agent interacts more with the environment and continues to improve the hypotheses by collecting more positive examples, until the target hypotheses are learnt.

3.4 Planning with Answer Set Programming

The learnt hypotheses in the inductive learning phase are used to generate an action plan that the agent should follow. In the following subsections, we explain how to create an ASP program and use the answer sets by solving for the ASP to make an action plan in a maze environment.

3.4.1 Answer Set Program

The ASP program should be constructed such that answer sets of the ASP program are a sequence of actions and states at each time step in ASP syntax. The ASP program for ILP(RL) is summarised in Listing 3.2:

Listing 3.2: Answer set program for ILP(RL)

```
% Hypotheses are given from ILASP.

% Choice rule for choosing an action at each time T.
1{action(down,T);
```

```
  action(up,T);
  action(right,T);
  action(left,T)}1 :-time(T), not finished(T).

% T_start is the current time step, T_max is the maximum time
    steps.
% The agent can do a planning between these time steps.
time(T_start..T_max).

% Check whether the agent reaches a terminal state.
finished(T):- goal(T2), time(T), T >= T2.
goal(T):- state_at((X_terminal, Y_terminal), T),
          not finished(T-1).
goalMet:- goal(T).
:- not goalMet.

% Walls are cumulatively collected from
% context dependent examples.
wall((X_1, Y_1)).
wall((X_2, Y_2)).
...

% Current state of the agent at time T,
% which is the start of the planning
state_at((X_start, Y_start), T).

% The output of ASP should include only state_at and action
#show state_at/2.
#show action/2.

% Find a shortest path to a terminal state, thus
% minimise the number of actions to reach the terminal state.
#minimize{1, X, T: action(X,T)}.

% The size of the maze
cell((0..X_width, 0..Y_height)).

% Similar to ILASP, adjacent rules are given
adjacent(right,(X+1,Y),(X,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(left,(X,Y),(X+1,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(down,(X,Y+1),(X,Y)):-cell((X,Y)),cell((X,Y+1)).
adjacent(up,(X,Y),(X,Y+1)):-cell((X,Y)),cell((X,Y+1)).
```

First, we use the learnt hypotheses returned from ILASP as part of the ASP program. In the inductive learning phase in ILP(RL), we only need to differentiate between s_t and s_{t+1} as state_before and state_after respectively. For the planning with ASP, however, the answer sets contain a sequence of actions and states for more than two time steps, such as $s_t, s_{t+1}, s_{t+2}, \ldots$. In order to capture the notion of the time sequences, the ASP syntax of the hypotheses needs to be modified from inductive learning phase to ASP planning by adding

time *T*. Specifically, the following mapping is required between ILASP and ASP planning syntax.

No	Inductive learning syntax	ASP planning syntax
1	state_before(V)	state_at(V1, T)
2	state_after(V)	state_at(V1, T+1)
3	(empty body)	time(T)
4	action(A)	action(A, T)

Table 3.5: ASP syntax mapping between inductive learning and ASP planning

The conversion from (empty body) to time(T) means that the body of all the hypotheses includes time(T) in ASP planning syntax.

Example 3.4.1. (Mapping of ASP syntax between inductive learning and ASP planning). Suppose that an agent learnt the following hypothesis in the inductive learning phase.

state_after(V0) :- adjacent(right, V0, V1), state_before(V1), action(right), not wall(V0).

This hypothesis is converted into the following syntax for ASP planning.

state_at(V0, T+1) :- time(T), adjacent(right, V0, V1), state_at(V1, T), action(right, T), not wall(V0).

The converted hypothesis is used to generate a sequence of actions and states to the "right" direction for more than two time steps for ASP planning, given there is no wall in the right adjacent cell.

An action for the agent is given as a choice rule, which is of the form:

$$1\{\text{action(down,T); action(up,T); action(right,T); action(left,T)}\}1 \text{ :- time(T), not finished(T).} \tag{3.9}$$

The choice rule states that an action must be exactly one of four actions: down, up, right, or left at each time step T, as defined by the maximum and minimum integers 1. The choice rule specifies that one of four actions must be true unless not finished(T) or time(T) are satisfied, as defined in the body of the choice rule.Therefore there is always an action to be taken until the agent reaches a terminal state. When the agent reaches a terminal state, finished(T) is satisfied, otherwise time step T exceeds the maximum time step that is allocated to the agent.
The maximum time step is specified externally and is of the form:

$$\text{time}(T_{start}..T_{max}) \tag{3.10}$$

where T_{start} is the current time step, or starting time of planning, and T_{max} is the maximum time step. For example, if an agent is at time step 0, and can take actions up to 100 time step within an episode until it finds a terminal state, time is defined as time(0..100).
finished(T) determines whether the agent reaches the goal, which is specified as follows:

$$\begin{aligned}&\text{finished(T):- goal(T2), time(T), T} \geq \text{T2.}\\&\text{goal(T):- state\_at((}X_{terminal}\text{, }Y_{terminal}\text{), T), not finished(T-1).}\\&\text{goalMet:- goal(T).}\\&\text{:- not goalMet.}\end{aligned} \tag{3.11}$$

$\mathsf{state\_at}((\mathsf{X}_{\mathsf{terminal}}, \mathsf{Y}_{\mathsf{terminal}}))$ is the location of the terminal state, which is unknown to the agent before the learning starts. The agent explores the environment until it finds the terminal state. The terminal state is stored in the background knowledge.
Once the agent reaches the goal at time t and $\mathsf{finished(T)}$ is satisfied, there will not be any actions at time $t+1$ since the body of the choice rule defined in Equation 3.9 is not satisfied. The locations of walls are what the agent collects from the context of context dependent examples, which are specified as follows:

$$\mathsf{wall((X,Y))} \tag{3.12}$$

These wall information are accumulated as background knowledge for the ASP planning.
The starting state for the planning is provided as part of ASP. It is the current location of the agent at time T when the plan is generated. It is specified as follows:

$$\mathsf{state\_at}((\mathsf{X}_{\mathsf{start}}, \mathsf{Y}_{\mathsf{start}}), \mathsf{T}) \tag{3.13}$$

In addition, the definitions of adjacent and cell type are also provided, and are the same as the definition of adjacent in inductive learning phase defined in Equation 3.2 and 3.3.
In addition, we need to incorporate a notion of rewards. Instead of maximising the total rewards, which is the objective of most RL methods, we assume that the rewards for any states except the terminal state in an environment are -1, so that we can use an optimisation statement in ASP to minimise the weighted sum of the atoms in the answer sets. Because of this assumption, the maximising the total reward is equivalent to using #minimize. The optimisation statement in ILP(RL) is specified as follows:

$$\#\mathsf{minimize}\{1, \mathsf{X}, \mathsf{T}: \mathsf{action(X,T)}\}. \tag{3.14}$$

While this assumption limits the RL problems that ILP(RL) could solve, this project is a preliminary research of using a ASP-based ILP in RL problems. This assumption is further discussed in Section 6.2.
Finally, we are only interested in a sequence of actions and corresponding states as the output of the ASP program. Clingo, a ASP solver, can selectively include the atoms of certain predicates in the output and hide other predicates. This is specified as follows:

$$\begin{aligned} &\#\mathsf{show\ state\_at}/2. \\ &\#\mathsf{show\ action}/2. \end{aligned} \tag{3.15}$$

This way the answer sets of the ASP program contain only state_at and action.

3.4.2 Plan Execution

Having defined the ASP program, we describe how to use the answer sets generated by solving the ASP program in order to execute actions. The output of an ASP program is the plan that the agent follows, and it is of the form:

$$\begin{aligned} &\mathsf{state\_at}((\mathsf{x_t},\mathsf{y_t}),\mathsf{t}),\ \mathsf{action}(\mathsf{a_t},\mathsf{t}), \\ &\mathsf{state\_at}((\mathsf{x_{t+1}},\mathsf{y_{t+1}}),\mathsf{t+1}),\ \mathsf{action}(\mathsf{a_{t+1}},\mathsf{t+1}), \\ &\mathsf{state\_at}((\mathsf{x_{t+2}},\mathsf{y_{t+2}}),\mathsf{t+2}),\ \mathsf{action}(\mathsf{a_{t+2}},\mathsf{t+2}), \\ &\ldots \\ &\mathsf{state\_at}(((\mathsf{x_{t+n}},\mathsf{y_{t+n}}),\mathsf{t+n}),\ \mathsf{action}(\mathsf{a_{t+n}},\mathsf{t+n}), \\ &\mathsf{state\_at}((\mathsf{x_{goal}},\mathsf{y_{goal}}),\mathsf{t+n+1}). \end{aligned} \tag{3.16}$$

where n is the number of time steps taken to reach the terminal state. action(A,T) tells the agent which action to take at each time. Given the correct answer set planning, the agent follows the plan and reaches the terminal state.
The correctness of the planning is based on the correctness of the hypotheses as well as the wall information in the agent's background knowledge. For example, if the agent does not have a hypothesis for how to move right, the ASP plan does not generate an answer set containing action(right). Similarly, if the agent has not seen enough surrounding walls, and one of the state(x,y) in the plan is a wall, the agent fails to follow the plan by being blocked by the wall. If this happens, either the current hypotheses were incomplete, or the agent needs more background knowledge. The agent iteratively improve the hypotheses by accumulating more context dependent examples, which also contains a new wall information. This way the ASP planning also improves and the agent is able to correctly navigate through an environment using ASP planning. This completes the whole cycle of ILP(RL) framework shown in Figure 3.1.

Example 3.4.2. (Answer Set Program and Plan Execution).

