Behavior Tree Learning for Robotic Task Planning through Monte Carlo DAG Search over a Formal Grammar

Emily Scheide, Graeme Best, and Geoffrey A. Hollinger

***Abstract*— We present an algorithm for learning behavior trees for robotic task and motion planning, which alleviates the need for time-intensive or infeasible manual design of control architectures. Our method involves representing the search space of behavior trees as a formal grammar and searching over this grammar by means of a new generalization of Monte Carlo tree search for directed acyclic graphs, named MCDAGS. Additionally, our method employs simulated annealing to expedite the aggregation of the most functional subtrees. We present simulated experiments for a marine target search and response scenario, and an abstract task selection problem. Our results demonstrate that the learned behavior trees compare favorably with a manually-designed tree, and significantly outperform baseline learning methods. Overall, these results show that our method is a viable technique for the automatic design of behavior trees for robotic task planning.**

I. Introduction

As robots are employed in increasingly complex domains, such as for marine monitoring [2], search and rescue [3], and manipulation [4], they must be capable of adaptively switching between autonomous behaviors. A control architecture specifies how the behavior switching reacts to online observations and the changing state of the environment. The manual design of control architectures is often infeasible due to it being inherently time-intensive and requiring expert knowledge, particularly as robotic tasks become increasingly complex or applications require larger multi-robot teams. This motivates the need to automatically generate control architectures that function well in challenging task domains.

Recently, behavior trees have become a popular control architecture in robotics and computer games [5]. They offer advantages in readability, recursivity, and modularity [2], [6], [7], [8], [9], [10] as compared to finite state machines, decision trees, and various other controlled hybrid systems [7], [11]. These advantages are inherent to the behavior tree design, which is built with respect to tasks rather than states. Put simply, a behavior tree is a directed rooted tree that is comprised of leaf nodes, which evaluate conditions and activate actions, and internal nodes, which describe a logic structure. Behavior tree operation occurs through the switching between a number of tasks, based on changing observed input signals [7]. Due to the advantages such a structured tree brings, it is more feasible to manually design behavior trees than state-based methods.

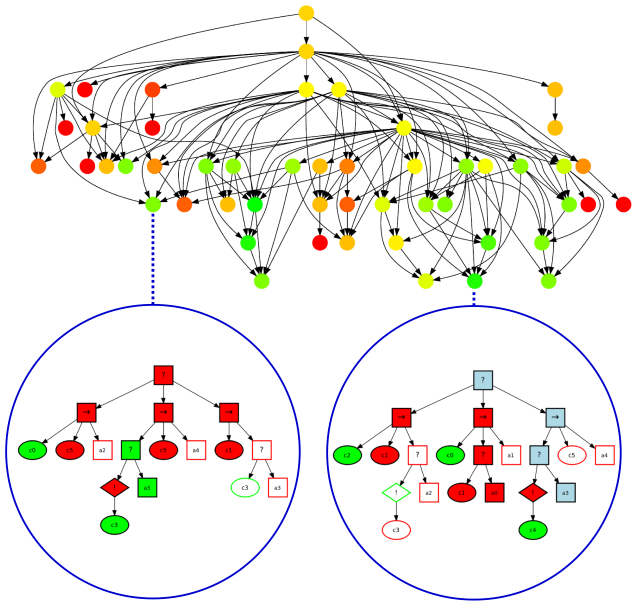

Fig. 1. MCDAGS searches over a formal grammar that describes behavior trees by incrementally constructing a DAG (above). Leaf nodes in the DAG correspond to feasible behavior trees (below). Green DAG nodes represent grammar derivations that lead to the most promising behaviors trees.

Even so, the manual design of a behavior tree can still become too time-intensive or even impossible for sufficiently complex robotic task domains. To meet the increasing demand for autonomous robotics, it is vital that this design be as expedited as possible, without sacrificing functionality. In order to expedite this process given the difficulties of manual design, we require an automatic behavior tree generation method that is robust to task complexity. Given a set of robot capabilities and a task simulator, the goal of the learning method is to find the behavior tree structure that maximizes the expected reward. This is challenging because it is difficult to design a concise and complete representation of the large search space of all behavior trees. Additionally, the learning is required to be with respect to a task simulator, which typically is computationally expensive and noisy.

In this paper, we propose a new learning algorithm for generating behavior trees that maximize task performance. Our algorithm, illustrated in Fig. 1, learns an optimal behavior tree by searching over a formal grammar that represents the set of well-structured behavior trees. The search is carried out by a generalization of Monte Carlo tree search

*Approved for public release; distribution is unlimited. This work was in part sponsored by DARPA under agreement #HR00111820044 and Office of Naval Research Grant N00014-17-1-2581. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

*A preliminary version of this work appeared as a workshop paper [1].

*The authors are with the Collaborative Robotics and Intelligent Systems (CoRIS) Institute, Oregon State University, Corvallis, OR, USA. {scheidee, bestg, geoff.hollinger}@oregonstate.edu

Manuscript 904 submitted to 2021 IEEE International Conference on Robotics and Automation (ICRA). Received November 2, 2020.

