2019 International Conference on Robotics and Automation (ICRA)
Palais des congres de Montreal, Montreal, Canada, May 20-24, 2019

Learning Behavior Trees From Demonstration

Kevin French Shiyu Wu Tianyang Pan Zheming Zhou Odest Chadwicke Jenkins

***Abstract*— Robotic Learning from Demonstration (LfD) allows anyone, not just experts, to program a robot for an arbitrary task. Many LfD methods focus on low level primitive actions such as manipulator trajectories. Complex multi-step task with many primitive actions must be learned from demonstration if LfD is to encompass the full range of task a user may desire. Existing methods represent the high level task in various forms including, finite state machines, decision trees, formal logic, among others. Behavior trees are proposed as an alternative representation of high level task. Behavior trees are an execution model for the control of a robot designed for real time execution, modularity, and, consequently, transparency. Real time execution allows the robot to reactively perform the task. Modularity allows the reuse of learned primitive actions and high level task in new situations, speeding up the process of learning in new scenarios. Transparency allows users to understand and interactively modify the learned model. Behavior trees are used to represent high level tasks by building on the relationship it has with decision trees. We demonstrate a human teaching our Fetch robot a household cleaning task.**

I. Introduction

Robots are ever more ubiquitous in the work place and at home. Single purpose robots, like Roombas, are common; yet, general purpose robots are not. The lack of general purpose robots can, in large part, be attributed to the need for an expert programmer. Programing every behavior a user may desire is practically impossible. The responsibility, necessarily, falls on the user, but the average user will not have the skill or time to program their desired behaviors. Learning from Demonstration (LfD) [1] provides a solution. LfD studies how a robot could learn from a natural human demonstration. Instead of using code, the user will just show the robot how to do what the user wants.

LfD has made significant progress, especially for motion primitives. It is clear that motion primitives are not enough to solve high level complex multi-step goal directed task, such as loading a bag of groceries. The robot must learn primitive actions, but also segment the primitive actions for learning and learn the transitions between the primitive actions. High level LfD methods require all three abilities. Techniques for high level LfD range from regression based [2], [3], [4], to declarative [5], [6], [7].

High level LfD methods disagree on how they represent a learned policy. A good policy is modular, transparent, reactive, and readily executable. A modular policy leads to faster learning overtime, as learned structures are reused.

K. French, Z. Zhou, are with the Robotics Institute and S. Wu, T. Pan, and O.C. Jenkins are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, 48109-2121 `[kdfrench|shiyuwu|typan|zhezhou|ocj]@umich.edu`

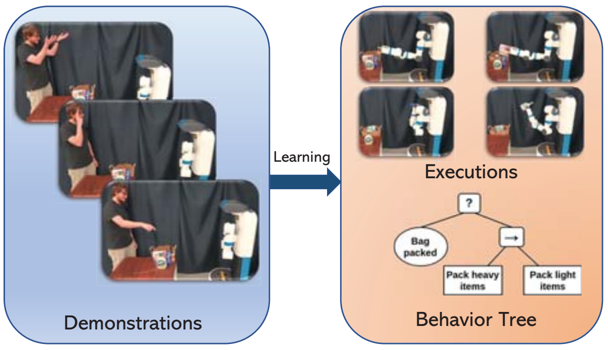

Fig. 1. Demonstrations to Behavior Tree. A human tells a robot what actions to take to pack a grocery bag. The robot learns from these demonstrations a behavior tree representing the task.

A transparent policy promotes interactive learning and allows a user to better understand the learned behaviors. A reactive policy is highly responsive, real time, and adaptive to changing stimuli. A readily executable policy is its own execution layer, directly outputting robot commands. No policy currently in use rates highly in all four of these properties, but behavior trees [8] do.

Behavior trees are a policy representation explicitly designed for modularity and speed. They have gained popularity in video games, as a replacement to finite state machines [9]. Recently they have been adapted, formalized, and commercially used in robotics [8]. Consider the grocery packing task depicted in figure 1. The behavior tree, modularly composed of subtrees *pack heavy items* and *pack light items*, in figure 1 will reactively pack all heavy items before light items until the bag is packed all while indicating the status of execution to the user. The approach section explains in detail how a behavior tree provides these benefits.

