Learning Reward Structure with Subtasks in Reinforcement Learning

Shuai Han
Utrecht University
Utrecht, the Netherland
s.han@uu.nl

Mehdi Dastani
Utrecht University
Utrecht, the Netherland
m.m.dastani@uu.nl

Shihan Wang
Utrecht University
Utrecht, the Netherland
s.wang2@uu.nl

Abstract

Improving sample efficiency of Reinforcement Learning (RL) in sparse-reward environments poses a significant challenge. In scenarios where the reward structure is complex, accurate action evaluation often relies heavily on precise information about past achieved subtasks and their order. Previous approaches have often failed or proved inefficient in constructing and leveraging such intricate reward structures. In this work, we propose an RL algorithm that can automatically structure the reward function for sample efficiency, given a set of labels that signify subtasks. Given such minimal knowledge about the task, we train a high-level policy that selects optimal subtasks in each state together with a low-level policy that efficiently learns to complete each sub-task. We evaluate our algorithm in a variety of sparse-reward environments. The experiment results show that our method significantly outperforms the state-of-art baselines as the difficulty of the task increases.

1 Introduction

As a powerful technique to optimize the intelligent behaviors of agents, reinforcement learning (RL) has been applied in a variety of domains, such as traffic signal control [6, 24], chemical structure prediction [26, 40], radio resource management [46, 10] and games [30]. The successful training of RL agents often relies on reward functions that are designed based on domain knowledge. Such reward functions allow agents to receive immediate reward signals. Without those handcrafted signals, the sparse rewards can result in RL algorithms suffering from low sample efficiency [1, 19]. Numerous methods have been proposed to enhance sample efficiency of RL in sparse-reward environments, such as building goal-conditioned reinforcement learning (GCRL) to provide intrinsic rewards [14, 28, 7], applying hierarchical reinforcement learning (HRL) for improving credit assignment [35, 45, 27], or employing reward machines (RM) to expose the structure of reward functions [22, 4, 3].

In many scenarios, accomplishing a task involves the sequential completion of multiple subtasks. Especially in sparse-reward settings where immediate feedback is scarce, evaluating action selection relies heavily on precise information about past subtask completions and their specific order. However, previous methodologies, such as GCRL and HRL, do not incorporate precise information about the sequential order of subtasks into their policy learning frameworks. On the other side, the recently proposed RM specify the reward function structure as an automaton [22, 32], which provides crucial information about the sequential nature of subtasks. In such an automaton, nodes represent the completion of subtasks, while the transitions between nodes signify the order in which subtasks are to be achieved. Incorporating such information during policy learning can significantly improve the sample efficiency of RL algorithms [44]. Nevertheless, for complex applications, the reward structure is not always available due to the lack of sufficient domain knowledge [20, 41]. To construct a reward machine for a given set of subtasks, previous methods have proposed to use automata learning to infer a automaton to describe and exploit the reward function structure [43, 20]. However, learning an exact automaton from trace data is a NP-complete problem [17]. Although heuristic methods can be used to speed up the learning [43], inferring an automaton that is representative to the reward structure relies on trace data which is collected by an adequate exploration. When the exploration of agents is inadequate, the automaton derived from the incomplete trace data could be either inaccurate or partial, which leads to the RL algorithm learning sub-optimal policies or even failing to learn.

Aiming at improving sample efficiency of RL in the above-mentioned sparse-reward scenarios that involve sequential completion of multiple subtasks, we propose a novel algorithm, which we call Automatically Learning to Compose Subtasks (ALCS). It automatically learns the structure of reward based on a given set of subtasks (i.e. constituting the minimal domain knowledge of the task). The key idea of ALCS is to learn the best sequences of subtasks to achieve the learning task. To accomplish this, we develop a framework with two-level hierarchy of policy learning. The low-level policy learns to take the next action toward completing a given subtask, while the high-level policy learns to specify a subtask to be achieved next. There are two main characteristics of the high-level policy learning. One is that the next subtask is selected based on the exact sequence of completed subtasks, which considers precise information about subtask sequences during learning. Another characteristic is that at the end of an episode, the subtasks selected by the high-level policy are modified based on the subtasks actually achieved by the low-level policy. This is necessary to consider the impact of all achieved subtasks on the reward gains so that those achieved subtasks can be reinforced as the policy selection. We verify the performance of our method on 8 sparse-reward environments. The results show that when the difficulty of tasks increases, our method produces a significant improvement over the previous most sample-efficient methods. We also analyze the good interpretability of ALCS. The results reveal that when there are multiple possible sequences of subtasks to complete a task, ALCS is able to interpret all these sequences and indicate the best one.

2 Related Work

Among the plethora of work on sample efficiency [44, 18, 31, 29, 25], we summarize the literature in three subfields, which our algorithm closely relates to.

Reward machines. By specifying reward structure with Mealy Machines, named reward machines, QRM [22] is proposed to improve the sample efficiency for RL by assuming a RM to expose reward structure to the learning agent. The integration of reward machines in RL has lead to a series of proposals for exploration [3], reward shaping [4], offline learning [37] and multi-agent learning [32, 9]. When the domain knowledge minimally structures the reward function, (i.e., only specifying the possible subtasks), the RM assumed by these methods are hard to construct. In order to address this problem, methods are proposed to learn the unknown reward structure of the RL task from trace data. LRM [41] uses discrete optimization method to learn the RM in partially observable settings. JIRP [43] adapts classical automata learning algorithms [33, 23] to learns the RM. SRM [8] extends the RM learning to environments containing noisy rewards. ISA [15] uses inductive logic programming to learn reward structures for RL tasks. DeepSynth [20] employs an automata synthesis method to obtain the reweard structure. However, all those methods require at least one positive trace (i.e., the task is solved in such trace) to learn a representative reward structure that is helpful for policy learning. This requirement is not easy to fulfill because in a sparse-reward environment, obtaining the positive trace can be difficult. Besides, the automata synthesis method employed in DeepSynth is NP-complete, which could be computationally expensive when the reward structure of the task is complex.

Goal-conditioned reinforcement learning. Goal-conditioned reinforcement learning (GCRL) trains an agent to achieve different goals [5, 28, 7] by training a goal-conditioned policy. The multiple goals provide intrinsic rewards to train self-motivated agents in sparse-reward environments [1]. Since rewards are intrinsic, the algorithm can compute multiple rewards towards different goals for a single transition to reuse the data, thus improving sample efficiency. The training of goal-conditioned policy can be improved with efficient techniques. For example, [14] generates a series of goals of varying difficulty levels based on a curiosity-driven approach to constitute easy-to-difficult curriculum learning. [13] applies contrastive representation learning to action-labeled trajectories to constitute goal-conditioned value function. [12] constructs causal graphs through causal discovery and in this way improves the generalization of GCRL. GCRL trains policies conditioned on goals to be achieved. To the best of our knowledge, there are no GCRL methods that train policies conditioned on goals that have been completed previously, which makes it difficult for GCRL to efficiently learn tasks that require some exact sequential completion of multiple subgoals.

