

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Symbolic Reinforcement Learning using Inductive Logic Programming

Author:
Kiyohito Kunii

Supervisor:
Prof. Alessandra Russo
Mark Law
Ruben Vereecken

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

September 2018

Abstract

Reinforcement Learning (RL) has been applied and proven to be successful in many domains. However, most of current RL methods face limitations, namely, low learning efficiency, inability to use abstract reasoning, and inability of transfer learning to similar environments. In order to tackle these shortcomings, we introduce a new 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 hypothesis is highly expressive and transferable to a similar environment. While there are a number of past papers that attempt to incorporate symbolic representation to RL problems in order to achieve efficient learning, there has not been any attempt of applying 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 promising, and open up an avenue for future research directions.

Acknowledgements

I would like to thank Prof. Alessandra Russo for supervising my project, and for her enthusiasm for my work and invaluable guidance throughout.

I would also like to thank Mark Law for his expertise on inductive logic programming and answer set programming, as well as many fruitful discussions, and Ruben Vereecken for his expertise on reinforcement learning and for providing me with advice and assistance for technical implementation.

Contents

List of Figures and Tables

Chapter 1

Introduction

1.1 Motivation

Reinforcement learning (RL) is a subfield of machine learning concerned 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-and-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 presence of language bias which defines what the logic program can learn. Second, the learnt hypothesis is expressed using a symbolic representation and therefore is easy 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 learning in RL scenarios.

Since the recent ILP frameworks 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 learning 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 divided into the following high-level objectives:

1. **Translation of state transitions into ASP syntax.**

In RL, an agent interacts with an environment by taking an action and observing a state and rewards (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 applicability 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 stochastic 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 learning 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:

$$\underbrace{h}_{\text{head}} \leftarrow \underbrace{a_1, a_2, \dots, a_n}_{\text{body}} \quad (2.1)$$

h and a_1, \dots, a_n are all atoms. h is the *head* of the rule and a_1, \dots, a_n are the *body* of the rule.

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

$$\underbrace{h}_{\text{head}} \leftarrow \underbrace{a_1, a_2, \dots, a_n, \text{not } b_{n+1}, \dots, \text{not } b_{n+k}}_{\text{body}} \quad (2.2)$$

where h, a_1, \dots, a_n , and b_{n+1}, \dots, b_{n+k} are all atoms.

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

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

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

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.

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

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)

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

$ground(P)$ 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)

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

Where X is a set of atoms. $\text{ground}(P)$ is

$\text{male}(\text{john}) \leftarrow \text{not female}(\text{john}).$
 $\text{male}(\text{anna}) \leftarrow \text{not female}(\text{anna}).$
 $\text{female}(\text{john}) \leftarrow \text{not male}(\text{john}).$
 $\text{female}(\text{anna}) \leftarrow \text{not male}(\text{anna}).$

The first step removes $\text{male}(\text{anna}) \leftarrow \text{not female}(\text{anna}).$ and $\text{female}(\text{john}) \leftarrow \text{not male}(\text{john}).$

$\text{male}(\text{john}) \leftarrow \text{not female}(\text{john}).$
 $\text{female}(\text{anna}) \leftarrow \text{not male}(\text{anna}).$

The second step removes negation atoms from the body.

Thus reduct P^X is $(\text{ground}(P))^X = \{\text{male}(\text{john}), \text{female}(\text{anna}).\}$

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)

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

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

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 $\text{not } p$ (*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:

$$\leftarrow a_1, \dots, a_n, \text{not } b_n, \dots, \text{not } b_{n+k} \quad (2.3)$$

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 $\text{ground}(P)_x$, the empty head becomes \perp , 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:

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

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 $l \leq |X \cap \{h_1, \dots, h_m\}| \leq u$??.

Reduct in ASP 4 steps. ??

Optimisation statement is useful to sort the answer sets in terms of preference, which is of the form:

$$\begin{aligned} &\# \text{minimize}[a_1=w_1, \dots, a_n=w_n] \text{ or} \\ &\# \text{maximize}[a_1=w_1, \dots, a_n=w_n] \end{aligned} \quad (2.5)$$

where w_1, \dots, w_n are integer weights and a_1, \dots, 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 ??.

$$B \wedge H \models E \quad (2.6)$$

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* task¹ 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. $H \in \text{ILP}_{\text{cautious}} \langle B, E^+, E^- \rangle$ if and only if there is at least one answer set A of $B \cup H$ ($B \cup H$ is satisfiable) such that for every answer set A of $B \cup H$:

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

Example 2.2.1. (Cautious Induction)

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

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

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)

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

Neither of $\text{situation}(\text{john}, \text{awake})$ nor $\text{situation}(\text{john}, \text{sleep})$ is false in all answer sets. In this example, the hypothesis only contains $\text{person}(\text{john})$. 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. $H \in \text{ILP}_{\text{brave}} \langle B, E^+, E^- \rangle$ if and only if there is at least one answer set A of $B \cup H$ such that:

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

Example 2.2.3. (Brave Induction)

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

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

¹This 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.

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 constraints must rule out all answer sets that meet the conditions of the Brave Induction.

Example 2.2.4. (Limitation of Brave Induction)

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

In order to learn the constraint hypothesis $H = \{ \leftarrow not\ situation(P, awake), super_person(P) \}$, 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:

$$E = \langle e^{inc}, e^{exc} \rangle. \quad (2.7)$$

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

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

$$T = \langle B, S_M, E^+, E^- \rangle \quad (2.8)$$

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 $ILP_{LAS}(T)$, and a hypothesis H is an inductive solution of $ILP_{LAS}(T) \langle B, S_M, E^+, E^- \rangle$ such that:

1. $H \subseteq S_M$
2. $\forall e^+ \in E^+ : \exists A \in \text{Answer Sets}(B \cup H)$ such that A extends e^+
3. $\forall e^- \in E^- : \nexists A \in \text{Answer Sets}(B \cup H)$ 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 S_M 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 $ILP_{LAS}(T) \langle B, S_M, E^+, E^- \rangle$, and is based on two fundamental concepts: *positive solutions* and *violating solutions*.

A hypothesis H is a positive solution if and only if

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

A hypothesis H is a violating solution if and only if

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

Given both definitions of positive and violating solutions, $ILP_{LAS} \langle B, S_M, E^+, E^- \rangle$ is positive solutions that are not violating solutions.

Context-dependent Learning from Answer Sets

Context-dependent learning from answer sets ($ILP_{LAS}^{context}$) 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:

$$\langle e, C \rangle \quad (2.9)$$

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

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

1. $H \subseteq S_M$

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

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

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.

$ILP_{LAS}^{context}$ is used in ILASP2i, which we use for our new framework.

Example 2.2.6. ($ILP_{LAS}^{context}$).

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 ??.

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

$$H_t = S_1, R_1, A_1, \dots, A_{t-1}, S_t, R_t. \quad (2.10)$$

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

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \dots, S_t]. \quad (2.11)$$

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.

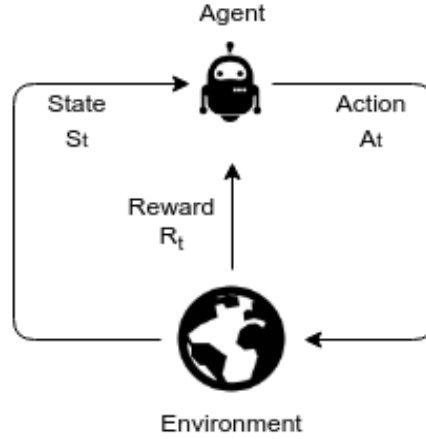


Figure 2.1: The interaction between an agent and an environment

Definition 2.18. Markov Decision Process (MDP)

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

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

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

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2.12)$$

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

2.3.2 Policies and Value Functions

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

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

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

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

Example 2.3.1. (Value Functions).

XXX

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

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

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

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

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

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

Example 2.3.2. (State-action value Functions).

XXX

2.3.3 Temporal-Difference (TD) Learning

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

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

where $R_{t+1} + \gamma V(S_{t+1})$ is the target for TD update, which is a biased estimate of $v_{\pi}(S_t)$, and $\delta = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called *TD error*, which is the error in $V(S_t)$ available at time $t+1$. Since the TD learning only needs to know the estimate of one step ahead and does not need the final outcome of the episodes (also called TD(0)), it can learn online after every time step. TD learning also works without the terminal state, which is the goal for an agent. TD(0) is proved to converge to v_{π} in the table-based case.

Q-learning is off-policy TD learning introduced 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:

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

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-learning is off-policy because a policy being evaluated and improved is different from the policy being used to take an action.