This paper shows how behavior trees can be learned from demonstration. A state space and action space is assumed to exist for the world. Human demonstrations are provided to the robot. The Classification and Regression Tree algorithm (CART) [10] is used to produce a decision tree from the demonstrations. We propose BT-Espresso (algorithm 2) to convert the decision tree into a behavior tree. Finally, the behavior tree is executed to robustly achieve the demonstrated task. A Fetch robot is taught to use a micro fiber dusting tool to dust a table, as a proof of concept.

II. Related Work

In this section high level learning from demonstration is explained, followed by behavior trees, and finally how behavior trees have been used in learning from demonstration.

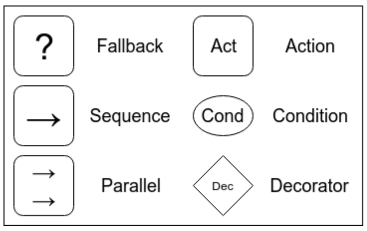

Fig. 2. Standard behavior tree nodes. A table of graphical symbols representing the six standard behavior tree node types.

A. High level Learning From Demonstration

This work focuses on a subset of Learning from Demonstration (LfD) [1] techniques, high level LfD. High level LfD learns complex multi-step policies composed of several primitive actions sequenced together, such as cleaning a kitchen, from user demonstrations. High level LfD can be broken down into three major components; segmenting primitive actions, learning the primitive actions, and learning the transitions between primitive actions. Segmenting primitive actions is the process of finding repeated substructure in demonstrations. The repeated structures are grouped together and passed to a primitive action learner. Learning primitive actions, low level LfD, has been well studied and the reader is referred to [1] for a broad review of LfD. Learning transitions between primitive actions is the generalization of observation of the world before and after actions are taken. Learning the transitions is critical to generating high level complex multi-step policies.

Early work in learning high level LfD involved the used of regression methods such as Gaussian mixture models. In [4] sparse online Gaussian processes are used to segment and learn primitive actions. The transitions between learned actions are modeled as a finite state machine. Visual aliasing, two highly similar observations mapping to two different actions, prevented the successful learning of a finite state machine in [4]. Building on the work of [4], [2] used a beta process auto-regressive hidden Markov model (BP-AR-HMM) to segment demonstrations into primitive actions. The primitive actions are then modeled by Dynamic Motion Primitives. Then the visual aliasing problem is overcome by incorporating a history and interactive user demonstrations. A finite state machine is generated using a nearest neighbors classifier for state transitions. Note that both of these methods [4], [2] learn all three of the components of a high level complex multi-step policy.

Learning just the transitions between primitive actions, [3] takes an information theoretic approach to high level learning from demonstration. Using crowd sourced demonstrations a decision tree is learned maximizing information gain at every node of the decision tree. The decision tree is used to directly map the observation of the world to a primitive action.

Until this point only classification/regression based learners have been discussed but others have used declarative methods. Declarative methods seek to represent the world as logical statements that can be reasoned over. [5] learns grounded symbols of a formal logic, Planning Domain Definition language (PDDL) and Probabilistic PDDL (PPDDL), from demonstration. These symbols can then be used to learn and or declare the goal of a high level task. A planner then plans actions that move the current state of the world to the goal of the high level task. In [7] a human demonstrator narrates his actions to help a robot actively learn a hierarchical task network (HTN). During the process the primitive actions are learned by learning the constraints on the motion instead of generalizing a trajectory through their Task Space Region (TSR).

B. Behavior Trees

Behavior trees were originally designed as a formal representation for functional requirements of a piece of software [11]. Behavior trees then became popular in the video game industry for non-player characters [12]. More recently, behavior trees have been adapted for use in robotics [8] and shown to be a generalization of Hybrid Control Systems, Generalize Sequential Behavior Compositions, Subsumption Architecture and decision trees [13]. Behavior trees have been used in various real robotic systems including Rethink Robotics's industrial robotics Intera software, JIBO the social robot, and autonomous mining vehicles from Scania iQmatic project [14].

