Adapting In-Game Agent Behavior by Observation of Players Using Learning Behavior Trees

Emmett Tomai
University of Texas – Pan American
1201 W. University Dr.
Edinburg, TX 78539, USA
tomaie@utpa.edu

Roberto Flores
University of Texas – Pan American
1201 W. University Dr.
Edinburg, TX 78539, USA
rfloresx@broncs.utpa.edu

ABSTRACT
In this paper we describe *Learning Behavior Trees*, an extension of the popular game AI scripting technique. Behavior Trees provide an effective way for expert designers to describe complex, in-game agent behaviors. Scripted AI captures human intuition about the structure of behavioral decisions, but suffers from brittleness and lack of the natural variation seen in human players. Learning Behavior Trees are designed by a human designer, but then are trained by observation of players performing the same role, to introduce human-like variation to the decision structure. We show that, using this model, a single hand-designed Behavior Tree can cover a wide variety of player behavior variations in a simplified Massively Multiplayer Online Role-Playing Game.

Categories and Subject Descriptors
J [**Computer Applications**]

General Terms
Algorithms, Design, Experimentation, Human Factors.

Keywords
Artificial intelligence, game AI, agents, learning, MMORPG.

1. INTRODUCTION
Artificial Intelligence in games remains primarily the domain of simple, fast approaches such as scripting. With a few notable exceptions, such as Orkin's planning AI for Monolith's *F.E.A.R.* [14], Evan's work on Lionhead Studio's *Black & White* and his later formal logic for storytelling in Linden Lab's *Versu*, most games have stayed away from complex AI techniques, which can be computationally expensive and difficult for designers to control. Scripting is advantageous in both those regards, as it is simple to write, cheap to run and well understood in the industry. For most games, giving the designer the ability to precisely specify what will happen in-game is a higher priority than creating more dynamic interactions. However, the high cost of developing finely hand-tuned game play, which players consume far more quickly than it can be created, has created more interest in automatic content generation [cf. 20], and supported a wide range of efforts to learn human-like agent behavior in games [cf. 6]. Experienced players are increasingly looking for new experiences, creating new opportunities for AI in games. At the same time, scripting approaches have become more sophisticated. As games have gotten more ambitious, bigger, and harder to maintain, the ad hoc tangles of finite state machines (FSMs) used for agent control have become more and more unwieldy. Advanced engineering techniques such as hierarchical FSMs and *Behavior Trees* [11] have been used to attempt to address these concerns.

Human designed scripts capture expert intuition as to how in-game agents should behave. Through time consuming iterative development and testing, they are able to create agents that entertain human players effectively. However, as with most hand-engineered approaches, scripts suffer from repetitiveness, predictability and lack of naturally nuanced variations. The gap between playing with or against scripted AI and playing with or against other players is vast. In this paper, we consider how designer-created Behavior Trees could be automatically modified to display characteristics of human players performing the same role. We begin with straightforward, deterministic Behavior Trees for agents that play the role of human players in a Massively Multiplayer Online Role-Playing Game (MMORPG). Such online virtual worlds are an increasingly significant venue for human interaction, and provide an interesting problem for agents because of the high degree of freedom afforded to players. A well-designed Behavior Tree might capture optimal behavior for a player, according to some metrics, but would be hard-pressed to cover the range of behavior variation seen across a population of players in these open world games. This problem is notable because these games are at the mercy of difficult to predict population dynamics, making the use of intelligent agents for preliminary testing and creating on-demand populations very desirable. The potential of embodied, virtual interaction also extends to education, training and scientific research [cf. 2,5], where virtual agents could play an important role as guides and assistants. We present *Learning Behavior Trees*, an extension of Behavior Trees to observe human player traces and adapt a human-designed tree to cover the variations that are observed.

2. RELATED WORK
In the domain of video games, particular interest has been shown in developing human-like behavior for agents in the *first-person shooter (FPS)* genre. Geisler noted the high predictability and manual labor involved in traditional AI scripting of game agent opponents (*bots*) as motivation for automatic learning of human-like behavior [8]. These behaviors include low-level movement primitives such as changing direction, changing speed and

jumping, as well as basic game actions such as aiming and firing a weapon at opponents. Gamez showed that a global workspace architecture combining independent, hand-tuned neural networks can deliver human-like bot control [7], while Thurau used self-organizing maps and artificial neural networks to learn those primitive actions based on position and relative enemy positions [18]. Geisler evaluated both naïve Bayes and neural network approaches to this problem with promising results [8]. Additionally, a number of evolutionary approaches have been evaluated for developing human-like agent controllers, focusing primarily on human-like movement. Graham used a genetic algorithm to evolve an artificial neural network that implements dynamic obstacle avoidance while following a direct path [10]. Togelius evaluated several co-evolution strategies for creating car racing controllers with the aim of deploying a diverse population of human-like AI opponents in a car racing game [19]. These approaches were evaluated according to whether they effectively traverse space while avoiding obstacles and hitting checkpoints. Similarly, Lim [] and Perez [] used evolution to assemble Behavior Trees from sub-tree options, to maximize certain functional evaluations. All of these, and numerous other results [cf. 6] have demonstrated that machine learning and evolutionary computation are well suited to optimizing behavior control, particularly in domains where the problem has a reactive nature (e.g. following a twisting path, positioning relative to other agents, strategic responses) and a small number of output dimensions (e.g. movement and facing). However, Bakkes argues that more complex behaviors require working at multiple levels of abstraction (e.g. long-term goals and planning) [1].

