INTRODUCTION

WHAT IS AI TREE?

This plugin aims to allow you to easily create AI behavior trees within Godot. Let’s get an overview of what AI Tree can do!

1. BEHAVIOR TREE GRAPH

This is a visual programming editor that allows you to create behavior trees without writing a line of code! Place, connect, and change aspects of nodes on the graph and watch those nodes give your entities live in your game.

2. NODES

The behavior tree runs off of nodes that are connected to create the tree. These nodes can be added and connected in the graph, or they can manually be put together in your code base.
While not truly established yet, it is possible to create your custom nodes that will fit in with existing nodes. I will be working to create a base class for the graph node so that this is possible.

3. AI NODE

This is a custom node that is responsible for bridging the behavior tree graph to your game on runtime. This node is responsible for storing the information generated by the graph and for taking that information and converting it into a behavior tree usable by your entity.

GETTING STARTED

This tutorial teaches you how to install AI Tree and provides a quick overview of how to use the graph.

CONTENT

Add List of Contents

1. INSTALLATION & ACTIVATION

Discussing how to install and activate the plugin. I will need to research how to do this.

2. CREATING AN AI ENTITY

Covers creating a new Entity that the player should connect the AI to. This should include adding the AITree Node and creating a dummy script to use in the editor

3. ACCESSING THE GRAPH EDITOR

Now that we have created an AI Entity, let’s look at the editor graph. While viewing your scene, click on the new tab labeled “AI Tree” at the very top of the editor (Next to 2D, 3D, etc.)

There should now be two new screens in the middle of your screen: A panel on the left with a column of buttons and a graph on the right with a single node labeled “Root Node”.

The graph on the right is where all of the graph nodes for the tree will be placed, organized, and connected. Here we can press left-click to select and hold left-click to drag existing nodes around the graph area. Additionally, two nodes can be connected by left-clicking and dragging from one port on a node (The little white circles) to another port on a different node.

The panel on the left contains two groups of buttons: The node buttons and the utility buttons. The top group of buttons are the node buttons, and clicking those buttons will create and add a new node of the associated type to the graph. The buttons in the lower group of buttons have various uses depending on the button. The save button is currently the only utility button present and is how you save the information of the tree so the game can build the tree.

Covers how to enter the graph editor and gives an overview of the basic layout

4. CREATING A BASIC TREE

Let’s create a basic tree using the entity we created and its script.

First, press the button labeled “Conditional” in the left panel of the graph AI Tree tab. You should see a new node being added to the graph with the “Conditional Node” title (If you don’t see it, move around the graph to look for it, as they all currently spawn in the center of the graph). This new node should have 3 ports (an input port on the left and two output ports on the right) and a dropdown menu. If you open the dropdown menu, you should be able to see the two signals we created while making the entity. Move this node to the right of the “Root Node” and left-click drag from the “Root Node”’s port to the “Conditional Node”’s input (left) port.

Next, press the button labeled “Action” in the left panel of the graph AI Tree tab. You should see a new node being added to the graph with the “Action Node” title. This new node should only have 1 port, the input port, and a dropdown menu. This dropdown contains a list of all of the methods we have added to the script attached to our entity’s scene root. Create a second action node by pressing the “Action” button again and drag both action nodes to the right of the conditional node. Connect one of the action nodes to the conditional’s true port and the other action node to the conditional’s false port.

On one of the two action nodes, select the options dropdown and select [*the second method we created]*. You should see a new row on that action node with a label matching the variable of the method we created. Type your name into the text edit in that row and your tree is finished! Simply press the save button and your tree has been fully created! **NOTE: YOU MUST PRESS THE SAVE BUTTON BEFORE SWITCHING SCENES OR ALL YOUR WORK WILL BE LOST. NODES THAT ARE NOT CONNECTED IN THE TREE WILL NOT BE SAVED.**

Goes through the process of creating a basic tree. Should only use the conditional node and action node for this basic tree, the later documentation will cover in detail using each of the specific nodes

5. RUNNING THE TREE IN YOUR GAME

Goes through the process of saving the tree and having it run in the game

PROJECT ELEMENTS

1. TREE DATA

TreeData is a custom resource whose sole purpose is to store the data for a behavior tree made in the graph editor. The resource consists of a variable that stores data in the form of a dictionary. The resource is in turn stored in an AI Tree Node (See Below), where it can be referenced by the editor to load the tree into the graph or it can be used to build a tree during runtime.