CoSTAR [15] is an industrial robotics research project designed to make low cost industrial robots like Universal Robots UR5 robot usable in small manufacturing entities. CoSTAR uses behavior trees as the core task representation. The capabilities of the robot are expressed as individual behaviors of the robot. Task predicates are expressed as behavior conditions. By exposing the behavior tree through a graphical user interface that allows generation and editing of behavior trees, CoSTAR allows for transparent analysis, general structure, and classification of execution traces relative to a dictionary of known task [15].

C. Behavior Trees In Learning From Demonstration

We believe that behavior trees have appeared in learning from demonstration contexts only three times before [14], [13], [12]. In [14], it was mentioned that Rethink Robotics's Intera software has the capability to either build behavior trees graphically or construct them from demonstration. We believe that this code has not been made public nor is there a publication related to how behavior trees are learned from demonstration.

In [13], the equivalence between decision trees and behavior trees is shown. It is stated that this allows for the use of learning from demonstration work to leverage behavior tree research. This work formalizes the claim and shows how behavior trees can be used in learning from demonstration, taking into account practical considerations of a real robotic system. The equivalence between decision trees and behavior trees will be discussed further in the approach section.

In [12], behavior trees are used for non-player characters in a video game. A human provides demonstrations to the system and then the C4.5 algorithm is used to learn a decision

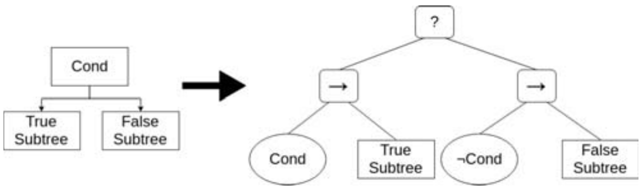

Fig. 3. Decision tree conversion to behavior tree. A generic decision tree left is converted to a behavior tree right. The true and false subtrees are protected by a guard condition preventing execution of the subtree when its preconditions are not met. The guards allow actions to fail and succeed.

tree. The tree is flattened into a set of rules, which are further simplified using a greedy algorithm. From the simplified rules, a behavior tree is constructed. Our paper shares a similar process, but generalizes the greedy algorithm to any logic simplification algorithm and is deployed on a Fetch robot to do household task.

III. Approach

A. Background

A common policy representation in LfD is the finite state machine. Finite state machine nodes can be thought as, the arguably defunct, goto statement. Control is transfered from node to node; resulting in a large web of transitions akin to spaghetti code. Behavior trees, on the other hand, can be thought of as functions. A node is called by its parent, executes its program, and returns a status from $\{Success, Failure, Running\}$. The parent's execution is altered by its children's return statuses. Control flows back and forth between parent and children; resulting in concise, transparent, and modular behavior trees.

The reader is refereed to [8] for a thorough definition of a Behavior tree. Below is a brief overview for the readers convenience.

A Behavior tree is a directed acyclic graph composed of 2 different types of nodes: control, and action. Control nodes, branches, have exactly one parent and $n \in \mathbb{N}$ children. Listed below (and shown in figure 2) are the standard control nodes:

- Sequence - Children are executed in order.
- Fallback - Children are executed in order until any child returns *Success*.
- Parallel - Children are executed simultaneously.
- Decorator - The status of the only child is mapped to a new status.

Action nodes, leafs, have exactly one parent and no children. They represent the capabilities of the system. Action nodes are physical actions and evaluation of logical conditions.

Behavior trees are reactive. Long running processes are run in a separate thread so the tree can poll the status and return *Running* until the process has returned *Success* or *Failure*.

B. BT-Espresso Algorithm

An algorithm is developed for high level learning from demonstration (LfD) using a behavior tree as the policy model. The behavior tree is used for its modularity, transparency, reactivity and ability to be directly executed.