Several established cognitive architectures, designed for deep, complex, human-like reasoning, have been applied to the problem of learning goal-based movement in games. Soar was proposed for creating synthetic adversaries in the MOUT (Military Operations on Urbanized Terrain) domain, emphasizing believability and diversity [22]. It was evaluated on its ability to show transfer learning for different goal locations and topologies [9]. Best detailed how ACT-R could be used in the same domain with lower-level perceptual input only [3]. Both systems learn from experience how to accomplish a certain goal. Several approaches have augmented this idea be combining human-encoded knowledge with learner behaviors. Spronck applied *Dynamic Scripting* to both group combat in the *Role-Playing Game (RPG)* genre and strategic decision-making in the *Real-Time Strategy (RTS)* genre [17]. A knowledge base of manually created rules is combined with learning inclusion and ordering of those rules into scripts. Marthi used *Hierarchical Reinforcement Learning* for learning joint movement of units in the RTS domain [13]. The reinforcement learning of movement is embedded in a manually created *concurrent ALisp* program. The program encodes knowledge about the task context and controls both the training and execution of the learned behaviors in that context. We propose a similar approach in this work, with a more explicit, declarative composition. Finally, Schrum has created a FPS bot architecture that learns combat behavior using *Neuroevolution* [15] and won the 2K Games' 2012 BotPrize while being judged as human more than 50% of the time [12]. The learned combat behavior is one component of the architecture, organized in a Behavior Tree-like structure that encodes human intuition about the priority and trigger conditions for that behavior and others. In this work we look more generally at Behavior Trees as a flexible controlling architecture for mixing learned and procedural behaviors.

3. PLAYER BEHAVIOR IN MMORPGS

In an MMORPG, players control avatar characters in a physically simulated virtual world that is shared and persistent. In contrast to more reactive and/or linear environments in other genres, players roam freely in the world, picking up tasks and completing them at their own discretion. Many tasks, or *quests*, are acquired from *non-player characters (NPCs)* which are system-controlled agents that provide static, motivating dialogue along with the task assignments. To complete a quest, a player usually travels to other regions of the world where they fight enemies and interact with other entities to fulfill the task requirements. A major part of those interactions is collecting useful virtual items, for example looting the corpse of a defeated foe to find new weapons. The most prevalent quest tasks ask the user to kill or collect a certain number of a certain type of entity or item. When the tasks for a quest are complete, the player will often return to an NPC to receive credit. Players can hold several quests at once, and start and stop pursing them at any time. Unlike many avatar-based genres, MMORPGs do not have a strong element of racing against time, and allow players to idle around and socialize. In this environment, there is an extremely wide range of player behaviors, even though the actual set of in-game character actions is very small. This makes it challenging to script any sort of player-like activity for in-game agents.

To collect player behavior data, we created a lightweight, research focused MMORPG-type game. The game collects a data for each player, including movement, avatar actions (attack, loot, interact and gather), per-player events (e.g. progress made on a task) and UI actions. In post-processing, the actions and events for each player were divided into sequential *journeys*: segments starting and ending with productive NPC interactions. Productive is defined as accepting a new quest, or turning in a completed one. Figure 1 shows the system visualization of two player journeys. Both players received the same quest from the NPC (N) at the top, to fight and kill three enemies (called *mobs* in the genre) in a nearby region. Each fight that contributed to the quest goal is shown as a white circle (F), while fights that did not contribute to any quest goal held by the player are shown as green circles (F). The player on the left was very efficient, going from the NPC to three fights and back. The player on the right, in contrast, added numerous fights that did not advance quest goals, and traveled to another region in the process.

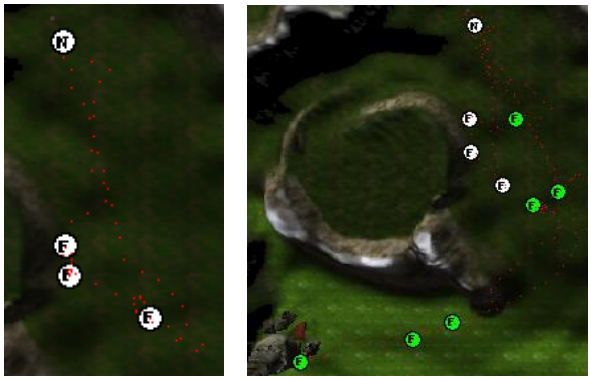

Figure 1. Two different player journeys for the same quest.

The obvious AI agent for performing this type of journey would be deterministic and optimal for speed and loot collected. It would move in straight lines, attack the closest available needed mobs, and loot at the end of each fight. That behavior is not a good

match even for the player on the left. He or she may have wasted time wandering or idling, may not have attacked the closest available mobs and may have declined to loot or looted after other fights. Some of those possibilities could be added to the agent script, such as only considering mobs in front of the character, passing by mobs already engaged with other players, or looting corpses left by other players. But if the player was confused, or trying to help out someone else, or simply wanting to explore, the script could not easily be made to account for those cases. Our goal is to automatically adapt the script based on observed data.