Algorithm 2 Q-learning for learning $Q : X \times A \rightarrow \mathbb{R}$

```

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

```

$$Q(s_t, a_t) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t] \quad (2.20)$$

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

2.3.4 Model-based and Model-free RL

A *model* is a representation of an environment that an agent can use to understand how the environment should look like, which is T and R in MDP. There are two different types of RL methods: *model-based* and *model-free* RL method. Model-based RL is where, when a model of the environment is known, the agent uses the model to plan for a series of actions to achieve the agent's goal. The model itself can be learnt by interacting with the environment by taking actions, which return states and rewards, and the parameters of the action models can be estimated with maximum likelihood methods [?]. Using a model, an agent can do planning to generate or improve a policy for the modelled environment.

By contrast, model-free RL is where a model is unknown and the agent learns to achieve the goal solely by interacting with the environment without using the model. Thus the agent knows only possible states and actions, and the transition state and reward probability functions are unknown. When the model of the environment is correct, the model-based RL agent can reach an optimal policy without trial-and-error interactions with the environment. When the model is not a true representation of the environment, or the true MDP, however, the planning algorithms will not lead to the optimal policy, but a sub-optimal policy.

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.

2.3.5 Transfer Learning

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 knowledge in the first task. Since training a RL algorithm tends to be time consuming and computational expensive, transfer learning allows the trained agent to be applied in a different setting. While transfer learning is an active research areas in machine learning, not many have been done in RL. 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. 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 knowledge in a new task.

Chapter 3

Framework

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

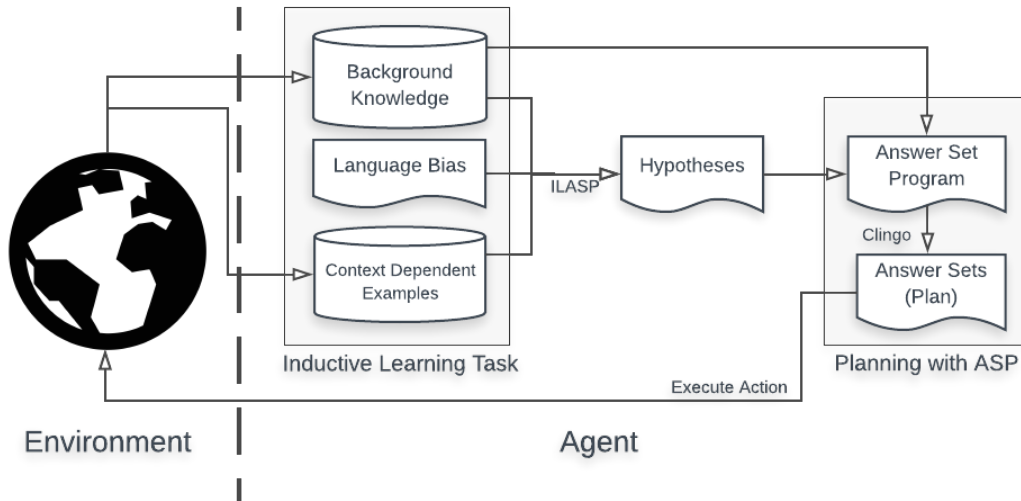


Figure 3.1: ILP(RL) overview

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.

The first step is inductive learning. An agent interacts with an unknown environment, and receives state transition experiences as context dependent examples. Together with pre-defined background knowledge and language bias, these examples are used to inductively learn and improve hypotheses, which are state transition function in the environment.

The 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 terminal state. The agent remembers these information as background knowledge, and, together with the learnt

Algorithm 3 ILP(RL) Algorithm

```

1: Input:
2:   Current state  $s$ 
3:   Actions  $A = \{1, \dots, n_a\} A : X \rightarrow A$ 
4:   States transition  $T : X \times A \rightarrow X$ 
5:    $\epsilon \in [0, 1]$ 
6:   Background knowledge for ILASP and ASP  $B$ 
7:   Language bias  $S_M$ 
8:
9: procedure ILP(RL)( $S, A, T, \epsilon, B, S_M$ )
10:    $H = \emptyset$ 
11:    $E = \emptyset$ 
12:   Start  $s \in S$ 
13:   while  $s$  do is not terminal
14:      $a \leftarrow \text{POLICY}(B, H, s, A, \epsilon)$ 
15:      $\triangleright$  observe the next state  $s'$  by taking action  $a$ 
16:      $s' \leftarrow T(s, a)$ 
17:      $\triangleright$  generate a positive example
18:      $e \leftarrow \text{MakeDCPI}(s', s, a)$ 
19:      $E = E + e$ 
20:      $\triangleright$  if the current  $B \cup H$  without language bias does not cover  $E$ , update  $H$ 
21:     if  $\text{ILASP}(B \cup H, S_M = \emptyset, E) == \text{UNSATISFIABLE}$  then
22:        $H = \text{ILASP}(B, S_M, E)$ 
23:      $\text{Update}(B)$ 

```

hypotheses, uses them to make an action plan by solving an ASP program. The plan is a sequence of actions and it navigates the agent in the environment.

The agent iteratively executes this cycle in order to learn and improve the hypotheses as well as an action planning. Mechanisms of each step are explained in details in the following sections.

3.2 Environment

Since there is no existing base frameworks for ILP(RL), our focus on this project is to develop a preliminary version of the framework. Therefore, we use a very simple environment 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 a 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 experiences: a reward R_t , the next state S_{t+1} and surrounding information of the current state. The base environment mainly consists of three different elements: a goal, walls and paths. The goal cell is the terminal state and the agent receives a positive reward when the agent receives a negative reward in any states except the terminal state. When the agent reaches the terminal state, the agent receives a positive reward and the current 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 state transition in the environment using inductive learning. 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 for a more complex environment is not considered in this project and it is discussed in Further Research in Section ??.

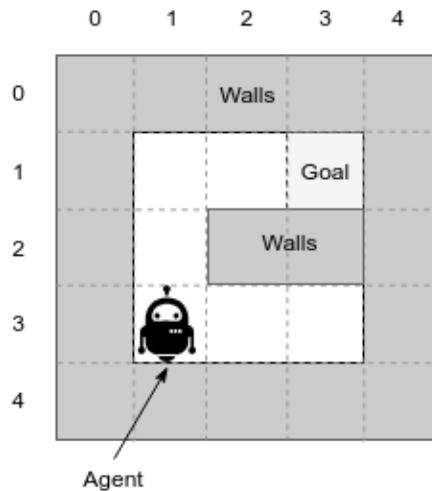


Figure 3.2: 5×5 grid maze example

3.3 Inductive Learning Task

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

$$\begin{aligned}
&\text{state_after}(V1):-\text{adjacent}(\text{right}, V0, V1), \text{state_before}(V1), \text{action}(\text{right}), \text{wall}(V0). \\
&\text{state_after}(V0):-\text{adjacent}(\text{right}, V0, V1), \text{state_before}(V1), \text{action}(\text{right}), \text{not wall}(V0). \\
&\text{state_after}(V1):-\text{adjacent}(\text{left}, V0, V1), \text{state_before}(V1), \text{action}(\text{left}), \text{wall}(V0). \\
&\text{state_after}(V0):-\text{adjacent}(\text{left}, V0, V1), \text{state_before}(V1), \text{action}(\text{left}), \text{not wall}(V0). \\
&\text{state_after}(V1):-\text{adjacent}(\text{down}, V0, V1), \text{state_before}(V1), \text{action}(\text{down}), \text{wall}(V0). \\
&\text{state_after}(V0):-\text{adjacent}(\text{down}, V0, V1), \text{state_before}(V1), \text{action}(\text{down}), \text{not wall}(V0). \\
&\text{state_after}(V1):-\text{adjacent}(\text{up}, V0, V1), \text{state_before}(V1), \text{action}(\text{up}), \text{wall}(V0). \\
&\text{state_after}(V0):-\text{adjacent}(\text{up}, V0, V1), \text{state_before}(V1), \text{action}(\text{up}), \text{not wall}(V0).
\end{aligned} \tag{3.1}$$

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. $\text{adjacent}(D, V0, V1)$ specifies that $V0$ is next to $V1$ in the direction of D . For example, $\text{adjacent}(\text{right}, V0, V1)$ means $V0$ is right next to $V1$. $\text{wall}(V)$ and $\text{not wall}(V)$ specify whether a state is a wall or not wall in a state V respectively. We describe the details of how to construct the background knowledge, context dependent examples and the language bias in the following sections, and these are the necessary components for the agent to learn the target hypotheses. 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 the inductive learning. In order to learn the state transition for each direction as shown in ??, we need to define the meaning of "being next to" a state. This is defined as a rule *adjacent*, which is of the form:

$$\begin{aligned}
&\text{adjacent}(\text{right}, (X+1,Y),(X,Y)):-\text{cell}((X,Y)), \text{cell}((X+1,Y)). \\
&\text{adjacent}(\text{left}, (X,Y), (X+1,Y)):-\text{cell}((X,Y)), \text{cell}((X+1,Y)). \\
&\text{adjacent}(\text{down}, (X,Y+1),(X,Y)):-\text{cell}((X,Y)), \text{cell}((X,Y+1)). \\
&\text{adjacent}(\text{up}, (X,Y), (X,Y+1)):-\text{cell}((X,Y)), \text{cell}((X,Y+1)).
\end{aligned} \tag{3.2}$$

where cell corresponds to a state, and X and Y represent x-coordinate and y-coordinate respectively. The rules ?? are given as background knowledge and allow the agent to understand the relation of two adjacent states. $\text{cell}((X,Y))$ is defined as follows:

$$\text{cell}((0..X, 0..Y)). \tag{3.3}$$

where, X and Y are the size of width and height of an environment respectively. For example, a grid maze shown in Figure?? has a height and width of 4, thus the type cell is defined in the background knowledge as $\text{cell}((0..4, 0..4))$.