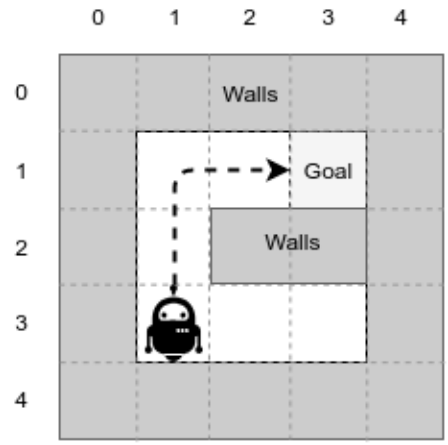

Figure 3.6: 5×5 grid maze example (ASP planning)

We use the same 5×5 grid maze as an example to highlight how the ASP planning works. Suppose that the agent is at (1,3) and knows the location of the terminal state (3,1). Thus the agent can make an action plan to reach the terminal state. The ASP program for this example is provided as follows:

Listing 3.3: Example of ASP program

```
% Learnt hypotheses
state_at(V0, T+1):-time(T),adjacent(right, V0, V1),
                   state_at(V1,T),action(right,T),not wall(V0).
state_at(V0, T+1):-time(T),adjacent(left, V0, V1),
                   state_at(V1,T),action(left,T), not wall(V0).
state_at(V0, T+1):-time(T),adjacent(down, V0, V1),
                   state_at(V1,T),action(down,T), not wall(V0).
state_at(V0, T+1):-time(T),adjacent(up, V0, V1),
                   state_at(V1,T),action(up,T), not wall(V0).

1{action(down, T);
  action(up, T);
```

```
  action(right, T);
  action(left, T)}1 :- time(T), not finished(T).

% The maximum time step is 4
time(0..4).

finished(T):- goal(T2), time(T), T >= T2.
goal(T):- state_at((3, 1), T), not finished(T-1).
goalMet:- goal(T).
:- not goalMet.

% Walls that the agent knows so far
wall((1, 0)).
wall((2, 0)).
wall((3, 0)).
wall((0, 1)).
wall((0, 2)).
wall((0, 3)).
wall((2, 2)).
wall((3, 2)).
wall((1, 4)).

% Starting state
state_at((1, 3), 0).

#show state_at/2.
#show action/2.
#minimize{1, X, T: action(X,T)}.

% Size of the maze
cell((0..4, 0..4)).

adjacent(right, (X+1,Y),(X,Y))   :- cell((X,Y)), cell((X+1,Y)).
adjacent(left,(X,Y),  (X+1,Y)) :- cell((X,Y)), cell((X+1,Y)).
adjacent(down, (X,Y+1),(X,Y))   :- cell((X,Y)), cell((X,Y+1)).
adjacent(up,   (X,Y),  (X,Y+1)) :- cell((X,Y)), cell((X,Y+1)).
```

The learnt hypotheses given by ILASP contain state transition for all four directions, and the agent has already seen necessary wall locations. The ASP program defined in Listing 3.3 returns the answer sets as a plan. By following the answer sets, the agent is able to reach the terminal state at (3,1).

state_at((1,3),0),action(up,0),
state_at((1,2),1),action(up,1),
state_at((1,1),2),action(right,2),
state_at((2,1),3),action(right,3),
state_at((3,1),4).

3.5 Exploration Policy

In this section, we explain the exploration policy of ILP(RL). While the agent can learn the hypotheses and execute a plan by interacting with the environment as shown in Figure 3.1, there are two main reasons why ILP(RL) requires exploration. First, the ILP(RL) agent can execute ASP planning only if it finds a terminal state. Therefore, the ILP(RL) agent needs to continue exploring the environment until the terminal state is found. Second, even if the terminal state is found and the agent generates a plan based on the hypotheses and background knowledge, the plan does not guarantee that the agent always finds an optimal policy. While the agent exploits what is already learnt and follows the best plan based on the current hypotheses and background knowledge, the agent also needs to explore the environment using its exploration policy, which is an action different from a plan in order to discover a new state, which might allow the agent to find an even shorter path and therefore receive higher total rewards in the long term.
One of the simple exploration policies in RL is called *ϵ-greedy policy*. ϵ-greedy policy states that an agent follows the optimal policy with probability of (1- ϵ) where $\epsilon \in [0, 1]$, and the agent takes a random action with probability of ϵ. ϵ is a hyper-parameter and chosen externally.
When the agent deviates from the planning by taking an random action and moves to a new state, the agent needs generate a new ASP plan from the new state. Since the ILP(RL) agent does not know when the optimal plan is found, it is necessary for ILP(RL) to continues the ϵ-greedy policy.

3.6 Implementation

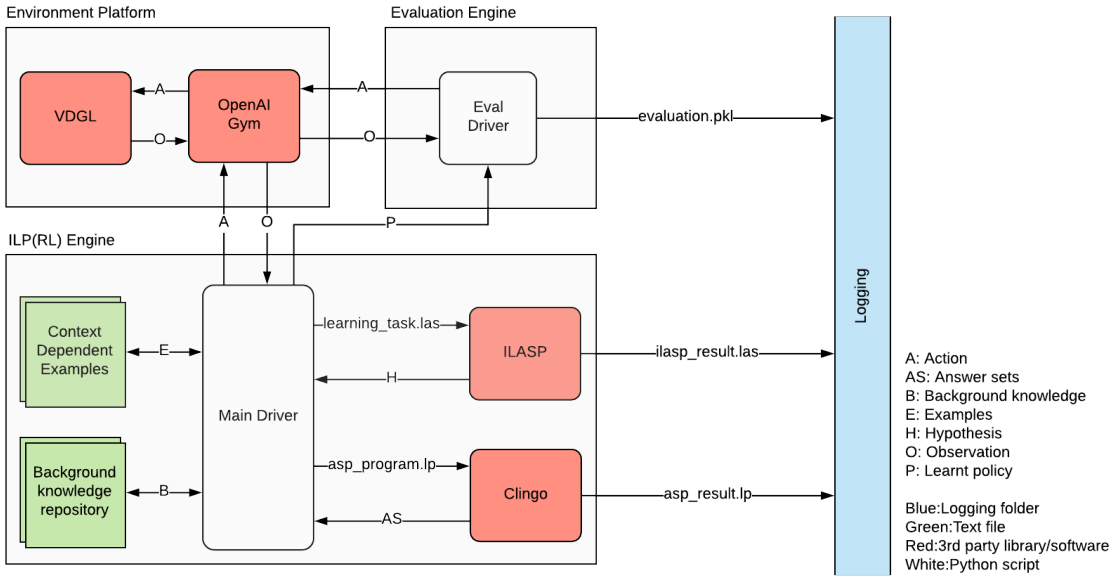

Figure 3.7: Data flow diagram for ILP(RL) engine, environment and evaluation components

We divide our implementation of the design outline into three separate software components: ILP(RL) Engine, Environment Platform and Evaluation Engine. The overview of our software implementation is shown in the data flow diagram in Figure 3.7.

ILP(RL) Engine is our main framework shown in Figure 3.1. As shown in Figure 3.7, the main driver is written in Python and executes the learning cycle shown in Figure 3.1. The main driver constructs the necessary files, such as a learning task and an ASP program, and handles the communications by sending input and output among third-party software and libraries as well as executing their programs: ILASP, Clingo and the environment platform including the Video Game Definition Language (VDGL) and OpenAI Gym. The main driver is also responsible for I/O operation for context dependent examples as well as background knowledge, both are being accumulated in text files.
Eval Driver is another Python script that executes the evaluation of ILP(RL) as well as a benchmark RL algorithm for evaluations. The details of the evaluation is described in the next chapter.
All the results of inductive learning with ILASP, ASP planning and evaluation are recorded in Logging folder as shown in Figure 3.7.
In the following sections, we explain how the main driver communicates with third-party softwaredetails.

3.6.1 Inductive Learning with ILASP

We use ILASP2i, an inductive learning system developed in [24]. ILASP2i is an iterative version of ILASP2, and is designed to scale with the number of examples. ILASP2i also introduces the use of context dependent examples. The latest ILASP available is ILASP3, which is designed to work on a noisy examples. Our context dependent examples do not contain noise since they are all true state transition that the agent has experienced by interacting with an environment, and therefore ILASP2i was sufficient for our implementation.
While ILASP2i is designed to work with a large number of examples, the inductive learning phase is the bottleneck of ILP(RL) framework in terms of computational time. Therefore we implemented a number of optimisations in order to reduce the computational time of inductive learning.
The first optimisation is the frequency of running ILASP. As described in Section 3.3.4, ILASP is called only if the current hypothesis does not cover the new positive example. This way each ILASP calls refines the hypotheses.
The next optimisation is through the command line. ILASP2i has a number of options, and we explain each option using the actual command we use to run ILASP in ILP(RL).

```
1    ILASP --version=2i FILENAME.las -ml=8 -nc --clingo5
2    --clingo "clingo5 --opt-strat=usc,stratify"
3    --cached-ref=PATH --max-rule-length=8
```

where,

- –version=2i specifies that we use ILASP2i.
- –ml=8 specifies the maximum number of body that each rule can have. The default length is 3.
- –nc means no constrains, and omits constrains from the search space. Since our target hypothesis is not a constraint, this option reduces the search space.
- –clingo5 –clingo "clingo5 –opt-strat=usc,stratify" generates a Clingo 5 program, which is the fastest Clingo, and specifies Clingo executable with the specified options. usc,

stratify is an unsatisfiable-core base optimisation with stratification using Gringo [25]. For details, see [26].

- –cached-ref=PATH enables the iterative mode, and keeps the output of the relevant example to a specified path, and start the learning from where it left before rather than going though all the examples.
- –max-rule-length=8 specifies the maximum number of literals in each rule of the hypothesis. The default maximum number is 5, and it is increased to 8.

3.6.2 Planning with ASP

Planning is computed using Clingo 5[1], and the actual execution command for ILP(RL) is as follows.

```
1    clingo5 --opt-strat=usc,stratify -n 0 FILENAME.lp
2    --opt-mode=opt --outf=2
```

- –clingo "clingo5 –opt-strat=usc,stratify" similar to ILASP, this option specifies clingo executable with the specified options.
- -n 0 -n is an abbreviation of models to specify the maximum number of answer sets to be computed. -n 0 means to compute all answer sets.
- –opt-mode=opt computes optimal answer sets.
- –outf=2 makes the output in. JSON[2] format.

