Reinforcement Learning with Option Machines

Floris den Hengst[1,2*] , **Vincent François-Lavet**[2] , **Mark Hoogendoorn**[2] , **Frank van Harmelen**[2]

[1]ING Bank N.V.
[2]Vrije Universiteit Amsterdam

Floris.den.Hengst@ing.com, {Vincent.FrancoisLavet, M.Hoogendoorn, Frank.van.Harmelen}@vu.nl

Abstract

Reinforcement learning (RL) is a powerful framework for learning complex behaviors, but lacks adoption in many settings due to sample size requirements. We introduce a framework for increasing sample efficiency of RL algorithms. Our approach focuses on optimizing environment rewards with high-level instructions. These are modeled as a high-level controller over temporally extended actions known as *options*. These options can be looped, interleaved and partially ordered with a rich language for high-level instructions. Crucially, the instructions may be *underspecified* in the sense that following them does not guarantee high reward in the environment. We present an algorithm for control with these so-called *option machines* (OMs), discuss option selection for the partially ordered case and describe an algorithm for learning with OMs. We compare our approach in zero-shot, single- and multi-task settings in an environment with fully specified and underspecified instructions. We find that OMs perform significantly better than or comparable to the state-of-art in all environments and learning settings.

1 Introduction

Reinforcement Learning (RL) is a powerful framework for learning complex behaviors. Sample effiency, however, remains an open challenge in RL and prevents adoption in many real-world settings [Dulac-Arnold *et al.*, 2019; den Hengst *et al.*, 2020]. Sample efficiency is often improved with knowledge of a good solution, e.g. with demonstrations, increasingly complex tasks [Bengio *et al.*, 2009], intermediate rewards [Ng *et al.*, 1999] and by decomposing the task into subtasks that are easier to learn [Dietterich, 2000].

Recently, approaches have become popular for making RL more sample efficient with high-level symbolic knowledge. These methods combine the clear semantics, verifiability and well-understood compositional and computational characteristics of symbolic methods at a high level of abstraction with the power and flexibility of RL at large, low-level action and state spaces [Yang *et al.*, 2018; Toro Icarte *et al.*, 2018b; Toro Icarte *et al.*, 2018a; Camacho *et al.*, 2019; Lyu *et al.*, 2019; Illanes *et al.*, 2020; den Hengst *et al.*, 2022]. These works demonstrate that symbolic instructions form a compelling complement to RL. A drawback of existing methods, however, is that they require the instructions to fully define the task at hand. Specifically, these assume that high rewards are *always* obtained if the instructions are followed. Such rich instructions, however, may be hard to attain in practice. Firstly, knowledge of a good solution may be tacit. Secondly, the solution space may be so large that only partial instructions are feasible, e.g. chess opening and closing strategies. Finally, the quality of a solution may not be known a priori, e.g. when it depends on the agents' capabilities or user preferences.

We therefore target a setting in which an agent is to optimize an environment reward with the help of *underspecified* instructions. These instructions define a solution at a high level of abstraction and, crucially, do not define the task at hand completely: following these instructions does not guarantee a high environment reward. Such instructions are abundant in a vast range of domains, including driving directions and clinical guidelines. In this paper, we propose and evaluate a framework for sample-efficient RL with underspecified instructions.

The framework consists of a high-level controller over a set of temporally extended actions known as *options* [Sutton *et al.*, 1999] and uses a formalism that allows for looping, interleaving and partial ordering of such options. The policies for these options are trained to optimize an environment return and can be reused both within a single task and across tasks. We compare our approach with the state of the art on an environment with instructions that fully specify the task and an environment in which the instructions are underspecified.

In summary, the contributions of this paper are:

- the first approach to increase sample efficiency of an RL agent with high-level and underspecified instructions;
- methods for specification, control and learning for options with rich initiation and termination conditions;
- intuitive instruction semantics that allow reuse of options both within a single task and across multiple tasks;
- state of the art performance in a single-task setting and significant outperformance of the state of the art in zero-

[*] Contact Author

shot and multi-task settings across environments with fully specified and underspecified instructions.

After comparing our approach to related work and introducing preliminaries, we introduce our framework in Section 4. We detail how instructions are formalized and used for control, then present a learning algorithm in Section 5, an experimental evaluation in Section 6 and a discussion in Section 7.

2 Related Work

The literature on improving RL sample efficiency is vast and contains many task- or domain-specific approaches. We limit the discussion here to generic methods for expressing and supplying knowledge to the learner.

2.1 Hierarchical RL

Our work uses the expressive formalism of finite state transducers (FSTs) to specify initiation and termination conditions of temporally extended actions and can hence be seen an extension of the options framework [Sutton *et al.*, 1999], see Section 3.1. Our framework specifically proposes the use of a, to the best of our knowledge, novel kind of option with non-Markovian initiation and termination conditions, see Section 4.3. In the context of hierarchical RL, both sequential [Singh, 1992] and subroutine-based [Dietterich, 2000] formalisms have been used to define options. Unlike our proposed approach, these formalisms do not allow for interleaving, looping or partial ordering of options.

2.2 Classical Planning and RL

High-level control with classical planning and primitive control with RL goes back to Ryan [2002] who proposed to use plans obtained from high-level teleo-operators mapping states to suitable behaviors. Another early example used STRIPS planning and was extended with reward shaping [Grounds and Kudenko, 2005; Grzes and Kudenko, 2008]. More recently, Yang *et al.* [2018] and Lyu *et al.* [2019] proposed to use an action language from which subtasks are derived. Solutions to these are combined to solve new tasks and are optimized using intrinsic rewards. Illanes *et al.* [2020] introduced the problem of 'taskable RL' and propose a solution based on decomposition. Unfortunately, these works all require a planning goal that specifies the task completely and requires a planning model whereas our approach is robust against underspecified instructions and relies on instructions formalized as an FST which can be specified as e.g. LTL constraints.

2.3 Automata, Temporal Logics and RL

The first to recognize that automata can drastically improve RL sample efficiency were Parr and Russell [1998]. They proposed a 'hierarchy of abstract machines' to constrain the agent action space. This work was extended by iteratively refining the automata with data [Leonetti *et al.*, 2012; Leonetti *et al.*, 2016]. These automata operate on primitive actions and have no abstraction over actions.

Another line of work proposes to specify tasks in temporal logic formulas. These formulas are then converted into a reward function with the aim for the agent is to learn how to satisfy the formula [Sadigh *et al.*, 2014; Fu and Topcu, 2014; Li *et al.*, 2017; Brafman *et al.*, 2018; den Hengst *et al.*, 2022]. These works require the full task to be specified whereas we target optimizing an unknown environment reward function using possibly underspecified instructions.

Some works consider decomposition of tasks specified in a temporal logic formula with the option framework. Andreas *et al.*[2017] introduced an approach for learning modular behaviors over sequences of subtasks. This approach optimizes an environment reward but does not support looping or interleaving subtasks and requires learning when to switch to a new subtask. Toro Icarte *et al.* [2018a] similarly learn a policy per subtask, but infer subtasks from an LTL formula using LTL progression. The same authors propose to learn a policy per state of an automaton representation of the formula [Toro Icarte *et al.*, 2018b; Camacho *et al.*, 2019]. These approaches specify temporally extended behaviors *implicitly*, i.e. there is no transparency at the meta-controller level, whereas we use *explicitly* named options. Reuse of options is therefore limited and their approach may not be applicable to certain zero-shot settings. On top of this, many policies may need to be learned, as the size of the automaton may grow exponentially in the size of the formula. Most importantly, these approaches also require that the entire task is specified upfront, whereas we target optimizing an unknown environment reward with possibly underspecified instructions.

