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 learnig efficiency, unability to use abstract reasoning, and inability of tranfer learning to similar environments. In order to tackle these shortcomings, we introduce a new approach called ILP(RL). ILP(RL) learns a general concept of a valid move in an environment, called a hypothesis, using ILASP, and generates a plan for a sequence of actions to the destination using ASP. The learnt hypotheses is highly expressive and transferrable 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 ILP into a RL problem. We examined ILP(RL) in a various simple maze environments, and show that ILP(RL) learns faster than existing RL methods. We also show that transfer learning of the learnt hypothesis successfully improves learning on a new but similar environment. This proof of concept approach shows potentials for this new way of learning using ILP. Although the experiments were conducted in a simple environment, the results of the experiments are promissing, and oepn up an avenue for future research directions.

Acknowledgments

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 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 domains 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. Especially together with deep learning (DL), there have been many successful applications of RL in a number of domains, such as video games [**?**], the game of Go [**?**] and robotics [**?**].

Despite these successful applications of RL, there are still a number of issues that RL has to overcome. First, most of the RL algorithms requires large number of trial-end-error interactions with long time of training, which is also computationally expensive. Second, most of the RL algorithms have no thought process to 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 to perform 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 based on logic programming and 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 compared to RL. First, unlike most of statistical machine learning methods, ILP requires a small number of training data due to the presense 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 to interpret for human users. Third, since the learnt hypothesis is a abstract and general concept, it can be easily applied to a different learning task, making transfer learning possible. The disadvantages of ILP system are that examples, or training data, have to be clearly defined. 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 has been a number of advance in ILP, especially ILP frameworks based on Answer Sets Programming (ASP), a declarative logic programming which defines semantics based on Herbrand models (Gelfond and Lifschitz 1988). Because

of the progress and limitations of RL and ILP, we developed a new RL approach by incorporating a new ASP-based ILP framework called Learning from Answer Set (ILASP), which learns a valid move within an environment, and uses the learnt hypothesis and background knowledge to generate a sequence of actions in the environment.
In recent RL research, there is a number of research on 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 the system [**?**]. This approach is promising and shows potentials. In addition, there has been works on using ASP into RL to achieve data-efficient learning. However, none of the papers attempt to apply inductive learnig in RL senarios.
Since the recent ILP framworks enable us to learn a complex hypothesis in a more realistic environments,
Finally, the recent advance of Inductive Logic Programming (ILP) research has enabled us to apply ILP in more complex situations and there are a number of new algorithms 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 my motivation is to combine these two different subfields of machine learnig and devise a new way of RL algorithm in order to overcome some of the RL problems.
Particularly since [**?**], there have been several researches that further explored the incorporation of symbolic reasoning into RL, but the combining of ILP and RL has not been explored.

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 potentials of ILP(RL) for improving the learning efficiency and transfer learning capability that current RL methods suffer.
The objective of this project is devided 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 (MDP). Since ASP-based ILP algorithms require 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. **Using the learnt concept to execute actions.**
   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 the hypothesis.

4. **Evaluation of the new framework in various environments.**
   In order to investigate the applicabilities of ILP-based approach, evaluation of the new framework in various scenarios is conducted in order to gain insight into its potential, especially 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 therefore more complex environments such as continuous states or schocastic environments are not considered in this project. The possibilities of applying these more complex environments are discussed in Section**??**.

1.3 Contributions

The main contribution of this project is the development of a novel ILP based approach to reinforcement learning . This project contributes to the incorporation of ASP-based ILP ilearning frameworks into Reinforcement Learning by applying the latest ILP framework called Learning from Answer Sets. To my knowledge, this is the first attempt of incorporating an ASP-based ILP learning framework is incorporated into an RL scenario.
In simple environments, we show that an agent learns the rules of the game and reaches an optimal policy faster than existing RL algorithms. We also show that the learnt hypothesis is easy to understand for human users, and can be applied to other environments to optimise the agent's learning process.
The full hypotheses were learnt in the early stage of the learning and exploration phase. Thus with sufficient exploration, the model of the environment is correct and therefore the agent is able to find the optimal policy, or the shortest path.
We show that ILP(RL) is able to solve a reduced MDP where the rewards are assumed to be associated with a sequence of actions planned as answer sets. Although this is a limited solution, there is a potential to expand it to solve full MDP as discussed in Further Research.
TODO more details on the strength of the algorithm. Validity

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 the learning efficiency and the capability of the transfer learning are compared it against two existing RL algorithms. 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 framework and experiments of ILP(RL) 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 [**?**]. 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:

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

where h, a_1, ... a_{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 fromconstants 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.

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. 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*.

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

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 2.8. 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*.

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). Since there is no unique least Herbrand Model for a normal logic program, Stable Model of a normal logic program is defined in [**?**]. 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 2.9. 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 2.10. X is a *stable model* of P if it is the least Herbrand Model of P^X.

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 $(ground(P))^x = \{male(john), female(anna).\}$

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

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 **??**.

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.
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 $1 \leq i \leq m$ are atoms. 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 $l \leq |X \cap \{h_1, ..., h_m\}| \leq u$ **??**.
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 [**?**]. We will use Clingo 5 for the implementation of our new framework.

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 [**?**]. 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 [**?**]. ILP is based on learning from entailment, as shown in Equation **??**.

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

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 [**?**].

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* ([**?**]), which are the foundations of *Learning from Answer Sets* discussed in Section **??**, a state-of-the-art ILP framework that we will use for our new framework (for other non-monotonic ILP frameworks, see [**?**], [**?**], [**?**] and [**?**]).

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 $\in$ $ILP_{cautious}$ $\langle$ 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 $\cup$ H:

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

Example 2.2.1. (Cautious Induction)

$$B = \begin{cases} exercises \leftarrow not\ \ eat\_out. \\ eat\_out \leftarrow exercises. \end{cases} \qquad E^+ = \{\text{tennis}\},\ E^- = \{\text{eat\_out }\}$$

One possible H $\in$ $ILP_{cautious}$ is {tennis$\leftarrow$ exercises, $\leftarrow$ not tennis }.

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.

Example 2.2.2. (Limitation of Cautious Induction)

$$B = \begin{cases} 1\{situation(P,\ awake),\ situation(P,\ sleep)\}1 \leftarrow person(P). \\ person(john). \end{cases}$$

Neither of *situation(john, awake)* nor *situation(john, sleep)* is false in all answer sets. In this example, the hypothesis only contains person(john). Thus no examples could be given to learn the choice rule.

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_{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

Example 2.2.3. (Brave Induction)

$$B = \begin{cases} exercises \leftarrow not\ \ eat\_out. \\ tennis \leftarrow holiday \end{cases} \qquad E^+ = \{\text{tennis}\},\ E^- = \{\text{eat\_out }\}$$

One possible H $\in$ ILP_{brave} is {tennis}, which returns {tennis, holiday, exercises} as answer sets.

The limitation of Brave Induction is that it is not possible to learn constraints, since the conditions of the Definition **??** 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.

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

Example 2.2.4. (Limitation of Brave Induction)

$$\text{B} = \begin{cases} 1\{\textit{situation(P, awake), situation(P, sleep)}\}1 \leftarrow \textit{person(P)}. \\ \textit{person(C)} \leftarrow \textit{super\_person(C)}. \\ \textit{super\_person(john)}. \end{cases}$$

In order to learn the constraint hypothesis H = { ← not situation(P, awake), super_person(P)}, it is not possible to find an optimal solution.

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 rules and constraints. S_M is specified by *language bias* of the learning task using *mode declaration*. 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 $\text{ILP}_{\text{LAS}}(\text{T})$, and a hypothesis H is an inductive solution of $\text{ILP}_{\text{LAS}}(\text{T})$ $\langle$ B, S_M, E^+, E^- $\rangle$ such that:

1. $\text{H} \subseteq S_M$
2. $\forall\ e^+ \in E^+ : \exists$ A $\in$ Answer Sets(B $\cup$ H) such that A extends e^+
3. $\forall\ e^- \in E^- : \nexists$ A $\in$ Answer Sets(B $\cup$ H) such that A extends e^-

Example 2.2.5. (LAS). Suppose a learning task T is as follows:

#modeh(1, drive(var(person))).
#modeb(1, hasCar(var(person))).
#pos(e1, {hasCar(john), drive(john)}, {})
#pos(e2, {hasCar(anna), drive(anna)}, {})
person(john).
person(anna).

where, both person(john) and person(anna) are background knowledge B, #modeh and #modeb are language bias SM and two positive examples #pos are given.

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 $\text{ILP}_{\text{LAS}}(\text{T}) \langle \text{B}, \text{S}_\text{M}, \text{E}^+, \text{E}^- \rangle$, and is based on two fundamental concepts: *positive solutions* and *violating solutions*.
A hypothesis H is a positive solution if and only if

1. $\text{H} \subseteq \text{S}_\text{M}$
2. $\forall \text{e}^+ \in \text{E}^+ \exists \text{A} \in$ Answer Sets$(\text{B} \cup \text{H})$ such that A extends e^+

A hypothesis H is a violating solution if and only if

1. $\text{H} \subseteq \text{S}_\text{M}$
2. $\forall \text{e}^+ \in \text{E}^+ \exists \text{A} \in$ Answer Sets$(\text{B} \cup \text{H})$ such that A extends e^+
3. $\exists \text{e}^- \in \text{E}^- \exists \text{A} \in$ Answer Sets$(\text{B} \cup \text{H})$ such that A extends e^-

Given both definitions of positive and violating solutions, $\text{ILP}_{\text{LAS}} \langle \text{B}, \text{S}_\text{M}, \text{E}^+, \text{E}^- \rangle$is positive solutions that are not violating solutions.

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*[**?**][2]. *Context-dependent examples* are examples in which each unique background knowledge, or *context*, only applies to specific examples. This way the background knowledge is more structured rather than one fixed background knowledge that is applied to all examples.
Formally, partial interpretation is called *context-dependent partial interpretation (CDPI)*, which is of the form:

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

where *e* is a partial interpretation and C is a context, or an ASP program without weak constraints.

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

1. $\text{H} \subseteq \text{S}_\text{M}$
2. $\forall \langle \text{e}, \text{C} \rangle \in \text{E}^+, \exists \text{A} \in$ Answer Sets $(\text{B} \cup \text{C} \cup \text{H})$ such that A extends e
3. $\forall \langle \text{e}, \text{C} \rangle \in \text{E}^-, \nexists \text{A} \in$ Answer Sets $(\text{B} \cup \text{C} \cup \text{H})$ such that A extends e

One of the main advantages of using context dependent examples is that when a hypothesis is computed, only the relevant set of examples are used for search rather than using all examples at once. Relevant examples are counterexamples for the previously computed hypothesis in the previous iterations. This iterative approach increases the efficiency of solving learning tasks, and enables more expressive structure of the background knowledge to particular examples.
$ILP^{context}_{LAS}$ is used in ILASP2i, which we use for our new framework.

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

Example 2.2.6. ($ILP^{context}_{LAS}$).
Suppose a learning task T is as follows:

#modeh(1, drive(var(person))).
#modeb(1, hasCar(var(person))).
#pos(e1, {hasCar(john), drive(john)}, {})
#pos(e2, {hasCar(anna), drive(anna)}, {})
person(john).
person(anna).

where, both person(john) and person(anna) are background knowledge B, #modeh and #modeb are language bias SM and two positive examples #pos are given.

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 **??**, the agent interacts with an environment, and at each time step 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 benchmarks of the experiments discussed in Chapter **??**.

