Learning from Less: Guiding Deep Reinforcement Learning with Differentiable Symbolic Planning

Zihan Ye[1,2] **Oleg Arenz**[1] **Kristian Kersting**[1,2,3,4]
[1]Computer Science Department, TU Darmstadt, Germany
[2]Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany
[3]Centre for Cognitive Science, TU Darmstadt, Germany
[4]German Research Center for Artificial Intelligence (DFKI), Darmstadt, Germany
{firstname.lastname}@tu-darmstadt.de

Abstract

When tackling complex problems, humans naturally break them down into smaller, manageable subtasks and adjust their initial plans based on observations. For instance, if you want to make coffee at a friend's place, you might initially plan to grab coffee beans, go to the coffee machine, and pour them into the machine. Upon noticing that the machine is full, you would skip the initial steps and proceed directly to brewing. In stark contrast, state-of-the-art reinforcement learners, such as Proximal Policy Optimization (PPO), lack such prior knowledge and therefore require significantly more training steps to exhibit comparable adaptive behavior. Thus, a central research question arises: *How can we enable reinforcement learning (RL) agents to have similar "human priors", allowing the agent to learn with fewer training interactions?* To address this challenge, we propose **d**ifferentiable s**y**mbolic p**lan**ner (Dylan), a novel framework that integrates symbolic planning into Reinforcement Learning. Dylan serves as a reward model that dynamically shapes rewards by leveraging human priors, guiding agents through intermediate subtasks, thus enabling more efficient exploration. Beyond reward shaping, Dylan can work as a high-level planner that composes primitive policies to generate new behaviors while avoiding common symbolic planner pitfalls such as infinite execution loops. Our experimental evaluations demonstrate that Dylan significantly improves RL agents' performance and facilitates generalization to unseen tasks.

1 Introduction

Reinforcement learning (RL) has demonstrated remarkable success across a wide range of domains, including game playing [24, 37], robotic manipulation [27, 2], and more recently, post-training in Large Language Models (LLMs) [31, 21, 22, 15]. Despite these advances, the challenge of sparse rewards remains a significant barrier to the broader applicability and efficiency of RL methods [29, 1, 10]. In sparse environments, the reward signals are infrequent or delayed, making effective exploration difficult, and the learning process can become prohibitively expensive in terms of computation [32]. Furthermore, sparse rewards often lack the granularity needed to explicitly guide agents toward desirable behaviors, frequently resulting in suboptimal or unintended outcomes. For example, while LLMs such as DeepSeek-R1 demonstrate a certain level of reasoning ability in solving complex problems, they may still exhibit issues such as language mixing or inconsistent linguistic patterns during multi-step reasoning—largely due to the limited structure provided by sparse rewards during post-training [15]. Similarly, in robotics, sparse reward signals can lead agents to exploit unintended shortcuts or converge on behaviors that diverge from human expectations [29, 6].

RLC 2025 Workshop on Programmatic Reinforcement Learning.

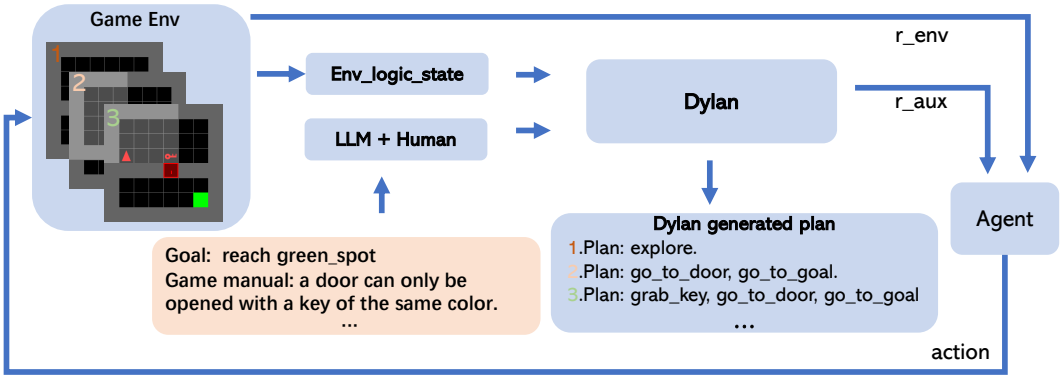

Figure 1: **An overview of Dylan working as a reward model.** Given the goal and game rules, Dylan first generates candidate plans to achieve the goal and then shapes rewards based on the generated plans, balancing among all candidate plans or selecting the best one to follow through.

Prior works [16, 29] have explored reward shaping as a means of providing agents with additional learning signals to address this challenge. These signals are typically derived from potential-based functions [28], expert-crafted heuristics [19, 9], or learned reward models based on human preferences [5] and auxiliary tasks [16]. While these approaches can reduce the amount of data required for learning to some extent, a fundamental question remains: *Can we design a reward model that not only enables agents to learn with fewer training interactions, but also aligns with human intent and remains inherently interpretable?*

To address this question, we propose DYLAN, a differentiable symbolic planner that serves as a reward model to guide reinforcement learning agents. Unlike prior reward models, DYLAN decomposes the goal task into modular subgoals and assigns rewards through logical reasoning over these subgoals (e.g., *the coffee beans are at hand*, *the agent is at the coffee machine*). These subgoal-based rewards are semantically aligned with human understanding of task structure, allowing agents to learn more efficiently while preserving interpretability. Beyond its role as a reward model, DYLAN can also work as a symbolic planner that composes different behaviors by stitching together reusable policy primitives. This capability alleviates a key limitation of conventional RL agents: they are overfit to a single task and fail to generalize. For instance, an agent trained to *navigate to a red door* may perform poorly when asked to *pick up a blue key and open a blue door*. Without modular reasoning and task composition, current RL systems must be retrained for every new task, incurring significant cost. DYLAN's compositional structure supports generalization across related tasks and facilitates knowledge transfer through its symbolic task grounding. Moreover, DYLAN's differentiable nature overcomes a common limitation of traditional symbolic planners, which are typically non-adaptive and prone to failure in environments requiring flexible search strategies.

Overall, we make the two following major, novel contributions:

- We introduce DYLAN, a differentiable symbolic planner that can be integrated as a reward model into existing reinforcement learning frameworks, offering interpretable intermediate feedback that guides agents with fewer interactions while remaining aligned with human intent.
- Beyond reward model, DYLAN can also serve as a differentiable planner. Acting as a high-level policy in a hierarchical RL setting, it composes policy primitives in a modular and flexible way, allowing the agent to generate new behaviors.

To this end, we proceed as follows. We start off by discussing background and related work in Sec. 2, followed by a detailed introduction of DYLAN in Sec. 3. Before concluding, we touch upon the experimental section in Sec. 4.

2 Background and Related Work

DYLAN builds upon several research areas, including first-order logic, differentiable reasoning, reinforcement learning and reward shaping.

