UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

Learning Portable Symbolic Representations

Steven James

475531

Supervised by: George Konidaris & Benjamin Rosman

October 2021

A thesis submitted to the Faculty of Science, University of the Witwatersrand,in fulfilment of the requirements for the degree of Doctor of Philosophy.

Abstract

A major goal of artificial intelligence (AI) is to create agents capable of acting effectively in a wide variety of complex environments. A popular framework for modelling decision-making agents is reinforcement learning (RL), where an agent learns from interaction with its environment. Though RL has proven successful in solving a number of challenging tasks, one major hurdle to the development of truly autonomous agents is the need to specify appropriate task representations. In RL, the most common approach is for a human designer to simply provide the agent with the task description by defining the state space, rewards, goals and actions available to the agent. While this approach is feasible within the bounds of narrowly defined tasks, it must clearly be dispensed with if we are ever to construct agents with full autonomy.

In this thesis, we concern ourselves with the question of how an agent can acquire its own representations from sensory data. We restrict our focus to learning representations for long-term planning, a class of problems that state-of-the-art learning methods are unable to solve. We take inspiration from the way humans reason about the world—although we must sense and act in the real world, we do not reason at such a low level. Rather, we use mental abstractions of our environment that ignore irrelevant minutiae. When acting, we can make use of abstraction to employ high-level skills, known in RL as options. By learning and planning with both state and action abstractions, we are ultimately able to construct plans consisting of thousands of actions.

Importantly, a feature of human intelligence is that we are proficient at a wide array of tasks. One key aspect that allows us to quickly solve new problems is our ability to reuse previously learned abstract representations. For example, once we acquire a conceptual representation of a *door*, we can simply apply this to any new doors we may encounter, independent of the lighting conditions, the location of the door or its colour. Since *tabula rasa* learning is infeasible for robots, learning transferable representations is key to scaling AI approaches to real-world agents.

We propose various methods for autonomously learning symbolic representations of an agent's environment. Importantly, these symbols are task-independent, and so can be recycled to solve new tasks. In particular, we make three main contributions. First, we demonstrate how an agent can use an existing set of options to acquire representations from egocentric observations. Since the resulting abstractions are agent-centric, they can immediately be reused by the same agent in new environments. We show how to combine these portable representations with problem-specific ones to generate a sound task description that can be used for abstract planning. Our results demonstrate that our approach allows an agent to transfer previous knowledge to new tasks, improving sample efficiency as the number of tasks increase.

Our second contribution is to leverage the fact that the real world consists of objects that an agent can observe and interact with. Based on this assumption, we show how to construct object-centric abstractions that can be used when an agent finds itself in a new task containing similar objects. As a result, an agent can convert observations from a high-dimensional environment (such as a video game) to object-centric textual symbols that can be given as input to classical planners. Once more, the transferability of the learned representations allows an agent to learn subsequent tasks using fewer environment observations.

Finally we show how to autonomously construct a multi-level hierarchy consisting of increasingly abstract representations. Since these hierarchies are transferable, higher-order concepts can be reused in new tasks, precluding the agent from relearning them and improving sample efficiency. The hierarchy further allows the agent to plan at a variety of levels, reducing the size of the problem and thereby improving planning efficiency.

Declaration

I, Steven James, hereby declare the contents of this thesis to be my own work unless otherwise explicitly referenced. This thesis is submitted for the degree of Doctor of Philosophy at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, nor for any other degree.

Signature

10/10/2021

Date

Acknowledgements

If I had to compare the last decade of my life to a movie, it would undoubtedly be *Sliding Doors*. This is true of everyone, of course, but I feel like it applies particularly to me. It is a testament to where I am, and the people I have met along the way, that I look back at these inflection points with absolute dread. Had Wits not offered the Applied Computing programme back in 2010, I may very well have become an actuary. Had said programme not been scrapped three days into my university career, I'd very well be an electrical engineer. Had Dean and Pravesh not created the 2nd year AI challenge, I may very well not have known Pravesh. Had I not known Pravesh, I wouldn't have been introduced to George. Had I not known George, I wouldn't have pursued an MSc. Had I not been around for Masters, I would never have met Benji. And so on, and so on. Looking back on all of these events, my heart starts to pound...how different things could have been. Imagine not knowing George, or Benji, or Pravesh! I do not like it. *I do not like it one bit!*[1]

To George: it is not an exaggeration to say that I am where I am because of you. I still remember our very first conversation all those years ago. My internet was down, so you called me on my phone and we discussed a potential topic for masters. I'll come clean and admit now that I had no idea what a *distribution over policies* was at the time, but your passion was absolutely contagious—you could have pitched me a machine learning approach to analysing sewer water, and I would have said yes. Thank you for all the time and effort, the years of weekly meetings and the in-person lab visits. If I can be half the researcher, the teacher, the supervisor and the friend you've been to me, I'd consider myself a wild success. It's been a wonderful 6 years, and I look forward to the next 60 (at least).

To Benji: I remember when you first returned to Wits. There was a department meeting in a small room in Senate House. Comparing where we are now with where we were then is absolutely mind-blowing, and it is in large part down to your leadership, your enthusiasm, and your desire to build something from the ground up. I am deeply grateful for everything you have done for me—sending me to IJCAI in 2016 was the moment I knew that *this* is what I wanted to do with the rest of my life. Thank you for all the advice, the support and the friendship over the past years. I look forward to decades more as well.

[1] This is why I try not to do too much introspection—bad for my heart.

Together, George and Benji make up the supervisor dream team. I scoff at the emails I receive inviting me to seminars and training on how to supervise students. What do I need that for?! I'll just copy George and Benji! I thoroughly recommend that everyone in the world pursue a PhD with them, and you can do so by calling them directly at the following numbers: **[redacted under threat of severe maiming]**.

Anyone who knows me will know that I *love* Wits. Now Wits itself is cool, but it's really about the Witsies themselves. Pravesh and Richard have been constants throughout my past ten years at Wits (on account of their being so old and all). Thank you for the friendship and camaraderie, and for showing me what it means to be great teachers. Despite being contrarians, thank you for also sometimes agreeing with me (c.f. Bitstream Charter).

Dean Wookey showed me what it meant to be truly altruistic. Thank you for all the time and effort you put into making everything and everyone around you better.

To Benji[2] and Adam: I'm glad I got to spend the majority of my PhD with you guys, and I hope we get to hang out in person very soon. I promise to make the research house a reality someday!

I have met so many people, and made so many friends, along the way: Matt, Andrew, Raph, Jeremy, Shahil, Kwanda, Ofir, Geraud, Branden, and the entire RAIL lab; Arnu, Elan, and the Stellenbosch crowd; Cam, Ben A., Ben B., Akhil, Dave, Barrett, Nakul, Vanya, Yuu, Eric, Max, and everyone at Brown. Thank you for your friendship and for being part of my journey.

To Shun and the rest of the MSS team (Brian, MJ and Senzo). Thank you for bending over backwards, forwards and sideways whenever I needed anything. It is no exaggeration to say that the department wouldn't be able to do the work we do without you guys. The experiments in this thesis required large amounts of computation, and would not have been possible without the support of the High Performance Computing infrastructure provided by the Mathematical Sciences Support unit at the University of the Witwatersrand. Large experiments were also made possible by the Centre for High Performance Computing, who are absolutely fantastic and should get all the public funding they want!

Last but not least, to my family and friends: thank you for your love and support throughout my life. To Ariela: even when none of my experiments worked and it was all going horribly wrong, you made everything alright. To my mom and dad: thank you for everything you have ever done for me, for all the love, and for giving me the freedom to pursue whatever my heart desired. Thank you for your infinite patience, and for always asking how things were going (even if you had to hear the standard response of: *I don't want to talk about it*). I am who I am because of you.

[2] A different Benji from the aforementioned one.

Funding Acknowledgements

The research presented in this thesis was supported in part by the National Research Foundation of South Africa (Grant Number 117808) and by a Google PhD Fellowship.

If you saw through my eyes
This view was worth the climb
Believe me when I say
There's no place in this world I'd rather be.

– Simple Plan & State Champs, *Where I Belong*

Contents

Chapter 1

Introduction

The ultimate goal of artificial intelligence (AI) is not just to understand the nature of intelligence, but to construct entities or *agents* possessing the self-same characteristics [Russell and Norvig 2009]. Although the definition of "intelligence" is open to interpretation, we can generally agree that humans are examples of such agents, and so a successful implementation of AI should be able to accomplish a multitude of tasks in the real world, much like any human.

A popular paradigm for constructing decision-making agents is *reinforcement learning* (RL), where an agent learns how best to act through trial-and-error interaction with its environment [Sutton and Barto 1998]. Recently, state-of-the-art RL approaches have made several significant advances in challenging domains, ranging from video games [Mnih *et al.* 2015] to Go [Silver *et al.* 2016], and even robotics [Levine *et al.* 2016]. Despite these successes, it is clear that these approaches do not capture a fundamentally remarkable aspect of human intelligence—namely, that humans can solve not just a single problem, but a massively diverse array of tasks.

There appears to be a disconnect, then, between the above approaches and the capabilities of humans—the former are aimed at solving a restricted set of tasks near-optimally, while the latter achieve competence in a wide range of tasks. For example, while the AlphaZero agent may possess superhuman abilities in playing chess [Silver *et al.* 2017], it cannot drive a car, cook a meal or assemble a bookcase. While the former approach is certainly useful in designing narrow, application-specific solutions, it falls well short of the ultimate aim of generally intelligent agents.

There are many challenges to overcome before we reach such a point. In this work, we highlight three that we believe must be tackled if we are ever to develop flexible, real-world agents.

i) **Sample efficiency**: Collecting data for training an RL agent can often be expensive, since it requires that the agent interact with its environment. Consequently, methods that can succeed using only a small amount of data are highly desirable or, in many cases, a strict requirement. While this is less of an issue in simulated environments, it is necessary for training agents that, like humans, operate in the real world. For example, recent algorithms for

playing Atari video games require *billions* of environment interactions [Badia *et al.* 2020]. Such approaches clearly cannot be applied to real robots, since training would be prohibitively time consuming, and wear and tear on the physical agent would result in failure.

ii) **Task representation learning**: As we will see in the next chapter, tasks in RL are formulated by human designers and provided to agents in a standardised form. For example, a chess-playing agent may be given a compact encoding representing the position of the pieces on the board, the legal moves available, and whose turn it is to play. Though this practice is widespread, it sidesteps an important question: *where do these representations come from in the first place?*

   It is obvious that this approach is infeasible in the long run: we cannot pre-program an agent with every task it may encounter before deploying it in the real world. Nor can we require that a human designer accompany the agent throughout its lifetime, providing task representations as and when it requires them. Clearly then, as Konidaris [2019] argues, the only option is for the agent to learn its own representation for any newly encountered task directly from its observations of the world.

iii) **Multitask learning**: If our goal is to have generally intelligent agents acting in the real world, it is insufficient to design agents capable of optimally solving only a handful of tasks. Rather, they must exhibit competence in a wide variety of tasks. Owing to the sample efficiency requirements, it is infeasible for an agent to solve every new task it encounters from scratch. Therefore, we require learning approaches that can be leveraged by an agent to solve new tasks quickly.

If we are to design a single agent capable of solving multiple tasks in the real world, it must necessarily have a complex sensorimotor space. However, decision making using high-dimensional observations requires large amounts of data, which runs counter to our sample efficiency requirements. Therefore, we need sample-efficient algorithms that allow a single agent to learn a suitable task representation for any given task using its sensor data.

1.1 Sample efficient planning with learned abstractions

A class of particularly challenging problems involves those that require high-level or long-term planning. Robots, in particular, face the difficult task of formulating plans while sensing and acting in high-dimensional, continuous spaces. Planning at this raw sensorimotor level is typically infeasible—the robot's innate action space involves directly actuating motors at a high frequency, but it would take thousands of such actuations to accomplish most useful goals. This is further exacerbated by its sensors, which often provide high-dimensional noisy signals.

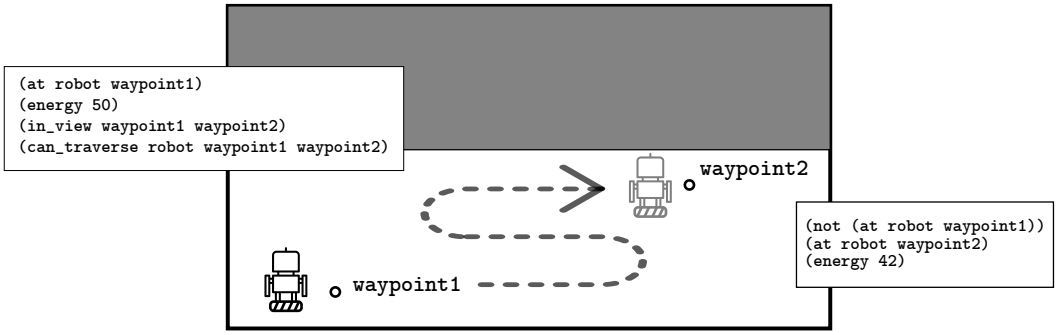
That being said, humans face a similar problem, and yet are still able to construct long-term plans consisting of (at the very lowest level) millions of actions. One likely reason for this is that we plan and reason about the world in terms of high-level *abstractions* [Botvinick and Weinstein 2014]. For instance, when planning a vacation, we do not construct a plan that includes every single step our legs must take (not to mention the low-level muscle activations involved in walking) to deliver us to our destination. Rather, we reason and plan using higher-order concepts, such as *being at an airport*, which abstract away these low-level control details.

One way of building high-level concepts is *action abstraction*. In the *options* framework, low-level actions are chained together to create *skills*, which are then used when learning and planning [Sutton *et al.* 1999]. In the above example, an agent might possess the skill `GetToAirport`, encapsulating all of the necessary low-level actions required to transport the agent to the airport.

The use of skills alleviates the problem of having to reason using low-level actions. However, the observations an agent makes about its environment may also be low-level. These observations make up what is known as the agent's *state*; examples of low-level state representations include pixels from a video camera, and continuous values representing the pose and joint positions of the agent. It is not hard to see that, even with skills, planning using these low-level states remains a great challenge. We may therefore also seek to perform *state abstraction*, where states are aggregated into high-level states, reducing the size of the problem. Examples of abstract states might be `AtHome` and `AtAirport`, representing all of the low-level observations (likely consisting of many gigabytes of pixel data) that correspond to the agent being at home or at the airport respectively.

One particular form of state abstraction is based on the *physical symbol system hypothesis* [Newell and Simon 1976], which states that the capacity to represent a problem using abstract patterns (*symbols*), as well as the ability to manipulate said symbols, is all that is necessary for truly intelligent behaviour. This is the approach taken in the classical planning literature [Fikes and Nilsson 1971; Ghallab *et al.* 2004]. Here a symbolic description of the world and available actions are provided to the agent, which is then tasked with discovering a plan of action to attain some goal. An example of a symbolic domain is given by Figure 1.1. However, constructing a symbolic representation of a real-world domain requires substantial domain knowledge that must be provided by a human designer.[1]

[1]Throughout this thesis, we use the term "symbolic representations" to refer to discrete abstractions that can take a small, finite number of values. For example, a proposition is a symbol that can take the values 0 or 1. Such abstractions are desirable for a number of reasons. Since each symbol has a finite domain, there is little uncertainty about its value at any given time. Further, the discrete nature of the symbolic representations allows an agent to enumerate potential outcomes of its actions quickly and precisely, allowing for fast forward-planning techniques. Finally, the decisions and plans of an agent can be output in a human-readable form.




```
(:action navigate
:parameters (?x - rover ?y - waypoint ?z - waypoint)

:precondition (and (can_traverse ?x ?y ?z)
                   (at ?x ?y)
                   (in_view ?y ?z)
                   (>= (energy ?x) 8)
               )
:effect (and (decrease (energy ?x) 8)
             (not (at ?x ?y))
             (at ?x ?z)
        )
)
```

```
Description of a navigation operation
for a rover moving between start and end points

If the rover can traverse from start to end
and it is at the start waypoint
and the end waypoint is visible from the start
and the rover has sufficient energy...

then the rover's energy levels decrease
and it is no longer at the start waypoint
and it is at the end waypoint.
```

Figure 1.1: An illustration and snippet of the "Mars Rover" domain from the 2002 International Planning Competition [Long and Fox 2003]. In the above diagram, the robot wishes to navigate between two waypoints. The `navigate` action (bottom left block) states that, in order for the agent to navigate between two waypoints `x` and `y`, the agent must be present at `x` and be able to see `y`, the path between the waypoints must be traversable, and the agent must have sufficient energy (see description in bottom right block). Fortunately, the current state of the agent (top left block) matches these conditions for `waypoint1` and `waypoint2`. After executing the action, the agent will find itself at `waypoint2` and its energy will have decreased by 8 (right block). Note how the difficulties and low-level details of navigating a robot over unknown terrain are abstracted by a human designer—it is assumed that the actions are able to take care of such details—and that these high-level symbolic states are sufficient for reasoning and planning in the world.

Abstraction, then, is clearly both desirable and necessary, but the question of how to link high-level reasoning with low-level sensing and control remains open. If an agent's abstractions are too high-level, it risks omitting important and necessary details. However, if it seeks to preserve every last detail of the environment, then its representations will be too low-level and planning will once again be infeasible. The key question is how best to construct an abstract model of an environment while retaining only the information required for planning.

Recent model-based RL algorithms learn a model of the environment's dynamics directly from observation data, and then perform planning in the learned latent space [Hafner *et al.* 2019; Schrittwieser *et al.* 2020; Hafner *et al.* 2021]. Despite some success, these approaches require an inordinate amount of data and engineering effort, and do not possess any soundness guarantees. An alternate approach is the recent framework proposed by Konidaris *et al.* [2018], which uses both state and action abstraction to construct a symbolic representation of an agent's environment.

In particular, given a set of high-level skills that have either been learned or provided to an agent, abstract states are constructed to support planning with those specific skills. Most importantly, the resulting symbolic representations are provably sound for long-term planning.

1.2 The need for transfer

If the approach taken by Konidaris *et al.* [2018] can be used to learn a representation suitable for long-term planning, then the obvious question is: *why is this not sufficient?* A major (and potentially fatal) deficiency in this approach is its lack of generalisability—the learned symbols apply only to the current task, so an agent must relearn the appropriate representation for each new task it encounters. This is a data- and computation-intensive procedure that operates on extremely high-dimensional data and requires repeated execution of actions within the environment.

To alleviate this issue, we can turn to the notion of *transfer*, where an agent leverages its knowledge from previous tasks to solve new ones [Taylor and Stone 2009]. Much like abstraction, the ability to transfer knowledge is a key component in our ability to generalise and adapt to new tasks quickly. For example, recent work has shown that humans leverage their knowledge of objects and affordances when playing video games—without this knowledge transfer we are no more efficient than an RL agent learning from scratch [Dubey *et al.* 2018].

1.3 Aims and contributions

In this work we seek to develop algorithms capable of learning transferable symbolic representations from low-level data that can be used for long-term planning. More concretely, we extend the symbol-learning framework of Konidaris *et al.* [2018] so that the learned representations are portable—given a new task, an agent can reuse the symbols it has learned previously to speed up learning. Our approach results in agents that are i) more sample efficient; ii) able to learn their own representation; and iii) able to use their learned representations to solve a variety of tasks. In particular, we make the following three contributions (summarised by Figure 1.2):

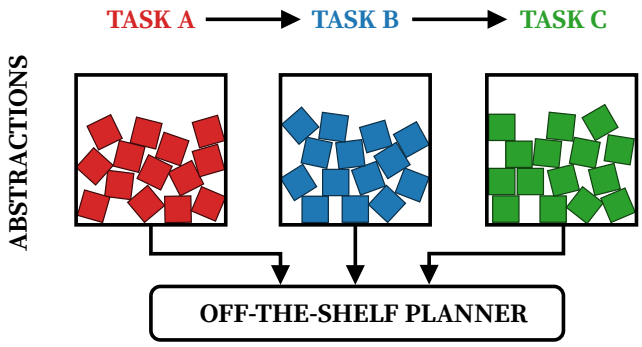
Agent-centric abstractions In Chapter 3 we propose a method that allows an agent to learn an egocentric representation of its environment directly from data. By assuming the existence of a single agent equipped with sensors that is common across all tasks, we demonstrate how to learn a high-level symbolic representation that can not only be used by an off-the-shelf planner, but can also be transferred to new domains.[2]

[2]This chapter is based on work presented in James *et al.* [2020].

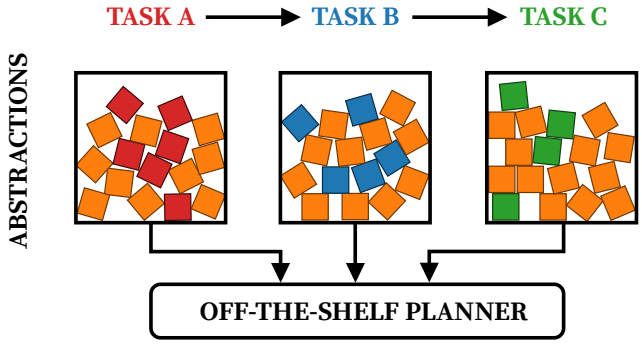
Object-centric abstractions We can extend our assumptions to include not only the existence of agents but also of objects. In Chapter 4 we demonstrate how such an assumption can be leveraged to learn object-centric representations that can be transferred to new environments that share similar types of objects.

Portable abstraction hierarchies The previous methods focus on building a single level of abstraction, which is then used for planning. In Chapter 5 we show how this can be extended to produce hierarchies of increasingly abstract representations that can also be transferred to new tasks. Constructing these hierarchies does not require further environment interaction, and we demonstrate how it can be used to decrease the size of the problem, thereby making planning easier.

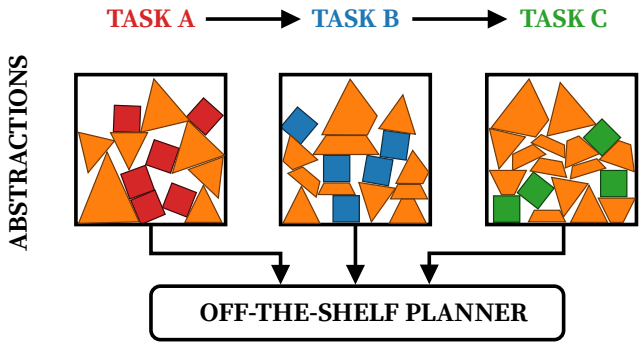
Taken together, these approaches allow an agent to autonomously learn symbolic representations that are decoupled from the task in which they were learned. These representations can be accumulated over the lifetime of an agent, and then deployed to quickly solve any new task an agent encounters. We see this as an important step towards creating adaptable agents that sense, act and learn in the real world.



(a) Adopting the framework of Konidaris *et al.* [2018] results in task-specific abstractions, represented by the different coloured building blocks. Representations from one task cannot be reused in a new task, which must be learned from scratch.



(b) In Chapters 3 and 4 we learn agent- and object-centric representations (orange blocks) that can be transferred between tasks. We combine these portable representations with non-portable abstractions to construct a model suitable for planning.



(c) In Chapter 5 we learn portable hierarchies of abstractions (orange triangles). These hierarchies, as well as parts thereof (orange trapezoids), can be transferred to new tasks and combined with task-specific abstractions to produce a model for the new task.

Figure 1.2: Illustration of our approach to learning abstractions. (a) Abstractions are task-specific and cannot be transferred to new tasks. (b) Learning a mixture of portable and task-specific abstractions. We can transfer the portable representations to a new task, and then learn fewer task-specific abstractions to complete the model. (c) Learning a mixture of task-specific abstractions with portable hierarchies of abstractions. We can transfer the hierarchies to a new task, and then complete the model by learning certain task-specific representations.

Chapter 2

Background

There are broadly speaking two fundamental problem areas when it comes to designing intelligent decision-making agents: *learning* and *planning*. Learning is characterised by an agent interacting with the world and using the resulting data to decide on the best course of action. Planning, on the other hand, involves an agent using a *model* of the world to decide how best to act. This model, which allows an agent to simulate the outcomes of different actions "in its head", can either be given to the agent or learned itself through environment interactions.

In this chapter, we outline the formalisms used for learning and planning with both unstructured and structured representations, and then show how the two can be linked. In Section 2.1 we outline the unstructured representations used in reinforcement learning, and then contrast them with the structured ones used in classical planning (Section 2.2). Then we outline in detail the framework of Konidaris *et al.* [2018], which can be used to learn a structured representation from low-level unstructured data. Finally, we conclude with a discussion of alternate techniques for representation learning in Section 2.4.

2.1 Unstructured representations in reinforcement learning

Reinforcement learning (RL) is concerned with constructing agents capable of acting optimally in *sequential decision processes*. Here an agent attempts to maximise its utility by making a sequence of action choices. At each time step, the agent receives an observation from its environment[1] and acts accordingly. As a consequence of its action, the agent receives feedback (*reward*) and finds itself, by way of some transition function, in a new environment configuration, or *state*. Whereas the rewards represent only the instantaneous outcome of an action, utility captures the

[1]The environment can be defined as that which cannot be arbitrarily changed by the agent [Sutton and Barto 1998].

long-term consequences of actions. The goal of the agent is simply to maximise its utility over time.

2.1.1 Markov decision processes

One of the most common classes of sequential decision processes, especially in RL, are *Markov Decision Processes* (MDPs). A sequential decision problem is an MDP if its transition and reward functions are *Markovian*—that is, the probability of reaching state s_{t+1} from state s_t, as well as the associated reward, depends only on s_t and the selected action and not on the history of earlier states [Puterman 2009]. Formally,

$$\Pr(s_{t+1}, r_{t+1} \mid s_0, a_0, s_1, a_1, \ldots, s_{t-1}, a_{t-1}, s_t, a_t) = \Pr(s_{t+1}, r_{t+1} \mid s_t, a_t). \quad (2.1)$$

An MDP is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where

- $\mathcal{S} \subseteq \mathbb{R}^n$ is the (possibly infinite) n-dimensional set of environment states;
- $\mathcal{A}$ is the (possibly infinite) set of actions;
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ defines the probability of transitioning between states under an action;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function; and
- $\gamma \in [0, 1]$ is used to discount future rewards.

Together the transition and reward functions are known as the *model* and govern the dynamics of the MDP. At time t, the agent receives state $s_t \in \mathcal{S}$ and executes an action $a \in \mathcal{A}$ according to its *policy* π—a mapping from states to actions. The agent then receives a scalar reward $r_{t+1} \in \mathbb{R}$ and transitions to the new state s_{t+1}. In stochastic shortest path problems [Bertsekas and Tsitsiklis 1991], an agent begins at some initial state distribution ρ_0 and must learn a policy to reach a set of goal states $\mathcal{G} \subseteq \mathcal{S}$ optimally.

The series of states, actions and rewards constitutes the agent's *experience*. This is used to modify the agent's policy to maximise the total expected utility or *return* G_t, given by

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \quad (2.2)$$

2.1.2 Solving MDPs with optimal policies

The ultimate aim of the agent is to learn an *optimal policy* π^* that maximises its expected return at all states. To define the optimal policy, we first require the notion of a *value function*. Under policy π, the value function at state s is defined as

$$V^\pi(s) = \sum_{s' \in \mathcal{S}} \Pr\left(s' \in \mathcal{S} \mid s, \pi(s)\right) \left[\mathcal{R}(s, \pi(s)) + \gamma V^\pi(s')\right], \quad (2.3)$$

which captures the expected return from a state under the given policy. A policy π^* is said to be optimal if $\forall s \in \mathcal{S}$, $V^{\pi^*}(s) = \max_\pi V^\pi(s)$. It is also useful to define the related *action-value function*, defined as

$$Q^\pi(s,a) = \sum_{s' \in \mathcal{S}} \Pr\left(s' \in \mathcal{S} \mid s, a\right) \left[\mathcal{R}(s,a) + \gamma V^\pi(s')\right]. \tag{2.4}$$

This captures the expected return after executing a in state s, and thereafter following policy π. The optimal action-value function Q^* is given by the set of *Bellman equations*:

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} \Pr\left(s' \mid s, a\right) \left[\mathcal{R}(s,a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a')\right]. \tag{2.5}$$

For deterministic transitions, the above are $|\mathcal{S}|$ non-linear equations in $|\mathcal{S}|$ unknowns, and can be solved with techniques such as dynamic programming [Bellman 1957]. Having computed the optimal action-value function, it is then trivial to derive the optimal policy π^*, which is greedy with respect to Q^*: $\pi^*(s) = \text{argmax}_{a \in \mathcal{A}} Q^*(s,a) \forall s \in \mathcal{S}$.

2.1.3 Reducing the state space

While the above approach to computing optimal policies is feasible when the MDP is relatively small, we can see from Equation 2.5 that it quickly becomes problematic for extremely large (or infinite) state spaces. Unfortunately, this is often the case in many real-world problems.

One way to overcome this is through the use of *function approximation*, where the value function is represented as a parameterised function. Here we approximate $\mathcal{S}$ using a set of *features* or basis functions $\boldsymbol{\phi}$, the dimension of which is much smaller than that of the state space. A common approach is to use a weighted linear combination of features to represent the value function, which then takes the compressed form

$$\tilde{V}(s) \approx \sum_{i=1}^{N} \theta_i \phi_i(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s), \tag{2.6}$$

where $N \ll |\mathcal{S}|$ is the dimension of $\boldsymbol{\phi}$, $\boldsymbol{\theta} \in \mathbb{R}^N$ is a weight vector learned by an RL algorithm, and $\boldsymbol{\phi}$ is a mapping $\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^N$. The feature vector $\boldsymbol{\phi}$ may be provided to the agent, or it may be learned directly from data. The latter is most commonly achieved by combining neural networks with RL algorithms [Riedmiller 2005; Mnih *et al.* 2015]. In either case, function approximation allows an agent to generalise its state values to similar, but unseen, states.

Another closely related approach is *state aggregation*, where sets of states are treated as single units. For example, a continuous state space may be discretised so that similar states (defined using an appropriate distance metric) are aggregated into a single logical abstract state. Often states are grouped to preserve some property of the original MDP; for example, *bisimulation* [Givan *et al.* 2003] creates an

abstraction that preserves the transition and reward function, while *utile distinction* [McCallum and Ballard 1996] preserves the optimal action and its value. We discuss other state aggregation methods in Section 2.4, but refer the reader to Li *et al.* [2006] for a thorough overview.

2.1.4 Action abstraction

Another major challenge is posed by environments that require long sequences of low-level actions to be executed before any reward is received. For example, a robot manipulating an object may only receive a reward once it has positioned its body and arm appropriately, and grasped the object in its hand. If this requires hundreds of low-level actions to achieve, then the reward signal will be extremely sparse and learning will be difficult, even when state abstraction is used.

Hierarchical reinforcement learning [Barto and Mahadevan 2003] tackles this problem by abstracting away low-level control details, allowing the agent to learn and plan using high-level actions called *skills*. We focus here on the *options* framework [Sutton *et al.* 1999], but note that there exists other approaches, such as Hierarchicies of Abstract Machines [Parr and Russell 1998] and MAXQ [Dietterich 2000], for creating high-level actions.

An option o is a temporally-extended action defined by the tuple $\langle \mathcal{I}_o, \pi_o, \beta_o \rangle$, where $I_o = \{s \mid o \in \mathcal{O}(s)\}$ is the *initiation set* that specifies the states in which the option can be executed, $\pi_o : \mathcal{S} \rightarrow \mathcal{A}$ represents the *option policy*, and β_o is the *termination condition*, where $\beta_o(s)$ is the probability of option o halting in state s. Note that options are strict generalisations of regular (*primitive*) actions, since an action a can be formulated as a one-step option that is executable everywhere, has a policy that selects a at all states, and terminates in all states with probability 1.

Replacing the set of actions in an MDP with a set of options results in a *semi-Markov decision process* (SMDP). This is characterised by the tuple $\langle \mathcal{S}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where

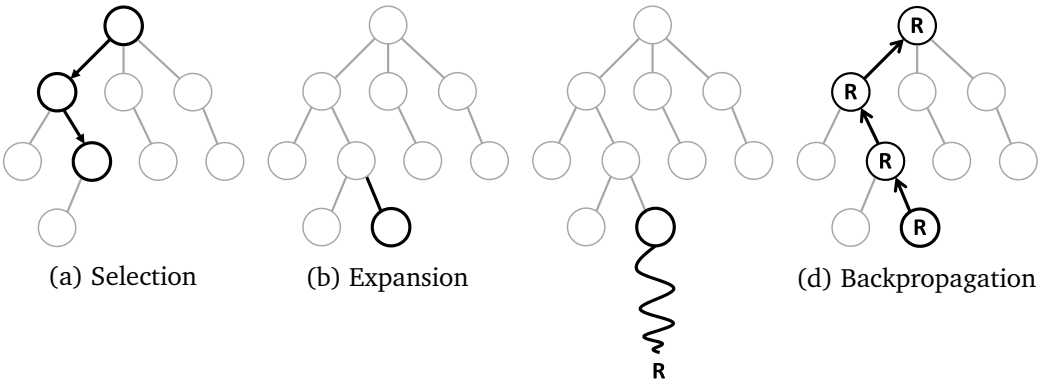
- $\mathcal{S}$ is the set of states as before;
- $\mathcal{O}(s)$ is the set of *options* available to the agent at state s;
- $\mathcal{T}$ specifies the probability of arriving in state s' after option o is executed from s for τ timesteps: $\Pr(s', \tau \mid s, o)$;
- $\mathcal{R}(s, o, \tau, s')$ specifies the feedback the agent receives from the environment when it executes option o from state s and arrives in state s' after τ steps; and
- γ is the discount factor as before.

2.1.5 MDP-based planning

The main aim of an RL agent is to learn an optimal policy through trial-and-error interaction with its environment. One approach is to optimise the policy or value function directly, using samples from the environment. Alternatively, we can use knowledge of $\mathcal{T}$ and $\mathcal{R}$ to compute an optimal policy without explicit environment

interaction. In the case of certain domains, such as board games, it is feasible for a human to specify the transition and reward functions for the agent. More generally, though, an agent may be required to first use environment samples to estimate the model, and then use those estimates to compute an optimal policy. Using data generated from a model (either given or learned) is known as *planning*.

For planning in small domains, dynamic programming techniques such as value or policy iteration [Bellman 1957; Howard 1960] can be used to compute an optimal policy. For larger domains, a popular planning algorithm is Monte Carlo tree search (MCTS) [Coulom 2006; Kocsis and Szepesvári 2006]. MCTS constructs an asymmetric lookahead tree starting at the agent's current state, and uses randomised simulations to estimate the value of states in the tree. This tree is continually grown until some time budget has elapsed, at which point the best action is selected and executed and the planning procedure repeats (see Figure 2.1). Note that, unlike other RL approaches, MCTS does not compute a value function or policy, which are defined at every state. Rather, it is an online planning algorithm that attempts to estimate the best action for *only* the current state.



(a) Selection (b) Expansion (c) Simulation (d) Backpropagation

Figure 2.1: Phases of the Monte Carlo tree search algorithm. Each node in the tree is a state, with the current state serving as the root node. The tree is grown by performing a lookahead search, adding new nodes (states) to the tree when a leaf node is encountered, and then simulating a full episode using a model to estimate the value of the new state. This is repeated until a fixed time budget has been exhausted, at which point the action leading to the next state with the highest estimate is selected.

2.2 Structured representations in classical planning

In contrast to MDP-centric formulations, the classical planning literature describes environments using a more structured, symbolic representation. In this paradigm, the aim is not to interact with an environment, but rather to use the provided domain model to compute an optimal plan (a sequence of actions) that solves a specific task.

2.2.1 Classical planning domains

In the classical representation, environments are modelled as *classical planning domains* (CPDs) $\Sigma = \langle \tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \tilde{\gamma}, \text{cost} \rangle$, representing the state space, action space, transition dynamics and cost function respectively.[2] The state space is described using vocabulary $\mathcal{L}$, consisting of *predicate* symbols that take zero or more variable arguments. We refer to a predicate as *ground* if it possesses no arguments (*a proposition*), or if its arguments have been instantiated with constants or objects in the world. Each state in the environment is represented as a conjunction of ground predicates, and all predicates not explicitly mentioned are assumed to be false. The planning domain definition language (PDDL) [McDermott *et al.* 1998] is one of the most common standards used to encode these representations.

Each action or *operator* a in $\tilde{\mathcal{A}}$ is represented by the triple

$$\langle \text{pre}(a), \text{effects}^{-}(a), \text{effects}^{+}(a) \rangle,$$

where $\text{pre}(a)$ is the precondition specifying the conditions that must hold in order for a to be executable; $\text{effects}^{-}(a)$ are the negative effects (the set of predicates set to false by the operator); and $\text{effects}^{+}(a)$ are the positive effects (the set of predicates made true).

The transition function $\tilde{\gamma}$ predicts the next state given the current state s and operator a, where

$$\tilde{\gamma}(s, a) = \begin{cases} \{s \setminus \text{effects}^{-}(a)\} \cup \text{effects}^{+}(a) & \text{if } \text{pre}(a) \subseteq s \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The dynamics can also model stochastic transitions by assigning each potential outcome its own probability, as in the probabilistic planning domain definition language (PPDDL) [Younes and Littman 2004]. Finally, the cost function represents some metric that should be minimised (or maximised) when planning, analogous to the negative reward function in MDPs. If omitted, the cost function is assumed to be 1 everywhere.

Example 1. *Consider the environment illustrated by Figure 2.2 consisting of two robots and two doors. In the MDP formulation, the state of the world might be specified by a vector containing the robots' xy-positions and the angles of the doors. By contrast, a classical representation of the state is as follows:*

```
near(r1, d1) and is-closed(d1) and is-closed(d2).
```

[2] We use $\tilde{\cdot}$ to disambiguate the classical representation and MDP variables when they are identical. For example $\mathcal{S}$ is an MDP state space, while $\tilde{\mathcal{S}}$ is a classical planning state space.

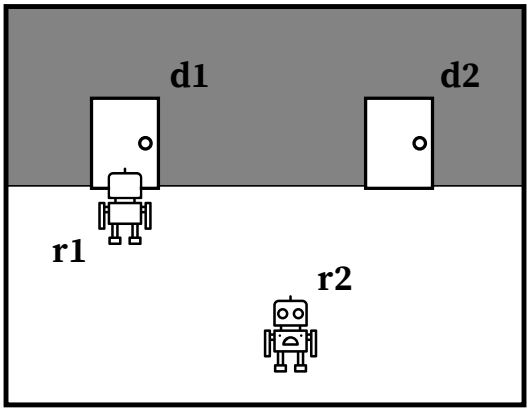


Figure 2.2: An environment with two robots (r1,r2) and two doors (d1,d2). One robot is near the first door and can interact with it, while the other is out of range of either door.

An operator for opening the door might take the following form:

```
(action: open-door
 parameters: (?x - robot, ?y - door)
 precondition: near(x, y) and is-closed(y)
 negative effect: is-closed(y)
 positive effect: is-open(y)
)
```

The operator is parameterised by two objects—a robot and door—and states that, in order for a robot to open a door, it must be near the door and the door must be closed. As a result of executing the operator, the door is no longer closed but open. Note that given this action operator, it is clear that it can be instantiated with `r1` and `d1`, but not with `r2` and any other door, since the precondition would not be satisfied in the initial state.

2.2.2 Solving CPDs with optimal plans

In the classical planning formalism, the domain description is decoupled from the task the agent is required to solve. While the CPD specifies the dynamics of the environment, the problem description specifies what the agent is required to achieve. In particular, a classical planning problem is given by the tuple $\langle \Sigma, s_0, g \rangle$, where Σ is the domain description, s_0 is the initial state and g is a set of ground predicates representing the goal.

The aim of an agent is to discover an optimal plan that satisfies the goal condition starting from the initial state. A plan $\tilde{\pi}$ is simply a finite sequence of operators

$$\tilde{\pi} = \langle a_1, a_2, \ldots, a_n \rangle,$$

and the cost of a plan is given by sum of each operator's cost in the plan. We say that $\tilde{\pi} = \langle a_1, a_2, \ldots, a_n \rangle$ is a *solution* to a planning problem $\langle \Sigma, s_0, g \rangle$ if there exists states $s_1, \ldots, s_n$ such that $\tilde{\gamma}(s_i, a_i) = s_{i+1}$ for all $1 \leq i < n$ and $g \subseteq s_n$. In other words, a plan is a solution to a problem if it can be executed according to the transition

function, starting at the initial state and reaching (satisfying) the goal condition. Finally, we refer to a plan as *optimal* if there is no other solution that incurs smaller cost: $\tilde{\pi} = \underset{\tilde{\pi}' \in \text{ all plans}}{\operatorname{argmin}} \{\text{cost}(\tilde{\pi}') \mid \tilde{\pi}' \text{ is a solution}\}$ [Ghallab *et al.* 2004].[3]

2.2.3 Classical planning methods

One of the main differences between MDP-based planning and classical planning is the structure imposed by the latter's representation. Since the problem definition specifies the start and goal states, we can apply backward- or forward-search algorithms, such as breadth-first or A* search, to find the shortest path between start and end states.

Importantly, the domain structure can be leveraged to automatically produce heuristics that can then be combined with forward search algorithms to speed up planning. For example, the Heuristic Search Planner [Bonet and Geffner 1999] performs search on a relaxation of the given domain—in particular, the negative effects of operators are not accounted for. The resulting solution never overestimates the length of the optimal plan in the original problem, and can therefore serve as an admissible heuristic for various search algorithms. Owing to the domain structure and heuristics, classical planners can solve extremely large problems efficiently—domains consisting of more than 330 000 ground actions were easily solved at the 2003 International Planning Competition [Long and Fox 2003].

2.3 From unstructured to structured representations

In the previous sections, we reviewed two approaches to representing agents acting in environments. The MDP-based formulation can be used to capture low-level details of real-world tasks, such as high dimensional observations and continuous actions. However, learning in such a regime is often difficult and computationally expensive. On the other hand, the classical representation uses symbols to describe an idealised world where low-level details are dispensed with.

Abstract symbolic representations of a low-level task are extremely attractive, since agents can efficiently construct plans to solve tasks of a hierarchical nature. This allows agents to tackle sparse-reward problems by planning at a high level, reducing the effective horizon of the problem. A major issue, however, is the connection between the symbolic representation of the environment, and the environment itself. In the paradigm described in Section 2.2, symbols are typically provided to the agent by a human expert, but clearly this approach does not scale to real-world agents. Therefore, the agent must concern itself with first acquiring these symbols autonomously. This must be done in such a way that the symbols (or the logical conclusions reached by manipulating them) are grounded in reality, and not

[3] In the stochastic setting, a plan is optimal if there is no other solution that incurs smaller *expected* cost [Younes and Littman 2004].

simply imaginary hallucinations of the agent. Since we are interested in planning, the obvious question then is: *how can an agent sensing and acting in a low-level, high-dimensional domain learn a sound symbolic CPD useful for planning?*

In this section, we discuss the *skills-to-symbols* framework [Konidaris *et al.* 2018] for learning a probablistic representation in the form of PPDDL. A defining feature of this approach is that it uses both state and action abstraction to construct a representation that is provably sound and useful for planning. Throughout this work, we assume the agent is equipped with a set of options; these options may be hand-designed and provided to the agent, but more generally may be learned using any appropriate skill discovery algorithm (e.g. [Stolle and Precup 2002; Konidaris and Barto 2009b; Ranchod *et al.* 2015; Machado *et al.* 2017; Bacon *et al.* 2017; Jinnai *et al.* 2019; Bagaria and Konidaris 2019]). In either case, the framework then constructs abstract states to support planning with those very options. Consequently, the learned representation is provably sound and complete, forming a bridge between low-level control and high-level planning.

2.3.1 Learning preconditions and effects

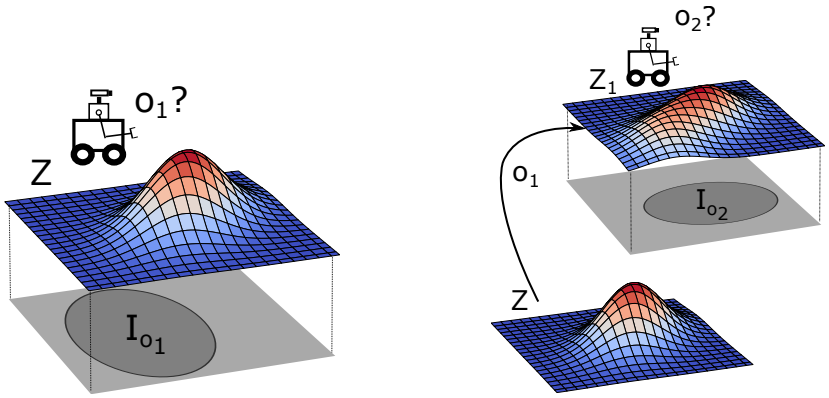
The first question to ask is: *what exactly should an agent learn?* To answer this, recall first that we wish to learn an abstract representation to facilitate planning. In particular, while classical planning primarily centres around deterministic domains, we are instead interested in the stochastic case. This will allow us to model MDPs with stochastic dynamics and account for the uncertainly inherent in learning with a finite number of samples.