2. AI TREE NODE

Should cover the elements of the tree node. Two parts, this part and a second part that covers the code elements of the project

The AI Tree node is a custom Godot Node that is responsible for storing a TreeData resource and creating the behavior tree during runtime from a given resource. This node also defines if a scene is an AI scene, with the editor querying the scene for this node.

An AI Tree node should be added as a direct child of the scene root to mark that scene as an AI scene. THE NODE MUST BE LABELED AS “AINode”, AS THIS IS HOW THE EDITOR CHECKS IF IT IS AN AI SCENE. You should also create a new “TreeData” resource and attach the resource to the “AINode”’s “TreeData” export slot in the inspector.

2.1 BEHIND THE SCENES

The script for this node is simple. It contains two variables, one exported variable for the “TreeData” resource and one local variable that will hold the tree’s root node when the tree is built during run time.

The most important (and only) function of this script is the “build_node” function, which uses recursion to read through the TreeData to create a usable tree node at runtime. Unless you are planning on creating a custom tree node, then you will likely never need to touch this function. If, however, you are making a custom node, then you will need to add a match case to the match statement in the function using your node’s type as the case. This match case should create the new node you are looking to add alongside any relevant information that needs to be passed into that node. In most cases, your node will have outputs, in which case you should also pass in the data for the child node(s) to “build_node” functions to fully build out the tree. **NOTE: I WILL CREATE AN EXAMPLE AT SOME POINT TO CLARIFY THIS STEP FOR DEVELOPERS.**

3. AI NODES

This should cover the different node types that can exist and how they function in code. This should cover how each node works, how to use these outside the editor, and how to create new custom nodes

“AI Nodes” refers to the variety of nodes that are created during runtime to build an AI Behavior Tree. While these nodes share a similar name to their respective graph nodes, they have different purposes and function differently. These nodes are only responsible for working during runtime and managing their state during runtime. They have no impact on the graph editor’s operation.

3.1 TreeNode

This is the base class for all of the “AI Nodes”. While it is not abstract (GDScript won’t let me **:(**), you should never create an instance of “TreeNode”. Instead, all node types should extend this class, and then instances of those node types should be made instead.

This class contains the “node_started” and “node_finished” signals, as well as an empty “nodeProcess” function as a reminder that all nodes should have that function.

3.2 ActionNode

The “Action Node” stands as the only “leaf” node native to the project. The role of the “Action Node” is to have the AI entity perform and action specified by the node. These actions could attacks that the entity can make or having the entity patrol an area. Essentially, “Action Nodes” are used at the end of a tree’s branch to tell the entity what to do.

Each “Action Node” calls a provided function to run as its associated Action. As such, each “Action Node” should be given an Object (the object where the callable is stored) and a StringName (the name of the callable to trigger) via the “assignCallable” function. Additionally, if the desired callable has variables associated with it, then an array of variables should be appended or assigned to the “variableList” field. During the “nodeProcess” function, these fields are used to call the function from the object with the provided variables.

3.2.1 ActionNode SDK

Class Name: ActionNode
Extends: TreeNode

Properties:

- Object, callableObject, null
- StringName, callableName, “”
- Array, variableList, [ ]

Methods:

- void assignCallable(Object object, StringName name)
- RESULT nodeProcess()

Method Descriptions:

Void assignCallable(Object object, StringName name)
Assigns the callableObject and the callableName for the node

RESULT nodeProcess()
Processes the TreeNode. Calls the given callableName from the given callableObject using Object.callv, providing the variableList if present.

3.3 ConditionalNode

The “ConditionalNode” is a branching “TreeNode” with two possible children, a “True” node and a “False” node. This node stores a boolean value which is updated when a signal this node connects to is emitted. When this node is processed, this node checks the boolean value and processes the corresponding child node, “TrueNode” if the bool is true, and “FalseNode” if the bool is false.