First-order logic and Differentiable Forward-Chaining Reasoning. We refer readers to Appendix H for a review of first-order logic fundamentals. Build upon this foundation, differentiable forward-chaining inference [11, 35, 42] enables logical entailment to be computed in a differentiable manner using tensor-based operations. This technique bridges symbolic reasoning with gradient based learning, allowing logic driven models to be trained end-to-end. Building on this foundation, a variety of extensions have emerged, including reinforcement learning agents that incorporate logical structures into policy learning [17, 8], differentiable rule learners capable of extracting interpretable knowledge from complex visual environments [36] and differentiable meta-reasoning frameworks that enable a reasoner to perform self-inspection [42].

Dylan builds upon these developments by introducing differentiable meta-level reasoning within the forward-chaining framework [36], thereby enabling symbolic planning capabilities in a fully differentiable architecture. Unlike classical symbolic planners [12, 13], Dylan learns to adaptively compose and select rules based on task demands, thus alleviating the non-adaptivity limitation of previous classical symbolic planners.

Deep Reinforcement Learning. Reinforcement Learning (RL) has been extensively studied as a framework for sequential decision-making, where an agent learns an optimal policy through trial and error. Traditional RL approaches, such as Q-learning [39] and policy gradient methods [38], have been foundational to modern RL advancements. With the rise of deep learning, Deep Reinforcement Learning (DRL) has demonstrated success in high-dimensional environments, particularly in Advantage Actor-Critic (A2C) [23] and Proximal Policy Optimization (PPO) [34]. However, standard RL methods often struggle with sparse or delayed rewards, leading to inefficient exploration. To address this, intrinsic motivation [32] and curriculum learning [3] have been proposed to encourage exploration and guide policy learning in complex environments. Despite significant progress, RL algorithms still suffer from sample inefficiency, reward sparsity, and generalization issues. Recent research focuses on improving stability and efficiency through techniques such as hierarchical RL [26], meta-learning [14] and model-based RL [18].

Dylan builds upon these advancements by serving as a reward model, aiming to enhance learning efficiency. Unlike traditional model-based reinforcement learning [25, 41], the objective of Dylan is not to construct a world model via the logic planner. Instead, Dylan leverages human prior knowledge encoded in the logic planner to shape the agent's behavior through informed reward signals, thus guiding and accelerating the learning process, rather than modeling environment dynamics.

Reward Shaping. Reward shaping has been extensively explored to improve sample efficiency and learning stability in RL, particularly under sparse or delayed feedback. Classical potential-based reward shaping methods [29] preserve policy invariance by incorporating scalar potential functions into the reward signal. While this method is theoretically well-grounded, their expressiveness is limited. This limitation arises from their reliance on hand-crafted heuristics and their inability to capture complex task structures. Potential-based shaping has been applied to multi-agent systems [9], showcasing its effectiveness in cooperative settings through the use of spatial and role-specific heuristics. However, the approach still lacks compositionality and offers limited interpretability. To address the limitations of heuristic-driven shaping, recent approaches have explored unsupervised learning signals as an alternative means of guiding agent behavior. Unsupervised auxiliary tasks [16], generate pseudo-rewards to facilitate representation learning and encourage exploration. However, the resulting shaping signals are typically task-agnostic, static, and misaligned with high-level semantic objectives. To overcome the limitations of generic auxiliary signals, subsequent work has explored leveraging human preferences to ground reward learning in semantically meaningful objectives. Preference-based reward learning [5] aligns the agent's behavior with human preference by inferring reward functions from trajectory comparisons. While effective, this method suffers from poor sample efficiency and produces reward models that are often opaque and difficult to interpret or generalize.

Unlike existing reward shaping approaches that employ static heuristics or task-agnostic signals, Dylan introduces a differentiable symbolic planner as the reward model. This enables the dynamic assignment of shaped rewards that reflects task structure. Being interpretable and modular, Dylan facilitates learning through subgoal decomposition and compositional planning.

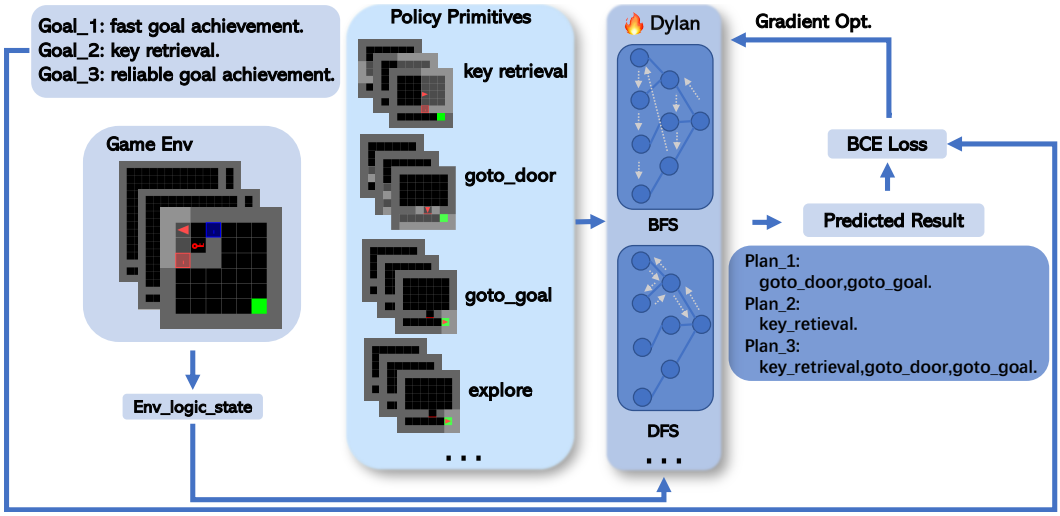

Figure 2: **Dylan, as a differentiable symbolic planner**, is capable of adapting to tasks that require different search strategies and stitching together different primitive policies to generate novel behaviours. (Best viewed in color)

