Procedural Content Generation: Evolving Non-Player Character (NPC) Intelligence

Andrew Lam
lam00139@morris.umn.edu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

Modern game developers face challenges in manually producing responsive non-player character (NPC) behaviors. Procedural Content Generation (PCG) offers a promising approach to reducing the manual labor of scripting complex game behaviors by automating design space exploration while preserving designer control. This paper evaluates an evolutionary PCG method for NPC behavior: *EvolvingBehavior*, which uses genetic programming to evolve behavior trees. This method supports designers through co-creation to produce NPC behaviors that are effective, interpretable, and aligned with the designer's vision.

Keywords: procedural content generation, non-player character behavior, evolving behavior tree, genetic programming

1 Introduction

The process of producing an interactive and believable non-player character (*NPC*), an autonomous game entity controlled by AI rather than the player, has long been considered and agreed as a time-consuming task in modern game development by [3, 11]. Traditionally, developers spend substantial effort manually scripting behaviors and response patterns for each character, which limits their ability to scale and be creative. To address these limitations, Procedural Content Generation (PCG) becomes a promising option for developers who want to take this burden off their hands by automating repetitive parts of the design process while still leaving space for developers to guide and shape the outcomes toward their intended goals.

To explore ways to address this problem, this paper describes an approach to PCG of NPC intelligence through an evolutionary approach: genetic-programming-based evolution of behavior trees (*EvolvingBehavior*). This approach combines behavior trees with genetic programming, a tree-structured program-synthesis method that naturally matches behavior tree structure and has shown promising results on simple 2D games [7]. This method offers the ability for co-creative design by evolving interpretable behavior structures that align with designer intent.

Focusing on EvolvingBehavior, which applies genetic programming to evolve behavior trees under designer-defined fitness, we examine when AI serves as a co-creative partner rather than a substitute for human design. Specifically, this paper introduces the necessary background concepts and explains the motivation behind the method, followed by describing the tools, platforms, and evaluation setup. After that, this paper details the training procedure and reports quantitative fitness-based results comparing evolved, random, and manually authored behavior trees across a range of map and density conditions. The paper concludes with a discussion of the system's limitations and a summary.

2 Background and related concepts

2.1 Procedural content generation in game development

Procedural Content Generation (PCG) refers to the use of algorithms to produce elements in a game, such as levels, worlds, rules, or character behaviors, with minimal human input. The process methods may be purely rule-based or AI-driven. As Smith explains in his previous research, the connection between AI techniques and design goals made by PCG helps reduce the burden of manual work while shaping unique player experiences [10].

2.2 Behavior trees for game AI

Behavior trees (BTs) are a widely used architecture for NPC decision making in modern games [4]. Damian Isla's work on the *Halo 2* AI system is widely cited as an early large-scale commercial use of BT-style control [4]. Behavior trees were introduced as the solution for managing increasingly complex game logic while keeping the designer in control.

A behavior tree is organized as a rooted hierarchy in which the root dispatches to composite nodes (selectors and sequences) and eventually to leaf nodes (as shown in Figure 1). A selector evaluates its children from left to right (from highest to lowest priority) and returns as soon as one child succeeds; if all children fail, the selector itself fails. A sequence executes its children in order and fails upon the first failure; it succeeds only if all children succeed. Leaf nodes implement actions (e.g., move, attack) and simple condition checks (e.g., IsEnemyClose?, SeeEnemyInSight?). Optional decorators (single-child modifiers) can guard or alter a child's execution, as described in [1].

In Figure 1, the root is a selector that prioritizes the left-most sequence. We illustrate two common ways to place a condition. The first way sets conditions as a step inside the sequence. In this pattern, the sequence performs one or more actions, then evaluates the condition, and if the condition fails, the sequence fails at that point. The second approach uses conditions as a decorator that wraps the entire sequence. Here, the condition is checked before any actions run, and if it is false, the branch fails immediately, and the selector proceeds to the next option. Using these patterns, characters gain clearer decision-making structures with easier debugging and future expansion, while allowing developers to build more complex and reactive behaviors from simple building blocks.