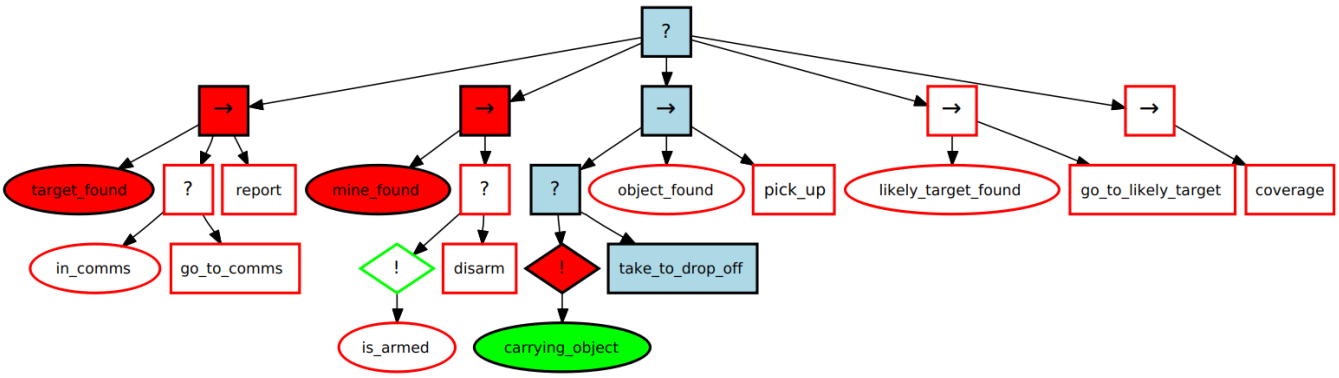

Fig. 2. A manually-designed behavior tree for a marine target search and response application. In the state shown, the robot is carrying an object and taking it to the drop-off location. Leaf squares are *actions*, ovals are *conditions*, ? denotes a *fallback*, → denotes a *sequence*, and ! denotes a *not-decorator*. The flow of execution is from left to right. Red denotes a node that has failed, green denotes a node that has succeeded, and blue denotes that a node is currently executing. Unshaded nodes are currently inactive as they are currently not being evaluated or performed.

(MCTS) [12] that searches over a directed acyclic graph (DAG). This DAG representation enhances the connectivity between related regions of the search space (see Fig. 1), and thus allows for better propagation of information that aids the learning. Our MCTS generalization is inspired by [13], [14], with improvements regarding how information is propagated and utilized. MCTS periodically alternates with rounds of simulated annealing (SA) to expedite the aggregation of functional subtrees into a more desirable behavior tree. Unlike previous behavior tree learning methods, our algorithm is suitable for learning high-level robot behaviors, without requiring introspection within a problem-specific simulator.

We demonstrate the efficacy of our method in two problem domains. In an abstract action selection problem, we highlight the computational benefits of the proposed learning algorithms. In a marine target search and response problem, we show the applicability of our method to robotics scenarios. Our method achieves significantly better task performance compared to methods that have a simplified grammar, use a tree rather than a DAG, or do not use simulated annealing. Overall, we demonstrate that our learning algorithm is capable of generating behavior trees that achieve equal performance to a manually-designed tree (see Fig. 2).

II. Related Work

Behavior trees have been adopted for a wide range of applications [5], ranging from video games [15] to marine robotics [2], [16]. However, the design of behavior trees requires significant human expertise to be effective for the problem at hand. Common methods for learning behavior trees are genetic programming [15], [17], [18], [19], Q-learning [6], [20], and backchaining [21]. While our learning algorithm is inspired by these methods, we found them to be not suitable for robotics applications as the focus is typically on learning low-level actions, rather than autonomous behaviors, often require introspection within a problem-specific simulator, or need significant human input.

A key difference between these learning approaches is how they represent a behavior tree, where different representations warrant different construction methods. One such representation is a formal grammar, which has the advantage of having well-defined rules that incrementally build behavior trees. Grammars have been proposed in [18], [19], which do not enforce any specific structure. In contrast, [22] uses a grammar that introduces structural guidance. The grammar we design in this paper generalizes these grammars to enforce a more functional behavior tree structure.

These grammar representations of behavior trees open up the possibility of employing general grammar-search algorithms developed for other contexts. Grammar search algorithms seek to find the sequence of production rules that derives the word that maximizes reward. In [23], an architectural shape grammar is searched using Q-learning. In [13], they propose a generalization of MCTS that searches over a DAG representation of a grammar. We adapt this method to be robust to challenges in behavior tree learning.

Game tree search can also be described as a DAG search, where transpositions represent states reachable through multiple move sequences. MCTS generalizations for DAGs have been proposed for game trees [14], [24] and adapted for other contexts [25], [26]. In [14], a spectrum of MCTS variants are explored, and counterexamples are presented that show that standard MCTS methods (similar to [13]) fail when the multiple ancestor paths are not accounted for correctly. Inspired by the study in [14], we propose several improvements to the grammar search of [13].