A *probabilistic plan* $\tilde{\pi}_Z = \{o_1, \ldots, o_n\}$ is defined as a sequence of options to be executed, starting from some state drawn from distribution Z. It is useful to introduce the notion of a *goal option*, which can only be executed when the agent has reached its goal. Appending this option to a plan means that the probability of successfully executing a plan is equivalent to the probability of reaching some goal. Our representations should therefore allow us to compute the probability of $\tilde{\pi}_Z$ succeeding.

As a plan is simply a chain of options, it is therefore necessary (and sufficient) to learn when an option can be executed, as well as the outcome of doing so [Konidaris *et al.* 2018]. This corresponds to learning the *precondition* $\text{Pre}(o) = \Pr(s \in I_o)$, which expresses the probability that option o can be executed at state $s \in \mathcal{S}$, and the *image*

$$\text{Im}(Z, o) = \frac{\int_{\mathcal{S}} \Pr(s' \mid s, o) Z(s) \Pr(s \in I_o) ds}{\int_{\mathcal{S}} Z(s) \Pr(s \in I_o)},$$

which represents the distribution of states an agent may find itself in after executing o from states drawn from distribution Z. Note that even though the precondition for an option is a *set*, we allow for it to be probabilistic to account for uncertainty and error in its estimation. Figure 2.3 illustrates how the precondition and image are used to calculate the probability of executing a two-step plan.



(a) The agent begins at distribution Z, and must determine the probability with which it can execute the first option o_1.

(b) The agent estimates the effect of executing o_1, given by Z_1. It must then determine the probability of executing o_2 from Z_1.

Figure 2.3: An agent attempting to calculate the probability of executing the plan $\tilde{\pi}_Z = \{o_1, o_2\}$, which requires knowledge of the conditions under which o_1 and o_2 can be executed, as well as the effect of o_1. Reused with permission from Konidaris *et al.* [2018].

2.3.2 Partitioned subgoal options

For large or continuous state spaces, representing the image of an arbitrary option presents difficulties, since in the worst case we would need to model $\Pr(\cdot \mid s, o)$ conditioned on every s. However, we can do so for a subclass known as *subgoal options* [Precup 2000], of which there are two variants [Konidaris *et al.* 2018]. An option is said to possess the *weak subgoal property* if the probability of executing the *next* available options is the same, regardless of the initial state of the agent. In other words, for all options $o, n \in \mathcal{O}$ and states $s, t \in I_o$

$$\Pr(s' \in I_n \mid s, o) = \Pr(t' \in I_n \mid t, o),$$

where $s' \sim \Pr(\cdot \mid s, o)$ and $t' \sim \Pr(\cdot \mid t, o)$. Alternatively, an option possesses the *strong subgoal property* if its terminating states are independent of its starting states. That is, for any strong subgoal option o, $\Pr(s' \mid s, o) = \Pr(s' \mid o)$. We can thus substitute the option's image for its *effect*: $\text{Eff}(o) = \text{Im}(Z, o)\ \forall Z$. Although the strong subgoal condition is more restrictive, it will prove more useful practically; how best to exploit the weak subgoal condition remains an open question.

Subgoal options are not overly restrictive, since they refer to options that drive an agent to some set of states with high reliability, which is a common occurrence in robotics owing to the use of closed-loop controllers. Nonetheless, it is likely an option may not be subgoal. It is often possible, however, to *partition* an option's initiation set into a finite number of subsets, so that it possesses the strong subgoal property when initiated from each of the individual subsets. In other words, we partition an option's initiation set into classes $\mathcal{C}$ such that $\Pr(s' \mid s, c) \approx \Pr(s' \mid c), c \in \mathcal{C}$ (see Figure 2.4). More formally, we define a *partitioned option* as follows:

Definition 1. *Given an option $o \in \mathcal{O}$, define a relation $\sim_o$ on I_o so that $s \sim_o t \iff \Pr(s' \mid s, o) = \Pr(s' \mid t, o)$ for all $s, t, s' \in \mathcal{S}$. Then $\sim_o$ is an equivalence relation which partitions I_o. Label each equivalence class in $I_o / \sim_o$ with a unique integer α. A partitioned subgoal option in $\mathcal{S}$ is then the parameterised option $o(\alpha) = \langle [\alpha], \pi_o, \beta_o \rangle$, where $[\alpha]$ is the set of states in equivalence class α.*

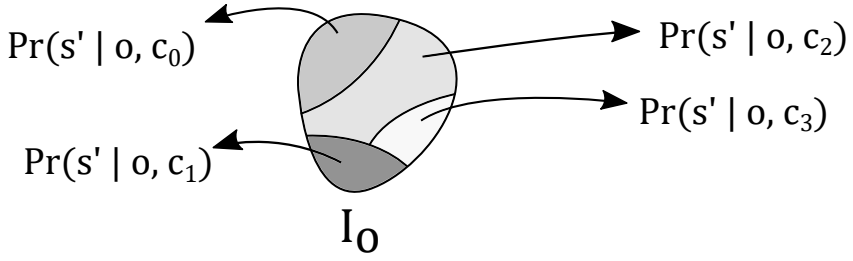
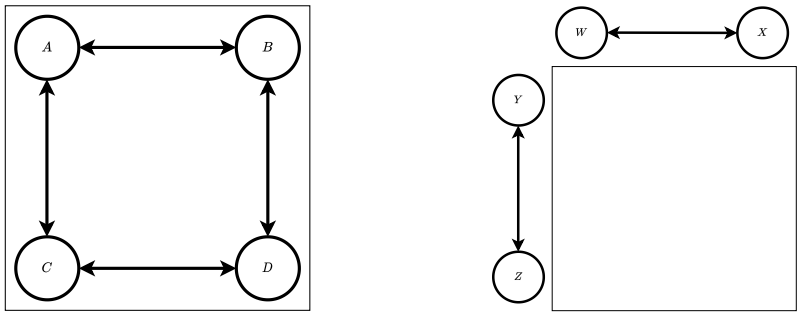


Figure 2.4: Option o is not subgoal, but we can partition the initiation set I_o into 4 subsets $c_0, \ldots, c_3$, such that the option is subgoal when initiated from each of these sets. Note that each of the differently shaded regions is an equivalence class on I_o.

Abstract options for factored representations

The classical representation makes several assumptions about the world. These include the *frame assumption* (any aspect of the world not explicitly changed by the agent remains the same) and the *action outcomes assumption* (each action affects the world in only a small number ways) [Pasula *et al.* 2004]. Along similar lines, we may assume that our options are *abstract*—that is, they obey the frame and action outcomes assumptions [Konidaris *et al.* 2018]. For each option, we decompose the state into two sets of variables $s = [a, b]$, such that executing the option results in state $s' = [a, b']$, where a is the subset of variables that remains unchanged. We refer to the variables that are modified by an option as its *mask*. If the options violate the action outcomes assumption (that is, all state variables are always changed by the options), then the resulting representation will be a planning graph or abstract MDP. However, if these assumptions hold, then the learned representation will take the form of a *factored* abstract MDP or STRIPS-like representation [Fikes and Nilsson 1971].

Example 2. *Consider an agent whose state is characterised by its xy-coordinates in a square room. The agent starts in the top-left corner of the room, and possesses options that allow it to navigate from any corner to the adjacent corners. If all options modify the x and y position simultaneously, then there is no opportunity for factorisation. The resulting representation consists of a graph with four nodes, each of which is an abstract state (see Figure 2.5a). However, if the options can modify x and y independently, then we can factorise our representation: one factor represents the agent's x position, and the other the y position. The state is then the Cartesian product of these two factors (see Figure 2.5b).*



(a) (b)

Figure 2.5: An illustration of the difference between factored and unfactored abstractions. (a) The abstraction is unfactored and the abstract states are given by the set $\{A, B, C, D\}$. (b) The abstraction is factored into x and y components and the abstract states are given by the product $\{W, X\} \times \{Y, Z\}$.

2.3.3 Constructing a CPD

We can now describe the framework that allows an agent to learn its own classical representation from low-level data. This procedure is illustrated by Figure 2.6 and summarised below, but for more detail see Konidaris *et al.* [2018].

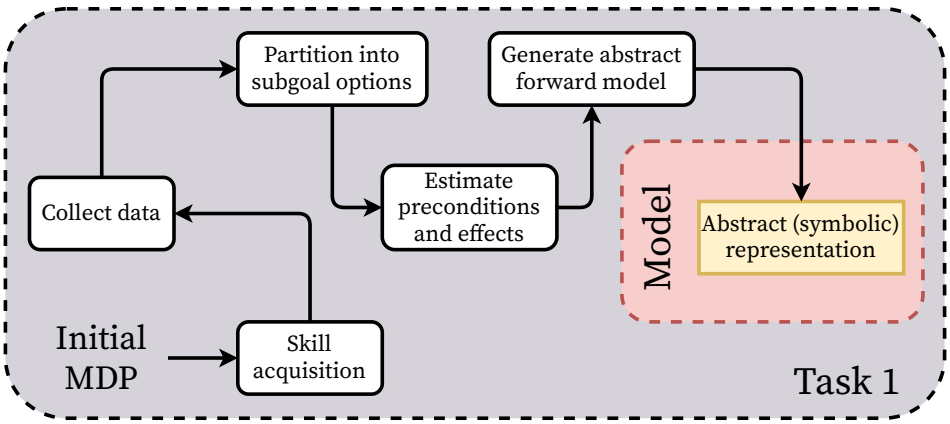


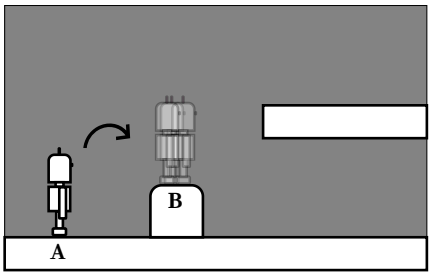
Figure 2.6: The process of learning symbolic representations [Konidaris *et al.* 2018]. The abstract model can take various forms, such as a factored MDP or a PPDDL description.

Step 1: Option partitioning

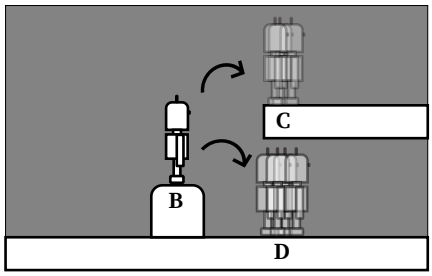
As mentioned, estimating the effects of options may be difficult if they do not already possess the subgoal property. The first step is therefore to partition the options into approximately subgoal options based on data collected by interacting with the environment. Let $\bar{I}_o$ and $\bar{\beta}_o$ be the empirical sets of initial and terminating states for a given option o. The agent first partitions $\bar{I}_o$ into subsets $K \subseteq \bar{I}_o$ such that $\Pr(s' \mid s_i, o) = \Pr(s' \mid s_j, o) \forall s_i, s_j \in K, s' \in \bar{\beta}_o$, as per Definition 1.

Unfortunately, finding an exact partitioning is computationally expensive. Prior work has approximated this procedure by assuming that distinct effects are spatially distant, and then clustering state transition samples based on effect states [Andersen and Konidaris 2017; Konidaris *et al.* 2018; Ames *et al.* 2018]. Each cluster is assigned to a partition, and each pair of partitions is examined to determine whether their start states overlap significantly. If they do overlap, the partitions are merged to account for probabilistic effects.[4]

Example 3. *To illustrate this approximation procedure, consider the example of an agent executing a* `Jump-Right` *skill to reach a higher ledge. The state of the environment is the xy-location of the agent, and the domain is stochastic. As a result, occasionally the agent jumps and fails to reach the ledge, falling to the ground. There are two cases to consider, illustrated by Figure 2.7 below.*



(a) Jumping from position `A`, the agent always manages to land on the small ledge. However, its exact position varies slightly owing to stochasticity in the environment.



(b) Jumping from position `B`, the agent is sometimes able to reach the higher ledge at location `C`. However, because of stochasticity, the agent occasionally falls to the ground at position `D`.

Figure 2.7: Examples of an agent jumping onto a higher ledge from various locations. The domain is stochastic, and so the effects of executing the option are also stochastic. This stochasticity is visualised by transparency in the outcomes.

Using the procedure described above, we first cluster option data based on the terminating states. This results in three clusters at locations `B`*,* `C` *and* `D`*, each of which is assigned to its own partitioned option. Our next step is to determine whether any of these three are in fact stochastic outcomes of a single partitioned option. The initial states for the option* `A` $\rightarrow$ `B` *do not overlap with any of the other partitions, and so are left as its own partitioned option. However, the initation set for the option* `B` $\rightarrow$ `C` *does overlap with* `B` $\rightarrow$ `D`*. Therefore, we merge these into a single probabilistic partitioned option. If there are N empirical samples of the agent making the transition* `B` $\rightarrow$ `C`*, and M for the transition* `B` $\rightarrow$ `D`*, then the associated probabilities are simply $N/(N+M)$ and $M/(N+M)$.*

[4]In this context, clustering is used to approximate the subgoal property. However, it has also been used in prior work to discover effects that can be reliably predicted [Ugur *et al.* 2012]. The latter approach has achieved empirical success and provides an avenue for adaptively selecting the clustering algorithm's hyperparameters. However, it lacks the theoretical guarantees of the subgoal condition.

Step 2: Learning preconditions

Having ensured that the options are approximately subgoal, the agent next learns a precondition classifier for each of these approximately partitioned options. Recall that the precondition corresponds to determining whether an option can be executed at a given state, and is thus a classification problem. We can fit a probabilistic classifier to the data; states inside each partitioned option's initiation set are treated as positive examples, and all others as negatives. However, it is not sufficient to simply fit a classifier to the data, since it may be the case that the precondition depends only on a *subset* of state variables. Therefore, we first employ a feature selection procedure to determine which state variables are relevant to the option's precondition. We refer to these variables as the *precondition mask*. The framework is agnostic to the manner in which feature selection is performed, but a simple approach is outlined in Figure 2.8. Having determined the precondition mask, a probabilistic classifier is fit to the relevant state variables' data.

procedure FEATURESELECTION
  Given: positive start states p, negative start states n, state dimension N, threshold ϵ.
  ▷ Fit a classifier over all state variables
  $allVars \leftarrow \{1, \ldots, N\}$
  $initScore \leftarrow$ FITANDSCORE$(start, negative, allVars)$
  $mask \leftarrow \emptyset$
  for each $variable \in \{1, \ldots, N\}$ **do**
    $subsetScore \leftarrow$ FITANDSCORE$(start, negative, allVars \setminus \{variable\})$
    if $initScore - subsetScore > \epsilon$ **then**
      ▷ Keep the variable if it causes score to decrease when removed
      $mask \leftarrow mask \cup \{variable\}$
    end if
  end for
  $latestScore \leftarrow$ FITANDSCORE$(start, negative, mask)$
  for each $variable \in allVars \setminus Mask$ **do**
    $newScore \leftarrow$ FITANDSCORE$(start, negative, Mask \cup \{variable\})$
    if $newScore - latestScore > \epsilon$ **then**
      ▷ Keep the variable if adding it improves the score
      $mask \leftarrow mask \cup \{variable\}$
      $latestScore \leftarrow newScore$
    end if
  end for
  return $mask$
end procedure

Figure 2.8: Pseudocode for a simple feature selection procedure to determine the relevant state variables for an option's precondition. First, a classifier is fit to all the data and the overall score computed. Then, any state variable that reduces the score when removed is kept as part of the precondition mask. Finally, we attempt to add the remaining state variables back to the mask; any that improve the classifier's performance are kept and added to the mask.

Step 3: Learning effects

Computing the effect of a partitioned option is a problem of density estimation. For a given partitioned option, we consider its terminating states[5] and compute the mask—the set of state variables that have been changed by the option. Next we fit a density estimator to the data, taking only the state variables in the mask into account. Each of these density estimators (representing a distribution over a subset of state variables) forms a single proposition in our classical representation vocabulary $\mathcal{L}$.

Step 4: Generating a PPDDL model

For each partitioned option o, the agent has learned a precondition classifier $\hat{I}_o$ and effect distribution $\hat{\beta}_o$. However, to construct a PPDDL representation, both the precondition and effects must be specified in terms of state distributions (propositions) only. Effects are modelled as such and so pose no problem, but the learned precondition is a classifier rather than a state distribution. The agent must therefore iterate through all possible effect distributions to compute whether the skill can be executed there. This procedure is exponential in the number of factors—more precisely, its time complexity is $O(|\mathcal{O}||\mathcal{L}|^{|F|})$, where $|\mathcal{O}|$ is the number of options, $|\mathcal{L}|$ is the size of the symbolic vocabulary, and $|F|$ is the number of factors [Konidaris *et al.* 2018]. Fortunately, the number of factors is usually small in practice, and so the above can be executed in a reasonable amount of time.

This is achieved by replacing o's precondition classifier with every $\mathcal{P} \in \wp(\mathcal{L})$ such that $\int_{\mathcal{S}} \hat{I}_o(s)\mathcal{G}(s)ds > 0, \mathcal{G} = \prod_{p\in\mathcal{P}} p$, where $\mathcal{L}$ is the vocabulary and $\wp(\mathcal{L})$ denotes the powerset of $\mathcal{L}$. In other words, the agent considers every combination of effect distributions and draws samples from their conjunction. If these samples are classified as positive by $\hat{I}_o$, then the conjunction $\mathcal{P}$ is used to represent the precondition. The preconditions and effects are now specified using distributions over state variables, where each distribution is a proposition. We have now learned a probabilistic CPD, which is sound and suitable for planning. The procedure for constructing the classical operators is described in Figure 2.9, while the full pseudocode for the entire framework is given by Figure 2.10.

2.4 Related model-learning approaches

We have presented one approach to bridging the gap between low-level MDPs and smaller, discrete representations; however, there are many other ways of doing so. These can be broadly categorised into approaches that (a) learn models *given* a set of predicates; (b) learn symbolic models from low-level sensory data; and (c) perform state aggregation to construct smaller, property-preserving MDPs. We briefly describe these approaches below.

[5] In the case of a probabilistic option, there will be multiple sets of terminating states, and we would perform density estimation for each probabilistic outcome in turn.

procedure BUILDPPDDLOPERATOR
  Given: precondition classifier $classifier$, option effect $effect$, all symbols $vocabulary$
  $operators \leftarrow \emptyset$
  for each $candidate \in \wp(vocabulary)$ **do** ▷ For all possible effect combinations
    $samples \leftarrow$ SAMPLE$(candidate)$ ▷ Sample from the distributions
    $prob \leftarrow$ SCORE$(classifier, samples)$ ▷ Query the classifier with the data
    if $prob > 0$ **then**
      if $prob = 1$ **then**
        ▷ Construct the new operator with the existing effects
        $operator \leftarrow \{candidate, effect\}$
      else
        ▷ Add a probabilistic failure case
        $newEffect \leftarrow \begin{cases} \texttt{fail}, \text{with probability } (1 - prob) \\ effect, \text{with probability } prob \end{cases}$
        $operator \leftarrow \{candidate, newEffect\}$
      end if
      $operators \leftarrow operators \cup \{operator\}$
    end if
  end for
  return $operators$
end procedure

Figure 2.9: Pseudocode for constructing probabilistic operators. The procedure accepts the precondition classifier and effect for a single partitioned option, along with the entire propositional vocabulary. It then determines which combination of propositions are evaluated as positive by the classifier. Those that match are used as the precondition for the operator.

Top-down model acquisition

One line of work learns symbolic models *given* symbolic state descriptors. Here, states represented as conjunctions of ground predicates are provided, and the challenge is to learn rules that best describe the effects of actions on these predicates. This class of approaches is illustrated by Figure 2.11, which provides a symbolic representation of a block-stacking domain with three blocks and a gripper.

For example, Khardon [1999] uses inductive logic programming to learn solutions to problems in a particular domain, given the domain description and example solutions. Finney *et al.* [2002] attempt to learn the transition dynamics of a block-stacking domain by estimating the dynamics with respect to each block in the environment. Unfortunately, partial observability is introduced by considering only each block in turn and, as a result, learning performance is severely hampered. Pasula *et al.* [2004] present an algorithm for learning a probabilistic model of relatively simple environments by inferring parameterised rules. Since these rules are relational, they can be reused in new task configurations. Zettlemoyer *et al.* [2005]

procedure LEARNREPRESENTATION
  Given: T state-option transitions $\mathcal{D} = \{(s_i, o_i, s'_i) \mid 0 \leq i \leq T\}$, state dimension N

  ▷ Partition options into subgoal options
  $subgoalOptions \leftarrow \emptyset$
  for each $o \in \mathcal{O}$ **do**
    $\bar{I} \leftarrow \{s \mid (s, o, \cdot) \in \mathcal{D}\}$ ▷ Set of initial states for option o
    $\bar{\beta} \leftarrow \{s' \mid (\cdot, o, s') \in \mathcal{D}\}$ ▷ Set of terminating states for option o
    for all $K \subseteq \bar{I}$ such that $\Pr(s' \mid s_i, o) = \Pr(s' \mid s_j, o) \forall s_i, s_j \in I, s' \in \bar{\beta}$ **do**
      ▷ Extract the start and end states for a partition
      $P \leftarrow \{o, K, \{s' \mid \forall s \in K, (s, o, s') \in \mathcal{D}\}\}$
      $subgoalOptions \leftarrow subgoalOptions \cup \{P\}$
    end for
  end for

  ▷ Estimate preconditions and effects
  $estimatedOperators, vocabulary \leftarrow \emptyset$
  for each $\{\cdot, start, end\} \in subgoalOptions$ **do**
    $mask \leftarrow$ COMPUTEMASK$(start, end)$ ▷ List the objects that change state
    $negative \leftarrow \mathcal{S} \setminus start$
    $features \leftarrow$ FEATURESELECTION$(start, negative, N)$
    $classifier \leftarrow$ FITCLASSIFIER$(start, negative, features)$
    $estimator \leftarrow$ FITESTIMATOR$(mask, end)$ ▷ Fit over mask only
    $vocabulary \leftarrow vocabulary \cup \{estimator\}$
    $estimatedOperators \leftarrow estimatedOperators \cup \{\{classifier, estimator\}\}$
  end for
  ▷ Build propositional PPDDL
  $operators, \leftarrow \emptyset$
  for each $precondition, effect \in estimatedOperators$ **do**
    $op \leftarrow$ BUILDPPDDLOPERATOR$(precondition, effect, vocabulary)$
    $operators \leftarrow Operators \cup \{op\}$
  end for
  return $vocabulary, operators$
end procedure

Figure 2.10: The full *skills-to-symbols* framework for autonomously constricting a propositional PPDDL representation given a set of options. Some subroutines used in the pseudocode below are outlined in previous sections.

and Pasula *et al.* [2007] extend this approach to more complex domains, allowing an agent to learn relational rules in fully observable stochastic environments.

To handle partial observability, Amir and Chang [2008] ignore the possibility of action failures and propose an algorithm for learning operator preconditions and effects. Approaches by Halbritter and Geibel [2007] and Mourão *et al.* [2010] estimate action effects in partially observable stochastic domains, but the repre-

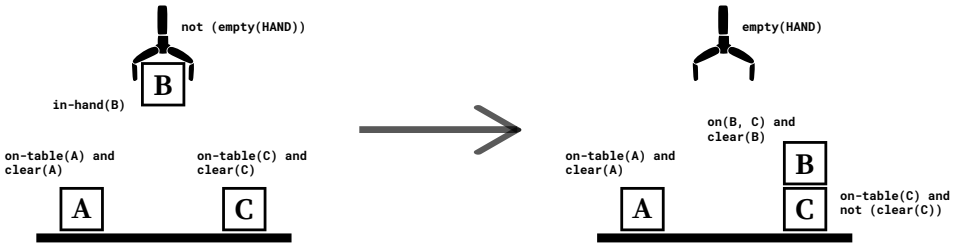


Figure 2.11: Top-down approach to model learning. The agent must estimate the classical planning operators from transition data, but the states are already represented symbolically. Here the state of the world is represented by the predicates such as `on-table` and `clear`, and must infer general rules from the way actions modify these predicates.

sentations cannot be explicitly represented using PDDL, precluding the use of fast classical planners. Mourão *et al.* [2012] extend these approaches by using classifiers to determine the *mask* of each action from which PDDL preconditions and effects can be extracted, while Aineto *et al.* [2019] present an approach for learning symbolic representations given examples of plan executions that may be partially or wholly unobservable.

More recently, deep learning approaches have been applied to symbolic representations to learn policies and abstractions that generalise to new problem definitions [Toyer *et al.* 2018; Bajpai and Garg 2018; Bonet *et al.* 2019]. In all of these cases, however, the symbols modelling the state space are given—the origin of these symbols therefore remains an open question.

Bottom-up symbol acquisition

Unlike the previous section, certain approaches can learn symbolic representations given low-level raw observations, as illustrated by Figure 2.12. Bonet and Geffner [2020] learn first-order symbolic representations given a graph that encodes the structure of the state space and transition dynamics. While their approach does not require predefined symbols, it is unlikely that a real-world agent acting in novel environments will have access to such a graph.

Ugur and Piater [2015a;b] learn object-centric PPDDL representations for an object manipulation task by clustering the effects of actions based on the observed change in state. While their system produces a PPDDL representation that can be used for planning on a real robot, object features are specified prior to learning, and discrete relations between object properties such as width and height are given. Furthermore, certain predicates are manually inserted to generate a sound representation. Ahmetoglu *et al.* [2020] extend this approach by using deep neural networks, allowing the agent to learn from raw object observations instead of handcrafted object features. However, in order to generate a PPDDL representation suitable for constructing towers of blocks, predicates that track the number of objects in a tower must still be manually inserted.

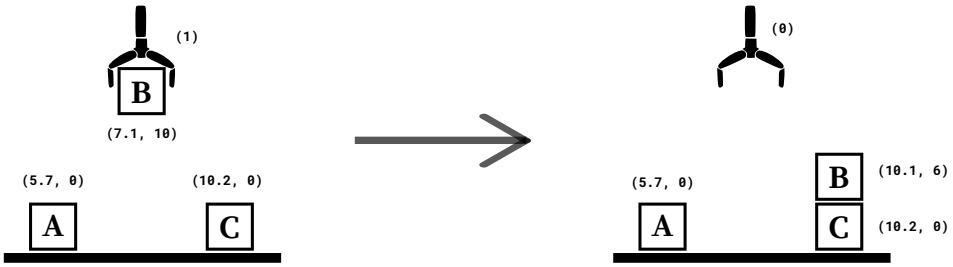


Figure 2.12: Bottom-up approach to model learning. Here an agent must estimate the classical planning operators and state predicates from transition data. However, states are not represented symbolically, and so the symbolic states, as well as the dynamics, must both be learned. In the above block-stacking example, the state of each block is represented by its xy-position instead of the symbolic representations in Figure 2.11.

Jetchev *et al.* [2013] are able to learn relational symbols and operators directly from geometric data. The approach is framed as an optimisation problem—the procedure attempts to search for symbol groundings that maximise the predictive abilities over both the transition and reward functions while penalising overly complex models. However, the size of the search space is large, requiring one dimension for every parameter of every symbol, which restricts its ability to scale to large problems.

Mugan and Kuipers [2009; 2011] iteratively discretise a continuous state space to construct a model suitable for planning. Options are then learned to reach these discretised states. This can be seen as a "symbols-first" approach, where skills are learned to achieve an initial discretisation, which is refined if necessary. This contrasts to the "skills-first" approach described in Section 2.3, where a symbolic representation is constructed from a set of skills. One issue with the former is that it relies on the symbols being *reachable* from one another, which is not a consideration when constructing symbols to support skills.

Asai and Fukunaga [2018] learn deterministic PDDL action operators directly from pixels using a binarised autoencoder, where the bottleneck layer represents the set of propositions set to true and false, but it is unclear how to extend the approach to the stochastic setting. Asai [2019] builds on this to learn deterministic object-centric representations. However, the representations are encoded implicitly and cannot be transformed into a language that can be used by existing planners. This limitation is removed by Asai and Muise [2020], who propose a deep learning approach to extract PDDL representations from image-based transition data. However, their framework does not provide soundness guarantees, nor does it consider stochastic dynamics.[6]

[6]Ahmetoglu *et al.* [2020] demonstrate that modelling probabilistic effects is critical to success in complex tasks, and that determinising probabilistic operators leads to significant degradation in performance.

Finally, approaches in the *task and motion planning* (TAMP) setting learn operator preconditions and effects from data [Wolfe *et al.* 2010; Kaelbling and Lozano-Pérez 2011]. For example, Kaelbling and Lozano-Pérez [2017] learn the continuous parameters of preconditions and effects to construct operators that can be chained together, while Wang *et al.* [2020] extend this to learn probabilistic preconditions that account for uncertainty. While TAMP approaches are used to perform high-level planning, they also capture the low-level details required by motion planners to achieve each of the individual subtasks in a plan. Consequently, the learned representations are a hybrid of discrete and continuous symbols. By contrast, approaches such as *skills-to-symbols* assume the agent's options are sophisticated enough to deal with this complexity. As such, the details for low-level motion planning need not be modelled, and the resulting representations are fully discrete.

State representation learning

The above symbol acquisition approaches have attempted to construct representations of environments using symbolic predicates. Representation learning, on the other hand, aims to discover compact state spaces from high-dimensional observations [Lesort *et al.* 2018]. A more compact representation may be formed by, for instance, discarding irrelevant state variables [Jong and Stone 2005] or selecting appropriate abstractions from a library [Konidaris and Barto 2009a; Wookey and Konidaris 2015]. Alternate approaches learn a mapping to a lower-dimensional space that preserves certain properties, such as real-world physical constraints [Jonschkowski and Brock 2015] or the Markov property [Allen *et al.* 2021b].

More recently, deep learning techniques have been applied to map the high-dimensional observations to a feature space suitable for learning a policy [Mnih *et al.* 2015], and autoencoders have been pretrained to discover a latent representation capable of reconstructing the original input [Lange and Riedmiller 2010; Oh *et al.* 2015; Finn *et al.* 2016]. The learning agent then attempts to learn a policy or value function in this latent space. In all of these cases, the learned representation is still continuous, and planning remains difficult.

State aggregation

Instead of discovering a lower-dimensional representation, an alternate approach is to aggregate collections of states into abstract states, such that certain properties of the resulting MDP are preserved. The aim here is to construct a smaller discrete MDP, where learning and planning are likely easier. As Gopalan *et al.* [2017] demonstrate, abstract MDPs (in their case, a hand-constructed hierarchy of abstractions) can be used to speed up planning significantly.

In general, these approaches involve treating groups of states as a single unit so that certain properties of the resulting MDP are preserved. For example, the strictest form of state aggregation is *bisimulation* [Givan *et al.* 2003], where only states with the same reward and transition functions are grouped. Dean and Givan [1997] introduce an approximation, where states are grouped so that the resulting clusters have transition and reward functions that are ϵ-close to the true model. Jong and

Stone [2005] propose an approach that groups states with the same optimal actions together, while Boutilier *et al.* [2000] and Chapman and Kaelbling [1991] propose an aggregation method that preserves the optimal action-value function.

All of these approaches construct a smaller, abstract MDP for which learning an optimal policy is easier. However, the resulting MDP may still be relatively large, and so constructing an optimal plan or policy using dynamic programming or MCTS can still be relatively slow. By contrast, CPDs provide structure that can be leveraged to develop domain-independent heuristics and apply fast goal-directed planning algorithms such as A* search [Hart *et al.* 1968]. Since we are interested in constructing agents capable of long-term planning, methods that produce a classical representation are better suited to our purpose.

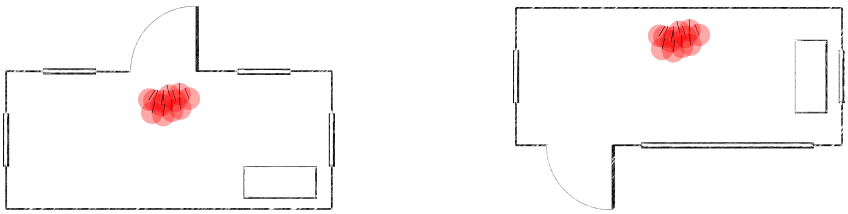
2.5 Conclusion

In this chapter, we discussed the differences between representations based on MDPs and CPDs. Although CPDs can be leveraged for fast planning, real-world agents do not receive symbolic observations. Therefore, methods that allow an agent to autonomously learn symbolic CPDs from raw observations are highly desirable. While several approaches can achieve this (see Section 2.4), they either assume that the symbolic predicates are given, or they lack the soundness guarantees of the *skills-to-symbols* framework [Konidaris *et al.* 2018]. However, even this approach is not without issues. Importantly, the learned representations are distributions over low-level state variables, and so are tied to the task in which they were learned. They therefore cannot be reused in new tasks or environments, which may be a fatal issue in real-world scenarios. In the next few chapters, we investigate ways of overcoming this deficiency.

Chapter 3

Agent-Centric Representations

In the previous chapter we described a framework for learning a classical planning representation from raw observation data. However, a major shortcoming of that framework is the lack of generalisability. Since the learned symbols are grounded in the current task, an agent must relearn the appropriate representation for each new task—or even any small change to a task— that it encounters (see Figure 3.1).



(a) The distribution over positions from where the agent is able to interact with the door.

(b) In the new task, the learned distribution is no longer useful since the door's location has changed.

Figure 3.1: An illustration of the shortcomings of learning task-specific state abstractions [Konidaris *et al.* 2018]. (a) An agent (represented by a red circle) learns a distribution over states ($\langle x, y, \theta \rangle$ tuples describing its position and orientation) in which it can interact with a door. (b) However, this distribution cannot be reused in a new room with a different layout.

Unfortunately this issue is inevitable, since state spaces are problem-specific *by definition*. Thus, apart from certain special cases, abstractions learned in one task cannot be transferred to another. To overcome this, we introduce a framework for deriving a *portable* symbolic abstraction over an agent-centric or egocentric observation space [Agre and Chapman 1987; Guazzelli *et al.* 1998; Finney *et al.* 2002; Konidaris *et al.* 2012].[1] Because such observation spaces are relative to the

[1]Egocentric observation spaces have also been adopted by recent reinforcement learning frameworks, such as *VizDoom* [Kempka *et al.* 2016], *Minecraft* [Johnson *et al.* 2016] and *Deepmind Lab* [Beattie *et al.* 2016].

agent, they provide a suitable avenue for representation transfer between tasks.

However, abstractions learned in this agent-centric space are necessarily non-Markov (since agent space is not a Markov state space by definition) and so are insufficient for planning. Our second contribution is thus to prove that the addition of very particular problem-specific information (learned autonomously from the task) to the portable abstractions results in a representation that *is* sufficient for planning. This combination of portable abstractions and task-specific information results in lifted action operators that are transferable across tasks, but which must be grounded on a per-task basis (see Figure 3.2).

We describe our framework using a simple toy domain, and then demonstrate successful transfer in two domains. Our results show that an agent is able to learn abstractions that generalise across tasks, reducing the experience required to learn a representation of a new task.

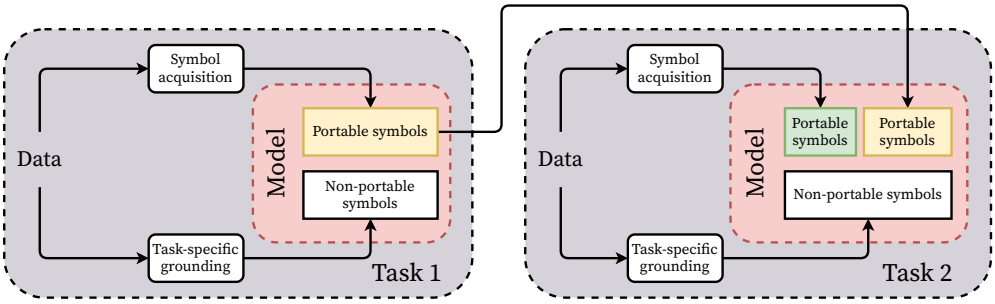


Figure 3.2: An overview of our approach. In the first task, the agent uses data to acquire transferable symbolic representations. These are then combined with problem-specific symbols, also learned from data, to form a model suitable for planning. In the second task, this process is repeated; however, symbols from the first task can be reused to construct a model for the current task.

3.1 Agent-centric observations for transfer

To develop an approach for multitask learning, we turn to the idea of *transfer learning* [Taylor and Stone 2009], the goal of which is to create an agent capable of leveraging knowledge gained in one task to improve its performance in a different but related task. We are interested in a collection of tasks modelled by a family of SMDPs.

We first consider the most basic definition of an agent, which is anything that can perceive its environment through sensors, and act upon it with effectors [Russell and Norvig 2009]. In practice, a human designer will usually build upon the observations produced by the agent's sensors to construct the Markov state space for the problem at hand, while discarding unnecessary perceptual information. Instead we will seek to effect transfer by using both the agent's sensor information—which is typically egocentric—in addition to the Markov state space.

We assume that tasks are related because they are faced by the same agent

[Konidaris *et al.* 2012]. For example, consider a robot equipped with various sensors that is required to perform a number of as yet unspecified tasks. The only aspect that remains constant across all these tasks is the presence of the robot and, more importantly, its sensors, which map the state space $\mathcal{S}$ to a portable, lossy and egocentric observation space $\mathcal{D}$ known as *agent space*. We define an observation function $\phi : \mathcal{S} \to \mathcal{D}$ that maps states to observations and depends on the sensors available to the agent. We assume the sensors may be noisy, but that the noise has mean 0 in expectation, so that if $s, t \in \mathcal{S}$, then $s = t \implies \mathbb{E}[\phi(s)] = \mathbb{E}[\phi(t)]$. To differentiate, we refer to $\mathcal{S}$ as *problem space* [Konidaris and Barto 2007].

Augmenting an SMDP with this egocentric data produces the tuple $\mathcal{M}_i = \langle \mathcal{S}_i, \mathcal{O}_i, \mathcal{T}_i, \mathcal{R}_i, \mathcal{D} \rangle$ for each task i, where the egocentric observation space $\mathcal{D}$ remains constant across all tasks. We can use $\mathcal{D}$ to define portable options, whose option policies, initiation sets and termination conditions are all defined egocentrically. Because $\mathcal{D}$ remains constant regardless of the underlying SMDP, these options can be transferred across tasks [Konidaris and Barto 2007].

3.2 Building a portable symbolic vocabulary

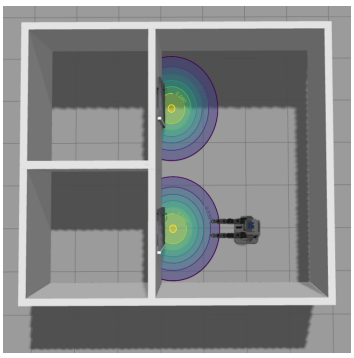
As we observed in Chapter 2, *symbols* represent precondition and effect distributions over low-level states, which are directly tied to the SMDP in which they were learned. There is thus opportunity for defining a more general representation.

Consider a navigating robot learning an abstract representation of a map, where the robot possesses a motor skill for opening a door. For any specific map, its state can be described by its orientation and xy-coordinates. If there are N doors in the environment, then the robot will learn N predicates, each of which is a distribution over coordinates specific to each door; when placed in a new map, it would have to learn everything from scratch. However, if we were to consider an egocentric state representation, we may find that the view when standing in front of each door looks identical (see Figure 3.3). We can therefore instead learn a single, generalisable symbol that names a distribution in agent space. This distribution describes the precondition of opening the multiple doors in the current map, as well as those to be found in new, as yet unseen, maps.

We therefore propose learning a symbolic representation in agent space $\mathcal{D}$. Transfer can be effected in this manner provided ϕ is *non-injective*,[2] because $\mathcal{D}$ remains consistent both within the same SMDP and across SMDPs, even if the state space or transition function do not.

Having augmented the SMDP with an agent space, we allow for the agent to possess options defined over either $\mathcal{S}$ (*problem-space options*) or $\mathcal{D}$ (*agent-space options*) [Konidaris and Barto 2007].

[2] We require that ϕ be non-injective since the ability to effect transfer relies on two distinct states in $\mathcal{S}$ being identical in $\mathcal{D}$. If that is not the case, we can do no better than learning propositional symbols in $\mathcal{S}$.



(a) Illustration of symbols as distributions over xy-coordinates, illustrated with bivariate Gaussian distributions.

(b) A single symbol viewed from an agent-centric perspective.

Figure 3.3: Representations for the symbol `InFrontOfDoor` in different state spaces in a navigation task with two doors. (a) When operating in xy-space, we require two symbols, each of which is a distribution over particular coordinates. If the doors were located at different positions, then the distributions would need to be relearned. (b) However, an agent-centric definition requires only one symbol that is independent of the door's location. We must still, however, determine *which* particular door it is, which requires information specific to the current task.

Definition 2. *Let $\mathcal{O}^{\mathcal{X}}$ be the set of all options defined over some state space $\mathcal{X}$. That is, each option $o \in \mathcal{O}^{\mathcal{X}}$ has a policy $\pi_o : \mathcal{X} \to \mathcal{A}$, an initiation set $\mathcal{I}_o \subseteq \mathcal{X}$ and a termination function $\beta_o : \mathcal{X} \to [0, 1]$.*

Using the above definition, we have that $\mathcal{O} = \mathcal{O}^{\mathcal{S}} \cup \mathcal{O}^{\mathcal{D}}$, where $\mathcal{O}^{\mathcal{S}}$ are problem-space options and $\mathcal{O}^{\mathcal{D}}$ are agent-space options. All options are assumed to obey the subgoal property in their respective state spaces. Our aim is to construct a symbolic representation suitable for planning using both types of options.

Since we wish to learn symbols in $\mathcal{D}$, a mixture of problem- and agent-space options presents a challenge. For example, given only a current distribution over agent-space observations, *we cannot determine the probability with which a problem-space option can be executed*. This follows naturally from the property that ϕ is non-injective: consider two states $s, t \in \mathcal{S}$ such that $s \neq t$ and $\phi(s) = \phi(t) = d \in \mathcal{D}$. If $s \in \mathcal{I}_o$, but $t \notin \mathcal{I}_o$, then knowledge of d alone is insufficient to determine whether we can execute o. We therefore require additional information to disambiguate such situations.

We can accomplish this by partitioning our agent-space options based on their effects in $\mathcal{S}$ (using the same procedure described in Section 2.3.3), resulting in options that are subgoal in both $\mathcal{D}$ and $\mathcal{S}$. This necessitates having access to both problem- and agent-space observations. Recall that options are partitioned to ensure the subgoal property holds, and so each partition defines its own unique image distribution. If we label each partitioned option, then each label refers to a unique distribution in $\mathcal{S}$ and is sufficient for disambiguating our agent-space symbols. Figure 3.4 illustrates the partitioning of an agent-space subgoal option that causes the agent to approach the door in its field of view.

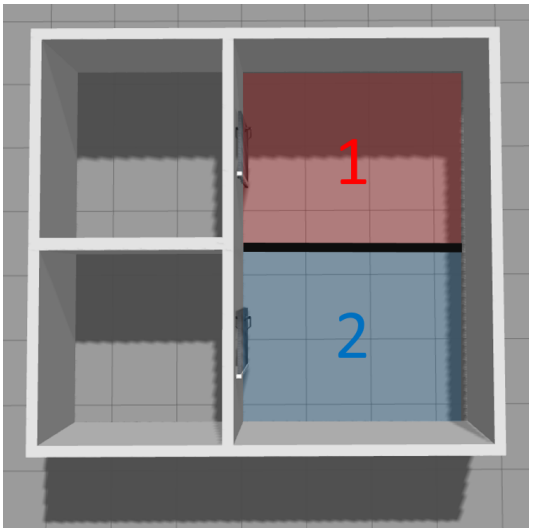


Figure 3.4: An option `GoToDoor` is subgoal in $\mathcal{D}$ but not in xy-space, since the door it approaches depends upon the agent's location. However, we can partition the option into two options, `GoToDoor(1)` and `GoToDoor(2)`, which possess the subgoal property in $\mathcal{S}$. In the above image, the red and blue regions represent the two partitions, each of which is assigned an identifier.

We can combine these partition labels with the symbol in Figure 3.3b to produce the lifted symbol `InFrontOfDoor(X)`, where `InFrontOfDoor` is the name for a distribution over $\mathcal{D}$, and `X` is simply a partition number. Note that the only time problem-specific information is required is in determining the values of `X`.