3 Preliminaries

3.1 Reinforcement Learning

The RL framework can be used to maximize the amount of collected rewards in an environment by selecting an action at each time step [Sutton and Barto, 2018]. Such problems are formalized as a Markov Decision Problem (MDP) $M : \langle S, A, T, R, \gamma, S_0 \rangle$ with a set of environment states $S = \{s^1, \dots, s^n\}$, a set of agent actions $A = \{a^1, \dots, a^m\}$, a probabilistic transition function $T : S \times A \rightarrow \mathbb{P}(S)$ function $R : S \times A \times S \rightarrow [R_{\text{min}}, R_{\text{max}}]$ with $R_{\text{min}}, R_{\text{max}} \in \mathbb{R}$, a discount factor $\gamma \in [0, 1)$ to balance current and future rewards and S_0 a distribution of initial states at time step $t = 0$. At each time step t, the agent observes an environment state s_t and performs some action $a_t \sim \pi \in \Pi : S \rightarrow \mathbb{P}(A)$ and collects reward $r_t = R(s_t, a_t, s_{t+1})$. An optimal policy π^* yields the highest obtainable discounted cumulative rewards. For complex tasks it may be difficult to discover any positive rewards. The agent can be given progressively more complex tasks known as *curriculum learning* [Bengio *et al.*, 2009].

Actor-Critic Methods

Actor-critic (AC) methods optimize a set of weights $\boldsymbol{\theta}$ on which the policy is conditioned: $a \sim \pi(s, \cdot; \boldsymbol{\theta})$ [Williams, 1992; Konda and Tsitsiklis, 2000]. This *actor* is itself optimized with an estimated state-value $\hat{v}_\pi(s; \mathbf{w})$, conditioned on a second set of weights $\mathbf{w}$ referred to as the *critic*. Both sets of weights can then be optimized with the following update rules for given step sizes $\alpha^{\boldsymbol{\theta}}, \alpha^{\mathbf{w}} > 0$ and a given interaction with the environment (s_t, a_t, r_t, s_{t+1}) and resulting return $g = \sum_{j=t}^{\infty} \gamma^{j-t} R(s_j, a_j, s_{j+1})$ at time t:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \big(\nabla \hat{v}(s_t; \mathbf{w})\big) \big(g - \hat{v}(s_t; \mathbf{w})\big) \qquad (1)$$

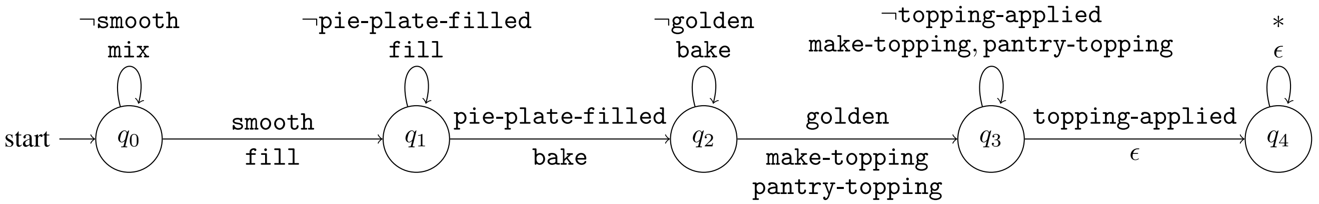

Figure 1: Option Machine for the pie recipe from Example 1 with environment events $\{\texttt{smooth}, \texttt{pie-plate-filled}, \texttt{golden}, \texttt{topping-applied}\}$ and options $\{\texttt{mix}, \texttt{fill}, \texttt{bake}, \texttt{make-topping}, \texttt{pantry-topping}\}$.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \big(\nabla \log \pi(s_t, a_t; \boldsymbol{\theta})\big) \hat{v}(s_t; \mathbf{w}) \qquad (2)$$

Options

The option framework introduces an abstraction over the space of actions [Sutton *et al.*, 1999]. The agent selects a 'primitive' action $a \in A$ or 'multi-step' action at each time step. These *options* are formalized as a tuple $\langle \mathcal{I}, \pi, \beta \rangle$ where $\mathcal{I} : S \rightarrow \{0, 1\}$ a function indicating in which states the option can be initiated, π a policy that controls the agent when the option is active and $\beta : S \rightarrow \{0, 1\}$ a termination function that determines when the option becomes inactive. If the options are trained with an actor-critic method then each option o can have its own actor $\boldsymbol{\theta}_o$ and critic $\mathbf{w}_o$. We denote the sets of all actors and critics for all options as $\boldsymbol{\Theta}$ and $\mathbf{W}$.

3.2 Finite State Transducers

Transducers are a generalization of finite state machines for control and define a mapping between two different types of information. We focus on deterministic FSTs whose output is determined by its current state and input, known in literature as a Mealy machine. We define a FST as a tuple $\varphi : \langle \Sigma, \Omega, Q, I, F, \delta \rangle$ where Σ is a finite input alphabet, Ω a finite output alphabet, Q a finite set of states, $I \subseteq Q$ the set of initial states, $F \subseteq Q$ the set of terminal or final states, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow Q \times (\Omega \cup \{\epsilon\})$ a transition functions where ϵ the empty string [Mealy, 1955]. A FST can be specified in a temporal logic such as LTL and then converted to a FST with out-of-the-box tools [Michaud and Colange, 2018].

4 The Option Machine Framework

In this section we introduce a framework for using underspecified instructions in RL. Specifications for Option Machines (OMs) can be underspecified in two ways. Firstly, the instructions specify what to do at a high level of abstraction rather than at the level of primitive actions. Secondly, a policy following the instructions in OMs is not assumed to always get high environment rewards. This contrasts with most related works, in which following the instructions is equated to high environment rewards. OMs, in contrast, use the environment reward as the canonical definition of the task and leverage instructions for reuse of obtained knowledge, improved exploration and better reward attribution.

Example 1. *A recipe gives instructions for a particular type of pie. While each type of pie is a separate task, recipes refer to common steps such as mixing ingredients, pouring, baking etc. Solutions for these steps can be reused across recipes. A recipe may be underspecified and not guarantee a tasty result as baking requires more knowledge than just the recipes.*

We now introduce OMs formally from the perspective of a curriculum of tasks. An OM curriculum is defined as a tuple $C : \langle S, A, T, \gamma, \mathcal{R}, P, \Phi, L \rangle$ where S, A, T, γ are defined as usual in RL, see Section 3.1. Tasks $\mathcal{R}$ are formalized as a set of environment reward functions, P a probability distribution over tasks $\mathcal{R}$ and instructions Φ as a set of FSTs. Each $\varphi_i \in \Phi$ corresponds to a particular task R_i and has some Σ_i of environment events as its input alphabet. We assume that a function for detecting these events $L : S \rightarrow \bigcup_\Phi \Sigma_i$ is available. The main loop can be found in Algorithm 1 and contains components for control and learning.

4.1 Instructions as an Option Machine

Our approach uses high-level instructions for a given task. In particular, instructions define traces of high-level behaviors based on high-level descriptions of environment states. This allows for the intuitive formalization of e.g. a recipe.

Example 1. ***(cont.)*** *A recipe 'mix ingredients until smooth, fill pie plate and bake in oven at* $180°C$ *until golden. Apply a home-made topping or use a topping from the pantry to finalize the pie.' See Figure 1 for an example OM.*

High-level descriptions of states consist of events that the agent can detect in the environment. These are formalized a set of atomic propositions AP^I, to which some truth value in $\Sigma : 2^{AP^I}$ can be assigned. Σ corresponds to the input alphabet for the FST associated with the current task. We assume that some function $L : S \rightarrow \Sigma$ for detecting these events in states is available, e.g. as a handcrafted or pretrained component. We return to our running example before we look at how events are used for high-level control.