While MCTS methods provide asymptotic convergence guarantees [27], [28], these guarantees do not hold when generalizing to DAGs [14]. Importantly, MCTS techniques for DAGs are prone to being stuck in local optima. To overcome this issue, we complement our MCTS generalization with alternating rounds of simulated annealing.

A. Background: Behavior Trees

An example behavior tree is shown in Fig. 2. Behavior trees [7] are directed rooted trees, which are comprised of control flow, execution, and decorator nodes. The types of control flow nodes are *fallback*, *sequence*, and *parallel*, and they determine which nodes are active. Execution node types are *action* and *condition* nodes. *Action* nodes trigger

execution of robot actions, and take on a value of either *success*, *running*, or *failure*. *Condition* nodes are either *true* or *false* depending on the state of the robot and the world. Decorator nodes have one child *condition* node by definition, and, given the state of that *condition* node as input, return a state. A common example is the *not-decorator* node, which returns the logical complement of the child condition. In this work, we consider all of these behavior tree components except *parallel* nodes, which are less commonly used.

III. Problem Formulation

We address the problem of learning behavior trees as a means of automating the design of a robot's control architecture. A robot's capabilities are described as a set of actions and its sensors are described as a set of conditions. The goal is to learn the best assembly of these conditions and actions in the form of a behavior tree that maximizes task performance. We formalize this problem as follows.

A robot's behavior capabilities are defined as actions $\mathcal{A}$ and sensing capabilities are defined as Boolean conditions $\mathcal{C}$. The action and condition sets are divided into overlapping groups $g \in \mathcal{G}$ by category of application, with $g = [\mathcal{A}_g, \mathcal{C}_g]$ where $\mathcal{A}_g \subseteq \mathcal{A}$ is the actions and $\mathcal{C}_g \subseteq \mathcal{C}$ is the conditions in a group. These groups guide the learning to connect related actions and conditions within subtrees. We introduce a default action $a^\star \in \mathcal{A}$ as an action the robot should execute if all else fails.

These conditions and actions are to be assembled into a behavior tree. Each behavior tree b in the set of all valid behavior trees $\mathcal{B}$ can be evaluated by running it through a black box simulator that returns a reward $f(b)$. This reward function is application-specific and encodes the task performance achieved if behavior tree b is used for decision making; we present an example reward function for a marine target search and response domain later in Sec. V.

Ultimately, we aim to find the behavior tree, $b^* \in \mathcal{B}$, that achieves the maximum reward; i.e.,

$$b^* = \underset{b \in \mathcal{B}}{\operatorname{argmax}}\, f(b). \tag{1}$$

The goal is for the resulting behavior tree to be the most efficient and well-suited for autonomously guiding the robot through the execution of all tasks within its task domain.

IV. Behavior Tree Learning Algorithm

We propose a new algorithm for learning behavior trees. We introduce a formal grammar that encodes the search space of well-structured behavior trees. This grammar formulates a directed acyclic graph, as illustrated in Fig. 1, which we search over with a new generalization of MCTS, named MCDAGS. MCDAGS finds high-performing behavior trees and a set of promising subtrees. Optimized behavior trees are constructed by aggregating these promising subtrees through simulated annealing. This alternating process continues as shown in Alg. 1. We detail the formal grammar, MCDAGS, and SA components of our algorithm as follows, then provide a brief analysis of the computation time and optimality.

Algorithm 1 Behavior tree learning with MCDAGS and SA

Input: formal grammar F consisting of actions $\mathcal{A}_g$ and conditions $\mathcal{C}_g$ in groups $g \in \mathcal{G}$
Output: behavior tree b^*

$b^* \leftarrow (), f^* \leftarrow -\infty$ ▷ Best tree and reward
$shortcuts \leftarrow \{\}$ ▷ Promising subtrees
for $i = 1$ to num_rounds **do**
  if i is odd or $i < 10$ **then**
    $b, f, new\_shortcuts \leftarrow \text{MCDAGS}(F)$
  else
    $b, f \leftarrow \text{SA}(shortcuts, b^*)$
  $shortcuts \leftarrow shortcuts \cup new\_shortcuts$
  $F \leftarrow \text{UPDATEGRAMMAR}(F, shortcuts)$
  if $f > f^*$ **then** $b^* \leftarrow b,\ f^* \leftarrow s$
return b^*

A. Behavior Tree Formal Grammar

We design a set of production rules, which together form a formal grammar. This set of rules guides the autonomous production of well-structured behavior trees. This set of rules is universal, and is applicable to any robot or scenario, as it takes in any set of groups of actions, $\mathcal{A}_g$, and conditions, $\mathcal{C}_g$, as input.

The structure the grammar enforces has a *fallback* node at the root, and then a layer of *sequence* nodes. Next, it allows *fallback*, *action*, *condition*, or *not-decorator* nodes with the constraint that *conditions* must appear to the left of *actions*. If a *fallback* was chosen at the previous level, the deepest level is allowed to contain *action*, *condition*, or *not-decorator* nodes only. We also enforce that each subtree only contains *actions* and *conditions* that are in the same group. We also enforce that the right-most subtree is the default action $a^\star$. Figs. 1-3 present example behavior trees that satisfy this structure.

