Differentiable Synthesis of Behavior Tree Architectures and Execution Nodes

Yu Huang[1] HUANGYU@NUDT.EDU.CN
Ziji Wu[1] WUZIJI@NUDT.EDU.CN
Kexin Ma[2] MAKEXIN@NUDT.EDU.CN
Ji Wang[1,*] WJ@NUDT.EDU.CN
1 State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China
2 Institute for Quantum Information & State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China

Editors: G. Pappas, P. Ravikumar, S. A. Seshia

Abstract

Deep reinforcement learning (DRL) has achieved remarkable success in solving complex control tasks. However, neural network policies often lack interpretability and struggle to generalize to new scenarios without further training. Behavior trees (BTs) offer a more interpretable policy representation, making them a promising alternative. Yet, the automatic synthesis of BTs remains a challenge due to the discrete search space and the need to adapt to diverse scenarios. Prior works often come at the cost of fixed or constrained architectures, or rely on customized execution nodes. We propose an end-to-end synthesis framework that simultaneously generates the architectures and execution nodes of BTs solely from environment rewards. We first conduct architecture search on top of a continuous relaxation of the architecture search space derived from a given grammar. To tackle the discrete execution mechanism and non-differentiable semantics of BTs, we redefine the execution mechanism and interpret the semantics in terms of a differentiable approximation. We also propose an efficient extraction algorithm that leverages the fallback structure of BTs to instantiate a valid BT architecture. This algorithm recovers the performance damaged by the co-adaptation and continuous approximation. Experiment results show the superior performance and generalization of our synthesized BTs, demonstrating the efficacy of proposed framework.

Keywords: Differentiable Synthesis, Behavior Trees, Explainable Reinforcement Learning, Neurosymbolic Programming

1. Introduction

Deep reinforcement learning has achieved tremendous success in a variety of domains, such as games Silver et al. (2016) and robotic control Ibarz et al. (2021). However, neural network policies often struggle with poor interpretability and generalization. To address these issues, recent research has explored the synthesis of symbolic policies, which express task-solving logic in a more transparent form. One such symbolic policy, Behavior Tree, has been widely praised for its interpretability, compositionality, and generalization abilities Colledanchise and Ögren (2017). Despite its potential, the automatic synthesis of BTs still faces significant challenges. It requires not only a reasonable BT architecture but also diverse agent capabilities that are represented as execution nodes within

* Corresponding Author.

the BTs. Hence, two major challenges persist: the first lies in the discrete and rapidly growing architecture search space, and the second involves customizing agent capabilities, which necessitates considerable human expertise. Current BT synthesis methods encounter limitations in architectural flexibility. To manage the extensive search space, previous approaches typically focus on optimizing execution parameters within fixed architectures Mayr et al. (2021) or rely on narrow structural templates for incremental construction French et al. (2019). Although these constrained architectures enhance search efficiency, they significantly limit policy expressiveness and are often tailored to specific tasks. Recent efforts attempt to address these limitations through two paradigms: (1) Formal grammar-based methods Scheide et al. (2021), which introduce formal grammars of BTs and search over them using the variants of Monte-Carlo searching algorithm, and (2) Automated planning techniques Colledanchise et al. (2019), that build BTs using back-chaining mechanisms. However, both paradigms rely on predefined action/condition libraries, which require extensive expert knowledge to construct. Moreover, this reliance damages the learning ability, preventing BTs from proactively learning primitive behaviors during environment interactions.

We propose a novel end-to-end synthesis framework that jointly learns the architecture and execution nodes of BTs while maximizing the expected task reward. Unlike prior methods, our framework defines both the production rules for BT architectures and the domain-specific languages (DSLs) for execution nodes. The introduction of DSLs enables the learning of execution nodes that are typically predefined by experts and allows for adaptation through environment interactions. For instance, in a Pendulum environment, different controllers like PID or linear controllers may be needed for swinging the pendulum up and keeping it upright. Our framework automatically learns suitable controller types and their parameters, overcoming limitations in existing approaches. Inspired by recent advances in differentiable architecture search Liu et al. (2019); Qiu and Zhu (2022), we conduct the architecture search process on top of a continuous relaxation of the discrete search spaces. It converts the selection problem of production rules into a continuous weight optimization problem, facilitating the use of gradient-based methods. After training the architecture weights, we extract a discrete BT architecture and further fine-tune the parameters of execution nodes using standard reinforcement learning algorithms. However, BTs' synthesis presents unique challenges. First, BTs have various control structures with different semantics, which are more complex than the if-then-else structure in Qiu and Zhu (2022). The richer search space makes it more difficult to instantiate a discrete architecture from the trained weights. Second, the discrete execution mechanism inherent in BTs and the non-differentiable semantics of control nodes are incompatible with gradient-based architecture search. To address these challenges, we redefine the execution mechanism of BTs to include continuous outputs and interpret the non-differentiable semantics of control nodes using continuous approximation. The improved differentiable BTs preserve the consistent interface throughout the entire BT and their modularity. Moreover, we find that the continuous approximation and co-adaptation phenomenon discovered in Cui and Zhu (2021) hinder valid architecture extraction by greedily selecting the most likely architecture. Consequently, we present an effective BT extraction algorithm that leverages the fallback of BTs.

To the best of our knowledge, this framework is the first end-to-end solution capable of simultaneously generating BT architectures and execution nodes while maximizing task performance. We evaluated it across various control scenarios, ranging from classic control environments with both discrete and continuous action spaces to more complex tasks in MuJoCo. The results demonstrate that our framework excels in synthesizing BTs with superior performance and generalization.

2. Preliminaries

2.1. Behavior Tree

A Behavior Tree is a rooted directed tree structure composed of execution nodes which interact directly with the environment, and control nodes which govern the triggering logic of their child nodes. During execution, the BT propagates tick signals in depth-first order based on the environment states and the rules of different node types. A node executes upon receiving a tick signal and subsequently transmits its return status to its parent (if any). This paper focuses specifically on the four BT nodes: **Condition:** An execution node that checks the proposition related to current states and returns *success* if it satisfies; otherwise, it returns *failure*. **Action:** An execution node that performs a specified behavior and returns *success*, *failure*, or *running* depending on the execution results. **Sequence:** A control node that ticks its children from left to right. It returns *failure* or *running* immediately when a child returns either *failure* or *running*. Otherwise, it returns *success*. **Fallback:** A control node that ticks its children from left to right. It returns *success* or *running* immediately when a child returns either *success* or *running*. Otherwise, it returns *failure*. More details about BTs and their execution can be found in Appendix A.

Behavior Tree Construction Grammar. We design an extensible grammar for constructing BTs. As illustrated in Fig. 1 (left), the symbol $\rightarrow$ and ? denote *Sequence* and *Fallback* respectively, while *Act, Cond* represents Action and Condition. The hyperparameter n denotes the maximum number of nonterminals allowed in a control node. For $n \geq 2$, the BT construction grammar parameterized by n affects only topology while preserving expressivity. We fix $n = 2$ hereafter. In this grammar, we do not utilize black-box execution nodes but employ different DSLs to unlock encapsulated execution nodes. It allows users to scale this grammar appropriately according to various task requirements. For example, in certain tasks, a simple linear function can be sufficient to evaluate possible conditional judgments or actions. The construction grammar for the instantiated BT is depicted in Fig. 1 (right). We further describe BT as a pair (g, θ), where g represents the discrete architecture and θ is a matrix of real-valued parameters associated with the execution nodes.

$$
\begin{aligned}
BT &:= (\rightarrow Cond\ BT_1\ \ldots\ BT_n) \\
&\quad \mid (?\ Cond\ BT_1\ \ldots\ BT_n) \mid Act
\end{aligned}
$$

$$
\begin{aligned}
BT &:= (\rightarrow Cond\ BT_1\ \ldots\ BT_n) \\
&\quad \mid (?\ Cond\ BT_1\ \ldots\ BT_n) \mid Act \\
+\ Cond &:= \theta_{c1} + \theta_{c2} * x \geq 0 \\
+\quad Act &:= \theta_{a1} + \theta_{a2} * x
\end{aligned}
$$

Figure 1: General BT's construction grammar (left) and Instantiated version (right).

Discussion of Sequence Nodes. A Sequence node is used for actions that should be carried out in order. However, it poses a great synthesis challenge in RL control tasks. Executing multiple actions in a single tick may cause temporal inconsistencies, as subsequent action in Sequence nodes depends on environmental changes from preceding actions, potentially creating variable-length environment steps. On the other hand, the sequential execution of actions can also be achieved by synchronizing the duration of environment steps with tick duration. Therefore, this work primarily focuses on Sequence nodes that have only two child nodes ($n = 1$).

2.2. Differentiable Architecture Synthesis

Architecture search is challenging due to the discrete and combinatorial search space. *Differentiable ARchiTecture Search* (DARTs) Liu et al. (2019) proposed a differentiable search framework

for neural architecture search problems. It transforms the operation selection problem for a fixed set of neural network blocks into weight optimization. Recent work Qiu and Zhu (2022); Cui and Zhu (2021) applies this method to program synthesis. Given a grammar, those methods perform an architecture search to this grammar over a program derivation graph where nodes contain partial or complete program architectures, as depicted in Fig. 2. Edges encode extending choices between different architectures and are relaxed to trainable weights ω. Then, efficient gradient-descent methods are utilized to learn the probability distribution over all potential program architectures within the derivation graph. Upon convergence, a discrete program architecture is extracted top-down by replacing each node with the architecture that has the highest probability.

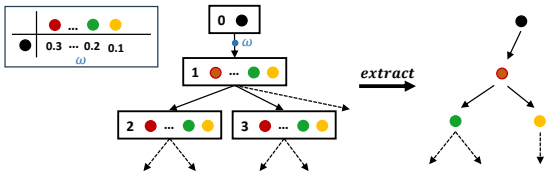


Figure 2: An example derivation graph with weights ω. Based on ω, a discrete program is extracted.

2.3. Problem Formulation

Formally, a RL system can be formulated as a Markov Decision Process (MDP) defined by a tuple M[π] = ($\mathcal{S}$, $\mathcal{A}$, P, r, $\mathcal{S}_0$, π), where $\mathcal{S}$ is the environment state space, $\mathcal{A}$ represents the action space, $\mathcal{S}_0$ is a set of initial states, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ denotes the transition probabilities, and $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ captures the task rewards. π is a certain policy, which takes states as input and outputs actions. At time step t, the agent in state s_t performs actions selected by policy π, then receives a reward r_t from the environment and transitions to a new state s_{t+1}. A MDP system parameterized with a policy π can generate trajectories $\zeta_\pi = s_0, a_0, \ldots, a_{t-1}, s_t, \ldots$. The long-term reward of π is $R(\pi) = E_{(\zeta_\pi = s_0, a_0, \ldots, a_{t-1}, s_t, \ldots) \sim M[\pi]} \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$, where a discount factor $\gamma \in [0, 1]$ is hired to avoid infinite total rewards. This paper develops a new RL framework using BTs as policy representations. We aim to synthesize a BT policy π_{BT} that satisfies the grammar and maximizes the long-term discount reward $R(\pi_{BT})$, by simultaneously learning the architecture g and parameters θ. Since the input space of Condition nodes or Action nodes can be different from that of the whole environment state, we assume $\mathcal{X}$ is the set of input variables where $\mathbb{R}^{|\mathcal{X}|} \subset \mathcal{S}$.