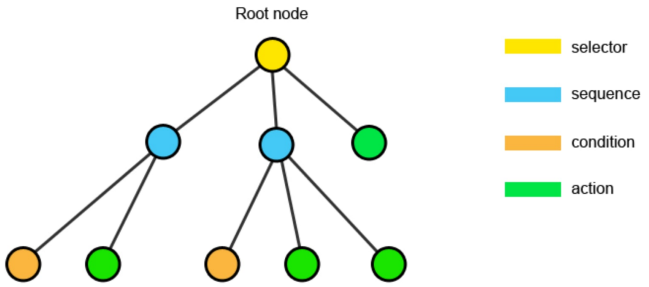


Figure 1. Simplified behavior tree showing selector (yellow), sequence (blue), condition (orange), and action (green) nodes [2].

Building on this, Champandard and Dunstan formalized BT design principles in *The Behavior Tree Starter Kit*, presenting a standard framework for structuring game AI [1]. Their approach aims to improve the behavior tree to be more readable, easier to scale, and simpler to reuse—qualities that turned the behavior tree into a standard across both commercial and academic game development.

2.3 Evolutionary computation

Evolutionary computation (EC) is a trial and feedback search: generate controller variants, score them, keep and recombine the best, mutate to explore, and iterate. This work will focus on the use of genetic programming (GP) to evolve the structure of behavior trees. Figure 2 illustrates the EC loop cycle. The process begins with an initialized population of candidate controllers, which are evaluated using multiple test metrics to obtain fitness scores. Candidates with higher fitness are selected as parents and used to produce offspring through reproduction. A mutation step is then applied to introduce new variation. The resulting offspring form a new generation that replaces the previous one, and the cycle repeats. In the EvolvingBehavior experiments, each candidate is a BT while reproduction uses subtree crossover (swap subtrees), and mutation makes small edits (add/replace/delete nodes), producing new behavior trees for the next generation. Further information on fitness function settings used in the experiments will be introduced in Section 4.1.

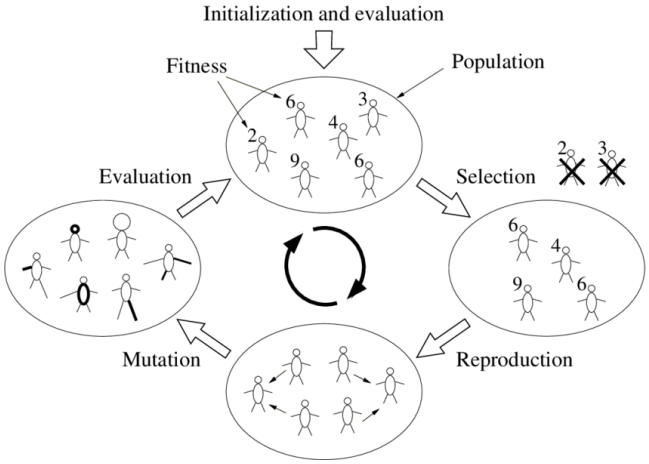


Figure 2. Illustration of the evolutionary computation cycle [13].

2.4 NPC decision-making methods

Non-player characters (NPCs) in the game need to continuously make decisions on what to do, choosing between a variety of actions such as movement, reactions, and strategy based on the current game state. Established common approaches consist of finite state machines, rule-based systems, goal-oriented action planning, tree-based control, and artificial neural networks [12]. Many games use a hybrid method combining these ideas together, but this paper will concentrate on tree-based methods, specifically Behavior Trees, offering a clear, hierarchical way to sequence conditions and actions.

3 EvolvingBehavior: a co-creative tool for behavior tree evolution

3.1 The reason behind EvolvingBehavior

EvolvingBehavior acts as a co-creative tool, which allows designers to evolve NPC behavior trees (BTs) using the Unreal Engine 4 platform with the help of genetic programming. The ultimate goal is a co-creative workflow where the designer supplies building blocks, such as BT nodes or templates, and a fitness score definition of a good behavior. The tool then continues to test and refine trees until they meet the specific designer vision in a real game context.

3.2 System setup and process

The designers took authority over supplying materials such as "initial tree, library of additional pre-defined nodes, templates, controls, other parameters and settings for mutation of the evolutionary process" [7], that the genetic programming algorithm needs to build a modified tree. With this setup, the initial tree is said to be either complete, incomplete, or empty since the system allows mutation and will expand it over time.

The EvolvingBehavior tool converts the behavior trees into an editable chromosome, which is a compact and manipulable tree data structure. The chromosome stores mapped nodes, which are direct references to Unreal BT nodes (tracked by unique IDs), and generated nodes, which are templates with properties the designer bounds (integers, floats, booleans, and blackboard values). This representation allows chromosomes to be later copied, stored, and modified through mutation operators, including changes to node properties. At the end of the evolution step, these chromosomes are then translated back into properties of Unreal behavior tree nodes via reflection.