4. BEHAVIOR TREES

Behavior Trees are a technique for controlling video game AI agents, made popular by Bungie's *Halo* series [11]. Procedural behaviors are composed into trees using non-leaf composition nodes that explicitly specify traversal semantics. Every tree is itself a behavior, composed of sub-behaviors. The key advantage of this, from a game AI point of view, is that non-programmers can utilize the explicit semantics of each behavior, shown in a convenient graphical format, to compose new behaviors out of existing ones. From a research point of view, this composition of sub-trees represents structured knowledge about the decision process being modeled. Even though the leaf behaviors are procedural black boxes, the decision structure is entirely declarative and visible. Behavior Trees are typically limited to a small (e.g. 3 or 4) set of well-defined composition nodes, and there are no hidden transitions between behaviors [4].

For this experiment, we created two deterministic Behavior Trees, *Btree1* and *Btree2*, that model behavior for the combat/collection part of quest fulfillment journeys such as seen in Figure 1. We will use these as examples in this description. Each Behavior Tree-controlled agent in the simulation has its own tree instance that is recursively updated from the root with every discrete time step. Every sub-tree returns *Success*, *Failure* or *Running* on update. Figure 3 shows our *Advance* sub-tree, which makes an agent move to stay in range of a target entity. The target entity must already be set as a *control variable* for that tree. Figure 2 gives the legend of node types that applies to all the Behavior Tree Figures in this paper. The root of the Advance tree is a *Sequence Selector*, which updates its children sequentially. It is set to *Quit on Success*, meaning that it will continue updating until one child returns Success or all return Failure. When a child returns Running, the Sequence Selector pauses at that child and also returns Running. On the next time step, it either restarts from the first child or from the last Running child, depending on the *Restart* parameter. The leaf nodes in the tree are procedural behaviors, divided into three classes: *Action*, *Check* and *Set Control Var*. Actions cause the agent to perform actions. Checks access the game state and return Success or Failure. Set Control Var nodes assign values to one or more control variables based on other control variables, and return Success unless required values are missing.

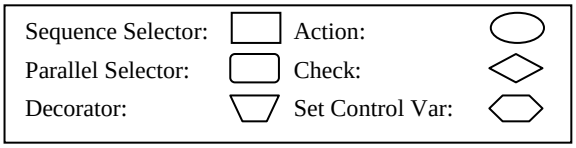

Figure 2. Legend for nodes in Behavior Tree Figures.

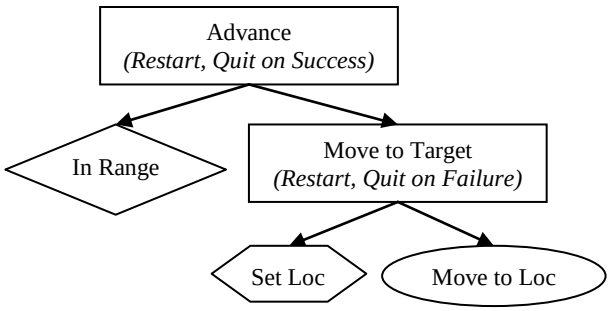

Figure 3. Sub-tree for *Advance* behavior.

When the Advance tree is updated, it first checks to see if the target is already in range by updating the *In Range* node. If the In Range node returns Success, then the Advance node also returns Success. Otherwise, the *Move to Target* tree is updated. Move to Target is also a Sequence Selector, but is set to *Quit on Failure*, failing as soon as one of its children does and only succeeding if they both do. The *Set Loc* node sets the *location* control variable to the current location of the target. If the location cannot be set for some reason, the node returns Failure, causing Move to Target and Advance to also return Failure. The *Move to Loc* node causes the agent to step towards the location control var. If the agent does not arrive at the location in that step, Move to Loc returns Running, and so do Move to Target and Advance. On the next time step, In Range and Set Loc will be re-run due to the Restart settings, making the tree properly reactive. When Move to Loc returns Success or Failure, the whole Advance tree does as well.

There are different popular definitions and terminologies for Selectors, which we group into Sequence and *Parallel*. Parallel Selectors always update all their children, and are parameterized by how their return value is determined: *Success on All*, *Success on 1*, *Failure on All* or *Failure on 1*. The other type of non-leaf node is the *Decorator*. Decorator nodes are inserted between a parent and child and can control whether that child is updated and/or modify its return value. For example, a *Continue* Decorator converts Success to Running, allowing a child tree to be run repeatedly without modifying the parent. An *Optional* Decorator converts Failure to Success, allowing a child to be run to completion, ignoring the outcome.

The full structure of Btree1 and Btree2 are shown in Figure 4. Even considering mostly optimal behavior for the limited task of combat and collection, there are significant decisions to be made, as shown by the structure of the trees. The major difference between the two trees is in how targets are acquired. In Btree1, the closest target is acquired first, whether it is an entity to attack or a corpse to loot, then the tree branches based on that target. In Btree2, combat is always preferred and targets are acquired after that decision is made. The trees were created by research team members and show how substantially structurally different trees may be created for the same agent capabilities and task.

Our Learning Behavior Trees require two additional declarative annotations to the leaf nodes. Each is annotated with the control variables it uses as input and output, and the names of the actions it can cause the agent to perform. These actions match the actions that are logged by the game engine to support recording and playback of player traces. Both of these annotations are minor tasks for the author, and clearly define the impact that a leaf node can have on the world: it can produce an action or not, and it can

mutate its output control variables or not. With simple recursive functions, the possible actions, control variables used and control variables set by any sub-tree can be generated.