3. Differentiable Synthesis of Behavior Trees

3.1. Relaxing Architectures Search Space

Formally, behavior tree architecture synthesis constrained by a grammar is performed over a *derivation graph* $\mathcal{G} = \{V, E\}$ where a node $v \in V$ contains a set of partial architectures with nonterminals or complete behavior tree architectures permissible by grammar. An edge $(v, v') \in E$ connects two nodes v and v' if architectures in v' can be obtained by expanding a nonterminal within partial architectures in v following some production rules. We formulate the architecture derivation graph in a top-down manner and Fig. 3 depicts a derivation graph for the grammar in Fig. 1.

$\mathcal{G}$ essentially expresses all possible BTs up to a certain depth bound d. In the derivation graph, $BT_{i,j}^k$ represents the k-th nonterminal on j-th partial architecture in node i (start counting at 0). We assign a trainable weight ω to each edge and relax the choice of all possible production rules into a softmax. For example, on root node 0, we have four choices to expand the initial nonterminal BT.

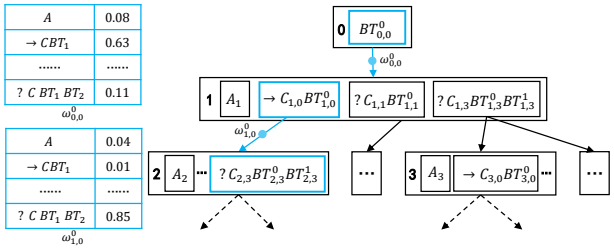


Figure 3: Derivation graph of grammar. It omits the derivation of execution nodes for clarity.

Thus, node 1 contains four partial architectures, and a weight $\omega^0_{0,0}$ is assigned to the edge (v_0, v_1). To learn architecture weights, a derivation graph itself is encoded as a differentiable function $\mathcal{F}_{\omega,\theta}$, where ω represents architecture weights and θ includes unknown parameters of all behavior trees in the graph. $\mathcal{F}_{\omega,\theta}$ takes the environment state as input and outputs actions. The semantic computation of $BT^k_{i,j}$ in node v_i is defined as a softmax over possible grammar production rules for $BT^k_{i,j}$:

$$\|BT^k_{i,j}\|(s) = \sum_{g \in \mathcal{N}(v')} \frac{exp(\omega_e[BT^k_{i,j}, g])}{\sum_{g' \in \mathcal{N}(v')} exp(\omega_e[BT^k_{i,j}, g'])} \cdot \|g\|(s) \tag{1}$$

where v' is the child node directly connected to node v_i, and e=(v_i, v'). $\mathcal{N}(v')$ represents the set of architectures in node v'. The output of $\mathcal{F}_{\omega,\theta}$ is delegated to $BT^0_{0,0}$ which is the weighted sum by the outputs of all behavior trees contained in the derivation graph.

Complexity. Let d be the depth of a derivation graph, m be the maximum number of production rules, and n be the maximum number of nonterminals in any rules. The computation complexity of $\mathcal{F}_{\omega,\theta}$ is $O((m \cdot n)^d)$. In practice, we employ many strategies to reduce the complexity and run-time, like Node Sharing Cui and Zhu (2021). We leave the details in Appendix D.1.

We formulate the training process as a bi-level optimization problem and jointly optimize (ω, θ) with a differentiable object function $\mathcal{J}$ defined on the derivation graph's output. The optimization proceeds as an iterative two-phase procedure: on the architecture optimization step, we freeze the execution parameters θ and optimize architecture parameters ω with respect to (2); on the execution parameters optimization step, we freeze ω and train θ with respect to (3).

$$\omega_{i+1} \leftarrow \omega_i + \alpha \cdot \nabla_\omega J_{\theta_i,\omega_i} \tag{2}$$

$$\theta_{i+1} \leftarrow \theta_i + \alpha \cdot \nabla_\theta J_{\theta_i,\omega_i} \tag{3}$$

Those two steps are alternated across training iterations until convergence. The object function $\mathcal{J}$ has different forms depending on the learning methods. Take proximal policy optimization (PPO) Schulman et al. (2017) as an example, object function $\mathcal{J}$ and learning target are:

$$\text{maximize}_{\theta,\omega}\ J_{\theta_{\text{old}},\omega_{\text{old}}}(\theta,\omega) = \mathbb{E}_{s\sim\rho_{\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}},a\sim\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}}\left[\frac{\mathcal{F}_{\theta,\omega}(s,a)}{\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}(s,a)} A_{\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}}(s,a)\right] \tag{4}$$

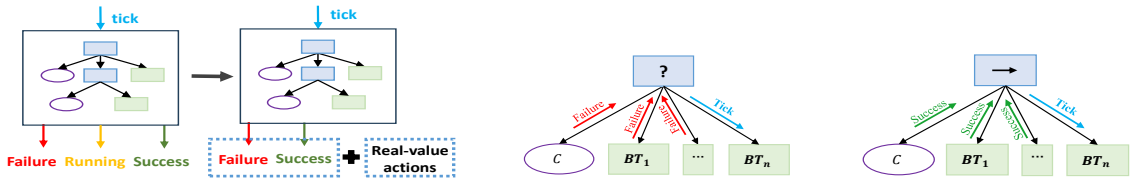
where $\rho_{\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}}$ is the discounted state visitation frequency of $\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}$, $A_{\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}}(s,a)$ is an advantage estimator over a batch of samples from $\mathcal{F}_{\theta_{\text{old}},\omega_{\text{old}}}$, and θ_{old}, ω_{old} are execution node parameters and architecture weights before the update.

3.2. Differentiable BTs

Originally, concrete actions are performed exclusively in *Action* nodes and BTs only return one of the status signals among *failure*, *running*, and *success*. While this execution mechanism facilitates task switching, its discrete returns are incompatible with gradient-based methods for differentiable

architecture searches. To avoid discontinuities, we redefine the output of BTs which returns not just discrete status but also the real-valued actions performed in the environment, as shown in Fig. 4(a).

In this paper, we stipulate that an Action node returns *success* and its real-valued actions to the parent once it is executed. To prevent non-uniform interfaces inside the BT hierarchy, we redefine the outputs of all types of control nodes to ensure a consistent interface across the entire BT. A control node receives real-valued actions from its children and similarly passes them to its parent after computation. We define a real-valued function $\|\cdot\|(s)$ as the semantics of each grammatical construct, for example, $\|A_i\|(s) = a_i[s]$ reads the result of action A_i in state s, where a_i is the action function, and $\|BT\|(s)$ represents the output of the entire BT in state s.



(*a*) Execution machnisms (*b*) *Fallback* Semantic (*c*) *Sequence* Semantic

Figure 4: (a) In addition to returning one of the status signals, the BT also returns real-valued actions in our framework. (b) Semantics of *Fallback* nodes. (c) Semantics of *Sequence* nodes.

As illustrated in Fig. 4(b) and Fig. 4(c), a child node can be executed only if its former sibling node failed in Fallback nodes or succeeded in Sequence nodes. The discrete nature of control node semantics also creates discontinuous gradients. To overcome this problem, we perform a continuous approximation of control nodes' semantics by iteratively employing the *sigmoid* function:

$$\| \rightarrow C\, BT_1\| = \sigma(C) \cdot \|BT_1\| \tag{5}$$

$$\|?\, C\, BT_1 ... BT_n\| = (1-\sigma(C))[\|BT_1\| + \sum_{i=2}^{n} \prod_{j=1}^{i-1} (1-\sigma(C_j))\|BT_i\|] \tag{6}$$

where σ is *sigmoid* function, C_j is the top-leftmost condition node of BT_j, $\sigma(\cdot)$ and $1-\sigma(\cdot)$ approximate the continuous representation of Sequence and Fallback control logic, respectively. A nonterminal BT may be extended to Act, which is not guarded by a condition. Since an action node returns *success* once it is executed, we set $\sigma(\cdot) = 1$. Note that we assume the action space is additive, as in the Ant environment where moving down and left are two additive actions. Otherwise, the aforementioned formulas represent only a relatively reasonable approximation.

3.3. Extraction Algorithm

Once we have learned the derivation graph, we can obtain a discrete BT architecture based on ω. A conceivable way would be greedily replacing each node with the most likely architecture. However, the performance ranked by weights in the derivation graph may be inaccurate. As shown in Fig. 3, a node contains several partial architectures, which may be co-adapted during training via node sharing Cui and Zhu (2021). Additionally, continuous approximation of the control nodes' semantics also brings trouble in extracting discrete architectures. Considering a BT with Sequence as root, when its conditional judgment is *False*, which causes $\sigma(C) = \mathbf{0}$, the whole output of the BT will be $\mathbf{0}$ too. This anomaly arises not from BT computation, but from the continuous approximation of control node semantics, which introduces misleading signals that degrade learning.

We introduce an extraction algorithm by utilizing Fallback nodes to address those potential issues. The algorithm traverses the derivation graph in a breadth-first manner, maintaining a queue

that contains the nodes to be processed. In each iteration, it dequeues one node and identifies the following processed candidates by comparing the probabilities. The insight is that when the probabilities of multiple architectures are particularly close, they may be co-adaptive. We then add a special Fallback without condition node to preserve all these architectures in ascending order of probability. If one dominates the choice, we directly replace the current node with this architecture. To overcome the disadvantage of continuous approximation, we add a Fallback as a new root and an Action node to capture unexpected actions. The extraction procedure is depicted in Fig. 5. Finally, we fine-tune all execution nodes' parameters within the extracted BT using standard RL algorithms, leveraging the parameter values previously learned during the architecture search step. The pseudocode is depicted in Algorithm 1 in the appendix.

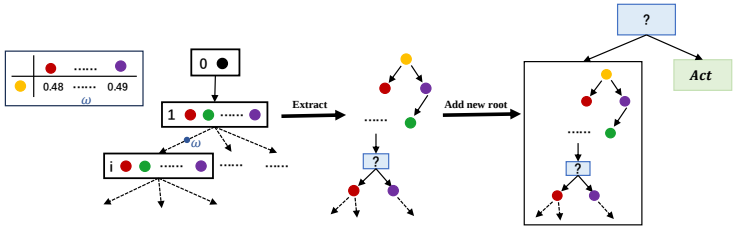


Figure 5: Extraction with identifying co-adaptation and adding new root node.

4. Experiments And Evaluations

Our experiments are designed to answer three research questions: **RQ1** Is our proposed *behavior tree differentiable synthesis* framework more effective than the state-of-the-art methods? **RQ2** Can the baselines and BTs synthesized by our framework generalize to novel scenarios without further learning? **RQ3** How significant is the use of *extraction algorithm* before the fine-tune process?