This grammar is defined as $F = [T, N, P, S]$. Terminal characters are $T = \{?, \rightarrow, !, [a], (c),), (, z\}$, where ? denotes *fallback*, $\rightarrow$ denotes *sequence*, ! denotes *not-decorator*, $[a]$ denotes an *action* $a \in \mathcal{A}$, (c) denotes a *condition* $c \in \mathcal{C}$, (denotes going down a level, and) denotes going up a level. Character $z \in \mathcal{Z}$ denotes a shortcut subtree learned by the MCDAGS rounds. Non-terminal characters are $N = \{S, s, s^+, A^\star, s_g, f_g, A_g, C_g, \check{C}_g, r_g^1, l_g^1, r_g^2, l_g^2\}, \forall g \in \mathcal{G}$, where S is the start character.

The production rules, P, are defined in Table I. Each behavior tree is constructed by applying a sequence of production rules. Derivation of the behavior tree starts at the start character S, which then expands to a *fallback* node ? at the root with subtrees below it on level 1, each denoted by the non-terminal character s (see Table I Part 1). An additional subtree containing the $[a^\star]$ action is added to the right side.

Each subtree s is assigned a group $g \in \mathcal{G}$, and is expanded to a sequence node $\rightarrow$ at the root with child nodes below it on level 2, specific to group g. The children allowed at this level are non-terminal characters A_g, $\check{C}_g$, r_g^1, l_g^1, and f_g (Part 2.1). A_g converts into any *action* node $[a]$ in group

TABLE I
BEHAVIOR TREE FORMAL GRAMMAR PRODUCTION RULES.
Terminal characters are shown in red and non-terminals in blue.

1. Setup a *fallback* node with *sequence* subtrees:
$S \to ?\ (\ s\ s^+\ A^\star\ )$ $\quad s^+ \to s\ s^+$ $\quad s^+ \to s$
$A^\star \to \to\ (\ [a^\star]\ )$

2. Subtree structure, $\forall g \in \mathcal{G}$:
2.1 Level 1 to Level 2:
$s \to s_g$ $\quad s_g \to \to\ (\ A_g\ r_g^1\ )$ $\quad s_g \to \to\ (\ f_g\ r_g^1\ )$
$s_g \to \to\ (\ l_g^1\ A_g\ )$ $\quad s_g \to \to\ (\ l_g^1\ f_g\ )$
2.2 Level 2:
$r_g^1 \to f_g\ r_g^1$ $\quad r_g^1 \to f_g$ $\quad r_g^1 \to A_g\ r_g^1$
$r_g^1 \to A_g$ $\quad l_g^1 \to l_g^1\ f_g$ $\quad l_g^1 \to f_g$
$l_g^1 \to l_g^1\ \check{C}_g$ $\quad l_g^1 \to \check{C}_g$
2.3 Level 2 to Level 3:
$f_g \to ?\ (\ A_g\ r_g^2\ )$ $\quad f_g \to ?\ (\ l_g^2\ A_g\ )$
2.4 Level 3:
$r_g^2 \to A_g\ r_g^2$ $\quad r_g^2 \to A_g$
$l_g^2 \to l_g^2\ \check{C}_g$ $\quad l_g^2 \to \check{C}_g$
2.5 Conditions and not-decorators:
$\check{C}_g \to C_g$ $\quad \check{C}_g \to !\ (\ C_g\ )$

3. Insert actions $\forall a \in \mathcal{A}_g$ and conditions $\forall c \in \mathcal{C}_g$:
$A_g \to [a]$ $\quad C_g \to (c)$

4. Add shortcut subtrees $\forall z \in \mathcal{Z}$:
$s \to z$

g (Part 3). $\check{C}_g$ converts into any *condition* node (c) or not-decorator node with any child *condition* $!\ (c)$ in group g (Parts 2.5 and 3). We also enforce that a particular *action* or *condition* can only appear once in a subtree.

Character r_g^1 either adds just one A_g or f_g character to the right of the original location of r_g^1, or adds one A_g or f_g and another instance of r_g^1 to allow for the addition of multiple *action* and *fallback* nodes (Part 2.2). l_g^1 does the same except to the left of the original location of l_g^1, and it adds $\check{C}_g$ characters (which lead to *conditions*) instead of A_g (*actions*) (Part 2.2). This definition ensures that *conditions* appear before sibling *actions*.

Character f_g sets up an optional level 3 of the tree by converting to a *fallback* node ? at level 2 with another level of children below it (Part 2.3). The children allowed below on level 3 are A_g, $\check{C}_g$, r_g^2, and l_g^2. The r_g^2 and l_g^2 characters are equivalent to r_g^1 and l_g^1 except they do not allow the addition of *fallback* nodes and subsequent children (Part 2.4).

Information is passed to subsequent rounds by adding the most promising learned subtrees to the grammar. This allows the expedited rediscovery of these subtrees in subsequent rounds. The grammar is updated to include production rules that represent each of these shortcut subtrees z (Part 4).

B. Monte Carlo DAG Search over a Formal Grammar

Our learning algorithm seeks to find the best behavior tree b^* by searching through the grammar F. The grammar encodes the search space as a DAG. The root represents word S, edges represent production rules, internal nodes represent partial derivations, and leaf nodes represent behavior trees.