Hierarchical reinforcement learning. Hierarchical Reinforcement Learning (HRL) methods exploit temporal abstraction [35] or spatial abstraction [45] to improve sample efficiency of RL. When designing the two-level policy, we are inspried by HRL with temporal abstraction, such as HAM [34], MAXQ [11] and h-DQN [27]. These methods abstract the environment as a Semi-MDP [39], which specifies different time scales for different level of policies. A specific HRL method is Interrupting Options [39], where the option is interrupted in extreme case at every step so that the high-level policy and low-level policy make decisions at the same time scales. HRL approaches select subtasks based on the environment state, which leads to existing HRL methods being inappropriate for solving the problem of structuring rewards using domain knowledge in the case where the agent needs to complete subtasks in some order.

We propose ALCS which uses the idea from the mentioned three subfields. In particular, we follow RM and incorporate the precise information of the sequential order of completed subtasks into policy learning. Moreover, inspired by HRL, we train two-level policies and follow the technique of GCRL to reuse the data to train multiple policies jointly. ALCS differs from RM because ALCS does neither assume a given RM nor infer a reward machine for the learning agent. By training low-level policies with subtasks, ALCS does not rely on positive trace to start the policy learning, which makes ALCS more sample efficient than those methods that infer a reward machine. Besides, the high-level policy of ALCS takes into account the sequence of historically completed subtasks when making decisions, which makes ALCS different from existing HRL methods. Moreover, ALCS trains policies conditioned not only on a subtask to be achieved but also on a sequence of completed subtasks, which is different from existing GCRL methods. We have a detailed analysis of the distinctions and merits of our method in Appendix A.

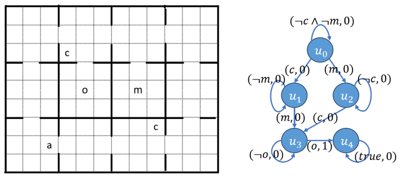
3 Automatically Learning to Compose Subtasks

3.1 Problem Setting

The RL problem considers an agent interacting with an unknown environment [38]. Such environment can be modeled as a finite Markov Decision Process (MDP), $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma)$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition function, $\gamma \in [0, 1)$ is a discount factor and $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a reward function. An agent employs a deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ to interact with the environment. At a time step t, the agent takes action $a_t = \pi(s_t)$ according to the current state s_t. The environment state will transfer to next state s_{t+1} based on the transition probability $\mathcal{T}$. The agent will receive the reward $r_t = R(s_t, a_t, s_{t+1})$. Then, the next round of interaction begins. The goal of this agent is to find the optimal policy π^* that maximizes the expected return: $\pi^* = argmax_\pi \mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t | \pi]$, where T is the terminal time step. The Q function for policy π on a state-action pair (s, a) is defined as: $Q(s, a) = \mathbb{E}_{\pi,\mathcal{T}}[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a]$. When the estimated Q function converges to the optimal Q^*, policy $\pi(s) = argmax_a Q^*(s, a)$ is an optimal policy.

In this work, we extend the standard RL problem with domain knowledge consisting of a finite vocabulary set $\mathcal{P}$ and a labeling function $L : S \to \mathcal{P} \cup \{\emptyset\}$. A vocabulary $p \in \mathcal{P}$ is seen as a subtask that is potentially helpful for achieving the final learning task. Given $\mathcal{P}$ and the labeling function L, $L(s) = p$ means that p is the subtask that is achieved at state s and $L(s) = \emptyset$ means that no subtask is achieved at state s. Note that $L(s)$ does not represent the subtasks that have been achieved prior to state s.

We demonstrate this using an example in the *OfficeWorld* domain [22]. As shown in Figure 1(a), the vocabulary set $\{c, m, o\}$ specifies three possible subtasks in this environment. During interaction with the environment, agents can use L to detect whether a subtask has been completed at s. For example, $L(s) = c$ or $L(s) = o$ mean the subtask 'picking up coffee' or 'arriving at office' is achieved by the agent at s respectively. When $L(s)$ is $\emptyset$, no subtask achieved at s.



(a) (b)

Figure 1: (a) *Coffee&mail* task. The 'a', 'c', 'm' and 'o' indicates the agent, coffee, mail and office respectively. The agent receives reward only upon reaching the 'o' after reaching 'c' and 'm'. (b) RM introduced by [22] to expose the reward structure of this task.

In recent literature, the solution to such a problem is to specify [22] or learn [43, 20] a reward machine. An example of a reward machine for *Coffee&mail* task is shown in 1(b). An RL agent with this reward machine starts from the beginning of an episode (u_0). If at any MDP state s that $L(s) = c$, then the RL agent moves to u_1 with reward 0. Similarly, if at u_3 the RL agent encounters an MDP state s that $L(s) = o$ then the RL agent moves to terminal state u_4 with reward 1. With such a reward machine, previous methods such as [22, 43, 20] learn Q-values over the cross-product $Q(s, u, a)$, which allows the agent to consider the MDP state s and RM state u to select the next action a [22]. In our setting, how the subtasks contribute to rewards is not required and given. Instead, the knowledge given by $\mathcal{P}$ and L is limited to subtasks (not to, for example, their ordering structures). Given such minimal knowledge of the learning task, our algorithm will learn to capture these knowledge during the training.

3.2 Two-level policy formalization

In the framework of ALCS, a high-level policy is designed to select a next subtask to be achieved based on the environment state and the sequence of subtasks that have already achieved in the history of this episode. We denote it as $\pi_h : \mathcal{S} \times \mathcal{P}^* \rightarrow \mathcal{P}$, where $\mathcal{P}^*$ is the Kleene closure on $\mathcal{P}$. For example, in Figure 1 where $\mathcal{P} = \{c, m, o\}$, the Kleene closure on $\mathcal{P}$ is $\mathcal{P}^* = \{\emptyset, c, m, o, cc, cm, co, mc, mm, mo, oc, om, oo, ccc, ...\}$. Given a state s, π_h selects the next subtask p by:

$$p = \pi_h(s, p^*) \tag{1}$$

where the sequence $p^* \in \mathcal{P}^*$ represents the order of subtasks that have been temporally achieved in the history of a given episode. The completed subtasks and their order are key information in the learning tasks that requires sequential completion of subtasks. Therefore, the agent will benefit from training the policy π_h with such sequence being considered. Moreover, the learned behavior based on p^* empowers the high-level policy to effectively support non-Markovian decision-making for a current state.