Example 1. ***(cont.)*** *Events* $\{\texttt{smooth}, \texttt{pie-plate-filled}, \texttt{golden}\}$ *can be identified from pixel-level states.*

High-level Behaviors are actions that take multiple time steps and can be reused across tasks. These are formalized as *options* and denoted with a set of atomic propositions AP^O, to which some truth values in $\Omega : 2^{AP^O}$ can be assigned. At each time step, the permissible options in an OM are determined by this FST output. The current FST state q_t and detected events $L(s_t)$ trigger some FST transition $\delta\big(q_t, L(s_t)\big)$ which produces a new FST state q_{t+1} and an output $\omega_t \in \Omega \cup \epsilon$. The 'true' propositions in ω_t are interpreted as the set of permissible options at that particular time step and are denoted $O_t \subseteq AP^O$. An OM consists of policies associated with options, a FST that specifies which options are permissible and a mechanism to select from these. We discuss selection mechanisms in the next section. If no options are explicitly defined, then this is represented by the empty string $O_t = \{\epsilon\}$. We treat this is a particular output for which the agent uses a dedicated fallback option.

Example 1. ***(cont.)*** *Figure 1 shows that* `mix` *is the only permissible option until the event* `smooth` *is detected. From this point onward, the option* `fill` *is permissible until the event* `pie-plate-filled` *becomes true etc. When the event* `golden` *has been detected, the two options* `make-topping` *and* `pantry-topping` *become permissible simultaneously.*

4.2 Control with Option Machines

Control in the OM framework assumes a given task R_i with corresponding FST φ_i and has a two-level structure, see Algorithm 2. At the upper, meta-controller level, a suitable option is selected using φ_i. The policy for this option is then executed at the lower level and generates a primitive action $a_t \in A$ to be executed by the agent. In particular, an option is selected based on the FST output. This output defines one or multiple permissible options O_t. For now, we simply assume these policies to exist and leave the details on how these are optimized from interactions with the environment to Section 5.

Example 1. ***(cont.)*** *It may not be clear to a recipe author whether their audience has the right actuators to create a topping. Further, it may not be known whether e.g. pantry toppings are available.*

We compare three approaches to select an option from O_t. The first approach assumes a total ordering over all options AP^O which fixes the selected option as the highest-ranked permissible option in O_t. This 'fixed' approach does not incorporate learning in the upper level of control but it comes with the benefit of stability of agent behavior. The other two approaches do incorporate learning in the upper level of decision-making and both use option-specific state-value estimates $\hat{v}(s; \mathbf{w}_o)$. The first of these simply selects an option o from the permissible options O_t greedily:

$$f(O_t, s, \mathbf{W}) = \arg\max_{o \in O_t} \hat{v}(s; \mathbf{w}_o) \tag{3}$$

The greedy approach, however, may result in frequent switches between options, e.g. when estimates are inaccurate during early phases of learning or when all permissible options yield a similar return. To mitigate this, we introduce a 'sticky' mechanism that defaults to selecting the previous option o_{t-1} if it is permissible and greedily otherwise:

$$f(O_t, s, o_{t-1}, \mathbf{W}) = \begin{cases} o_{t-1} & \text{if } o_{t-1} \in O_t \\ \arg\max_{o \in O_t} \hat{v}(s; \mathbf{w}_o) & \text{otherwise} \end{cases} \tag{4}$$

4.3 Reusable Policies and Non-Markovian Options

Policies in the OM framework have names AP^O and can therefore easily be reused within a task or across tasks. For example, the policy for mixing ingredients can be used for mixing both the dough and the filling in a single cake recipe. Additionally, multiple recipes may require mixing dough. Named options enable reuse of policies in e.g. a zero-shot setting where an unseen task can be solved by combining previously encountered options.

Algorithm 1 Main loop

```
Input: curriculum $C : \langle S, A, T, \gamma, \mathcal{R}, P, \Phi, L \rangle$,
parameterizations $\pi(\cdot; s, \boldsymbol{\theta})$ and $\hat{v}(s; \mathbf{w})$
Parameters: learning steps $N$, batch size $D$
Output: set of actors $\boldsymbol{\Theta}$ and set of critics $\mathbf{W}$

$i \leftarrow 0, \mathcal{D} \leftarrow \emptyset, \boldsymbol{\Theta} \leftarrow \emptyset, \mathbf{W} \leftarrow \emptyset$
$\forall o \in AP^O \cup \epsilon$, add random weights $\boldsymbol{\theta}_o$ to $\boldsymbol{\Theta}$, $\mathbf{w}_o$ to $\mathbf{W}$.
while $i < N$ do
    while $|\mathcal{D}| < D$ do
        sample $(R \in \mathcal{R}, \varphi \in \Phi) \sim P$.
        $d \leftarrow$ rollout for task $R$ and instructions $\varphi$. {Alg. 2}
        $\mathcal{D} \leftarrow \mathcal{D} \cup d$.
    end while
    update parameters $\boldsymbol{\Theta}, \mathbf{W}$ with $\mathcal{D}$.            {Alg. 3}
    $i \leftarrow i + 1$.
end while
return $\boldsymbol{\Theta}, \mathbf{W}$.
```

Algorithm 2 Control with an Option Machine

```
Input: finite-state transducer $\varphi$, actors $\boldsymbol{\Theta}$, critics $\mathbf{W}$,
labelling $L : S \rightarrow \Sigma$
Parameters: shaping reward $\rho \geq 0$
Output: episode $d$

initialize $o$, $d \leftarrow \emptyset$, $q \leftarrow q_0 \in \varphi$, observe $s$.
while $q$ and $s$ are not terminal do
    $(q', O) \leftarrow \delta(q, L(s))$.
    $o \leftarrow$ select from $O$.                    {Equation 3 or 4}
    perform action $a \sim \pi(\cdot|s, \boldsymbol{\theta}_o)$.
    observe $r$ and $s'$.
    append $(s, o, q, a, r, s')$ to $d$.
    $s \leftarrow s', q \leftarrow q'$.
end while
return $d$.
```

The initiation and termination condition of options in our framework are defined by the FST and based on the history of observed events $L(s_0), L(s_1), \ldots, L(s_t)$. These conditions are therefore non-Markovian. This enables powerful yet intuitive control, including looping and interleaving of options.

5 Learning with Option Machines

In this section we look at the problem of learning optimal policies for options from environment interactions generated by a sequence of these options. A key challenge here is to attribute rewards to the appropriate option. If an option was in control at a particular point in time, should future rewards be attributed to this option or not? First, however, we detail how instructions in OMs can be used to guide the agent with shaping rewards.

Shaping rewards are small positive (or negative) intermediate rewards for actions or states that are promising (or to be

Algorithm 3 Learning with Option Machines

Input: actors $\boldsymbol{\Theta}$, critics $\mathbf{W}$, episodes $\mathcal{D}$
Parameters: learning rates $\alpha^{\boldsymbol{\theta},\mathbf{w}}$, shaping reward ρ, discount factor γ
Output: updated actors $\boldsymbol{\Theta}$ and critics $\mathbf{W}$

```
1:  for all $d \in \mathcal{D}$ do
2:      $d' \leftarrow$ reverse episode $d$.
3:      $q' \leftarrow q \in d'[0]$.
4:      $o' \leftarrow o \in d'[0]$.                              {option to train}
5:      $g_e \leftarrow 0$.                                 {environment return}
6:      $g_s \leftarrow 0$.                                     {shaping return}
7:      for all $(s, o, q, a, r, s') \in d'$ do
8:          if $q \neq q'$ then
9:              $o' \leftarrow o$.                       {update option to train}
10:             $g_s \leftarrow \rho$.                         {add shaping rewards}
11:         else
12:             $g_s \leftarrow \gamma g_s$.                {discount shaping return}
13:         end if
14:         $g_e \leftarrow \gamma g_e + r$.             {update environment return}
15:         $g \leftarrow g_e + g_s$.                                  {total return}
16:         $\boldsymbol{\theta}_{o'} \overset{+}{\leftarrow} \alpha^{\boldsymbol{\theta}} \big(\nabla \log \pi(a|s, \boldsymbol{\theta}_{o'})\big)\big(g - \hat{v}(s, \mathbf{w}_{o'})\big)$.
17:         $\mathbf{w}_{o'} \overset{+}{\leftarrow} \alpha^{\mathbf{w}} \big(\nabla \hat{v}(s, \mathbf{w}_{o'})\big)\big(g - \hat{v}(s, \mathbf{w}_{o'})\big)$.
18:     end for
19: end for
20: return $\boldsymbol{\Theta}, \mathbf{W}$
```

avoided). These can be defined based on prior knowledge of a good solution. For example, a small positive reward can be given for solving a subtask such as successfully baking a pie crust. The usage of the FST formalism gives a very natural way to delineate subtasks using FST states. In particular, if the FST transitions from some state q to another state $q' \neq q$, a preset shaping reward ρ can be applied to inform the agent that it is progressing according to the instructions. Shaping rewards can be defined naturally in our approach.