3 Dylan: Guiding RL using Differentiable Symbolic Planning

A key feature of Dylan is its ability to incorporate human prior knowledge into (deep) RL in a structured and actionable end-to-end fashion. To this end, Dylan uses (differentiable) symbolic logic to represent prior knowledge and guide the agent's through reward shaping. Let us illustrate this using a navigation task from the MiniGrid-DoorKey environment [4], where the agent must pick up a key, unlock a door, and reach the goal. By consulting the environment's manual, we extract high-level rules like: "a door can only be opened with a key of the same color" and "the agent can reach the goal only by passing through an open door." Instead of representing such rules as raw text or hardcoded logic, Dylan uses them as structured symbolic transitions: each step describes an action (e.g., go to the key) that transforms one state (e.g., no key) into another (e.g., has key), provided certain preconditions are met. These transitions are represented in a format inspired by STRIPS [12], commonly used in classical planning. This symbolic abstraction allows Dylan to build a high-level plan that sequences primitive actions to achieve a given goal. To obtain task-specific rules as structured symbolic transitions, we prompt GPT-4o [30] to generate and transform candidate game rules, and then rely on human supervision to verify and refine them. We provide the detailed environment logic, game rules, the logic planner and the prompt format in Appendix A. Akin to [35, 36], by defining the initial and t-th step valuation of ground atoms as $\mathbf{v}^{(0)}$ and $\mathbf{v}^{(t)}$, we make the symbolic planner differentiable in three steps: **(Step 1)** We encode each planning rule $C_i \in \mathcal{C}$ as a tensor $\mathbf{I}_i \in \mathbb{N}^{G\times S\times L}$, where S is the maximum number of possible substitutions for variables, L is the maximum number of body atoms and G is the number of grounded atoms. Specifically, the tensor $\mathbf{I}_i$ stores at position $[j,k,l]$ the index (0 to $G-1$) of the grounded atom that serves as the l-th body atom when rule C_i derives grounded head j using substitution k. **(Step 2)** To be able to learn which rules are most relevant during forward reasoning, a weight matrix $\mathbf{W}$ consisting of M learnable weight vectors, $[\mathbf{w}_1,\ldots,\mathbf{w}_M]$, is introduced. Each vector $\mathbf{w}_m \in \mathbb{R}^C$ contains raw weights for the C planning rules. To convert these raw weights into normalized probabilities for soft rule selection, a *softmax* function is applied independently to each vector $\mathbf{w}_m$, yielding $\mathbf{w}^*_m$. **(Step 3)** At each step t, we compute the valuation of body atoms using the *gather* operation over the valuation vector $\mathbf{v}^{(t)}$, looping over the body atoms for each grounded rule. These valuations are combined using a soft logical AND (*gather* function) followed by a soft logical OR across substitutions:

$$b^{(t)}_{i,j,k} = \prod\nolimits_{1\le l\le L} \mathbf{gather}(\mathbf{v}^{(t)}, \mathbf{I}_i)[j,k,l], \quad c^{(t)}_{i,j} = softor^{\gamma}(b^{(t)}_{i,j,1},\ldots,b^{(t)}_{i,j,S}). \tag{1}$$

Here, i indexes the rule, j the grounded head atom, and k the substitution applied to existentially quantified variables. The resulting body evaluations $c^{(t)}_{i,j}$ are weighted by their assigned rule weights

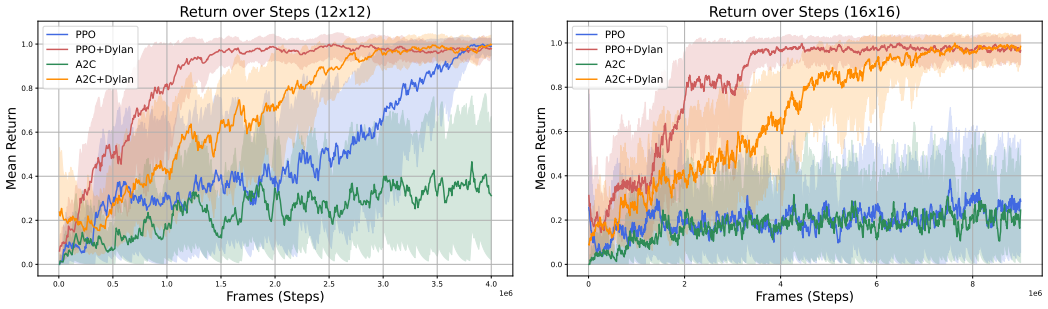

Figure 3: **Dylan boosts RL agents' performances as a static reward model.** Return comparisons of methods with and without Dylan in 12×12 and 16×16 MiniGrid environments during training. The return curves are averaged over three runs, with the solid lines representing the mean values and the shaded areas indicating the minimum and maximum values. All curves are smoothed using exponential moving average (EMA) for improved readability. (Best viewed in color)

$w^*_{m,i}$, and then aggregated across rules and rule sets:

$$h^{(t)}_{j,m} = \sum\nolimits_{1 \le i \le C} w^*_{m,i} \cdot c^{(t)}_{i,j}, \quad r^{(t)}_j = softor^{\gamma}(h^{(t)}_{j,1}, \ldots, h^{(t)}_{j,M}), \quad v^{(t+1)}_j = softor^{\gamma}(r^{(t)}_j, v^{(t)}_j). \tag{2}$$

We provide full details of this differentiable reasoning procedure in Appendix B.

3.1 Dylan as Reward Model

With Dylan at hand, we now incorporate it into the reinforcement learning and use it as a static **reward model**. As the agent interacts with the environment, a logical state is provided by the environment and used as an input to Dylan. Based on this received state, goal and rules, Dylan performs reasoning to identify the most promising plan to reach the goal. Initially, an exploratory plan is executed to gather information about the environment. After this exploratory phase, Dylan selects the optimal *plan*, the one with the highest estimated probability of achieving the desired goal. Once the plan is determined, the reward distribution follows the plan during subsequent decision-making.

Let the selected action sequence be denoted as $[\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3, \ldots, \mathtt{a}_n]$, where each action $\mathtt{a}_i$ represents a high-level action. Unlike the low-level actions commonly used in reinforcement learning, these high-level actions encode semantically meaningful behaviors, such as `get_key`. For each action, the planner also receives the corresponding expected state transition, represented as $\mathtt{move}(\mathtt{a}_i, \mathtt{s}^{(i)}_{\text{pre}}, \mathtt{s}^{(i)}_{\text{post}})$, where $\mathtt{s}^{(i)}_{\text{pre}}$ and $\mathtt{s}^{(i)}_{\text{post}}$ denote the pre- and post-action symbolic states, respectively as introduced previously. The **reward function** is designed to provide feedback only when the agent follows the planned action sequence in the correct order. Specifically, the agent must achieve each planned post-condition state $\mathtt{s}^{(i)}_{\mathtt{post}}$, corresponding to action $\mathtt{a}_i$, before proceeding to the next action in the sequence. No reward is given if the agent deviates from the prescribed order or reaches states out of sequence. The auxilary reward function $r_{\text{reasoner}}(s, a, i)$ at `i`-th transition is defined as:

$$r_{\text{reasoner}}(s, a, i) = \begin{cases} \mathtt{max}\big((\lambda - \mathtt{num\_steps}/\mathtt{total\_steps}), 0\big), & \text{if } s = s^{(i)}_{post}, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Here $\lambda > 0$, is a hyperparameter which denotes the reward. Dylan ensures the rewards are assigned **sequentially**, strictly following the planned order of actions $[\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]$. The agent must complete each transition $\mathtt{move}(\mathtt{a}_i, \mathtt{s}^{(i)}_{\mathtt{pre}}, \mathtt{s}^{(i)}_{\mathtt{post}})$ before progressing to the next step `i+1`. No rewards are granted if the agent skips steps, performs actions out of order, or transitions to incorrect states. Additionally, the reward is penalized by the number of steps taken `num_steps` relative to the total allowed steps `total_steps`. This incentivizes the agent to follow the planned sequence as efficiently as possible and discourages unnecessary actions. The shaped reward is defined as the sum of the environment reward and the reasoner reward:

$$r'(s, a, i) = r(s, a)_{\text{env}} + r_{\text{reasoner}}(s, a, i) \tag{4}$$

The objective becomes maximizing the expected cumulative discounted shaped reward:

$$J'(\pi) = \mathbb{E}_{\tau \sim \pi} \left[\sum\nolimits_{t=0}^{\infty} \gamma^t \left(r(s, a)_{\text{env}} + r_{\text{reasoner}}(s, a, i) \right) \right] \tag{5}$$

By integrating the planner as a reward model, we provide structured and goal-aligned shaped rewards. This can lead to faster convergence and fewer training interactions, as shown empirically in the experimental section.