3.3.2 Context Dependent Examples

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

$$\#pos(\{E_{ILP(RL)}^{inc}\}, \{E_{ILP(RL)}^{exc}\}, \{C_{ILP(RL)}\}) \quad (3.4)$$

It is equivalent to context-dependent partial interpretation (CDPI) in $ILP_{LAS}^{context}$. 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 component in CDPI in ILP(RL) is defined as follows:

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

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

- $s_{t+1} = s_{t+1}^*$
- $\forall A \in AS(B \cup H_t \cup C_{ILP(RL)}) \mid s_{t+1} \notin A$

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

- $s_{t+1} \neq s_{t+1}^*$
- $\exists A \in AS(B \cup H_t \cup C_{ILP(RL)}) \mid s_{t+1} \in A$

where s_{t+1}^* is the true next state where the agent is at $t+1$, B is the background knowledge, H_t is the hypotheses at t , S is all the states in the environment.

Example 3.3.1. (Context dependent examples).

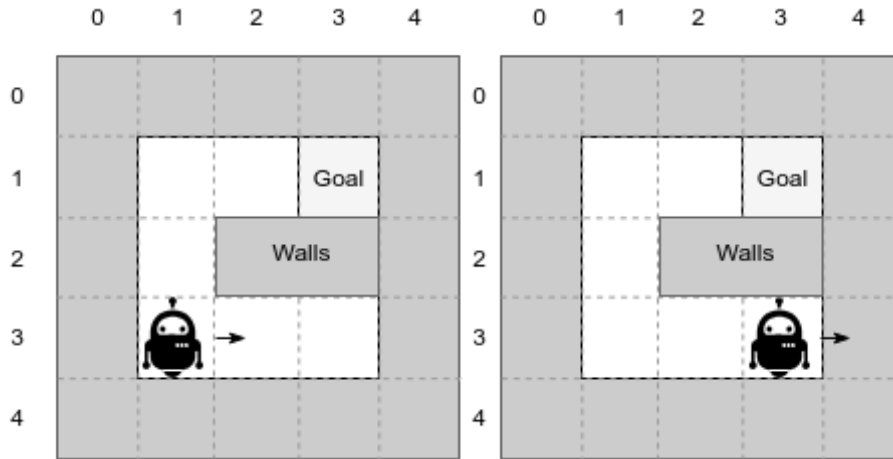


Figure 3.3: 5×5 grid maze example (context dependent example)

We use a simple 5×5 grid maze environment to highlight how an agent gains a positive example. Suppose $H_t = \emptyset$ and an agent takes an action "right" to move from (1,3) to (2,3) cell, as shown on the left in Figure ?. a_t is "right", s_t is (1,3) and s_t^* is (2,3). According to the Definition ??, $A = \emptyset$ since $H_t = \emptyset$, thus $state_after((2,3))$ is in $E_{ILP(RL)}^{exc}$, $A = \emptyset$ is there

is no exclusions in this case. $C_{ILP(RL)}$ includes a_t , s_t and adjacent walls of s_t . The following positive example is generated.

$$\begin{aligned} &\#pos(\{state_after((2,3)), \\ &\quad \{\}, \\ &\quad \{state_before((1,3)).\ action(right).\ wall((0,3)).\ wall((1,4)).\}\}) \end{aligned} \quad (3.5)$$

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

$$\begin{aligned} &\#pos(\{state_after((4,4)), \\ &\quad \{state_after((4,3)), state_after((3,4)), state_after((5,4)), state_after((4,5))\} \\ &\quad \{state_before((4,4)).\ action(right).\ wall((5,4)).\ wall((4,5)).\}\}). \end{aligned} \quad (3.6)$$

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 $ILP_{LAS}^{context}$, thus in order to learn the target hypotheses, S_M is specified as follows:

$$\begin{aligned} &\#modeh(state_after(var(cell))). \\ &\#modeb(1, adjacent(const(action), var(cell), var(cell)), (positive)). \\ &\#modeb(1, state_before(var(cell)), (positive)). \\ &\#modeb(1, action(const(action)), (positive)). \\ &\#modeb(1, wall(var(cell))). \end{aligned} \quad (3.7)$$

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 we specify 1 for $\#modeb$ in ILP(RL). $var(t)$ is a placeholder for a variable of type t . In ??, we use $cell$ for the variable type, which is grounded using $cell$ specified in ?? in the background knowledge. $const(t)$ is a placeholder for a constant term of type t , and type t must be specified as $\#constant(t, c)$, where c is a constant term., $const(t)$ is specified as follows:

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

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

(positive) in `#modeb` specifies that the body predicates only appear as positive and not negation as failure, which reduces the search space. We use (positive) for all `#modeb` except `wall`. Thus `wall(var(cell))` could appear as not `wall` in a hypothesis, and all other body predicates should only be positive.

Finally, we define `#max_penalty` to specify the maximum size of the hypothesis. By default it is 15, however our target hypotheses defined in ?? is larger than 15, and thus it is increased to 50. Increasing `#max_penalty` allows ILASP to learn longer hypotheses at the expense of longer computation.

3.3.4 Hypothesis

Having defined the $ILP_{LAS}^{context}$ task $T = \langle B, S_M, E^+, E^- \rangle$, ILP(RL) is able to learn hypotheses H . Since B and S_M are fixed, the hypotheses vary based on context dependent examples that the agent accumulates by interacting with an environment. For example, in the early phase of learning, the agent does not have many positive examples, and learns a hypothesis that is subset of the full target hypotheses that the agent could learn in the environment. Thus ILP(RL) needs to iteratively improve the hypotheses as illustrated in Figure ?. As specified in line 19 of Algorithm , $ILP_{LAS}^{context}$ is executed when the current hypotheses do not cover the new context example. If the current hypotheses cover the new positive example, there is no need to re-execute ILASP.

The learnt hypotheses will be used for ASP planning, which we describe in the next section.

Example 3.3.2. (Hypothesis). Using the context dependent examples illustrated in Example ??, 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

```

1 % Background knowledge
2 cell((0..5, 0..5)).
3 adjacent(right, (X+1,Y), (X,Y)):-cell((X,Y)),cell((X+1,Y)).
4 adjacent(left, (X,Y), (X+1,Y)):-cell((X,Y)),cell((X+1,Y)).
5 adjacent(down, (X,Y+1), (X,Y)):-cell((X,Y)),cell((X,Y+1)).
6 adjacent(up, (X,Y), (X,Y+1)):- cell((X,Y)),cell((X,Y+1)).
7 % Context dependent examples
8 #pos({state_after((3,4))},
9     {state_after((2,4)),state_after((1,5)),
10     state_after((0,4)),state_after((1,4))},
11     {state_before((2,4)). action(right).
12     wall((1, 4)). wall((4, 2)).}).
13 % Language bias
14 #modeh(state_after(var(cell))).
15 #modeb(1, adjacent(const(action),
16     var(cell),var(cell)), (positive)).
17 #modeb(1, state_before(var(cell)), (positive)).
18 #modeb(1, action(const(action)), (positive)).
19 #modeb(1, wall(var(cell))).
20
21 #max_penalty(50).
```

```

22
23 #constant(action, right).
24 #constant(action, left).
25 #constant(action, down).
26 #constant(action, up).

```

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

$$\text{state_after}(V1) \text{ :- adjacent(left, } V0, V1), \text{ state_before}(V0).$$

After having learnt the hypothesis above, suppose the agent takes another action and gains one extra example as shown in Listing ??.

Listing 3.2: Additional context dependent example

```

1 #pos({state_after((4,4))}, % Inclusion
2   {state_after((4,3)),state_after((3,4))},% Exclusion
3   state_after((5,4)),state_after((4,5))},
4   {state_before((4,4)).action(right). % Context
5   wall((5,4)).wall((4,5)).}).

```

The current hypothesis, does not cover the new positive example, ILP(RL) runs ILASP with the new positive example. The revised hypotheses are as follows:

XXX

ILP(RL) keeps improving the hypotheses this way until the target hypotheses are learnt.