3.3.1 ConditionalNode SDK

Class Name: ConditionalNode

Extends: TreeNode

Properties:

- TreeNode, falseNode, null
- TreeNode, trueNode, null
- bool, boolValue, false

Methods:

- Void updateBoolValue(bool value)
- Void assignSignal(Object object, StringName name)
- RESULT nodeProcess()

Method Descriptions:

Void updateBoolValue(bool value)

Updates the value of the boolValue. Called when the signal connected in "assignSignal" is emitted.

Void assignSignal(Object object, StringName name)

Connects the signal "name" from "object" to the "updateBoolValue" method. This connection is how this node tracks the boolean's value. A signal must be connected or this node will not work properly.

RESULT nodeProcess()

Processes the node. If the boolean is true and a "trueNode" is present, the process the "trueNode". If the boolean is false and a "falseNode" is present, then process the "falseNode". Returns RESULT.FAILED if the required node is not present. If the required child node is present, then returns the RESULT of said child node.

3.4 RandomNode

The "RandomNode" is a branching "TreeNode" that randomly selects one of its children to process. The random selection of the child is node is weighted, with the weight of each being defined as they are added as a child to the "RandomNode"

3.4.1 RandomNode SDK

Class Name: RandomNode

Extends: TreeNode

Properties:

- Array[TreeNode], associatedNodes, [ ]
- Array[float], nodeWeights, [ ]
- float, nodeWeightCombo, 0.0
- RandomNumberGenerator, rng, null

Methods:

- void _init()
- void addNode(TreeNode node, float weight)
- RESULT nodeProcess()

Method Descriptions:

void _init()

The initialization override for this node. Creates a new RandomNumberGenerator and assigns it to rng.

void addNode(TreeNode node, float weight)

Adds a "TreeNode" to this node as a child node. "node" is a "TreeNode" that is being added as a child node. "weight" is a float that is related to the child and determines the likelihood of that node being selected. If each node should have an equal chance of being selected, then all weights should be equal, otherwise weights should be larger if you want to increase their odds or lower if you want to decrease their odds. Increments "nodeWeightCombo" by the "weight" define the maximum value for the generated number when determining which child to pick.

RESULT nodeProcess()

Processes the node. Randomly generates a number from 0 to the "nodeWeightCombo", then compares against the weights to determine which child node to process. Returns the RESULT of the child node. If for some reason it does not select a node, this node will return RESULT.FAILED.

3.5 SequenceNode

The "SequenceNode" is a branching node that progresses through its child nodes in order each time it is run. This node has a "reset" boolean, which forces the node to reset the sequence index to 0 when the tree is processed without processing the node.

3.5.1 SequenceNode SDK

Class Name: SequenceNode
Extends: TreeNode

Properties:

- bool, resetOnMiss, true
- Array[TreeNode], associatedNodes, = [ ]
- int, sequenceIndex, 0

Methods:

- void _init(bool reset = true)
- void appendItemToSequence(TreeNode node)
- void addBlockingNode(TreeNode node)
- RESULT nodeProcess()

Method Descriptions:

void _init(bool reset = true)
The initialization override for this function. “reset” gets assigned to the “resetOnMiss” property.

void appendItemToSequence(TreeNode node)
Adds the provided node to the end of the sequence.

void addBlockingNode(TreeNode node)
Adds a node as a blocking node. A blocking node is used by the sequence node to determine if the sequence node should be reset.

NOTE ON USING BLOCKING NODES: If you are using the graph editor to create your tree, adding blocking nodes to sequence nodes will be done for you by the “AI Tree” node. If you are creating your AI trees in code, it is recommended that you take the time to draw out your graph to determine where the best nodes are in the tree to define as blocking nodes. The best spots for a blocking node are the earliest spots where a sequence can no longer be processed. ***ADD EXAMPLE IMAGE***

RESULT nodeProcess()
Process the Node. Runs the child node at the “sequenceIndex” of the “associatedNodes” array. If at the end of the sequence, it resets the “sequenceIndex” back to 0. Returns the RESULT of the child node.

