RE:BT-Espresso: Improving Interpretability and Expressivity of Behavior Trees Learned from Robot Demonstrations

Adam Wathieu*[1], Thomas R. Groechel*[2], Haemin Jenny Lee[2],
Chloe Kuo[2], and Maja J. Matarić[2]

***Abstract*— Behavior trees (BTs) are hierarchical agent control architectures popular for robot task-level planning that can be autonomously learned from robot demonstrations via decision tree (DT) intermediaries, making them accessible to non-expert users. Conversion algorithms from DTs to BTs, such as the BT-Espresso algorithm, focus on replicating DT logic in a BT format but do not exploit the strengths of the BT architecture. We introduce the Representation Exploitation of BT-Espresso (RE:BT-Espresso) algorithm, which builds on BT-Espresso and improves the learned BT's *interpretability* and *expressivity*. RE:BT-Espresso improves interpretability by removing logical redundancies in the generated BTs and improves expressivity by exploiting desired BT structures, such as adding Inverter nodes, Repeater sequences, and Parallel Selector Action nodes that gives the user a choice of actions for state spaces that did not resolve to a concise action in the DT. The RE:BT-Espresso algorithm was evaluated against BT-Espresso using demonstration data synthesized by BTs. When compared to the synthesized BTs using graph edit distance (GED), RE:BT-Espresso outscored BT-Espresso on 54 subtrees, tied on 178, and lost on 2. Further, the proposed reduction strategies reduced the number of nodes in a generated tree by a median of 7.82%. The results validate improved interpretability and expressivity of learned RE:BT-Espresso task-level BT policies from robot demonstration.**

I. INTRODUCTION

Programming high-level robot tasks involving many primitive actions typically requires a scalable control architecture and robot programming expertise. Task-level control architectures sequence a robot's primitive actions toward a high-level goal-directed task. Various architectures, such as Finite State Machines (FSMs), decision trees (DTs), and neural networks (NNs), have been used to model task-level behavior by representing the robot states and their respective transitions. However, they have various drawbacks with regard to modularity, reactivity, and interpretibility [1]. FSMs, for example, become unwieldy with increased robot behavior complexity [2], DTs are not designed to be reactive, and NNs are difficult to readily interpret [3].

Recently, behavior trees (BTs) have become a popular alternative for task-level control due to their interpretability, modularity, and reactivity [4]. Creating a handcrafted BT policy is challenging and often produces behaviors that do not adjust to the robot's environment or user needs. Prior work has proposed learning BTs from demonstration, which allows the robot to learn a task by modeling the data collected from a demonstration [5], [6].

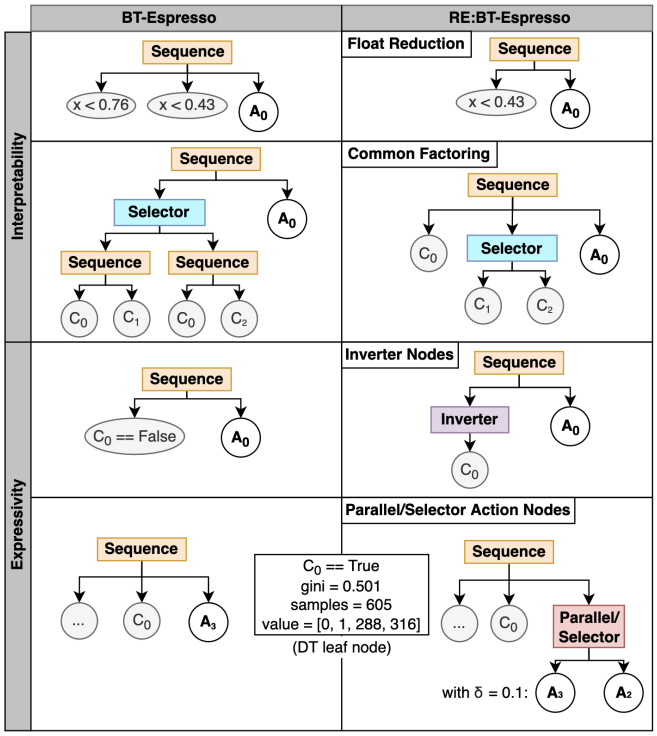

Fig. 1: Left: Behavior tree structures generated from BT-Espresso algorithm. Right: Improvements of the same structures with respect to interpretability and expressivity from the RE:BT-Espresso algorithm, as described in Sec. III. Last action taken diagrams (Sec. III-B) are shown in Figs. 2 and 3, further improving expressivity.