Algorithm 1 Naive Decision Tree to Behavior Tree. Convert a decision tree (dt) into a behavior tree (bt)

INPUT: node - a node from a decision tree either with an associated action (leaf nodes) or a condition, true child, and false child (branch nodes)
OUTPUT: a behavior tree

```
 1: function DT_TO_BT(node)
 2:     if IS_LEAF_NODE(node) then
 3:         return node.action
 4:     else                              ▷ Construct behavior tree nodes
 5:         root ← Fallback node
 6:         true_seq ← Sequence node
 7:         false_seq ← Sequence node
 8:         true_cond ← Condition(node.condition) node
 9:         false_cond ← Condition(¬node.condition) node
10:         true_action ← DT_TO_BT(node.true_child)
11:         false_action ← DT_TO_BT(node.false_child)
                                                      ▷ Build tree
12:         root.add_children(true_seq, false_seq)
13:         true_seq.add_children(true_cond, true_action)
14:         false_seq.add_children(false_cond, false_action)
15:         return root
16:     end if
17: end function
```

The equivalence between a behavior tree and a decision tree is shown in [13], but it is assumed that the action nodes always return running. Actions do not always return running; they may succeed or fail. Success and failure are accommodated by adding a guard. A guard is a predicate that ensures a precondition is met before a subtree is executed. Taking success and failure into account leads to algorithm 1. Algorithm 1, visualized in figure 3, is the proof of equivalence between decision trees and behavior trees, shown in [13], but in algorithmic form accounting for success and failure.

Algorithm 1 naively converts a decision tree into a behavior tree. The naivety comes from not taking advantage of properties of decision trees to improve the behavior tree. The remainder of this section is devoted to developing BT-Espresso (algorithm 2)which does take advantage of the properties a decision tree.

Decision trees are not guaranteed to be minimal, and are often possible to perform logic simplification on. A greedy simplification algorithm is proposed in [12]. This work goes a step further. First, it is shown that the output of algorithm 1 can be rearranged to allow for parallel execution. Second, logic minimization is used to generate a behavior tree that requires less computation per execution cycle. Finally, exact logic minimization via the Quine-McCluskey algorithm [16] is compared to a heuristic based approach, UC Berkeley Espresso [17].

Algorithm 1 generates a sequential behavior tree. A bet-

Algorithm 2 Behavior Tree 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.

INPUT: dt - a decision tree
OUTPUT: a behavior tree

```
1:  function BT_ESPRESSO(dt)
2:      rules ← DT_TO_RULES(dt)
3:      rule_dnfs ← LOGIC_MINIMIZER(rules)
4:      root ← Parallel()
5:      for dnf in rule_dnfs do
6:          seq ← Sequence node
7:          act ← dnf.action
8:          or ← Fallback node
9:          seq.add_child(or)
10:         seq.add_child(act)
11:         for minterm in dnf do
12:             and ← Sequence node
13:             for predicate in minterm do
14:                 cond ← Condition(predicate) node
15:                 and.add_child(cond)
16:             end for
17:             or.add_child(and)
18:         end for
19:         root.add_child(or)
20:     end for
21:     return root
22: end function
```

ter algorithm would exploit the parallel control node. BT-Espresso (algorithm 2)starts by expressing a decision tree as a set of boolean equations, through an exhaustive depth first traversal of the decision tree. Each leaf node generates a new boolean conjunctive clause of the form $c = \bigwedge_{i=0}^{n} h(f_i)$ where c is the class output defined by the leaf node, f_i is the i-th feature split on the way to the leaf node, and $h(f_i)$ returns $\neg f_i$ if to get to the leaf node feature f_i must be false otherwise f_i is returned. All classes c with multiple equations have all equations joined into a single disjunctive. The set of equations forms a tautology, because only one path through a decision tree can be true at a time.