3.6 CREATING A CUSTOM NODE

When creating a custom node for the tree, I recommend starting by creating said node as an “AI Node” and ensuring it works as intended before moving on to make a “Graph Node” equivalent for it.

Every custom “AI Node” should extend from the “TreeNode” class, and should implement the “nodeProcess” function from the parent class. The first line of the “nodeProcess” function should emit the “node_started” signal, and the last line should emit the “node_finished” signal. **NOTE: THE NODE FINISHED SIGNAL WILL EVENTUALLY NEED TO EMIT A “RESULT”**. Between these two signals, you should then carry out the function you are creating.

If your custom node will have children (i.e. other nodes will process after your custom node), then you should call the “nodeProcess” function on the node’s child/children and await the result. If you are doing multithreading, then you will need to call “wait_to_finish() for all of your threads before the “node_finished” signal is called or the tree may not operate properly.

4. GRAPH EDITOR

This should cover how to use the graph editor to make a branch. This should just cover the basics of adding and connecting nodes, as well as how to add buttons to the main panel and how to save and open a tree.

The graph editor is the main focus of this plugin. This graph editor allows the user to take the “AI Nodes” above and visually create a behavior tree for their entities. This makes it easy to visualize how the tree will behave while also making it easy for designers to make changes to the tree and variables within the tree.

The graph editor consists of two major components: the tool bar on the left of the scene and the graph editor on the right of the screen. The tool bar is responsible for allowing the user to add new nodes to the tree and perform other utility functions. Currently, the user can create 4 new nodes (conditional, random, sequence, and action) and can save their tree. The graph editor to the right is where graph nodes are moved, edited, and connected. It is in this graph where the behavior tree is designed and modified.

The graph editor will only appear while the currently selected scene contains an “AITree” node as a child of the scene root. When a scene without this node is selected, a screen will appear where the graph will be requesting to open a scene with an “AITree” node as a child of the root. ***This may be changed in the future to allow more options to the users, such as having multiple separate AI running in the same entity.***

Show some pictures of the editor, maybe some gifs of nodes moving and connecting

5. GRAPH NODES

This should cover the different nodes that exist in the graph editor. This should cover what each one represents, how their various components work, and how to create new graph nodes for custom nodes.

“Graph Nodes” are directly related to the “AI Nodes” and “TreeNodes” above. “Graph Nodes” are the graphical representation of the various nodes that can exist in a tree. For each “AI Node” that exists there is an equivalent “Graph Node” that collects the needed information from the user in the graph editor. These nodes, as opposed to the “AI Nodes”, are scenes that are placed into the tree graph editor to visually create the tree. When saved in the graph editor, these nodes pass their information to the “TreeData” resource saved in the associated “AITree” node. These nodes are the main components that make the graph editor work.

Note on Custom Nodes: If you want to make a custom “Graph Node”, it is recommended to make its equivalent “AI Node” first, as it is will define what goes into the “Graph Node”

5.1 Base Graph Node

Does not exist yet, however, I intend to create this as a class so I will leave space for it here in the documentation.

5.2 Root Graph Node

This graph node is a node that is present in every AI Tree. This node represents the root of the tree and is used by the main panel to know where the tree starts. Nodes are saved starting from this node.

The node contains one output. The node that this output connects to is the first node of the behavior tree, and when the tree is being built during runtime this connected node will be the root node for the tree. The root graph node cannot be deleted.

5.2.1 Root Graph Node SDK

Properties:

- BaseGraphNode, rootNode, null
- int, rootPort, -1

Methods:

- void addOutput(GraphEdit graph, Node node, int from_port, int to_port)
- void remove_connection(int from_port)
- void _delete_connection(GraphEdit graph, int toPort)
- Dictionary save()

Method Descriptions:

void addOutput(GraphEdit graph, Node node, int from_port, int to_port)

Adds another graph node as the output of the root node. Removes the previously connected node is one is present.

void remove_connection(int from_port)

Clears the connection information from the root node.

void _delete_connection(GraphEdit graph, in toPort)
Triggered when the root node is leaving the tree. Disconnects the node and clears the connection information from the root node.