With the definition of good behavior differing by game and purpose, this system offers developers "an event-driven architecture for defining their own" [7], rather than a set of prescribed fitness functions. The designers have room to define their own key for each measure, and events will then be sent to update the score of each agent, allowing space for flexibility. By default, at the end of the simulation, "a linear combination of the fitness measures with designer-defined weights" [7] will be computed. However, this combining rule can be swapped in with a different one if needed. Although the definition of good behavior for each game differs, most games share the same important aspect of fitness, which is "negative modifier for the complexity of the tree" [7]. To discourage too simple and over-complicated trees, a penalty using the tree size is included to control "bloat"—growth of unnecessarily complex, unused structures in the genetic program over many generations that could result in a less human-readable output [9].

After the preparation of fitness measurements, parents are picked through tournament selection. In each tournament, the system randomly samples k individuals from the current population and selects the one with the highest fitness as a parent. All candidates remain in the population and can be sampled again in later tournaments. For the experiment performed by the author of EvolvingBehavior, *k is set to 4* to balance the selective pressure and diversity [7]. At the end of each generation, the new offspring replace the entire population, except for a small elite: the top $n\%$ individuals that stand out in the prior generation are copied over unmodified to maintain strong solutions and limit bloat.

Once the system selects pairs of parents, each pair produces a child for the next generation. The child starts as an exact copy of the primary parent, but then undergoes the process of crossover and one or more points of mutation. Unlike other genetic programming systems, which only allow at most one mutation type per child, it is said that there is often a significant probability of performing no mutation at all [8]. EvolvingBehavior provides the ability for multiple mutation types to be applied, with a separate probability for each, giving designers finer control over rates [7].

During crossover, a randomly chosen subtree in the child is swapped with a subtree from the second parent. Subtrees are chosen by uniformly sampling a tree depth, which tends to exchange larger, behaviorally meaningful parts of the tree rather than mostly leaves. For point mutations, the system could either replace, add, or delete a node to introduce small random changes. In the EvolvingBehavior experiments, numeric properties of generated nodes are perturbed with Gaussian noise using a standard deviation of 10% of the current value [7], and boolean and blackboard properties are re-sampled from their valid options.

The process represented in Figure 3 acts as a full workflow summarizing the loop. The left column represents the game, where the engine loads a map, spawns agents (individuals), runs a trial, and records fitness events. The right column represents the EvolvingBehavior Plugin, which first encodes the current behavior trees as chromosomes, selects parents using tournament selection, applies crossover and point mutations (while keeping a portion of the population unchanged), and builds the next generation. In this process, designers provided inputs (initial tree, node library, templates, and settings) for the EvolvingBehavior Plugin, whose internal components exchange data with the running game via the connections shown by the arrows. This cycle repeats across generations until a satisfactory tree is found, and each iteration of this process corresponds to a single generation.

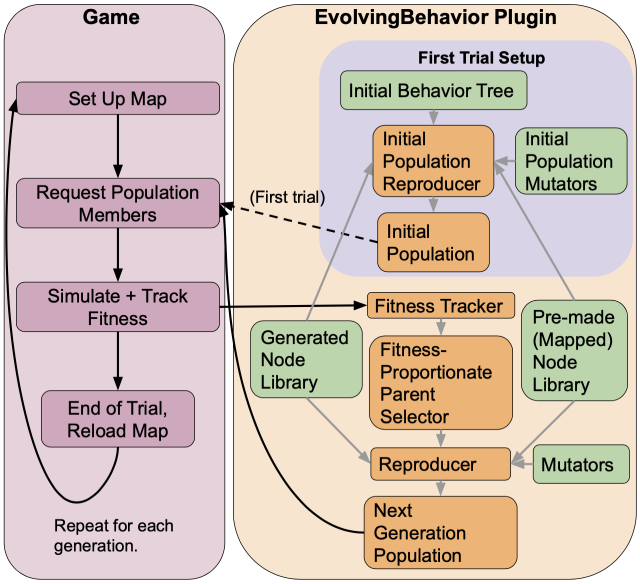


Figure 3. An overview of the process and data flow for evolving behavior trees using the EvolvingBehavior tool. [7]