3.6.3 Environment Platform

3.6.4 The Video Game Definition Language (VGDL)

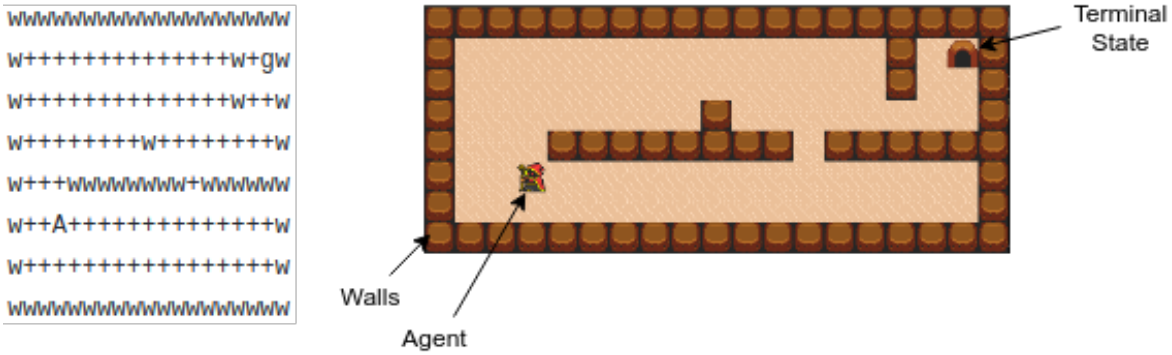

Figure 3.8: Map sketch of a VGDL game (left) and its high-level representation (right)

We use the Video Game Definition Language (VGDL), which is a high-level description language for 2D video games providing a platform for computational intelligence research [27].

[1] http://potassco.sourceforge.net/
[2] http://json.org/

The VGDL allows users to easily craft their own game environments, which makes it possible to do various experiments without relying on a default environment.
The base game we used to implement ILP(RL) is shown in Figure3.8. The map sketch is a plain text file and it is easy to modify the configuration of the game, and can be displayed as a high-level representation as visualisation.

Listing 3.4: VGDL description of a maze game

```
BasicGame
  SpriteSet
      floor > Immovable img=oryx/backBiege
      structure > Immovable
      goal  > color=GREEN img=oryx/door2
      avatar > MovingAvatar img=oryx/mage1
      wall > Immovable img=oryx/dirtWall_0 autotiling=True

  InteractionSet
      random wall structure     > stepBack
      avatar wall      > stepBack
      goal   avatar    > killSprite scoreChange=1
      avatar portalentry > teleportToExit

  TerminationSet
      SpriteCounter stype=goal   limit=0 win=True
      SpriteCounter stype=avatar limit=0 win=False

  LevelMapping
      g > floor goal
      w > floor wall
      A > floor avatar
      + > floor
```

The behaviours of the game can be specified using VGDL as shown in 3.4. All objects in the game can be described as sprites in the *SpriteSet*, where users can define the objects' properties. *InteractionSet* specifies the effects of objects when two objects interact in the game. *TerminationSet* specifies the conditions for ending the game. The representation of each object can be specified in *LevelMapping* and allows users to customise an original map [28].

3.6.5 OpenAI Gym

The VGDL platform provides an interface with OpenAI Gym [29], a commonly used benchmark platform for RL research. The communication between the VGDL environment and an agent is through the OpenAI Gym interface. 3.5 shows the functions provided by OpenAI Gym as well as the simple implementation for the interaction with an environment.

Listing 3.5: The OpenAI gym interface

```
import gym # import OpenAI gym package
import gym_vgdl # used to connect VGDL and OpenAI gym

```

```
num_episodes = 100 # num_episodes is specified by users
time_steps = 100 # time_steps is specified by users

# initialise an instance of a VDGL game
env = gym.make('VDGL_ENVNAME')

for i_episode in range(num_episodes):
    env.reset()  # the agent starts from a starting point
    for t in range(time_steps):
        action = 0 # an integer between 0 and 3
                   # and is chosen by your RL algorithm.
                   # Action 0: Up, 1: Down, 2: Left, 3: Right

        # take an action and get an observation
        next_state, reward, done, _ = env.step(action)
        env.render() # update the frame of the environment

        # when done is True, the agent is at a terminal state
        # and the current episode is finished
        if done:
            break
```

- env.reset() resets the game, and the agent restarts from the starting position. The function is called when the agent starts a new episode.
- env.step(action) returns an observation of taking an action, which include the state location of the agent in terms of the x and y coordinates, the reward of the state, and a Boolean value indicating whether the agent reaches an terminal state. The action is chosen by an RL algorithm of your choice. In the case of ILP(RL), action is chosen by the ASP planning or random exploration policy.
- env.render() renders one frame of the environment to visualise the movement of the agent in pygame.

Chapter 4

Evaluation

In this Chapter, we conducted four different evaluations in simple maze environments to investigate how the ILP(RL) agent learns and finds the optimal policy.

4.1 Experimental Setup

4.1.1 Evaluation Metrics

As introduced in Section 1.1, our motivation is to improve the learning efficiency and capability of transfer learning in RL. Therefore, these are the two main measurements for the performance of ILP(RL). The learning efficiency is measured in three different ways: performance of optimal policy, convergence of inductive learning and runtime of ILP(RL). Due to the random exploration for both ILP(RL) and a benchmark, the performance of each experiment varies. Particularly ASP planning of ILP(RL) starts only when the agent finds a terminal state and it is dependent on an random exploration policy. In order to smooth the impact of the randomness, we ran 30 experiments per evaluation and computed an average for all the evaluation metrics. Within an experiment, an agent is allocated 250 time steps per episode, 100 episode per experiment.
In order to compare the performance of ILP(RL) with an existing RL method, we use Q-learning as our base benchmark. Q-learning is widely used RL technique, and given the environments used for the experiments are a simple environment in that it is discrete and deterministic, this method is sufficient as a benchmark.

Performance of optimal policy

The performance of ILP(RL) is compared with a benchmark in terms of optimal policy, which is measured in terms of the total reward that an agent gains per episode by following its optimal policy. For all the evaluations, the agent receives -1 for any states except a terminal state and receives +10 when it reaches the terminal state. For example, if an agent requires the minimum 10 actions to get to a terminal state, the maximum total reward that the agent could gain per episode is 0 (-1 reward at each action +10 for reaching the terminal state).
Thus at after every episode, we disable an random exploration and inductive learning, and run the same experiment, so that the agent can only follow its optimal policy. In the case of ILP(RL), if the agent does not know a terminal state or hypothesis, it does not have any policy because ASP planning cannot be executed, and therefore the agent gets -250 total reward.

Convergence of Inductive Learning

The convergence of inductive learning is measured to see the learning curve of inductive learning phase in ILP(RL), which is specified as follows:

$$\frac{\text{The cumulative number of ILASP calls per time step}}{\text{The total number of ILASP calls in all episodes}} \in [0, 1] \tag{4.1}$$

This gives a normalised convergence rate of inductive learning with the maximum 1. For example, if the total number of ILASP calls in all episodes is 10 and there is a ILASP call at time step 1 episode 1, it is recorded as 0.1. When there is another ILASP call at time step 2 episode 1 after the first ILASP call, it is recorded as 0.2. The 10th ILASP call in this evaluation is recorded as 1.

Runtime

We recorded runtime of both ILP(RL) and a benchmark in each episode, and plot the cumulative runtime over episodes. For ILP(RL), we also recorded the average runtime of ILASP calls per evaluation. All the evaluations were conducted in Linux Operating System with Intel i7-6560U CPU and 8GB RAM.

4.1.2 Parameters

Parameter	ILP(RL)	Q-learning
The number of experiment per evaluation	30	30
The number of episode per evaluation	100	100
Time steps per episode	250	250
α (learning rate)	N/A	0.5
ϵ (epsilon)	0.1	0.1
Reward for any states except a terminal state	-1	-1
Reward for a terminal state	10	10

Table 4.1: List of parameters used in the evaluations

All the parameters used in the evaluations are summarised in Table 4.1. We trained the agents with maximum 250 time step, 100 episodes, and conduct the same experiment 30 times in each environment. The number of time steps should be sufficient for the both algorithms to reach a terminal state using the ϵ-greedy exploration strategy, in which we specify 250 time steps for all evaluations. The number of episode is specified such that both ILP(RL) and Q-learning eventually reaches the optimal policy within the specified number of episode. Every episode starts from a starting state. If the agent reaches the terminal state within 250 time steps, the episode is complete and the next episode starts with the fixed starting state.
The learning rate α for Q-learning, as shown in Equation 2.19, determines how much Q-value is updated each time. Since our environments are relatively simple, we use 0.5 for α. For the same reason, ϵ for both ILP(RL) and Q-learning is 0.1.
The rewards were assigned -1 for all states except the terminal state, and 10 for the terminal state.
We conducted four different evaluations using different environments to highlight each aspect of ILP(RL) algorithm.

4.2 Learning Evaluation

4.2.1 Evaluation 1: Baseline

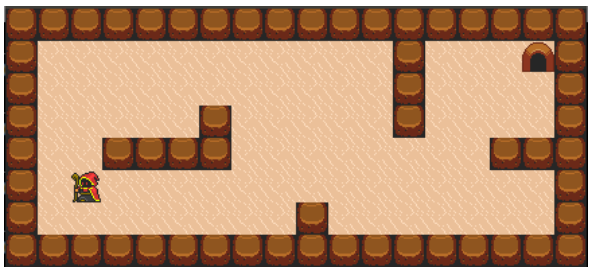

Figure 4.2: Game environment for Evaluation 1

The purpose of the first evaluation is to highlight how ILP(RL) agent learns the hypotheses using inductive learning and executes an ASP planning. The environment is a simple maze where the terminal state is located the right upper corner as shown in Figure 4.2.