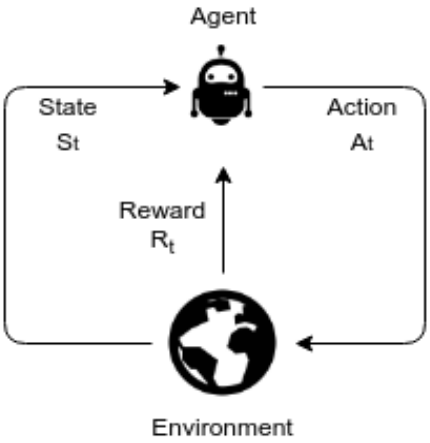

Figure 2.1: The interaction between an agent and an environment

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

$$H_t = S_1, R_1, A_1, ..., A_{t-1}, S_t, R_t. \tag{2.10}$$

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

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

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 ([**?**]). 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 ⟨S, A, T, R⟩ 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 × A × S → [0,1], which is a probability of reaching a future state $s' \in$ S by taking an action a ∈ A in the current state s ∈ S.
- R is a reward function $R_a(s, s') = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$, 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*,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.
For RL programs, it is not necessary for the agent to know T and R in advance, but they are present in the environment and can be realised each time the agent takes an action.

2.3.2 Policies and Value Functions

Most RL algorithms are concerned with estimating *Value functions*. Value functions estimate the expected return, or expected future reward, for a given action in a given state. The expected reward for an agent is dependent on the agent's action. Value functions are defined by *policies*, which maps from states to probabilities of choosing each 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:

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

Example 2.3.1. (Value Functions).
XXX

Similar to the state value function v_π, we define an *action-value function* under a policy π, which represents the value of taking action a in state s following a policy π.

$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] \text{ for all } s \in \mathcal{S} \tag{2.15}$$

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

$$q^*(s,a) = \max_\pi q_\pi(s,a) \tag{2.16}$$

A solution to MDP is called a *policy* π, a sequence of actions that leads to a solution. An optimal policy achieves the optimal value function (or action-value function), and it can be computed by maximising over the optimal value function (or action-value function).

Example 2.3.2. (Action-value Functions).
XXX

2.3.3 Model-based and Model-free RL

A *model* is a representation of an environment that an agent can use to understand how the environment should look like. 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 [**?**]. Using a model, an agent can do planning to generate or improve a policy for the modeled environment. Most of the RL methods are model-free RL, where the model is unknown and the agent learns to achieve the goal solely by interacting with the environment. 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, unlike with model-free RL, the agent does not require trial-and-error interactions with the environment and therefore model-based RL is faster to reach an optimal policy. However, when the model is not a true representation of the environment, or the true MDP, the planning algorithms will not lead to the optimal policy, but a suboptimal policy.
One algorithm which combines both aspects of model-based and model-free learning to solve the issue of sub-optimality is called Dyna [**?**]. Dyna learns a model from real experience and uses the model to generate simulated experience to update the evaluation functions. This approach is more efficient because the simulated experiences are relatively easy to generate compared to real experiences, thus less iterations are required.
This idea of learning a model of the environment and using it to execute planning is a core idea of our new framework.

2.3.4 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 updates the estimates of value function as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.17}$$

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 method 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 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 (non-function approximation).
Q-learning is off-policy TD learning defined in [**?**], 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:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma max(a+t)Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.18}$$

where α is a constant step-size parameter, or learning rate, α between 0 and 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.

$$Q(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|s_t, a_t] \tag{2.19}$$

Q-learning is guaranteed to converge to an optimal policy in a finite tabular representation. 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.

2.3.5 Function Approximation

Q-learning with a tabular representation works when every state-action value can be represented. In case of very large MDPs, however, it may not be possible to represent all state-action values with a tabular representation. For example, robot arms have a continuous states in 3D dimensional space. This problem motivates the use of *function approximation*, which estimates value function with function approximation. Not only is it represented in tabular form, but also in the form of a parameterised function with *weight vector w*. Unlike Q-table, changing one weight updates the estimated value of not only one state, but many states, and this generalisation makes it more flexible to apply to different scenarios where the tabular approach could not be applied.
The reason we are introducing the function approximation is that it is used as another benchmark for our new framework.

The Prediction Objective ($\overline{VE}$)

With function approximation, an update at one state changes many other states, and therefore the values of all states will not be exactly accurate. There is a tradeoff among states as to which state we make it more accurate, while others might be less accurate. The error in a state s is the square of the difference between the approximate value $\hat{v}$(s,w) and the true

value $v_\pi(s)$. The objective function can be defined by weighting it over the state space by μ, the *Mean Squared Value Error*, denoted $\overline{VE}$.

$$\overline{VE}(w) \doteq \sum_{s \in S} \mu(s) \big[v_\pi(s) - \hat{v}(s,w)\big]^2. \tag{2.20}$$

Stochastic gradient descent (SGD)

Stochastic gradient descent (SGD) methods are commonly used to learn function approximation in value prediction, which works well for online reinforcement learning. The weight vector w in SGD is defined as:

$$w \doteq \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \tag{2.21}$$

and $\hat{v}$(s,w) is a differentiable function of w for all s $\in$ S. SGD adjusts the weights vector by a fraction of α, a step-size parameter, in the direction that reduces the error on that example the most. Formally, it is defined as:

$$\begin{aligned} w_{t+1} &\doteq w_t - \frac{1}{2}\alpha \bigtriangledown \big[v_\pi(S_t) - \hat{v}(S_t, w_t)\big]^2. \\ &= w_t + \alpha \big[v_\pi(S_t) - \hat{v}(S_t, w_t)\big] \bigtriangledown \hat{v}(S_t, w_t). \end{aligned} \tag{2.22}$$

The gradient of J(w), which is the vector of partial derivatives with respect to each weight vector, is defined as:

$$\bigtriangledown_w J(w) = \begin{pmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{pmatrix} \tag{2.23}$$

Linear Value Function Approximation

A *linear function* of a weight vector is a type of simple function approximation to approximate the action-value function.

$$\hat{q}(s,a) \approx q_\pi(s,a) \tag{2.24}$$

The vector x(s,a) represents a *feature vector*, which is of the form:

$$x(S,A) = \begin{pmatrix} x_1(S,A) \\ \vdots \\ x_n(S,A) \end{pmatrix} \tag{2.25}$$

where each x(s,a) is a feature of the corresponding action-state pair.
Using the SGD update with linear function approximation, the gradient of the approximate value function with respect to w is:

$$\bigtriangledown \hat{q}(s,a,w) = x(s,a) \tag{2.26}$$

Thus the general SGD update defined in (**??**) can be simplified to the following:

$$w_{t+1} = w_t + \alpha\big[q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, w_t)\big]x(S_t, A_t) \tag{2.27}$$

Unlike non-linear value function approximation, the linear method is guaranteed to converge to a global optimum. One disadvantage of the linear method is that it cannot express any relationship between features. For example, it cannot represent that feature *i* is useful only if feature *j* is not present. Nevertheless, this approach is sufficient for our experiments, which are described in Chapter **??**.

Tile Coding (TBD)

There are different linear function approximation methods to represents states as features. Feature construction depends on a problem you are solving. We introduce *Tile Coding* which will be used for our benchmark.

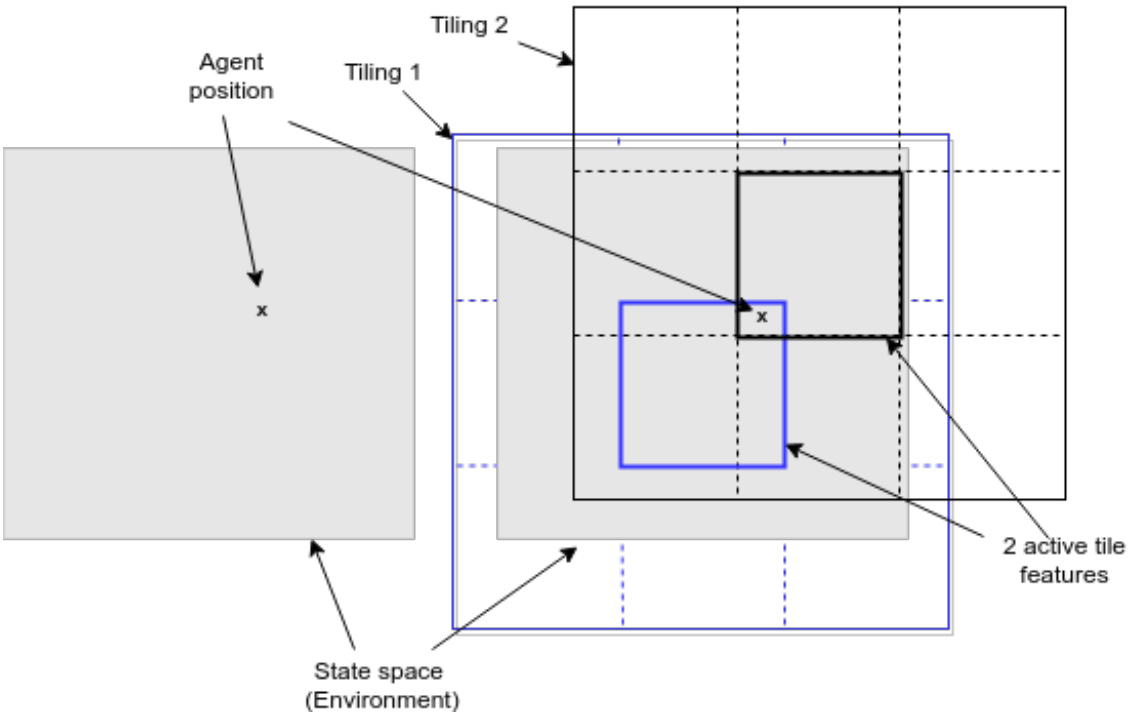

Figure 2.2: Tiling coding illustration

Illustration of tile coding is shown in Figure **??**. In Tile Coding, state set is represented as a continuous two-dimentionala space. If a state is within the space, then the value of the correnspondingfeature is set to be 1 to indicate that the feature is present, while 0 indicates that the feature is absent. This way of representing the feature is called *binary feature*. *Coarse coding* represents a state with which binary features are present within the space. One area is associated with one weight w, and training at a state will affect the weight of all the areas overlapping that state. the approximate value function will be updated within at all states within the union of the areas, and a point that has more overlap will be more affected. The size and shape of the areas will determine the degree of the generalisation. Large areas will have more generalisation. In addition, tiles can be overlap, and the change of the weight

in that state will affect all other states within the intersection of the spaces. The degree of overlap within a space will determined the degree of the generalisation. The shape of the space also affect how it is generalised.
Tile coding is a type of coarse coding. *Tiling* is a partition of state space, and each element of the partition is called a *tile*.
The state stpace is partitioned into multiple tiles with multiple tilings. Each tile in each tiling is associated with
In order to do coarse coding with tile coding, multiple tilings are required, each tiling is offset from one another by a fraction of a tile width.
As illustrated in Figure XXX, when a state occurs, several features with corresponding tiles become active,
Tile coding has computational advantage, since each component of tiling is binary value, XXXX.
a trained state will be generalised to other states if they are within any of the same tiles.
Similar to coarse coding, the size and shape of tiles will determine the degree of approximation.

Example 2.3.3. (Tile Coding).
XXX

2.3.6 Transfer Learning (TBD)

Transfer learning is a method that knowledge learnt in one or more tasks can be used to learn a new task better than if without the knowlege in the first task.
Transfer learning is an active research areas in machine learning, but not many have been done in RL. Since training tend to be time consuming and computational expensive, transfer learning allow the trained model to be applied in a different setting.
Transfer learning in RL is particularly important since most of the RL research have been done in a simulation or game scenarios, and training RL models in a real physical environment is more expensive to conduct.
Even in a virtual environments like games, the transfer learning between different tasks will greatly will have a big impact on potential applications.
This will also speed up learning
Transfer learning in ILP domain have been proved to be successful in many fields,
Since this project is combining ILP into RL senarios, this has a potential for extending this particular research.
We conducted experiements on transfer learning capabilities, which we describe in XXX.
One of the purposes of transfer learning is so that the agent requires less time to learn a new task with the help of what was learn in previous tasks.
Another goal would be to measure how effectively the agent reuses its knoledge in a new task. In this case the performan of learningon the first task is usally not measured.
There are many different matrices used to measure the performance of the transfer learning. Five common matrics are defined in XX as follow.
TODO source task selection
Since each matric measures different aspect of transfer learning, using multiple metrics would provide more comprehensive views of the performance of an RL algorithm.