4 Experiments

4.1 Methodology

Environment testbed. The EvolvingBehavior tool was tested through Tom Looman's "Epic Survival Game" sample [5] by evolving AI functionality for simple enemies NPC in the game. The game is a 3D, over-the-shoulder style standard third-person shooter in which zombie NPCs act as the enemy

of the human-controlled character. As the main objective of this game, the player must live as long as possible, enduring the population of leveled zombie-like enemies. In the base game, zombies perform actions such as patrolling, perceiving the player via sound or vision, chasing, and inflicting damage. The player can kill and do damage to zombies if they do enough damage, while only having 100 health and slowly losing health when an enemy is nearby. After each death, the player will respawn after 2 seconds at a random position. During the experiment, perception is not handled by the behavior tree. Instead, this experiment uses an Unreal's perception system to update a shared blackboard, a key–value store that holds AI state (e.g., the nearest detected player). The zombie behavior tree reads this blackboard entry and only determines which actions to execute.

Player Stand-ins and World Variants setup. A simple "human character" behavior tree is created to replace the human player, with a simple behavior of selecting a nearby point to move toward approximately every half a second. A priority is set for the human character to favor paths and locations farther from zombies, visible to the character but not to zombies, and aligned with the current heading. A stamina mechanic is also introduced to enable the option for player stand-in characters to briefly sprint at chase onset, followed by slowing down and a recovery state. Three maps were used, including a small enclosed map with dense obstacles and a large open outdoor map with sparse obstacles. These two maps were provided by the Epic Survival Game's map pool [5] and were modified to better fit the designer's intent. The third "medium map" was introduced as a duplicate version of the small map, with space and obstacles copied four times, and the walls between the copies being removed.

Initialization for Co-Creative Improvement. In this test case scenario, to better emulate the iterative improvement of a partially human-designed behavior tree, the functionality of the zombie behavior tree originally included with the "Epic Survival Game" was removed. Mostly, focusing on removing patrol and chase nodes, preserving some structure, but eliminating movement behavior.

Evolutionary operators and rates. The initial population is set to go through 10 iterations of random modification, yielding 40% probability of at least one point mutation and 40% probability of crossover. Subsequent generations used the same mutators at half intensity (approximately 20% overall), implemented as 12-point mutators at 1.84% each (20% chance that at least one fires). The elitism was set at 12% to maintain the balance of strong individuals with population diversity.

Fitness function. In the experiment, a system rewards meaningful zombie behaviors and provides penalties for zombie failure movements. Throughout the game run, the start and end of each chase is examined based on consistent movement at an angle towards and within a short distance of a player, or the lack thereof [7]. This was introduced in Nathan Partlan's research with the intention that these penalties help prevent NPCs from repeatedly chasing a player too frequently [6]. To prevent creating behavior trees that were excessively complex or too simple, a penalty is set to help lower the chances of both bloat and trivial trees that would take too long to evolve into more complex behavior. During each 90-second trial, we accumulate weighted rewards and penalties from the scoring rules shown in the Figure 4. The system then adds score for each success actions such as Movement (reward per second if agent move), Chase Start (one-time bonus when a chase begins), Chase Maintain (reward per second while the chase condition holds), Damage (event bonus per hit/HP dealt), and Search Near Last-Known (reward per second spent searching within a radius after target loss). The system would subtract from each individual's score for Failed Movement (event penalty on invalid or stuck moves) and Rapid Chase Reset (event penalty when a new chase starts shortly after the last one ended). A chase is detected when the agent's velocity persistently points toward the target within an angle and distance window, and ends after those criteria fail for a brief grace period. After the 90-second trial, the researchers apply one-off structural penalties based on tree size: Chromosome Size (anti-bloat penalty for over-large/complicated trees) and Chromosome Size Under Six (anti-trivial penalty if the tree has fewer than six nodes). Per-agent scores are averaged to yield the behavior tree's fitness, and the same fitness specification was used across all experimental conditions.