The set of equations may not be minimal, so BT-Espresso (algorithm 2)applies logic minimization to reduce the complexity of the final behavior tree (Note, that any tautology, not just one produced from a decision tree, could be used at this point). Logic minimization is an NP-Hard problem with exponential complexity. Small equations are exactly minimized in reasonable time using the the Quine-McCluskey algorithm [16]; however, the algorithm is intractable for any realistically sized problem. Thus, the heuristic based UC Berkeley Espresso logic minimization algorithm is used [17]. The output of the logic minimizer will be a set of disjunctive normal form equations for each class c.

BT-Espresso (algorithm 2)converts each disjunctive nor-

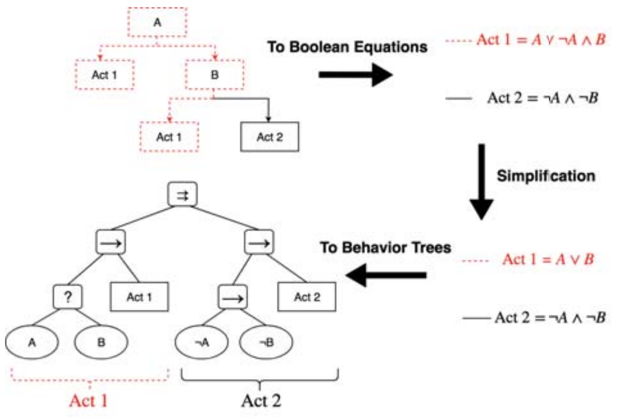

Fig. 4. Conversion of decision tree into parallel behavior tree. In this graphical representation of algorithm 2 a decision tree is flattened to boolean equations, simplified, and converted to a behavior tree. Highlighted in dotted red lines is Act 1's conditions. Notice that each class, Act 1 and Act 2, are generated as their own child of the parallel node. Also notice how fallback nodes act as a $\vee$ operator and sequence nodes act as a $\wedge$ operator.

mal form equation, independently, into a behavior tree. Boolean *And* and *Or* functions are directly represented by a sequence node and a fallback node respectively. Finally, each individual behavior tree is combined as a child of a parallel node.

Three different behavior trees are generated by changing the minimizer in line 3 of BT-Espresso (algorithm 2). Skipping minimization is equivalent to algorithm 1, but turns makes a parallel behavior tree. Using the Quine-McCluskey algorithm [16] as the minimizer results in a guaranteed minimal behavior tree in that the conditions for each classification are minimal, but for any realistically sized behavior tree is intractable. Using a heuristic minimizer, such as UC Berkley's Espresso algorithm, leads to a minimal or nearly minimal behavior tree, in a tractable manner. A visualization of BT-Espresso (algorithm 2) is provided by figure 4.

It is reasonable to wonder if one could directly execute the flattened minimal logic, but, in actuality, it can not be executed directly. These logical equations do not define how they should be executed; what rate should they be evaluated, what happens if the equation becomes false before the action is complete, should the equations be evaluated in sequence or parallel, etc. An implementation layer must be designed on top of the logic. A behavior tree is a principled way to define the implementation layer with beneficial properties such as transparency, modularity, and reactivity.

C. System Overview

The robot is equipped with a state space and action space for use in learning the tasks. The state space and action space were made by hand for the purpose of these experiments, but a complete system should generated the spaces with low level learning from demonstration techniques. While demonstrating, every action that the robot is told to perform