3.4 Planning with Answer Set Programming

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

3.4.1 Answer Set Program

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

Listing 3.3: Answer set program for ILP(RL)

```

1 % Hypotheses are given from ILASP.
2
3 % Choice rule for choosing an action at each time T.
4 1{action(down,T);
5   action(up,T);
6   action(right,T);
7   action(left,T)}1 :-time(T), not finished(T).
8
9 % T is the current time step, T_max is the maximum time steps.

```

```

10 % The agent can do a planning between these time steps.
11 time(T..T_max).
12
13 % Check whether the agent reaches a terminal state.
14 finished(T):- goal(T2), time(T), T >= T2.
15 goal(T):- state_at((X_terminal, Y_terminal), T), not finished(T
    -1).
16 goalMet:- goal(T).
17 :- not goalMet.
18
19 % walls are cumulatively collected from
20 % context dependent examples
21 wall((X_1, Y_1)).
22 wall((X_2, Y_2)).
23 ...
24
25 % Current state of the agent at time T,
26 % which is the start of the planning
27 state_at((X_start, Y_start), T).
28
29 % The output of ASP should include only state_at and action
30 #show state_at/2.
31 #show action/2.
32
33 % Find a shortest path to a terminal state, thus
34 % minimise the number of actions to reach the terminal state.
35 #minimize{1, X, T: action(X,T)}.
36
37 % The size of the maze
38 cell((0..X_width, 0..Y_height)).
39
40 adjacent(right, (X+1,Y), (X,Y)):-cell((X,Y), cell((X+1,Y))).
41 adjacent(left, (X,Y), (X+1,Y)):-cell((X,Y), cell((X+1,Y))).
42 adjacent(down, (X,Y+1), (X,Y)):-cell((X,Y), cell((X,Y+1))).
43 adjacent(up, (X,Y), (X,Y+1)):-cell((X,Y), cell((X,Y+1))).

```

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

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

Table 3.4: ASP syntax mapping between inductive learning and ASP planning

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

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

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

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

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

The converted hypothesis is used to generate a sequence of actions and states to the "right" direction for more than two time steps for ASP planning.

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

$$\begin{aligned} &1\{\text{action}(\text{down}, T); \text{action}(\text{up}, T); \text{action}(\text{right}, T); \text{action}(\text{left}, T)\}1 \\ &:- \text{time}(T), \text{not finished}(T). \end{aligned} \quad (3.9)$$

The choice rule states that an 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 rule specifies that one of four actions must be true unless not finished(T) or time(T) are satisfied, as defined in the body of the rule. Therefore there is always an action to be taken until the agent reaches a terminal state. When the agent reaches a terminal state, finished(T) is satisfied, otherwise time step T exceeds a maximum time step allocated to the agent.

The maximum time step is specified externally and it is of the form:

$$\text{time}(T_t..T_{\max}) \quad (3.10)$$

where T_t is the current time step and T_{\max} is the maximum time step. For example, if an agent is at time step 0, and can take actions up to 100 time step within an episode until it finds a terminal state, time is defined as time(0..100).

finished(T) determines whether the agent reaches the goal, which is specified as follows:

$$\begin{aligned} &\text{finished}(T) :- \text{goal}(T_2), \text{time}(T), T \geq T_2. \\ &\text{goal}(T) :- \text{state_at}((X_{\text{terminal}}, Y_{\text{terminal}}), T), \text{not finished}(T-1). \\ &\text{goalMet} :- \text{goal}(T). \\ &:- \text{not goalMet}. \end{aligned} \quad (3.11)$$

state_at((X_{terminal} , Y_{terminal})) is the location of the terminal state, which is unknown to the agent before the learning starts. The agent explores the environment until it finds the terminal state.

Once the agent reaches the goal at time t and $\text{finished}(T)$ is satisfied, there will not be any actions at time $t + 1$ since the body of the choice rule defined in ?? is not satisfied.

The locations of walls are what the agent collects from the context of context dependent examples, which are specified as follows:

$$\text{wall}((X,Y)) \quad (3.12)$$

These wall information are accumulated as background knowledge, and they are used for ASP planning.

The starting state for the planning is provided as part of ASP. It is the current location of the agent at time t when the plan is generated. It is specified as follows:

$$\text{state_at}((X_{\text{start}}, Y_{\text{start}}), T) \quad (3.13)$$

In addition, the definition of adjacent and cell type are also provided, which is the same as the definition of adjacent in inductive learning phase defined in Equation ?? and ??.

Next, we need to incorporate a notion of rewards. Instead of maximising the total rewards, which is the objectives of most RL methods, we assume that the rewards for any states except the terminal state in an environment are -1, so that we can use an optimisation statement in ASP in order to maximise the total rewards. Due to this assumption, the current framework of ILP(RL) works only subset of MDP, and our preliminary research focuses on solving this particular MDP problem. The optimisation statement in ILP(RL) is specified as follows:

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

Finally, we are only interested in a sequence of actions and corresponding states as the output of the ASP program. Clingo can selectively include the atoms of certain predicates in the output and hide other predicates. This is specified as follows:

$$\begin{aligned} \# \text{show state_at}/2. \\ \# \text{show action}/2. \end{aligned} \quad (3.15)$$

This way the answer sets of the ASP program contains only `state.at` and `action`.

3.4.2 Plan Execution

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

$$\begin{aligned} & \text{state_at}((x_t, y_t), t), \text{ action}(a_t, t), \\ & \text{state_at}((x_{t+1}, y_{t+1}), t+1), \text{ action}(a_{t+1}, t+1), \\ & \text{state_at}((x_{t+2}, y_{t+2}), t+2), \text{ action}(a_{t+2}, t+2), \\ & \dots \\ & \text{state_at}(((x_{t+n}, y_{t+n}), t+n), \text{ action}(a_{t+n}, t+n), \\ & \text{state_at}((x_{\text{goal}}, y_{\text{goal}}), t+n+1). \end{aligned} \quad (3.16)$$

where n is the number of time steps taken to reach the goal. $\text{action}(A, T)$ tells the agent which action to 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 information in the agent's background knowledge. For example, if the agent does not have a hypothesis for how to move right, the ASP plan does not generate an answer set containing `action(right)`. Similarly, if the agent has not seen enough surrounding walls, and one of the `state(x,y)` in the plan is a wall, the agent fails to follow the plan. If this happens, either the current hypotheses were incomplete, or the agent needs more background knowledge. The agent constantly refine the hypotheses by accumulating more context dependent examples, which also contains a new wall information. This way the ASP planning also improves and the agent is able to correctly navigate through the maze using a ASP plan. This completes the whole cycle of ILP(RL) framework shown in Figure ??.

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

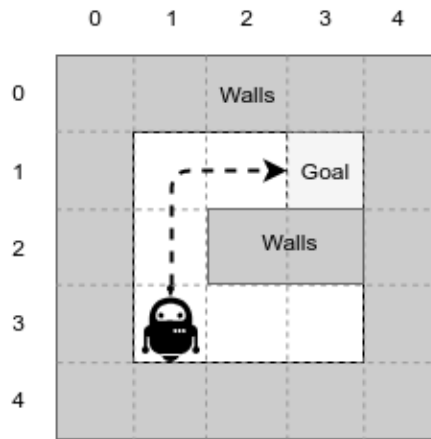


Figure 3.5: 5×5 grid maze example (ASP planning)

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

Listing 3.4: Example of ASP program

```

1 % Learnt hypotheses
2 state_at(V0, T+1):-time(T),adjacent(right, V0, V1),
3                     state_at(V1,T),action(right,T),not wall(V0).
4 state_at(V0, T+1):-time(T),adjacent(left, V0, V1),
5                     state_at(V1,T),action(left,T), not wall(V0).
6 state_at(V0, T+1):-time(T),adjacent(down, V0, V1),
7                     state_at(V1,T),action(down,T), not wall(V0).
8 state_at(V0, T+1):-time(T),adjacent(up, V0, V1),
9                     state_at(V1,T),action(up,T), not wall(V0).
10
11 1{action(down, T);
12   action(up, T);
13   action(right, T);
14   action(left, T)}1 :- time(T), not finished(T).
15
16 % The maximum time step is 4
17 time(0..4).

```

```

18
19 finished(T):- goal(T2), time(T), T >= T2.
20 goal(T):- state_at((3, 1), T), not finished(T-1).
21 goalMet:- goal(T).
22 :- not goalMet.
23
24 % Walls that the agent knows so far
25 wall((1, 0)).
26 wall((2, 0)).
27 wall((3, 0)).
28 wall((0, 1)).
29 wall((0, 2)).
30 wall((0, 3)).
31 wall((2, 2)).
32 wall((3, 2)).
33
34 % Starting state
35 state_at((1, 3), 0).
36
37 #show state_at/2.
38 #show action/2.
39 #minimize{1, X, T: action(X,T)}.
40
41 % Size of the maze
42 cell((0..4, 0..4)).
43
44 adjacent(right, (X+1,Y),(X,Y)) :- cell((X,Y)), cell((X+1,Y)).
45 adjacent(left, (X,Y), (X+1,Y)) :- cell((X,Y)), cell((X+1,Y)).
46 adjacent(down, (X,Y+1),(X,Y)) :- cell((X,Y)), cell((X,Y+1)).
47 adjacent(up, (X,Y), (X,Y+1)) :- cell((X,Y)), cell((X,Y+1)).