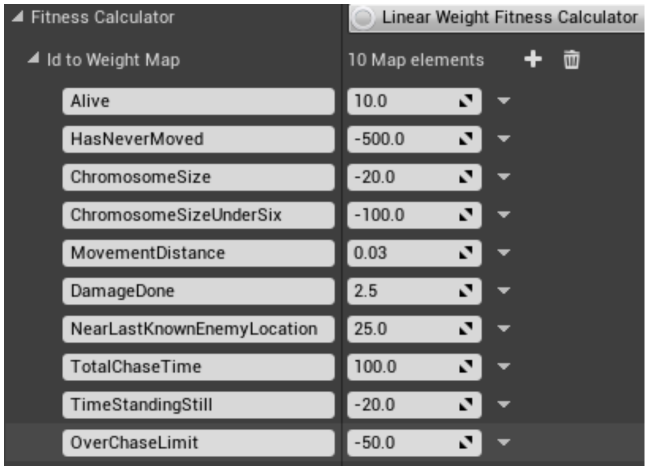


Figure 4. The identifiers and weight settings for the piece-wise linear fitness function used in the experiments. [7]

Behavior Tree Node The evolution operated on a mixed library of mapped and generated behavior tree nodes. The set included tasks from the original zombies in the "Epic Survival Game" (e.g., selecting random patrol points and moving through Unreal's pathfinding), plus variations of built-in Unreal nodes such as wait timers, decorators, and actions for rotation and angle checks, and direct movement toward a specified object or location (bypassing the full pathfinding). To

probe robustness, several intentionally less-effective nodes were retained so that evolution had to improve partially functional behavior. Additional utilities supported search behavior after target loss by finding positions near the last known player location and explicitly forgetting that location. In total, the library comprised 8 mapped and 4 generated decorators, 13 mapped and 5 generated tasks, and the standard composite types (selectors and sequences).

Experimental conditions. Agents are evolved using the three maps (Small, Medium, Large), crossed with three different counts of human stand-in characters (3, 6, and 12), and players will respawn 2 seconds after death at a random location, producing a total of nine experimental conditions. Zombie counts in the population are set as follows: Small with 20 zombies, Medium had 50 zombies, and Large had 40 zombies. As a reminder for the fitness function settings, the same fitness function is used across all experimental conditions, and these conditions offer a variety of densities and conditions to evolve zombie behavior.

Run Length and Simulation Settings. Each evolutionary run proceeded for 1,000 generations, exceeding the preliminary fitness plateau. The evaluations used 15× game speed and 90 seconds of simulated in-game time (approximately 6 seconds in real time), matching preliminary grid-search settings and balancing computational cost with gameplay fidelity.

Model selection. After each run, depending on the fitness function measurements setting, the generation with the highest average score will be identified, in which the highest-fitness individual from that generation will then be selected as the representative behavior tree.

Replications and Aggregation Evolution is run three times with different initial conditions for each experimental condition. The selected standout representative behavior tree from each run will be compared, and fitness results will be averaged across the three runs. With that being said, the small number of replications means the computational cost might be high, and the set of resulting trees needs to be kept manageable for manual inspection and qualitative evaluation across all nine conditions. This mirrors expected designer use, where frequent re-runs would slow iteration and complicate selection among behavior trees.

4.2 Comparison with Manual Design and Random Baseline

Two additional setups were included alongside evolution: firstly the manually authored behavior trees by three researchers (Researcher 1 had prior professional game-AI experience, while Researcher 2 and Researcher 3 had no significant behavior-tree experience and received a brief overview of Unreal's behavior-tree system). Secondly, a random-baseline procedure that reproduced and mutated all trees of the previous generation without fitness-based parent selection. All other operators, rates, and the final tree selection procedure matched the evolutionary setup, and all started from the same partial initial trees.

4.3 Results

Figure 5. Graph of the average fitness per generation of the evolved vs. the random baseline zombies, on the Medium Map, with 50 zombies and 6 humans. [7]

Figure 5 graph shows the average fitness by generation using the Medium map (50 zombies vs 6 human stand-ins). The Evolved runs climb rapidly in the first few dozen generations and stay steady well before 1000 generations, indicating that evolution consistently discovers effective behavior-tree policies and gives a stable result. In contrast, the Random baseline remains substantially lower and shows little upward trend, confirming that without using guidance for mutation/crossover with a proper setup of fitness-based selection, it would show no substantial improvements for NPC behavior tree. The diagram clearly shows that evolved behavior trees tend to outperform random trees while also maintaining a steady performance level under this configuration.