4.1. Experimental Setup

Evaluation Task. We evaluated our method on both discrete and continuous control benchmarks. All those benchmarks are collected from Gym[1], MiniGrid[2], and Mujoco[3]. We choose five tasks from Gym: Acrobot, Cart Pole, Mountain Car (discrete and continuous), and Pendulum. We choose two classes of MiniGrid tasks: Empty and Crossing. We also selected four tasks from a more complex benchmark Mujoco: Half Cheetah, Ant-Random, Ant-Maze, and Pusher. The correspondence between benchmarks and research questions is shown in Table 1. Details about tasks can refer to Appendix C.

Table 1: Correspondence between benchmarks and research questions

Target/Task	Gym				MiniGrid		Mujoco		
	Acrobot	Cart Pole	Mountain Car	Pendulum	Empty	Crossing	Half Cheetah	Ant	Pusher
Effectiveness	✔	✔	✔	✔	✔	✔	✔	✔	✔
Generalization					✔	✔	✔	✔	
Ablation			✔	✔				✔	

Evaluation Metrics. We compare the average final scores over 10,000 runs for the Gym. For MiniGrid tasks, we record the mean rewards and success rates over 1000 random seeds. In Mujoco tasks, we compare the agent's final distance from the target position.

1. https://gymnasium.farama.org/
2. https://minigrid.farama.org/environments/minigrid/
3. https://robotics.farama.org/envs/MaMuJoCo/#

Main Baselines. We select six baselines involving BT synthesis frameworks IRLBT Zhao et al. (2023) and RL frameworks including DDQN, DDPG, SAC, TRPO, and PPO. Table 6 lists their applicable action spaces. We use the original settings and do not consider variants. IRLBT defines two basic units Execution Units (EUs) and Behavior Units (BUs), and the sizes of generated BTs are determined by the total number of EUs and BUs. We chose different sizes for comparison. The implementation details and hyperparameters are shown in Appendix D and F.

4.2. Main Results

4.2.1. RQ1: EVALUATION OF EFFECTIVENESS.

We evaluate our method across discrete/continuous environments. See Appendix H for learned BTs.
Discrete Action Space. These tasks include Acrobot, Mountain Car, and Cart Pole. Our baselines include IRLBT, DDQN, and PPO. We chose sizes 4, 16, and 32 in IRLBT for comparison, respectively. Table 2 presents the mean and standard deviation of final scores. Our framework successfully solves all tasks and outperforms or matches the performance of RL methods. Compared to IRLBT, our framework excels in all benchmarks, while IRLBT cannot even solve some tasks (represented by /). In the Cart Pole task, IRLBT could hardly balance the pole (less than 10 steps) of any size. This is probably because IRLBT myopically chooses one variable at each step and cannot backtrack. In contrast, our approach constantly adjusts the probabilities of all possible BTs.

Table 2: Performance comparison with discrete action space.

Task	Method					
	IRLBT-4	IRLBT-16	IRLBT-32	DDQN	PPO	OURS
Acrobot	-311.60±52.36	-263.44±31.73	-182.15±26.55	-92.94±31.00	-89.75±23.81	**-84.30±22.67**
Mountain Car	/	-189.65±36.48	-175.07±22.04	-150.90±11.96	-167.09±20.27	**-145.22±4.42**
Cart Pole	9.52±0.97	9.45±0.94	9.59±1.07	**500.00±0.00**	**500.00±0.00**	449.70±66.55

Continuous Action Space. We also evaluate our method with continuous tasks: Mountain Car, Pendulum, Half Cheetah, Ant (Random and Maze), and Pusher. The baselines include DDPG, SAC, TRPO, and PPO. Fig. 6 depicts the learning curves for the rewards or final distance. Results are averaged over three random seeds. In Mujoco tasks, TRPO and PPO both failed to reach the target, but our framework can successfully solve those tasks. The high interpretability of BTs is not sacrificed at the expense of performance. In the Ant task, a TRPO policy with a three-layer feed-forward network has approximately 100,000 parameters, which is 2438 times our learned BTs. BTs' rich control structures implicitly compress the exploration space and provide an advantage of low-dimensional parameters.

4.2.2. RQ2: EVALUATION OF GENERALIZATION.

An evaluation of the generalization is conducted across two critical dimensions: scalability to larger sizes and adaptability to more complex environmental configurations. Specifically, for the Empty task, we first train a policy from scratch on a 5×5 grid (in the green box), then directly evaluate this policy on larger 6×6, 8×8, 16×16 and even 100×100 grids. In the Crossing task, we train on the N9S1 scenario (in the green box) and then test on four distinct scenarios: N9S2, N9S3, N11S5 and N21S9, where N denotes the size of the world and S represents the number of horizontal or vertical walls that traverse the grid world. We are also interested in whether the policies can generalize to unseen configurations. We evaluate the policy on **Lava Crossing** which is a variant of Crossing with lava streams that end the episode with a zero reward if the agent touches them. Those tasks

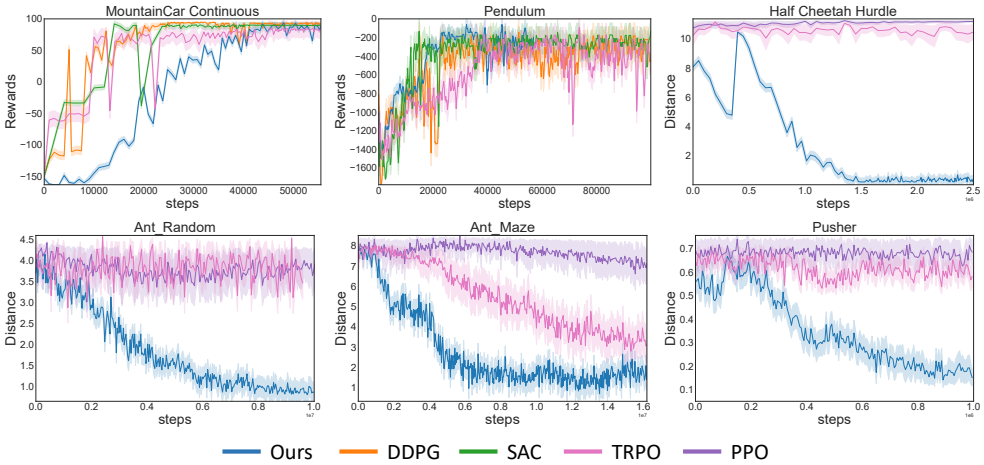


Figure 6: Comparison with baselines for tasks with continuous action space. The x-axis denotes the number of environment interaction steps and the y-axis records the rewards or final distance.

are known to be challenging due to their sparse reward and partial observability. We modify raw input with ***abstract states*** deriving from abstract state predicate evaluations (Appendix D.3). PPO and PPO-abs which also take abstract states as inputs are considered as two important baselines.

Table 3: Performance and success rate (in parentheses) comparison for generalization.

Task		Methods		
		Ours	PPO	PPO-abs
Empty	5 × 5	**0.94 ± 0.03 (1.000)**	0.81 ± 0.18 (0.992)	0.75 ± 0.23 (0.974)
	6 × 6	**0.95 ± 0.03 (1.000)**	0.68 ± 0.30 (0.911)	0.71 ± 0.26 (0.953)
	8 × 8	**0.94 ± 0.03 (1.000)**	0.42 ± 0.31 (0.750)	0.56 ± 0.26 (0.895)
	16 × 16	**0.96 ± 0.02 (1.000)**	0.23 ± 0.29 (0.468)	0.66 ± 0.18 (0.975)
	100 × 100	**0.99 ± 0.01 (1.000)**	0.16 ± 0.28 (0.310)	0.97 ± 0.01 (**1.000**)
Crossing	N9S1	**0.85 ± 0.11 (0.916)**	0.39 ± 0.34 (0.631)	0.40 ± 0.36 (0.597)
	N9S2	**0.74 ± 0.23 (0.870)**	0.26 ± 0.32 (0.467)	0.36 ± 0.36 (0.559)
	N9S3	**0.76 ± 0.26 (0.907)**	0.25 ± 0.31 (0.453)	0.39 ± 0.35 (0.618)
	N11S5	**0.74 ± 0.34 (0.841)**	0.14 ± 0.24 (0.313)	0.46 ± 0.34 (0.727)
	N21S9	**0.46 ± 0.47 (0.479)**	0.03 ± 0.10 (0.078)	0.12 ± 0.22 (0.317)
Lava Crossing	N9S1	**0.68 ± 0.31 (0.757)**	0.02 ± 0.12 (0.022)	0.11 ± 0.29 (0.139)
	N9S2	**0.63 ± 0.40 (0.702)**	0.00 ± 0.04 (0.002)	0.04 ± 0.17 (0.045)
	N9S3	**0.68 ± 0.39 (0.743)**	0.00 ± 0.06 (0.005)	0.03 ± 0.15 (0.038)
	N11S5	**0.69 ± 0.39 (0.748)**	0.00 ± 0.00 (0.000)	0.01 ± 0.09 (0.011)

The result is shown in Table 3. All baselines perform significantly worse than before on both tasks. In contrast, the BTs synthesized by our framework achieve zero-shot generalization to larger task instances. Especially on the Lava Crossing task, our method outperforms baselines by a large margin. This indicates that our synthesized BTs have learned how to avoid obstacles (no matter whether walls or lava streams) and perform target navigation.

We also test the generalization of our learned BTs in continuous-space tasks (Appendix G). We reshaped the maze's size in the Ant Maze task and changed the number and height of hurdles in the Half Cheetah Hurdle task. The results of Ant Maze show that as the size becomes larger, the generalization of BTs deteriorates with an effective generalization limit of roughly 150% relative to the original maze size. For Half Cheetah Hurdle, the learned BT adapts well across scenarios with varying hurdle numbers and can handle scenarios up to 7.5× of the original height.

4.2.3. RQ3: ABLATION EVALUATION FOR EXTRACTION ALGORITHM.

We conduct an ablation study to evaluate the necessity of the extraction algorithm. The baseline (denoted by Greedy) directly extracts a BT architecture by replacing each node with the most likely architecture. Table 4 summarizes the mean and standard deviation of rewards (or final distances) and the number of environment steps (in ten thousand) until convergence for various algorithms. Our extraction algorithm outperforms the greedy extraction method in those tasks. Considering the impact of co-adaptation and continuous approximation, we take advantage of Fallback nodes to preserve potential architectures. Particularly in Pendulum, the greedy method gets much worse because it is trapped in training a single PID controller, which makes it impossible to solve this task.