Dictionary save()
Takes all of the data for the root node and stores it in a dictionary. Gets the save data from the connected node.

5.3 Action Graph Node

Action Graph nodes represent action nodes in the graph editor. As such, these graph nodes contain elements that let you define a function to be run when the action node is processed as well as set the variables that the function needs to operate.

This graph node has one input and no outputs. When you first create an action graph node, there will be a dropdown button. The options in this dropdown menu are made from the list of the user's custom functions attached to the root node of the currently selected scene. Selecting a function from this dropdown will tell the behavior tree which function should be called when that node is processed. If the function you selected has parameters, then below the dropdown menu a number of rows for entering data for those parameters will appear, one for each parameter. Note that for this to work properly your parameters must have an explicit type declaration so the action graph node knows what type of editor to display for that variable.

5.3.1 Action Graph Node SDK

Properties:

- bool, inputUsed, false
- Array[String], arrayOfMethods, [ ]
- int, indexDifference, 0
- Array[int], varTypeArray, [ ]
- onready Node, callableOptions, get_node("FunctionDefinition/CallableOptions")
- Dictionary, providedData, { }

Methods:

- void _ready()
- void _on_callable_options_item_selected(int index)
- void add_variable(String variableName, int variantType)

- void load_data()
- void update_input(bool value)
- bool check_input()
- void delete_node(GraphEdit graph)
- Dictionary save()

Method Descriptions:

void _ready()

Function called when the node is ready. Prepares the callableOptions dropdown menu with all of the options being the functions that are in the script attached to the root of the selected scene. It also checks if data has been provided, in which case it loads that data.

void _on_callable_options_item_selected(int index)

Called when an option is selected in the dropdown menu. Determines what method has been selected and creates a row for each of the arguments of the method for the user to fill out.

void add_variable(String variableName, int variantType)

Adds a new argument row to the graph node, enabling the user to define the value for that argument. Creates a label from the provided string name. The variantType is used to determine what form of input to provide the user. Bool is a checkbox, int, float, and string are all TextEdits, and Vector2 and Vector3 have two and three TextEdits, respectively.

void load_data()

Takes the provided data and loads that information into the graph node. The callable option is selected in the options menu and the arguments are added. Once the arguments are added, the arguments are set to the arguments in the providedData.

void update_input(bool value)

Updates if this node has a connection to its input port

bool check_input()

Returns if the input port is connected.

void delete_node(GraphEdit graph)

Is called when this node is deleted. For this node it simply passes. It should remain in the script, however, as the function is called when any BaseGraphNode is deleted.
NOTE This will likely be moved to the BaseGraphNode class, meaning this function will no longer be needed.

Dictionary save()

Saves all the data for the graph node. Takes the argument data from the user's inputs and stores them as needed, depending on the data type. Returns all of this information, plus node type and position, in a dictionary.

5.4 Conditional Graph Node

Conditional graph nodes represent conditional nodes in the graph editor. This graph node enables you to define what signal should be used to update the conditional's value. This node is also responsible for determining what node should be processed depending on the value the conditional is checking.

This graph node has one input and two outputs. The first output is labeled as the "True Node", and the graph node it is connected to will determine what node is processed if the conditional is true. Likewise, the second output is labeled as the "False Node", and its connected graph node will be the node that is processed if the conditional is false. There is a dropdown menu attached to this node that contains a list of signals the script attached to the root of the selected scene. This signal determines what signal will be connected to the update function in the ConditionalNode instance to track the variable for checking the conditional.

5.4.1 Conditional Graph Node SDK

Properties:

- bool inputUsed = false
- onready Node signalOptions = get_node("BottomRow/SignalOptions")
- Array[String] arrayOfSignals = [ ]
- Array[GraphNode] associatedNodes = [null, null]
- Array[int] associatedPorts = [-1, -1]
- Dictionary providedData = { }

Methods:

- void _ready()
- bool check_input()
- void update_input(bool value)
- void add_output(GraphEdit graph, Node node, int from_port, int to_port)
- void remove_connection(int from_port)
- void delete_node(GraphEdit graph)
- void _delete_connection(GraphEdit graph)
- void load_data()
- Dictionary save()

Method Descriptions:

5.5 Random Graph Node

Random graph nodes represent random nodes in the graph editor. This graph node allows you to connect many nodes as children and assign weights to them for the random selection.