Chapter 3

Framework

This chapter introduces a new proof-of-concept framework called *ILP(RL)*, a new learning framwork for RL problems 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

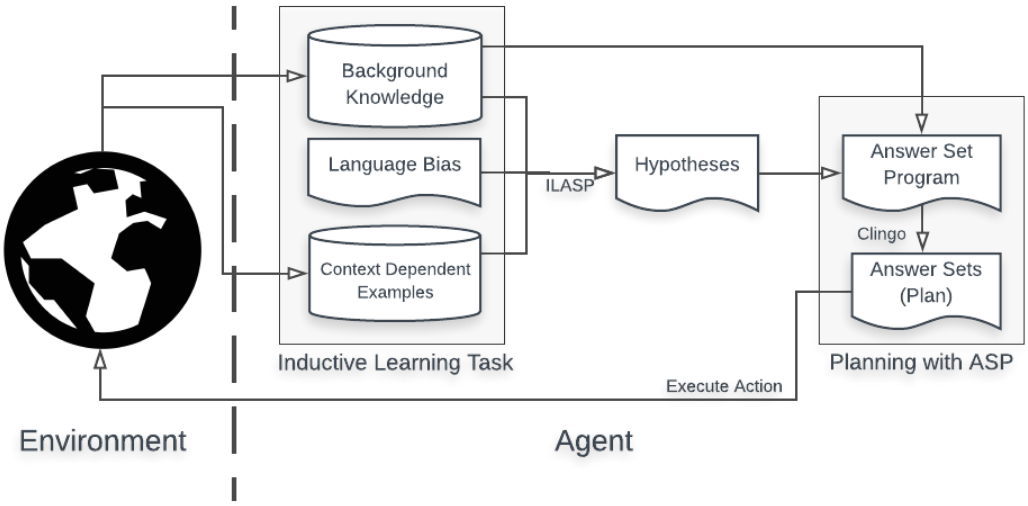

Figure 3.1: ILP(RL) overview

The overall architecture of ILP(RL), is shown in Figure **??**. ILP(RL) mainly consists of two components: inductive learning with ILASP and planning with ASP.
First step is inductive learning. An agent interacts with an unknown environment, and receives state transition experiences as context dependent examples. Together with predefined background knowledge and language bias, these examples are used to inductively learn and improve hypotheses, which are valid moves in the environment.
Second step is ASP planning. The interaction with the environment also gives the agent information about the environment, such as locations of walls or a goal. The agent remembers these information as background knowledge, and, together with the learnt hypotheses, uses

it to make an action plan by solving an answer set program. The plan is a sequence of actions and it navigates the agent in the environment.
The agent iteratively execute this cycle in order to learn and improve the hypotheses as well as an action planning. Mechanisms of each step are explained in details in the following sections.

3.2 Environment

Since there is no exisitng base framworks for ILP(RL), our focus on this project is to develop the preliminary version of the framework. Therefore, we use a very simple environmnet that allows us to see the potentials of our proposed architecture. The base environment is a simple grid maze, and we assume that the environment is a discrete deterministic environment.
We use an simple example to explain the environment as shown in Figure **??**. States are expressed as X and Y coordinates. In Figure **??**, for example, the agent is located at {X=2, Y=4}. The agent can take one of four possible actions at each time: up, down right and left. Every time the agent takes an action, the agent receives the following experience: a reward R_t, the next state S_{t+1}, surrounding information of the current state. The base environment mainly consists of three different elements: a goal, walls and paths. The goal cell is the terminal state and the agent receives a positive reward when and the agent receives an negative rewards in any states except the terminal state. When the agent reaches the terminal state, the agent receives a positive reward and the current episode is complete. Since the agent's goal is to maximise the total estimate rewards over time, this goal is equivalent to finding the shortest path from a starting state to a terminal state.
We also assume that, unlike an agent with other RL algorithms, the agent can see states of vertical and horizontal, but not diagonal, states around the agent. This assumption allows a agent to learn a valid move in the environment. For example, the agent at {X=2, Y=4} can see that there are walls at {X=1, Y=4} and {X=2, Y=5}. More details of how to use these surrounding information is described in **??**.
The applicability in more complex environment is not considered in this project and it is discussed in Further Research in Section **??**.

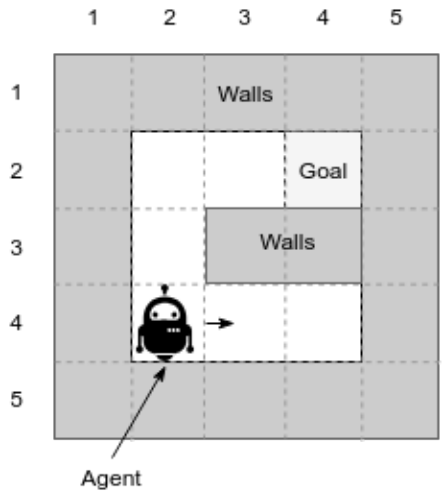

Figure 3.2: 5×5 grid maze example

3.3 Inductive Learning Task

The first step is constructing learning task using ILASP. The objective of the inductive learning is to learn a valid move in a given environment, which is used for generating action plan at the later step. The definition of a valid move is 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}, 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 (e.g adjacent(right, V0, V1) means V0 is right next to V1). wall(V) and not wall(V) specify whether there is a wall or not wall in a state V respectively. We describe the details of how to gain background knowledge, context dependent examples and language bias, necessary components for the agent to learn the above hypotheses. The summary of a full ILASP learning task can be found in Appendix **??**.

3.3.1 Background Knowledge

First we define necessary background knowledge for inductive learning. In order to learn a valid move for each direction in the form of state transition as shown in **??**, we need to define the meaning of "being next to" a state. This is defined as *adjacent*, which is 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 coordinates respectively. The rules **??** are given as background knowledge and allow the agent to understand the relation of two adjacent states. *cell((X,Y))* is defined as follows:

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

where, 0..X defines the range of X coodinates and X and Y are the size of width and height of the environment respectively. For example, a grid maze shown in Figure**??** has a height and width of 5, thus the type cell is defined in the background knowledge as cell((0..5, 0..5)).

3.3.2 Context Dependent Examples

Context dependent examples contain the results of the agent's interaction 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 as a following ASP form:

$$\#\mathsf{pos}(\{e^{\text{inc}}\}, \{e^{\text{exc}}\}, \{\mathsf{C}\}) \tag{3.4}$$

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

Definition 3.1. e^{inc} of CDPI for ILP(RL) is the next state of an agent $\forall$ $s_{t+1} \in$ S such that answer sets of B $\cup$ H does not cover.

Definition 3.2. e^{exc} of CDPI for ILP(RL) is the next state of an agent $\forall$ $s_{t+1} \notin$ S such that answer sets of B $\cup$ H covers, as well as $\forall$ $s'_{t+1} \in S_{neighbor}$ such that $s_{t+1} \neq s'_{t+1}$.

where B is the current background knowledge, H is the current hypotheses learnt by previous inductive learning using ILASP, S is all the states in the environment, and $S_{neighbor}$ is vertial and horizontal adjacent states of s_t as well as s_t itself.

Definition 3.3. Context C of CDPI for ILP(RL) contains an action a_t, a state s_t, and adjacent walls of s_t.

Both e^{inc} and e^{exc} contain only the next state s_{t+1}, expressed as *state_after((x,y))*, or empty. For context C, a_t is translated int *action((a))*, s_t is translated into *state_before((x,y))*, and adjacent walls of s_t are translated into *wall((x',y'))*.
In this report, we assume that part of context contains only whether a wall exists or not, with the presence of a wall, the agent cannot move to the state where a wall exist.

Example 3.3.1. (Context dependent examples).

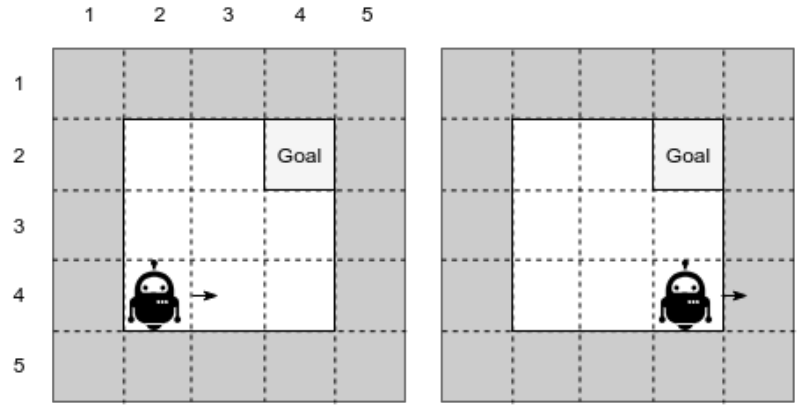

Figure 3.3: 5×5 grid maze example

We use a simple 5x5 gridworld environment to hilight how an agent gains a positive example. Suppose the agent does not know a hypothesis and takes an action "right" to move from (2,4) to (3,4) cell, as shown on the left in Figure **??**. a_t is "right" s_t is (2,4) and s_{t+1} is (3,4). According to the Definition **??**, the answer sets of B $\cup$ H, does not cover s_{t+1} since H = $\emptyset$, thus state_after((3,4)) is in e^{inc}. All other alternative next states S_{t+1} that the agent could

have ended up by taking different actions (down, up, and left) are in the exclusions. Context includes a_t, s_t and $S_{neighbor}$ of walls. The following positive example is generated.

```
#pos({state_after((3,4))},
    {state_after((2,4)),state_after((1,5)),state_after((0,4)),state_after((1,4))},
    {state_before((2,4)). action(right). wall((1, 4)). wall((4, 2)).})
```
(3.5)

The next example illustrates a senario when the agent tries to move to a state where a wall is. The agent is at (4,4) and tries to move to a cell (5,4) by taking an action "right", as shown on the left in Figure **??**. In this case, however, there is a wall at (5,4) and therefore the agent cannot go to that cell. Because of the blocking wall, s_t and s_{t+1} are (4,4). The same the previous case, $H = \emptyset$ and therefore the answer sets of $B \cup H$, does not cover s_{t+1}. All other alternative next adjacent states S_{t+1} (4,3), (3,4), (5,4) and (4,5) are exclusions, and the context is collected. From this example, the following positive example is generated:

```
#pos({state_after((4,4))},
    {state_after((4,3)),state_after((3,4)),state_after((5,4)),state_after((4,5))}
    {state_before((4,4)). action(right). wall((5,4)). wall((4,5)).}).
```
(3.6)

3.3.3 Language Bias

We now define a search space using a language bias specified by *mode declaration*. As defined in **??**, $H \subseteq S_M$ for $ILP^{context}_{LAS}$, thus in order to learn a valid move, S_M is specified as follows:

```
#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))).
```
(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*), which is 1 for all rules in ILP(RL).
var(t) is a placeholder for variable of *type* t. In **??**, we use cell for the variable type, which is grounded using on cell defined in **??** in the background knowledge. const(t) is a placeholder for 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:

```
#constant(action, right).
#constant(action, left).
#constant(action, down).
#constant(action, up).
```
(3.8)

action type is specified as constant since ILASP needs to learn different hypothesis for each action, and can be found in each context dependent examples as specified in Definition **??**.

(positive) in #modeb specifies that the predicates only appears as positive and not negation as failure, which reduces the seach space. In ILP(RL), wall(var(cell)) could appears as not wall in a hypothesis, and all other body declarations should only be positive due to the specification of (positive).
Finally, we define #max_penalty to specify the maximum size of the hypothesis. By default it is 15, and we increased to 50 as the maximum. Increasing #max_penalty allows ILASP to learn a longer hypothesis at the expense of longer computation.

