PEDAGOGICAL REPORT

Teaching Behavior Trees & AI Decision-Making in Unity

---

1. Teaching Philosophy

1.1 Target Audience

The instructional materials are designed for:

- Beginner to intermediate Unity students
- Individuals with basic C# scripting familiarity
- Game design or computer science learners exploring AI systems
- Students who have never used Behavior Trees before

This audience typically understands core Unity components (objects, components, basic scripts) but lacks exposure to structured AI decision-making architectures like Finite State Machines or Behavior Trees.

1.2 Learning Objectives

By the end of the training module, learners should be able to:

1. **Explain the purpose and structure of Behavior Trees**
    - Understand node types, evaluation flow, and states (Success, Failure, Running).
2. **Read and interpret BT diagrams**
    - Recognize Selector and Sequence logic and how they influence behavior transitions.
3. **Construct a simple Behavior Tree**
    - Specifically implement Patrol and Chase logic using modular behavior nodes.
4. **Integrate BT logic with Unity's NavMesh system**
    - Understand how decision making connects to real-time movement and navigation.

5. **Evaluate and extend the Behavior Tree**
    - Add new behaviors such as Idle, Attack, or Search based on exercises.
6. **Debug AI behavior effectively**
    - Use visual cues, gizmos, and systematic reasoning to identify errors.

1.3 Instructional Rationale

Behavior Trees were selected as the instructional AI concept because:

- They represent industry-standard AI architecture in modern games.
- They are modular and visually intuitive, making them ideal for teaching.
- They encourage systematic thinking, rather than ad-hoc scripting.
- Students gain exposure to scalable decision-making structures early in their development career.
- BTs unify multiple fields: logic, architecture, navigation, and emergent gameplay.

This topic satisfies academic goals while providing practical, industry-relevant experience.

---

2. Concept Deep Dive

2.1 Technical Structure of Behavior Trees

A Behavior Tree is a **rooted, directed tree** where each node evaluates to one of:

- **SUCCESS**
- **FAILURE**
- **RUNNING**

This evaluation propagates upward through the tree to determine the agent's final action each frame.

Formally, a Behavior Tree can be represented as:

BT = { N, E, root }

Where:

- **N**: set of nodes

- **E**: edges defining parent-child structure
- **root**: top-most node evaluated each frame

Each node defines:

f: state(t) -> {Success, Failure, Running}

Where **state(t)** represents the world state at frame *t*.

2.2 Composite Node Logic

Selector Node

A Selector node S with children $C_1 \ldots C_n$ returns:

$S(t) = \text{first}(C_i(t) \neq \text{Failure})$

If all children fail:

S(t) = Failure

This models OR logic.

Sequence Node

A Sequence node Q with children $C_1 \ldots C_n$ returns:

$Q(t) = \text{Failure} \quad \text{if any } C_i(t) = \text{Failure}$

$Q(t) = \text{Running} \quad \text{if any } C_i(t) = \text{Running}$

$Q(t) = \text{Success} \quad \text{if all } C_i(t) = \text{Success}$

This models AND logic.

2.3 Action Node Logic (Movement & Distance)

The Chase behavior uses Euclidean distance:

$d = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2]}$

Chasing is triggered if:

$d \leq \text{chaseRange}$

This provides a clear mathematical decision mechanism and connects learning to concrete formulas.

2.4 Game Design Connections

Behavior Trees provide direct benefits to game designers:

- **Predictability:** Designers know exactly when behaviors occur.
- **Modularity:** New actions plug into the BT without modifying existing logic.
- **Fallback Logic:** Allows fail-safe behavior transitions (e.g., Patrol → Chase → Patrol).
- **Scalability:** Adding new actions does not increase code complexity exponentially.

From a gameplay perspective:

- Patrol creates believable world activity.
- Detection radius encourages player strategy.
- Chase behavior adds tension and reactivity.

This establishes a fundamental understanding of how in-game characters perceive and react to the player.

---

3. Implementation Analysis

3.1 Architecture

Your implementation consists of the following structural layers:

1. Behavior Tree Framework

- Base node
- Selector composite
- Future extensibility for Sequence, Decorators, etc.

2. Concrete Behavior Nodes

- **Patrol Node**
- **Chase Node**

These adhere to single-responsibility principles and keep logic isolated.

---

3. EnemyAI Controller

- Initializes behavior tree
- Passes Unity references (waypoints, player, NavMeshAgent)
- Evaluates tree each frame

This demonstrates a clean separation of architecture (BT) and execution (AI controller).

---

4. Unity Integration

- NavMeshAgent handles all actual movement
- Patrol and chase behaviors interact minimally with Unity components
- The decision-making remains framework-agnostic

This is an ideal, scalable approach.

3.2 Performance Considerations

Your BT implementation is lightweight:

- Evaluation occurs once per frame
- Only two nodes are evaluated (Chase → Patrol fallback)
- No expensive operations such as raycasts or physics checks
- Distance calculations are constant time

This ensures:

- Smooth performance
- Negligible CPU overhead
- Easy scalability to more enemies

For larger BTs:

- Composite nodes help control branching complexity
- Decorators can gate behaviors (e.g., run every N frames)

3.3 Scalability

The design allows easy extension:

Add an Attack behavior

Just create an AttackNode and plug it into a Sequence or Selector.

Add Idle or LookAround behavior

Insert into Patrol branching.

Add Search behavior

Add fallback to Selector:

Selector:

1. Chase
2. Search last known location
3. Patrol

Add sensors (hearing, line-of-sight)

These become conditional nodes feeding into Sequences.

The architecture scales linearly.

---

4. Assessment & Effectiveness

4.1 Evaluation Criteria

A student successfully completes the module if they can:

1. Explain Behavior Tree concepts clearly
2. Interpret Selector and Sequence diagrams
3. Build a BT with Patrol and Chase behaviors
4. Integrate BT logic with Unity's NavMesh system
5. Debug issues (e.g., incorrect waypoint assignments)
6. Demonstrate responsive AI behavior in Play Mode

4.2 Expected Student Challenges

1. Misunderstanding BT flow

Students may assume behaviors "run simultaneously," not realizing Selector stops evaluating after a valid result.

2. Incorrect waypoint setup

Missing or zero-size waypoint arrays can break patrol behavior.

3. NavMesh baking issues

Common mistakes:

- Forgetting to bake
- Baking on wrong surface
- Agent height too large

4. Player starting inside chase range

This leads to no patrol → students may misdiagnose this as AI not working.

5. Confusion with node states

Misinterpreting RUNNING as SUCCESS leads to unexpected flow.

4.3 Mitigation Strategies

To improve comprehension:

- Use diagrams to illustrate Selector/Sequence flow
- Show BT evaluation frame-by-frame
- Encourage students to print or log node states
- Provide debugging tips (gizmos for chase radius)

4.4 Overall Effectiveness

This module effectively teaches AI decision structures because:

- It begins with a simple BT structure
- Provides immediate, visual feedback in the Unity scene
- Demonstrates clear cause-and-effect in player proximity
- Allows for modular extension into more advanced AI topics
- Reinforces architectural thinking in gameplay programming

Students leave with practical knowledge directly applicable to real game projects and industry workflows.