4.2.2 Evaluation 1: Result

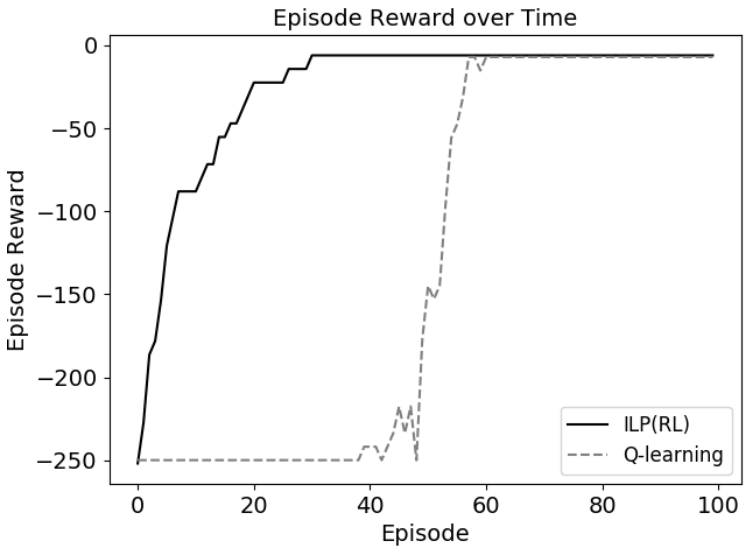

Figure 4.3: Evaluation 1: optimal policy

Figure 4.3 shows the performance of optimal policy. ILP(RL) reaches the optimal policy faster than Q-learning: ILP(RL) reaches the optimal policy at before 40 episode, whereas Q-learning reaches the optimal policy at around 60 episode. ILP(RL) learns the optimal policy at an earlier episode because once the terminal state is found, the planning is the correct path to the terminal state since there is only one way to reach the terminal state. The variations of the convergence to a optimal policy for ILP(RL) is dependent on how quickly the agent finds the terminal state. Since the exploration of ILP(RL) is based on ϵ-greedy policy with the current ILP(RL), this result shows that there is a potential that a better exploration policy further improves the ILP(RL) learning.

The convergence of Q-learning for reaching an optimal policy is steeper than that of ILP(RL). This is because the state-action value functions in Q-learning are updated with the rate of α, and the optimal policy for Q-learning is achieved when the optimal state-action value functions are achieved.
Overall this result shows that ILP(RL) converges to the optimal policy faster than Q-learning in this simple scenario, achieving more data-efficient learning.

Listing 4.1: Hypotheses learnt in Evaluation 1

```
state_after(V1):-adjacent(right, V0, V1), state_before(V1),
                    action(right), wall(V0).
state_after(V0):-adjacent(right, V0, V1), state_before(V0),
                    action(left), wall(V1).
state_after(V0):-adjacent(right, V0, V1), state_before(V1),
                    action(right), not wall(V0).
state_after(V0):-adjacent(left, V0, V1), state_before(V1),
                    action(left), not wall(V0).
state_after(V1):-adjacent(down, V0, V1), state_before(V1),
                    action(down), wall(V0).
state_after(V0):-adjacent(down, V0, V1), state_before(V1),
                    action(down), not wall(V0).
state_after(V0):-adjacent(up, V0, V1), state_before(V1),
                    action(up), not wall(V0).
state_after(V1):-adjacent(up, V0, V1), state_before(V1),
                    action(up), wall(V0).
```

In addition to the data-efficient learning, the hypothesis that the agent has learnt with ILP(RL) is expressive. Learnt hypotheses are shown in 4.1, which are state transition for all directions, both when an adjacent state is a wall or not wall. The learnt state transitions are easy to understand for human users, and general in that they can be applied to a different similar environment. The full learning task for this evaluation is in Appendix B.

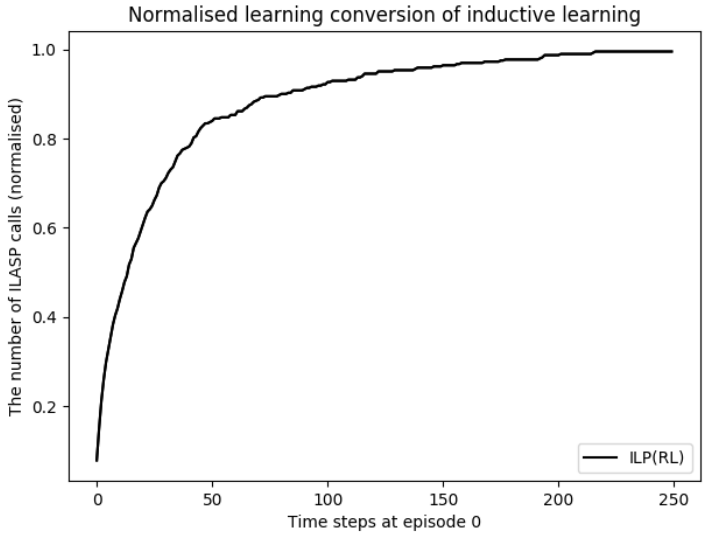

Figure 4.4: Normalised learning convergence by ILASP for experiment 1

In addition, we see how the agent learns the hypotheses over time. We plot part of the learning convergence for ILASP at episode 0 in Figure 4.4. It converges to 1.0 after 200 episode, and shows that the agent learns the target hypotheses at the episode 0. This shows that inductive learning of ILP(RL) happens at a very early stage of learning.
Finally, we compare the runtime of two algorithms. The Figure 4.5 shows that the runtime of ILP(RL) in the first few episodes is significantly high. This is due to the fact that ILP(RL) runs ILASP calls to learn the hypotheses at the beginning of episodes. The Figure 4.5 therefore highlights that inductive learning is likely the bottleneck in terms of computational time. This issue may not be critical in cases where the time of the time between the time steps is not an issue. If the performance is measured in terms of computation time rather than the number of iterations, ILP(RL) does not perform better than Q-learning. The average runtime of inductive learning is 5.58 seconds, and there are on average 12.83 instances of inductive learning per episode in this environment. The ASP planning is not a bottleneck of ILP(RL), but still takes longer time than Q-learning, as can be observed by the divergence of cumulative runtime between the two algorithms. This evaluation shows that while ILP(RL) learns faster than Q-learning in terms of the number of episodes, it suffers from increasing computational time due to inductive learning as well as ASP planning.

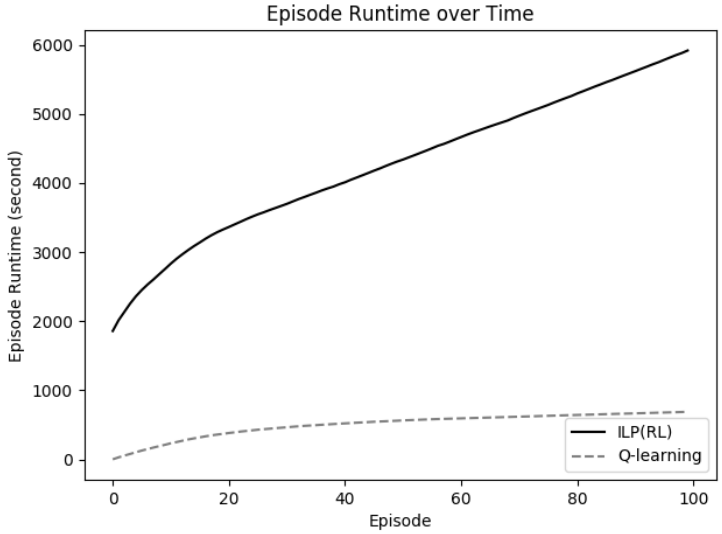

Figure 4.5: Evaluation 1: runtime comparison

4.2.3 Evaluation 2: Extended Baseline

Evaluation 2 was conducted to see if the agent learns a teleport and finds an optimal path using the teleport. The environment is the same as Evaluation 1 except the presence of teleport link and three extra walls to surround the destination teleport. In the environment shown in Figure 4.6, there are two ways to reach the goal: using a floor path to reach the goal located on the top right corner, or using the teleport. The environment is designed such that using the teleport is a shorter path and therefore gives a higher total reward. Compared to Evaluation 1, two extra language biases are added as follows:

```
#modeb(1, link_start(var(cell)), (positive)).
#modeb(1, link_dest(var(cell)), (positive)).
```

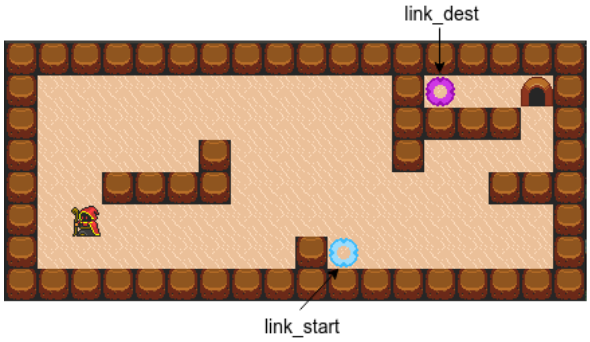

Figure 4.6: Game environment for Evaluation 2

link_start(var(cell)) is a state for departure of the teleport and link_dest(var(cell)) is the destination of the teleport. The teleport link is one-way: link_start takes the agent to link_dest, but link_dest does not take the agent back to link_start. This extra types allows ILASP to learn additional hypotheses. The full learning task for this evaluation is in Appendix C.
Also link_start and link_dest need to be stored in background knowledge rather than as context dependent examples when the agent finds them, because ILP(RL) needs to generate exclusions regarding the teleport link behaviour. The link locations need to be available for all positive examples so that ILASP correctly learns different a valid move for floor and teleport.
Because of the teleport link, the shortest path is 13 steps to reach the terminal state. Thus the maximum total reward that the agent could gain is -3.

4.2.4 Evaluation 2: Result

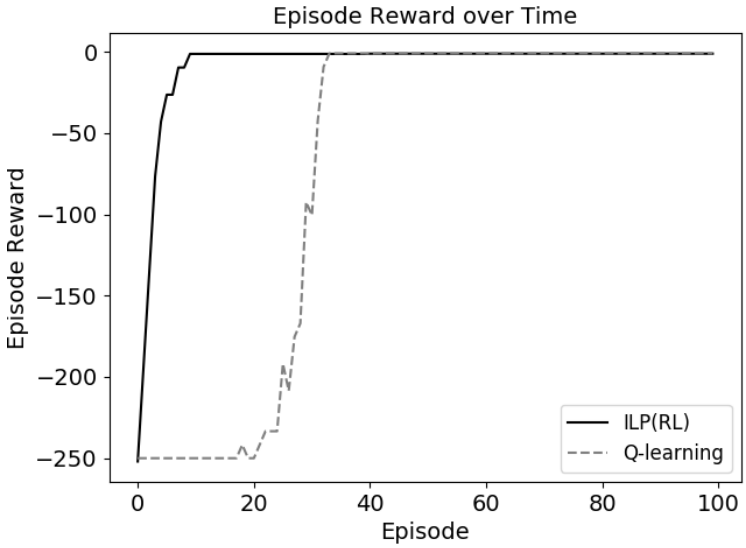

Figure 4.7: Evaluation 2: optimal policy

Similar to Evaluation 1, as shown in Figure 4.7, ILP(RL) finds an optimal policy faster than Q-learning. Compared to the environment in Evaluation 1, ILP(RL) finds an optimal policy at earlier episode, because finding a terminal state is easier for ILP(RL) in this environment,

since finding link_start immediately leads the agent to the terminal state rather than having to go through a floor path to the upper right corner of the environment.