```

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

```

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

```

3.5 Exploration

In this section, we explain exploration of ILP(RL). While the agent can learn the hypotheses and execute a planning by interacting with the environment as shown in Figure ??, there are mainly two reasons why ILP(RL) requires exploration. First, the ILP(RL) agent can only

do ASP planning only if it finds a terminal state. Therefore the agent needs to continue exploring the environment until the terminal state is found. Second, even if the terminal state is known and the agent generates a planning based on the background knowledge and the hypotheses, this does not guarantee that the agent always finds an optimal plan. While the agent exploits what is already learnt and follows the best plan so far, the agent 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 long term.

One of the simple exploration policies in RL is called ϵ -greedy policy. ϵ -greedy policy states that an agent follows the optimal policy with probability of $(1 - \epsilon)$ where $\epsilon \in [0, 1]$, and the agent takes a random action with a probability of ϵ . ϵ is a hyper-parameter and is chosen externally. Based on ϵ -greedy policy, we define the exploration policy for ILP(RL) in ??:

Algorithm 4 ILP(RL) Exploration Policy

```

1: procedure POLICY
2:   if Terminal state is found then
3:      $\triangleright \text{rand}(0,1)$  generates a random number between 0 and 1
4:     if  $\text{rand}(0, 1) < \epsilon$  then
5:        $\triangleright \text{randInt}(0,A)$  generates a uniformly random integer between 0 and  $|A|$ 
6:        $i = \text{randInt}(0, |A|)$  return  $a_i$ 
7:     else
8:        $a_i = \text{ASP Plan of ILP(RL)}$  return  $a_i$ 
9:   else
10:     $i = \text{randInt}(1, |A|)$  return  $a_i$ 

```

state, the agent generates a new planning from the new state. Since the ILP(RL) agent does not know when the optimal plan is found, it is necessary for ILP(RL) to continue ϵ -greedy strategy.

3.6 Implementation

In this section, we describe how we developed ILP(RL) framework. 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 VGDG with OpenAI interface. All of them are communicated through the main driver written in Python. In the following section, we explain each component of ILP(RL) in details.

3.6.1 Inductive Learning with ILASP

We use ILASP2i, an inductive learning system developed in XX. ILASP2i is an interactive 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 noisy examples. Our positive examples do not contain noise, state transition positive examples, and ILASP2i was sufficient for our implementation, which is discussed in Section XX. Although 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 optimisation 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 reduces the search space.
- `--clingo5` generates Clingo 5 programs, which is faster, 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 JSON¹ format

3.6.3 Environment Platform

The Video Game Definition Language (VGDL)

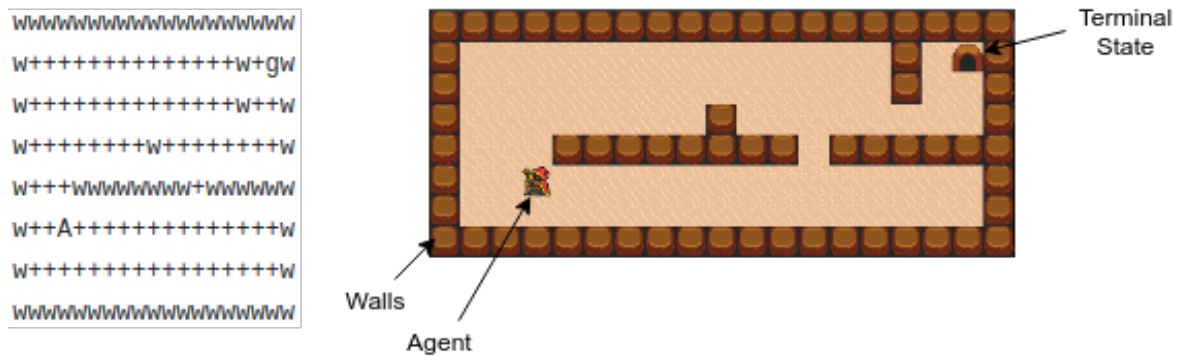


Figure 3.6: Map sketch of a VGDL game (left) and its high-level representation (right)

We use the Video Game Definition Language (VGDL), which is a high-level description language for 2D video games providing a platform for computational intelligence research ([?]). The VGDL allows users to easily craft their own environments, which makes us possible to do various experiments without relying on a default environment.

The base game we used to implement ILP(RL) is shown in Figure???. The map sketch is a plain text file and is easy to modify the configuration of the game.

Listing 3.5: VGDL description of a maze game

```

1  BasicGame
2      SpriteSet
3          floor > Immovable img=oryx/backBiege
4          structure > Immovable
5          goal > color=GREEN img=oryx/door2
6          avatar > MovingAvatar img=oryx/mage1
7          wall > Immovable img=oryx/dirtWall_0 autotiling=True
8
9      InteractionSet
10         random wall structure > stepBack
11         avatar wall > stepBack
12         goal avatar > killSprite scoreChange=1
13         avatar portentry > teleportToExit
14
15     TerminationSet
16         SpriteCounter stype=goal limit=0 win=True

```

¹<http://json.org/>

```

17         SpriteCounter stype=avatar limit=0 win=False
18
19     LevelMapping
20         g > floor goal
21         w > floor wall
22         A > floor avatar
23         + > floor

```

The behaviours of the game can be specified using VGDL as shown in ???. All objects in the games can be described as a sprite in the *SpriteSet*, where users can define the objects' properties. *InteractionSet* specify the effects of objects when two objects interact in the game. *TerminationSet* specify the conditions for ending the game. The representation of each object can be specified in *LevelMapping* and allows users to customise an original map. We use PyVGDL², which is a high-level VGDL on top of pygame³, a Python modules designed for writing video games.

OpenAI Gym

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

In this report, an agent receives a reward of -1 for any state except the terminal state, and receives reward of +10 for the terminal state, or the goal.

Listing 3.6: OpenAI gym interface

```

1  import gym # import OpenAI gym package
2  import gym_vgdl # used to connect VGDL and OpenAI gym
3
4  num_episodes = 100 # num_episodes is specified by users
5  time_steps = 100 # time_steps is specified by users
6
7  # initialise an instance of a VDGL game
8  env = gym.make('VDGL_ENVNAME')
9
10 for i_episode in range(num_episodes):
11     env.reset() # the agent starts from a starting point
12     for t in range(time_steps):
13         action = 0 # an integer between 0 and 3
14                 # and is chosen by your RL algorithm.
15                 # Action 0: Up, 1: Down, 2: Left, 3: Right
16
17         # take an action and get an observation
18         next_state, reward, done, _ = env.step(action)
19         env.render() # update the frame of the environment
20
21     # when done is True, the agent is at a terminal state

```

²<https://github.com/schaul/py-vgdl/>

³<https://www.pygame.org>


```
22         # and the current episode is finished
23         if done:
24             break
```

- `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 coordinates, 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 visualise the movement of the agent in pygame.

Chapter 4

Evaluation

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

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 three 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. For example, if an agent requires 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 plot the total reward at each episode. Since the agent follows a random exploration throughout the learning, the total reward never converges to the maximum in learning. 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 specified as follows:

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

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 few inductive learning 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 inductive learning happens after episode 0, this included in the denominator, and the maximum convergence plot is strictly less than 1.

Third, we log runtime of both ILP(RL) and a benchmark at each episode, and plot the cumulative runtime over episodes. For ILP(RL), we also measure the number of ILASP calls and the average runtime of ILASP calls.

4.1.2 Benchmarks

We use Q-learning as our base benchmark. Q-learning is widely used RL technique, and given the environments used for the experiments are a simple environment in that it is discrete and deterministic, this method is sufficient as a benchmark.

TODO Q-leaning Implementation

4.1.3 Parameters

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

Table 4.1: List of parameters used in the experiments

All the parameters used in the experiments are summarised in Table ?? . Agents are trained in 250 time steps with 100 episodes, and conducted the same experiment 30 times in each environment. 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. The number of episode is specified such that both ILP(RL) and Q-learning eventually reaches the optimal policy. The number of time steps should be sufficient for the both algorithms to reach a terminal state by the ϵ -greedy exploration strategy, which we specify 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 a 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 fixed starting point. We conducted several experiments using different environments to highlight each aspect of the algorithm.