MCDAGS incrementally constructs and searches over a single-rooted DAG $\mathcal{D}$. Like MCTS [12], MCDAGS cycles between four phases: *selection*, *expansion*, *simulation*, and *backpropagation*. The key differences to MCTS are in the

Algorithm 2 Search for best behavior tree over grammar F with Monte Carlo DAG search

1: **function** MCDAGS(F)
2: $\quad \mathcal{D} \leftarrow \{\}$ $\quad \triangleright$ DAG
3: $\quad b^* \leftarrow (), f^* \leftarrow -\infty$ $\quad \triangleright$ Best tree and reward
4: $\quad$ **for** fixed number of samples **do**
5: $\quad\quad n \leftarrow$ SELECTNODE($\mathcal{D}$)
6: $\quad\quad n^+ \leftarrow$ EXPANDORCONNECT(n, F)
7: $\quad\quad f \leftarrow$ ROLLOUT(n^+) $\quad \triangleright$ Reward $f(b)$
8: $\quad\quad$ **if** $f > 0$ **then** ADDBACKWARDSEDGES($\mathcal{D}, n^+$)
9: $\quad\quad$ BACKPROPAGATE($\mathcal{D}, n^+, f$)
10: $\quad\quad$ **if** $f > f^*$ **then** $b^* \leftarrow$ BT(n^+), $f^* \leftarrow f$
11: $\quad shortcuts \leftarrow$ GETSHORTCUTS($\mathcal{D}$)
12: $\quad$ **return** $b^*, f^*, shortcuts$

expansion and *backpropagation* phases, as well as how intermediate solutions are constructed to pass to subsequent rounds. Pseudocode is provided in Alg. 2.

In the *selection* phase (line 5), a leaf node n of $\mathcal{D}$ is selected. We use the standard UCT selection procedure [27], which recursively follows nodes from the root that maximize an upper confidence bound computed from learned statistics.

In the *expansion* phase (line 6), a production rule is applied to the word represented by n. If the resulting word already appears in $\mathcal{D}$, then an edge is added from n to this node. If the resulting word is not in $\mathcal{D}$, then a new node n^+ is added with an edge from n.

The *rollout* phase (line 7) applies random production rules to the word represented by n^+ until a terminal word, representing behavior tree b, is reached. The reward function $f(b)$ is evaluated. Additionally, we create a reconstructed tree b' that only contains nodes of b that were activated while evaluating f. b' has identical functionality to b, but ensures the most relevant information is retained for the next round.

If the behavior tree is promising then we add edges to n^+ from valid ancestors (line 8), which are found via backward induction of the production rules up to a fixed depth. This ensures that promising solutions are well connected in $\mathcal{D}$.

In the *backpropagation* phase (line 9), the statistics at ancestor nodes of n^+ are updated. Specifically, the reward $f(b)$ is incorporated into the averages of all ancestors. The count of visits is incremented for ancestors only on the selection path. This technique and alternatives are compared in [14]; we found this technique achieved the best results.

The most promising subtrees are passed to subsequent rounds by incorporating them into the grammar (Part 4). These subtrees are selected by extracting the nodes at each level with the highest reward. The trees b' are separated into level 1 subtrees and merged into the retained set.

C. Simulated Annealing for Subtree Selection and Ordering

Although MCDAGS learns a variety of promising subtrees, it is not always successful at combining these subtrees into a single behavior tree. To aid this process, we alternate with rounds of simulated annealing, which aims to select and order promising subtrees into an effective behavior tree.

The SA algorithm is initialized with the current overall best-performing behavior tree. At each iteration, neighboring solutions are constructed from the current solution by swapping two subtrees, removing a single subtree, or inserting a new subtree, where the possible subtrees are retained from previous learning rounds. A neighbor solution is selected at random, evaluated, and compared to the current solution. If the neighbor is better, then it replaces the current solution. If it is worse, then it replaces it with some probability that decreases as the learning progresses. SA returns to the previous best solution if no progress is being made.

D. Analysis

Our algorithm is an anytime algorithm such that the solution is incrementally improved. In most practical domains, the computation time will be dominated by calls to the simulator $f(b)$, which occur once per iteration. MCTS provides guarantees for asymptotic convergence rates to the optimal sequence when searching over trees [27], [28]. Unfortunately, these guarantees do not carry over to searching over DAGs, as demonstrated with counterexamples in [14]. However, our results empirically demonstrate performance improvements due to the DAG structure enabling better information propagation. Simulated annealing provides a theoretical guarantee for the probability of finding the optimal solution approaching one [29], although in practice restarts are commonly used to avoid local optima [30], as in our algorithm. While this analysis does not yield strong guarantees of optimality, this is unlikely to be achievable by any practical algorithm due to the complex structure of the search space; hence, randomized heuristic algorithms are imperative for this learning problem.

V. Simulation Experiments

We evaluate the performance of our algorithm at learning high-performing behavior trees in two example domains: (1) an abstract task assignment problem, and (2) a marine robotic target search and response scenario. We compare to the manually-designed tree and a variety of comparison methods to evaluate the components of our algorithm.