We now turn to the problem of attributing rewards to options and propose a method to address it using FST state information. We first consider the simple case where a single option o was active while visiting a FST state q. In this case, a transition from q to another state $q' \neq q$ must have been caused by the actions sampled according to that options' policy $\boldsymbol{\theta}_o$. Hence, future rewards should be used to update that options' policy. The case of multiple options executing before a transition to a new state, however, poses a problem. Reaching the event that triggers this transition requires different policies for the used options. Hence, these interactions should not be used to update both options policies naively. We propose to use all interactions for updating only the policy of the last option executing in a FST state instead.

Example 1. ***(cont.)*** *In Figure 1, a single option will execute while visiting states* $\{q_0, q_1, q_2\}$*. During visits to* q_3 *both* `make-topping` *and* `pantry-topping` *may execute (although not at the same time) until the* `topping-applied` *event is observed. If* `pantry-topping` *last executes before the event* `topping-applied` *is observed then this option's policy will be updated during learning.*

			Option Machines		
Env.	Setting	Sketch	Fixed	Greedy	Sticky
maze	isolation	0.09	**0.60**	N/A	N/A
	holdout	0.49	**0.54**		
craft	isolation	0.03	**0.90**	0.74	0.73
	holdout	0.05	**0.86**	0.13	0.22

Table 1: Zero-shot total environment reward on 1K test episodes. **Bold** denotes significant best (Mann-Whitney U, $p < 0.01$).

Algorithm 3 lists a learning algorithm that implements these ideas on reward shaping and reward attribution. First, the final automaton state and active option are extracted and both the discounted cumulative environment return g_e and shaping return g_s are initialized (lines 1-6). In lines 8-12, the last executing option o in a particular FST state q is set as the target option o' to optimize and the shaping rewards are calculated. These are added to the total reward (lines 14-15) and used to update the actor and critic parameters (lines 16-17). The learning algorithm thus leverages FST state information in two ways: firstly, shaping rewards can be supplied to promote exploration and reinforce subtask completion and secondly, interactions are mapped to a single option to ensure that the parameters of the appropriate option are updated.

6 Experiments

In this section, we provide an empirical evaluation of OMs in an environment with both fully specified and underspecified instructions. We evaluate OMs in single-task, multi-task and two zero-shot settings to answer the research questions:

1. Do the instructions improve sample efficiency?
2. What are effects of named options and reward shaping?
3. Which option selection method to use?

We include versions of OMs for each of the option selection mechanisms described in Section 4.2: OM-fixed selects based on an arbitrarily fixed order, OM-greedy selects according to Equation 3 and OM-sticky according to Equation 4.

6.1 Baselines

We compare option machines to three state-of-the-art approaches. Firstly, we include the 'sketch'-based approach proposed by Andreas *et al.*[2017]. This approach targets the multi-task setting, uses a *sequence* of subtasks rather than the richer representation proposed here and learns option termination conditions. Secondly, we compare to reward machines (RM) by Icarte *et al.* [2018b] which assume that the instructions specify the task fully and require that the training and evaluation subtasks use the same events. This is not the case for the tasks included here and we therefore do not include RM in the zero-shot setting. For all algorithms, we use AC as the base learner and we include a vanilla AC baseline per task in the single-task setting, denoted 'RL'.

6.2 Experimental Setup

Two benchmark environments by [Andreas *et al.*, 2017] are used to evaluate the approach. In the 'craft' environment,

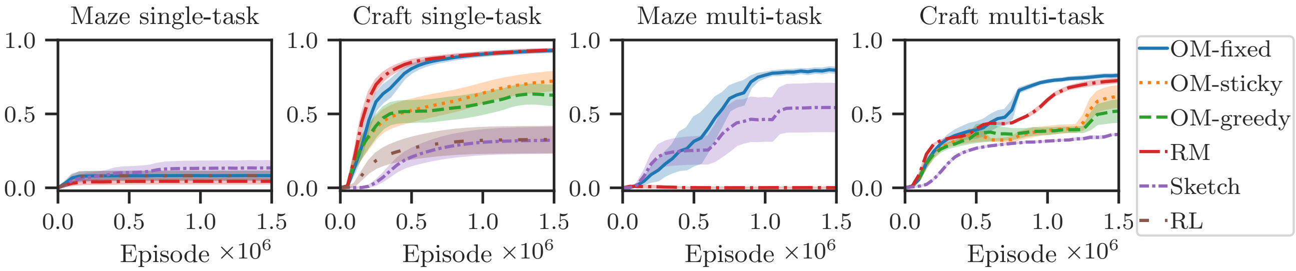

Figure 2: Total environment rewards per episode in the single- and multi-task setting on two environments.

items can be obtained by collecting resources such as wood and iron and combining them at workshop locations. Instructions may specify multiple permissible options simultaneously or may fully specify tasks. In the 'maze' environment, the agent must navigate a series of rooms with doors. An event detector describes whether the agent is in a door or not. Critically, it does not differentiate doors leading to the desired room from other doors. As a result, instructions are underspecified. Furthermore, instructions only permit one option at a time. We therefore do not include OM-greedy and OM-sticky in this environment.

An existing curriculum learning setup was used for multi-task learning [Andreas *et al.*, 2017]. Initially, only tasks associated with two options are presented. Once the mean reward on these reaches a threshold of 0.8, this limit is incremented. Tasks within this limit are sampled inversely proportional to the obtained reward. Results were selected with a grid search over hyperparameters. Shaping reward hyperparameters $\rho = 0$ and $\rho = 0.1$ were selected for the maze and craft environment respectively. We report averages over five random seeds. A detailed description of the environments, tasks, hyperparameters etc. can be found in the Appendix.

6.3 Results

Single-task Results

The two leftmost graphs in Figure 2 show the single-task results on all tasks consisting of more than two options. The maze environment proves too challenging. The reason is its inherent exploration problem which cannot be mitigated by the instructions. Following these does not guarantee solving the task and hence shaping rewards do not help. In the craft environment, shaping is useful: the RM and OM-fixed approaches significantly outperform all others. The usage of named options has negligible effects as RM and OM-fixed perform similarly. Finally, we see a slight advantage of using the sticky option selection over its greedy counterpart.

Multi-task Results

The two rightmost graphs in Figure 2 show that the instructions improve sample efficiency as our approach significantly outperforms all baselines. In the maze environment, this can all be attributed to the usage of named options since there are no shaping rewards with $\rho = 0$. Also, note that RMs fail to perform in the multi-task setting because they use the instructions as the full specification of the task. In the craft environment, the instructions *do* fully specify the task and shaping rewards increase sample efficiency. A comparison between OM-fixed and RM indicates that the usage of named options increases sample efficiency significantly. Again, we see that OM-fixed outperforms the other OM variants and that using sticky option selection provides a slight benefit.