3.3.4 Hypothesis

Having defined the $ILP^{context}_{LAS}$ task T = $\langle$ B, S_M, E^+, E^- $\rangle$, ILP(RL) is able to learn an hypothesis H. Since B and S_M are fixed, the hypotheses varies based on context dependent examples that the agent receives by interacting with an environment. In the early phase of ILP(RL) learning, the agent does not have many positive examples, and learns an hypothesis that is part of the full hypothesis that the agent can run in the environment. Therefore it is important to iteratively improve hypotheses. Inductive learning algorithm is executed by the following rule.

Definition 3.4. ILP(RL) runs $ILP^{context}_{LAS}$ to relearn H_{new} if and only if $\forall\langle$ e, C$\rangle \in E^+$, $\exists$A $\in$ Answer Sets (B $\cup$ C $\cup$ H) such that A extends e

where H is the current hypothesis that the agent has learnt so far. Learnt hypotheses are improved when a new positive example is added and the current hypothesis does not cover the new positive example. If the current hypothese cover the new positive example, there is no need to re-execute ILASP.
The learnt hypothesis will be used to generate a plan, which we describe in the next section.

Example 3.3.2. (Hypothesis). Using the context dependent examples illustrated in Exaple **??**, 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 **??**. The full learning task for this simple case is shown as follows:

Listing 3.1: Learning task example

```
% Background knowledge
cell((0..5, 0..5)).
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((3,4))},
     {state_after((2,4)),state_after((1,5)),
     state_after((0,4)),state_after((1,4))},
     {state_before((2,4)). action(right).
     wall((1, 4)). wall((4, 2)).}).
% 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).
```

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

$$\text{state\_after(V1) :- adjacent(left, V0, V1), state\_before(V0).} \tag{3.9}$$

Listing 3.2: Additional context dependent example

```

#pos({state_after((4,4))},
{state_after((4,3)),state_after((3,4)),
state_after((5,4)),state_after((4,5))},
{state_before((4,4)). action(right).
wall((5,4)). wall((4,5)).}).
```

This hypothesis, for example, does not explain how to move ”down”. In order to learn how to move ”down”, it needs an positive example of moving up.

3.4 Planning with Answer Set Programming

The learnt hypotheses using $ILP^{context}_{LAS}$ can be used to generate a sequence of action plan that the agent should follow. In the following subsections, we explain how to create an answer set program and use the answer sets to make a plan in a maze environment.

3.4.1 Answer Set Program

The ASP program should be constructed such that answer sets of ASP are a sequence of actions and states at each time step in the form of ASP syntax. We explain the details of how the ASP program is constructed in the planning phase.
First, we use the learnt hypotheses by $ILP^{context}_{LAS}$ 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. However, for the planning with ASP, the answer sets contains a sequence of actions and states at each time steps. In order to capture the notion of time sequences, we modify the ASP syntax by introducing T. Specifically, the following mapping is required
XX

Example 3.4.1. (Mapping of ASP syntax between $ILP^{context}_{LAS}$ and ASP).
XXX

Second, we define a choice rule of actions, which is of the form:

$$\begin{aligned}&1\{\text{action(down,T); action(up,T); action(right,T); action(left,T)}\}1\\ &\text{:- time(T), not finished(T).}\end{aligned} \tag{3.10}$$

Action is given as a choice rule, and this choice rule states that action must be one of four actions: down, up, right, or left at each time step T, as defined in the maximum and minimum integers 1. The choice rules specify that one of four actions must be true unless not finished(T) or time(T) are satisfied, as defined in the body of the rule. In RL scenarios, this means there is always 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 a maximum time steps allocated to the agent.
The maximum time steps are specified external and it is of the form:

$$\text{time}(\text{T}_\text{t}..\text{T}_\text{max}) \tag{3.11}$$

where T_t is the current time step and T_max is the maximum time steps. For example, if an agent is at time step 0, and can take actions 100 times to find a goal time is defined as time(0..100).
finished(T) determines whether the agent reaches the goal, which is defined in the following ASP form:

$$\begin{aligned}&\text{finished(T):- goal(T2), time(T), T} \geq \text{T2.}\\&\text{goal(T):- state\_at((}\text{X}_\text{goal}\text{, Y}_\text{goal}\text{), T), not finished(T-1).}\\&\text{goalMet:- goal(T).}\\&\text{:- not goalMet.}\end{aligned} \tag{3.12}$$

state_at((X_goal, Y_goal)) is the location of the goal, which is unknown to the agent in the beginning of the training. The agent explores the environment until it finds the goal location. In the other word, the agent cannot generate a plan to the goal untile the goal is found. Once the agent reaches the goal and finished(T) is satisfied, there will not be any actions at time T+1 since the body of the action choice rule defined in **??** is not satisfied.
Next, facts of walls information is provided as follows:

$$\text{wall((X,Y))} \tag{3.13}$$

As defined in Definition **??**, the agent is assumed to be able to see ajacent walls and used it as a context in positive examples in $ILP^{context}_{LAS}$. For ASP planning part, these wall information is accumulated as background knowledge as a separate repository, and used it for solving ASP.
Next, the starting state for the planning provided as part of ASP. It is simply the current location of the agent where the actions plan is a starting point.

$$\text{state\_at((}\text{X}_\text{start}\text{, Y}_\text{start}\text{), T)} \tag{3.14}$$

In addition, the definition of ajacent and cell type are also provided, which are the same as what was defined as background knoweldge of $ILP^{context}_{LAS}$ (Equation **??** and **??**) ajacents.
Next, we need to incorporate a nortion of rewards in each state. Instead of maximising the total rewards, which is the objectives of most RL methods, we use optimisation statements as follow.

$$\#\text{minimize}\{1, \text{X}, \text{T: action(X,T)}\}. \tag{3.15}$$

The use of optimisation staetment is based on the assumption that the total rewards can be maximised by searching for optimal answer sets. While this works only subset of MDP, our preliminary research focuses on solving this particular MDP problem.
Finally, we are only interested in a sequecne of actions and corresponding state as answer sets. Clingo can selectively include the atoms of certain predicates in the output and hide unselected ones. This is defined as follows:

$$\begin{aligned} &\#\text{show state\_at}/2. \\ &\#\text{show action}/2. \end{aligned} \tag{3.16}$$

Example 3.4.2. (Answer Set Program).

$$\begin{aligned} &\text{state\_at}((X_{start}, Y_{start}), T) \\ &\#\text{minimize}\{1, X, T: \text{action}(X,T)\}. \\ &\#\text{show state\_at}/2. \\ &\#\text{show action}/2. \\ &\text{wall}((X,Y)) \end{aligned} \tag{3.17}$$

3.4.2 Plan Execution

Having defined the ASP, we explain the answer sets generated by the ASP and how to use the answer sets to execute the planning.
Since the rules **??** involves $\text{state\_at}((X_{goal}, Y_{goal})$ and it is found only when the agent reaches the goal state, the plan generation is executed only after the agent finds the goal. Until the goal is found, the agent continues to explore the environment while impoving the hypotheses and collecting surrounding information. Once the agent reaches the goal, the agent is told whether the state is a terminal state, and can generate a plan using the current hypotheses by solving Answer Set Program defined in **??**.
The output of the ASP is of the form:

$$\begin{aligned} &\text{state\_at}((x_t,y_t),t),\ \text{action}(a_t,t), \\ &\text{state\_at}((x_{t+1},y_{t+1}),t+1),\ \text{action}(a_{t+1},t+1), \\ &\text{state\_at}((x_{t+2},y_{t+2}),t+2),\ \text{action}(a_{t+2},t+2), \\ &\dots \\ &\text{state\_at}(((x_{t+n},y_{t+n}),t+n),\ \text{action}(a_{t+n},t+n), \\ &\text{state\_at}((x_{goal},y_{goal}),t+n+1). \end{aligned} \tag{3.18}$$

where n is the number of time steps to take to reach the goal. *action(A,T)* tells which action the agent should take at each time. Given the answer set planning is correct, the agent follows the plan and reach the goal. The correctness of the planning is based on the correctness of the hypotheses as well as the wall surrounding information in the agent's background knowledge. For the correctness of the hypotheses, since the agent does not know all the information about the environment in the beginning of the learning, clingo might not generate correct sequence of actions leading to the goals. Also if the agent has not seen enough surrounding walls, the plan of actions might be blocked by a wall that the agent has not seen.

Example 3.4.3. (Plan Execution).
Together with hypothesis, the background knowledge will used to solve for answer sets program. However, since hypothesis is not complete, there is more than one answer set at each time step. Since one of the answer sets state_at is correct, the rest will be in the exclusions in the answer set, which is used to further improve the hypothesis in the next iteration of inductive learning.
In this example, the following is the answer set program
The answer set using the hypothesis XXX is
XXX. The answer set using the improved hypothesis is XXX, Which correctly returns a sequence of actions and predicted states.

This uncorrect plan is also a source for learning a better hypothesis. As shown in Example XX, if the hypothesis is not a full hypotheses, outputs of ASP contain lots of state_at((X,Y), T) at the same states. Given the agent is at one state at each time step, these duplicates are all included in exclusions, as defined in Definition **??**.

3.5 Exploration

Exploration is one of the active research areas in RL, and is often discussed as a tradeoff between exploration and exploitation. While the agent exploit what is already learnt and follow the best policy and action selection so far. It also needs to explore by taking a new action to discover a new state, which might make the agent discover an even shorter path and therefore higher total rewards in the future or in the long term. One of the most commonly used strategies is ϵ-greedy strategy. As described in Chapter XXX, the agent takes an random action with a probability of ϵ, which is a hiper-parameter.
ILP(RL) planning starts once the agent reaches a goal once. However it is likely that the agent has not seen all the environment and therefore is likely to generates a sub-optimal plan. Therefore, similar to RL algorithms, the agent also has to explore a new state.
In our case, given the goal is found and the model is correct, it is likely that following the plan will maximize the total rewards, To avoid the agent from being stuck in a sub-optimal plan, the agent deliberately discards the plan and takes an random action with a probability of epsilon (which is less than 1) TODO define this mathematically.
When the agent deviates from the planning, it discovers new information, which will be added to B. Exploration is therefore necessary to make sure that the agent might discovers a shorter path than the current plan, which will be demonstrated in the experiment.

Example 3.5.1. (Plan Generation). Suppose the plan of the agent is the following.
Suppose the agent decides to take an random action and moves ”up”, which may help it discover a new state. From the new state, the agent again plan to the goal from the current state. In this case, the new plan is Sub-optimal plan -¿ random exploration -¿ optimal policy.

While statistical RL algorithms, such as Q-learning or Tile-coding explained in XX, update Q-value by a facter of alpha, ILP(RL) tends to strictly follow the plan the generated. Also it is known that model-based RL tends to get stuck in a sub-optimal if the model is incorrect, ILP(RL) also needs a exploration And since the agent does not know when the perfect hypothesis in a particular environment is fully realised, it needs to keep exploring the environment even though
This strategy may not be appropriate in cases there safety is a priority (since it is random action.)

When the agent takes an random action and move into a new state, the agents creates a new plan from the new state and continue to move forward.
It is simple to implement.
The reason for using random exploration is that it can be used for both benchmark and ILP(RL) and thus enables us to do a fair comparision between them.
After taking an random action, the agent again has a probably of epsilon for taking an random action. Throughout following the new plan, there is a small probalbility of chance that the agent takes an random action.
The current framework simply uses a simple random exploration, therefore even if the agent takes an random action and goes to a different state other than the planed one, the agent does the replan from the new state and quickly correct to the original plan path. This means it is likely that the agent of ILP(RL) only explores the adjacent cells and if there is a shorter path or new state In other words, if the shorter path is far from the current state, the agent is unlikely be able to find the new state unless epsilon value is very high.

3.6 Implementation

As shown in Figure **??**, there are mainly three different components that need to be communicated: inductive learning with ILASP, planning with ASP and the external environment platform provided by VGDL with OpenAI interface. All of them are communicated through the main driver written in Python. In the following section, part of low-level implementation of each part of the framework.