A. Comparison Methods

To evaluate the overall performance and specific functionality of various components of our algorithm, we compare to six restricted versions of our algorithm, as well the manually-designed tree. Due to the limited availability of comparable method implementations for direct comparison, we focus on evaluating restricted versions of our proposed method to motivate and study the individual components:

- *Manual Design*: A behavior tree we designed by studying the internals of the problem-specific simulator, and ensuring that all possible ways of reward collection are encompassed
- *MCDAGS+SA*: Our method as described in Sec. IV
- *No Default*: Our method without providing a default action $a^\star$
- *MCTS+SA*: Our method except with a tree search structure instead of a DAG
- *MCDAGS*: Our method except without the SA rounds
- *No Groups*: Our method without the input actions and conditions sorted into groups
- *No Restarts*: MCDAGS without periodic restarts
- *No Structure*: *No Groups* with a simplified grammar that does not enforce the structure described in Sec. IV-A:

$S \to ?\ (\ l\ r\ )$	$S \to \to\ (\ l\ r\ )$	
$r \to l\ r$	$r \to l$	$l \to S$
$l \to [a]\ \ \forall a \in \mathcal{A}$	$l \to (c)\ \ \forall c \in \mathcal{C}$	$S \to z\ \ \forall z \in \mathcal{Z}$

Each trial was run for 50 rounds with an Intel Xeon 3.7GHz CPU. This took on the order of 1-10 hours, although our implementation has not been optimized for efficiency.

B. Abstract Task: Multiplication of Conditions

We begin by presenting results for an abstract task that requires switching between several actions based on which conditions are currently active. These experiments served as a useful domain for refining and validating our method ready for the following marine robotics search scenario.

1) Experimental setup: For these experiments, we define an abstract set of conditions $\mathcal{C} = \{0, 1, 2, 3\}$ and actions $\mathcal{A} = \{0, 1, ..., 6\}$. At each timestep, the correct action to be chosen is the multiplication of the conditions that are currently true. If no conditions are true, then the correct action is 1. The simulator iterates through the power set of the conditions, and the total reward is defined as the sum of correct actions. The simulator is a black box, in the sense that the behavior tree learning algorithms do not know which specific action-condition combination contributed to the reward. No groups $\mathcal{G}$ or default action $a^\star$ are used here. While this abstract problem is relatively small, it requires learning complex combinations of subtrees, and therefore serves as a useful domain for evaluating the algorithms.

2) Results: The results are presented in Fig. 4. Firstly, the methods with MCDAGS significantly outperforms the MCTS method. This shows that the DAG better captures the structure of the search space than a tree, and this structure is successfully exploited by our method. Interestingly, MCDAGS without SA performed better than with SA, we believe due to the MCDAGS rounds being much more useful than SA; this is not the case in the following domain. Searching over our grammar clearly outperforms the less-structured grammar, which shows the benefit of restricting the search space to more meaningful behavior trees.

C. Marine Robotic Target Search and Response

1) Experimental setup: We next present a marine target search and response scenario. A robot moves through a marine environment locating targets, disarming sea mines, and retrieving objects of interest. The goal is to maximize the completion of these tasks. Each task has a reward, and the total reward is the sum of these task rewards. The robot's conditions and actions are divided into groups: g_1 = [{go to comms, report},{in comms, target found, at surface}], g_2 = [{disarm},{mine found, is armed}], g_3 = [{pick up, take to drop off},{object found, carrying object}], g_4 = [{go to likely target},{likely target found}], g_5 = [{random walk},{}], g_6

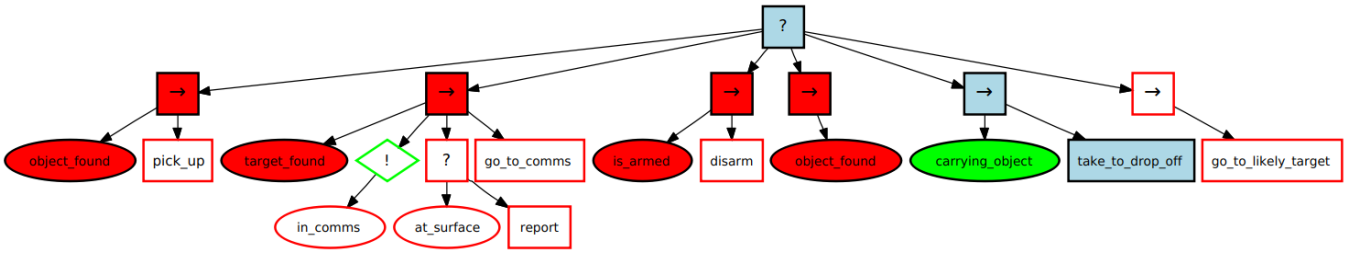

Fig. 3. A learned behavior tree for the *marine robotic target search and response* domain. Five subtrees were successfully learned. Although the structure differs from the manually-designed tree, the functionality is near-equivalent and achieves similar task performance.