Zero-shot Results

We evaluate applicable approaches in two zero-shot settings. In the first setting, policies for all options are trained in isolation and then evaluated on tasks composed of these options. We include all tasks here. In the second setting, policies are trained on a set of training tasks and then evaluated on two unseen, held out, tasks. For OM-based approaches, we execute Algorithm 2 in both settings. Table 1 shows that all of the OM versions significantly outperform the baseline in both environments. OM-fixed outperforms all OM versions. The difference here is striking in the holdout case.

The holdout setting is challenging since policies are optimized in the context of tasks other than the evaluation task. As a result, a policy associated with some option o is positively reinforced if it completes a subtask associated with a later option o'. If this subtask is not part of the evaluation task, completing it may harm performance. It could take time and affect later subtasks if these are not commutative. OM-fixed is less susceptible to this failure mode then the other variants, as it uses the same delineation across all episodes. This does not show in the 'isolation' training setting where the greedy and sticky variants perform significantly better than their counterparts trained in the holdout setting.

7 Discussion

We proposed a framework for sample efficient RL with underspecified instructions. These are represented with powerful and intuitive FSTs as a natural way to define shaping rewards and use named options for the reuse of learned behaviors. Experimental evaluations show state of the art performance in a single-task setting and significant outperformance of the state of the art in zero-shot and multi-task settings across environments with fully specified and with underspecified instructions. We have found indications that shaping rewards should not be used when instructions do not cover the task at hand completely but that named options provide a significant benefit. Finally, results indicate that named options significantly increase performance in the multi-task and zero-shot settings.

Future work includes the development of a calculus of instructions for RL with FST operations and the study of ways to derive OMs from interactions to communicate learned strategies with other agents and humans.

References

[Andreas *et al.*, 2017] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. In *ICML*, pages 166–175. PMLR, 2017.

[Bengio *et al.*, 2009] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *ICML*, volume 26, pages 41–48, 2009.

[Brafman *et al.*, 2018] R. Brafman, G. De Giacomo, and F. Patrizi. Ltlf/ldlf non-markovian rewards. In *AAAI*, volume 32. AAAI, 2018.

[Camacho *et al.*, 2019] A. Camacho, R. Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, pages 6065–6073, 2019.

[den Hengst *et al.*, 2020] F. den Hengst, E.M. Grua, A. El Hassouni, and M. Hoogendoorn. Reinforcement learning for personalization: A systematic literature review. *Data Science*, 3(2):107–147, 2020.

[den Hengst *et al.*, 2022] F. den Hengst, V. François-Lavet, M. Hoogendoorn, and F. van Harmelen. Planning for potential: efficient safe reinforcement learning. *Machine Learning*, pages 1–20, 2022.

[Dietterich, 2000] T.G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *JAIR*, 13:227–303, 2000.

[Dulac-Arnold *et al.*, 2019] G. Dulac-Arnold, D. Mankowitz, and T. Hester. Challenges of real-world reinforcement learning. *arXiv:1904.12901*, 2019.

[Fu and Topcu, 2014] J. Fu and U. Topcu. Probably approximately correct mdp learning and control with temporal logic constraints. In *Robotics: Science and Systems*, number 10, 2014.

[Grounds and Kudenko, 2005] M. Grounds and D. Kudenko. Combining reinforcement learning with symbolic planning. In *AAMAS*, pages 75–86. Springer, 2005.

[Grzes and Kudenko, 2008] M. Grzes and D. Kudenko. Plan-based reward shaping for reinforcement learning. In *IntelliSys*, volume 2, pages 10–22. IEEE, 2008.

[Illanes *et al.*, 2020] L. Illanes, X. Yan, R. Icarte, and S. McIlraith. Symbolic plans as high-level instructions for reinforcement learning. In *ICAPS*, volume 30, pages 540–550, 2020.

[Konda and Tsitsiklis, 2000] V. Konda and J. Tsitsiklis. Actor-critic algorithms. In *NIPS*, pages 1008–1014, 2000.

[Leonetti *et al.*, 2012] M. Leonetti, L. Iocchi, and F. Patrizi. Automatic generation and learning of finite-state controllers. In *AIMSA*, pages 135–144. Springer, 2012.

[Leonetti *et al.*, 2016] M. Leonetti, L. Iocchi, and P. Stone. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence*, 241:103–130, 2016.

[Li *et al.*, 2017] X. Li, C. Vasile, and C. Belta. Reinforcement learning with temporal logic rewards. In *IROS*, pages 3834–3839. IEEE/RSJ, 2017.

[Lyu *et al.*, 2019] D. Lyu, F. Yang, B. Liu, and S. Gustafson. Sdrl: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *AAAI*, volume 33, pages 2970–2977, 2019.

[Mealy, 1955] G.H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Michaud and Colange, 2018] T. Michaud and M. Colange. Reactive synthesis from ltl specification with spot. In *7th Workshop on Synthesis, SYNT@ CAV*, 2018.

[Ng *et al.*, 1999] A.Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, pages 278–287, 1999.

[Parr and Russell, 1998] R. Parr and S. Russell. Reinforcement learning with rhierarchies of machines. *NIPS*, pages 1043–1049, 1998.

[Ryan, 2002] M. Ryan. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *ICML*, volume 2, pages 522–529, 2002.

[Sadigh *et al.*, 2014] D. Sadigh, E.S. Kim, S. Coogan, S.S. Sastry, and S. Seshia. A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In *CDC*, pages 1091–1096. IEEE, 2014.

[Singh, 1992] S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3):323–339, 1992.

[Sutton and Barto, 2018] R.S. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sutton *et al.*, 1999] R.S. 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.

[Toro Icarte *et al.*, 2018a] R. Toro Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Teaching multiple tasks to an rl agent using ltl. In *AAMAS*, pages 452–461, 2018.

[Toro Icarte *et al.*, 2018b] R. Toro Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *ICML*, volume 35, pages 2107–2116. PMLR, 2018.

[Williams, 1992] R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[Yang *et al.*, 2018] F. Yang, D. Lyu, B. Liu, and S. Gustafson. Peorl: integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. In *IJCAI*, pages 4860–4866, 2018.

Appendices

Appendix A: Details Experimental Setup

The implementation of the environments and the ‘sketch’ baseline used in all experiments is the same as the one described by Andreas *et al.*. We include a list of all parameters and a description of the setup and the environments here for completeness. Instructions (see below) were defined in LTL and then converted into FST meta-controllers with the ltlsynt tool by [Michaud and Colange, 2018] and then implemented in Python. In all of our experiments, we implement each policy as a feedforward neural network with ReLU activations and critics as a linear function of the state. Both are optimized with the RMSProp optimizer. Table 2 lists all (hyper)parameters used.

Parameter	Value(s)	Description
step size	0.001	RMSProp optimization step size.
γ	0.9	Parameter to balance immediate and long-term rewards.
D	2000	Training algorithm batch size.
ρ	$\{0, 0.1\}$	Intrinsic or shaping rewards.
seeds	$\{0, 1, 2, 3, 4\}$	Initialization of the pseudo-random number generator.

Table 2: Parameters used in all experiments.

Craft Environment

The deterministic ‘craft’ environment is a 10×10 grid in which the agent senses the (x, y) position of locations of interest such as resources and workshops, relative to its own location. The state representation for this environment is a vector of dimensionality 1075, consisting of indicator parameters for each possible item in the agent inventory, indicator parameters for the position of locations of interest relative to the agents position and indicator parameters for the direction the agent is facing. The action space is defined as $\{\text{up}, \text{down}, \text{left}, \text{right}, \text{use}\}$ where the first four always move the agent in the particular direction in the grid. Options are terminated based on instructions or after fifteen time steps and episodes are terminated after 100 time steps.