We will now show that this combination of agent-space symbols with problem-space partition numbers provides a sufficient symbolic vocabulary for planning. For convenience, we will assume that all options are already partitioned subgoal options. We denote ω as an agent-space option; $\omega(\alpha)$ as an agent-space option that has further been partitioned in *problem space* with partition label α; and $o(\alpha)$ as a problem-space option with $I_o = [\alpha] \subseteq \mathcal{S}$. We first define a function that maps problem-space partitions to subsequent problem-space partitions:

Definition 3. *A linking function L is a function that specifies the problem-space partition the agent will enter, given the current problem-space partition and executed option. That is, $L(\alpha, o, \beta) = \Pr(\beta \mid \omega, \alpha)$, where $\omega \in \mathcal{O}$, $\alpha, \beta \in \Lambda$ and Λ is the set of problem-space partitions induced by all options.*

Recall from the previous chapter that the outcome of an option execution is given by its image $\text{Im}(Z, \omega)$. If Z is an agent-space distribution and ω an agent-space option, then this information alone is insufficient for representing the true next state distribution, since agent space provides only a lossy view of the world. We will therefore model the image of an option by incorporating problem-space partition information:

$$\widehat{\text{Im}}(Z, \omega(\alpha); \beta) = \frac{\text{Im}(Z, \omega) \cdot \mathbb{1}[\alpha = \beta]}{\int_{\mathcal{S}} \Pr(s \in [\alpha]) ds}$$

The above operator states that if the current problem-space partition matches the partition of the option, then the transition probabilities are exactly those under the subgoal option. Otherwise, we cannot in fact execute the option, and so the probability is 0. Thus we consider only starting states in $[\alpha]$ and renormalise to ensure that the transition remains a proper distribution.

Theorem 1. *The ability to represent the preconditions and image of each option in agent space, together with the partitioning in $\mathcal{S}$, is sufficient for determining the probability of being able to execute any probabilistic plan p from starting distribution Z.*

Proof. We assume, without loss of generality, that p_Z is a plan consisting of a number of agent-space options followed by a problem-space option: $p_Z = \{\omega_0(\alpha_0), \ldots, \omega_{n-1}(\alpha_{n-1}), o(\alpha_n)\}$. We denote the initial agent- and problem-space distributions as D_0 and S_0 respectively. The image of an option in agent space is specified by the operator

$$Z_{i+1} = \widehat{\mathrm{Im}}(Z_i, \omega_i(\cdot); \alpha_i), \text{ with } Z_0 = D_0.$$

The probability of being able to execute p_Z is given by

$$\Pr(x_0 \in I_{\omega_0}, \ldots, x_{n-1} \in I_{\omega_{n-1}}, s_n \in I_{o(\alpha_n)}),$$

where $x_i \sim Z_i$ and $s_n \sim \Pr(\cdot \mid s_0, \omega_0, \ldots, \omega_{n-1})$. By the Markov property, we can write this as

$$\Pr(s_n \in I_{o(\alpha_n)}) \prod_{i=0}^{n-1} \left[\Pr(x_i \in I_{\omega_i})\right].$$

If we can estimate the starting problem-space partition α_0 and linking function L, then we can evaluate this quantity as follows:

$$\begin{aligned}
\Pr(s_n \in I_{o(\alpha_n)}) &= \Pr(s_n \in [\alpha_n]) \\
&= \Pr(s_0 \in [\alpha_0]) \prod_{i=0}^{n-1} L(\alpha_i, \omega_i(\alpha_i), \alpha_{i+1}) \\
&= \int_{\mathcal{S}} \Pr(s \in [\alpha_0]) S_0(s) ds \times \prod_{i=0}^{n-1} L(\alpha_i, \omega_i(\alpha_i), \alpha_{i+1}),
\end{aligned}$$

and

$$\Pr(x_i \in I_{\omega_i}) = \prod_{j=1}^{i} L(\alpha_{j-1}, \omega_{j-1}, \alpha_j) \times \int_{\mathcal{D}} \Pr(x_i \in I_{\omega_i}) Z_i(x, \omega_i(\alpha_i); \alpha_i) dx.$$

Thus by learning the precondition and image operators in $\mathcal{D}$, partitioning the options in problem space, and learning the links between these partitions, we can evaluate the probability of an arbitrary plan executing. □

3.3 Generating a forward model

The previous section provided the sufficient vocabulary for learning a problem-specific model. We now show how to build a forward model by combining agent-space distributions and partition labels, which can be viewed as a two-step process. The agent first learns domain-independent symbols in agent space (which are not strictly tied to the current task), and then determines their parameters and how they link to each other, which depends on $\mathcal{S}$ and thus the current task.

The first phase is equivalent to learning propositions in agent space, which are portable because agent space is shared between SMDPs. An example of this is an agent learning that, after passing through a door, it finds itself in a room. There can be multiple such effects (e.g. it may land in a hallway), and we learn one rule for each. The second phase learns the specifics of the current environment—which doors connect to which rooms, for example. This involves learning how the parameter of a rule's precondition relates to the parameter of its effect.

3.4 Learning portable representations

To aid in explanation, we make use of a simple continuous task where a robot navigates the building illustrated in Figure 3.5a. The problem space is the xy-coordinate of the robot, while additionally the robot possesses sensors that allow it to detect nearby walls and windows. The agent space is the output of these egocentric sensors, observations from which are illustrated by Figures 3.5(b–d).

The robot is equipped with options to move between different regions of the building, halting when it reaches the start or end of a corridor. It possesses the following four options: (a) `Clockwise` and `Anticlockwise`, which move the agent in a clockwise or anticlockwise direction respectively; (b) `Outward`, which moves the agent down a corridor away from the centre of the building; and (c) `Inward`, which moves it towards the centre.

We could adopt the approach of Konidaris *et al.* [2018] to learn an abstract representation using transition data in $\mathcal{S}$. However, that procedure generates symbols that are distributions over xy-coordinates, and are thus tied directly to the particular problem configuration. If we were to simply translate the environment along the plane, the xy-coordinates would be completely different, and our learned representation would be useless.

Instead, the agent proceeds to learn a symbolic representation using transition data in $\mathcal{D}$ produced by its sensors. After applying the skills-to-symbols framework, the agent identifies three portable symbols, which are exactly those illustrated by Figures 3.5(b–d). The learned operators are listed in Table 3.1, where it is clear that naïvely considering egocentric observations alone is insufficient for planning purposes: the agent does not possess an option with probabilistic outcomes, but the `Inward` option appears to have probabilistic effects. This is due to aliasing, where a single option can be instantiated in different states that have the same egocentric view.

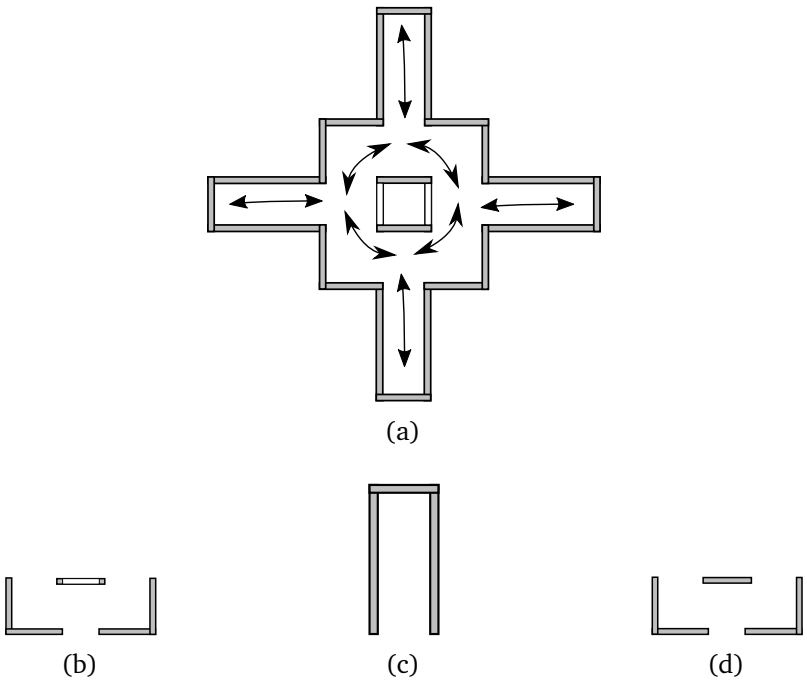


Figure 3.5: (a) A continuous navigation task where an agent can navigate between different regions in xy-space. Walls are represented by grey lines, while the two white bars represent windows. Arrows describe the agent's options. (b–d) Local observations produced by the agent's sensors, regardless of the agent's orientation. We name these `window-junction`, `dead-end` and `wall-junction` respectively. Note that these observations are independent of the agent's position and building configuration.

Option	**Precondition**	**Effect**
`Clockwise1`	`wall-junction`	`window-junction`
`Clockwise2`	`window-junction`	`wall-junction`
`Anticlockwise1`	`wall-junction`	`window-junction`
`Anticlockwise2`	`window-junction`	`wall-junction`
`Outward1`	`wall-junction`	`dead-end`
`Outward2`	`window-junction`	`dead-end`
`Inward`	`dead-end`	$\begin{cases}$ `window-junction` w.p. 0.5 `wall-junction` w.p. 0.5

Table 3.1: A list of the seven learned subgoal options, specifying their preconditions and effects in agent space only.

As mentioned, we can correct this by *re-partitioning* our option using problem-space data, labelling the subsequent partitions, and then combining these labels

with our portable symbols to generate a sound representation. Figure 3.6 annotates the domain with labels according to their problem-space partitions, but note that the partition numbers are completely arbitrary.

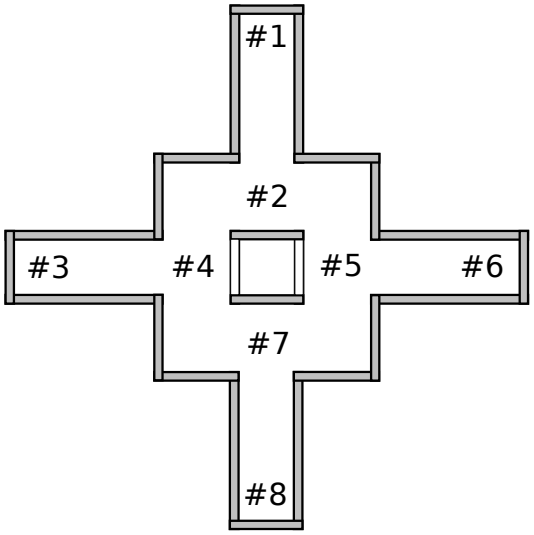


Figure 3.6: Each number refers to the initiation set of an option partitioned in problem space. For readability, we merge identical partitions. For instance, `#2` refers to the initiation sets of a single problem space partition of `Outward`, `Clockwise` and `Anticlockwise`.

Generating agent-space symbols results in lifted symbols such as `dead-end(X)`, where `dead-end` is the name for a distribution over $\mathcal{D}$, and `X` is a partition number that must be determined on a per-task basis. Note that the *only* time problem-specific information is required is to determine the values of `X`, which grounds the portable symbol in the current task.

3.5 Generating a task-specific model

We now summarise our approach, which can be viewed as a two-step process. The first phase uses the procedure illustrated in Figure 2.6 in the previous chapter to learn portable symbolic rules using agent-space transition data only. The second phase uses problem-space transitions to partition options in $\mathcal{S}$. The partition labels are then used as parameters to ground the previously learned portable operators in the current task. We use these labels to learn *linking functions* that connect precondition and effect parameters. For example, when the parameter of `Anticlockwise2` is `#5`, then its effect should take parameter `#2`. Figure 3.7 illustrates this grounding process.

These linking functions are learned by simply executing options and recording the start and end partition labels of each transition. We use a simple count-based approach that records the fraction of transitions from one partition label to another for each option. Let $\Gamma^{(o)}$ be the set of problem-space partition labels for option o, and $\Lambda = \bigcup_{o \in \mathcal{O}} \Gamma^{(o)}$ the set of all partition labels over all options. Note that each

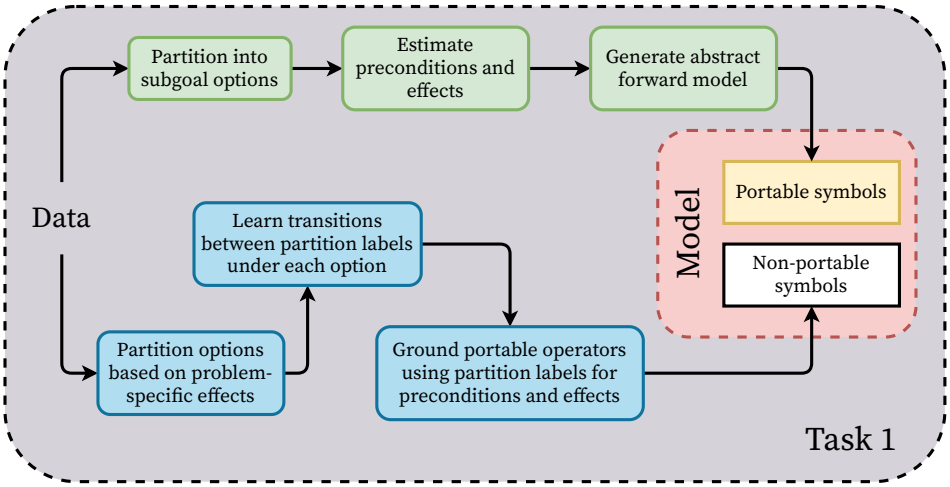


Figure 3.7: The full process of learning portable representations from data. In green nodes, the agent learns representations using egocentric data, while blue nodes denote where the agent learns using problem-space data from the current task only.

label $\lambda \in \Lambda$ refers to a set of initiation states $[\lambda] \subseteq \mathcal{S}$. Our approach is as follows:

1. Given a set of agent-space subgoal options, partition them further so that each of the partitioned options possesses the subgoal property in problem-space $\mathcal{S}$. Gather data from trajectories, and record tuples $\langle s, d, o, s', d' \rangle$ representing initial states in both $\mathcal{S}$ and $\mathcal{D}$, the executed option, and the subsequent states.

2. Determine the start and end partitions of the transition. The start partition is the singleton $c = \{\gamma \mid \gamma \in \Gamma^{(o)}, s \in [\Gamma^{(o)}]\}$, while the end labels are given by the set $\beta = \{\lambda \mid \lambda \in \Lambda, s' \in [\lambda]\}$. In practice, we keep all states belonging to each partition and then calculate the L2 norm to the closest states in each partition. We select those partitions whose distance is less than some threshold.

3. Denote L_o as the linking function for option o which stores the number of times transitions between different partition labels occur. Increment the existing count stored by $L_o(c, \beta)$, and keep count of the number of times the entry (o, c) has been updated.

4. Normalise the linking functions L_o by dividing the frequency counts by the number of times the entry for c was updated. We have now learned the link between the parameters of the precondition and effect for each option.

A combination of portable operators and partition numbers reduces planning to a search over the space $\Sigma \times \mathbb{N}$, where Σ is the set of generated symbols. Alternatively (and equivalently), we can generate either a factored MDP or a PPDDL representation [Younes and Littman 2004]. To generate the latter, we first specify predicates for the three symbols derived in the previous sections: `window-junction`, `dead-end` and `wall-junction`. We can then use a fluent (a real-valued predicate) named `partition` to store the current partition number. Figure 3.8 lists an example operator, where the effects depend on `partition` and the values of `partition` are reflected in the labelling in Figure 3.6. The full domain description is provided in Appendix A.1.

```
(:action Outward_2
:parameters()
:precondition ((wall-junction))
:effect (and (when (= (partition) 2) (and (dead-end)
             (not (wall-junction) (assign (partition) 1)))
             (when (= (partition) 7) (and (dead-end)
             (not (wall-junction) (assign (partition) 8)))
        )
)
```

Figure 3.8: Generated PPDDL for the `Outward` option initiated at a wall-junction node. In this instance, the linking function has recorded that transitions occur between partitions 2 and 1, as well as between 7 and 8.

3.6 Inter-task transfer

In the above example, it is not clear why one would want to learn portable symbolic representations—we perform symbol acquisition in $\mathcal{D}$ *and* instantiate the operators for the given task, which requires more computation than directly learning symbols in $\mathcal{S}$. We now demonstrate the advantage of doing so by learning portable models of two different domains, both of which feature continuous state spaces and probabilistic transition dynamics.

3.6.1 Domain descriptions

In our first domain, which we term *Rod-and-Block*, a rod is constrained to move along a track. The rod can be rotated into an upward or downward position, while a number of blocks are arranged to impede the rod's movement. Two walls are also placed at either end of the track. One such task configuration is illustrated by Figure 3.9.

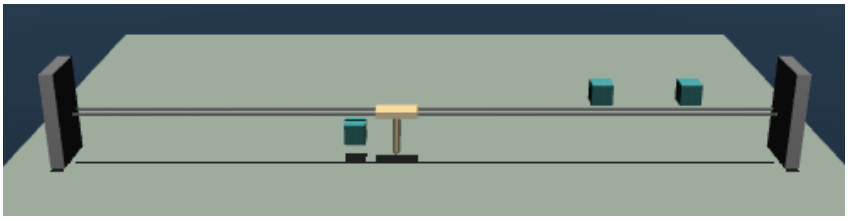


Figure 3.9: The *Rod-and-Block* domain consisting of a number of blocks and a rod that can be rotated. This particular task consists of three obstacles that prevent the rod from moving along the track when the rod is in either the upward or downward position. Different tasks are characterised by different block placements.

The problem space consists of the rod's angle and its x position along the track. Egocentric observations return the types of objects (i.e. boxes or walls) that are in close proximity to the rod, as well as its angle. In Figure 3.9, for example, there is a block to the left of the rod, which has an angle of π. The high-level options given to the agent are `GoLeft`, `GoRight`, `RotateUp`, and `RotateDown`. The first two translate

the rod along the rail until it encounters a block or wall while maintaining its angle. The remaining options rotate the rod into an upward or downward position, provided it does not collide with an object. These rotations can be done in both a clockwise and anti-clockwise direction.

Our second domain is a higher-dimensional video game known as the *Treasure Game* [Konidaris *et al.* 2015], where an agent navigates a maze in search of treasure. This domain contains ladders and doors which impede the agent. Some doors can be opened and closed with levers, while others require a key to unlock. Figure 3.10 illustrates one level of the game.

Figure 3.10: The layout of a single *Treasure Game* level.[3] Note that in order to retrieve the treasure, the agent must execute a high-level plan, which in practice consists of hundreds of low-level actions. First, it must climb down the ladder and pull the lever to open the door. After collecting the key, it must use it to unlock the bottom-most door. Only then is the treasure accessible.

The problem space consists of continuous variables denoting the xy-position of the agent, key and treasure, the angle of the levers (which determines whether a door is open) and the state of the lock. We construct the agent space by first tiling the screen into cells.[4] We then produce a vector of length 9, the elements of which are the type of sprites in each of the eight adjacent neighbouring cells centred on the agent. This represents the agent's local view of its environment. We also append the "bag" of items carried by the agent, which could be the key or the treasure, to its local view. The agent possesses the following high-level options:

- `GoLeft` and `GoRight`: the agent moves left or right continuously until it reaches a point of interest, such as a ladder, a lever, a wall or a ledge.
- `UpLadder` and `DownLadder`: the agent climbs up and down a ladder. These skills can only be executed when the agent is standing below or above a ladder respectively.

[4]In both domains we defined the agent space to be a local view about the agent. This does not imply that agent space should always be defined in this manner—the choice of agent space depends on the domain, but is directly specified by an embodied agent's sensors.

- `FallLeft` and `FallRight`: the agent jumps down from a ledge to the ground below.
- `JumpLeft` and `JumpRight`: the agent jumps up to a higher ledge from below. Occasionally the agent will not jump far enough and will fall to the ground.
- `Interact`: the agent pulls a lever when standing near one, which in turn opens or closes a door. The skill also unlocks a padlock when the agent is nearby and holding a key.

The outcome of options are stochastic—for jumping skills, the agent will sometimes fail to reach its target and fall to the ground, while other navigation skills have termination conditions that vary by a few pixels.

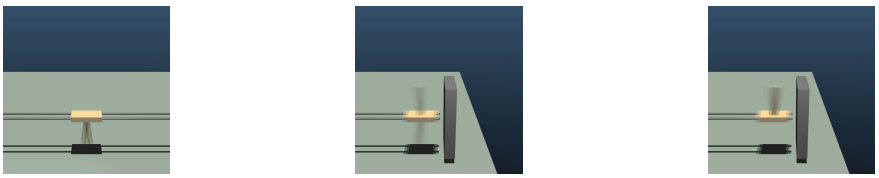
3.6.2 Learning portable representations

For both domains we learn a symbolic representation using agent-space transitions only, following the same procedure described in Section 2.3. We list the hyperparameters for these various steps in Tables A.1 and A.2 in the appendix. First, we collect data from a task by executing options uniformly at random and scaling the observations to be in the range $[0, 1]$. We record state transition data and which options could be executed at each state. We then partition options following the procedure in Section 2.3.3 using the DBSCAN clustering algorithm [Ester *et al.* 1996], which approximately preserves the subgoal property.

Next, the agent learns a precondition classifier for each of these approximately partitioned options using an SVM with Platt scaling [Cortes and Vapnik 1995; Platt 1999]. We use the feature selection procedure in Figure 2.8 to determine which state variables are relevant to each option's precondition. We first compute the accuracy of the SVM applied to all variables, performing a grid search to find the best hyperparameters for the SVM using 3-fold cross validation. Then we check the effect of removing each state variable in turn, recording those that cause the accuracy to decrease. Finally we check whether adding each of the state variables back improves the SVM's accuracy, in which case they are kept too. Having determined the features, we fit a probabilistic SVM to the relevant state variables' data.

A kernel density estimator [Rosenblatt 1956; Parzen 1962] with Gaussian kernel is next used to estimate the effect of each partitioned option. We learn distributions over only the variables in the option's mask (i.e. those variables modified by the option). We use a grid search with 3-fold cross validation to find the best bandwidth hyperparameter for each estimator.

For each partitioned option, we now have a classifier and set of effect distributions. We finally use the approach outlined in Figure 2.9 to generate the PPDDL representations. Examples of learned operators are illustrated by Figures 3.11 and 3.12, while more examples for both domains can be found in Appendices A.4 and A.5. These operators can be reused for new tasks—we need not relearn them when we encounter a new task, although we can always use data from a new task to improve them. We contrast this with a non-portable operator learned by the framework of Konidaris *et al.* [2018], which is illustrated by Figure 3.13.



(a) symbol_18 (b) symbol_11 (c) symbol_12

```
(:action Up Clockwise_1
 :parameters()
 :precondition (and (symbol_18) (symbol_11))
 :effect (and (symbol_12) (not (symbol_18)))
)
```

(d)

Figure 3.11: (a–b) The precondition for RotateUpClockwise1, which states that in order to execute the option, the rod must be left of a wall facing down. The precondition is a conjunction of these two symbols—the first (symbol_18) is a distribution over the rod's angle only, while the second (symbol_11) is a distribution over its position relative to the wall. (c) The effect of the option, with the rod adjacent to the wall in an upward position. (d) PPDDL description of the above operator, which is used for planning.

(a) symbol_34 (b) symbol_32 (c) symbol_9 (d) symbol_15 (e) symbol_18

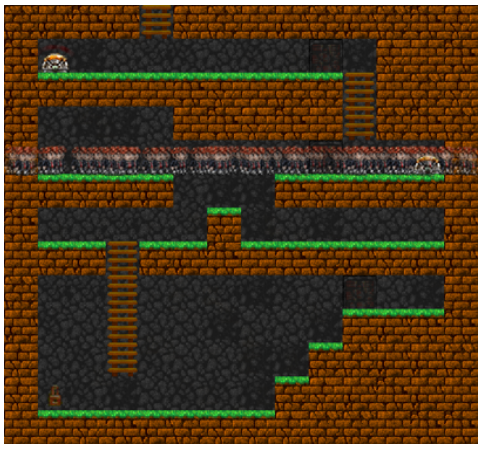
```
(:action interact_option-partition-2-70
 :parameters ()
 :precondition (and (notfailed) (symbol_34) (symbol_32))
 :effect (and (symbol_9) (symbol_15) (symbol_18)
         (not (symbol_32)) (not (symbol_34)))
)
```

(f)

Figure 3.12: (a–b) The precondition for an Interact option, which states that the agent must be standing at a closed lock (symbol_34) and must possess the key (symbol_32). (c–e) After executing the option, the agent no longer has the key (symbol_9), the door to its right is open (symbol_15) and the lock is open (symbol_18).

(a) `symbol_16`

(b) `symbol_24`



(c) `symbol_9`

```
(:action down_ladder-partition-2
 :parameters()
 :precondition (and (symbol_16)
                    (symbol_24))
 :effect (and (symbol_9)
              (not (symbol_24)))
)
```

(d)

Figure 3.13: (a–b) The precondition of a non-portable operator for climbing down a ladder. In order to execute the option, the agent must be at a particular xy-location (`symbol_16` and `symbol_24` respectively). (c) The effect of the option, where only the y-position of the agent has changed (`symbol_9`). (d) PPDDL description of the above operator, which is used for planning. Note that because the operator's preconditions and effects depend on the agent's xy-position, it is unlikely to be applicable in a new task.

3.6.3 Transfer experiments

Once we have learned sufficiently accurate portable operators, they need only be instantiated for the given task by learning the linking between partitions. This requires far fewer samples than classification and density estimation over the state space $\mathcal{S}$, which is required to learn a task-specific representation.

To illustrate this, we construct a set of ten tasks $\rho_1, \ldots, \rho_{10}$ for each domain. For the *Rod-and-Block* domain, we randomly select the number of blocks and then randomly position them along the track. Because tasks have different configurations, constructing a symbolic representation in problem space requires relearning a model

of each task from scratch. However, when constructing an agent-centric representation, symbols learned in one task can immediately be used in subsequent tasks. We gather k transition samples from each task by executing options uniformly at random, and use these samples to build both task-specific and egocentric (portable) models.

In order to evaluate a model's accuracy, we randomly select 100 goal states for each task, as well as the optimal plans for reaching each from some start state. Each plan consists of two options, and we denote a single plan by the tuple $\langle s_1, o_1, s_2, o_2 \rangle$. Let $\mathcal{M}_k^{\rho_i}$ be the model consisting of high-level preconditions and effects constructed for task ρ_i using k samples. We calculate the likelihood of each optimal plan under the model: $\Pr(s_1 \in I_{o_1} \mid \mathcal{M}_k^{\rho_i}) \times \Pr(s' \in I_{o_2} \mid \mathcal{M}_k^{\rho_i})$, where $s' \sim \text{Eff}(o_1)$. We build models using increasing numbers of samples in steps of 250, until the likelihood averaged over all plans is greater than some acceptable threshold (we use a value of 0.75), at which point we continue to the next task. The results are given by Figure 3.14.

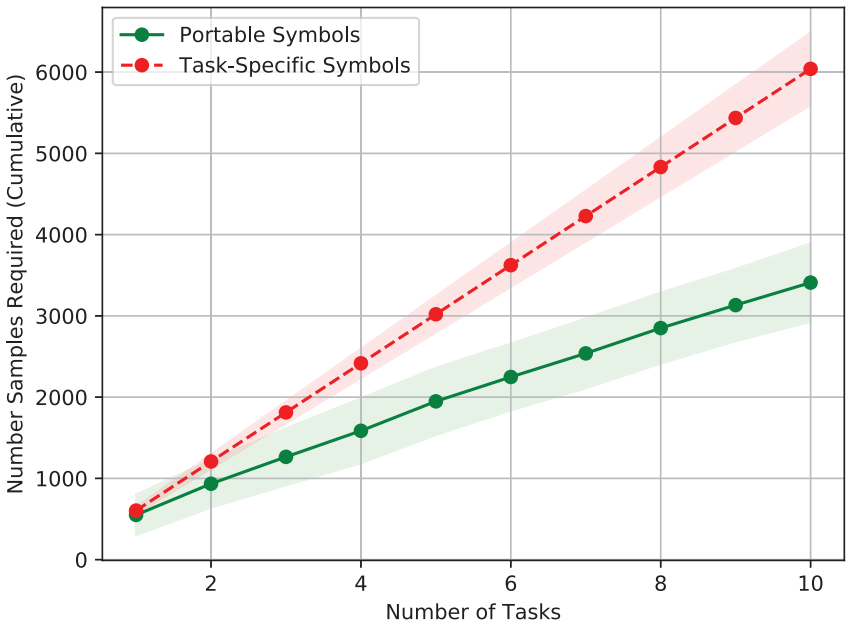


Figure 3.14: Results for the *Rod-and-Block* domain. Cumulative number of samples required to learn sufficiently accurate models as a function of the number of tasks encountered. Results are averaged over 100 random permutations of the task order. Standard errors are specified by the shaded areas. The agent requires about 600 samples to learn a task-specific model of each *Rod-and-Block* configuration. Conversely, when the agent learns and reuses portable representations, the number of samples required decreases to roughly 330 after only two tasks.

We repeat this process for the *Treasure Game* by constructing ten different levels,[5] and investigate the sample efficiency of our approach. An example of a portable operator, as well as its problem-space partitioning, is given by Figure 3.15, while the number of samples required to learn a good model of all 10 levels is given by Figure 3.16.

(a) (b)

```
(:action DownLadder_1
 :parameters()
 :precondition (and (symbol_80))
 :effect (and (symbol_622) (not symbol_80))
)
```

(c)

Figure 3.15: (a) The precondition (top) and positive effect (bottom) for the `DownLadder` operator, which states that in order to execute the option, the agent must be standing above the ladder. As a result, the agent finds itself standing on the ground below the ladder. The black spaces refer to unchanged low-level state variables. (b) Three problem-space partitions for the `DownLadder` operator. Each of the circled partitions is assigned a unique label and combined with the portable rule in (a) to produce a grounded operator. (c) The PPDDL representation of the operator specified in (a).

3.6.4 Discussion

Naturally, learning problem-space symbols results in a sample complexity that scales linearly with the number of tasks, since we must learn a model for each new task from scratch. Conversely, by learning and reusing portable symbols, we can reduce the number of samples we require as we encounter more tasks, leading to a *sublinear* increase. The agent initially requires about 600 samples to learn a task-specific model of each *Rod-and-Block* configuration, which decreases to roughly 330 after

[5] We made no effort to design tasks in a curriculum-like fashion. The levels are given in Appendix A.6.

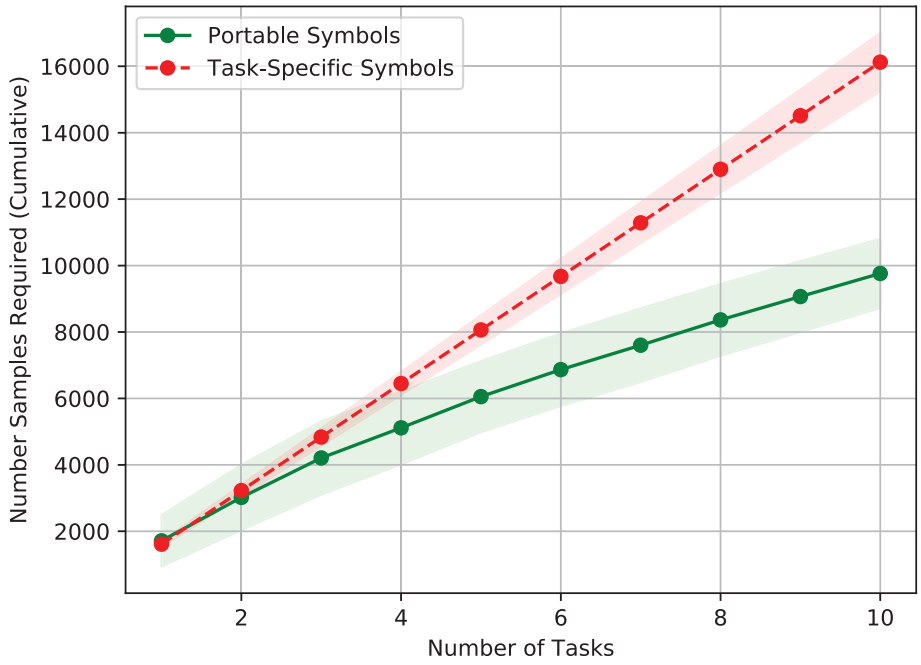


Figure 3.16: Results for the *Treasure Game* domain. Cumulative number of samples required to learn sufficiently accurate models as a function of the number of tasks encountered. Results are averaged over 100 random permutations of the task order. Standard errors are specified by the shaded areas. The agent requires 1600 samples for each level when learning task-specific representations. However, after learning portable representations in four tasks, the agent requires only 900 samples to learn a model of the next task, and about 700 after seven tasks.

only two tasks. Similarly, 1600 samples are initially needed for each level of the *Treasure Game*, but only 900 after four levels, and about 700 after seven.

Intuitively one might expect the number of samples to plateau as the agent observes more tasks. That we do not is as a result of the simple exploration policy used here—the agent must observe all relevant partitions at least once, and selecting actions uniformly at random is naturally suboptimal. Nonetheless, we still require far fewer samples to learn the links between partitions than learning a full model from scratch.

In both of our experiments, we construct a set of 10 domain configurations and then test our approach by sampling 100 goals for each, for a total of 1000 tasks per domain. Our model-based approach learns 10 forward models, and then uses them to plan a sequence of actions to achieve each goal. By contrast, a model-free approach would be required to learn all 1000 policies, since every goal defines another unique SMDP that must be solved. Furthermore, it is unclear how to extend these techniques to deal with tasks with different state space dimensionalities.

Our approach treats the problem as a two-step procedure: we first learn representations using only $\mathcal{D}$, and then use only $\mathcal{S}$ to ground the representations to our current task. A naïve alternative would be to simply combine $\mathcal{S}$ and $\mathcal{D}$ and learn

representations over the combined state space. However, doing so would result in models that are not wholly transferable. For example, in the *Treasure Game*, the agent would learn that to climb down a ladder, it must be standing on top of the ladder *and* at some specific xy-position. In a new task, the agent would recognise that it is standing on a ladder, but its coordinates would likely be different and so the precondition would not apply.

We depart from most model-based approaches in that we rely on the portable observation space $\mathcal{D}$ for transfer. This raises questions regarding how difficult it is to specify $\mathcal{D}$, and the sensitivity of the egocentric observation space to the resulting representations. Fortunately, it is not too hard to provide an egocentric view of the agent: as mentioned, for many real-world problems with embodied agents, this amounts to the agent carrying its own sensors, while for simulated problems (such as those presented here) one can simply centre the input observation on the agent's reference frame. We note, too, that there has been work on autonomously discovering portable observation spaces [Snel and Whiteson 2010], but this is orthogonal to our work.

Finally, we remark that transfer will naturally depend on, and be sensitive to, the characteristics of $\mathcal{D}$. The question of sensitivity has been extensively studied in the context of learning a single policy [Konidaris *et al.* 2012, Section 4.3.4], where results indicate that policy learning erodes gradually with the usefulness of $\mathcal{D}$. Practically, this is a concern for learning the option policies, and so we will only state that if $\mathcal{D}$ is sufficient to learn the options (which we assume has already taken place), then it is sufficient to learn the corresponding representations.

3.7 Related work

Several methods in the fields of meta-learning and lifelong learning focus on discovering an internal or latent representation that generalises across a distribution of tasks [Jonschkowski and Brock 2015; Higgins *et al.* 2017; Kirkpatrick *et al.* 2017; Finn *et al.* 2017; de Bruin *et al.* 2018]. When presented with a new task, agents subsequently learn a policy based on its internal representation in a model-free manner. In contrast, our approach learns an explicit model which supports forward planning, and is independent of the task or reward structure.

Relocatable action models [Sherstov and Stone 2005; Leffler *et al.* 2007] assume states can be aggregated into "types" which determine the transition behaviour. State-independent representations of the outcomes from different types are learned and improve the learning rate in a single task. However, the mapping from lossy observations to states is provided to the agent, since learning this mapping is as difficult as learning the full MDP.

More recently, Zhang *et al.* [2018] propose a method for constructing portable representations for planning, but the mapping to abstract states is provided, and planning is restricted solely to agent space. Similarly, Srinivas *et al.* [2018] learn a goal-directed latent space in which planning can occur. However, the goal must be known upfront and be expressible in the latent space; both are therefore unsuited

to tasks with goals defined in problem space.

Finally, certain approaches have extended or adapted the *skills-to-symbols* framework. Ames *et al.* [2018] learn PPDDL operators for parameterised actions by intelligently discretising the parameters. The resulting representation is fully propositional (even if the actions are not) and cannot be transferred across tasks. Pacheck *et al.* [2019] leverage the symbol-learning framework to construct symbols formulated using linear temporal logic, and then automatically learn new skills if the resulting plan fails. However, the same dynamics are shared across tasks. Gopalan *et al.* [2020] cluster effects using agent-centric observations, resulting in portable symbols that can be transferred to new tasks. These symbols (clusters) are then sequenced by incorporating natural language instructions, which mitigates the aliasing caused by partial observability. In contrast, we use the problem-specific partition labels from the state space itself to overcome this issue.

3.8 Summary

We have introduced a framework for autonomously learning portable representations for planning. Previous work [Konidaris *et al.* 2018; Ames *et al.* 2018] has shown how to learn a high-level representation suitable for planning, but these representations are directly tied to the task in which they were learned. Ultimately, this is a fatal flaw—should any of the environments change even slightly, the entire representation would need to be relearned from scratch. Conversely, we demonstrate that an agent is able to learn a portable representation given only data gathered from option execution. We also show that the addition of particular problem-specific information results in a representation that is provably sufficient for learning a sound representation for planning, which allows agents to leverage past experience in solving new unseen tasks.

Chapter 4

Object-Centric Representations

In the previous chapter, we used agent-centric observations to construct portable symbolic representations. One downside to this approach is that an agent will generally only have a partial view of its environment. If there are state variables that the agent cannot always observe, then these must be included in problem space, which is *not* transferable. For example, consider a domain consisting of several objects that must be manipulated. There is no issue if all objects are in full view of the agent's sensors, such as table-top manipulation tasks. In this case, the objects' states are contained in agent space and can be transferred to new tasks. However, if the agent cannot observe all of the objects simultaneously (for instance, if the objects are spatially distant), then their states must be included in our problem space definition, hampering transfer.

In this chapter, we tackle this problem by incorporating additional structure—namely, by assuming that the world consists of objects, and that similar objects are common amongst tasks. This assumption can substantially improve learning efficiency, because an object-centric model can be reused wherever that same object appears (within the same task, or across different tasks) and can also be generalised across objects that behave similarly—object *types*.

We assume that an agent is able to individuate the objects in its environment and possesses a set of skills that can be applied to them. Using these assumptions, we propose a framework for building portable object-centric abstractions given only the data collected by executing high-level skills. These abstractions specify both the abstract object attributes that support high-level planning and an object-relative lifted transition model. We then show how to integrate problem-specific information to instantiate these representations in a new task. Our approach reduces the samples required to learn a new task by allowing the agent to avoid relearning the dynamics of previously seen objects.

We demonstrate our approach on a Blocks World domain, and then apply it to a series of hard Minecraft tasks where an agent autonomously learns a PPDDL representation of a high-dimensional task from raw pixel input. Our results show that an agent can leverage these portable abstractions to learn a representation of new Minecraft tasks using a diminishing number of samples, allowing it to quickly construct plans consisting of hundreds of low-level actions.

4.1 Learning object-centric representations

In the previous chapter, we assumed the existence of an agent equipped with sensors, which led to the idea of agent space. Since we are now assuming the existence of objects, a natural extension is to introduce the notion of *object space*. We adopt the object-centric formulation from Ugur and Piater [2015a]: in a task with n objects, the state is represented by the set $\{\mathbf{f}_a, \mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n\}$, where $\mathbf{f}_a$ is a vector of the agent's features and $\mathbf{f}_i$ is a vector of features particular to object i. Note that the feature vector describing each object can itself be arbitrarily complex, such as an image or voxel grid—in this work we use pixels. Such a representation is common in robotics, where each object is often isolated from the environment and represented as a point cloud or voxelised occupancy grid.

The object-space representation assumes the state space has already been factored into its constituent objects. Practically, this means that the agent is aware that the world consists of objects, but is unaware of what the objects are, or whether there are multiple instantiations of the same object present. It is also easy to see that different tasks will likely have differing numbers of objects with potentially arbitrary ordering; any learned abstract representation should be agnostic to this.

We now introduce an object-centric generalisation of a learned symbolic representation that admits transfer in tasks when the state-space representation consists of features centred on objects in the environment. We summarise our approach in Figure 4.1, and describe the details below.[1]

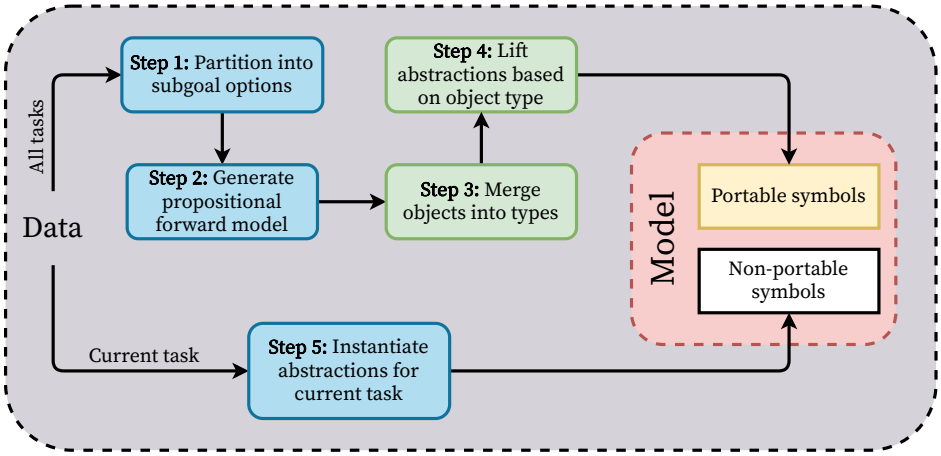


Figure 4.1: Learning lifted representations from data. Blue nodes represent problem-specific representations, while green nodes are abstractions that can be transferred between tasks.

[1]Pseudocode describing our entire approach to building a typed, object-centric PPDDL representation can be found in Appendix B.1.

4.1.1 Generating a propositional model (Steps 1–2)

We first follow the same procedure outlined in Section 2.3.3 to construct a non-portable, propositional PPDDL representation of a single task. Since object space is already factored into the constituent objects, each proposition in the resulting vocabulary will refer to a distribution over a particular object's state.

4.1.2 Generating a typed model (Steps 3–4)

At this point, there is no opportunity for transfer (both within the task and between different tasks), because each object is treated as unique and is tied to its exact index in the state vector. To overcome this, we propose to estimate object *types* using the PPDDL preconditions and effects learned in the previous section.

Definition 4. *Assume that option o has been partitioned into n subgoal options $o(1), \ldots, o(n)$. Object i's profile under option o is denoted by the set*

$$\textit{Profile}(i, o) = \left\{ \{\textit{Pre}_i^{o(1)}, \mathcal{E}_i^{o(1)}\}, \ldots, \{\textit{Pre}_i^{o(n)}, \mathcal{E}_i^{o(n)}\} \right\},$$

where $\textit{Pre}_i^{o(k)}$ is the distribution over object i's states present in the precondition for partition k, and $\mathcal{E}_i^{o(k)}$ is object i's effect distribution.[2]

Definition 5. *Two objects i and j are option-equivalent if, for a given option o, $\textit{Profile}(i, o) = \textit{Profile}(j, o)$. Furthermore, two objects are equivalent if they are option-equivalent for every o in $\mathcal{O}$.*

The above definition implies that objects are equivalent if one object can be substituted for another while preserving every operator's abstract preconditions and effects. Such objects can be grouped into the same *object type*, since they are functionally indistinguishable for the purposes of planning. In practice, however, we can use a weaker condition to approximate object types. Since an object-centric skill will usually modify only the object being acted upon, and because we have subgoal options that do not depend on the initial state, we can take a similar approach to Ugur and Piater [2015a] and group objects by effects only:

Definition 6. *Assume that option o has been partitioned into n subgoal options $o(1), \ldots, o(n)$. Object i's effect profile under option o is denoted by the set*

$$\textit{EffectProfile}(i, o) = \left\{ \mathcal{E}_i^{o(1)}, \ldots, \mathcal{E}_i^{o(n)} \right\},$$

where $\mathcal{E}_i^{o(k)}$ is object i's effect distribution. Two objects i and j are effect-equivalent if $\textit{EffectProfile}(i, o) = \textit{EffectProfile}(j, o)$ for every o in $\mathcal{O}$.

The notion of effect equivalence was first proposed by Şahin *et al.* [2007]. Such an approach assumes that an object's type depends on both the intrisic properties of the object itself, as well as the agent's endowed behaviours [Chemero 2003].