On the other hand, a low-level policy, denoted as $\pi_l : \mathcal{S} \times \mathcal{P} \rightarrow \mathcal{A}$, is designed to learn to achieve the selected subtask efficiently. Taking the current state and a subtask given from high-level policy, π_l decides an environment action to achieve the subtask. π_l selects actions by: $a = \pi_l(s, p)$. With the two-level policy, our agent interacts with the environment as follows. At the beginning of an episode, the agent initializes an empty sequence p^* to store the achieved subtasks in the environment. At each time step t, the agent first employs the high-level policy to select a subtask p_t to be achieved based on the current state s_t and the achieved subtask sequence p_t^*, i.e., $p_t = \pi_h(s_t, p_t^*)$. Then, the agent uses the low-level policy to choose action $a_t = \pi_l(s_t, p_t)$ to interact with the environment. When a subtask is achieved in s_{t+1}, it will be appended into the sequence:

$$p_{t+1}^* = \begin{cases} p_t^* \oplus L(s_{t+1}) & \text{if} \quad L(s_{t+1}) \neq \emptyset \\ p_t^* & \text{otherwise} \end{cases} \tag{2}$$

where $\oplus$ represents appending $L(s_{t+1})$ into the end of sequence p_t^*.

We next describe the definition of the Q functions of two policies and the detailed corresponding training process.

3.3 Low-level training

The goal of low-level policy π_l in ALCS is to achieve the given subtask $p \in \mathcal{P}$. Therefore, π_l is not trained with respect to the MDP reward function R. For a certain $p \in \mathcal{P}$, $\pi_l(s, p)$ is trained with the following rewards:

$$R^p(s_t, a_t, s_{t+1}) = \begin{cases} 1 & \text{if} \quad p = L(s_{t+1}) \quad \text{and} \quad p \neq L(s_t) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

According to Equation (3), it can be known that for different subtasks, π_l is trained with different reward functions. Given Equation (3), we define the low-level Q function $Q_l(s, p, a)$ as the excepted return based on the subtask p: $Q_l(s, p, a) = \mathbb{E}_{\pi_l, \mathcal{T}}[\sum_{t=0}^{T} \gamma^t R^p(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a]$. With this low-level function Q_l, π_l can select actions by doing argmax operation: $a_t = argmax_a Q_l(s_t, p, a)$ at time step t. In the training, Q_l is updated with experience $(s_t, a_t, s_{t+1}, r_t^p)$, where $r_t^p = R^p(s_t, a_t, s_{t+1})$ is the reward value with respect to the subtask p.

We observe that the reward function in Equation (3) is computed by labeling function L. Inspired by goal relabeling [1, 28] and counterfactual experiences [22], to improve the training efficiency, we use L to generate multiple experience for different subtasks to update Q_l, instead of using a single experience $(s_t, a_t, s_{t+1}, r_t^p)$. To do so, for a single transition (s_t, a_t, s_{t+1}) sampled from the environment, we can generate a set of experiences: $\{(s_t, a_t, s_{t+1}, r_t^p)\}$ for all $p \in \mathcal{P}$, where $r_t^p = R^p(s_t, a_t, s_{t+1})$ is the reward value with respect to subtask p at time step t. We put the pseudo-code for training the low-level policy and a detailed description of it in the appendix B.

3.4 High-level training

The goal of the high-level policy π_h in ALCS is to compose subtasks for the original task in MDP. So π_h is designed to be trained with the MDP reward function. The corresponding Q function defined as the excepted return from the environment following π_h and π_l: $Q_h(s, p^*, p) = \mathbb{E}_{\pi_l, \pi_h, \mathcal{T}}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = \pi_l(s, p)]$. With this high-level function Q_h, π_h selects a subtask p_t by doing the argmax operation: $p_t = argmax_p Q_h(s_t, p_t^*, p)$ for given s_t and p_t^*. In the decision-making process of selecting the subsequent subtask p_t for completion, ALCS considers the sequence of previously accomplished subtasks. This functionality empowers the high-level policy to determine the execution order of subtasks, thereby facilitating ALCS in its support for non-Markovian decision-making capabilities.

In the general RL methods, Q function should be updated based on the trajectories collected from exploration. However, when the trajectories of high-level policy in exploration are applied to training, it may lead to key subtasks in terms of the task completion not being learned. For example, a subtask p_t is chosen by $\pi_h(s_t, p_t^*)$. When $\pi_l(s_t, p_t)$ takes an action to achieve p_t, this action causes another subtask p' being completed. In this case, if p' is the key subtask that brings reward in future time step, the importance of p' to reward cannot be learned. Because $\pi_h(s_t, p_t^*)$ does not choose p' as its decision and therefore p' does not appear in the exploration trajectories of π_h.

To solve this problem, we propose to generate the high-level experiences based on the actual completed subtask in the environment (subtask p' in the above example), instead of based on the subtask selected by the high-level policy (subtask p). In other words, the actual completed subtask will be rewarded and the corresponding data $((s_t, p_t^*), p', (s_{t+1}, p_{t+1}^*), r_t)$ is used for the update. Whenever a subtask is detected as 'achieved' by the labeling function, we assume that π_h has chosen this subtask. And experiences will be generated to update Q_h based on this assumption. An example for generating experiences in an episode is shown in Figure 2. We include the pseudo-code for training the high-level policy and a detailed description of it in the appendix C.

4 Experiments

We evaluate our method in 8 different sparse-reward environments from the two commonly used domains [1], *OfficeWord* and *MineCraft* [22, 43]. We introduce the characteristics of these environments and the reasons for choosing them as follows. **Coffee** (Go to office after taking a coffee), **Coffee**&**Mail** (Go to office after taking both a coffee and a mail. The order of taking the coffee and mail does not matter), **Collecting** (Go to office after collecting four packages A, B, C and D in the

[1]The code is released on: https://github.com/shan0126/ALCS/tree/master.