This graph has one input and an infinite number of outputs. When the node is first created, there are zero outputs. There is a button at the bottom of the node that, when pressed will at a new column of information and a new output. Any number of these rows can be added, and rows can be deleted by pressing the button at the rightmost part of the row. Each row contains a text edit which corresponds to the connected nodes weight, or chance of being randomly selected.

5.5.1 Random Graph Node SDK

Properties:

- bool inputUsed = false
- Array[GraphNode] associatedNodes = [ ]
- Array[int] associatedPorts = [ ]
- Dictionary providedData = { }

Methods:

- void _ready()
- void _on_add_node_button_pressed()
- void _on_delete_button_pressed()
- void update_input(bool value)
- bool check_input()
- void add_output(GraphEdit graph, Node node, int from_port, int to_port)
- void remove_connection(int from_port)
- void delete_node(GraphEdit graph)
- void _delete_connection(GraphEdit graph, int fromPort, Node toNode, int toPort)
- void load_data()
- Dictionary save()

Method Descriptions:

void _ready()

Th ready function for the node. Checks if data has been provided. If it has, then loads in they data.

Void _on_add_node_button_pressed()
Triggered when the add node button at the bottom of the graph node is pressed. Creates a new row in the graph node with a weight text edit and an output port.

void _on_delete_button_pressed()
Triggered when the delete button is pressed in a row. Removes the row from the data and shifts all of the connections to the graph node appropriately.

void update_input(bool value)
Updates if a node is connected to this graph node's input port.

bool check_input()
Returns whether the input port is in use.

void add_output(GraphNode graph, int from_port, int to_port)
Adds a connection to one of the output ports. If a node is already connected to one of those output ports, then disconnects the currently connected node.

void remove_connection(int from_port)
Clears a connection from an output port.

Void delete_node(GraphEdit graph)
Triggered when this node is deleted. Disconnects all of its associated nodes so it can be deleted cleanly.

void _delete_connection(GraphEdit graph, int fromPort, Node toNode, int toPort)
When a connection delete request is entered. Disconnects the connection to the provided node and clears the data from this node.

Void load_data()
Loads in the provided data, creating all the necessary output ports and assigning the provided weights

void save()
Saves the data of the node. First gets all of the save data from its child nodes, then returns a dictionary of the saved data.

5.6 Sequence Graph Node

Sequence graph nodes represent sequence nodes in the graph editor. This graph node allows you to connect many nodes as children, with their order determining the order in which they are processed as the sequence node is processed multiple times in a row.

This graph as one input and an infinite number of outputs. There is a checkbox labeled "Reset", which is used to determine if the sequence resets when not run simultaneously during runtime. When the node is first created, there are zero outputs. There is a button at the bottom of the node that, when pressed will at a new column of information and a new output. Any number of these rows can be added, and rows can be deleted by pressing the button at the rightmost part of the row. During runtime, the sequence node will start by processing the top-most node (Node 1) and work its way down.

5.6.1 Sequence Graph Node SDK

Properties:

- bool inputUsed = false
- Array[GraphNode] associatedNodes = [ ]
- Array[int] associatedPorts = [ ]
- Dictionary providedData = { }

Methods:

- void _ready()
- void _on_add_node_button_pressed()
- void _on_delete_button_pressed()
- void update_input(bool value)
- bool check_input()
- void add_output(GraphEdit graph, Node node, int from_port, int to_port)
- void remove_connection(int from_port)
- void delete_node(GraphEdit graph)
- void _delete_connection(GraphEdit graph, int fromPort, Node toNode, int toPort)
- void load_data()
- Dictionary save()

Method Descriptions:

void _ready()

Th ready function for the node. Checks if data has been provided. If it has, then loads in they data.