Table 4: Performance and convergence comparison for different extraction methods

Method	MountainCar-Continuous		Pendulum		Ant Random		Ant Cross	
	Rewards	Steps	Rewards	Steps	Distance	Steps	Distance	Steps
Ours	**92.26 ± 1.05**	**2.3**	**-272.60 ± 161.81**	**5.7**	**0.87 ± 1.03**	**214.4**	**1.49 ± 1.73**	**149.2**
Greedy	83.89 ± 3.23	7.6	-1020.70 ± 171.34	10.9	1.83 ± 1.02	655.0	4.17 ± 1.27	500.0

5. Related Work and Conclusion

Policy representation is crucial in RL systems, usually categorized into neural network and symbolic types. Neural network policies suffer from poor interpretability and generalization, while symbolic policies are constructed through predefined rules or logical structures Alur et al. (2013), which are more interpretable and verifiable. It traditionally depends on expert-designed control algorithms Thomason et al. (2024) without incorporating learning techniques. Neural symbolic policy synthesis methods have emerged recently, which employ neural networks to facilitate synthesis processes. A wealth of literature has applied imitation learning to abstract networks' control logic into symbolic forms, like finite automata Giles et al. (1992), decision tree Bastani et al. (2018), program Verma et al. (2018, 2019), and behavior tree Zhao et al. (2023). However, imitation-guided methods often produce suboptimal policies due to the distillation gap. Balog et al. (2017); Parisotto et al. (2017) learn a probability distribution over DSL operators or programs given the input-output examples and then use it to guide search. While the above methods require stronger supervision, our framework generates BTs solely using rewards. Trivedi et al. (2021) synthesis programs by first learning an embedding space and then searching this space to find programs. This method requires a large dataset and only works on tasks with discrete states and actions. Our method applies to both discrete and continuous domains. Qiu and Zhu (2022); Cui and Zhu (2021) apply the method in Liu et al. (2019) to program synthesis, learning the probability distribution over all possible program derivations induced by a given grammar. However, applying this method to BTs synthesis is more challenging due to the larger search space, the greater diversity in control structures, and the discrete nature of outputs. More work about policy representation and BTs generation is in Appendix E.

Conclusion We propose a novel differentiable BT synthesis framework that jointly learns the architectures and execution nodes of BTs. Experiment results demonstrate that our method is effective. The ablation study indicates that our extraction algorithm excels in discovering valid architectures.

Acknowledgments

This research was supported by the NSFC Programs (No. 62032024).

References

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013. URL https://ieeexplore.ieee.org/document/6679385/.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 2499–2509, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html.

Zhongxuan Cai, Minglong Li, Wanrong Huang, and Wenjing Yang. BT expansion: a sound and complete algorithm for behavior planning of intelligent robots with behavior trees. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 6058–6065. AAAI Press, 2021. doi: 10.1609/AAAI.V35I7.16755.

Yue Cao and C. S. George Lee. Robot behavior-tree-based task generation with large language models. In *Proceedings of the AAAI 2023 Spring Symposium on Challenges Requiring the Combination of Machine Learning and Knowledge Engineering (AAAI-MAKE 2023)*, volume 3433 of *CEUR Workshop Proceedings*, 2023.

Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019*, pages 1294–1303. IEEE, 2019. doi: 10.1109/ICCV.2019.00138.

Xinglin Chen, Yishuai Cai, Yunxin Mao, Minglong Li, Wenjing Yang, Weixia Xu, and Ji Wang. Integrating intent understanding and optimal behavior planning for behavior tree generation from human instructions. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 6832–6840. International Joint Conferences on Artificial Intelligence Organization, 2024. doi: 10.24963/ijcai.2024/755.

Michele Colledanchise and Petter Ögren. Behavior trees in robotics and AI: an introduction. *CoRR*, abs/1709.00084, 2017. URL http://arxiv.org/abs/1709.00084.

Michele Colledanchise, Diogo Almeida, and Petter Ögren. Towards blended reactive planning and acting using behavior trees. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8839–8845, 2019. doi: 10.1109/ICRA.2019.8794128.

Guofeng Cui and He Zhu. Differentiable synthesis of program architectures. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 11123–11135, 2021.

Renato de Pontes Pereira and Paulo Martins Engel. A framework for constrained and adaptive behavior-based agents, 2015.

Kevin French, Shiyu Wu, Tianyang Pan, Zheming Zhou, and Odest Chadwicke Jenkins. Learning behavior trees from demonstration. In *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*, pages 7791–7797. IEEE, 2019. doi: 10.1109/ICRA.2019.8794104.

Yanchang Fu, Long Qin, and Quanjun Yin. A reinforcement learning behavior tree framework for game ai. In *Proceedings of the 2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering*, pages 573–579. Atlantis Press, 2016/08. ISBN 978-94-6252-220-6. doi: 10.2991/essaeme-16.2016.120.

C. Lee Giles, Clifford B. Miller, Dong Chen, Hsing-Hen Chen, Guo-Zheng Sun, and Yee-Chun Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Comput.*, 4(3):393–405, 1992. doi: 10.1162/NECO.1992.4.3.393. URL https://doi.org/10.1162/neco.1992.4.3.393.

Simona Gugliermo, Erik Schaffernicht, Christos Koniaris, and Federico Pecora. Learning behavior trees from planning experts using decision tree and logic factorization. *IEEE Robotics Autom. Lett.*, 8(6):3534–3541, 2023. doi: 10.1109/LRA.2023.3268598.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. volume abs/1812.05905, 2018. URL http://arxiv.org/abs/1812.05905.

Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *Int. J. Robotics Res.*, 40(4-5), 2021. doi: 10.1177/0278364920987859. URL https://doi.org/10.1177/0278364920987859.

Matteo Iovino, Jonathan Styrud, Pietro Falco, and Christian Smith. Learning behavior trees with genetic programming in unpredictable environments. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4591–4597, 2021. doi: 10.1109/ICRA48506.2021.9562088.

Mart Kartasev and Aron Granberg. Integrating reinforcement learning into behavior trees by hierarchical composition. 2019.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016*, 2016. URL http://arxiv.org/abs/1509.02971.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=S1eYHoC5FX.

Artem Lykov and Dzmitry Tsetserukou. Llm-brain: Ai-driven fast generation of robot behaviour tree based on large language model. 2023. URL https://arxiv.org/abs/2305.19352.

Matthias Mayr, Konstantinos I. Chatzilygeroudis, Faseeh Ahmad, Luigi Nardi, and Volker Krüger. Learning of parameters in behavior trees for movement skills. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - Oct. 1, 2021*, pages 7572–7579. IEEE, 2021. doi: 10.1109/IROS51168.2021.9636292.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=rJ0JwFcex.

Shubham Pateria, Budhitama Subagdja, Ah-Hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.*, 54(5):109:1–109:35, 2022. doi: 10.1145/3453160. URL https://doi.org/10.1145/3453160.

Nicholas Potteiger and Xenofon D. Koutsoukos. Safe explainable agents for autonomous navigation using evolving behavior trees. In *IEEE International Conference on Assured Autonomy, ICAA 2023, Laurel, MD, USA, June 6-8, 2023*, pages 44–52. IEEE, 2023. doi: 10.1109/ICAA58325.2023.00014. URL https://doi.org/10.1109/ICAA58325.2023.00014.

Nicholas Potteiger and Xenofon D. Koutsoukos. Safeguarding autonomous UAV navigation: Agent design using evolving behavior trees. In *IEEE International Systems Conference, SysCon 2024, Montreal, QC, Canada, April 15-18, 2024*, pages 1–8. IEEE, 2024. doi: 10.1109/SYSCON61195.2024.10553469. URL https://doi.org/10.1109/SysCon61195.2024.10553469.

Nicholas Potteiger, Ankita Samaddar, Hunter Bergstrom, and Xenofon D. Koutsoukos. Designing robust cyber-defense agents with evolving behavior trees. In *International Conference on Assured Autonomy, ICAA 2024, Nashville, TN, USA, October 10-11, 2024*, pages 1–10. IEEE, 2024. doi: 10.1109/ICAA64256.2024.00011. URL https://doi.org/10.1109/ICAA64256.2024.00011.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=6Tk2noBdvxt.

Emily Scheide, Graeme Best, and Geoffrey A. Hollinger. Behavior tree learning for robotic task planning through monte carlo DAG search over a formal grammar. In *IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021*, pages 4837–4843. IEEE, 2021. doi: 10.1109/ICRA48506.2021.9561027.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning,*

ICML 2015, Lille, France, 6-11 July 2015, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889–1897. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/schulman15.html.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. volume abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. doi: 10.1038/NATURE16961. URL https://doi.org/10.1038/nature16961.

Jonathan Styrud, Matteo Iovino, Mikael Norrlöf, Mårten Björkman, and Christian Smith. Combining planning and learning of behavior trees for robotic assembly. In *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*, pages 11511–11517. IEEE, 2022. doi: 10.1109/ICRA46639.2022.9812086.

Ioan Alexandru Sucan, Mark Moll, and Lydia E. Kavraki. The open motion planning library. *IEEE Robotics Autom. Mag.*, 19(4):72–82, 2012. doi: 10.1109/MRA.2012.2205651. URL https://doi.org/10.1109/MRA.2012.2205651.

Wil Thomason, Zachary K. Kingston, and Lydia E. Kavraki. Motions in microseconds via vectorized sampling-based planning. In *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*, pages 8749–8756. IEEE, 2024. doi: 10.1109/ICRA57147.2024.10611190. URL https://doi.org/10.1109/ICRA57147.2024.10611190.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 25146–25163, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/d37124c4c79f357cb02c655671a432fa-Abstract.html.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2094–2100. AAAI Press, 2016. doi: 10.1609/AAAI.V30I1.10295. URL https://doi.org/10.1609/aaai.v30i1.10295.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5052–5061. PMLR, 2018. URL http://proceedings.mlr.press/v80/verma18a.html.

Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 15726–15737, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/5a44a53b7d26bb1e54c05222f186dcfb-Abstract.html.

Adam Wathieu, Thomas R. Groechel, Haemin Jenny Lee, Chloe Kuo, and Maja J. Mataric. RE: bt-espresso: Improving interpretability and expressivity of behavior trees learned from robot demonstrations. In *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*, pages 11518–11524. IEEE, 2022. doi: 10.1109/ICRA46639.2022.9812046.

Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *CoRR*, abs/1911.10635, 2019. URL http://arxiv.org/abs/1911.10635.

Qi Zhang, Lin Sun, Peng Jiao, and Quanjun Yin. Combining behavior trees with maxq learning to facilitate cgfs behavior modeling. In *2017 4th International Conference on Systems and Informatics (ICSAI)*, pages 525–531, 2017. doi: 10.1109/ICSAI.2017.8248348.

Chenjing Zhao, Chuanshuai Deng, Zhenghui Liu, Jiexin Zhang, Yunlong Wu, Yanzhen Wang, and Xiaodong Yi. Interpretable reinforcement learning of behavior trees. In *Proceedings of the 15th International Conference on Machine Learning and Computing, ICMLC 2023, Zhuhai, China, February 17-20, 2023*, pages 492–499. ACM, 2023. URL https://doi.org/10.1145/3587716.3587798.

Appendix A. More Details about Behavior Trees

A.1. Classic Formulation of BTs

A BT is a directed tree where we apply the standard meanings of root, child, parent, and leaf nodes, which control the agent's decision-making behavior. The leaf nodes of BTs are *execution nodes* that can be classified into *action nodes* (represented by green rectangles) and *condition nodes* (represented by purple ovals). The non-leaf nodes are called *control flow nodes*, including four types of node: *Sequence, Fallback, Parallel* and *Decorator*. In this work, we mainly focus on the usage of *Sequence nodes* (represented by →) and *Fallback nodes* (represented by ?).