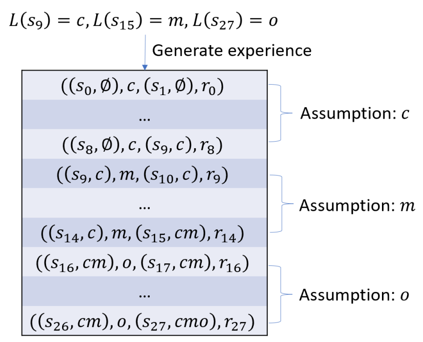

$L(s_9) = c, L(s_{15}) = m, L(s_{27}) = o$
Generate experience
$((s_0, \emptyset), c, (s_1, \emptyset), r_0)$
...
$((s_8, \emptyset), c, (s_9, c), r_8)$
$((s_9, c), m, (s_{10}, c), r_9)$
...
$((s_{14}, c), m, (s_{15}, cm), r_{14})$
$((s_{16}, cm), o, (s_{17}, cm), r_{16})$
...
$((s_{26}, cm), o, (s_{27}, cmo), r_{27})$
Assumption: c
Assumption: m
Assumption: o

Figure 2: Generating the high-level experiences for Q_h in the task of Figure 1. Each row contains a transition: $((s_t, p_t^*), p_t, (s_{t+1}, p_{t+1}^*), r_t)$, where p_t here is the assumed selected subtask and r_t is the environmental reward. In this example, subtask c, m, and o are achieved at time step 9, 15, and 27, respectively. Then, the assumed subtask p_t selected by π_h are c from time step $0 \sim 8$, m from time step $9 \sim 14$, and o from time step $15 \sim 27$.

corners. The order of taking those packages does not matter), **Bonus** (Same task with the Collecting environment. The agent will receive extra bonus if all packages are collected), **Plant** (Get wood, use toolshed), **Bridge** (Get iron, get wood, use factory. The agent recieves 1 reward after achieving this task. The iron and wood can be gotten in any order), **Bed** (Get wood, use toolshed, get grass, use workbench. The grass can be gotten at any time before using the workbench), and **Gem** (Get wood, use workbench, get iron, use toolshed, use axe. The iron can be gotten at any time before using the toolshed). In those environments, the former four environments are from the *OfficeWord* domain, while the latter four are from *MineCraft*. The agent in all environments expected for Bonus is only rewarded when completing the task (sparse-reward environments). These environments have increasing task complexity in terms of the number of subtasks and states, which is helpful to show how the performance of methods as the task difficulty increases. Moreover, *OfficeWord* and *MineCraft* have 108 and 400 discrete states, respectively. Comparing the performance of methods in the two domains can show the impact of the increased state space on the performance.

In addition, Bonus is a special environment. In this environment, the agent can get 1 reward for every package, but is preferred to collect all the packages. Note that the goal in the Bonus environment is too sparse for providing rewards. Thus, in this environment, there is also an environment reward for completing individual subtasks (collecting one or more packages), which helps the methods learn the skill of collecting packages. But at the same time those rewards can lead to the agent misalignment policy from learning to collect all packages to learning to collect individual packages. Therefore in this environment, it is challenging for the RL agent to utilize the subtask reward without such policy misalignment.

We compare the following seven different methods as follows. **ALCS** is a implement prototype with learning rates 0.1 for both Q_l and Q_h. In the training, the rate for $\epsilon-$greedy exploration is 20% to provide stochastic exploration. **JIRP** [43] infers an RM using the $libalf$ library [2] from the RL trajectory. JIRP employs the same level of domain knowledge to our method, i.e., a set of labels that signify subtasks. **DeepSynth** [20] uses automata synthesis [23] to learn a high-level automaton model to guide RL policy learning. Likewise, DeepSynth employs the same level of domain knowledge to our method. **HRL** [27] learns hierarchical policies. We follow the implementation [43] where treats subtasks $p \in \mathcal{P}$ as options and the termination condition of an option is whenever the corresponding p is achieved. **Interrupting options** [39] is a HRL method in which the high-level policy can interrupt low-level policies if there are better subtasks to be completed. Interrupting options is an indispensable method for avoiding the missing of key subtasks. **HER** [1] is a well-known GCRL method. The goal in HER is defined as the subtask $p \in \mathcal{P}$ and the corresponding goal-conditioned reward is the same as Equation (3). HER is an efficient method to train to complete multiple subtasks. **Q learning** [42] is a well-known general RL algorithm. We use the implementation version from [22].

Notably, JIRP and DeepSynth are the state-of-the-art methods on the above-mentioned domains. In general, JIRP learns better when there is only a binary reward at the end of the episode, and DeepSynth is more adaptable in non-binary cases. They all have access to a given set of subtasks, as well as a label function and the return of this function is the identity of the subtask based on the environmental state when this subtask is completed for the first time in an episode.

4.1 Comparison

We first compare our method with baselines on 8 environments from *OfficeWord* and *MineCraft* domains to validate the superiority of ALCS. The results are shown in Figure 3. The error bounds (i.e., shadow shapes) indicate the upper and lower bounds of the performance with 20 runs excluding 2 best and 2 worst results.

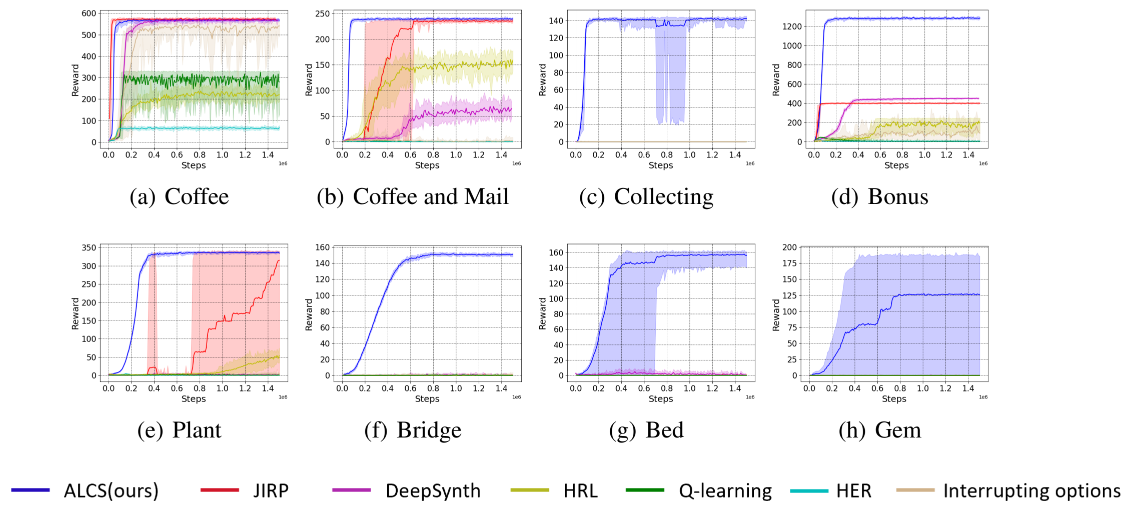


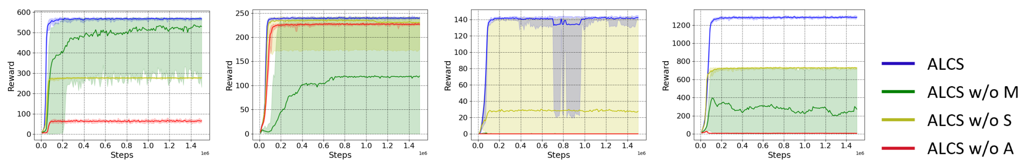
Figure 3: Learning curves of various RL algorithms on 8 environments from *OfficeWord* and *MineCraft* domains.