3.2 Dylan as Adaptive Reward Model

In this subsection, we use Dylan as an adaptive reward model for reinforcement learning agents, leveraging probabilities of all possible plans from Dylan.

Building on the reward model described in Sec. 3.1, we introduce an additional dense reward r_{adaptive} for each action. This reward encourages the agent to take steps that more effectively lead toward subgoals, thereby further accelerating learning process. The dense reward r_{adaptive} is computed using the *log-sum-exp* [7] over the probabilities of all candidate plans, ensuring numerical stability while leveraging the full distribution of plan probabilities. To obtain these probabilities, Dylan performs reasoning at each step of the agent's trajectory, producing updated likelihoods for each plan. The probability of each plan is obtained by $p_{\text{targets}} = \mathbf{v}^{(T)}\left[I_{\mathcal{G}}(\text{targets})\right]$, where $I_{\mathcal{G}}(x)$ returns the index of the target atom within the set $\mathcal{G}$. $\mathbf{v}^{(T)} = f_{\text{infer}}(\mathbf{v}^{(0)})$ denotes the valuation tensor obtained after T-step forward reasoning. $\mathbf{v}[i]$ represents the i-th entry of the valuation tensor. Details on how $\mathbf{v}^{(0)}$ is initialized are provided in Appendix I.

We aggregate the probabilities of all plans using the *log-sum-exp* [7] operation for numerical stability, resulting in the dense reward:

$$r_{\text{adaptive}} = \log\left(\sum\nolimits_{p_j \in \mathcal{P}_{a_t}^{(i)}} \exp\left(\log(p_j)\right)\right).$$

To ensure that the agent's learning is not dominated solely by the dense reward, we scale it by a factor ω. As positive dense rewards could discourage the agent from finishing an episode by entering zero-reward absorbing states, we subtract a positive constant λ ensuring that the dense auxiliary reward is always negative. The sparse rewards, discussed in Sec. 3.1 does not suffer from such survival bias, as it only given when the agent makes progress towards the goal. The resulting reward function, which integrates the adaptive reward with both the environment reward and the reasoner reward described in Sec. 3.1, is defined as:

$$r'(s,a,i) = \begin{cases} \omega * r_{\text{adaptive}} - \lambda + r(s,a)_{\text{env}} + r_{\text{reasoner}}(s,a,i), & \text{successful transition,} \\ \omega * r_{\text{adaptive}} - \lambda + r(s,a)_{\text{env}}, & \text{otherwise.} \end{cases}$$

Instead of sticking to a single fixed plan, our adaptive reward model takes all possible plans into consideration, guiding the agent to choose actions aligned with high-probability, goal-achieving strategies while discouraging stagnation or ineffective behavior.

3.3 Dylan as Differentiable Planner

In addition to serving as a reward model within the reinforcement learning training, Dylan can also operate as a standalone planner, enabling the integration of multiple policies outside the training process. As a planner, Dylan's primary objective is to stitch together diverse primitive policies to generate new behaviors, thus enabling the agent to finish new tasks without retraining. However, combining various policies within limited reasoning steps presents a significant challenge: the risk of becoming trapped in loops or suffering from inefficient exploration strategies. Consider a planning task where the objective is to reach state C from state A, with available transitions including A→B, B→A, and B→C. When using depth-first search (DFS), the planner may enter an infinite loop, repeatedly cycling through states without reaching the goal. For instance, if the planner selects actions in alphabetical order, it may continually choose A→B and then B→A, resulting in an endless A-B-A-B sequence. This pathological behavior illustrates how naïve DFS can fail in symbolic planners without mechanisms to detect or prevent cycles. On the other hand, using breadth-first search (BFS) can avoid such looping issues but may introduce inefficiencies due to exhaustive exploration. This challenge raises a critical question: **How can the planner determine search strategies to accomplish tasks within constrained reasoning steps?**

To address this adaptation issue, **Dylan** employs a rule weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_M]$ to dynamically select planning rules. By applying a *softmax* function to each weight vector $\mathbf{w}_j \in \mathbf{W}$,

we choose M rules from a total of C rules. The weight matrix $\mathbf{W}$ is initialized randomly and optimized via gradient descent, minimizing a Binary Cross-Entropy (BCE) loss between the target probability p_{target} and the predicted probability $p_{\text{predicted}}$:

$$\underset{\mathbf{W}}{\texttt{minimize}} \quad \texttt{L}_{\texttt{loss}} = \texttt{BCE}(p_{\text{target}}, p_{\text{predicted}}(\mathbf{W})). \tag{6}$$

Where the predicted probability $\mathbf{p}_{\text{predicted}} = \mathbf{v}^{(T)}\left[I_{\mathcal{G}}(\text{target})\right]$, $I_{\mathcal{G}}(x)$ returns the index of the target atom within the set $\mathcal{G}$. $\mathbf{v}^{(T)}$ denotes the valuation tensor obtained after T-step forward reasoning. $\mathbf{v}[i]$ represents the i-th entry of the valuation tensor. Due to its differentiability, **Dylan** can dynamically adapt its search strategies based on the tasks, which we demonstrate in Section 4.

4 Experimental Evaluation

In this section, we show how Dylan does reward shaping using human prior and its planning capability by composing different policy primitives. Overall, we aim to answer four research questions. **Q1:** Can Dylan enable reinforcement learning agents to learn more effectively with fewer interactions? **Q2:** Can Dylan further improve RL agent learning performance by working as an adaptive reward model? **Q3:** Can Dylan compose policy primitives to generalize to a different task instead of retraining new policies? **Q4:** Can Dylan adapt itself to tasks that require different search strategies?

Environment setup. To evaluate and compare different reinforcement learning methods, we conduct a series of experiments within the MiniGrid environment suite [4]. MiniGrid offers a range of partially observable, grid-based tasks that are designed to test an agent's generalization and exploration capabilities. For consistency and reproducibility, we focus on a subset of the MiniGrid-DoorKey environment (as shown in Fig. 2), in which the agent must first acquire a key, use it to open a door, and then navigate to the goal position. While the original MiniGrid-DoorKey environment offers only a single viable solution path, we have customized the MiniGrid-DoorKey environments to support multiple solution paths. Each environment presents different exploration challenges and sparse reward settings, making them ideal benchmarks for evaluating learning performance.