Condition: A condition node that checks the proposition related to current states and returns *success* if it satisfies the specified situation; otherwise, it returns *failure*. Like *Low battery*, *At pos*. Note that a Condition node never returns a status of *running*.

Action: An action node like *PickUp* and *Goto pos*, that performs a specified behavior and returns *success*, *failure*, or *running* depending on the execution results. For example, while the action *Goto pos* is ongoing, it returns *running*. If it is successfully completed at the next tick, it returns *success*.

Sequence: A control node that ticks its children from left to right. A Sequence node is used when some actions, or condition checks, are meant to be carried out in sequence. It returns *failure* or *running* immediately when a child returns either *failure* or *running*. It returns *success* if and only if all its children return *success*. Note that when a child returns *running* or *failure*, the Sequence node does not route the ticks to the next child (if any). For example, in Fig. 7, if *At pos* is false, the Sequence node will return *failure* immediately and will not tick *PickUp* and *Goto pos*.

Fallback: A control node that ticks its children from left to right. A Fallback node is used when a set of actions represents alternative ways of achieving a similar goal. It returns *success* or *running* immediately when a child returns either *success* or *running*. Otherwise, it returns *failure*. Note that when a child returns *running* or *success*, the Sequence node does not route the ticks to the next child (if any). For example, in Fig. 7, if *Low battery* is true, the Fallback node will return *success* immediately and no more nodes will be ticked.

A.2. Execution Example

Fig. 7 shows a BT example that controls the robot to pick up a key from a specific position when the robot has sufficient power.

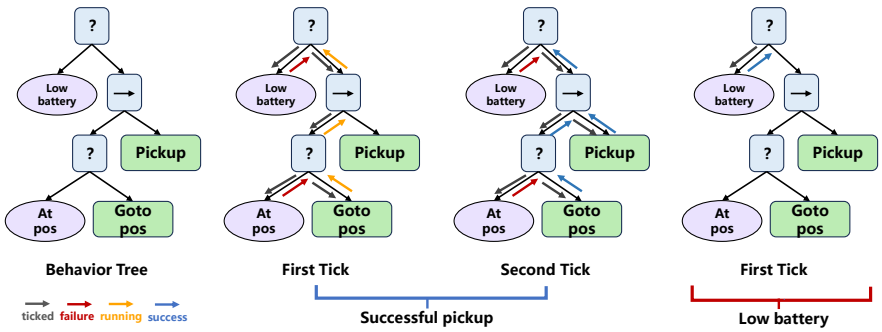


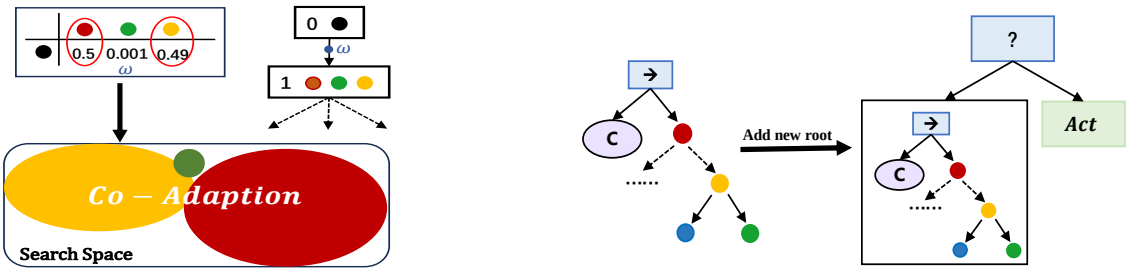
Figure 7: BT example.

The execution of BTs starts from the root and infinitely delivers ticks to its children at a particular frequency. *Ticks* can be seen as a signal to identify which node is executable. We take one possible execution of the BT example for illustration, which requires that the robot first go to the specific position, then pick up a key. The whole process must be carried out with a high battery and stopped if the battery is low. We assume that the conditions *Low-battery* and *AtPos* don't hold initially. In the first tick, the condition node *Low-battery* is ticked but returns failure, then the sequence node is ticked due to the functionality of these control flow nodes. *AtPos* also returns failure, and the action node *GotoPos* is ticked and returns running. Status *running* means the action is in progress but not completed. The status running is propagated to the root since all the control flow nodes will return running if one of their children returns running. Similar to the first tick, the action node *GotoPos* is ticked again but returns success in the second tick. During the second tick, the action node *Pickup* is also ticked since the first child of the sequence node returns success, and *Pickup* returns success. The success of *Pickup* means the pickup task succeeds. If the conditions *Low-battery* hold initially, it returns success and the whole BT returns success. The rest of the nodes will not be ticked in this situation. This alerts the operator that the robot needs to be recharged.

Appendix B. The Extraction Algorithm and Complexity Analysis

B.1. Co-Adaptation and The Abnormality Caused by Continuous Approximation

As shown in Fig. 8(a), a node contains several partial architectures, and those architectures may partition the search space into disjoint segments and collaboratively model the entire search space. In other words, it is the weighted sum of those partial architectures' outputs that produces satisfying performance, not the one with the largest probability. Therefore, when obtaining a discrete BT structure, the greedy approach that selects the partial architectures with the highest probability will lead to an unreasonable structure. In our work, *Fallback* nodes are utilized to address this potential issue.



(*a*) *Fallback* Semantic (*b*) *Sequence* Semantic

Figure 8: (a) Co-adaptation phenomenon. (b) The abnormality caused by the continuous approximation.

Continuous approximation of the control nodes' semantics also brings trouble in extracting discrete architectures. Considering a BT with *Sequence* as root, as shown in Fig. 8(b), when the *Condition* node (represented by purple ovals) returns failure, the whole output of the BT will be **zero**. However, this anomalous action, especially when **zero** is in the action space like Mountain-car, will introduce misleading signals that degrade learning performance. In this paper, we add a Fallback as a new root node and an Action node to capture unexpected actions.

B.2. Pseudocode of the Extraction Algorithm

As illustrated in Algorithm 1, the extraction algorithm takes a trained derivation graph $\mathcal{G}$ and a threshold λ as input and outputs a BT with the discrete architecture. The derivation graph $\mathcal{G}$ is implemented as a multi-pointer linked list. Each node contains multiple pointers that indicate the partial architectures it holds, along with a weight representing the probability of selecting these partial structures. Besides, our algorithm maintains a queue of nodes to be processed, which is initialized to $[root]$ (cf., line 1 to line 2). At each iteration, the algorithm dequeues a node and identifies processed candidates using the IDENTIFY function (cf., line 15 to line 23) with a threshold λ (cf., line 3 to line 5). This function extracts the most likely partial architectures q and compares the probabilities of the remaining candidates with pro_q. Any child with probabilities particularly close to pro_q will be added to the candidate set all_cand. This set will be empty if the current node is an execution node that does not require further extensions. After identification, we determine which new nodes should be added to the queue (*len*$(next\_n) > 0$) and detect co-adaptation by evaluating the candidate set's length ($len(next\_n) > 1$) (cf., line 6 to line 11). Finally, we add a new root node and a new action node (cf., line 12 to line 14).

Algorithm 1: Extraction Algorithm

Input: Trained derivation graph $\mathcal{G}$, threshold λ
Output: Synthesized behavior tree BT

```
1  root = G.root
2  queue = [root]
3  while len(queue) != 0 do
4      n = queue.pop()
5      next_n = IDENTIFY(n, λ)
6      if len(next_n) > 0 then
7          if len(next_n) > 1 then
8              next_n = [Fallback(next_n)]
9          n.next = next_n
10         for cand in next_n do
11             queue.append(cand)
12 act = Action()
13 new_root = Fallback([root, act])
14 return new_root
15 Function IDENTIFY(node, λ) is
16     all_cand = []
17     if node.type == "Control" then
18         q, pro_q = Max_Probability(node)
19         resort(node.w)
20         for w' in node.w do
21             if pro_q - w' < λ then
22                 all_cand.append(node with w')
23     return all_cand
```

B.3. Complexity Analysis of the Extraction Algorithm

The time complexity of the extraction algorithm depends on several factors, including the depth of the tree (d), the number of production rules (m), and the maximum number of nonterminals contained in a rule (n). In the best-case scenario, each node has a child that dominates the weights, resulting in a time complexity of $O(n^d)$. This occurs when the threshold is too strict, limiting exploration to exactly one production rule. When there is no co-adaptation, our extraction algorithm degenerates into a greedy method. Conversely, the worst-case scenario arises when the threshold is too permissive, requiring the algorithm to explore nearly all possible rule combinations across the tree. In this case, the time complexity would be $O((n \cdot m)^d)$. In practice, the time complexity typically lies between the best and worst cases due to the threshold-based pruning mechanism. The threshold plays a critical role in the extraction algorithm's efficiency as we mentioned above. The choice of threshold is a trade-off between accuracy and efficiency and can be tuned based on the requirements of specific applications.

Based on the ablation studies shown in Fig. 4 in the experimental section, we observe that the extraction algorithm significantly enhances the overall efficiency and reduces total training time. This improvement occurs because the extraction algorithm generates a well-structured initial BT, which decreases the search space for the RL fine-tuning process. Regarding the running time of the extraction algorithm itself, it is negligible compared to the time required for RL training. The extraction algorithm operates efficiently, and its computational cost is minimal within the context of the overall pipeline. Thus, while RL fine-tuning does add to the running time, the extraction algorithm effectively offsets this by accelerating the convergence of the RL process.

B.4. Discussion about Real-world Applications

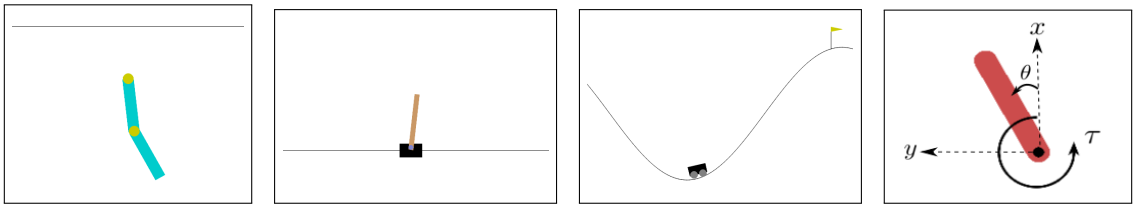
While our proposed framework achieves significant performance in both discrete and continuous tasks, some may question the potential of this framework for real-world applications. There are some works Potteiger and Koutsoukos (2023, 2024); Potteiger et al. (2024) that learns BTs in abstract environments and then transfer to the real world. Using the abstract environments leads to an efficient and safe learning process since the agent is not interacting with the real environment. For example, in UAV navigation and obstacle avoidance tasks, Potteiger and Koutsoukos (2023, 2024) first automatically construct BT using GP in an abstract grid environment, and then deploy and transfer this BT to a realistic simulation. Potteiger et al. (2024) designs BTs for robust cyber-defense agents to defend against adaptive cyber-attackers. This method also starts by synthesizing BTs in a novel *Cyber-Firefighter Abstract Environment*. Inspired by these works, we believe that our proposed method can also be well transferred to real-world applications.