[2] These precondition and effect distributions can be null where appropriate.

However, this definition does not take into account the importance of preconditions, nor does it consider interactions involving multiple objects. Nonetheless, the approach will prove sufficient for our purposes (see Section 4.2), and we leave a more complete definition to future work.

The pseudocode in Figure 4.2 demonstrates how the effect profile for each object can be computed using the propositional representation. Having done so, the agent determines whether objects i and j are similar (using an appropriate measure of distribution similarity) and, if so, merges them into the same object type. Propositions representing distributions over individual objects can now be replaced with predicates that are parameterised by types (see Figure 4.3).

procedure COMPUTEEFFECTS
  Given: object i, option o, PPDDL operators $Operators$
  ▷ Get only the operators that model option o
  $Operators \leftarrow \{operator \mid operator \in Operators, \textsc{RefersTo}(operator, o)\}$
  $Effects \leftarrow \emptyset$
  for each $\{\cdot, effect\} \in Operators$ **do**
    ▷ Extract the effect propositions that refer to distributions over object i
    $OperatorEffect \leftarrow \{prop \mid prop \in effect, \textsc{RefersTo}(prop, i)\}$
    $Effects \leftarrow Effects \cup \{OperatorEffect\}$
  end for
  return $Effects$
end procedure

Figure 4.2: Pseudocode for computing the effect distributions under an option for a given object. Note that the effect profile for each object is represented as an unordered set of distributions.

Example 4. *Consider a domain illustrated by Figure 4.4 with three objects—two identical doors and a block—and an agent with a single option to open a door. Since the option can affect both of the doors, it would first be partitioned into two subgoal options (one for each door). Given this, the effect profile for the first and second doors would be the sets* $\{\texttt{open1}, \varnothing\}$ *and* $\{\varnothing, \texttt{open2}\}$ *respectively. The effect profile for the block, which cannot be acted upon, would simply be* $\{\varnothing, \varnothing\}$*. If the two distributions* `open1` *and* `open2` *are similar enough (determined using an appropriate similarity metric), then the agent can conclude that the sets representing the two doors' effects are equal. The agent can then merge the two door objects into a single type and replace the* `open1` *and* `open2` *propositions with a single* `open` *predicate parameterised by objects of a new type—which might be called* `door`*.*

procedure MERGE
  Given: objects $\mathcal{M}$, type T, PPDDL operators $Operators$, all propositions $Propositions$
  ▷ Find the first object matching the type
  $archetype \leftarrow \emptyset$
  for each $object \in \mathcal{M}$ **do**
    if ISTYPE$(object, T)$ **then**
      $archetype \leftarrow object$
      break
    end if
  end for
  ▷ Remove propositions with objects of type T that are not the archetype
  $Removed \leftarrow \{prop \mid prop \in Propositions, \text{ISTYPE}(prop, T), \neg\text{REFERSTO}(prop, archetype)\}$
  ▷ Keep operators that do not contain the removed propositions
  $Operators \leftarrow \{op \mid \forall op \in Operators, Removed \cap op = \emptyset\}$
  return $Operators$, $Propositions \setminus Removed$
end procedure

Figure 4.3: Pseudocode for lifting propositions to typed predicates. The procedure takes in an object type and identifies one object instance of that type. Distributions over that object's state are kept and treated as parameterised predicates, while propositions and operators that refer to all other objects of that type are removed.

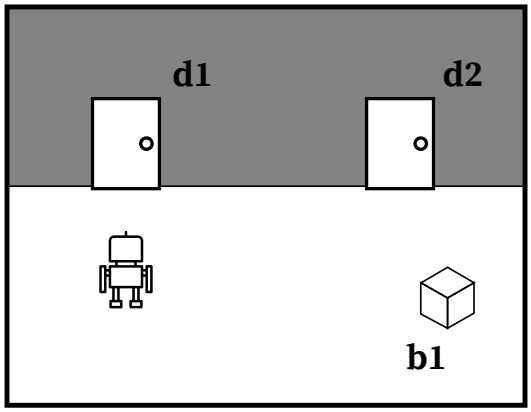


Figure 4.4: A domain with two doors, `d1` and `d2`, and a block `b1`. The agent can interact with the doors, but not the block.

The manner in which we group objects into types is an instance of *effect equivalence*, where entities are divided into equivalence classes based on the effects they undergo [Şahin *et al.* 2007]. Such an approach assumes that an object's type depends on both the intrisic properties of the object itself, as well as the agent's endowed behaviours [Chemero 2003]. However, this definition does not take into account the importance of preconditions, nor does it consider interactions involving multiple objects. Furthermore, although they are critical issues, we do not consider the question of what constitutes an object or complex interactions involving

deformable objects. Nonetheless, the approach will prove sufficient for our purposes (see Section 4.2), and we leave a more complete definition to future work.

4.1.3 Problem-specific instantiation (Step 5)

If the task dynamics are completely described by the state of each object, as is the case in object-oriented MDPs [Diuk *et al.* 2008], then our typed representation is sufficient for planning. However, in many domains the object-centric state space is *not* Markov. For example, in a task where only a particular key opens a specific door, the state of the objects alone is insufficient to describe dynamics—the identities of the key and door are also necessary. As in the previous chapter, we can augment the object-centric observation space with problem-specific, allocentric information to preserve the Markov property. To keep our notation consistent, we denote $\mathcal{D}$ as the object-centric observation space, while $\mathcal{S}$ represents the space of problem-specific state variables.

For a given partioned option, the agent repeats the partitioning procedure, but this time using only problem-specific state data. This forms n partitioned options that are subgoal in both $\mathcal{D}$ *and* $\mathcal{S}$. Denote κ_i and λ_i for $i \in \{1, \ldots, n\}$ as the sets of start and end states for each of these newly partitioned options. The agent now *grounds* the operator by appending each κ_i and λ_i to the precondition and effect, treating each κ_i and λ_i as problem-specific propositions. Finally, these propositions must be linked with the grounded objects being acted upon. The agent therefore adds a precondition predicate conditioned on the identity of the grounded objects (see Figure 4.5).

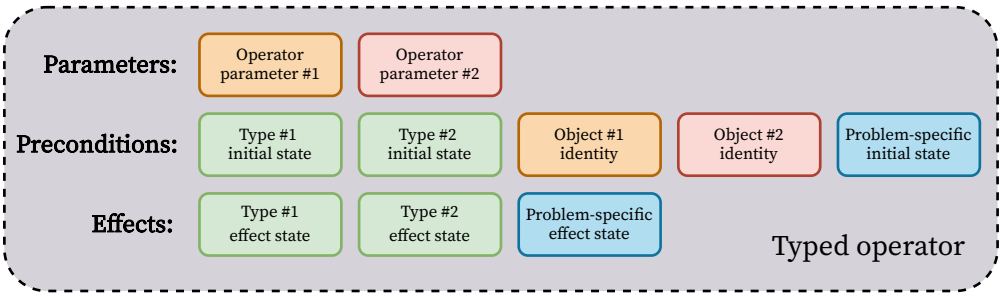


Figure 4.5: An example of a typed operator "template" that accepts two parameters. In order to instantiate the operator in the current task, we must determine the identity of the objects and include them as preconditions (orange and red nodes). We must also include any problem-specific symbols in the precondition and effect, much as we did in Chapter 3. The green nodes represent the portable aspects of the operator that can be transferred to new tasks.

4.2 Experiments

We first demonstrate our framework on the classic Blocks World domain (Section 4.2.1). While the high-level operators and predicates describing the domain are usually given, we show how such a representation can be learned autonomously from scratch. We then demonstrate that our method scales to significantly harder problems by applying it to a high-dimensional Minecraft task (Section 4.2.2).[3] Finally, we investigate the transferability of the learned abstractions by transferring them to additional procedurally generated Minecraft tasks (Section 4.2.3).

4.2.1 Learning a representation of Blocks World

The Blocks World domain consists of a number of blocks which can be stacked on top of one another by an agent (hand). The agent possesses options that allow it to pick up a block (`Pick`), put a block back on the table (`Put`), and stack one block on another (`Stack`). Blocks cannot be picked up if they are covered or if the hand is occupied, and can only be put down or stacked if already gripped. We consider the task consisting of three blocks `A`, `B` and `C`, where each block has attributes describing whether there is nothing, another block, or a table directly above or below it. This representation allows us to determine whether a given block is on a table, on another block, or grasped in the hand, and similarly whether another block has been stacked upon it. The hand is characterised by a single boolean indicating whether it is holding a block. Thus a state is described by $\{\mathbf{f}_H, \mathbf{f}_A, \mathbf{f}_B, \mathbf{f}_C\}$, corresponding to the hand and blocks' features respectively. Note that the agent is initially unaware that the blocks are identical and interchangeable.

Generating a Propositional Model (Steps 1–2) Using the approach outlined in Section 4.1.1, the agent partitions the options using transition data collected from the environment. This results in a total of 15 partitions of the `Pick` option, 3 partitions of the `Put` option, and 12 partitions of the `Stack` option, described by Table 4.1. It then fits a classifier to each partition's initiation states, and a density estimator to its terminating states. Finally, the agent generates a propositional PPDDL using these learned preconditions and effects. Figure 4.6 illustrates a learned propositional operator, while the full PPDDL, learned entirely from data, can be found in Appendix B.2.

Generating a Lifted Typed Model (Steps 3–4) Using the effects from the propositional representation, the agent determines that objects `A`, `B` and `C` all possess the same effect profiles for all options and so can be grouped into a single type, while the hand belongs to its own type. The agent next lifts its representation by replacing the learned propositions with predicates parameterised by the above types. For example, after generating the model, there are three propositions: `AOnTable`,

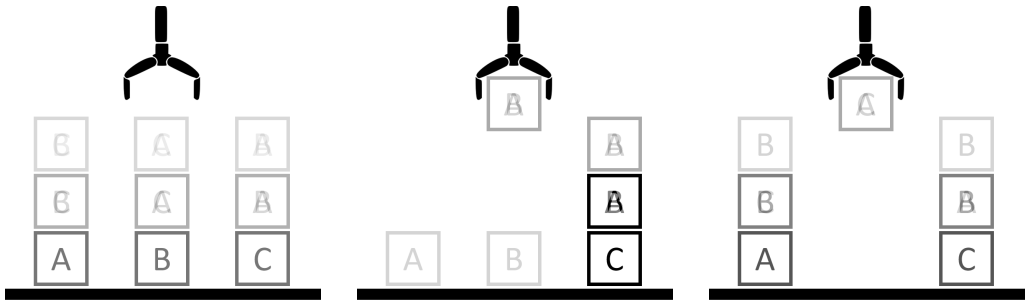
[3] The hyperparameters for the entire procedure are listed in Appendix B.4.

Option	# partitions	Description of start states	Description of end states
`PickOffTable(X)`	3	`X` is on the table, `X` is clear, and the hand is empty.	`X` is grasped in the hand.
`PickOffSingleBlock(X, Y)`	6	`X` is on block `Y` which is on the table, `X` is clear, and the hand is empty.	`X` is grasped in the hand and `Y` is clear and on the table.
`PickOffDoubleBlock(X, Y)`	6	`X` is on block `Y` which is on another block, `X` is clear, and the hand is empty.	`X` is grasped in the hand and `Y` is clear and on another block.
`StackOnSingleBlock(X, Y)`	6	`X` is in the hand, and `Y` is clear and on the table.	`X` is on block `Y` which is on the table, and the hand is empty.
`StackOnDoubleBlock(X, Y)`	6	`X` is in the hand, and `Y` is clear and on another block.	`X` is on block `Y` which is on another block, and the hand is empty.
`Put(X)`	3	`X` is grasped in the hand.	`X` is on the table and the hand is empty.

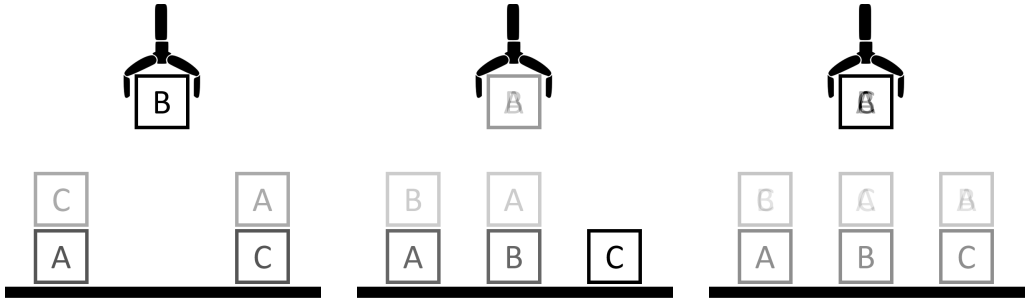
Table 4.1: Descriptions of the different option partitions. The description of start and end states includes only the relevant information.

`BOnTable`, and `COnTable`.[4] Since these are distributions over objects determined to be the same type, the agent replaces them all with a single predicate `OnTable(X)`, which accepts block objects. As a result, the agent reduces the number of operators from 30 to 6, resulting in a more compact representation with a smaller branching factor. Figure 4.7 illustrates how the propositional operator in Figure 4.6 has been lifted to describe picking any block `X` off any block `Y`.

[4]For readability, we have given each symbol a semantically meaningful name. However, they are generated by the agent autonomously and so in practice would have generic names such as `symbol0` and `symbol1`.



(a) `symbol_10` (b) `symbol_15` (c) `symbol_6`



(d) `symbol_3` (e) `symbol_4` (f) `symbol_1`

```
(:action Pick-partition-10
 :parameters()
 :precondition (and (symbol_10) (symbol_15) (symbol_6))
 :effect (and (symbol_3) (symbol_4) (symbol_1)
              (not symbol_6) (not symbol_10) (not symbol_15))
)
```

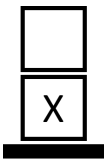
(g) Propositional PPDDL operator for one of the `Pick` option partitions.

Figure 4.6: The learned propositional operator for a `Pick` action describing picking `B` off `C`. To execute the action, the hand must be empty (`symbol_10`), `C` must be on the table and covered by a block (`symbol_15`), and `B` must be on top of a block and uncovered (`symbol_6`). After execution, `B` is in the hand (`symbol_3`), `C` is on the table and clear (`symbol_4`), and the hand is full (`symbol_1`). We visualise each propositional symbol by sampling from it, and randomly sampling the remaining independent state variables (since each symbol is a distribution over a subset of state variables). The transparency is due to the averaging over the independent state variables. Note that we must learn one operator for every pair of blocks.

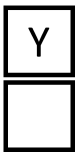
4.2.2 Learning a representation of a Minecraft task

In the above example, objects were represented using pre-specified features and were sufficient to describe the environment dynamics. However, our approach is capable of scaling beyond this simple case and learning these features from pixels. We now demonstrate this in a Minecraft task [Johnson *et al.* 2016] consisting of five rooms with various items positioned throughout. Rooms are connected with either regular doors which can be opened by direct interaction, or puzzle doors which

(a) symbol_4

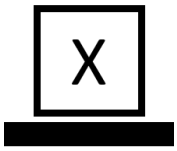


(b) symbol_5



(c) symbol_2

(d) symbol_0



(e) symbol_3

(f) symbol_1

```
(:action Pick-partition-10
 :parameters (?w - type0 ?x - type1 ?y - type1)
 :precondition (and (symbol_4 ?w) (symbol_5 ?x) (symbol_2 ?y))
 :effect (and (symbol_0 ?y) (symbol_3 ?x) (symbol_1 ?w)
         (not (symbol_2 ?y)) (not (symbol_4 ?w)) (not (symbol_5 ?x)))
)
```

(g) PPDDL operator for a `Pick` action.

Figure 4.7: The learned lifted operator for a `Pick` action describing picking a block off another. In order to pick up block `Y`, it must be on block `X` which itself is on the table, and the hand must be empty. As a result, the hand is not empty, `Y` is now in the hand, and `X` is on the table and clear. `type0` refers to the "hand" type, while `type1` refers to the "block" type.

require the agent to pull a lever to open. The world is described by the state of each of the objects (given directly by each object's appearance as a 600×800 RGB image), the agent's view and current inventory. Figure 4.8 illustrates the state of each object in the world at the beginning of one of the tasks.

The agent is provided with the following high-level skills:

(i) `WalkToItem`: the agent will approach an item if it is in the same room.

(ii) `AttackBlock`: the agent will break a block, provided it is near the block and holding the pickaxe.

(iii) `PickupItem`: the agent will collect the item if it is standing in front of it.

(iv) `WalkToNorthDoor`: the agent will approach the northern door in the current room.

(v) `WalkToSouthDoor`: the agent will approach the southern door in the current room.

(vi) `WalkThroughDoor`: the agent will walk through an open door to the next room.

(vii) `CraftItem`: the agent will create a new clock from ingredients in its inventory, provided it is near the crafting table.

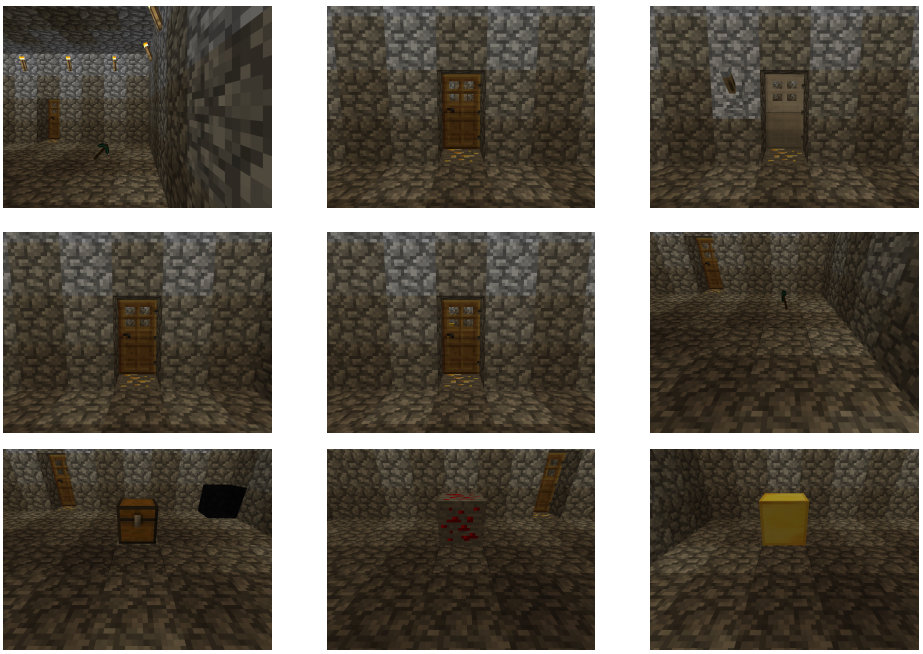


Figure 4.8: The state of each object in the world at the start of the task. From left to right the images represent the agent's point of view, the four doors, the pickaxe, the chest, and the redstone and gold blocks. The inventory is not shown here.

(viii) `OpenChest`: the agent will open the chest, provided it is standing in front of it and possesses the clock.

(ix) `ToggleDoor`: the agent will open or close the door directly in front of it.

Execution is stochastic—opening doors occasionally fails, and the navigation skills are noisy in their execution.

To solve the task, an agent must first collect the pickaxe, use it to break the gold and redstone blocks and collect the resulting items. It must then navigate to the crafting table, where it uses the collected items to first craft gold ingots and subsequently a clock. Finally, it must navigate to the chest and open it to complete the task. This requires a long-horizon, hierarchical plan—the shortest plan that solves the task consists of 29 options consisting of *hundreds* of low-level continuous actions.

Generating a propositional model (Steps 1–2)

To simplify learning, we first apply a series of preprocessing steps to reduce the dimensionality of the state space. Images are first downscaled to 160×120 and then converted to greyscale. Following the methodology of prior work [Konidaris *et al.* 2018], we apply principal component analysis [Pearson 1901] to a batch of

images collected from the different tasks and keep the top 40 principal components.[5] Consequently, each object is represented as a vector of length 40, while the inventory is simply a one-hot encoded vector of length 5.

We collect data by executing options uniformly at random. We record state transition data and the options that could be executed at each state. Options are partitioned using the DBSCAN clustering algorithm [Ester *et al.* 1996] to cluster the terminating states of each into separate effects. For each pair of partitioned options, we check whether there is significant overlap in their initiating states (again using DBSCAN). If the initiating states overlap significantly, the partitions are merged to account for probabilistic effects.

For each of these approximately partitioned options, the agent learns a precondition classifier using an SVM [Cortes and Vapnik 1995] with Platt scaling [Platt 1999]. We use states from the partitioned option as positive examples, and all other states as negative ones. A simple feature selection procedure then determines which objects are relevant to the option's precondition. We first compute the accuracy of the SVM applied to the objects in the option's mask, performing a grid search to find the best hyperparameters for the SVM using 3-fold cross validation. Then, for every other object in the environment, we compute the SVM's accuracy when that object's features are added to the SVM. Any object that increases the SVM accuracy is kept. Pseudocode for this procedure is outlined in Figure 4.9.

Having determined the relevant objects, we fit a probabilistic SVM to these objects' data. Note that we learn a single SVM for a given precondition. Thus if the precondition includes two objects, then the SVM will learn a classifier over both objects' features jointly.

A kernel density estimator (KDE) [Rosenblatt 1956] with Gaussian kernel is used to estimate the effect of each partitioned option. We learn distributions over only the objects affected by the option, learning one KDE for each object. Again, a grid search with 3-fold cross validation is used to find the best bandwidth hyperparameter for each estimator. We fit a single KDE to each object separately, since the state space has already been factored into these objects. Each of these KDEs is an abstract symbol in our propositional PPDDL representation, which is then used to construct the propositional PPDDL representation as previously.

Generating a typed model (Steps 3–4)

Using the effects from the propositional representation, the agent next groups objects into types based on their effect profiles. This is made easier because certain objects do not undergo effects under certain options. For example, the chest cannot be toggled while a door can, and thus it is immediately clear that they are not of the same type. To determine the type of each object, we first assume that they all

[5] Empirical experimentation found that 40 components were sufficient to distinguish between open and closed doors.

```
1: procedure FEATURESELECTION
2:     Given: affected objects Mask, positive start states p, negative start states n,
   set of objects M
3:     ▷ Fit a classifier over only objects in the mask
4:     classifier ← FITCLASSIFIER(start, negative, Mask)
5:     initScore ← classifier.score
6:     Keep ← ∅
7:     for each object ∈ M \ Mask do
8:         classifier ← FITCLASSIFIER(start, negative, Mask ∪ {object})
9:         if classifier.score > initScore then
10:            ▷ Keep the object if it improves the score
11:            Keep ← Keep ∪ {object}
12:        end if
13:    end for
14:    return Mask ∪ Keep
15: end procedure
```

Figure 4.9: Pseudocode for a simple feature selection procedure. To determine which objects should form part of the precondition, we fit a classifier to objects in the option's mask. We then iterate through the remaining objects, adding them to the classifier to determine if they improve the score. Those that result in improved accuracy are kept.

belong to their own type. For each object, we compute its effect profile by extracting the effect propositions that occur under each option (see Figure 4.2).

For every pair of objects, we then determine whether the effect profiles are similar by checking whether the KL-divergence is less than a certain threshold. Having determined the types, we replace all similar propositions with a predicate parameterised by an object of that type.

Problem-specific instantiation (Step 5)

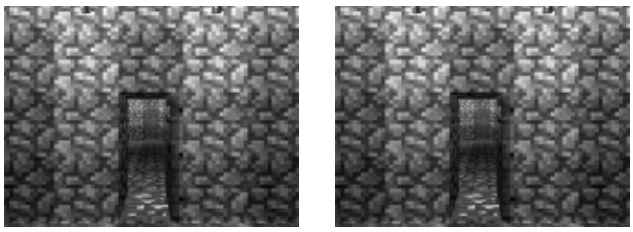
The agent now has a representation whose operators can be transferred between tasks. However, unlike Blocks World, a complication arises because the object-centric state space is *not* Markov. For example, a state where all the doors are closed and the agent is in front of the first door is indistinguishable from a state where the agent is in front of the second door. As described in Section 4.1.3, the agent must ground the representations in the current task by incorporating additional problem-specific state variables to preserve the Markov property. These state variables are fixed across the family of MDPs; in this case, they are the agent's xyz-location.

For each partitioned option, the agent again uses DBSCAN to cluster end states $\mathcal{S}$ to form partitioned subgoal options in both $\mathcal{D}$ and $\mathcal{S}$. Each of these clusters in $\mathcal{S}$ is a problem-specific proposition, which can be added to the learned operators to ground the problem. To ground the operators, we add the start and end clusters (problem-specific propositions) to the precondition and effects of the PPDDL operator. We also record the exact object that appears in the parameter list of each operator, and

add a precondition predicate (fluent) to ensure that only *those* particular objects can be modified. Without this final step, the agent would, for example, believe it can open *any* door while standing in front of a door at a particular location.

Figure 4.10 illustrates a learned operator for opening a particular door, where the problem-specific symbol has been tied to the door being opened in this manner. More learned operators are illustrated in Appendix B.5. The generated PPDDL description is provided as input to the MINI-GPT planner [Bonet and Geffner 2005] which uses a variant of real-time dynamic programming [Bonet and Geffner 2003] to compute an optimal plan. This plan is illustrated by Figure 4.11.

(a) symbol_37 (b) symbol_9 (c) psymbol_24



(d) symbol_64 (e) symbol_65

```
(:action Toggle-Door-partition-1a
:parameters (?w - type0 ?x - type1)
:precondition (and (notfailed) (symbol_37 ?w) (symbol_9 ?x)
                   (= (id ?x) 1) (psymbol_24))
:effect (and (symbol_64 ?x) (symbol_65 ?w) (not (symbol_9 ?x))
             (not (symbol_37 ?w)))
)
```

(f) A learned typed PPDDL operator for one partition of the Toggle-Door option. The predicates underlined in red must be relearned for each new task, while the rest of the operator can be transferred.

Figure 4.10: Our approach learns that, in order to open a particular door, the agent must be standing in front of a closed door (symbol_37) at a particular location (psymbol_24), and the door must be closed (symbol_9). The effect of the skill is that the agent finds itself in front of an open door (symbol_64) and that the door is open (symbol_65). type0 and type1 refer to the "agent" and "door" classes, while id is a fluent specifying the identity of the grounded door object, and is linked to the problem-specific symbol underlined in red.

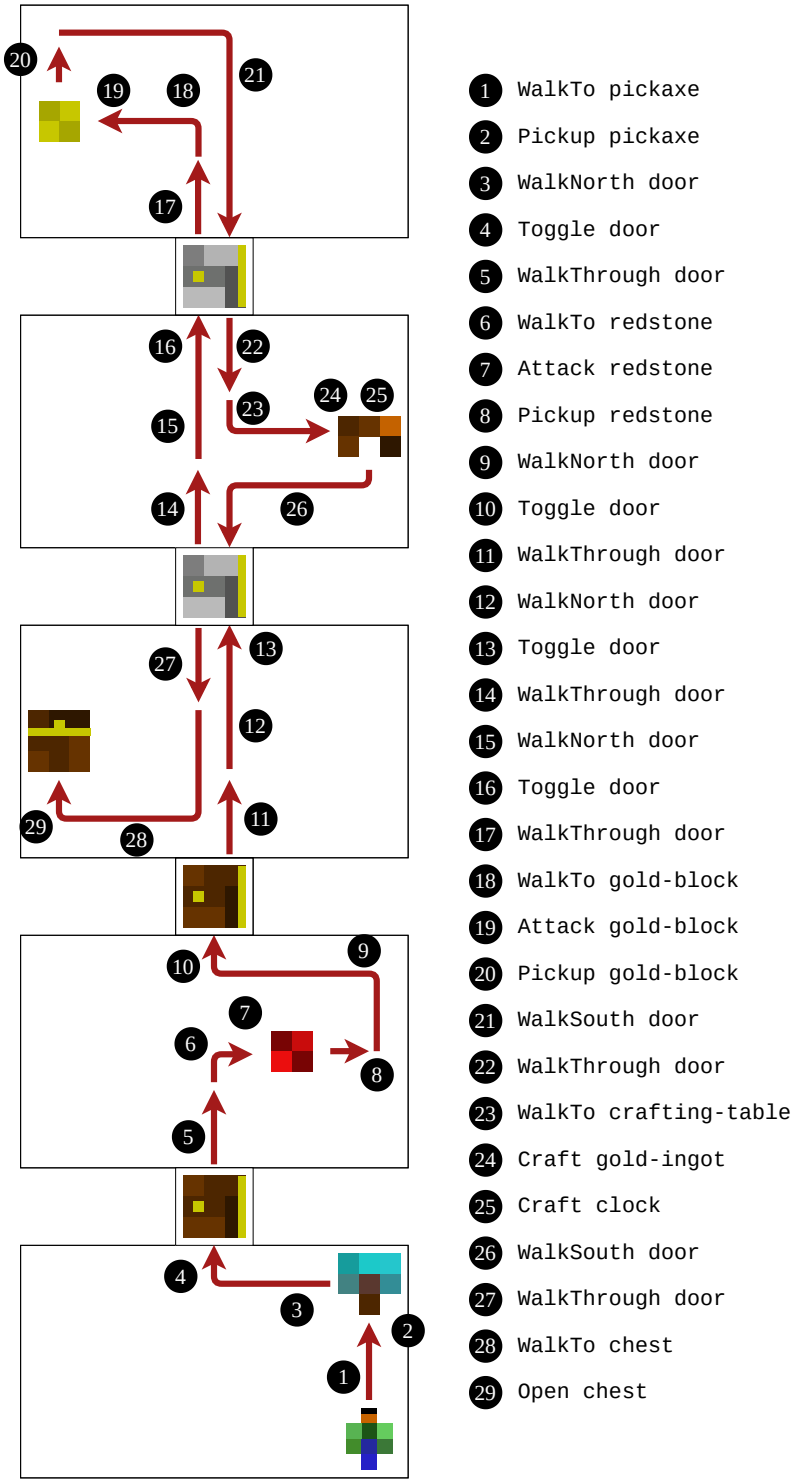


Figure 4.11: Path traced by the agent executing different options while solving the first task.

4.2.3 Inter-task Transfer in Minecraft

We next investigate transferring operators between five procedurally generated tasks, where each task differs in the location of the objects and doors—the agent cannot thus simply use a plan found in one task to solve another. For a given task, the agent transfers all operators learned from previous tasks, and continues to collect samples using uniform random exploration until it produces a model which predicts that the optimal plan can be executed. Figure 4.12 shows the number of operators transferred between tasks, while Figure 4.13 and the number of samples required to learn a model of a new task.

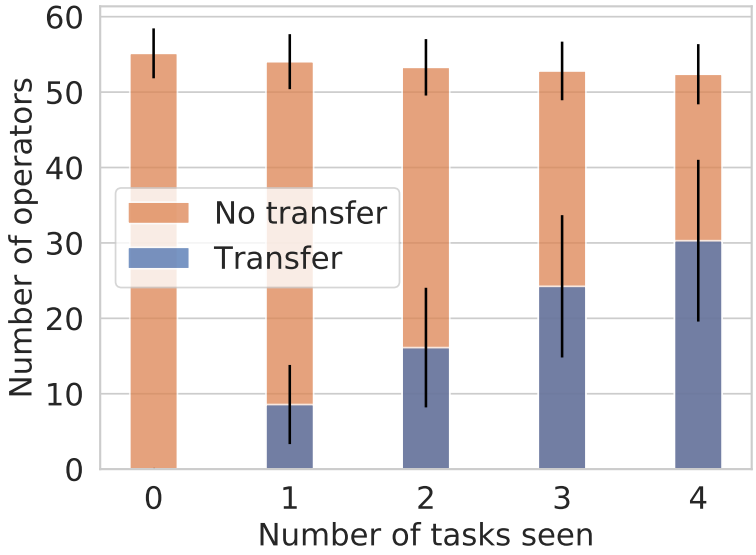


Figure 4.12: Orange bars represent the number of operators that must be learned to produce a sufficiently accurate model to solve the task. Blue bars represent the number of operators transferred between tasks. As the number of tasks increases, the number of new operators that must be learned decreases. By contrast, the number of operators required to learn when no transfer occurs is constant. We report the mean and standard deviation averaged over 80 runs with random task orderings.

The lower bound on sample complexity depends on the exploration strategy, since we must discover all problem-specific symbols to complete the model. Figure 4.13 shows that the number of samples required to learn a model decreases over time towards this lower bound. Inter-task transfer could be further improved by leveraging the agent's existing knowledge to perform non-uniform exploration, but we leave this to future work.

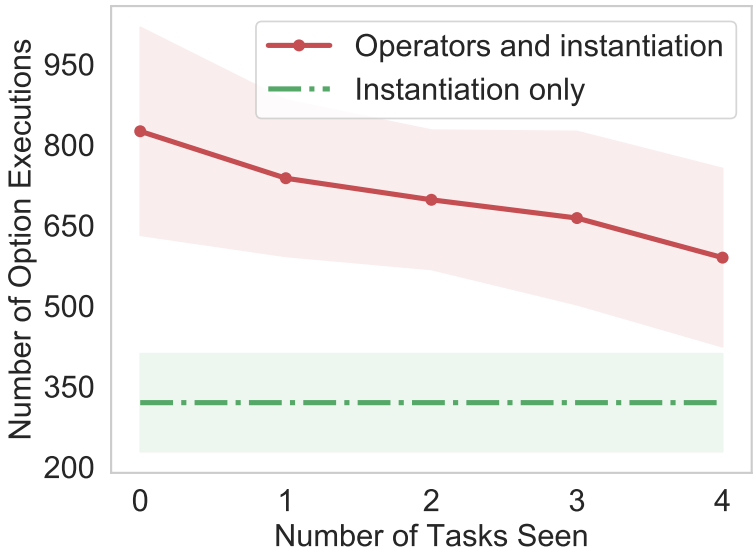


Figure 4.13: Number of samples required to learn sufficiently accurate models as a function of the number of tasks encountered. The red line represents the number of samples required to learn all the operators and the instantiation, while the green line accounts for the instantiation phase only. This line is the lower bound on the sample complexity, and refers to the number of options that must be executed to discover the necessary problem-specific propositions. We report the mean and standard deviation averaged over 80 runs with random task orderings.

4.3 Discussion of failure cases

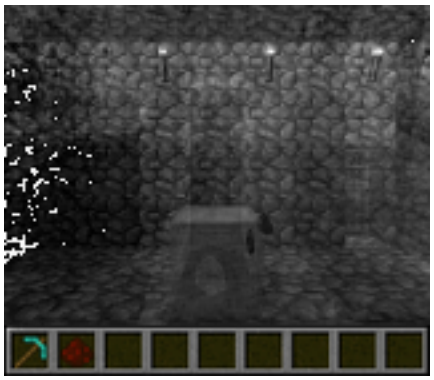
Although our approach is able to learn a model of a complex Minecraft task, it relies on several phases involving clustering, classification and density estimation. Owing to the complexity of each of these steps, it is unsurprising to note that we observe various learning errors throughout. These errors could be caused by a variety of factors, such as insufficient data or suboptimal hyperparameters. We highlight some of the most common ones below to serve as a guide for where future improvements to the underlying estimation procedures can be made.

4.3.1 Partitioning errors

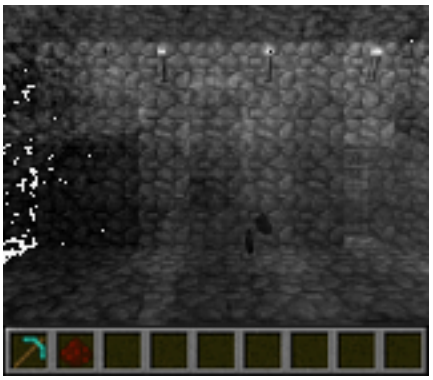
Since the weak subgoal property is intractable to compute precisely, we rely on an approximation that uses the DBSCAN clustering algorithm. DBSCAN requires a single hyperparameter that determines radius of a neighbourhood with respect to some point. However, this hyperparameter is extremely sensitive and depends on the exact task. In Figures 4.14 and 4.15, we illustrate an example of imperfect partitioning which results in redundant partitioned options.

(a) Set of start states for the first partition of the `Attack` option, with the agent standing in front of a gold block.

(b) Set of end states for the first partition of the `Attack` option, with the agent standing in front of a smashed block.



(c) Set of start states for the second partition of the `Attack` option, with the agent standing in front of a gold block.

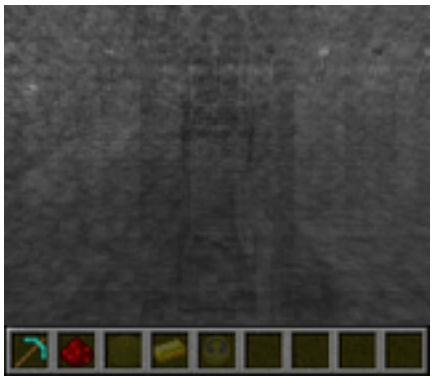


(d) Set of end states for the second partition of the `Attack` option, with the agent standing in front of a smashed block.

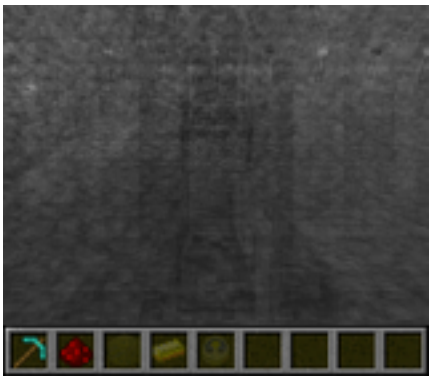
Figure 4.14: In the above example, the partitioning procedure has generated two partitioned options for breaking the gold block, where there should only be one. They are functionally equivalent, but because of the strange shadows on the left of the image (caused by rendering bugs in the Malmo platform), the clustering algorithm has produced one extra partition. As a result, an additional unnecessary operator and predicate would be created. This issue is due to the approximation of the subgoal property, and could be overcome with a more robust clustering approach.

(a) Set of start states for the first partition of the `ToggleDoor` option, where the agent is standing in front of an open door.

(b) Set of end states for the first partition of the `ToggleDoor` option, where the agent is standing in front of a closed door.



(c) Set of start states for the second partition of the `ToggleDoor` option.

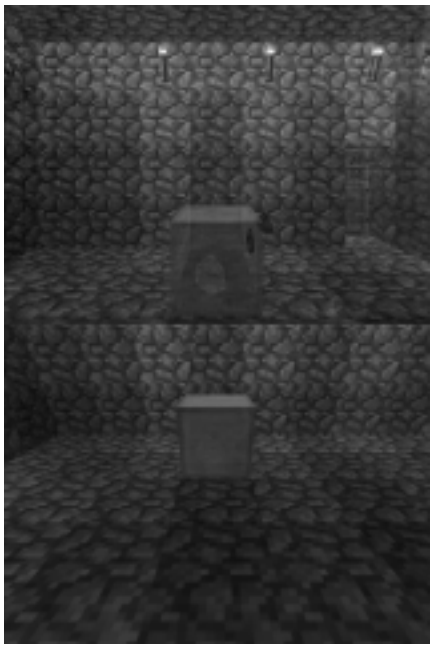


(d) Set of end states for the second partition of the `ToggleDoor` option.

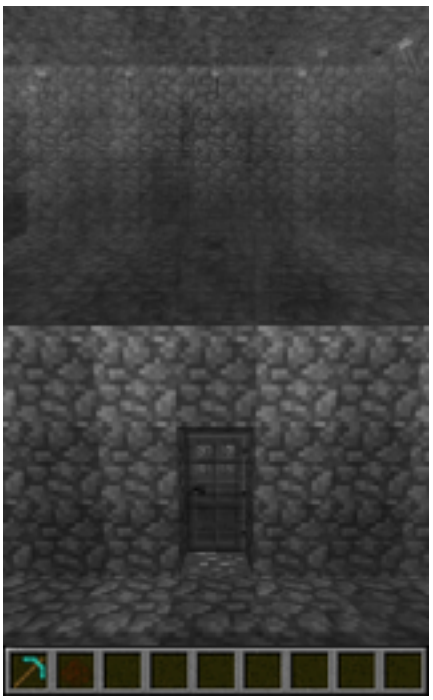
Figure 4.15: In this example, the partitioning has clustered noisy samples into an additional partition of the `ToggleDoor` option. While the top row shows the case where the state of the door changes from open to closed, the bottom row is a relatively useless noisy operator. We will subsequently learn a precondition and effect for this partition, but it likely will not be used by the planner. This error is due to noisy data and could be corrected by modifying the clustering algorithm's hyperparameters to ignore clusters whose sizes are below a certain threshold.

4.3.2 Precondition errors

When estimating the preconditions, we find that applying a grid-search to the hyperparameters of an SVM results in a fairly robust classifier. However, determining which objects should belong to an operator's precondition is more challenging. In Figure 4.16, we illustrate two failure cases: the first case excludes a necessary object, while the second includes an unnecessary one.



(a) The precondition for attacking the gold block. The top image represents the agent's view (in front of the block) while the bottom image is the state of the block (unbroken).

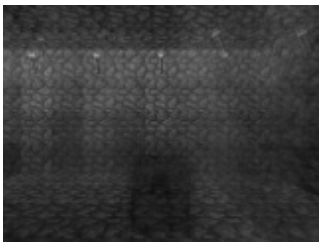


(b) The precondition for walking to a closed door. The top image represents the agent's view (in a room) while the bottom image is the state of the door (closed) and the state of the inventory.

Figure 4.16: In the left example, the classifier predicts that the gold block can be broken when the agent is in front of it. However, this is not quite correct, since the agent must also have the pickaxe to break the block. In this case, this issue occurs because the data only included states where the agent reached the gold block with the pickaxe. The agent therefore did not observe states where it was in front of the block without the pickaxe, and thus concludes that the pickaxe is irrelevant to the precondition. In the right example, the classifier overfits to the data and predicts that the agent can only walk to the door when it has the pickaxe. This issue can be overcome with more data, provided the data includes samples where the agent finds itself in front of the block without the pickaxe.

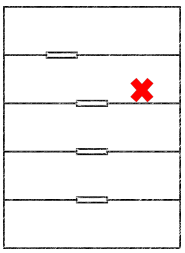
4.3.3 PPDDL construction errors

The quality of the PPDDL operators depends on how accurately the precondition classifiers and effect estimators are learned. An error can result in imperfect PPDDL operators, as seen in Figure 4.17.



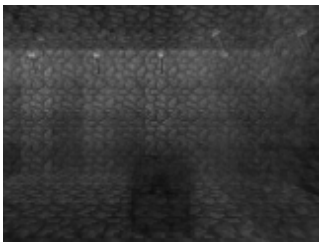
(a) symbol_56

(b) symbol_29



(c) psymbol_1

(d) symbol_57



(e) symbol_11

```
(:action Craft-partition-1-240a
 :parameters (?w - type0 ?x - type9)
 :precondition (and (notfailed) (symbol_56 ?w) (symbol_29 ?x)
                    (psymbol_1))
 :effect (probabilistic 0.21 (not (notfailed))
                        0.79 (and (symbol_57 ?x) (symbol_11 ?w)
                                  (not (symbol_29 ?x))
                                  (not (symbol_56 ?w))))
)
```

(f) Typed PPDDL operator for a partition of the Craft option.

Figure 4.17: Abstract operator that models the agent crafting a gold ingot. In order to do so, the agent must be standing in front of the crafting table (symbol_56) at a particular location (psymbol_1), and must have the gold block in its inventory (symbol_29). As a result, the agent finds itself in front of the crafting table (symbol_11), and now has a gold ingot in its inventory (symbol_57). This option is deterministic; however, due to estimation errors, the PPDDL operator predicts that it will only succeed with probability 0.79. This issue is caused by the precondition classifier's failure to generalise to samples drawn from the predicates. Solutions include deploying more sophisticated models such as neural networks, or creating a more robust classfier using methods such as data augmentation [Wong *et al.* 2016].

4.3.4 Type Inference Error

Finally, we observe that occasionally the procedure will not discover the correct types. In Table 4.2, instead of discovering a single type for all four doors, our approach predicts that one door is different from the others. This occurs because the predicate describing the door's effect is considered to be dissimilar to those of the other doors, despite being semantically identical.

Type	Name	Object(s)
0	Agent	0
1	Pickaxe	1
2	Door1	2, 3, 4
3	Door2	5
4	Redstone Block	6
5	Gold Block	7
6	Chest	8
7	Inventory	9

Table 4.2: A grouping of objects into types. Note that one of the doors is allocated its own type. This issue is due to the measure used to determine whether two effects are similar. In future, we may be able to incorporate additional knowledge, such as the options available before and after interacting with an object, to improve the type inference step.