The results show that algorithms that cannot utilize information about subtasks already performed, such as Q learning, are unable to learn the optimal policy in these domains. Our method significantly outperforms the baseline methods (including the state-of-the-art methods JIRP and DeepSynth) in all environments except Coffee where the reward structure is simple and easy to explore. In such an environment, learning an exact model with JIRP in Coffee environments to specify the reward structure is a better solution. However, when the reward structure of the task becomes complex, the sample efficiency of ALCS can significantly outperform all other methods.

Another conclusion can be drawn by comparing those methods between *OfficeWord* and *MineCraft* domains. It is well known that the sample efficiency of the RL algorithm can decrease on environments with large state space. Comparing the performance of those RL algorithms in *OfficeWord* domain, their performance in the *MineCraft* are reduced because of the larger state space. In contrast, our method is less affected by increased state space compared to the baselines. In addition to the state space, the problem difficulty increases when the number of subtasks increases. By comparing the performance of ALCS across Bridge, Bed and Gem environments, it can be seen that the performance variance of ALCS grows with the increasing task complexity. This suggests that our method can suffer from the exponential growth of subtasks but still be more scalable than state-of-the-art methods. Moreover, in the Collecting environment where all baselines fail on policy learning, these baselines can learn some policies for subtasks when rewards are shaped for completing the collection of one or more packages (in the Bonus environment). However, these shaped rewards can misalign the learned policy from collecting all packages to collecting individual packages for baselines. Our two-layer policy learning can overcome such policy misalignment.

To keep the consistency, all the results in Figure 3 are based on the same learning rate 0.1 as JIRP. We tested the sensitivity of ALCS to high learning rates in the coffee task. The results show that ALCS has a stable performance when the learning rate does not exceed 0.5.

4.2 Ablation study



(a) Coffee (b) Coffee and Mail (c) Collecting (d) Bonus

Figure 4: Learning curves for ablation experiment of ALCS on 4 environments from *OfficeWord*.

In order to explore the contribution that each proposed technique to the whole algorithm, we compare the ALCS with its ablative variants on the *OfficeWord* domain. The results are shown in Figure 4, where 'ALCS w/o M' stands for ALCS algorithm without multiple experiences generating for updating Q_l (as proposed in Subsection 3.3), 'ALCS w/o S' represents ALCS algorithm without sequence of completed subtask for π_h (as presented in Subsection 3.2), and 'ALCS w/o A' indicates ALCS algorithm without the assumed choice of subtask for training π_h (as presented in Subsection 3.4) which can be also seen as the on-policy version of ALCS.

As shown in Figure 4, in case of ALCS w/o M, when not generating multiple experiences for Q_l, the performance of the algorithm drops in all environments due to insufficient learning of the low-level policy, and in Collecting environment, ALCS w/o M cannot even learn. In ALCS w/o S, when π_h learns without a sequence of completed subtask, it cannot select the optimal subtask to be achieved due to the inability to obtain sufficient information on the completed subtasks. In case of ALCS w/o A, we see that the off-policy learning in ALCS with the assumed choice of the subtask is the key to high-level π_h to efficiently learn to compose subtasks. In addition, in the coffee environment, the subtask on picking coffee has a high probability of being completed while executing the subtask on arriving at the office. In this case, the selection of the subtask for picking a coffee will not be learned by the π_h. Therefore, even if the tasks in the Coffee environment are not complex, without assuming a choice of subtask, ALCS cannot to learn good policies.

4.3 Interpretability

How to make the RL algorithms interpretable has widely raised concerns recently for ethical and legal considerations in application [36, 21, 16]. In this section, based on the given minimal domain knowledge, we interpret the behavior of agents trained by ALCS.

In the training process of ALCS, we can use a tree structure to record all the sequences of subtasks that have been achieved. This tree begins with a root node '$\emptyset$'. Each descendant node of the root is a subtask $p \in \mathcal{P}$. The nodes in the path from the root to a descendant can form a sequence $p^* \in \mathcal{P}^*$. In the beginning of an episode, the agent always starts from the root node '$\emptyset$', which means no subtask has been achieved in the history of this episode. Then, if a subtask p is achieved, a new child node with respect to p is added in the tree. Besides, the reward from the environment when subtask p is achieved will be recorded on the edge to the corresponding child node. Given an achieved sequence of subtask p_t^* at time step t, a node associated with the current state can be uniquely identified in the tree. We call this node as a current node at t, denoted as $N(p_t^*)$. At the start of the episode, the agent's current node $N(p_t^*)$ is reset to the root node, i.e. p_t^* is reset to $\emptyset$. An example of this is shown in Figure 5.

With this tree, the RL agent can interpret its behavior based on the given domain knowledge during execution. The interpretation consists of the following three parts. **1) What has already happened.** During execution, the current p_t^* can be seen as an interpretation of what subtasks have happened in which order. **2) Current best subtask.** The current p_t selected by π_h is interpreted as the current best subtask. **3) Planning for the future.** With $p_t^* \oplus p_t$, the agent can localize a node on the tree. By performing a breadth-first search (BFS) on the subtree with this node as root, the agent can find a sequence of subtasks that can obtain the reward. This sequence of subtasks is interpreted as the planning for the future. If the BFS algorithm cannot find a satisfied descendant node in limited depth, then we say there is no interpretation to future plan for the current decision.

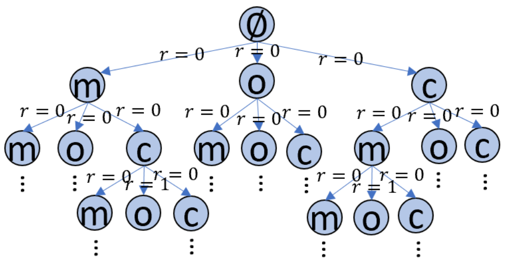


Figure 5: An example tree to record sequences of subtasks during training in the task of Figure 1. For finite MDP the tree is finite as the depth of the tree will be limited by the episode steps and the width will be limited by $|\mathcal{P}|$.

The advantage of this is that the interpretation takes into account both the high-level information constructed by domain knowledge (i.e., the tree) and the environment state information through $p_t = \pi(s_t, p_t^*)$. When the environment state changes, both the behavior and its interpretation could change, which makes the decision-making procedure of an agent more transparent.

We demonstrate the interpretability of our learned policy in the Coffee and Mail environment. In this task, there are two possible sequences of subtasks to bring reward: $c \to m \to o$ and $m \to c \to o$. However, the optimal sequence of subtasks is different when the agent starts from different positions. As shown in Figure 6, starting from position ①, the agent takes fewer steps with the sequence $c \to m \to o$. At position ② $m \to c \to o$ is a better sequence with fewer steps to finish the task. The previous approaches with an automaton in interpretation will consider the two sequences to be equivalent.