Task	LTL specification
Plank	$((\texttt{get-wood} \wedge \neg\texttt{w0}))\mathbf{W}(\texttt{wood} \vee \texttt{plank})) \wedge (\mathbf{F}\texttt{wood} \implies \mathbf{F}(\texttt{w0}\mathbf{W}\texttt{plank}))$
Stick	$((\texttt{get-wood} \wedge \neg\texttt{w1})\mathbf{W}(\texttt{wood} \vee \texttt{stick})) \wedge (\mathbf{F}\texttt{wood} \implies \mathbf{F}(\texttt{w1}\mathbf{W}\texttt{stick}))$
Cloth	$((\texttt{get-grass} \wedge \neg\texttt{w2})\mathbf{W}(\texttt{grass} \vee \texttt{cloth})) \wedge (\mathbf{F}\texttt{grass} \implies \mathbf{F}(\texttt{w2}\mathbf{W}\texttt{cloth}))$
Rope	$((\texttt{get-grass} \wedge \neg\texttt{w0})\mathbf{W}(\texttt{grass} \vee \texttt{rope})) \wedge (\mathbf{F}\texttt{grass} \implies \mathbf{F}(\texttt{w0}\mathbf{W}\texttt{rope}))$
Bridge	$\mathbf{G}(\neg(\texttt{w2} \wedge \texttt{get-grass}) \wedge \neg(\texttt{w2} \wedge \texttt{get-iron})) \wedge (\texttt{get-wood}\mathbf{W}\texttt{wood}) \wedge (\texttt{get-iron}\mathbf{W}\texttt{iron}) \wedge ((\mathbf{F}\texttt{wood} \wedge \mathbf{F}\texttt{iron}) \implies \mathbf{F}(\texttt{w2}\mathbf{W}\texttt{bridge})$
Bed	$\mathbf{G}(\neg(\texttt{w1} \wedge \texttt{get-wood}) \wedge \neg(\texttt{w1} \wedge \texttt{get-grass}) \wedge \neg(\texttt{w1} \wedge \texttt{w0})) \wedge (\texttt{get-grass}\mathbf{W}\texttt{grass}) \wedge (\texttt{get-wood}\mathbf{W}\texttt{wood}) \wedge (\mathbf{F}(\texttt{wood}) \implies (\texttt{w0}\mathbf{W}\texttt{plank})) \wedge ((\mathbf{F}\texttt{wood} \wedge \mathbf{F}\texttt{plank} \wedge \mathbf{F}\texttt{grass}) \implies \mathbf{F}(\texttt{w1}\mathbf{W}\texttt{bed})$
Axe	$\mathbf{G}(\neg(\texttt{w1} \wedge \texttt{get-wood}) \wedge \neg(\texttt{w1} \wedge \texttt{get-iron}) \wedge \neg(\texttt{w1} \wedge \texttt{w0})) \wedge (\texttt{get-iron}\mathbf{W}\texttt{iron}) \wedge (\texttt{get-wood}\mathbf{W}\texttt{wood}) \wedge (\mathbf{F}(\texttt{wood}) \implies (\texttt{w1}\mathbf{W}\texttt{stick})) \wedge ((\mathbf{F}\texttt{wood} \wedge \mathbf{F}\texttt{stick} \wedge \mathbf{F}\texttt{iron}) \implies \mathbf{F}(\texttt{w0}\mathbf{W}\texttt{axe})$
Shears	$(\texttt{get-iron}\mathbf{W}\texttt{iron}) \wedge (\texttt{get-wood}\mathbf{W}\texttt{wood}) \wedge (\mathbf{F}(\texttt{wood}) \implies (\texttt{w1}\mathbf{W}\texttt{stick})) \wedge ((\mathbf{F}\texttt{wood} \wedge \mathbf{F}\texttt{stick} \wedge \mathbf{F}\texttt{iron}) \implies \mathbf{F}(\texttt{w1}\mathbf{W}\texttt{shears})$
Gold	specification ‘Bridge’ + $\wedge((\mathbf{F}\texttt{wood} \wedge \mathbf{F}\texttt{iron} \wedge \mathbf{F}\texttt{bridge}) \implies \mathbf{F}(\texttt{get-gold}\mathbf{W}\texttt{gold})$
Gem	specification ‘Axe’ + $\wedge((\mathbf{F}\texttt{wood} \wedge \mathbf{F}\texttt{stick} \wedge \mathbf{F}\texttt{axe}) \implies \mathbf{F}(\texttt{get-gem}\mathbf{W}\texttt{gem})$

Table 3: Curriculum of tasks and nondeterministic specifications in the ‘craft’ environment where $AP^I = \{\texttt{axe}, \texttt{bed}, \texttt{bridge}, \texttt{cloth}, \texttt{door}, \texttt{gem}, \texttt{gold}, \texttt{grass}, \texttt{iron}, \texttt{plank}, \texttt{rope}, \texttt{shears}, \texttt{stick}, \texttt{wood}\}$ each referring to having an item in the agent inventory and behaviors $AP^O = \{\texttt{get-iron}, \texttt{get-wood}, \texttt{get-grass}, \texttt{get-gold}, \texttt{get-gem}, \texttt{w0}, \texttt{w1}, \texttt{w2}\}$ where the latter refer to using three different workshops. These specifications were made deterministic by a total order over all available behaviors.

Maze Environment

The ‘maze’ environment, of which an example is depicted in Figure 5, is a grid environment of varying size. The environment consists of various adjacent rooms. The agent is placed in one of these rooms and is tasked with reaching a particular other room, possibly by traversing some intermediate rooms. Some rooms are connected by doors, which can be open or locked. Locked doors can be opened by acquiring a key to that particular door and using it on the lock. These keys are placed in a position that is reachable for the agent. The agent senses keys, locked doors and open doors in all cardinal directions and cannot sense through walls. The state representation consists of a vector describing the distance to rooms and keys in all cardinal directions, i.e. it is of dimensionality 12. The action space is defined as $\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}, \texttt{key}\}$ where the first

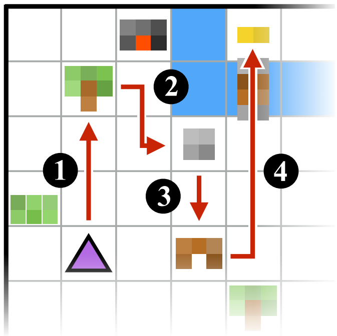

Figure 3: Visualization of craft world for the 'Gold' task. This task consists of executing (1) get-wood, (2) get-iron, (3) w0, (4) get-gold. The labelling in this world consists of whether an item such as 'wood' is present in the agent inventory. This can be encoded into an LTL specification with vocabulary AP^I : $\{\texttt{wood}, \texttt{iron}, \texttt{bridge}, \texttt{gold}\}$ to describe the environment state and options AP^O : $\{\texttt{get-wood}, \texttt{get-iron}, \texttt{w0}, \texttt{get-gold}\}$.