The primary way to learn a BT from demonstration is to first train a decision tree (DT), and then convert the DT to a BT [5], [6]. However, current DT-to-BT conversion algorithms, such as the naive algorithm [5] and BT-Espresso algorithm [6], focus on replicating the DT logic in a BT format, but do not fully leverage the interpretable and expressive power of the hierarchial BT architecture. As a result, these algorithms generate formulaic BTs equivalent to evaluating the original DT at every timestep.

We introduce the Representation Exploitation of BT-Espresso (RE:BT-Espresso) algorithm, which builds on BT-Espresso and improves the learned BT's *interpretability* and

*Adam Wathieu and Thomas Groechel are co-first authors.

This work was supported by grant NSF IIS-1925083 and the USC Robotics and Autonomous Systems REU summer program

[1] Author is with the Department of Computer Science, Northwestern University, Evanston, IL 60208, USA adamwathieu2022@u.northwestern.edu

[2] Authors are all with the Interaction Lab, Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA {groechel,haeminle,cmkuo,mataric}@usc.edu

expressivity. Murdoc et al. [7] define Post Hoc Interpretation as the analysis of a trained model to provide insights into the learned relationships between labels and features. To improve the *interpretability* of generated BTs, the RE:BT-Espresso algorithm removes redundant nodes that complicate the relationships between labels and features without contributing logically to the control policy. The *expressivity* of a BT is defined as the breadth of behaviors that can theoretically be constructed [8]. The RE:BT-Espresso algorithm implements features that expand the breadth of behaviors that can be generated as compared to the BT-Espresso algorithm.

To evaluate RE:BT-Espresso, simulated robot BT demonstration data were executed on both BT-Espresso and RE:BT-Espresso. The algorithms were evaluated using graph edit distance (GED) for similarity to the original simulated BT. From 234 simulated subtrees, RE:BT-Espresso outscored BT-Espresso on 54 subtrees, tied on 178, and lost on 2. Furthermore, the proposed reduction strategies reduced the number of nodes in a generated tree by a median of 7.82%. The results demonstrate the ability of RE:BT-Espresso to generate BTs with enhanced interpetability and expressivity.

II. BACKGROUND AND RELATED WORK

A. Behavior Trees

BTs were popularized by the video game industry to create AI for non-player characters (NPCs) [9]. Widely used in popular video games like *Halo* and *Spore*, BTs served as an intermediary data structure between the game designers and programmers [10]. They were intuitive enough for the designers to use to construct the behavior of the NPCs and technical enough for the programmers to execute in their game engine. In robotics, the notion of BTs relates to behavior-based control [11] and has recently been more widely adopted over other high-level robot control methods, such as finite state machines and neural networks [4].

In high-level task planning in robotics, a good control policy should be interpretable, modular, and reactive [1]. An interpretable policy allows users to understand how and when the behaviors of the robot are triggered. A modular control policy is one whose design allows the user to modify and reuse parts of the policy in real time. A reactive control policy should execute in real time, and be responsive to sudden changes to its world states.

BTs have all these desired qualities, and have therefore emerged as a popular alternative to other control architectures for high-level task control in robotics [6], [12]. BTs are interpretable, as the user can trace from the root down to the execution nodes and see in real time what action the robot is taking and why. Additionally, since every node has the same interface, subtrees can be moved and manipulated, allowing BTs to be modular. Lastly, BTs are reactive. By rerunning at every tick, they react to changes in the state space or action space in real time.

Iovino et al. [4] and Colledanchise et al. [12] provide detailed reviews of the function and use of BTs in robotics.

B. Learning Behavior Trees

Constructing BTs for robot control policies is an active area of study. Manually designed BTs have been commonly used, but result in rigid, inflexible robot behavior.

Reinforcement Learning Theory and Genetic Programming have both been used to generate BTs. Banerjee et al. [13] used an action-quality function to choose the best action given a certain state, and converted this function to a BT via a *decanonicalization* algorithm. Li et al. [14] merged reinforcement learning nodes with BT nodes to create a mixed MDRL-BT. Zhang et al. [15] used genetic programming, evolving a BT with hybrid constraints and using an evaluation function to select higher fitness nodes to reproduce. These methods required the use of reward functions that are task-specific and hence difficult to generalize. Additionally, they require many iterations of the simulation in order to create an accurate BT. Alternatively, learning from demonstration (LfD) allows for learning an arbitrary task by creating the BT from demonstration data [4] [16].

