

# ASP-Based Abstractions for Reinforcement Learning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Logic and Computation**

eingereicht von

**Rafael Philipp Bankosegger, BSc.**

Matrikelnummer 11837316

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Thomas Eiter

Mitwirkung: Dipl.-Ing. Johannes Oetsch

Wien, 4. September 2025

---

Rafael Philipp Bankosegger

---

Thomas Eiter



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# ASP-Based Abstractions for Reinforcement Learning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Logic and Computation**

by

**Rafael Philipp Bankosegger, BSc.**

Registration Number 11837316

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Thomas Eiter

Assistance: Dipl.-Ing. Johannes Oetsch

Vienna, September 4, 2025

---

Rafael Philipp Bankosegger

---

Thomas Eiter



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Erklärung zur Verfassung der Arbeit

Rafael Philipp Bankosegger, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 4. September 2025

---

Rafael Philipp Bankosegger



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Acknowledgements

First and foremost, thank you, Karoline and Alois! Without your unwavering support, this thesis would not have been possible. Thank you, Johannes, for being very generous with your time and knowledge, and for keeping me on track many times. Thank you, Thomas, for your patience with me and for your insightful comments and edits. Thank you, Celine, Sibylle, my friends, and my larger family, for supporting me in many ways during the creation of this thesis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit dem modellfreien, bestärkenden Lernen in relationalen Markov-Entscheidungsproblemen (RMDPs), die Zustände und Aktionen durch logische, atomare Formeln repräsentieren. Ein fundamentales Hindernis für viele interessante Anwendungsdomänen in diesem Bereich ist eine kombinatorische Explosion derer Zustands-Aktions-Räume, wodurch wertbasierte, tabulare Lernalgorithmen wie “*Q*-learning” schnell unbrauchbar werden. Praktikable Algorithmen überwinden dieses Hindernis, indem sie tabulare Lernalgorithmen mit Funktionsapproximationstechniken erweitern. Dies erlaubt die Darstellung von sowohl kompakten, als auch generalisierbaren Repräsentationen für Nutzenfunktionen und Strategien. Vorgeschlagene Techniken reichen von hochentwickelten, wie “Relational Reinforcement Learning”, “Deep Relational Reinforcement Learning” und “Neuro-Symbolic Reinforcement Learning”, zu konzeptuell einfacheren, aber immer noch relevanten Techniken, wie Zustands- und Zustands-Aktions-Paar-Abstraktionen. Solche Abstraktionen wurden bereits in Kombination mit einer Vielfalt an Wissensrepräsentationsformalissen untersucht. Allerdings ist das logische Programmieren mit Antwortmengen, das “Answer Set Programming” (ASP), nur spärlich im diesem Rahmen untersucht, obwohl es, dank seine Deklarativität und seiner Fähigkeit zur nichtmonotonen Inferenz, ein interessanter Kandidat für diese Anwendung ist. Um diese Forschungslücke zu füllen, untersucht diese Arbeit die Anwendung von ASP zur Repräsentation von Zustands- und Zustands-Aktions-Paar-Abstraktionen für RMDPs. Es wird eine generelle Methode konzeptualisiert, um Abstraktionen in ASP zu kodieren, und ein Prototyp derselben implementiert. Basierend darauf werden Abstraktionen in zwei konkreten Anwendungsdomänen modelliert und getestet, nämlich für Stapelaufgaben in Blockwelten und für Navigationsaufgaben in “Minigrid“-Umgebungen. Empirische Analysen zeigen, dass es, unter der Nutzung von “*Q*-learning” und mithilfe der Abstraktionen, möglich ist, Strategien mit akzeptabler Qualität stabil und reproduzierbar zu lernen, und zwar mit weniger Interaktionen als nötig sind, um vergleichbare Strategien ohne die Abstraktionen zu lernen. Diese Resultate sind im Einklang mit der betrachteten Abstraktions-Theorie und einer großen Menge an vorangegangenen empirischen Untersuchungen. Allerdings sind zusätzliche Studien nötig um die Brauchbarkeit unseres Zuganges in praktischen Anwendungen zu prüfen. Nach Wissen des Authors ist die vorliegende Arbeit die erste, die sich im vorgestellten Rahmen mit der Anwendung von ASP zur Repräsentation spezifisch von Zustands-Aktions-Paar-Abstraktionen befasst.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Abstract

This thesis considers the problem of model-free reinforcement learning (RL) in relational Markov decision processes (RMDPs), where states and actions are represented by atoms, formed from predicates over a domain of objects. A fundamental issue for many interesting task environments in this setting is a combinatorial explosion of their state-action spaces, rendering value-based, tabular learning algorithms, such as  $Q$ -learning, intractable. Practical value-based learning algorithms have dealt with this issue by extending tabular learning algorithms with function approximation techniques, allowing for both compact and generalisable representations of learned value functions and policies. Proposed techniques range from highly sophisticated techniques, such as relational RL, deep relational RL, and neuro-symbolic RL, to the conceptually more simple but still relevant techniques of state- and state-action pair abstractions. Such abstractions have been represented in a variety of knowledge representation formalisms but we found answer set programming (ASP) to be under-explored in this use case, even though it is an interesting candidate due to its declarative knowledge processing and non-monotonic reasoning capabilities. In order to close this research gap, ASP is explored as a means of representing state- and state-action pair abstractions for RMDPs. To this end, a general method for encoding abstractions in ASP is devised and implemented as a proof of concept. Two specific abstractions are modelled and tested, one for blocks world stacking tasks and one for navigation tasks in grid worlds from the Minigrid library. An empirical analysis shows that, using  $Q$ -learning on the abstract representations, policies of acceptable quality can be learned with high consistency. These policies can also be obtained in a smaller number of samples than what is needed to learn comparable policies without the abstraction. These results are consistent with both the considered theory of abstraction and a large body of previous empirical research but additional studies are needed to test the viability of our approach in real-world applications. To the knowledge of the author, this thesis is the first piece of research to consider ASP as a representation method for state-action pair abstractions in this setting.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Contents

<b>Kurzfassung</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>Contents</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Preliminaries and Theoretical Background</b>	<b>9</b>
2.1 Notation . . . . .	9
2.2 Answer Set Programming . . . . .	10
2.3 Reinforcement Learning . . . . .	19
2.4 Tabular Solution Methods and $Q$ -Learning . . . . .	26
2.5 State-Action Pair Abstraction . . . . .	32
<b>3 ASP-Based State-Action Pair Abstractions</b>	<b>39</b>
3.1 ASP and Prolog . . . . .	39
3.2 The CARCASS Framework . . . . .	41
3.3 Translation to ASP . . . . .	45
3.4 Example . . . . .	53
<b>4 Case Study: Blocks World</b>	<b>59</b>
4.1 The Blocks World as RMDP . . . . .	61
4.2 ASP-Based Blocks World CARCASS . . . . .	64
4.3 Empirical Evaluation . . . . .	71
4.4 Discussion . . . . .	81
<b>5 Case Study: Minigrid</b>	<b>83</b>
5.1 The Minigrid setting . . . . .	86
5.2 Relational State Description . . . . .	91
5.3 ASP-Based Minigrid CARCASS . . . . .	92
5.4 Empirical Evaluation . . . . .	104
5.5 Discussion . . . . .	113
	xiii

<b>6 Related Work</b>	<b>115</b>
6.1 Representing Abstractions in RMDPs . . . . .	115
6.2 Adjacent Research Areas . . . . .	118
<b>7 Conclusion and Future Work</b>	<b>121</b>
<b>Overview of Generative AI Tools Used</b>	<b>125</b>
<b>Bibliography</b>	<b>127</b>

# Introduction

The field of *reinforcement learning* (Sutton and Barto 2018) provides frameworks and practical methods for building agents with the ability to learn rational behaviour online, through repeated interactions with an unknown or partially known task environment. The *discrete-time Markov decision process (MDP)* in particular is a well-understood and broadly applicable framework for modelling agent-environment interactions as discrete time points. At any such time point, the agent perceives the current state of the environment and performs an action selected by its current policy, a stochastic mapping from states to actions. The state of the environment is then updated and a numerical reward signal, acting as a measure of the agent's success, is generated. A solution to the MDP is a policy that maximises the expected discounted sum of rewards, called the expected return, over an infinite time horizon and for all states. Value-based, tabular learning algorithms, such as *Q-learning*, store and update policy-dependent estimates of the expected returns, called value functions, for every state and every action admissible in that state. After an interaction between the agent and the environment the estimates are updated and the agent's policy is iteratively improved. The asymptotic convergence to the optimal policy is guaranteed under certain requirements, such as the finiteness of the MDP (Tsitsiklis 1994). In practice, the learning process can usually be stopped after a finite number of interactions, as soon as the learned policy complies to a desired quality standard.<sup>1</sup>

The space requirements and the amount of interactions needed to sufficiently explore the state-action space are however limiting factors of the tabular approach, with many environments of practical relevance being intractable due to a combinatorial explosion in the size of the state-action space. Another issue for some environments is that positive rewards become increasingly sparse and require longer chains of interactions to reach as the state space increases in size. In extreme cases, there might exist only a single state

---

<sup>1</sup>This assumes that the environment is stationary, i.e. that the dynamics of the environment and the reward signal stay constant over time.

with a positive reward in the entire state space.<sup>2</sup> This means that, although there exist algorithmically efficient methods of exploration that provide theoretical upper bounds for such worst-case scenarios (Li 2012), there is a limit to the applicability of pure *tabula rasa* algorithms that learn behaviour with no prior knowledge about the environment. These three problems have been referred to as the *storage problem*, the *learning problem*, and the *needle-in-a-haystack problem*, respectively (e.g. van Otterlo 2008, p. 74).

**Function Approximation.** An essential step in overcoming these problems is the extension of value-based learning algorithms with *function approximation* techniques (Sutton and Barto 2018, Chapters 9–11), which allow for representations of value functions that are both compact and generalisable beyond the so-far encountered and well-explored state-action pairs to completely new or less-explored state-action pairs. In order to apply function approximation techniques, we first need to reconsider how states and actions are represented. Tabular methods use *atomic* representations of states and actions, meaning that they are treated as no more than distinct elements in sets with no underlying structure. Consequently, value functions can only be represented as lookup tables, mapping individual states and state-action pairs to numeric values. However, it is often possible to decompose a state or an action into several distinct elements. Examples of these so-called *factored* representations include numeric *feature vectors* and RGB-images.

**Relational Markov Decision Processes.** Of special interest to us are *relationally factored* representations, where states are described using *atoms* that are formed from *predicates* over a domain of objects.<sup>3</sup> The *relational MDP (RMDP)* in particular extends the standard MDP framework by defining states as sets of ground atoms (i.e. *Herbrand interpretations*) and actions as single ground atoms (van Otterlo 2008, p. 168). The atoms in a state represent true facts about that state and atoms not in the state are considered to be false in accordance with the *closed world assumption*. As an example, consider a simple *blocks world* environment with a domain consisting of two blocks and a table. The binary predicate **on** is used to state the fact that one block is positioned directly on top of another block or the table. A state where both blocks are positioned on the table is therefore represented as the set  $\{\mathbf{on}(b1, \text{table}), \mathbf{on}(b2, \text{table})\}$ . The binary predicate **move** is used to represent the agent's actions. An action that picks block "b1" and places it on top of the other block "b2" is denoted as **move**(b1, b2). The resulting new state after applying this action is the set  $\{\mathbf{on}(b1, b2), \mathbf{on}(b2, \text{table})\}$ . As a second example, consider a *grid world* environment where the agent is situated in a two-dimensional grid. The domain consists of the grid coordinates, the agent, and other objects of interest. We use the predicate **obj** to make statements about the location of objects on the grid. In addition, we use the predicate **room** to state the existence of rectangular rooms. A state where the agent is located in the corner of a small room, facing east, and with

---

<sup>2</sup>A prominent example is the blocks world task environment, see Chapter 4.

<sup>3</sup>For precise definitions of these concepts see Section 2.2.

---

the task to reach a goal tile in the opposite corner of the room can be described as  $\{\mathbf{room}((0, 0), (7, 7)), \mathbf{obj}(\mathbf{agent}(\mathbf{east}), (1, 1)), \mathbf{obj}(\mathbf{goal}, (6, 6))\}$ .

An elaborate case can be made for why relational representations are useful in the context of reinforcement learning (van Otterlo 2008, Chapter 4). When combined with knowledge representation formalisms based on *first-order logic*, the relational setting provides a natural and comprehensible format for representing and reasoning about value functions, policies, and other aspects of RMDPs. The expressiveness of these formalisms allows for the modelling of concepts that are both compact and generalisable across states and actions, in some cases with levels of compactness and generalisability that are higher than what is possible with techniques based on feature vectors or propositional representations.<sup>4</sup> Critically, generalisation can occur not only across states and actions from the same domain but also across different domains with arbitrary numbers of objects. In the context of reinforcement learning, this can be useful in formulating value functions and policies that are transferable across multiple RMDPs with comparable environmental dynamics, comparable reward structures and built from the same predicates. Finally, the relational setting allows us to make use of principles from knowledge representation and reasoning. A declarative view on knowledge and a clear separation between knowledge representation and knowledge processing allows for a modular approach, which in turn allows for the integration of reinforcement learning techniques into larger applications. Prior knowledge can be used in the formulations of policies and value functions. In addition, the reinforcement learning process can serve as a knowledge acquisition component, as learned policies and other learned knowledge about the environment can be integrated into the knowledge base.

Over the years, many methods for representing and learning value functions and policies in the relational setting have been suggested. Function approximation techniques from the field of *relational learning*, designed specifically with the relational setting in mind, have been used to extend tabular reinforcement learning techniques, resulting in a subfield called *relational reinforcement learning* (Dzeroski, De Raedt, and Blockeel 1998; Džeroski, Raedt, and Driessens 2001; Driessens and Ramon 2003; Gärtner, Driessens, and Ramon 2003; Bloch 2018; Das et al. 2020). Recent developments include the application of *deep relational learning* techniques (Zambaldi et al. 2019; Janisch, Pevný, and Lisý 2020; Garg, Bajpai, and Mausam 2020; Sharma et al. 2023) and of *neuro-symbolic* methods (Dong et al. 2019; Hazra and Raedt 2023).

**State Abstraction.** A fundamental technique is *state abstraction* (van Otterlo 2008, p. 92; Sutton and Barto 2018, p. 203), where the idea is to partition the state space into sets of states that share some similarity. Each block of the partition is considered to be an *abstract state*. The technique of *state-action pair abstraction* generalises state abstraction by introducing *abstract actions*. Here, the partition is formed over the set of state-action pairs (Goetschalckx 2009, pp. 51–56; Ravindran 2004, p. 18). A simple way to incorporate

---

<sup>4</sup>In the context of supervised machine learning, this has been shown using examples such as the *parity problem* and the *Bongard problem* (van Otterlo 2008, pp. 155–161).

these types of abstraction into the reinforcement learning process is to define them a priori and to keep them fixed as the environment is explored. Value functions and policies can be constructed on the basis of these abstract states and state-action pairs instead of their concrete counterparts. The  $Q$ -learning algorithm and other tabular methods can be applied as usual on these abstract representations, but their success depends on the quality of the abstraction, as there is no general guarantee of convergence (Li, T. J. Walsh, and Littman 2006). Abstraction techniques designed for the relational setting can be distinguished by the formalisms used to represent abstract states and state-action pairs and by the decision procedures used to compute their coverage. To start with, abstract states can be represented as formulae in first-order logic. In this context, concrete states are elements of (or covered by) an abstract state if and only if they model its corresponding formula.<sup>5</sup> As an example based on the previously introduced blocks world environment, consider an abstract state represented by the formula  $\exists X.\exists Y.\exists Z.\mathbf{on}(X, Y) \wedge \mathbf{on}(Y, Z)$ . Intuitively, this abstract state covers concrete states where there exists a tower of two or more blocks stacked on top of each other. In a 2-blocks environment, the formula has two concrete states as models, namely  $\{\mathbf{on}(b1, b2), \mathbf{on}(b2, \text{table})\}$  and  $\{\mathbf{on}(b2, b1), \mathbf{on}(b1, \text{table})\}$ . The concrete state  $\{\mathbf{on}(b1, \text{table}), \mathbf{on}(b2, \text{table})\}$  however is not a model and therefore does not belong to the represented abstract state. In a logic programming context, abstract states can be represented as conjunctions of literals where existential quantification is implicitly assumed. Thus, the formula above can be rewritten as the set  $\{\mathbf{on}(X, Y), \mathbf{on}(Y, Z)\}$  and subsumption testing (e.g. Santos and Muggleton 2010) can be used as the decision procedure for deciding coverage. Abstract state-action pairs can be represented similarly. For example, again in the 2-blocks world, the abstract state-action pair represented by  $(\{\mathbf{on}(X, Y), \mathbf{on}(Y, Z)\}, \mathbf{move}(X, \text{table}))$  covers the concrete state-action pairs  $(\{\mathbf{on}(b1, b2), \mathbf{on}(b2, \text{table})\}, \mathbf{move}(b1, \text{table}))$  and  $(\{\mathbf{on}(b2, b1), \mathbf{on}(b1, \text{table})\}, \mathbf{move}(b2, \text{table}))$ . These concrete state-action pairs are similar in the sense that, for both, executing the action in its corresponding state, leads to a new state where all blocks are unstacked.

In the literature there exist several approaches to abstraction based on logic programming. These include *logical Markov decision programs* (Kersting and De Raedt 2004), *rQ-learning* (Morales 2003), the CARCASS framework (van Otterlo 2003; van Otterlo 2008, pp. 252–270) and *stochastic abstract policies* (Koga, Silva, and Costa 2015). The *Prolog*-based CARCASS framework is particularly noteworthy for explicitly allowing for the integration of domain knowledge in the form of logic programs. Besides logic programming, other knowledge representation paradigms have been considered. Karia and Srivastava (2022) used a description logic to model vectors of first-order features, which in turn were used to define abstract states and actions. Kokel, Natarajan, et al. (2023) presented a hierarchical framework where a planner was used to decompose a given RMDP into a sequence of subtasks. State abstraction was achieved by removing atoms that were irrelevant, which was decided individually for each subtask and based on a formal language of *dynamic first-order conditional influence (D-FOCI)* statements.

<sup>5</sup>Note that, in order to achieve a well-defined set partition, the models of the used formulae need to be mutually exclusive.

---

The more recent research also gives a motivation for the use of fixed abstraction techniques in state-of-the-art reinforcement learning frameworks. Fixed abstraction techniques are conceptually simple, thus easily understandable, and have a strong theoretical foundation. Abstraction techniques are also flexible in their range of applications, as they can be used both directly with tabular methods but also in combination with other function approximation methods. Both Kokel, Natarajan, et al. (2023) and Karia and Srivastava (2022) use fixed abstractions as a pre-processing (or feature selection) step before applying advanced function approximation methods, such as deep learning.

**Answer Set Programming.** When it comes to the modelling of abstractions for RMDPs, we found that the problem solving approach of *answer set programming (ASP)* (Lifschitz 2019; Brewka, Eiter, and Truszczyński 2011) is under-explored. Rooted in knowledge based systems and logic programming, ASP offers a unique combination of features that make it an interesting candidate for modelling abstractions. Unlike Prolog, ASP is truly declarative with a strict separation of logic from control. Unlike description logics, ASP supports non-monotonic reasoning, allowing for reasoning under incomplete knowledge. These features make ASP elaboration tolerant (McCarthy 1998), a highly desired property, as it makes it easy to iterate on abstractions. In addition, the language of ASP is expressive enough to model and reason about both action languages for planning applications and D-FOCI statements.

We are aware of two pieces of research that have used ASP for representing various aspects of RMDPs in an abstract form. Mitchener et al. (2022) presented a hierarchical framework where ASP was used to model a high-level policy. The ASP encoding of the policy was constructed automatically, based on a value function derived from previously collected histories of interactions. The resulting policy was compactly represented and able to generalise beyond the experienced interactions. The histories and value functions were however maintained on a much finer level of abstraction, making them susceptible to the state explosion problem and a potential bottleneck. Sridharan and Meadows (2018) presented a robot architecture in which ASP was employed, among other usages, to identify and eliminate atoms irrelevant to solving a given reinforcement learning task, based on pre-existing domain axioms, including casual laws, constraints, and executability conditions. This amounts to state abstraction. To our knowledge, there exists as of yet no ASP-based representation of full state-action pair abstraction for both value functions and policies.

**Objective and Research Questions.** The general objective of this thesis is to explore ASP as a representation method for state and state-action pair abstractions in RMDPs. For the scope of this thesis, we limit our exploration to two types of environments: classical blocks world environments and grid world environments as provided in the *Minigrad* simulation library (Chevalier-Boisvert et al. 2023). The reinforcement learning algorithm of choice is  $Q$ -learning with an  $\epsilon$ -greedy exploration policy. Learning is considered on both concrete and abstract representations of the state-action space and we refer to those cases as *concrete  $Q$ -learning* and *abstract  $Q$ -learning*, respectively. Our primary assumption is

that state-action pair abstractions make RMDPs with large state-action spaces tractable by reducing both the space and sample requirements of the learning process, with coarser abstractions, i.e. with fewer abstract state-action pairs, resulting in larger reductions. But, as Li, T. J. Walsh, and Littman (2006) observed, the coarseness of an abstraction is also associated with a cost. When used with abstract  $Q$ -learning in particular, convergence is not guaranteed in general and, if convergence occurs nevertheless, there is also no guarantee that the learned abstract policy is optimal. To understand the effects of our abstraction, we therefore need to study three constructs, namely the stability of the learning process, the quality of the learned policy, and the observed sample efficiency, which roughly relates to the number of samples needed for the learning process to produce a policy of acceptable quality.<sup>6</sup> Based on these considerations, we formulate the following main research question.

*For the mentioned task environments, can we build ASP-based state-action pair abstractions and, using abstract  $Q$ -learning with an  $\epsilon$ -greedy exploration policy, consistently learn policies of acceptable quality while achieving a significant increase in sample efficiency, compared to concrete  $Q$ -learning?*

For our empirical research we derive secondary research questions comparing concrete  $Q$ -learning (as a control condition) to abstract  $Q$ -learning:

- Do the proposed abstractions cause differences in sample efficiency?
- Do the proposed abstractions cause stability issues during the learning process?
- Do the proposed abstractions cause differences in the quality of the learned policy?

**Contributions.** In an effort to answer the above questions, we make the following research contributions in this thesis.

1. We present a general method to encode state-action pair abstractions in ASP, based on the CARCASS framework (van Otterlo 2008, pp. 252–270). The resulting encodings can be combined with background knowledge in the form of other ASP programs. We also show how to translate existing, CARCASS-based abstractions to ASP.
2. We build a proof of concept of the above method, making use of the *clingo* answer-set solver (Gebser, Kaminski, Kaufmann, Ostrowski, et al. 2016).
3. We present a hand-coded, ASP-based state-action pair abstraction in the blocks world, making use of procedural domain knowledge in the form of pre-existing blocks world planning algorithms.

---

<sup>6</sup>The constructs are explained in detail in section 4.3.1 and in section 5.4.1. The quality criteria are defined individually for every task environment.

- 
4. We present a relational state description and a hand-coded, ASP-based state abstraction for Minigrid environments. We use ASP to derive a high-level room layout from low-level facts about the grid world and to plan high-level navigation paths through that layout. This knowledge is used to derive a coarse state abstraction based on the next object or location on the navigation path, ignoring facts that are irrelevant to the task at hand.
  5. We present empirical research conducted in both blocks world environments with up to 20 blocks and in several Minigrid environments. The experiments use  $Q$ -learning with  $\epsilon$ -greedy exploration and compare the different representations (concrete and abstract). We measure the stability of the learning process to understand convergence, the quality of the learned solutions, and the sample efficiency. The results show that, under certain parameter configurations, it is possible for abstract  $Q$ -learning to reliably produce policies of acceptable quality and in a smaller number of samples, compared to concrete  $Q$ -learning.

The results are consistent with a large body of research on abstraction and support the theoretical claims about the effect of abstraction on sample efficiency but additional studies are needed to test the viability of our approach in real-world applications.

**Structure.** The thesis is organised as follows. In Chapter 2, we discuss the theoretical foundation of the thesis, including ASP, reinforcement learning and abstraction. In Chapter 3, we present the original CARCASS concept and how we adapted it to ASP. In Chapters 4 and 5, we present case studies in blocks worlds and in *Minigrid* environments, respectively. In Chapter 6 we glance over the related work and finally, in Chapter 7 we present our conclusions and ideas for future work.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Preliminaries and Theoretical Background

In this chapter, we review the theory on which we base this thesis. Section 2.1 introduces basic notation. Section 2.2 introduces answer set programming. Sections 2.3, 2.4, and 2.5 are concerned with reinforcement learning, introducing Markov decision processes in a relational domain,  $Q$ -learning as a solution method and a basic theory of abstraction, respectively.

## 2.1 Notation

Unless otherwise specified, we use the notation detailed in the following subsections.

### 2.1.1 Sets and Set Partitions

We denote sets with uppercase letters  $X = \{x_1, x_2, \dots\}$ ,  $Y = \{y_1, y_2, \dots\}$  and elements of those sets with indexed lowercase letters  $x_i \in X$ ,  $y_j \in Y$ . The number of elements in a set  $X$  are denoted as  $|X|$ . Sets of sets will be denoted with calligraphic letters  $\mathcal{X} = \{X, Y\}$ . In particular, the powerset is denoted as  $\mathcal{P}(X) = \{X' \mid X' \subseteq X\}$ . In addition to common set operations like the Cartesian product  $X \times Y$ , union  $X \cup Y$  and intersection  $X \cap Y$ , we define the following operations:  $X^1 \doteq X$ ,  $X^n \doteq X \times X^{n-1}$  for  $n > 1$ .

Let  $X$  be a set. Following the notation of Ravindran (2004, pp. 12–14), a *partition* of  $X$  is a set of *blocks*  $\mathcal{B} \doteq \{B_1, B_2, \dots, B_n\}$  with  $B_i \subseteq X$  (for  $1 \leq i \leq n$ ),  $B_i \cap B_j = \emptyset$  (for  $1 \leq i, j \leq n$  and  $i \neq j$ ) and  $\bigcup_i B_i = X$ . Let  $x \in X$ . The block of  $\mathcal{B}$  to which  $x$  belongs is denoted by  $[x]_{\mathcal{B}}$ , i.e.,  $[x]_{\mathcal{B}} = B_i$  iff  $x \in B_i$  for  $B_i \in \mathcal{B}$ . A mapping  $f : X \rightarrow Y$  induces the partition  $\mathcal{B}_f$  on  $X$  as follows:  $[x]_{\mathcal{B}_f} = [x']_{\mathcal{B}_f}$  iff  $f(x) = f(x')$  (for all  $x, x' \in X$ ). For some  $x \in X$ , we use the shortcut notation  $[x]_f \doteq [x]_{\mathcal{B}_f}$ . Two partitions  $\mathcal{B}_1$  and  $\mathcal{B}_2$  of  $X$  can be

ordered by their *coarseness*. We write  $\mathcal{B}_1 \succeq \mathcal{B}_2$  ( $\mathcal{B}_1$  is *coarser than or equally coarse to*  $\mathcal{B}_2$ ) iff  $[x]_{\mathcal{B}_2} = [x']_{\mathcal{B}_2}$  implies  $[x]_{\mathcal{B}_1} = [x']_{\mathcal{B}_1}$  for all  $x, x' \in X$ .

### 2.1.2 Probability Theory

We base our notation for probability theory on the one presented by Schickinger and Steger (2002, Chapter 1) and Beierle and Kern-Isberner (2019, Appendix A). Let  $\Omega = \{\omega_1, \omega_2, \dots\}$  be a countable set, called a *sample space*. We call  $E \subseteq \Omega$  an *event*. If  $|E| = 1$  we call  $E$  an *elementary event*. A *discrete probability distribution* over  $\Omega$  assigns a probability to all possible events by its *probability function*  $\Pr : \mathcal{P}(\Omega) \rightarrow [0, 1]$  with  $\sum_{\omega \in \Omega} \Pr(\{\omega\}) = 1$  and  $\Pr(E) = \sum_{\omega \in E} \Pr(\{\omega\})$  for all  $E \subseteq \Omega$ . We use  $\Pr\{\omega_1, \omega_2, \dots\}$  as abbreviation for  $\Pr(\{\omega_1, \omega_2, \dots\})$ .

We denote random variables by uppercase letters  $X, Y$  in sans-serif font to distinguish them from sets. Formally, given a sample space  $\Omega$ , a *random variable* is a mapping  $X : \Omega \rightarrow X$  to some set  $X$ . The *domain* of  $X$  is defined as  $Dom(X) \doteq \{X(\omega) \mid \omega \in \Omega\}$ . A random variable can also be used in a predicate to define an event. For example,  $\{X = x\} \doteq \{\omega \in \Omega \mid X(\omega) = x\}$ . We call  $X$  a *numeric random variable* if  $Dom(X) \subseteq \mathbb{R}$  and denote its *expected value* by  $\mathbb{E}(X) \doteq \sum_{x \in Dom(X)} x \cdot \Pr\{X = x\}$ .

Let  $A, B \subseteq \Omega$  be events with  $\Pr\{B\} > 0$ . The *conditional probability* of  $A$  occurring provided that  $B$  occurred is defined as  $\Pr(A \mid B) \doteq \frac{\Pr(A \cap B)}{\Pr(B)}$ . This can be extended to *conditional random variables* denoted by  $X \mid A$ , where we define the expected value as  $\mathbb{E}(X \mid A) = \sum_{x \in Dom(X)} x \cdot \Pr\{X = x \mid A\}$ .

We refer to the act of drawing an elementary event  $\omega \in \Omega$  according to the probability distribution characterised by  $\Pr$  as *sampling*, *drawing a sample*, or *realising a sample* from that distribution. This is denoted by  $\omega \sim \Pr(\cdot)$ . Given an event  $E \subseteq \Omega$ , a sample can also be drawn from a conditional probability distribution, denoted by  $\omega \sim \Pr(\cdot \mid E)$ .

## 2.2 Answer Set Programming

*Answer set programming (ASP)* (Lifschitz 2019; Brewka, Eiter, and Truszczyński 2011) is an expressive formalism that can be used to encode and solve combinatorial search and optimisation problems. Its features make it attractive for many applications in science and industry (Erdem, Gelfond, and Leone 2016; Falkner et al. 2018). At the core of ASP is its declarative modelling language. When following a *declarative problem solving approach* (Eiter, Ianni, and Krennwallner 2009, p. 3), there is no need to develop a custom algorithm unique to the problem at hand. Instead, the problem specification itself can be encoded in the language of ASP, which often is a much easier task. The search for a solution, i.e. the hard part, is left up to computationally efficient off-the-shelf answer set solvers, or ASP solvers in short. (Gebser, Kaminski, Kaufmann, Ostrowski, et al. 2016; Alviano et al. 2017). A solution to the ASP encoding is called a *model* or *answer set*.

The sophistication of contemporary ASP solvers was made possible in part due to the rigorous semantics of the ASP modelling language (Gelfond and Lifschitz 1988; Calimeri et al. 2020). The strong connection to the field of Knowledge Representation and Reasoning and to other logic-based formalisms (Eiter, Ianni, and Krennwallner 2009, p. 4) inspired the adaptation of already existing theoretical results and algorithms, such as the DPLL-procedure and conflict-driven learning (Gebser, Kaminski, Kaufmann, and Schaub 2012, Chapter 6). A consensus on the semantics also enabled the research community to collaborate and to foster development in regularly held, friendly solver competitions (Gebser, Maratea, and Ricca 2020).

What makes ASP stand out from adjacent approaches such as *SAT Solving*, *Prolog* and *Constraint Programming* is its unique combination of features, as summarised by Brewka, Eiter, and Truszczyński (2011). Compared to the propositional nature of SAT solving, ASP has a relational aspect and allows for first-order predicates and variables in its syntax. The presence of variables allows for the modelling of recursive relations such as the transitive closure of a graph. Compared to Prolog, ASP has a stronger commitment to the declarativity of its language. This limits the expressiveness of ASP but makes modelling more accessible and intuitive, especially to users with no strong background in logic. The nonmonotonic nature of ASP is useful in dealing with incomplete knowledge and allows to model default assumptions in the absence of information. In addition, an ASP encoding can yield more than one answer set, a feature that can be exploited in a *model-based problem specification methodology*: A problem specification in ASP can be formulated such that there is a one-to-one mapping between solutions of the original problem and the answer sets admitted by the ASP specification. In many cases, only a single answer set is needed, but the one-to-one correspondence is still a useful intuition that supports many tasks related to modelling in ASP, such as when checking for correctness of the encoding by enumerating all answer sets and when debugging a *spurious* answer set (which does not correspond to any solution).

In the following subsections, we present the ASP modelling language as laid out in the ASP-Core-2 specification by Calimeri et al. (2020). First, we describe the core syntax of ASP, which introduces *rules* and *programs* as the central syntactic objects. Second, we describe the core semantics of ASP as a mapping  $\mathcal{AS}$  from programs to answer sets. Given these definitions, a problem specification can be modelled as a program  $P$ . An ASP solver, which implements the semantics of ASP, can then be used to compute  $\mathcal{AS}(P)$ . We conclude this section with discussions on core extensions, related complexity results, and on ASP solvers. For further introductory reading we suggest (Lifschitz 2019) for a gentle introduction to ASP and the *clingo* solver, (Eiter, Ianni, and Krennwallner 2009) for a primer on the subject and (Beierle and Kern-Isberner 2019) for a discussion of ASP in the context of other knowledge representation formalisms.

### 2.2.1 Core Syntax

Like for any logical language, we start our discussion of ASP by defining the syntax. We use a first-order predicate logic (PL1) as a basis and define its signature as follows.<sup>1</sup>

**Definition 2.2.1** (Signature: Beierle and Kern-Isberner 2019, p. 48). A *(PL1)-Signature*  $\Sigma = (Func, Pred)$  consists of a set *Func* of *function names* and a set *Pred* of *predicate names*. Items in *Func* are represented as  $f/\alpha \in Func$ , where  $f$  denotes the function name itself and  $\alpha$  its arity. Analogously, items in *Pred* are denoted  $\mathbf{p}/\beta \in Pred$ . Function names with arity zero are called *symbolic constants*. Usually, the existence of at least one symbolic constant in *Func* is assumed.

The next definition concerns the terms of the language, representing the objects in our domain of discourse.

**Definition 2.2.2** (Terms: Calimeri et al. 2020). Let  $\Sigma = (Func, Pred)$  be a signature and let *Var* be a set of variables. In its most basic form, a  $(\Sigma)$ -*term* is a symbolic constant  $c$  (for  $c/0 \in Func$ ), a *variable*  $V \in Var$ , or a *functional term*  $f(t_1, \dots, t_n)$ , built from a function name  $f/n \in Func$  and other terms  $t_1, \dots, t_n$ .

To reason about our domain, we need to make statements about the relations between its objects. As usual, this is done via atoms and literals.

**Definition 2.2.3** (Atoms and literals: Calimeri et al. 2020). Let  $\Sigma = (Func, Pred)$  be a signature. A *predicate atom*  $\mathbf{p}(t_1, \dots, t_n)$  consists of a predicate name  $\mathbf{p}/n \in Pred$  and  $\Sigma$ -terms  $t_1, \dots, t_n$ . A *classical literal* is a predicate atom  $\mathbf{p}(t_1, \dots, t_n)$  or its strong negation  $\neg\mathbf{p}(t_1, \dots, t_n)$ .<sup>2</sup> A *naf-literal* is of the form  $l$  and **not**  $l$ , where  $l$  is a classical literal. Here, **not** refers to negation as failure.

Next, we need a way to formulate more complex statements about the interaction of multiple atoms and literals.

**Definition 2.2.4** (Rules and programs: Calimeri et al. 2020). A *rule* is of the form

$$h_1 \mid \dots \mid h_m \leftarrow b_1, \dots, b_n.$$

with  $h_1, \dots, h_m$  being classical literals and  $b_1, \dots, b_n$  being naf-literals for  $m \geq 0$  and  $n \geq 0$ . We call  $head(r) \doteq h_1 \mid \dots \mid h_m$  the *head* of the rule and  $body(r) \doteq b_1, \dots, b_n$  its *body*. A rule with an empty body, i.e.  $n = 0$ , is called a *fact*. A rule with an empty head, i.e.  $m = 0$ , is called a *constraint*. A set of rules forms a *program*  $P$ .

<sup>1</sup>Calimeri et al. (2020) imply the existence of a signature in their definitions, but do not mention it explicitly. We adopt the definition of Beierle and Kern-Isberner (2019, p. 48) who use it as a basis for their definition of logic programs and ASP (pp. 278–313).

<sup>2</sup>In the literature, strongly negated atoms have been referred to as both (classical) literals (Beierle and Kern-Isberner 2019, p. 290; Leone et al. 2006; Gelfond and Lifschitz 1991) and (classical) atoms (Calimeri et al. 2020).

In natural language, a rule can be read as follows: If all of  $b_1, \dots, b_n$  are true, then at least one of  $h_1, \dots, h_m$  has to be true as well. A program can be viewed as a collection of such statements.

Finally, we need to distinguish syntactic objects that are free of variables.

**Definition 2.2.5** (Ground objects: Calimeri et al. 2020). We call a syntactic object (term, atom, literal, rule, program, etc.), *ground* if it contains no variables.

### 2.2.2 Core Semantics

The goal of this subsection is to define the mapping  $\mathcal{AS}$ , which assigns answer sets to programs. Roughly speaking, an answer set is a judgement about the truth values of ground atoms constructed from a signature. Formally, this can be represented as follows:

**Definition 2.2.6** (Herbrand universe, base and interpretation: Calimeri et al. 2020; Eiter, Ianni, and Krennwallner 2009, p. 8; Beierle and Kern-Isberner 2019, p. 290–297). Given a signature  $\Sigma = (Func, Pred)$ , we call  $\mathcal{HU}(\Sigma)$  the *Herbrand universe* of  $\Sigma$ . It is the set of all ground terms that can be formed from constants and function names in  $Func$ . Further, the *Herbrand literal base*  $\mathcal{HB}_{Lit}(\Sigma)$  of  $\Sigma$  is the set of all ground classical literals, formed with predicate symbols in  $Pred$  and terms in  $\mathcal{HU}(\Sigma)$ . A set of literals  $I \subseteq \mathcal{HB}_{Lit}(\Sigma)$  is *consistent* if, for all predicate atoms  $q$ ,  $\{q, \neg q\} \not\subseteq I$ . A *Herbrand literal interpretation*  $I \subseteq \mathcal{HB}_{Lit}(\Sigma)$  is a consistent subset of the Herbrand literal base. The set of all Herbrand literal interpretations is denoted as  $\mathcal{HI}_{Lit}(\Sigma)$ . For some atom  $q$ , we say that  $q$  is *true* if  $q \in I$ ,  $q$  is *false* if  $\neg q \in I$ , and  $q$  is *undefined* (or unknown) otherwise.

Herbrand literal interpretations are often used in the context of a program without an explicit mention of the signature. We denote by  $\Sigma_P$  the signature which is constructed by collecting all function and predicate names that occur in  $P$ . We also introduce the shorthand notation:  $\mathcal{HU}(P) \doteq \mathcal{HU}(\Sigma_P)$ ,  $\mathcal{HB}_{Lit}(P) \doteq \mathcal{HB}_{Lit}(\Sigma_P)$  and  $\mathcal{HI}_{Lit}(P) \doteq \mathcal{HI}_{Lit}(\Sigma_P)$ .

The notion of truth needs to be extended to other objects in our language. This is done via a satisfaction relation. For now, we consider the ground case.

**Definition 2.2.7** (Ground satisfaction: Calimeri et al. 2020). Given a signature  $\Sigma$  and a consistent interpretation  $I \in \mathcal{HI}_{Lit}(\Sigma)$ , a *satisfaction relation*  $\models$  can be defined for various objects formed from function and predicate names in  $\Sigma$ .<sup>3</sup> In the following, we denote ground classical literals by  $l$ , ground rules by  $r = h_1 \mid \dots \mid h_m \leftarrow b_1, \dots, b_n$ . and ground programs by  $P$ .

- $I \models l$  iff  $l \in I$ .

<sup>3</sup>Although not used by Calimeri et al. (2020) in particular, the notation  $I \models f$  is common when talking about the truth of a logical formula  $f$ . In an ASP context, the notation is used by e.g. by Eiter, Ianni, and Krennwallner (2009, p. 10).

- $I \models \mathbf{not} \ l$  iff  $l \notin I$ .
- $I \models h_1 \mid \dots \mid h_m$  iff  $I \models h_i$  for some  $1 \leq i \leq m$ .
- $I \models b_1, \dots, b_n$  iff  $I \models b_i$  for all  $1 \leq i \leq n$ .
- $I \models h_1 \mid \dots \mid h_m \leftarrow b_1, \dots, b_n$  iff  $I \models h_1 \mid \dots \mid h_m$  or  $I \not\models b_1, \dots, b_n$ .
- $I \models P$  iff  $I \models r$  for all rules  $r \in P$ .

If  $I \models o$  for some syntactic object  $o$  (literal, rule, program, etc.), we also say that  $o$  is *true* with respect to  $I$ .

Let's consider the case of general programs including variables. Here, satisfaction is defined indirectly, by first obtaining an equivalent ground program and then applying ground satisfaction as just defined. The first step is formalised as follows:

**Definition 2.2.8** (Substitution and ground instantiation: Calimeri et al. 2020). A mapping  $\theta : Var \rightarrow \mathcal{HU}(P)$  from a set of variables  $Var$  to the Herbrand universe of a program  $P$  is called a *substitution*. The application of a substitution is denoted  $o\theta$  for some object  $o$  (a term, atom, literal, or rule) and replaces each occurrence of a variable  $V \in Var$  in  $o$  by  $\theta(V)$ . The application  $r\theta$  is a *ground instance* of an object  $o$  if the domain of  $\theta$  is the set of all variables in  $o$ . The *ground instantiation*  $Gnd(P)$  of a program  $P$  is the set of all ground instances of all the rules in  $P$ .

Now, we are ready to define answer sets in terms of the stable model semantics, first introduced by Gelfond and Lifschitz (1988) and later extended to aggregates (Faber, Leone, and Pfeifer 2004; Faber, Pfeifer, and Leone 2011), which are introduced later in this section.

**Definition 2.2.9** (Model, reduct and answer sets: Calimeri et al. 2020). Given a program  $P$  and a consistent interpretation  $I \in \mathcal{HI}(P)$ , we call  $I$  a *model* of  $P$  if  $I \models Gnd(P)$ . We define the *reduct*  $P^I$  of  $P$  with respect to  $I$  as  $P^I \doteq \{r \in Gnd(P) \mid I \models \mathit{body}(r)\}$ . We call a consistent interpretation  $I \in \mathcal{HI}(P)$  an *answer set* of  $P$  if  $I$  is a  $\subseteq$ -minimal model of  $P^I$ . The set of all answer sets of  $P$  is denoted by  $\mathcal{AS}(P)$ .

### 2.2.3 Core Extensions

The so-far presented language of ASP reflects the current consensus on its basic features, but the development and addition of more features is still an active research topic. Some such extensions are simply cosmetic in nature, making the syntax more concise and intuitive. Others have a strong complexity-theoretic impact on the language and the kinds of problems that can be described in it. In this subsection, we take a short look at some of the more established extensions (also included in the ASP-Core-2 specification). This is not a complete and rigorous treatment of the syntax and semantics, but rather intended to give the intuition behind these features. For the full specification we refer to Calimeri et al. (2020).

## Reasoning About Strings and Integers

Besides generic symbolic constants, it is possible to express *string constants* and *integers* in ASP-Core-2. Syntactically, the set of constants is first extended to contain quoted texts and the integer numbers, which can be used to form terms and predicates just like other constants. Second, *arithmetic terms* like  $-(t)$ ,  $(t + u)$ ,  $(t - u)$ ,  $(t \star u)$  and  $(t/u)$  can be formed from terms  $t$  and  $u$ . Third, *built-in atoms* are introduced to the language, allowing to make comparative statements between terms  $t$  and  $u$ , such as  $t \leq u$ ,  $t = u$  or  $t \neq u$ . Semantically, the notions of *well-formed substitutions* and *arithmetic evaluations* are introduced. These are used to modify the grounding  $Gnd(P)$  of a program  $P$  such that all ground instances of rules adhere to the arithmetic principles and nested arithmetic expressions are fully evaluated. For evaluating built-in atoms, such as  $t = u$ , a total order  $\preceq$  over  $\mathcal{HU}(P)$  is constructed. For the given example then,  $I \models t = u$  iff  $t \preceq u$  and  $u \preceq t$ . Within integers,  $\preceq$  corresponds to the natural order of integers. Within strings and symbolic constants, a lexicographical order is defined. Further orderings are defined between terms of different types as well as for functions.

## Aggregates

Often, it is useful to aggregate information over a set of terms. In ASP-Core-2, this can be formulated with *aggregate atoms* of the form  $\#aggr E \prec u$ . Intuitively, this expression summarises the elements in the set  $E$  via a function  $\#aggr$  into a single term (for example by counting the elements in  $E$ ) and compares the result to a term  $u$  with respect to  $\prec$ . An element in  $E$  is referred to as *aggregate element* and has the form  $t_1, \dots, t_m : l_1, \dots, l_n$ , where  $t_1, \dots, t_m$  are terms and  $l_1, \dots, l_n$  are naf-literals.  $\#aggr \in \{\#count, \#sum, \#max, \#min\}$  is referred to as an *aggregate function name* and  $\prec \in \{<, \leq, =, \neq, >, \geq\}$  as an *aggregate relation*. Given an aggregate atom  $a$ , *aggregate literals* are formed as usual by  $a$  and its negation **not**  $a$ .

The semantics of aggregates are accounted for by extending the definitions of *ground satisfaction* and *ground instantiation*. Given an interpretation  $I$  and a collection of ground aggregate elements  $E$ , the mapping  $Eval(E, I) \doteq \{(t_1, \dots, t_m) \mid t_1, \dots, t_m : l_1, \dots, l_n \in E \text{ and } I \models l_1, \dots, l_n\}$  defines the set of tuples of terms to be aggregated. Also, an *aggregate function* is defined for every aggregate function name, which maps a set of tuples of terms to a single term. For example,  $\#count(T) \doteq |T|$  if  $T$  is finite and  $+\infty$  otherwise. Then, the condition for ground satisfaction is as follows:  $I \models \#aggr E \prec u$  iff  $I \models t \prec u$ , where  $t \doteq \#aggr(Eval(E, I))$  is the resulting term after applying the aggregate function and  $t \prec u$  can be evaluated using the semantics for built-in atoms. The ground satisfaction of an aggregate literal is treated like that of any other (naf-)literal.

The ground instantiation,  $Gnd(P)$ , needs to be adjusted as well. The introduction of aggregates demands a distinction between *global substitutions* (from the set of variables appearing outside of aggregate elements) and *local substitutions* (from the set of variables in an aggregate element). For a collection  $E$  of aggregate elements, the *instantiation* of  $E$  is defined as:  $Inst(E) \doteq \bigcup_{e \in E} \{e\theta \mid \theta \text{ is a well-formed substitution for } e\}$  Given a

rule  $r$ , a ground instance is generated in two steps: First, apply a well-formed global substitution  $\theta$  to  $r$ . Second, consider every aggregate atom  $\#aggr E \prec u$  that appears in  $r\theta$ , and replace  $E$  by  $Inst(E)$ .  $Gnd(P)$  contains all rules obtainable by this procedure.

The definition of aggregates can be extended to include aggregate atoms of the form  $u \prec \#aggr E$  and  $u_1 \prec_1 \#aggr E \prec_2 u_2$ . Rules with these formulations can be translated into rules with the original formulation.

### Choice Rules

Another useful concept in the ASP-Core-2 specification is that of a *choice rule*. It captures the intuition behind choosing a subset of certain size from a set. In ASP, such a set is expressed as a collection  $C$  of *choice elements*  $h : l_1, \dots, l_k$  (where  $h$  is a classical literal and  $l_1, \dots, l_k$  are naf-literals). Intuitively,  $h$  is available for the choice if  $l_1, \dots, l_k$  are satisfied. The number of elements chosen from the set is defined with a *choice atom*  $u \prec C$ , for an aggregate relation  $\prec$  and term  $u$ . Putting a choice atom in the head of a rule results in a *choice rule*  $u \prec C \leftarrow b_1, \dots, b_n$ . The semantics of a choice rule are defined indirectly, as choice rules can be translated into rules without choice atoms (by using aggregates).

### Optimisation

Sometimes, it is desired to indicate preference of one answer set over another. In the ASP-Core-2 input language, such preference orderings can be expressed through *weak constraints*.<sup>4</sup> They have the form  $:\sim b_1, \dots, b_n. [w@l, t_1, \dots, t_m]$ . Intuitively, if the literals  $b_1, \dots, b_n$  in the body of the weak constraint are true, then a weight penalty  $w$  is applied to the answer set. The term  $l$  defines the priority level at which  $w$  is applied. This enables the tracking of several distinct optimisation criteria at different levels, with higher levels having higher priority.

Weak constraints are grounded as usual. Then, given a program  $P$  and an answer set  $I \in \mathcal{AS}(P)$ , the tuples  $(w@l, t_1, \dots, t_m)$  of all ground weak constraints where  $I \models b_1, \dots, b_n$  are collected into the set  $Weak(P, I)$ .<sup>5</sup> Assuming that the number of non-zero weights in  $Weak(P, I)$  is finite and that all weights  $w$  are integers, the sum of weights at priority level  $l$  is denoted by  $P_l^I \doteq \sum_{(w@l, t_1, \dots, t_m) \in Weak(P, I)} w$ . Given two answer sets  $I, I' \in \mathcal{AS}(P)$ , we say that  $I$  is *dominated* by  $I'$  if  $P_l^{I'} < P_l^I$  for some level  $l$  and  $P_{l'}^{I'} = P_{l'}^I$  for all higher levels  $l' > l$ . An answer set  $I \in \mathcal{AS}(P)$  is *optimal* if it is not dominated by any other answer set.

<sup>4</sup>Gebser, Kaminski, Kaufmann, and Schaub (2012, p. 30) provide alternative minimisation statements of the form  $minimize \{l_1 = w_1@p_1, \dots, l_n = w_n@p_n\}$ . where  $l_i$  are literals,  $w_i$  are weights and  $p_i$  are priority levels.

<sup>5</sup>Note that tuples with exactly the same terms will be merged into one (as is usual for sets), even if they originate from different weak constraints. In this case, the modeller can add more terms to distinguish those tuples and make sure they are summed up as intended.