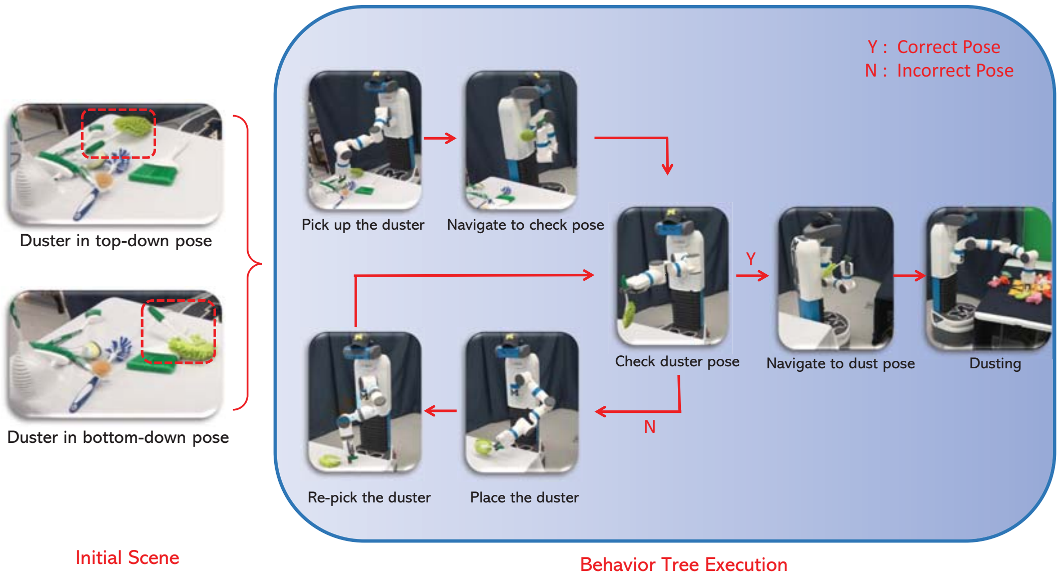

Fig. 5. Execution of a learned behavior tree. The robot executes a dusting sequence learned from demonstration. The robot has learned that it can only use the duster when it is correctly orientation, face-down. The robot has learned how to flip the duster to correct the orientation.

and current state pair is recorded. In this way, the record mapping, teacher execution to recorded execution, and the embodiment mapping, recorded execution to recorded states and actions to learning algorithm, are both identity. Many works focus on the record and embodiment mappings and could be applied to this work to allow for more natural teaching by a human teacher [5].

The Classification and Regression Tree algorithm (CART) [10] is used on all demonstrations associated with the task to get a decision tree, when the task has been fully demonstrated. Finally the decision tree is converted to a behavior tree with BT-Espresso (algorithm 2). The behavior tree is then used to execute the task without human input.

IV. Results

A. Experimental Setup

To test algorithm 1 and BT-Espresso (algorithm 2), a series of test were developed. The robot was given a series of household task to complete. One of the task that was taught to the robot was to dust. The robot had no prior knowledge of the task, and had to be taught how to execute the task from scratch.

A state space and action space was developed for the dusting task. The state space used in these experiments consists of the last ten actions, the state of the joints, the position of the robot in the world, consolidated point cloud data from the camera (such as centroids of Euclidean segmentations), and properties of the duster orientation (face-up or face-down). The action space includes around 40 high level actions including object detection, grasping, navigation, and text to speech actions.

The state space is used as the input to the classifier and the actions are the outputs or classifications. Different settings were tested for the Classification and Regression Tree (CART) [10] algorithm. It was empirically determined that for these experiments, the best settings for the CART algorithm [10], as implemented by scikit-learn [18], are as follows: no max tree depth, no minimum number of samples for a leaf node, and balanced class weight. The number of demonstrations required was minimal with these settings. These settings worked well because there are a relatively small number of demonstrations and there are dominant classes. The nature of the experiments requires some task to be repeated many times, generating the dominant classes, while others only have one example per execution of the task. The tree becomes biased without the balancing of class weights.

The user is provided with a graphical user interface (GUI) for providing demonstrations. The GUI is comprised of three windows: RVIZ, navigation, and action selection. RVIZ, from the Robot Operating system, graphically represents the world. Navigation, from [19], provides planning and visualization of navigation task. Action selection provides buttons for all possible actions and shows the state vector.

A user teaches the robot how to dust with a micro fiber duster using the GUI. The task is initiated with the robot at a cluttered pile of cleaning supplies of which one is the duster. The goal of the task is for the robot to clean up a surface with the duster.