C. Decision Tree Intermediaries for LfD Behavior Trees

The primary way to use LfD to generate BTs is to first train a DT classifier on robot demonstration data using the actions as labels and world state space as features. The DT is then converted to a BT, since BTs have been shown to be a generalization of DTs, and can mimic the logic of DTs [17], [18]. Multiple machine learning algorithms are effective for this purpose, such as ID3, C4.5, and CART [18].

To convert the DT to a BT, Sagredo-Olivenza et al. [5] used the naive DT-to-BT algorithm, shown in [6], with some manual optimizations. The naive DT-to-BT algorithm leverages the fact that the Sequence control node is analogous to the logical AND, and that the Selector control node is analogous to the logical OR. French et al. [6] built on the naive algorithm with the BT-Espresso algorithm, which uses the UC Berkeley Espresso logic minimization algorithm to heuristically reduce the DT logical statements, before converting it to a BT. The BT-Espresso algorithm generates BTs that are consistent with the original DT. However, the generated BTs leave room for further logic minimization and improved interpretability. Additionally, no techniques are used to utilize and exploit unique BT structures and nodes, and thus the algorithm fails to leverage the expressive nature of BTs. This work addresses both of these limitations using the algorithms and techniques described below.

III. TECHNICAL APPROACH

A. Definitions and Assumptions

The following definitions and assumptions serve as motivation for our improvements of the BT-Espresso algorithm:

1. Murdoc et al. [7] define Post Hoc Interpretation as the analysis of a trained model to provide insights into the learned relationships between labels and features. Since logical redundancies in a BT model make the relationships between labels and features superfluous, we assume that removing logical redundancies from an existing BT model makes the model more interpretable.

2. BT expressiveness is defined as the breadth of behaviors that can theoretically be constructed [8]. This work adds more unique control nodes and BT structures thus increasing BT expressivity when compared to the original BT-Espresso algorithm [6].

B. RE:BT-Espresso algorithm

We improved the RE:BT-Espresso algorithm, which converts the DT to a BT, to make the BTs both more *interpretable* and more *expressive* (Alg. 1). Note the root node can be switched out for a Selector or Sequence node depending on the task and actions. A Parallel node was chosen to stay consistent with the original BT-Espresso algorithm.

Increasing Interpretability

To create BTs that are more interpretable, the RE:BT-Espresso algorithm generates less redundancy in the BTs while maintaining the same logic. This is achieved by using the following two methods.

Float Representation Reduction: The Berkeley Espresso logic minimization algorithm [19] minimizes the complexity of Boolean expressions, but forgoes logic reductions of the conditional statements that make up each Boolean value. Consequently, the BT-Espresso algorithm generates trees that have Condition nodes that logically subsume other Condition nodes under the same parent node (see Fig. 1). This leads to redundant logic and bloated BTs with reduced interpretability. The RE:BT-Espresso algorithm removes all float conditions that are logically subsumed by another float condition within the same expression, leading to fewer Condition nodes while maintaining the same logic (Alg. 1, line 5). The resulting trees are less redundant and more interpretable.

Factoring-Out Condition Nodes: The Berkeley Espresso logic minimization algorithm returns the set of Boolean expressions in disjunctive normal form (DNF). This often results in the same conditional statements appearing in every conjunction of a Boolean expression, making the generated BT-Espresso BT evaluate some Condition nodes many times. The RE:BT-Espresso algorithm factors out common Condition nodes via the distributive law. Specifically, the algorithm removes the common Condition nodes from each conjunction and places them under a Sequence node. The reduced conjunctions are then placed beneath a Selector node, which is placed under the Sequence node, to the right of the common Condition nodes (Alg. 1, line 6). This factoring places conditions that must be true for an action to trigger higher in the BT while also removing repetitive nodes, resulting in a more interpretable BT (Fig. 1).

Increasing Expressivity

To create more expressive BTs, the RE:BT-Espresso algorithm generates more unique Control nodes and BT structures when compared to its predecessor, which serve to increase the breadth of behaviors that can theoretically be constructed.

As will be seen, Parallel Selector Action nodes and Last Action Taken Sequences serve as *Suggested Alternate Expressive Subtrees (SAES).* These subtrees are created as a set of possible trees for an end user to explore adding to the main tree. Collisions would be created if these subtrees were attached to the root of the generated tree without end user oversight. Thus, an end user is needed to manually edit and add the subtrees to the generated tree. For simplicity of the algorithm, however, both Parallel Selector Action nodes and Last Action Taken Sequences are described as being attached to the root of the main tree.