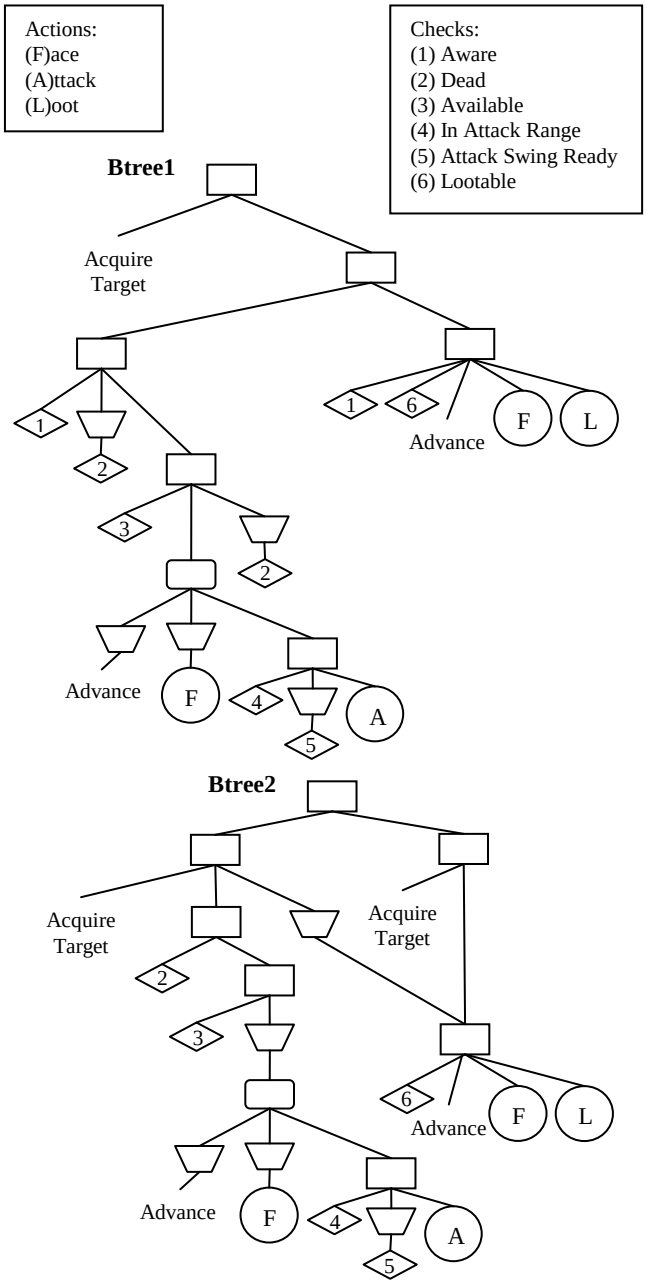

Figure 4. Behavior Trees for combat/collection quest fulfillment. The Advance sub-tree is shown in Figure 3 and the Acquire Target sub-tree is partially shown in Figure 8.

5. LEARNING BEHAVIOR TREES

5.1 Adapting from Player Traces

Hierarchical machine learning approaches to modeling player behavior have used procedural decision process models, created by the investigators, to contextualize learning (cf. [13], [15]). We apply this approach to Behavior Trees, which have already been proven as a way for game designers to formalize their intuitions about desired behavior. Starting with a deterministic Behavior Tree, our goal is to automatically adapt it to cover a range of observed human behaviors. Our method does this by inserting Decorator nodes called *modifiers* that provide certain stochastic interventions that give the tree the desired coverage. The modifiers also store positive examples of when they intervene, enabling later training. Because they are Decorators, inserting these modifiers makes minimal change to the structure of the authored tree, maintaining a high degree of its readability. This means that this method cannot create new structure, but is limited to variations in behavior that come from altering the update traversal and control variables.

To adapt a deterministic Behavior Tree, we run an agent controlled by the tree in the game, in sync with playback of a human player trace. The agent is updated at each time step until the player trace indicates an observable action other than movement. If the player action was not matched by the agent, then the system attempts to adapt the tree. For example, after killing an enemy, Btree1 always has the agent stop to loot the corpse. If a player chooses not to loot that particular corpse, then the player's next action will not match what the agent did, and the tree will be adapted to cover the case where players make that choice. This may involve inserting a new modifier, or adding another positive case to an existing one. During adaptation, the modifiers that are in place intervene deterministically based on the cases they cover in that player trace. This is so that further necessary modifications can be detected. Adaptation is complete for a player trace when all actions taken by the player either are predicted by the agent, or cannot be explained by any available modifier. As the tree is run against each available player trace, more modifiers are inserted, and the ones in place collect additional training samples. This method does not yet attempt to cover player movement variations, such as running in a circle prior to attacking. Movement, unlike the other actions, is not what we refer to as a *direct effect action*, where it is possible to map from the observed action (e.g. attacking) back to control variables (e.g. target). If a player is observed moving and stopping at point A, it cannot be assumed that point A was the intended destination. This creates a number of additional challenges which are out of the scope of this study. We have other work on mimicking human-like movement [21], which has not been integrated.