### 2.2.4 Computational Complexity

In this section, we briefly discuss complexity results related to ASP. Following the works of Eiter and Gottlob (1995) as well as Dantsin et al. (2001), we consider two problems. First, given a program  $P$ , the problem of  $\mathcal{AS}$ -consistency is to decide whether  $\mathcal{AS}(P) \neq \emptyset$ . Second, given a program  $P$  and a propositional formula  $f$ , the problem of  $\mathcal{AS}$ -entailment is to decide whether  $f$  is a logical consequence of every answer set  $I \in \mathcal{AS}(P)$  (Eiter and Gottlob 1995, p. 314).<sup>6</sup> Other relevant reasoning tasks can be found in e.g. Gebser, Kaminski, Kaufmann, and Schaub (2012, p. 29).

A range of complexity results exists for different fragments of the ASP programming language. We first consider a program  $P$  that is ground and defined as in Section 2.2.1 and Section 2.2.2 (without any of the extensions). The most simple case occurs when every rule  $r \in P$  is **not**-free, has a disjunction-free head (i.e. its head consists of exactly one atom  $h_1$ ), and does not contain strong negation (i.e. it has no literals of the form  $\neg p(t_1, \dots, t_n)$  in its head or body).<sup>7</sup> Then,  $\mathcal{AS}$ -consistency is trivial due to the guaranteed existence of exactly one answer set and  $\mathcal{AS}$ -entailment is P-complete (Dantsin et al. 2001, Theorem 4.2). If **not** is allowed in the rules of  $P$  (but still, strong negation is disallowed and rule heads are assumed to be disjunction-free), then deciding  $\mathcal{AS}$ -consistency is NP-complete (Dantsin et al. 2001, Theorem 5.7) and deciding  $\mathcal{AS}$ -entailment is co-NP-complete (Dantsin et al. 2001, Theorem 5.8). If, in addition, disjunctions are allowed in rule heads (but strong negation is still disallowed), deciding  $\mathcal{AS}$ -consistency is  $\Sigma_2^P$ -complete (Eiter and Gottlob 1995, Theorem 3.3) and deciding  $\mathcal{AS}$ -entailment is  $\Pi_2^P$ -complete (Dantsin et al. 2001, Theorem 6.2 citing Eiter and Gottlob 1995, Theorem 3.4). Adding strong negation does not change the complexity results: deciding  $\mathcal{AS}$ -consistency remains  $\Sigma_2^P$ -complete (Eiter and Gottlob 1995, Theorem 7.1) and deciding  $\mathcal{AS}$ -entailment remains  $\Pi_2^P$ -complete (Eiter and Gottlob 1995, Theorem 7.2).

Considering non-ground programs, both  $\mathcal{AS}$ -consistency and  $\mathcal{AS}$ -entailment are generally undecidable unless some syntactic restrictions are applied, as outlined in e.g. Dantsin et al. (2001, pp. 389–390) as well as Eiter, Ianni, and Krennwallner (2009, p. 28). The syntactic restrictions for ASP-Core-2 are discussed by Calimeri et al. (2020, Section 5). In the presence of restrictions, an additional grounding step is needed for computing  $Gnd(P)$ . In the worst case, the size of  $Gnd(P)$  is exponential in the size of  $P$ , resulting in an exponential blowup in complexity (Eiter, Ianni, and Krennwallner 2009, p. 28). More detailed results are presented in Dantsin et al. (2001, e.g. Theorems 4.5, 5.8, 6.2, and 6.4).

<sup>6</sup>The exact definition of  $\mathcal{AS}$ -entailment varies in some of the presented complexity results. For once, the definition depends on the semantics used to evaluate the given fragment of logic programming. To this end, the problems of  $S$ -consistency and  $S$ -entailment can be defined for a general semantics  $S$  (Eiter and Gottlob 1995, pp. 294, 314). Other definitions of the problem check entailment for a set of ground atoms  $A$  instead of a (more general) propositional formula (Dantsin et al. 2001, p. 380).

<sup>7</sup>Rules of this type are known as *definite Horn clauses* and programs consisting of only these rules are referred to as *classical logic programs* (Beierle and Kern-Isberner 2019, p. 279) or as *pure logic programs* (Dantsin et al. 2001, p. 375).

With the addition of weak constraints, it becomes possible to express problems beyond the complexity of  $\Sigma_2^P$ , even in the ground case (Eiter, Ianni, and Krennwallner 2009, p. 51). Problems relevant to optimal answer sets and their complexity results are summarised by Gebser, Kaminski, Kaufmann, and Schaub (2012, p. 30) citing (upon others) Gebser, Kaminski, and Schaub (2011).

The complexity results show that the ASP formalism is capable of expressing a wide range of complex search and optimisation problems. But the high complexity can also be a problem when one is not careful in the design of ASP encodings. This needs to be kept in mind when modelling in ASP.

### 2.2.5 Answer-Set Solvers and the Clingo System

The expressive power of ASP and its inherent computational complexity demand efficient tools for computing answer sets. Fortunately, such tools exist in the form of ASP solvers. Eiter, Ianni, and Krennwallner (2009, p. 48) provide a short overview over the basics of ASP solver technology. Typically, the architecture follows a two-level structure, analogous to how the semantics are defined. First, a *grounding step* is applied to translate arbitrary programs into equivalent ground versions. Second, *model search* is applied to the ground program, which returns one or more answer sets.

Both steps are highly optimised in state-of-the-art ASP solvers. For this thesis, we work with *clingo* (Gebser, Kaminski, Kaufmann, Ostrowski, et al. 2016), a system that utilises the *gringo* and *clasp* sub-components for grounding and solving, respectively.<sup>8</sup> *Clingo* complies with the ASP-Core-2 specification and introduces other extensions beyond the core specifications. For an introduction we suggest Gebser, Kaminski, Kaufmann, and Schaub (2012), who describe the modelling language in Chapters 2 and 3, and have an in-depth discussion on the solver technology in Chapters 4–7.

When modelling with *clingo*, one usually does not use the mathematical notation that we used so far to define the syntax and semantics. Instead, the user writes code in a raw input format, which is also part of the ASP-Core-2 specification (Calimeri et al. 2020). We use both the mathematical notation and the raw input format. From the *clingo*-specific syntax, we make use of *#show* directives, which can be used to hide atoms in the answer set (Gebser, Kaminski, Kaufmann, Ostrowski, et al. 2016, p. 2:4–2:5).

**Example.** For illustration purposes, we consider the logic program  $P$  from the *penguin problem* (Beierle and Kern-Isberner 2019, pp. 289). In mathematical notation, we write

---

<sup>8</sup>Information about the distribution of *clingo* can be found at <https://potassco.org/clingo/>.

it as:

$$P \doteq \left\{ \begin{array}{l} \mathbf{bird}(X) \leftarrow \mathbf{penguin}(X). \\ \mathbf{flies}(X) \leftarrow \mathbf{bird}(X), \mathbf{not} \neg\mathbf{flies}(X). \\ \neg\mathbf{flies}(X) \leftarrow \mathbf{penguin}(X). \\ \mathbf{penguin}(\mathbf{tweety}). \\ \mathbf{bird}(\mathbf{polly}). \end{array} \right\}$$

In raw input format, this program is written as:

```
1 bird(X) :- penguin(X).
2 flies(X) :- bird(X), not -flies(X).
3 -flies(X) :- penguin(X).
4 penguin(tweety).
5 bird(polly).
```

## 2.3 Reinforcement Learning

In this chapter, we develop the theoretical foundations of reinforcement learning (Sutton and Barto 2018). We start with a generic intuition about artificial agents and how they may interact with an environment in which they are situated. We then formalise the agent-environment interaction using probability-theoretic notions and introduce the concept of Markov decision processes. Using this framework, we then define a clear optimisation goal for artificial agents.

### 2.3.1 The Agent-Environment Interface

In Artificial Intelligence, it is common to model tasks as a series of interactions between an *agent* and an *environment* in which that agent is situated (Russell and Norvig 2013). The environment has an internal state and the agent is able to observe that state to some degree. The agent is also able to influence the environment's state by performing actions in the environment. *Task environments* may be seen as the formulation of a problem to be solved by the agent. A task environment typically has some performance measure, such as a numerical reward after each action, specifying whether the behaviour of an agent was desirable or not.

The agent-environment interface is a flexible framework for modeling a broad variety of tasks. Russell and Norvig (2013, pp. 42–47) develop several dimensions in which environments can be differentiated: If the agent's ability to observe is sufficient to fully describe the environments state, the environment is *fully observable*. If only part of the environment is revealed with each observation, the environment is *partially observable*. For example, partial observability can occur due to noisy or missing sensor data. If only one agent performs actions on the environment, we speak of a *single-agent* setting. In a *multi-agent* setting, multiple agents perform actions in the same environment. Additionally, each agent tries to optimise its behaviour according to some performance measure, which is also influenced by the behaviour of the other agents. One may further

distinguish between *cooperative* and *competitive* multi-agent environments, depending on whether the performance measures of the agents are aligned or not. An environment is *deterministic* if, given some current state and some performed action, the next state is always the same. An environment is *stochastic* if the next state is sampled from some probability distribution, which in general does not need to depend on the current state and action. Note that a deterministic, partially observable environment can appear stochastic due to the uncertainty introduced by partial observability. In a *static* environment, the environment changes only after an agent performs some action. This is opposed to a *dynamic* environment, which may also change in between actions, i.e. while the agent is deliberating. If each percept-action cycle corresponds to a discrete time point, we speak of a *discrete* environment. On the other hand, if the environment state exhibits continuous change, the environment is considered to be *continuous*. An environment may also be *known* or *unknown* depending on whether the environment’s dynamics are given or not. Sutton and Barto (2018, p. 57) make a further distinction: In *episodic* environments, the agent-environment interaction is broken up into several distinct episodes.<sup>9</sup> Within each episode, a series of agent-environment interactions occurs. Some choice of action early in the episode can influence future states and rewards within the same episode, but not in other episodes. In a *continuing* environment, all interactions happen in a single, indefinitely long episode. Thus, when choosing an action, an agent needs to consider the influence of an action for an indefinite horizon of future states and rewards.

For this thesis, we put the focus on environments that are fully observable, single-agent, static, discrete and episodic (in the sense of Sutton and Barto 2018, p. 57). The discussed methods work for both deterministic and stochastic environments. As typical for reinforcement learning, we assume that the environment dynamics are unknown. Thus, in order to succeed in a given task environment, an agent needs to experience the environment on-line and improve its behaviour along the way. The environment dynamics are not known to the agent, but we assume that domain knowledge about the environment and its dynamics is known in general and can be used for modelling purposes (i.e. by hand-crafting an abstraction, as discussed in Section 2.5, Section 3.2 and in our case studies).

### 2.3.2 Stochastic Control Processes

The following references mainly (Sutton and Barto 2018, Chapter 3) but borrows notation and concepts by Hutter (2009) and Schickinger and Steger (2002, Chapter 4). By assumption, the environment is discrete, which allows for time to be modelled as an infinite sequence of discrete *time points*  $t \in \{0, 1, 2, \dots\} = \mathbb{N}_0$ .<sup>10</sup> We also assume full

<sup>9</sup>Russell and Norvig (2013, p. 44) use the term “episodic” to define a different concept: Each episode consists only of a single percept and action and the feedback is immediate (e.g. in a classification task). This is opposed to *sequential* environments in which multiple percept-action loops may be performed.

<sup>10</sup>It is also possible to model time as a finite sequence of discrete time points, as done e.g. in Kakade (2003, p. 22). Generally, a distinction can be made between *finite-horizon*, *infinite-horizon* and *indefinite-horizon* models, with infinite-horizon models being mathematically most convenient (van Otterlo 2008, p. 41).

observability. Thus, at any time point  $t$ , the agent perceives a full description of the current environmental state  $s_t \in S$  from a set of possible states  $S$ .<sup>11</sup> The agent then reacts by performing some action  $a_t \in A_{s_t} \subseteq A$ , where  $A_{s_t}$  denotes the set of actions admissible in state  $s_t$  and  $A$  denotes the set of all actions available to the agent at some time during the interaction.<sup>12</sup> This causes the environment to update its state and the global clock will advance by one time point. Finally, the agent perceives a new state  $s_{t+1} \in S$ .

After each interaction, the agent additionally receives a numerical reward  $r_{t+1} \in R \subseteq \mathbb{R}$  from a set of possible rewards. This indicates the immediate value of the agent's last action. The action-perception loop continues indefinitely, producing a history  $h \in H$  of interactions, where  $h \doteq (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots)$  and  $H \subseteq S \times (A \times R \times S)^\infty$ .<sup>13</sup> The set  $H$  of all possible histories forms a sample space for a probability distribution by assigning each  $h \in H$  a probability value  $\Pr\{h\}$ .<sup>14</sup> For every time point  $t$ , we define the random variables  $R_t : H \rightarrow R$  to describe the reward received at  $t$ ,  $S_t : H \rightarrow S$  for the environmental state at  $t$  and  $A_t : H \rightarrow A$  for the action performed at  $t$ . The history up to state  $t$  can be described by another random variable  $H_t : H \rightarrow S \times (A \times R \times S)^t$  defined by  $H_t \doteq (S_0, A_0, R_1, S_1, \dots, A_{t-1}, R_t, S_t)$ .

The rewards in a history are usually aggregated into the *return*, a numeric measurement of the agent's performance. Intuitively, a history with a high return is to be preferred over histories with lower ones. Depending on the exact setting, multiple aggregation methods exist and may be appropriate. See Sutton and Barto (2018, p. 54) and van Otterlo (2008, p. 41) for some background and overview of the basic aggregation methods. For this thesis, we use the following aggregation method:

**Definition 2.3.1** ( $\gamma$ -discounted return: Sutton and Barto 2018, p. 55). Given a discount rate  $\gamma \in [0, 1]$  and a time point  $t \in \mathbb{N}_0$ , the  $\gamma$ -discounted return at  $t$  is defined as random variable  $G_t : H \rightarrow \mathbb{R}$  with:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

<sup>11</sup>More generally, one may assume partial observability. In this case, the current internal state  $s_t$  of the environment is not fully known to the agent. Instead, the agent may receive an observation  $o_t \in O$  from an observation space. This allows for modelling hidden information as well as noisy sensors, as used for example in the original definition of the *history-based decision process* (Majeed and Hutter 2018).

<sup>12</sup>We assume that  $|A_s| > 0$  for all states  $s \in S$ .

<sup>13</sup>Hutter (2009) slightly disagrees with Sutton and Barto (2018, p. 48) in this definition. In addition to an initial state (or, in their case, initial observation), Hutter assumes also an initial reward, which would give a slightly different definition of  $H \doteq S \times R \times (S \times A \times R)^\infty$ . To be precise, Hutter allows for histories of arbitrary (finite) length, i.e.  $H \doteq S \times R \times \bigcup_{n \in \mathbb{N}} (S \times A \times R)^n$ , which corresponds to an *indefinite-horizon* setting according to e.g. van Otterlo (2008, p. 41). In Section 2.3.5, we discuss how histories of finite length can be modelled using absorbing states. Also note that our definition turned the  $=$  into a  $\subseteq$  to account for the possibility that some actions may not be admissible in some states. To account for this exactly (in light of definition 2.3.3) we should state  $H \doteq (\Psi \times R)^\infty \times S$ .

<sup>14</sup>Note that Sutton and Barto (2018) do not define the sample space in the way it is done here. Our inspiration for the formulation came from Schickinger and Steger (2002, p. 169) where the sample space of a Markov chain is described explicitly.

The discount rate determines how much future rewards are worth in relation to immediate rewards (Sutton and Barto 2018, p. 55). If  $\gamma = 0$ , only the immediate reward is considered. As  $\gamma$  approaches 1, future rewards become more and more influential.

### 2.3.3 Markov Environments

A core assumption of many reinforcement learning algorithms is that the environment has the *Markov property*:

**Definition 2.3.2** (Markov property: Sutton and Barto 2018, p. 49; van Otterlo 2008, p. 39). The environment has the *Markov property* (is said to be *Markov*) if for all time points  $t \in \mathbb{N}_0$ , sub-histories  $h_t = (s_0, a_0, r_0, s_1, \dots, a_{t-1}, r_t, s_t)$ , possible current actions  $a_t \in A_{s_t}$ , possible next states  $s_{t+1} \in S$  and rewards  $r_{t+1} \in R$  it holds that:

$$\begin{aligned} & \Pr(\{R_{t+1} = r_{t+1} \wedge S_{t+1} = s_{t+1}\} \mid \{H_t = h_t \wedge A_t = a_t\}) \\ &= \Pr(\{R_{t+1} = r_{t+1} \wedge S_{t+1} = s_{t+1}\} \mid \{S_t = s_t \wedge A_t = a_t\}) \end{aligned}$$

In other words, only the current state and action have influence on the probability of the next state and reward. When deliberating which action  $a_t$  is best to perform next, an agent can ignore the entire history  $h_t$  besides its last state  $s_t$  and still make an optimal decision. A Markov environment is usually formalised as a Markov decision process.

**Definition 2.3.3** (Markov decision process: Sutton and Barto 2018, p. 48; Ravindran 2004, p. 9). Let  $S$  be a finite set of states,  $A$  be a finite set of actions, and  $R \subset \mathbb{R}$  be a finite set of numerical rewards. Let  $\gamma \in [0, 1]$  be a discount rate. Let  $\Psi \subseteq S \times A$  define the admissibility of actions in each state. Let  $p : \Psi \times R \times S \rightarrow [0, 1]$  with  $\sum_{s' \in S} \sum_{r \in R} p(s', r \mid s, a) = 1$  for all  $(s, a) \in \Psi$ . Given some  $s' \in S$ ,  $r \in R$  and  $(s, a) \in \Psi$ ,  $p(s', r \mid s, a)$  denotes the probability of reaching  $s'$  as a next state with a reward of  $r$  after performing action  $a$  in state  $s$ . We call the tuple  $(S, A, R, \Psi, p, \gamma)$  a *finite, discrete-time and  $\gamma$ -discounted Markov decision process (MDP)*, whose *dynamics* are defined by  $p$ . Further, let  $A_s \doteq \{a \mid (s, a) \in \Psi\}$  define the set of *admissible actions* in state  $s$ , for all  $s \in S$ .<sup>15</sup>

An MDP characterises a Markov environment if the following holds for all  $t \in \mathbb{N}_0$ ,  $(s, a) \in \Psi$ ,  $r \in R$  and  $s' \in S$ :

$$p(s', r \mid s, a) = \Pr(\{S_{t+1} = s' \wedge R_{t+1} = r\} \mid \{S_t = s \wedge A_t = a\})$$

In some of our referenced literature, the transition function  $p$  and reward function  $r$  are defined separately.<sup>16</sup>To ensure compatibility we give the following definition:

<sup>15</sup>The notation is borrowed from Ravindran (2004) and Ravindran and Barto (2003).

<sup>16</sup>See for example the definitions in van Otterlo (2008, p. 38) and in the work on abstraction presented in Section 2.5.

**Definition 2.3.4** (Transition and reward functions: Sutton and Barto 2018, p. 49). Let  $(S, A, R, \Psi, p, \gamma)$  be an MDP. The probability of ending up in state  $s'$  after performing action  $a$  in state  $s$  is denoted by the *transition function*  $p(s' | s, a)$ . The expectation of the immediate reward to be received when performing action  $a$  in state  $s$  is denoted by the *reward function*  $r(s, a)$ . Both functions are defined as follows:

$$p(s' | s, a) \doteq \sum_{r' \in R} p(r', s' | s, a)$$

$$r(s, a) \doteq \sum_{r' \in R} r \sum_{s' \in S} p(r', s' | s, a)$$

### 2.3.4 Formalisation of the Agent and Its Goals

Next, we specify the agent's behaviour within the MDP-framework. At every time point  $t \in \mathbb{N}_0$ , after perceiving the current state  $s_t$ , the agent needs to select an admissible action  $a_t \in A_{s_t}$ . This choice is modelled via a policy.

**Definition 2.3.5** (Policy: Sutton and Barto 2018, p. 58; Ravindran 2004, p. 9). Given an MDP  $(S, A, R, \Psi, p, \gamma)$ , a *stochastic policy* is a mapping  $\pi : \Psi \rightarrow [0, 1]$  with  $\sum_{a \in A_s} \pi(a | s) = 1$  for all  $s \in S$ .  $\pi$  induces a probability distribution over the actions  $a \in A_s$  for every state  $s \in S$ . The probability of choosing any single action is denoted by  $\pi(a | s)$ .<sup>17</sup>

Having both  $p$  and  $\pi$  defined, we can revisit the interaction of agent and environment and how it affects the probability distribution  $\Pr\{h\}$  across histories in  $H$ . At every time point  $t \in \mathbb{N}_0$ , the agent perceives  $s_t$ , and performs a sampled action  $a_t \sim \pi(\cdot | s_t)$ . Next, the environment samples  $r_{t+1}, s_{t+1} \sim p(\cdot | s_t, a_t)$  and the process repeats. The probability of any history  $h = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots) \in H$  generated by  $p$  and  $\pi$  is defined by:<sup>18</sup>

$$\Pr_{p, \pi}\{h\} \doteq \text{init}(s_0) \cdot \prod_{t \in \mathbb{N}_0} \pi(a_t | s_t) \cdot p(r_{t+1}, s_{t+1} | s_t, a_t)$$

where  $\text{init} : S \rightarrow [0, 1]$  denotes some initial state distribution with  $\sum_{s \in S} \text{init}(s) = 1$ .

Recall from definition 2.3.1 that an agent's performance is measured in terms of returns. Given the probability distribution induced by  $p$  and  $\pi$ , we can use the notion of returns to assign utility measures to states and actions.

**Definition 2.3.6** (Value functions: Sutton and Barto 2018, p. 58). The utility of being in state  $s$  following  $\pi$ , is expressed as *state-value function*  $v_\pi(s)$  for policy  $\pi$ . Similarly, the *action-value function*  $q_\pi(s)$  for policy  $\pi$  expresses the utility of being in state  $s$ , taking

<sup>17</sup>The definition in Sutton and Barto (2018, p. 58) does not provide mathematical notation with respect to admissible actions. We borrow notation from Ravindran and Barto (2003) to account for this.

<sup>18</sup>This formulation was inspired by a similar formulation for Markov chains by Schickinger and Steger (2002, p. 169), but we did not see it mentioned in MDP descriptions from the literature.

action  $a$  and then following  $\pi$  for all future actions. Both are defined in terms of the expected (discounted) return:<sup>19</sup>

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_{p,\pi}(G_t \mid \{S_t = s\}) & \forall s \in S \\ q_\pi(s, a) &\doteq \mathbb{E}_{p,\pi}(G_t \mid \{S_t = s \wedge A_t = a\}) & \forall (s, a) \in \Psi \end{aligned}$$

Given these definitions, the task of “solving” an MDP can be understood as an optimisation problem, searching the policy space for some policy  $\pi$  that maximises  $v_\pi$  in all states. To this end, a partial ordering can be defined with respect to the set of all policies.

**Definition 2.3.7** (Ordering of policies, optimal policy: Sutton and Barto 2018, p. 62). Let  $\pi$  and  $\pi'$  be policies. Then  $\pi$  is *better than or equal* to  $\pi'$ , denoted by  $\pi \succeq \pi'$ , iff  $v_\pi(s) \geq v_{\pi'}(s)$  for all  $s \in S$ . A policy  $\pi$  is *optimal* iff  $\pi \succeq \pi'$  for all policies  $\pi'$ . An optimal policy is denoted by  $\pi_\star$ .

The existence of an optimal policy is ensured, but not its uniqueness (Sutton and Barto 2018, p. 62). More than one optimal policy may exist in general, but all share the same optimal value functions:

**Definition 2.3.8** (Optimal value functions: Sutton and Barto 2018, pp. 62–64).

$$\begin{aligned} v_\star(s) &\doteq \max_{\pi} \{v_\pi(s)\} & \forall s \in S \\ q_\star(s, a) &\doteq \max_{\pi} \{q_\pi(s, a)\} & \forall (s, a) \in \Psi \end{aligned}$$

### 2.3.5 Episodic MDP

In this thesis we focus on agent-environment interactions in an *episodic task setting* (Sutton and Barto 2018, pp. 54, 57; van Otterlo 2008, p. 40). The agent does not experience the environment as one long interaction history, as is the case in a *continuous task setting*, but as a series of distinct, finite histories (called episodes) with clearly defined start and terminal states.<sup>20</sup> This motivates the following definition:

**Definition 2.3.9** (Episodic MDP: van Otterlo 2008, p. 40). Given an MDP  $(S, A, R, \Psi, p, \gamma)$ , a set of *terminal states*  $S_{\text{term}} \subseteq S$  and an initial state distribution determined by  $\text{init} : S \rightarrow [0, 1]$  with  $\sum_{s \in S} \text{init}(s) = 1$ , we define the tuple  $(S, A, R, \Psi, p, \gamma, \text{init}, S_{\text{term}})$  as an *episodic MDP*.

<sup>19</sup>Note that we implicitly universally quantify over all time points  $t \in \mathbb{N}_0$  where  $\Pr_{p,\pi}\{S_t = s\} > 0$  in the case of  $v_\pi$  and  $\Pr_{p,\pi}\{S_t = s \wedge A_t = a\} > 0$  for  $q_\pi$ . This is necessary due to the definition of conditional probabilities. So, if the chance of visiting some state (resp. state-action pair) is zero for all time points, then its corresponding value function is undefined.

<sup>20</sup>We follow the description of van Otterlo (2008, p. 40) here, with some discrepancies. Instead of goal states, we use the more neutral notion of terminal states as described in Sutton and Barto (2018, pp. 54, 57).

In addition to the usual conditions for general MDPs, an EMDP characterises a Markov environment if the initial states are accounted for in the probability distribution, i.e.  $\Pr\{S_0 = s\} = \text{init}(s)$  for all  $s \in S$ , and all terminal states are modelled as *absorbing states* (Sutton and Barto 2018, p. 57; van Otterlo 2008, p. 40). Formally,  $p(0, s | s, a) = 1$  for all terminal states  $s \in S_{\text{term}}$  and available actions  $a \in A_s$ . So, actions performed in a terminal state will always result in a reward of zero and a transition to the same terminal state. This formulation enables us to integrate EMDPs into the so-far established framework by mapping histories of finite length to histories of infinite length with equivalent returns.

### 2.3.6 Relational MDP

So far, we treated states and actions as monolithic symbols with no information besides distinguishability from other states.<sup>21</sup> In many domains, more is known about the structure of a state. To reflect this, the state space  $S$  can be expressed as the Cartesian product of  $n$  state variables:  $(x_1, x_2, \dots, x_n) = s \in S \subseteq X_1 \times X_2 \times \dots \times X_n$ , where  $X_i$  is the domain of values assignable to variable  $x_i$  (for  $1 \leq i \leq n$ ). The action space can be expressed similarly.<sup>22</sup>

For this work, we are mainly interested in states and actions with a relational structure. Many domains are naturally described using a domain of objects and their relations, formalised as atoms (Definition 2.2.3). This is well motivated in van Otterlo (2008, Chapter 4). Formally, such a relational state space can be expressed by assigning every atomic formula a state variable with a Boolean domain, indicating its truth value. An equivalent but more concise formulation is to represent every state as a Herbrand interpretation (Definition 2.2.6).<sup>23</sup>

**Definition 2.3.10** (Relational MDP: van Otterlo 2008, p. 168). Let  $\Sigma = (F, P_S \cup P_A)$  be a signature of state predicate definitions  $P_S$ , action predicate definitions  $P_A$  and function definitions  $F$ , with  $P_S \cap P_A = \emptyset$ .<sup>24</sup> Let  $(S, A, R, \Psi, p, \gamma)$  be an MDP. We

<sup>21</sup>This MDP representation can be called *monolithic* (Ravindran 2004, p. 49), *atomic* (Moore 2018, p. 24) or simply *flat* (Grounds and Kudenko 2007).

<sup>22</sup>The notation was borrowed from Grounds and Kudenko (2007). An MDP represented in this way is sometimes called a *factored MDP* (e.g. Ravindran and Barto 2003; Givan, Dean, and Greig 2003). Using state variables, it is also possible to represent the MDP dynamics in a structured fashion, for example by deploying a dynamic Bayesian Network (Ravindran and Barto 2003; Ravindran 2004, p. 50). Such an MDP is called a *structured MDP*. For an introduction to this topic we refer to van Otterlo (2008, Chapter 3.5).

<sup>23</sup>A logical structure can be modelled in many ways. Moore (2018, p. 24) distinguishes between atomic representations, propositional representations, relational representations and deictic representations. Also, first-order representations can be considered, where states correspond to full first-order structures instead of Herbrand interpretations (van Otterlo 2008, p. 219).

Instead of (Herbrand) interpretations as states, the logical structure can also be expressed as a labelling function, which is more in the style of Kripke structures. Examples of this include Camacho et al. (2019) and Icarte et al. (2019). Note that, in order to recreate the above definition, a labelling function must be assumed to be bijective, such that the state can be recovered from the labelling.

<sup>24</sup>Only constants are allowed in the function signature of the original definition (van Otterlo 2008, p. 168). Extending this definition to include function symbols generally leads to Herbrand universes of

call  $(\Sigma, S, A, R, \Psi, p, \gamma)$  a *Relational MDP (RMDP)* if  $S \subseteq \mathcal{HLit}((F, P_S))$  and  $A \subseteq \mathcal{HB}_{Lit}((F, P_A))$  such that  $S$  and  $A$  are finite. We require that no strong negation is used in both  $S$  and  $A$ ,<sup>25</sup> such that a state  $s \in S$  can be viewed as a set of ground atoms  $s \doteq \{\mathbf{p}_1(\bar{t}_1), \dots, \mathbf{p}_n(\bar{t}_n)\}$  and an action  $a \in A$  as a single ground atom  $a \doteq \mathbf{p}(\bar{t})$ , where  $\bar{t}, \bar{t}_1, \dots, \bar{t}_n$  are used to denote various ground terms.<sup>26</sup> Ground satisfaction is defined as in Section 2.2.7, with  $s \models q$  iff  $q \in s$ . This definition can be extended to an *episodic RMDP*  $(\Sigma, S, A, R, \Psi, p, \gamma, init, S_{term})$  using definition 2.3.9.

## 2.4 Tabular Solution Methods and Q-Learning

In the following we discuss solution methods for reinforcement learning. We assume an episodic Markov environment characterised by  $(S, A, R, \Psi, p, \gamma, init, S_{term})$  in which the dynamics  $p$  are unknown. The admissible actions  $A_s$  are assumed to be provided to the agent whenever it reaches state  $s \in S$ . The goal of the agent is to learn  $\pi_*$  through a series of online interactions, divided into episodes.

A plethora of solution methods are available for the problem of reinforcement learning. Many of those are based on the important theoretical result of the *Bellman optimality equations* (Sutton and Barto 2018, pp. 63–64):

$$\begin{aligned} v_*(s) &\doteq \max_{a \in A_s} \{q_*(s, a)\} & \forall s \in S \\ q_*(s, a) &\doteq \sum_{r', s'} p(r', s' \mid s, a) \cdot (r' + \gamma v_*(s')) & \forall (s, a) \in \Psi \end{aligned}$$

To begin with, an agent can estimate an approximate model  $\tilde{p} \approx p$  of the environmental dynamics through interaction. Having a model available enables the use of dynamic programming and planning methods as discussed by Sutton and Barto (2018) in Chapters 4 and 8. One can even construct the appropriate system of Bellman equations in an attempt to solve them analytically or by using linear programming (Papadimitriou and Tsitsiklis 1987). These methods are appropriately called *model-based* methods. Another approach is offered by *model-free* methods, such as Monte Carlo methods and temporal-difference learning (Sutton and Barto 2018, Chapters 5, 6).<sup>27</sup> These aim to learn the

infinite size. We see no harm in this extension as long as  $S$  and  $A$  remain finite subsets in the spirit of definition 2.3.3. Note that van Otterlo (2008, p. 183) also mentions the possibility of RMDPs defined on infinitely large Herbrand interpretations.

<sup>25</sup>The original definition uses the *Herbrand atom base* (not the *Herbrand literal base*) and operates under the closed world assumption (where atoms not in  $s \in S$  are assumed to be *false*). We adapt this version to stay compatible with the CARCASS Framework, which is based on normal logic programs (Chapter 3). We do not use strong negation in our own ASP encodings. Allowing strong negation in  $S$  does however make sense and could be very useful in a partially observable context. It is not immediate whether allowing strong negation in  $A$  (i.e. the inclusion of strongly negated actions) makes sense.

<sup>26</sup>Strictly speaking, since states are now sets, notation would demand renaming states from lowercase  $s$  to uppercase  $S$  and sets of states from  $S$  to  $\mathcal{S}$ . However, this would be more confusing than helpful, so we stick with the original notation of denoting states by lowercase  $s$  and the set of states by uppercase  $S$ .

<sup>27</sup>The difference between *model-based* and *model-free* methods is e.g. discussed in Sutton and Barto (2018, p. 159) and van Otterlo (2008, p. 44).

optimal policy directly from experience, without a need to estimate  $p$ . All methods mentioned so far are closely related conceptually, as they all organize their search around the approximation of value functions. The programmatic representation of value functions as tables earns them the name of *tabular solution methods* (Sutton and Barto 2018, p. 23). A third category of methods focuses on the search in policy space. This includes a range of heuristic policy search methods, using e.g. evolutionary algorithms (van Otterlo 2008, p. 129) but also policy gradient methods (Sutton and Barto 2018, Chapter 13).

### 2.4.1 Q-Learning

In this thesis, we focus on *Q-learning* (Watkins 1989; Sutton and Barto 2018, p. 131), which falls into the category of (tabular) temporal difference methods. *Q-learning* is also an *off-policy* method (Sutton and Barto 2018, p. 110): The samples gathered during online interaction come from a *behaviour policy* which we denote by  $\pi_{\text{Behaviour}}$ . This policy is different from the policy we want to learn, the *target policy*, denoted by  $\pi_{\text{Target}}$ . Behaviour and target policy can be completely decoupled but don't need to be. *Q-learning* uses the samples gathered from  $\text{Pr}_{p, \pi_{\text{Behaviour}}}$  to obtain an estimate  $q \approx q_*$ , which in turn is used to construct the target policy  $\pi_{\text{Target}} \approx \pi_*$  analogous to the structure of the Bellman equation for  $v_*$ . The behaviour policy usually follows to some degree the target policy but also adds random actions such that all state-action pairs in  $\Psi$  are visited continually (see correctness of *Q-learning* below). This is formalised in terms of ( $\epsilon$ )-greedy policies as follows.

**Definition 2.4.1** ( $\epsilon$ -greedy policy: Sutton and Barto 2018, pp. 64, 100). Given an action-value function  $q : \Psi \rightarrow \mathbb{R}$ , we denote its *greedy policy* by  $\pi_q$ . Further, given some  $\epsilon \in [0, 1]$ , we denote the  $\epsilon$ -*greedy policy* of  $q$  by  $\pi_{q, \epsilon}$ . Both are defined according to the following equations, where ties related to  $\arg \max$  are split arbitrarily and  $\pi_{\text{rnd}}$  denotes a policy that samples actions from a uniformly random distribution.

$$\begin{aligned} \pi_{\text{rnd}}(a \mid s) &\doteq 1/|A_s| && \forall (s, a) \in \Psi \\ \pi_q(a \mid s) &\doteq \begin{cases} 1 & \text{for } a = \arg \max_{a' \in A_s} \{q(s, a')\} \\ 0 & \text{otherwise} \end{cases} && \forall (s, a) \in \Psi \\ \pi_{q, \epsilon}(a \mid s) &\doteq \epsilon \cdot \pi_{\text{rnd}}(a \mid s) + (1 - \epsilon) \cdot \pi_q(a \mid s) && \forall (s, a) \in \Psi \end{aligned}$$

We now consider *Q-learning* as presented in Algorithm 2.1. Two nested loops iterate over the episodes and the time points within each episode, respectively. At the start of an episode, the initial state  $s_0$  is sampled (line 2). Then, for every time point  $t$  in that episode, an action  $a_t$  is sampled according to the current state  $s_t$  and the behaviour policy  $\pi_{\text{Behaviour}} \doteq \pi_{q, \epsilon}$  (line 4). Next, the reward  $r_{t+1}$  and the state  $s_{t+1}$  are sampled according to the current state  $s_t$ , the sampled action  $a_t$ , and the dynamics of the environment (line 5). Having obtained a full sample  $(s_t, a_t, r_{t+1}, s_{t+1})$ , the value function is updated (line 6). The inner loop continues until the current state is terminal or until an artificial time limit  $\tau \in \mathbb{N}$  is reached. The outer loop continues indefinitely. The main output of

the algorithm is its target policy  $\pi_{\text{Target}} \doteq \pi_q \simeq \pi_*$ . Note that the artificial time limit was not part of the original definition by Sutton and Barto (2018, p. 131).

---

**Algorithm 2.1:**  $Q$ -learning using  $\epsilon$ -greedy exploration policies (Sutton and Barto 2018, p. 131) with time limit

---

**Input:** Episodic MDP  $(S, A, R, \Psi, p, \gamma, \text{init}, S_{\text{term}})$   
**Parameter:** Exploration factor  $\epsilon > 0$   
**Parameter:** Fixed learning rate  $\alpha \in (0, 1]$   
**Parameter:** Initial q-values  $q : \Psi \rightarrow \mathbb{R}$ ,  
with  $q(s, a) = 0$  for all  $a \in A_s$  if  $s \in S_{\text{term}}$   
**Parameter:** Time limit  $\tau \in \mathbb{N}$   
**Output:**  $q \simeq q_*$ ,  $\pi_q \simeq \pi_*$

```

1 foreach episode do
2    $s_0 \sim \text{init}(\cdot)$ 
3   repeat for every time point  $t = 0, 1, 2, \dots$ 
4      $a_t \sim \pi_{q, \epsilon}(\cdot \mid s_t)$ 
5      $r_{t+1}, s_{t+1} \sim p(\cdot \mid s_t, a_t)$ 
6      $q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A_{s_{t+1}}} \{q(s_{t+1}, a)\} - q(s_t, a_t)]$ 
7   until  $s_{t+1} \in S_{\text{term}}$  or  $t \geq \tau$ 
8 end

```

---

### 2.4.2 Correctness of $Q$ -Learning

A core principle of  $Q$ -learning is that of *value iteration* (Jaakkola, Jordan, and S. P. Singh 1994; Sutton and Barto 2018, p. 83), a dynamic programming method that may be applied when the dynamics of the environment  $p$  are known. It turns the Bellman equation

$$q_*(s, a) = \mathbb{E}_{p, \pi_*} (R_{t+1} + \gamma \max_{a'} \{q_*(S_{t+1}, a')\} \mid \{S_t = s \wedge A_t = a\}) \quad \forall (s, a) \in \Psi$$

into a series of functions  $(q_k : \Psi \rightarrow \mathbb{R})_{k \in \mathbb{N}_0}$  with the recurrence relation:<sup>28</sup>

$$q_{k+1}(s, a) = \sum_{s' \in S} \sum_{r' \in R} p(r', s' \mid s, a) \cdot (r' + \gamma \max_{a'} \{q_k(s', a')\}) \quad \forall (s, a) \in \Psi, \forall k \in \mathbb{N}_0$$

The series has  $q_*$  as a fixed point (due to the equality in the Bellman optimality equation), and indeed can be shown to converge to  $q_*$  for arbitrary initial values  $q_0$ . The operation in the equation above is called an *expected update* as defined by Sutton and Barto (2018, p. 74).

---

<sup>28</sup>Jaakkola, Jordan, and S. P. Singh (1994) present a slightly different recurrence relation using separate transition and reward functions, with  $q_{k+1}(s, a) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) v_k(s')$  and  $v_k(s) = \max_{a \in A_s} \{q_k(s, a)\}$ . We adapt the formulation to the Bellman optimality equation presented by Sutton and Barto (2018, p. 83).

Without the availability of  $p$ , there is no way to compute the expected values directly. Instead,  $Q$ -learning uses samples obtained from interactions with the environment to approximate the expected values. Sutton and Barto (2018, pp.121, 172) refer to these updates as *sample updates*. Proofs of correctness exist for multiple versions of  $Q$ -learning, including proofs due to Watkins (1989), Tsitsiklis (1994) and Jaakkola, Jordan, and S. P. Singh (1994). We follow the proof of Tsitsiklis (1994), who define  $Q$ -learning as a random process over some sample space  $\Omega$ , subject to:<sup>29</sup>

$$\mathbf{Q}_{k+1,s,a} \doteq (1 - \alpha_{k,s,a}) \cdot \mathbf{Q}_{k,s,a} + \alpha_{k,s,a} (\mathbf{R}'_{k,s,a} + \gamma \max_{a'} \{\mathbf{Q}_{k,S'_{k,s,a},a'}\}) \quad \forall (s,a) \in \Psi, \forall k \in \mathbb{N}_0$$

The random variables can be interpreted as follows. At every iteration  $k \in \mathbb{N}_0$  and for every  $(s,a) \in \Psi$ ,  $\mathbf{Q}_{k,s,a} : \Omega \rightarrow \mathbb{R}$  describes an estimate of  $q_*(s,a)$  and  $\alpha_{k,s,a} : \Omega \rightarrow [0,1]$  describes a step-size coefficient (analogous to the learning rate  $\alpha$  in Algorithm 2.1). The random variables  $\mathbf{R}'_{k,s,a} : \Omega \rightarrow R$  and  $\mathbf{S}'_{k,s,a} : \Omega \rightarrow S$  describe an independent sample obtained from the environment, thus following  $\Pr_{\Omega}\{\mathbf{R}'_{k,s,a} = r' \wedge \mathbf{S}'_{k,s,a} = s'\} = p(r', s' | s,a)$  for all  $r' \in R$  and  $s' \in S$ . There are two peculiarities that differentiate the above recurrence relation from Algorithm 2.1. First, at every step  $k$ , the environment is sampled for every state-action pair. Second, the samples are gathered independently, without the need to adhere to a consistent history. Running this algorithm in its full generality would thus require that the environment can be reset to any state at any point in time, a level of control that we do not assume in episodic environments.<sup>30</sup> Luckily, the recurrence relation and convergence theorems are general enough to include our use case. Algorithm 2.1 can be derived by setting  $\alpha_{k,s,a}$  to zero for all state-action pairs, except the one currently visited, removing the need to sample these pairs. Analogous to  $t$ ,  $k$  can be advanced after every agent-environment interaction and across episodes. The proof presented by Tsitsiklis (1994) shows that, for every  $(s,a) \in \Psi$ ,  $\mathbf{Q}_{k,s,a}$  converges to  $q_*(s,a)$  with probability 1, under the following conditions:

1. Finiteness. It is assumed that  $\Phi$  is finite and that the variance of  $\mathbf{R}_{k,s,a}$  is finite for every  $k \in \mathbb{N}_0$  and  $(s,a) \in \Psi$ .
2. Outdated information. The convergence proof includes the possibility of outdated information (e.g. in an asynchronous setting). This requires the assumption that all outdated information is eventually updated.
3. Stepsize coefficients.
  - a) The assumptions on  $\alpha_{k,s,a}$  are relatively lax, including the possibility of incorporating information from the past (i.e. steps before  $k$ ) into the decision of which state-action pairs to update and with which learning rate, as long as the decision on  $\alpha_{k,s,a}$  is made before any samples at steps  $\geq k$  are realised.

<sup>29</sup>Tsitsiklis (1994) define the optimal policy in terms of minimality with respect to a cost signal, and not in terms of maximality with respect to a reward signal. We adjusted the formula to fit our definition.

<sup>30</sup>In general, one can assume different sampling models with respect to the MDP, as outlined e.g. in Kakade (2003, p. 28). In the context of this publication, we assume an *online simulation model* in episodic MDPs.

- b)  $\sum_{k=0}^{\infty} \alpha_{k,s,a} = \infty$  for all  $(s, a) \in \Psi$ . As a consequence, every state-action pair needs to be visited and sampled an infinite number of times.
  - c)  $\sum_{k=0}^{\infty} \alpha_{k,s,a}^2 \leq c$  for some constant  $c$  and for all  $(s, a) \in \Psi$ .
4. Discounting. If the MDP is discounted (i.e.  $\gamma < 1$ ), no further assumptions are required. If no discounting is applied ( $\gamma = 1$ ), the situation is more complicated and several cases can be distinguished. For details we refer to Tsitsiklis (1994).

Let's see how the above results apply to Algorithm 2.1. Condition 1 is satisfied by definition 2.3.3. Condition 2 is satisfied trivially, since all information is updated at every step. The satisfaction of condition 3a depends on the choice of learning rates, as well as the choice of state-action pairs to be sampled at step  $k$ . The learning rate is kept constant (except for when it is set to zero). The state-action pair to sample is fully determined by the past history of interactions of the  $\epsilon$ -greedy behaviour policy with the environment, which uses information about the past (i.e. the current  $q_*$ -estimate) but none from the future. So, condition 3a is satisfied.

If we assume an infinite time limit ( $\tau = \infty$ ), condition 3b is satisfied because of two reasons. First, the learning rate is constant. Second, since every admissible action has a non-zero chance of being chosen by the  $\epsilon$ -greedy behaviour policy, every (non-terminal) state-action pair is updated an infinite number of times. The argument for exploration based on  $\epsilon$ -greedy policies is developed by Sutton and Barto (2018, Chapter 5.4) in the context of Monte Carlo Control. If the time limit is finite, some states may not be reachable from the start state within the time limit. So, for convergence it needs to be ensured that the time limit is higher than the shortest path between every state and some initial state.<sup>31</sup> The  $q$ -updates for terminal states are generally ignored. However, due to their definition (modelling them as absorbing states), we know that all future rewards will be zero and so  $q(s, a) = 0$  for any action  $a \in A_s$ . This is precisely the assumption for initial  $q$ -values in Algorithm 2.1. So, no  $q$ -updates are needed because the initial  $q$ -values are already optimal.

Condition 3c is *not* satisfied assuming a constant  $\alpha$  but insight into this case can be gained by considering the theory of stochastic iterative algorithms, on which the proof above is based. Bertsekas and Tsitsiklis (1996, p. 135), cited by Sutton and Barto (2018, pp. 33, 44), provide an intuition for the behaviour of such algorithms when a constant stepsize parameter is used.<sup>32</sup> The intuition, applied to  $Q$ -learning, is the following. Although the convergence of  $Q_{k,s,a}$  to  $q_*(s, a)$  exactly is generally not possible, as the estimates continue to change based on the more recently obtained samples. It is however possible to obtain estimates within a neighbourhood of  $q_*$ . The size of that neighbourhood can be influenced by adjusting  $\alpha$ , with lower values leading to a smaller neighbourhood.

<sup>31</sup>Technically for both finite and infinite  $\tau$ , we also need to assume that every state within the episodic MDP is in principle reachable from one of the initial states. If this is not the case, then some states may never be visited. But such a state would not be relevant to solving the EMDP anyways.

<sup>32</sup>Another proof of convergence of  $Q$ -learning, using the theory of stochastic approximation, is also provided in the same work (Bertsekas and Tsitsiklis 1996, pp. 247–249).

Finally, condition 4 needs to be checked individually for every MDP.

### 2.4.3 Complexity of $Q$ -Learning and Efficient Exploration

Strehl, Li, and Littman (2009) identified several dimensions in which the complexity of online reinforcement learners can be studied. The *space complexity* can be determined as usual. The computational complexity however needs some more theoretical groundwork, because online reinforcement learning is a never-ending process in principle. To gain more insight, the computational complexity can be divided into two dimensions. First, *per-timestep computational complexity* measures the amount of computation time needed during a single interaction with the environment. In the case of Algorithm 2.1, this involves selecting the next action  $a_t$ , as well as updating  $q(s_t, a_t)$  and  $\pi_{q,\epsilon}$ . Second, *sample complexity* (also called *learning complexity*) provides an upper bound for the number of interactions needed (within some likelihood with respect to all possible runs of the learning process) for the algorithm to learn a policy that is expected to operate within some error bound of the optimal policy (Strehl, Li, and Littman 2009, p. 2417). For an introduction we refer to Li (2012). Note that the notion of sample complexity is of particular importance in settings where obtaining samples is expensive, as is the case in many real-world applications.

For  $Q$ -learning, per-timestep computational complexity is in  $\mathcal{O}(\ln|A|)$  and space complexity is in  $\Theta(|S| \cdot |A|)$  (e.g. Li 2012, p. 196). These results do not include the complexity of the exploration strategy, i.e. the computation needed to choose a next action  $a_t$  (in our case using the  $\epsilon$ -greedy strategy). The sample complexity of  $Q$ -learning in an online setting depends on several factors, one being the exploration strategy. For  $\epsilon$ -greedy behaviour policies in particular, MDPs can be constructed where the sample complexity is exponential in  $|S|$  (Li 2012, p. 177).

This highlights the need for more efficient exploration strategies. A wide range of such strategies has been proposed, some of which are discussed in van Otterlo (2008, pp. 53, 58). Efforts to determine general bounds on the sample complexity of certain exploration techniques have led to the development of PAC-MDP algorithms, such as the Rmax algorithm (Li 2012, p. 186) and Delayed  $Q$ -learning (Li 2012, p. 196). These algorithms are guaranteed to find the optimal policy (within some error margin) in a polynomially bounded number of samples (with some probability).

For the purposes of this thesis however, the  $\epsilon$ -greedy strategy will suffice, in combination with another exploration strategy called *optimistic  $q$ -values initialisation* (e.g. van Otterlo 2008, p. 54). Recall that the initial  $q$ -values are provided to Algorithm 2.1 as parameters. Initialising them "optimistically" means setting them to a high value in relation to the rewards that are believed to be encountered. When state-action pairs are visited, their  $q$ -values will decrease as lower-valued rewards are encountered. Critically, the so-far least explored state-action pairs will have the highest  $q$ -values and are thus chosen next by the  $\epsilon$ -greedy strategy. The time limit  $\tau$  influences exploration as well, by introducing a bias

towards initial states and away from states reachable only via long paths, which is useful in some cases but detrimental in others.

## 2.5 State-Action Pair Abstraction

In the previous section, we established that tabular reinforcement learning algorithms can be computationally costly in relation to the size of the state space. This is a problem for very large state and action spaces, which, according to Sutton and Barto (2018, p. 195), are commonly found in practical applications of reinforcement learning. As an example, Grounds and Kudenko (2007) observe that the state space of factored MDPs (see Section 2.3.6) grows exponentially in the number of variables. The same can be said about a relationally factored MDP  $(\Sigma, S, A, R, \Psi, p, \gamma)$ , where the potential size of the state space equals the number of Herbrand interpretations for the given signature  $\Sigma$ .<sup>33</sup> This phenomenon of combinatorially large state spaces has been referred to as *state space explosion* (e.g. Grounds and Kudenko 2007, p. 77) or the *curse of dimensionality* (van Otterlo 2008, p. 74 citing Bellman 1957). It is the cause of three particular (van Otterlo 2008, p. 74) problems. First, the space required to store  $q_\pi : \Psi \rightarrow \mathbb{R}$  can simply be too large. Second, even if the convergence of the reinforcement learning algorithm is theoretically guaranteed, the learning process often takes too long in practice to produce an acceptable policy. This is due to the large number of state-action pairs that need to be explored. Third, a growing state space does not imply that the number of high-reward states grows as well. These can become increasingly hard to find as their number shrinks in proportion to low-reward states. This is particularly problematic in planning domains such as the blocks world (Chapter 4), where there might be just one success state for any number of blocks (van Otterlo 2008, p. 74). The problem has an interdisciplinary nature and different research fields provide different perspectives on possible solution methods. An overview of different approaches is provided by van Otterlo (2008, Chapter 3), including an introduction to state abstraction (Chapter 3.4), a method that we investigate below. For an in-depth introduction we refer to Sutton and Barto (2018, pp. 195–338), where function approximation, a particularly popular approach, is discussed.