3.6.1 Inductive Learning with ILASP

We use ILASP2i, a inductive learning system developed in XX. ILASP2i is an interative version of ILASP2, and is designed to scale with the numbers of examples. ILASP2i also introduces the use of context dependent examples. The latest ILASP officially available at the time of writing is ILASP3, which is designed to work on a noisy examples. Our positive examples do not contain noise, state transition positive examples, and ILASP2i was sufficient for our implemnetation. which is discussed in Section XX. Alghough ILASP2i is designed to work with a large number of examples, inductive learning part is the bottleneck of our framework in terms of computational time. We did a number of optimisation in order to mitigate this.
The first optimastion is the frequency of running ILASP. As already described in Definition **??**, ILASP is ran only if the current hypothesis does not cover all the examples accumulated so far. Because of this, inductive learning takes place at the very early stage of learning, which is highlighted in the experiments in Chapter XX.
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

```
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 specified the maximum numbers 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 recudes the search space.
- –clingo5 generates clingo 5 programs, which is fater, instead of clingo 4.3.
- –clingo "clingo5 –opt-strat=usc,stratify" specifies clingo executable with the specified options. usc, stratify is unsatisfiable-core base optimisation with stratification using Gringo [**?**], a core XXX introduced in gringo version 3. REFERENCE.
- –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 The default maximum number is 5.

Last optimisation is specifying search space.
Complexity.
TODO the number of search space XX.

3.6.2 Planning with ASP

Planning is computed using Clingo 5.

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

- –clingo "clingo5 –opt-strat=usc,stratify" specifies clingo executable with the specified options. usc, stratify is unsatisfiable-core base optimisation with stratification using Gringo [**?**], a core XXX introduced in gringo version 3. REFERENCE.
- -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[1] format

3.6.3 Environment Platform

The Video Game Definition Language (VGDL)

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 ([**?**]). The VGDL allows users to easily craft their own environments, which makes us possible to do various experiments without relying on a default environment. All objects in the games can be described as a sprite in the *SpriteSet*, and users can define the objects' properties such

[1] http://json.org/

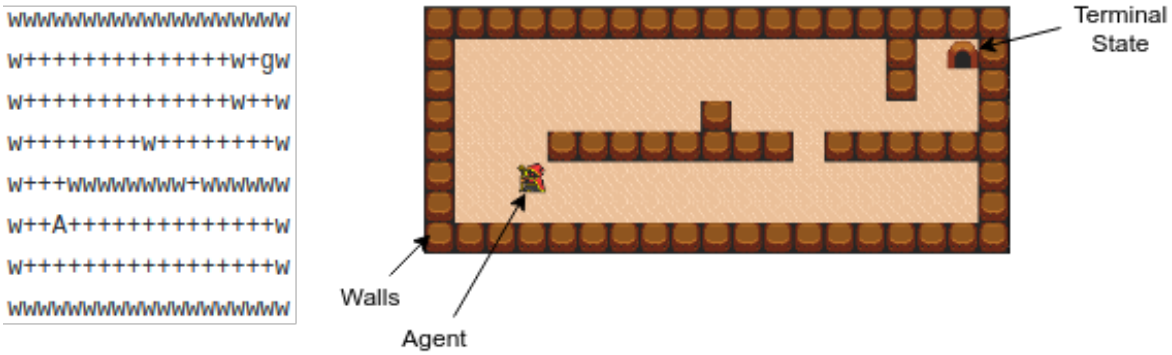

Figure 3.4: The VGDL (Video Game Definition Language) environment example

as orientation or movements. The representation of each object can be specified in *LevelMapping* and allows users to customize an original map. *InteractionSet* specify the effects of objects when two objects interact in the game. *TerminationSet* specify the conditions for ending the game. XX VDGL mapping example.
The VGDL platform provides an interface with OpenAI Gym ([**?**]), which is a commonly used benchmark platform.
The base game is a simple maze as shown in Figure **??**. There are 3 different types of cells: a goal cell, walls and paths. The agent can take 4 different actions: up, down, right and left. The environment is not known to the agent in advance, and it attempts to find the goal by exploring the environment. In all experiments, the agent receives -1 in any states except the goal state, where it gains a reward of 10. Once the agent reaches the goal, or termination state, that episode is finished and the agent start the next episode from the starting point.
TODO write more details of how you make your own environment We use PyVGDL[2], which is a high-level VGDL on top of pygame[3], a Python modules designed for writing video games. We develop our framework ILP(RL) using an environment of VDGL game.

OpenAI Gym

The communication between VGDL environment and an agent, ILP(RL), is through OpenAI Gym, which is

Listing 3.3: OpenAI gym interface

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

env = gym.make('VDGL_ENVNAME') # initialise an instance of a
 game
env.reset()
action = 0 # integer between 0 and 3
# 0: Up
# 1: Down
# 2: Left
# 3: Right
```

[2]https://github.com/schaul/py-vgdl/
[3]https://www.pygame.org

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

- env.reset() resets the game and the agent starts from the starting position. We call it 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 x and y coodinates, reward of the state, an 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 strategy between 0 and 3.
- env.render() renders one frame of the environment to see the movement of the agent in pygame.

3.6.4 The Main Driver

All of the above are connected in Python script.
The main roles of the driver is handling the communications between an environment and an agent as well as the communications within the agent.

Communication between an environment and an agent When an agent takes an action in a VDGL game environment, the output of the environment is returned by OpenAI gym environment, which is of the form:

This works the same for any RL algorithms when using OpenAI gym environment.

subprocess

Communication within the agent

Chapter 4

Evaluation

In this Chapter, we conducted 4 different experiments in simple maze environments to investigate how the ILP(RL) agent learns and reaches the optimal policy.

4.1 Experimental Setup

4.1.1 Evaluation Metrics

As introduced in **??**, our motivation is to improve the RL learning efficiency and capability of transfer learning. Therefore, these are the two main measurements for the performance of ILP(RL).
The learning efficiencies are measured in two different ways. First, the performance of ILP(RL) is compared with benchmarks in terms of convergence rate, which is measured in terms of the number of episodes that the agent requires to get to an optimal policy. The optimal policy can be measured in terms of the total reward that the agent gains at each episode. Throughout the experiments, we designed the environment such that the agent receives reward of -1 for any state except the terminal state, and receives reward of +10 for the terminal state, or the goal. For example, if the agent needs the shortest 10 actions to get to the terminal state, the maximum reward the agent could get per episode is 0 (-1 reward at each action + 10 for reaching the goal). We log the rewards at each time step and calculate the total reward at each episode. Since the agent follows an random exploration throughout the learning, the total reward never converges to the maximum in learining. Therefore at each episode, we also measure the performance without exploration to see the pure optimal policy.
Second, the convergence of learning by ILASP is measured to see the learning curve of ILP(RL), which is defined as follows:

$$\frac{\text{The cumulative number of ILASP calls per time step at episode 0}}{\text{The total number of ILASP calls in all episodes}} \tag{4.1}$$

The reason we are measuring it only at episode 0 is that empirically the agent learns most of the target hypotheses within episode 0 and there is no hypothesis refinement after episode 0. This gives a normalised convergence rate of ILASP learning with the maximum 1, and we plot it across each time steps. If an indutive learning happens after episode 0, this included in the deniminator, and the maximum convergence plot is strictly less than 1.
RUNTIME

4.1.2 Benchmarks

We use different benchmarks for learning evaluation and transfer learning evaluation.
For learning evaluation, we use two existing RL methods as benchmarks: Q-learning and tile-coding. Q-learning is widely used RL technique, and given the environments used for the experiments are discrete and deterministic, this method is sufficient for our experiments. XX the Q-values are initialised as XX. (Optimistic and pessimistic initialisation.)
Another benchmark is tile coding, which is a type of linear function approximation techniques described in Chapter XX. The reason for using an extra benchmark is that the performance comparision of ILP(RL) with q-learning might not be a fair comparision, since ILP(RL) has one extra assumption: the agent knows surrounding information (whether there are walls in adjacent cells), which is not a common assumption for Q-learning. Thus we incorporate the same surrounding information as features, and update the weights of each feature as a learning. We compare the performance of ILP(RL) with these two methods.
For transfer learnig evaluation,

4.1.3 Parameters

Parameter	ILP(RL)	Benchmarks
The number of episode	100	100
Time steps per episode	250	250
The number of experiments	30	30
Alpha	N/A	0.5
Epsilon	0.1	0.1

Table 4.1: List of parameters used in the experiments

Agents are trained in 250 time steps with 100 episodes, and conducted the same experiment 30 times in each environment. The resulting average score is plotted for each experiment. All the matrices used in the experiments are summarised in Table **??**. The number of episode is set such that both the benchmarks as well as ILP(RL) eventually eaches the optimal policy. The number of time steps should be sufficient for the both algorithms to reach the terminal state by the random exploration, which we set 250 time steps for all experiments. If the agent does not find the terminal state by 250 time steps, the agent receives the reward of -250 and start the next episode from the starting point. If the agent reaches the terminal state within 250 time steps, it receives the reward of 10 and start the next episode with the same starting point.
Starting point is fixed every episode. We conducted several experiments using different environments to highlight each aspect of the algorithm.
Each experiment is conducted 30 times and the perform is averaged across the experiments, since the performance of the agent is affected by the randomness of the exploration, and ILP(RL) is highly dependent on how quickly the agent finds the goal.

4.2 Learning Evaluation

4.2.1 Experiment 1: Setup

The purpose of the first experiment is to highlight how ILP(RL) agent learns hypotheses in ILASP. The environment are designed as a simple maze where the terminal state is located

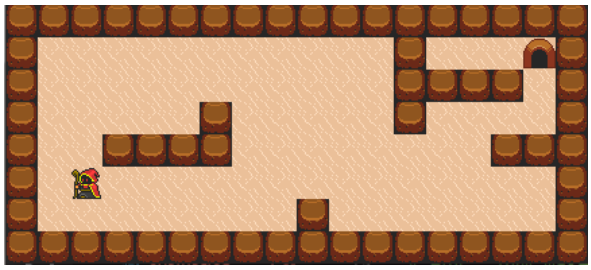

Figure 4.2: Game environment for experiment 1

the right uppper corner as shown in Figure **??**.
The agent's starting location is the same at every episodes. The agent's goal is to learn hypotheses for valid move of the game, and to use the hypotheses to correctly plan a sequence of actions to reach the goal. The shortest path between the agent's starting point and the terminal state is 18 steps, thus the maximum total reward the agent could gain is -8.