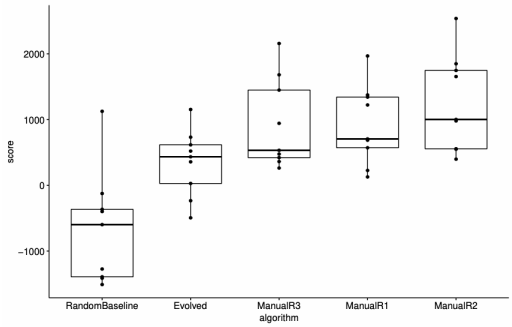


Figure 6. Box plots of the average fitness results for each algorithm across all problem configurations. [7]

Figure 6 summarizes final fitness across all map×density conditions. The Random baseline shows the lowest medians with wide interquartile ranges, indicating weak and inconsistent performance. Both Manual and Evolved distributions sit

substantially higher, with tighter boxes and fewer low outliers, demonstrating robust gains over random search. Overall medians for Evolved are close to Manual, showing that evolved behavior trees frequently approach hand-authored quality.

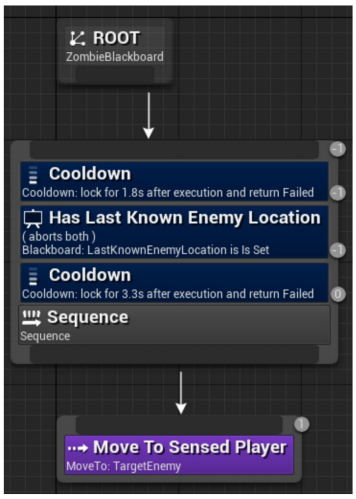
Figure 7 shows the comparison between three trees (Random0, ManualR3, Evolved2) in the Medium map 50vs3 condition. ManualR3 and Evolved2 both show a clear patrol → chase flow: patrol when no target, then pursue/attack on detection. Random0 lacks a solid patrol branch and mostly moves only when a target is sensed, leading to brittle behavior. Visually, the evolved tree's structure looks close to the manual design one, while the random tree is simpler and less organized.

5 Limitations

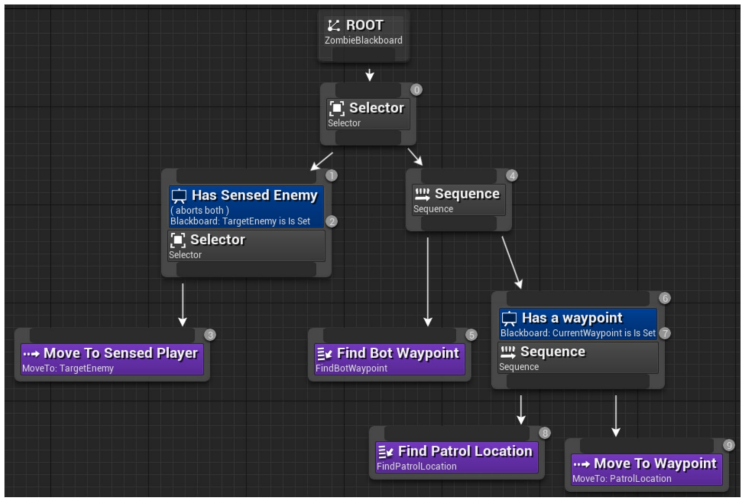
This study used the researchers themselves in the role of developers, individuals with prior knowledge of the tool, which may have biased design choices, parameter tuning, and expectations. The experimental scope was modest (three runs per nine related conditions) and based on a relatively simple sample game, limiting behavior-tree complexity and reducing the ability to establish statistical significance for some treatment differences. Usability and co-creative fit were not validated with independent designers, and key interaction tasks (e.g., defining/tuning fitness functions, configuring test environments, selecting parameters) were not user-tested. Consequently, generalizability to richer game mechanics, broader design goals, and real-world workflows remains uncertain and warrants future study.

6 Conclusion

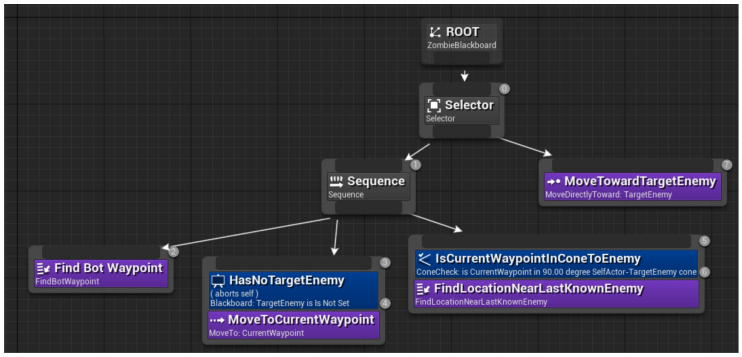
This paper examined EvolvingBehavior as an evolutionary, co-creative tool for producing NPC behavior trees in Unreal Engine 4 and found that it reliably automates the search for effective behaviors while preserving designer control and interpretability. In the study's settings, evolved trees improved rapidly, consistently exceeded a random-search baseline, and often approached hand-authored performance, demonstrating meaningful time savings without sacrificing editability or alignment with design intent. While the evaluation used a simplified game and researcher-authored setups, the results indicate that behavior tree evolution can reduce authoring effort, accelerate iteration, and serve as a practical partner for designers, providing a meaningful opportunities for advancement in game development practice. However, future work is required and should validate usability with independent designers and richer game case scenarios.