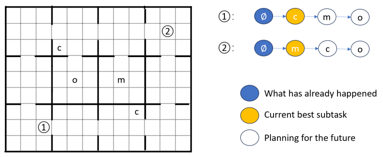


Figure 6: Interpretation on Coffee&Mail. The three colors on the right of the figure represent the three corresponding parts of interpretation.

The high-level policy learned in ALCS can explain the difference, because our high-level policy can output an exact subtask to be achieved based on the environment state. As shown in Figure 6, when the agent starts in different positions, with different state being input into π_h, π_h will select different 'current best subtask' to achieve based on Equation (1). As shown in right of Figure 6, 'c' is selected at ① and 'm' is selected at ②. Then, according to the recording tree, it provides future planning after achieving the selected subtask. These constitute a complete interpretation on how to compose subtasks exactly to finish the task with fewer steps.

5 Conclusion and future work

In this paper, we propose ALCS, an RL algorithm to automatically structure the reward function to improve the sample efficiency in sparse-reward tasks. ALCS uses a two-level policy to learn composing subtasks and to achieve them in the best order for maximizing the excepted return from environments. Besides, three optimization methods are designed for this two-level policy learning framework, which includes providing information of the completed subtask sequence for a better high-level decision, generating multiple experience for sample efficiency, and training high-level policy with assumed selection based on the actually achieved subtasks. We show that in a variety of sparse-reward environments, ALCS significantly outperforms the state-of-the-art methods in environments with high task difficulty and demonstrates the interpretative capacity for indicating the selected sequence among multiple possible sequences. In future work, we want to explore the convergence of the proposed algorithm and to continue our efforts to automatically learn to structure the reward function. In future work, we will extend our method with partial information about the specified subtasks.

References

[1] Marcin Andrychowicz, Dwight Crow, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. pages 5048–5058, 2017.

[2] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R Piegdon. libalf: The automata learning framework. In *Proceedings of the 22nd International Conference on CAV*, pages 360–364, 2010.

[3] Hippolyte Bourel, Anders Jonsson, Odalric-Ambrym Maillard, and Mohammad Sadegh Talebi. Exploration in reward machines with low regret. In *International Conference on Artificial Intelligence and Statistics*, pages 4114–4146, 2023.

[4] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *Proceedings of the Twenty-Eighth IJCAI*, pages 6065–6073, 2019.

[5] Elliot Chane-Sane, Cordelia Schmid, and Ivan Laptev. Goal-conditioned reinforcement learning with imagined subgoals. In *International Conference on Machine Learning*, pages 1430–1440. PMLR, 2021.

[6] Chacha Chen, Hua Wei, Nan Xu, Guanjie Zheng, Ming Yang, Yuanhao Xiong, Kai Xu, and Zhenhui Li. Toward A thousand lights: Decentralized deep reinforcement learning for large-scale traffic signal control. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 3414–3421, 2020.

[7] Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, 2022.

[8] Jan Corazza, Ivan Gavran, and Daniel Neider. Reinforcement learning with stochastic reward machines. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 6429–6436, 2022.

[9] Michael Dann, Yuan Yao, Natasha Alechina, Brian Logan, and John Thangarajah. Multi-agent intention progression with reward machines. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, pages 215–222, 2022.

[10] James Delaney, Steve Dowey, and Chi-Tsun Cheng. Reinforcement-learning-based robust resource management for multi-radio systems. *Sensors*, 23(10):4821, 2023.

[11] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[12] Wenhao Ding, Haohong Lin, Bo Li, and Ding Zhao. Generalizing goal-conditioned reinforcement learning with variational causal reasoning. *Advances in Neural Information Processing Systems*, 35:26532–26548, 2022.

[13] Benjamin Eysenbach, Tianjun Zhang, Sergey Levine, and Russ R Salakhutdinov. Contrastive learning as goal-conditioned reinforcement learning. *Advances in Neural Information Processing Systems*, 35:35603–35620, 2022.

[14] Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided hindsight experience replay. *Advances in neural information processing systems*, 32, 2019.

[15] Daniel Furelos-Blanco, Mark Law, Alessandra Russo, Krysia Broda, and Anders Jonsson. Induction of subgoal automata for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3890–3897, 2020.

[16] Claire Glanois, Paul Weng, Matthieu Zimmer, Dong Li, Tianpei Yang, Jianye Hao, and Wulong Liu. A survey on interpretable reinforcement learning. *arXiv preprint arXiv:2112.13112*, 2021.

[17] E. Mark Gold. Complexity of automaton identification from given data. *Inf. Control.*, 37(3):302–320, 1978.

[18] Zijing Guo, Chendie Yao, Yanghe Feng, and Yue Xu. Survey of reinforcement learning based on human prior knowledge. *Journal of Uncertain Systems*, 15(01):2230001, 2022.

[19] Abhishek Gupta, Aldo Pacchiano, Yuexiang Zhai, Sham Kakade, and Sergey Levine. Unpacking reward shaping: Understanding the benefits of reward engineering on sample complexity. *Advances in Neural Information Processing Systems*, 35:15281–15295, 2022.

[20] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. Deepsynth: Automata synthesis for automatic task segmentation in deep reinforcement learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*, pages 7647–7656, 2021.

[21] Alexandre Heuillet, Fabien Couthouis, and Natalia Díaz-Rodríguez. Explainability in deep reinforcement learning. *Knowledge-Based Systems*, 214:106685, 2021.

[22] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *J. Artif. Intell. Res.*, pages 173–208, 2022.

[23] Natasha Yogananda Jeppu, Thomas Melham, Daniel Kroening, and John O'Leary. Learning concise models from long execution traces. In *57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[24] Qize Jiang, Minhao Qin, Shengmin Shi, Weiwei Sun, and Baihua Zheng. Multi-agent reinforcement learning for traffic signal control through universal communication method. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, pages 3854–3860, 2022.

[25] Mu Jin, Zhihao Ma, Kebing Jin, Hankz Hankui Zhuo, Chen Chen, and Chao Yu. Creativity of ai: Automatic symbolic option discovery for facilitating deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7042–7050, 2022.

[26] Hwanhee Kim, Soohyun Ko, Byung Ju Kim, Sung Jin Ryu, and Jaegyoon Ahn. Predicting chemical structure using reinforcement learning with a stack-augmented conditional variational autoencoder. *J. Cheminformatics*, 14(1):83, 2022.

[27] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[28] Minghuan Liu, Menghui Zhu, and Weinan Zhang. Goal-conditioned reinforcement learning: Problems and solutions. *arXiv preprint arXiv:2201.08299*, 2022.