Baselines. We compare the following reinforcement learning algorithms: **A2C [23]:** Advantage Actor-Critic, a synchronous, on-policy policy gradient method. **PPO [34]:** Proximal Policy Optimization , an on-policy actor-critic method which is the state of the art reinforcement learning algorithms. **hDQN [20]:** Hierarchical Deep Q-Network, a hierarchical reinforcement learning approach that decomposes complex tasks into a hierarchy of subgoals.

All methods are implemented using torch-ac [40], unless otherwise specified, hyperparameters are selected from the literature without fine-tuning (training hyperparameters in Appendix C and D).

4.1 Dylan as reward model

In this experiment, we evaluate Dylan's effectiveness as a static reward model for guiding reinforcement learning agents during training. As shown in Fig. 2, the modified MiniGrid-DoorKey environment provides two solutions for the agent to reach the goal position (indicated by the green marker). The agent can either directly go through an already opened blue door or alternatively acquire a red key, use it to unlock a red door, and navigate to the goal.

Figure 4: **Dylan further boosts RL agents' performances as an adaptive reward model**, shown by the returns over training of the PPO baseline, and PPO with reasoner and adaptive reasoner, on the 8×8 MiniGrid-DoorKey environment using static rewards, adaptive rewards, and pure PPO. Averages over 3 runs; solid lines show means, shaded areas show standard error. (Best viewed in color)

The primary goal of our experiments is to quantify how Dylan influences the learning efficiency and convergence speed across different reinforcement learning algorithms. We conduct comparative evaluations using two benchmark MiniGrid-Doorkey environments: a moderately complex 12×12 grid and a more challenging 16×16 grid. Both environments emphasize the necessity for effective exploration due to their sparse reward setting.

	Key Retrieval	Red Door Reaching	Goal Reaching	Safe Goal Reaching
A2C [23]	$59.2_{\pm 12.5}$	$50.2_{\pm 13.6}$	$98.6_{\pm 1.6}$	$41.2_{\pm 17}$
PPO [34]	$63.8_{\pm 12.2}$	$53.2_{\pm 13.4}$	$100_{\pm 0}$	$42_{\pm 17.2}$
hDQN [20]	$50_{\pm 0}$	$35.8_{\pm 4.7}$	$92.6_{\pm 3.8}$	$46.8_{\pm 2.1}$
Dylan (ours)	$100_{\pm 0}$●	$100_{\pm 0}$ ●	$100_{\pm 0}$ ●	$98.2_{\pm 2.2}$ ●

Table 1: **Performance on multitasks setting (Successrate; the higher, the better).** We compare Dylan with the baseline method PPO [34], A2C [23] and hDQN [20]. Success rates are averaged on fifty runs with its standard deviation. The best-performing models are denoted using ●.

Fig. 3 presents a detailed comparison of training performance across several reinforcement learning algorithms, specifically PPO and A2C, evaluated both with and without Dylan's auxiliary reward signals. Results are averaged over three independent runs; solid lines indicate mean performance, while shaded regions denote the min and max values. To enhance readability, all curves are smoothed using a time-based exponential moving average (EMA). Our empirical results clearly demonstrate Dylan's substantial positive impact on learning efficiency, particularly as the complexity of the environment increases. While all evaluated algorithms exhibit improved convergence rates when assisted by Dylan in the 12×12 environment, the improvement becomes notably greater in the more demanding 16×16 scenario. Remarkably, baseline agents such as PPO, which reliably converge in simpler settings, fail to converge in the challenging 16×16 environment without Dylan's guidance. In contrast, incorporating Dylan effectively resolves these exploration bottlenecks, significantly accelerating convergence. In summary, our findings strongly suggest that Dylan effectively mitigates common reinforcement learning challenges, such as sparse rewards and inefficient exploration strategies. Importantly, these improvements are achieved simply by adding Dylan as an auxiliary reward provider, without altering the original reinforcement learning algorithm.

4.2 Dylan as adaptive reward model

In this experiment, we further assess Dylan's ability to guide the agent's learning by comparing two reward models (static and adaptive) alongside a baseline pure PPO approach. The static reward model delivers sparse rewards based on the agent's immediate success, while the adaptive reward model provides denser and more informative rewards by dynamically evaluating the agent's progress. Fig. 4 illustrates the learning performance of different methods in the MiniGrid-DoorKey 8×8 environment, using both static and adaptive reward settings. Results are averaged over three independent runs, with solid lines indicating mean returns and shaded regions denoting standard error.

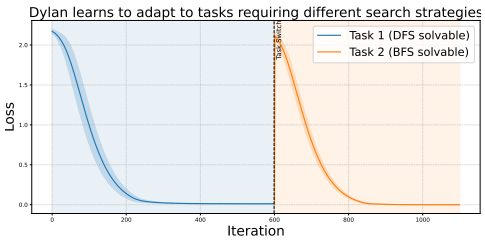

Figure 5: **Training loss curve as Dylan learns to adapt to multiple tasks requiring different search strategies.** Results are averaged over three runs, with solid lines indicating mean values and shaded areas representing standard deviation.

Our results show that incorporating Dylan's auxiliary rewards consistently improves convergence speed compared to the pure PPO [34] baseline. Notably, the adaptive reward model achieves slightly faster convergence than the static one, suggesting that accounting for multiple possible plans can further reduce training interactions. It is important to note that we did not perform exhaustive hyperparameter tuning for the adaptive reward model. As such, the current configuration (hyperparameters in Appendix J) may not be optimal, and further tuning could potentially yield even greater improvements in learning efficiency.

4.3 Dylan as differentiable planner

In this experiment, we demonstrate Dylan's ability to adapt its planning strategy to varying task structures and to compose policy primitives in order to generate novel behaviors. Specifically, we evaluate two core capabilities: adaptive search strategy selection and compositional generalization.

Dylan's adaptivity. Certain scenarios may require different search strategies. For example, in the scenario (task in Appendix. G) Depth-First Search (DFS) may result in an infinite loop, where the agent aims to reach a state defined by `reach_goal` by combining two policy primitives `go_through_red_door` and `go_through_blue_door`. However, when employing DFS,

the recursive definitions of `get_through_door` may result in repeated exploration of the same paths. For instance, the agent may continuously attempt to execute the `go_through_red_door` or `go_through_blue_door` actions without progressing towards the final goal. This is particularly problematic when the agent encounters cyclic rules or tasks with high branching factors, where DFS tends to prioritize depth exploration over breadth, thereby getting stuck in loops or excessively deep branches. Under such conditions, the agent should autonomously recognize the inefficiency of DFS and accordingly shift to a more suitable method, such as Breadth-First Search (BFS). In this task (task provided in Appendix E), the planner is asked to solve each task within three reasoning steps. The first task is best addressed using DFS, while the second is more efficiently solved with BFS. To perform well across both scenarios, the agent must dynamically adjust its planning strategy to suit the structure of each task. Fig. 5 shows the training loss curves for Dylan on these two representative tasks (planner rules provided in Appendix F). All results are averaged over three runs, with the solid line representing the mean and the shaded areas indicating the standard deviation.