In this thesis, we treat state- and state-action pair abstraction techniques in the spirit of Li, T. J. Walsh, and Littman (2006) citing Giunchiglia and T. Walsh (1992): abstraction can be viewed as a mapping from one problem representation to another. A solution to the original, *concrete* problem can be found by first solving the abstract representation and then transferring back the abstract solution to the concrete representation. The

<sup>33</sup>There are  $2^{\mathcal{HB}((F, P_S))}$  possible Herbrand atom interpretations (based on a Herbrand atom base  $\mathcal{HB}$ ) in a if  $F$  consists of constants only, and even more if we consider Herbrand literal bases. If there are no restrictions on the arity of function names, then the size of the Herbrand Universe can be infinite, and with it also the number of possible interpretations. That being said, a logical theory can be used to restrict the size of the state space. In the blocks world for example (see Chapter 4), it is not possible to have more than one block stacked directly on top of another and interpretations with this property can be discarded. The number of possible state is however still enormous.

success of this approach depends on how well problem-relevant properties of the concrete problem are preserved in the abstract representation. Properties that are not relevant to the solution may be ignored, which hopefully reduces the complexity of the abstract problem, making it easier to solve than the concrete one.

This notion can be adopted to MDPs. If the concrete problem is framed as finding  $\pi_*$  in an MDP  $M = (S, A, R, \Psi, p, \gamma)$  and  $|\Psi|$  is too large, an abstract problem representation may be induced by a mapping  $\phi : \Psi \rightarrow \hat{\Psi}$  from the concrete state-action space to a smaller, abstract state-action space. Here,  $\hat{\Psi} \subseteq \hat{S} \times \hat{A}$  defines the admissible set of abstract actions in abstract states for the new representation. The interaction between agent and environment is lifted into the abstract representation as well. An abstract policy  $\hat{\pi}$  determines the choice of abstract actions and the environment responds by providing new abstract states and (the original) rewards. A new set of histories  $\hat{H}$  is generated, with  $(\phi(s_0, a_0), r_1, \phi(s_1, a_1), r_2, \dots) \in \hat{H}$  iff  $(s_0, a_0, r_1, s_1, a_1, r_2, \dots) \in H$ ,  $H$  being the set of histories for the concrete MDP. Ideally, if the right properties are preserved, one can apply reinforcement learning algorithms to the abstract problem, compute  $\hat{q}_*$  and  $\hat{\pi}_*$  and use these results to derive a solution  $\pi_*$  to the concrete problem. In the following, we present abstraction frameworks by Li, T. J. Walsh, and Littman (2006)<sup>34</sup> as well as Ravindran (2004) in an attempt to understand abstraction mappings and the properties that they need to preserve.

### 2.5.1 State Abstraction

First, we present abstraction mappings that are limited to abstract state spaces, leaving the action space unchanged. This subset of abstraction mappings has been studied extensively and detailed results are available about the convergence of  $Q$ -learning and the transferability of abstract solutions back into the concrete state space.

**Definition 2.5.1** (State abstraction: Li, T. J. Walsh, and Littman 2006). Let  $M = (S, A, R, \Psi, p, \gamma)$  be an MDP with  $\Psi = S \times A$ .<sup>35</sup> We define a *state abstraction function*  $\phi : S \rightarrow \hat{S}$  as a mapping from  $S$  to an abstract set  $\hat{S}$  of states. Let  $w : S \rightarrow [0, 1]$  be a weighting function subject to  $\sum_{x \in [s]_\phi} w(x) = 1$  for all  $s \in S$ . The state abstraction function  $\phi$  together with  $w$  induce an abstract MDP  $\hat{M} = (\hat{S}, A, R, \hat{\Psi}, \hat{p}, \gamma)$  with  $\hat{\Psi} \doteq \hat{S} \times A$  and, for all  $(s, a) \in \Psi$ ,  $s' \in S$  and  $r' \in R$ , it holds that:<sup>36</sup>

$$\hat{p}(r', \phi(s') \mid \phi(s), a) \doteq \sum_{x \in [s]_\phi} w(x) \cdot \sum_{y \in [s']_\phi} p(r', y \mid x, a)$$

<sup>34</sup>This work was also cited by van Otterlo (2008, p. 92).

<sup>35</sup>In their definition of MDPs and policies, Li, T. J. Walsh, and Littman (2006) do not mention the admissibility of actions. We interpret this as  $\Psi = S \times A$ . If the admissibility of some actions is restricted, then  $[s_1]_\phi = [s_2]_\phi$  should imply  $A_{s_1} = A_{s_2}$ , which would need adjustment in the definitions of the provided abstraction types. At the very least, one might decide to ignore (possibly optimal) actions that are not admissible by all states forming the abstract state, i.e.:  $\hat{\Psi} \doteq \{(\phi(s), a) \mid (s, a) \in \Psi, a \in \bigcap_{x \in [s]_\phi} A_x\}$ .

<sup>36</sup>Li, T. J. Walsh, and Littman (2006) defined the abstract MDP in terms of separate transition and reward models. Using our definition of  $\hat{p}$  and the equations from definition 2.3.4, we can reconstruct the

In the context of an abstract MDP  $\hat{M}$ , it is possible to define abstract policies  $\hat{\pi} : \hat{\Psi} \rightarrow [0, 1]$  as well as abstract value functions  $\hat{q}_{\hat{\pi}} : \hat{\Psi} \rightarrow \mathbb{R}$  and  $\hat{v}_{\hat{\pi}} : \hat{S} \rightarrow \mathbb{R}$ , just as usual. Note that many different abstract MDPs can be induced from the concrete MDP, based on the choice of  $w$ . We come back to that later. First, let's see how state abstraction may be applied in an online, off-policy learning setting, such as  $Q$ -learning.

The interaction with the environment happens via a behaviour policy  $\hat{\pi}_{\text{Behaviour}}$ , which now realises actions based on abstract states, i.e.  $a_t \sim \hat{\pi}_{\text{Behaviour}}(\cdot \mid \phi(s_t))$ . The environment is used to sample  $r_{t+1}, s_{t+1} \sim p(s_t, a_t)$  exactly as before. An estimate  $\hat{q} : \hat{\Psi} \rightarrow \mathbb{R}$  is maintained and updated as follows:<sup>37</sup>

$$\hat{q}(\phi(s_t), a_t) \leftarrow \hat{q}(\phi(s_t), a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A} \{\hat{q}(\phi(s_{t+1}), a)\} - \hat{q}(\phi(s_t), a_t)]$$

The abstract target policy can be derived from the  $q$ -value estimates as usual, with  $\hat{\pi}_{\text{Target}} \doteq \hat{\pi}_{\hat{q}}$ . Similarly, an  $\epsilon$ -greedy behaviour policy may be defined as  $\hat{\pi}_{\text{Behaviour}} \doteq \hat{\pi}_{\hat{q}, \epsilon}$ . Ideally, we would like this process to converge in a way that allows us to construct an optimal policy from the abstract target policy, i.e. we want the following to hold after convergence:

$$\hat{\pi}_{\text{Target}}(a \mid \phi(s)) \approx \pi_{\star}(a \mid s) \quad \forall (s, a) \in \Psi$$

Unfortunately, as shown by Li, T. J. Walsh, and Littman (2006), convergence is not guaranteed. To understand why, we need to investigate the role of the weighting function  $w$ . It reflects that not all concrete states are equally likely to be encountered during online learning and thus that the amount of contribution of individual states from the concrete MDP to the dynamics of the abstract MDP may vary. Critically, the likelihood of encountering states depends on the behaviour policy. In other words, different behaviour

original definitions as follows:

$$\begin{aligned} \hat{r}(\phi(s), a) &= \sum_{r \in R} r \sum_{\hat{s} \in \hat{S}} \hat{p}(\hat{s}, r \mid \phi(s), a) = \sum_{r \in R} r \sum_{\hat{s} \in \hat{S}} \sum_{x \in [s]_{\phi}} w(x) \sum_{y \in \phi^{-1}(\hat{s})} p(y, r \mid x, a) \\ &= \sum_{x \in [s]_{\phi}} w(x) \sum_{r \in R} r \sum_{\hat{s} \in \hat{S}} \sum_{y \in \phi^{-1}(\hat{s})} p(y, r \mid x, a) \\ &= \sum_{x \in [s]_{\phi}} w(x) \sum_{r \in R} r \sum_{y \in S} p(y, r \mid x, a) = \sum_{x \in [s]_{\phi}} w(x) \cdot r(x, a) \\ \hat{p}(\phi(s') \mid \phi(s), a) &= \sum_{r \in R} \hat{p}(\phi(s'), r \mid \phi(s), a) = \sum_{r \in R} \sum_{x \in [s]_{\phi}} w(x) \sum_{y \in [s']_{\phi}} p(y, r \mid x, a) \\ &= \sum_{x \in [s]_{\phi}} w(x) \sum_{y \in [s']_{\phi}} \sum_{r \in R} p(y, r \mid x, a) = \sum_{x \in [s]_{\phi}} w(x) \sum_{y \in [s']_{\phi}} p(y \mid x, a) \end{aligned}$$

<sup>37</sup>Compare with line six in Algorithm 2.1 and with the version of  $Q$ -learning presented by Li, T. J. Walsh, and Littman (2006). Note that we left  $\alpha$  constant in the update presented here, to make it consistent with Algorithm 2.1. But the convergence conditions discussed in Section 2.4.2 still apply and are also mentioned by Li, T. J. Walsh, and Littman (2006) in a slightly different formulation.

policies can induce abstract MDPs with different weighting functions.<sup>38</sup> This means that behaviour policies that change over time (e.g.  $\epsilon$ -greedy policies) may lead to changing (non-Markov) dynamics of the abstract decision problem. Li, T. J. Walsh, and Littman (2006) observe the effect of this as a “chattering” phenomenon which occurs when applying  $Q$ -learning in certain abstract MDPs, witnessing the fact that  $Q$ -learning applied to abstract MDPs does not converge in general. However, if the behaviour policy is assumed to be stationary, then  $Q$ -learning can be shown to converge to  $\hat{q}_*$  (S. Singh, Jaakkola, and Jordan 1994). But even if  $\hat{\pi}_{\text{Target}}$  converges to  $\hat{\pi}_*$  for some abstract MDP  $\hat{M}$ , there is no guarantee that this solution is also optimal in the concrete MDP, i.e. that  $\hat{\pi}_*(a \mid \phi(s)) \approx \pi_*(a \mid s)$  for all  $(s, a) \in \Psi$ .

While there are no guarantees that  $Q$ -learning under state abstraction works in general, Li, T. J. Walsh, and Littman (2006) do provide a more detailed analysis considering different families of abstraction functions based on the following properties:

- P1 *Model-irrelevance*. For all states  $s_1, s_2 \in S$ , abstract states  $\hat{s} \in \hat{S}$ , and actions  $a \in A$ ,  $\phi(s_1) = \phi(s_2)$  implies  $\hat{r}(s_1, a) = \hat{r}(s_2, a)$  and  $\sum_{x \in \phi^{-1}(\hat{s})} p(x \mid s_1, a) = \sum_{x \in \phi^{-1}(\hat{s})} p(x \mid s_2, a)$ .
- P2  *$q_\pi$ -irrelevance*. For all states  $s_1, s_2 \in S$ , policies  $\pi$ , and actions  $a \in A$ ,  $\phi(s_1) = \phi(s_2)$  implies  $q_\pi(s_1, a) = q_\pi(s_2, a)$ .
- P3  *$q_*$ -irrelevance*. For all states  $s_1, s_2 \in S$  and actions  $a \in A$ ,  $\phi(s_1) = \phi(s_2)$  implies  $q_*(s_1, a) = q_*(s_2, a)$ .
- P4  *$a_*$ -irrelevance*. For every abstract state  $\hat{s}$  there exists an action  $a_*$ , such that for all states  $s_1, s_2 \in S$ ,  $\phi(s_1) = \phi(s_2) = \hat{s}$  implies that  $q_*(s_1, a_*) = \max_a q_*(s_1, a) = \max_a q_*(s_2, a) = q_*(s_2, a_*)$ .
- P5  *$\pi_*$ -irrelevance*. For every abstract state  $\hat{s}$  there exists an action  $a_*$ , such that for all states  $s_1, s_2 \in S$ ,  $\phi(s_1) = \phi(s_2) = \hat{s}$  implies that  $q_*(s_1, a_*) = \max_a q_*(s_1, a)$  and  $q_*(s_2, a_*) = \max_a q_*(s_2, a)$ .

The results that Li, T. J. Walsh, and Littman (2006) obtained under consideration of these properties (relevant to  $Q$ -learning) can be summarised as follows: First, it can be shown that  $P1 \implies P2$ ,  $P2 \implies P3$ ,  $P3 \implies P4$ , and  $P4 \implies P5$ , but that they are not equivalent (e.g., there exists a mapping  $\phi$  exhibiting  $P2$  but not  $P1$ ). Further, mappings with different properties can be ordered by the coarseness of their resulting partitions. So, if  $\phi_1$  exhibits  $P1$  and  $\phi_2$  exhibits  $P2$  but not  $P1$ , then  $\mathcal{B}_{\phi_2} \succeq \mathcal{B}_{\phi_1}$ . This reflects the intuition that abstraction mappings with increasing coarseness will lose more and more of the mentioned properties.

<sup>38</sup>The dependence of the weighting function on the policy is made explicit e.g. in the convergence proof of S. Singh, Jaakkola, and Jordan (1994), who show the convergence of  $Q$ -learning (guided by a stationary policy) in the more general context of *soft state aggregation*.

At the bare minimum, to guarantee that  $\pi_*$  is at least representable by some abstract policy, P5 needs to be preserved. But in general, the abstract representation of  $\pi_*$  does not need to coincide with the optimal abstract policy  $\hat{\pi}_*$ . This is shown by providing a counterexample in which  $\hat{\pi}_*$  is suboptimal in the concrete MDP (Li, T. J. Walsh, and Littman 2006, figure 1).

If P4 is preserved however, it can be shown that  $\hat{\pi}_*$  for any induced abstract MDP (independent of  $w$ ) is optimal also for the concrete MDP, i.e. that  $\hat{\pi}_*(a | \phi(s)) \approx \pi_*(a | s)$  for all  $(s, a) \in \Psi$  (Li, T. J. Walsh, and Littman 2006, theorem 3). With respect to  $Q$ -learning, convergence to  $\hat{q}_*$  can be shown also for non-stationary behaviour policies, as long as P3 is preserved (Li, T. J. Walsh, and Littman 2006, theorem 4).<sup>39</sup> For coarser abstractions, the convergence of  $Q$ -learning is not guaranteed, but may still occur.

To summarise, asymptotic convergence of abstract  $Q$ -learning and the optimality of  $\hat{\pi}_*$  in the concrete MDP representation are generally not a given and depend on the preserved properties. Ideally, when using  $Q$ -learning, the coarseness of abstraction mappings should be chosen such that P3-P5 are preserved. However, if only P5 is preserved, it may still be possible to achieve convergence for  $Q$ -learning and optimality in the concrete case. The choice of properties to preserve needs to be balanced with the desired state space reduction, which is an interesting trade-off. The results of Li, T. J. Walsh, and Littman (2006) are mainly concerned with the asymptotic convergence of reinforcement learning algorithms using state abstraction and with their optimality in the concrete problem representation. But more can be said about the effect of state abstraction on learning efficiency, in particular about the influence of to the weighting function  $w$  (Li, T. J. Walsh, and Littman 2006 citing Roy 2006).

### 2.5.2 State-Action Pair Abstraction

As just discussed, the framework by Li, T. J. Walsh, and Littman (2006) provides useful tools for understanding state abstractions. However, it fails to capture some aspects that need to be treated in a more general abstraction framework for reinforcement learning. First, state abstraction does not modify the action space. In domains with large action spaces, such as the blocks world (see Chapter 4), it is essential that both state and action spaces are abstracted. Second, Li, T. J. Walsh, and Littman (2006) assume admissibility of all actions in every state. A possible extension to the general case of limited admissibility is mentioned as future work, with a reference to Ravindran and Barto (2003). Third, some MDPs exhibit symmetries that cannot be captured by state abstraction alone. Ravindran (2004, p. 5) provide a symmetric grid world problem as an example and show how different actions (such as moving north and moving east in the given grid world) lead to equivalent results under a state abstraction which exploits that particular grid worlds symmetry. This highlights the need for a more general *state-action*

<sup>39</sup>Also note that a similar result is available from Majeed and Hutter (2018), which treats  $Q$ -learning and the preservation of value functions in the context of history-based decision processes.

*pair abstraction*<sup>40</sup> function  $\phi : \Psi \rightarrow \hat{\Psi}$ , like formulated in the introduction of this section. To this end, Ravindran (2004) introduce the concept of *MDP Homomorphisms*:<sup>41</sup>

**Definition 2.5.2** (MDP homomorphism: Ravindran 2004, p. 18). Given are two MDPs  $M = (S, A, R, \Psi, p, \gamma)$  and  $\hat{M} = (\hat{S}, \hat{A}, \hat{R}, \hat{\Psi}, \hat{p}, \gamma)$ . Let  $h : \Psi \rightarrow \hat{\Psi}$  be a surjective mapping defined by the tuple  $(f : S \rightarrow \hat{S}, \{g_s : A_s \rightarrow \hat{A}_{f(s)} \mid s \in S\})$ , such that  $h((s, a)) = (f(s), g_s(a))$  for all  $(s, a) \in \Psi$ . We call  $h$  an *MDP homomorphism* if it holds that for all  $(s, a) \in \Psi$  and  $s' \in S$ :

$$\begin{aligned}\hat{p}(f(s') \mid f(s), g_s(a)) &= \sum_{x \in [s']_f} p(x \mid s, a) \\ \hat{r}(f(s), g_s(a)) &= r(s, a)\end{aligned}$$

In this definition,  $f$  acts as state abstraction function and  $g_s$  as action abstraction function for state  $s$ , mapping each admissible action  $a \in A_s$  to an abstract action  $\hat{a} \in A_{f(s)}$  admissible in the abstract state  $f(s)$ . Using this framework, online, off-policy reinforcement learning algorithms can be developed analogous to our discussion of state abstraction. But due to the introduction of abstract actions, we need to be careful when mapping abstract policies back to concrete ones. To this end, Ravindran (2004, p. 22) introduce *lifted policies*.

**Definition 2.5.3** (Lifted policy: Ravindran 2004, p. 22). Let  $h = (f, g_s \mid s \in S)$  define a homomorphism from  $M$  to  $\hat{M}$ . Let  $\hat{\pi} : \hat{\Psi} \rightarrow [0, 1]$  be a policy for  $\hat{M}$ . We define  $\pi_{\hat{\pi}}$  as  $\hat{\pi}$  *lifted to  $M$* , subject to:

$$\pi_{\hat{\pi}}(s, a) \doteq \frac{\hat{\pi}(f(s), g_s(a))}{|[a]_{g_s}|} \quad \forall (s, a) \in \Psi$$

In the presence of a homomorphism  $h = (f, g_s \mid s \in S)$  from  $M$  to  $\hat{M}$ , Ravindran (2004) shows the equivalence of optimal value functions for  $M$  and  $\hat{M}$ , i.e. that  $q_{\star}(s, a) = \hat{q}_{\star}(f(s), g_s(a))$  for all  $(s, a) \in \Psi$  (Theorem 1, p. 21). Further, it is established that the image of an optimal abstract policy is also optimal in the concrete MDP, i.e. that  $\pi_{\star} \approx \pi_{\hat{\pi}_{\star}}$  (Theorem 2, p. 23). In Theorem 6 (p. 32), a relation between MDP homomorphisms and the concept of *stochastic bisimulation* is established, which was also cited by Li, T. J. Walsh, and Littman (2006) in the context of state abstraction, as example of P1 (model-irrelevance).

Although the convergence of  $Q$ -learning in the context of MDP homomorphisms is suggested by this link between MDP homomorphisms and P1, we are not aware of such a proof, at least not in the case of non-stationary behaviour policies. If the behaviour

<sup>40</sup>This term was inspired by Goetschalckx (2009, pp. 51–56), who provide a discussion on the similar concept of *state-action pair aggregation* and provide a convergence proof of  $Q$ -learning when the behaviour policy is assumed to be stationary.

<sup>41</sup>SMDP Homomorphisms (Ravindran and Barto 2003), a more general version based on Semi-Markov decision processes, is also cited by Li, T. J. Walsh, and Littman (2006).

## 2. PRELIMINARIES AND THEORETICAL BACKGROUND

---

policy is assumed to be stationary though, the convergence of  $Q$ -learning is guaranteed for any  $h$ , even if it is not a homomorphism (Goetschalckx 2009, pp. 53–55), just like it was the case for state abstractions.

# ASP-Based State-Action Pair Abstractions

In this chapter we present our approach to modelling RMDP abstraction functions in ASP. Our approach is based on the *CARCASS framework* (van Otterlo 2008, pp. 252–270), which uses *Prolog* to model state-action pair abstraction functions and *SLDNF-Resolution* as the central computational mechanism. Moving away from Prolog, we provide a method to encode CARCASS based abstractions in ASP, include some useful extensions, and show how the resulting ASP encoding can be used in an online learning setting to compute abstractions.

The chapter is structured as follows. We first give a brief introduction to Prolog, focusing on the aspects that are relevant to the CARCASS framework. We also compare ASP to Prolog and give our motivation for using ASP over Prolog. Next, we introduce the CARCASS framework as defined by van Otterlo (2008). We consider CARCASS-based  $Q$ -learning and relate its convergence properties to the theory of abstraction as discussed in Section 2.5. We then present the core contribution of this thesis, which is the implementation of the CARCASS framework in ASP. We show how ASP can be used to implement the CARCASS semantics and how this implementation can be used in combination with  $Q$ -learning in order to solve large-scale RMDPs. To this end, we also provide a worked-out example of a translated CARCASS.

## 3.1 ASP and Prolog

Recall our discussion of answer-set programming (ASP) in Section 2.2. ASP takes a model-based approach to logic programming, in which every answer set is considered to be a solution (i.e. answer) to the problem encoded by a given program. Prolog takes a proof-based approach instead. In addition to a *normal program*  $P$ ,<sup>1</sup> Prolog expects a

*normal goal* of the form  $g \doteq (\leftarrow b_1, \dots, b_n.)$  as input, with  $b_1, \dots, b_n$  being naf-literals (excluding strong negation) for  $n \geq 0$  (Nienhuys-Cheng and Wolf 1997, p. 131). The goal of Prolog's decision procedure is to decide whether the body of  $g$  is true with respect to all intended models<sup>2</sup> of  $P$  by searching for an *SLDNF-refutation* for  $P \cup \{g\}$  (Nienhuys-Cheng and Wolf 1997, p. 141; van Otterlo 2008, pp. 179–187). Given an SLDNF-refutation, a substitution from the variables in  $g$  can be formed, which constitutes a *computed answer* to the original query (Nienhuys-Cheng and Wolf 1997, p. 141). Multiple SLDNF-refutations may exist and each one induces its own computed answer. The search for SLDNF-refutations in Prolog is organised depth-first and follows the order of rules and the literals within rule bodies (Nienhuys-Cheng and Wolf 1997, pp. 155–157). To this end, programs are interpreted not as sets of rules but as ordered lists of rules. Further, a *cut operator* is introduced, which allows to directly modify the proof search procedure by pruning parts of the search space (Nienhuys-Cheng and Wolf 1997, pp. 157–159). The way in which the proof search is organised and the influence a programmer can exert on it make Prolog a decision procedure which is unsound, incomplete and not fully declarative. Sources for unsoundness are *floundering* (Nienhuys-Cheng and Wolf 1997, pp. 141, 156) and possible interactions between the cut operator and default negation (Nienhuys-Cheng and Wolf 1997, pp. 158–159). Sources for incompleteness are the cut operator and the imposed order, which may cause an infinite loop to occur during in the proof search before producing all (or any) computed answers (Nienhuys-Cheng and Wolf 1997, pp. 156–157). This lays the burden on the programmer to understand not only the language but also the underlying decision procedure.

As mentioned by Gebser, Kaminski, Kaufmann, and Schaub (2012, pp. 1–4), the motivation to create a fully declarative modelling language was one of the key ambitions behind the development of ASP. They also describe several key differences between ASP and Prolog. First, there is the difference in the reasoning approach (model-based and proof-based) as just discussed. Second, the computational procedures used in computing "answers" for each approach are quite different. While major evaluation approaches for ASP employ techniques from propositional satisfiability testing for computing models and database techniques for computing groundings, Prolog relies on proof-search techniques such as unification and exposes its computational mechanisms to the modeller. ASP on the other hand offers a fully declarative formalism, the usage of which requires no knowledge about the inner workings of the grounding and solving procedures. This strict separation of logic from control in the sense of Kowalski (1979) makes ASP a better fit to the original motivation of logic programming (Gebser, Kaminski, Kaufmann, and Schaub 2012, pp. 1). A third difference is the expressiveness of the formalisms. As discussed in Section 2.2.4, the complexity of ASP is constrained while it still covers a large range of

<sup>1</sup>*Normal programs* are a syntactic restriction of *programs* as introduced in Definition 2.2.4. They consist of *normal rules* of the form  $(h \leftarrow b_1, \dots, b_n.)$  with  $h$  being an atom and  $b_1, \dots, b_n$  being naf-literals (excluding strong negation) for  $n \geq 0$  (Nienhuys-Cheng and Wolf 1997, p. 130). In the sense of Definition 2.2.4, these are rules without disjunction in the heads (i.e.  $m=1$ ) and without strong negation.

<sup>2</sup>The intended models of a program are characterised by the models of the program's *completion* (Nienhuys-Cheng and Wolf 1997, Section 8.5). Compare with the characterisation of a program's answer sets by its *completion* and *loop formulas* (Gebser, Kaminski, Kaufmann, and Schaub 2012, Section 5.1).

practical problems. Prolog on the other hand is computationally complete due to the unrestricted use of function symbols in the language (Dantsin et al. 2001, Section 4.1). The differences between the formalisms leads to different modelling paradigms in each (Marek and Truszczyński 1998).

The declarative aspect of ASP makes it well-suited to model abstraction functions for reinforcement learning. In particular, we believe the elaboration tolerance offered by ASP (Gebser, Kaminski, Kaufmann, and Schaub 2012, pp. 2, 9, 45 citing McCarthy 1998) to be very useful in a reinforcement learning setting, enabling elaborations on the abstraction function and on relevant background knowledge over time. However, due to the limited expressive power of ASP compared to Prolog, there exist abstraction functions that are expressible in Prolog but not in ASP.

## 3.2 The CARCASS Framework

The acronym *CARCASS* stands for *Compact Abstraction using Relational Conjunctions for Aggregation of State-action Spaces* (van Otterlo 2008, p. 252). It was conceived as a technique for the implementation of state-action pair abstractions using logic programming. The framework was published in several works (van Otterlo 2003; van Otterlo 2004; van Otterlo 2008, pp. 252–270). For our discussion, we follow van Otterlo (2008, pp. 252–270), the most comprehensive treatment.

### 3.2.1 Syntax and Semantics

In the following, we present the central definitions of the CARCASS framework, starting with abstract states and actions.

**Definition 3.2.1** (Abstract states and actions: van Otterlo 2008, p. 253). Let  $\Sigma = (Func, Pred)$  be a PL1-Signature and  $Var$  a set of variables.

- An *abstract state*  $\hat{s}$  is of the form  $\hat{s} \doteq l_1, \dots, l_k$ , such that  $k \geq 1$  and every  $l_i$  ( $1 \leq i \leq k$ ) is a naf-literal, excluding strong negation, over  $\Sigma$  and  $Var$ .<sup>3</sup>
- An *abstract action*  $\hat{a}$  is a single predicate atom  $\hat{a} \doteq \mathbf{p}(\bar{t})$  over  $\Sigma$  and  $Var$ , where  $\bar{t}$  is meant as a placeholder for various terms.

The syntax of the CARCASS is defined as follows.

**Definition 3.2.2** (CARCASS syntax: van Otterlo 2008, p. 253). Let  $\Sigma = (Func, Pred)$  be a PL1-Signature and  $Var$  a set of variables. A *CARCASS*  $\hat{C} = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$

<sup>3</sup>This definition was originally given in a normal logic programming setting, where strong negation is not part of the language. Therefore we need to exclude it. But we see no harm in including strongly negated predicate atoms in abstract states when encoded in ASP. Also, built-in atoms (e.g. equality) can be included to some extent, as was done in examples by van Otterlo (2008, p. 253) and in Section 3.4 but one needs to be careful when translating them from Prolog to ASP.

is a finite, ordered list with  $n \geq 1$  elements, such that  $\hat{s}_1, \dots, \hat{s}_n$  are abstract states and  $\hat{A}_{\hat{s}_1}, \dots, \hat{A}_{\hat{s}_n}$  are sets of abstract actions over  $\Sigma$  and  $Var$ . Every  $\hat{A}_{\hat{s}_i} = \{\hat{a}_{i,1}, \dots, \hat{a}_{i,m_i}\}$  denotes the set of *admissible* abstract actions over the abstract state  $\hat{s}_i$ . For every  $1 \leq i \leq n$  and  $1 \leq j \leq m_i$ , the variables occurring in  $\hat{a}_{i,j}$  must also occur in  $\hat{s}_i$ . For ease of notation, we also define  $\hat{S} \doteq \{ \hat{s}_i \mid 1 \leq i \leq n \}$  and  $\hat{\Psi} \doteq \{ (\hat{s}_i, \hat{a}_{i,j}) \mid 1 \leq i \leq n, 1 \leq j \leq m_i \}$ .

A CARCASS is designed with an RMDP  $(\Sigma, S, A, R, \Psi, p, \gamma)$  of matching signature in mind.<sup>4</sup> The intuition behind the CARCASS is as follows. Let  $(s, a) \in \Psi$  be a state-action pair of the RMDP. To find a matching abstract pair, the CARCASS list can be traversed in order until some  $\hat{s}_i$  is found which *covers*  $s$  (to be defined shortly). Then, an abstract action  $\hat{a}_{i,j} \in \hat{A}_{\hat{s}_i}$  needs to be found such that  $(\hat{s}_i, \hat{a}_{i,j})$  *covers*  $(s, a)$ . This intuition describes two aspects of the CARCASS: The covering relation and the decision list characteristic. First, the covering relation is defined as follows.

**Definition 3.2.3** (CARCASS covering relation: van Otterlo 2008, p. 254). Let  $P \vdash_{SLDNF} g$  denote the existence of an *SLDNF-refutation* of  $P \cup \{\leftarrow g\}$  (Nienhuys-Cheng and Wolf 1997, p. 141; van Otterlo 2008, pp. 179–187), where  $P$  is a normal program and  $\leftarrow g$  is a normal goal clause. Let  $(\Sigma, S, A, R, \Psi, p, \gamma)$  be an RMDP and let  $\langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$  be a CARCASS. We denote by  $S_{\hat{s}_i}$  the *set of states covered by  $\hat{s}_i$*  and by  $\Psi_{\hat{s}_i, \hat{a}_{i,j}}$  the *set of state-action pairs covered by  $(\hat{s}_i, \hat{a}_{i,j})$* , defined as follows:<sup>5</sup>

$$S_{\hat{s}_i} \doteq \{ s \in S \mid s \vdash_{SLDNF} \hat{s}_i \}$$

$$\Psi_{\hat{s}_i, \hat{a}_{i,j}} \doteq \{ (s, a) \in \Psi \mid s \vdash_{SLDNF} \hat{s}_i \theta \text{ and } a = \hat{a}_{i,j} \theta \}$$

In light of this definition, a state  $s \in S$  is seen as a program consisting of facts and an abstract state  $\hat{s}_i = l_1, \dots, l_k$  is seen as a normal goal  $g \doteq (\leftarrow l_1, \dots, l_k)$ . In words, a state is covered by an abstract state  $\hat{s}$  if there exists an *SLDNF-refutation* of  $s \cup \{g\}$ . The state-action pairs covered by  $(\hat{s}_i, \hat{a}_{i,j})$  are found by enumerating the *computed answers* (Nienhuys-Cheng and Wolf 1997, p. 141) for  $s \cup \{g\}$  and applying them to  $\hat{a}_{i,j}$ .

Second, we formalise the decision list characteristic. In combination with the covering relation, the following definition will provide us with the full CARCASS semantics.

**Definition 3.2.4** (CARCASS semantics: van Otterlo 2008, p. 254). Let  $(\Sigma, S, A, R, \Psi, p, \gamma)$  be an RMDP and let  $\hat{C} = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$  be a CARCASS. For every  $\hat{s}_i \in \hat{S}$

<sup>4</sup>Generally, the signatures of the RMDP and a related CARCASS need to match. But van Otterlo (2008, pp. 252–253) allows for the possibility to include *background predicates* in the CARCASS’s signature, which do not occur directly in the RMDP but are meant to be derived using additional background knowledge.

<sup>5</sup>The definition in (van Otterlo 2008, p. 254) may also be read as  $\Psi_{\hat{s}_i, \hat{a}_{i,j}} \doteq \{ (s, a) \in S \times A \mid s \vdash_{SLDNF} \hat{s}_i \theta \text{ and } a \equiv \hat{a}_{i,j} \theta \}$ . So, an abstract state-action pair may cover a pair  $(s, a)$  which is not admissible, i.e.  $(s, a) \notin \Psi$ .

This possibility is mentioned by van Otterlo (2008, p. 253), who leaves it up to the designer to ensure that  $\Psi_{\hat{s}_i, \hat{a}_{i,j}} \subseteq \Psi$ .

and  $(\hat{s}_i, \hat{a}_{i,j}) \in \hat{\Psi}$ , we define the CARCASS's *semantics* by:

$$\begin{aligned} \llbracket \hat{s}_i \rrbracket_{\hat{C}} &\doteq \{ s \in S_{\hat{s}_i} \mid s \notin S_{\hat{s}_k} \text{ for all } 1 \leq k < i \} \\ \llbracket \hat{s}_i, \hat{a}_{i,j} \rrbracket_{\hat{C}} &\doteq \{ (s, a) \in \Psi_{\hat{s}_i, \hat{a}_{i,j}} \mid s \notin S_{\hat{s}_k} \text{ for all } 1 \leq k < i \} \\ \llbracket \hat{s}_i, \hat{a}_{i,j} \rrbracket_{\hat{C}}^s &\doteq \{ a \mid (s, a) \in \llbracket \hat{s}_i, \hat{a}_{i,j} \rrbracket_{\hat{C}} \} \end{aligned}$$

Intuitively, the semantics can be understood as follows.  $\llbracket \hat{s}_i \rrbracket_{\hat{C}}$  gives us the set of states  $s \in S$  that are attributed to the abstract state  $\hat{s}_i$ . While the same state can be covered by multiple abstract states, the order restriction ensures that the semantics induce a partitioning  $\mathcal{B} = \{ \llbracket \hat{s}_1 \rrbracket_{\hat{C}}, \llbracket \hat{s}_2 \rrbracket_{\hat{C}}, \dots, \llbracket \hat{s}_n \rrbracket_{\hat{C}}, \}$  over the state space.  $\llbracket \hat{s}_i, \hat{a}_{i,j} \rrbracket_{\hat{C}}$  gives us the set of state-action pairs  $(s, a) \in \Psi$  that are attributed to the abstract state-action pair  $(\hat{s}_i, \hat{a}_{i,j})$ , and  $\llbracket \hat{s}_i, \hat{a}_{i,j} \rrbracket_{\hat{C}}^s$  gives us the set of actions  $a \in A_s$  that belong to the abstract action  $\hat{a}_{i,j}$ .

### 3.2.2 CARCASS and Q-Learning

In the following we look at an application of the CARCASS framework to  $Q$ -learning. The procedure (Algorithm 3.1, van Otterlo 2008, p. 258) is analogous to  $Q$ -learning in the concrete MDP (Algorithm 2.1) and to the considerations from Section 2.5 on online learning under abstraction. Besides the CARCASS semantics, the algorithm makes use of abstract policies  $\hat{\pi} : \hat{\Psi} \rightarrow [0, 1]$ , with  $\sum_{\hat{a} \in \hat{A}_{\hat{s}}} \hat{\pi}(\hat{a} \mid \hat{s}) = 1$  for all  $\hat{s} \in \hat{S}$ , and their related abstract value functions  $\hat{q}_{\hat{\pi}} : \hat{\Psi} \rightarrow \mathbb{R}$ . A greedy policy  $\hat{\pi}_{\hat{q}}$  (resp.  $\epsilon$ -greedy policy  $\hat{\pi}_{\hat{q}, \epsilon}$ ) is defined in analogy to Definition 2.4.1.

We now consider Algorithm 3.1. The two loops describe the interaction cycle between the agent and the environment just as before, but both the  $Q$ -learning update and the  $\epsilon$ -greedy policy now operate on the abstract states and actions. At any given time point  $t$ , a reward  $r_t$  and a state  $s_t$  are observed. Then, the abstract state  $\hat{s}_t$  associated with  $s_t$  is computed, along with the set of admissible abstract actions  $\hat{A}_{\hat{s}_t}$  and their associated concrete actions  $\{ \llbracket \hat{s}_{t+1}, \hat{a} \rrbracket_{\hat{C}}^{s_{t+1}} \mid \hat{a} \in \hat{A}_{\hat{s}_{t+1}} \}$ . Then, an abstract action  $\hat{a}_t$  is selected according to the abstract behaviour policy. With  $s_t$ ,  $\hat{s}_t$  and  $\hat{a}_t$  available, a method is needed to select a concrete action. This is done by sampling  $a_t$  from a random uniform probability distribution over  $\llbracket \hat{s}_t, \hat{a}_t \rrbracket_{\hat{C}}^s$ , i.e.:

$$\pi_{\hat{C}, \hat{s}, \hat{a}}(a \mid s) \doteq \frac{1}{|\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^s|}$$

The execution of  $a_t$ , marks the transition to the next time point. After observing  $r_{t+1}$ ,  $s_{t+1}$  and computing all relevant abstract components for  $t+1$ , the abstract value function is updated.

To investigate the convergence of Algorithm 3.1, we can apply the theories discussed in Section 2.4.2 and Section 2.5. For some RMDP  $(\Sigma, S, A, R, \Psi, p, \gamma)$ , state abstraction is implemented by any CARCASS  $\hat{C} = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$  in which all abstract actions are ground. Then, a *state abstraction function*  $\phi_{\hat{C}} : S \rightarrow \hat{S}$  can be defined as

---

**Algorithm 3.1:**  $Q$ -learning for CARCASSs (van Otterlo 2008, p. 258) using  $\epsilon$ -greedy exploration policies (Sutton and Barto 2018, p. 131) with time limit

---

**Input:** Episodic MDP  $M = (S, A, R, \Psi, p, \gamma, \text{init}, S_{\text{term}})$   
**Input:** CARCASS  $\hat{C} = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$   
**Parameter:** Exploration factor  $\epsilon > 0$   
**Parameter:** Fixed learning rate  $\alpha \in (0, 1]$   
**Parameter:** Initial values  $\hat{q} : \hat{\Psi} \rightarrow \mathbb{R}$ ,  
 with  $\hat{q}(\hat{s}, \hat{a}) = 0$  for all  $\hat{a} \in \hat{A}_{\hat{s}}$  if  $\hat{s} \in \hat{S}_{\text{term}}$   
**Parameter:** Time limit  $\tau \in \mathbb{N}$   
**Output:**  $\hat{q} \simeq \hat{q}_*$ ,  $\hat{\pi}_{\hat{q}} \simeq \hat{\pi}_*$  (in case of convergence)

- 1 **foreach** episode **do**
- 2      $s_0 \sim \text{init}(\cdot)$
- 3      $(\hat{s}_0, \hat{A}_{\hat{s}_0}, \{ \llbracket \hat{s}_0, \hat{a} \rrbracket_{\hat{C}}^{s_0} \mid \hat{a} \in \hat{A}_{\hat{s}_0} \}) \leftarrow \text{Compute s.t. } s_0 \in \llbracket \hat{s}_0 \rrbracket_{\hat{C}}$ .
- 4     **repeat** for every time point  $t = 0, 1, 2, \dots$
- 5          $\hat{a}_t \sim \hat{\pi}_{\hat{q}, \epsilon}(\cdot \mid \hat{s}_t)$
- 6          $a_t \sim \pi_{\hat{C}, \hat{s}_t, \hat{a}_t}(\cdot \mid s_t)$
- 7          $r_{t+1}, s_{t+1} \sim p(\cdot \mid s_t, a_t)$
- 8          $(\hat{s}_{t+1}, \hat{A}_{\hat{s}_{t+1}}, \{ \llbracket \hat{s}_{t+1}, \hat{a} \rrbracket_{\hat{C}}^{s_{t+1}} \mid \hat{a} \in \hat{A}_{\hat{s}_{t+1}} \}) \leftarrow \text{Compute s.t. } s_{t+1} \in \llbracket \hat{s}_{t+1} \rrbracket_{\hat{C}}$ .
- 9          $\hat{q}(\hat{s}_t, \hat{a}_t) \leftarrow \hat{q}(\hat{s}_t, \hat{a}_t) + \alpha [r_{t+1} + \gamma \max_{\hat{a}' \in \hat{A}_{\hat{s}_{t+1}}} \{ \hat{q}(\hat{s}_{t+1}, \hat{a}') \} - \hat{q}(\hat{s}_t, \hat{a}_t)]$
- 10     **until**  $s_{t+1} \in S_{\text{term}}$  or  $t \geq \tau$
- 11 **end**

---

$\phi_{\hat{C}}(s) \doteq \hat{s}$  iff  $s \in \llbracket \hat{s} \rrbracket_{\hat{C}}$ . Due to the order of abstract states and the resulting state partition, it is ensured that every  $s \in S$  is associated with at most one abstract state  $\hat{s} \in \hat{S}$ . However, it is technically possible that some state is not covered by any abstract state. It is up to the designer of the CARCASS to ensure that this does not happen.

We can use CARCASSs also to implement MDP homomorphisms. Let  $\hat{C}$  be a CARCASS of an RMDP  $(\Sigma, S, A, R, \Psi, p, \gamma)$ . A surjective mapping  $h_{\hat{C}} = \Psi \rightarrow \hat{\Psi}$  from the concrete state-action space to the abstract state-action space is induced by  $\hat{C}$  as follows:  $h_{\hat{C}}((s, a)) \doteq (f_{\hat{C}}(s), g_{\hat{C}, s}(a))$  with  $f_{\hat{C}}(s) \doteq \hat{s}$  iff  $s \in \llbracket \hat{s} \rrbracket_{\hat{C}}$  and  $g_{\hat{C}, s}(a) \doteq \hat{a}$  iff  $a \in \llbracket f_{\hat{C}}(s), \hat{a} \rrbracket_{\hat{C}}^s$ . Analogous to state abstractions, we are guaranteed that  $f_{\hat{C}}(s)$  is a function due to the order-induced partitioning of the state space. But again, we need to ensure that every  $s \in S$  is covered by some abstract state and that every  $(s, a) \in \Psi$  is covered by some abstract state-action pair. Furthermore,  $\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^s$  cannot be empty for any  $\hat{s}, \hat{a}$  and  $s$ . Another issue might occur for  $g_{\hat{C}, s}$ . Using an example, van Otterlo (2008, p. 256) shows that different abstract actions may have overlapping coverage relations, so there might exist  $\hat{a} \neq \hat{a}' \in \hat{A}_{\hat{s}}$  such that  $a \in \llbracket f(s), \hat{a} \rrbracket_{\hat{C}}^s$  and  $a \in \llbracket f(s), \hat{a}' \rrbracket_{\hat{C}}^s$ . In such a case, the application of the homomorphism framework is not immediate. However, if a designer ensures that every  $(s, a) \in \Psi$  is associated with exactly one abstract state-action pair  $(\hat{s}, \hat{a}) \in \hat{\Psi}$ , then we can use the MDP homomorphism framework to analyse Algorithm 3.1

as, for this special case, the sampling process of  $a$  is identical to the one in Definition 2.5.3.

### 3.3 Translation to ASP

In this section, we show how the CARCASS can be modelled using ASP and discuss online learning as an application. Finally, we consider an example CARCASS to illustrate the process.

#### 3.3.1 Translation

Let  $\hat{C} = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$  be a CARCASS as defined above, in correspondence to the RMDP  $M = (\Sigma, S, A, R, \Psi, p, \gamma)$  with a shared signature  $\Sigma = (F, P_S \cup P_A)$ . We assume that none of the function- and predicate names introduced in the translation below occur in  $\Sigma$ .

In the following, we define several answer-set programs which, in various combinations, can model the different aspects of the CARCASS framework.

- $P_s$  encodes a concrete state  $s \in S$  as facts (Definition 3.3.2).
- $P_{A_s}$  encodes the set of admissible concrete actions  $A_s$  for a state  $s \in S$  as facts (Definition 3.3.2).
- $P_{\hat{s}_i}$  encodes the covering relation for a single abstract state  $\hat{s}_i \in \hat{S}$  (Definition 3.3.3).
- $P_{\hat{S}}$  collects all the programs  $P_{\hat{s}_i}$  for  $\hat{s}_i \in \hat{S}$  (Definition 3.3.4). It also contains a choice rule, such that there is a one-to-one correspondence between the answer sets of  $P_s \cup P_{\hat{S}}$  and the abstract states that cover  $s \in S$ .
- $P_{<}$  is a constraint encoding the decision list characteristic, such that  $P_s \cup P_{\hat{S}} \cup P_{<}$  has only a single answer set, corresponding to the covering state with the smallest index (Definition 3.3.4).
- $P_{\hat{s}_i, \hat{a}_{i,j}}$  (together with  $P_{\hat{S}}$ ) encode the covering relation for a single state-action pair  $\hat{s}_i, \hat{a}_{i,j} \in \hat{\Psi}$  (Definition 3.3.5).
- $P_{\hat{\Psi}} \doteq P_{\hat{S}} \cup \bigcup_{i,j} P_{\hat{s}_i, \hat{a}_{i,j}}$  encodes the complete covering relation (Definition 3.3.6). The answer sets of  $P_s \cup P_{A_s} \cup P_{\hat{\Psi}}$  correspond one-to-one with the abstract states that cover  $s \in S$ . Each answer set contains a listing of the admissible abstract actions and their covered admissible concrete actions for that answer set's corresponding abstract state.
- $P_{\hat{\Psi}} \cup P_{<}$  encodes the full CARCASS semantics, with  $P_s \cup P_{A_s} \cup P_{\hat{\Psi}} \cup P_{<}$  yielding a single answer set corresponding to the abstract state attributed to  $s$ .

**Labelling Function.** Recall that CARCASS-based abstract states and actions are themselves logical expressions, possibly containing variables. In order to reason about them as domain objects, we also need to represent them as terms. To this end, we assume a labelling function as follows.

**Definition 3.3.1** (Labelling function). A *labelling function* is a one-to-one mapping  $\lambda : \hat{S} \cup \hat{\Psi} \mapsto \mathbb{S}$  from abstract states and state-action pairs to the set of strings (i.e. sequences of characters) that can be used in the raw input format for answer-set solvers.

The labels are an opportunity to choose descriptive names or phrases of the labelled expressions, but they can also be chosen arbitrarily.

**Concrete States and Actions.** We define proper logic program representations for states and actions in  $M$  as follows:

**Definition 3.3.2** (ASP translation: States and actions). Let  $s \in S$  be a state and let  $A_s$  be the admissible actions in  $s$ . Recall that  $s$  is represented as a finite set of ground atoms and that every action  $a \in A_s$  is represented as a single ground atom. With this in mind, we formulate the following programs, where  $\bar{t}$  is meant as a placeholder for various terms.

$$\begin{aligned} P_s &\doteq \{\mathbf{p}(\bar{t}) \mid \mathbf{p}(\bar{t}) \in s\} \\ P_{A_s} &\doteq \{\mathbf{p}(\bar{t}) \mid \mathbf{p}(\bar{t}) \in A_s\} \end{aligned}$$

A single state-action pair  $(s, a) \in \Psi$ , with  $a = \mathbf{p}(\bar{t})$ , is represented by the program  $P_s \cup \{\mathbf{p}(\bar{t})\}$ .

Intuitively, an interpretation satisfying the facts in  $P_s \cup P_{A_s}$  characterises the state  $s$  and its admissible actions.

**Abstract states.** We encode the CARCASS covering relation (Definition 3.2.3) for a single abstract state as follows:

**Definition 3.3.3** (ASP translation: Covering relation for an abstract state). Let  $\hat{s}_i = l_1, \dots, l_k \in \hat{S}$  be an abstract state, let  $V_1, \dots, V_m$  denote all the variables occurring in  $\hat{s}_i$  and let  $\lambda : \hat{S} \cup \hat{\Psi} \mapsto \mathbb{S}$  be a labelling function. Then,  $\hat{s}_i$  is encoded as the program.

$$P_{\hat{s}_i} \doteq \left\{ \begin{array}{l} \mathbf{cState}(\lambda(\hat{s}_i), i). \\ \mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1, \dots, V_m)) \leftarrow l_1, \dots, l_k. \end{array} \right\}$$

The formulation makes use of two new predicates.

- **cState/2** denotes the existence of an abstract state  $\hat{s}_i$ . Its first argument identifies the abstract state by its label and its second argument marks its position (or index) in the decision list.

- **cStateCovers**/2 denotes the existence of a covering relation between  $\hat{s}_i$  and some state  $s$ . The first argument is again the label of the abstract state in question. The second argument is a tuple of terms, representing a computed answer for  $P_s \cup \{\leftarrow l_1, \dots, l_k\}$  in the context of SLDNF-Resolution.

The first rule is a fact defining **cState**/2 as just described. The head of the second rule defines **cStateCovers**/2, its body consists of the literals in  $\hat{s}_i$ . The ground instances of this rule reflect all of the possible ground substitutions for the variables in  $\hat{s}_i$ . If, for some ground instance, the body is satisfied by an interpretation, the rule concludes that a covering relation exists for  $\hat{s}_i$ .

**Decision List.** Next, we translate the decision list characteristic. This can be done by following the *guess and check* methodology (e.g. Eiter, Ianni, and Krennwallner 2009, p. 42), where a program consists of two parts:<sup>6</sup> The *guessing* part produces solution candidates, for example by using a choice rule. The *checking* part usually consists of constraints which eliminate solution candidates, such that only the intended answer sets remain. For our application, we aim to produce one solution candidate for each covering abstract state. Then, a constraint eliminates all solution candidates except for the one corresponding to the minimal covering state with respect to the decision list ordering. We formalise our considerations in the following two programs.