4.4 Related work

Existing approaches have long assumed that the world consists of objects, and that similar objects are shared across tasks. The classical planning literature, for example, represents problems in terms of the objects that constitute a domain, and operators that can affect their states [McDermott *et al.* 1998]. Another approach is that of object-oriented MDPs, which exploit the presence of objects by providing the agent with an object-oriented representation. This results in compact representations that are transferable between tasks sharing the same object classes and dynamics [Guestrin *et al.* 2003; Diuk *et al.* 2008; Marom and Rosman 2018]. Here the state space consists of a set of object classes $\mathcal{C}$, where each class $C \in \mathcal{C}$ has its own set of attributes $\text{Att}(C) = \{C.\alpha_1, \ldots C.\alpha_n\}$. A particular task consists of a grounded set of objects, where each object belongs to one of the classes in $\mathcal{C}$. The state of a single object is the value assignment to all of its attributes, and the state of the task is the union of all object states.

In general, the state space is provided to the agent, which must then learn the transition dynamics of the environment. These dynamics are represented in terms of the effects an agent's action has on the different objects, with additive, subtractive, and setting effects commonly considered [Diuk *et al.* 2008]. Marom and Rosman [2018] extend this model-learning approach to learn dynamics with respect to each object in turn, resulting in a lifted representation that improves the transfer.

In the above settings, however, the question arises as to the most appropriate way of building an object-oriented representation of a problem, especially one experienced by the agent at the pixel level. This includes deciding which attributes

should be chosen to characterise a particular type, as well as which objects should belong to each class or type.

There has been work autonomously learning parameterised, transferable representations of skills from raw data. Ugur and Piater [2015a] learn object-centric PPDDL representations for robotic object manipulation tasks. As in our work, they estimate object types by clustering objects based on how actions affect their states, but the object features are specified prior to learning, and discrete relations between object properties such as width and height are given. Moreover, additional predicates are manually inserted to generate a sound representation. Asai [2019] learns object-centric abstractions directly from pixels, but it is unclear how to extend the approach to the stochastic setting. Furthermore, the representations cannot be transformed into a language that can be used by existing task-level planners. By contrast, we show how to learn an object-centric PPDDL representation along with the object types, and the abstract high-level dynamics model directly from raw data.

4.5 Conclusion

We have shown a method for learning high-level, object-centric representations that are sound and useful for planning. In particular, we have demonstrated how to learn the type system, predicates and high-level operators all from pixel data. Our representation generalises across objects and can be transferred to new tasks.

Our definition of types considered the effects of objects in isolation. This is a similar setting to object-oriented MDPs [Diuk *et al.* 2008], where the dynamics are described by pairwise object interactions (and one of those objects is the agent). While this proved sufficient for the cases considered here, the definition would be inadequate in certain situations involving multi-object interactions. For example, consider two objects x and y that undergo the same effects in isolation, but which can also interact with a third object z. If the effect of an option on z is dependent on whether it was executed in conjunction with x or y, then x and y would need to be considered as different, but related, types. We leave the task of developing a theoretically sound definition of types, as well as the important question of what constitutes an object, to future work.

Although we have injected structure by assuming the existence of objects, this reflects the nature of many environments: fields such as computer vision assume that the world consists of objects, and there is evidence to suggest that infants do the same [Spelke 1990]. This assumption allows us to convert complex, high-dimensional environments to abstract representations that serve as input to task-level planners. Our approach provides an avenue for solving sparse-reward, long-term planning problems, such as the MineRL competition [Guss *et al.* 2019], which are currently beyond the reach of state-of-the-art approaches.

Chapter 5

Portable Hierarchies

Previous chapters demonstrated how to learn an abstract representation using agent- and object-centric observations. These approaches can be thought of as constructing a two-level hierarchy—the bottom level being the original MDP and the top level the symbolic classical planning domain (CPD).

It has long been understood that hierarchies of abstractions can be used to simplify decision making and accelerate planning [Sacerdoti 1974]. One such example is the hierarchical task network (HTN) framework, where planning is performed using a hierarchy of tasks and subtasks [Tate 1977; Ghallab *et al.* 2004]. Decomposing tasks into subtasks requires a large amount of domain-specific knowledge, but as a result, these planners can solve problems consisting of thousands of objects and action operators, and have seen wide deployment in the real world [Wilkins and desJardines 2001]. Given the espoused advantages of hierarchical planning, the next logical step is to wonder whether there is any advantage to learning a *multi-level abstraction hierarchy* within our framework.

In this chapter, we propose a method for constructing a hierarchy of increasingly abstract representations. As with our previous approaches, these representations can be reused in new tasks that share similar structure. We first describe our approach in Section 5.1 by manually constructing an agent-centric hierarchy in a gridworld domain. We then demonstrate how it can be learned autonomously by interleaving action and state abstraction (Section 5.2). In Section 5.3 we apply our method to the high-dimensional continuous video game from Chapter 3, where our results demonstrate that the learned multi-level hierarchy improves planning, while its portability improves the agent's planning efficiency when faced with a new task.

5.1 Constructing a portable hierarchy

Recall from previous chapters that the agent begins in a continuous, low-level MDP. After acquiring skills, it constructs an abstract representation of the task used for planning. Konidaris [2016] proposes an approach that simply extends this by acquiring new skills in this abstract domain, and then learning a new abstraction

using these higher-order skills. This interleaving of skill and symbol acquisition—a *skill-symbol loop*—is illustrated by Figure 5.1.

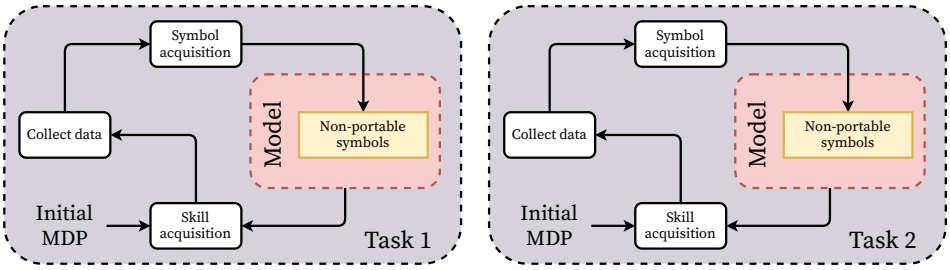


Figure 5.1: Illustration of the *skill-symbol loop* [Konidaris 2016]. Starting with a continuous MDP, an agent acquires a set of skills and then constructs a symbolic representation as described previously (see Chapter 2). Given this new abstract domain, we can simply acquire new skills and then repeat the procedure to learn an increasingly abstract representation. This can be iterated to construct an entire hierarchy of state and action abstractions. However, this must be done for each task independently.

While this hierarchy can be used to plan at various levels of abstraction, there is once again no opportunity for transfer—given a new task, an agent must construct a new hierarchy from scratch. We now outline our approach to constructing such a hierarchy, parts of which can be transferred to new tasks that share similar structure.

5.1.1 Abstract state semantics

In previous chapters, the predicates in our learned symbolic vocabulary represent distributions over low-level states in the original MDP. The obvious question, then, is: *what do higher-order predicates represent?* We represent an n-level hierarchy as a collection of increasingly abstract domain representations

$$\{\mathcal{M}_0, \tilde{\mathcal{M}}_1, \ldots, \tilde{\mathcal{M}}_{n-1}\},$$

where $\mathcal{M}_0$ is the original low-level MDP, and each $\tilde{\mathcal{M}}_{i>0}$ is a PPDDL domain description.

In the agent-space formulation of Chapter 3, the low-level MDP is represented by $\mathcal{M}_0 = \langle \mathcal{S}_0, \mathcal{D}_0, \mathcal{A}_0, \mathcal{T}_0, \mathcal{R}_0 \rangle$.[1] $\mathcal{S}_0$ and $\mathcal{D}_0$ represent the problem-specific and agent-centric state variables, while the transition dynamics and reward function are denoted by $\mathcal{T}_0$ and $\mathcal{R}_0$. After applying our portable symbol-learning method, the resulting CPD is given by $\tilde{\mathcal{M}}_1 = \langle \tilde{\mathcal{S}}_1, \tilde{\mathcal{D}}_1, \tilde{\mathcal{A}}_1, \tilde{\gamma}_1 \rangle$. Here $\tilde{\mathcal{A}}_1$ is the set of action operators, $\tilde{\mathcal{S}}_1$ and $\tilde{\mathcal{D}}_1$ are problem-specific and agent-centric predicates representing distributions over ground states, and $\tilde{\gamma}_1$ is the transition function, modelled by the learned PPDDL operators.

[1] Although we adopt the agent-centric representation in this chapter, it readily extends to the object-centric approach in Chapter 4.

By treating $\tilde{\mathcal{A}}_1$ as low-level actions in $\tilde{\mathcal{M}}_1$, we can acquire new, higher-level options and then repeat the process to construct increasingly abstract representations [Konidaris 2016]. Importantly, because every abstract state space is constructed from a set of discrete option effects, we have that every $\tilde{\mathcal{M}}_i$ at level $i > 0$ is necessarily discrete, even if $\mathcal{M}_0$ itself is continuous. As such, every abstract state at level $i > 1$ is a *categorical distribution* over states at level $i - 1$.

Definition 7. *An abstract state space at level $i > 0$ is the tuple $\langle \tilde{\mathcal{S}}_i, \tilde{\mathcal{D}}_i \rangle$, where each $s \in \tilde{\mathcal{S}}_i$ and $d \in \tilde{\mathcal{D}}_i$ is a distribution over states in $\tilde{\mathcal{S}}_{i-1}$ and $\tilde{\mathcal{D}}_{i-1}$ respectively.*

Given an abstract state at level i, we can compute the distribution over ground agent-space and problem-space states it represents, $\mathcal{G}$, by recursively computing the distribution over states at level $i-1, i-2, \ldots 0$. This computation is illustrated by Figure 5.2 and will prove useful when determining the probability that an agent has reached the goal, given its current abstract state.

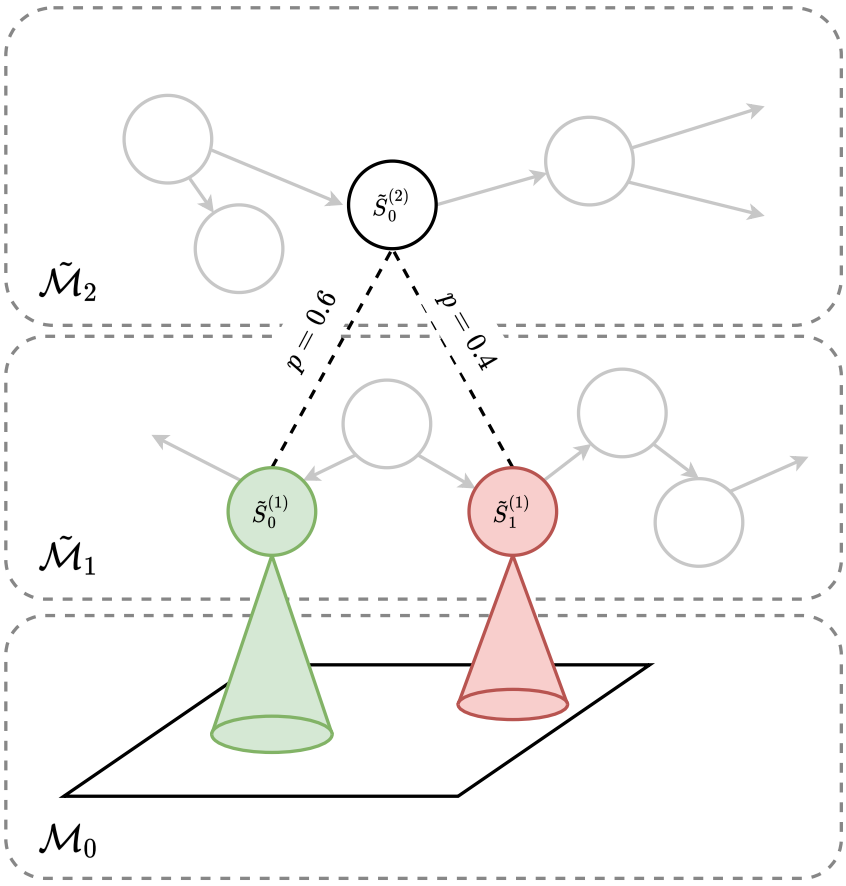


Figure 5.2: A three-level hierarchy where state $\tilde{S}_0^{(2)}$ is a categorical distribution over states $\tilde{S}_0^{(1)}$ and $\tilde{S}_1^{(1)}$ at level 1, which in turn are distributions over low-level states in the original MDP $\mathcal{M}_0$. Let $\mathcal{G}(s), s \in \tilde{\mathcal{M}}_1$ be the distribution over states in the original MDP. We can then calculate the low-level states that $\tilde{S}_0^{(2)}$ represents as $0.6 \times \mathcal{G}(\tilde{S}_0^{(1)}) + 0.4 \times \mathcal{G}(\tilde{S}_0^{(1)})$.

The above formulation means that, as we construct more levels in the hierarchy,

the resulting representations become increasingly compact and faster to plan with. This may also complicate planning, however, since it is unclear how best to leverage the hierarchy [Konidaris 2016; Gopalan *et al.* 2017]. For example, consider an agent required to reach a ground state in the original MDP. Then, as is evident in Figure 5.2, planning at a high level introduces uncertainty as to whether the plan produced does in fact satisfy the goal. Conversely, planning at too low a level negates the advantages of the hierarchy to begin with.

Finally, note that since each $\mathcal{D}_i$ in our abstract hierarchy is agent-centric, it provides an avenue for transfer between tasks that share similar local observations. By contrast, because each $\mathcal{S}_i$ is constructed from problem-specific state variables, it must be relearned from scratch for every new task encountered. Similarly, when operators consist of both portable and non-portable predicates, much of the operator can still be transferred; only the problem-specific predicates must be relearned in the new task.

5.1.2 Constructing a hierarchy

We illustrate our approach by hand-constructing a portable hierarchy on a simple domain, and then show how it can be used to plan at various levels of abstraction. We consider an extension of the Four Rooms domain [Sutton *et al.* 1999] where an agent must navigate a building with four floors. Each floor consists of four rooms separated by doorways, and each room contains a single staircase that leads to higher and lower levels (see Figure 5.3). The agent space consists of a local agent-centric view (such as RGB observations from an egocentric camera), while the problem space is the agent's xyz-coordinates. The agent possesses 6 actions: four that allow it to move in any cardinal direction, and two for climbing up and down staircases. The final two actions are only executable when the agent is near a staircase—executing them leaves the agent in the centre of the room directly above or below its initial location. At every timestep, the agent receives a reward of -1.

Level 1 We assume that the agent possesses the following seven options:

(i) `WalkToDoorway (x2)`: the agent will walk from a room to each of the two doorways in a clockwise and anticlockwise direction.

(ii) `WalkToRoom (x2)`: the agent will walk from a doorway to each of the two adjacent rooms in a clockwise and anticlockwise direction.

(iii) `WalkToStaircase`: the agent will walk from the centre of a room to the room's staircase.

(iv) `ClimbStaircase`: the agent will climb a staircase and find itself in the centre of the room directly above.

(v) `DescendStaircase`: the agent will descend a staircase and find itself in the centre of the room directly below.

After constructing a PPDDL representation using the approach in Chapter 3, our first abstract representation consists of the following five portable symbols:

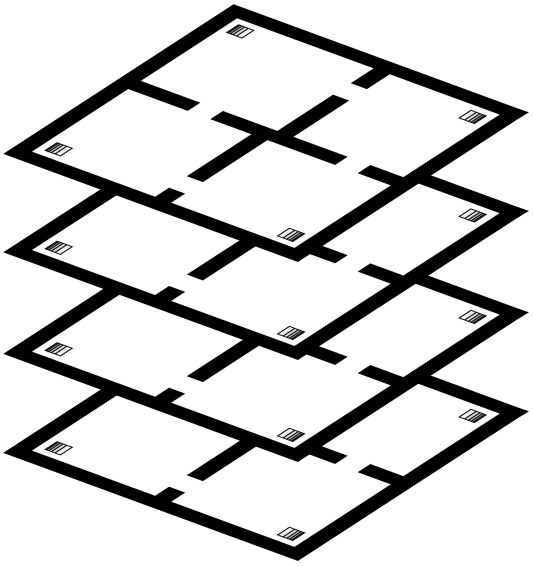


Figure 5.3: Illustration of the extended Four Rooms domain. Each floor consists of four rooms with doorways connecting adjacent rooms. Additionally, each room has a staircase that leads to higher or lower levels, visualised as a box in the corner of each room.

(i) `InRoom`, where the agent is in the centre of the room; (ii) `AtGroundStairs`, where the agent is at a staircase on the ground floor; (iii) `AtTopStairs`, where the agent is at a staircase on the top floor; (iv) `AtStairs`, where the agent is at a staircase on either of the middle floors; and (v) `InDoorway`, where the agent is in a doorway. There are also 48 problem-space propositions referring to distributions over various locations in the domain (12 for each floor). Figure 5.4 illustrates this abstract representation.

Level 2 Next, the agent acquires an option to navigate between adjacent rooms directly. For illustrative purposes, this is a stochastic option: with probability 0.7, the agent finds itself in the centre of the room, and with probability 0.3 at the staircase inside the room. The agent also acquires two additional options to climb up and down a level, even if it is not initially near a staircase. Once again, these options terminate in the centre of the room with probability 0.7, and at the room's staircase with probability 0.3.

As a result, the new representation consists of three agent-space predicates:

(i) `InBottomRoom`, a categorical distribution over the symbols `AtGroundStairs` and `InRoom`;

(ii) `InMiddleRoom`, a distribution over `InRoom` and `AtStairs`; and

(iii) `InUpperRoom`, a distribution over `InRoom` and `AtTopStairs`.

There are 16 problem-space propositions, corresponding to the location of each of the rooms on each of the four floors. Figure 5.5 illustrates the abstract representation at this level.

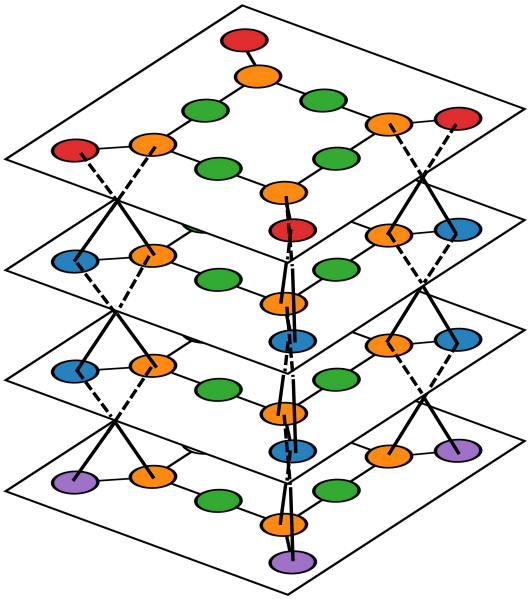


Figure 5.4: The abstract representation at level 1 of the hierarchy. Each node represents a state made up of a single agent- and problem-specific predicate. Colours indicate different agent-centric predicates, while a node's position indicates its problem-specific proposition. The `InRoom` symbol is represented by orange nodes, `AtGroundStairs` purple, `AtTopStairs` red, `AtStairs` blue, and `InDoorway` green. Edges represent options that connect neighbouring nodes.

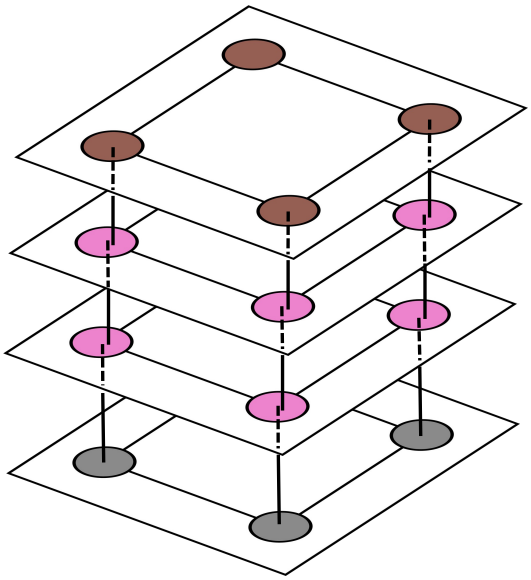


Figure 5.5: The abstract representation at level 2 of the hierarchy. Each node represents a state, with its colour denoting the agent-centric predicate and its position indicating the problem-specific proposition. The `InBottomRoom` symbol is represented by grey nodes, `InMiddleRoom` pink nodes, and `InUpperRoom` brown nodes. Edges represent options that connect neighbouring nodes.

Level 3 Finally, the agent acquires a single option that allows it to climb up or down floors, regardless of its current location. As a result, we have the same portable predicates as level 2, along with four problem-space propositions, where each proposition represents a unique floor. Figure 5.6 illustrates this final abstract representation.

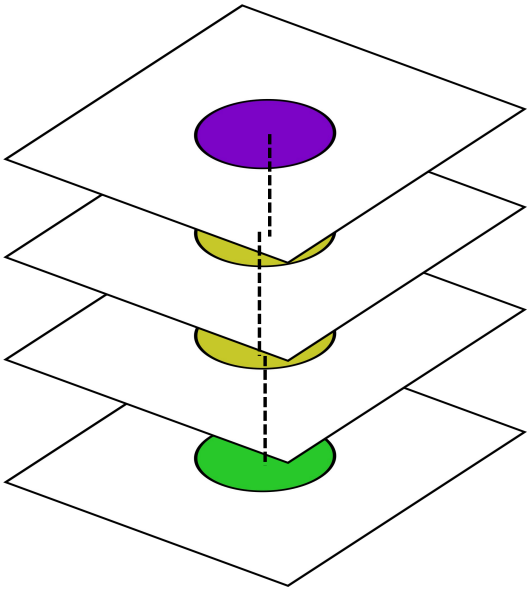


Figure 5.6: The abstract representation at level 3 of the hierarchy. Each node represents a state, with its colour denoting the agent-centric predicate and its position indicating the problem-specific proposition. The `InBottomRoom` symbol is represented by the green node, `InMiddleRoom` yellow, and `InUpperRoom` purple. Edges represent options that connect neighbouring nodes.

5.1.3 Planning with a hand-constructed hierarchy

Having constructed this 3-level hierarchy, we now examine how it can be used to plan at a variety of levels. Intuitively, using higher-order levels results in faster planning, but there may be greater uncertainty as to whether the plan is in fact a solution to the current problem. We consider a *plan query* to be the tuple $\langle B, G \rangle$, representing the low-level set of starting and goal states respectively. Note that these conditions may be satisfied by more than one abstract state or predicate. We investigate four possible queries, and then show the implications of planning at each of the levels in the hierarchy.

Query 1 This query requires the agent to compute a plan that starts in a particular doorway on the bottom floor, and terminates in the equivalent doorway on the top floor. Planning at level 1 is straightforward, since the agent can simply use its options to traverse all floors to reach the top and then navigate to the doorway. However, planning at levels 2 and 3 is not possible, since the predicates used at these levels do not refer to ground states where the agent is in the doorway. For example, the low-level states that `InBottomRoom` refers to are disjoint from the states

referred to by `InDoorway`: $\text{support}(\mathcal{G}(\texttt{InBottomRoom})) \cap \texttt{InDoorway} = \varnothing$, where $\text{support}(\mathcal{G}(\tilde{\mathcal{S}})) = \{s \in \mathcal{S} \mid f_{\mathcal{G}(\tilde{S})}(s) > 0\}$ and $f_{\mathcal{G}(\tilde{S})}$ is the probability density function defined by $\mathcal{G}(\tilde{\mathcal{S}})$.

Query 2 This query requires the agent to compute a plan that starts at a particular staircase on the bottom floor, and terminates at the equivalent staircase on the top floor. Once more, planning at level 1 is straightforward, since the agent can simply use its options to traverse all floors to reach the top and then navigate to the correct staircase. Planning at level 2 is also possible, but introduces uncertainty. Recall that `InUpperRoom` is a categorical distribution over lower-level abstractions. Therefore, while the agent need only execute four level 2 options to reach `InUpperRoom`, the probability that the agent has attained the goal is only 0.3. There is thus a tradeoff between planning at a higher level—the solution length is shorter, but we cannot guarantee the success of the plan. We can also use level 3 where planning is trivial, but this introduces even more uncertainty—the probability that the agent is in the correct room is only 0.25, and so the overall probability of reaching the goal is 0.075.

Query 3 This query requires the agent to compute a plan that starts in a particular room on the bottom floor, and terminates somewhere in the equivalent room on the top floor. Planning at level 1 is the same as before, but now the use of level 2 abstractions introduces no uncertainty, because reaching `InUpperRoom` is a valid solution. Planning at level 3 still introduces uncertainty, although not as much as previously, since the probability that the agent is in the correct room is 0.25.

Query 4 The final query requires the agent to reach the top floor, starting on the bottom floor. Now no uncertainty is introduced when planning at any of the three levels of the hierarchy— planning at levels 1 and 2 is the same as before, while planning at level 3 is also certain, since the goal condition is independent of the exact room location.

The above suggests that there are advantages to planning using all levels of the hierarchy. Although higher levels introduce more uncertainty, they are also more efficient to plan with. A naïve approach is to first identify the highest level at which planning should take place. As we observed in **Query 1**, planning at a particular level is only feasible if there are abstract states that match the start and goal conditions. Having constructed a plan to reach the goal at this level, the agent can then use the level directly below to refine the solution. This can be repeated until the probability of a plan's success exceeds some threshold. In general, though, how best to leverage the hierarchy and tradeoffs associated with its various levels remains an open question that we leave to future work.

5.2 Learning a portable hierarchy

In the above example as well as in prior work [Konidaris 2016], the higher-order options were manually constructed to produce a useful and minimal hierarchy. In

this section, we propose a method for autonomously constructing the entire hierarchy given only the initial set of options. Before outlining our approach, we first discuss some basic concepts used in the hierarchical planning literature.

5.2.1 Hierarchical planning

Throughout this thesis, we have concerned ourselves with classical planning, the aim of which is to construct goal-achieving plans. An alternate approach is known as Hierarchical Task Network (HTN) planning [Ghallab *et al.* 2004], which inherits many of the same concepts as classical planning such as symbolic predicates and action operators. However, rather than achieving a goal, the aim of HTN planners is to decide how best to perform a set of *tasks*. For simplicity, we consider a subset of HTN planning known as Simple Task Network (STN) planning.

In addition to the state predicates and action operators, STN domains consist of a set of *tasks* $t_i(p_1, \ldots, p_n)$, where t_i is the name of the ith task, and $p_1, \ldots, p_n$ are the task parameters. An example of a task might be `travel(a, x ,y)` which represents agent `a` travelling between two locations `x` and `y`. The purpose of the planner is to determine how best to complete the given task. To achieve this, STN planners are provided with several *methods*, which specify different ways for decomposing a given task into a set of subtasks. Formally, a method m is defined by the tuple

$$m = \langle \text{name}(m), \text{task}(m), \text{pre}(m), \text{network}(m) \rangle,$$

where (i) name(m) is the method's name; (ii) task(m) is the non-primitive task that the method achieves; (iii) pre(m) is the precondition of the method; and (iv) network(m) is a *task network* or graph that defines a sequence of subtasks and their ordering.

The task network specifies the subtasks that, when executed in the appropriate order, achieve the method's task. While the network specifies a partial ordering on tasks, we may also have a totally ordered network. In this case, the task network is simply a sequence of subtasks that must be executed one after the other.

To continue our example, assume `travel` has two methods to achieve the task: `travel-by-foot` and `travel-by-taxi`. When travelling by foot, the agent need only walk between the two locations. To travel by taxi, however, the agent must first hail a taxi, ride it to the destination, and then pay the driver. Both of these methods will allow the agent to travel between `x` and `y`, provided the agent is at location `x`. However, an additional precondition for `travel-by-foot` is that `x` and `y` are close enough. Figures 5.7 and 5.8 illustrate these two methods based on the above requirements.

The `travel` task is known as a *non-primitive* task, which can be achieved in two different ways. Given a non-primitive task, a planner must use the various methods to decide how best to decompose the task into subtasks. However, tasks can also be *primitive*—in this case, no decomposition need occur. In the above example, let us assume that the agent possesses an action operator for walking between two locations: `walk`. The task network of `travel-by-foot` contains a single task, whose name matches this action operator. The task `walk` is therefore primitive, and

```
method: travel-by-foot(a,x,y)
   task: travel
   precondition: at(a,x) and distance(x,y) < 10
   task network: walk(a,x,y)
```

Figure 5.7: A method for travelling between two locations by foot, applicable only if the distance between the two is less than ten kilometres. The method only has one subtask—walking between locations.

```
method: travel-by-taxi(a,x,y)
   task: travel
   precondition: at(a,x)
   task  network: hail-taxi(a,x) → ride-taxi(a,x,y) → pay(a,y)
```

Figure 5.8: A method for travelling between two locations by taxi, consisting of a sequence of three totally ordered subtasks.

the agent can simply execute the appropriate operator provided the state matches the operator's precondition. The overall aim of a planner, then, is to recursively decompose a task to determine the sequence of primitive tasks (operators) that must be executed.

5.2.2 Learning higher-order options with subgoal discovery

We now describe our approach to construct an abstraction hierarchy automatically using the extended Four Rooms domain from Section 5.1.2, and then apply it to a significantly harder pixel-based video game. We implement the extended Four Rooms domain using the gym minigrid framework [Chevalier-Boisvert *et al.* 2018], with the layout illustrated by Figure 5.9. The egocentric observation is a 3×3 window about the agent, while the problem space is given by its location in the grid.

We assume we are provided with a set of options. In this case these options are exactly those specified at the first level in Section 5.1.2. Our first step is to construct an abstract representation using the approach in Chapter 3, which produces a factored PPDDL representation. Next we must decide how best to discover higher-order skills in this new representation.

Recall that classical planning decouples the domain definition from the task specification. This poses a challenge, since the agent must learn skills in the absence of any particular goal, with the hope that these skills will prove useful in solving future unseen tasks. This is analogous to learning skills in reward-free MDPs [Houthooft *et al.* 2016; Eysenbach *et al.* 2019], but is made easier here because our representation is discrete. Fortunately, prior approaches have used graph-based representations to discover skills in the absence of any reward or goal [Şimşek *et al.* 2005; Machado *et al.* 2017], such as adding edges (options) to minimise the cover time of a transition graph [Jinnai *et al.* 2019]. Therefore, we take a naïve approach and *unfactorise* our representation to create a transition graph, illustrated by Figure 5.10.

We next attempt to identify "important" nodes in the graph, and then construct options to reach these *subgoal* nodes. There are a number of metrics that can be

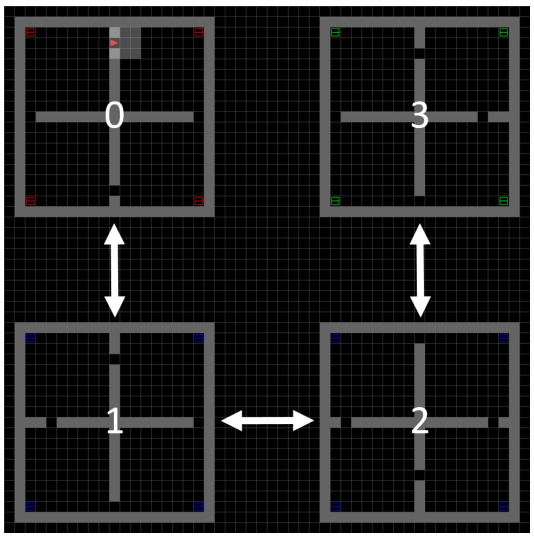


Figure 5.9: The extended Four Rooms domain from Section 5.1.2. The red boxes symbolise stairs that can only be climbed, green boxes are those that can only be descended, and blue boxes are stairs that can either be climbed or descended. The red triangle indicates the agent's position and orientation, and the highlighted cells its current observation. For visualisation purposes, we display the layout on a 2D plane: each floor is labelled with an integer, with white arrows indicating which floor can be accessed by climbing or descending stairs. For example, an agent can transition between floors 2 and 3, but not between floors 0 and 3.

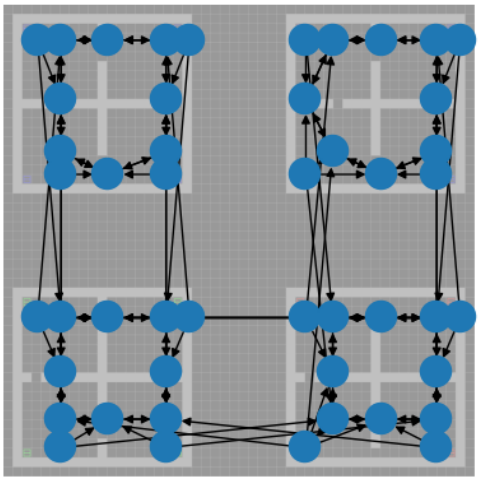


Figure 5.10: A transition graph representing $\tilde{\mathcal{M}}_1$ overlaid on the original domain. The graph is constructed by unfactorising the PPDDL representation. Each node is therefore a conjunction of agent-centric and problem-specific predicates. The location of the nodes corresponds to the xy-position represented by the problem-specific predicates, and each edge in the graph corresponds to a PPDDL operator.

used to measure a node's influence and importance in a graph. For example, Şimşek *et al.* [2005] identifies "bottleneck" nodes using the *betweenness centrality* measure,

which computes the shortest paths between all pairs of nodes and then determines the proportion of paths that pass through a given node. While our approach is agnostic to the metric, we find that the VOTERANK metric [Kitsak *et al.* 2010; Zhang *et al.* 2016], which measures the "influence" of nodes within connected components, discovers more useful subgoals in practice. One intuitive reason for this is that VOTERANK accounts for a node's connectivity within a local neighbourhood, and so identifies spatially distant nodes. Figure 5.11 demonstrates the difference between the two metrics in a small example, while Figure 5.12 indicates the identified subgoals in our transition graph.

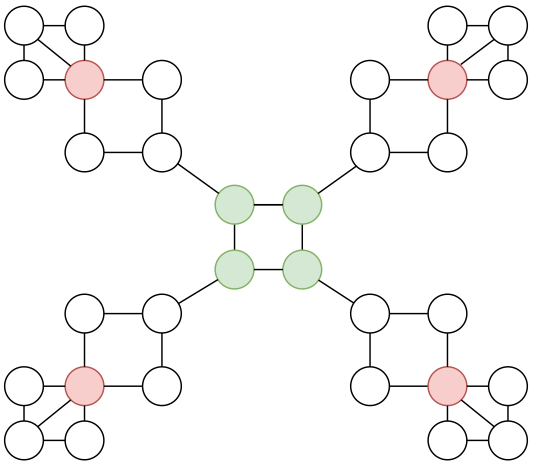


Figure 5.11: Difference between centrality metrics on an example graph. Nodes with the largest betweenness centrality measure are highlighted in green, while the pink nodes indicate those most important according to VOTERANK. One potential issue of betweenness centrality is that it identifies bottleneck states, and so if there are separate "hubs" in the graph, the measure is biased towards nodes that connect these hubs. In this simple example, it is not particularly useful to learn options to reach all green nodes, since they are adjacent to one another. By contrast, VOTERANK identifies nodes *within* each hub, resulting in options that reach more diverse regions of the graph.

We next construct options to reach relevant nodes using a shortest path algorithm, such as Djikstra's algorithm [Dijkstra 1959]. Edges along these paths constitute our higher-order options. The predicates representing the identified subgoal nodes constitute the new state space $\langle \tilde{\mathcal{S}}_2, \tilde{\mathcal{D}}_2 \rangle$ for CPD $\tilde{\mathcal{M}}_2$, while the options comprise the action operators $\tilde{\mathcal{A}}_2$. Our proposed approach to skill acquisition is provided by Figure 5.13. Our option discovery method first identifies relevant subgoal nodes to reach (lines 4–6), and then uses a shortest path algorithm to reach each subgoal from all other nodes (lines 12–14). Each higher-order option is associated with a single subgoal node, and all paths to that node constitute the option's policy. We also restrict the maximum length of higher-order options (line 15), since longer options are less likely to transfer. Note that since all the options contain only a single node in their termination set, they are *subgoal* options by construction.

Having learned representation $\tilde{\mathcal{M}}_2$, we can now simply repeat the process. We first acquire the symbolic representation at the next level using our newly-discovered options. Note that even though we learned these options in an unfactored setting, we use them only to collect data from which we build predicates, and so the resulting

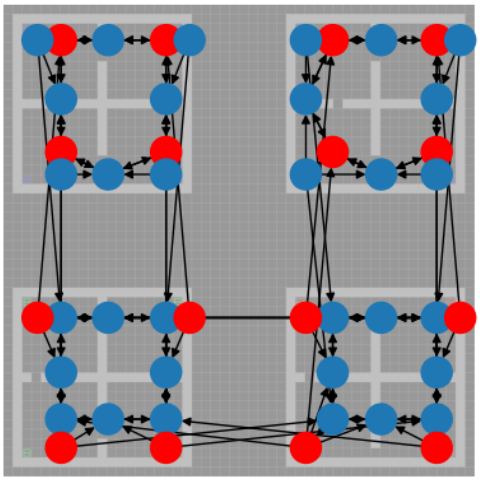
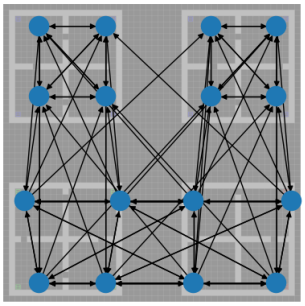


Figure 5.12: Red nodes indicate those identified as influential by VOTERANK.

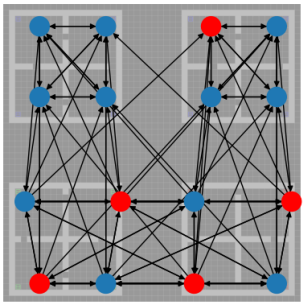
procedure SUBGOALOPTIONDISCOVERY
  Given: transition graph $graph$, importance metric $\mathcal{I}$, max_goals, max_length
  ▷ Find the most important nodes in the graph
  $scores \leftarrow \{\mathcal{I}(n) \mid n \in graph.nodes\}$ ▷ Compute importance of each node
  $subgoals \leftarrow \text{SORT}(graph.nodes, scores)$ ▷ Sort from most to least important
  $subgoals \leftarrow \text{LIMIT}(subgoals, max\_goals)$ ▷ Use only the top few nodes
  ▷ Compute options to reach each node
  $options \leftarrow \emptyset$
  for each $target \in subgoals$ **do**
    $I_o \leftarrow \emptyset; \pi_o \leftarrow \emptyset$
    for each $source \in graph.nodes$ **do**
      ▷ Compute the shortest path from source to target
      $path \leftarrow \text{SHORTESTPATH}(graph, source, target)$
      if $1 < \text{LENGTH}(path) \leq max\_length$ **then**
        $I_o \leftarrow I_o \cup \{source\}$
        $\pi_o \leftarrow \pi_o \cup \{path\}$
      end if
    end for
    $\beta o : target \rightarrow 1$
    $o \leftarrow \langle I_o, \pi_o, \beta_o \rangle$
    $options \leftarrow options \cup \{o\}$
  end for
  return $options$
end procedure

Figure 5.13: Pseudocode for computing options from a learned classical planning domain. The approach accepts a graph representing the transition dynamics, identifies relevant nodes to reach, and then uses any shortest path algorithm to compute options to reach those nodes.

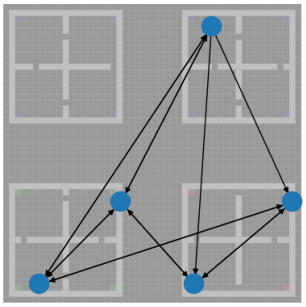
representation is still factorised. We can then construct the unfactorised graph, identify subgoal nodes, and learn new options to reach them. Since we halt when the number of nodes in the graph is less than 2, the process iterates a further two times, as illustrated by Figure 5.14. Pseudocode for our approach to constructing a multi-level hierarchy from a low-level MDP is given by Figure 5.15.



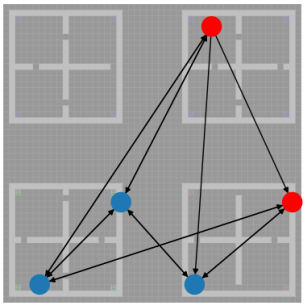
(a) Graph representing $\tilde{\mathcal{M}}_2$



(b) Identified subgoals in $\tilde{\mathcal{M}}_2$



(c) Graph representing $\tilde{\mathcal{M}}_3$



(d) Identified subgoals in $\tilde{\mathcal{M}}_3$

Figure 5.14: The procedure iteratively constructs a hierarchy by converting the PPDDL representation into a graph, identifying nodes, and then learning new higher-order options until the size of the graph drops below some threshold. Red nodes indicate subgoal states which are used to construct skills at the next level in the hierarchy.

5.2.3 Higher-order options as STN methods

In the previous section, we took a simple approach to option discovery by identifying subgoal nodes and then using path planning to construct options. Recall that the edges in the graph representing $\tilde{\mathcal{M}}_1$ are PPDDL operators, and thus the higher-order options consist of a sequence of PPDDL operators. In the STN formalism, these would be considered a totally ordered primitive subtask network. We can therefore treat the attainment of predicates in each target node as an STN task, and the different paths to that node as methods for that task. Figure 5.16 provides an example of this on a simple graph.

Finally, we note that since each higher-order skill consists of a chain of actions at the level below it, the semantics of actions will vary by level. For example, actions $\tilde{\mathcal{A}}_1$ at level 1 are individual PPDDL operators, consisting of sequences of low-level

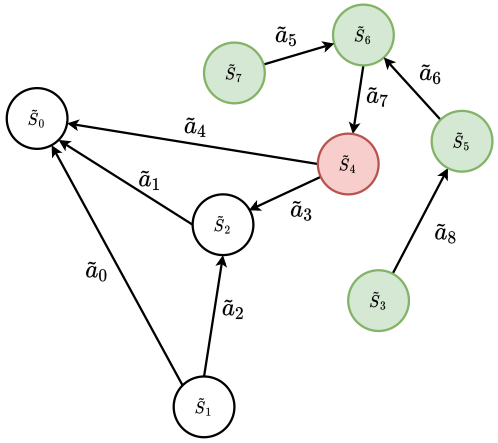
procedure SKILLSYMBOLLOOP
  Given: low-level MDP $\mathcal{M}_0$, initial options $\mathcal{O}$, abstraction algorithm $\mathscr{A}$, importance metric $\mathscr{I}$, graph size reduction N, maximum option length L
  $hierarchy \leftarrow \{\mathcal{M}_0\}$
  $i \leftarrow 0$
  $env \leftarrow \mathcal{M}_0$
  $graph \leftarrow \varnothing$
  $total\_skills \leftarrow \varnothing$
  while $graph = \varnothing$ or $|graph.nodes| > 2$ **do**
    ▷ Acquire skills for the current domain
    $skills \leftarrow \begin{cases} \mathcal{O} & \text{if } graph = \varnothing, \\ \text{OPTIONDISCOVERY}(graph, \mathscr{I}, \frac{|graph.nodes|}{N}, L) & \text{otherwise.} \end{cases}$
    $total\_skills \leftarrow total\_skills \cup \{skills\}$
    ▷ Learn a symbolic representation
    $\mathcal{M}_{i+1} \leftarrow \mathscr{A}(env, skills)$
    ▷ Compute transition graph by unfactorising states
    $graph \leftarrow \text{CONSTRUCTGRAPH}(\mathcal{M}_{i+1})$
    $hierarchy \leftarrow \{\mathcal{M}_{i+1}\}$
    $env \leftarrow \mathcal{M}_{i+1}$
    $i \leftarrow i + 1$
  end while
  return $hierarchy, total\_skills$
end procedure

Figure 5.15: Our approach to learning a hierarchy of abstractions given an initial MDP and set of options. When discovering new options, we use the VOTERANK measure to identify the most influential nodes and restrict the subsequent options to be no longer than a given length.

actions, while $\tilde{\mathcal{A}}_2$ are sequences of PPDDL operators. Table 5.1 categorises the actions at the various levels.

Level	Action semantics
0	Primitive actions in the original MDP
1	PPDDL operators
2	STN methods with totally ordered *primitive* task networks
3+	STN methods with totally ordered *non-primitive* task networks

Table 5.1: Categorising the actions at each level of the hierarchy.



(a) A small graph with target node (pink) and source nodes (green). Edges represent PPDDL operators.

```
(:method method-1
 :task (reach-s4)
 :precondition (s3)
 :ordered-tasks (a8, a6, a7)
)
```

```
(:method method-2
 :task (reach-s4)
 :precondition (s5)
 :ordered-tasks (a6, a7)
)
```

```
(:method method-3
 :task (reach-s4)
 :precondition (s6)
 :ordered-tasks (a7)
)
```

```
(:method method-4
 :task (reach-s4)
 :precondition (s7)
 :ordered-tasks (a5, a7)
)
```

(b) STN methods that reach the target node.