Dylan's compositionality. After demonstrating Dylan's differentiable adaptivity, we further evaluate its ability to compose policy primitives to produce novel behaviors, enabling the agent to solve previously unseen tasks without retraining. In this experiment, the planner is provided with a set of reusable primitives, such as `get_key` and `go_through_door`, and is asked to solve a range of goal-directed tasks using these building blocks. We designed four tasks in this experiment. In these tasks, the agent must retrieve a key, navigate to the red door, and reach the goal position while avoiding the blue door, which leads to a trap. Note that we use the environment shown in Figure 2, where the agent has multiple possible paths to the goal. As a result, possessing a key is not necessarily a prerequisite for opening a door. For comparison, we include PPO, A2C and hDQN agents trained specifically to navigate to the goal. Table. 1 summarizes the various task settings within the MiniGrid-Doorkey environment. Dylan successfully composes low-level primitives to generalize across diverse tasks, whereas the PPO, A2C and hDQN agents, which are trained solely for goal navigation, fail to generalize to tasks requiring more complex behavior.

Overall, the experimental results provide affirmative answers to all four questions **Q1**–**Q4**.

5 Limitations

Although Dylan has demonstrated impressive results in enabling agents to learn from less environment interactions and generalizing to new tasks, it also has certain limitations. One limitation lies in its reliance on symbolic states provided directly by the environment. In future work, we aim to explore the use of vision foundation models to extract symbolic representations directly from raw game images. Another limitation involves the generation of game rules by large language models (LLMs). Currently, we prompt the GPT-4o to produce game rules and rely on human supervision to verify and correct them. A promising direction for future research is to incorporate an automated error-correction mechanism—such as leveraging multiple LLMs in a multi-round discussion framework—to improve the accuracy and reliability of the generated rules.

6 Conclusions

In this paper, we introduced Dylan, a novel reward-shaping framework that leverages human prior knowledge to help reinforcement learning agents learn with less training interactions. Beyond its role as a reward model, Dylan is, to the best of the authors' knowledge, the first differentiable symbolic planner that alleviates traditional symbolic planner's non-adaptable limitation. Dylan is capable of dynamically combining primitive policies to synthesize novel, complex behaviors. Our empirical evaluations demonstrate Dylan's effectiveness in enabling agents to learn from fewer interactions, accelerating convergence, and overcoming exploration bottlenecks—especially in environments with increasing complexity. These results illustrate Dylan's potential as a robust framework bridging symbolic reasoning and reinforcement learning.

Promising avenues for future research include automating the acquisition of symbolic abstractions, for example, through predicate invention. Additionally, investigating Dylan's scalability and broader applicability across both symbolic and subsymbolic domains remains an interesting research direction. We further want to apply Dylan to multimodal scenarios, where it could leverage multimodal inputs to more effectively guide agent learning across diverse and complex environments.

7 Acknowledgements

This work is supported by the Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) project "The Third Wave of AI" and "The Adaptive Mind". This work also benefits from preparing the Cluster of Excellence "Reasonable AI", EXC 3057/1 – project number 533677015 and the Deutsche Forschungsgemeinschaft (German Research Foundation, DFG) under Germany's Excellence Strategy (EXC 3066/1 "The Adaptive Mind", Project No. 533717223). It further benefited from the Hessian research priority programme LOEWE within the project "WhiteBox" and the EU-funded "TANGO" project (EU Horizon 2023, GA No 57100431) and Federal Ministry of Education and Research under the funding code 01|S25002B. Furthermore, the authors would like to thank Quentin Delfosse, Cedric Derstroff and Jannis Brugger for proofreading our paper and thus improving the clarity of the final manuscript.

References

[1] Marcin Andrychowicz et al. Hindsight experience replay. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[2] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 2020.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, 2009.

[4] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *Advances in Neural Information Processing Systems*, 2023.

[5] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2023.

[6] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 2017.

[7] Marco Cuturi and Mathieu Blondel. Soft-dtw: a differentiable loss function for time-series. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[8] Quentin Delfosse, Hikaru Shindo, Devendra Dhami, and Kristian Kersting. Interpretable and explainable logical policies via neurally guided symbolic abstraction. *Advances in Neural Information Processing Systems*, 2023.

[9] Sam Devlin, Daniel Kudenko, and Marek Grzes. An empirical study of potential-based reward shaping and advice in complex, multi-agent systems. *Adv. Complex Syst.*, 2011.

[10] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[11] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 2018.

[12] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 1971.

[13] Richard E Fikes and Nils J Nilsson. Strips, a retrospective. *Artificial intelligence*, 1993.

[14] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, 2017.

[15] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv*, 2025.

[16] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *International Conference on Learning Representations*, 2017.

[17] Zhengyao Jiang and Shan Luo. Neural logic reinforcement learning. In *International conference on machine learning*, 2019.

[18] Łukasz Kaiser, Mohammad Babaeizadeh, Piotr Miłos, Błażej Osiński, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model based reinforcement learning for atari. In *International Conference on Learning Representations*, 2019.

[19] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 2013.

[20] 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*, 2016.

[21] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv*, 2024.

[22] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv*, 2024.

[23] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 2016.

[24] 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*, 2015.

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

[26] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 2018.

[27] Michael Neunert, Abbas Abdolmaleki, Markus Wulfmeier, Thomas Lampe, Tobias Springenberg, Roland Hafner, Francesco Romano, Jonas Buchli, Nicolas Heess, and Martin Riedmiller. Continuous-discrete reinforcement learning for hybrid control in robotics. In *Proceedings of the Conference on Robot Learning*, 2020.

[28] A. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning*, 1999.

[29] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, 1999.

[30] OpenAI. Chatgpt-4o, 2024.

[31] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 2022.

[32] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, 2017.

[33] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009.

[34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, 2017.

[35] Hikaru Shindo, Masaaki Nishino, and Akihiro Yamamoto. Differentiable inductive logic programming for structured examples. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[36] Hikaru Shindo, Viktor Pfanschilling, Devendra Singh Dhami, and Kristian Kersting. α ilp: thinking visual scenes as differentiable logic programs. *Machine Learning*, 2023.

[37] Tom Silver, Rohan Chitnis, Joshua Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning symbolic operators for task and motion planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[38] Richard S Sutton, Satinder Singh, and David McAllester. Comparing policy-gradient algorithms. 2000.

[39] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 1992.

[40] Lucas Willems. torch-ac: Pytorch implementations of advantage actor-critic methods. `https://github.com/lcswillems/torch-ac`, 2019.

[41] Zifan Wu, Chao Yu, Chen Chen, Jianye Hao, and Hankz Hankui Zhuo. Plan to predict: Learning an uncertainty-foreseeing model for model-based reinforcement learning. *Advances in Neural Information Processing Systems*, 2022.

[42] Zihan Ye, Hikaru Shindo, Devendra Singh Dhami, and Kristian Kersting. Neural meta-symbolic reasoning and learning. *arXiv preprint arXiv:2211.11650*, 2022.