**Definition 3.3.4** (ASP translation: decision list). Let  $\hat{C} = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$  be a CARCASS. Its decision list characteristic is encoded by the union  $P_{\hat{S}} \cup P_{<}$  of the following two programs.

$$P_{\hat{S}} \doteq \bigcup_{1 \leq i \leq n} P_{\hat{s}_i} \cup \left\{ 1 = \{ \mathbf{cStateChoice}(R) : \mathbf{cStateCovers}(R, \_) \}. \right\}$$

$$P_{<} \doteq \left\{ \leftarrow \mathbf{cStateChoice}(R1), \mathbf{cStateCovers}(R2, \_), \right. \\ \left. \mathbf{cState}(R1, N1), \mathbf{cState}(R2, N2), N2 < N1. \right\}$$

The formulation introduces another unary predicate:

- **cStateChoice**/1 denotes some abstract state (by its label in the argument) as "chosen" for a given interpretation.

The program  $P_{\hat{S}}$  contains the definitions of the covering relations for all abstract states. It also contains a choice rule, in which the labels of all covering abstract states are made available for the choice, but only one can be chosen per answer set. This is the *guessing* part: Given a state  $s \in S$ ,  $\mathcal{AS}(P_s \cup P_{\hat{S}})$  contains one answer set for every abstract state  $\hat{s} \in \hat{S}$  covering  $s$ .

<sup>6</sup>This methodology is also called *generate and test* e.g. by Gebser, Kaminski, Kaufmann, and Schaub 2012, p. 39).

The program  $P_{<}$  consists of a constraint eliminating all interpretations in which the index of the chosen covering abstract state is not minimal. This is the *checking* part of the program. Given a state  $s \in S$ ,  $\mathcal{AS}(P_s \cup P_{\hat{S}} \cup P_{<})$  contains at most one answer set, corresponding to the covering abstract state  $\hat{s} \in \hat{S}$  with the lowest index in the decision list.

**Abstract state-action pairs.** The covering relation for a single abstract state-action pair can be encoded as follows:

**Definition 3.3.5** (ASP translation: Covering relation for an abstract state-action pair). Let  $(\hat{s}_i, \hat{a}_{i,j}) \in \hat{\Psi}$ , with  $\hat{a}_{i,j} = \mathbf{p}(\bar{t})$ . Further let  $V_1, \dots, V_l$  denote all the variables in  $\hat{s}_i$ . Then,  $(\hat{s}_i, \hat{a}_{i,j})$  is encoded as the following program, containing a newly introduced function name  $\mathbf{p}/n$  with the same arity as the predicate name  $\mathbf{p}/n$ . The following program contains a single rule, defining the covering relation for an abstract state-action pair:

$$P_{\hat{s}_i, \hat{a}_{i,j}} \doteq \left\{ \begin{array}{l} \mathbf{cActionCovers}(\lambda(\hat{s}_i, \hat{a}_{i,j}), \mathbf{p}(\bar{t})) \\ \leftarrow \mathbf{cStateChoice}(\lambda(\hat{s}_i)), \\ \mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1, \dots, V_l)), \\ \mathbf{p}(\bar{t}). \end{array} \right\}$$

The existence of a covering relation for  $(\hat{s}_i, \hat{a}_{i,j})$  is denoted by a new predicate as follows:

- **cActionCovers**/2 has as arguments the label  $\lambda(\hat{s}_i, \hat{a}_{i,j})$  for the abstract state-action pair and a covered action  $\mathbf{p}(\bar{t})$  represented as a functional term. So, in our encoding, an action  $\hat{a}_{i,j}$  can occur both as an atom  $\mathbf{p}(\bar{t})$  and as a functional term  $\mathbf{p}(\bar{t})$ .

**Complete covering relation.** Collecting the previously defined programs  $P_{\hat{S}}$  and  $P_{\hat{s}_i, \hat{a}_{i,j}}$  for  $\hat{s}_i, \hat{a}_{i,j} \in \hat{\Psi}$ , the complete covering relation can be defined as follows:

**Definition 3.3.6** (ASP translation: Complete CARCASS covering relation). The full program defining the covering relation for all abstract states and state-action pairs is constructed as follows:

$$P_{\hat{\Psi}} \doteq P_{\hat{S}} \cup \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq m_i} P_{\hat{s}_i, \hat{a}_{i,j}}$$

**Correctness.** We now consider how our translation relates back to the original definition based on SLDNF resolution. In the following we denote by  $\models$  the ground ASP satisfaction relation (Definition 2.2.7).

**Proposition 3.3.1.** Let  $P = P_s \cup P_{A_s} \cup P_{\hat{\Psi}}$ ,  $I \in \mathcal{AS}(P)$ ,  $\hat{s}_i = l_1, \dots, l_k \in \hat{S}$  with variables  $V_1, \dots, V_m$ ,  $s \in S$ , and  $\lambda : \hat{S} \cup \hat{\Psi} \mapsto \mathbb{S}$ . Further let  $\theta$  be a substitution such that  $\hat{s}_i\theta$  is a ground instance of  $\hat{s}_i$ . Then,  $s \vdash_{\text{SLDNF}} \hat{s}_i\theta$  iff  $\mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1\theta, \dots, V_m\theta)) \in I$ .

**Proof ( $\Rightarrow$ ):** Assume that  $s \vdash_{\text{SLDNF}} \hat{s}_i\theta$ . The SLDNF-refutation of  $s \cup \{\leftarrow \hat{s}_i\}$  is a sequence of derivation steps with goal clauses:  $(\leftarrow l_1\theta, \dots, l_{k-1}\theta, l_k\theta)$ ,  $(\leftarrow l_1\theta, \dots, l_{k-1}\theta)$ ,  $\dots$ ,  $(\leftarrow l_1\theta)$ ,  $\square$ . At every derivation step, a literal  $l_j\theta$  is deleted from the body (assuming an arbitrary order without loss of generality). If  $l_j\theta = \mathbf{p}(\bar{t})$  is positive, then  $\leftarrow l_1\theta, \dots, l_{j-1}\theta$  is a binary resolvent of  $\leftarrow l_1\theta, \dots, l_{j-1}\theta, \mathbf{p}(\bar{t})$  and of  $\mathbf{p}(\bar{t}) \in s$ . It follows, since  $\mathbf{p}(\bar{t})$  is a fact in  $P_s$ , that  $I \models \mathbf{p}(\bar{t}) = l_j\theta$ . If  $l_j\theta = \mathbf{not} \mathbf{p}(\bar{t})$  is negative, then there exists a finitely failed subsidiary derivation tree with  $\leftarrow \mathbf{p}(\bar{t})$  as root and no children. Thus,  $\mathbf{p}(\bar{t}) \notin s$  and, as there are no rules in  $P$  with  $\mathbf{p}$  in their head, that  $I \models \mathbf{not} \mathbf{p}(\bar{t}) = l_j\theta$ . Now observe that there exists a ground rule  $r = \mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1\theta, \dots, V_m\theta)) \leftarrow l_1\theta, \dots, l_k\theta \in \text{Gnd}(P)$ . This is clear since a non-ground rule of this form is in  $P_{\hat{s}_i} \subset P$ , and because  $\hat{s}_i\theta$  is a ground instance of  $\hat{s}_i$ , the body of  $r$ . Since  $P_s \models l_j\theta$  for all  $i$ , it follows that  $I \models \hat{s}_i = \text{body}(r)$  and by  $I \models r$  that  $I \models \mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1\theta, \dots, V_m\theta))$ , i.e.  $\mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1\theta, \dots, V_m\theta)) \in I$ .

**Proof ( $\Leftarrow$ ):** Assume that  $\mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1\theta, \dots, V_m\theta)) \in I$ . This atom must be supported by some rule in  $r \in \text{Gnd}(P)$  with  $I \models \text{body}(r)$  (Eiter, Ianni, and Krennwallner 2009, Theorem 6), which can only be of the form  $r = \mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1\theta, \dots, V_m\theta)) \leftarrow l_1\theta, \dots, l_k\theta$ . We can inductively construct an SLDNF-Refutation  $s \cup \{\leftarrow \hat{s}_i\}$  with the following key insight: If some  $l_j\theta = \mathbf{p}(\bar{t})$  is positive, then it follows that  $I \models \mathbf{p}(\bar{t})$  by  $I \models \hat{s}_i = \text{body}(r)$ . Thus,  $\mathbf{p}(\bar{t})$  must be supported, which can only be due to the fact  $\mathbf{p}(\bar{t}) \in P_s$ . So,  $\mathbf{p}(\bar{t}) \in s$  and  $\leftarrow l_1\theta, \dots, l_{j-1}\theta$  is a binary resolvent of  $\leftarrow l_1\theta, \dots, l_{j-1}\theta, l_j\theta$  and of  $\mathbf{p}(\bar{t})$ . If  $l_j\theta = \mathbf{not} \mathbf{p}(\bar{t})$  is negative, then by  $I \models l_j$  it follows that  $\mathbf{p}(\bar{t}) \notin I$  and that  $\mathbf{p}(\bar{t}) \notin s$ . So, the derivation  $\leftarrow \mathbf{p}(\bar{t})$  fails finitely and  $l_j$  can be deleted from the goal clause  $\leftarrow l_1\theta, \dots, l_{j-1}\theta, l_j\theta$ .

From this result it follows that there is a one-to-one correspondence between the answer sets of  $P_s \cup P_{A_s} \cup P_{\hat{\Psi}}$  and the computed answers  $\theta$  s.t.  $s \vdash_{\text{SLDNF}} \hat{s}_i\theta$ . This is clear because of the choice rule in  $P_{\hat{\Psi}} \subset P_{\hat{\Psi}}$ .

The rule in  $P_{\hat{s}_i, \hat{a}_{i,j}}$  has a ground instance for every occurrence of  $\mathbf{cStateCovers}$  in  $I$ . The body of this ground instance will be true if (1)  $\hat{s}_i$  is "selected" in  $I$  (e.g. by the choice rule in  $P_{\hat{\Psi}}$ ), (2) the substitution  $\theta$  related with the ground instance represents a computed answer (i.e.  $s \vdash_{\text{SLDNF}} \hat{s}_i\theta$ ), and (3) there exists an admissible ground action matching  $(\mathbf{p}(\bar{t}))\theta$ , which is the case if  $(\mathbf{p}(\bar{t}))\theta \in A_s$  (i.e. is a fact in  $P_{A_s}$ ). These criteria correspond with the ones from Definition 3.2.3 and the rule head rightly asserts that  $(\hat{s}_i, \hat{a}_{i,j})$  covers  $(s, a)$ . Generalising this argument, we can see how  $P_{\hat{s}_i, \hat{a}_{i,j}}$  identifies all admissible actions for which the covering relation holds.

**Summary.** The answer sets of the program  $P_s \cup P_{A_s} \cup P_{\hat{\Psi}}$  correspond one-to-one with the abstract states that cover  $s$ , as marked by  $\mathbf{cStateChoice}$ . Each answer set contains also a complete listing (via  $\mathbf{cActionCovers}$ ) of the admissible abstract actions and of their covered admissible concrete actions. The program  $P_s \cup P_{A_s} \cup P_{\hat{\Psi}} \cup P_{<}$  yields at most one such answer set, corresponding to the covering abstract state with the smallest index.

### 3.3.2 Online Learning with the ASP-based CARCASS

In this subsection, we show how our ASP-based CARCASS translation can be applied to abstraction-based online learning methods such as Algorithm 3.1. For every agent-environment interaction cycle, we need to compute the abstract state  $\hat{s}$  corresponding to the observed concrete state, its admissible abstract actions  $\hat{A}_{\hat{s}}$  and the concrete actions covered by each of them. Algorithm 3.2 provides an example of how this can be done using the ASP translation.

In the first line, the ASP-translation is handed to an answer-set solver, which yields exactly one answer set  $I$ , as described previously. Lines 2–6 are then concerned with the extraction of all needed information from  $I$ . Note that we use the labelling function only during modelling and solving. When extracting abstract states and state-action pairs, we convert them back from their label to their original representations. This is not necessary but ensures compatibility with Algorithm 3.1.

---

**Algorithm 3.2:** CARCASS online interaction, implemented using ASP
 

---

**Input:** Current state  $s$   
**Input:** Available actions  $A_s$   
**Parameter:** CARCASS  $C = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$   
**Parameter:** Labelling function  $\lambda$   
**Output:** Abstract state  $\hat{s}$  such that  $s \in \llbracket \hat{s} \rrbracket_C$   
**Output:** Available abstract actions  $\hat{A}_{\hat{s}}$   
**Output:** Mappings to ground actions  $\llbracket \hat{s}, \hat{a} \rrbracket_C^s$  for all  $\hat{a} \in \hat{A}_{\hat{s}}$

- 1  $\{I\} \leftarrow \mathcal{AS}(P_{\hat{\Psi}} \cup P_{<} \cup P_s \cup P_{A_s})$
- 2  $\hat{s} \leftarrow \lambda^{-1}(t)$  for the only **cStateChoice**( $t$ )  $\in I$
- 3  $\hat{A}_{\hat{s}} \leftarrow \{\hat{a} \mid \mathbf{cActionCovers}(x, a) \in I \wedge \lambda(\hat{s}, \hat{a}) = x\}$
- 4 **for**  $\hat{a} \in \hat{A}_{\hat{s}}$  **do**
- 5    $\llbracket \hat{s}, \hat{a} \rrbracket_C^s \leftarrow \{a \mid \mathbf{cActionCovers}(x, a) \in I \wedge \lambda(\hat{s}, \hat{a}) = x\}$
- 6 **end**

---

Next, we discuss several extensions to Algorithm 3.2, resulting in Algorithm 3.3. A premise of the CARCASS framework is the ability to add background knowledge, as exemplified by van Otterlo (2008, p. 262) in the context of Prolog. In ASP, the addition of background knowledge in the form of a program  $P_B$  is as simple as joining it with our previous program, i.e.,  $P_{\hat{\Psi}} \cup P_{<} \cup P_s \cup P_{A_s} \cup P_B$ . The restrictions are that this joined program produces still just one answer set and that  $P_B$  does not make use of the atoms used in our CARCASS translation. The former restriction can be slightly lifted if  $P_B$  contains weak constraints. In this case, we allow the above program to produce multiple answer sets. However, only one of them is allowed to be optimal.<sup>7</sup> The computation of

---

<sup>7</sup>If multiple optimal answer sets are allowed, they can disagree on the chosen CARCASS state. This motivates our restriction. As an idea to loosen this restriction, a weak constraint of the form

this optimal answer set (Algorithm 3.3, line 1), and which one is selected in case there are multiple optimal answer sets despite our restriction, is left to the answer-set solver.

So far, we required every concrete state to be covered by at least one abstract state. If some state is not covered by any abstract state, there is no answer set and Algorithm 3.2 breaks. This is also observed by van Otterlo (2008, p. 255), who recommends the addition of an abstract state which covers all states and is always last in the decision list. In our translation, we refer to this as the *gutter* state, formalised as follows:

$$P_{\text{gutter}} \doteq \left\{ \begin{array}{l} \mathbf{cState}(\text{"gutter"}, \#sup). \\ \mathbf{cStateCovers}(\text{"gutter"}, ()). \end{array} \right\}$$

This program joined with our previous programs (Algorithm 3.3, line 1). The special constant  $\#sup$  (as in *supremum*) is interpreted as  $\infty$  (Gebser, Kaminski, Kaufmann, and Schaub 2012, p. 115). The gutter state is therefore always at the last position of the decision list. In terms of actions, we assign one abstract action  $\rho$  to the gutter state, with  $A_s$  as the set of covered actions (Algorithm 3.3, lines 3–6).

Another peculiarity of both Algorithm 3.2 and Algorithm 3.3 is the treatment of an abstract action  $\hat{a}$  not covering any concrete actions. In this case, no related instances of  $\mathbf{cActionCovers}$  are part of the answer set and  $\hat{a}$  will not count as admissible in our implementation. So, it is possible for one abstract state  $\hat{s}$  to have different sets of admissible actions in different concrete states. In our implementation, this is handled by adding the set of admissible actions to the state description, forming a refined abstract state  $\hat{x}_t = (\hat{s}_t, \hat{A}_{\hat{s}_t})$ . This way, if at any time points  $t_1, t_2$  during the learning process, we encounter the same CARCASS state ( $\hat{s}_{t_1} = \hat{s}_{t_2}$ ) but their sets of admissible actions are decided to be different ( $\hat{A}_{\hat{s}_{t_1}} \neq \hat{A}_{\hat{s}_{t_2}}$ ), we conclude that they are different abstract states ( $\hat{x}_{t_1} \neq \hat{x}_{t_2}$ ) for the purposes of online learning. This is formalised in a new  $q$ -update, which is amended in Algorithm 3.1.

$$\hat{q}(\hat{x}_t, \hat{a}_t) \leftarrow \hat{q}(\hat{x}_t, \hat{a}_t) + \alpha[r_{t+1} + \gamma \max_{\hat{a}' \in \hat{A}_{\hat{s}_{t+1}}} \{\hat{q}(\hat{x}_{t+1}, \hat{a}')\} - \hat{q}(\hat{x}_t, \hat{a}_t)]$$

### 3.3.3 Computing the CARCASS Semantics with ASP

Our ASP translation can be used to model the various set definitions forming the CARCASS semantics, namely:  $S_{\hat{s}}$ ,  $\llbracket \hat{s} \rrbracket_{\hat{C}}$ ,  $\Psi_{\hat{s}, \hat{a}}$ ,  $\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}$ , and  $\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^s$ . Apart from the programs defined above we need a representation of the full state-action space  $\Psi$ . To this end, we assume the existence of a program  $P_{\Psi}$  such that  $\mathcal{AS}(P_{\Psi}) = \{s \cup \{\mathbf{a}(\bar{t})\} \mid (s, \mathbf{a}(\bar{t})) \in \Psi\}$ .<sup>8</sup>In words, every answer set of  $P_{\Psi}$  should model one state-action pair in  $\Psi$ .

<sup>8</sup> $:\sim \mathbf{cStateChoice}(R), \mathbf{cState}(R, N). [N@1, R]$  can be added to  $P_{<}$ . This constraint acts as a tie breaker on the lowest priority level, provided that other optimization statements from  $P_B$  are defined only on priority levels 2 or higher. This way, if multiple optimal answer sets exist, it is at least ensured that they all correspond to the same CARCASS state.

---

**Algorithm 3.3:** CARCASS online interaction, implemented using ASP.

Extended with background knowledge, optimisation and gutter

---

**Input:** Current state  $s$   
**Input:** Available actions  $A_s$   
**Parameter:** CARCASS  $C = \langle (\hat{s}_1, \hat{A}_{\hat{s}_1}), \dots, (\hat{s}_n, \hat{A}_{\hat{s}_n}) \rangle$   
**Parameter:** Labelling function  $\lambda$   
**Parameter:** Program  $P_B$  with background knowledge, possibly including optimisation statements  
**Output:** Abstract state  $\hat{s}$  such that  $s \in \llbracket \hat{s} \rrbracket_{\hat{C}}$   
**Output:** Available abstract actions  $\hat{A}_{\hat{s}}$   
**Output:** Mappings to concrete actions  $\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^s$  for all  $\hat{a} \in \hat{A}_{\hat{s}}$

- 1  $\{I\} \leftarrow$  The only optimal answer set of  $\mathcal{AS}(P_{\Psi} \cup P_{<} \cup P_s \cup P_{A_s} \cup P_{\text{gutter}} \cup P_B)$
- 2  $\hat{s} \leftarrow \lambda^{-1}(t)$  for the only **cStateChoice**( $t$ )  $\in I$
- 3 **if**  $\lambda(\hat{s}) = \text{"gutter"}$  **then**
- 4      $\hat{A}(\hat{s}) \leftarrow \{\rho\}$
- 5      $\llbracket \hat{s}, \rho \rrbracket_{\hat{C}}^s \leftarrow A_s$
- 6 **end**
- 7 **else**
- 8      $\hat{A}_{\hat{s}} \leftarrow \{\hat{a} \mid \mathbf{cActionCovers}(x, a) \in I \wedge \lambda(\hat{s}, \hat{a}) = x\}$
- 9     **for**  $\hat{a} \in \hat{A}_{\hat{s}}$  **do**
- 10          $\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^s \leftarrow \{a \mid \mathbf{cActionCovers}(x, a) \in I \wedge \lambda(\hat{s}, \hat{a}) = x\}$
- 11     **end**
- 12 **end**

---

At the core of our approach is the program  $P_{\Psi} \cup P_{\Psi}$ , which has as its answer sets every possible combination of state-action pair and covering abstract state. Recall from the definition of  $P_{\Psi}$  that different covering abstract actions do not get their own answer set, but are listed in the answer set corresponding with their abstract state. In line with the generate-and-test methodology, we can use constraints to delete unintended answer sets. The following constraints can be used to delete any answer set that does not correspond

---

<sup>8</sup>The program  $P_{\Psi}$  may be a compact representation of the state-action space using MDP-specific background knowledge or a general enumeration scheme like the following: Given an RMDP  $(\Sigma, S, A, R, \Psi, p, \gamma)$  and a predicate **stateChoice**/ $1$  not occurring in  $\Sigma$ , we could define  $P_{\Psi}$  by the rules:

$$P_{\Psi} \doteq \left\{ \begin{array}{ll} 1 = \{ \mathbf{stateChoice}(I) : I = 1.. \mu \}. & \text{with } \mu \doteq |S| \\ \mathbf{p}(\bar{t}) \leftarrow \mathbf{stateChoice}(i). & \text{for all } s_i \in S \text{ and for all } \mathbf{p}(\bar{t}) \in s_i \\ 1 = \{ \mathbf{a}_1(\bar{t}_1); \mathbf{a}_2(\bar{t}_2); \dots \} \leftarrow \mathbf{stateChoice}(i). & \text{for all } s_i \in S \text{ with } A_{s_i} = \{ \mathbf{a}_1(\bar{t}_1), \mathbf{a}_2(\bar{t}_2), \dots \} \end{array} \right\}$$

With this definition,  $\mathcal{AS}(P_{\Psi}) = \{s_i \cup \{\mathbf{a}_j(\bar{t}_j)\} \cup \{\mathbf{stateChoice}(i)\} \mid (s_i, \mathbf{a}_j(\bar{t}_j)) \in \Psi\}$ . Note that the inclusion of **stateChoice** and other background atoms is no issue for our formulation and can be projected away if not needed using *#show* statements.

to a given abstract state  $\hat{s}$  or pair  $(\hat{s}, \hat{a})$ , respectively.

$$\begin{aligned} P_{\hat{s}} &\doteq \{ \leftarrow \mathbf{not\ cStateChoice}(\lambda(\hat{s})). \} \\ P_{\hat{s}, \hat{a}} &\doteq \{ \leftarrow \mathbf{not\ cActionCovers}(\lambda(\hat{s}, \hat{a}), \_). \} \end{aligned}$$

In addition, we can use clingo's *projection* feature to hide atoms in our answer sets. If we are only interested in atoms belonging to the signature of an RMDP  $\Sigma = (P_S \cup P_A, F)$ , we can project away the rest as follows.

$$\begin{aligned} P_{\text{proj-}S} &\doteq \{ \#show\ \mathbf{p}/\beta. \mid \mathbf{p}/\beta \in P_S \} \\ P_{\text{proj-}A} &\doteq \{ \#show\ \mathbf{p}/\beta. \mid \mathbf{p}/\beta \in P_A \} \end{aligned}$$

Different configurations of these constraints give us answer sets corresponding with different aspects of the CARCASS semantics.

$$\begin{aligned} S_{\hat{s}} &= \mathcal{AS}(P_{\Psi} \cup P_{\hat{\Psi}} \cup P_{\hat{s}} \cup P_{\text{proj-}S}) \\ \llbracket \hat{s} \rrbracket_{\hat{C}} &= \mathcal{AS}(P_{\Psi} \cup P_{\hat{\Psi}} \cup P_{\hat{s}} \cup P_{<} \cup P_{\text{proj-}S}) \\ \Psi_{\hat{s}, \hat{a}} &\approx \{ s \cup \{a\} \mid (s, a) \in \Psi_{\hat{s}, \hat{a}} \} = \mathcal{AS}(P_{\Psi} \cup P_{\hat{\Psi}} \cup P_{\hat{s}} \cup P_{\hat{s}, \hat{a}} \cup P_{\text{proj-}S} \cup P_{\text{proj-}A}) \\ \llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}} &\approx \{ s \cup \{a\} \mid (s, a) \in \llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}} \} = \mathcal{AS}(P_{\Psi} \cup P_{\hat{\Psi}} \cup P_{\hat{s}} \cup P_{\hat{s}, \hat{a}} \cup P_{<} \cup P_{\text{proj-}S} \cup P_{\text{proj-}A}) \\ \llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^s &= \mathcal{AS}(P_{\Psi} \cup P_{\hat{\Psi}} \cup P_{\hat{s}} \cup P_{\hat{s}, \hat{a}} \cup P_{<} \cup P_{\text{proj-}A}) \end{aligned}$$

### 3.4 Example

In this subsection, we introduce an example CARCASS, model it in ASP, and apply it in an online learning situation. As RMDP, we use a *3-blocks world*. For now, we give a brief description of this domain with only the concepts necessary to understand the example. For a proper introduction to the blocks world domain, see Chapter 4. The objects in this domain are the blocks  $a$ ,  $b$  and  $c$  as well as a table. The state description is formed from two predicates: **on/2** and **clear**. For a given state  $s$ , a block  $x \in \{a, b, c\}$  is considered to be stacked on top of a location  $y \in \{a, b, c, \text{table}\}$  iff  $\mathbf{on}(x, y) \in s$  is true. If nothing is stacked on top of some block  $x \in \{a, b, c\}$ , then  $x$  is considered clear, i.e.  $\mathbf{clear}(x) \in s$ . The following example describes a situation in which blocks  $a$  and  $c$  are on the table and  $b$  is stacked on top of  $a$ . Nothing is stacked on blocks  $b$  and  $c$ , so they are considered to be clear.

$$s_1 = \{ \mathbf{on}(a, \text{table}), \mathbf{on}(b, a), \mathbf{on}(c, \text{table}), \mathbf{clear}(a), \mathbf{clear}(c) \}$$

The actions available to the agent are formed from the predicate **move/2**. Any block  $x \in \{a, b, c\}$  can be moved to a new location  $y \in \{a, b, c, \text{table}\}$ , as long as  $x$  and  $y$  are considered to be clear (the table is always clear). Such an action is denoted by **move**( $x, y$ ). For state  $s_1$ , the following actions are available.

$$A_{s_1} = \{ \mathbf{move}(b, \text{table}), \mathbf{move}(b, c), \mathbf{move}(c, b) \}$$

After performing an action, the state changes deterministically in the obvious way. For example, performing the action  $a_1 = \mathbf{move}(b, \text{table})$  yields:

$$s_2 = \{ \mathbf{on}(a, \text{table}), \mathbf{on}(b, \text{table}), \mathbf{on}(c, \text{table}), \mathbf{clear}(a), \mathbf{clear}(b), \mathbf{clear}(c) \}$$

$$A_{s_2} = \{ \mathbf{move}(a, b), \mathbf{move}(b, a), \mathbf{move}(a, c), \mathbf{move}(c, a), \mathbf{move}(b, c), \mathbf{move}(c, b) \}$$

A reward may be given for reaching certain configurations (such as stacking all blocks), but this is not important for the purposes of this demonstration. The CARCASS which we want to investigate is taken from van Otterlo (2008, Example 5.2.1, p. 253) and looks as follows:

$\hat{s}_1:$	$\{ \mathbf{on}(A, B), \mathbf{on}(B, \text{table}), \mathbf{on}(C, \text{table}), A \neq B, B \neq C \}$
$\hat{a}_{1,1}, \hat{a}_{1,2}, \hat{a}_{1,3}:$	$\mathbf{move}(A, C), \mathbf{move}(C, A), \mathbf{move}(A, \text{table})$
$\hat{s}_2:$	$\{ \mathbf{on}(A, \text{table}), \mathbf{on}(B, \text{table}), \mathbf{on}(C, \text{table}), A \neq B, B \neq C, A \neq C \}$
$\hat{a}_{2,1}, \dots, \hat{a}_{2,6}:$	$\mathbf{move}(A, B), \mathbf{move}(B, A), \mathbf{move}(A, C),$ $\mathbf{move}(C, A), \mathbf{move}(B, C), \mathbf{move}(C, B)$
$\hat{s}_3:$	$\{ \mathbf{on}(A, B), \mathbf{on}(B, C), \mathbf{on}(C, \text{table}), A \neq B, B \neq C, C \neq \text{table} \}$
$\hat{a}_{3,1}:$	$\mathbf{move}(A, \text{table})$

For the translation to ASP, we need to define a labelling function  $\lambda$ . The following will do:

$$\begin{array}{lll} \lambda(\hat{s}_1) = \text{"s1"} & \lambda(\hat{s}_2) = \text{"s2"} & \lambda(\hat{s}_3) = \text{"s3"} \\ \lambda((\hat{s}_1, \hat{a}_{1,1})) = \text{"a11"} & \lambda((\hat{s}_2, \hat{a}_{2,1})) = \text{"a21"} & \lambda((\hat{s}_3, \hat{a}_{3,1})) = \text{"a31"} \\ \lambda((\hat{s}_1, \hat{a}_{1,2})) = \text{"a12"} & \lambda((\hat{s}_2, \hat{a}_{2,2})) = \text{"a22"} & \\ \lambda((\hat{s}_1, \hat{a}_{1,3})) = \text{"a13"} & \lambda((\hat{s}_2, \hat{a}_{2,3})) = \text{"a23"} & \\ & \lambda((\hat{s}_2, \hat{a}_{2,4})) = \text{"a24"} & \\ & \lambda((\hat{s}_2, \hat{a}_{2,5})) = \text{"a25"} & \\ & \lambda((\hat{s}_2, \hat{a}_{2,6})) = \text{"a26"} & \end{array}$$

According to the definitions, we construct  $P_{\hat{\Psi}}$  (Listing 3.1) and the order constraint  $P_{<}$  (Listing 3.2).

Let's now consider an example from online learning, assuming that the agent is currently in  $s_2$  as described above.<sup>9</sup> Constructing  $P_{s_2}$ , we get Listing 3.3. Next, the construction of  $P_{A_{s_2}}$  yields Listing 3.4.

Following Algorithm 3.2, we compute  $\mathcal{AS}(P_{\hat{\Psi}} \cup P_{<} \cup P_{s_2} \cup P_{A_{s_2}})$  and get the following atoms from Listing 3.5 as the single answer set.

<sup>9</sup>Compare with the online learning example given by van Otterlo (2008, Example 5.2.1, pp. 257–258)

```

1  cState("s1",1).
2  cStateCovers("s1", (A,B,C)) :-
3    on(A,B), on(B,table), on(C,table), A!=B, B!=C.
4  cActionCovers("a11", move(A,C))
5    :- cStateChoice("s1", cStateCovers("s1", (A,B,C)), move(A,C).
6  cActionCovers("a12", move(C,A))
7    :- cStateChoice("s1", cStateCovers("s1", (A,B,C)), move(C,A).
8  cActionCovers("a13", move(A,table))
9    :- cStateChoice("s1", cStateCovers("s1", (A,B,C)), move(A,table).
10
11 cState("s2",2).
12 cStateCovers("s2", (A,B,C)) :-
13   on(A,table), on(B,table), on(C,table), A!=B, B!=C, A!=C.
14 cActionCovers("a21", move(A,B)) :-
15   cStateChoice("s2", cStateCovers("s2", (A,B,C)), move(A,B).
16 cActionCovers("a22", move(B,A)) :-
17   cStateChoice("s2", cStateCovers("s2", (A,B,C)), move(B,A).
18 cActionCovers("a23", move(A,C)) :-
19   cStateChoice("s2", cStateCovers("s2", (A,B,C)), move(A,C).
20 cActionCovers("a24", move(C,A)) :-
21   cStateChoice("s2", cStateCovers("s2", (A,B,C)), move(C,A).
22 cActionCovers("a25", move(B,C)) :-
23   cStateChoice("s2", cStateCovers("s2", (A,B,C)), move(B,C).
24 cActionCovers("a26", move(C,B)) :-
25   cStateChoice("s2", cStateCovers("s2", (A,B,C)), move(C,B).
26
27 cState("s3",3).
28 cStateCovers("s3", (A,B,C)) :-
29   on(A,B), on(B,C), on(C,table), A!=B, B!=C, C!=table.
30 cActionCovers("a31", move(A,table)) :-
31   cStateChoice("s3", cStateCovers("s3", (A,B,C)), move(A,table).
32
33 1 = { cStateChoice(R) : cStateCovers(R, _) }.

```

Listing 3.1: The translated program  $P_{\hat{\psi}}$  of the example CARCASS.

```

1 :- cStateChoice(R), cStateCovers(R2, _),
2   cState(R,N), cState(R2,N2), N2 < N.

```

Listing 3.2: The the order constraint  $P_{<}$ .

In this case, only one abstract state was identified to cover  $s_2$ , namely:

$$\hat{s} = \lambda^{-1}("s2") = \mathbf{on}(A, \text{table}), \mathbf{on}(B, \text{table}), \mathbf{on}(C, \text{table}), A \neq B, B \neq C, A \neq C$$

The admissible abstract actions can be read off as instructed in Algorithm 3.3 (line 3):

$$A_{\hat{s}} = \left\{ \begin{array}{l} \mathbf{move}(A, B), \mathbf{move}(B, A), \mathbf{move}(A, C), \\ \mathbf{move}(C, A), \mathbf{move}(B, A), \mathbf{move}(C, B) \end{array} \right\}$$

### 3. ASP-BASED STATE-ACTION PAIR ABSTRACTIONS

---

```

1 on(a,table). on(b,table). on(c,table).
2 clear(a). clear(b). clear(c)

```

Listing 3.3: The facts of the state description  $P_{s_2}$ .

```

1 move(a,b). move(b,a). move(c,a).
2 move(a,c). move(b,c). move(c,b).

```

Listing 3.4: The admissible actions  $P_{A_{s_2}}$ .

In particular, we can conclude that  $\mathbf{move}(A, B) \in A_{\hat{s}}$  since both  $\mathbf{cActionCovers}(\text{"a21"}, a) \in I$  (where e.g.  $a = \mathbf{move}(c, b)$ ) and  $\lambda(\hat{s}, \mathbf{move}(A, B)) = \text{"a21"}$ .

This CARCASS illustrates how different substitutions (or computed answers, in the Prolog sense) form the coverage relation between abstract and concrete actions. In the given situation, every abstract action covers every single concrete action. For example, we conclude that the following concrete actions are covered by  $\mathbf{move}(A, B)$ :

$$\llbracket \hat{s}, \mathbf{move}(A, B) \rrbracket_{\hat{C}}^{s_2} = \left\{ \begin{array}{l} \mathbf{move}(a, b), \mathbf{move}(b, a), \mathbf{move}(a, c), \\ \mathbf{move}(c, a), \mathbf{move}(b, c), \mathbf{move}(c, b) \end{array} \right\}$$

The extracted values for  $\hat{s}$ ,  $A_{\hat{s}}$  and  $\llbracket \hat{s}, \hat{a} \rrbracket_{\hat{C}}^{s_2}$  (for all  $\hat{a} \in A_{\hat{s}}$ ) are now ready to be used in the current online learning step.

```

1 on(a,table)    on(b,table)    on(c,table)
2 clear(a)      clear(b)      clear(c)
3
4 move(a,b)     move(b,a)     move(c,a)
5 move(a,c)     move(b,c)     move(c,b)
6
7 cState("s1",1)
8 cState("s2",2)
9 cState("s3",3)
10
11 cStateCovers("s2", (c,b,a))    cStateCovers("s2", (b,c,a))
12 cStateCovers("s2", (c,a,b))    cStateCovers("s2", (a,c,b))
13 cStateCovers("s2", (b,a,c))    cStateCovers("s2", (a,b,c))
14 cStateChoice("s2")
15
16 cActionCovers("a21",move(c,b))    cActionCovers("a22",move(b,c))
17 cActionCovers("a21",move(b,c))    cActionCovers("a22",move(c,b))
18 cActionCovers("a21",move(c,a))    cActionCovers("a22",move(a,c))
19 cActionCovers("a21",move(a,c))    cActionCovers("a22",move(c,a))
20 cActionCovers("a21",move(b,a))    cActionCovers("a22",move(a,b))
21 cActionCovers("a21",move(a,b))    cActionCovers("a22",move(b,a))
22
23 cActionCovers("a23",move(c,a))    cActionCovers("a24",move(a,c))
24 cActionCovers("a23",move(b,a))    cActionCovers("a24",move(a,b))
25 cActionCovers("a23",move(c,b))    cActionCovers("a24",move(b,c))
26 cActionCovers("a23",move(a,b))    cActionCovers("a24",move(b,a))
27 cActionCovers("a23",move(b,c))    cActionCovers("a24",move(c,b))
28 cActionCovers("a23",move(a,c))    cActionCovers("a24",move(c,a))
29
30 cActionCovers("a25",move(b,a))    cActionCovers("a26",move(a,b))
31 cActionCovers("a25",move(c,a))    cActionCovers("a26",move(a,c))
32 cActionCovers("a25",move(a,b))    cActionCovers("a26",move(b,a))
33 cActionCovers("a25",move(c,b))    cActionCovers("a26",move(b,c))
34 cActionCovers("a25",move(a,c))    cActionCovers("a26",move(c,a))
35 cActionCovers("a25",move(b,c))    cActionCovers("a26",move(c,b))

```

Listing 3.5: The computed answer set.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

## Case Study: Blocks World

In this chapter we investigate the effects of state-action pair abstraction in the *blocks world* (Slaney and Thiébaux 2001), the canonical example for planning problems in many textbooks.<sup>1</sup> Roughly speaking, the  $n$ -blocks world domain consists of  $n \geq 1$  blocks and a table. In any given state, every block is located on the table or stacked on top of another block, forming a tower. The state space consists of the various configurations in which blocks can be stacked. Transitioning to a different state is possible by performing discrete actions, in which a single block with no other blocks stacked on top of it can be moved on top of another tower or onto the table. Example states and actions are given for a 3-blocks world in Section 3.4 and a formal description is provided later in this chapter.

When viewing the blocks world as a planning problem, one is interested in finding a sequence of actions that, when performed in order, cause a transition from some initial state to some goal state. Finding such a sequence of minimal length has been shown to be NP-Hard (Gupta and Nau 1991). Note however that finding any sequence is trivial. One can move all blocks to the table and then re-stack blocks in the desired configuration, which overall takes at most  $2(n - 1)$  actions (unstack-stack algorithm, Slaney and Thiébaux 2001, p. 129).

Another interesting property of the blocks world is the size of its state space. For a  $n$ -blocks world, the number of possible configurations is (Slaney and Thiébaux 2001, p. 124)<sup>2</sup>:

$$\sum_{i=1}^n \binom{n}{i} \frac{(n-1)!}{(i-1)!}$$

<sup>1</sup>The blocks world can even be called the "Hello World" of planning (Slaney and Thiébaux 2001, p. 147). Occurrences in textbooks include Gebser, Kaminski, Kaufmann, and Schaub (2012, p. 157), Lifschitz (2019, p. 129), and Beierle and Kern-Isberner (2019, p. 364).

<sup>2</sup>The same simplification was presented by Datler (2020, p. 15).

A lower bound of the growth rate is  $n!$ , which counts the number of states with exactly one tower, i.e. permutations of  $\{1, 2, \dots, n\}$ . The growth of the number of states is therefore superpolynomial in the number of blocks. Similarly interesting is the number of state-action pairs, which grows faster than the number of states by another polynomial factor.<sup>3</sup> As an illustration consider the following table:

$n$	Number of states	Number of state-action pairs <sup>4</sup>
5	501	2140
10	$5.894 \cdot 10^7$	-
15	$6.557 \cdot 10^{13}$	-
20	$3.277 \cdot 10^{20}$	-

The relational nature of the blocks world domain, the complexity of finding an optimal sequence of actions and the exploding state space make the blocks world an interesting test-bed for reinforcement learning and the application of abstraction functions. Further, there exists a rich background of theoretical results (Slaney and Thiébaux 2001) to put our experiments into context.

In the following, we give a formal description of the blocks world planning problem as RMDP. We then present an ASP-based CARCASS, which encodes an abstraction function inspired by the algorithms GN1 and GN2 (Slaney and Thiébaux 2001, pp. 129–131 citing Gupta and Nau 1992, pp. 229–230). With this abstraction function, we manage to keep the abstract state-action space constant in size as the number of blocks increases. As a trade-off, the abstraction does not preserve the optimal policy for the blocks world planning problem in general. We are not aware of any theoretical guarantees of convergence for the proposed abstraction.

To investigate the effect of abstract  $Q$ -learning under the presented abstraction function, we present exploratory empirical results, aimed at answering the following research questions:

- Q1** Does the proposed abstraction cause differences in sample efficiency?
- Q2** Does the proposed abstraction cause stability issues during the learning process?
- Q3** Does the proposed abstraction cause differences in quality of the learned policy?

The chapter is structured as follows: First, we present an RMDP formulation of the blocks world planning task, followed by a discussion of our abstraction function. We then present our empirical research in the form of three small experiments and close with a discussion of the results.

---

<sup>3</sup>Roughly speaking, we can move each of the  $n$  blocks on top of another one or onto the table. This gives a loose upper bound of  $|S| \cdot n^2$ .

<sup>4</sup>To get this number for  $n = 5$ , we enumerated all state-action pairs for five blocks in ASP, using the following definitions. This is unfeasible for  $n \geq 10$ , which is why we left those entries empty.

## 4.1 The Blocks World as RMDP

In the following, we provide a formulation of the blocks world planning problem as episodic RMDP (Definition 2.3.10)  $(\Sigma, S, A, R, \Psi, p, \gamma, \text{init}, S_{\text{term}})$ . Here, we adapt the ASP-based characterisation of Datler (2020, Chapter 3). To start, the signature  $\Sigma = (F, P_S \cup P_A)$  consists of the function symbols  $F = \{\text{table}/0, 0/0, 1/0, \dots, n-1/0\}$ , representing the table and each of the blocks, respectively. The state predicates are given by  $P_S = \{\text{on}/2, \text{subgoal}/2\}$ ,<sup>5</sup> where  $\text{on}(B, L)$  represents the fact that some block  $B$  is situated directly on top of some location  $L$  (i.e. the table or another block). Similarly,  $\text{subgoal}(B, L)$  symbolises that, in any goal state,  $B$  must be placed on  $L$ . This formulation allows us to specify partial and full goal descriptions, which correspond to elementary and primitive blocks world instances in the sense of Gupta and Nau (1991), respectively. The action predicates are given by  $P_A = \{\text{move}/2\}$ , where  $\text{move}(B, L)$  symbolises the action of moving block  $B$  to  $L$ .

We also make use of a list of extended state predicates  $P_{S+} = \{\text{block}/1, \text{location}/1, \text{occupied}/1, \text{clear}/1, \text{supported}/1, \text{terminal}/0\}$ . Here,  $\text{block}(B)$  specifies that  $B$  is a block and  $\text{location}(L)$  specifies that  $L$  is a location. If  $B$  is a block, it may be occupied by another block, as symbolised by  $\text{occupied}(B)$ . If a location  $L$  is the table, or a block not occupied by another block, it is considered to be clear (i.e.  $\text{clear}(L)$ ). A block  $B$  is supported (i.e.  $\text{supported}(B)$ ) if some block beneath  $B$  is placed on the table. Finally,  $\text{goal}$  indicates a goal state, in which all blocks are in the location as specified by the sub-goals. These intuitions are formalised in the following background knowledge:

```

1 block(B) :- B=0..(n-1).
2 location(table).
3 location(X) :- block(X).
4 occupied(B) :- block(B), on(_, B).
5 clear(L) :- location(L), not occupied(L).
6 supported(X) :- on(X, table).
7 supported(X) :- on(X, Y), supported(Y).
8 goal :- 0 = { not on(B, L) : subgoal(B, L) }.

```

### 4.1.1 State Space

We characterise the state space using the following ASP encoding (Datler 2020, p. 20):

```

1 { on(X, L) : location(L), L != X } = 1 :- block(X).
2 :- block(X), { on(Y, X) : block(Y) } > 1.
3 :- block(X), not supported(X).
4 subgoal(0, table).
5 subgoal(B2, B1) :- block(B1), block(B2), B2=B1+1.

```

<sup>5</sup>The presented formulation is slightly different than our actual encoding. Having the planning background in mind, and in an attempt to provide a general interface for all ASP-based RMDPs, we treat state predicates as *fluents* (e.g. Lifschitz 2019, Chapter 8.3). For example,  $\text{on}(B, L)$  may be represented as  $\text{tic}(\text{on}(B, L), T)$ , where  $T$  is a time point. Similarly, the occurrence of an action at time point  $T$  is symbolised by  $\text{act}(\text{move}(B, L), T)$ . For more on the origins of our blocks world implementation see Datler (2020).

In the encoding, we use a choice rule to place every block exactly on top of one location and ensure with a constraint that at most one block can be directly on top of any other block. Also, we make sure that every block is supported by the table. The sub-goals are fixed to specify just one state as the goal state, in which all blocks are stacked in order. It can be shown that the answer sets of this program in addition with the background knowledge above correspond one-to-one with the states of our RMDP.

#### 4.1.2 Terminal States and the Initial State Distribution

Generally, the set of terminal states is defined as the set of states in which all blocks are stacked as specified by the given **subgoal** atoms. Formally, a state at time point  $T$  is terminal if **goal/0** can be derived using the background knowledge. For the specified state space, there is only one such state. Therefore:

$$S_{\text{term}} = \{ \{ \mathbf{on}(0, \text{table}). \mathbf{subgoal}(0, \text{table}). \mathbf{on}(1, 0). \mathbf{subgoal}(1, 0). \dots \\ \dots \mathbf{on}(n-1, n-2). \mathbf{subgoal}(n-1, n-2). \} \}$$

The initial states are drawn from a random uniform distribution excluding the terminal state.

$$\text{init}(s) \doteq \begin{cases} \frac{1}{|S \setminus S_{\text{term}}|} & \text{for } s \in S \setminus S_{\text{term}} \\ 0 & \text{otw.} \end{cases}$$

For blocks world RMDPs with nine blocks or less, we use the state space characterisation above to enumerate all states as answer sets, and simply choose one of them randomly whenever an initial state needs to be sampled. For larger blocks worlds, this is not feasible and we turn to Algorithm 4.1.<sup>6</sup> In both cases, if we by chance generate the terminal state, we simply repeat the sampling process.

#### 4.1.3 Admissible Actions

To mark moves as admissible, we introduce an additional predicate **admissibleMove/2**. Given a state, the set of admissible moves is defined by:

<pre> 1  admissibleMove(B,L) :- block(B), clear(B), clear(L), B!=L, 2  not on(B,L), not goal.</pre>
---

Note that we do not admit any moves in the terminal state.<sup>7</sup> To compute answer sets corresponding one-to-one with  $\Psi$ , we can use the above definition, the state space characterisation, and the following choice rule:

<sup>6</sup>The described algorithm is limited in application due to its space requirements. For our purposes, it suffices. Slaney and Thiébaux (2001) describe possible optimisations and another algorithm which can handle generating random states for worlds beyond 10000 blocks.

<sup>7</sup>Admitting no actions for some state naturally labels it as being terminal. Although this formulation appears inconsistent with the idea of absorbing states, they achieve the same conceptually. If  $A_{s_{t+1}} = \emptyset$  for some transition  $(s_t, a_t, r_{t+1}, s_{t+1})$ , the agent simply stops and computes the realised return as  $g_t = r_{t+1}$ , omitting the infinite sum of zero-rewards that would be added by further traversing an absorbing state.

---

**Algorithm 4.1:** Generation of random blocks world states (Slaney and Thiébaux 2001, p. 126)

---

**Input:** The number of blocks  $n \geq 0$

**Output:** Sampled initial state  $s_0 \sim \text{init}(\cdot)$

- 1 Let  $g(n, k) = \sum_{i=0}^n \binom{n}{i} \frac{(n+k-1)!}{(i+k-1)!}$
  - 2 Start with an empty table and  $n$  ungrounded towers each consisting of a single block
  - 3 **repeat**
  - 4     Arbitrarily select one of the  $\phi$  yet ungrounded towers
  - 5     Select the table with probability  $\frac{g(\phi-1, \tau+1)}{g(\phi, \tau)}$  or one of the other towers (grounded or not) each with probability  $\frac{g(\phi-1, \tau)}{g(\phi, \tau)}$ , and place the selected ungrounded tower onto it
  - 6 **until** all towers are grounded
- 

1  $1 = \{ \text{move}(B, L) : \text{admissibleMove}(B, L) \}.$

#### 4.1.4 Dynamics and Discounting

In order to describe the RMDP dynamics, we introduce the predicates **nextOn**/2, **nextSubgoal**/2, and **nextReward**/1. Let the tuple  $(s, a, r_{\text{next}}, s_{\text{next}})$  correspond with the set:

$$\begin{aligned}
 & s \cup \{a\} \cup \{\mathbf{nextReward}(r_{\text{next}})\} \\
 & \cup \{\mathbf{nextOn}(B, L) \mid \mathbf{on}(B, L) \in s_{\text{next}}\} \\
 & \cup \{\mathbf{nextSubgoal}(B, L) \mid \mathbf{subgoal}(B, L) \in s_{\text{next}}\}
 \end{aligned}$$

Then, the dynamics are defined by the program  $P_{\text{Dynamics}}$ :

```

1 nextOn(B, L) :- on(B, L), not move(B, _).
2 nextOn(B, L) :- move(B, L).
3 nextSubgoal(B, L) :- subgoal(B, L).
4 nextReward(99) :- goal.
5 nextReward(-1) :- not goal.

```

So, after the transition, the moved block is at its new location and all other blocks are located where they were before. The sub-goals stay as they are. For every transition, a negative reward of  $-1$  is given as penalty. When reaching a goal state, a reward of 99 is given.<sup>8</sup> Formally,  $p(s_{\text{next}}, r_{\text{next}} \mid s, a) = 1$  iff the only answer set in  $\mathcal{AS}(P_s \cup \{a.\} \cup P_{\text{Dynamics}})$  corresponds with the tuple  $(s, a, r_{\text{next}}, s_{\text{next}})$ , and zero otherwise.

<sup>8</sup>The represented program is a simplification of our implementation. For a more detailed discussion see Datler (2020, p. 22).

The discount factor is set to  $\gamma = 1$ . Thus, if the goal state is reached in some episode, the return will be 100 minus all moves that occurred during the episode.<sup>9</sup> Thus, shorter episodes are associated with higher returns at time point zero. The optimal policy corresponds to the one which always generates episodes of minimal length, implementing an optimal blocks world planner.