Appendix C. Detailed Descriptions of the Tasks and the Input of BTs

C.1. Gymnasium Tasks

Tasks from the Gymnasium that occurred in our experiments are briefly introduced below. For more details, please refer to the Gymnasium Documentation. As aforementioned, we utilize *raw states* as the input in these benchmarks.

a) Acrobot. This system features two links connected linearly to form a chain, with one end fixed. The joint between the two links is actuated. The objective is to apply torques at the actuated



(*a*) **Acrobot** (*b*) **Cart Pole** (*c*) **Mountain Car** (*d*) **Pendulum**

Figure 9: Gymnasium Tasks used in our experiments.

joint to swing the free end of the chain above a specified height, starting from an initial position where it hangs downward.

b) Cart Pole. In this setup, a pole is connected by an unactuated joint to a cart that moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

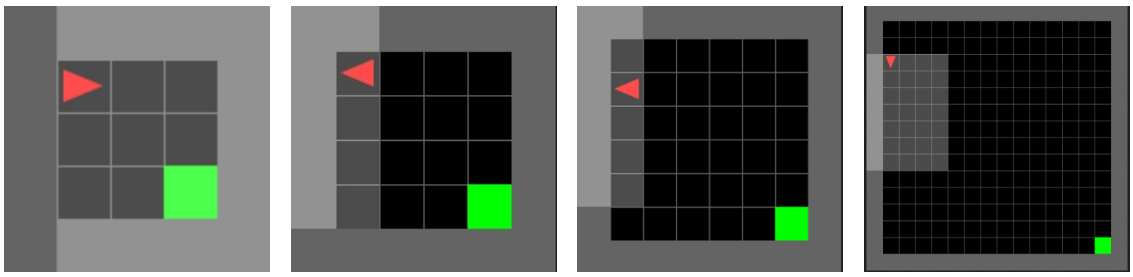
c) Mountain Car. The Mountain Car MDP is deterministic and consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The objective is to accelerate the car strategically to reach the goal state on top of the right hill. There are two versions of this task in the Gymnasium: one with discrete actions and one with continuous actions. We evaluated our method on both of them.

d) Pendulum. This system consists of a pendulum attached at one end to a fixed point, with the other end free. The pendulum starts in a random position, and the goal is to apply torque at the free end to swing it upright, positioning its center of gravity directly above the fixed point.

C.2. MiniGrid Tasks

We conducted experiments using two types of MiniGrid tasks: the Empty Task and the Crossing Task. Please refer to MiniGrid Documentation for more information. In this benchmark, a positive reward is given only when the goal condition is achieved. Otherwise, the reward is always 0. We select increasingly larger map sizes and more complex environment configurations to verify the generality of our method. Note that we defined *abstract states* as inputs for BTs, which are detailed in Appendix D.3.

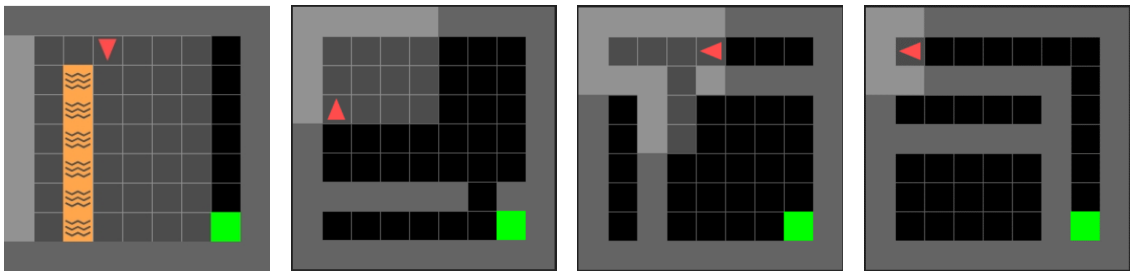
a) The Empty Task. This environment is an empty room where the goal for the agent is to reach a designated green square, which provides a sparse reward. A small penalty is subtracted from the number of steps to reach the goal. This environment is useful, with small rooms, to validate that the RL algorithm works correctly, and with large rooms to experiment with sparse rewards and exploration. In the random variants of this environment, the agent begins at a random position for each episode.



(*a*) **Empty** (Default) (*b*) Empty 6 × 6 (*c*) Empty 8 × 8 (*d*) Empty 16 × 16

Figure 10: The Empty Tasks in MiniGrid that are used in our experiments.

b) The Crossing Task. In this task, the agent must reach the green goal square located at the opposite corner of the room while avoiding rivers of deadly lava that can terminate the episode in failure. Each lava stream runs across the room either horizontally or vertically, and each has a single crossing point that can be safely used. Fortunately, a safe path to the goal is guaranteed to exist. This environment is useful for studying safety and safe exploration. An alternative version of this task replaces lava with walls and was also utilized for evaluating our methods.



(*a*) **Crossing** (Default) (*b*) Crossing S9N1 (*c*) Crossing S9N2 (*d*) Crossing S9N3

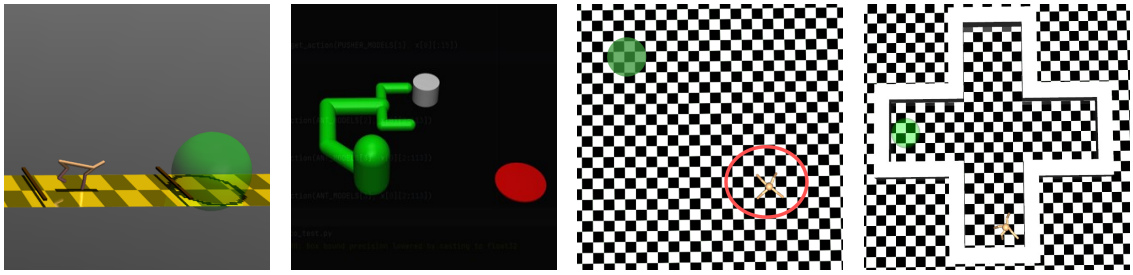
Figure 11: The Crossing Tasks in MiniGrid that are used in our experiments.

C.3. Mujoco Tasks

As illustrated in Fig. 12, we selected three representative simulation tasks in Mujoco to evaluate the effectiveness of our method: Half Cheetah, Pusher, and Ant (including both Random and Maze maps). Below are brief descriptions of each task; please refer to Mujoco Documentation for further information.

a) Half Cheetah (Hurdle). The Half Cheetah is a 2D robot with nine parts and eight joints, including two legs. The goal is to make the cheetah run forward (to the right) as fast as possible to reach the target with a radius of 1. In the Half Cheetah Hurdle version, the Cheetah is also required to jump over hurdles. In this task, the input $\mathcal{X}$ includes the position of the next hurdle x_{next} and the Half Cheetah's back foot x_{back}, the distance from Half Cheetah's back foot to next hurdle $||x_{back}, x_{next}||_1$ are then computed as additional input for the Condition nodes.

b) Pusher. The Pusher task involves a multi-jointed robot arm that consists of shoulder, elbow, forearm, and wrist joints. The goal is to move a target cylinder (the object) to a specified goal position using the robot's end effector. The input $\mathcal{X}$ of behavior tree are $x_{obj}, y_{obj}, x_{arm}, y_{arm}$, and then computes $||x_{obj}, y_{obj}||_2$, $||x_{arm}, y_{arm}||_2$ as inputs for the Condition nodes.



(*a*) **Half Cheetah** (*b*) **Pusher** (*c*) **Ant** (Random) (*d*) **Ant** (Maze)

Figure 12: Simulation Tasks in Mujoco that are used in our experiments.

c) Ant. The Ant is a 3D quadruped robot with a torso that has free rotational movement and four attached legs. In the Random task, the Ant starts at the origin and is required to reach randomly sampled targets within a confined circular region. In the Maze task, the Ant must reach a randomly designated goal area (including coordinates like $[6, 6]$, $[6, -6]$, and $[12, 0]$) while navigating through a maze with walls as obstacles. The input $\mathcal{X}$ of behavior tree consists of $x_{ant}, y_{ant}, x_{goal}, y_{goal}$, along with $arctan(\frac{y_{ant}}{x_{ant}})$ and $||x_{ant}, y_{ant}||_2$ as the overall inputs.

Appendix D. Implementation Details

In this section, we present implementation details, including a complexity analysis of the derivation graph computation, the DSL designs of Condition and Actions nodes, and the abstract state of MiniGrid tasks.

D.1. Complexity Analysis of the Derivation Graph Execution

The complexity of the derivation graph computation is exponential (EXP), as discussed in Sec. 3. In the code implementation, we perform the computation in a bottom-up manner: we invoke each execution node and pass its result to wherever it is used in the tree. Similarly, intermediate results computed by shorter ones can be reused by longer ones. More importantly, for each BT contained in the derivation graph, we compute its results exactly once since the input to any BT is always the current environment state. This shared computation approach can significantly reduce the time cost of derivation graph execution. Notably, we applied an effective optimization strategy called *Node sharing* to our derivation graph, which can significantly decrease space complexity form $O((m \cdot n)^d)$ to $O(n^d)$. More information can be found in Cui and Zhu (2021).

For more complex tasks that require more production rules and a deeper derivation graph, we could optimize the architecture search procedure by applying further optimization strategies. For example, Progressive DARTS Chen et al. (2019) and Iterative Graph Unfolding Cui and Zhu (2021) gradually increase the tree depth d during the search procedure, focusing on higher-quality derivations by dropping the lowest-weighted derivations. These strategies would allow our framework to scale to more complex tasks.

D.2. Detailed DSL designs for Condition and Action nodes

In this paper, we argue that a linear function can sufficiently evaluate potential conditional judgments and serve as the DSL for Condition nodes. We consider four DSLs for Action nodes.

Affine Action. The DSL for affine action is defined as: $A := \theta_{a1} + \theta_{a2} \cdot x$, where θ_{a1}, θ_{a2} are parameters. Particularly, affine action can simply be a learnable constant θ_{a1} as θ_{a2} is $\mathbf{0}$. The DSL for discrete action space is always a trainable constant θ_{a1}. During execution, this constant resizes itself according to the constraints of the task's action space. Consider the Cart Pole task, where the valid actions are limited to the values zero and one. The actions generated by the derivation graph often need to be adjusted to the nearest valid action. This is typically implemented using the *round* function in code.