4.2 Learning Evaluation

4.2.1 Experiment 1: Setup

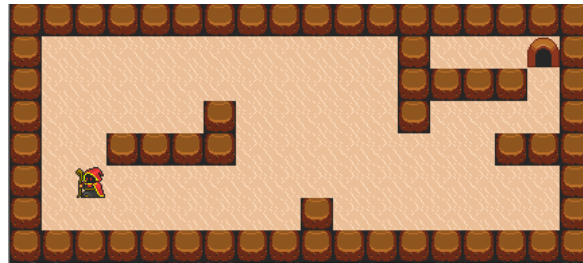


Figure 4.2: Game environment for experiment 1

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

the right upper corner as shown in Figure ?? . 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

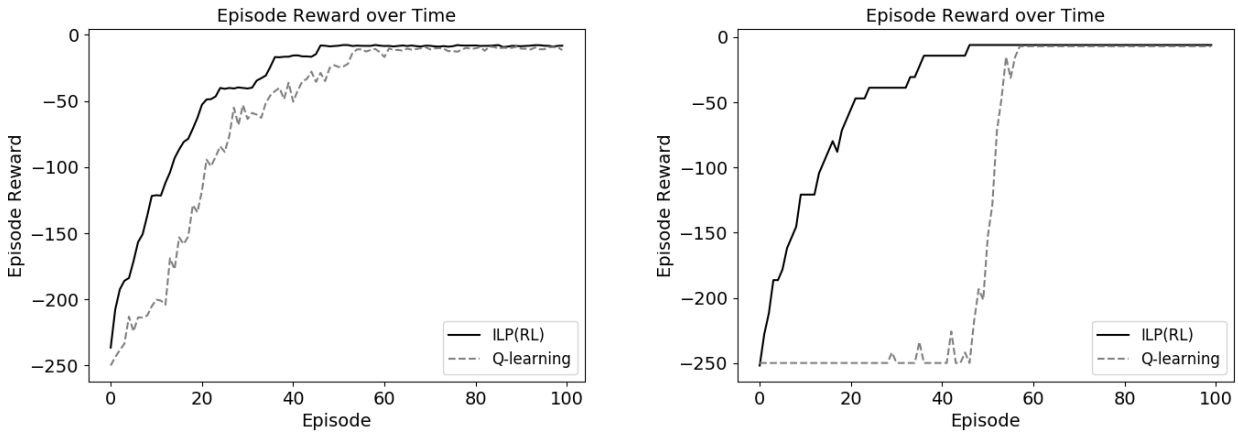


Figure 4.3: Results of experiment 1: learning curve (left) and testing (right)

Figure ?? shows the learning and testing performance. The convergence rate of ILP(RL) is faster than Q-learning: ILP(RL) reaches the maximum reward at between 40 and 50 episode, 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 α , 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 on the right in Figure ??, where we measure only the performance of the policy without random exploration. Overall this results shows that ILP(RL) converges to the optimal policy faster than Q-learning in this simple scenario, achieving more data-efficient learning.

Listing 4.1: Hypotheses 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 environment. This transfer learning capability is also described in Experiment 3 and 4.

Next, we see how the agent learns the hypotheses over time. We plot the learning convergence for ILASP at episode 0 in Figure ?. This graph shows that the agent learns almost all of the hypotheses at the episode 0. Since the two variables that the agent require to construct optimal answer set planning are the hypotheses 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 learns the full hypotheses at episode 0, without the terminal state, the agent continues to randomly explore the environment. Since the exploration of ILP(RL) is based on ϵ -greedy strategy, there is a potential that a better exploration strategy further improve the ILP(RL) learning.

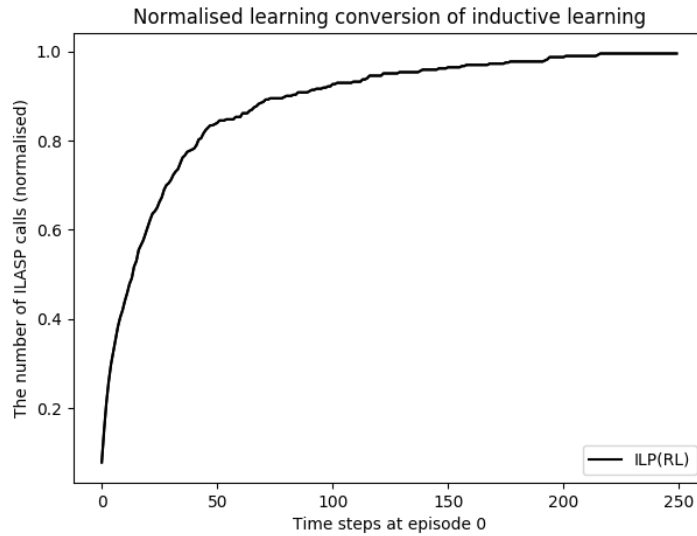


Figure 4.4: Normalised learning convergence by ILASP for experiment 1

Finally, we compare the runtime of two algorithms. The Figure ?? shows that the runtime of ILP(RL) in the first few episodes is significantly high. This is due to the fact that ILP(RL) runs ILASP calls at the beginning of episodes. The Figure ?? therefore highlights an issue that inductive learning is likely the bottleneck in terms of computational time. This issue may not be critical in cases where the time between the time steps is not an issue. If the performance is measured in terms of computation time rather than the number of iterations, ILP(RL) is not likely to perform existing 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 ASP planning is not a bottleneck of ILP(RL), but still takes longer time than Q-learning, as can be observed by the divergence of cumulative runtime between the two algorithms. This experiment show that while ILP(RL) learns faster than Q-learning in terms of the number of episodes, it suffers from increasing computational time due to inductive learning as well as ASP planning.

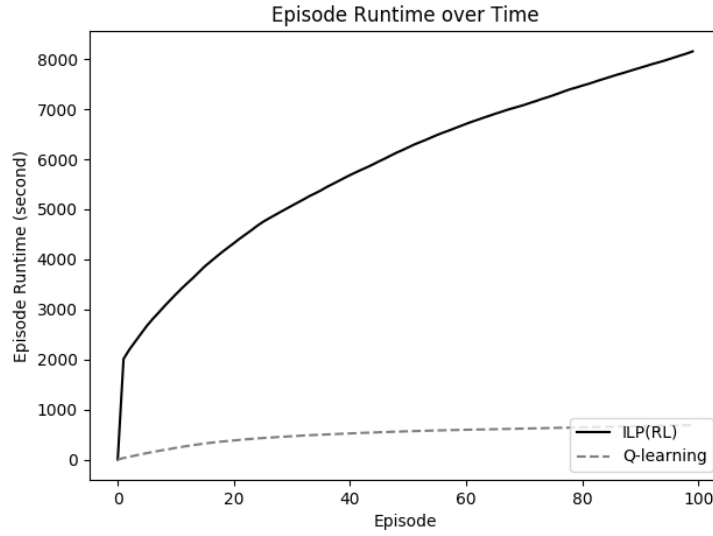


Figure 4.5: Runtime comparison

4.2.3 Experiment 2: Setup

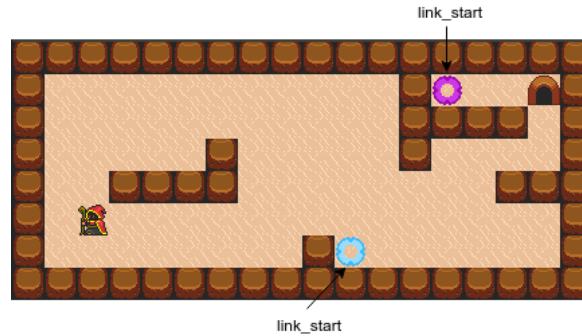


Figure 4.6: Game environment for experiment 2

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

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

`link_start(var(cell))` is a state for departure of the teleport and `link_dest(var(cell))` is the destination of the teleport. The teleport link is one-way: `link_start` takes the agent to `link_dest`, but `link_dest` does not take the agent back to `link_start`. This extra type allows ILASP to learn additional hypothesis. The full learning task for this experiment is in Appendix XX.