## 4.2 ASP-Based Blocks World CARCASS

In the following, we propose an ASP-based CARCASS abstraction function for the blocks world RMDP. Although the blocks world planning problem is NP-hard, there exist some simple algorithms that have been shown to achieve near-optimal performance (Slaney and Thiébaux 2001, pp. 129, 141). In particular, we take inspiration from the *GN1* and *GN2* algorithms (ibid. citing Gupta and Nau 1991; Gupta and Nau 1992).

In order to explain the CARCASS, we first need to introduce some concepts. We will do so in the form of domain knowledge, to be used by the CARCASS states. To keep this section self-contained, we repeat some of the predicate definitions from before. States and actions are represented just as before, using the predicates **on**/2, **subgoal**/2, and **move**/2. In addition we define the predicates **clear**/1, **above**/2, and **goalRelevant**/1 by the rules:

```

1 clear(A) :- on(A, _), not on(_, A).
2 clear(A) :- table = A.
3 above(A, B) :- on(A, B).
4 above(A, B) :- on(A, C), above(C, B).
5 goalRelevant(A) :- subgoal(A, _).
6 goalRelevant(A) :- subgoal(_, A).

```

The meaning of **clear**/1 was described previously. The predicate **above**/2 captures the transitive closure of **on**/2 and **goalRelevant**/1 collects all blocks which appear in sub-goals. For the blocks world RMDP as defined above, we fully specified the goal state. Therefore, all blocks appear in sub-goals. However, the CARCASS is designed with partial goal state descriptions in mind, in which some blocks may not be goal relevant.<sup>10</sup>

Next, we need a method to identify whether blocks are in their final position. To this end, let's map out some intuitions by considering an example state: { **on**(4, table), **on**(0, 4), **on**(1, 0), **on**(2, 1), **on**(3, table), **subgoal**(1, 0), **subgoal**(2, 1), **subgoal**(3, 2) }. The sub-goals specify that the blocks 0, 1, 2, and 3 need to be stacked in order. However, the location of blocks 0 and 4 are not specified, making this a partial goal description. Block 0 is the bottom-most block mentioned in our goal description and needs to be in

<sup>9</sup>In the extreme case, the goal state is never reached and the return will be  $-\infty$ . In practice, this case can be avoided by setting a time limit, as discussed for *Q*-learning in Algorithm 2.1.

<sup>10</sup>For example, consider the state {**on**(0, table), **on**(1, table), **on**(2, table), **subgoal**(1, 2)}. This state represents a valid 3-blocks world planning problem, but does not occur in the state space of our RDMP specification. One can think of three states which satisfy the sub-goal. In every case, block 1 must be stacked on top of block 2. But block 0 can occur on top of block 1, beneath block 2, or on the table.

its final position before stacking other goal-relevant blocks on top of it. We therefore call block 0 a *basis* with respect to the given goal description. Is block 0 in its final position? The location of block 0 is not specified in the partial goal description. Neither is the location of block 4, the only block below. Therefore, block 0 does not need to be moved in order to reach a goal state and can be considered to be in its final position. In the given state, blocks 1, 2 and 3 are stacked on top of block 0 as specified by our goal description, so their locations can be considered to be final as well. Block 4 however is not in its final position and we consider the stack of goal relevant blocks to be *incomplete*.

To formalise these intuitions, we define the predicates **finalBasis**/1, **final**/2, and **incomplete**/2 by the rules:<sup>11</sup>

```

1 finalBasis(X) :- subgoal(_, X), not subgoal(X, _),
2                 0 = #count { B : above(X, B), goalRelevant(B) }.
3 final(X, X) :- finalBasis(X).
4 final(X, B1) :- final(X, B2), on(B1, B2), subgoal(B1, B2).
5 incomplete(X, Top) :- final(X, Top), subgoal(B, Top), not on(B, Top).

```

In words, **finalBasis**( $X$ ) holds for block  $B$  if  $B$  occurs in the goal description only as a location, but the location of  $B$  itself is not specified and furthermore no other goal-relevant blocks can be located below  $B$ . A block  $B$  can be considered to be in its final position with respect to some basis  $X$  (as symbolised by **final**( $X, B$ )) if it is itself the basis or if  $B$  and all blocks below it (up to the basis) are stacked according to the given goal description. A block  $B$  that is in its final position (w.r.t. a basis  $X$ ), is marked as a the current incomplete top w.r.t  $X$  (as symbolised by **incomplete**( $X, B$ )) if there exists some other block which, according to the given goal description, needs to be stacked on top of  $B$ . Applying these definitions to the above example yields { **finalBasis**(0), **final**(0, 0), **final**(0, 1), **final**(0, 2), **incomplete**(0, 2), } as part of the only answer set. For partial goal state descriptions, it is generally possible to have more than one basis.<sup>12</sup> In full state descriptions like the one specified for our RMDP, the only complete basis is always the table, since **subgoal**(0, table) occurs in all RMDP states.

<sup>11</sup>Note that many atoms from the implementation have been renamed, mostly to improve clarity. The presented predicates can be translated as follows: **finalBasis**  $\mapsto$  **goodPartialTowerBase**, **final**  $\mapsto$  **goodPartialTower**, **incomplete**  $\mapsto$  **goodPartialTowerTop**. CARCASS-specific predicates were also renamed and can be translated as follows: **cState**  $\mapsto$  **rule**, **cStateCovers**  $\mapsto$  **applicable**, **cActionCovers**  $\mapsto$  **abstractAction**, and **cStateChoice**  $\mapsto$  **choose**.

Also note that there may exist a more concise formulation of the CARCASS states, involving the predicate **nextCompletingMove**/2 as follows: **nextCompletingMove**( $X, \text{Top}, \text{Next}$ ) :- **final**( $X, \text{Top}$ ), **subgoal**( $\text{Next}, \text{Top}$ ), not **on**( $\text{Next}, \text{Top}$ ).

Finally, note that there is a mistake in the definition of **finalBasis**. As written, the rule cannot detect a basis being in its final position if it is stacked on top of another goal tower which is already complete and in its final position. For example, consider the state {**on**(1, 2), **on**(2, 3), **on**(3, 4), **on**(4, table), **subgoal**(1, 2), **subgoal**(3, 4)}, where **finalBasis**(2) should be derivable but isn't. For the provided RMDP, this mistake is irrelevant as we only ever have to deal with one basis, namely the table.

<sup>12</sup>As example, consider the state { **on**(0, table), **on**(1, table), **on**(2, table), **on**(3, table), **subgoal**(1, 0), **subgoal**(3, 2) }

### 4.2.1 CARCASS Abstraction Function

Equipped with the above domain knowledge, we are ready to specify the CARCASS-based abstraction function. There are five abstract states, modelled in ASP using the translation scheme described in Section 3.3 but with some modifications. First, we use a slightly different formulation for  $P_{\hat{a}_{i,j}}$ , omitting the admissible action atom in the rule body.<sup>13</sup> Second, we introduce a *gutter action* as an abstract action admissible in all abstract states. Its intention is to cover all concrete actions that are not covered by any other abstract action. Formally:

```

1 cActionCovers("gutterAction", move(X, Y)) :-
2   clear(X), clear(Y), X!=Y, X!=table,
3   not cActionCovers(move(_, _), move(X, Y)), not on(X, Y).

```

Further modifications to the translation scheme are mentioned as they occur. For now, let's specify the CARCASS states, starting with `r0`.

```

1 cState("r0", 0).
2 cStateCovers("r0", (Top, Next)) :-
3   incomplete(_, Top), clear(Top), subgoal(Next, Top), clear(Next).
4 cActionCovers("move(next,top)", move(Next, Top)) :-
5   cStateChoice("r0"), cStateCovers("r0", (Top, Next)).

```

Intuitively, `r0` covers all states in which there exists goal-relevant tower that is in its final position but incomplete. Also, the top of the tower is clear, and so is the next block which belongs on top of it.

For example, consider the state  $\{ \mathbf{on}(0, \text{table}), \mathbf{on}(1, 0), \mathbf{on}(2, \text{table}), \mathbf{subgoal}(0, \text{table}), \mathbf{subgoal}(1, 0), \mathbf{subgoal}(2, 1) \}$ , in which the finally-positioned, goal-relevant but incomplete tower consists of the table, block 0, and 1. To advance this state towards a goal state, the next block needs to be moved on top of the clear tower, a general move modelled by the `move(next, top)` action. Note that if multiple incomplete towers with clear tops and clear next blocks exist, then all moves of this type are covered by `move(next, top)`. The remaining actions are covered by `gutterAction`. As a comparison, `r0` covers case (1) of the GN1 algorithm (Slaney and Thiébaux 2001, p. 129) and the moves covered by `move(next, top)` are *constructive moves*.

What if an incomplete tower exists but is not clear? This case is covered by `r1`.

```

1 cState("r1", 1).
2 cStateCovers("r1", (Basis, Top, BadTop)) :-

```

<sup>13</sup>The used formulation is more akin to the scheme below. Compared with Definition 3.3.5, also inadmissible actions (or actions that do not exist) may be covered by the relation as defined here. Without this "safeguard", the designer needs to be more careful in designing the CARCASS.

$$P_{\hat{s}_i, \hat{a}_{i,j}} \doteq \left\{ \begin{array}{l} \mathbf{cActionCovers}(\lambda(\hat{s}_i, \hat{a}_{i,j}), p(\bar{t})) \\ \leftarrow \mathbf{cStateChoice}(\lambda(\hat{s}_i)), \mathbf{cStateCovers}(\lambda(\hat{s}_i), (V_1, \dots, V_i)). \end{array} \right\}$$

```

3 | incomplete(Basis, Top), not clear(Top), above(BadTop, Top), clear(
   |   BadTop).
4 | cActionCovers("move(badTop,table)", move(BadTop, table)) :-
5 |   cStateChoice("r1"), cStateCovers("r1", (_, _, BadTop)).
6 | cActionCovers("move(badTop,other)", move(BadTop, Other)) :-
7 |   cStateChoice("r1"), cStateCovers("r1", (Basis, Top, BadTop)),
8 |   clear(Other), Other != BadTop, Other != table.

```

The covering relation is similar to that of `r0`, but `Top` is not clear. The clear block above `Top`, called `BadTop`, therefore needs to be moved.

For example, consider the state  $\{ \mathbf{on}(0, \text{table}), \mathbf{on}(2, 0), \mathbf{on}(1, \text{table}), \mathbf{subgoal}(0, \text{table}), \mathbf{subgoal}(1, 0), \mathbf{subgoal}(2, 1) \}$ , in which the finally-positioned, goal-relevant but incomplete tower consists of the table and of block 0. In order to advance this state towards a goal state, block 2 needs to be moved away. There are two ways to move `BadTop`. First, it can always be moved to the table, a move covered by the abstract action `move(badTop, table)`. Second, if another tower exists (which may or may not be goal relevant, incomplete and/or in its final position), the block can be moved on top of that other tower, as covered by `move(badTop, other)`.<sup>14</sup> All other actions are covered by `gutterAction`.

How does this relate to GN1 and GN2 algorithms? First, observe that, due to order of rules built-in to the CARCASS semantics, state `r1` will only be associated with concrete states not covered by `r0`. Therefore, no constructive actions are available in `r1` and all subsequent states. By definition, this state and all following states cover situations in which the blocks world state is deadlocked (Slaney and Thiébaux 2001, p. 128). The state `r1` does not cleanly map onto any of the cases for GN1 or GN2 but is related to the second case in the definition of the function  $\delta_\pi$  (ibid. p. 130). Note that `move(badTop, other)` is not considered to be a sensible action due to the risk of introducing new deadlocks. Thus, a learned abstract policy, should generally prefer `move(badTop, table)` over `move(badTop, other)`.

Next, consider the case in which a goal-relevant tower exists, is in its final position, incomplete and clear but in which the next block belonging on top of it is not clear. This case is covered by `r2`.

```

1 | cState("r2", 2).
2 | cStateCovers("r2", (Top, BadTop)) :-
3 |   incomplete(Basis, Top), clear(Top), subgoal(Next, Top),
4 |   not clear(Next), above(BadTop, Next), clear(BadTop).
5 | cActionCovers("move(badTop,table)", move(BadTop, table)) :-
6 |   cStateChoice("r2"), cStateCovers("r2", (_, BadTop)).
7 | cActionCovers("move(badTop,other)", move(BadTop, Other)) :-
8 |   cStateChoice("r2"), cStateCovers("r2", (Top, BadTop)),
9 |   clear(Other), Other != Top, Other != BadTop, Other != table.

```

<sup>14</sup>Note that the rule body of the second action deviates from the CARCASS-ASP translation scheme. A more clean (but less concise) formulation would split the above into two CARCASS states, one where only a single tower is present and another where more than one tower exists.

We identify the next block to be moved onto the goal-relevant tower as `Next` and the clear block on top of it as `BadTop`. As previously described for `r1`, there are two abstract actions, covering moves of `BadTop` to the table or on top of another tower (and the `gutterAction` as third abstract action).

As an example, consider  $\{ \mathbf{on}(0, \text{table}), \mathbf{on}(1, \text{table}), \mathbf{on}(2, 1), \mathbf{subgoal}(0, \text{table}), \mathbf{subgoal}(1, 0), \mathbf{subgoal}(2, 1) \}$ . Compared with (Slaney and Thiébaux 2001), this state is related to the first case in the definition of the function  $\delta_\pi$ .

The three abstract states discussed so far cover all states in our blocks world RMDP specification. To handle partial state descriptions as well, two more states are needed: State `r3` covers the case in which a basis is not in its final position and not clear, the sensible actions being to move blocks on top of the basis. State `r4` covers the case in which a basis is not in its final position but clear, in which case the basis can be moved to the table constructively. In the goal description as defined for our RMDP, these cases cannot occur: the basis is always the table, which can only be in its final position.

```

1 cState("r3",3).
2 cStateCovers("r3", (BadTop)) :-
3   not finalBasis(_, subgoal(_, Basis), not subgoal(Basis, _),
4   not clear(Basis), above(BadTop, Basis), clear(BadTop)).
5 cActionCovers("move(badTop,table)", move(BadTop, table)) :-
6   cStateChoice("r3"), cStateCovers("r3", (BadTop)).
7 cActionCovers("move(badTop,other)", move(BadTop, Other)) :-
8   cStateChoice("r3"), cStateCovers("r3", (BadTop)), clear(Other),
9   Other != BadTop, Other != table.
```

```

1 cState("r4",4).
2 cStateCovers("r4", (Basis)) :-
3   not finalBasis(_, subgoal(_, Basis),
4   not subgoal(Basis, _), clear(Basis)).
5 cActionCovers("move(basis,table)", move(Basis, table)) :-
6   cStateChoice("r4"), cStateCovers("r4", (Basis)).
```

#### 4.2.2 Convergence Guarantees, Correctness and Other Observations

Before discussing the experiments, let's apply the theory of abstraction (Section 2.5) to the presented abstraction function. First, the results from state abstraction are not immediately applicable as it is not intended to be used with abstract actions.

Looking at the presented results from state-action pair abstraction, we can check if our abstraction function is an MDP homomorphism. To this end, let  $(s_0, a_0)$  and  $(s_1, a_1)$  be

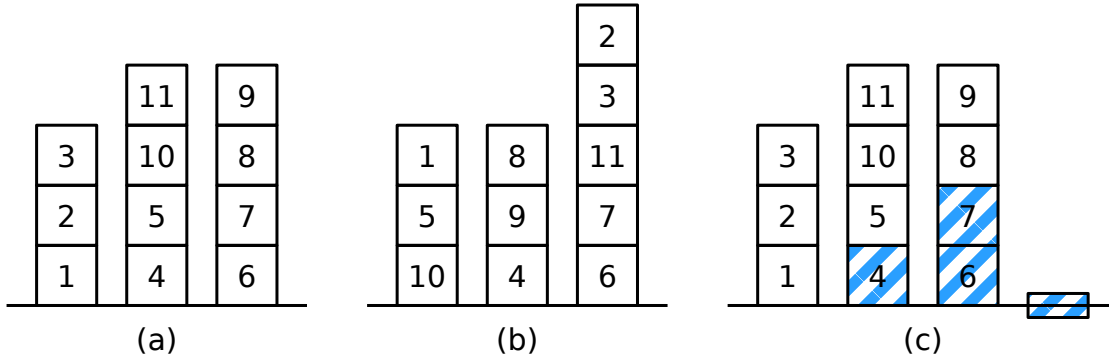


Figure 4.1: A blocks world planning problem from Slaney and Thiébaux (2001, p. 146) citing Kautz and Selman (1996). The initial state is shown in (a), the goal state in (b). In (c), the initial state is shown with in-position, goal-relevant towers marked in color. Blocks 4, 7 and the table are marked as the incomplete tops of the towers, all sharing the table as a basis.

two state-action pairs defined as:

$$\begin{aligned}
 s_0 &= \{\mathbf{on}(0, \text{table}), \mathbf{on}(1, \text{table}), \mathbf{on}(2, \text{table}), \\
 &\quad \mathbf{subgoal}(0, \text{table}), \mathbf{subgoal}(1, 0), \mathbf{subgoal}(2, 1)\} \\
 a_0 &= \mathbf{move}(1, 0) \\
 s_1 &= \{\mathbf{on}(0, \text{table}), \mathbf{on}(1, 0), \mathbf{on}(2, \text{table}), \\
 &\quad \mathbf{subgoal}(0, \text{table}), \mathbf{subgoal}(1, 0), \mathbf{subgoal}(2, 1)\} \\
 a_1 &= \mathbf{move}(2, 1)
 \end{aligned}$$

Both pairs are associated with the abstract state-action pair  $(r_0, \text{move}(\text{next}, \text{top}))$ . The expected rewards in the concrete MDP are  $r(s_0, a_0) = -1$  and  $r(s_1, a_1) = 99$ . By the definition of homomorphisms, it is necessary that  $r(s_0, a_0) = r(s_1, a_1)$ , which is not the case. Therefore, the presented abstraction function does not resemble an MDP homomorphism. So, we are not aware of any theoretical guarantees of convergence for the given abstraction function.

Next, let's see if there exists an abstract policy corresponding to the concrete optimal policy. In other words, given our CARCASS, can we represent abstract policies which resemble solutions to the optimal blocks world planning problem? For this, recall how our abstract states relate to GN1 and GN2 (Slaney and Thiébaux 2001, p. 129). The abstract state  $r_0$  covers states in which constructive actions are available and  $r_1$  and  $r_2$  cover cases in which the given state is deadlocked. Resolving deadlocks optimally is actually where the complexity of blocks world planning lies (Gupta and Nau 1992, p. 8). Consider the example in Figure 4.1. The initial state  $s_0$  is covered by both  $r_1$  (since blocks 4 and 7 are not clear) and  $r_2$  (since the table is clear), with  $r_1$  taking precedence as the associated abstract state. The abstract action  $\text{move}(\text{badTop}, \text{table})$  covers the concrete actions  $\mathbf{move}(11, \text{table})$  and  $\mathbf{move}(9, \text{table})$ . The abstract action  $\text{move}(\text{badTop}, \text{other})$

covers the actions `move(11, 3)`, `move(11, 9)`, `move(9, 3)`, and `move(9, 11)`. Other actions, such as `move(3, table)` or `move(3, 11)` are covered by `gutterAction`. Observe that  $s_0$  is deadlocked, the deadlocks being the sets  $\{8, 11\}$  and  $\{9, 11\}$  (Slaney and Thiébaux 2001, p. 146). As outlined in algorithm *PERFECT* (ibid. pp. 136–138), finding the optimal action amounts to finding a *minimal size hitting set* for all deadlocks, which in this case is the set  $\{11\}$ . Therefore, `move(11, table)` is the optimal action to take.<sup>15</sup> From the next state, advancing to the goal state is a matter of applying eight constructive moves. Knowing the optimal plan length, we can compute the optimal value function as  $v_*(s_0) = q_*(s_0, \text{move}(11, \text{table})) = 91$ . But `move(badTop, table)` covers `move(9, table)` as a second action, which resolves only one of the two deadlocks, namely  $\{8, 11\}$ . An additional action is needed to resolve  $\{9, 11\}$  and  $q_*(s_0, \text{move}(9, \text{table})) = 90$ . Choosing `move(badTop, table)` as the next abstract action, both of these actions have the same chance of being chosen as the next concrete action, as laid out in Algorithm 3.1. Thus, the choice of which deadlocks to resolve is (at least in part) left to chance.

The abstraction function circumvents the major source of complexity for the blocks world problem but, as a trade-off, cannot represent the optimal policy. In any case, we cannot expect the learned abstract policy to represent optimal solutions to the blocks world planning problem in general.

Note however that the above example cannot occur in the state space of our RMDP specification. There, the goal is to stack all blocks in order, so for any state there can only be one goal-relevant tower and therefore only one concrete action covered by `move(badTop, table)` for both `r1` and `r2`. So, we have reason to believe that an optimal policy can be represented for the chosen goal state in particular. Of course, a formal proof is needed to confirm this.

Another characteristic of the proposed abstraction function is the unequal partitioning of concrete actions. For example, consider a concrete state  $s$  associated with `r1`. Its admissible concrete actions  $A_s$  are partitioned across two blocks represented by `move(next, top)` and `gutterAction`. The former block is always singleton while the latter contains all other actions. So, if the abstract action is chosen according to a uniform distribution (e.g. by an  $\epsilon$ -greedy policy or at the start of the learning process when  $q$ -values are equal), there is a  $\frac{1}{2}$  chance of choosing the constructive concrete action and a  $\frac{1}{2(|A_s|-1)}$  chance of choosing any other action. Similar examples can be given for the other abstract states. Therefore, the abstraction function introduces a strong exploration bias. Still, every action has a non-zero chance of being selected by the abstract  $\epsilon$ -greedy policy.

<sup>15</sup>Actually, when applying the ASP-based planner in (Datler 2020), one can find `move(11, 3)` as a second optimal action. It also resolves the deadlocks but without moving block 3 to the table. This action is covered by `move(badTop, other)`, and therefore aggregated with other suboptimal actions.

## 4.3 Empirical Evaluation

Having established the RMDP and our proposed abstraction function, we now turn towards our empirical results in an attempt to understand the effects of the chosen abstraction function on abstract  $Q$ -learning. As discussed in section 2.5, the promise of state-action pair abstraction is to make RMDPs with large state-action spaces tractable by reducing both the space and sample requirements of the learning process, with coarser abstractions resulting in larger reductions. But the coarseness of an abstraction is also associated with a cost. Used with abstract  $Q$ -learning in particular, convergence is not guaranteed in general and, if convergence occurs nevertheless, there is also no guarantee that the learned abstract policy is optimal. To understand the effects of our abstraction, we therefore need to study three constructs, namely (1) the observed sample efficiency, (2) the stability of the learning process, and (3) the quality of the learned policy. This leads to our research questions, formulated in comparison with the concrete RMDP representation (as control condition) and limited in scope to the specified RMDP.

**Q1** Does the proposed abstraction cause differences in sample efficiency?

**Q2** Does the proposed abstraction cause stability issues during the learning process?

**Q3** Does the proposed abstraction cause differences in quality of the learned policy?

In the rest of this section, we present and discuss the results of three experiments. The first experiment investigates Q1 in the 5-blocks world RMDP with various parameter configurations. In the second, one particular parameter configuration is picked to answer Q2 and Q3. The third experiment examines all three research questions in the 20-blocks world RMDP, to see how the previous results generalise for larger blocks worlds.

### 4.3.1 Metrics and Operationalisation

Having discussed the intuition of our research questions, we next need to operationalise them. Let's consider a single realisation of the learning process, consisting of a series  $h_1, h_2, \dots$  of episodes (or histories). The quality of the learned policy for each episode  $h_i$  is directly measurable in terms of the realised *return*  $G_0(h_i)$  at time point zero, which can be computed according to Definition 2.3.1 once the episode has ended.<sup>16</sup> For brevity's sake, we refer to  $G_0(h_i)$  as just the return of  $h_i$ . To establish a baseline, we consider the worst-case performance of a hypothetical *naive* unstack-stack policy in an  $n$ -blocks world. Such a policy needs to move at most  $n - 1$  blocks in order to reach a state in which every block is on the table and at most another  $n - 1$  blocks to stack the blocks in the desired configuration. The total number of actions of this algorithm results therefore in a worst-case return of  $100 - 2(n - 1)$ . This serves as a loose bound for tracking the

<sup>16</sup>Note that we are working with finite episodes here, which may be artificially cut off by a time limit. So, the realised "return" which we measure does not necessarily correspond with the expected return in particular in cases where it is negative.

success of the learned policy. We call an episode  $h_i$  *successful* if  $G_0(h_i) \geq 100 - 2(n - 1)$ . Of course, with the threshold only representing the worst case, surpassing it does not generally imply a policy of better quality. Not surpassing it however is a clear sign of poor quality.

By definition, the learning process converges if the quality of the learned policy increases monotonically. Stability issues are therefore present if the quality of the learned policy decreases at any point during the learning process. We study stability by observing the return of episodes across the learning process. If no stability issues are present, we expect the return to converge to a range of positive values (depending on the starting state) as learning progresses. Note however that this method does not allow us to identify chattering phenomena in the abstract state-action value function (Li, T. J. Walsh, and Littman 2006), especially if they are too small to change the greedy policy.

In order to detect differences in sample efficiency, we record the *normalised cumulative return* (NCR), defined by the formula  $NCR(h_i) \doteq \frac{1}{i} \cdot \sum_{j=1}^i G_0(h_j)$ . Intuitively, this metric computes the average return across all episodes realised so far. As a second metric, we count the total number of successful episodes. Both metrics correlate with sample efficiency: All else equal, higher sample efficiency leads to higher returns earlier in the learning process, therefore increasing the normalised cumulative return. Note however that there is also a correlation between the specified metrics and solution quality, as well as a negative correlation with stability: For example, a "slow" learning process achieving very high returns late in the learning process can achieve higher cumulative returns over time than a "faster" learning process, achieving low returns early but never achieving the high returns of the former process. Also, a "fast" learning process with stability issues might achieve a lower cumulative return over time than a "slow" learning process with no stability issues. To establish a causal relationship between differences in sample efficiency and the used metrics, we need to rule out such alternative explanations.

### 4.3.2 Experiment Setup and Control

We use Algorithm 3.1 as implementation of abstract  $Q$ -learning and Algorithm 3.3 to compute the abstraction function. Algorithm 2.1 is used as implementation of concrete  $Q$ -learning. For all experiments, the initial  $q$ -values were set to  $-1$  and the time limit was set to  $3n + 1$  for an  $n$ -blocks world. The learning rate  $\alpha$  and the exploration factor  $\epsilon$  were varied such that  $\alpha \in \{0.001, 0.003, 0.005, 0.01, 0.03, 0.05, 0.1, 0.3, 0.5\}$  and  $\epsilon \in \{0.05, 0.1, 0.2, 0.3\}$ . For every configuration, the experiment was repeated 20 times. For every repetition, the learning process was stopped after realising the first 3000 episodes.

To summarise the collected data, we use the following statistical descriptions:

**Definition 4.3.1** (Quantiles, median, interquartile range: Eid, Gollwitzer, and Schmitt 2013, p. 111-113).

- $Q_1$ , the *first quantile*, is a value such that at least 25% of the samples in the data set are below or equal to  $Q_1$  and at least 75% of the samples are above or equal to  $Q_1$ .
- $Q_2$ , the *median* or *second quantile*, is a value such that at least 50% of the samples in the data set are below or equal to  $Q_2$  and at least 50% of the samples are above or equal to  $Q_2$ .
- $Q_3$ , the *third quantile*, is a value such that at least 75% of the samples in the data set are below or equal to  $Q_3$  and at least 25% of the samples are above or equal to  $Q_3$ .
- The *interquartile range (IQR)* is the range of values between  $Q_1$  and  $Q_3$ .

The algorithms were implemented in *Python 3*<sup>17</sup>, and *clingo 5.5*<sup>18</sup>. The full source code can be found online.<sup>19</sup> The experiments in the 5-blocks world were run on a cluster with 13 nodes, each having two CPUs of type Intel Xeon E5-2650 v4. The experiments in the 20-blocks world were run on a desktop PC with one AMD Ryzen 5 Pro 4650G CPU and 16GB memory.

### 4.3.3 Experiment 1 - Results

For the first experiment, a parameter study was conducted in the 5-blocks world. The results are presented in Figure 4.2. Focusing on the normalised cumulative return (NCR), we can observe a clear difference between abstract and concrete  $Q$ -learning, as shown in plots (a) and (b), respectively. The median NCR for the abstract case is consistently higher for almost all configurations (except when  $\alpha = 0.5$ ), with interquartile ranges being far apart. Observing the influence of parameters in the abstract case, all configurations achieve a similar median NCR when  $\alpha \leq 0.1$ , indicating small effects (if any) of both  $\alpha$  and  $\epsilon$ . However, a variation in interquartile range is observable as  $\alpha$  changes. For  $\alpha \geq 0.3$ , an interaction effect between  $\alpha$  and  $\epsilon$  is indicated, with increases in  $\alpha$  and  $\epsilon$  suggesting a decrease in NCR. In the concrete case, influences of  $\alpha$  and  $\epsilon$  on NCR can be observed. For  $\alpha \leq 0.05$ , an increase in  $\alpha$  can be associated with an increase in NCR. For  $\alpha \geq 0.05$ , increases in  $\alpha$  are associated with little (if any) change in median NCR. Also, an inverse correlation can be observed between  $\epsilon$  and the median NCR, with lower values of  $\epsilon$  generally leading to higher values of median NCR (except when  $\alpha = 0.001$ ).

Focusing on the number of successful episodes as shown in plots (c) and (d), abstract  $Q$ -learning achieves higher median values than concrete  $Q$ -learning for every configuration, with interquartile ranges being far apart. Regarding the effect of  $\alpha$  on abstract  $Q$ -learning, no clear trend can be spotted for  $\alpha \leq 0.1$  due to the similarity of median values and high interquartile ranges. However, values of  $\alpha \geq 0.3$  can generally be associated with a

<sup>17</sup><https://www.python.org/>

<sup>18</sup><https://potassco.org/clingo/>

<sup>19</sup><https://github.com/rbankosegger/RLASP-core>.

lower number of successful episodes. A clearer trend on abstract  $Q$ -learning is suggested for  $\epsilon$ , with lower values of  $\epsilon$  generally leading to higher numbers of successful episodes. For concrete  $Q$ -learning, the effect of  $\alpha$  is as follows: For  $\alpha \leq 0.03$ , higher values of  $\alpha$  can be associated with an increase in the number of successful episodes. For  $\alpha \geq 0.1$ , higher values of  $\alpha$  can be associated with a decrease in the number of successful episodes. Therefore, a peak is present between  $0.03 \leq \alpha \leq 0.1$ . For  $\epsilon$ , an inverse correlation is suggested once again.

Our main results can be summarised as follows:

- R1 All else equal, a change in representation from the concrete to the proposed abstract one can be associated with an increase in both the normalised cumulative return and the number of successful episodes.
- R2 For concrete  $Q$ -learning, a decrease in  $\epsilon$  (within the explored range) can generally be associated with an increase in both metrics. The same is true for the number of successful episodes in abstract  $Q$ -learning.
- R3 The effect of  $\alpha$  is different for both algorithms and both metrics. Without more sophisticated statistical tools, no clear conclusions can be drawn about its relationship with the metrics in the abstract case.

#### 4.3.4 Experiment 1 - Interpretation

Most relevant to our research question is result R1. The increase in both the normalised cumulative return and the number of successful episodes when using abstract  $Q$ -learning can be explained by an increase in higher sample efficiency as caused by the abstraction. Recall however that both metrics also correlate with solution quality and possible stability issues. For concrete  $Q$ -learning the quality of the learned solution is limited by the choice of  $\alpha$ . But the same applies also to abstract  $Q$ -learning, in addition to the quality concerns introduced by the abstraction. Therefore, we are sure that the quality of the learned abstract policy is at most as good as the quality of the learned concrete policy and an increase in quality can be ruled out as an explanation of the increases in normalised cumulative return and number of successful episodes. To explain the difference with stability issues, we would have to assume that abstract  $Q$ -learning is more stable than concrete  $Q$ -learning, therefore achieving high returns more often. The stability of concrete  $Q$ -learning learning is again influenced by the choice of  $\alpha$ . But so is the stability of abstract  $Q$ -learning, in addition to the stability concerns introduced by the abstraction. We therefore deem it unlikely that abstract  $Q$ -learning is more stable than concrete  $Q$ -learning. To support these claims empirically, it is necessary to investigate the learning curves of the learning processes. This is addressed in the next two experiments.

Based on this argument, we conclude that using abstract  $Q$ -learning indeed causes an increase in sample efficiency, which in turn explains the increases in both normal cumulative return and the number of successful episodes. The causal relationship itself

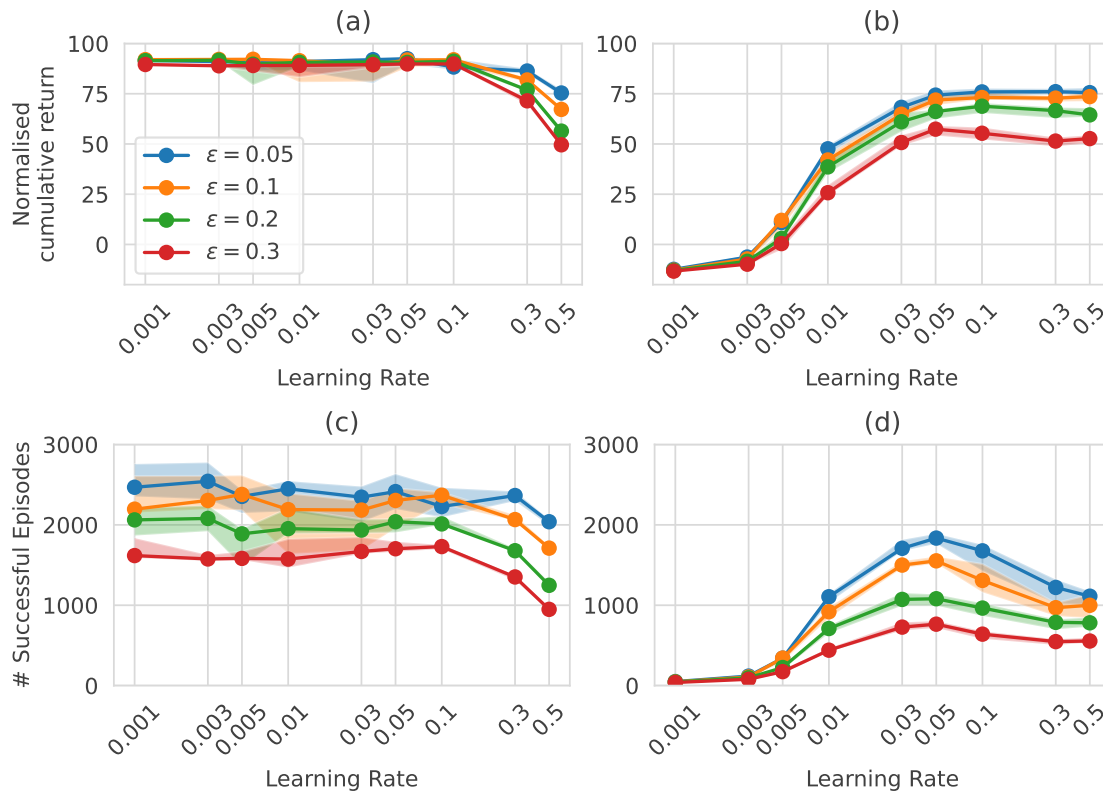


Figure 4.2: A parameter study in the 5-blocks world. Plot (a) shows the normalised cumulative return at episode 3000, as achieved by abstract  $Q$ -learning with various parameter configurations. Plot (b) shows the same for concrete  $Q$ -learning. Plot (c) shows the total number of successful episodes (out of 3000) for abstract  $Q$ -learning. Plot (d) shows the same for concrete  $Q$ -learning. For all plots, the dots show the median value across 20 repetitions. The shaded areas show the interquartile ranges (IQR).

can be explained by the state-action space reduction and the exploration bias, both properties of the proposed abstraction function. To see the extent to which either of these factors contributes to the causal relationship, more experiments are needed. How well does this conclusion generalise? For now, we have to restrict it to the 5-blocks world RMDP as presented, the chosen abstraction function and the specified values for the initialisation of  $q$ -values and the time limit. The conclusion does however generalise for a reasonable range of choices for  $\alpha$  and  $\epsilon$ .

To properly explain results R2 and R3, more experimentation is needed. We can however speculate to identify some plausible explanations: The inverse correlation between  $\epsilon$  and both metrics may (at least in part) be due to a decrease in quality of the learned solution, which in turn is caused by the random explorative actions becoming more frequent as  $\epsilon$  increases. Regarding the effects of varying  $\alpha$  in concrete  $Q$ -learning, the following explanations come to mind: As  $\alpha$  decreases, more experience is needed to learn the

optimal policy. Therefore, the sample efficiency decreases and both the and the number of successful episodes decrease as a result. If  $\alpha$  is chosen to be too large, stability issues may be introduced, lowering the observed number of successful episodes. This effect may be less noticeable in the NCR if these issues cause episodes to achieve returns just slightly below the threshold of "success".

#### 4.3.5 Experiment 2 - Results

The next experiment aims to investigate the behaviour of abstract  $Q$ -learning in terms of the quality of its learned solution and possible stability issues, with results presented in Figure 4.3. The experiment was conducted in the 5-blocks world with the parameters set to  $\alpha = 0.05$  and  $\epsilon = 0.05$ , the best configuration found for concrete  $Q$ -learning in the previous experiment. The baseline (i.e. the worst-case return of the naive unstack-stack policy) is 92 in the 5-blocks world.

Let's start with investigating the learning curves shown in plot (a). For abstract  $Q$ -learning, returns start out at  $-16$ , the first positive median return being recorded after episode 10. Values above the baseline are first recorded at episode 14. After the initial jump, the median return stays consistently above a value of 90 as the learning process continues. The interquartile range is too small to be noticed in the plot. For concrete  $Q$ -learning, a positive median return is first reached at episode 440. After episode 1038, the median is consistently above 90. The interquartile range is noticeable around episode 500 but becomes diminishingly small after episode 1000. Considering the normalised cumulative returns across episodes as shown in plot (b), a significant difference between the concrete and abstract representations can be observed across all episodes except for the first few, with interquartile ranges being far apart. At episode 3000, a median NCR of 92.4 ( $Q_1 = 89.6$ ,  $Q_3 = 93.1$ ) is achieved by abstract  $Q$ -learning and a median NCR of 74.3 ( $Q_1 = 72.3$ ,  $Q_3 = 76.6$ ) is achieved by concrete  $Q$ -learning. These values are also observable in the previous experiment.

Next, let's look at the distribution of returns between episodes 201 and 300 shown in plot (c). For concrete  $Q$ -learning, 1842 episodes yield a negative return. These are 92.1% out of a total of 100 considered episodes with 20 repetitions each. For abstract  $Q$ -learning only 109 (5.5%) episodes yield a negative return, with a median return of 93 ( $Q_1 = 92$ ,  $Q_3 = 94$ ). Finally, consider the distribution of returns between episodes 2901 and 3000 shown in plot (d). For concrete  $Q$ -learning, no negative returns are recorded. The median return is at 93 ( $Q_1 = 92$ ,  $Q_3 = 94$ ). For abstract  $Q$ -learning, 9 negative returns (0.5%) are recorded. The median return is at 93 ( $Q_1 = 92$ ,  $Q_3 = 94$ ). Comparing the distribution of returns for the two configurations, a slight shift towards higher values can be observed for abstract  $Q$ -learning. This is however not enough to deduce any significant difference, for which a more sophisticated statistical analysis is necessary. Comparing the two distributions of returns for abstract  $Q$ -learning (for episodes 201 to 300 and episodes 2901 to 3000, respectively), the later distribution is slightly skewed towards higher values and the number of episodes with negative returns is reduced from

109 (5.5%) to 9 (0.5%). But again, one needs to be cautious about concluding significant differences without a statistical test.

To summarise the main observations:

1. For both configurations, median returns are consistently above 90 for episodes 1038–3000.
2. There is a clear difference in the distribution of returns for episodes 201–300 between abstract and concrete  $Q$ -learning, with more than 75% of episodes achieving returns above the "naive" baseline but 5.5% of episodes yielding negative returns in the abstract case.
3. Without a formal analysis, no difference in the distribution of returns for episodes 2901–3000 can be concluded between abstract and concrete  $Q$ -learning. In both cases, more than 75% of episodes are above the "naive" baseline.
4. Without a formal analysis, no difference in the distribution of returns between episodes 201–300 and 2901–3000 can be concluded for abstract  $Q$ -learning. There is however a notable downward trend in the percentage of episodes with negative returns, which decreases from 5.5% to 0.5%.

#### 4.3.6 Experiment 2 - Interpretation

In the examined configurations, what can be deduced about the quality of found solutions for abstract and concrete  $Q$ -learning? To answer this question, we consult the results concerning the distribution of returns for episodes 2900–3000. To start with, no significant difference in distribution of the return was detected between the two cases. Therefore, we conclude that, after 3000 episodes, both processes learned policies of comparable quality. For both algorithms, more than 75% of episodes achieved returns above the baseline. Therefore, both policies are able to achieve returns comparable to those of a naive unstack-stack algorithm. A significant part of the episodes is however below the baseline, indicating that the learned policies are not as consistent as a naive algorithm might be.

Regarding the consistency issues, we offer the following possible explanations, to be investigated in future experiments: First, some decrease in returns might be caused by the random explorative actions of the  $\epsilon$ -greedy strategy. Second, it is possible that the consistency of the policies is improved when the learning processes are continued beyond beyond the 3000 episodes considered here.

Next we discuss the possible occurrence of stability issues, in particular for abstract  $Q$ -learning. After a large jump within the first few episodes, the returns for abstract  $Q$ -learning stay consistently high, with no visible fluctuations in either the median nor the interquartile range. Therefore, if stability issues are present, we conclude that the majority of episodes realised by abstract  $Q$ -learning are unaffected. Still, it might be the

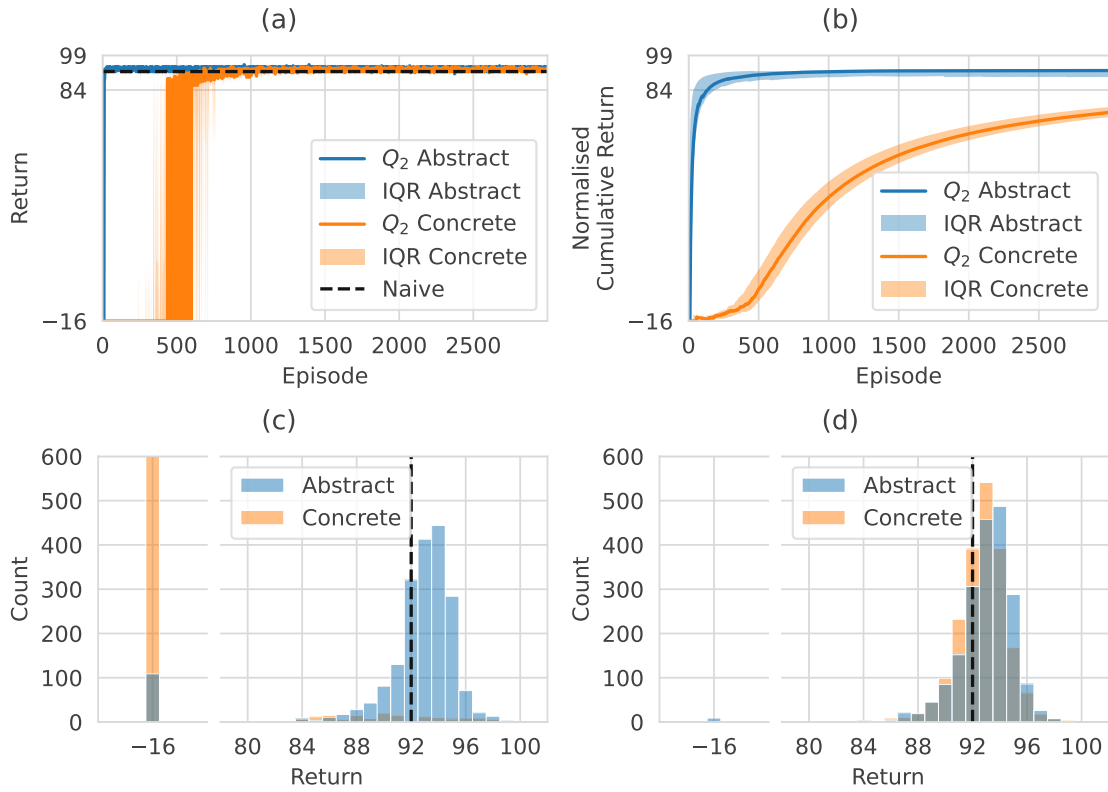


Figure 4.3: Comparison of the behaviour policies for concrete and abstract  $Q$ -learning in the 5-blocks world ( $\alpha = 0.05$ ,  $\epsilon = 0.05$  with 20 repetitions per configuration). In (a), the learning curves for both abstract and concrete representations of the RMDP are shown in terms of the median return  $Q_2$  and its interquartile range. The dashed line shows the worst-case return of a naive unstack-stack policy. In (b), the normalised cumulative return is shown in terms of its median and interquartile range. In (c), a histogram shows the distribution of the return of episodes 201 to 300 for both abstract and concrete  $Q$ -learning. Having 20 repetitions over 100 episodes gives a total count of 2000 per configuration. In (d), a histogram shows the distribution of the return of episodes 2901 to 3000.

case that stability issues are present in a minority (i.e. less than 25%) of experiment repetitions. Also note that after 2900 episodes of learning, there are still some episodes with negative returns, which cannot be explained through explorative actions (otherwise they would also occur in the concrete case). The observed downward trend suggests however that the number of episodes with negative returns tends to zero as learning progress beyond 3000 episodes.

In terms of sample efficiency, the results concur with those of the previous experiment. Abstract  $Q$ -learning clearly achieves high returns faster than concrete  $Q$ -learning, which manifests in the normalised cumulative return.

To conclude, the experiment strengthens our previous conclusions about the relationship between abstraction and sample efficiency. In addition, the quality of the learned abstract policy was found to be no worse than that of the learned concrete policy, at least up to episode 3000. No stability issues were detected. Note however that these conclusions do not necessarily generalise to other parameter configurations.

### 4.3.7 Experiment 3 - Results

In the third experiment, we aim to test whether previous results generalise to larger blocks worlds. In particular, we investigate the behaviour of abstract  $Q$ -learning in a 20-blocks world RDMP. The results are shown in Figure 4.4. Regarding concrete  $Q$ -learning, note that no positive returns were observed within the first 3000 episodes and for a variety of parameter configurations. The baseline (i.e. the worst-case return of the naive unstack-stack policy) is 62 in the 20-blocks world.

First, we consider the parameter configuration  $\alpha = 0.01$ ,  $\epsilon = 0.05$ , with the learning curve shown in plot (a). The first positive median return is observed at episode 376, with median returns being consistently above 62 after episode 1543. After episode 2000, there are only 10 instances for which  $Q_1$  drops below the baseline of 62, and none below 60. Considering the same configuration, plot (b) shows the distribution of returns for episodes 2901–3000. Within this range, 49 episodes (2.5%) are recorded with negative returns and a total of 194 episodes (9.7%) are recorded below the baseline. The median return is 67 ( $Q_1 = 65$ ,  $Q_3 = 69$ ).

The parameter configuration  $\alpha = 0.03$  and  $\epsilon = 0.05$ , as depicted in plot (c), achieves first positive median returns after episode 261. The median return is consistently above the baseline for episodes 500–1397 and 2335–3000. In between these two ranges however, the median drops below zero at multiple stages. The parameter configuration of  $\alpha = 0.005$ ,  $\epsilon = 0.05$ , as depicted in plot (d), achieves first positive median returns only after episode 1292. Median returns above the baseline are achieved after episode 1586, but never consistently within the range of 3000 episodes. The parameter configuration of  $\alpha = 0.01$ ,  $\epsilon = 0.1$  as depicted in plot (e), achieves first positive median returns after episode 411. The median return never drops below 59 after episode 924 and stays consistently above the baseline for episodes 1390–2187. The first quartile however does not achieve consistent positive values, dropping back to zero at multiple stages and for an extended range after episode 2888.

### 4.3.8 Experiment 3 - Interpretation

Abstract  $Q$ -learning with parameters set to  $\alpha = 0.01$ ,  $\epsilon = 0.05$  yields median returns that are well above the worst-case return of the naive unstack-stack policy from episode 1543 onward. Within the range of episodes from 2901–3000, below one tenth of episodes yield a return worse than the naive policy. Therefore we conclude that a policy of acceptable quality (although not necessarily the optimal one) was successfully learned after 3000 episodes. In addition, given the presented plots, no stability issues can be detected.

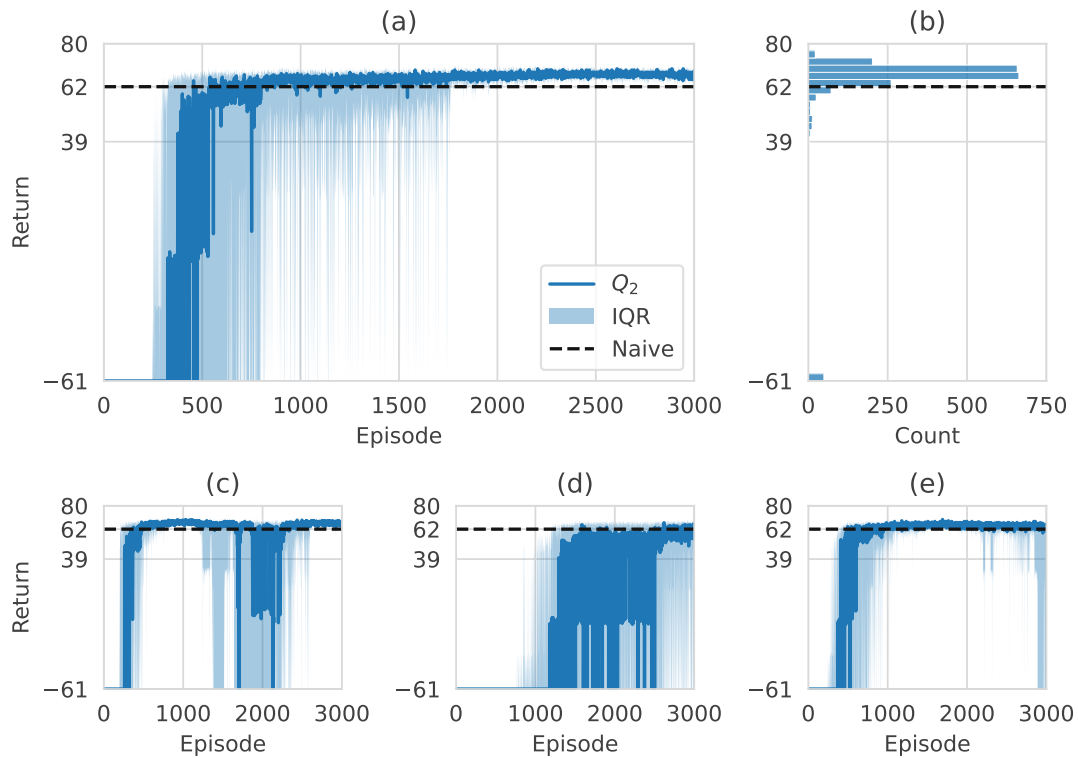


Figure 4.4: Abstract  $Q$ -learning in the 20-blocks world. Plots (a), (c), (d), (e) show learning curves with percentiles ( $n = 20$ ) for the following parameter configurations: (a)  $\alpha = 0.01$ ,  $\epsilon = 0.05$ , (c)  $\alpha = 0.03$ ,  $\epsilon = 0.05$ , (d)  $\alpha = 0.005$ ,  $\epsilon = 0.05$ , and (e)  $\alpha = 0.01$ ,  $\epsilon = 0.1$ . Plot (b) shows the last 100 episodes of the trials in (a) as histogram. Having 20 repetitions per episode gives a total count of 2000. The dotted lines show the worst-case return achieved by a naive stack-unstack policy.