PID Action. A PID controller is a type of feedback loop commonly used in control systems to maintain a desired setpoint. The "PID" stands for Proportional, Integral, and Derivative, which are the three terms in the controller's output equation. The DSL of PID action is: $\mathbf{PID}_{\theta_P,\theta_I,\theta_D}(\epsilon, h, s) = \theta_P \cdot P + \theta_I \cdot I + \theta_D \cdot D$, where $P = (\epsilon - s)$ is proportional term, $I = \mathbf{fold}(+, \epsilon - h)$ is integral, and $D = \mathbf{peek}(h, -1) - s$ is derivative term. $\theta_P, \theta_I, \theta_D$ are parameters to be trained, ϵ is the target, and h is a history of previous states. The function **fold** takes as input an anonymous function **fun** x·f(x) that evaluates an expression f(x) over the input x. The function **peek** returns the most recent state in the history memory. Taking the Pendulum task as an example, the observation of Pendulum contains $\cos(\omega), \sin(\omega)$ and $\dot{\omega}$, where ω is the angle the pendulum makes with the vertical and $\dot{\omega}$ is the angular velocity. The goal is to stay the pendulum upright, so the fixed goal ϵ is set to [1, 0, 0].

Linear Recomposed Action. As complexity grows, it is beneficial to compose and reuse task-agnostic primitive actions. The DSL of linear recomposed action is: $A_r := \theta_1 \cdot a_1 + \cdots + \theta_M \cdot a_M$, where $\{a_i\}_{i=0}^{M}$ are pre-trained primitives. A_r recomposes primitive actions into a more complex action with parameters $\theta_1, \cdots, \theta_M \in \mathbb{R}^1$. In practice, we usually compute a weighted sum of primitives as: $\|A_r\|(s) = \sum_{i=0}^{M} p_i \cdot a_i(s)$ where composition weights $\{p_i\}_{i=0}^{M}$ are calculated using softmax. In the Mujoco benchmark, we require multiple primitive actions to compose higher-level skills. For example, we equipped Ant with four basic primitive policies, i.e., a_{up}, a_{down}, a_{left}, and a_{right}. For pusher, it requires two primitive skills a_{pusher_down} and a_{pusher_left}. In the Half Cheetah Hurdle environment, two simple policies a_{jump} and $a_{forward}$ are provided. All the primitives we used are borrowed from Qiu and Zhu (2022).

D.3. Abstract State of MiniGrid Tasks

In MiniGrid, each grid is encoded as a 3-dimensional tuple (OBJECT_IDX, COLOR_IDX, STATE) that indicates the object's types, object's colors and object's states. The agent moves through this grid with a partially visible field, interacting with objects and navigating toward goals. To make behavior tree synthesis more practical and interpretable, we modified the raw environment state to a *"abstract state"*. We defined several *predicates*, including: *front_is_clear*, *left_is_clear*, *right_is_clear*, *goal_on_left*, *goal_on_right*, *goal_present*, *front_is_obj*. *front_is_obj* can be instantiated according to the environment configuration and goal condition, for instance, *front_is_obj(key)*. Those predicates construct a higher-level representation of the environment based on raw state data. We form an abstract state by concatenating the Boolean values of the predicates as a binary vector. We also incorporate the agent's direction information into the abstract state to facilitate faster convergence, shown in Fig. 13.

Appendix E. Additional Related Works

Policy Representation. Policy representation is the key to a control system. In modern approaches, it can be categorized into two types: neural network policy and symbolic policy. Neural network

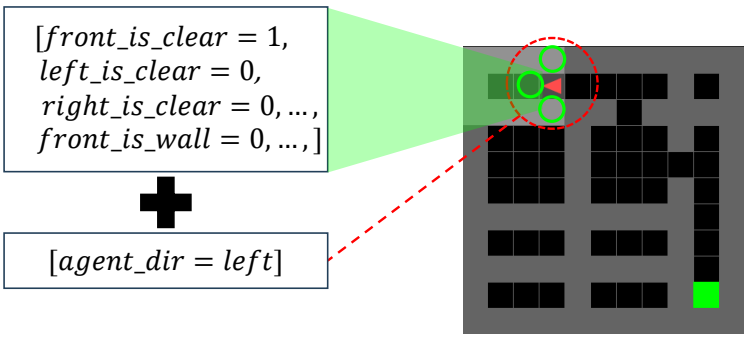


Figure 13: An example abstract state from the Crossing task in MiniGrid.

policies have achieved promising successes in many challenging control tasks due to their strong fitting capabilities. Classic algorithms are divided into value-based (e.g. DDPG van Hasselt et al. (2016)), policy-based (e.g. TRPO Schulman et al. (2015), PPO Schulman et al. (2017)) and hybrid methods (e.g. SAC Haarnoja et al. (2018), DDPG Lillicrap et al. (2016)). Neural network policies have made remarkable progress in recent years, such as HRLPateria et al. (2022) and MARL Zhang et al. (2019). However, neural network policies still suffer from poor interpretability and generalization.

Symbolic policies represent strategies through predefined rules or logical structures Alur et al. (2013), which have better interpretability and verifiability. Symbolic policy representation includes programs, finite state machines, decision trees, and behavior trees. This type of policy always relies solely on traditional control algorithms designed by experts based on domain knowledge Thomason et al. (2024); Sucan et al. (2012), without incorporating learning techniques. In recent years, neural symbolic policy synthesis methods have emerged, which employ neural networks to facilitate a synthesis process. Giles et al. (1992) extracted finite automata out of neural networks. VIPER Bastani et al. (2018) learns decision tree policies guided by a DNN policy via model compression and imitation learning. PIRL Verma et al. (2018) and PROPEL Verma et al. (2019) also employ imitation learning to synthesize programs. DeepCoder Balog et al. (2017) trains neural networks to predict a probability distribution over DSL operators and then uses it to guide the program synthesis. While the above methods require stronger supervision, LEAPS Trivedi et al. (2021) synthesizes programs solely from reward signals by first learning a program embedding space and then searching this space to find a program. pi_PRL Qiu and Zhu (2022) applies the method in DARTS Liu et al. (2019) to programmatic reinforcement learning, which synthesizes differentiable programmatic policies without a pre-trained oracle. However, applying this method to BTs synthesis is more challenging due to the larger architecture search space, the greater diversity of semantics in construction structures, and the discrete nature of the outputs.

Behavior Tree Generation. Various studies have been explored for synthesizing BTs automatically. Some methods Fu et al. (2016/08); Zhang et al. (2017); Kartasev and Granberg (2019); de Pontes Pereira and Engel (2015) replace nodes in a fixed BT with RL components, endowing BTs with learning capacity.

Learning from Demonstration (LfD) French et al. (2019); Wathieu et al. (2022); Gugliermo et al. (2023) has also been utilized to generate BTs via translating decision trees to BTs. Zhao et al. (2023) defines two basic units and builds BTs on those units. For the above methods, the expressiveness of the generated BTs is usually limited by fixed architectures or constrained transformation rules.

Some works synthesize BTs on the given agent's capabilities. Scheide et al. (2021) utilizes a Monte Carlo search to find BTs in a search space defined by a formal grammar and a set of capabilities. Some methods combine automated planners with BT synthesis Colledanchise et al. (2019); Cai et al. (2021) and construct a BT based on back-chaining, which starts from the goal conditions and iteratively links the actions that satisfy them. Genetic Programming Styrud et al. (2022); Iovino et al. (2021) is also widely used for BT synthesis, and its initial population is generated according to the given capabilities. More recent work Chen et al. (2024); Cao and Lee (2023); Lykov and Tsetserukou (2023) employs Large Language Models (LLMs) to generate BTs from natural language instructions. However, predefined execution nodes always link to static functions, with the drawbacks of being scenario-specific and requiring relearning when the setup changes.

Appendix F. Hyperparameters and Training Details

F.1. Configurations of the Derivation Graph

Table 5 summarizes the derivation graph depth and action DSLs used for different benchmarks.

Table 5: Derivation graph depth and action DSLs

Tasks	Depth	Action DSL Type
Gym(except Pendulum)	3	Constant
Pendulum	4	Constant/PID Controller
MiniGrid	4	Constant
Mujuco	6	Linear Recomposed Actions

F.2. Hyperparameters of Baseline Algorithms

The baseline algorithms and their hyperparameters we used are listed in Table 6. All neural network architectures of RL algorithms are two- or three-layer feed-forward networks with a hidden size of 256.

IRLBT Zhao et al. (2023) presents a behavior trees generation method that directly represents the policies generated by Q-learning and its derived algorithms in the form of BTs. It defines two basic units: Branching Units (BUs) which is a Sequence node with one condition and one Fallback, and Execution Units (EUs) which is a Sequence node with one condition and one action. At time step t with an environment state s_t, this method searches the units corresponding to s_t and updates the visit frequency, Q-values, potential division objects list and its highest expected reward increase ΔQ. If ΔQ exceeds the dynamic threshold Q_{TH}, the tree grows. This algorithm is similar to decision tree generation methods but differs in its representation of decision conditions, which are explicitly stored in Condition nodes.

Table 6: Baseline Methods and detailed Hyperparameters

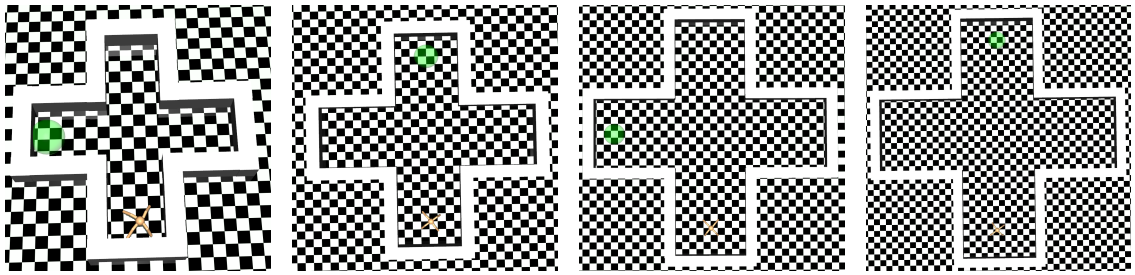
Baseline Methods	Description	Action Spaces	Hyperparameters
IRLBT Zhao et al. (2023)	A Q-learning-based BT synthesis framework	Discrete	Discounter factor: 0.8 Visit decay: 0.99 Split_max: 10e10 Split_decay: 0.99 Number of split: 7
DDQN van Hasselt et al. (2016)	Double Deep Q-Learning algorithms	Discrete	Learning rate: 0.002 Batch size: 64 Replay buffer size: 10000 Update interval: 10 Discounter factor: 0.98 Exploration Rate: 0.1
DDPG Lillicrap et al. (2016)	Actor-Critic algorithms with policy gradient	Continuous	Actor learning rate: 0.0005 Critic learning rate: 0.005 Batch size: 128 Replay buffer size: 100000 Update interval: 20 Discounter factor: 0.99 Exploration Rate: gradually decreases from 0.25 to 0.05.
SAC Haarnoja et al. (2018)	Soft actor-critic algorithms	Continuous	Actor learning rate: 0.0005 Critic learning rate: 0.005 Batch size: 256 Replay buffer size: 100000 Update interval: 5 Discounter factor: 0.99 Soft update τ: 0.005
TRPO Schulman et al. (2015)	Trust region policy optimization algorithms	Continuous	Batch size: 512 Replay buffer size: 100000 Update interval: 10 Discounter factor: 0.99 GAE: 0.97 KL-Divergence limit: 0.05 L2 regularization regression: 0.001
PPO Schulman et al. (2017)	Proximal policy optimization algorithms	Discrete Continuous	Actor learning rate: 0.0003 Critic learning rate: 0.001 Batch size: 128 Replay buffer size: 100000 Update interval: 10 Discounter factor: 0.99 GAE: 0.97 Clip ratio eps: 0.02

Appendix G. More Results on the Generalization Experiments

We investigate the generalization of our learned BTs in tasks involving continuous spaces. Additionally, we assess how well these BTs can adapt to larger state spaces and new configurations.