[29] Daoming Lyu, Fangkai Yang, Bo Liu, and Steven Gustafson. Sdrl: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *Proceedings of the AAAI*, pages 2970–2977, 2019.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[31] Thomas M Moerland, Joost Broekens, Aske Plaat, Catholijn M Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

[32] Cyrus Neary, Zhe Xu, Bo Wu, and Ufuk Topcu. Reward machines for cooperative multi-agent reinforcement learning. In *Proceedings of the 2021 International Conference on Autonomous Agents and Multiagent Systems*, pages 934–942, 2021.

[33] Daniel Neider and Nils Jansen. Regular model checking using solver technologies and automata learning. In *NASA Formal Methods Symposium*, pages 16–31. Springer, 2013.

[34] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

[35] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54(5):1–35, 2021.

[36] Erika Puiutta and Eric MSP Veith. Explainable reinforcement learning: A survey. In *International cross-domain conference for machine learning and knowledge extraction*, pages 77–95, 2020.

[37] Haoyuan Sun and Feng Wu. Less is more: Refining datasets for offline reinforcement learning with reward machines. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 1239–1247, 2023.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[39] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[40] Luca A. Thiede, Mario Krenn, AkshatKumar Nigam, and Alán Aspuru-Guzik. Curiosity in exploring chemical spaces: intrinsic rewards for molecular reinforcement learning. *Mach. Learn. Sci. Technol.*, 3(3):35008, 2022.

[41] Rodrigo Toro Icarte, Ethan Waldie, Toryn Klassen, Rick Valenzano, Margarita Castro, and Sheila McIlraith. Learning reward machines for partially observable reinforcement learning. *Advances in neural information processing systems*, 32, 2019.

[42] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[43] Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. Joint inference of reward machines and policies for reinforcement learning. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling*, pages 590–598, 2020.

[44] Chao Yu, Xuejing Zheng, Hankz Hankui Zhuo, Hai Wan, and Weilin Luo. Reinforcement learning with knowledge representation and reasoning: A brief survey. *CoRR*, abs/2304.12090, 2023.

[45] Mehdi Zadem, Sergio Mover, et al. Goal space abstraction in hierarchical reinforcement learning via set-based reachability analysis. In *2023 IEEE International Conference on Development and Learning (ICDL)*, pages 423–428. IEEE, 2023.

[46] Mohammad Zangooei, Niloy Saha, Morteza Golkarifard, and Raouf Boutaba. Reinforcement learning for radio resource management in RAN slicing: A survey. *IEEE Commun. Mag.*, 61(2):118–124, 2023.

A ALCS revisited

Like previous methods inferring an RM for policy learning, ALCS aims to improve the sample efficiency for sparse-reward learning tasks that involve sequentially completing multiple subtasks by constructing the reward structure. Such problems with complex reward structures have not typically been addressed by previous HRL and GCRL methods. This subsection will revisit how ALCS leverages the strengths from RM methods while surmounting challenges that are difficult to overcome with the HRL and GCRL methodologies.

A reward machine succinctly exposes the structure of a reward function to the RL agent via an automaton, as shown in Figure 1(b). This reward structure reveals the necessary subtasks and their ordering through paths from the start state to the reward state in the automaton. Following such structure, an abstract state on these paths characterizes an intermediate status towards completing the subtasks in some order. For example, u_1 represents the intermediate status that ‘c’ has been achieved and u_3 represents that both ‘c’ and ‘m’ have been achieved. Augmenting the MDP state with such high-level abstract state provides key information of achieved subtasks in history to support low-level

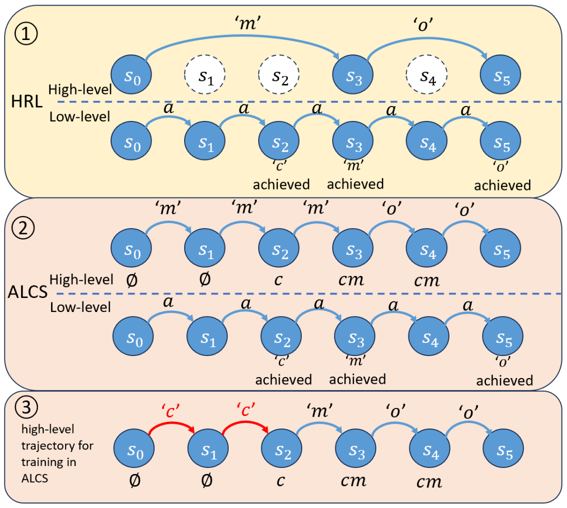


Figure 7: Example to show different methods in the *Coffee&mail* task. ① and ② show the decision process of two-level policy in previous HRL methods and in the proposed ALCS, respectively. ③ indicates the high-level trajectory for training generated from the decision process in ②.

decision-making. Inspired by this, in the absence of given RM, we argue to enable RL agents to make decisions based on achieved subtasks as well.

To do this, we develop a two-level policy framework to make decisions and learn to automatically structure the reward function by composing subtasks. A case to show how ALCS make decisions in an episode is shown in ② of Figure 7. The high-level policy first selects a subtask to be achieved at each time step based on the current MDP state augmented with previously completed subtasks. Then the low-level policy selects an action to interact with the environment based on the current MDP state and the subtask chosen by the high-level policy. For example, at s_3, the high-level policy selects subtask o based on the MDP state as well as on previously completed subtasks c and m, then the low-level policy selects an action to achieve o based on s_3. As an episode ends, the high-level decision for each time step on this episode trajectory will be modified based on the subsequently completed subtask according to subsection 3.4. For example, in ③ of Figure 7, the decisions at s_0 and s_1 are modified as c based on the subsequently achieved subtask at s_2.

ALCS is quite different compared to previous HRL methods, the decision process of which is demonstrated in ① of Figure 7. In the common case of HRL [11, 27], at s_0 the high-level policy first selects a subtask to be achieved. The low-level policy will persistently aim to complete this subtask as the goal to select actions until the subtask is achieved. Then the high-level policy selects a subtask based on the current state s_3 again. By allowing the two level policies to make decisions on different time scales, the higher level obtains a shorter trajectory $\{s_0, `m', s_3, `o', s_5\}$, which enables efficient credit assignment over long timescales [35].

However, this representative HRL approach is not suitable for learning to automatically structure reward function with subtasks. On the one hand, high-level policy in HRL always makes decisions based on the current MDP state, which does not contain precise information about completed subtasks as the RM state does. Thus this high-level decision making is limited in its contribution to the overall action selecting of the RL agent. On the other hand, HRL uses an abstracted MDP trajectory to train the high-level policy, which can lead the learning of high-level policies to miss key subtasks. As in Figure 7, although the agent completes the task, the high-level learning in HRL only reinforces the selection of subtasks ‘m’ and ‘o’, missing another key subtask ‘c’ for the underlying reward structure. As a contrast, such critical subtasks are not missed in the ALCS training.