A Symbolic transformation for Dylan

Prompt: read the game manual and return the game rules in first-oder logic format such as:

```
get_through_door:- initial, go_through_red_door.
get_through_door:- get_through_door,go_through_blue_door.
get_through_door:- get_through_door,go_through_red_door.
reach_goal :- get_through_door,go_to_goal.
```

Dylan leverages human prior knowledge to guide reinforcement learning. As the first step, this prior knowledge is converted into a structured, logic-based format. For example, consider the agent navigation task illustrated in Fig. 2, drawn from MiniGrid Doorkey environment [4], After reviewing the environment's manual, we extract a rule stating that a locked door can only be opened with a key of the corresponding color, and that the agent can reach the goal by passing through an opened door. This rule can be expressed as the logic program:

```
get_red_key:-init, go_red_key.     go_through_door:-init,go_blue_door.
go_through_door:-get_red_key,go_open_red_door.
reach_goal:-go_through_door,go_to_goal.
```

We categorize the atoms in these clauses into two types: **state atoms** and **policy atoms**. Within each clause, the **head** and the **first body atom** are considered state atoms, while the **second body atom** corresponds to a policy atom. Following the STRIPS-style representation [12, 13], we transform each clause into a single atom using the predicate:

$$\texttt{move/3/[action, pre\_condition, post\_condition]}.$$

For example, the clause: `get_blue_key :- initial, go_blue_key.` is rewritten as one atom: `move(go_blue_key, initial, get_blue_key)`. In this transformation, the **head** (`get_blue_key`) becomes the **postcondition**, the **first body atom** (`initial`) serves as the **precondition**, and the **second body atom** (`go_blue_key`) represents the **action**.

With this transformation at hand, we now a symbolic planner:

```
plan(Start, New, Goal, [Act, Old_stack]) :-
        move(Act, Old, New),condition_met(Old, Current),
        change_state(Current, New),plan(Start, Current, Goal, Old_stack).
plan_final(Start, Goal, Move_stack) :-
        plan(Start, Current, Goal, Move_stack),equal(Current, Goal).
```

The first rule establishes the recursive process for generating a plan. It selects an appropriate policy that transitions the agent from an old state to a new state, verifies the necessary conditions, updates the current state accordingly, and recursively continues the planning process until the goal is reached. The second rule defines the termination condition, ensuring that the planning process concludes once the agent's current state matches the desired goal state.

B Differentialize Symbolic planner

We now describe each step in detail on how to differentialize the symbolic planner.

(Step 1) Encoding Logic Programs as Tensors. To enable differentiable forward reasoning, Each meta-rule is transformed into a tensor representation for differentiable forward reasoning. Each meta-rule $C_i \in \mathcal{C}$ is encoded as a tensor $\mathbf{I}_i \in \mathbb{N}^{G \times S \times L}$, where S denotes the maximum number of substitutions for existentially quantified variables in the rule set , and L is the maximum number of atoms in the body of any rule. For instance, $\mathbf{I}_i[j, k, l]$ stores the index of the l-th subgoal in the body of rule C_i used to derive the j-th fact under the k-th substitution.

(Step 2) Weighting and Selecting Meta-Rules. We construct the reasoning function by assigning weights that determine how multiple meta-rules are combined. (i) We fix the size of the target meta-program to be M, meaning the final program will consist of M meta-rules selected from a total of C candidates in $\mathcal{C}$. (ii) To enable soft selection, we define a weight matrix $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_M]$, where each $\mathbf{w}_i \in \mathbb{R}^C$ assigns a real-valued weight. (iii) We then apply a *softmax* to each $\mathbf{w}_i$ to obtain a probability distribution over the C candidates, allowing the model to softly combine multiple meta-rules.

(Step 3) Perform Differentiable Inference. Starting from a single apply of the weighted meta-rules, we iteratively propagate inferred facts across T reasoning steps.

We compute the valuation of body atoms for every grounded instance of a meta-rule $C_i \in \mathcal{C}$. This is achieved by first gathering the current truth values from the valuation vector $\mathbf{v}^{(t)}$ using an index tensor $\mathbf{I}i$, and then applying a multiplicative aggregation across subgoals:

$$b^{(t)}_{i,j,k} = \prod_{l=1}^{L} \mathbf{gather}(\mathbf{v}^{(t)}, \mathbf{I}i)[j,k,l], \tag{7}$$

where the $\mathbf{gather}$ operator maps valuation scores to indexed body atoms:

$$\mathbf{gather}(\mathbf{x}, \mathbf{Y})[j,k,l] = \mathbf{x}[\mathbf{Y}[j,k,l]]. \tag{8}$$

The resulting value $b^{(t)}_{i,j,k} \in [0,1]$ reflects the conjunction of subgoal valuations under the k-th substitution of existential variables, used to derive the j-th candidate fact from the i-th meta-rule. Logical conjunction is implemented via element-wise product, modeling the "and" over the rule body.

To integrate the effects of multiple groundings of a meta-rule C_i, we apply a smooth approximation of logical *or* across all possible substitutions. Specifically, we compute the aggregated valuation $c^{(t)}_{i,j} \in [0,1]$ as:

$$c^{(t)}_{i,j} = softor^{\gamma}(b^{(t)}_{i,j,1}, \ldots, b^{(t)}_{i,j,S}), \tag{9}$$

where $softor^{\gamma}$ denotes a differentiable relaxation of disjunction. This operator is defined as:

$$softor^{\gamma}(x_1, \ldots, x_n) = \gamma \log \sum_{i=1}^{n} \exp(x_i/\gamma), \tag{10}$$

with temperature parameter $\gamma > 0$ controlling the smoothness of the approximation. This formulation closely resembles a softmax over valuations and serves as a continuous surrogate for the logical *max*, following the log-sum-exp technique commonly used in differentiable reasoning [7].

(ii) Weighted Aggregation Across Meta-Rules. We compute a weighted combination of meta rules using the learned soft selections:

$$h^{(t)}_{j,m} = \sum_{i=1}^{C} w^{m,i} \cdot c^{(t)} i, j, \tag{11}$$

where $h^{(t)}_{j,m} \in [0,1]$ represents the intermediate result for the j-th fact contributed by the m-th slot. Here, $w^{m,i}$ is the softmax-normalized score over the i-th meta-rule:

$$w^{*}_{m,i} = \frac{\exp(w_{m,i})}{\sum_{i'} \exp(w_{m,i'})}, \quad w_{m,i} = \mathbf{w}_m[i].$$

Finally, we consolidate the outputs of the M softly selected rule components using a smooth disjunction:

$$r^{(t)}_j = softor^{\gamma}(h^{(t)}_{j,1}, \ldots, h^{(t)}_{j,M}), \tag{12}$$

which yields the t-step valuation for fact j. This mechanism allows the model to integrate M soft rule compositions from a larger pool of C candidates in a fully differentiable way.