Algorithm 1: RE:BT-Espresso. Convert a decision tree (dt) into a behavior tree (bt) heuristically minimizing the number of nodes. Rules are Boolean algebra expressions and dnfs are Boolean algebra expressions in disjunctive normal form. Algorithm additions to BT-Espresso [6] marked in blue. Algorithm notation consistent with notation used in [6].

```
   Input: dt - a decision tree
   Output: a behavior tree
 1: function RE:BT_ESPRESSO(dt)
 2:     rules ← DT_TO_RULES_PSA(dt)
 3:     inv_set ← BUILD_INV_SET(rules)
 4:     rule_dnfs ← LOGIC_MINIMIZER(rules)
 5:     rule_dnfs ← FLOAT_REDUCTION(rule_dnfs)
 6:     rule_dnfs ← PULLOUT_COMMON(rule_dnfs)
 7:     root ← Parallel()
 8:     for dnf in rule_dnfs do
 9:         seq ← Sequence node
10:         act ← dnf.action
11:         or ← Selector node
12:         seq.add_child(or)
13:         seq.add_child(act)
14:         for conjunction in dnf do
15:             and ← Sequence node
16:             for literal in conjunction do
17:                 if literal in inv_set then
18:                     inv ← Inverter(INV(literal))
19:                     and.add_child(inv)
20:                 else
21:                     cond ← Condition(literal)
22:                     and.add_child(cond)
23:                 end if
24:             end for
25:             or.add_child(and)
26:         end for
27:         root.add_child(seq)
28:     end for
29:     LAT_trees ← GEN_LAT_TREES(rule_dnfs)
30:     for subtree in LAT_trees do
31:         root.add_child(subtree)
32:     end for
33:     return root
34: end function
```

Inverter Nodes: BTs generated from the BT-Espresso algorithm cannot reveal inversely related behaviors due to the algorithm's inability to generate Inverter nodes. This limits the breadth of possible behaviors that can be constructed, and thus the expressivity of the generated BTs is reduced. The RE:BT-Espresso algorithm accommodates Inverter nodes by identifying inversely related conditional statements in the DT Boolean expressions, and representing one as the inverse of the other (Alg. 1, lines 3, 17-19). For example, if both condition $C_1 = (dist < 0.5)$ and $C_2 = (dist \geq 0.5)$ are present in a DT Boolean expression, RE:BT-Espresso

Algorithm 2: GEN_LAT_TREES. Generate behavior trees given an input *rules*, a list of Boolean expressions that represent the conditions for an action to be triggered. *Repeater* generates an LAT tree, replacing the Sequence node with a Repeater node. Algorithm notation consistent with notation used in [6].

Input: $rules$ - list of Boolean expressions
Output: list of behavior trees

```
function GEN_LAT_TREES(rules)
    G ← GENERATE_GRAPH(rules)
    G, cycles_d ← REMOVE_CYCLES(G)
    for path in G.paths do
        SN ← Sequence node
        for n in path do
            if is_cycle_node(n): then
                SN.add_child(Repeater(n, cycles_d))
                continue
            end if
            if first_node(n) then
                rule ← GET_DNF_RULE(n.action)
            else
                rule ← GET_LAT_CONJUNCT(n.action)
                rule ← rule \ lat_rules
            end if
            for seq_rule in rule do
                SN.add_child(Condition(seq_rule))
            end for
            SN.add_child(Action(n.action))
        end for
        return_list.append(SN)
    end for
    return return_list
end function
```

algorithm will detect the inverse relation, and represent C_2 as $\neg C_1$ via an Inverter node with the C_1 Condition node as the child (Fig. 1). When building the BT, all notted conditions are detected and given an Inverter node above the original Condition node.

Parallel Selector Actions: The leaf nodes of the learned DT have a class distribution which represents the number of examples for each action from the action space for the given decision path. BT-Espresso chooses the most probable action when constructing a corresponding DNF logical statement, even for high impurity leaf nodes. RE:BT-Espresso extends this by finding all actions in the DT leaf nodes that are δ-close to the most probable action according to a configurable fraction threshold δ. For a DT leaf node with class distribution $A = \{a_0, a_1, ..., a_n\}$, all actions $a_i \geq (1-\delta) \cdot max(A)$ are treated as a singular action during the Berkeley minimization, then separated again and placed under a "Parallel/Selector" node when the BT is constructed (Alg. 1, line 2). The suggested "Parallel/Selector" node is the root of a SAES, and allows the node to be changed to either a Parallel or Selector node at runtime via a visual editor, depending on what the user sees fit. This increases the expressivity of the BT, since a greater breadth of behaviors can be generated. Specifically, the end user can now choose which actions to include in the BT for a DT decision path that ended in a high impurity leaf node.