Clearly, abstract  $Q$ -learning causes a decrease in sample efficiency compared to concrete  $Q$ -learning, which was not able to find positive rewards in the given configuration of parameters.

These results however do not generalise to other parameter configurations. Even slight variations on the parameters have a significant effect on the learning process, as demonstrated in plots (c), (d), and (e) of Figure 4.4. For plot (c), the learning rate was increased to  $\alpha = 0.03$ . As a result, stability issues are detected in the learning curve. For plot (d), the learning rate was decreased to  $\alpha = 0.005$ . By episode 3000, the learning process has not yet produced an acceptable solution. Consistent returns above the baseline of 62 are not yet achieved by the learned abstract policy. For plot (e), the exploration factor was increased to  $\epsilon = 0.1$ . Again, stability issues are detected when looking at the first quartile of the return.

Looking at the progression of learning rates from 0.005 over 0.01 to 0.03 (respectively plots (d) (a) and (c)), a trend can be spotted with respect to stability issues and sample efficiency: As  $\alpha$  increases, sample efficiency decreases but stability issues become more severe. To get the best of both worlds, employing a managed (i.e. non-constant) learning rate seems to be a fruitful idea.

To summarise, the experiment demonstrates the existence of a parameter configuration for which abstract  $Q$ -learning, within 3000 episodes, is able to produce a policy of acceptable quality in the 20-blocks world. The parameters need to be carefully chosen though, as small deviations can already effect both the sample efficiency and the stability of the learning process.

## 4.4 Discussion

In this chapter, we proposed and evaluated an ASP-based abstraction function for the blocks world planning problem. For evaluation, we applied concrete and abstract versions of  $Q$ -learning with an  $\epsilon$ -greedy behaviour policy in blocks worlds with 5 and 20 blocks. Experiments were performed with various different parameter configurations. Particular attention has been paid to the effects of the learning rate  $\alpha$  and the exploration factor  $\epsilon$ . A distinct subset of blocks world planning tasks was investigated, with random initial states and a fixed goal state, in which all blocks need to be stacked in one particular order. So, we cannot claim any generalisability of the presented results for other parameter configurations and blocks world planning problems beyond the investigated subset. We answer the posed research questions as follows.

### Q1 - Does the proposed abstraction cause differences in sample efficiency?

In the 5-blocks world we concluded that using abstract  $Q$ -learning with the proposed abstraction causes an increase in sample efficiency, compared to concrete  $Q$ -learning. In the 20-blocks world, where concrete  $Q$ -learning failed to achieve any positive returns over a span of 3000 episodes, learning an abstract policy of acceptable quality within the same number of episodes was possible. These results are consistent with the discussed theory of abstraction, claiming increased sample efficiency as a consequence of a smaller abstract state-action space. In addition, we identified a large exploration bias built into our abstraction function as an additional plausible explanation for the observed decrease in sample efficiency. The extent to which each of these factors contribute to sample efficiency remains to be investigated.

### Q2 - Does the proposed abstraction cause stability issues during the learning process?

In both the 5-blocks world and the 20-blocks world, parameter configurations have been found with no detected stability issues in the first 3000 episodes. For different parameter configurations, stability issues were detected in the 20-blocks world. So, using the

proposed abstraction function can indeed cause stability issues depending on the choice of parameters. The effects of  $\alpha$  and  $\epsilon$  on stability were not systematically investigated. We did however spot a trend for  $\alpha$ , suggesting less stability issues for smaller learning rates. Changes in  $\epsilon$  can also effect stability, although no general trends were identified.

#### **Q3 - Does the proposed abstraction cause differences in quality of the learned policy?**

In the 5-blocks world, no significant differences in quality were found between the abstract and concrete policies after 3000 episodes. In the 20-blocks world no comparison was done, due to concrete  $Q$ -learning being intractable.

The returns achieved in episodes 2900 to 3000 were also compared to a baseline, chosen to be the worst-case return of a naive unstack-stack policy. Under certain parameter configurations, this baseline was surpassed by a majority of episodes within the given interval, for both concrete and abstract  $Q$ -learning in the 5-blocks world and for abstract  $Q$ -learning in the 20-blocks world. In all configurations, the baseline was not surpassed consistently, which might be in part due to explorative random actions taken by the  $\epsilon$ -greedy policy and/or because more experience is needed to further improve the policies.

## Case Study: Minigrid

In this case study, we investigate the effects of state abstraction in the *Minigrid* simulation library (Chevalier-Boisvert et al. 2023), which provides a suite of episodic, *grid world* type task environments. The agent is situated in a two-dimensional grid of tiles, along with a set of objects, such as walls, doors, and keys. Most objects have additional properties, such as a colour or an internal state. For example, the internal state of a door can be open, closed or locked. Solving a Minigrid task environment typically requires the agent to navigate through a fixed layout of connected, rectangular rooms in order to reach the location of a goal tile as fast as possible. Along the way, the agent is tasked with solving simple puzzles and with avoiding hazards. By default, all Minigrid task environments have the same set of actions, which allow the agent to take a step forward, turn left and right, pick up and drop items, and to toggle the internal state of some objects. The effects of the agent’s actions usually depend on and influence only the agent itself and the object that is directly in front of the agent. As all task environments are constructed from the same building blocks, these effects stay consistent in the Minigrid setting.

We focus on task environments belonging to the families of *Door Key*, *Multi Room*, and *Four Rooms* (see Farama Foundation 2023b for the documentation and Farama Foundation 2024 for the source code).

- In Multi Room environments, the room layout consists of a series of two or more rectangular rooms, connected by closed but unlocked doors. An example initial state is presented in Figure 5.1(b). The agent is placed on one end of this series of rooms, the goal tile on the other. The doors are initially closed but require no key to open.
- In Door Key environments, the room layout consists of two rooms, connected by a locked door. An example initial state is presented in Figure 5.1(a). The agent and a key are located in one room, the goal tile in the other. To reach the goal tile,

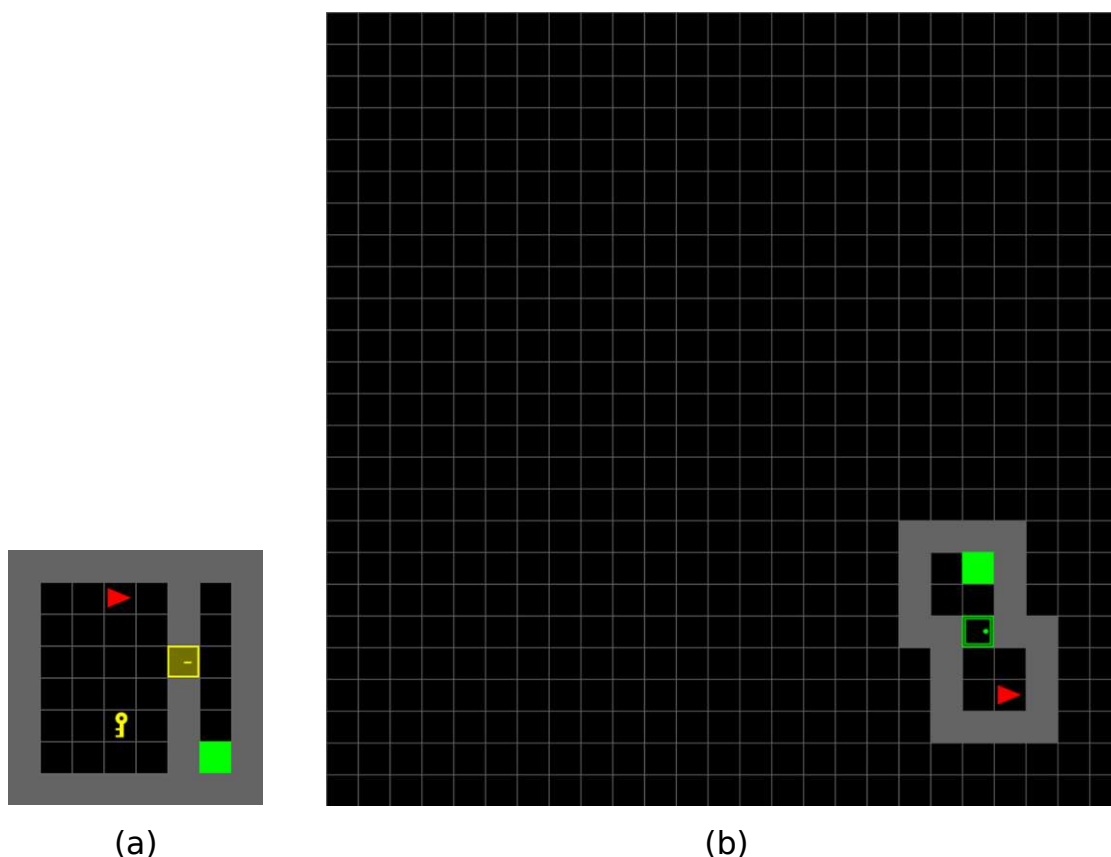


Figure 5.1: Examples of Minigrid states. In (a), an initial state of the MiniGrid-DoorKey-8x8-v0 task environment is shown. The grey squares mark walls. The red triangle marks the location of the agent and its facing. The green square marks a goal tile. In order to reach it, the yellow door has to be unlocked using the yellow key. In (b), an initial state of the MiniGrid-MultiRoom-N2-S4-v0 task environment is shown. The green door is unlocked, requiring no key to open.

the agent must navigate to the key, pick it up, navigate to the door, toggle it, and finally navigate to the goal itself.

- In the Four Rooms environment, the room layout consists of four square rooms that are arranged in two rows and two columns. Adjacent rooms are connected through gaps in the walls. An example initial state is presented in Figure 5.2(a).

For our treatment of these task environments, we assume that the agent can fully observe all objects on the grid, with the same information as in the top-down views of the presented figures. When determining the initial state for a new episode in any of these environments, many aspects of the state are randomised. This includes the location and orientation of the agent, the locations of other movable objects such as keys, and even

the room layout as determined by the locations and colours of walls, doors and other immovable objects. Constraints are placed on the randomisation procedure to ensure that the essence of the task stays the same. Let's consider the Multi Room task environment as an example. The room layout from the sampled initial state will always consist of a series of connected rooms and the following is randomly determined: the location of these rooms on the grid, their size, through which walls they are connected, the location and colour of the doors, the location and orientation of the agent in the first room, and the location of the goal tile in the last room.

Grid worlds suffer from the state space explosion problem in principle, which motivates the use of abstraction techniques.<sup>1</sup> For a Minigrad task environment in particular, the possible combinations of room layouts, the possible locations and orientations of the agent, and the possible states and locations of other objects contribute to the size of the state space, which often is considerably large. To illustrate, consider the Multi Room environment from Figure 5.1(b). By horizontal and vertical translation of the shown layout within the bounds of the grid, we can generate  $21 \cdot 19 = 399$  different room layouts. Also considering the six possible door colours and the four possible locations for the goal tile raises this number to 9586. As described later in this chapter, more layouts can be generated when also considering the different ways in which the two rooms can attach to each other. Further, the possible locations and orientations of the agent, as well as the state of the door need to be considered for each layout in order to count the state space. The example also shows the potential of applying abstraction functions in the Minigrad setting. A significant state space reduction without any loss in task-relevant information can be achieved simply by omitting redundant information and by exploiting superficial symmetries.<sup>2</sup> In the example, the door colour can be omitted. Further, all horizontal and vertical translations of the layout from Figure 5.1(b) can be grouped into one abstract state by considering grid coordinates relative to the agent's location.

In this chapter, we develop two variants of an ASP-based, CARCASS-style state abstraction function and test them in experiments featuring Door Key, Multi Room, and Four Rooms task environments. To this end, we present a relational state representation that can be applied to any task environment in the Minigrad setting. We then identify a set of assumptions that hold for a range of Minigrad task environments. In short, we make the following assumptions:

- A1 We assume that the environments are all constructed from the same building blocks with common actions having effects that are localised and consistent. Therefore, we can expect the results of interactions with objects to be consistent across the three task environments.

---

<sup>1</sup>Without constraints on the layout, there are  $\binom{w \cdot h}{k+1}$  possibilities to place  $k$  objects plus the agent on a  $w \times h$  grid without overlap. Multiply this with the number of possible combinations for the agent's orientation and the possible values for each object's internal state and colour.

<sup>2</sup>We use the term *symmetries* in the sense of Ravindran 2004.

A2 We further assume a common structure for the room layouts. Every encountered room layout is assumed to consist of a set of rectangular rooms that are connected via gaps in the room boundaries. Finally,

A3 we assume a common task structure in which the main task of a given task environment can be broken down into an ordered set of subtasks.

Based on these assumptions, we develop ASP-based background knowledge and use it to build an ASP-based, CARCASS style state abstraction function. At decision time the domain knowledge is used in conjunction with the current state description to infer a single subtask that needs to be solved next. The abstract state is constructed such that it contains only information relevant to this subtask. In one variant of this abstraction function, the set of admissible actions is preserved. In the other variant, the available domain knowledge is used to restrict the set of admissible actions such that actions with no effect on the environment (e.g. trying to move forward while fronting a wall) are removed. Within the theoretical framework discussed in this thesis, there are no theoretical guarantees of convergence for the proposed abstractions.

To investigate the effect of abstract  $Q$ -learning under the presented abstraction functions, we present exploratory empirical results, aimed at answering the following research questions:

**Q1** Do the proposed abstractions cause differences in sample efficiency?

**Q2** Do the proposed abstractions cause stability issues during the learning process?

**Q3** Do the proposed abstractions cause differences in quality of the learned policy?

The chapter is structured as follows. We start by discussing the Minigrid setting in more detail. We then present our relational state description for the setting, the developed domain knowledge, and the abstraction function. Finally, we show the results of our empirical research in the form of three small experiments and close with a discussion of the results.

## 5.1 The Minigrid setting

In this section, we introduce the Minigrid setting (Chevalier-Boisvert et al. 2023) and highlight the aspects which are relevant to our work. For further details we refer to the public documentation (Farama Foundation 2023b) and the source code (Farama Foundation 2024).

A state in a Minigrid task environment can be described as a collection of objects, each having a location in a two-dimensional, discrete coordinate system  $\{0, 1, \dots, w - 1\} \times \{0, 1, \dots, h - 1\}$  for  $w, h \in \mathbb{N}$ . Such objects can be *goal tiles*, *lava tiles*, *walls*, *doors*, *keys*, *floors*, *balls* and *boxes*. In addition to their location, some objects have an internal state.

Doors can be *open*, *closed* and *locked*. Doors, walls, floors, balls, keys and boxes also have one of six colours (*red*, *green*, *blue*, *purple*, *yellow*, *grey*). The agent is represented on the grid as well, with a location, an orientation of *north*, *south*, *east*, or *west*, and an inventory to carry one object. We say that the agent is *fronting* a location if the agent is directly adjacent to and oriented towards it. We say that the agent is *fronting* an object if that object is in the location that the agent is fronting. For our experiments we assume full observability of all objects, their location on the grid and their internal state.<sup>3</sup>

The action space is defined as the set { **left**, **right**, **forward**, **pickup**, **drop**, **toggle**, **done** }. Although some actions are not needed to solve certain task environments, we assume all actions to be admissible in all states for all task environments.<sup>4</sup> Their effects are as follows. The actions **left** and **right** cause the agent to change its orientation. The action **forward** causes the agent to move forward one step into the fronted location. This happens only if the location to enter is either empty or contains an object which can overlap with the agent. Otherwise, for example if the agent is fronting a wall, the action has no effect. The action **pickup** allows the agent to pick up a fronted object. Only some objects, such as keys and balls, can be picked up and only one object can be carried at any given time point. If there is no appropriate fronted object to pick up or if the agent is already carrying another object, the action has no effect. The action **drop** allows the agent to drop the object it is currently carrying. If it carries no object, the action has no effect. The action **toggle** enables the agent to interact with fronted doors and boxes but has no effect if there is no appropriate object to toggle. A locked door can be opened via the **toggle** action, but only if the agent is carrying a key of the same colour as the door. The action **done** has no effect for the purposes of this thesis.<sup>5</sup>

Note how the effects of all of these actions depend on and influence only the contents of the location that the agent is fronting and the properties of the agent, i.e. its orientation, location, and inventory. We therefore say that the effects of these actions are *localised*.

The default task for a Minigrid environment is to navigate from the agent's initial location to the location of a goal tile on the grid.<sup>6</sup> This needs to be achieved within a finite

<sup>3</sup>Minigrid task environments are characterised by *partially observable Markov Decision Processes* (POMDP, Chevalier-Boisvert et al. 2023 citing Kaelbling, Littman, and Cassandra 1995) and do not generally assume full observability. By default, the agent can only observe its immediate surroundings, the  $7 \times 7$  area of the grid directly in front of it. The agent's line of sight is further blocked by walls and closed doors. For our experiments, we use the *Fully Obs Wrapper* (Farama Foundation 2023b), which provides full observability of the grid, similar to the illustrations in Figure 5.1. As discussed below, partial observability is still present in the reward structure and the termination conditions.

<sup>4</sup>In the documentation of the task environments, actions are labelled *unused* when they are not needed in the context of the given task (see e.g. Door Key, Multi Room, Farama Foundation 2023b). But in the source code, we found no coded restrictions of the action space (Farama Foundation 2024).

<sup>5</sup>This is the default behaviour for Minigrid environments. Only a few environments make use of the **done** action, see e.g. *Go To Object* (Farama Foundation 2023b).

<sup>6</sup>Other tasks are possible as well. See for example the *Go To Object* family of environments (Farama Foundation 2023b). In order to distinguish between different tasks, Minigrid states can be augmented with a natural language description of the current task, called the *mission space*. In our experiments, we are concerned only with the default task, which is why we can ignore the mission space in our state

sequence of time points bounded by a fixed time horizon  $t_{\max}$ . The discount rate is set to  $\gamma = 1$ . The rewards are zero for all time points except when a goal tile is entered by the agent. The amount of the received positive reward is proportional to the time needed to achieve that task. An optimal policy must therefore navigate the agent to the goal tile with a minimal number of actions. The reward is defined as follows:

$$R_t \doteq \begin{cases} 1 - 0.9 \cdot \frac{t}{t_{\max}} & \text{if } A_{t-1} \text{ is a } \mathbf{forward} \text{ action, moving the agent onto a goal tile.} \\ 0 & \text{otherwise} \end{cases}$$

Minigrid environments have an episodic character in the spirit of Section 2.3.5. Each task environment is associated with a distinct set of initial states, which define the type of room layouts that can be encountered in that environment (i.e. the locations of walls, doors, goal tiles, etc.), as well as the locations of the agent and movable objects within. Episodes are terminated when the goal is reached or when the time limit is exceeded. In specific situations, some actions cause termination as well, for example when moving onto a lava tile. Note that both the rewards and the termination conditions for episodes depend on the number of time points spent in the episode. However, by default time is not an observable part of the state description. So, a Minigrid task environment with the presented configuration can not be modelled with the MDP framework.<sup>7</sup> Having discussed the basic building blocks of the Minigrid setting, we proceed to describe several families of task environments in detail.

### 5.1.1 Multi Room Environments

In environments belonging to the Multi Room Environments family, the room layout consists of a chain of rooms, divided by closed doors. To complete the task, the agent needs to navigate from one end of the chain to the other, opening doors along the way. The procedure for generating new initial states and their room layouts is the same for all task environments belonging to this family. The environments can differ in the grid dimensions  $w \times h$ , the number of rooms to place and the possible sizes of rooms. We focus on the `MiniGrid-MultiRoom-N2-S4-v0` environment, in which  $w = h = 25$ , the number of rooms to generate is 2, and the size of generated rooms (including walls) is always  $4 \times 4$ . For an example state, see Figure 5.1(b). The procedure for generating layouts with an arbitrary number of rooms of different sizes is quite involved. In the following, we sketch the essential parts of the procedure for layouts with two  $4 \times 4$  rooms. For further details

description.

<sup>7</sup>An EMDP characterisation in the sense of Section 2.3.5 is however possible by augmenting the state description with the number of time points spent in the current episode. Assuming that  $S$  is the state space as presented so far, we can add the time  $t$  such that  $s' = (s, t) \in S' = S \times \{0, 1, 2, \dots, t_{\max}\}$ . With the augmented state space  $S'$ , it is possible to define Markov rewards and terminal states as usual. Within the Minigrid framework, such an augmentation can be achieved using the *Time Aware Observation Wrapper* from Gymnasium (Farama Foundation 2023a, Observation Wrappers). From the POMDP perspective (Chevalier-Boisvert et al. 2023, p. 3),  $S'$  would be considered the *state space* and  $S$  would be considered the *observation space*, with an *observation function*  $\Omega : S' \rightarrow S$  defined such that  $\Omega((s, t)) = s$ . Note that  $\Omega$  can also be viewed as a state abstraction function in the sense of Section 2.5.

we refer to the source code (Farama Foundation 2024, `envs/multiroom.py`). To start, a  $w \times h$  grid is generated and the first room is placed at a random location within those bounds. Next, a door connecting the two rooms must be placed. The location of that door is randomly selected from the wall tiles of the first room which are not corners. A door of random colour is placed at that location. The second room is then attached to the first room such that they share the door and some other wall tiles. The exact location of the second room is determined randomly as well. During the process, it can happen that the second room is placed outside the bounds of the grid. If this is the case, the placement procedure for the door and the second room is repeated. If this occurs eight consecutive times, the placement procedure for the first room is repeated as well. To complete the room layout, the goal is placed in the second room, at a random location. Finally, the agent is placed in the first room, at a random location and orientation. Per default, the time horizon is set to  $t_{\max} = 40$ . The mission space string says "traverse the rooms to get to the goal".

### 5.1.2 Door Key Environments

In environments belonging to the Door Key Environments family, a wall with a locked door is placed as an obstacle between the agent and the goal coordinates. To complete the task, the agent needs to pick up a key and unlock the door with it. Particular task environments belonging to this family have varying grid dimensions of  $w \times h$ . Their initial states are randomly generated following a common procedure. The generation process determines the room layout, the initial location and rotation of the agent, as well as the initial location of the key. For illustrative purposes, consider the following algorithm sketch: Given the  $w \times h$  grid, a room is created by placing walls on all four grid borders, with a  $(w - 2) \times (h - 2)$  walkable area as the result. The goal tile is placed in the south-east corner of the room, at  $(w - 2, h - 2)$ . At a random  $x$ -coordinate with  $2 \leq x < w - 2$ , a vertical line of walls is placed such that the room is split in two. At a random  $y$ -coordinate with  $1 \leq y < h - 2$ , a locked, yellow door is inserted into that splitting wall. The agent is placed west to the splitting wall, at a random location and orientation. Finally, a yellow key is placed west to the splitting wall, at a random location and not overlapping the agent. The full source code for generating the initial states is available at Farama Foundation (2024, `envs/doorkey.py`). Figure 5.1(a) shows an example state for the `MiniGrid-DoorKey-8x8-v0` environment. Per default, the time horizon is set to  $t_{\max} = 10 \cdot w \cdot h$ . Based on a grid of size  $w = h = 8$ , the time horizon is therefore  $t_{\max} = 640$ . The mission space string says "use the key to open the door and then get to the goal".

### 5.1.3 Four Rooms Environments

The only environment that belongs to this family is `MiniGrid-FourRooms-v0`. Four square rooms of equal size are arranged in two rows and two columns. With,  $w = h = 19$ , the internal size of each room (excluding walls) is  $8 \times 8$ . The rooms can be traversed via gaps in the walls, which are created by removing a random wall segment for every

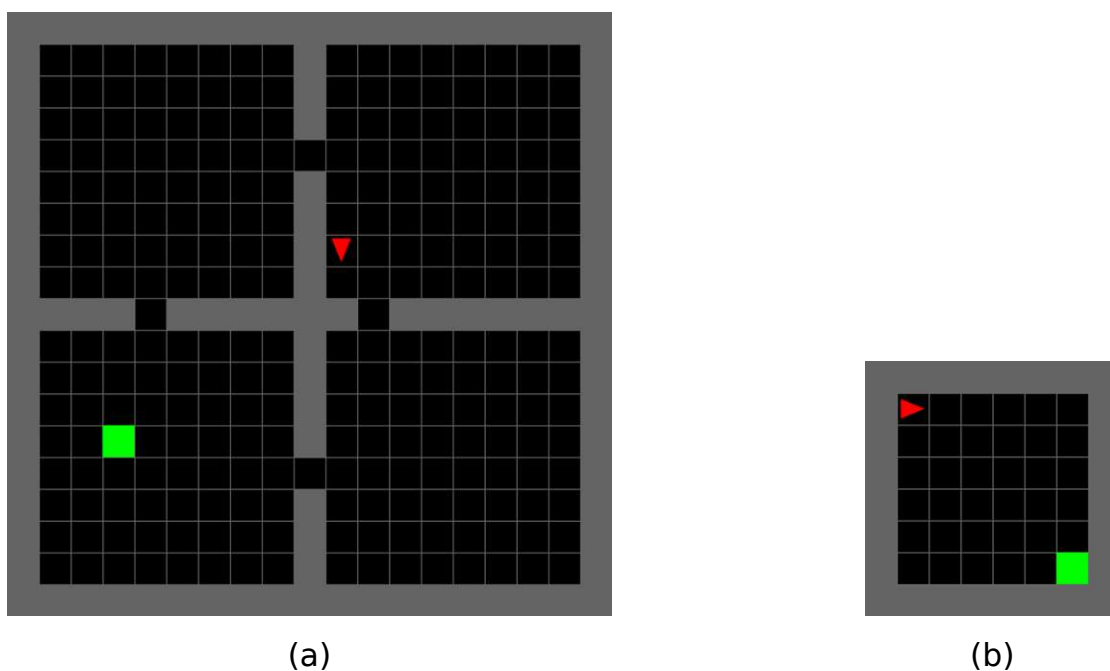


Figure 5.2: More examples of Minigrid states. In (a), a starting state of the MiniGrid-FourRooms-v0 task environment is shown. In (b), a starting state of the MiniGrid-Empty-8x8-v0 task environment is shown.

connecting wall. Both the agent and the goal tile are placed randomly in a free location. Figure 5.2(a) shows an example state for the MiniGrid-FourRooms-v0 environment. The time horizon is set to  $t_{\max} = 100$  and the mission space string says "reach the goal".

#### 5.1.4 Other Notable Environments

For the design of our abstraction function, we also considered other task environments, which we mention only briefly here.<sup>8</sup> In *Empty* environments, a single room is generated, containing only the agent and the goal tile. See Figure 5.2(b) for an example. In *Lava Gap* environments, a strip of lava tiles with a gap is placed between the agent and the goal tile. In *Dynamic Obstacles* environments, the agent and the goal tile are placed in the same room just like in *Empty* environments. In addition, a number of blue balls are placed in the room. With every time point, these balls change their location. Lava tiles and blue balls represent dangerous elements in the environment that must be avoided. Stepping onto a lava tile or onto a blue ball terminates an episode.

<sup>8</sup>Preliminary experiments in these task environments show promise but no results were included in the empirical evaluation section. This was done to keep the scope of this thesis manageable.

## 5.2 Relational State Description

By default, a Minigrad state consists of a numeric matrix and the mission space string.<sup>9</sup> Many alternative state descriptions exist (see the *Wrapper* section in Farama Foundation 2023b). But to our knowledge, no official relational state description exists. To work with the environments, we thus need to create our own.

In the grammar below, the set  $B$  represents the set of atoms (i.e. the base) from which states are constructed. The exact state space  $S \subseteq \mathcal{P}(B)$  for each task environment in particular depends on the possible room layouts.

$$\begin{aligned}
B &::= \mathbf{obj}(O, L) \mid \mathbf{carries}(O_{\text{canPickup}}) \mid \mathbf{terminal} \\
O &::= \text{goal} \mid \text{lava} \mid \text{wall}(C) \mid \text{floor}(C) \mid O_{\text{doors}} \mid O_{\text{agent}} \mid O_{\text{canPickup}} \\
O_{\text{doors}} &::= \text{door}(C, \text{open}) \mid \text{door}(C, \text{closed}) \mid \text{door}(C, \text{locked}) \\
O_{\text{agent}} &::= \text{agent}(\text{north}) \mid \text{agent}(\text{south}) \mid \text{agent}(\text{west}) \mid \text{agent}(\text{east}) \\
O_{\text{canPickup}} &::= \text{ball}(C) \mid \text{box}(C) \mid \text{key}(C) \\
C &::= \text{red} \mid \text{green} \mid \text{blue} \mid \text{purple} \mid \text{yellow} \mid \text{grey} \\
L &::= (X, Y) \\
X &::= 0 \mid 1 \mid 2 \mid \dots \mid w - 1 \\
Y &::= 0 \mid 1 \mid 2 \mid \dots \mid h - 1
\end{aligned}$$

An atom of the form  $\mathbf{obj}(o, l)$  asserts that some object  $o \in O$  is placed in some location  $l \in \mathcal{L}$  on the  $w \times h$  grid. An atom of the form  $\mathbf{carries}(o)$  asserts that some object  $o \in O_{\text{canPickup}}$  is carried by the agent. The atom  $\mathbf{terminal}$  asserts that the current state is terminal. The set of objects  $O$  is a catch-all category for everything that can be placed on the grid. It includes goal tiles, lava tiles, coloured walls, and coloured floors. It also includes the sets  $O_{\text{doors}}$ ,  $O_{\text{agent}}$ , and  $O_{\text{canPickup}}$ . The set  $O_{\text{doors}}$  consists of doors in all possible internal states and colors. The set  $O_{\text{agent}}$  consists of the agent in all possible orientations. The set  $O_{\text{canPickup}}$  consists of the objects that can be picked up by the agent, namely balls, boxes, and keys in different colors. As before, the action space is defined as  $\{ \mathbf{left}, \mathbf{right}, \mathbf{forward}, \mathbf{pickup}, \mathbf{drop}, \mathbf{toggle}, \mathbf{done} \}$ , with all actions being propositions.

For examples consider the relational state descriptions of two previously discussed states. For brevity's sake, most atoms asserting the locations of grey walls are omitted.

<sup>9</sup>After applying the *Fully Obs Wrapper* (Farama Foundation 2023b), a  $w \times h$  Minigrad state description contains a matrix  $\mathbf{A} = (a_{x,y,i})_{0 \leq x < w, 0 \leq y < h, 0 \leq i < 3} \in \mathbb{N}_0^{w \cdot h \cdot 3}$ . For every grid location  $x, y$ , the value  $a_{x,y,0}$  describes the type of object or terrain at that location. The values  $a_{x,y,1}$  and  $a_{x,y,2}$  can be used to describe additional features of that object, for example its colour and internal state.

The state from Figure 5.1 (a)	The state from Figure 5.1 (b)
<code>obj(goal, (6, 6))</code>	<code>obj(goal, (20, 17))</code>
<code>obj(door(yellow, locked), (5, 3))</code>	<code>obj(door(green, closed), (20, 19))</code>
<code>obj(key(yellow), (3, 5))</code>	<code>obj(agent(east), (21, 21))</code>
<code>obj(agent(east), (3, 1))</code>	<code>obj(wall(grey), (18, 16))</code>
<code>obj(wall(grey), (0, 0))</code>	<code>obj(wall(grey), (18, 17))</code>
<code>obj(wall(grey), (0, 1))</code>	<code>⋮</code>
<code>⋮</code>	<code>⋮</code>

### 5.3 ASP-Based Minigrid CARCASS

In this section, we propose an ASP-based CARCASS state abstraction function which can be applied to a selection of Minigrid task environments. In particular, the abstraction function is intended to work with task environments of the families Empty, Door Key, Multi Room, Four Rooms, Lava Gap, and Dynamic Obstacles.

To successfully apply a single abstraction function across this range of task environments, we need to base it on a set of common assumptions. These include the following.

- First, we assume that the task environments are composed of the same basic building blocks with common actions having effects that are localised and consistent across all of the chosen task environments.
- Second, we assume that states and actions are presented in the relational form described in the previous section.
- Third, we assume a common room layout structure. For any state in the chosen task environments, the room layout can be viewed as a set of connected, rectangular rooms with either wall tiles or lava tiles acting as room boundaries. Adjacent rooms are accessible via doors or gaps (i.e. missing wall or lava tiles) in the boundaries.
- Fourth, we assume a common task structure.

With regard to the last assumption, as laid out previously, the overarching task in any of the chosen task environments is to reach the goal tile. This task can be deconstructed into a series of subtasks that need to be achieved in a linear order. Each subtask has a navigation component, an interaction component, and is associated with some object.

**Example.** Consider the Door Key state from Figure 5.1(a). The subtasks can be enumerated as (1) getting the key, (2) unlocking the locked door, (3) moving onto the open door and (4) moving onto the goal tile. For subtask (1), the associated object is the key. Fulfilling the navigation component requires the agent to move such that it is fronting the key. Fulfilling the interaction component requires the agent to perform a **pickup** action.

Additional examples are provided below. The procedure by which this task structure was determined is presented later in this section.

### 5.3.1 Towards an abstraction function

Before presenting the details of the ASP encoding, we build up the intuition behind the abstraction function. In the most simple case, for example in Figure 5.2(b), the room layout consists of a single room. There is only one subtask with the goal tile as the associated object. Fulfilling the navigation component requires the agent to move such that the agent is fronting the goal tile. Fulfilling the interaction component requires the agent perform a **forward** action. An abstraction function for this particular case can look as follows. For a state  $s$ , let  $o_a(s)$ ,  $x_a(s)$ , and  $y_a(s)$  denote the agent's orientation,  $x$ -location, and  $y$ -location, respectively, such that  $s \models \mathbf{obj}(\mathbf{agent}(o_a(s)), (x_a(s), y_a(s)))$ . Also, let  $x_g(s)$  and  $y_g(s)$  denote the  $x$  and  $y$  locations of the goal tile, such that  $s \models \mathbf{obj}(\mathbf{goal}, (x_g(s), y_g(s)))$ . Finally, let  $\text{sgn}$  denote the sign function.<sup>10</sup> We define the state abstraction function  $\phi_1$  as:

$$\phi_1(s) = \{ \mathbf{oriented}(o_a(s)), \mathbf{xdir}(\text{sgn}(x_g(s) - x_a(s))), \mathbf{ydir}(\text{sgn}(y_g(s) - y_a(s))) \}$$

The resulting abstract state is a Herbrand interpretation consisting of atoms stating the agent's orientation, the direction of the goal tile relative to the agent on the  $x$  axis and the direction of the goal tile relative to the agent on the  $y$  axis.<sup>11</sup> As an example, the state  $s_0$  from Figure 5.2(b) yields  $o_a(s_0) = \text{east}$ ,  $x_a(s_0) = y_a(s_0) = 1$ ,  $x_g(s_0) = y_g(s_0) = 6$ , and therefore  $\phi_1(s_0) = \{ \mathbf{oriented}(\text{east}), \mathbf{xdir}(1), \mathbf{ydir}(1) \}$ .

Next, we consider task environments with two or more subtasks. As stated in the assumptions, we can expect subtasks to be linearly ordered. Therefore, it is always clear which particular subtask needs to be pursued next. More strongly, we assume that the object associated with the current subtask is always located in the same room as the agent. This is because the movement through open doors and gaps in the walls between rooms are considered to be subtasks too, which need to be fulfilled before any subtasks in those adjacent rooms can be fulfilled. Later, when discussing the ASP encodings, we show in detail how the linear order of subtasks is generated.

As all rooms are assumed to be rectangular, we can reuse parts of the previous abstraction for the navigation component of the current subtask. To this end, we define the functions  $x_{st} : S \rightarrow X$  and  $y_{st} : S \rightarrow Y$  such that they provide the  $x$  and  $y$  locations of the object associated with the current subtask, analogous to the role of  $x_g$  and  $y_g$  in  $\phi_1$ . Further, we define the function  $t_{st} : S \rightarrow O \cup \{\text{none}\}$  such that it provides information about the properties of the associated object. This information is needed to distinguish the actions required to complete the interaction components for subtasks with different associated

<sup>10</sup>As usual, let  $\text{sgn}(x) \doteq 1$  iff  $x > 0$ ,  $\text{sgn}(x) \doteq -1$  iff  $x < 0$ , and  $\text{sgn}(x) \doteq 0$  iff  $x = 0$ .

<sup>11</sup>For a less coarse abstraction, we can also define  $\phi'_1(s) = \{ \mathbf{oriented}(o_a(s)), \mathbf{xrel}(x_g(s) - x_a(s)), \mathbf{yrel}(y_g(s) - y_a(s)) \}$ , where  $\mathbf{xrel}$  and  $\mathbf{yrel}$  denote the grid coordinates of the agent relative to the goal.

objects. Note that this distinction is necessary only when the agent is located directly next to the associated object. Otherwise, let  $t_{st}(s) = \text{none}$ . A special case occurs when the agent is located in a gap between two rooms. Here, the agent cannot move as freely as when fully inside a rectangular room. To treat this case, we introduce the function  $g(s) : S \rightarrow \{\text{horizontal}, \text{vertical}, \text{none}\}$  which differentiates situations in which the agent is in a horizontal gap, a vertical gap or in no gap at all. The ASP encodings of all of these functions are described in detail below. With them, we define the abstraction function  $\phi_2$  as:

$$\phi_2(s) = \{ \mathbf{oriented}(o_a(s)), \mathbf{xdir}(\text{sgn}(x_{st}(s) - x_a(s))), \mathbf{ydir}(\text{sgn}(y_{st}(s) - y_a(s))), \\ \mathbf{touching}(t_{st}(s)), \mathbf{ingap}(g(s)) \}$$

**Example.** Consider the state  $s_1$  from Figure 5.1(b). The subtasks can be enumerated as (1) opening the closed door, (2) moving onto the open door and (3) moving onto the goal tile. Subtask (1) is associated with the closed door. Therefore,  $x_{st}(s_1) = 20$  and  $y_{st}(s_1) = 19$ . The agent is not yet touching the door, so  $t_{st}(s_1) = \text{none}$ . Also, the agent is not in a gap, so  $g(s_1) = \text{none}$ . The state is mapped to  $\phi_2(s_1) = \{ \mathbf{oriented}(\text{east}), \mathbf{xdir}(-1), \mathbf{ydir}(-1), \mathbf{touching}(\text{none}), \mathbf{ingap}(\text{none}) \}$ . After performing the actions **left**, **forward**, **left**, **forward**, **right**, we get to a new state  $s_2$  with the agent fronting the door. The navigation component for subtask (1) is now complete. The interaction component remains to be completed. Subtask (1) is therefore still the current subtask. The associated object is still the door, so  $x_{st}(s_2) = x_{st}(s_1) = 20$  and  $y_{st}(s_2) = y_{st}(s_1) = 19$ . Note that, with the agent being adjacent to the door, we get  $t_{st}(s_2) = \text{door}(\text{green}, \text{closed})$ . The abstract state is  $\phi_2(s_2) = \{ \mathbf{oriented}(\text{north}), \mathbf{xdir}(0), \mathbf{ydir}(-1), \mathbf{touching}(\text{door}(\text{green}, \text{closed})), \mathbf{ingap}(\text{none}) \}$ . After performing a **toggle** action, we get to state  $s_3$ , in which subtask (1) is completed. Moving on to subtask (2), the associated object is the now open door. With the agent already fronting the door, the navigation component is trivially complete. Completing the interaction component requires the agent to perform a **forward** action. The abstract state is  $\phi_2(s_3) = \{ \mathbf{oriented}(\text{north}), \mathbf{xdir}(0), \mathbf{ydir}(-1), \mathbf{touching}(\text{door}(\text{green}, \text{open})), \mathbf{ingap}(\text{none}) \}$ . After performing the **forward** action, we get to state  $s_4$  with subtask (2) complete. Moving on to subtask (3), the associated object is the goal tile. Therefore,  $x_{st}(s_4) = 20$  and  $y_{st}(s_4) = 17$ . Note that the agent is now located in a vertical gap. The abstract state is  $\phi_2(s_4) = \{ \mathbf{oriented}(\text{north}), \mathbf{xdir}(0), \mathbf{ydir}(-1), \mathbf{touching}(\text{none}), \mathbf{ingap}(\text{vertical}) \}$ . After performing another **forward** action, we get to state  $s_5$ . The agent is located completely in the second room and the navigation component is complete. What remains is the interaction component, which requires the agent to perform a **forward** action. The abstract state is  $\phi_2(s_5) = \{ \mathbf{oriented}(\text{north}), \mathbf{xdir}(0), \mathbf{ydir}(-1), \mathbf{touching}(\text{goal}), \mathbf{ingap}(\text{none}) \}$ . With a last **forward** action, the task environment is completed.

This abstraction can be applied to Empty environments and to Multi Room environments with two rooms, as shown in the examples above. For Multi Room environments with more than two rooms, we can extend the list of subtasks for each additional door

that needs to be opened and traversed. The abstraction can also be applied without modification to Door Key environments with the task structure laid out in a previous example and to Four Rooms environments by treating gaps in the room boundaries as waypoints just like open doors. For application in Lava Gap and Dynamic Obstacles environments, additional information is required to distinguish dangerous situations, in which a **forward** action might cause the agent to step onto a lava tile or onto a blue ball.

### 5.3.2 Detecting Room Layouts

With the intuitions and assumptions mapped out, we are now ready to describe the ASP encoding of our abstraction. Although it is based on the same principles as  $\phi_2$ , the abstraction function function outlined in the previous section, there are differences which we make clear during the discussion.

The first part of our encoding, concerned with detecting room layouts, is presented in Listing 5.1. It starts by identifying locations that are **blocked** by walls or lava tiles and locations that can be identified as a **gap** (lines 1–6). A gap surrounded by two gaps of the same type can be identified as part of an **alley**, a special room type with either a width or a height of one (lines 7–10). Locations that are blocked and locations that are gaps but not alleys are defined as **pcb** (short for possible corner piece), see lines 12–13. If three such pieces occur in an "L" shape, the possible corners of a rectangle can be identified and marked by **rtl** (short for "rectangle top left"), **rbl** ("rectangle bottom left"), **rtr** ("rectangle top right"), and **rbr** ("rectangle bottom right"), see lines 14–17. Using these four corners, a rectangle can be identified and denoted by **rect** $((x_1, y_1), (x_2, y_2))$ , where  $(x_1, y_1)$  marks the top right corner and  $(x_2, y_2)$  marks the bottom left corner (lines 19–20). Rooms are then defined as rectangles that do not contain any smaller rectangles (lines 21–25). The existence of a room is denoted by **room** $((x_1, y_1), (x_2, y_2))$ , where the terms have the same meaning as for rectangles.<sup>12</sup>

### 5.3.3 Identifying the Subtasks

Next we consider the problem of generating the linear order of subtasks. The corresponding program is presented in Listing 5.2.

We rephrase the problem as a shortest path problem in an unweighted graph. As vertices, we consider the locations of objects that can be associated with possible subtasks, namely doors, the goal, keys, and gaps that are not alleys (lines 1–5).<sup>13</sup> The location of the agent is also considered to be a vertex (line 6). The existence of a vertex is denoted by **v** $((x, y))$ , where  $(x, y)$  is the corresponding object's location. Two vertices are connected by an edge if the two corresponding objects are located in the same room (lines 8–10). In the special case that an object is located on the border of two rooms (e.g. a door or

<sup>12</sup>This encoding is tested in the previously mentioned Minigrad task environments. However, we are aware of an issue in *Crossing* environments, in particular in cases where two alleys cross.

<sup>13</sup>In the code we also define the location of balls that are not blue as vertices, although considering objects of this type is not strictly necessary for the treated task environments.

```

1 blocked(XY) :- obj(wall(_),XY).
2 blocked(XY) :- obj(lava,XY).
3 gap((X,Y),horizontal) :- not blocked((X,Y)),
4   blocked((X,Y-1)), blocked((X,Y+1)).
5 gap((X,Y),vertical) :- not blocked((X,Y)),
6   blocked((X-1,Y)), blocked((X+1,Y)).
7 alley((X,Y) :- gap((X,Y), horizontal), gap((X+1,Y), horizontal).
8 alley((X,Y) :- gap((X,Y), horizontal), gap((X-1,Y), horizontal).
9 alley((X,Y) :- gap((X,Y), vertical), gap((X,Y+1), vertical).
10 alley((X,Y) :- gap((X,Y), vertical), gap((X,Y-1), vertical).
11
12 pcp(U) :- blocked(U).
13 pcp(U) :- gap(U,_), not alley(U).
14 rtl((X,Y) :- pcp((X,Y)), pcp((X+1,Y)), pcp((X,Y+1)).
15 rbl((X,Y) :- pcp((X,Y)), pcp((X+1,Y)), pcp((X,Y-1)).
16 rtr((X,Y) :- pcp((X,Y)), pcp((X-1,Y)), pcp((X,Y+1)).
17 rbr((X,Y) :- pcp((X,Y)), pcp((X-1,Y)), pcp((X,Y-1)).
18
19 rect(((X1,Y1), (X2,Y2))) :- rtl((X1,Y1)), rbl((X1,Y2)),
20   rtr((X2,Y1)), rbr((X2,Y2)), X2-X1>0, Y2-Y1>0.
21 contains(((X11,Y11), (X12,Y12)), ((X21,Y21), (X22,Y22))) :-
22   rect(((X11,Y11), (X12,Y12))), rect(((X21,Y21), (X22,Y22))),
23   X11<=X21, Y11<=Y21, X12>=X22, Y12>=Y22,
24   ((X11,Y11), (X12,Y12))!=(X21,Y21), (X22,Y22)).
25 room(R) :- rect(R), not contains(R,_).

```

Listing 5.1: Detecting room layouts.

a gap), we consider the object to be located in both rooms. Therefore, a vertex on the border of two rooms has edges to objects in both rooms and can be used to build a path from one room to the other. The existence of an edge is denoted by  $e((x_1, y_1), (x_2, y_2))$ , where  $(x_1, y_1)$  and  $(x_2, y_2)$  are vertices.

We compute the shortest path from the vertex representing the location of the agent to the vertex representing the location of the goal tile (lines 12–16). The vertices on the path are denoted by  $\mathbf{p}(i, (x, y))$ , where  $(x, y)$  is the  $i$ -th vertex on the path. Choice rules generate all possible paths with the location of the agent as the 0-th vertex on the path, up to a length equal to the total number of vertices (line 13). A constraint ensures that the location of at least one goal tile is on the path (line 14). Using weak constraints, we optimize for paths with minimal length (lines 15–16). In case multiple paths with minimal length exist, we use weak constraints on a lower priority level to prefer paths where the 1-st vertex has minimal *Manhattan distance* to the location of the agent.

**Example.** Consider the state from Figure 5.2(a), where multiple paths of minimal length exist. If the second weak constraint is removed, both the door west to the agent and the door north to the agent are valid next subtasks and the order on the subtasks is not linear.

```

1 v(U) :- obj(door(_,_), U).
2 v(U) :- obj(goal, U).
3 v(U) :- obj(key(_), U).
4 v(U) :- obj(ball(C), U), not C = blue.
5 v(U) :- gap(U,_), not alley(U).
6 v(U) :- obj(agent(_), U).
7
8 inRoom((X,Y),((X1,Y1),(X2,Y2))) :- v((X,Y)), room((X1,Y1),(X2,Y2))
9
10 X1<=X, Y1<=Y, X2>=X, Y2>=Y.
11 e(U,V) :- v(U), v(V), inRoom(U,R), inRoom(V,R), U!=V.
12
13 p(0, U) :- obj(agent(_), U).
14 0 { p(T+1, V) : e(U,V) } 1 :- p(T,U), T<#count{ X : v(X) }.
15 :- obj(goal,_), 0=#count{ U : p(_,U), obj(goal,U) }, not terminal.
16 :~ p(T,_). [1@2, T]
17 :~ p(1,(X1,Y1)), obj(agent(_),(X2,Y2)), D=|X2-X1|+|Y2-Y1|. [D@1]
18
19 needsKeyBefore(C,T) :- p(T, U), obj(door(C,locked), U),
20 not carries(key(C)).
21 keyOnPath(C,T) :- p(T,U), obj(key(C), U).
22 :- 0=#count{ T : keyOnPath(C,T), T<T1 }, needsKeyBefore(C,T1).
23
24 nextOnPath(V) :- p(1,V).

```

Listing 5.2: Identifying the subtasks.

Adding the second weak constraint is not a perfect solution however, as there are still edge cases in which multiple valid next subtasks exist. Also note that the minimal path does not necessarily coincide with the path representing the least number of actions needed to solve the task environment. For more on these issues see Section 5.3.7.

If the shortest path contains a locked door and the agent is not yet carrying a key of the same color, a key of this type needs to be picked up before the door is visited. To cover this case, another constraint is added (lines 18–22). Having established the shortest path, we identify the next subtask to be the 1-st index on the path and distinguish the corresponding location via the atom `nextOnPath((x,y))`, see line 23.

Note that the location of a completed subtask is no longer part of the shortest path the next time the abstraction is computed.