Listing 4.2: Incomplete hypotheses for Evaluation 2

```
state_after(V1):-link_dest(V1).
state_after(V0):-link_dest(V0), state_before(V0),action(right).
state_after(V1):-adjacent(left, V0, V1), state_before(V0),
                 action(right), not wall(V1).
state_after(V0):-adjacent(left, V0, V1), state_before(V1),
                 action(left), not wall(V0).
state_after(V1):-adjacent(up, V0, V1), state_before(V0),
                 action(down), not wall(V1).
state_after(V0):-adjacent(up, V0, V1), state_before(V1),
                 action(up), not wall(V0).
state_after(V1):-adjacent(left, V0, V1), state_before(V1),
                 action(left), wall(V0).
state_after(V1):-adjacent(down, V0, V1), state_before(V1),
                 action(down), wall(V0).
state_after(V1):-adjacent(up, V0, V1), state_before(V1),
                 action(up), wall(V0).
```

To highlight the inductive learning process of the new concept of teleport link, Listing 4.2 is an intermediate incomplete hypotheses learnt by ILASP. These hypotheses are generated just after the agent steps onto the link_start. However, the first hypothesis in Listing 4.2 states that when link_dest is available state_after is true. Since link_dest is available in background knowledge rather than context in the context dependent examples, when solving for answer sets to generate a plan, it generates incorrect state_after at every time step.
However, as shown in Definition 3.3, these generated state_after are all incorrect and therefore will be added to exclusions of the next positive example. These exclusions will later refine hypotheses and the final complete hypotheses are shown in Listing 4.3.

Listing 4.3: Complete hypotheses for Evaluation 2

```
state_after(V1):-link_start(V0), link_dest(V1),
                 state_before(V0).
state_after(V1):-adjacent(left, V0, V1), state_before(V0),
                 action(right), not wall(V1).
state_after(V0):-adjacent(left, V0, V1), state_before(V1),
                 action(left), not wall(V0).
state_after(V1):-adjacent(up, V0, V1), state_before(V0),
                 action(down), not wall(V1).
state_after(V0):-adjacent(up, V0, V1), state_before(V1),
                 action(up), not wall(V0).
state_after(V0):-adjacent(left, V0, V1), state_before(V0),
                 action(right), wall(V1).
state_after(V1):-adjacent(left, V0, V1), state_before(V1),
                 action(left), wall(V0).
state_after(V0):-adjacent(up, V0, V1), state_before(V0),
                 action(down), wall(V1).
state_after(V1):-adjacent(up, V0, V1), state_before(V1),
```

```
18                    action(up), wall(V0).
```

Compared to the Evaluation 1, there are two new hypotheses due to the presence of the teleport links. These learnt hypotheses are also applicable to an environment where there is no teleport links, such as the environment in Evaluation 1. In this case, the first two hypotheses in Listing 4.3 are never be used since the body predicates relating to link_start(V0), link_dest(V1) are never be satisfied.

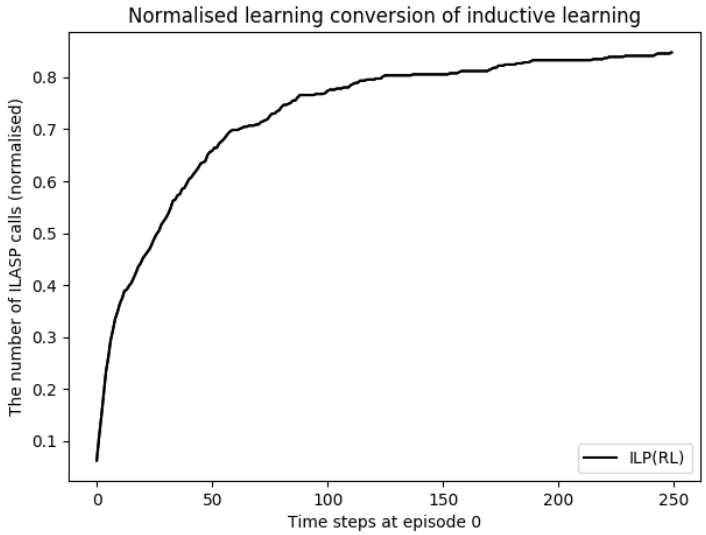

Figure 4.8: Normalised learning convergence by ILASP for Evaluation 2

Figure 4.8 shows the learning convergence of inductive learning at episode 0. Similar to Evaluation 1, most of ILASP calls occur at the beginning of the episode, as shown in Figure 4.9. The results of Evaluation 1 and 2 confirm that ILP(RL) learns state transition at the beginning of learning. If the agent finds the teleport link state at a later episode, ILP(RL) refines the hypotheses by learning a new state transition regarding the teleport link.

Figure 4.9 shows the runtime of ILP(RL) and Q-learning in Evaluation 2. Despite the fact that the size of the environment is the same as Experiment 1, there is a significant increase of runtime for ILP(RL) at the beginning of episode. This is because of the increase of search space in order for ILP(RL) to learn a new state transition regarding the teleport link. The average runtime of inductive learning is 95.47 seconds, and there are on average 16.23 instances of inductive learning per episode.

To highlight the increase of runtime, Table 4.10 summarises the comparison of runtime, ILASP calls and search space between Evaluation 1 and 2. Because of the two extra language biases for learning the teleport link, the search space in Evaluation 2 is significantly larger than that of Evaluation 1. This increase affects each ILASP call, resulting in much longer ILASP runtime in Evaluation 2. ILP(RL) calls ILASP an average of 3.4 more times in order to learn new hypotheses regarding the teleport link state.

While ILP(RL) still learns faster than Q-learning in terms of the number of iterations, the result of Experiment 2 shows that, the learning time per episode increases with respect to the size of search space, which corresponds to the number of state transition that the agent needs to learn in the environment.

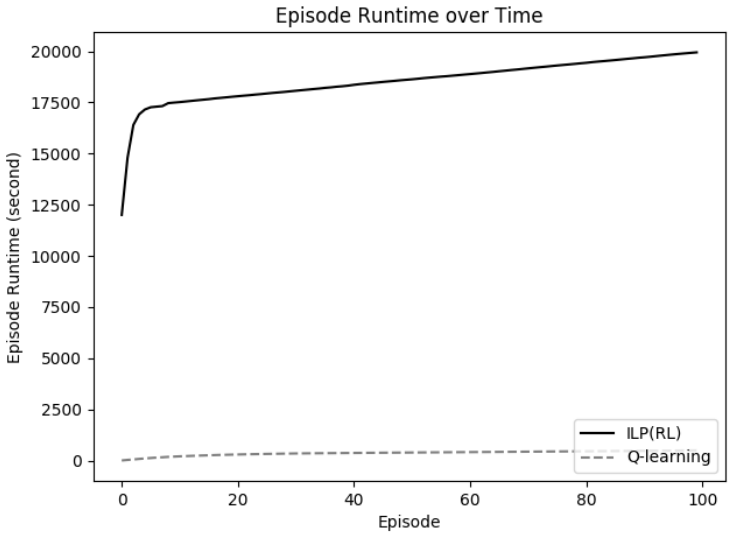

Figure 4.9: Evaluation 2: runtime comparison

Metrics	Evaluation1	Evaluation2
Average ILASP runtime time (seconds)	5.58	95.47
The number of ILASP calls	12.83	16.23
Search space	1690	32755

Table 4.10: Comparison of runtime, ILASP calls and search space between Evaluation 1 and 2

4.3 Transfer Learning Evaluation

4.3.1 Evaluation 3: Transfer Learning

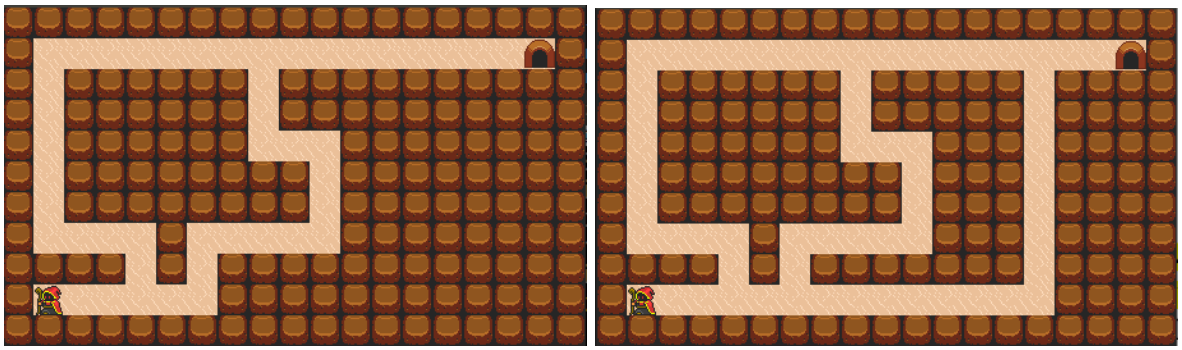

Figure 4.11: Game environment for Evaluation 3: before (left) and after (right) transfer learning

In Evaluation 3, we investigate the possibilities of transfer learning between similar environments. We trained the ILP(RL) agent using the environment on the left in Figure 4.11, and transfer the learnt hypotheses as well as context dependent examples to a new environment on the right in Figure 4.11. The terminal state is located at the same location, and there is an extra shorter path to the terminal state in the environment on the right of Figure 4.11. Context dependent examples are transferred to the new environment. This is because if there is a new hypothesis that the agent needs to learn in the new environment, ILP(RL)

needs to refine the hypothesis using these transferred context dependent examples. Thus all the context dependent examples are also transferred as well as the learnt hypotheses.
Background knowledge is not transferred since the wall locations are different in a new environment. The agent therefore starts the exploration of the new environment with an empty background knowledge and gradually collects them over time. The terminal state is the same as that in the first environment, but the shortest path to the goal is different between the two environments as new shorter path is introduced in the right environment in Figure 4.11.
For transfer learning evaluation, we use three agents for comparison as follows.