Figure 5.16: (a) In the above example, we identify $\tilde{S}_4$ as a target subgoal node. We can therefore view this as the STN *task* `(:task reach-s4)`. To achieve this task, we can compute the optimal paths from different source nodes. (b) Each of these four paths can be considered a *method* for the task of reaching $\tilde{S}_4$, consisting of a sequence of ordered primitive tasks (PPDDL operators).

5.2.4 Planning with a learned hierarchy

One artefact of our approach is that the state space at level $i > 1$ is a subset of the states at level $i - 1$. This is due to our skill acquisition method—since skills are constructed to reach individual subgoal nodes, the effects of the actions at $\tilde{\mathcal{M}}_i$ are simply these very nodes which represent the states at $\tilde{\mathcal{M}}_{i-1}$. Since we have that $\tilde{\mathcal{D}}_i \subseteq \tilde{\mathcal{D}}_{i-1}$ and $\tilde{\mathcal{S}}_i \subseteq \tilde{\mathcal{S}}_{i-1}$, the question of how best to plan with a hierarchy is moot, because the states at higher levels are *not* more abstract than the ones below. Rather, they are point distributions over lower-level states.[2]

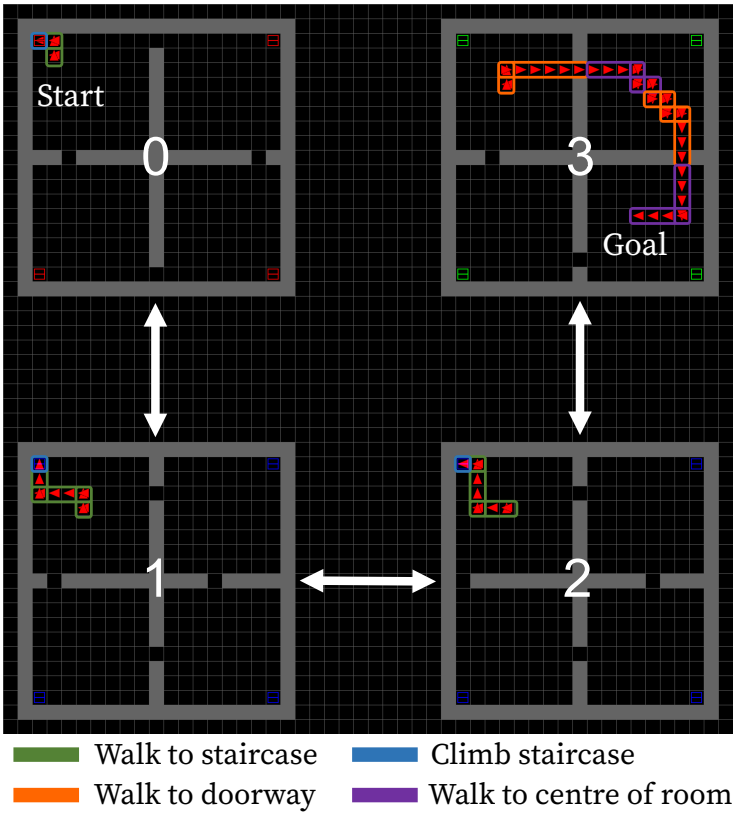
Therefore to plan with these abstractions in practice, we simply combine the PPDDL representation of $\tilde{\mathcal{M}}_1$ with all action operators at each of the above levels. In other words, our operators become more abstract, but our predicates remain the same. This leads to a planning domain consisting of a set of predicates, PPDDL operators, and STN tasks and methods, which can be formalised using a hierarchical planning domain definition langauge such as HPDDL or HDDL [Alford *et al.* 2016; Höller *et al.* 2019].

Unfortunately, there are two issues that prevent us from applying off-the-shelf HTN planners to the resulting representation. One problem is that no existing HTN planners support probabilistic effects. However, an even greater problem is the mismatch between the objectives of HTN planning and the manner in which new problem instances are presented to the agent. To be more precise: a given problem is defined by its *goal*, but the HTN planner requires a *task* as input. Since we have autonomously learned all operators and tasks, *the agent cannot determine which task should be completed in the first place*. Furthermore, the ultimate effect of the task (which is unknown) should align with the goal, but there is no guarantee that such a task has been learned.

To overcome this, we simply "flatten" the STN methods into PPDDL operators [Alford *et al.* 2009], which can be done in a straightforward recursive manner since the subtask network is totally ordered. We then use MINI-GPT [Bonet and Geffner 2005] to construct a plan to reach the centre of the bottom right room on the third floor starting at the top-left room on floor 0. Figures 5.17 and 5.18 illustrate the solutions found when using a single level and all levels of the hierarchy respectively.

As mentioned, one particularly powerful aspect of learning a symbolic model is that it can be used to solve any number of subsequent tasks, characterised by start and goal states. To measure the usefulness of the hierarchy from a planning perspective, we plot the distribution of shortest paths between *all* pairs of start and end states. Figure 5.19 illustrates the advantage of using a multi-level hierarchy, where the lengths of the optimal plans are skewed to be shorter than those of a single-level hierarchy, thereby reducing the planning horizon and improving efficiency.

[2] We can achieve true state abstraction by identifying *groups* of nodes instead of individual subgoal nodes. Our simple approach here is sufficient for the task, and so we leave alternate approaches to future work.

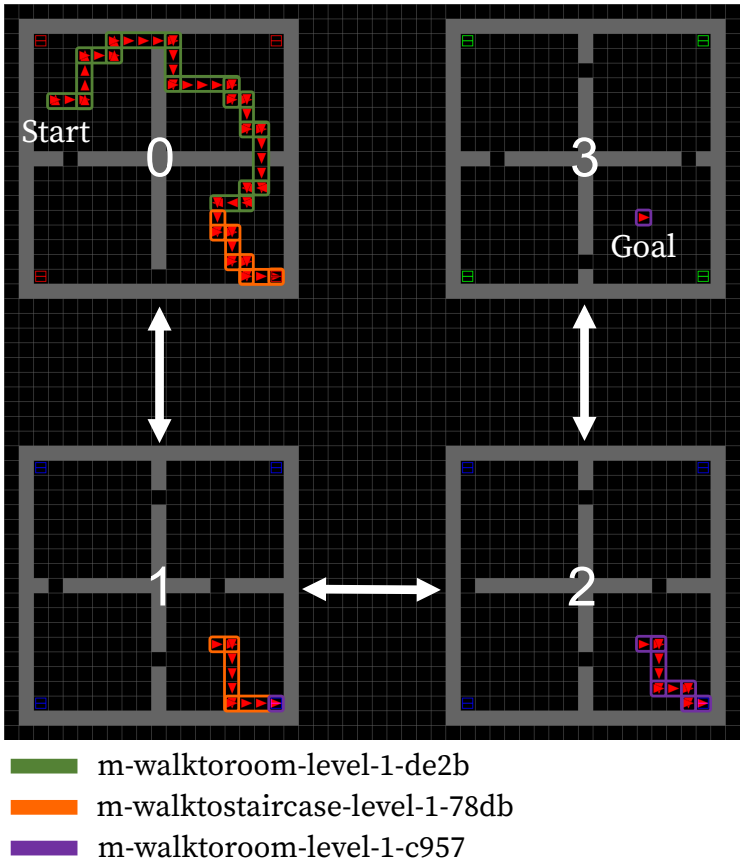


(a) Trajectory of the discovered plan. The legend specifies the meaning of the coloured paths, which represent different options in the full trajectory.

```
['walktostaircase-6758', 'climbup-56e4', 'walktostaircase-46d0',
 'climbup-5b44', 'walktostaircase-7c2b', 'climbup-30a5',
 'walktodoorway1-c56b', 'walktoroom2-981a', 'walktodoorway2-f003',
 'walktoroom2-1f1']
```

(b) Output of the MINI-GPT planner.

Figure 5.17: The trajectory of the solution found by the planner when using only the states and actions at the first level of the abstraction hierarchy. The plan consists of ten PPDDL operators, specified by (b).



(a) Trajectory of the discovered plan. The legend specifies the meaning of the coloured paths, which represent different high-level operators in the full trajectory.

```
['m-walktoroom-level-1-de2b', 'm-walktostaircase-level-1-78db',
 'm-walktoroom-level-1-c957']
```

(b) Output of the MINI-GPT planner, which produces a high-level plan consisting of three higher-order operators to navigate from the start to goal state.

```
(:method m-walktoroom-Level-1-de2b
 :parameters ()
 :task (WalkToStairCase-WalkToRoom-WalkToStairCase-WalkToRoom-Level-1)
 :precondition (and (notfailed) (symbol_0) (psymbol_1))
 :ordered-tasks (and (WalkToDoorway1-87d4) (WalkToRoom2-ef1a)
                 (WalkToDoorway2-889b) (WalkToRoom2-25eb))
)
```

(c) An HTN method for the first task in the plan. This method executes when the agent is in the centre of a room (`symbol_0`) and in the top-left room on floor 0 (`psymbol_1`). The method subsequently executes four lower-level tasks specified by `ordered-tasks`.

```
(:method m-walktostaircase-level-1-78db
 :parameters ()
 :task (WalkToStairCase-ClimbUp-WalkToStairCase-Level-1)
 :precondition (and (psymbol_28) (notfailed) (symbol_0))
 :ordered-tasks (and (WalkToStairCase-ef79) (ClimbUp-46f9)
                (WalkToStairCase-b87e))
)
```

(d) An HTN method for the second task in the plan. This method executes when the agent is in the centre of a room (`symbol_0`) and in the bottom-right room on floor 0 (`psymbol_28`). The method subsequently executes three lower-level tasks specified by `ordered-tasks`.

```
(:method m-walktoroom-level-1-c957
 :parameters ()
 :task (ClimbUp-WalkToStairCase-ClimbUp-Level-1)
 :precondition (and (symbol_2) (psymbol_36) (notfailed))
 :ordered-tasks (and (ClimbUp-4b94) (WalkToStairCase-652b)
                (ClimbUp-a14))
)
```

(e) An HTN method for the final task in the plan. This method executes when the agent is at a staircase (`symbol_2`) and in the bottom-right room on floor 1 (`psymbol_36`). The method subsequently executes three lower-level tasks specified by `ordered-tasks` to reach the final goal.

Figure 5.18: The trajectory of the solution found by the planner when using states and actions at all levels of the abstraction hierarchy. (b) The plan consisting of three methods (converted to PPDDL operators). (c–e) The definition of each of the three learned methods used in the plan, specified using the Hierarchical Domain Definition Language (HDDL) [Höller *et al.* 2019]. Task-specific predicates are highlighted in red.

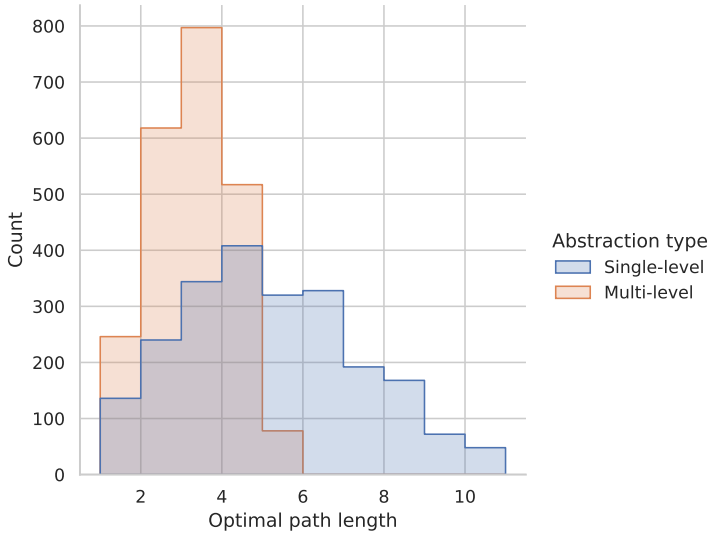
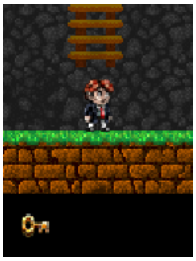


Figure 5.19: Distribution of optimal plan lengths for all pairs of start and goal states. The blue histogram illustrates plan lengths for a single-level hierarchy, while the orange histogram is a result of planning with all layers in the hierarchy.

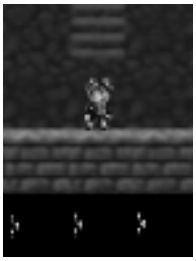
5.3 Experiments in the *Treasure Game*

The above results were generated in a relatively simple environment where there was no opportunity for transfer. In this section, we apply our approach to the *Treasure Game* environment first described in Chapter 3. While the options remain the same, we modify the environment to make it significantly harder—in particular, our agent space is described by a 144×144 pixel window about the agent and a 48×144 window around the inventory. As in Chapter 4, we preprocess these images using PCA [Pearson 1901] to reduce the dimensionality to 25 and 5 respectively. Figure 5.20 illustrates a single level in this modified environment together with the observations before and after preprocessing.

(a)



(b)



(c)

Figure 5.20: An example of the *Treasure Game* environment using pixel observations as the agent space. (a) A single level indicating the position of the agent at the bottom of a ladder. (b) The agent space representation, specified as a local window centred on the agent. (c) A PCA reconstruction of the agent's observation.

5.3.1 Single-task experiments

We begin by applying our approach to each of the levels individually. For each task we use uniform random exploration to collect data over 30 episodes, and then construct abstraction hierarchies using the skill-symbol loop outlined in Figure 5.15 and

the abstraction algorithm from Chapter 3.[3] Figure 5.21 illustrates the interleaving of transition graph construction and subgoal identification for one of the tasks.

As mentioned, the resulting higher-order options can be viewed as STN methods. We formalise these STN tasks and methods using HDDL [Höller *et al.* 2019] and visualise the decomposition of several learned tasks. Figure 5.22 illustrates a single STN task for reaching a particular location along with two of its method decompositions, while Figure 5.23 shows an STN task constructed at level 3 in the hierarchy.

Finally, we compute the distribution of the length of all pairs of shortest paths for each of the tasks when using abstractions from varying levels of the hierarchy. Results for the first two tasks are given by Figure 5.24 and indicate that incorporating information at increasingly abstract levels of the hierarchy reduces the size of the graphs.[4] Consequently, the maximum planning horizon is shortened, which greatly simplifies the planning problem.

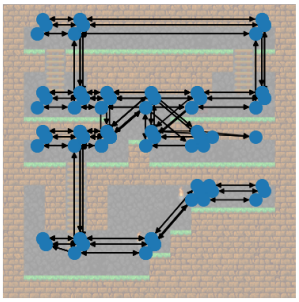
5.3.2 Transfer with hierarchies

The previous section demonstrates the advantage of learning a hierarchy for a single task. However, we have yet to quantify the effect of a *portable* hierarchy. To achieve this, we construct ten different levels of the *Treasure Game*, the layouts of which are listed in Appendix C.1. We then investigate transfer by presenting the agent with each of the ten tasks in sequence. Given task i, the agent must construct a model of the task by combining representations learned in the current task with those learned in previous tasks $i-1, i-2, \ldots, 0$. Unlike Chapter 3, portable representations here consist of not just PPDDL operators, but also STN methods at various levels in the hierarchy.

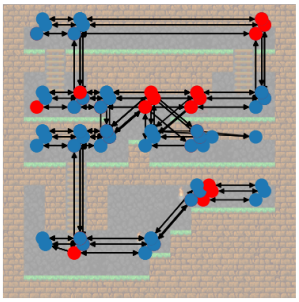
To measure the correctness of a learned model, we randomly sample 100 start and goal states for each task, and compute the optimal plan between them. These serve as a set of test problems with which we can evaluate a learned model—we query the learned model with each test problem, and record whether the resulting plan coincides with the ground-truth plan.

We use the non-portable *skills-to-symbols* approach [Konidaris *et al.* 2018] as a baseline to measure the effect of transfer. Since each task varies in difficulty, we compare the *relative* performance of the two approaches. For the baseline approach, we collect up to 50 episodes' worth of data for each task, and determine the number of episodes needed to build a model that maximises the test score. When evaluating transfer, we determine how many episodes are required to *outperform* the baseline for the current task; once this threshold is reached, the agent proceeds on to the next task. For both approaches, we use uniform random exploration and randomise the order of tasks over 100 trials.

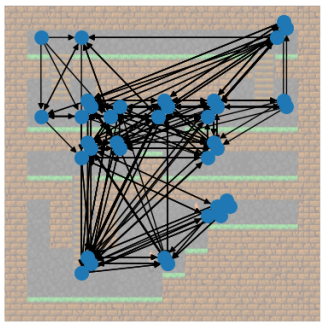
[3] The hyperparameters for the procedure are listed in Appendix C.2.
[4] Histograms for all tasks are provided in Appendix C.3.



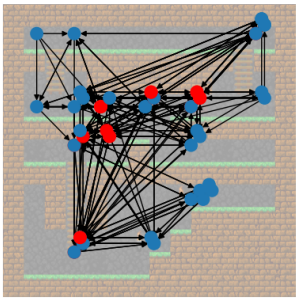
(a) Graph representing $\tilde{\mathcal{M}}_1$



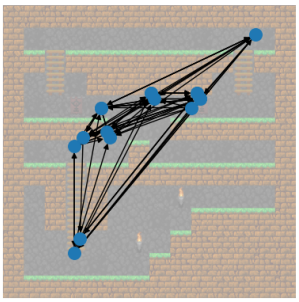
(b) Identified subgoals in $\tilde{\mathcal{M}}_1$



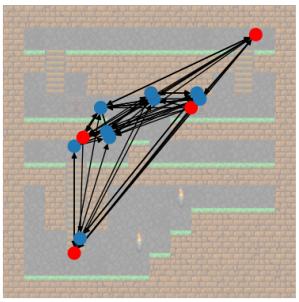
(c) Graph representing $\tilde{\mathcal{M}}_2$



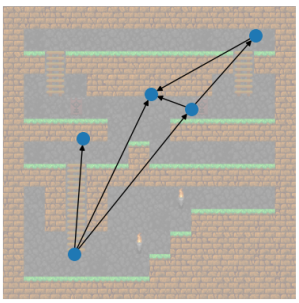
(d) Identified subgoals in $\tilde{\mathcal{M}}_2$



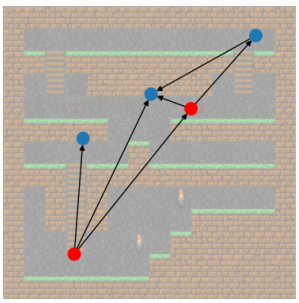
(e) Graph representing $\tilde{\mathcal{M}}_3$



(f) Identified subgoals in $\tilde{\mathcal{M}}_3$

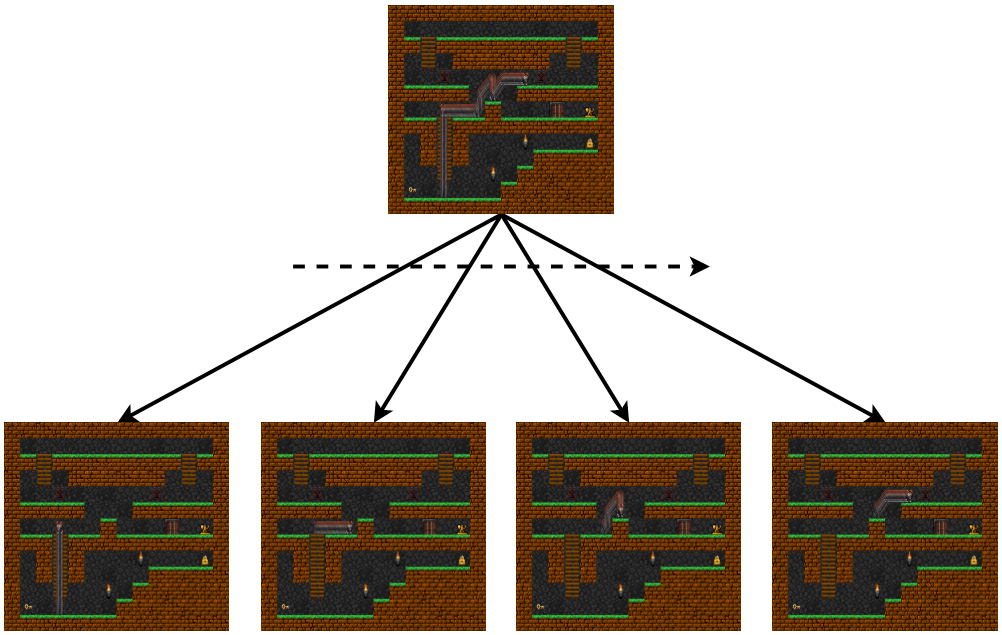


(g) Graph representing $\tilde{\mathcal{M}}_4$



(h) Identified subgoals in $\tilde{\mathcal{M}}_4$

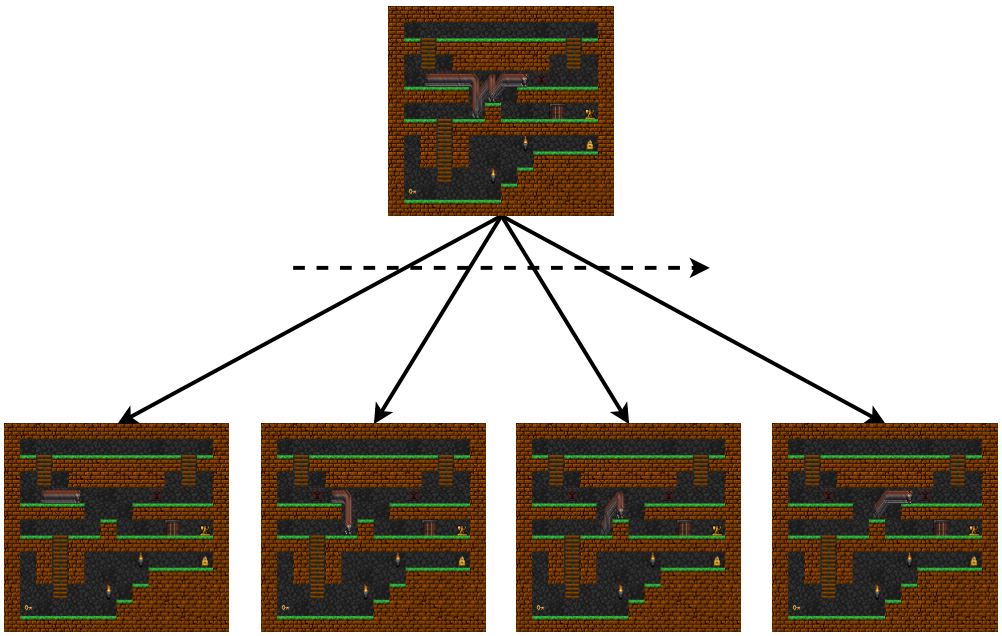
Figure 5.21: Our iterative procedure for automatically constructing a hierarchy in a single level of the *Treasure Game*.



(a) Task decomposed into four primitive subtasks.

```
(:method m-Level-2-0-30d9
 :parameters ()
 :task (up_ladder-Level-2-b881)
 :precondition (and (notfailed) (psymbol_0) (symbol_3))
 :ordered-tasks (and (up_ladder_option-25ea) (go_right_option-e2a3)
                (jump_right_option-5580) (jump_right_option-fd48))
)
```

(b) HDDL representation of the method in (a).

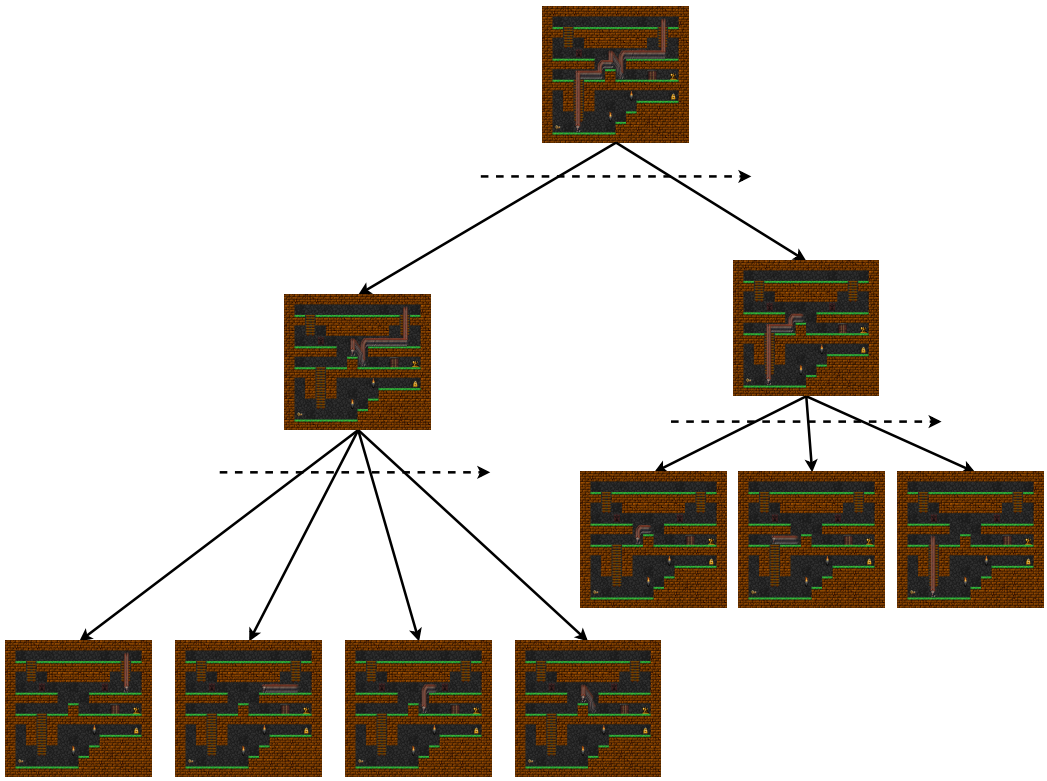


(c) Task decomposed into four primitive subtasks.

```
(:method m-Level-2-0-1a76
 :parameters ()
 :task (up_ladder-Level-2-b881)
 :precondition (and (notfailed) (symbol_1) (psymbol_2) (symbol_3))
 :ordered-tasks (and (go_right_option-40f5) (down_right_option-9b97)
                (jump_right_option-1933) (jump_right_option-fd48))
)
```

(d) HDDL representation of the method in (c).

Figure 5.22: A representation of a single task for reaching a location between a ledge and flag, along with two of its methods. The dashed arrow indicates the order in which subtasks must be executed. The red symbol is a problem-specific predicate—a distribution over problem-space state variables such as the agent's xy-position—that must be relearned for a new task.



(a) Task decomposed into two non-primitive subtasks. Each of these subtasks is further decomposed into a set of primitive subtasks.

```
(:method m-Level-3-a949
 :parameters ()
 :task (down_left-Level-3-a447)
 :precondition (and (symbol_24) (symbol_1) (notfailed) (psymbol_13))
 :ordered-tasks (and (down_right-Level-2-1c97) (down_left-Level-2-d5c3))
)
```

(b) HDDL representation of the method at level 3 of the hierarchy.

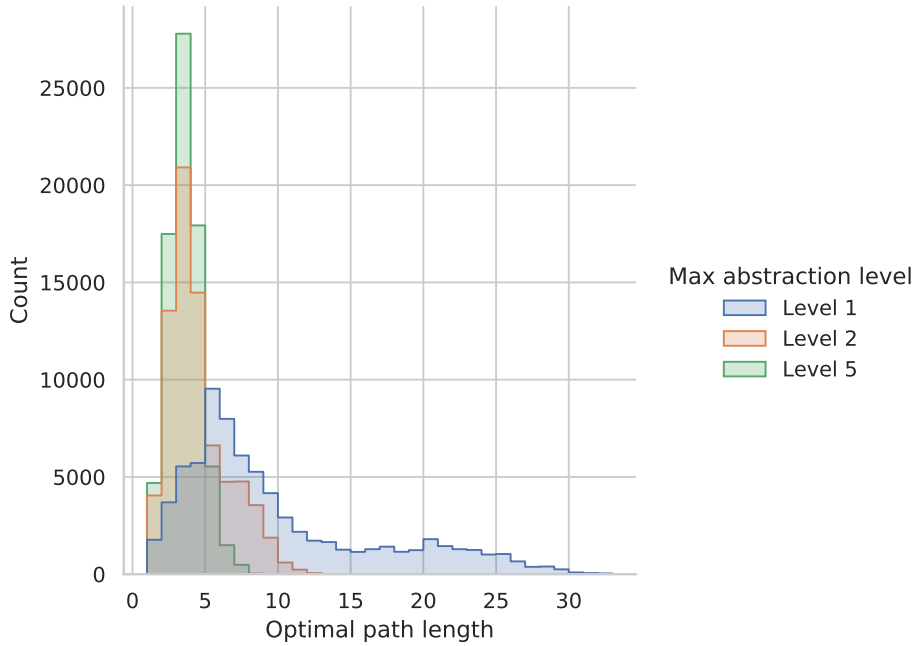
```
(:method m-down_right-Level-2-65b3
 :parameters ()
 :task (down_right-Level-2-1c97)
 :precondition (and (psymbol_13) (notfailed) (symbol_1) (symbol_24))
 :ordered-tasks (and (down_ladder_option-362c) (go_left_option-18ab)
                (down_left_option-5453) (jump_left_option-84a7))
)
```

(c) HDDL representation of a method that decomposes the first subtask of (b).

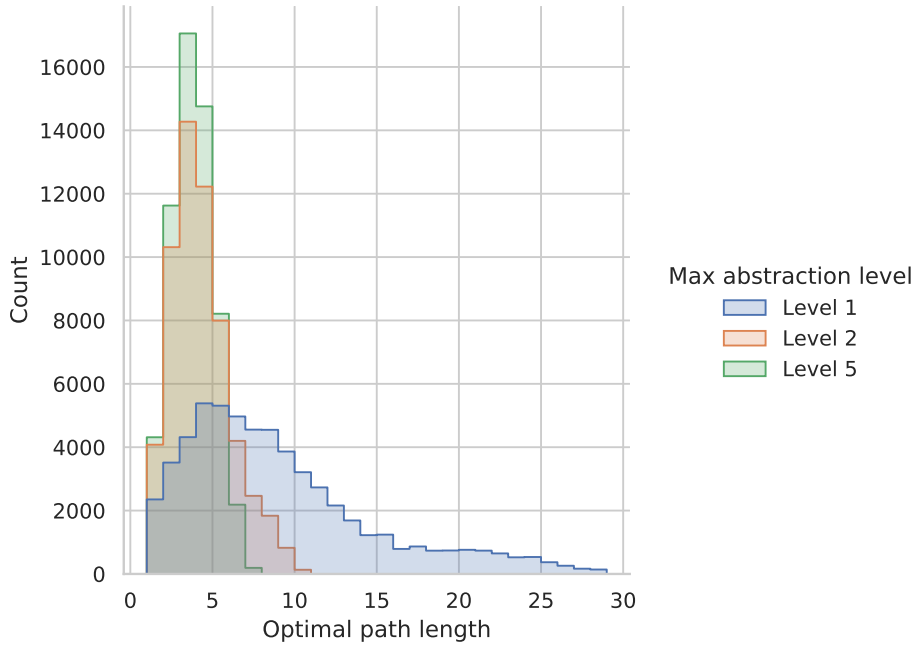
```
(:method m-down_left-Level-2-ed21
 :parameters ()
 :task (down_left-Level-2-d5c3)
 :precondition (and (notfailed) (psymbol_38) (symbol_30))
 :ordered-tasks (and (down_left_option-6e4b) (go_left_option-c084)
                (down_ladder_option-460d7))
)
```

(d) HDDL representation of a method that decomposes the second subtask of (b).

Figure 5.23: A representation of a single task at level 3 of the hierarchy, consisting of two non-primitive subtasks. The dashed arrow indicates the order in which subtasks must be executed, while red symbols are problem-specific predicates that must be relearned for a new task.



(a) Distribution of optimal plan lengths in the first task when using hierarchies of varying heights.



(b) Distribution of optimal plan lengths in the second task when using hierarchies of varying heights.

Figure 5.24: Distribution of optimal plan lengths for all pairs of start and goal states. For visualisation purposes, we omit the histograms at levels 3 and 4 of the hierarchy, since they are very similar to the histogram at level 5.

We first investigate the effect of transfer on an agent's sample efficiency. Since an agent can reuse previous predicates, the model of any single task will potentially consist of predicates learned in the current task, as well as those learned in previous tasks. We record the proportion of predicates in a task's model that were transferred from previous tasks, with the results given by Figure 5.25. We then measure the number of samples required to learn a model of a new task, with the results illustrated by Figure 5.26. Although the results exhibit high variance (due to the exploration strategy, the differences in tasks, and the randomised task order), sample efficiency is clearly improved when an agent is able to reuse past knowledge.

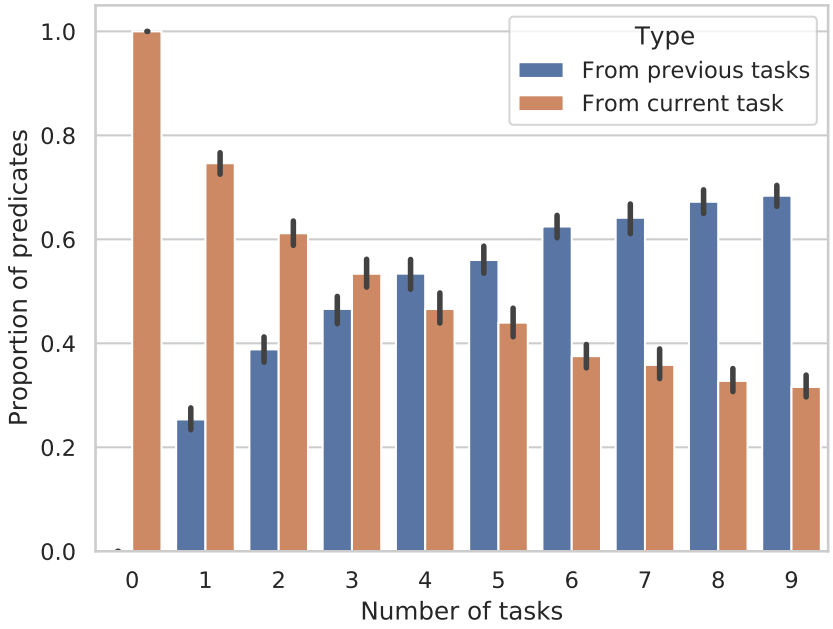


Figure 5.25: The proportion of new predicates learned in the current task compared with previously learned ones. The agent is able to reuse an increasing number of predicates as it observes more and more tasks. Results are averaged over 100 random task orderings, with the variance indicated by black error bars.

Unfortunately, when considering sample efficiency alone, we observe no advantage to transferring more than one level of the hierarchy. We attribute this result to a number of factors, the most important of which is the use of uniform random exploration. In order for a PPDDL operator or STN method to transfer, the agent must instantiate it with problem-specific predicates. Using undirected exploration means that the probability of an agent executing all actions along a method's trajectory (and therefore instantiating it) is inversely proportional to the length of its task network. Furthermore, higher-order skills are less likely to transfer by definition—these skills are composed of a long chain of actions at the lowest level, which must all be executable in precisely the same order for the skill to transfer. While the tasks share some structure, they have not been specifically designed as a curriculum, and so long chains of actions in one task are unlikely to be applicable in another.

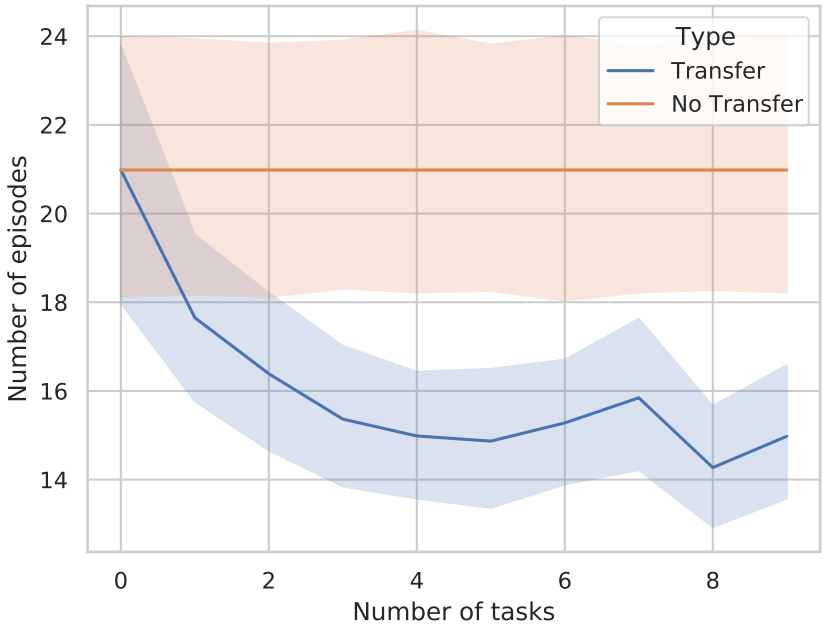


Figure 5.26: Number of episodes required to learn a model of a given task. The amount of data required to learn a task model decreases as the agent observes an increasing number of tasks. Results are averaged over 100 random task orderings, with the variance indicated by shaded bars.

Figure 5.27 indicates that the majority of transfer is due to the representations at the first level of the hierarchy.

Despite this, there are still advantages to transferring all levels of the hierarchy. Although we do not observe improvements in sample efficiency in aggregate, recall that the hierarchy is constructed using the PPDDL representation $\tilde{\mathcal{M}}_1$ itself. As a result, building a multi-level hierarchy uses the same number of environment samples as the first level alone. We can therefore expect that transferring a hierarchy will never *worsen* sample efficiency. However, one stated advantage of a hierarchical representation is its effect on planning. To quantify this, we compute the diameter of the resulting graph—the maximum shortest path length between every pair of nodes—when restricting transfer to varying levels of the hierarchy. The results given by Figure 5.28 indicate that transferring all layers of the hierarchy does confer advantages by reducing the size of the planning horizon. Interestingly, our results support the findings of Riemer *et al.* [2018], who learn a hierarchy of increasingly abstract options, but find that there are diminishing returns as the options become more abstract.

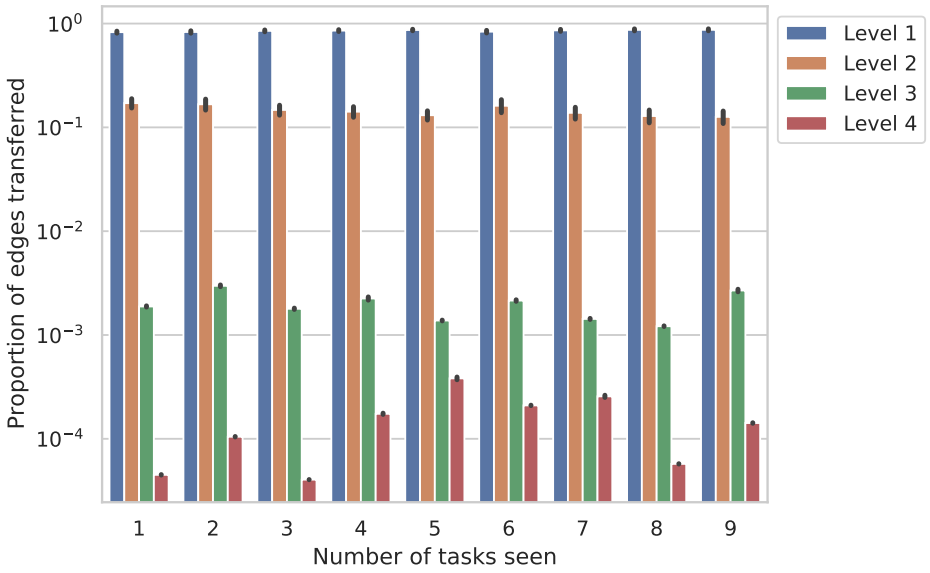


Figure 5.27: PPDDL operators (blue) make up a significant portion of the transferred edges. Note that the y-axis is plotted on a log scale. Results are averaged over 100 random task orderings, with the variance indicated by black error bars.

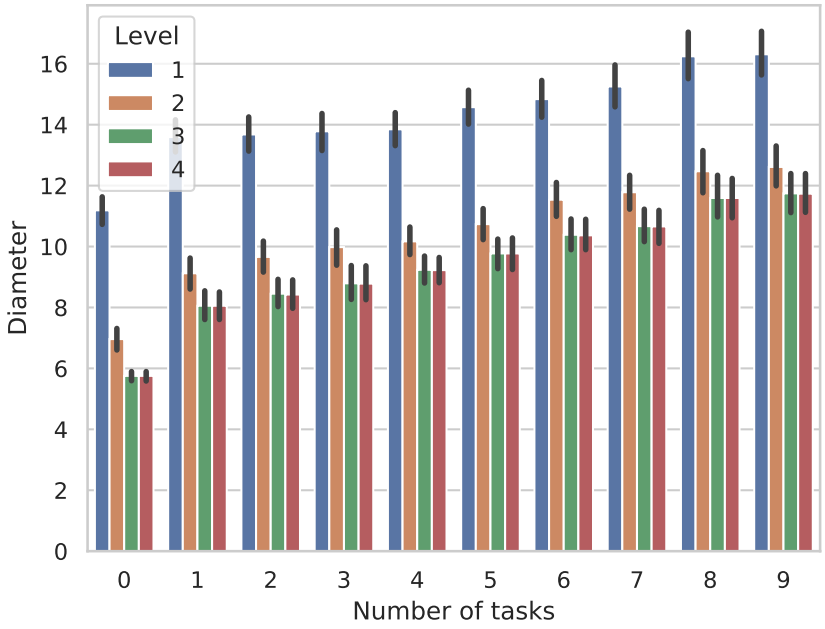


Figure 5.28: Results are averaged over 100 random task orderings, with the variance indicated by black error bars.

5.4 Related Work

There are several approaches to building a multi-level hierarchy using state and action abstraction. As mentioned, Konidaris [2016] constructs a hierarchy of increasingly abstract representations using a skill-symbol loop. However, only deterministic tasks are accounted for, and the skills at each level of the hierarchy are handcrafted. Riemer *et al.* [2018] learn a hierarchy of increasingly abstract actions in an end-to-end manner using a neural network. Much like our approach, actions at level i consist of a sequence of actions at $i - 1$. However, the state space remains the same as in the original MDP.

Hengst [2002] decomposes an MDP into sub-MDPs by ordering state variables based on how often they change. Variables are grouped into MDPs at various levels depending on how often they change—those that change frequently serve as lower-level MDPs—and transition dynamics between these decompositions are learned. Barry *et al.* [2011] aggregate MDP states into clusters while attempting to minimise both the number of transitions between abstract states under an optimal policy, and the variance in state values within each cluster. This results in a two-level hierarchy, consisting of a high-level policy for transitioning between abstract states, and a low-level one for acting within any cluster. In both of these cases, the algorithms only succeed if transitions between the high-level abstractions occur with probability 1, which is not the case in stochastic environments.

Other approaches model the causal relationships between state variables of factored MDPs [Boutilier *et al.* 1995], and then learn options to modify these variables [Jonsson and Barto 2006; Vigorito and Barto 2010]. The original MDP is decomposed into option-specific tasks which are significantly easier to solve: since only a subset of state variables are causally related to the one modified by the option, the remaining variables can be ignored. In all of the above, the hierarchy cannot be reused when an agent encounters a new domain.

An alternate approach is MAXQ framework [Dietterich 2000; Mehta *et al.* 2008], which decomposes an MDP into a hierarchy of subtasks with associated subgoals. However, there is no model to capture transitions between higher-order abstractions in the hierarchy, and so planning cannot be done at an abstract level. The Abstract MDP framework [Gopalan *et al.* 2017] allows for transitions to occur at varying levels of abstraction, and demonstrates that using a hierarchy can significantly improve planning. However, the hierarchy itself is hand-designed.

Recent work has focused on feudal RL [Dayan and Hinton 1993], where high-level layers in the hierarchy provide information to layers below them. For example, Nachum *et al.* [2018; 2019] construct a two-level hierarchy, where the top layer proposes goals for the lower level to attain. Levy *et al.* [2019] extend this approach to learn a multi-level hierarchy using hindsight experience replay [Andrychowicz *et al.* 2017]. Hejna *et al.* [2020] adopt a similar two-level approach, but divide the state space into agent- and problem-space state variables. The top layer proposes goals in problem space, and the lower layer learns how to achieve them. This allows for transfer between *agents*—since the top layer can be reused, only the lower layer must be relearned. Christen *et al.* [2021] assume that the problem space consists

of a top-down map, and train a neural network to predict the value function given the map and goal location [Nardelli *et al.* 2018]. The top layer is therefore able to generalise over new maps, resulting in improved transfer in new configurations. All of these approaches are model-free, and do not produce explicit representations that can be used by planners.