The dusting task requires the robot to performing the following functions:

- Find the handle of the duster in clutter
- Grasp the handle of the duster in clutter
- Determine the orientation of the grasped duster
- Fix the orientation of the grasped duster if required
- Navigate to the dirty surface

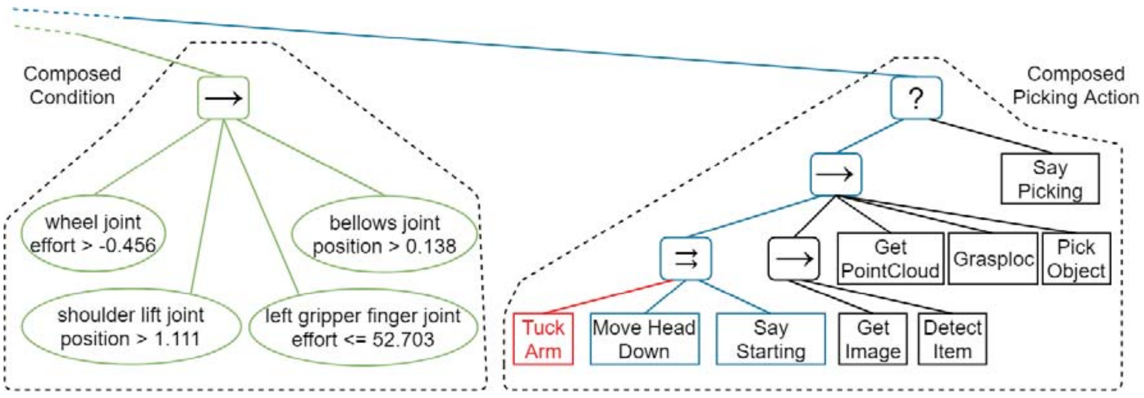

Fig. 6. BT-Espresso (algorithm 2) learned behavior tree. A small section of a behavior tree learned with BT-Espresso (algorithm 2) is shown. The composed condition in the dotted lines expresses the last minterm in the disjunctive normal form boolean equation that must be true for the composed action in the dotted lines on the right to activate. This particular action was chosen to highlight the modularity of behavior trees. An action, in this case a pick action, can be represented by a behavior tree that was previously learned or constructed. It is also easy to see how introspection can be done, the colors indicate the states of the nodes, green for success, blue for running and red for failure. Best viewed in color.

- Dust the surface

In finding and checking the pose of the duster, we deployed Faster-RCNN as the object detector [20]. The duster in training data is labeled into different parts (e.g. handle, feather-top, feather-down) and fine-tuned on VGG-16 pre-trained model. For robot grasping, we use a custom manipulation pipeline developed by the Laboratory for Progress. The pipeline uses our implementation of handle grasp localization as proposed by ten Pas and Platt [21].

The duster has an orientation that it must be used in to achieve contact with the microfiber cloth and the surface. A human can easily use dexterous motions of the fingers to orient the duster, but the robot cannot do the same by the motions available at the gripper. To fix the orientation of the duster, the robot places its grasper near a clear surface on a table to flip the duster. If flipping fails, the robot must repeat the process until the orientation is correct.

B. Discussion

The demonstration sequence is naturally taught by a user. If the robot fails to flip the duster, a human will tell it to try again. Effectively, the demonstration process is teaching the robot an affordance for the object. The robot learns that to use the affordance of dusting, it must hold the duster in the correct orientation. Learning the affordances can be applied to other objects in the same way.

The dusting task is fully learned after an average 7 runs of the task. The task is learned in 40 minutes, at an average of 5 demonstrations a minute. The Classification and Regression tree algorithm [10] and post processing with BT-Espresso (algorithm 2) takes less than a second. For comparison, using Quine-McCluskey algorithm [16] for logic minimization took 7 minutes on the same task. The speed of learning allows testing the learned behavior at any point in the demonstration process.

Figure 5 shows the initial scene as well as the behavior tree execution of the experiment. In the initial scene, the duster could be in either face-up pose or face-down pose. If the robot grabs the duster in face-up pose, it is unable to dust due to the wrong orientation of the duster.