**Example.** Consider the state from Figure 5.1(a). Recall the subgoals that should be computed for this state. First, the key requires a **pickup** action. Second, the locked door requires a **toggle** action while the key is in the inventory. Third, the open door requires a **forward** action. Fourth, the goal tile requires a **forward** action. The computed shortest path is  $((3,1), (3,5), (5,3), (6,6))$ , corresponding with the location of the agent, the location of the key, the location of the door and the location of the goal tile. Directly after the key is picked up, an answer set with a shorter path exists,

```

1  adj((X+1, Y)) :- obj(agent(_), (X, Y)).
2  adj((X-1, Y)) :- obj(agent(_), (X, Y)).
3  adj((X, Y+1)) :- obj(agent(_), (X, Y)).
4  adj((X, Y-1)) :- obj(agent(_), (X, Y)).
5
6  fronting((X, Y-1)) :- obj(agent(north), (X, Y)).
7  fronting((X, Y+1)) :- obj(agent(south), (X, Y)).
8  fronting((X+1, Y)) :- obj(agent(east), (X, Y)).
9  fronting((X-1, Y)) :- obj(agent(west), (X, Y)).
10
11 dangerous(XY) :- obj(ball(blue), XY).
12 dangerous(XY) :- obj(lava, XY).
13
14 oriented(O) :- obj(agent(O), _).
15
16 xdir(west)      :- obj(agent(_), (AX, AY)), nextOnPath((GX, GY)), AX > GX.
17 xdir(east)     :- obj(agent(_), (AX, AY)), nextOnPath((GX, GY)), AX < GX.
18 xdir(on_axis)  :- obj(agent(_), (AX, AY)), nextOnPath((GX, GY)), AX = GX.
19
20 ydir(north)    :- obj(agent(_), (AX, AY)), nextOnPath((GX, GY)), AY > GY.
21 ydir(south)    :- obj(agent(_), (AX, AY)), nextOnPath((GX, GY)), AY < GY.
22 ydir(on_axis)  :- obj(agent(_), (AX, AY)), nextOnPath((GX, GY)), AY = GY.
23
24 touching(goal) :- nextOnPath(G), adj(G), obj(goal, G).
25 touching(key)  :- nextOnPath(G), adj(G), obj(key(_), G).
26 touching(door(S)) :- nextOnPath(G), adj(G), obj(door(_, S), G).
27 touching(gap)  :- nextOnPath(G), adj(G), gap(G, _), not alley(G),
28   not touching(goal), not touching(door(_)), not touching(key).
29 touching(none) :- not touching(goal), not touching(door(_)),
30   not touching(gap), not touching(key).
31
32 ingap(D) :- obj(agent(_), XY), gap(XY, D), not alley(XY).
33 ingap(none) :- not ingap(horizontal), not ingap(vertical).

```

Listing 5.3: Predicate definitions for the abstract state.

namely  $((x_1, y_1), (5, 3), (6, 6))$ , where  $(x_1, y_1)$  is the location of the agent from which the key got picked up. The second subtask is the locked door at location  $(5, 3)$ . Directly after the door is unlocked, the shortest path is  $((4, 3), (5, 3), (6, 6))$ , where  $(4, 3)$  is by necessity the location of the agent. The third subtask is the open door at location  $(5, 3)$ . After performing a **forward** action, the location of the agent becomes identical with the location of the door and the shortest path becomes  $((5, 3), (6, 6))$ .

### 5.3.4 Predicates for the Abstract State

Having established the subtasks, we can give encodings for the functions and atoms that were used to construct the abstraction function  $\phi_2$ . These are presented in Listing 5.3.

First, additional predicates are defined (lines 1–12):

- **adj** $((x, y))$  is true for all locations  $(x, y)$  that are directly adjacent to the agent;
- **fronting** $((x, y))$  is true for the location that the agent is fronting; and
- **dangerous** $((x, y))$  is true for all locations that, when stepped onto, can terminate an episode before the task is completed.

The predicates **oriented**/1, **xdir**/1, **ydir**/1, **touching**/1, and **ingap**/1, to be used in the definition of the abstraction function, are defined in lines 14–32.<sup>14</sup>

There are two discrepancies to how  $\phi_2$  was presented before. First, in the definitions of **xdir** and **ydir** (lines 16–22), the values of the sign function are replaced with more descriptive terms. For example, **xdir**(-1) is renamed to **xdir**(west). Second, although we had  $t_{st}$  map into  $O$  previously, we now omit object properties that are not needed to solve the presented task environments (lines 24–30). For example, **touching**(door(green, closed)) is renamed to **touching**(door(closed)), omitting the fact that the door is green.

### 5.3.5 Abstract States

With the program described above as background knowledge, we can use the CARCASS in Table 5.1 to model the desired state abstraction function. The first state  $\hat{s}_1$  covers concrete states where the agent is fronting dangerous objects. The abstract states  $\hat{s}_2$  to  $\hat{s}_{757}$  represent all possible combinations of atoms from  $\phi_2$ .<sup>15</sup> The procedure from Section 3.3 can be used to translate this CARCASS to ASP.

However, we use the custom translation shown in Listing 5.4 instead, resulting in a more concise encoding.<sup>16</sup> This encoding starts with the usual choice rule and constraint for selecting an applicable abstract state (lines 1–3). The abstract state  $\hat{s}_1$  is translated in the usual way, with  $\lambda(\hat{s}_1) = \text{danger}$  (lines 5–6). The rules from lines 8–9 are used to translate the abstract states  $\hat{s}_2$  to  $\hat{s}_{757}$ . To understand this, let’s consider what happens when the above ASP encoding is applied to a state  $s$ . Note that any answer set contains at most one ground instance for each of the predicates **oriented**/1, **xdir**/1, **ydir**, **touching**/1, and **ingap**/1. For example, it is impossible that both **xdir**(west) and **xdir**(east) occur in the same answer set. Therefore,  $s$  can be covered by at most one of  $\hat{s}_2$  to  $\hat{s}_{757}$ . In other words, these abstract states cover mutually exclusive sets of states. This allows us to neglect parts of the CARCASS’s list characteristic and to translate only the one abstract state  $\hat{s}_i$  from the range that actually covers  $s$ . As a label  $\lambda(\hat{s}_i)$ , we use a tuple representation of  $\hat{s}_i$ .

<sup>14</sup>The functions  $x_a, y_a, o_a, x_{st}, y_{st}, t_{st}$ , and  $g$ , which were previously used to introduce  $\phi_2$ , are not defined explicitly here. Their values can however be read off from the atoms in which they are used.

<sup>15</sup>There exist four ground instances of **oriented**, three ground instances of both **xdir** and **ydir**, seven ground instances of **touching**, and three ground instances for **ingap**. The total number of abstract states is therefore  $1 + 4 \cdot 3 \cdot 3 \cdot 7 \cdot 3 = 1 + 756 = 757$ .

<sup>16</sup>The encoding is more concise from a human-readability point of view. It remains to be seen if the presented encoding results in a more concise grounding as well.

---

$\hat{s}_1$ :	<b>fronting</b> ( $T$ ), <b>dangerous</b> ( $T$ )
$\hat{A}_{\hat{s}_1}$ :	{ <b>left</b> , <b>right</b> , <b>forward</b> , <b>pickup</b> , <b>drop</b> , <b>toggle</b> , <b>done</b> }
$\hat{s}_2$ :	<b>oriented</b> (north), <b>xdir</b> (west), <b>ydir</b> (north), <b>touching</b> (goal), <b>ingap</b> (horizontal)
$\hat{A}_{\hat{s}_2}$ :	{ <b>left</b> , <b>right</b> , <b>forward</b> , <b>pickup</b> , <b>drop</b> , <b>toggle</b> , <b>done</b> }
$\hat{s}_3$ :	<b>oriented</b> (north), <b>xdir</b> (west), <b>ydir</b> (north), <b>touching</b> (goal), <b>ingap</b> (vertical)
$\hat{A}_{\hat{s}_3}$ :	{ <b>left</b> , <b>right</b> , <b>forward</b> , <b>pickup</b> , <b>drop</b> , <b>toggle</b> , <b>done</b> }
⋮	
$\hat{s}_{757}$ :	<b>oriented</b> (east), <b>xdir</b> (on_axis), <b>ydir</b> (on_axis), <b>touching</b> (none), <b>ingap</b> (none)
$\hat{A}_{\hat{s}_{757}}$ :	{ <b>left</b> , <b>right</b> , <b>forward</b> , <b>pickup</b> , <b>drop</b> , <b>toggle</b> , <b>done</b> }

---

Table 5.1: Faithful CARCASS abstraction.

Note that the covering abstract state  $\hat{s}_i$  (for  $2 \leq i \leq 757$ ) is just the conjunction of the set of ground atoms that is  $\phi_2(s)$ . To represent this in the encoding, we use the atom **phi2**( $t$ ), where  $t$  is a 5-tuple of terms containing the same information.

**Example.** Let some state  $s$  cover the abstract state  $\hat{s}_2 = \mathbf{oriented}(\text{north}), \mathbf{xdir}(\text{west}), \mathbf{ydir}(\text{north}), \mathbf{touching}(\text{goal}), \mathbf{ingap}(\text{horizontal})$ . Then,  $\phi_2(s) = \{ \mathbf{oriented}(\text{north}), \mathbf{xdir}(\text{west}), \mathbf{ydir}(\text{north}), \mathbf{touching}(\text{goal}), \mathbf{ingap}(\text{horizontal}) \}$ . In an optimal answer set we get the atom **phi2**((oriented(north), xdir(west), ydir(north), touching(goal), ingap(horizontal))). So,  $\lambda(\hat{s}_2) = (\text{oriented}(\text{north}), \text{xdir}(\text{west}), \text{ydir}(\text{north}), \text{touching}(\text{goal}), \text{ingap}(\text{horizontal}))$ .

For the special case that no **phi2**-atom is in the answer set, the gutter state is added. In particular, the gutter state covers concrete states in which the agent is not fronting any danger and in which the agent's location and the goal location are equivalent. Then, there exists no **nextOnPath** atom in an optimal answer set because the optimal path has length zero.

### 5.3.6 Abstract and Concrete Actions

Next, the sets of admissible actions need to be treated. In the following, we give two possible translations. The first preserves the sets of admissible actions from the Minigrid setting, with all actions admissible in all states. This can be seen as a direct translation of the CARCASS above. The second uses previously established domain knowledge to remove useless actions from the set of admissible actions.

**Unrestricted Variant.** In the presented CARCASS, all abstract states have the same admissible actions. In the translation from Section 3.3, a rule is generated for every

```

1 1 = { cStateChoice(R) : cStateCovers(R, _) }.
2 :- cStateChoice(R1), cStateCovers(R2, _),
3    cState(R1, N1), cState(R2, N2), N2 < N1.
4
5 cState(danger, 1).
6 cStateCovers(danger, ()) :- fronting(T), dangerous(T).
7
8 phi2((oriented(O), xdir(X), ydir(Y), touching(T), ingap(G))) :-
9    oriented(O), xdir(X), ydir(Y), touching(T), ingap(G).
10 cState(S, 2) :- phi2(S).
11 cStateCovers(S, ()) :- phi2(S).
12
13 cState(gutterState, #sup).
14 cStateCovers(gutterState, ()).

```

Listing 5.4: Custom CARCASS encoding.

```

1 cActionCovers(left, left).
2 cActionCovers(right, right).
3 cActionCovers(forward, forward).
4 cActionCovers(pickup, pickup).
5 cActionCovers(drop, drop).
6 cActionCovers(toggle, toggle).
7 cActionCovers(done, done).

```

Listing 5.5: Abstract and concrete actions - unrestricted variant.

state-action pair. Instead, with all actions being admissible in all abstract states, we can simply write a list of facts, as presented in Listing 5.5.

**Restricted Variant.** This variant is presented in Listing 5.6. Using the previously established domain knowledge, it is possible to identify actions that have no effect if performed in the given concrete state. The **forward** action has no effect when the agent is fronting an object that blocks the way. The **pickup** and **toggle** actions have an effect only if the agent is fronting an appropriate object. The **drop** action has an effect only if an object is carried. The **done** action never has an effect. Actions that have no effect can safely be removed from the set of admissible actions.<sup>17</sup>

### 5.3.7 Convergence Guarantees, Correctness and Other Observations

We apply the theory of state abstraction (Section 2.5) to the Minigrad setting and to the presented abstraction function. From the previous discussion we know that a Minigrad task environment with state descriptions as presented so far is not Markov. In order to obtain an MDP representation, we need to augment the previously presented state

<sup>17</sup>There is room to be more aggressive with the removal of admissible actions. For example, if concerned about safety, one can add the rule: `forwardMoveBlocked :- fronting(T), dangerous(T)`.

```

1  cActionCovers(left, left).
2  cActionCovers(right, right).
3
4  forwardMoveBlocked :- fronting(XY), obj(wall(_, XY)).
5  forwardMoveBlocked :- fronting(XY), obj(door(_, closed), XY).
6  forwardMoveBlocked :- fronting(XY), obj(door(_, locked), XY).
7  forwardMoveBlocked :- fronting(XY), obj(key(_, XY)).
8  cActionCovers(forward, forward) :- not forwardMoveBlocked.
9
10 cActionCovers(pickup, pickup) :- fronting(XY), obj(key(_, XY)).
11 cActionCovers(toggle, toggle) :- fronting(XY), obj(door(_, _), XY).
12 cActionCovers(drop, drop) :- carries(_), not forwardMoveBlocked.

```

Listing 5.6: Abstract and concrete actions - restricted variant.

description  $s_t$  with the current time point  $t$ . The result is the state description  $(s_t, t)$ . To get back to the state description as presented so far, we introduce a state abstraction function  $\omega$ , with  $\omega((s_t, t)) \doteq s_t$ . On top of this, we apply the abstraction function  $\phi_3$ , which is defined by the ASP encoding above. For our experiments, we consider  $\omega((s_t, t))$  to be the *concrete* state description and  $\phi_3(\omega((s_t, t)))$  to be the *abstract* state description. Unless otherwise noted, the analysis assumes that the set of admissible actions is preserved.

As a first result, we show the following: For all Minigrid task environments,  $\omega$  is not  $q_*$ -irrelevant. Towards constructing a counterexample, consider the timestep-augmented state  $(s_0, 0)$  from the Empty environment in Figure 5.2(b). After performing the actions **left, left, left, left**, we get another state  $(s_4, 4)$ , with  $\omega(s_0, 0) = s_0 = s_4 = \omega(s_4, 4)$ . For both states, the optimal course of actions is to perform five **forward** actions, a **left** action and another five **forward** actions. Thus, a minimum of 11 steps is needed to complete the task. With  $t_{\max} = 256$  for `MiniGrid-Empty-8x8-v0`, we get  $q_*(s_0, \mathbf{forward}) \cong 0.961 \neq 0.947 \cong q_*(s_4, \mathbf{forward})$ . This violates the property P3. Analogous counterexamples can also be obtained for the other Minigrid task environments. We conclude that, in light of the state abstraction framework as described in this thesis, there are no theoretical convergence guarantees, even for the concrete state description. The same holds for  $\phi_3$ , which is applied on top of  $\omega$ .

Second, we observe that  $\phi_3$  is not  $\pi_*$ -irrelevant for Four Rooms and Multi Room environments. To see this, consider Figure 5.3. All of the presented states map to the same abstract state. The optimal action to take for the states depicted in (a) and (b) is **forward**. The optimal action to take for the state depicted in (c) however is **right**. Therefore, there exists not a single optimal action, which violates the property P5. Similar situations can be constructed in both the Four Rooms and the Multi Room environment. For this particular example, the problem is mitigated if we use the abstraction variant with restricted action sets. In (c), the **forward** action is removed from the set of admissible actions. This makes it possible to distinguish (c) from (a) and (b), by their different sets of admissible actions. When using  $Q$ -learning with the unrestricted

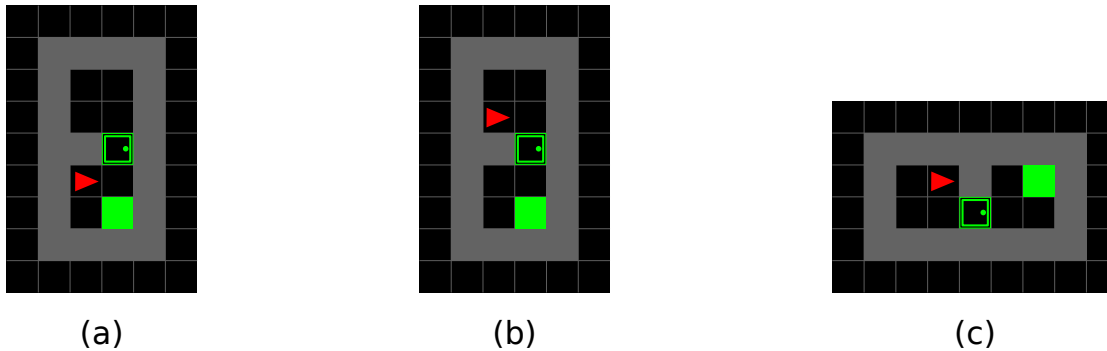


Figure 5.3: Three partial states from `MiniGrid-MultiRoom-N2-S4-v0` that all map to the same abstract state:  $\{ \mathbf{oriented}(\mathbf{east}), \mathbf{xdir}(\mathbf{east}), \mathbf{ydir}(\mathbf{south}), \mathbf{touching}(\mathbf{none}), \mathbf{ingap}(\mathbf{none}) \}$ . In (a), the current subtask is associated with the goal tile. In (b) and (c), the current subtask is associated with the closed door.

abstraction variant, we expect the different optimal actions to compete, with stability issues as a consequence.

Another unrelated problem, also preventing  $\pi_*$ -irrelevance, exists for Four Rooms environments in particular. The ASP encoding is written such that answer sets with minimal subtask-paths are to be preferred. There exist however situations where the optimal policy requires following a subtask-path that is not minimal. To illustrate, consider a `MiniGrid-FourRooms-v0` state in which the following is true: the agent is oriented south and located at (8, 8). The goal tile is located at (10, 8). The gaps in the room boundaries are located at (8, 9), (9, 1), (9, 11), (10, 9), respectively. In this example, the path with the minimal number of subgoals is ((8, 8), (9, 1), (10, 8)). Completing these two subgoals requires a total of 20 time points.<sup>18</sup> There exists however a different path ((8, 8), (8, 9), (9, 10), (10, 9), (10, 8)) which only requires 10 actions to complete.<sup>19</sup> Now consider a state identical to the first one except that the gap from (9, 1) is moved to (9, 7). The path with the minimal number of subgoals is (8, 8), (9, 7), (10, 8)), which requires 8 time points.<sup>20</sup> Both states map to the same abstract state:  $\{ \mathbf{oriented}(\mathbf{south}), \mathbf{xdir}(\mathbf{east}), \mathbf{ydir}(\mathbf{on\_axis}), \mathbf{touching}(\mathbf{none}), \mathbf{ingap}(\mathbf{none}) \}$ . In the former state however, the optimal action is to move **forward**. In the latter state, the optimal action is to turn **left**.

Finally, again in the Four Rooms environment, we are aware of a case where the abstraction function is not well-defined because of the existence of multiple optimal answer sets. Consider a `MiniGrid-FourRooms-v0` state in which the following is true. The agent is oriented west and located at (13, 5). The goal tile is located at (7, 11). The gaps in the room boundaries are located at (7, 9), (9, 5), (9, 16), and (13, 9), respectively. In

<sup>18</sup>Turn **left** twice. Move **forward** seven times. Turn **right**. Move **forward** twice. Turn **right**. Move **forward** seven times.

<sup>19</sup>Move **forward** three times. Turn **left**. Move **forward** twice. Turn **left**. Move **forward** three times.

<sup>20</sup>Turn **left** twice. Move **forward**. Turn **right**. Move **forward** twice. Turn **right**. Move **forward**.

this state, there exist two distinct shortest paths, namely  $((13, 5), (9, 5), (7, 9), (7, 11))$  and  $((13, 5), (13, 9), (9, 16), (7, 11))$ . Also, the first vertices of the paths have the same *Manhattan distance* to the location of the agent. Therefore, there exist two optimal answer sets corresponding with different abstract states. Ideally, this edge case should be resolved, for example by adding a third weak constraint on a lower priority level.

For our experiments, we did not make attempts to repair the issues mentioned above.

## 5.4 Empirical Evaluation

In this section, we study the effects of the two variants of the presented state abstraction using explorative empirical methods. For the same reasons as laid out in the previous case study (Section 4.3), we consider the observed sample efficiency, the stability of the learning process, and the quality of the learned policy. We compare both variants of abstract  $Q$ -learning to concrete  $Q$ -learning in light of the following research questions:

- Q1** Do the proposed abstractions cause differences in sample efficiency?
- Q2** Do the proposed abstractions cause stability issues during the learning process?
- Q3** Do the proposed abstractions cause differences in quality of the learned policy?

In the rest of this section, we present and discuss the results of three experiments. The first experiment investigates the effects of abstraction in a Door Key environment for both abstraction variants. The second and third experiments investigate the effects of the abstraction variant with restricted action sets in Multi Room and Four Rooms environments, respectively.

### 5.4.1 Metrics and Operationalisation

We use the same metrics as detailed in Section 4.3.1. The main metric for solution quality of each episode  $h_i$  is the realised *return*  $G_0(h_i)$  at time point zero. In addition, we call an episode  $h_i$  *successful* if it achieved a positive return, i.e.  $G_0(h_i) > 0$ . For studying stability, we observe the return of episodes across the learning process. For studying sample efficiency, we observe the *normalised cumulative return* (NCR) as defined by  $NCR(h_i) \doteq \frac{1}{i} \cdot \sum_{j=1}^i G_0(h_j)$  and count the total number of successful episodes.

### 5.4.2 Experiment Setup and Control

For concrete  $Q$ -learning, we use a variant of Algorithm 2.1 with the following state representation.<sup>21</sup> Starting from the provided *Gymnasium* interface, we first apply the *Fully Obs Wrapper* (Farama Foundation 2023b) and then translate the resulting states into our custom relational state description. This is what we refer to as the *concrete state*

<sup>21</sup>New states and rewards are realised using the Minigrid framework.

*representation.* The abstract state representation was obtained by using the presented ASP encodings with a variant of Algorithm 3.3.<sup>22</sup> This was applied on top of the concrete state representation. We use a variant of Algorithm 3.1 as implementation of abstract  $Q$ -learning.<sup>23</sup>

For all experiments, the initial  $q$ -values were set to 0.3 and the exploration factor was set to  $\epsilon = 0.2$ . No time limit was set (i.e.  $\tau = \infty$ ), since all of the investigated Minigrid environments have a finite time horizon  $t_{\max}$  that acts as a natural time limit. The learning rate was set to different values for the different task environments. For the Door Key environment, it was set to  $\alpha = 0.00001$ . For the Multi Room and Four Rooms environments, it was set to  $\alpha = 0.001$ . For every configuration, the experiment was repeated 20 times. For every repetition in the Door Key environment, the learning process was stopped after realising the first 20000 episodes. For every repetition in the Multi Room and Four Rooms environments, the learning process was stopped after realising the first 10000 episodes. We chose these parameter combinations specifically because they showed promise in preliminary experiments. No structured parameter study was performed.

To summarise the collected data, we use the first quartile  $Q_1$ , the median  $Q_2$ , the third quartile  $Q_3$  and the interquartile range (IQR) (see Definition 4.3.1).

The algorithms were implemented in *Python 3.12*<sup>24</sup>, and *clingo 5.7*<sup>25</sup>. The full source code can be found online.<sup>26</sup> The experiments were run on a desktop PC with one AMD Ryzen 5 Pro 4650G CPU and 16GB memory.

### 5.4.3 Experiment 1 - Results

The first experiment was conducted in the `MiniGrid-DoorKey-8x8-v0` environment. The results are presented in Figure 5.4.

Let's start with investigating the learning curves shown in plot (a). Returns start out at zero for all configurations. For the variant of abstract  $Q$ -learning with unrestricted action sets, the first positive median return is recorded after episode 450. The first positive  $Q_1$  return is recorded after episode 1636. After episode 2092, all recorded median returns are positive. After episode 3932, all recorded  $Q_1$  returns are positive. For the variant of abstract  $Q$ -learning with restricted action sets, the first positive median return is recorded after episode 68. The first positive  $Q_1$  return is recorded after episode 122. After episode 195, all recorded median returns are positive. After episode 2048, all recorded  $Q_1$  returns are positive. The results for concrete  $Q$ -learning are not depicted in the plot, as  $Q_1$ ,  $Q_2$ , and  $Q_3$  are all zero for all episodes.

<sup>22</sup>Line 1 of the algorithm needs to be adjusted to account for our custom CARCASS translation.

<sup>23</sup>New states and rewards are realised using the Minigrid framework.

<sup>24</sup><https://www.python.org/>

<sup>25</sup><https://potassco.org/clingo/>

<sup>26</sup><https://github.com/rbankosegger/RLASP-core>.

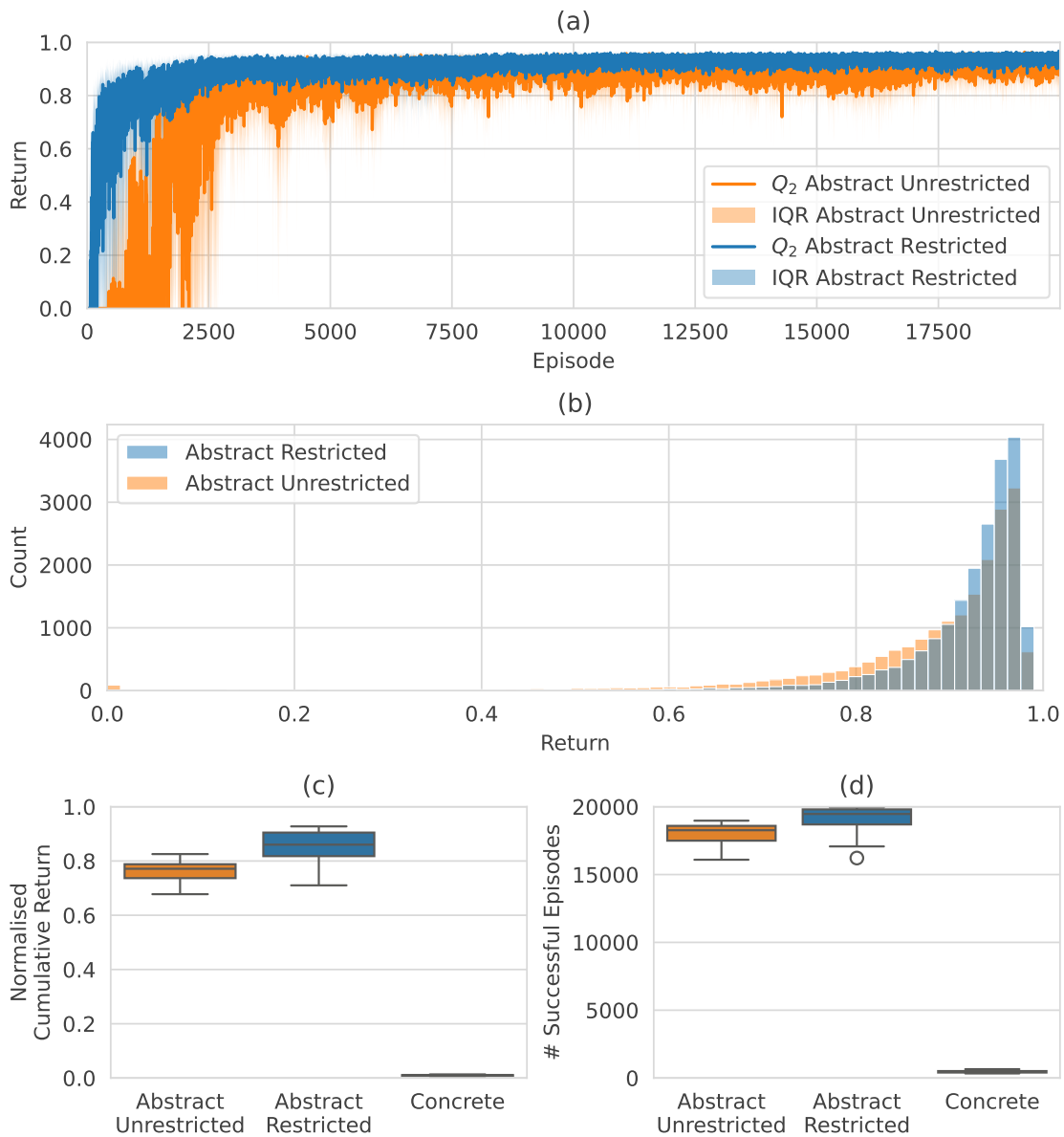


Figure 5.4: Concrete and abstract  $Q$ -learning in the MiniGrid-DoorKey-8x8-v0 environment. In (a), the learning curves for the two abstract representations of the task environment are shown in terms of the median return  $Q_2$  and its interquartile range. In (b), a histogram shows the distribution of the return of episodes 19001 to 20000 for the two abstract representations. Aggregating 20 repetitions over 1000 episodes gives a total count of 20000 samples per configuration. In (c), box plots of the normalised cumulative returns at episode 20000 are shown for both the concrete representation and for the two abstract representations. In (d), box plots of the counts of total successful episodes after episode 20000 are shown for both the concrete representation and for the two abstract representations.

Next, let's look at the distribution of returns, aggregating all repetitions of all episodes between 19001 and 20000 as shown in plot (b). For the variant of abstract  $Q$ -learning with unrestricted action sets, a median ( $Q_1, Q_3$ ) return of 0.923 (0.852, 0.956) is observed. This equates to a median ( $Q_1, Q_3$ ) of 55 (105, 31) time points spent per episode. Out of the total of 20000 episodes, 88 (0.440%) have a return of zero. The maximum observed return is 0.987, which equates to nine time points spent in that particular episode. For the variant of abstract  $Q$ -learning with restricted action sets, a median ( $Q_1, Q_3$ ) return of 0.941 (0.902, 0.962) is observed. This equates to a median ( $Q_1, Q_3$ ) of 42 (70, 27) time points spent per episode. Out of the total of 20000 episodes, 6 (0.030%) have a return of zero. The maximum observed return is 0.990, which equates to a total of seven steps needed to solve that particular episode. For concrete  $Q$ -learning, a median ( $Q_1, Q_3$ ) return of 0.000 (0.000, 0.000) is observed. Out of the total of 20000 episodes, 19459 (97.295%) have a return of zero. The maximum observed return is 0.951, which equates to 35 time points spent in that particular episode. These results are not shown in the histogram.

Plot (c) shows box plots of the achieved NCRs. For the variant of abstract  $Q$ -learning with unrestricted action sets, the median ( $Q_1, Q_3$ ) NCR is 0.771 (0.737, 0.788), with a minimum of 0.678 and a maximum of 0.826. For the variant of abstract  $Q$ -learning with restricted action sets, the median ( $Q_1, Q_3$ ) NCR is 0.860 (0.818, 0.905), with a minimum of 0.710 and a maximum of 0.927. For concrete  $Q$ -learning, the median ( $Q_1, Q_3$ ) NCR is 0.009 (0.08, 0.011), with a minimum of 0.006 and a maximum of 0.013

Plot (d) shows box plots of the total counts of successful episodes. For the variant of abstract  $Q$ -learning with unrestricted action sets, the median ( $Q_1, Q_3$ ) count of successful episodes is 18268 (17498, 18591) with a minimum of 16094 and a maximum of 18972. For the variant of abstract  $Q$ -learning with restricted action sets, the median ( $Q_1, Q_3$ ) count of successful episodes is 19467 (18695, 19803) with a minimum of 16218 and a maximum of 19948. For concrete  $Q$ -learning, the median ( $Q_1, Q_3$ ) count of successful episodes is 468 (404, 513) with a minimum of 338 and a maximum of 653.

For both variants of abstract  $Q$ -learning, when compared to concrete  $Q$ -learning, there are obvious differences in the shapes of the learning curves, the distributions of realised returns in the last 1000 episodes, the normalised cumulative returns, and the number of successful episodes. Comparing the two variants of abstract  $Q$ -learning, there are tentative differences. Statistical testing is needed however to evaluate whether these differences are significant.

#### 5.4.4 Experiment 1 - Interpretation

For both variants of abstract  $Q$ -learning, there are clear differences in the NCR, in the number of successful episodes, and in the shape of the learning curves when compared to concrete  $Q$ -learning. From this we conclude that both variants of abstract  $Q$ -learning have a higher sample efficiency than concrete  $Q$ -learning. The only plausible explanation

of this observation is the change from the concrete representation to one of the abstract representations. This lets us infer a causal relationship.

Regarding stability issues, observe that, after episode 3932,  $Q_1$  is consistently above zero in the learning curves for both abstract variants of  $Q$ -learning. In addition, the interquartile ranges of the observed NCRs and the observed numbers of successful episodes are small, suggesting there are only small differences in the behaviour of the learning process between different repetitions. From the histogram of returns from the last 1000 episodes of the learning processes, it can be seen that the vast majority of episodes ( $> 99.5\%$ ) are successful for both variants of abstract  $Q$ -learning. Although we cannot exclude the presence of instability based on these results, we conclude that its influence is negligible.

Regarding solution quality, it is not sensible to make comparative statements involving the learned policy of concrete  $Q$ -learning, which has not yet produced a policy of reasonable quality by episode 20000. Also, no general statements about the optimality of the learned solutions can be made for either variant of abstract  $Q$ -learning based on the available results. Note however that there exist states for which the optimal solution was displayed. In particular, the variant of abstract  $Q$ -learning with restricted action sets realised an episode with the maximum return of 0.990, which equates to a total of seven steps needed to solve that particular episode.<sup>27</sup> Considering the threshold of success, i.e. whether the return of any particular episode is positive, we consider the learned solutions of both abstract variants of  $Q$ -learning to have a satisficing solution quality.

#### 5.4.5 Experiment 2 - Results

The second experiment was conducted in the `MiniGrid-MultiRoom-N2-S4-v0` environment with the parameters set to  $\alpha = 0.001$  and  $\epsilon = 0.2$ . The results are presented in Figure 5.5.

The learning curve of the variant of abstract  $Q$ -learning with restricted action sets is shown in plot (a). Starting out at a return of zero, the first positive median return is recorded after episode 133. The first positive  $Q_1$  return is recorded after episode 639. After episode 1012, all recorded median returns are positive. After episode 1962, all recorded  $Q_1$  returns are positive. The results for concrete  $Q$ -learning, are not depicted in the plot, as  $Q_1$  and  $Q_2$  are zero for all episodes. There are exactly two instances for which  $Q_3$  is positive: episodes 5513 and 9803, both with a  $Q_3$  return of 0.025.

The histogram in plot (b) shows the distribution of returns of the variant of abstract  $Q$ -learning with restricted action sets, aggregating all repetitions of all episodes between 9001 and 10000. A median ( $Q_1, Q_3$ ) return of 0.798 (0.730, 0.843) is observed. This equates to a median ( $Q_1, Q_3$ ) of 4.5 (6, 3.5) time points spent per episode. Out of the total of 20000 episodes, 5 (0.025%) have a return of zero. The maximum observed return

<sup>27</sup>This is only possible in an initial state with the following conditions. The agent is touching and fronting the key. The agent is touching the door. The door is located at (5, 5). The optimal set of actions in this case is **pickup, right, toggle, forward, forward, right, forward**.

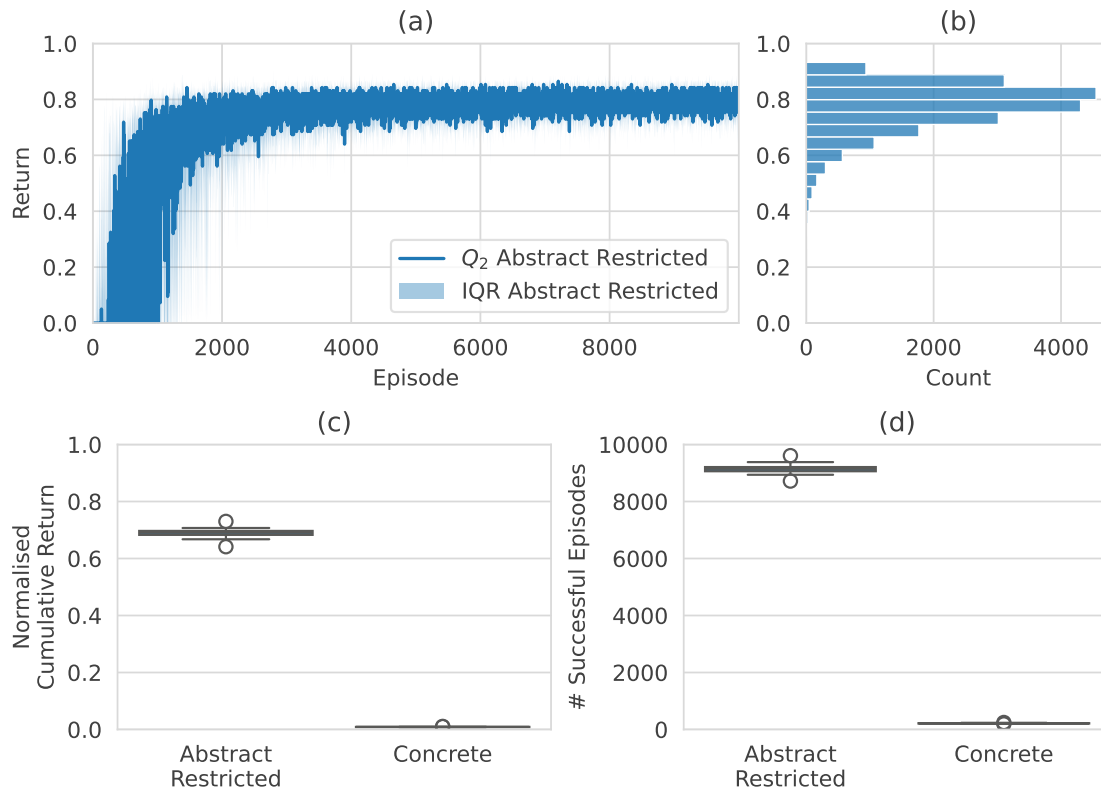


Figure 5.5: Comparison of the behaviour policies for concrete  $Q$ -learning and for the variant of abstract  $Q$ -learning with restricted action sets in the MiniGrid-MultiRoom-N2-S4-v0 environment. In (a), the learning curve for the abstract representation is shown in terms of the median return and its interquartile range. In (b), a histogram shows the distribution of the return of episodes 9001 to 10000 for the abstract representation. Aggregating 20 repetitions over 1000 episodes gives a total count of 20000 samples. In (c), box plots of the normalised cumulative returns are shown for both the concrete representation and for the abstract representation. In (d), box plots of the counts of total successful episodes are shown for both the concrete representation and for the abstract representation.

is 0.933, which equates to three time points spent in that particular episode. For concrete  $Q$ -learning, a median ( $Q_1, Q_3$ ) of 0.000, (0.000, 0.000) is observed. Out of the total of 20000 episodes, 19554 (97.770%) have a return of zero. The maximum observed return is 0.933, which equates to three time points spent in that particular episode. These results are not shown in the histogram.

Plot (c) shows box plots of the achieved NCRs. For the variant of abstract  $Q$ -learning with restricted action sets, the median ( $Q_1, Q_3$ ) NCR is 0.691 (0.683, 0.697) with a minimum of 0.641 and a maximum of 0.730. For concrete  $Q$ -learning, the median ( $Q_1, Q_3$ ) NCR is 0.009 (0.009, 0.009) with a minimum of 0.008 and a maximum of 0.011.

Plot (d) shows box plots of the total counts of successful episodes. For the variant of abstract  $Q$ -learning with restricted action sets, the median ( $Q_1, Q_3$ ) count of successful episodes is 9148 (9061, 9208) with a minimum of 8717 and a maximum of 9609. For concrete  $Q$ -learning, the median ( $Q_1, Q_3$ ) count of successful episodes is 215 (210, 221) with a minimum of 194 and a maximum of 246.

Comparing the variant of abstract  $Q$ -learning with restricted action sets to concrete  $Q$ -learning, there are obvious differences in the shapes of the learning curves, the distributions of realised returns in the last 1000 episodes, the normalised cumulative returns, and the number of successful episodes.

#### 5.4.6 Experiment 2 - Interpretation

From the differences in the NCR, in the number of successful episodes, and in the shape of the learning curves we conclude that the variant of abstract  $Q$ -learning with restricted action sets has a higher sample efficiency than concrete  $Q$ -learning. The only plausible explanation of this observation is the change from the concrete representation to the abstract representation. This lets us infer a causal relationship.

Regarding stability issues for abstract  $Q$ -learning with restricted action sets, observe that after episode 1962,  $Q_1$  is consistently above zero in the learning curve. In addition, the interquartile range of the NCR and the number of successful episodes is small, suggesting there are only small differences in the behaviour of the learning process between different repetitions. From the histogram of returns from the last 1000 episodes of the learning processes, it can be seen that the vast majority of episodes ( $> 99.9\%$ ) are successful. Although we cannot exclude the presence of instability based on these results, we conclude that its influence is negligible.

Regarding solution quality, it is not sensible to make comparative statements involving the learned policy of concrete  $Q$ -learning, which has not yet produced a policy of reasonable quality by episode 10000. Also, no general statements about the optimality of the learned solutions can be made based on the available results. Considering the threshold of success, i.e. whether the return of any particular episode is positive, we consider the learned solution of the variant of abstract  $Q$ -learning with restricted action sets to have a satisfactory solution quality.

#### 5.4.7 Experiment 3 - Results

The third experiment was conducted in the MiniGrid-FourRooms-v0 environment. The results are presented in Figure 5.6.

The learning curve of the variant of abstract  $Q$ -learning with restricted action sets is shown in plot (a). Starting out at a return of zero, the first positive median return is recorded after episode 287. The first positive  $Q_1$  return is recorded after episode 1403. After episode 3137, all recorded median returns are positive. After episode 4649, all recorded  $Q_1$  returns are positive. The results for concrete  $Q$ -learning are not depicted

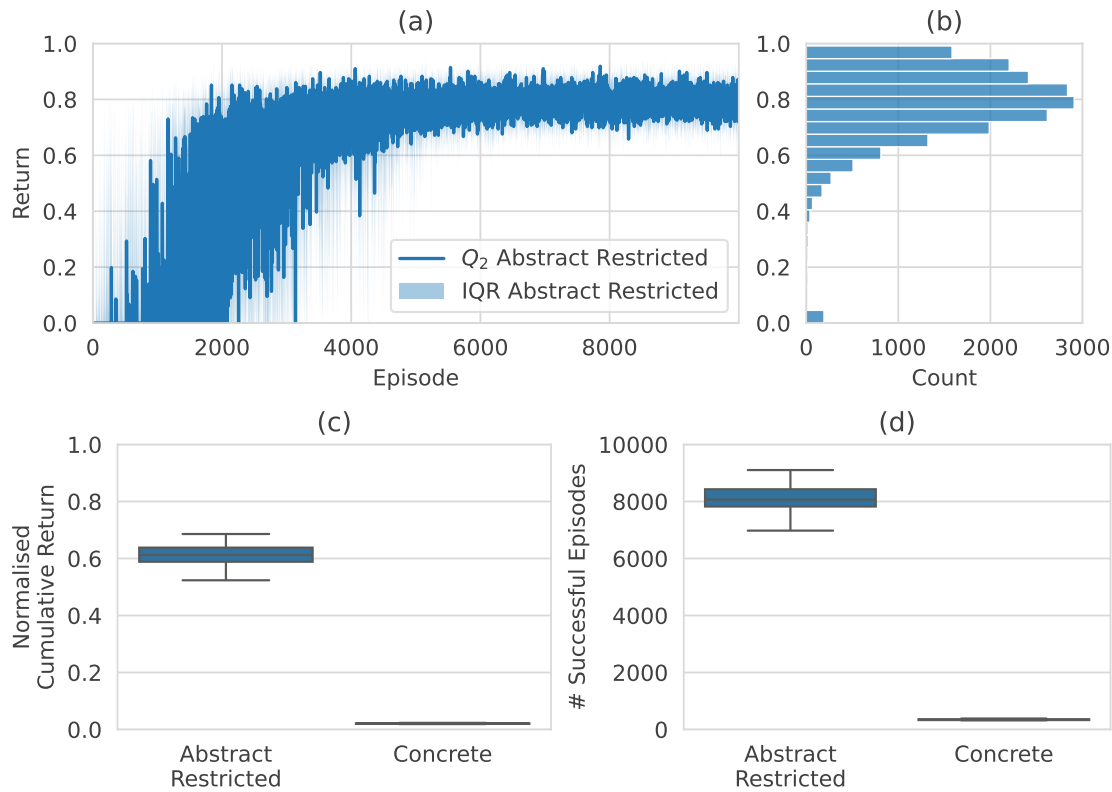


Figure 5.6: Comparison of the behaviour policies for concrete  $Q$ -learning and for the variant of abstract  $Q$ -learning with restricted action sets in the Concrete and abstract  $Q$ -learning in the MiniGrid-FourRooms-v0 environment. In (a), the learning curve for the abstract representation is shown in terms of the median return and its interquartile range. In (b), a histogram shows the distribution of the return of episodes 9001 to 10000 for the abstract representation. Aggregating 20 repetitions over 1000 episodes gives a total count of 20000 samples. In (c), box plots of the normalised cumulative returns are shown for both the concrete representation and for the abstract representation. In (d), box plots of the counts of total successful episodes are shown for both the concrete representation and for the abstract representation.

in the plot, as  $Q_1$  and  $Q_2$  are zero for all episodes. There are exactly four instances for which  $Q_3$  is positive: episodes 4022, 7164, 8323, and 8898 with returns of 0.025, 0.169, 0.449, and 0.041, respectively.

The histogram in plot (b) shows the distribution of returns of the variant of abstract  $Q$ -learning with restricted action sets, aggregating all repetitions of all episodes between 9001 and 10000. A median ( $Q_1, Q_3$ ) return of 0.793, (0.712, 0.874) is observed. This equates to a median ( $Q_1, Q_3$ ) of 23 (32, 14) time points spent per episode. Out of the total of 20000 episodes, 192 (0.960%) have a return of zero. The maximum observed return is 0.991, which equates to one time point spent in that particular episode. For

concrete  $Q$ -learning, a median ( $Q_1, Q_3$ ) return of 0.000 (0.000, 0.000) is observed. Out of the total of 20000 episodes, 19240 (96.200%) have a return of zero. The maximum observed return is 0.991, which equates to one time point spent in that particular episode.

Plot (c) shows box plots of the achieved NCRs. For the variant of abstract  $Q$ -learning with restricted action sets, the median ( $Q_1, Q_3$ ) NCR is 0.612 (0.589, 0.638) with a minimum of 0.524 and a maximum of 0.686. For concrete  $Q$ -learning, the median ( $Q_1, Q_3$ ) NCR is 0.020526 (0.019838, 0.021713) with a minimum of 0.019 and a maximum of 0.023.

Plot (d) shows box plots of the total counts of successful episodes. For the variant of abstract  $Q$ -learning with restricted action sets, the median ( $Q_1, Q_3$ ) count of successful episodes is 8068 (7824, 8429) with a minimum of 6978 and a maximum of 9103. For concrete  $Q$ -learning, the median ( $Q_1, Q_3$ ) count of successful episodes is 346 (332, 357) with a minimum of 322 and a maximum of 385.

Comparing the variant of abstract  $Q$ -learning with restricted action sets to concrete  $Q$ -learning, there are obvious differences in the shapes of the learning curves, the distributions of realised returns in the last 1000 episodes, the normalised cumulative returns, and the number of successful episodes.

### 5.4.8 Experiment 3 - Interpretation

From the differences in the NCR, in the number of successful episodes, and in the shape of the learning curves we conclude that the variant of abstract  $Q$ -learning with restricted action sets has a higher sample efficiency than concrete  $Q$ -learning. The only plausible explanation of this observation is the change from the concrete representation to the abstract representation. This lets us infer a causal relationship.

Regarding stability issues for abstract  $Q$ -learning with restricted action sets, observe that, after episode 4649,  $Q_1$  is consistently above zero in the learning curve. In addition, the interquartile range of the NCR and the number of successful episodes is small, suggesting there are only small differences in the behaviour of the learning process between different repetitions. From the histogram of returns from the last 1000 episodes of the learning processes, it can be seen that the vast majority of episodes (> 99%) are successful. Although we cannot exclude the presence of instability based on these results, we conclude that its influence is negligible.

Regarding solution quality, it is not sensible to make comparative statements involving the learned policy of concrete  $Q$ -learning, which has not yet produced a policy of reasonable quality by episode 10000. Also, no general statements about the optimality of the learned solutions can be made based on the available results. Considering the threshold of success, i.e., whether the return of any particular episode is positive, we consider the learned solution of the variant of abstract  $Q$ -learning with restricted action sets to have a satisfactory solution quality.

## 5.5 Discussion

In this chapter, we proposed and evaluated two variants of an ASP-based state abstraction function for several Minigrid task environments. The two variants of abstract  $Q$ -learning use the same state abstraction function but differ in their sets of admissible actions. The first preserves the sets of admissible actions, such that every action is admissible in every abstract state. The second restricts the action sets such that actions with no effect on the environment are removed. For evaluation, we applied concrete and abstract versions of  $Q$ -learning with an  $\epsilon$ -greedy behaviour policy in three Minigrid task environments, namely `MiniGrid-DoorKey-8x8-v0`, `MiniGrid-FourRooms-v0`, and `MiniGrid-MultiRoom-N2-S4-v0`, belonging to the families of Door Key, Multi Room, and Four Rooms, respectively. The experiments were performed with one parameter configuration per task environment. We cannot claim any generalisability of the presented results for other parameter configurations and for other task environments beyond the investigated ones. The posed research questions are answered as follows.

### Q1 - Do the proposed abstractions cause differences in sample efficiency?

In the Door Key environment, we concluded that using either of the proposed variants of abstract  $Q$ -learning causes an increase in sample efficiency when compared to concrete  $Q$ -learning. In the Four Rooms and Multi Room environments, we concluded that using the variant of abstract  $Q$ -learning with restricted action sets causes an increase in sample efficiency when compared to concrete  $Q$ -learning. Concrete  $Q$ -learning failed to achieve consistently positive returns over a span of 20000 episodes in the Door Key environment and over a span of 10000 episodes in the other two environments. Learning abstract policies of acceptable quality was possible within the same episode spans.

These results are consistent with the discussed theory of state abstraction, claiming increased sample efficiency as a consequence of a smaller abstract state space. This suffices as an explanation of the results for the unrestricted variant of abstract  $Q$ -learning. For the variant of abstract  $Q$ -learning with restricted action sets, two additional factors need to be taken into account. First, there is also a reduction in the size of the action space. But note that this may in turn increase the size of the state space, as abstract states need to be split further by their sets of admissible actions. Second, the restrictions on the action sets help with exploration, since actions without an effect are eliminated. The extent to which each of these factors contribute to the sample efficiency remains to be investigated.

### Q2 - Do the proposed abstractions cause stability issues during the learning process?

In the case of restricted action sets, parameter configurations have been found for all three environments with no detectable stability issues in the first 10000 (resp. 20000) episodes. Also, in the case of unrestricted action sets, a parameter configuration has been

found for the tested Door Key environment with no detectable stability issues in the first 20000 episodes.

### **Q3 - Do the proposed abstractions cause differences in quality of the learned policy?**

No direct comparison of the quality of the learned policies was possible due to concrete  $Q$ -learning being intractable. Also, based on the available results, no statements about the optimality of the learned abstract policies can be made. Within the scope of the discussed theoretical framework, the optimality of the learned abstract policies can not be guaranteed. In fact, we provided negative results concerning Four Rooms and Multi Rooms environments, showing the existence of Minigrid states for which there is no abstract representation of the optimal policy. Despite these problems with the abstraction function, we found that the learned abstract policies are able to complete the vast majority of given tasks successfully, i.e. with positive rewards.