Besides, what subtask the policy is based on distinguishes ALCS from previous GCRL methods. GCRL methods train policies based on the subtask to be accomplished. And how to select the subtask to be accomplished is either by means of a predefined subtask selection function or by using an HRL approach to learning [7]. This still prevents GCRL from automatically learning composing subtasks with specific order for a given task. And although the low-level policy in ALCS takes a similar decision-making approach based on the subtasks to be completed as GCRL, the high-level policy in ALCS learns automatically to select the next subtask to be achieved based on the exact sequence of completed subtasks. The high-level policy is conditioned on completed subtasks instead of a subtask to be achieved. To the best of our knowledge, this is different from all previous GCRL methods and gives ALCS the ability to automatically compose subtasks with valid orders.

B Generating multiple experiences and updating Q_l

Algorithm 1 Generating multiple experiences and updating Q_l.

```
    Input: Transition $(s_t, a_t, s_{t+1})$, original $Q_l$, learning rate $\alpha$
 1: Initialize $experiences \leftarrow \{\}$
 2: for $p \in \mathcal{P}$ do
 3:     if $p = L(s_{t+1})$ and $p \neq L(s_t)$ then
 4:         $r_t^p \leftarrow 1, done \leftarrow True$
 5:     else
 6:         $r_t^p \leftarrow 0, done \leftarrow False$
 7:     end if
 8:     Add tuple $(s_t, a_t, s_{t+1}, r_t^p, p, done)$ into $experiences$
 9: end for
10: for $(s, a, s', r, p, done)$ in $experiences$ do
11:     if $done$ then
12:         $y \leftarrow r$
13:     else
14:         $y \leftarrow r + \gamma \max_{a'} Q_l(s', p, a')$
15:     end if
16:     $Q_l(s, p, a) \leftarrow (1 - \alpha) Q_l(s, p, a) + \alpha \cdot y$
17: end for
```

Algorithm 1 describes the detailed practice for updating Q_l with multiple experiences generated from single transition. Given an input transition (s_t, a_t, s_{t+1}), Algorithm 1 in Lines 2 $\sim$ 7 assigns different rewards following Equation 3 for all possible $p \in \mathcal{P}$ to this transition, which yields a collection of experiences. Then those experiences will be used to update Q_l in Lines 10 $\sim$ 17. After sufficiently updating for Q_l, given a subtask p, π_l can repeatedly perform this subtask. But this is generally not enough to maximize the environment rewards. So we also need to train the π_h, which is responsible for learning to compose subtasks in any order to maximize the environment rewards.

C Automatically Learning to Compose Subtasks

Algorithm 2 shows the overall training procedure of our method. When an episode begins, we initialize some variables in Lines 2 $\sim$ 4, where $experience\_h$ is the experiences for this episode to update Q_h, e_temp is a temporary experience set, and p_t^* stores the sequence of achieved subtasks at the time step t. After the initialization for an episode, the algorithm will repeat the process in Line 6 $\sim$ 36 until the episode terminates. In each episode, the agent first choose a subtask p_t in Line 6. Then an action a_t is selected following the $\epsilon-$greedy exploration in Line 7 $\sim$ 11, where $rand()$ represents a function for sampling a variable between 0 and 1 uniformly. After executing a_t, the Q_l is updated using Algorithm 1 with respect to the current transition in Line 13. Then, high-level experiences are generated in Line 14 $\sim$ 28. The experiences e in line 15 and Line 21 are differentiated according to whether there is a subtask being achieved at current time step. If a subtask is achieved (i.e., $L(s_{t+1}) \neq \emptyset$), this achieved subtask, denoted as p_{act} in Line 19, will be appended into p_t^* in Line 20. Besides, p_{act} will be treated as the assumed subtask selected by π_h. Based on p_{act} the high-level experiences for updating are generated in Line 23 $\sim$ 26. Finally, Q_h will be updated for each experience in the episode in Line 30 $\sim$ 33. According to Algorithm 2, Q_h and Q_l are jointly updated with the same transition data but different rewards.

Algorithm 2 Automatically Learning to Compose Subtasks.

```
    Input: Total episode $M$, learning rate $\beta$, exploration rate $\epsilon$
    Output: $Q_l, Q_h$
 1: for $episode = 1 \rightarrow M$ do
 2:     Reset $Env$ and get $s_0$, $t \leftarrow 0$
 3:     $experience\_h \leftarrow \{\}$, $e\_temp \leftarrow \{\}$
 4:     $p_t^* \leftarrow []$
 5:     while not terminal do
 6:         Obtain subtask $p_t = argmax_p Q_h(s_t, p_t^*, p)$
 7:         if $rand() > \epsilon$ then
 8:             Take the action $a_t = argmax_a Q_l(s_t, p_t, a)$
 9:         else
10:             Take a random action as $a_t$
11:         end if
12:         Execute $a_t$ and get $s_{t+1}$ and $r_t$ from $Env$
13:         Update $Q_l$ using Algorithm 1 with $(s_t, a_t, s_{t+1})$
14:         if $L(s_{t+1}) = \emptyset$ then
15:             $e \leftarrow ((s_t, p_t^*), p_t, (s_{t+1}, p_t^*), r_t)$
16:             Add experience tuple $e$ into $e\_temp$
17:             $p_{t+1}^* \leftarrow p_t^*$
18:         else
19:             $p_{act} \leftarrow L(s_{t+1})$
20:             $p_{t+1}^* \leftarrow p_t^* \oplus p_{act}$
21:             $e \leftarrow ((s_t, p_t^*), p_t, (s_{t+1}, p_{t+1}^*), r_t)$
22:             Add experience tuple $e$ into $e\_temp$
23:             for $((s, p^*), p, (s', p^{*'}), r)$ in $e\_temp$ do
24:                 $e\_assumed \leftarrow ((s, p^*), p_{act}, (s', p^{*'}), r)$
25:                 Add $e\_assumed$ into $experience\_h$
26:             end for
27:             $e\_temp \leftarrow \{\}$
28:         end if
29:         if terminal then
30:             for $((s, p^*), p, (s', p^{*'}), r)$ in $experiences\_h$ do
31:                 $y \leftarrow r + \gamma \max_{p'} Q_h(s', p^{*'}, p')$
32:                 $Q_h(s, p^*, p) \leftarrow (1 - \beta) Q_h(s, p^*, p) + \beta \cdot y$
33:             end for
34:             $experiences\_h \leftarrow \{\}$
35:         end if
36:         $t \leftarrow t + 1$
37:     end while
38: end for
```