(iii) Iterative Forward Reasoning. We iteratively apply the forward reasoning procedure for T steps. At each step t, we update the valuation of each fact j by softly merging its newly inferred value $r_j^{(t)}$ with its previous valuation:

$$v_j^{(t+1)} = softor^{\gamma}(r_j^{(t)}, v_j^{(t)}). \tag{13}$$

This recursive update mechanism approximates logical entailment in a differentiable form, enabling the model to perform T-step reasoning over the evolving fact valuations. The whole reasoning computation Eq. 7-13 can be implemented using efficient tensor operations.

C Hyperparameter

Table 2: Summary of Training Hyperparameters

Parameter	Training Parameters
`-epochs`	4 (PPO optimization epochs per update)
`-batch-size`	256 (Batch size for PPO updates)
`-frames-per-proc`	128 (Frames per process before update)
`-discount`	0.99 (Discount factor γ)
`-lr`	0.0001 (Learning rate)
`-gae-lambda`	0.95 (λ for GAE)
`-entropy-coef`	0.01 (Entropy regularization coefficient)
`-value-loss-coef`	0.5 (Value loss coefficient)
`-max-grad-norm`	0.5 (Gradient clipping norm)
`-optim-eps`	1×10^{-8} (Optimizer epsilon)
`-optim-alpha`	0.99 (RMSprop alpha)
`-clip-eps`	0.2 (PPO clipping parameter ϵ)
`-recurrence`	1 (Recurrent steps, LSTM if > 1)
`-text`	`False` (Enable GRU for text input)

D hDQN architechture

As the official implementation of hDQN is not publicly released, we provide our own PyTorch implementation following the methodology described in the original paper. Our architecture consists of two key neural network components: the **MetaController** and the **ControllerQNetwork**. These models form a hierarchical structure where the MetaController selects subgoals or abstract directives, and the ControllerQNetwork executes primitive actions conditioned on those directives.

D.1 MetaController

The `MetaController` is a multi-layer feedforward network responsible for high-level decision making. It takes as input a feature vector representing the environment state and outputs a latent code or subgoal.

Architecture

- Input: Feature vector of dimension `in_features`
- Linear layer: Linear(`in_features` $\rightarrow 512$)
- Layer Normalization: LayerNorm(512)
- LeakyReLU activation with slope 0.01
- Dropout: $p = 0.1$
- Linear layer: Linear$(512 \rightarrow 256)$
- Layer Normalization: LayerNorm(256)
- LeakyReLU activation with slope 0.01
- Output Linear layer: Linear$(256 \rightarrow$ `out_features`$)$

Output

A latent vector or high-level action representation of dimension `out_features`.

D.2 ControllerQNetwork

The `ControllerQNetwork` is a value-based deep Q-network that operates at the lower level. It estimates Q-values for primitive actions given the current state and optionally the selected subgoal.

Architecture

- Input: Feature vector of dimension `input_dim`
- Linear layer: Linear($\texttt{input\_dim} \to 512$)
- Layer Normalization: LayerNorm(512)
- LeakyReLU activation with slope 0.1
- Dropout: $p = 0.1$
- Linear layer: Linear($512 \to 256$)
- Layer Normalization: LayerNorm(256)
- LeakyReLU activation with slope 0.1
- Output Linear layer: Linear($256 \to \texttt{output\_dim}$)

Output

A vector of Q-values for `output_dim` discrete actions.

E Differentiable Planning Task

In this appendix, we present two pathological tasks that require a planner to adapt its search strategy—specifically between Depth-First Search (DFS) and Breadth-First Search (BFS)—for efficient goal finding within three steps. These examples demonstrate how a fixed strategy may underperform depending on the search structure and goal location.

Task 1 favors DFS, as the goal `plan(a, h)` lies deep along a single branch:

```
edge(a, c). edge(a, b). edge(a, d).
edge(b, e). edge(c, f). edge(d, g).
edge(e, h).
```

Task 2 favors BFS, as the goal `plan(a, e)` is close to the root but may be delayed by DFS exploring deeper branches first:

```
edge(a, c). edge(a, b).edge(b, d).
edge(c, e). edge(d, f).
```

F Differentiable Planning rules

We define the planning behavior of two search algorithms—**DFS** and **BFS**—using logical rules. These rules describe how each algorithm explores the search space and identifies successful plans.

```
dfs(B, F, G, r(F, D)) :− edge(E, F), dfs(B, E, G, D).
plan(B, G) :− dfs(B, F, G, H), equal(F, G).
bfs(k(B, D), S, E) :− findall(A, F), append(C, F, k(B, D)), bfs(k(A, C), S, E).
plan(S, E) :− bfs(k(A, C), S, E), equalbfs(A, C, E).
```

G Task that DFS can fail

```
get_through_door:- initial, go_through_red_door.
get_through_door:- get_through_door,go_through_blue_door.
get_through_door:- get_through_door,go_through_red_door.
reach_goal:- get_through_door,go_to_goal.
```

H First-Oder Logic

In first-order logic, a term can be a constant, a variable, or a function term constructed using a function symbol. We denote an n-ary predicate $\mathtt{p}$ as $\mathtt{p}/(n, [\mathtt{dt_1}, \ldots, \mathtt{dt_n}])$, where $\mathtt{dt_i}$ represents the data type of the i-th argument. An atom is an expression of the form $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$, where $\mathtt{p}$ is an n-ary predicate symbol and $\mathtt{t_1}, \ldots, \mathtt{t_n}$ are terms. If the atom contains no variables, it is referred to as a ground atom, or simply a fact.

A literal is either an atom or the negation of an atom. We refer to an atom as a positive literal, and its negation as a negative literal. A clause is defined as a finite disjunction ($\vee$) of literals. When a clause contains no variables, it is called a ground clause. A definite clause is a special case: a clause that contains exactly one positive literal. Formally, if $A, B_1, \ldots, B_n$ are atoms, then the expression $A \vee \neg B_1 \vee \ldots \vee \neg B_n$ constitutes a definite clause. We write definite clauses in the form of A :- $B_1, \ldots, B_n$. where A is the *head* of the clause, and the set $\{B_1, \ldots, B_n\}$ is referred to as the body. For simplicity, we refer to definite clauses as clauses throughout this paper. The forward-chaining inference is a type of inference in first-order logic to compute logical entailment [33].

I initial valuation obtained for adaptive reward model

To obtain the initial valuation v_0 for the adaptive reward model, we initialize both the environment's logic state and the high-level action atoms. Since the environment logic states are externally provided, the only variable component is the valuation of the high-level action atoms. We assign these valuations based on the agent's distance to each subtask, using the inverse of the distance (plus a small positive constant) to produce a meaningful and smoothly varying probability. This design ensures that as the distance changes, the corresponding probability in the plan is updated accordingly.

In our adaptive reward model experiments, we define the valuation of each high-level action atom as $0.5 + \frac{1}{\text{distance}+2}$. The plan probability is then computed based on the initial valuation of both the action atoms and the environment atoms.

J hyperparameters for adaptive reward model

$\lambda = 0.01 \qquad \omega = 1/20$