- Agent(TL): The agent with transferred hypotheses, examples and also remembers the location of the terminal state.
- Agent(noTL)$_{\text{Goal}}$: The agent with no transferred information, but knows the location of the terminal state.
- Agent(noTL)$_{\text{noGoal}}$: The agent with no transferred information, including the location of the terminal state.

Listing 4.4: Hypotheses for Evaluation 3

```
state_after(V0):-adjacent(right, V0, V1), state_before(V1),
                 action(right), not wall(V0).
state_after(V0):-adjacent(left, V0, V1), state_before(V1),
                 action(left), not wall(V0).
state_after(V1):-adjacent(down, V0, V1), state_before(V0),
                 action(up), not wall(V1).
state_after(V0):-adjacent(down, V0, V1), state_before(V1),
                 action(down), not wall(V0).
state_after(V1):-adjacent(right, V0, V1), state_before(V1),
                 action(right), wall(V0).
state_after(V1):-adjacent(left, V0, V1), state_before(V1),
                 action(left), wall(V0).
state_after(V0):-adjacent(up, V0, V1), state_before(V0),
                 action(down), wall(V1).
state_after(V1):-adjacent(up, V0, V1), state_before(V1),
                 action(up), wall(V0).
```

Listing 4.4 is the hypotheses that is transferred to a new environment, which is acquired by training the agent in the environment once on the left of Figure 4.11. The learnt hypotheses are the same hypotheses are that in Evaluation 1 shown in Listing 4.1 in the environment since the hypotheses are general state transition that work in any similar environments.

4.3.2 Evaluation 3: Result

The result is shown in Figure 4.12. For Agent(TL), since the complete hypotheses are already known to the agent as well as the terminal state, the agent can execute ASP planning from episode 0. There is no ILASP calls in the new environment since the transferred hypotheses are already complete and cover all the examples the agent encounters in the new environment. The only information required is background knowledge for the ASP planning, namely

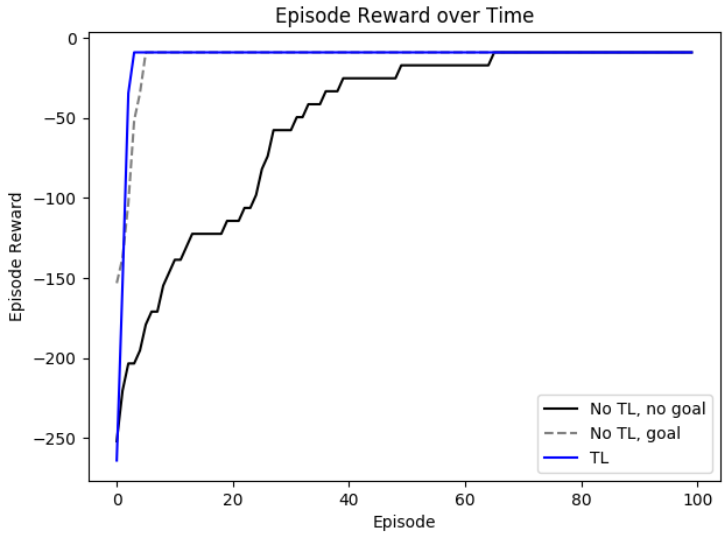

Figure 4.12: Evaluation 3: optimal policy

the locations of the walls, which are quickly acquired and reached the maximum total reward at the very beginning of episodes.
The next best agent in terms of convergence rate is Agent(noTL)$_{\text{Goal}}$. Since the terminal state is known to the agent, the agent can do the planning from episode 0. However, the agent needs to learn the hypothesis. The reason that the convergence rate is almost the same as that of Agent(TL) is that, the ILP(RL) learns the complete hypotheses at very early episode, as observed in Evaluation 1 and 2. Thus there is little difference between Agent(TL) and Agent(noTL)$_{\text{Goal}}$ in terms of finding an optimal policy, and information of the terminal state plays an important rule for ILP(RL).
Thus the transfer learning works particularly when the terminal state is transferred, because the ASP planning of ILP(RL) depends on the terminal state. In addition, this evaluation further confirms that there is a promising potential for improving the exploration strategy to find the terminal state as soon as possible.

4.3.3 Evaluation 4: Extended Transfer Learning

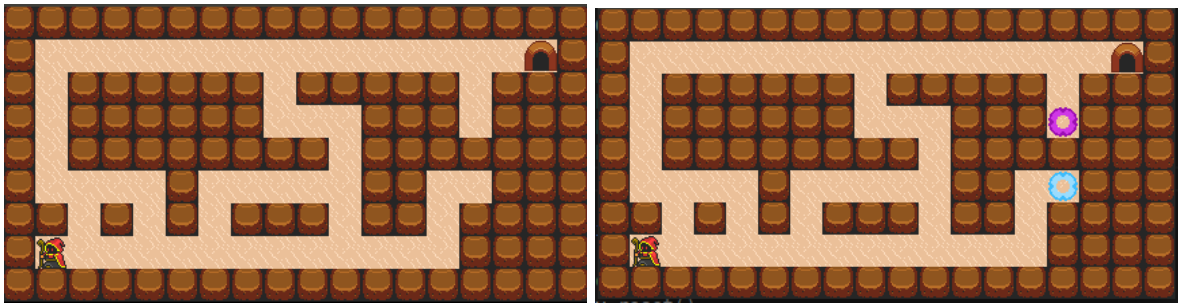

Figure 4.13: Game environment for Evaluation 4: before (left) and after (right) transfer learning

In Evaluation 4, we trained ILP(RL) on the left in Figure 4.13, and transferred context dependent examples as well as the learnt hypotheses. The objective of this experiment is to see

how the transferred agent learns a new state transition on top of the transferred hypotheses. In the new environment on the right of Figure 4.13, there is a teleport link and using the teleport is the shortest path to the terminal state. This is a new concept that did not exist in the trained environment and therefore the agent needs to learn it after the hypotheses are transferred. The same as Evaluation 3, we use four different agents.

4.3.4 Evaluation 4: Result

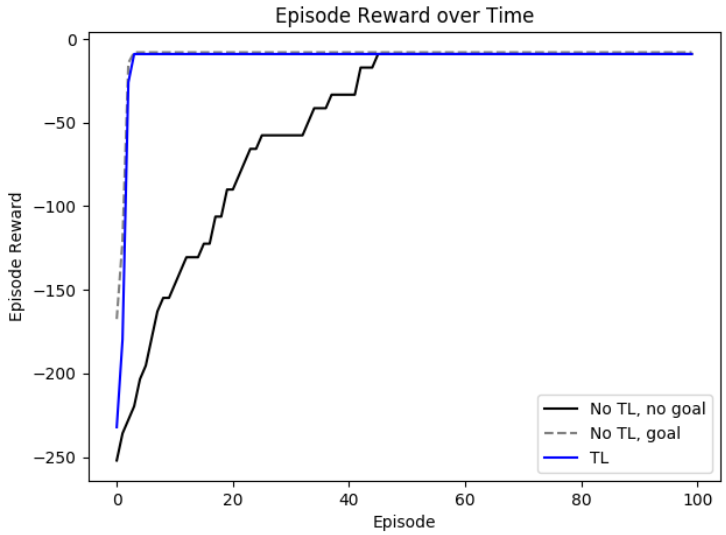

Figure 4.14: Evaluation 4: optimal policy

Figure 4.14 shows the results of optimal policy. Agent(TL) is able to successfully learn the new concept and quickly finds the optimal policy at the early episode. Similar to Evaluation 3, Agent(noTL)$_{\text{Goal}}$ also quickly reaches the optimal policy. This experiment shows that the hypotheses is transferable even in cases where there is a new hypothesis the agent needs to learn in a new environment.
Also the difference of the state where the link is located does not cause any problems with ILP(RL) algorithm even when the positive examples in the previous environment are transferred. This is because information of ajacent walls are within the context of each example rather than background knowledge. This experiment shows the flexibility of context dependent examples in RL scenarios.
The new hypotheses the agent learns are the same as that of Evaluation 2. The hypotheses regarding link_start and link_dest are what the Agent(TL) learnt in the new environment.

4.4 Discussion

We evaluated the properties of ILP(RL) using simple maze environments. While the development of ILP(RL) is still at an early stage and only a proof-of-concept, we observe both strengths as well as weakness of the current framework of ILP(RL), which are summarised in the following sections.

4.4.1 Strengths of ILP(RL)

As observed in Evaluation 1-4, there are several advantages of ILP(RL) over Q-learning.

Faster learning of an optimal policy The ILP(RL) agent learns the general state transitions of the environment at an early stage of the learning, mostly at episode 0, and as soon as it finds the terminal state, is able to generate an ASP plan as a sequence of actions to find an optimal policy. While this is a proof-of-concept approach and more work needs to be done, our approach is first attempt for incorporating an ASP-based ILP in RL problems and the evaluations shows that this is a promising direction for research.

Transfer learning Unlike Q-learning, where state-action value functions are updated, ILP(RL) learns state transition in the form of hypotheses, which can be applied to similar but different environments. We confirmed the possibility of transfer learning with the evaluations. Especially when the goal is known to the agent in Evaluation 3 and 4, the current ILP(RL) works the best because ASP planning is based on the terminal state. While this is a limited transfer learning since the terminal state is known in advance, this is still a useful transfer in cases where the terminal state is the same but the rest of the environment changes.

We also observed that the agent can learn a new hypothesis on top of the transferred hypotheses, the learn hypothesis is very flexible in terms of applicability in other environments.