Last Action Taken Sequences: Task-level robot policies often execute many actions sequentially, regardless of the state space during the sequence. Since the BT-Espresso algorithm trains the DT solely on the world state space without considering patterns in the action being triggered, it cannot detect and generate sequences of actions found in the demonstration data. Since sequences of actions cannot be captured and constructed by the algorithm, the resulting BTs have limited expressivity.

The RE:BT-Espresso implements Last Action Taken Sequences, a SAES that captures common sequences of actions from the demonstration data. A last action taken (LAT) column is first added to the data for each action a_t which reflects the most recent action a_{t_0} such that $t_0 < t$ and $a_{t_0} \neq a_t$. This column is captured within the DT rules. As illustrated in Fig. 2, after the Berkeley minimization, conjunctions containing a LAT condition are found. A graph G is constructed with directed edges constructed from LAT action to action. The graph represents the action sequences learned from the demonstration data, where each node is an action, and the adjacent nodes of incoming edges represent last taken actions.

If G is a directed acyclical graph (DAG), a Sequence node SN is constructed for all paths from source nodes to sink nodes. For every path of the DAG, RE:BT-Espresso traverses the nodes. For every Action node, the Boolean expression is retrieved, and all conditions within the conjunction that contains the LAT condition are added to SN, not including the LAT condition itself. Finally, the Action node is added to SN (see Alg. 2 and Fig. 2).

Last Action Taken Repeaters: Cycles within G are turned into Repeater nodes when G is not a DAG. Repeater nodes act as Sequences nodes that loops the sequence continually until a failure. All cycles are found within G and eventually output as Repeater nodes as shown in Fig. 3.

First, all cycles in G are found. For all cycles, a node must be chosen from the cycle as the start of the path. There are two proposed strategies for determining the starting node. The first is to choose an arbitrary node. The second searches the original demonstration data for the various combinations

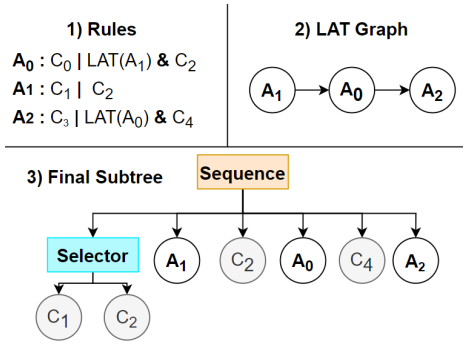

Fig. 2: Creating Last Action Taken (LAT) Sequence trees as described in Alg. 2. 1) Rule look up from action to conditions. 2) The LAT graph with edges [LAT action, action]. 3) The resulting LAT subtree with conditions. All conjunctions of A_1 are included. For A_0 and A_2, only the LAT conjunction is considered with LAT literals removed as the sequence guarantees the last action. Note C_2 must be checked again before A_0 as C_2 may change after completing A_1. This subtree is later added to the root node.

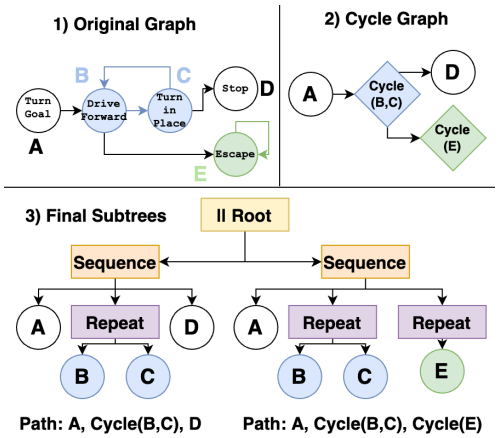

Fig. 3: Creating Last Action Taken (LAT) Repeater trees as described in Alg. 2 using a turn-go-turn mobile robot policy example. 1) The original LAT graph with edges [LAT action, action]. 2) The result of replacing cycles with cycle nodes. The original cycle paths are stored in a lookup for Repeater subtrees. 3) The resulting subtrees with respect to all paths within the cycle graph following the LAT Sequence builder as shown in Fig. 2. All subtrees are added to the root node. Condition nodes omitted for brevity.

of possible cycle sequences and chooses the starting node to be the starting node of the most frequent sequence. The cycle path starting at this node follows the same steps as constructing LAT Sequences above. The parent SN is then replaced with a Repeater node RN. The linked code in Sec. IV implements the first strategy.