4.2.2 Experiment 1: Result

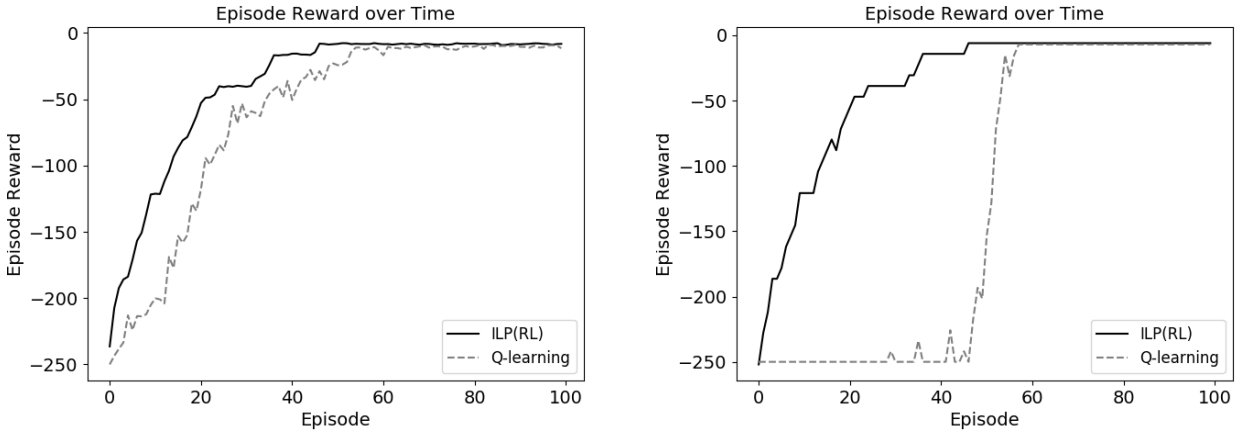

Figure 4.3: Results of experiment 1: learning curve (left) and testing (right)

Figure **??** shows the traning performance between ILP(RL) and Q-learning. The convergence rate of ILP(RL) is faster than Q-learning: ILP(RL) reaches the maximum reward between 40 and 50 episodes, whereas Q-learning reaches the same level at between 60 and 70 episodes. This is because unlike Q-learning where the value function is updated with the rate of alpha, ILP(RL) gradually builds the model of the environment and use the background knowledge to accurately plan. This result is also consistent with the general notion that model-based RL is more data-efficient than model-free RL. The same trend is also shown in Figure **??**, where we measure only the performance of the policy without random exploration. The reason Q-learning converges slower in the testing than that of training is that Q values are initialised to 0, which means that XXX. Overall this results shows that ILP(RL) converges to the optimal policy faster than benchmarks in a simple scenarios, achieving more data-efficient learning.

Listing 4.1: Hypotheses for experiment 1

```
1 state_after(V1):-adjacent(right, V0, V1), state_before(V1),
2                  action(right), wall(V0).
```

```
3 state_after(V0):-adjacent(right, V0, V1), state_before(V0),
4                   action(left), wall(V1).
5 state_after(V1):-adjacent(down, V0, V1), state_before(V1),
6                   action(down), wall(V0).
7 state_after(V1):-adjacent(up, V0, V1), state_before(V1),
8                   action(up), wall(V0).
9 state_after(V0):-adjacent(right, V0, V1), state_before(V1),
10                  action(right), not wall(V0).
11 state_after(V0):-adjacent(left, V0, V1), state_before(V1),
12                  action(left), not wall(V0).
13 state_after(V0):-adjacent(down, V0, V1), state_before(V1),
14                  action(down), not wall(V0).
15 state_after(V0):-adjacent(up, V0, V1), state_before(V1),
16                  action(up), not wall(V0).
```

In addition to the data-efficient learning, what the agent has learnt with ILP(RL) is expressive. Learnt hypotheses are shown in **??**, which is the rule of the game and easy to understand for human users. Since the learnt hypothesis is a general concept, which can be used in a different environmet. This transfer learning capability is also described in Experiement 3 and 4.

Next, we see how the agent learns the hypotheses during over time. We plot the learning convergence for ILASP at episode 0 in Figure **??**, measured in terms of the number of hypothesis refinement to reach the final hypothesis as shown in **??**. This shows that the agent learns most of the hypotheses at the episode 0. Since the two variables that the agent require to construct optimal answer set planning are the hypothese and background knowledge, in this case the location of the walls, the reason that the agent reaches the maximum reward at between 40 and 50 episodes, is mostly dependent on how quickly the agent finds the goal location. Even though the agent acquires the full hypotheses at episode 0, without the terminal state, the agent continues to randomly explore the environment. Since our exploration strategy is expilon random choice, there is a promissing that a better exploration strategy further accelerates the learning process.

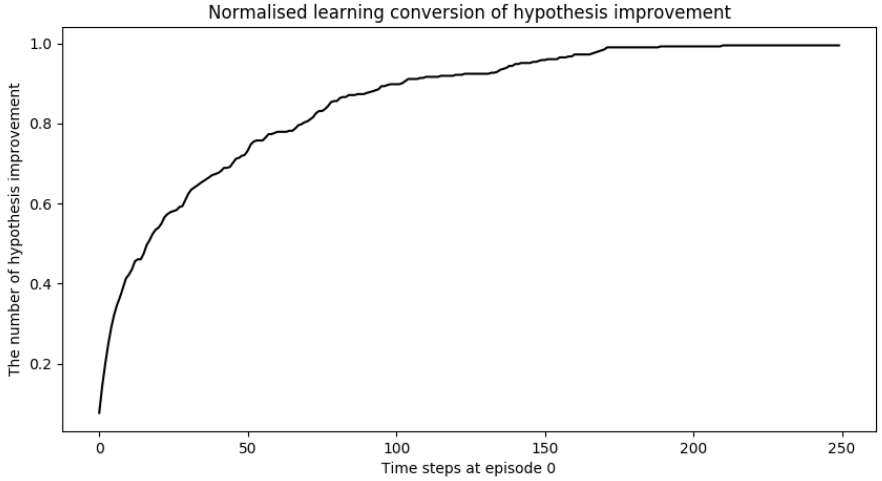

Figure 4.4: Normalised learning convergence by ILASP for experiment 1

Finally, we compare the runtime of two algorithms.
This result confirms that inductive learning part is likely the bottleneck in terms of
This issue may not be cretical in cases where the time between the time steps is not an issue. If the performace is measured in terms of computation time rather than the number of iterations, ILP(RL) is not likely to perform exisitng RL algorithms. The average runtime of inductive learning is 5.579 seconds, and there are on average 12.83 times inductive learning per episode. The plannning part is not a bottleneck of ILP(RL) compared to Q-leaning, but still takes longer time than Q-learning. While the planning with this environment is relatively simple planning. As discussed in XX, ASP also has its scalability issues. XXX

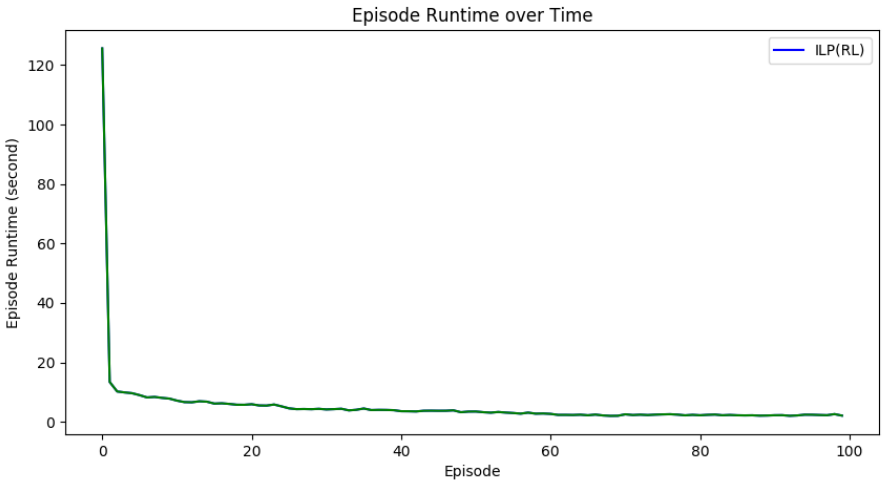

Figure 4.5: Runtime comparision

4.2.3 Experiment 2: Setup

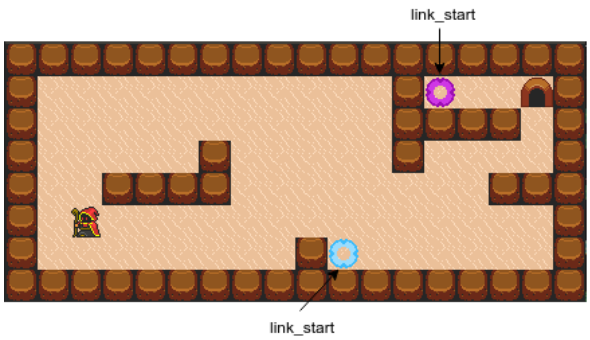

Figure 4.6: Game environment for experiment 2

Experiment 2 was conducted to see if the agent find a optimal path of using a teleport. The environment is the same as experiment 1 except the presence of teleport link. In the environment shown in Figure **??**, there are two ways to reach the goal: using a normal path to get the goal located on the top right corner, or using a telport. The environment is designed such that using a teleport is a shorter path and therefore gives higher total reward.

Compared to Experiment 1, two extra search spaces and concepts are added as follows:

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

Where teleport links are added to the environment. 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 hypothesis. The full learning task for this experiment is in Appendix XX.
In VDGL game environmet, when the agent steps onto link_start, it immediately takes the agent to link_dest. This means that the agent moves two states in one time step. Since our ASP translation part takes the state transition as state_before and state_after, we deal with this situation such that the agent receives two positive experiences in this time step rather than two time steps.
Also link_start and link_dest need to be stored in background knowledge rather than as contex examples, because the framework need to generate exclusions regarding the link behaviour (TODO EXPLAIN MORE). link locations need to be available for all positive examples so that ILASP correctly learn non-link for normal valid move rather than teleport, which is shown in Figure XX below.
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 Experiment 2: Result

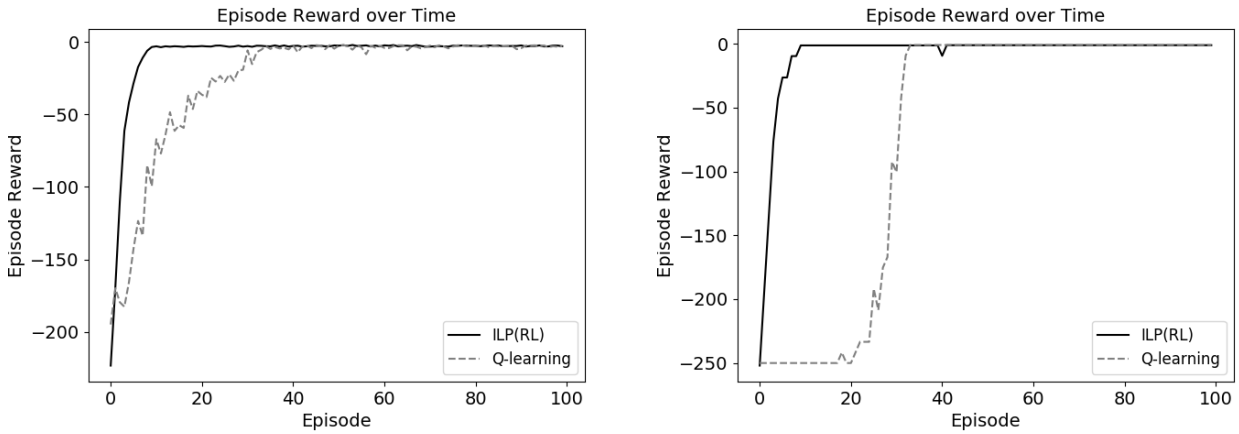

Figure 4.7: Results of experiment 3: learning curve (left) and testing (right)

The same as the Experiment 1, both training and test performance converges faster than that of Q-learning.