Saxe *et al.* [2017] use the linearly-solvable MDP (LMDP) framework [Todorov 2007] to construct a hierarchy of LMDPs, where higher layers share reward information with lower layers, allowing for planning at various levels. However, LMDPs require a predefined "passive dynamics" model, which is quadratic in the number of states. It is therefore unclear how to extend this approach beyond the tabular setting.

Finally, our approach results in the construction of STN tasks and methods. Other approaches to learning how best to decompose HTN tasks into methods exist [Zhuo *et al.* 2009; Hogg *et al.* 2010]; however, in these cases the symbolic vocabulary is already given.

5.5 Conclusion

We have outlined a method for autonomously constructing a hierarchy of abstractions given an initial set of options. These hierarchies can be learned from continuous, high-dimensional environments, and are constructed in the absence of any particular goal information. They are thus suitable for planning in any future task an agent may encounter. The hierarchies can also be transferred to new tasks where, although sample efficiency is not significantly improved, planning is made more efficient.

Although we have shown that a portable hierarchy can be learned and transferred to new tasks, our approach raises many new questions about exploration, how best to discover higher-order options, and how best to plan with them. For example, our resulting representations can be viewed as STN tasks and methods. Unfortunately, owing to shortcomings of HTN planners, we cannot directly leverage this formulation. We discuss potential solutions to these problems and other avenues for research in the next chapter.

Chapter 6

Conclusion

Our aim in this thesis was to develop techniques for autonomously learning reusable symbolic representations. To this end, we proposed three approaches: first, we showed how to learn an agent-centric representation that can be used for symbolic planning. These representations can be transferred to new tasks, reducing the number of times an agent is required to interact with the world. We then extended this approach to learn transferable object-centric representations, allowing us to construct parameterised symbolic representations that have long been staples in classical planning. Finally, we showed how to construct a portable hierarchy of abstractions that can be used to plan at different levels. Altogether, our results indicate that the learned abstractions can be reused in new tasks, reducing the number of times an agent is required to interact with its environment. This improvement in sample efficiency will be critical to scaling symbol acquisition approaches to real world tasks in the future.

6.1 Future work

There are, of course, several ways that these approaches can be directly improved. One particular avenue is to use feedback from the various system components to improve the symbol acquisition process. For example, in our implementations, predicates learned in one task were "frozen" and transferred to subsequent tasks. However, these predicates may capture the idiosyncrasies of the task in which they were learned. We may therefore be able to use data from a new task to refine these symbols and maximise their generalisability. Another potential source of feedback is the planner itself. For example, if the planner outputs a solution that is not executable in the real domain, it suggests that the agent may have learned an "invalid" symbol. This could occur because the agent only uses a finite amount of data to construct the symbols, and so may have overgeneralised its experience. Such information could be used to identify symbols that should be refined (e.g. by collecting more data) or discarded. These feedback loops would result in a more robust system overall.

In our experiments, we adopted the underlying machine learning algorithms first used by Konidaris *et al.* [2018]—partitioning was accomplished with the DBSCAN

algorithm, and the preconditions and effects were learned using support vector machines and kernel density estimators respectively. Furthermore, we used principal component analysis to reduce the dimensionality of state spaces. These techniques were advantageous for two reasons: they allowed us to be consistent with prior work, and they are extremely sample efficient. An alternative approach is to use neural networks, and train classifiers and density estimators in an end-to-end manner with stochastic gradient descent [Rumelhart *et al.* 1986]. While neural networks are likely more robust and capable of better generalisation, they require magnitudes more data to train. Additionally, it is not immediately clear how to incorporate factors, nor how best to perform partitioning and effect estimation for image-based input. A promising direction for future work is therefore to investigate whether approaches such as mixture density networks [Bishop 1994] and deep clustering techniques [Caron *et al.* 2018] are suited to the task, and quantify the tradeoffs thereof.

One goal of this work is to learn representations that can be used by existing planners, leveraging decades of work in the planning community. To this end, our agent learned representations described using PPDDL. This was necessary because the low-level MDP was stochastic, and so probabilistic effects must be accounted for. Unfortunately, recent versions of PDDL only support deterministic planning [Gerevini *et al.* 2009], which restricts the set of planners available to us. Modifying our approach to produce an alternate planning language, such as the more recent Relational Dynamic Influence Diagram Language (RDDL) [Sanner 2010], would make our methods applicable to a wider variety of off-the-shelf planners.

Our approach is similarly restricted when it comes to hierarchical planning—in Chapter 5 we observe an objective mismatch between HTN and classical planners. Since MDPs are goal-oriented, we were required to convert our STN methods to classical representations. However, it is possible to combine the two approaches. For example, Shivashankar *et al.* [2013] propose a goal-oriented HTN formalism, while Gerevini *et al.* [2008] run classical and HTN planners in parallel, deciding which one to use based on the current context. Applying similar approaches to our work will likely improve planning efficiency even further.

Finally, recall that our overall aim is to develop agents capable of operating beyond the restrictions imposed by MDPs. While we show how to learn representations in an agent- and object-centric manner, the agent is still required to include additional variables in the state descriptor. This allows the agent to disambiguate similar-looking states, but necessitates a human designer's input. Future work should therefore focus on removing this requirement by allowing the agent to infer this additional information from observations. For example, navigation-based agents could perform simultaneous localisation and mapping [Leonard and Durrant-Whyte 1991], and use the output to disambiguate similar egocentric observations. Additionally, we assumed in Chapter 4 that the agent was able to individuate objects and had knowledge of which particular object it was interacting with. Removing these assumptions (for example, by applying techniques that can extract objects from scenes [Yuan *et al.* 2021]) will serve as another step towards truly autonomous agents.

6.2 Discussion

Beyond the immediate extensions above, our work suggests broader avenues for future research. There are, for example, several implications for skill acquisition. Our work suggests that acquiring skills is critical, since they are used to subsequently generate symbolic representations and object types. In order to construct preconditions and effects, however, we require that the skills have the subgoal property, but the clustering method used to achieve this is fragile and error-prone. However, this issue could be obviated by simply learning subgoal options initially. This suggests that option discovery methods that "funnel" an agent towards a set of termination states are highly desirable. Additionally, since we wish to chain sequences of skills, options should be learned such that the termination set of one overlaps with the initation set of another, as in approaches such as skill chaining [Konidaris and Barto 2009b; Bagaria and Konidaris 2019]. While options have generally been learned with the aim of improving policy learning in a model-free context [Sutton *et al.* 1999; Bacon *et al.* 2017], focusing instead on their implications for constructing useful and compact abstract representations may provide new insights into tackling the skill acquisition problem.

Since we are ultimately interested in agents that can solve multiple tasks, learning *portable* options is also an important goal. This suggests that future work focus on learning *factored* skills, whose initiation sets, policies and termination conditions are defined over as few state variables as possible. Reducing the dependence on the number of state variables will help prevent options from overfitting to the task in which they were learned, and maximise their probability of being useful in other tasks. Estimating the effects of such options would also be made easier, since we need not estimate a distribution over the entire state space. Learned options that explicitly modify only a small number of state variables have been the subject of recent work [Allen *et al.* 2021a], and progress in this direction will likely result in more useful and portable symbolic representations.

The ability to autonomously generate symbolic tasks and domains also suggests a further direction on the planning front. The generated domains often consist of tens or hundreds of duplicate predicates, extraneous operators and noisy symbols. Future work may therefore consider the construction of planners specifically designed to operate in such cases, instead of the clean hand-designed domains currently in use. For example, Helmert [2006] uses a causality graph to ignore irrelevant lifted predicates, while Kumar *et al.* [2020] use knowledge of the start and goal states to remove both predicates and real-valued symbols from the domain. Work in this direction could be used in conjunction with our framework to tackle extremely large environments.

Finally, the most salient question raised is how best to perform exploration within this framework. While Andersen and Konidaris [2017] propose a method for exploration that minimises a model's uncertainty within a single task, extensions to the multitask case present a number of challenges. In particular, how can we leverage operators from previous tasks to improve learning a model in a new task? Furthermore, as observed in Chapter 5, these operators may exist at different levels of the hierarchy. Addressing this problem adequately is a key challenge in the short term, and will likely lead to a significant increase in an agent's sample efficiency.

6.3 Concluding remarks

Reinforcement learning has made great strides in recent years, and has been used to solve a number of *grand challenges* [Silver *et al.* 2017; 2018]. Although these are important milestones, the sample (in)efficiency of these approaches means they are unlikely to be applicable in the real world. More importantly, though, these approaches make an unacknowledged yet ultimately fatal assumption—namely, that *MDPs are real*.

This thesis reflects the belief that artificial intelligent agents must, ultimately, be capable of operating in the real world. Since real agents can only observe the world through their sensors, they must necessarily be able to learn sufficiently accurate and compact representations of any task they may be required to solve. The agent may decide to encode a task as an MDP or a PPDDL domain, or any other number of representations, but whatever the case, they must be autonomously constructed given only what is available to the agent—its sensory observations. To achieve this, we require algorithms capable of successfully learning when operating in noisy observation spaces that are typical of the hardware sensors an agent might possess, as opposed to the clean Markovian state spaces currently in use.

Furthermore, a hallmark of human intelligence is our ability to tackle a massively diverse array of tasks. Importantly, though, we do not achieve (nor strive for) optimality—no one individual is simultaneously the best golfer, the best pianist and the best chess player. Rather, we define intelligence as being able to perform all of these tasks, and more, with a modicum of success. This suggests that the focus of future work should revolve around agents that can achieve *competence* across a large number of tasks directly from sensor data.

To see why we should aim for multitask competency, consider the game of chess. For many decades, defeating the world champion in chess was seen as *the* desirable goal in artificial intelligence research. However, because the aim was to outperform all humans, approaches were designed with this narrow task in mind, and therefore used specialised data structures and compressed representations that simplified the problem dramatically. Consequently, having designed a super-human chess agent, we discovered that we had learned less about how to design an intelligent agent, and more about how best to build a chess-playing program!

In our view, a *grander* challenge would be to develop a physical agent capable of playing a reasonable game of chess on a real chess board, given only its observations

and innate action space. Such an agent would need to learn a representation that encodes the position of the pieces and the general rules, while being agnostic to the surrounding conditions, such as lighting, size of the pieces, colour of the board, and so on. Furthermore, the same agent should then be able to apply the same techniques to play checkers, backgammon and Go, with the ultimate aim of being able to competently solve as many tasks as possible.

Taken as a whole, the methods presented in this work allow us to develop agents that are able to acquire high-level concepts that are abstracted away from low-level observations from which they were learned. Although there is still much to be done, we believe that this thesis represents a step towards the ultimate goal of constructing generally intelligent artificial agents.

References

[Agre and Chapman 1987] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, volume 87, pages 286–272, 1987.

[Ahmetoglu *et al.* 2020] A. Ahmetoglu, M. Seker, A. Sayin, S. Bugur, J. Piater, E. Oztop, and E. Ugur. DeepSym: Deep symbol generation and rule learning from unsupervised continuous robot interaction for planning. *arXiv preprint arXiv:2012.02532*, 2020.

[Aineto *et al.* 2019] D. Aineto, S. Celorrio, and E. Onaindia. Learning action models with minimal observability. *Artificial Intelligence*, 275:104–137, 2019.

[Alford *et al.* 2009] R. Alford, U. Kuter, and D. Nau. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1629–1634, 2009.

[Alford *et al.* 2016] R. Alford, H. Behnke, D. Höller, P. Bercher, S. Biundo, and D. Aha. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26, 2016.

[Allen *et al.* 2021a] C. Allen, M. Katz, T. Klinger, G. Konidaris, M. Riemer, and G. Tesauro. Efficient black-box planning using macro-actions with focused effects. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, pages 4024–4031, 2021.

[Allen *et al.* 2021b] C. Allen, N. Parikh, O. Gottesman, and G. Konidaris. Learning Markov state abstractions for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2021.

[Ames *et al.* 2018] B. Ames, A. Thackston, and G. Konidaris. Learning symbolic representations for planning with parameterized skills. In *Proceedings of the 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018.

[Amir and Chang 2008] E. Amir and A. Chang. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

[Andersen and Konidaris 2017] G. Andersen and G. Konidaris. Active exploration for learning symbolic representations. In *Advances in Neural Information Processing Systems*, pages 5016–5026, 2017.

[Andrychowicz *et al.* 2017] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, 2017.

[Asai and Fukunaga 2018] M. Asai and A. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[Asai and Muise 2020] M. Asai and C. Muise. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to STRIPS). In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, 2020.

[Asai 2019] M. Asai. Unsupervised grounding of plannable first-order logic representation from images. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 583–591, 2019.

[Bacon *et al.* 2017] P. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[Badia *et al.* 2020] A. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. Guo, and C. Blundell. Agent57: Outperforming the Atari human benchmark. In *Proceedings of the International Conference on Machine Learning*, pages 507–517. PMLR, 2020.

[Bagaria and Konidaris 2019] A. Bagaria and G. Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2019.

[Bajpai and Garg 2018] A. Bajpai and S. Garg. Transfer of deep reactive policies for MDP planning. In *Advances in Neural Information Processing Systems*, pages 10988–10998, 2018.

[Barry *et al.* 2011] J. Barry, L. Kaelbling, and T. Lozano-Pérez. DetH*: Approximate hierarchical solution of large Markov decision processes. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011.

[Barto and Mahadevan 2003] A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[Beattie *et al.* 2016] C. Beattie, J. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, et al. DeepMind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[Bellman 1957] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Bertsekas and Tsitsiklis 1991] D. Bertsekas and J. Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991.

[Bishop 1994] C. Bishop. *Mixture density networks*. Technical report, Aston University, 1994.

[Bonet and Geffner 1999] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *European Conference on Planning*, pages 360–372. Springer, 1999.

[Bonet and Geffner 2003] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 3, pages 12–21, 2003.

[Bonet and Geffner 2005] B. Bonet and H. Geffner. mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, 24:933–944, 2005.

[Bonet and Geffner 2020] B. Bonet and H. Geffner. Learning first-order symbolic representations for planning from the structure of the state space. In *European Conference on Artificial Intelligence*, 2020.

[Bonet *et al.* 2019] B. Bonet, G. Frances, and H. Geffner. Learning features and abstract actions for computing generalized plans. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 2703–2710, 2019.

[Botvinick and Weinstein 2014] M. Botvinick and A. Weinstein. Model-based hierarchical reinforcement learning and human action control. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 369(1655):20130480, 2014.

[Boutilier *et al.* 1995] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, volume 14, pages 1104–1113, 1995.

[Boutilier *et al.* 2000] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

[Caron *et al.* 2018] M. Caron, P. Bojanowski, A. Joulin, and M. Douze. Deep clustering for unsupervised learning of visual features. In *Proceedings of the European Conference on Computer Vision*, pages 132–149, 2018.

[Chapman and Kaelbling 1991] D. Chapman and L. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, volume 91, pages 726–731, 1991.

[Chemero 2003] A. Chemero. An outline of a theory of affordances. *Ecological Psychology*, 15(2):181–195, 2003.

[Chevalier-Boisvert *et al.* 2018] M. Chevalier-Boisvert, L. Willems, and S. Pal. *Minimalistic Gridworld Environment for OpenAI Gym*. `https://github.com/maximecb/gym-minigrid`, 2018.

[Christen *et al.* 2021] S. Christen, L. Jendele, E. Aksan, and O. Hilliges. Learning functionally decomposed hierarchies for continuous control tasks with path planning. *IEEE Robotics and Automation Letters*, 6(2):3623–3630, 2021.

[Cortes and Vapnik 1995] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[Coulom 2006] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

[Dayan and Hinton 1993] P. Dayan and G. Hinton. Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, 1993.

[de Bruin *et al.* 2018] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška. Integrating state representation learning into deep reinforcement learning. *IEEE Robotics and Automation Letters*, 3(3):1394–1401, 2018.

[Dean and Givan 1997] T. Dean and R. Givan. Model minimization in Markov decision processes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 106–111, 1997.

[Dietterich 2000] T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13(1):227–303, 2000.

[Dijkstra 1959] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[Diuk *et al.* 2008] C. Diuk, A. Cohen, and M. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 240–247, 2008.

[Dubey *et al.* 2018] R. Dubey, P. Agrawal, D. Pathak, T. Griffiths, and A. Efros. Investigating human priors for playing video games. In *Proceedings of the International Conference on Machine Learning*, 2018.

[Ester *et al.* 1996] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *2nd International Conference on Knowledge Discovery and Data Mining*, volume 96, pages 226–231, 1996.

[Eysenbach *et al.* 2019] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations*, 2019.

[Fikes and Nilsson 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[Finn *et al.* 2016] C. Finn, X. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. Deep spatial autoencoders for visuomotor learning. In *Proceedings of the 2016 IEEE International Conference on Robotics and Automation*, pages 512–519, 2016.

[Finn *et al.* 2017] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the International Conference on Machine Learning*, pages 1126–1135, 2017.

[Finney *et al.* 2002] S. Finney, N. Gardiol, L. Kaelbling, and T. Oates. The thing that we tried didn't work very well: Deictic representation in reinforcement learning. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 154–161, 2002.

[Gerevini *et al.* 2008] A. Gerevini, U. Kuter, D. Nau, A. Saetti, and N. Waisbrot. Combining domain-independent planning and HTN planning: The Duet planner. In *European Conference on Artificial Intelligence*, pages 573–577, 2008.

[Gerevini *et al.* 2009] A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[Ghallab *et al.* 2004] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[Givan *et al.* 2003] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1):163–223, 2003.

[Gopalan *et al.* 2017] N. Gopalan, M. Littman, J. MacGlashan, S. Squire, S. Tellex, J. Winder, M. desJardines, and L. Wong. Planning with abstract Markov decision processes. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, 2017.

[Gopalan *et al.* 2020] N. Gopalan, E. Rosen, G. Konidaris, and S. Tellex. Simultaneously learning transferable symbols and language groundings from perceptual data for instruction following. *Robotics: Science and Systems XVI*, 2020.

[Guazzelli *et al.* 1998] A. Guazzelli, M. Bota, F. Corbacho, and M. Arbib. Affordances, motivations, and the world graph theory. *Adaptive Behavior*, 6(3-4):435–471, 1998.

[Guestrin *et al.* 2003] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1003–1010, 2003.

[Guss *et al.* 2019] W. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. Mohanty, D. Liebana, R. Salakhutdinov, N. Topin, et al. The MineRL competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:1904.10079*, 2019.

[Hafner *et al.* 2019] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In *Proceedings of the International Conference on Machine Learning*, pages 2555–2565, 2019.

[Hafner *et al.* 2021] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba. Mastering Atari with discrete world models. In *International Conference on Learning Representations*, 2021.

[Halbritter and Geibel 2007] F. Halbritter and P. Geibel. Learning models of relational MDPs using graph kernels. In *Mexican International Conference on Artificial Intelligence*, pages 409–419. Springer, 2007.

[Hart *et al.* 1968] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hejna *et al.* 2020] D. Hejna, L. Pinto, and P. Abbeel. Hierarchically decoupled imitation for morphological transfer. In *Proceedings of the International Conference on Machine Learning*, pages 4159–4171. PMLR, 2020.

[Helmert 2006] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[Hengst 2002] B. Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the International Conference on Machine Learning*, volume 19, pages 243–250, 2002.

[Higgins *et al.* 2017] I. Higgins, A. Pal, A. Rusu, L. Matthey, C. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner. DARLA: Improving zero-shot transfer in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 1480–1490, 2017.

[Hogg *et al.* 2010] C. Hogg, U. Kuter, and H. Munoz-Avila. Learning methods to generate good plans: Integrating HTN learning and reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 2010.

[Höller *et al.* 2019] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, and R. Alford. HDDL—A language to describe hierarchical planning problems. In *International Workshop on HTN Planning (ICAPS)*, 2019.

[Houthooft *et al.* 2016] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. VIME: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1117–1125, 2016.

[Howard 1960] R. Howard. *Dynamic programming and Markov processes*. MIT Press, 1960.

[James *et al.* 2020] S. James, B. Rosman, and G. Konidaris. Learning to plan with portable symbols. In *Proceedings of the International Conference on Machine Learning*, pages 4682–4691. PMLR, 2020.

[Jetchev *et al.* 2013] N. Jetchev, T. Lang, and M. Toussaint. Learning grounded relational symbols from continuous data for abstract reasoning. In *Proceedings of the 2013 ICRA Workshop on Autonomous Learning*, 2013.

[Jinnai *et al.* 2019] Y. Jinnai, J. Park, D. Abel, and G. Konidaris. Discovering options for exploration by minimizing cover time. In *Proceedings of the International Conference on Machine Learning*, pages 3130–3139. PMLR, 2019.

[Johnson *et al.* 2016] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell. The Malmo platform for artificial intelligence experimentation. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 4246–4247, 2016.

[Jong and Stone 2005] N. Jong and P. Stone. State abstraction discovery from irrelevant state variables. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, volume 8, pages 752–757, 2005.

[Jonschkowski and Brock 2015] R. Jonschkowski and O. Brock. Learning state representations with robotic priors. *Autonomous Robots*, 39(3):407–428, 2015.

[Jonsson and Barto 2006] A. Jonsson and A. Barto. Causal graph based decomposition of factored MDPs. *Journal of Machine Learning Research*, 7(11), 2006.

[Kaelbling and Lozano-Pérez 2011] L. Kaelbling and T. Lozano-Pérez. Hierarchical task and motion planning in the now. In *Proceedings of the 2011 IEEE International Conference on Robotics and Automation*, pages 1470–1477, 2011.

[Kaelbling and Lozano-Pérez 2017] L. Kaelbling and T. Lozano-Pérez. Learning composable models of parameterized skills. In *Proceedings of the 2017 IEEE International Conference on Robotics and Automation*, pages 886–893, 2017.

[Kempka *et al.* 2016] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. Vizdoom: A Doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2016.

[Khardon 1999] R. Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[Kirkpatrick *et al.* 2017] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 2017.

[Kitsak *et al.* 2010] M. Kitsak, L. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. Stanley, and H. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, 2010.

[Kocsis and Szepesvári 2006] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

[Konidaris and Barto 2007] G. Konidaris and A. Barto. Building portable options: skill transfer in reinforcement learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, volume 7, pages 895–900, 2007.

[Konidaris and Barto 2009a] G. Konidaris and A. Barto. Efficient skill learning using abstraction selection. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009.

[Konidaris and Barto 2009b] G. Konidaris and A. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in Neural Information Processing Systems*, 22:1015–1023, 2009.

[Konidaris *et al.* 2012] G. Konidaris, I. Scheidwasser, and A. Barto. Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research*, 13(May):1333–1371, 2012.

[Konidaris *et al.* 2015] G. Konidaris, L. Kaelbling, and T. Lozano-Pérez. Symbol acquisition for probabilistic high-level planning. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, pages 3619–3627, 2015.

[Konidaris *et al.* 2018] G. Konidaris, L. Kaelbling, and T. Lozano-Pérez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61(January):215–289, 2018.

[Konidaris 2016] G. Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, volume 2016, pages 1648–1654, 2016.

[Konidaris 2019] G. Konidaris. On the necessity of abstraction. *Current opinion in behavioral sciences*, 29:1–7, 2019.

[Kumar *et al.* 2020] N. Kumar, M. Fishman, N. Danas, M. Littman, S. Tellex, and G. Konidaris. Task scoping: Building goal-specific abstractions for planning in complex domains. *arXiv preprint arXiv:2010.08869*, 2020.

[Lange and Riedmiller 2010] S. Lange and M. Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *International Joint Conference on Neural Networks*, pages 1–8, 2010.

[Leffler *et al.* 2007] B. Leffler, Mi. Littman, and T. Edmunds. Efficient reinforcement learning with relocatable action models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 7, pages 572–577, 2007.

[Leonard and Durrant-Whyte 1991] J. Leonard and H. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(3):376–382, 1991.

[Lesort *et al.* 2018] T. Lesort, N. Díaz-Rodríguez, J-F. Goudou, and D. Filliat. State representation learning for control: An overview. *Neural Networks*, 108:379–392, 2018.

[Levine *et al.* 2016] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[Levy *et al.* 2019] A. Levy, G. Konidaris, R. Platt, and K. Saenko. Learning multi-level hierarchies with hindsight. In *International Conference on Learning Representations*, 2019.

[Li *et al.* 2006] L. Li, T. Walsh, and M. Littman. Towards a unified theory of state abstraction for MDPs. In *In Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, 2006.

[Long and Fox 2003] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[Machado *et al.* 2017] M. Machado, M. Bellemare, and M. Bowling. A Laplacian framework for option discovery in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 2295–2304. PMLR, 2017.

[Marom and Rosman 2018] O. Marom and B. Rosman. Zero-shot transfer with deictic object-oriented representation in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2297–2305, 2018.

[McCallum and Ballard 1996] A. McCallum and D. Ballard. *Reinforcement learning with selective perception and hidden state*. PhD thesis, University of Rochester, 1996.

[McDermott *et al.* 1998] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. *PDDL—the planning domain definition language*. Technical report, 1998.

[Mehta *et al.* 2008] N. Mehta, S. Ray, P. Tadepalli, and T. Dietterich. Automatic discovery and transfer of MAXQ hierarchies. In *Proceedings of the International Conference on Machine Learning*, pages 648–655, 2008.

[Mnih *et al.* 2015] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[Mourão *et al.* 2010] K. Mourão, R. Petrick, and M. Steedman. Learning action effects in partially observable domains. In *European Conference on Artificial Intelligence*, pages 973–974, 2010.

[Mourão *et al.* 2012] K. Mourão, L. Zettlemoyer, R. Petrick, and M. Steedman. Learning STRIPS operators from noisy and incomplete observations. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 614–623, 2012.

[Mugan and Kuipers 2009] J. Mugan and B. Kuipers. Autonomously learning an action hierarchy using a learned qualitative state representation. In *Proceedings of the 21st International Jont Conference on Artifical intelligence*, pages 1175–1180, 2009.

[Mugan and Kuipers 2011] J. Mugan and B. Kuipers. Autonomous learning of high-level states and actions in continuous environments. *IEEE Transactions on Autonomous Mental Development*, 4(1):70–86, 2011.

[Nachum *et al.* 2018] O. Nachum, S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3303–3313, 2018.

[Nachum *et al.* 2019] O. Nachum, S. Gu, H. Lee, and S. Levine. Near-optimal representation learning for hierarchical reinforcement learning. In *International Conference on Learning Representations*, 2019.

[Nardelli *et al.* 2018] N. Nardelli, G. Synnaeve, Z. Lin, P. Kohli, P. Torr, and N. Usunier. Value propagation networks. In *International Conference on Learning Representations*, 2018.

[Newell and Simon 1976] A. Newell and H. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[Oh *et al.* 2015] J. Oh, X. Guo, H. Lee, R. Lewis, and S. Singh. Action-conditional video prediction using deep networks in Atari games. *Advances in Neural Information Processing Systems*, 2015.

[Pacheck *et al.* 2019] A. Pacheck, G. Konidaris, and H. Kress-Gazit. Automatic encoding and repair of reactive high-level tasks with learned abstract representations. In *Robotics Research: the 18th Annual Symposium*, 2019.

[Parr and Russell 1998] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems*, pages 1043–1049, 1998.

[Parzen 1962] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[Pasula *et al.* 2004] H. Pasula, L. Zettlemoyer, and L. Kaelbling. Learning probabilistic relational planning rules. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 73–81, 2004.

[Pasula *et al.* 2007] H. Pasula, L. Zettlemoyer, and L. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[Pearson 1901] K. Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[Platt 1999] J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in Large Margin Classifiers*, 10(3):61–74, 1999.

[Precup 2000] D. Precup. *Temporal abstraction in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 2000.

[Puterman 2009] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2009.

[Ranchod *et al.* 2015] P. Ranchod, B. Rosman, and G. Konidaris. Nonparametric Bayesian reward segmentation for skill discovery using inverse reinforcement learning. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 471–477. IEEE, 2015.

[Riedmiller 2005] M. Riedmiller. Neural fitted Q iteration: First experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328, 2005.

[Riemer *et al.* 2018] M. Riemer, M. Liu, and G. Tesauro. Learning abstract options. In *Advances in Neural Information Processing Systems*, 2018.

[Rosenblatt 1956] N. Rosenblatt. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, pages 832–837, 1956.

[Rumelhart *et al.* 1986] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[Russell and Norvig 2009] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[Sacerdoti 1974] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Şahin *et al.* 2007] E. Şahin, M. Cakmak, M. Doğar, E. Uğur, and G. Üçoluk. To afford or not to afford: A new formalization of affordances toward affordance-based robot control. *Adaptive Behavior*, 15(4):447–472, 2007.

[Sanner 2010] S. Sanner. *Relational Dynamic Influence Diagram Language (RDDL): Language Description*. Technical report, 2010.

[Saxe *et al.* 2017] A. Saxe, A. Earle, and B. Rosman. Hierarchy through composition with multitask LMDPs. In *Proceedings of the International Conference on Machine Learning*, pages 3017–3026. PMLR, 2017.

[Schrittwieser *et al.* 2020] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[Sherstov and Stone 2005] A. Sherstov and R. Stone. Improving action selection in MDPs via knowledge transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, volume 5, pages 1024–1029, 2005.

[Shivashankar *et al.* 2013] V. Shivashankar, R. Alford, U. Kuter, and D. Nau. The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 2013.

[Silver *et al.* 2016] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Silver *et al.* 2017] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

[Silver *et al.* 2018] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[Şimşek *et al.* 2005] Ö. Şimşek, A. Wolfe, and A. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the International Conference on Machine Learning*, pages 816–823, 2005.

[Snel and Whiteson 2010] M. Snel and S. Whiteson. Multi-task evolutionary shaping without pre-specified representations. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, pages 1031–1038. ACM, 2010.

[Spelke 1990] E. Spelke. Principles of object perception. *Cognitive Science*, 14(1):29–56, 1990.

[Srinivas *et al.* 2018] A. Srinivas, A. Jabri, P. Abbeel, S. Levine, and C. Finn. Universal planning networks: Learning generalizable representations for visuomotor control. In *Proceedings of the International Conference on Machine Learning*, pages 4732–4741, 2018.

[Stolle and Precup 2002] M. Stolle and D. Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[Sutton and Barto 1998] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Sutton *et al.* 1999] R. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[Tate 1977] A. Tate. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[Taylor and Stone 2009] M. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[Todorov 2007] E. Todorov. Linearly-solvable Markov decision problems. In *Advances in Neural Information Processing Systems*, pages 1369–1376, 2007.

[Toyer *et al.* 2018] S. Toyer, F. Trevizan, S. Thiébaux, and L. Xie. Action schema networks: Generalised policies with deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[Ugur and Piater 2015a] E. Ugur and J. Piater. Bottom-up learning of object categories, action effects and logical rules: from continuous manipulative exploration to symbolic planning. In *Proceedings of the 2015 IEEE International Conference on Robotics and Automation*, pages 2627–2633, 2015.

[Ugur and Piater 2015b] E. Ugur and J. Piater. Refining discovered symbols with multi-step interaction experience. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots*, pages 1007–1012. IEEE, 2015.

[Ugur *et al.* 2012] E. Ugur, E. Şahin, and E. Oztop. Self-discovery of motor primitives and learning grasp affordances. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3260–3267. IEEE, 2012.

[Vigorito and Barto 2010] C. Vigorito and A. Barto. Intrinsically motivated hierarchical skill learning in structured environments. *IEEE Transactions on Autonomous Mental Development*, 2(2):132–143, 2010.

[Wang *et al.* 2020] Z. Wang, C. Garrett, L. Kaelbling, and T. Lozano-Pérez. Learning compositional models of robot skills for task and motion planning. *arXiv preprint arXiv:2006.06444*, 2020.

[Wilkins and desJardines 2001] D. Wilkins and M. desJardines. A call for knowledge-based planning. *AI magazine*, 22(1):99–99, 2001.

[Wolfe *et al.* 2010] J. Wolfe, B. Marthi, and S. Russell. Combined task and motion planning for mobile manipulation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, 2010.

[Wong *et al.* 2016] S. Wong, A. Gatt, V. Stamatescu, and M. McDonnell. Understanding data augmentation for classification: When to warp? In *International Conference on Digital Image Computing: Techniques and Applications*, 2016.

[Wookey and Konidaris 2015] D. Wookey and G. Konidaris. Regularized feature selection in reinforcement learning. *Machine Learning*, 100(2):655–676, 2015.

[Younes and Littman 2004] H. Younes and M. Littman. *PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects*. Technical report, 2004.

[Yuan *et al.* 2021] W. Yuan, C. Paxton, K. Desingh, and D. Fox. SORNet: Spatial object-centric representations for sequential manipulation. In *Conference on Robot Learning*, 2021.

[Zettlemoyer *et al.* 2005] L. Zettlemoyer, H. Pasula, and L. Kaelbling. Learning planning rules in noisy stochastic worlds. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 911–918, 2005.

[Zhang *et al.* 2016] J. Zhang, D. Chen, Q. Dong, and Z. Zhao. Identifying a set of influential spreaders in complex networks. *Scientific Reports*, 6, 2016.

[Zhang *et al.* 2018] A. Zhang, A. Lerer, S. Sukhbaatar, R. Fergus, and A. Szlam. Composable planning with attributes. In *Proceedings of the International Conference on Machine Learning*, pages 5842–5851, 2018.

[Zhuo *et al.* 2009] H. Zhuo, D. Hu, C. Hogg, Q. Yang, and H. Munoz-Avila. Learning HTN method preconditions and action models from partial observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1804–1810, 2009.

Appendix A

Agent-Centric Representations

A.1 PPDDL description for the navigation task

The following is the full PDDL description for the toy navigation domain presented in Section 3.4.

```
; Automatically generated ToyDomainV0 domain PPDDL file.
(define (domain ToyDomain)
        (:requirements :strips :probabilistic-effects :conditional-effects :rewards :fluents)
        (:predicates
                (notfailed)
                (wall-junction)
                (window-junction)
                (dead-end)
        )

        (:functions (partition))

        ;Action Inward-partition-0
        (:action Inward_0
         :parameters()
         :precondition (and (dead-end) (notfailed))
         :effect (and (when (= (partition) 6) (and (wall-junction)
                              (not (dead-end)
                                           (decrease (reward) 1.00)
                                           (assign (partition) 5))))
                                   (when (= (partition) 3) (and (wall-junction)
                                          (not (dead-end)
                                                (decrease (reward) 1.00)
                                                (assign (partition) 4))))
                                   (when (= (partition) 1) (and (window-junction)
                                          (not (dead-end)
                                                (decrease (reward) 1.00)
                                                (assign (partition) 2))))
                                   (when (= (partition) 8) (and (window-junction)
                                          (not (dead-end)
                                                (decrease (reward) 1.00)
                                                (assign (partition) 7))))
                         )
        )


        ;Action Outward-partition-0
        (:action Outward_1
         :parameters()
         :precondition (and (wall-junction) (notfailed))
```

```
 :effect (and (when (= (partition) 2) (and (dead-end)
                     (not (wall-junction)
                                    (decrease (reward) 1.00)
                                    (assign (partition) 1))))
                          (when (= (partition) 5) (and (dead-end)
                                 (not (wall-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 6))))
                          (when (= (partition) 4) (and (dead-end)
                                 (not (wall-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 3))))
                          (when (= (partition) 7) (and (dead-end)
                                 (not (wall-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 8))))
                 )
)


;Action Outward-partition-0
(:action Outward_2
 :parameters()
 :precondition (and (window-junction) (notfailed))
 :effect (and (when (= (partition) 2) (and (dead-end)
                     (not (window-junction)
                                    (decrease (reward) 1.00)
                                    (assign (partition) 1))))
                          (when (= (partition) 5) (and (dead-end)
                                 (not (window-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 6))))
                          (when (= (partition) 4) (and (dead-end)
                                 (not (window-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 3))))
                          (when (= (partition) 7) (and (dead-end)
                                 (not (window-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 8))))
                 )
)


;Action Clockwise-partition-0
(:action Clockwise_3
 :parameters()
 :precondition (and (wall-junction) (notfailed))
 :effect (and (when (= (partition) 4) (and (window-junction)
                     (not (wall-junction)
                                    (decrease (reward) 1.00)
                                    (assign (partition) 2))))
                          (when (= (partition) 5) (and (window-junction)
                                 (not (wall-junction)
                                       (decrease (reward) 1.00)
                                       (assign (partition) 7))))
                 )
)


;Action Clockwise-partition-1
(:action Clockwise_4
 :parameters()
 :precondition (and (window-junction) (notfailed))
 :effect (and (when (= (partition) 7) (and (wall-junction)
                     (not (window-junction)
                                    (decrease (reward) 1.00)
```

```
                                        (assign (partition) 4))))
                                (when (= (partition) 2) (and (wall-junction)
                                       (not (window-junction)
                                             (decrease (reward) 1.00)
                                             (assign (partition) 5))))
                        )
        )


        ;Action Anticlockwise-partition-0
        (:action Anticlockwise_5
         :parameters()
         :precondition (and (window-junction) (notfailed))
         :effect (and (when (= (partition) 7) (and (wall-junction)
                             (not (window-junction)
                                        (decrease (reward) 1.00)
                                        (assign (partition) 5))))
                                (when (= (partition) 2) (and (wall-junction)
                                       (not (window-junction)
                                             (decrease (reward) 1.00)
                                             (assign (partition) 4))))
                        )
        )


        ;Action Anticlockwise-partition-1
        (:action Anticlockwise_6
         :parameters()
         :precondition (and (wall-junction) (notfailed))
         :effect (and (when (= (partition) 4) (and (window-junction)
                             (not (wall-junction)
                                        (decrease (reward) 1.00)
                                        (assign (partition) 7))))
                                (when (= (partition) 5) (and (window-junction)
                                       (not (wall-junction)
                                             (decrease (reward) 1.00)
                                             (assign (partition) 2))))
                        )
        )

)
```

A.2 Hyperparameters for the *Rod-and-Block* domain

Algorithm	**Hyperparameter**	**Value(s)**
DBSCAN	neighbourhood ϵ	0.03
Feature selection (see Figure 2.8)	threshold ϵ	0.02
SVM	C	grid search over $[1, 16)$
	γ	grid search over $[5, 20)$
KDE	bandwidth	grid search over $[0.001, 0.1)$
Linking (see Section 3.5)	neighbourhood ϵ	0.03

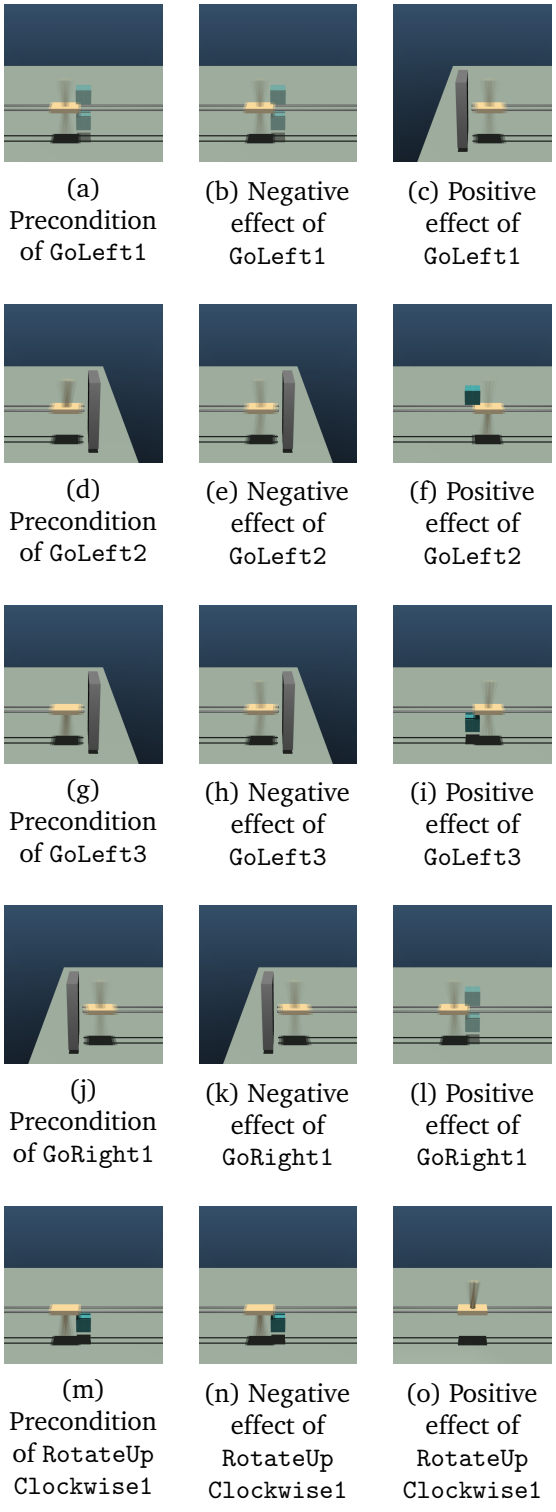
Table A.1: Hyperparameters used for constructing a PPDDL representation of the *Rod-and-Block* domain.

A.3 Hyperparameters for the *Treasure Game* domain

Algorithm	**Hyperparameter**	**Value(s)**
DBSCAN	neighbourhood ϵ	0.05
Feature selection (see Figure 2.8)	threshold ϵ	0.02
SVM	C	grid search over $[2, 16)$
	γ	grid search over $[0.01, 4)$
KDE	bandwidth	grid search over $[0.001, 0.1)$
Linking (see Section 3.5)	neighbourhood ϵ	0.05

Table A.2: Hyperparameters used for constructing a PPDDL representation of the *Treasure Game* domain.

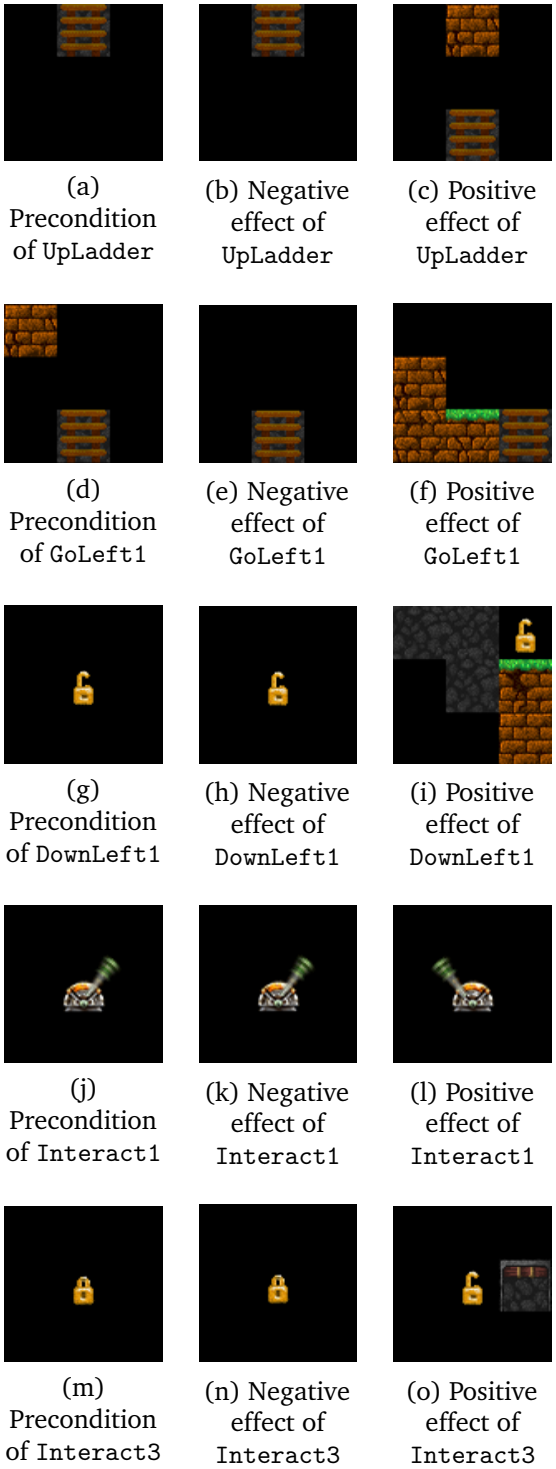
A.4 Learned operators for the *Rod-and-Block* task



(a) Precondition of `GoLeft1`
(b) Negative effect of `GoLeft1`
(c) Positive effect of `GoLeft1`
(d) Precondition of `GoLeft2`
(e) Negative effect of `GoLeft2`
(f) Positive effect of `GoLeft2`
(g) Precondition of `GoLeft3`
(h) Negative effect of `GoLeft3`
(i) Positive effect of `GoLeft3`
(j) Precondition of `GoRight1`
(k) Negative effect of `GoRight1`
(l) Positive effect of `GoRight1`
(m) Precondition of `RotateUp Clockwise1`
(n) Negative effect of `RotateUp Clockwise1`
(o) Positive effect of `RotateUp Clockwise1`

Figure A.1: A subset of symbolic rules learned for a *Rod-and-Block* task.