The center figure of figure 5 shows the most critical action, checking the orientation of the duster. The robot will navigate to the dust pose and dust if the orientation is correct; otherwise, the robot would try to flip the duster and check the orientation again.

The condition of the orientation of the duster enables the robot to handle different states of the world rather than executing everything in a single sequence. The property is inherited from the decision tree the robot learned.

The loop of checking duster orientation and flipping the duster shows that the robot can recover from failures when executing the behavior tree. The robot would continue to execute the flipping process until the orientation of the duster is correct.

Figure 6 shows a portion of the learned behavior tree. This particular section is the learned behavior for picking up the handle of the duster.

Last actions are used in the state space to allow the robot to learn temporal relations. Without last actions in the state space the robot cannot learn sequences of actions. In some cases this is not a problem because another part of the state space changes when the action is taken.

V. Conclusion

Learning from Demonstration studies how robots can be programed with natural human interfaces, a critical factor for general purpose robots realizability. General purpose robots need to learn complex high level goal oriented task. Major progress has been made towards this end, but many works do not focus on the learned policy, just how to learn it. This work proposes the behavior tree as a policy representation, because it is modular, transparent, and reactive. BT-Espresso (algorithm 2) was developed to learn a behavior tree from demonstration. BT-Espresso (algorithm 2) was validated by teaching a Fetch robot to dust.

REFERENCES

[1] 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.

[2] S. Niekum, S. Osentoski, G. Konidaris, S. Chitta, B. Marthi, and A. G. Barto, “Learning grounded finite-state representations from unstructured demonstrations,” *International Journal of Robotics Research*, vol. 34, no. 2, pp. 131–157, 2015.

[3] C. Crick, S. Osentoski, G. Jay, and O. C. Jenkins, “Human and robot perception in large-scale learning from demonstration,” in *Proceedings of the 6th international conference on Human-robot interaction*. ACM, 2011, pp. 339–346.

[4] D. H. Grollman and O. C. Jenkins, *Can We Learn Finite State Machine Robot Controllers from Interactive Demonstration?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 407–430. [Online]. Available: https://doi.org/10.1007/978-3-642-05181-4_17

[5] G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez, “From skills to symbols: Learning symbolic representations for abstract high-level planning,” *Journal of Artificial Intelligence Research*, vol. 61, pp. 215–289, 2018.

[6] Z. Zeng, Z. Zhou, Z. Sui, and O. C. Jenkins, “Scene-level Programming by Demonstration.”

[7] S. Chernova and M. Veloso, “Confidence-based policy learning from demonstration using Gaussian mixture models,” *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems - AAMAS ’07*, vol. 5, p. 1, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1329125.1329407

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

[9] G. Robertson and I. Watson, “A Review of Real-Time Strategy Game AI,” *AI Magazine*, vol. 35, no. 4, p. 75, 2014. [Online]. Available: https://aaai.org/ojs/index.php/aimagazine/article/view/2478

[10] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*, ser. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984. [Online]. Available: https://books.google.com/books?id=JwQx-WOmSyQC

[11] R. G. Dromey, “From requirements to design: Formalizing the key steps,” in *Proceedings. First International Conference on Software Engineering and Formal Methods*, 2003, pp. 2–11.

[12] 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*, 2017.

[13] 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, 2017.

[14] M. Colledanchise, “Behavior trees in robotics,” Ph.D. dissertation, 2017, qC 20170308. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-202926

[15] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, “A framework for end-user instruction of a robot assistant for manufacturing,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6167–6174.

[16] E. J. McCluskey Jr, “Minimization of boolean functions,” *Bell system technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

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

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[19] J. J. Park, C. Johnson, and B. Kuipers, “Robot navigation with model predictive equilibrium point control,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 4945–4952.

[20] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Advances in neural information processing systems*, 2015, pp. 91–99.

[21] A. Ten Pas and R. Platt, “Localizing handle-like grasp affordances in 3d point clouds,” in *Experimental Robotics*. Springer, 2016, pp. 623–638.