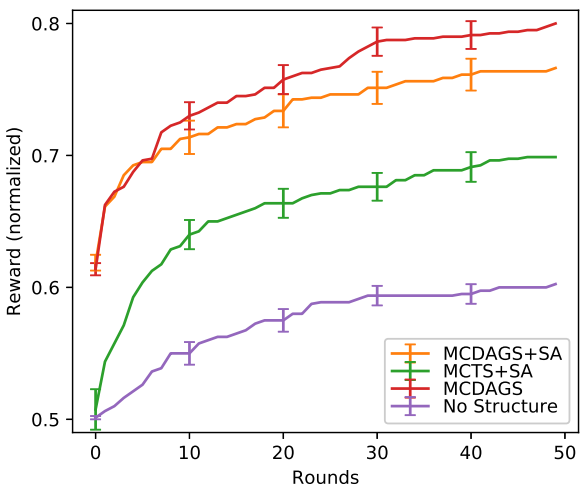

Fig. 4. Convergence of various learning algorithms for the *multiplication of conditions* task selection problem. The optimal solution in this domain has a reward of 1.0. Shown are means and standard errors over 50 trials.

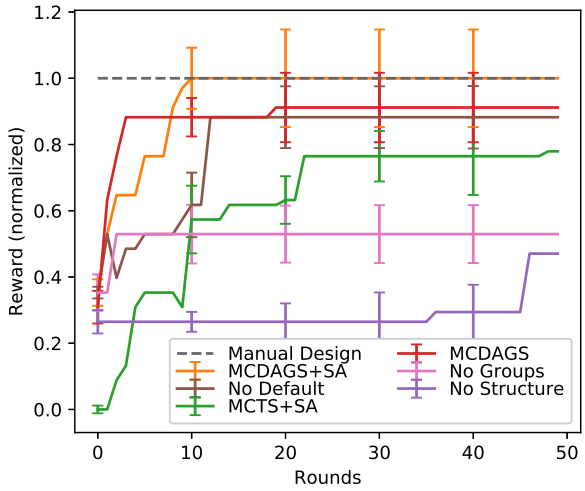

Fig. 5. Convergence of various learning algorithms for the *marine target search and response* scenario. Rewards are normalized with respect to *Manual Design*. Shown are medians and standard errors over 10 trials.

= [{shortest path},{}], and g_7 = [{coverage},{}]. We define the default action $a^\star$ to be the coverage planner to ensure the robot inherently moves. The robot moves on a roadmap with a random distribution of targets, mines, and objects of interest. The robot observes nearby objects with small false-positive and false-negative rates.

2) Results: In Fig. 3, we show a representative example behavior tree learned by our method. In comparison with the manually-designed behavior tree in Fig. 2, our method achieves similar functionality and performance, despite the variation in the structure. We observe that subtrees for g_1, g_2, g_3, and g_4 were learned such that all actions and crucial conditions are present. We note the learning chose to replace the default coverage planner with a different action in this instance, but this was not always the case. For 100 trials on different worlds, the learned tree achieves 2% better performance on average. This demonstrates that our algorithm is capable of learning a robust, high-performing behavior tree that is comparable to a manually-designed tree.

We compare the performance of various methods in Fig 5. *MCDAGS+SA* outperforms all of the comparison learning methods and reaches a similar performance to the manually-designed tree (reward 1.0). Our full method significantly outperforms the cases where MCDAGS is replaced with MCTS, which shows the benefits of searching over a DAG rather than a tree. Additionally, we observe a small improvement when alternating between SA and MCDAGS. When there is no enforced structure, the learning was relatively poor, which shows the benefit of our structured grammar. Similarly, our method without restarts also did poorly (not shown in plot), showing the need to divide the learning into rounds. The learning performed significantly better when the actions and conditions are grouped together. Marginal improvement was achieved when $a^\star$ is provided, showing that having this information speeds up the learning but good solutions are still found without this information.

VI. Future Work

We presented an algorithm that generates behavior trees with comparable performance to a manually-designed behavior tree, and showed the benefits of using a grammar that enforces functional structure, a DAG instead of a tree, and simulated annealing for expediting the aggregation of learned subtrees. In the future, it would be interesting to investigate online refinements of learned behavior trees. A related challenge would be to learn to formulate specific actions in conjunction with their optimal placement within a behavior tree. Additionally, it would be pertinent to demonstrate the efficacy of this method for more complex domains, such as those involving heterogeneous multi-robot teams [31].

VII. Acknowledgment

We would like to thank John Keller for the behavior tree implementation.

REFERENCES

[1] E. Scheide, G. Best, and G. A. Hollinger, "Learning behavior trees for robotic task planning by Monte Carlo search over a formal grammar," in *Proc. RSS Workshop on Learning (in) Task and Motion Planning*, 2020.

[2] P. Ögren, "Increasing modularity of UAV control systems using computer game behavior trees," in *Proc. AIAA Guidance, Navigation, and Control Conf.*, 2012, p. 4458.

[3] Y. Tian, K. Liu, K. Ok, L. Tran, D. Allen, N. Roy, and J. P. How, "Search and rescue under the forest canopy using multiple UAS," in *Proc. Int. Symp. on Experimental Robotics*, 2018, pp. 140–152.