## Related Work

In this chapter, we discuss related work concerning reinforcement learning in relational MDPs. In the first part of our discussion, we focus on relational representations of abstract value functions and of abstract policies as solution methods to overcome the state explosion problem. In the second part, we provide a brief overview of adjacent research areas.

### 6.1 Representing Abstractions in RMDPs

Most influential to our own work is the CARCASS framework (van Otterlo 2003; van Otterlo 2008, pp. 252–270). The abstraction methods we used in this thesis are represented as CARCASS’s but differ in the knowledge representation method: we use ASP instead of Prolog. The differences are discussed in detail in Chapter 3. In addition, van Otterlo (2008, pp. 261–264) presents two CARCASS abstractions for the blocks world environment with the task to stack all blocks in an arbitrary order. The first CARCASS, to be applied to the five-blocks world, preserves the Markov property of the concrete RMDP. It showcases how background knowledge can be used in the abstraction, utilizing some of Prolog’s built-in predicates `findall/3` and `length/2` in order to count the number of towers in any given state. Similar results can be achieved in ASP using aggregates. The second CARCASS is very coarse, with only four abstract states. It supports the learning of a generalised unstack-stack policy, to be applied in blocks worlds of any size. For the experiments, blocks worlds with up to 15 blocks were considered. Abstract  $Q$ -learning and *Prioritised Sweeping* were used as learning algorithms. This is different from our experiments in the blocks world, for which we assumed a more complex task with the goal to stack the blocks in one particular, predefined order. In our experiments, we considered blocks worlds with up to 20 blocks.

Detailed surveys of work closely related to the CARCASS framework are available (van Otterlo 2008, pp. 271–284; van Otterlo 2012). Most relevant to our own thesis are the

following two:

First, Kersting and De Raedt (2004) introduced *logical Markov decision programs (LOMDPs)* and presented the *logical TD( $\lambda$ )* algorithm, working on abstract state-action pairs. Analogous to the CARCASS framework, abstract states are represented as universally quantified conjunctions of atoms, abstract actions are represented as single atoms, and abstract policies are represented as finite sets of decision rules. In difference to the CARCASS framework, the coverage relation between abstract and concrete states is defined using  *$\theta$ -subsumption*, which is less expressive than the SLDNF-based covering relation of the CARCASS when it comes to the addition of background knowledge in the form of a logic program. Prolog is used for the implementation. Various experiments are performed in the blocks world, with the goal to stack blocks in arbitrary order, similar to the previously described work of van Otterlo (2008, pp. 261–264).

Second, Morales (2003) defined a set of abstract *r-states*, each a conjunction of atoms. A set of abstract *r-actions* is defined independently, with both preconditions and postconditions also given as conjunctions of atoms. A Prolog notation is used for representing these conjunctions. The *relational Q-learning (rQ-learning)* algorithm is presented, which works on the level of abstract *r-state-r-action pairs*. Experiments are presented in two grid world task environments and one chess-related domain. In the first grid world environment, the task is to reach a goal tile from some arbitrary location. The second grid world environment represents a taxi domain. The task is to navigate to a pick-up location, pick up a passenger, then navigate to a delivery location and finally to drop off the passenger. State-action pair abstractions are presented for both tasks, aggregating actions based on whether their execution increases or decreases the Manhattan distance to a particular *goal rank*. This is different from our own work in the Minigrad setting, where we presented a state abstraction and left the actions concrete. In addition, we provide detailed ASP encodings and propose methods to identify multiple subtasks and to navigate more complex room layouts.

Later, Koga, Silva, and Costa (2015) utilised *stochastic abstract policies*, defined as stochastic mappings from abstract states (again, conjunctions of atoms) to abstract actions. Based on a set of source task environments with known transition and reward models, a variation of dynamic programming is used to obtain a single abstract policy which, although possibly suboptimal for individual tasks environments, can be used as an exploration policy to guide learning in new and unknown task environments with comparable relational state descriptions. For the experimental evaluation, grid world navigation tasks with complex room layouts are considered. The presented concrete state descriptions include rooms, corridors and doors as unique objects, as well as a Manhattan distance based measure of how close these objects are to the goal, relative to the agents location. A state-action pair abstraction is built based on this knowledge. In contrast, our own work in the Minigrad setting derives a more complex, graph-based distance measure from the state description.

Karia and Srivastava (2022) used a description logic for modelling relational state-action pair abstractions. The logic used extends the standard  $\mathcal{ALC}$  syntax and semantics with the

equality operator on binary relations and the role inverse (Baader et al. 2003). Abstract state-action pairs are represented as numerical feature vectors. Each feature is defined using the description language, either counting the number of individuals (i.e. ground terms) in the relational state description that are instances of a given concept description or computing the minimum distance between two concept descriptions based on a role. The feature vector is used as the input of a multilayer perceptron based regression model to predict the value function. Note that the vector length is constant even when the number of objects in the state description varies, allowing for the regression model to generalise across similar task environments with different domains. The model is trained using *deep Q-learning*. The features are automatically generated before training, using the sampled state space of a smaller but similar problem. Experiments are presented in four RDDL task environments: a sysadmin domain, an academic advising domain, a game of life domain and a wildfire domain.

The state-of-the-art *RePreL* framework (Kokel, Natarajan, et al. 2023; Kokel, Manoharan, et al. 2021) uses a planner to decompose a given RMDP into a partially ordered sequence of *subgoal RMDPs*. The planner has access to a set of manually specified *operators*, each consisting of a primitive task as well as first-order formulae describing preconditions, effects, and termination conditions. Each subgoal RMDP is derived from a ground instance of one of these operators. The transition and reward models are based on the original, concrete RMDP, with modifications to reflect the specific termination conditions of the subgoal RMDP. In addition, an abstract state description is obtained for the subgoal RMDP by deleting irrelevant atoms and by replacing specific object names with generic (Skolem) constants, ensuring that the abstraction generalises over different substitutions for the same operator. The irrelevant atoms are inferred from the concrete state description, the substitution used to ground the operator, and a set of manually specified *dynamic first-order conditional influence (D-FOCI) statements*, describing the influence that operators, actions and state atoms have on the reward and on other state atoms in the current state and the next state. Policies for the subgoal RMDPs can be learned using model-free reinforcement learning techniques. The framework is evaluated in discrete and continuous domains with rich relational structures and compared to related work, including *hierarchical reinforcement learning*, other combinations of planning with reinforcement learning, and *deep reinforcement learning*. Of particular interest to our own work are the experiments in three grid world domains with task environments comparable to those used in the Minigrid setting. Abstract *Q-learning* is used to learn policies for the subgoal RMDPs. For our ASP-CARCASS based abstraction in the Minigrid setting, we also assumed that the main task can be decomposed into subtasks, but this was achieved differently. In *RePreL*, the task structure is explicitly modelled with an action language. The separation of the task structure from the state descriptions allows for the planner to also function in partially observable domains. This was demonstrated in experiments involving the relational box world. In our work, the task structure is used implicitly as a means to obtain state abstractions and represented as the solution to a shortest path problem in a graph, which is in turn inferred directly from the state description based on the room layout and on the locations of objects. Furthermore, the *RePreL* planning

step happens only once at the start of each episode. By contrast, in our work the ASP solver is used at every timestep. There are also differences in the abstraction methods. The *RePreL* framework uses model-irrelevant state abstraction functions, whereas the abstraction functions in our work are coarser. Finally, unlike the CARCASS framework, the *RePreL* framework cannot represent action space abstractions.

The *Detect, Understand, Act (DUA)* reinforcement learning architecture (Mitchener et al. 2022) aims to solve task environments from the Animal AI competition testbed (Crosby et al. 2019). In this setting, the agent is situated in a three-dimensional world, receiving two-dimensional RGB images as observations. Inspired by animal cognition experiments, the tasks are complex reasoning problems that involve navigation and object manipulation. In order to succeed, an agent must demonstrate a range of cognitive abilities, such as spatial reasoning, object permanence and tool use. DUA is tailored to this environment and consists of three components. First, the *Detect* component is a computer vision module that induces relational descriptions from the observed images by detecting objects and relations. Second, the *Act* component consists of a set of pre-defined, fixed policies that are capable of achieving distinct subtasks in the environment. These policies, along with termination conditions, form *options* in the sense of Sutton and Barto (2018, Chapter 17.2). Third, the *Understand* component is designed to learn and represent a *meta-policy*, mapping the relational observations from *Detect* to options from *Act*, analogous to how normal policies would map states to actions.<sup>1</sup> The meta-policy is represented as an answer-set program and learned using the *ILASP learner* (Law, Russo, and Broda 2020). The encoding consists of a relational description of the current observation, common sense background knowledge, the preconditions for options to be admissible, and a decision list built from weak constraints, representing the meta-policy itself. Answer sets correspond one-to-one with admissible options and optimal answer sets correspond with concrete options to execute next.

## 6.2 Adjacent Research Areas

In this section, we briefly mention adjacent research in the RMDP setting concerned with other function approximation techniques besides abstraction, with planning in the RMDP setting, and with other ways of combining ASP and reinforcement learning.

*Relational reinforcement learning* (Dzeroski, De Raedt, and Blockeel 1998) is concerned with studying the application of *relational learning* techniques to the field of reinforcement learning. To this end, classical function approximation techniques are lifted to handle relational representations. Examples include regression trees (Dzeroski, De Raedt, and Blockeel 1998; Džeroski, Raedt, and Driessens 2001), instance based regression (Driessens and Ramon 2003), kernel methods (Gärtner, Driessens, and Ramon 2003), tile coding (Bloch 2018), and gradient boosting (Das et al. 2020). In some methods, explicitly represented state-action pair abstractions are a by-product of the learning process. For

<sup>1</sup>The role of the meta-policy is comparable to the role of the planner in RePreL. Both RePreL and DUA are part of a wider research area called *hierarchical reinforcement learning* (Pateria et al. 2022).

example, the branches in a relational regression tree can be viewed as existentially quantified conjunctions of literals inducing a partition of the state-action space. Recent developments include the application of *deep relational learning* techniques to RMDPs (Zambaldi et al. 2019; Janisch, Pevný, and Lisý 2020; Garg, Bajpai, and Mausam 2020; Sharma et al. 2023) and of *neuro-symbolic* methods (Dong et al. 2019; Hazra and Raedt 2023).

*Decision theoretic planning* (or *probabilistic planning*) is closely related to model-based reinforcement learning. In the relational context, compact representations can be achieved not only for value functions and policies but also for other aspects of the RMDP framework, such as transition models and reward models. RMDPs can be represented using formal languages such as PPDDL (Younes et al. 2005; Taitler, Gimelfarb, et al. 2022) and RDDDL (Sanner 2010). These representations allow for the development of algorithms that solve RMDPs on a purely intensional level, without the need to enumerate concrete states or actions. Examples include the REBEL algorithm (van Otterlo 2008, Chapter 6; Kersting, van Otterlo, and De Raedt 2004), which was presented in the same PhD thesis as the CARCASS framework, and the FOALP system (Sanner and Boutilier 2009). For recent work we refer to the probabilistic track of the regularly held *international planning competition* (Taitler, Alford, et al. 2024).

Other methods that combine ASP and reinforcement learning have been proposed in the literature. To start with, employing ASP for planning and for reasoning about action languages can facilitate exploration by restricting admissible actions (Ferreira et al. 2017) or by acting as a behaviour policy (Datler 2020). ASP-generated plans can also be used to derive a distance measure for relational instance-based regression (Nickles 2011) or for high-level decision making in hierarchical frameworks (Lyu et al. 2019), with task planning in mobile robots as a special case (Yang et al. 2014). Other robot architectures employ ASP for a variety of tasks, such as reasoning with incomplete domain knowledge, planning, and diagnosis (Zhang, Sridharan, and Wyatt 2015; Mota, Sridharan, and Leonardis 2021). Also here, applications of abstraction techniques can be found, for example when discovering and learning new actions (Sridharan and Meadows 2018). Finally, Saad (2011) employ ASP to encode and solve RMDPs in their entirety, including histories and value functions.

Outside of reinforcement learning, a body of work exists on *abstract answer-set programs*, making use of approaches such as the clustering of domain objects (Saribatur, Eiter, and Schüller 2021) and the omission of atoms (Saribatur and Eiter 2021). Potential applications for such abstractions include also planning problems (Saribatur 2020).



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Conclusion and Future Work

In this chapter, we reflect on the objectives and results of our research, draw conclusions and provide ideas for future work.

**Research Objectives and Results.** The aim of this thesis was to explore use cases for ASP in reinforcement learning and in particular as a representation method for state and state-action pair abstractions applied to relational Markov decision processes. Focusing on blocks world and Minigrad task environments, we stated the following central research question.

*For the mentioned task environments, can we build ASP-based state-action pair abstractions and, using abstract Q-learning with an  $\epsilon$ -greedy exploration policy, consistently learn policies of acceptable quality while achieving a significant increase in sample efficiency, compared to concrete Q-learning?*

To achieve our research goals, we devised a method to encode state-action pair abstractions in ASP and implemented a proof of concept. We used the resulting framework to model ASP-based abstractions for tasks in blocks world and Minigrad environments. The abstractions were evaluated using empirical methods, comparing concrete and abstract versions of Q-learning in terms of their sample efficiency, stability, and the quality of the produced policies.

The results show that, under certain problem-specific parameter configurations, abstract Q-learning reliably produces policies of acceptable quality and in a smaller number of episodes as compared to concrete Q-learning, for all tested tasks and environments. This is consistent with the presented theory: the increase in sample efficiency can be explained in part by the reduction in size of the state-action space due to the abstract representation. We identified two additional factors with a possible effect on sample efficiency, namely

an exploration bias in the blocks world abstraction and the restricted admissibility of actions in the Minigrd abstraction. Summing up, we can positively answer our central research question.

**Generalisability and Impact.** Considering the presented results in isolation however, we cannot claim generalisability beyond the tested parameter combinations and task environments. Detailed discussions about the used parameter combinations can be found in the corresponding chapters. For blocks world environments, only a single task was tested, namely to stack all blocks into one tower with a specific order. This task is not representative of the full range of (NP-Hard) blocks world planning tasks and more experimentation is needed to see how abstract  $Q$ -learning behaves for other goal configurations. For the Minigrd setting, three task environments with a fully observable room layout and a similar goal condition were tested. These environments do not capture the variety of available task environments, which are by default partially observable and include instructions in natural language. Furthermore, there exist task environments (e.g. of the *Crossing* type), where the presented abstraction is not directly usable and we expect the  $\epsilon$ -greedy policy to be insufficient for exploring some of the more complex task environments.

It is also worth noting some of the commonalities between the chosen environments. Both blocks world and Minigrd environments have dynamics and reward structures that are idealised versions of real-world problems. They are essentially deterministic and clearly defined formally or in the source code, which made it easier to design the abstractions. The reward structures are also similar in that they produce positive rewards only if a certain goal state is reached and rewards of zero or small negative rewards otherwise. This does not reflect the full range of relational Markov decision processes that can theoretically be defined. For task environments encountered in real-world applications, we expect both the dynamics and reward structures to be more complex and ambiguous, making it difficult to manually find suitable abstractions of similar effectiveness.

The results regarding the stability of abstract  $Q$ -learning are however consistent with previous empirical research, even in the absence of theoretical guarantees. Indeed, Sutton and Barto (2018, p. 263) conjectured that convergence can be guaranteed in the case of linear function approximation (which generalises abstraction) for the special case of using  $Q$ -learning with an  $\epsilon$ -greedy exploration policy, pointing out that they are not aware of any observed cases of divergence for this case.

**Conclusion.** We explored ASP as a means of modelling state-action pair abstractions for relational Markov decision processes. To this end, we devised a general method for encoding abstractions, implemented a proof of concept, and used it to model and test two specific abstractions, one for blocks worlds and one for grid worlds from the Minigrd library. An empirical analysis showed that, using  $Q$ -learning on the abstract representations, policies of acceptable quality could be learned with high consistency. These policies could also be obtained in a smaller number of samples than what was

---

needed to learn comparable policies without the abstraction. Our empirical results are consistent with both the considered theory of abstraction and a large body of previous empirical research but additional studies are needed to test the viability of our approach in real-world applications. To our knowledge, this thesis is the first piece of research to consider ASP as a representation method for state-action pair abstractions in this setting, highlighting the potential usefulness of this elaboration tolerant problem solving approach thanks to its commitment to declarativity and its non-monotonic reasoning capabilities.

**Future Work.** Some related state-of-the-art research considers automatically generated abstractions as a preprocessing technique before applying more sophisticated function approximation methods. Both the automated generation of ASP-based abstractions in the reinforcement learning setting, e.g by making use of the abstraction refinement methodology presented by Saribatur, Eiter, and Schüller (2021), and their integration into larger state-of-the-art reinforcement learning frameworks are possible future directions of research.

Another possibility is to investigate the transfer capabilities of the learned abstract policies by testing their performance in previously unseen tasks and environments, such as on blocks worlds with a different numbers of blocks or with different goal configurations (where preliminary experiments show promise). For Minigrad environments, the learned policies can be tested in larger task environments of the same type as the ones they were trained in.

Further, the evaluation of our approach should be extended to consider the overall computational effort needed during learning, which is a function of the sample requirements and the computational effort needed for every time point. In the current framework, both grounding and model search are executed for every time point, having a clear impact on the computational effort. In this context it is also worth exploring the efficiency of our encodings and more sophisticated solving techniques, such as *multi-shot solving* and *solving under assumptions* (Gebser, Kaminski, Kaufmann, and Schaub 2019).

Finally, there are many ways to strengthen the results of this research, for example by considering other parameter configurations, task environments, exploration techniques, and learning algorithms, or by refining our research constructs.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Overview of Generative AI Tools Used

*GPT-5*<sup>1</sup>, *GPT-4o mini*<sup>2</sup>, and earlier versions were used as search engines. They were also used to answer questions about English grammar, spelling, and style in the context of individual sentences.

---

<sup>1</sup><https://chatgpt.com/>

<sup>2</sup><https://duck.ai/>



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Bibliography

- Alviano, Mario et al. (2017). „The ASP System DLV2“. In: *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*. Ed. by Marcello Balduccini and Tomi Janhunen. Vol. 10377. Lecture Notes in Computer Science. Springer, pp. 215–221. DOI: 10.1007/978-3-319-61660-5\\_19. URL: [https://doi.org/10.1007/978-3-319-61660-5\\\_19](https://doi.org/10.1007/978-3-319-61660-5%5C_19).
- Baader, Franz et al., eds. (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press. ISBN: 0-521-78176-0.
- Beierle, Christoph and Gabriele Kern-Isberner (2019). *Methoden wissensbasierter Systeme - Grundlagen, Algorithmen, Anwendungen, 6. Auflage*. Computational intelligence. SpringerVieweg. ISBN: 978-3-658-27083-4. DOI: 10.1007/978-3-658-27084-1. URL: <https://doi.org/10.1007/978-3-658-27084-1>.
- Bellman, Richard (1957). *Dynamic programming*. Princeton, New Jersey: Princeton Univ. Press.
- Bertsekas, Dimitri P. and John N. Tsitsiklis (1996). *Neuro-dynamic programming*. Vol. 3. Optimization and neural computation series. Athena Scientific. ISBN: 1886529108. URL: <https://www.worldcat.org/oclc/35983505>.
- Bloch, Mitchell Keith (2018). „Computationally Efficient Relational Reinforcement Learning“. PhD thesis. University of Michigan, USA. URL: <https://hdl.handle.net/2027.42/145859>.
- Brewka, Gerhard, Thomas Eiter, and Mirosław Truszczyński (2011). „Answer set programming at a glance“. In: *Commun. ACM* 54.12, pp. 92–103. DOI: 10.1145/2043174.2043195. URL: <https://doi.org/10.1145/2043174.2043195>.
- Calimeri, Francesco et al. (2020). „ASP-Core-2 Input Language Format“. In: *Theory Pract. Log. Program.* 20.2, pp. 294–309. DOI: 10.1017/S1471068419000450. URL: <https://doi.org/10.1017/S1471068419000450>.
- Camacho, Alberto et al. (2019). „LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning“. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, pp. 6065–6073. DOI: 10.24963/IJCAI.2019/840. URL: <https://doi.org/10.24963/ijcai.2019/840>.

- Chevalier-Boisvert, Maxime et al. (2023). „Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks“. In: *CoRR* abs/2306.13831. DOI: 10.48550/ARXIV.2306.13831. arXiv: 2306.13831. URL: <https://doi.org/10.48550/arXiv.2306.13831>.
- Crosby, Matthew et al. (2019). „The Animal-AI Testbed and Competition“. In: *NeurIPS 2019 Competition and Demonstration Track, 8-14 December 2019, Vancouver, Canada. Revised selected papers*. Ed. by Hugo Jair Escalante and Raia Hadsell. Vol. 123. Proceedings of Machine Learning Research. PMLR, pp. 164–176. URL: <http://proceedings.mlr.press/v123/crosby20a.html>.
- Dantsin, Evgeny et al. (2001). „Complexity and expressive power of logic programming“. In: *ACM Comput. Surv.* 33.3, pp. 374–425. DOI: 10.1145/502807.502810. URL: <https://doi.org/10.1145/502807.502810>.
- Das, Srijita et al. (2020). „Fitted Q-Learning for Relational Domains“. In: *CoRR* abs/2006.05595. arXiv: 2006.05595. URL: <https://arxiv.org/abs/2006.05595>.
- Datler, Elias (Aug. 2020). „Balancing Learning and Planning. A Case Study on Combining Reinforcement Learning With Answer Set Programming in the Blocks World“. Bachelor’s Thesis. Vienna, Austria: Technische Universität Wien. URL: <https://fuxgeist.com/thesis.pdf> (visited on 03/25/2021).
- Dong, Honghua et al. (2019). „Neural Logic Machines“. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: <https://openreview.net/forum?id=B1xY-hRctX>.
- Driessens, Kurt and Jan Ramon (2003). „Relational Instance Based Regression for Relational Reinforcement Learning“. In: *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*. Ed. by Tom Fawcett and Nina Mishra. AAAI Press, pp. 123–130. URL: <http://www.aaai.org/Library/ICML/2003/icml03-019.php>.
- Dzeroski, Saso, Luc De Raedt, and Hendrik Blockeel (1998). „Relational Reinforcement Learning“. In: *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998*. Ed. by Jude W. Shavlik. Morgan Kaufmann, pp. 136–143.
- Džeroski, Sašo, Luc De Raedt, and Kurt Driessens (2001). „Relational Reinforcement Learning“. In: *Mach. Learn.* 43.1/2, pp. 7–52. DOI: 10.1023/A:1007694015589. URL: <https://doi.org/10.1023/A:1007694015589>.
- Eid, Michael, Mario Gollwitzer, and Manfred Schmitt (2013). *Statistik und Forschungsmethoden: Lehrbuch, 3. korrigierte Auflage*. Beltz. ISBN: 978-3-621-27524-8.
- Eiter, Thomas and Georg Gottlob (1995). „On the Computational Cost of Disjunctive Logic Programming: Propositional Case“. In: *Ann. Math. Artif. Intell.* 15.3-4, pp. 289–323. DOI: 10.1007/BF01536399. URL: <https://doi.org/10.1007/BF01536399>.
- Eiter, Thomas, Giovambattista Ianni, and Thomas Krennwallner (2009). „Answer Set Programming: A Primer“. In: *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August*

30 - September 4, 2009, Tutorial Lectures. Ed. by Sergio Tessaris et al. Vol. 5689. Lecture Notes in Computer Science. Springer, pp. 40–110. DOI: 10.1007/978-3-642-03754-2\\_2. URL: [https://doi.org/10.1007/978-3-642-03754-2%5C\\_2](https://doi.org/10.1007/978-3-642-03754-2%5C_2).

Erdem, Esra, Michael Gelfond, and Nicola Leone (2016). „Applications of Answer Set Programming“. In: *AI Mag.* 37.3, pp. 53–68. DOI: 10.1609/aimag.v37i3.2678. URL: <https://doi.org/10.1609/aimag.v37i3.2678>.

Faber, Wolfgang, Nicola Leone, and Gerald Pfeifer (2004). „Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity“. In: *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*. Ed. by José Júlio Alferes and João Alexandre Leite. Vol. 3229. Lecture Notes in Computer Science. Springer, pp. 200–212. DOI: 10.1007/978-3-540-30227-8\\_19. URL: [https://doi.org/10.1007/978-3-540-30227-8%5C\\_19](https://doi.org/10.1007/978-3-540-30227-8%5C_19).

Faber, Wolfgang, Gerald Pfeifer, and Nicola Leone (2011). „Semantics and complexity of recursive aggregates in answer set programming“. In: *Artif. Intell.* 175.1, pp. 278–298. DOI: 10.1016/J.ARTINT.2010.04.002. URL: <https://doi.org/10.1016/j.artint.2010.04.002>.

Falkner, Andreas A. et al. (2018). „Industrial Applications of Answer Set Programming“. In: *Künstliche Intell.* 32.2-3, pp. 165–176. DOI: 10.1007/s13218-018-0548-6. URL: <https://doi.org/10.1007/s13218-018-0548-6>.

Farama Foundation (2023a). *Gymnasium Documentation*. URL: <https://gymnasium.farama.org/> (visited on 06/03/2024).

— (2023b). *Minigrid Documentation*. URL: <https://minigrid.farama.org/> (visited on 04/25/2024).

— (2024). *Minigrid Source Code*. URL: <https://github.com/Farama-Foundation/Minigrid> (visited on 04/25/2024).

Ferreira, Leonardo Anjoletto et al. (2017). „Answer set programming for non-stationary Markov decision processes“. In: *Appl. Intell.* 47.4, pp. 993–1007. DOI: 10.1007/s10489-017-0988-y. URL: <https://doi.org/10.1007/s10489-017-0988-y>.

Garg, Sankalp, Aniket Bajpai, and Mausam (2020). „Symbolic Network: Generalized Neural Policies for Relational MDPs“. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, pp. 3397–3407. URL: <http://proceedings.mlr.press/v119/garg20a.html>.

Gärtner, Thomas, Kurt Driessens, and Jan Ramon (2003). „Graph Kernels and Gaussian Processes for Relational Reinforcement Learning“. In: *Inductive Logic Programming: 13th International Conference, ILP 2003, Szeged, Hungary, September 29-October 1, 2003, Proceedings*. Ed. by Tamás Horváth. Vol. 2835. Lecture Notes in Computer Science. Springer, pp. 146–163. DOI: 10.1007/978-3-540-39917-9\\_11. URL: [https://doi.org/10.1007/978-3-540-39917-9%5C\\_11](https://doi.org/10.1007/978-3-540-39917-9%5C_11).

- Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, et al. (2016). „Theory Solving Made Easy with Clingo 5“. In: *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*. Ed. by Manuel Carro et al. Vol. 52. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:15. DOI: 10.4230/OASICS.ICLP.2016.2. URL: <https://doi.org/10.4230/OASICS.ICLP.2016.2>.
- Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub (2012). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers. ISBN: 978-3-031-00433-9. DOI: 10.2200/S00457ED1V01Y201211AIM019. URL: <https://doi.org/10.2200/S00457ED1V01Y201211AIM019>.
- (2019). „Multi-shot ASP solving with clingo“. In: *Theory Pract. Log. Program.* 19.1, pp. 27–82. DOI: 10.1017/S1471068418000054. URL: <https://doi.org/10.1017/S1471068418000054>.
- Gebser, Martin, Roland Kaminski, and Torsten Schaub (2011). „Complex optimization in answer set programming“. In: *Theory Pract. Log. Program.* 11.4-5, pp. 821–839. DOI: 10.1017/S1471068411000329. URL: <https://doi.org/10.1017/S1471068411000329>.
- Gebser, Martin, Marco Maratea, and Francesco Ricca (2020). „The Seventh Answer Set Programming Competition: Design and Results“. In: *Theory Pract. Log. Program.* 20.2, pp. 176–204. DOI: 10.1017/S1471068419000061. URL: <https://doi.org/10.1017/S1471068419000061>.
- Gelfond, Michael and Vladimir Lifschitz (1988). „The Stable Model Semantics for Logic Programming“. In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*. Ed. by Robert A. Kowalski and Kenneth A. Bowen. MIT Press, pp. 1070–1080.
- (1991). „Classical Negation in Logic Programs and Disjunctive Databases“. In: *New Gener. Comput.* 9.3/4, pp. 365–386. DOI: 10.1007/BF03037169. URL: <https://doi.org/10.1007/BF03037169>.
- Giunchiglia, Fausto and Toby Walsh (1992). „A Theory of Abstraction“. In: *Artif. Intell.* 57.2-3, pp. 323–389. DOI: 10.1016/0004-3702(92)90021-0. URL: [https://doi.org/10.1016/0004-3702\(92\)90021-0](https://doi.org/10.1016/0004-3702(92)90021-0).
- Givan, Robert, Thomas L. Dean, and Matthew Greig (2003). „Equivalence notions and model minimization in Markov decision processes“. In: *Artif. Intell.* 147.1-2, pp. 163–223. DOI: 10.1016/S0004-3702(02)00376-4. URL: [https://doi.org/10.1016/S0004-3702\(02\)00376-4](https://doi.org/10.1016/S0004-3702(02)00376-4).
- Goetschalckx, Robby (Sept. 2009). „The Use of Domain Knowledge in Reinforcement Learning (Het gebruik van domeinkennis in reinforcement learning)“. PhD thesis. Leuven, Belgium: Katholieke Universiteit Leuven. ISBN: 978-94-6018-125-2.
- Grounds, Matthew Jon and Daniel Kudenko (2007). „Combining Reinforcement Learning with Symbolic Planning“. In: *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning, 5th, 6th, and 7th European Symposium, ALA-MAS 2005-2007 on Adaptive and Learning Agents and Multi-Agent Systems, Revised*

- Selected Papers*. Ed. by Karl Tuyls et al. Vol. 4865. Lecture Notes in Computer Science. Springer, pp. 75–86. DOI: 10.1007/978-3-540-77949-0\_6. URL: [https://doi.org/10.1007/978-3-540-77949-0%5C\\_6](https://doi.org/10.1007/978-3-540-77949-0%5C_6).
- Gupta, Naresh and Dana S. Nau (1991). „Complexity Results for Blocks-World Planning“. In: *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 2*. Ed. by Thomas L. Dean and Kathleen R. McKeown. AAAI Press / The MIT Press, pp. 629–633. URL: <http://www.aaai.org/Library/AAAI/1991/aaai91-098.php>.
- (1992). „On the Complexity of Blocks-World Planning“. In: *Artif. Intell.* 56.2-3, pp. 223–254. DOI: 10.1016/0004-3702(92)90028-v. URL: [https://doi.org/10.1016/0004-3702\(92\)90028-v](https://doi.org/10.1016/0004-3702(92)90028-v).
- Hazra, Rishi and Luc De Raedt (2023). „Deep Explainable Relational Reinforcement Learning: A Neuro-Symbolic Approach“. In: *Machine Learning and Knowledge Discovery in Databases: Research Track - European Conference, ECML PKDD 2023, Turin, Italy, September 18-22, 2023, Proceedings, Part IV*. Ed. by Danai Koutra et al. Vol. 14172. Lecture Notes in Computer Science. Springer, pp. 213–229. DOI: 10.1007/978-3-031-43421-1\_13. URL: [https://doi.org/10.1007/978-3-031-43421-1%5C\\_13](https://doi.org/10.1007/978-3-031-43421-1%5C_13).
- Hutter, Marcus (2009). „Feature Reinforcement Learning: Part I. Unstructured MDPs“. In: *J. Artif. Gen. Intell.* 1.1, pp. 3–24. DOI: 10.2478/V10229-011-0002-8. URL: <https://doi.org/10.2478/v10229-011-0002-8>.
- Icarte, Rodrigo Toro et al. (2019). „Learning Reward Machines for Partially Observable Reinforcement Learning“. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach et al., pp. 15497–15508. URL: <https://proceedings.neurips.cc/paper/2019/hash/532435c44bec236b471a47a88d63513d-Abstract.html>.
- Jaakkola, Tommi S., Michael I. Jordan, and Satinder P. Singh (1994). „On the Convergence of Stochastic Iterative Dynamic Programming Algorithms“. In: *Neural Comput.* 6.6, pp. 1185–1201. DOI: 10.1162/NECO.1994.6.6.1185. URL: <https://doi.org/10.1162/neco.1994.6.6.1185>.
- Janisch, Jaromír, Tomáš Pevný, and Viliam Lisý (2020). „Symbolic Relational Deep Reinforcement Learning based on Graph Neural Networks“. In: *CoRR* abs/2009.12462. arXiv: 2009.12462. URL: <https://arxiv.org/abs/2009.12462>.
- Kaelbling, Leslie Pack, Michael L. Littman, and Anthony R. Cassandra (1995). „Partially Observable Markov Decision Processes for Artificial Intelligence“. In: *KI-95: Advances in Artificial Intelligence, 19th Annual German Conference on Artificial Intelligence, Bielefeld, Germany, September 11-13, 1995, Proceedings*. Ed. by Ipke Wachsmuth, Claus-Rainer Rollinger, and Wilfried Brauer. Vol. 981. Lecture Notes in Computer Science. Springer, pp. 1–17. DOI: 10.1007/3-540-60343-3\_22. URL: [https://doi.org/10.1007/3-540-60343-3%5C\\_22](https://doi.org/10.1007/3-540-60343-3%5C_22).

- Kakade, Sham Machandranath (Mar. 2003). „On the sample complexity of reinforcement learning“. PhD thesis. London, UK: University College London. URL: <https://discovery.ucl.ac.uk/id/eprint/10100726>.
- Karia, Rushang and Siddharth Srivastava (2022). „Relational Abstractions for Generalized Reinforcement Learning on Symbolic Problems“. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. Ed. by Luc De Raedt. ijcai.org, pp. 3135–3142. DOI: 10.24963/IJCAI.2022/435. URL: <https://doi.org/10.24963/ijcai.2022/435>.
- Kautz, Henry A. and Bart Selman (1996). „Pushing the Envelope: Planning, Propositional Logic and Stochastic Search“. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*. Ed. by William J. Clancey and Daniel S. Weld. AAAI Press / The MIT Press, pp. 1194–1201. URL: <http://www.aaai.org/Library/AAAI/1996/aaai96-177.php>.
- Kersting, Kristian and Luc De Raedt (2004). „Logical Markov Decision Programs and the Convergence of Logical TD( $\lambda$ )“. In: *Inductive Logic Programming, 14th International Conference, ILP 2004, Porto, Portugal, September 6-8, 2004, Proceedings*. Ed. by Rui Camacho, Ross D. King, and Ashwin Srinivasan. Vol. 3194. Lecture Notes in Computer Science. Springer, pp. 180–197. DOI: 10.1007/978-3-540-30109-7\\_16. URL: [https://doi.org/10.1007/978-3-540-30109-7%5C\\_16](https://doi.org/10.1007/978-3-540-30109-7%5C_16).
- Kersting, Kristian, Martijn van Otterlo, and Luc De Raedt (2004). „Bellman goes relational“. In: *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*. Ed. by Carla E. Brodley. Vol. 69. ACM International Conference Proceeding Series. ACM. DOI: 10.1145/1015330.1015401. URL: <https://doi.org/10.1145/1015330.1015401>.
- Koga, Marcelo Li, Valdinei Freire da Silva, and Anna Helena Reali Costa (2015). „Stochastic Abstract Policies: Generalizing Knowledge to Improve Reinforcement Learning“. In: *IEEE Trans. Cybern.* 45.1, pp. 77–88. DOI: 10.1109/TCYB.2014.2319733. URL: <https://doi.org/10.1109/TCYB.2014.2319733>.
- Kokel, Harsha, Arjun Manoharan, et al. (2021). „RePreL: Integrating Relational Planning and Reinforcement Learning for Effective Abstraction“. In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*. Ed. by Susanne Biundo et al. AAAI Press, pp. 533–541. URL: <https://ojs.aaai.org/index.php/ICAPS/article/view/16001>.
- Kokel, Harsha, Siraam Natarajan, et al. (2023). „RePreL: a unified framework for integrating relational planning and reinforcement learning for effective abstraction in discrete and continuous domains“. In: *Neural Comput. Appl.* 35.23, pp. 16877–16892. DOI: 10.1007/s00521-022-08119-y. URL: <https://doi.org/10.1007/s00521-022-08119-y>.

- Kowalski, Robert A. (1979). „Algorithm = Logic + Control“. In: *Commun. ACM* 22.7, pp. 424–436. DOI: 10.1145/359131.359136. URL: <https://doi.org/10.1145/359131.359136>.
- Law, Mark, Alessandra Russo, and Krysia Broda (2020). „The ILASP system for Inductive Learning of Answer Set Programs“. In: *CoRR* abs/2005.00904. arXiv: 2005.00904. URL: <https://arxiv.org/abs/2005.00904>.
- Leone, Nicola et al. (2006). „The DLV system for knowledge representation and reasoning“. In: *ACM Trans. Comput. Log.* 7.3, pp. 499–562. DOI: 10.1145/1149114.1149117. URL: <https://doi.org/10.1145/1149114.1149117>.
- Li, Lihong (2012). „Sample Complexity Bounds of Exploration“. In: *Adaptation, Learning, and Optimization* 12. Ed. by Marco A. Wiering and Martijn van Otterlo, pp. 175–204. DOI: 10.1007/978-3-642-27645-3\_6. URL: [https://doi.org/10.1007/978-3-642-27645-3\\_6](https://doi.org/10.1007/978-3-642-27645-3_6).
- Li, Lihong, Thomas J. Walsh, and Michael L. Littman (2006). „Towards a Unified Theory of State Abstraction for MDPs“. In: *International Symposium on Artificial Intelligence and Mathematics, AI&Math 2006, Fort Lauderdale, Florida, USA, January 4-6, 2006*. URL: <http://anytime.cs.umass.edu/aimath06/proceedings/P21.pdf>.
- Lifschitz, Vladimir (2019). *Answer Set Programming*. Springer. ISBN: 978-3-030-24657-0. DOI: 10.1007/978-3-030-24658-7. URL: <https://doi.org/10.1007/978-3-030-24658-7>.
- Lyu, Daoming et al. (2019). „SDRL: Interpretable and Data-Efficient Deep Reinforcement Learning Leveraging Symbolic Planning“. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, pp. 2970–2977. DOI: 10.1609/AAAI.V33I01.33012970. URL: <https://doi.org/10.1609/aaai.v33i01.33012970>.
- Majeed, Sultan Javed and Marcus Hutter (2018). „On Q-learning Convergence for Non-Markov Decision Processes“. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, pp. 2546–2552. DOI: 10.24963/IJCAI.2018/353. URL: <https://doi.org/10.24963/ijcai.2018/353>.
- Marek, Victor W. and Miroslaw Truszczyński (1998). „Stable models and an alternative logic programming paradigm“. In: *CoRR* cs.LO/9809032. URL: <https://arxiv.org/abs/cs/9809032>.
- McCarthy, John (1998). „Elaboration tolerance“. In: *Formalization of Commonsense Reasoning (FCS-98), London, England, January 7-9, 1998*, pp. 198–216. URL: <http://jmc.stanford.edu/articles/elaboration.html> (visited on 08/20/2025).
- Mitchener, Ludovico et al. (2022). „Detect, Understand, Act: A Neuro-symbolic Hierarchical Reinforcement Learning Framework“. In: *Mach. Learn.* 111.4, pp. 1523–1549. DOI: 10.1007/s10994-022-06142-7. URL: <https://doi.org/10.1007/s10994-022-06142-7>.

- Moore, Harry Paul (Oct. 2018). „Reinforcement Learning in Robotic Task Domains with Deitic Descriptor Representation.“ PhD thesis. Louisiana State University, Agricultural, and Mechanical College.
- Morales, Eduardo F. (2003). „Scaling up reinforcement learning with a relational representation“. In: *Proceedings of the Workshop on Adaptability in Multi-Agent Systems at AORC 2003, Sydney, Australia*.
- Mota, Tiago, Mohan Sridharan, and Ales Leonardis (2021). „Integrated Commonsense Reasoning and Deep Learning for Transparent Decision Making in Robotics“. In: *SN Comput. Sci.* 2.4, p. 242. DOI: 10.1007/s42979-021-00573-0. URL: <https://doi.org/10.1007/s42979-021-00573-0>.
- Nickles, Matthias (2011). „Integrating Relational Reinforcement Learning with Reasoning about Actions and Change“. In: *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*. Ed. by Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi. Vol. 7207. Lecture Notes in Computer Science. Springer, pp. 255–269. DOI: 10.1007/978-3-642-31951-8\_23. URL: [https://doi.org/10.1007/978-3-642-31951-8\\_23](https://doi.org/10.1007/978-3-642-31951-8_23).
- Nienhuys-Cheng, Shan-Hwei and Ronald de Wolf (1997). *Foundations of Inductive Logic Programming*. Vol. 1228. Lecture Notes in Computer Science. Springer. ISBN: 3-540-62927-0. DOI: 10.1007/3-540-62927-0. URL: <https://doi.org/10.1007/3-540-62927-0>.
- Papadimitriou, Christos H. and John N. Tsitsiklis (1987). „The Complexity of Markov Decision Processes“. In: *Math. Oper. Res.* 12.3, pp. 441–450. DOI: 10.1287/MOOR.12.3.441. URL: <https://doi.org/10.1287/moor.12.3.441>.
- Pateria, Shubham et al. (2022). „Hierarchical Reinforcement Learning: A Comprehensive Survey“. In: *ACM Comput. Surv.* 54.5, 109:1–109:35. DOI: 10.1145/3453160. URL: <https://doi.org/10.1145/3453160>.
- Ravindran, Balaraman (Feb. 2004). „An Algebraic Approach to Abstraction in Reinforcement Learning“. PhD thesis. University of Massachusetts Amherst.
- Ravindran, Balaraman and Andrew G. Barto (2003). „SMDP Homomorphisms: An Algebraic Approach to Abstraction in Semi-Markov Decision Processes“. In: *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*. Ed. by Georg Gottlob and Toby Walsh. Morgan Kaufmann, pp. 1011–1018. URL: <http://ijcai.org/Proceedings/03/Papers/145.pdf>.
- Roy, Benjamin Van (2006). „Performance Loss Bounds for Approximate Value Iteration with State Aggregation“. In: *Math. Oper. Res.* 31.2, pp. 234–244. DOI: 10.1287/MOOR.1060.0188. URL: <https://doi.org/10.1287/moor.1060.0188>.
- Russell, Stuart and Peter Norvig (2013). *Artificial Intelligence: A Modern Approach*. 3rd ed. London, UK: Pearson. ISBN: 1292024208.
- Saad, Emad (2011). „Bridging the Gap between Reinforcement Learning and Knowledge Representation: A Logical Off- and On-Policy Framework“. In: *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 11th European Conference,*

- ECSQARU 2011, Belfast, UK, June 29-July 1, 2011. Proceedings*. Ed. by Weiru Liu. Vol. 6717. Lecture Notes in Computer Science. Springer, pp. 472–484. DOI: 10.1007/978-3-642-22152-1\_40. URL: [https://doi.org/10.1007/978-3-642-22152-1\\_40](https://doi.org/10.1007/978-3-642-22152-1%5C_40).
- Sanner, Scott (2010). „Relational Dynamic Influence Diagram Language (RDDL): Language Description“. URL: [https://users.cecs.anu.edu.au/~ssanner/IPPC\\_2011/RDDL.pdf](https://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf) (visited on 08/20/2025).
- Sanner, Scott and Craig Boutilier (2009). „Practical solution techniques for first-order MDPs“. In: *Artif. Intell.* 173.5-6, pp. 748–788. DOI: 10.1016/J.ARTINT.2008.11.003. URL: <https://doi.org/10.1016/j.artint.2008.11.003>.
- Santos, Jose and Stephen H. Muggleton (2010). „Subsumer: A Prolog theta-subsumption engine“. In: *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*. Ed. by Manuel V. Hermenegildo and Torsten Schaub. Vol. 7. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 172–181. DOI: 10.4230/LIPICS.ICLP.2010.172. URL: <https://doi.org/10.4230/LIPICS.ICLP.2010.172>.
- Saribatur, Zeynep G. (2020). „Abstraction for ASP Planning“. In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by Giuseppe De Giacomo et al. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 2933–2934. DOI: 10.3233/FAIA200460. URL: <https://doi.org/10.3233/FAIA200460>.
- Saribatur, Zeynep G. and Thomas Eiter (2021). „Omission-Based Abstraction for Answer Set Programs“. In: *Theory Pract. Log. Program.* 21.2, pp. 145–195. DOI: 10.1017/S1471068420000095. URL: <https://doi.org/10.1017/S1471068420000095>.
- Saribatur, Zeynep G., Thomas Eiter, and Peter Schüller (2021). „Abstraction for non-ground answer set programs“. In: *Artif. Intell.* 300, p. 103563. DOI: 10.1016/J.ARTINT.2021.103563. URL: <https://doi.org/10.1016/j.artint.2021.103563>.
- Schickinger, Thomas and Angelika Steger (2002). *Diskrete Strukturen. Wahrscheinlichkeitstheorie und Statistik*. Vol. 2. Springer-Verlag Berlin Heidelberg.
- Sharma, Vishal et al. (2023). „SymNet 3.0: Exploiting Long-Range Influences in Learning Generalized Neural Policies for Relational MDPs“. In: *Uncertainty in Artificial Intelligence, UAI 2023, July 31 - 4 August 2023, Pittsburgh, PA, USA*. Ed. by Robin J. Evans and Ilya Shpitser. Vol. 216. Proceedings of Machine Learning Research. PMLR, pp. 1921–1931. URL: <https://proceedings.mlr.press/v216/sharma23c.html>.
- Singh, Satinder, Tommi S. Jaakkola, and Michael I. Jordan (1994). „Reinforcement Learning with Soft State Aggregation“. In: *Advances in Neural Information Processing Systems 7, [NIPS Conference, Denver, Colorado, USA, 1994]*. Ed. by Gerald Tesauro, David S. Touretzky, and Todd K. Leen. MIT Press, pp. 361–368. URL:

[https://proceedings.neurips.cc/paper%5C\\_files/paper/1994/hash/287e03db1d99e0ec2edb90d079e142f3-Abstract.html](https://proceedings.neurips.cc/paper%5C_files/paper/1994/hash/287e03db1d99e0ec2edb90d079e142f3-Abstract.html).

- Slaney, John K. and Sylvie Thiébaux (2001). „Blocks World revisited“. In: *Artif. Intell.* 125.1-2, pp. 119–153. DOI: 10.1016/S0004-3702(00)00079-5. URL: [https://doi.org/10.1016/S0004-3702\(00\)00079-5](https://doi.org/10.1016/S0004-3702(00)00079-5).
- Sridharan, Mohan and Benjamin Meadows (2018). „Knowledge Representation and Interactive Learning of Domain Knowledge for Human-Robot Interaction“. In: *Advances in Cognitive Systems* 7, pp. 77–96. URL: <http://www.cogsys.org/journal/volume7/> (visited on 08/20/2025).
- Strehl, Alexander L., Lihong Li, and Michael L. Littman (2009). „Reinforcement Learning in Finite MDPs: PAC Analysis“. In: *J. Mach. Learn. Res.* 10, pp. 2413–2444. DOI: 10.5555/1577069.1755867. URL: <https://dl.acm.org/doi/10.5555/1577069.1755867>.
- Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement learning - an introduction, 2nd Edition*. MIT Press. URL: <http://www.incompleteideas.net/book/the-book-2nd.html>.
- Taitler, Ayal, Ron Alford, et al. (2024). „The 2023 International Planning Competition“. In: *AI Mag.* 45.2, pp. 280–296. DOI: 10.1002/AAAI.12169. URL: <https://doi.org/10.1002/aaai.12169>.
- Taitler, Ayal, Michael Gimelfarb, et al. (2022). „pyRDDL Gym: From RDDL to Gym Environments“. In: *CoRR* abs/2211.05939. DOI: 10.48550/ARXIV.2211.05939. arXiv: 2211.05939. URL: <https://doi.org/10.48550/arXiv.2211.05939>.
- Tsitsiklis, John N. (1994). „Asynchronous Stochastic Approximation and Q-Learning“. In: *Mach. Learn.* 16.3, pp. 185–202. DOI: 10.1007/BF00993306. URL: <https://doi.org/10.1007/BF00993306>.
- van Otterlo, Martijn (2003). „Efficient Reinforcement Learning using Relational Aggregation“. In: *Proceedings of the Sixth European Workshop on Reinforcement Learning, EWRL-6, Nancy, France, September 4-5, 2003*.
- (2004). „Reinforcement Learning for Relational MDPs“. In: *Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands (BeNeLearn 2004), January 8-9, 2004*. Ed. by Ann Nowé, Tom Lenaerts, and Kris Steenhaut, pp. 138–145. URL: <https://web.archive.org/web/20040715063749/http://benelearn04.vub.ac.be/BENELEARN.pdf> (visited on 08/20/2025).
- (May 2008). „The logic of adaptive behavior. Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains“. PhD thesis. Enschede, the Netherlands: University of Twente. ISBN: 978-90-365-2677-7.
- (2012). „Solving Relational and First-Order Logical Markov Decision Processes: A Survey“. In: *Reinforcement Learning*. Ed. by Marco A. Wiering and Martijn van Otterlo. Vol. 12. Adaptation, Learning, and Optimization. Springer, pp. 253–292. DOI: 10.1007/978-3-642-27645-3\_8. URL: [https://doi.org/10.1007/978-3-642-27645-3%5C\\_8](https://doi.org/10.1007/978-3-642-27645-3%5C_8).

- Watkins, Christopher John Cornish Hellaby (May 1989). „Learning from Delayed Rewards“. PhD thesis. Cambridge, UK: King’s College. URL: [https://cs.rhul.ac.uk/~chrisw/new\\_thesis.pdf](https://cs.rhul.ac.uk/~chrisw/new_thesis.pdf).
- Yang, Fangkai et al. (2014). „Planning in Answer Set Programming while Learning Action Costs for Mobile Robots“. In: *2014 AAAI Spring Symposia, Stanford University, Palo Alto, California, USA, March 24-26, 2014*. AAAI Press. URL: <http://www.aaai.org/ocs/index.php/SSS/SSS14/paper/view/7727>.
- Younes, Håkan L. S. et al. (2005). „The First Probabilistic Track of the International Planning Competition“. In: *J. Artif. Intell. Res.* 24, pp. 851–887. DOI: 10.1613/JAIR.1880. URL: <https://doi.org/10.1613/jair.1880>.
- Zambaldi, Vinícius Flores et al. (2019). „Deep reinforcement learning with relational inductive biases“. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: <https://openreview.net/forum?id=HkxaFoC9KQ>.
- Zhang, Shiqi, Mohan Sridharan, and Jeremy L. Wyatt (2015). „Mixed Logical Inference and Probabilistic Planning for Robots in Unreliable Worlds“. In: *IEEE Trans. Robotics* 31.3, pp. 699–713. DOI: 10.1109/TRO.2015.2422531. URL: <https://doi.org/10.1109/TRO.2015.2422531>.