In VDGL game environment, 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 context examples, because ILP(RL) needs to generate exclusions regarding the link behaviour. The link locations need to be available for all positive examples so that ILASP correctly learns different a valid move for floor and teleport, 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

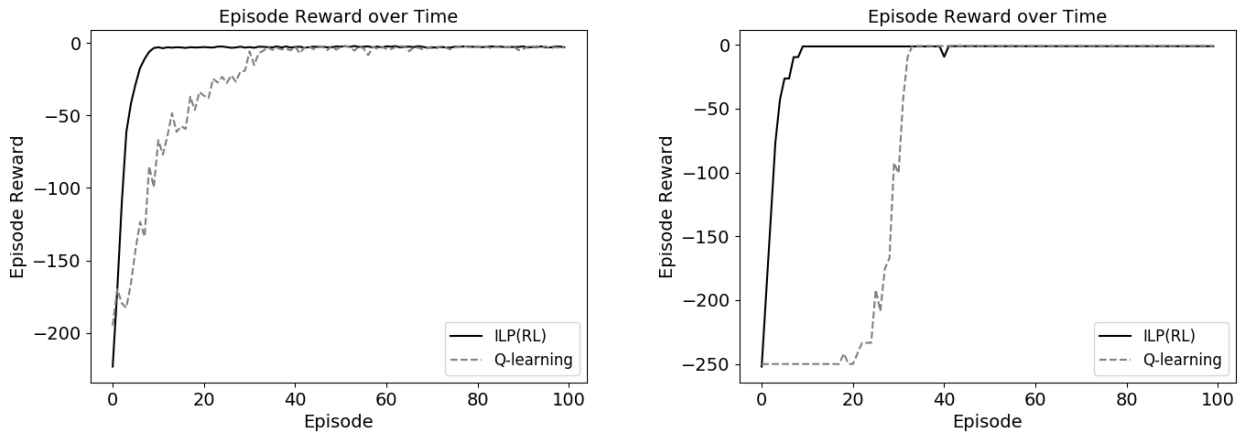


Figure 4.7: Results of experiment 3: learning curve (left) and testing (right)

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

```

1 state_after(V1):-link_dest(V1).
2 state_after(V0):-link_dest(V0), state_before(V0),action(right).
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(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 Definition ??, these generated `state_after` are all incorrect and therefore will be added to exclusions of the next positive examples. These exclusions will later refine hypotheses and results in Figure ?. The final complete hypotheses are shown in Listing ?.

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 applicable to an environment where there is no link, such as a game in experiment 1. In this case, the first two hypotheses in Listing ?? 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 learning at the beginning of episode.

Figure ?? show the runtime of ILP(RL) and Q-learning. Despite the fact that the size of the environment is the same as Experiment 1, there is a significant increase of runtime for ILP(RL) at the beginning of episode. This is because of the increase of search space as we added extra language bias.

The average runtime of inductive learning is 95.47 seconds, and there are on average 16.23 times inductive learning per episode.

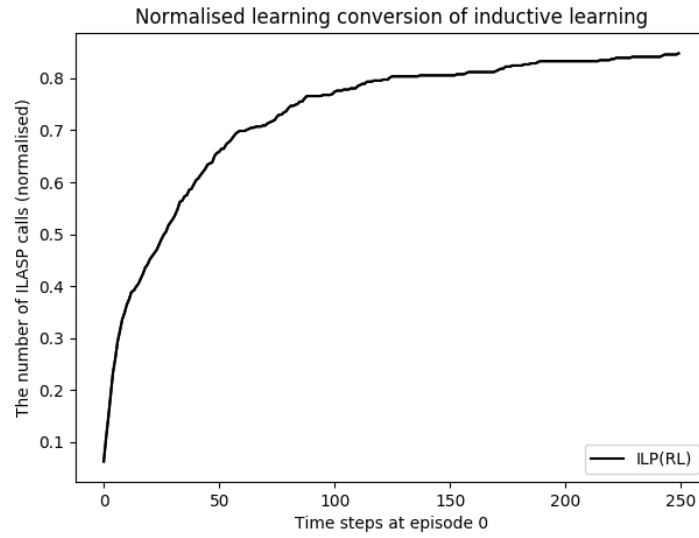
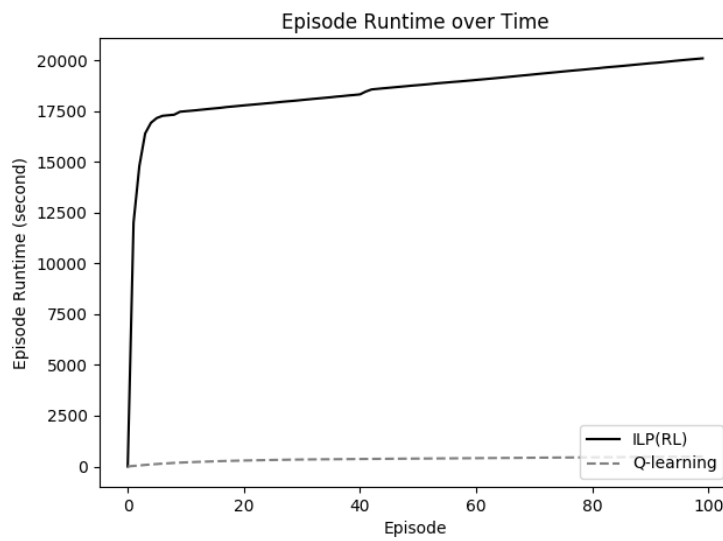
TODO is the table a good idea?

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

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: Comparison of runtime

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 comparison

4.3 Transfer Learning Evaluation

4.3.1 Experiment 3: Setup

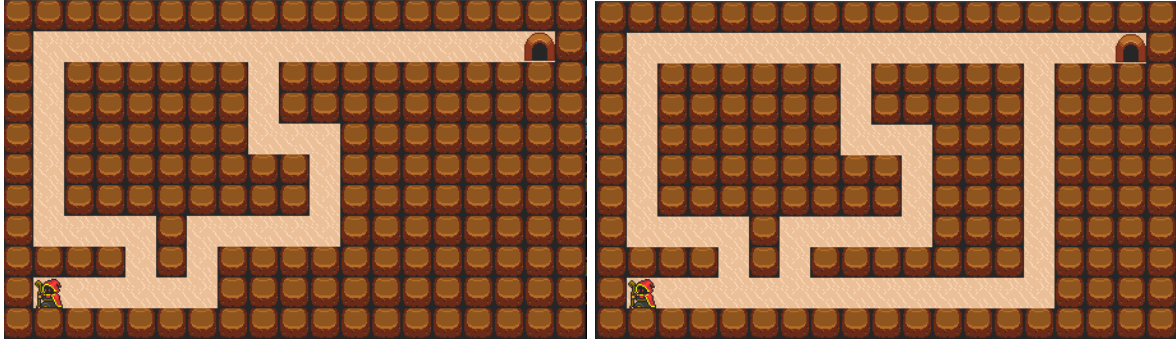


Figure 4.11: Game environment for experiment 3: before (left) and after (right) transfer learning

In Experiment 3, we investigated the possibilities of transfer learning between similar environments. We trained the ILP(RL) agent using the environment on the left in Figure ??, and transfer the learnt hypotheses as well as positive examples to a new environment on the right in Figure ?. The learnt hypotheses are valid moves in the environment and a general concept that is applicable to any similar environments. Positive examples are also transferred because, if there is a new hypothesis that the agent needs to learn in a new environment, ILP(RL) needs to refine the hypothesis by running ILASP in the new environment. Thus the all the positive examples are also transferred as well as the 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 terminal state is the same as that in the first environment, but the shortest path to the goal is different between the two environments as new shorter path is introduced in the right environment in Figure ?. In addition to Q-learning, we use three extra agents for comparison as follows.

- Agent(TL): The agent with transferred hypotheses, examples and also remembers the location of the terminal state.
- Agent(noTL)_{Goal}: The agent with transferred hypotheses, examples, but does not know the location of the terminal state.
- Agent(noTL)_{noGoal}: The agent with no transferred information, neither 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 terminal state is known in advance, this is still a useful transfer in cases where the terminal state is the same but the rest of the environment changes.

Listing ?? 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