Last Action Taken Sequences + Repeaters: To further capture more complex and expressive trees, LAT Repeaters and LAT Sequences are combined when G contains cycles. The following constitutes a single pass for the LAT Repeater subroutine. For graph G, all cycles are found and replaced with a cycle node CN as described in the above section. All nodes within the cycle are removed. Every edge going in or out of each node of the cycle are added to CN. CN is put into a lookup table from cycle node to Repeater subtree for the original cycle path. When nodes are part of multiple cycles, a single cycle can either be arbitrarily chosen for this pass on G or scored using a metric such as the most frequent cycle found within the original demonstration data.

A single pass does not guarantee a DAG for G if nodes are part of multiple cycles. A single pass, however, can lead to more paths as cycles are removed. A DAG can be guaranteed if this subroutine is repeated on the resultant graph G' until all cycles are removed. When all cycles are removed, the original LAT Sequence algorithm is run on the final resultant graph. When encountering a cycle node, the subtree within the cycle node look up is used with a Repeater parent node as opposed to a Sequence node parent (see Alg. 2 and Fig. 3). The linked code in Sec. IV runs the subroutine until all cycles have been removed.

IV. EXPERIMENT AND RESULTS

A. System Overview

The RE:BT-Espresso algorithm is implemented in `Python3` [20], tested, and available at `https://github.com/interaction-lab/RE-BT-Espresso`. The repository contains a BT simulator, BT builder pipeline, and experiment analysis.

The data simulator is designed as an agent-based modelling system [21]. The simulator creates BTs using `py_trees` [22] and simulates actions with response to environment variables. At each time step t, the simulator first updates environment variables to random values $[0.0, 1.0]$. The BT then performs a single tick. All actions taken, the action return value (i.e., *Success* or *Failure*), and environment variables are logged to a csv for t. All node types that RE:BT-Espresso can generate (e.g., Repeater nodes, Inverter nodes, Parallel Selector Action nodes) are supported. Actions and conditions further support configurable success and failure probabilities. To simulate parallel actions that happen over multiple time steps, actions have a configurable chance to return *Running* in which case no action is logged until a *Success* or *Failuire* is returned.

The BT builder pipeline accepts timeseries data containing actions as labels with a configuration file and outputs a set of BTs. These data are used to train DTs using `scikit-learn`'s [23] `DecisionTreeClassifier` using Gini Impurity as the measure of split quality. Multiple DTs are learned for all pruning values generated from minimal cost-complexity pruning [18]. Finally, the pipeline converts these DTs to BTs using RE:BT-Espresso and generates a folder of BTs, one for every level of pruning on the DT. The UC Berkeley Espresso logic minimization [19] within the pipeline is implemented via `pyeda`[24].

Experiment analysis consisted of comparing generated BTs (BT_{gen}) to the original simulated BT (BT_{sim}) by graph analysis using `networkx` [25]. For all levels of pruning BTs, each subtree of BT_{gen} is compared against BT_{sim}. Graph edit distance (GED) [26] is reported for graph similarity calculated using `networkx`'s heuristic `optimize_graph_edit_distance` for configurable number of iterations as exact GED is NP-Hard. A lower graph edit distance indicates a more similar graph. A custom node comparator is provided to equate nodes of the same type. The subtree with the lowest GED is reported.

B. Experiments

Subtrees were randomly auto-generated according to valid BT rules and simulated. These subtrees were combined in sets of 2 or 3 with a randomly chosen Parallel or Sequence root to form unique BT configuration files. The simulator was configured with the following: total number of time steps = 20,000; return *Running* chance = 40%. The BT builder pipeline was configured with the following: Parallel Selector Action node threshold δ = 0.3; max DT depth = 5; SVM-SMOTE Upsampling = `False`. Both RE:BT-Espresso and BT-Espresso were run to generate BTs to be compared in result analysis. Result analysis was configured to run the GED heuristic from `networkx` for 1 iteration.

For every subtree of the original simulated tree, every subtree of the generated tree from BT-Espresso was scored using GED choosing the lowest score among all generated subtrees.