Listing 4.2: Incomplete hypotheses for experiment 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),
```

```
6                   action(left), not wall(V0).
7 state_after(V1):-adjacent(up, V0, V1), state_before(V0),
8                   action(down), not wall(V1).
9 state_after(V0):-adjacent(up, V0, V1), state_before(V1),
10                  action(up), not wall(V0).
11 state_after(V1):-adjacent(left, V0, V1), state_before(V1),
12                  action(left), wall(V0).
13 state_after(V1):-adjacent(down, V0, V1), state_before(V1),
14                  action(down), wall(V0).
15 state_after(V1):-adjacent(up, V0, V1), state_before(V1),
16                  action(up), wall(V0).
```

To highlight the learning process of the new concept of teleport link, Figure **??** is an intermediate incomplete hypothesis learnt by ILASP. These hypotheses are generated just after the agent steps onto the link_start. However, the first hypothesis says when link_dest is available state_after is true. Since link_dest is available in background knowledge rather than context, when solving for answer sets to generate a plan, it generates incorrect state_after at every time step.
However, as shown in Algorithms XX, these generated state_after are all incorrect and therefore will be added to exclusions of the next positive examples. These exclusions will later refines hypotheses and results in Figure **??**, the final complete hypotheses.
Learnt hypotheses are shown in Figure **??**:

Listing 4.3: Complete hypotheses for experiment 2

```
1 state_after(V1):-link_start(V0), link_dest(V1),
2                   state_before(V0).
3 state_after(V1):-adjacent(left, V0, V1), state_before(V0),
4                   action(right), not wall(V1).
5 state_after(V0):-adjacent(left, V0, V1), state_before(V1),
6                   action(left), not wall(V0).
7 state_after(V1):-adjacent(up, V0, V1), state_before(V0),
8                   action(down), not wall(V1).
9 state_after(V0):-adjacent(up, V0, V1), state_before(V1),
10                  action(up), not wall(V0).
11 state_after(V0):-adjacent(left, V0, V1), state_before(V0),
12                  action(right), wall(V1).
13 state_after(V1):-adjacent(left, V0, V1), state_before(V1),
14                  action(left), wall(V0).
15 state_after(V0):-adjacent(up, V0, V1), state_before(V0),
16                  action(down), wall(V1).
17 state_after(V1):-adjacent(up, V0, V1), state_before(V1),
18                  action(up), wall(V0).
```

Compared to the Experiment 1, there are two new hypotheses due to the presence of the teleport links. These learnt hypotheses are also applicables to an environment where there is no link, such as a game in experiment 1. In this case, the first two hypotheses in Figure XX are never be used since the body predicates relating to link_start(V0), link_dest(V1) are never be satisfied.
Figure **??** shows the learning convergence of inductive learning at episode 0. Similar to Experiment 1, most of inductive learning occur at the beginning of the episode, as shown

in Figure **??**. This result confirms that the agent learns inductive learnig at the beginning of episode.
Despite the fact that the size of the environment is the same as Experiment 1, there is a significant increase of runtime at the beginning of episode. This is because of the increase of search space as we added extra language bias.
There are increase of runtime after episode 0. This is due to the fact that the agent later discovers a teleport link, which is a new hypothesis to be learnt. TODO is this really true??
The average runtime of inductive learning is 95.47 seconds, and there are on average 16.23 times inductive learning per episode.

Parameter	ILP(RL)	Benchmarks
Runtime per episode (seconds)	95.47	1
Average ILASP time (seconds)	XXX	N/A
The number of ILASP calls	16.23	N/A
Hypothesis space	XXX	N/A

Table 4.8: Comparision of runtime

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 symbolic representations that the agent needs to learn in the environment.

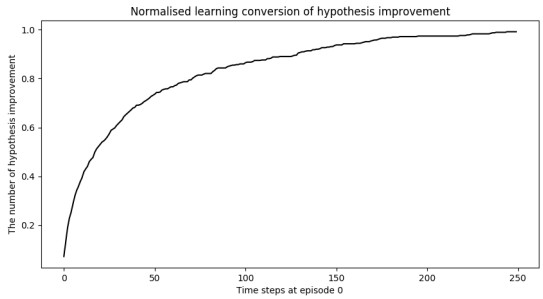

Figure 4.9: Normalised learning convergence by ILASP for experiment 2

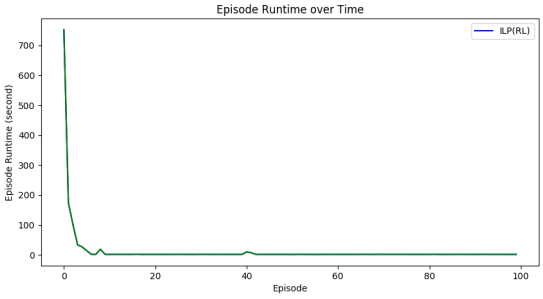

Figure 4.10: Runtime comparision

4.3 Transfer Learning Evaluation

4.3.1 Experiment 3: Setup

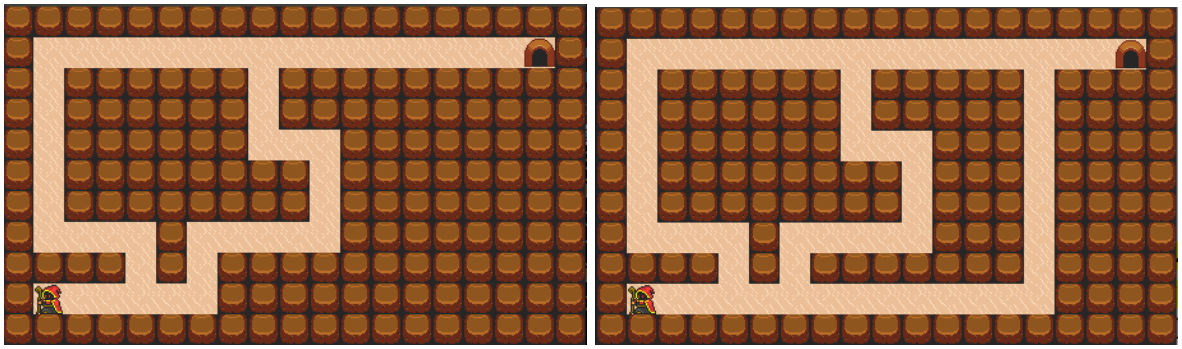

Figure 4.11: Game environment for experiment 3: before (left) and after (right) transfer learning

In Experiment 3, we investigated the possibilites of transfer learning betweeen similar environments. We trained the agent using the environment on the left in Figure **??**, and transfer the learnt hypothesis as well as positive examples to a new environment on the right in Figure **??** The learnt hypothesis is valid move of the game and a general concept that is applicable to any similar environments. Positive examples are also transferred since if there is a new hypothesis that the agent needs to learn in a new environment, the agent needs to refine the hypothesis by running ILASP in the new environment. Thus the all the positice examples are also transferred as well as learnt hypotheses. Background knowledge are 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 goal position is the same as in the first game, 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 **??**.
We use four different agents 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 transferred hypotheses, examples, but does not know the location of the terminal state.
- Agent(noTL)$_{\text{noGoal}}$: The agent with no transferred information, neighther the hypotheses nor the location of the terminal state is known to the agent.
- Q-learning

While this is a limited transfer learning since the goal position 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.
Figure XX 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 **??**. The positive examples to be transferred are those the agent accumulated in the left environment, XXX examples in total.

Listing 4.4: Hypotheses for experiment 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).
```

4.3.2 Experiment Result 3

The result is shown in Figure **??** and **??**. For Agent(TL) since the complete hypotheses are already known to the agent as well as the planning, the agent can do planning from episode 0. The only information it needs is background knowledge for planning, namely the locations of the walls, which is quickly acquired and reached the maximum total reward at the very beginning of episodes.
The next best agent in terms of the convergence to the maximum total reward is Agent(noTL)$_{\text{Goal}}$. Since the terminal state is konwn to the agent, the agent can do the planning from episode 0. However, the agent needs to learn the hypotheses. The reason that the convergence rate is almost the same as that os Agent(TL) is that the ILP(RL) learns the complete hypothese in episode 0, as observed in Experiment 1 and 2, there is no difference between Agent(TL) and Agent(noTL)$_{\text{Goal}}$.
What makes a difference for the convergence is whether the agent knows the terminal state, which can be seen by compering between Agent(noTL)$_{\text{Goal}}$ and Agent(noTL)$_{\text{noGoal}}$. The difference in terms of the iterations is that Agent(noTL)$_{\text{noGoal}}$ needs to find the terminal state first before starting the planning, which is a random exploration.
This observation shows that there is a promissing potential for improving the exploration strategy to find the terminal state as soon as possible. We discuss this in XX.
There was no ILASP calls in the new environments since the transferred hypotheses are already optimal and cover all the examples the agent encounters in the new environment.

4.3.3 Experiment 4: Setup

The transferred hypotheses are the same as that in Experiment 3, and the positive examples were collected the environment on the left in Figure **??**. The new environment, shown on the right in Figure **??**, is the same except that there is teleport links. This is a new concept that did not exist in the trained environment and therefore the agent needs to learn it after the hypothesis is transferred. The same as Experiment 3, we use four different agents.

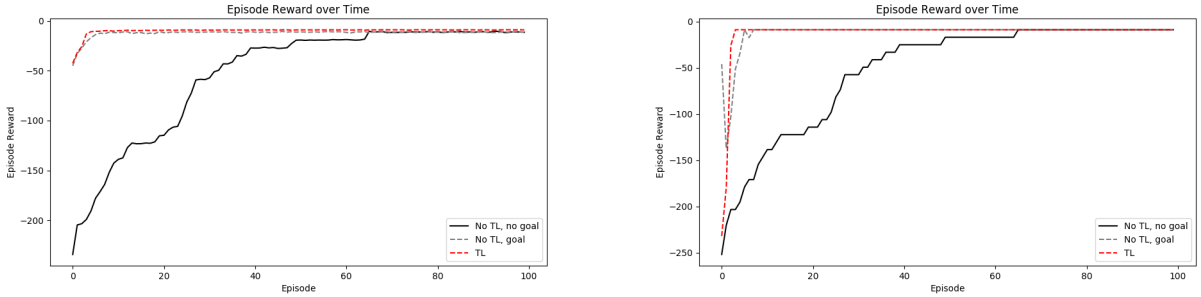

Figure 4.12: Results of experiment 3: learning curve (left) and testing (right)

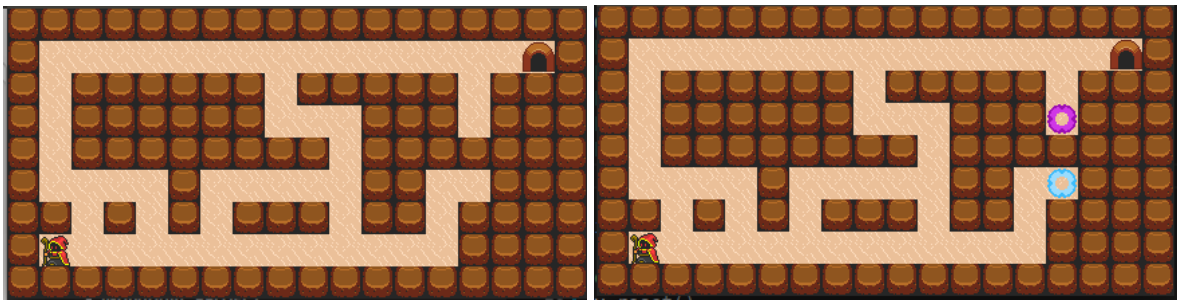

Figure 4.13: Game environment for experiment 4: before (left) and after (right) transfer learning

4.3.4 Experiment Result 4

Figure **??** show the results of the performance. The agent is able to succeffully learn the new concept and quickly finds the optimal policy at the early episodes. This confirms that the transferred agents learns on top of what is already learnt. This experiment shows that the hypotheses is transferable even in cases where there is something 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 even when the positive examples in the previous environment is transferred, since the surrounding information is within the context rather than background knowledge. This shows the power of context dependent examples in RL senarios.

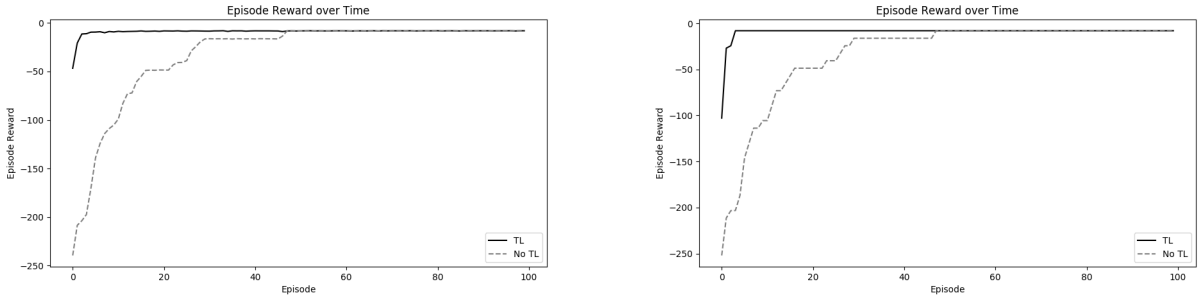

Figure 4.14: Result of experiment 4 (learning curve)

The new hypotheses the agent learns are the same as that of Experiment 2. The hypothesis containing link_start and link_dest is what the agent learnt in the new environment.