```

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

```

4.3.2 Experiment Result 3

The result is shown in Figure ?? . For Agent(TL), since the complete hypotheses are already known to the agent as well as the terminal state, the agent can do ASP planning from episode 0. The only information required is background knowledge for the ASP planning, namely the locations of the walls, which are quickly acquired and reached the maximum total reward at the very beginning of episodes.

The next best agent in terms of the convergence to the maximum total reward is Agent(noTL)_{Goal}. Since the terminal state is known to the agent, the agent can do the planning from episode 0. However, the agent needs to learn the hypotheses. The reason that the convergence rate is almost the same as that of Agent(TL) is that , the ILP(RL) learns the almost complete hypotheses in episode 0, as observed in Experiment 1 and 2. Thus there is no difference between Agent(TL) and Agent(noTL)_{Goal} in terms of learning speed.

What makes a difference for the convergence is whether the agent knows the terminal state, which can be seen by compering between Agent(noTL)_{Goal} and Agent(noTL)_{noGoal}. The difference in terms of the iterations is that Agent(noTL)_{noGoal} needs to find the terminal state first before starting the planning, which is a random exploration.

This observation shows that there is a promising potential for improving the exploration strategy to find the terminal state as soon as possible.

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 hypotheses are transferred. The same as Experiment 3, we use four different agents.

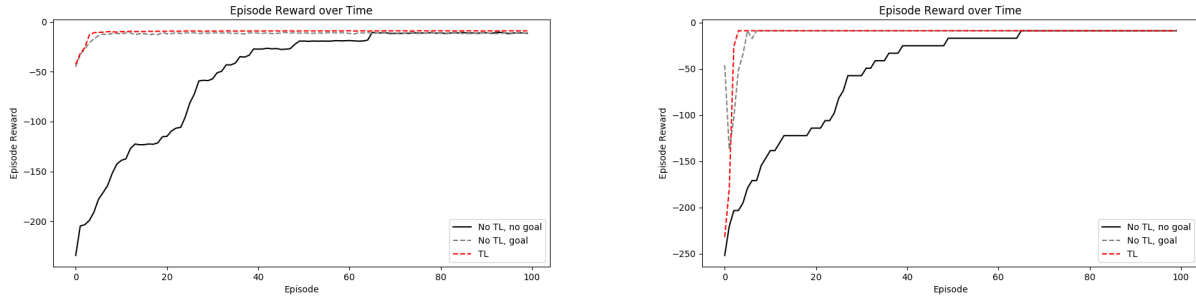


Figure 4.12: Results of experiment 3: learning curve (left) and testing (right)

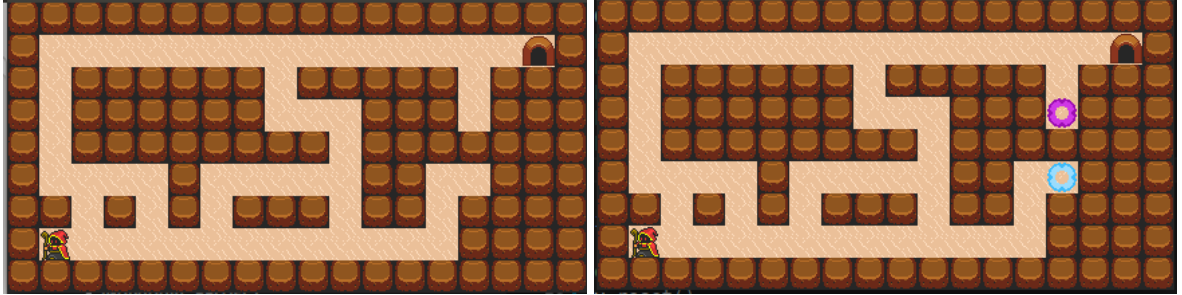


Figure 4.13: Game environment for experiment 4: before (left) and after (right) transfer learning

4.3.4 Experiment Result 4

Figure ?? show the results of the learning and test performance. The agent is able to successfully 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 scenarios.

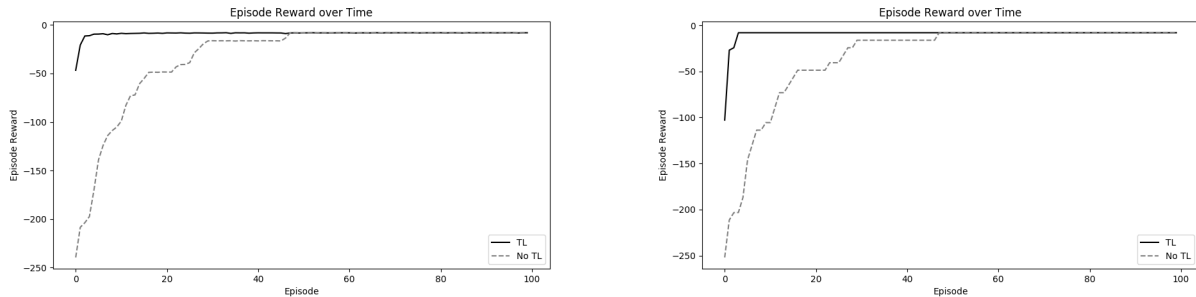


Figure 4.14: Result of experiment 4 (learning curve)

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

```
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).
```

4.4 Discussion

We investigated 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)

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

Faster learning convergence The ILP(RL) agent 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 generate 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 promising direction of the research.

Transfer Learning Unlike existing RL algorithms, where it learns value functions or Q values, ILP(RL) learns a valid move as a hypotheses, which can be applied to similar but different environments. We confirmed this with the experiments and, especially when the goal is known to the agent in experiment 3 and 4. 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.

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 framework. Some of these limitations are further elaborated in Further Research in the concluding chapter.

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 Q-learning due to the computation required for inductive learning with ILASP as well as ASP planning. This limitation indicates that ILP(RL) may not be suitable in an environment where there is a moving object based on time rather than time steps.

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 algorithms 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 limitation is theoretically discussed in XXX, where the complexity of deciding satisfiability is XX. (TODO Need to understand NP complete) 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 regardless of whether there is a new concept such as teleport links.

Another question remains to how to extend the current framework to more realistic scenarios. 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 other environments, however, 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 maximising total rewards, which is the common objective for most of RL algorithms, the application of the current framework is limited to particular types of problems.

There are some promising 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 reasoning.

However, most RRL algorithms focus on planning. RRL incorporates relational representations of states and actions to generalise 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 between entities in the environment to help improving policy. While RLL was applied to XX, this approach also shows an promising 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 networks (CNNs) [?] and an auto-encoder, which are transformed into symbolic representations for relevant object types and positions of the objects. These symbolic representations represent abstract state-space, which are the inputs for the Q-learning algorithm to learn a policy on this particular state-space. DSRL was shown to outperform DRL in stochastic variant environments. However, there are a number of drawbacks to this approach. First, the extraction of the individual objects was done by manually defined threshold of feature activation values, given that the games were geometrically simple. Thus this approach would not scale in geometrically complex games. 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 attempt 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 learns 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 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 consequences of each possible action. The more details of theoretical 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 framework focus on ASP-based ILP with RL, which has not been explored.

[?] proposed an architecture for interactively discovering previously unknown axioms using reinforcement learning. 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 researches. Due to advances of ASP-based ILP frameworks and there is no existing works of applying ASP-based ILP into RL scenarios, this motivates us to pursuing the potentials of ILP to solve RL problems. Because of the flexibility and expertise of ILASP, we decided to apply ILASP as our core learning framework.
- We considered the target hypotheses and how to construct learning tasks. Our target learning is a valid move of the agent, and which 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 promising 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 attempt 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 improvements 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 strategy 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, further experiments on different game environments are required for more robustness of the framework, such as the presence of dynamic enemies in the environment or non-stational environment. This might leave other aspects of using ILP in RL context and further bridge the gaps between the field 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 minimising 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 themselves 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 learning would return a sequence 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 promising direction.

Value Iteration Approach

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

Appendix A

Ethics

There is no particular ethical considerations for this particular research listed in Table ??.

Although this project is part of artificial intelligence in both inductive logic programming as well as reinforcement learning, it is a preliminary research and all the experiments were conducted using a game environment. Therefore there is no concerns for misuse or real applications.

	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.

Listing B.1: Learning tasks for Experiment 1

```
1 cell((0..7, 0..6)).
2 adjacent(right, (X+1,Y),(X,Y)):-cell((X,Y)), cell((X+1,Y)).
3 adjacent(left, (X,Y), (X+1,Y)):-cell((X,Y)), cell((X+1,Y)).
4 adjacent(down, (X,Y+1),(X,Y)):-cell((X,Y)), cell((X,Y+1)).
5 adjacent(up, (X,Y), (X,Y+1)):-cell((X,Y)), cell((X,Y+1)).
6 #modeh(state_after(var(cell))).
7 #modeb(1, adjacent(const(action),var(cell),var(cell))).
8 #modeb(1, state_before(var(cell)),(positive)).
9 #modeb(1, action(const(action)),(positive)).
10 #modeb(1, wall(var(cell))).
11 #max_penalty(50).
12 #constant(action, right).
13 #constant(action, left).
14 #constant(action, down).
15 #constant(action, up).
16 % Context dependent examples are added here
```

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.

Listing C.1: ASP program for Experiment 1

```
1 {action(down,T);
2   action(up,T);
3   action(right,T);
4   action(left,T);}1:-time(T), not finished(T).
5
6 state_at((1,4),3).
7 finished(T):-goal(T2),time(T), T >= T2.
8 goal(T):-state_at((5, 1),T), not finished(T-1).
9 goalMet:-goal(T).
10 :-not goalMet.
11
12 time(0..30).
13 cell((0..6, 0..5)).
14
15 adjacent(right,(X+1,Y),(X,Y)):-cell((X,Y)),cell((X+1,Y)).
16 adjacent(left,(X,Y),(X+1,Y)):-cell((X,Y)), cell((X+1,Y)).
17 adjacent(down,(X,Y+1),(X,Y)):-cell((X,Y)), cell((X,Y+1)).
18 adjacent(up,(X,Y),(X,Y+1)):-cell((X,Y)), cell((X,Y+1)).
19
20 #minimize{1,X,T: action(X,T)}.
21 #show state_at/2.
22 #show action/2.
```