Symbolic learning Since both the outcomes of ASP planning and inductive learning can be expressed in ASP syntax, the learning process as well as outcomes of ILP(RL) are easy to understand for human users, and the learnt hypotheses are very general state transitions of an environment.

4.4.2 Limitations of ILP(RL)

Although this first version of the ILP(RL) using inductive learning and ASP planning shows potentials for a new way of solving RL problems, it is a proof-of-concept and there are a number of limitations with the current framework. Some of these limitations are further elaborated on in the Further Research section in Chapter 6.2.

Runtime While we showed that ILP(RL) converges to an optimal policy in terms of the number of episodes, the computational time is significantly longer than that of Q-learning due to the computation required for inductive learning with ILASP as well as ASP planning. This limitation indicates that ILP(RL) may not be suitable in an environment where the runtime of learning is also a concern. ILP(RL) may also be unsuitable in an environment where there are dynamic moving objects based on time rather than time step.

Scalability issue for more complex environment ILP framework is known to be less scalable. The current framework is tested in a relatively simple environment, and proven to be work better than Q-learning in terms of learning an optimal policy by time step. However, learning runtime of ILP(RL) in each time step is slower than that of Q-learning, which is worsen when there are more hypotheses that ILP(RL) needs to learn. For example, as shown in Evaluation 1 and 2, adding two language biases significantly increases the runtime of the ILP(RL) algorithm, since the search space grows

significantly with respect to the language biases. While, Q-learning updates Q-value function regardless of whether there is a new concept such as teleport links, ILP(RL) needs to expands search space of hypotheses by adding more language bias. This is an important issue since many of RL research are focused on the development of RL algorithms in more complex environments.

Another question is the possibility of extending ILP(RL) to more realistic scenarios. Many RL works in more complex environments such as 3D or real physical environment, whereas the observations of an environment need to be expressed as ASP syntax for ILP(RL) to work.

Requirements of assumptions ILP(RL) requires initial assumptions such as background knowledge or specification of language bias for search space. While most RL works in different kinds of environment without any pre-configuration, as shown in the Evaluation 2, it was necessary to add two extra modeb before the learning starts. Thus the current framework of ILP(RL) is unfeasible in cases where these learning concepts were unknown or difficult to define with language bias.

In addition, not only it needs search space, but also it is assumed that an agent knows the definition of adjacent state and is able to see the adjacent states. While this assumption may be reasonable in many cases, this is not common in most RL algorithms.

solving limited MDP The current ILP(RL) framework does not make use of the rewards for inductive learning and only uses state transition as well as the terminal state for ASP planning. While our evaluations were conducted in a simple environment and we assumed that there is only one reward for any states except a terminal state. As described in Section2.3.4, other model-based RL methods learn a model of an environment, which tells the agent the reward and state transition functions. However, the current ILP(RL) only learns state transition and does not learn reward functions. In addition, some MDP problems contain no terminal state and instead there may be a different means to gain rewards. Since the current ILP(RL) is dependent on finding a terminal state for planning rather than maximising total rewards, the application of the current framework is limited to a particular type of MDP problems.

Chapter 5

Related Work

In this section, we review some of the related work in logic programming in reinforcement learning (RL), RL with answer set programming (ASP), symbolic RL, and model-based RL.
The combination of inductive logic programming (ILP) and RL has its root in *Relational Reinforcement Learning (RRL)*[30]. RRL equips with generalisation of ILP, and is based on first-order logic, and therefore does not cope with negation as failure or non-monotonic reasoning. Using the logic programming, RRL incorporates relational representations of states and actions and uses a relational function approximator to learn the Q-function[31]. However, most RRL algorithms focus on planning, where the model of an environment is known to the agent and the agent does not need to learn the model of the environment.
One algorithm which combines both aspects of model-based and model-free RL is called *Dyna* [32]. Dyna learns a model from real experience and uses the model to generate simulated experience to update the action-state value functions. This approach is more efficient than model-free RL because the simulated experiences are relatively easy to generate compared to real experiences, thus less interactions with the environment are needed to update the action-state value functions. The problem with Dyna is that if the learnt model is not a true representation of the environment, the agent may be stuck in a sub-optimal policy.
More recent work on using RRL is the paper in [33], which combines RRL and deep reinforcement learning (DRL) in order to improve the interpretability and an ability to generalise to more complex environments than the experienced. The proposed architecture uses self-attention mechanism [34] to reason about the relations between entities in the environment to help improving policy. While this is not an application of a ILP framework and only uses symbolic representations, the learnt works on very complex game environment called StarCraft II[1].
Similarly, [4] introduced *Deep Symbolic Reinforcement Learning (DSRL)*, a proof of concept for incorporating symbolic front end as a means of converting low-dimensional symbolic representation into spatio-temporal representations, which will be the state transitions input of reinforcement learning. DSRL extracts features using *Convolutional Neural Networks (CNNs)* [35], which are transformed into symbolic representations for relevant object types and positions of the objects. These symbolic representations represent abstract state-space, which are the inputs for the Q-learning algorithm to learn a policy on this particular state-space. DSRL was shown to outperform DRL in stochastic variant environments. However, there are a number of drawbacks to this approach. First, the extraction of the individual objects was done by manually defined threshold of feature activation values, given that the games were geometrically simple. Thus this approach would not scale in geometrically complex games.

[1] https://starcraft2.com/en-us/

Second, using deep neural network front-end might also cause a problem. [36] points out that a single irrelevant pixel could dramatically influence the state through the change in CNNs, as as demonstrated in [37]. In addition, while proposed method successfully used symbolic representations to achieve more data-efficient learning, there is still the potential to apply ILP to the extracted symbolic representations to further improve the learning efficiency and generalisation.
[36] further explored this symbolic abstraction approach in [4] by incorporating the relative position of each object with respect to every other object rather than absolute object position. They also assign priority to each Q-value function based on the relative distance of objects from an agent.
Another approach for using symbolic RL is to store heuristics expressed by knowledge bases [38]. An agent learns the concept of *Hierarchical Knowledge Bases (HKBs)* (which is defined in details in [39] and [40]) at every iteration of training, which contain multiple rules (state-action pairs). The agent then is able to decide itself when it should exploit the heuristic rather than the state-action pairs of the RL using *Strategic Depth*. This approach effectively uses the heuristic knowledge bases, which acts as a symbolic model of the game.
Another field related to our research is the combining of ASP and RL. The original concept of combining ASP and RL was in [41], where they developed an algorithm that efficiently finds the optimal solution of an MDP of non-stationary domains by using ASP to find the possible trajectories of an MDP. ASP is used to find a set of states of an MDP as choice rules describing the consequences of each possible action. In order to find stationary sets, an extension of ASP called BC^+, an action language [42], was used. BC^+ can directly translate the agent's actions into ASP form, and provide sequences of actions in answer sets. The more details of theoretical explanation of this approach is described in [43]. This approach focused more on efficient update of the Q function using ASP, and does not make use of inductive learning, whereas ILP(RL) focuses more on ASP-based ILP with RL.

Chapter 6

Conclusion

6.1 Summary of Work

In this project, we developed a new RL algorithm called *ILP(RL)* by applying an ASP-based ILP in order to develop a new learning framework in RL. Here we summarise the work of the project.

- We started off by looking at existing symbolic reinforcement learning approaches to understand the research fields and see the potential for improving further research. Advances in ASP-based ILP frameworks and the lack of existing works applying ASP-based ILP into RL scenarios motivated us to pursue the potentials of ILP to solve RL problems. We decided to apply ILASP as our core learning framework, because of its flexibility of applications.

- We considered the target hypotheses and how to construct learning tasks for ILASP. Our objectives of inductive learning is to learn state transition of an environment. As a way to use our learnt hypotheses to execute a sequence of actions, we used the Clingo ASP solver for our plan execution. The use of ASP optimisation is based on the assumption that the goal of the game is to find the shortest path to a terminal state.

- Having designed the overall framework of ILP(RL), we implemented the initial version of full-working software using Python engine, ILASP, Clingo and VDGL environment. This software allows us to evaluate our learning algorithm.

- We considered what kind of RL problems we would like to test with our new framework. Since this is a new proof-of-concept and it was necessary to test core functionalities of ILP(RL), we chose a simple maze game provided by VDGL and created original environments.

- We tested our new framework in various maze environments to highlight each aspect of the algorithm. We show that ILP(RL) learns an optimal policy faster than Q-learning for finding a shortest path, and show some capability for transfer learning. While the evaluations were conducted under limited conditions, we showed that there is potential in an ILP-based approach for RL problems.

6.2 Further Research

Since the development of ILP(RL) is a new attempt and we only developed the initial version and tested it on simple maze games, there are a number of directions for further research. We discuss some of the possible improvements and further research in this section.

Better exploration policy We used a random exploration policy for both ILP(RL) and Q-learning for the evaluations. As shown in the evaluations, however, ILP(RL) is heavily dependent on how quickly the agent finds a terminal state in order to start ASP planning. In RL research, the trade-off between exploration and exploitation is another active research field and a more sophisticated exploration strategy would improve ILP(RL) such as Count-based approach [44] or Interval Estimation [45].

Experiments on different environments Since the objective of this project is to develop the initial framework of ILP(RL) and run evaluations on the core part of the algorithms, further experimentation with different game environments is required in order to make the framework more robust, such as the presence of dynamic enemies in the environment or non-stationary environment. The more experimentation will lead to other aspects of using ILP in RL context and further bridge the gaps between the field of RL and ILP.

Learning reward function The current ILP(RL) can only learn state transitions and not reward function, and the rewards are assumed to be solved by using optimisation statement in ASP. In many RL environments, however, there may be more than one type of rewards. In order to improve our current framework to solve other types of MDP, the rewards themselves can be included as part of inductive learning using *Learning from ordered answer sets*, denoted ILP_{LOAS} [46]. This framework is an extension of ILASP by allowing the learning of ASP programs with *weak constraints* [47]. Examples under ILP_{LOAS} are called *ordered pairs of partial answer sets* which can represents which answer sets of a learnt hypothesis are preferred to the others. The preference learning can be applied for learning reward functions. With using ILP_{LOAS}, we could further generalise the current framework by removing ASP planning part and inductive learning could learn both reward and state transition functions.