(a) The Random0 tree for Medium50v3.



(b) The ManualR3 behavior tree.



(c) The Evolved2 tree for Medium50v3.

Figure 7. Examples of behavior trees discussed in the qualitative evaluation: Random0 and Evolved2 from the Medium Map 50v3 condition, and the ManualR3 hand-designed tree.

Acknowledgments

I would like to thank Professor Elena Machkasova for her guidance throughout this course and for providing valuable feedback and comments that helped improve this paper. I also thank my advisor, Professor Kristin Lamberty, for her continuous support and insightful feedback during the research and writing process.

References

[1] Alex J. Champandard and Philip Dunstan. 2013. The Behavior Tree Starter Kit. In *Game AI Pro: Collected Wisdom of Game AI Professionals*. A. K. Peters, Ltd., Natick, MA, USA.

[2] Philipp Grünauer. 2017. Flexible game AI. https://g-phil.com/flexible-game-ai/. Blog post on G-Phil (devlog).

[3] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. 2022. A survey of Behavior Trees in robotics and AI. *Robot. Auton. Syst.* 154, C (Aug. 2022), 18 pages. doi:10.1016/j.robot.2022.104096

[4] Damian Isla. 2005. Handling Complexity in the Halo 2 AI. In *Game Developers Conference*. https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

[5] Tom Looman. 2017. *EpicSurvivalGameSeries: Third person survival game (tutorial) series for Unreal Engine 4*. https://github.com/tomlooman/EpicSurvivalGameSeries GitHub repository.

[6] Nathan Partlan, Erica Kleinman, Jim Howe, Sabbir Ahmad, Stacy Marsella, and Magy Seif El-Nasr. 2021. Design-Driven Requirements for Computationally Co-Creative Game AI Design Tools. In *Proceedings of the 16th International Conference on the Foundations of Digital Games* (Montreal, QC, Canada) *(FDG '21)*. Association for Computing Machinery, New York, NY, USA, Article 34, 12 pages. doi:10.1145/3472538.3472573

[7] Nathan Partlan, Luis Soto, Jim Howe, Sarthak Shrivastava, Magy Seif El-Nasr, and Stacy Marsella. 2022. EvolvingBehavior: Towards Co-Creative Evolution of Behavior Trees for Game NPCs. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*. Association for Computing Machinery, New York, NY, USA, Article 36, 13 pages. doi:10.1145/3555858.3555896

[8] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd.

[9] Riccardo Poli, Nicholas Freitag McPhee, and Leonardo Vanneschi. 2008. Elitism reduces bloat in genetic programming. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation* (Atlanta, GA, USA) *(GECCO '08)*. Association for Computing Machinery, New York, NY, USA, 1343–1344. doi:10.1145/1389095.1389355

[10] Gillian Smith. 2014. Understanding procedural content generation: a design-centric analysis of the role of PCG in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 917–926. doi:10.1145/2556288.2557341

[11] Veronica Torres. 2022. *A Survey on Non-Player Character Believability*. Creative Component (M.S., Computer Engineering). Iowa State University, Ames, Iowa. https://dr.lib.iastate.edu/server/api/core/bitstreams/4841c6c3-b69c-4135-827a-294879484520/content Major Professor: Joseph A. Zambreno.

[12] Muhtar Çağkan Uludağlı and Kaya Oğuz. 2023. Non-player character decision-making in computer games. *Artif. Intell. Rev.* 56, 12 (April 2023), 14159–14191. doi:10.1007/s10462-023-10491-7

[13] Rasmus K. Ursem. 2003. *Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization*. PhD dissertation. Aarhus University, Department of Computer Science, Aarhus, Denmark.