[4] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard *et al.*, "An integrated system for autonomous robotics manipulation," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2012, pp. 2955–2962.

[5] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and AI," *arXiv preprint arXiv:2005.05842*, 2020.

[6] B. Banerjee, "Autonomous acquisition of behavior trees for robot control," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2018, pp. 3460–3467.

[7] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[8] A. Klöckner, "Interfacing behavior trees with the world using description logic," in *Proc. AIAA Guidance, Navigation, and Control (GNC) Conf.*, 2013, p. 4636.

[9] C. I. Sprague, Ö. Özkahraman, A. Munafo, R. Marlow, A. Phillips, and P. Ögren, "Improving the modularity of AUV control systems using behaviour trees," in *Proc. IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, 2018.

[10] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "CoSTAR: Instructing collaborative robots with behavior trees and vision," in *Proc. Int. Conf. on Robotics and Automation (ICRA)*, 2017, pp. 564–571.

[11] P. I. Cowling, M. Buro, M. Bida, A. Botea, B. Bouzy, M. V. Butz, P. Hingston, H. Muñoz-Avila, D. Nau, and M. Sipper, "Search in Real-Time Video Games," in *Artificial and Computational Intelligence in Games*. Schloss Dagstuhl, 2013, vol. 6.

[12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[13] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-based optimization on graphs with application to library performance tuning," in *Proc. Int. Conf. on Machine Learning (ICML)*, 2009, p. 729–736.

[14] A. Saffidine, T. Cazenave, and J. Méhat, "UCD: Upper confidence bound for rooted directed acyclic graphs," *Knowledge-Based Systems*, vol. 34, pp. 26–33, 2012.

[15] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *IEEE Trans. on Games*, vol. 11, no. 2, pp. 183–189, 2018.

[16] S. Bhat, I. Torroba, Ö. Özkahraman, N. Bore, C. Sprague, Y. Xie, I. Stenius, J. Severholt, C. Ljung, J. Folkesson *et al.*, "A cyber-physical system for hydrobatic AUVs: System integration and field demonstration," in *Proc. IEEE OES Autonomous Underwater Vehicles Symp.*, 2020.

[17] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game DEFCON," in *Proc. European Conf. on the Applications of Evolutionary Computation*, 2010, pp. 100–110.

[18] A. Neupane and M. Goodrich, "Learning swarm behaviors using grammatical evolution and behavior trees," in *Proc. Int. Joint Conference on Artificial Intelligence (IJCAI)*, 2019, pp. 513–520.

[19] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving behaviour trees for the Mario AI competition using grammatical evolution," in *Proc. European Conf. on the Applications of Evolutionary Computation*, 2011, pp. 123–132.

[20] R. Dey and C. Child, "QL-BT: Enhancing behaviour tree design and implementation with Q-learning," in *Proc. IEEE Conf. on Computational Intelligence in Games (CIG)*, 2013.

[21] M. Colledanchise, D. Almeida, and P. Ögren, "Towards blended reactive planning and acting using behavior trees," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2019, pp. 8839–8845.

[22] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary behavior tree approaches for navigating platform games," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 9, no. 3, pp. 227–238, 2016.

[23] M. Ruiz-Montiel, J. Boned, J. Gavilanes, E. Jiménez, L. Mandow, and J.-L. Pérez-de-la-Cruz, "Design with shape grammars and reinforcement learning," *Advanced Engineering Informatics*, vol. 27, no. 2, pp. 230–245, 2013.

[24] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in Monte Carlo tree search," in *Proc. IEEE Symp. on Computational Intelligence and Games*, 2008, pp. 389–395.

[25] D. Bauer, T. Patten, and M. Vincze, "Monte Carlo tree search on directed acyclic graphs for object pose verification," in *Proc. Int. Conf. on Computer Vision Systems*, 2019, pp. 386–396.

[26] A. Pélissier, A. Nakamura, and K. Tabata, "Feature selection as Monte-Carlo search in growing single rooted directed acyclic graph by best leaf identification," in *Proc. SIAM Int. Conf. on Data Mining*, 2019, pp. 450–458.

[27] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proc. European Conf. on Machine Learning*, 2006, pp. 282–293.

[28] G. Best, O. Cliff, T. Patten, R. R. Mettu, and R. Fitch, "Dec-MCTS: Decentralized planning for multi-robot active perception," *Int. J. Robotics Research*, vol. 38, no. 2-3, pp. 316–337, 2019.

[29] V. Granville, M. Krivanek, and J. Rasson, "Simulated annealing: A proof of convergence," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 16, no. 6, pp. 652–656, 1994.

[30] F. Mendivil, R. Shonkwiler, and M. Spruill, "Restarting search algorithms with applications to simulated annealing," *Advances in Applied Probability*, vol. 33, no. 1, pp. 242–259, 2001.

[31] M. Colledanchise, A. Marzinotto, D. V. Dimarogonas, and P. Ögren, "The advantages of using behavior trees in mult-robot systems," in *Proc. Int. Symp. on Robotics (ISR)*, 2016.