A.5 Learned operators for the *Treasure Game* task



(a) Precondition of `UpLadder` (b) Negative effect of `UpLadder` (c) Positive effect of `UpLadder`

(d) Precondition of `GoLeft1` (e) Negative effect of `GoLeft1` (f) Positive effect of `GoLeft1`

(g) Precondition of `DownLeft1` (h) Negative effect of `DownLeft1` (i) Positive effect of `DownLeft1`

(j) Precondition of `Interact1` (k) Negative effect of `Interact1` (l) Positive effect of `Interact1`

(m) Precondition of `Interact3` (n) Negative effect of `Interact3` (o) Positive effect of `Interact3`

Figure A.2: A subset of symbolic rules learned in Level 1

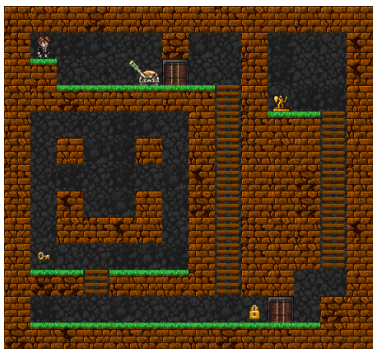
A.6 *Treasure Game* level layouts

(a) Level 1

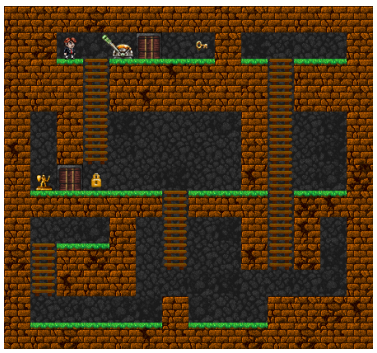
(b) Level 2

(c) Level 3

(d) Level 4



(e) Level 5



(f) Level 6

(g) Level 7

(h) Level 8

(i) Level 9

(j) Level 10

Appendix B

Object-Centric Representations

B.1 Pseudocode

Below we present pseudocode describing our approach to building a typed, object-centric PPDDL representation for an arbitrary domain.

procedure LEARNREPRESENTATION
  Given: T state-option transitions $Dataset = \{(d_i, s_i, o_i, d_i', s_i') \mid 0 \leq i \leq T\}$, set of objects $\mathcal{M}$
  ▷ Partition options into subgoal options
  $SubgoalOptions \leftarrow \emptyset$
  for each $o \in \mathcal{O}$ **do**
    $I \leftarrow \{d \mid (d, \cdot, o, \cdot, \cdot) \in Dataset\}$ ▷ Set of initial states for option o
    $\beta \leftarrow \{d' \mid (\cdot, \cdot, o, d', \cdot) \in Dataset\}$ ▷ Set of terminating states for option o
    for all $K \subseteq I$ such that $\Pr(d' \mid d_i, o) = \Pr(d' \mid d_j, o) \forall d_i, d_j \in I, d' \in \beta$ **do**
      $P \leftarrow \{o, K, \{d' \mid \forall d \in K, (d, \cdot, o, d', \cdot) \in Dataset\}\}$ ▷ Start and end states for a partition
      $SubgoalOptions \leftarrow SubgoalOptions \cup \{P\}$
    end for
  end for
  ▷ Estimate preconditions and effects
  $Preconditions, Effects \leftarrow \emptyset$
  for each $\{\cdot, start, end\} \in SubgoalOptions$ **do**
    $mask \leftarrow$ COMPUTEMASK$(start, end)$ ▷ List the objects that change state
    $negative \leftarrow \mathcal{S} \setminus start$
    $features \leftarrow$ FEATURESELECTION$(mask, start, negative)$
    $classifier \leftarrow$ FITCLASSIFIER$(start, negative, features)$
    $Preconditions \leftarrow Preconditions \cup \{classifier\}$
    $estimator \leftarrow$ FITESTIMATOR$(mask, end)$ ▷ Fit over only objects that change
    $Effects \leftarrow Effects \cup \{estimator\}$
  end for

▷ Build propositional PPDDL
$Operators, Propositions \leftarrow \emptyset$
for each $precondition, effect \in (Preconditions \times Effects)$ **do**
$op, symbols \leftarrow$ BUILDPPDDLOPERATOR$(precondition, effect, Effects)$
$Operators \leftarrow Operators \cup \{op\}$
$Propositions \leftarrow Propositions \cup symbols$
end for
▷ Infer object types
$EffProfile \leftarrow \emptyset$
for each object m **do**
for each $o \in \mathcal{O}$ **do**
$EffProfile(m, o) \leftarrow$ COMPUTEEFFECTS$(m, o, Operators)$
end for
end for
$Types \leftarrow \{K \mid EffProfile(m_i, o) \approx EffProfile(m_j, o) \forall o \in \mathcal{O}, m_i, m_j \in K, K \subseteq \mathcal{M}\}$
▷ Generate typed PPDDL
$TypedOperators, Predicates \leftarrow \emptyset$
for each $type \in Types$ **do**
▷ Replace propositions and operators over objects of same type with lifted versions
$ops, predicates \leftarrow$ MERGE$(\mathcal{M}, type, Operators, Propositions)$
$TypedOperators \leftarrow TypedOperators \cup ops$
$Predicates \leftarrow Predicates \cup predicates$
end for
▷ Instantiate typed PPDDL in new task
for each $\{o, start, end\} \in SubgoalOptions$ **do**
▷ Get problem-specific start states
$I_{\mathcal{S}} \leftarrow \{s \mid \forall d \in start, d' \in end, (d, s, o, d', \cdot) \in Dataset\}$
▷ Get problem-specific end states
$\beta_{\mathcal{S}} \leftarrow \{s' \mid \forall d \in start, d' \in end, s \in I_{\mathcal{S}}, (d, s, o, d', s') \in Dataset\}$
▷ For each partition label
for all $\kappa \subseteq I_{\mathcal{S}}$ such that $\Pr(s' \mid s_i, o) = \Pr(s' \mid s_j, o) \forall s_i, s_j \in I_{\mathcal{S}}, s' \in \beta_{\mathcal{S}}$ **do**
▷ Get end label
$\lambda \leftarrow \{s' \mid s \in start, d' \in end, s \in \kappa, (d, s, o, d', s') \in Dataset\}$
$Predicates \leftarrow Predicates \cup \{\kappa\} \cup \{\lambda\}$ ▷ Add problem-specific symbols
$mask \leftarrow$ COMPUTEMASK$(start, end)$ ▷ Computes the affected objects
▷ Link problem-specific symbols in precondition and effect to the affected objects
$TypedOperators \leftarrow$ GROUND$(TypedOperators, \kappa, \lambda, mask)$
end for
end for
return $TypedOperators, Predicates$
end procedure

B.2 Propositional PDDL description for the Blocks World task

Below is the automatically generated propositional PDDL description of the Blocks World domain with 3 blocks. In practice, the agent generates this description with arbitrary names for the propositions, but for readability purposes we have manually renamed them to match their semantics.

```
(define (domain BlocksWorld)
        (:requirements :strips)
        (:predicates (notfailed)
                (AInHand)
                (HandFull)
                (COnBlock)
                (BInHand)
                (COnTable)
                (AOnTable)
                (BOnBlock)
                (AOnBlock)
                (BOnTable)
                (CInHand)
                (HandEmpty)
                (BOnTable_BCovered)
                (COnBlock_CCovered)
                (AOnBlock_ACovered)
                (BOnBlock_BCovered)
                (COnTable_CCovered)
                (AOnTable_ACovered)
        )

        (:action Pick_0
         :parameters()
         :precondition (and (HandEmpty) (AOnTable) (notfailed))
         :effect (and (AInHand) (HandFull) (not AOnTable) (not HandEmpty) (not AOnTable)
                 (not HandEmpty))
        )

        (:action Pick_1
         :parameters()
         :precondition (and (HandEmpty) (AOnBlock) (COnBlock_CCovered) (notfailed))
         :effect (and (COnBlock) (AInHand) (HandFull) (not AOnBlock) (not HandEmpty)
                 (not COnBlock_CCovered) (not AOnBlock) (not HandEmpty)
                 (not COnBlock_CCovered))
        )

        (:action Pick_2
         :parameters()
         :precondition (and (HandEmpty) (COnTable_CCovered) (BOnBlock) (notfailed))
         :effect (and (BInHand) (COnTable) (HandFull) (not BOnBlock) (not HandEmpty)
                 (not COnTable_CCovered) (not BOnBlock) (not HandEmpty)
                 (not COnTable_CCovered))
        )

        (:action Pick_3
         :parameters()
         :precondition (and (HandEmpty) (AOnTable_ACovered) (BOnBlock) (notfailed))
         :effect (and (BInHand) (AOnTable) (HandFull) (not BOnBlock) (not HandEmpty)
                 (not AOnTable_ACovered) (not BOnBlock) (not HandEmpty)
                 (not AOnTable_ACovered))
        )

        (:action Pick_4
         :parameters()
```

```
 :precondition (and (HandEmpty) (AOnBlock) (BOnBlock_BCovered) (notfailed))
 :effect (and (BOnBlock) (AInHand) (HandFull) (not AOnBlock) (not HandEmpty)
         (not BOnBlock_BCovered) (not AOnBlock) (not HandEmpty)
         (not BOnBlock_BCovered))
)

(:action Pick_5
 :parameters()
 :precondition (and (HandEmpty) (AOnBlock_ACovered) (BOnBlock) (notfailed))
 :effect (and (BInHand) (AOnBlock) (HandFull) (not BOnBlock) (not HandEmpty)
         (not AOnBlock_ACovered) (not BOnBlock) (not HandEmpty)
         (not AOnBlock_ACovered))
)

(:action Pick_6
 :parameters()
 :precondition (and (HandEmpty) (AOnBlock) (BOnTable_BCovered) (notfailed))
 :effect (and (BOnTable) (AInHand) (HandFull) (not AOnBlock) (not HandEmpty)
         (not BOnTable_BCovered) (not AOnBlock) (not HandEmpty)
         (not BOnTable_BCovered))
)

(:action Pick_7
 :parameters()
 :precondition (and (HandEmpty) (BOnTable) (notfailed))
 :effect (and (BInHand) (HandFull) (not BOnTable) (not HandEmpty) (not BOnTable)
         (not HandEmpty))
)

(:action Pick_8
 :parameters()
 :precondition (and (HandEmpty) (COnTable) (notfailed))
 :effect (and (CInHand) (HandFull) (not COnTable) (not HandEmpty) (not COnTable)
         (not HandEmpty))
)

(:action Pick_9
 :parameters()
 :precondition (and (HandEmpty) (AOnTable_ACovered) (COnBlock) (notfailed))
 :effect (and (CInHand) (AOnTable) (HandFull) (not COnBlock) (not HandEmpty)
         (not AOnTable_ACovered) (not COnBlock) (not HandEmpty)
         (not AOnTable_ACovered))
)

(:action Pick_10
 :parameters()
 :precondition (and (HandEmpty) (AOnBlock) (COnTable_CCovered) (notfailed))
 :effect (and (COnTable) (AInHand) (HandFull) (not AOnBlock) (not HandEmpty)
         (not COnTable_CCovered) (not AOnBlock) (not HandEmpty)
         (not COnTable_CCovered))
)

(:action Pick_11
 :parameters()
 :precondition (and (HandEmpty) (AOnBlock_ACovered) (COnBlock) (notfailed))
 :effect (and (CInHand) (AOnBlock) (HandFull) (not COnBlock) (not HandEmpty)
         (not AOnBlock_ACovered) (not COnBlock) (not HandEmpty)
         (not AOnBlock_ACovered))
)

(:action Pick_12
 :parameters()
 :precondition (and (HandEmpty) (COnBlock) (BOnTable_BCovered) (notfailed))
 :effect (and (BOnTable) (CInHand) (HandFull) (not COnBlock) (not HandEmpty)
         (not BOnTable_BCovered) (not COnBlock) (not HandEmpty)
         (not BOnTable_BCovered))
)
```

```
(:action Pick_13
 :parameters()
 :precondition (and (HandEmpty) (COnBlock_CCovered) (BOnBlock) (notfailed))
 :effect (and (BInHand) (COnBlock) (HandFull) (not BOnBlock) (not HandEmpty)
         (not COnBlock_CCovered) (not BOnBlock) (not HandEmpty)
         (not COnBlock_CCovered))
)

(:action Pick_14
 :parameters()
 :precondition (and (HandEmpty) (COnBlock) (BOnBlock_BCovered) (notfailed))
 :effect (and (BOnBlock) (CInHand) (HandFull) (not COnBlock) (not HandEmpty)
         (not BOnBlock_BCovered) (not COnBlock) (not HandEmpty)
         (not BOnBlock_BCovered))
)

(:action Put_15
 :parameters()
 :precondition (and (HandFull) (AInHand) (notfailed))
 :effect (and (AOnTable) (HandEmpty) (not AInHand) (not HandFull) (not AInHand)
         (not HandFull))
)

(:action Put_16
 :parameters()
 :precondition (and (HandFull) (BInHand) (notfailed))
 :effect (and (BOnTable) (HandEmpty) (not HandFull) (not BInHand) (not HandFull)
         (not BInHand))
)

(:action Put_17
 :parameters()
 :precondition (and (HandFull) (CInHand) (notfailed))
 :effect (and (COnTable) (HandEmpty) (not HandFull) (not CInHand) (not HandFull)
         (not CInHand))
)

(:action Stack_18
 :parameters()
 :precondition (and (HandFull) (CInHand) (BOnTable) (notfailed))
 :effect (and (BOnTable_BCovered) (COnBlock) (HandEmpty) (not HandFull)
         (not BOnTable) (not CInHand) (not HandFull) (not BOnTable)
         (not CInHand))
)

(:action Stack_19
 :parameters()
 :precondition (and (HandFull) (COnBlock) (BInHand) (notfailed))
 :effect (and (BOnBlock) (COnBlock_CCovered) (HandEmpty) (not HandFull)
         (not COnBlock) (not BInHand) (not HandFull) (not COnBlock)
         (not BInHand))
)

(:action Stack_20
 :parameters()
 :precondition (and (HandFull) (AOnBlock) (BInHand) (notfailed))
 :effect (and (BOnBlock) (AOnBlock_ACovered) (HandEmpty) (not HandFull)
         (not BInHand) (not AOnBlock) (not HandFull) (not BInHand)
         (not AOnBlock))
)

(:action Stack_21
 :parameters()
 :precondition (and (HandFull) (AInHand) (BOnTable) (notfailed))
 :effect (and (BOnTable_BCovered) (AOnBlock) (HandEmpty) (not AInHand)
         (not HandFull) (not BOnTable) (not AInHand) (not HandFull)
```

```
                (not BOnTable))
        )

        (:action Stack_22
         :parameters()
         :precondition (and (HandFull) (CInHand) (BOnBlock) (notfailed))
         :effect (and (BOnBlock_BCovered) (COnBlock) (HandEmpty) (not HandFull)
                (not BOnBlock) (not CInHand) (not HandFull) (not BOnBlock)
                (not CInHand))
        )

        (:action Stack_23
         :parameters()
         :precondition (and (HandFull) (COnTable) (BInHand) (notfailed))
         :effect (and (BOnBlock) (COnTable_CCovered) (HandEmpty) (not HandFull)
                (not BInHand) (not COnTable) (not HandFull) (not BInHand)
                (not COnTable))
        )

        (:action Stack_24
         :parameters()
         :precondition (and (HandFull) (AInHand) (COnBlock) (notfailed))
         :effect (and (COnBlock_CCovered) (AOnBlock) (HandEmpty) (not AInHand)
                (not HandFull) (not COnBlock) (not AInHand) (not HandFull)
                (not COnBlock))
        )

        (:action Stack_25
         :parameters()
         :precondition (and (HandFull) (AOnTable) (CInHand) (notfailed))
         :effect (and (COnBlock) (AOnTable_ACovered) (HandEmpty) (not HandFull)
                (not AOnTable) (not CInHand) (not HandFull) (not AOnTable)
                (not CInHand))
        )

        (:action Stack_26
         :parameters()
         :precondition (and (HandFull) (AInHand) (COnTable) (notfailed))
         :effect (and (COnTable_CCovered) (AOnBlock) (HandEmpty) (not AInHand)
                (not HandFull) (not COnTable) (not AInHand) (not HandFull)
                (not COnTable))
        )

        (:action Stack_27
         :parameters()
         :precondition (and (HandFull) (AOnBlock) (CInHand) (notfailed))
         :effect (and (COnBlock) (AOnBlock_ACovered) (HandEmpty) (not HandFull)
                (not AOnBlock) (not CInHand) (not HandFull) (not AOnBlock)
                (not CInHand))
        )

        (:action Stack_28
         :parameters()
         :precondition (and (HandFull) (AOnTable) (BInHand) (notfailed))
         :effect (and (BOnBlock) (AOnTable_ACovered) (HandEmpty) (not HandFull)
                (not BInHand) (not AOnTable) (not HandFull) (not BInHand)
                (not AOnTable))
        )

        (:action Stack_29
         :parameters()
         :precondition (and (HandFull) (AInHand) (BOnBlock) (notfailed))
         :effect (and (BOnBlock_BCovered) (AOnBlock) (HandEmpty) (not AInHand)
                (not HandFull) (not BOnBlock) (not AInHand) (not HandFull)
                (not BOnBlock))
        )
)
```

B.3 Lifted PDDL description for the Blocks World task

In contrast, the lifted representation learned below is far more compact. Here, operators are parameterised by objects, which allows for better generalisation across instances with varying numbers of blocks.

Below we provide the learned representation for the domain. Again, we manually rename the predicates and types to help with readability.

```
(define (domain BlocksWorld)
        (:requirements :strips :typing)
        (:types hand block)
        (:predicates
                (BlockInHand ?w - block)
                (HandFull ?w - hand)
                (BlockOnBlock ?w - block)
                (BlockOnTable ?w - block)
                (HandEmpty ?w - hand)
                (BlockOnTable_BlockCovered ?w - block)
                (BlockOnBlock_BlockCovered ?w - block)
                (notfailed)
        )
        (:action Pick-partition-0
         :parameters (?w - hand ?x - block)
         :precondition (and (notfailed) (HandEmpty ?w) (BlockOnTable ?x))
         :effect (and (BlockInHand ?x) (HandFull ?w) (not (BlockOnTable ?x))
                 (not (HandEmpty ?w)))
        )

        (:action Pick-partition-1
         :parameters (?w - hand ?x - block ?y - block)
         :precondition (and (notfailed) (HandEmpty ?w) (BlockOnBlock ?x)
                       (BlockOnBlock_BlockCovered ?y))
         :effect (and (BlockOnBlock ?y) (BlockInHand ?x) (HandFull ?w)
                 (not (BlockOnBlock ?x)) (not (HandEmpty ?w))
                 (not (BlockOnBlock_BlockCovered ?y)))
        )

        (:action Pick-partition-10
         :parameters (?w - hand ?x - block ?y - block)
         :precondition (and (notfailed) (HandEmpty ?w) (BlockOnTable_BlockCovered ?x)
                       (BlockOnBlock ?y))
         :effect (and (BlockInHand ?y) (BlockOnTable ?x) (HandFull ?w)
                 (not (BlockOnBlock ?y)) (not (HandEmpty ?w))
                 (not (BlockOnTable_BlockCovered ?x)))
        )

        (:action Put-partition-0
         :parameters (?w - hand ?x - block)
         :precondition (and (notfailed) (HandFull ?w) (BlockInHand ?x))
         :effect (and (BlockOnTable ?x) (HandEmpty ?w) (not (BlockInHand ?x))
                 (not (HandFull ?w)))
        )

        (:action Stack-partition-0
         :parameters (?w - hand ?x - block ?y - block)
         :precondition (and (notfailed) (HandFull ?w) (BlockInHand ?x) (BlockOnTable ?y))
         :effect (and (BlockOnTable_BlockCovered ?y) (BlockOnBlock ?x) (HandEmpty ?w)
                 (not (HandFull ?w)) (not (BlockOnTable ?y))
                 (not (BlockInHand ?x)))
```

```
        )

        (:action Stack-partition-1
         :parameters (?w - hand ?x - block ?y - block)
         :precondition (and (notfailed) (HandFull ?w) (BlockOnBlock ?x) (BlockInHand ?y))
         :effect (and (BlockOnBlock ?y) (BlockOnBlock_BlockCovered ?x) (HandEmpty ?w)
                 (not (HandFull ?w)) (not (BlockOnBlock ?x))
                 (not (BlockInHand ?y)))
        )

)
```

A task might then be specified as follows:

```
(define (problem stack)
   (:domain BlocksWorld)

   (:objects hand - Hand
           A B C - Block
   )
   (:init (BlockOnTable A)
          (BlockOnTable B)
          (BlockOnTable C)
          (HandEmpty hand)
          (notfailed)
    )
   (:goal (and (BlockOnBlock A)
               (BlockOnBlock_BlockCovered C)
               (BlockOnTable_BlockCovered B)))
)
```

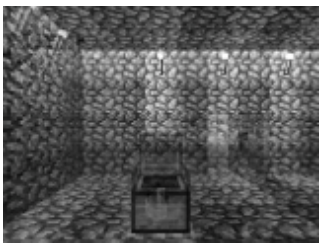
B.4 Hyperparameters for the *Minecraft* domain

Algorithm	Hyperparameter	Value(s)
DBSCAN	neighbourhood ϵ	0.98
SVM	C	grid search over logspace $[0.01, 0.5)$
	γ	grid search over logspace $[0.01, 1)$
KDE	bandwidth	grid search over $[0.001, 0.1)$
Distribution similarity	KL threshold	10000
Problem-specific linking	neighbourhood ϵ	0.5

Table B.1: Hyperparameters used for constructing a PPDDL representation of the *Treasure Game* domain.

B.5 Visualising operators for Minecraft

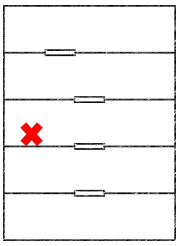
Here we illustrate some learned operators for the Minecraft task. To see all predicates and operators, please see the following URL: `https://sites.google.com/view/mine-pddl`.



(a) `symbol_13`

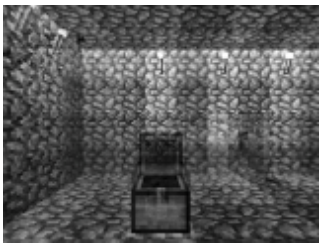
(b) `symbol_4`

(c) `symbol_55`



(d) `psymbol_8`

(e) `symbol_58`

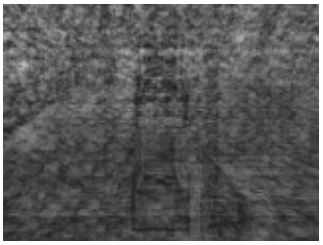


(f) `symbol_59`

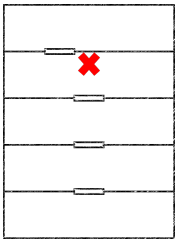
```
(:action Open-Chest-partition-0
:parameters (?w - type0 ?x - type6
             ?y - type9)
:precondition (and (notfailed) (symbol_13 ?w) (symbol_4 ?x)
                   (symbol_55 ?y) (psymbol_8))
:effect (and (symbol_58 ?x) (symbol_59 ?w) (not (symbol_4 ?x))
        (not (symbol_13 ?w)))
)
```

(g) A learned typed PDDL operator for the `Open-Chest` skill. The predicate underlined in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

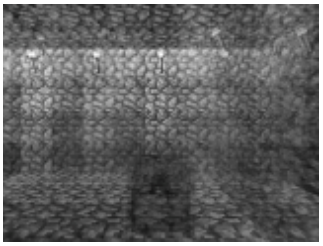
Figure B.1: Our approach learns that, in order to open a chest, the agent must be standing in front of a chest (`symbol_13`), the chest must be closed (`symbol_4`), the inventory must contain a clock (`symbol_55`) and the agent must be standing at a certain location (`psymbol_8`). The result is that the agent finds itself in front of an open chest (`symbol_58`) and the chest is open (`symbol_59`). `type0` refers to the "agent" type, `type6` the "chest" type and `type9` the "inventory" type.



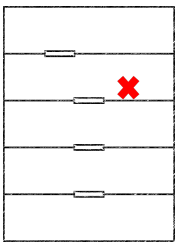
(a) symbol_46



(b) psymbol_0



(c) symbol_11



(d) psymbol_1

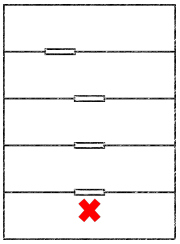
```
(:action Walk-to-partition-0-2a
 :parameters (?w - type0)
 :precondition (and (notfailed) (symbol_46 ?w)
                (psymbol_0))
 :effect (and (symbol_11 ?w) (psymbol_1)
         (not (symbol_46 ?w)) (not (psymbol_0)))
)
```

(e) Typed PDDL operator for a partition of the Walk-To option. The predicate underlined in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

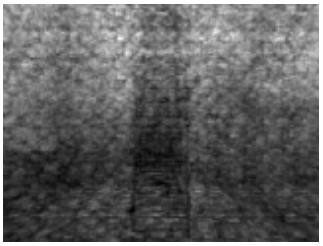
Figure B.2: Abstract operator that models the agent walking to the crafting table. In order to do so, the agent must be standing in the middle of a room (symbol_46) at a particular location (psymbol_0). As a result, the agent finds itself in front of the crafting table (symbol_1) at a particular location (psymbol_1).

(a) symbol_38

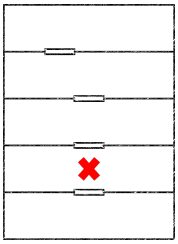
(b) symbol_64



(c) psymbol_24



(d) symbol_50

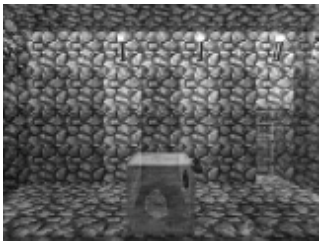


(e) psymbol_12

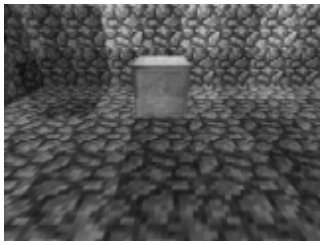
```
(:action Through-Door-partition-3-207a
 :parameters (?w - type0 ?x - type1)
 :precondition (and (notfailed) (symbol_38 ?w) (symbol_64 ?x)
                    (= (id ?x) 1) (psymbol_24))
 :effect (and (symbol_50 ?w) (not (symbol_38 ?w))
              (psymbol_12) (not (psymbol_24)))
)
```

(f) Typed PDDL operator for a partition of the Through-Door option. The predicate underlined in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

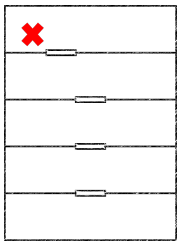
Figure B.3: Abstract operator that models the agent walking through a door. In order to do so, the agent must be standing in front of an open door (symbol_38) at a particular location (psymbol_24), and the door must be open (symbol_64). As a result, the agent finds itself in the middle of a room (symbol_50) at a particular location (psymbol_12).



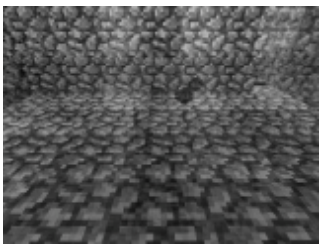
(a) symbol_15



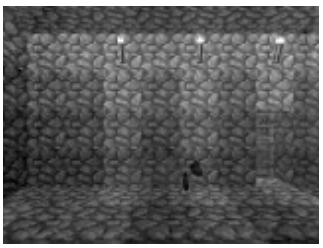
(b) symbol_2



(c) psymbol_17



(d) symbol_19



(e) symbol_20

```
(:action Attack-partition-0-76a
 :parameters (?w - type0 ?x - type7)
 :precondition (and (notfailed) (symbol_15 ?w) (symbol_2 ?x)
                    (psymbol_17))
 :effect (and (symbol_19 ?x) (symbol_20 ?w) (not (symbol_2 ?x))
              (not (symbol_15 ?w)))
)
```

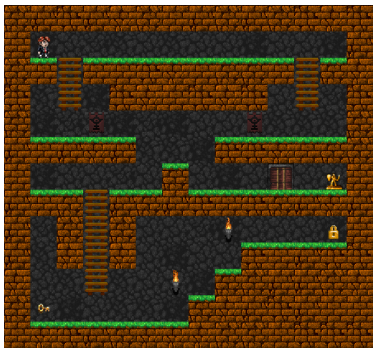
(f) Typed PDDL operator for a partition of the Attack option. The predicate underlined in red indicates a problem-specific symbol that must be relearned for each new task, while the rest of the operator can be safely transferred.

Figure B.4: Abstract operator that models the agent attacking an object. In order to do so, the agent must be standing in front of a gold block (symbol_15) at a particular location (psymbol_17), and the gold block must be whole (symbol_2). As a result, the agent finds itself in front of a disintegrated block (symbol_20), and the gold block is disintegrated (symbol_19).

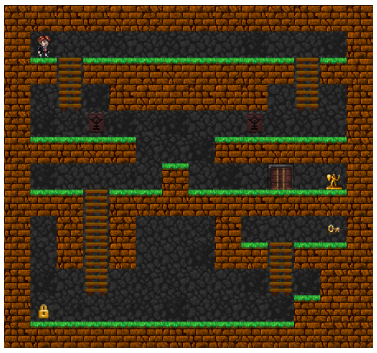
Appendix C

Portable Hierarchies

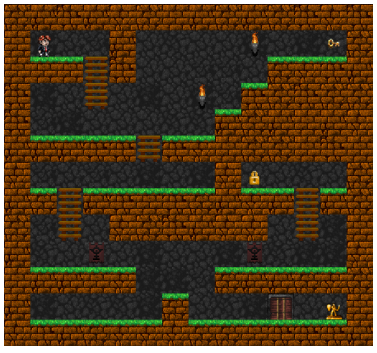
C.1 *Treasure Game* level layouts



(a) Level 1

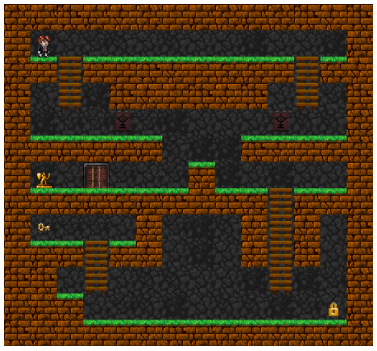


(b) Level 2

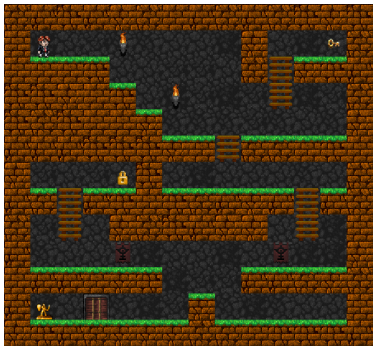


(c) Level 3

(d) Level 4



(e) Level 5

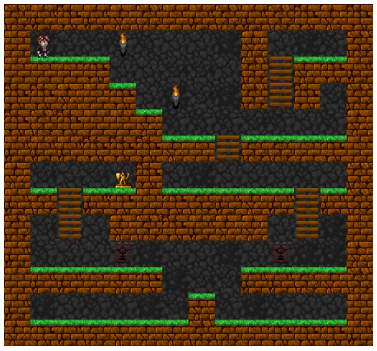


(f) Level 6

(g) Level 7

(h) Level 8

(i) Level 9



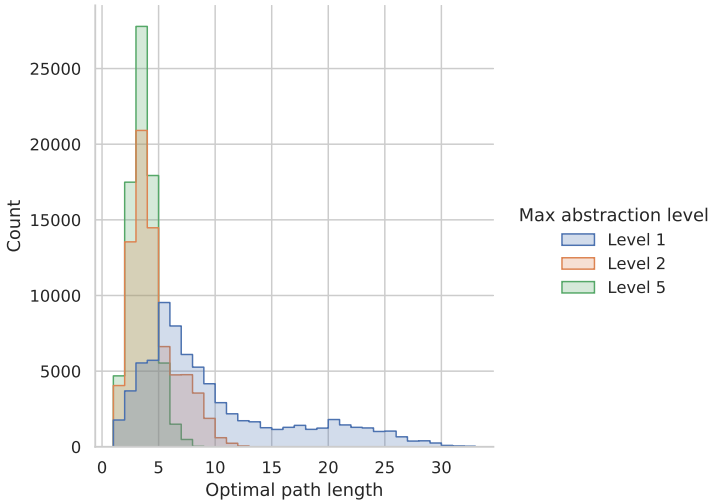
(j) Level 10

C.2 Hyperparameters for the *Treasure Game* domain

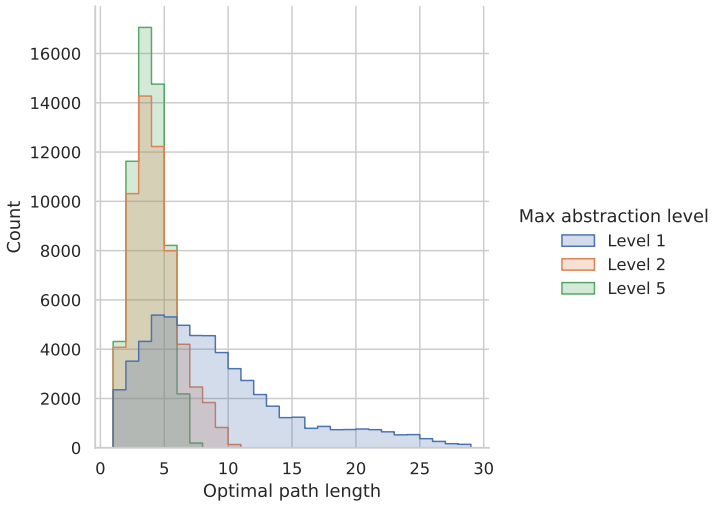
Algorithm	Hyperparameter	Value(s)
DBSCAN	neighbourhood ϵ	4
Feature selection	threshold ϵ	0.02
SVM	C	grid search over $[0.01, 0.5)$
	γ	grid search over $[0.01, 1)$
KDE	bandwidth	grid search over $[0.001, 0.1)$
Problem-specific linking	neighbourhood ϵ	0.1
Higher-order skill acquisition (see Figure 5.15)	maximum option length L	4
	graph size reduction N	3
	importance metric $\mathscr{I}$	VOTERANK

Table C.1: Hyperparameters used for constructing a hierarchical representation of the *Treasure Game* domain.

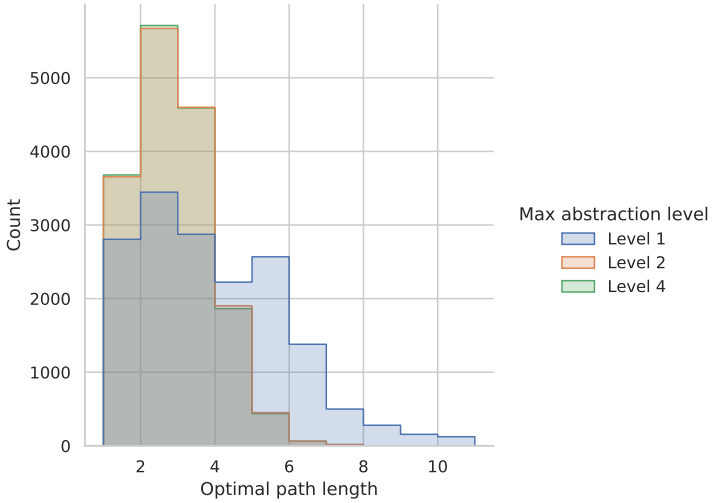
C.3 Distribution over shortest path lengths



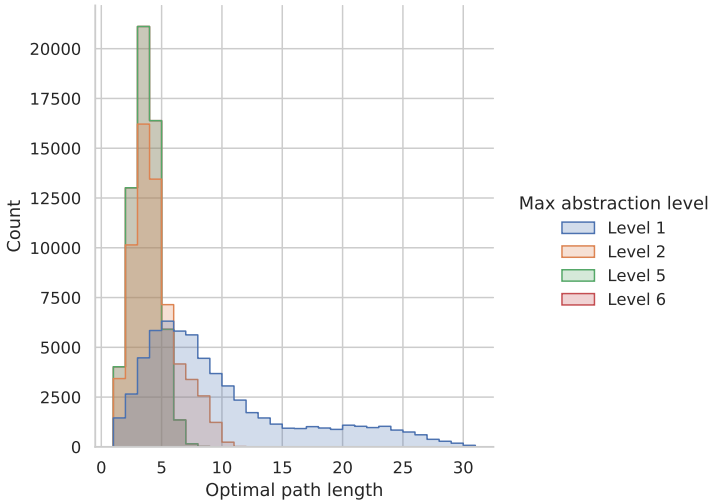
(a) Distribution of optimal plan lengths in Level 1 when using hierarchies of varying heights.



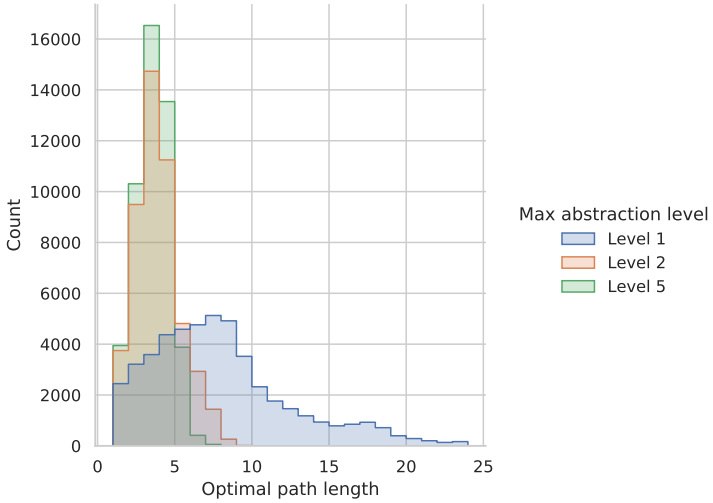
(b) Distribution of optimal plan lengths in Level 2 when using hierarchies of varying heights.



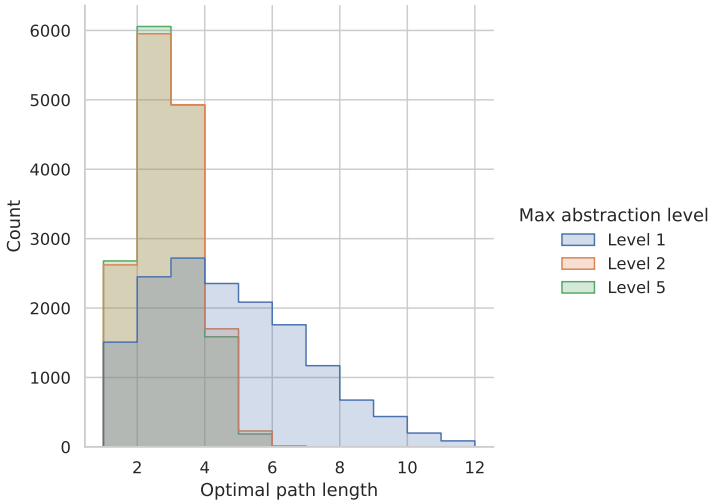
(c) Distribution of optimal plan lengths in Level 3 when using hierarchies of varying heights.



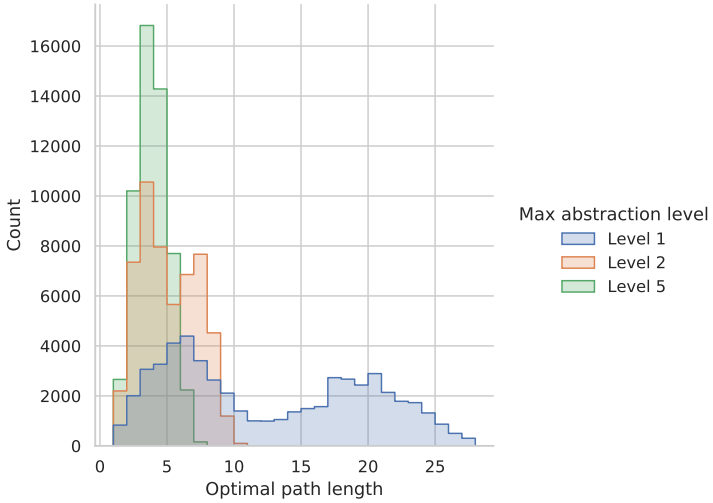
(d) Distribution of optimal plan lengths in Level 4 when using hierarchies of varying heights.



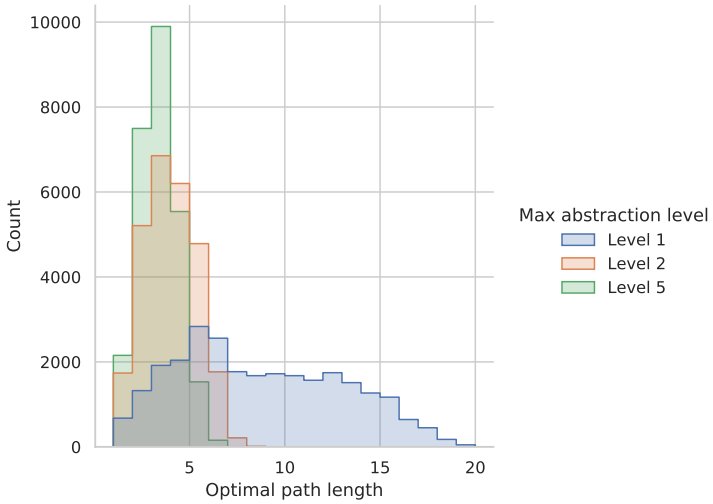
(e) Distribution of optimal plan lengths in Level 5 when using hierarchies of varying heights.



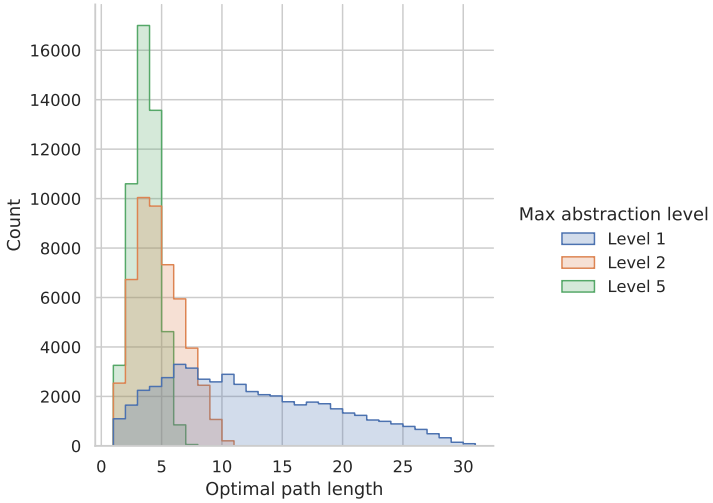
(f) Distribution of optimal plan lengths in Level 6 when using hierarchies of varying heights.



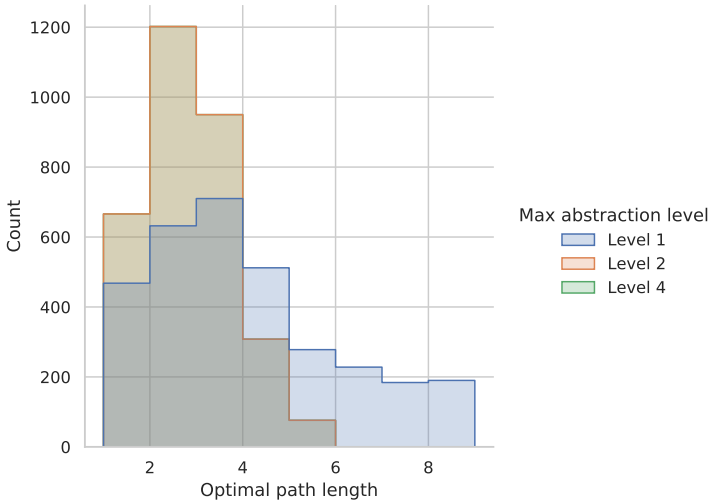
(g) Distribution of optimal plan lengths in Level 7 when using hierarchies of varying heights.



(h) Distribution of optimal plan lengths in Level 8 when using hierarchies of varying heights.



(i) Distribution of optimal plan lengths in Level 9 when using hierarchies of varying heights.



(j) Distribution of optimal plan lengths in Level 10 when using hierarchies of varying heights.