Void _on_add_node_button_pressed()

Triggered when the add node button at the bottom of the graph node is pressed. Creates a new row in the graph node with an output port.

void _on_delete_button_pressed()

Triggered when the delete button is pressed in a row. Removes the row from the data and shifts all of the connections to the graph node appropriately.

void update_input(bool value)

Updates if a node is connected to this graph node’s input port.

bool check_input()
	Returns whether the input port is in use.

void add_output(GraphNode graph, int from_port, int to_port)
	Adds a connection to one of the output ports. If a node is already connected to one of those output ports, then disconnects the currently connected node.

void remove_connection(int from_port)
	Clears a connection from an output port.

Void delete_node(GraphEdit graph)
	Triggered when this node is deleted. Disconnects all of its associated nodes so it can be deleted cleanly.

void _delete_connection(GraphEdit graph, int fromPort, Node toNode, int toPort)
	When a connection delete request is entered. Disconnects the connection to the provided node and clears the data from this node.

Void load_data()
	Loads in the provided data, creating all the necessary output ports and assigning the provided weights

void save()
	Saves the data of the node. First gets all of the save data from its child nodes, then returns a dictionary of the saved data.

5.7 CREATING A CUSTOM GRAPH NODE

6. RUNNING THE TREE

This should be a relatively short section that describes how the user can run the tree. The biggest thing is to explain how to use the "nodeProcess" function and give a couple of examples of when to trigger the "nodeProcess" function.

Once TreeData has been added to an AINode, or you have created a tree in code manually, the only thing left to do is actually run the tree. Everytime you want the tree to perform an action, simply grab the root node of the tree (either from the AINode or wherever you stored it) and call the nodeProcess() function on it. Everytime you call nodeProcess() on the root node, it should work its way through the tree to an action node and perform that action. If you call nodeProcess() again, then it will work through the tree again, potentially to a different action depending on random selection, conditionals, and sequence updates.

Generally, you will want to create a way to call the nodeProcess() function in a controllable maner, such as through cooldowns. Depending on your AI Entity and how you set up your tree/functions, you may be able to call nodeProcess every frame. However, generally speaking you will only want to call nodeProcess when you want the AI to take an action.

7. CREATING CUSTOM NODES

FUTURE CONTENT

More for me than anyone else. Details the changes I want to make in the future and new content I want to add.

1. RESULT ENUM

This will be an enumeration linked to the base class of nodes that will represent the possible results of running a node. This will allow for advanced features in nodes.

<u>Update 1:</u> I have made the enum and added it to the TreeNode script. While all nodes return a RESULT and the children receive that RESULT, there is still not much being done by the nodes with this RESULT. While this is mostly because because the node that will make the most use out of this feature doesn't exist yet, I do want to spend a little more time making sure it is working as intended and that there is nothing more I want to do with this feature in currently existing nodes.

<u>Update 2:</u> I think I do want to add a RUNNING state to the results enum. It will give me more options when working with nodes, especially when trying to determine when to retrigger the brain to process. I will want to stop trying to run the brain multiple times (and will want a catch to prevent this).

~~FINSHED 2. NEW BRANCH TRACKING SYSTEM~~

This is a tool that generates the various branches present in a tree and associates them with the corresponding node. This will allow for an easier deduction of what nodes are blocking to sequence nodes (And as opposed to currently will be accurate in all cases).

Update 1: This system has been finished. I may go back and rework the system if it becomes too slow or I think of a better way to create the system, but otherwise this is working well and as intended.

~~FINISHED 3. SELECTOR NODE~~

This will be a new node that will behave similarly to a sequence node. Instead of going from one connection to another based on the number of times in a row that sequence has been triggered, it will always start on the first node of the selector. If that node fails (Hence the need for a result enum), then the selector will go to its next connection.

Update 1: The node has been added and is seemingly working. I will probably want to start stress testing this node and the system as a whole to make sure everything works at scale.