Listing 4.5: Hypotheses for experiment 4

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

4.4 Discussion

We investigaged the properties of ILP(RL) using a simple maze environment. While the development of ILP(RL) is still in early stage and only a proof-of-concept, we observe both strengths as well as weakness of the current approach. We summarise both of these in the following sections.

4.4.1 Strengths of ILP(RL)

on a simple maze environment, which is the main target of our framework.
Although this is the first time and inductive logic programming is applied into reinforcement leaning and there are new interesting property for ILP(RL), there are two major limitations with the current framework.

Faster learning convergence The agent with ILP(RL) learns the valid move of the environment at the very early stage of learning and, as soon as it finds the terminal state, is able to genrate a plan as a sequence of actions to reach the goal. While this is a proof-of-concept approach, this way of RL is a new and the experiments show that this is a promissing direction of the research.

Transfer Learning Unlike existing RL algorithms, where it learns value functions ro Q values, ILP(RL) learns a valid move as a hypotheses, which can be applied to similar but different environments. We confimed with the experiments and, especially when the goal is known to the agent We also show that the agent can learn a new hypothesis on top of the transferred ones, which is very flexible in terms of applicability of the learnt hypotheses.

Symbolic learning Since both planning and learning can be expressed in ASP syntax, the learning process is easy to understand for human users, and the learnt hypotheses are a very general rule of the game.

In simple environments, we show that the agent learns rule of the game faster than existing RL algorithms, learnt concepts is easy to understand for human users.

4.4.2 Limitations of the current framework

Although the this first version of the ILP(RL) framework using ILASP and ASP show potentials, there are a number of limitations with the current frameworks. Some of these limitations are further elaborated in Further Research in the concluding chapter.
It was implement from scratch.

Learning time While we show that ILP(RL) improves the learning convergence in terms of the number of episodes required, the computational time is significantly longer than that of benchmarks due to the computation required for inductive learning with ILASP. This limitation indicates that ILP(RL) may not be suitable in an environment where there is a moving object based on time rather than time steps.

Unlike existing reinforcement learning, out algorithm refines hypothesis at every time steps within the same episode. Thus even though the efficiency in terms of the number of iteration is higher, training time within each iteration tends to be lower.

Scalability issue for more complex environment ILP framework is known to be less scalable. The current framework is tested in a relatively simple environments, and proven to be work better than RL algorithsm in terms of the number of episodes that is needed to converge to an optimal policy. However, learning in each episode is relatively slower than that of RL. This is shown in XXX, which shows average learnint time for ILASP.

This limitation is theoretically discussed in XXX, where the complexity of deciding satisfiability is As shown in Experiment 1 and Experiment 2, adding two language bias significantly increased the runtime of the algorithm, since the search space grows significantly with respect to the language bias.

$\sum_2^P$-complete. Since there is no negative examples used in our current framework, the complexity is NP-complete.

Whereas Q-learning update value function in the same way whether there is a new concept such as teleport links.

Another question remains to how to extend the framework to more realistic senarios. RL works in more complex environments such as 3D or real physical environment, whereas the experiences of the agent in the current framework need to be expressed as ASP syntax, thus expressing continuous states rather than discrete states is challenging.

Requirements of assumptions Language bias While most of existing reinforcement learning works in different kinds of environment without pre-configuration, our algorithm needs to define search space for learning hypothesis. As explained in the experiment 3, it was necessary to add two extra modeb before training. Thus the algorithm may not be feasible in cases where these learning concepts were unknown or difficult to define with language bias.

In addition, not only it needs search space, surrounding information is assumed to be known to the agent. While this assumption may be reasonable in many cases, this is not common in traditional reinforcement learning setting.

limited MDP The current framework does not make use of rewards the agent collects and mainly uses the location of the goal for planning. In some senarios, there may not be a termination state (goal) and instead there may be a different purpose to gain these rewards. Since the current implementation is dependent on finding the goal for planning rather than maximing total rewards, which is the common objective for most of RL algorithms, the application of the current framework may be limited to particular types of problems.

There are some promissing solutions for some of these limitations, which we discuss in the next chapter.

Chapter 5

Related Work

In this section, we review related work in logic programming, reinforcement learning combined with ASP, symbolic RL, and some model-based RL, in order to motivate our new approach. relational reinforcement learning.

The combination of inductive logic programming and reinforcement learning has its root in relational reinforcement learning.

The combining logic-based learning with reinforcement learning has its roots in what is called relational reinforcement learning (RRL) [**?**].

RRL equipped with generalisation of inductive logic programming.

RRL is based on first-order logic, and does not cope with negation as failure or non-monotonic reasonings.

However, most RRL algorithms focus on planning. RRL incorporates relational representations of states and actions to generalize Q-function and More recent work on using relational representation in RL is the paper in XX, which combines RRL and DRL in order to overcome the interpretability and an ability to generalise of DRL. The proposed architecture uses self-attention mechanism to reason about the relations bewtween entities in rhw environment to help imroving policy. While RLL was applied to XX, this approach also shows an promissing direction of using ILP in RL.

[**?**] 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 newtworks (CNNs) [**?**] and an autoencoder, 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 threhold of feature activation values, given that the games were geometrically simple. Thus this approach would not scale in geometrically complex games. Second, using deep neural network front-end might also cause a problem. As demonstrated in [**?**], a single irrelevant pixel could dramatically influence the state through the change in CNNs. In addition, while proposed method successfully used symbolic representations to achieve more data-efficient learning, there is still the potential to apply symbolic learning to those symbolic representations to further improve the learning efficiency, which is what we attemp to do in this paper. [**?**] further explored this symbolic abstraction approach 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.
[**?**] added relational reinforcement learning, a classical subfield of research aiming to combining reinforcement learning with relational learning or Inductive Logic Programming, which added more abstract planning on top of DSRL approach. The new mode was then applied to much more complicate game environment than that used by [**?**]. This idea of adding planning capability align with our approach of using ILP to improve a RL agent. We explore how to effectively learn the model of the environment and effectively use it to facilitate data-efficient learning and transfer learning capability.
Another approach for using symbolic reinforcement learning is storing heuristics expressed by knowledge bases [[**?**]). An agent lerans the concept of *Hierarchical Knowledge Bases (HKBs)* (which is defined in more details in [**?**] and [**?**]] 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 sym-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 [**?**], 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 consequnces of each possible action. The more details of theoretial explanation of this approach is described in XX. This approach focused more on efficient update of the Q function using ASP, and does not make use of inductive learning.
Similar works were conducted for learning XX in PAPER, for XX in XXX and for XX in XX. All of them are based on experiment in using robot.
Our framwork focues on ASP-based ILP with RL, which has not been explored.
[**?**] proposed an architecture for interactively discovering previously unkonwn axioms using reinforcement learnig. Axioms represent domain dynamics of preconditions and expected outcomes, as well as the relationship among actions of the agents and objects in the domain. These discovered axioms are used to generate more general axioms that can be use for subsequence reasoning. uses ASP program to encode a decision tree induction as well as relational representation in order to . The extension of [] is

Chapter 6

Conclusion

6.1 Summary of Work

In this paper, we developed a new RL algorithm by applying ILP to develop a new learning process. We summarise some of the works we have done throughout the project.

- We started off by looking at existing symbolic reinforcement learning approaches to understand the research fields and see the potentials of improving further researchs. Due to advances of ASP-based ILP frameworks and there is no existing works of applying ASP-based ILP into RL senarios, this motivates us to pursuing the potentials of ILP to solve RL problems. Because of the flexibility and experise of ILASP, we decided to apply ILASP as our core learning framework.

- We considerered the target hypotheses and how to construct learning tasks. Our target learning is a valid move of the agent, and whic can be expressed with state transition for each action. Also all the components of learning tasks have to be in ASP-syntax, we developed a Python engine for translating all output of the environment into ASP-syntax.

  surrounding information

- 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 a shortest path to a terminal state.

- We considered which kind of RL problems we would like to test with our new framework. Since this is a new proof-of-concept and testing core concepts were required, we chose a custom-maze game provided by VDGL and created original environments.

- We tested our new framework in a various maze environments to highlight each aspect of the algorithm. We show that ILP(RL) learns faster than benchmarks for finding a shortest path, and show a capability of transfer learning. While the experiments were conducted in a simple limited conditions, we show some promissing potentials of the ILP-based approach for RL problems.

We used a latest ILP algorithm called ILASP, Learning from Answer Set Program to iteratively improve hypotheses.

6.2 Further Research

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

Better exploration strategy We used a random exploration strategy for both algorithms in order to compare the performance of ILP(RL) with existing RL algorithms. As shown in the experiments, however, the convergence of ILP(RL) is heavily dependent on finding a terminal state in order to start planning part. In RL research, exploration stategy is another active research field and a more sophisticated exploration strategy would facilitate the learning of ILP(RL) such as Boltzman approach, Count-based and Optimistic Initial value TODO REFERENCE).

Experiments on different environments Since the purpose of this paper is develop the initial framework of ILP(RL) and experiments on the core part of the algorithms, furhter experiments on different game environments are required for more robustness of the framework, such as the presence of dynamic enermies in the environment or non-stational environment. This might releave other aspects of using ILP in RL context and further bridge the gaps between the fielf of RL and ILP.

Inclusion of reward as part of learning The current implementation of ILP(RL) can only solve a reduced MDP, where there is only one rewards and they can be solved by minising the answer sets. In many RL environments, however, there can be more than one type of rewards. In order to improve our current framework to solve other types of MDP, the rewards themselve can be included as part of inductive learning using *Learning from ordered answer sets*, denoted ILP_{LOAS}. This framework is an extension of ILASP by allowing the learning of ASP programs with weak constraints. 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. This element of preference learning can be used for different types of rewards. Weak constraints (Calimeri et al 2013) allows With using ILP_{LOAS}, we could further generalise the current framework by removing ASP planning part and inductive learnig would return a sequene of actions. Since our implementation in this paper is a proof-of-concept and generality of hypotheses are useful to highlight the strengths of our approach in terms of transfer learning. This flexibility of inductive learning is a promissing direction.

Value Iteration Approach

Reducing the initial assumption The current assumption requires the assumption of adjacent definition as a background knolwedge. This could be, however learnt using ILASP. Also the concept of adjacent is another general concpet and can be useful in many different environments.

Appendix A

Ethics

To our best knowledge, there is no particular ethical considerations for this particular research listed in Table XX. However, the field of RL ia an active research area and has been increasingly applied in industries these days, and therefore ethical frameworks for RL will be required for both academic research as well as industry applications.
Also the experiemnts of our algorithm were conducted using a game environment rather than real applications (e.g robots).
Rather compared to existing reinforcement learning methodologies.
Also there are a number of AI researchers discussing the ethics of AI in general. Since RL is considered to be part of AI research, these ethical considerations might be also applied.

	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 tasks

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

```
state_after(V1) :- link_dest(V1).
cell((0..7, 0..6)).
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))).
#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).
```

Appendix C

Answer Set Program

This is the full learning task for ILASP in the experiment 1. The syntax and time are added for planning purpose.
TODO put comment on the code

```
1{action(down,T); action(up,T); action(right,T); action(left,T); action(non,T)}1 :-
time(T), not finished(T).
#show state_at/2.
#show action/2.
finished(T):- goal(T2), time(T), T ¿= T2.
goal(T):- state_at((5, 1), T), not finished(T-1).
goalMet:- goal(T).
:- not goalMet.
time(0..30).
cell((0..6, 0..5)).
#minimize1, X, T: action(X,T).
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)).
state_at((1, 4), 3).
```