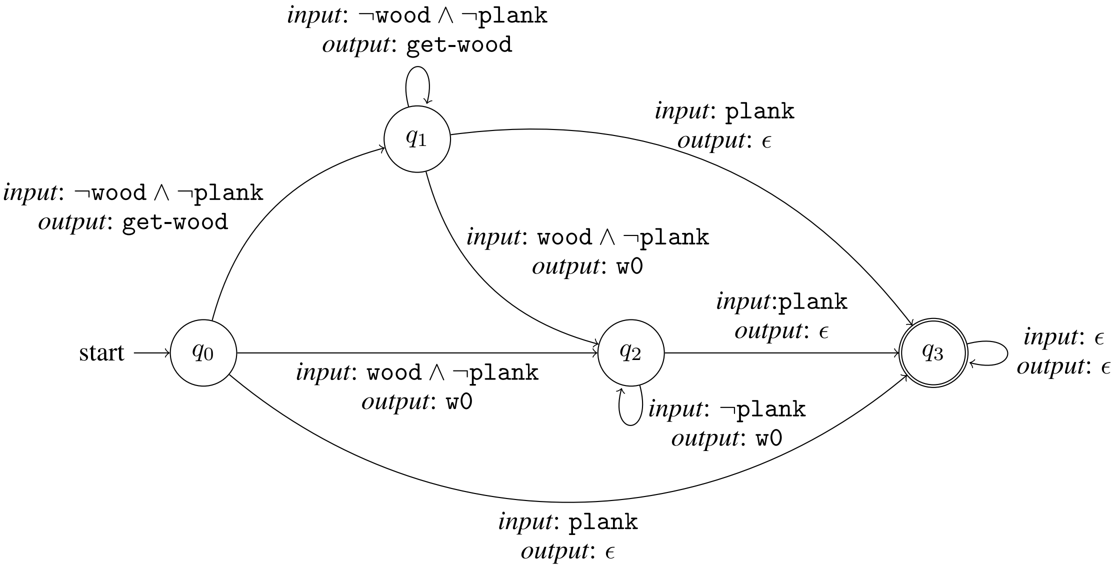

Figure 4: FST for the 'plank' task generated with the 'ltlsynt' tool of the Spot package by Michaud *et al.*. This specification has input alphabet AP^I : $\{\texttt{wood}, \texttt{plank}\}$ and output alphabet AP^O : $\{\texttt{get-wood}, \texttt{w0}\}$. Negative outputs such as ¬`get-wood` have been omitted in this representation for legibility whereas negative inputs have been included only where necessary to differentiate between available edges. For example, `wood` is not differentiating for any edges leaving q_2. Edges incoming to the terminal node q_3 produce no output: these are only visited if a plank is present in the agent inventory, i.e. upon completion of the task.

four always move the agent in the particular direction in the grid. Options are terminated based on instructions or after fifteen time steps and episodes are terminated after 100 time steps.

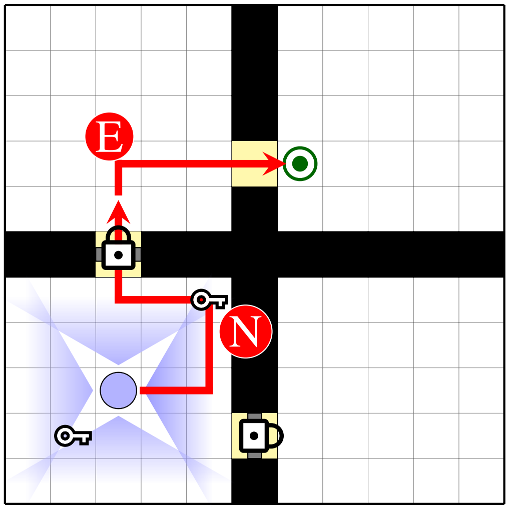

Figure 5: Visualization of a sample maze environment for the task ‘north-east’. Door states, i.e. states such that $L(s) = 1^{\texttt{door}}$, are visualized as a yellow cell 🟨. All other states are labelled as ‘not door’ states and visualized as a white cell □. The agent (blue circle, ●) senses its environment in four cardinal directions with three sensors per direction: the first detects open doors, the second senses locked doors and the last detects keys. The bottom left room contains two keys that look identical to the agent. One unlocks the door to the bottom right room, which need not be visited. The other unlocks the door to the top left and needs to be picked up to reach the target. Arrows denote options associated with ‘north’ and ‘east’ respectively for the specification $(\texttt{door}\mathbf{R}\texttt{north}) \wedge (\texttt{door} \implies \mathbf{XG}\texttt{east})$.

Task	LTL specification
West, West	$\mathbf{F}\text{door} \wedge \mathbf{G}\text{west}$
West, South	$(\text{door}\mathbf{R}\text{west}) \wedge (\text{door} \implies \mathbf{XG}\text{south})$
East, South	$(\text{door}\mathbf{R}\text{east}) \wedge (\text{door} \implies \mathbf{XG}\text{south})$
North, West	$(\text{door}\mathbf{R}\text{north}) \wedge (\text{door} \implies \mathbf{XG}\text{west})$
North, East	$(\text{door}\mathbf{R}\text{north}) \wedge (\text{door} \implies \mathbf{XG}\text{east})$
North, East, North	$(\text{door}\mathbf{R}(\text{north} \wedge \neg\text{east})) \wedge (\mathbf{F}(\text{door}) \implies (\mathbf{F}((\text{door}\mathbf{R}\text{east}) \wedge \mathbf{F}\text{door} \implies \mathbf{XG}\text{north})))$
South, East, North	$(\text{door}\mathbf{R}(\text{south} \wedge \neg\text{east})) \wedge (\mathbf{F}(\text{door}) \implies (\mathbf{F}((\text{door}\mathbf{R}\text{east}) \wedge \mathbf{F}\text{door} \implies \mathbf{XG}\text{north})))$
West, North, East	$(\text{door}\mathbf{R}(\text{west} \wedge \neg\text{north})) \wedge (\mathbf{F}(\text{door}) \implies (\mathbf{F}((\text{door}\mathbf{R}\text{north}) \wedge \mathbf{F}\text{door} \implies \mathbf{XG}\text{east})))$
West, West, South	$((\mathbf{F}(\text{door} \wedge \mathbf{X}(\mathbf{F}\text{door})))\mathbf{R}(\text{west} \wedge \mathbf{X}\text{west})) \wedge ((\mathbf{F}(\text{door} \wedge \mathbf{X}(\mathbf{F}(\text{door})))) \implies \mathbf{FG}(\text{south}) \wedge \mathbf{G}(\neg(\text{south} \wedge \text{west}))$
East, South, South	$(\text{door}\mathbf{R}\text{east}) \wedge \mathbf{F}(\mathbf{F}\text{door} \implies \mathbf{XG}(\text{south} \wedge \neg\text{east}))$

Table 4: Curriculum of tasks and specifications in the ‘maze’ environment where $AP^I = \{\texttt{door}\}$ and $AP^O = \{\text{west}, \text{east}, \text{north}, \text{south}\}$ each referring to a room to move to and not (!) primitive actions.

Appendix B: Results per task

The results in the main document are aggregated across tasks of different complexity. To highlight where difference in performance comes from, we split down the performance per task for both environments in Figures 6 and 7 and Tables 6 and 5.