The same was repeated for RE:BT-Espresso. Minimum and median were then calculated for each subtree. For example, a simulated tree with 2 subtrees and 20 pruning levels of DT resulted in 40 BT-Espresso GED scores and 40 RE:BT-Espresso GED scores. The cumulative statistics were taken from each subtree, leading to two scores for each statistic in this example (2 scores for each subtree for each algorithm).

The reduction of BT size was also used as a metric for interpretability. The simulated trees were rerun with a modified configuration that removed all SAES (Parallel Selector Actions, LAT Sequences, and Repeaters) from RE:BT-Espresso output to isolate the float representation reduction and factor out Condition nodes. The resultant trees were compared on the number of total nodes generated by each algorithm. The relative percent of nodes removed was calculated for all generated trees using the following metric: $\frac{num\_nodes(BTE) - num\_nodes(RE{:}BTE)}{num\_nodes(BTE)} * 100\%$

C. Results

Of the original 100 configuration files, 98 experiments completed with a total of 234 simulated subtrees. Two configuration files timed out for GED calculations (2 weeks of running the experiments) and were therefore not included in the results. The results for differences in GED can be found in Fig. 4. Of the 234 trees, RE:BT-Espresso's minimum score outscored BT-Espresso's minimum score on 54 subtrees, tied on 178, and lost on 2. RE:BT-Espresso's median score outscored BT-Espresso's median score on 52 subtrees, tied on 179, and lost on 3.

To measure the relative success of Repeater and Parallel Selector Action nodes, simulated subtrees containing a Repeater node or a number of Parallel Selector Action nodes were compared to RE:BT-Espresso's corresponding minimum scoring subtree. A total of 106 subtrees contained a Repeater node when generated by RE:BT-Espresso from the 125 simulated subtrees originally containing a Repeater node. A total of 65 of the 95 subtress contained exactly the same number of Parallel Selector Action nodes (simulated subtrees contained 0, 1, or 2 Parallel Selector Action nodes).

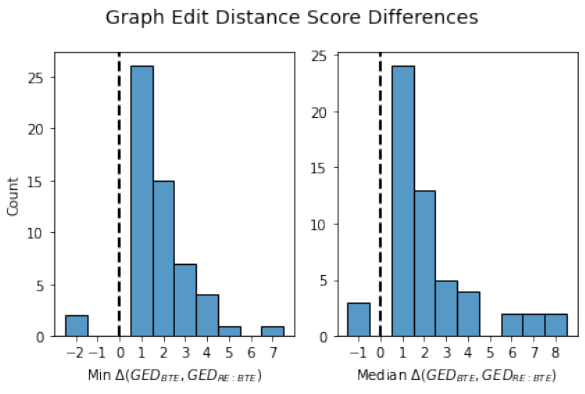

Fig. 4: Results of Graph Edit Distance (GED) [26] differences in minimum and median scores of BT-Espresso (GED_{BTE}) and RE:BT-Espresso ($GED_{RE:BTE}$) for all subtrees. See Sec. IV for details on scoring. Zero score differences (Min : 178, Median : 179) are removed for clarity.

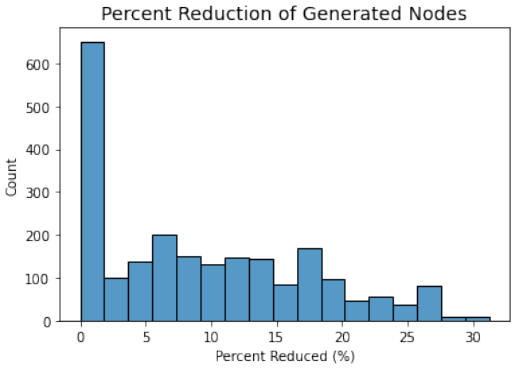

Fig. 5: Percent of reduction when applying float representation reduction and factoring out Condition nodes (Sec. III-B). The percent reduction is calculated as $\frac{num\_nodes(BTE) - num\_nodes(RE{:}BTE)}{num\_nodes(BTE)} * 100\%$ for all generated trees (2250 total). Median reduction: 7.82% fewer nodes; max reduction: 31.25% fewer nodes; and number of no difference found: 601.

.

The RE:BT-Espresso algorithm percent tree reduction from adding float representation reduction and factoring out Condition nodes is found in Fig. 5. Of the total 2250 created trees, 1649 trees ($\approx 73.29\%$) were reduced with a median reduction of 7.82% and a maximum reduction of 31.25%.