The first and simplest modification is introducing delays into the decision-making process. Human players do not react within a single frame to new environmental information, as a naively written game AI would. But learning appropriate delays is not simply a matter of saving the designer from coming up with a global distribution of random durations. Delays are dependent on a wide variety of contextual factors, both in-game and out. The structure of the Behavior Tree provides some of that context: a delay after moving into engagement range is different than a delay after completing a fight. When the agent predicts the correct action for the player trace playback, but at an earlier time, a *Delay* modifier is inserted above the leaf node that generated that action. The samples for a Delay are the game state and the duration of the delay. When that segment of the trace is replayed with that modification, the Delay returns Running instead of updating its child, for the exact duration in that sample. This causes the agent to match the player behavior, unless the Delay alone does not explain what the player did. In that case, further modification is explored. Post-adaptation, the duration samples could be used to generate, for example, a Gaussian distribution for delays at that point in the decision process.

```
For each step (player_action, delta) in the player trace:
    Update( agent, environment, delta ) => agent_actions
    If player_action matches first agent_actions:
        Synchronize agent, environment with end of step
        Continue from top            // success!
    If first agent_actions is player_action delayed:
        Modify( behavior_tree, Delay )
        Rewind agent, environment to start of step
        Continue from top            // retry to verify
    Otherwise:
        Rewind agent, environment to start of step
        If Explain( player_action, behavior_tree )
            //behavior_tree modified with possible explanation
            Continue from top      // retry to verify
    // cannot explain player_action
    Synchronize agent, environment with end of step
    Continue from top
```

Figure 5. Pseudo-code for the Behavior Tree adaptation algorithm

Figure 5 shows the outer loop of the adaptation algorithm, which attempts to explain each player action in the trace one by one. The Delay modification is so common and straightforward that we included it at this level. Whenever any modification is made to the tree, including Delay, the agent and environment are rewound back to the start of that step and re-run to verify that the modification was successful in predicting the player action. If a Delay is not sufficient, the algorithm will call the more complex *Explain* algorithm and see if it returns a candidate modification. If not, or the available modifications all fail, then that step in the trace cannot be explained and the system moves on to the next one. One minor detail not shown in Figure 5 is that the system can continue to step the playback forward past the first player action if the agent has not yet predicted an action.

The Explain algorithm, shown in Figures 6 and 7, begins with an initial environment and agent state that is known to fail to predict the next player action in the trace. It runs a regression starting with the lowest sub-trees that are capable of performing the actions performed by the player. Each tree has a set of *consumed* control variables, which are used but not set within the tree. If a sub-tree is given the correct values for those consumed control variables, inferred from the next player action, then when it is run by itself, it will either correctly predict that action, or it cannot explain it and is a dead-end. If it does predict it, then it must be the case that the original tree fails because either that sub-tree is not run at the right time, or it has the wrong control variables at that time. By regression up through the tree, our algorithm discovers the most specific node at which failure must be explained. We have developed four general-purpose modifiers to explain those failures. Importantly, these modifiers rely only on the structure of the tree and the simple control variable and action annotations discussed above. They do not require any other knowledge of the specific behaviors being used, which is critical for generality.

```
Explain( player_action, behavior_tree ):
    For each lowest sub_tree in behavior_tree
            where sub_tree can perform player_action:
        If Regress( player_action, sub_tree )
            Return True
    Return False
```

Figure 6. Pseudo-code for the outer loop of the explain algorithm.

```
Regress( player_action, sub_tree ):
    Consumed cvars from sub_tree => cvars
    If cannot infer cvars values from player_action:
        Return False
    Set sub_tree as root behavior tree in agent
    Step to (player_action, delta) in the player trace again
    Update( agent, environment, delta ) => agent_actions
    If player_action matches first agent_actions:
        If no parent to this sub_tree:
            Return True   // reached the root, success!
        Otherwise:
            Return Regress( player_action, parent )
    Otherwise:
            If Modify( sub_tree, MODIFIERS )
                // sub_tree was modified, re-try to verify
                Rewind agent, environment to start of step
                Return Regress( player_action, sub_tree )
    Return False
```

Figure 7. Pseudo-code for the explain algorithm's depth-first recursive regression.

The regression is a depth-first recursion that begins at a Selector sub-tree and moves up to the root. At each level, the Selector node being focused on has a child sub-tree which was successful in predicting the player action when run by itself (otherwise the regression would have stopped). We will refer to this as the *preferred child*. When the focused Selector is run, its updates can be broken up into non-overlapping temporal segments where either the preferred child was being updated, or it was being blocked by another child that was being updated. Our method is able to identify the blocking child due to the known set of Selector traversal options. Note that blocking can only occur with a Sequence Selector, as a Parallel always runs all its children. The modification algorithm considers the first of those segments that generates an agent action, or the first to overlap the player action we are attempting to predict. It then works backwards in time from there, attempting to find an applicable modification. When a modification is found, it rewinds the environment and agent state

and calls itself with the modified tree to test it. The standard depth-first search ensures that all options can be tried, but the first working one is taken.

5.2 Modifiers

The *IgnoreCondition* modifier applies when a *conditional child* is blocking the preferred child. A conditional child is defined as any child sub-tree that is not capable of performing any actions or *producing* any control variable values (setting values that it does not use internally). When a conditional child is blocking the preferred child, it is possible that the condition represented by that child is unimportant to the player's decision and should be ignored in that case. In Btree1, the designer indicated that players do not attack entities unless they are available (not fighting another player). Faced with an exception to that rule, our algorithm uses the tree structure to identify the available check as a candidate to be ignored. When an IgnoreCondition modifier is updated, it determines whether the status of its child has changed, and possibly ignores it until it changes back. The training sample includes the particular status to be ignored together with the game state.