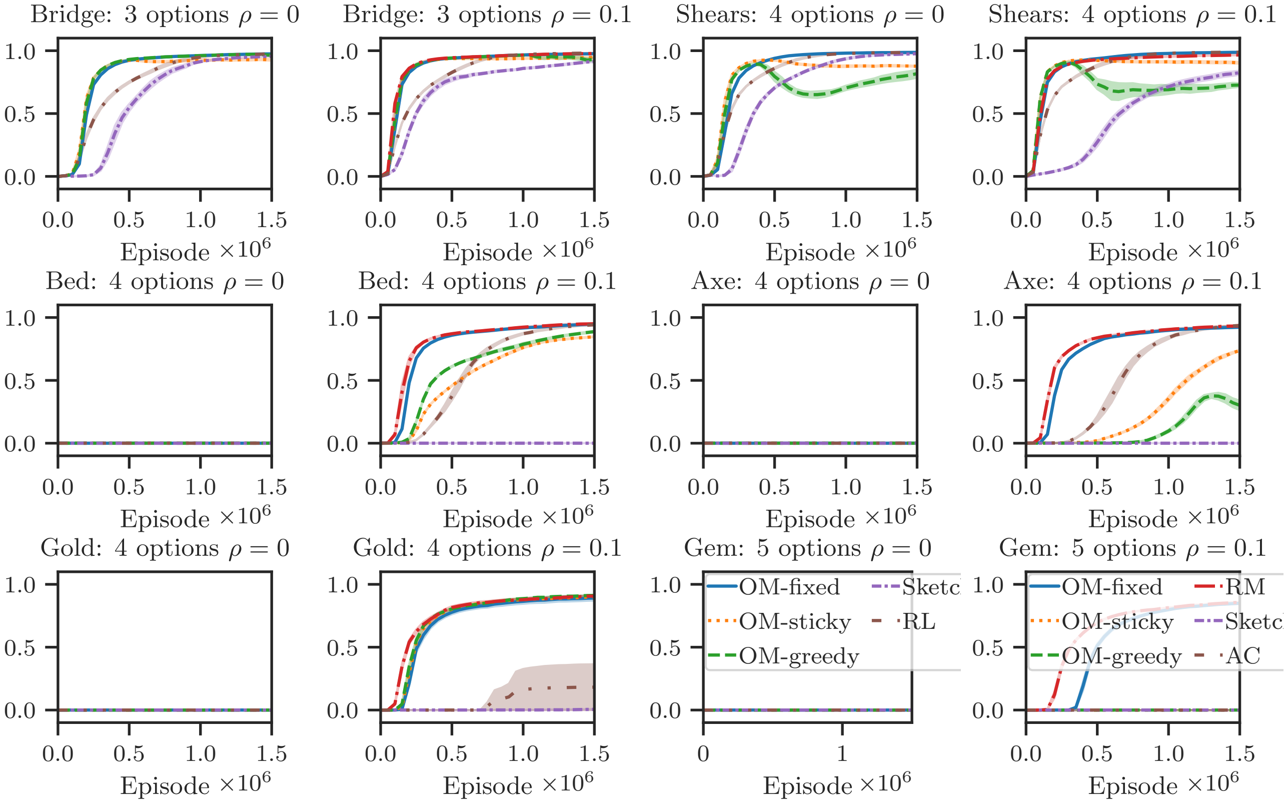

Figure 6: Total cumulative reward on craft world per task. The tasks 'bed', 'axe', 'gold' and 'gem' cannot be learned in the single-task setting if reward shaping is not applied as evidenced by graphs in the first and third columns. Reward shaping with our framework allows the learner to solve these hard problems. Additionally, the version of our framework with deterministic options and shaping (bottom right) is the only solution that learns to solve the 'gem' task.

Training	ρ	Model	WW	WS	ES	NW	NEN	ESS	SEN	WWS	WNE
holdout	0	Determ.				N/A				0.94	0.15
		Sketch								0.85	0.14
	0.1	Determ.								0.49	0.08
		Sketch								0.87	0.17
isolation	0	Determ.	0.98	0.94	0.30	0.93	0.08	0.07	0.03	0.91	0.84
		Sketch	0.83	0.00	0.01	0.02	0.00	0.00	0.03	0.00	0.00

Table 5: Maze environment zero-shot task completion rates for 1K evaluations, averaged over 5 random seeds. Each task consist of a sequence of rooms to reach in the cardinal directions 'North', 'South', 'West', 'East'. The task 'WNE', for example, consists of moving one room 'West', one room 'North' and one room 'East' in that order.

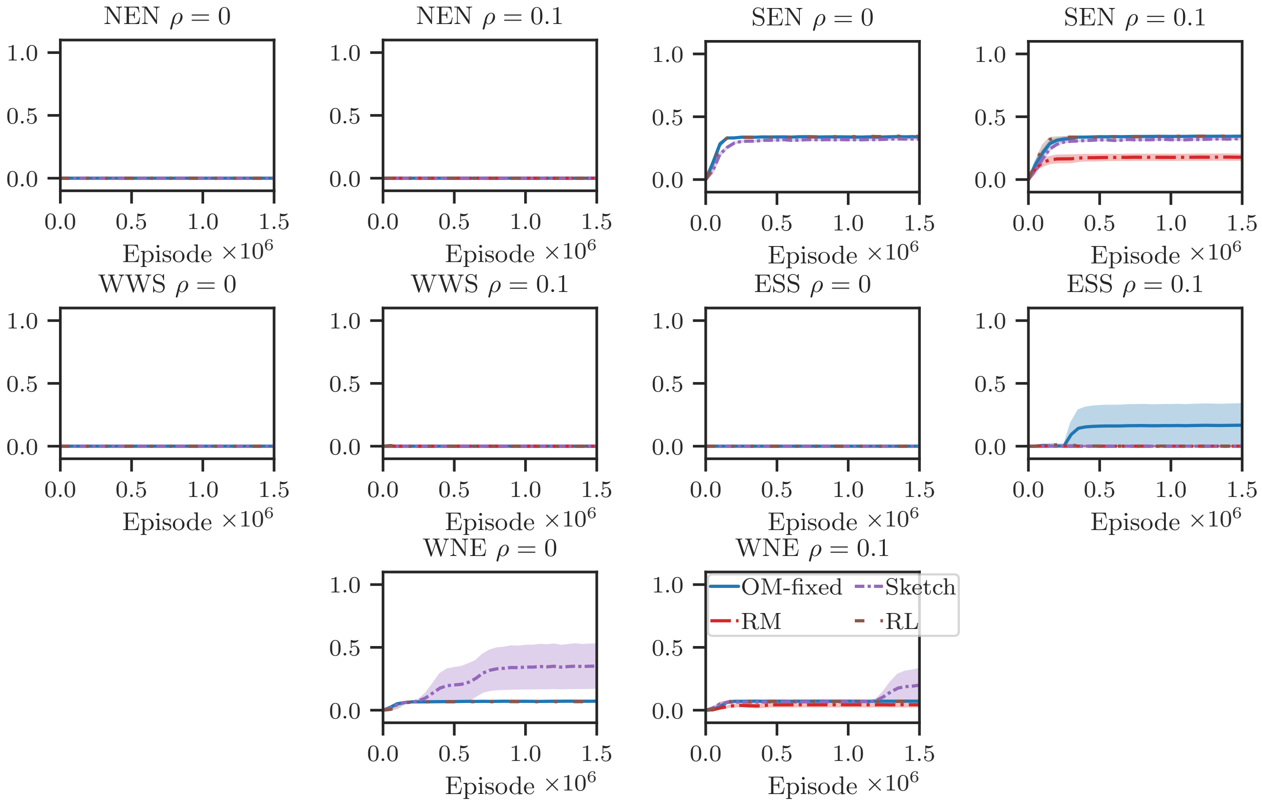

Figure 7: Task completion on maze world per task. Each task consist of a sequence of rooms to reach in the cardinal directions 'North', 'South', 'West', 'East'. The task 'NEN', for example, consists of moving one room 'North', one room 'East' and one room 'North' in that order.

Training	ρ	Model	Plank	Stick	Rope	Cloth	Bridge	Shears	Bed	Axe
holdout	0	Determ.	N/A						0.94	0.83
		Greedy							0.19	0.06
		Stable							0.29	0.10
		Sketch							0.09	0.00
	0.1	Determ.	N/A						0.87	0.84
		Greedy							0.2	0.06
		Stable							0.3	0.14
		Sketch							0.06	0.00
isolation	0	Determ.	0.96	0.80	0.97	0.97	0.94	0.91	0.91	0.76
		Greedy	0.96	0.80	0.97	0.97	0.94	0.94	0.15	0.01
		Stable	0.96	0.80	0.97	0.97	0.95	0.95	0.26	0.11
		Sketch	0.00	0.00	0.17	0.00	0.03	0.02	0.00	0.00

Table 6: Craft environment zero-shot results per task for 1K evaluations, averaged over 5 random seeds.