V. DISCUSSION

RE:BT-Espresso consistently and successfully regenerated trees that were more similar to the original simulated trees than BT-Espresso.

To the best of our knowledge, there is no known quantitative metric for evaluating BTs learned from robot demonstration data. DT intermediary approaches are typically validated by learning a BT from demonstration for a single high level task with task completion indicating a quality policy [5], [6]. Reinforcement Learning and Genetic Programming approaches are usually validated with their respective simulation and reward scoring to benchmark the simulated task [13], [14], [15]. The DT intermediary approaches do not have simulation, as the goal is to use real-world data to learn the policy; those data are expensive to collect, both in needed time and resources. We combined these approaches by simulating an exact BT for the demonstration data and comparing it to the algorithmically generated BT based on their respective graph edit distances. While imperfect, this approach provides a quantitative metric for evaluating differences between generated BTs.

End user non-expert programming of robot control policies is the ultimate goal of learning control policies from robot demonstrations. The RE:BT-Espresso pipeline outputs a BT of suggested subtrees that can be chosen from by the end user. The proposed optimal approach to choosing BTs is described in Sec. IV. The described tree coloring is designed to help users to be able to better understand common substructures of the interaction. These design choices, however, have yet to be explored with users. Future work will validate the usability of RE:BT-Espresso-generated BTs through a user study.

REFERENCES

[1] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5420–5427, IEEE, 2014.

[2] F. W. Heckel, G. M. Youngblood, and N. S. Ketkar, "Representational complexity of reactive agents," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pp. 257–264, IEEE, 2010.

[3] Y. Zhang, P. Tiňo, A. Leonardis, and K. Tang, "A survey on neural network interpretability," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2021.

[4] 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.

[5] I. Sagredo-Olivenza, P. P. Gómez-Martín, M. A. Gómez-Martín, and P. A. González-Calero, "Trained behavior trees: Programming by demonstration to support ai game designers," *IEEE Transactions on Games*, vol. 11, no. 1, pp. 5–14, 2017.

[6] K. French, S. Wu, T. Pan, Z. Zhou, and O. C. Jenkins, "Learning behavior trees from demonstration," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 7791–7797, IEEE, 2019.

[7] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu, "Definitions, methods, and applications in interpretable machine learning," *Proceedings of the National Academy of Sciences*, vol. 116, no. 44, pp. 22071–22080, 2019.

[8] O. Biggar, M. Zamani, and I. Shames, "An expressiveness hierarchy of behavior trees and related architectures," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5397–5404, 2021.

[9] G. Robertson and I. Watson, "A review of real-time strategy game ai," *Ai Magazine*, vol. 35, no. 4, pp. 75–104, 2014.

[10] D. Isla, "Gdc 2005 proceeding: Handling complexity in the halo 2 ai," *Retrieved October*, vol. 21, p. 2009, 2005.

[11] M. Matarić and F. Michaud, *Behavior-Based Systems*, pp. 891–909. 01 2008.

[12] M. Colledanchise and P. Ögren, "Behavior trees in robotics and ai," Jul 2018.

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

[14] L. Li, L. Wang, Y. Li, and J. Sheng, "Mixed deep reinforcement learning-behavior tree for intelligent agents design.," in *ICAART (1)*, pp. 113–124, 2021.

[15] Q. Zhang, J. Yao, Q. Yin, and Y. Zha, "Learning behavior trees for autonomous agents with hybrid constraints evolution," *Applied Sciences*, vol. 8, no. 7, p. 1077, 2018.

[16] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[17] M. Colledanchise and P. Ögren, "How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees," *IEEE Transactions on robotics*, vol. 33, no. 2, pp. 372–389, 2016.

[18] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Routledge, 2017.

[19] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, vol. 2. Springer Science & Business Media, 1984.

[20] Python Core Team, *Python: A dynamic, open source programming language*. Python Software Foundation, 2019. Python version 3.7.

[21] C. M. Macal, "Everything you need to know about agent-based modelling and simulation," *Journal of Simulation*, vol. 10, no. 2, pp. 144–156, 2016.

[22] *Py Trees*. 2021.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[24] C. Drake, "Pyeda: data structures and algorithms for electronic design automation," in *Proc. 14th Python in science conference (SciPy)*, pp. 26–31, 2015.

[25] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[26] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *4th International Conference on Pattern Recognition Applications and Methods 2015*, 2015.