In the Ant Maze benchmark, we scale the maze size by factors of 0.8, 1.2, 1.5, and 1.8 (as shown in Fig.14) and directly evaluate the learned BT obtained from the original task. Since target positions

are adjusted accordingly, we record the final distance to the goal normalized by the agent's initial distance from the goal. Success rates are also recorded. Table 7 shows that our method demonstrates strong generalization in shrinking mazes but exhibits decreasing performance as the scaling factor increases, with an effective generalization limit of roughly 150% relative to the original environment size.



(*a*) Ant Maze (0.8×) (*b*) Ant Maze (1.2×) (*c*) Ant Maze (1.5×) (*d*) Ant Maze (1.8×)

Figure 14: Ant Maze with reshaped sizes.

Table 7: Generalization results of generalization in Ant Maze tasks. Results are averaged over 50 random executions

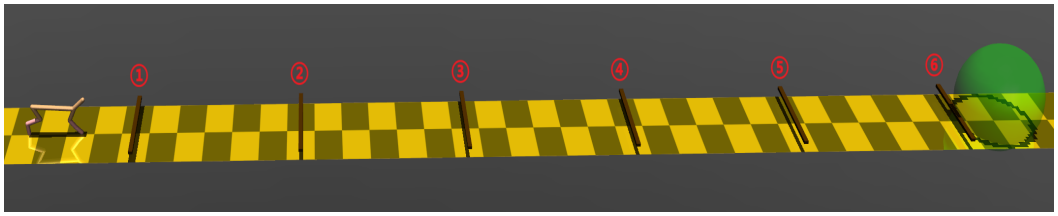
Tasks	Ant Maze				
	Original	0.8×	1.2×	1.5×	1.8×
Distance	0.128±0.184	0.182±0.292	0.144±0.192	0.156±0.158	0.180±0.154
Success_Rate	0.640	0.680	0.260	0.100	0.000

In the Half Cheetah Hurdle environment, we change the environment configurations from the number and height of hurdles. In the original tasks, a half cheetah must jump over three hurdles to reach the target with a radius of 1. We have set up the following variants: half cheetah with six hurdles, half cheetah with nine hurdles, and half cheetah with higher hurdles (shown in Fig.15). The results are shown in Table 8, where "6" and "9" indicate numbers and "n×" means height scalability. It can be seen that learned BT generalizes well across scenarios with varying numbers of hurdles. In environments with hurdles of varying heights, its generalization performance gradually deteriorates, successfully handling scenarios up to 7.5× the original height. At 10× scaling, the BT fails to guide the cheetah over the obstacle toward the target.

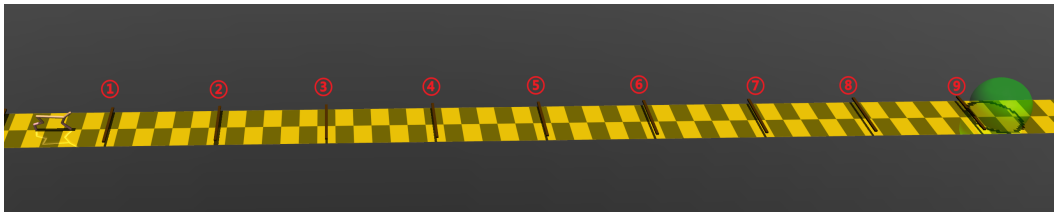
Table 8: Generalization results of generalization in Half Cheetah Hurdle tasks. Results are averaged over 50 random executions.

Tasks	Number			Height			
	Original	6	9	2×	5×	7.5×	10×
Distance	0.008±0.037	0.048±0.126	0.131±0.242	0.026±0.094	0.092±0.203	0.348±0.257	0.702±0.070
Success_Rate	0.960	0.840	0.720	0.900	0.800	0.180	0.000

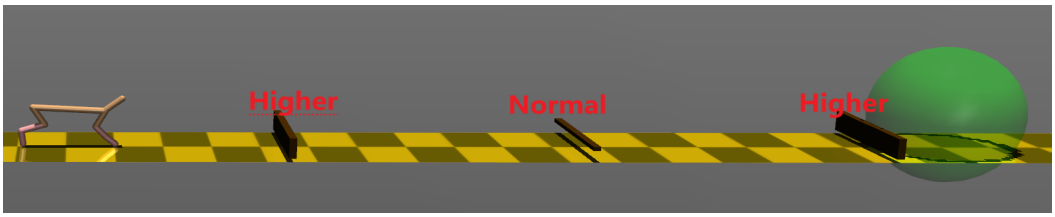
In continuous scenarios, both Ant Maze and Half Cheetah Hurdle, the BTs make conditional judgments and execute actions based on specific environmental data, which are highly dependent



(*a*) Half Cheetah Hurdle with six Hurdles



(*b*) Half Cheetah Hurdle with nine Hurdles



(*c*) Half Cheetah Hurdle with Higher Hurdles (the 1st and the 3rd)

Figure 15: Half Cheetah Hurdle with different hurdles.

on the environment's configuration. However, in discrete scenarios, such as MiniGrid, the behavior tree accepts abstract state inputs, offering better generalization.

Appendix H. Synthesized Behavior Trees

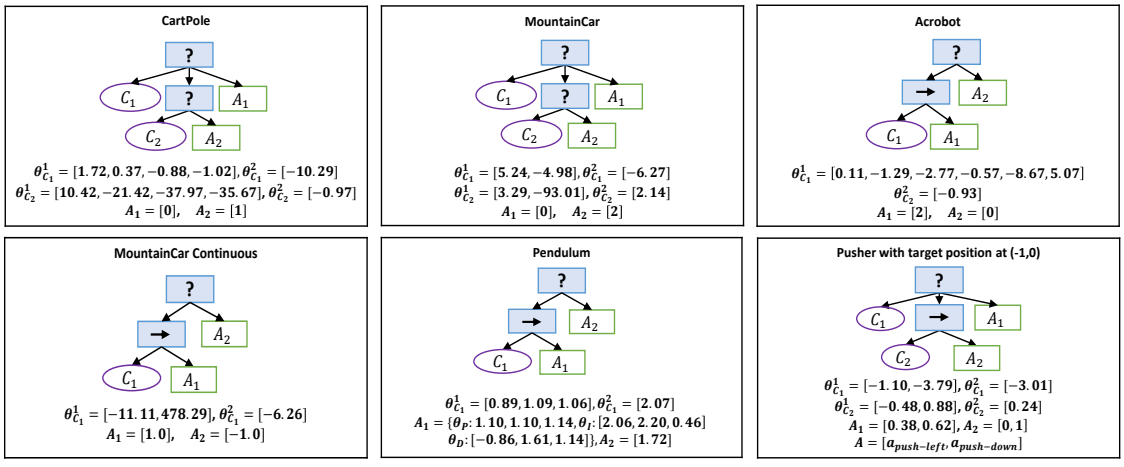


Figure 16: Synthesized Behavior trees. In Pendulum, A_1 is a PID controller that is specified by $\theta_P, \theta_I, \theta_D$; In Pusher, actions are composed of two primitive functions: a_{push_down} and a_{push_left}.

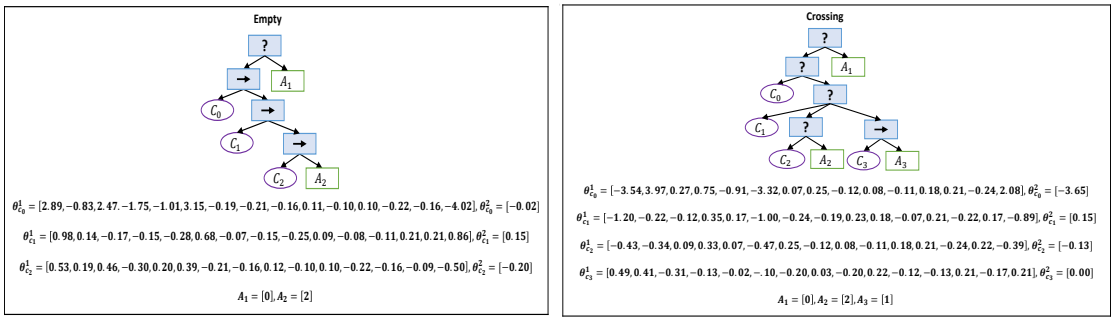


Figure 17: Behavior trees of MiniGrid tasks. The input is modified as an abstract state.

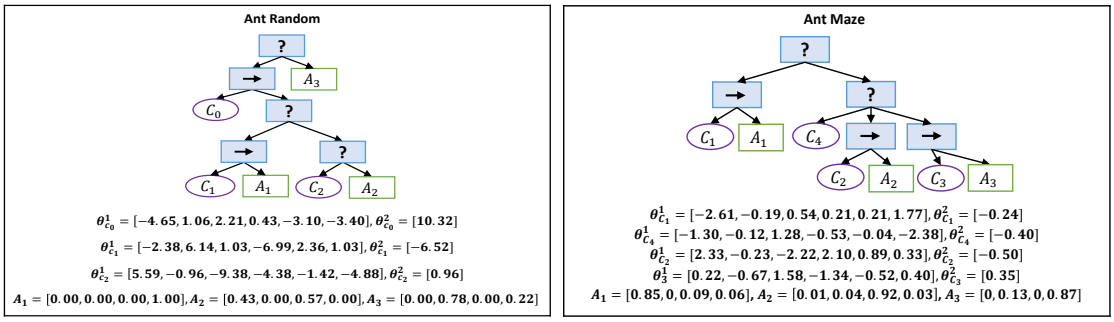


Figure 18: Behavior trees of Mujoco Ant tasks. The action nodes are composed of four primitive functions: a_{up}, a_{down}, a_{left}, and a_{right}.

In this section, we provide synthesized BTs of our proposed method which are complementary to the main experiments in the main paper. The BTs synthesized for the tasks Cart Pole, Mountain Car, Acrobot, Pendulum, and Pusher are shown in Fig. 16. The BTs synthesized for the task in MiniGrid are shown in Fig. 17, and BTs learned for Ant are depicted in Fig. 18. The learned behavior tree of the Half Cheetah Hurdle task is an Action node, which is composed of two primitives $a_{forward}$ and a_{jump} with weight [0.999, 0.001]. Moreover, we optimized the BT architecture based on empirical execution data. For instance, during the Cart Pole task execution, we observed that the action node added by the extraction algorithm remained inactive. Consequently, we removed the newly added root and action nodes from the extraction process.