The *RestartCaller* modifier applies when an *action child* is blocking the preferred child. An action child is defined as any sub-tree that can cause the agent to act on the environment. If the blocking child comes after the preferred child in the parent Selector, and that Selector is not set to Restart, it is possible that a restart of the Selector would represent the player interrupting what they are doing to consider doing something different. When a RestartCaller modifier is updated, it determines whether its child is Running, and possibly signals for a restart from its parent rather than updating its child. The training sample includes only the game state.

The *SetPreference* modifier applies when the preferred child is Running, but with the wrong control variable values. This situation is identified when a control variable is consumed by the preferred child, but not consumed by the parent Selector, indicating that it is produced by one of the other children. To apply this modification, the system identifies the children that can produce that control variable, and must see if they could have chosen the desired value. However, the logic of choosing is hidden in the SetControlVar nodes, and the algorithm does not have access to it. In order to automatically adapt, that logic has to be made explicit and declarative in the tree structure. We make this possible for the designer (who may or may not choose to) by providing two special classes of SetControlVar: *SetFromList* and *Filter*. SetFromList has a single input control variable which holds a list of values, and sets a single output control variable to one of those values. Filter takes in a list and outputs a subset of that list. Figure 8 shows part of the Acquire Target sub-tree using Set, Filter and Scan to explicitly generate and select from a list of potential targets. The actual choosing procedures (e.g. Alive, Needed, Available) are still in the designers control, but now our system can use the explicit structure to identify the point at which the desired value was available but not chosen.

For example, if the player targeted an unneeded entity, the system would detect that the desired target entity was available as input to the Filter:Needed node, but not beyond. Likewise, if the player targeted the second closest entity rather than the closest, the system would detect that the desired target entity was available as input to the SetFromList:First node. Once a node is identified that could set the desired value, it is decorated with a SetPreference that samples the game state, the input values and the output value. In the Filter case, it simply deactivates the Filter, letting all the values through. In the SetFromList case, is stores the input list and desired output as training samples.

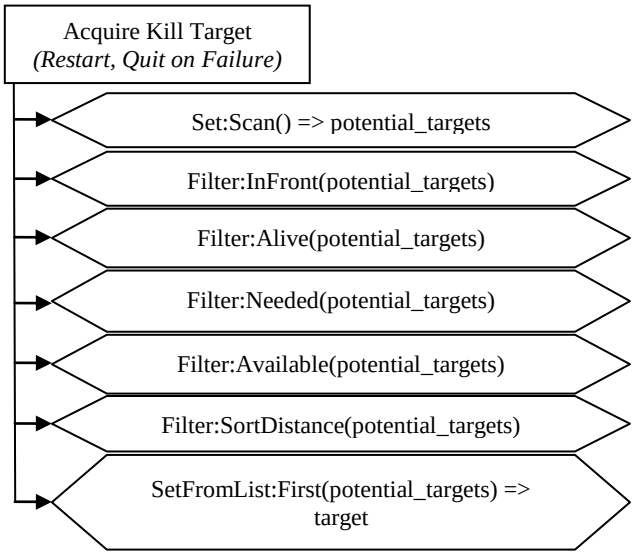

Figure 8. Sub-tree for *Acquire Kill Target* behavior.

The *Suppress* modifier is applied when the preferred child is running with wrong control variable values, but choosing the desired values cannot be explained. Instead, the preferred child sub-tree is decorated with a Suppress, which stores those wrong control variable values and does not update its child until the relevant control variables change. This forces the tree to go forward in another branch, exploring other possibilities that may explain the player behavior. Suppress captures the fact that even when all conditions are met, the player may simply choose not to pursue an otherwise appealing course at a certain time.

6. EVALUATION

In this phase of the project, we are evaluating the ability of this algorithm to adapt a deterministic Behavior Tree to cover a set of human player traces. We are concerned with generality over different players, different encounters and different Behavior Trees. For this evaluation, we gathered data from 25 human players playing a single session together in a laboratory setting. The experimental map was divided into two separate areas with similar but different topologies and tasks to perform. 15 of the players completed the quests in the A area of the map (*Data Set A*) while 10 others completed the quests in the B area (*Data Set B*). This evaluation uses the first combat-oriented journey for each player. These journeys ranged from 8 to 29 player actions (average 13), involving 3 to 10 different fights each.

The system was developed using 3 traces randomly selected from Data Set A. The other 22 traces were set aside. Several variations were noted in the 3 development traces, including ignoring available entities, not looting kills or looting other corpses, attacking already engaged entities, wandering off to other areas of the map and fighting entities there, and going back to talk to the NPCs halfway through the journey. We stopped at 3 because the four modifications we had developed (plus Delay) had already shown a great deal of robustness to unseen differences. We

hypothesized that those modifications would be sufficient to cover the majority of player behaviors observed in all the traces.

We used the two Behavior Trees created by the authors, Btree1 and Btree2, which have different decision structure over the same agent functionality. Btree1 was created due to perceived flaws in Btree2, so we hypothesized that Btree2 would have more unexplained discrepancies and be less adaptable than Btree1.