Relaxing the initial assumption The current assumption requires the assumption of adjacent definitions as background knowledge. This could ,however, be learnt using ILASP. Furthermore, the concept of adjacent is another general concept. If another inductive learning could learn the adjacent definition, this could be useful in many different environments.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, 2015.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[4] M. Garnelo, K. Arulkumaran, and M. Shanahan, "Towards deep symbolic reinforcement learning," *arXiv preprint arXiv:1609.05518*, 2016.

[5] V. Lifschitz, "What Is Answer Set Programming?.," *AAAI*, vol. 8, pp. 1594–1597, 2008.

[6] M. Sergot, "Minimal models and fixpoint semantics for definite logic programs," *Lecture Notes: Knowledge Representation (C491), Department of Computing, Imperial College London*, 2005.

[7] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," *5th International Conf. of Symp. on Logic Programming*, no. December 2014, pp. 1070–1080, 1988.

[8] M. Law, A. Russo, and K. Broda, "Inductive learning of answer set programs," *European Conference on Logics in Artificial Intelligence (JELIA)*, vol. 2, no. Ray 2009, pp. 311–325, 2014.

[9] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider, "Potassco: The Potsdam answer set solving collection," *AI Communications*, vol. 24, no. 2, pp. 107–124, 2011.

[10] S. Muggleton, "Inductive logic programming," *New Generation Computing*, vol. 8, no. 4, pp. 295–318, 1991.

[11] L. De Raedt, "Logical settings for concept-learning," *Artificial Intelligence*, vol. 95, no. 1, pp. 187–201, 1997.

[12] S. Muggleton, "Inductive Logic Programming: derivations, successes and shortcomings," *SIGART Bulletin*, vol. 5, pp. 1–5, 1993.

[13] C. Sakama and K. Inoue, "Brave induction: A logical framework for learning from incomplete information," *Machine Learning*, vol. 76, no. 1, pp. 3–35, 2009.

[14] R. P. Otero, "Induction of Stable Models," *Conference on Inductive Logic Programming (ILP)*, pp. 193–205, 2001.

[15] K. Inoue, T. Ribeiro, and C. Sakama, "Learning from interpretation transition," *Machine Learning*, vol. 94, no. 1, pp. 51–79, 2014.

[16] D. Corapi, A. Russo, and E. Lupu, "Inductive Logic Programming in Answer Set Programming," *Inductive Logic Programming*, pp. 91–97, 2012.

[17] M. Law, A. Russo, and K. Broda, "Iterative Learning of Answer Set Programs from Context Dependent Examples," aug 2016.

[18] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[19] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*, vol. 135. MIT press Cambridge, 1998.

[20] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[21] S. Ray and P. Tadepalli, "Model-Based Reinforcement Learning," *Encyclopedia of Machine Learning*, pp. 690–693, 2010.

[22] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1633–1685, 2009.

[23] M. Law, A. Russo, and K. Broda, "Inductive Learning of Answer Set Programs v3.1.0 User Manual," 2017.

[24] M. Law, A. Russo, and K. Broda, "Iterative learning of answer set programs from context dependent examples," *Theory and Practice of Logic Programming*, vol. 16, no. 5-6, pp. 834–848, 2016.

[25] M. Gebser, R. Kaminski, A. König, and T. Schaub, "Advances in gringo Series 3," in *Logic Programming and Nonmonotonic Reasoning* (J. P. Delgrande and W. Faber, eds.), (Berlin, Heidelberg), pp. 345–351, Springer Berlin Heidelberg, 2011.

[26] M. Alviano and C. Dodaro, "Unsatisfiable core shrinking for anytime answer set optimization," in *International Joint Conference on Artificial Intelligence*, pp. 4781–4785, 2017.

[27] T. Schaul, "A video game description language for model-based or interactive learning," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pp. 1–8, IEEE, 2013.

[28] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "Procedural level generation with answer set programming for general video game playing," in *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, pp. 207–212, IEEE, 2015.

[29] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[30] S. Dzeroski, L. De Raedt, and K. Driessens, "Thesis: Relational Reinforcement Learning," *Machine Learning*, vol. 43, pp. 7–52, 2001.

[31] L. De Raedt, *Logical and relational learning*. Springer Science & Business Media, 2008.

[32] R. S. Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," *Machine Learning Proceedings 1990*, vol. 02254, no. 1987, pp. 216–224, 1990.

[33] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, *et al.*, "Relational deep reinforcement learning," *arXiv preprint arXiv:1806.01830*, 2018.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

[35] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[36] A. d. Garcez, A. R. R. Dutra, and E. Alonso, "Towards symbolic reinforcement learning with common sense," *arXiv preprint arXiv:1804.08597*, 2018.

[37] J. Su, D. V. Vargas, and S. Kouichi, "One pixel attack for fooling deep neural networks," *arXiv preprint arXiv:1710.08864*, 2017.

[38] D. Apeldoorn and K.-I. Gabriele, "An Agent-Based Learning Approach for Finding and Exploiting Heuristics in Unknown Environments," *Proceedings of the Thirteenth International Symposium on Commonsense Reasoning*, pp. 1–8, 2017.

[39] D. Apeldoorn and G. Kern-Isberner, "When should learning agents switch to explicit knowledge?," in *GCAI*, pp. 174–186, 2016.

[40] D. Apeldoorn and G. Kern-Isberner, "Towards an understanding of what is learned: Extracting multi-abstraction-level knowledge from learning agents," in *Proceedings of the Thirtieth International Florida Artificial Intelligence Research Society Conference, V. Rus and Z. Markov, Eds. Palo Alto, California: AAAI Press*, pp. 764–767, 2017.

[41] L. A. Ferreira, R. A. Bianchi, P. E. Santos, and R. L. de Mantaras, "Answer set programming for non-stationary markov decision processes," *Applied Intelligence*, vol. 47, no. 4, pp. 993–1007, 2017.

[42] J. Babb and J. Lee, "Action language bc+: Preliminary report.," in *AAAI*, pp. 1424–1430, 2015.

[43] L. A. Ferreira, R. A. Bianchi, P. E. Santos, and R. L. de Mantaras, "Answer set programming for non-stationary markov decision processes," *Applied Intelligence*, vol. 47, no. 4, pp. 993–1007, 2017.

[44] J. Martin, S. N. Sasikumar, T. Everitt, and M. Hutter, "Count-based exploration in feature space for reinforcement learning," *arXiv preprint arXiv:1706.08090*, 2017.

[45] A. L. Strehl and M. L. Littman, "An analysis of model-based interval estimation for markov decision processes," *Journal of Computer and System Sciences*, vol. 74, no. 8, pp. 1309–1331, 2008.

[46] M. Law, A. Russo, and K. Broda, "Learning weak constraints in answer set programming," *Theory and Practice of Logic Programming*, vol. 15, no. 4-5, pp. 511–525, 2015.

[47] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub, "Asp-core-2: Input language format," *ASP Standardization Working Group, Technical report*, 2012.

Appendix A

Ethics

There are no particular legal and ethical considerations for this project listed in Table A.1.

- We do not have any considerations regarding human embryos or foetuses, human participants or human cells or tissues since none of them were involved (Section1-3 in Table A.1).
- The only data we used for our experiments are collected from an game environment, and none of them involved any personal data (Section 4 in Table A.1).
- Animals are not involved (Section 5 in Table A.1).
- Developing countries are not involved (Section 6 in Table A.1).
- This not project does not involve any environmental related issues or safety (Section 7 in Table A.1).
- There is no potentials for dual use. Although this project is part of artificial intelligence in both inductive logic programming as well as reinforcement learning, it is a preliminary research and all the experiments were conducted using a game environment (Section 8 in Table A.1).
- This project is a preliminary research and, there is no concerns regarding the misuse of applications in the foreseeable future (Section 9 in Table A.1).
- We use only open source software, and none of them have any legal issues for the use of a research (Section 10 in Table A.1).
- We do not have any other ethics issues regarding this project (Section 11 in Table A.1).

	Yes	**No**
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓

Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from Human Embryos/Foetuses i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?		✓
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓

Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		✓
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		✓
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
Section 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓
Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
Section 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓

Table A.1: Ethics Checklist

Appendix B

Learning task for Evaluation 1

This is the full learning task for ILASP in Evaluation 1.

Listing B.1: Learning tasks for Evaluation 1

```
cell((0..18, 0..8)).
adjacent(right, (X+1,Y),(X,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(left,(X,Y),(X+1,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(down,(X,Y+1),(X,Y)):-cell((X,Y)),cell((X,Y+1)).
adjacent(up,(X,Y),(X,Y+1)):-cell((X,Y)),cell((X,Y+1)).
#modeh(state_after(var(cell))).
#modeb(1,adjacent(const(action),var(cell),var(cell)),
      (positive)).
#modeb(1, state_before(var(cell)),(positive)).
#modeb(1, action(const(action)),(positive)).
#modeb(1, wall(var(cell))).
#max_penalty(50).
#constant(action,right).
#constant(action,left).
#constant(action,down).
#constant(action,up).
% Context dependent examples are added here
```

Appendix C

Learning task for Evaluation 2

This is the full learning task for ILASP in Evaluation 2.

Listing C.1: Learning tasks for Evaluation 2

```
cell((0..18, 0..8)).
adjacent(right,(X+1,Y),(X,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(left,(X,Y),(X+1,Y)):-cell((X,Y)),cell((X+1,Y)).
adjacent(down,(X,Y+1),(X,Y)):-cell((X,Y)),cell((X,Y+1)).
adjacent(up,(X,Y),(X,Y+1)):-cell((X,Y)),cell((X,Y+1)).

#modeh(state_after(var(cell))).
#modeb(1,adjacent(const(action),var(cell),var(cell)),
      (positive)).
#modeb(1,state_before(var(cell)),(positive)).
#modeb(1,action(const(action)),(positive)).
#modeb(1,wall(var(cell))).
% Two additional language biases
#modeb(1, link_start(var(cell)),(positive)).
#modeb(1, link_dest(var(cell)),(positive)).
#max_penalty(50).
#constant(action,right).
#constant(action,left).
#constant(action,down).
#constant(action,up).
#pos({state_after((2,6))},
% Context dependent examples are added here
```