We ran four experimental conditions in this evaluation. In each condition, the full adaptation algorithm was compared against a baseline of the adaptation algorithm using only the Delay modification. In all cases, the number of unexplainable player actions was recorded per trace. The four conditions are the Data Set A traces and the Data Set B traces, each run for Btree1 and Btree2.

Table 1. Mean and stddev for percentage of unexplained actions in each human player trace.

		Area A	Area B	**A + B**
Tree A	Delay	0.71±0.3	0.81±0.46	**0.75±0.37**
	Full	0.19±0.26	0.13±0.15	**0.17±0.22**
Tree B	Delay	0.67±0.21	0.79±0.4	**0.72±0.3**
	Full	0.28±0.28	0.18±0.24	**0.24±0.27**

Table 1 reports the mean and standard deviation for the percentage of unexplained player actions in each human player trace for the eight conditions. It also shows the numbers for Data Set A and Data Set B combined. As shown, the Full adaptation algorithm significantly outperformed the baseline Delay-only algorithm in all conditions (student's t-test, $p<0.01$). In fact, it substantially outperforms it, showing the effectiveness of the modifications and the regression algorithm in explaining behaviors from previously unseen players.

There was no significant difference between Data Set A and Data Set B for either Behavior Tree in either the Delay (student's t-test, $p=\sim0.50$) or Full (student's t-test, $p=\sim0.52$) conditions, providing some evidence that the method generalizes across different decision spaces within the limited quest model. Although we believed during development that Btree1 was superior to Btree2 in explaining player actions, there was no significant difference between the percentage errors in the Delay condition (student's t-test, $p=0.30$). There was also no significant difference in percentage errors in the Full condition (student's t-test, $p=0.08$), providing some evidence that the method generalizes across different Behavior Trees.

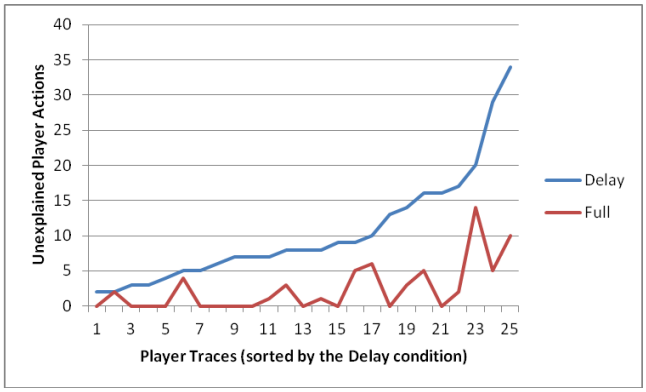

Figure 9. Number of unexplained actions for each human player trace using Btree1.

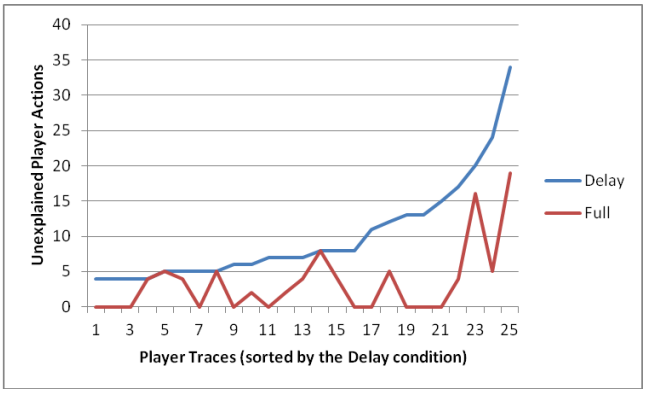

Figure 10. Number of unexplained actions for each human player trace using Btree2.

Figure 9 shows a chart of the number of unexplained player actions for each human player trace in the combined data set, working with Btree1. The player traces are sorted by the Delay condition values from least to greatest, to show the range of errors that the tree has with only the Delay modifier. The Full condition follows the same curve at a significantly lower level. Figure 10 show the same data for Btree2, at the same scale.

7. CONCLUSION AND FUTURE WORK

We have described Learning Behavior Trees, an extension of a well-known games industry technique for scripting in-game agents. Our extension maintains the advantages of Behavior Trees, namely graphical composition, easy reuse of sub-trees, simple but powerful composition semantics and the ability to use arbitrary procedural code in the leaf behaviors. Given a designer-built, deterministic Behavior Tree that expresses typical behavior, our algorithm is able to observe human players and automatically adapt the Behavior Tree to explain their choices. The resulting tree is capable of producing most of the behaviors observed, and stores contextual samples indicating the conditions for each variation. The tree is then prepared to make non-deterministic decisions based on those samples, resulting in a varied population of agents.

There are many machine learning approaches that may be appropriate to finding correlations and generating predictions using those contextual training samples. Due to the small numbers of samples in our test data, we have experimented with using simple Naïve Bayes and Inverse Transform Sampling to make those choices. The primary difficultly, besides scale, lies in evaluating a population of tree-controlled agents. The goal is not to reproduce a certain human behavior, but rather to show similarity to what a population of human players might be expected to do. This has proven quite difficult as even the straightforward but time consuming solution of having humans judge them is difficult when dealing with a diverse population. Is it humanlike for one agent over there to aimlessly run in circles jumping? In fact many players do this quite often. We believe that a more fully-functional agent architecture and game will be required to make such an experiment productive.

Part of that challenge is to integrate this work with our work on mimicking human-like movement in the same environment [21]. Our agents need the movement component in place in order to simulate entire sessions so that we can collect population-wide metrics for evaluation, and have agents run simultaneously with human players. One significant next step is figuring out how to

model and generate human-like movement target locations that may not be at all connected to task fulfillment.

The regression approach that we are using breaks the Behavior Tree down into very manageable parts, and the limited nature of the Selector variations suggests that perhaps the higher-level tree structure could be fully automatically learned given a set of independent low-level behavior trees. This would seem to invite a combinatorial explosion of possibilities, but what we have seen in this project is that actually running the trees in the simulation environment provides considerable constraint.

One limitation of this approach is that it requires that the game be instrumented for recording fairly fine-grained player actions, and playing back traces of game play. While this is a substantial undertaking, it is also necessary for big data analytics, sharing game play sessions and debugging complex player interactions. These are all becoming more and more required capabilities in a world where even single-player games have large-scale, real-time online components to build and maintain community.

8. ACKNOWLEDGEMENTS

This project was supported in part by the UTPA Center of Excellence in STEM Education Undergraduate Research Program, Office of Naval Research contract W911NF-11-1-0150.

9. REFERENCES

[1] Bakkes, S., Spronck, P., and van Lankveld, G. (2012). Player Behavioral Modeling for Video Games. *Entertainment Computing*, Vol. 3, Nr. 3, pp. 71–79.

[2] Bainbridge, W.S. (2007). The Scientific Research Potential of Virtual Worlds. *Science.* Vol. 317 no. 5837 pp. 472-476.

[3] Best, B., Lebiere, C., & Scarpinatto, C. (2002). A model of synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture. In Proceedings of the Eleventh Conference on Computer Generated Forces and Behavior Representation. Orlando, FL.

[4] Champandard, A. J. (2008). Getting started with decision making and control systems. *AI Game Programming Wisdom 4*. Boston, Massachusetts: Course Technology. 257-263.

[5] Dickey, M.D. (2005). Three-dimensional virtual worlds and distance learning: two case studies of Active Worlds as a medium for distance education. *British Journal of Educational Technology.* Volume 36, Issue 3, pages 439–451.

[6] Galway, L., Charles, D. and Black, M. (2008). Machine learning in digital games: a survey. *Artificial Intelligence Review*. Volume 29, Number 2, 123-161.

[7] Gamez, D., Fountas, Z., Fidjeland, A.K. (2013). A neurally controlled computer game avatar with humanlike behavior. *IEEE Transactions on Computational Intelligence in Games*, vol 5, no 1, March 2013.

[8] Geisler, B. (2004). Integrated machine learning for behavior modeling in video games. In: Fu D, Henke S, Orkin J (eds) *Challenges in game artificial intelligence: papers from the 2004 AAAI workshop*. AAAI Press, Menlo Park, pp 54–62.

[9] Gorski, N. and Laird, J. (2006). Experiments in Transfer Across Multiple Learning Mechanisms. In ICML Workshop on Structural Knowledge Transfer for Machine Learning.

[10] Graham, R. (2005). Realistic Agent Movement in Dynamic Game Environments. DiGRA 2005: Changing Views: Worlds in Play.

[11] Isla, D. (2005). Handling complexity in the Halo 2 AI. In *Proceedings of the GDC 2005.* Gamasutra.

[12] Karpov, I.V., Schrum, J., Miikkulainen, R. (2012). Believable Bot Navigation via Playback of Human Traces. In Philip F. Hingston, ed., *Believable Bots*, 151-170. Springer Berlin Heidelberg.

[13] Marthi, B., Russell, S., Latham, D. and Guestrin, C. (2005). Concurrent Hierarchical Reinforcement Learning. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, Edinburgh, Scotland, UK, July 30-August 5.

[14] Orkin, J. (2006). Three states and a plan: the AI of FEAR. In *Game Developers Conference* (Vol. 2006, p. 4).

[15] Schrum, J., Karpov, I.V. and Miikkulain, R. (2012). Humanlike Combat Behavior via Multiobjective Neuroevolution. In Philip F. Hingston, editors, *Believable Bots*, 119--150, 2012. Springer Berlin Heidelberg.

[17] Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. (2006). Adaptive Game AI with Dynamic Scripting. *Machine Learning*, Vol. 63, No. 3, pp. 217-248. (Springer DOI: 10.1007/s10994-006-6205-6)

[18] Thurau, C., Bauckhage, C., Sagerer, G. (2003). Combining self-organizing maps and multilayer perceptrons to learn bot-behavior for a commercial game. In Mehdi Q, Gough N, Natkin S (eds) *Proceedings of the 4th international conference on intelligent games and simulation*. Eurosis, pp 119–123.

[19] Togelius, J., Burrow, P. and Lucas, S.M. (2007). Multi-population competitive co-evolution of car racing controllers. *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 4043-4050.

[20] Togelius, J., Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. (2011). "Search-based procedural content generation: A taxonomy and survey." *Computational Intelligence and AI in Games, IEEE Transactions on* 3, no. 3: 172-186.

[21] Tomai, E., Salazar, R. and Flores, R. (2013). Mimicking Human-like Agent Movement in Open World Games with Path-Relative Recursive Splines. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

[22] Wray, R., Laird, J., Nuxoll, A., Stokes, D. and Kerfoot, A. (2005). Synthetic Adversaries for Urban Combat Training. *AI Magazine*, Volume 26, Number 3.