4. MULTI-NODE

This will be a new node that will have a collection of nodes in an order, similar to a sequence. This node will also have a boolean to determine if its children should be run in parallel or in sequence. If this node is ran and is to run in parallel, then all linked nodes will be triggered simultaneously, only ending this node when all are finished. If it is to run in sequence, then it should start at the first child node and work through all of its children, only finishing when its final child has been run.

Update 1: The process node has been (mostly) completed. I still need to add the functionality for if the simultaneous checkbox is unchecked, but that should be pretty straightforward. Otherwise, functions can be called simultaneously, with the onus being on the developer to create thread-safe functions that the node can call.

Update 2: The process node has been finished (outside of further testing). The simultaneous checkbox now works. When checked, the child nodes will run simultaneously. When unchecked, the child nodes will run one after the other.

Update 3: Something I realized while working on the tree is that I will likely not have the tree running multiple times overtop of itself. This made me think if I wanted to add delays to the nodes childed to this node so that they can run simultaneously but out of sync from each other. I think I would like to consider this.

~~FINISHED 5. EXCLUSIVE RANDOM~~

This will be an update to the random node that will add a boolean to make that random node exclusive. If a random node is marked as exclusive, then each child node can only be activated once at a time. This means that when a child node is selected it cannot be chosen again until that node is finished. This is to allow for a tree to be run multiple times while actions are running, but have certain actions to only have one instance active at a time.

Update 1: This has been completed, allowing users to mark a random node as exclusive. These nodes marked exclusive will run almost identically to a regular random node with the exception that these exclusive nodes will not allow for the same node to be ran until that node is finished.

6. BASE GRAPH NODE

This will simply be a base node for the graph nodes that all graph nodes will inherit (including the root node, even though most of them won't be used). This will allow me to take some of the functions that are reused by all nodes and put them in one place. It will also allow me to create virtual methods that will need to be overridden in their children and will make it easier to find out what nodes need to be added.

~~FINISHED 7. UPDATE TO CONDITIONAL NODES~~

I have realized there may be a way to use a boolean from the parent node instead of relying on a signal to update a conditional statement. If I have the StringName of the variable and get a reference to the parent node of the AITree Node, I should be able to get the property from the node. This will make it easier on the user when creating a new entity as they won't have to set up a property getter that emits a signal and instead can just add a boolean and update it.

Update 1: I have fully switched to using booleans instead of signals to track the boolean for the conditional nodes. This makes it easier on the user as it means they only need to create a boolean and don't need to create a whole signal system. It also make the system just need a single get call and that's it.

~~FINISHED 8. UPDATE TO ACTION FUNCTION REQUIREMENTS~~

I would like to for it so that every function declaration that will be used in the AI Tree Visual Editor will need to explicitly declare that it will be returning a RESULT enum. This will have two benefits. The first is that for the developers it will force them to make sure all routes within their function return a RESULT, which is important for proper traversal of the tree. The second is that I will be able to limit which functions are displayed in the AI Graph Visual Editor, showing the user only functions that are related to the AI Tree and forcing the player into the restrictions above if they want it to be usable in the Visual Editor.

Update 1: This system has been added. Action nodes will now only accept nodes that have an explicit return declaration for TreeNode.RESULT. This helps to clean up what is showing in the visual editor and it gives the developer a tool to make sure that only the functions they want to be called can be called in the visual editor.

~~FINISHED 9. ADD DELAY NODE~~

This will be a new node that simply adds a delay to the nodes being processed. This node will include a float value that represents how many seconds the node should delay for.

Update 1: This has been added and works for both the visual editor and through code. Note that the node needs a reference to the scene tree to work so it can get access to the timers on the tree.

10. FIX NULL NODES NOT CATCHING

Currently the system does not handle null values very well, especially the random node when a weight exists but the node does not. I will want to make a catch for this so that it the game doesn't crash when this situation occurs, but I will likley warn the developer that this is happening so that they fix the issue one way or another.

11. FIX ISSUE WITH SEQUENCE BEING ROOT NODE

Currently the branch ID system does not take into account if the root node is a sequence node, which results in a crash. This has two simple solutions. The first would be to simply tell users to not check reset on root sequence nodes, as they would never be reset anyways. The other is to try and find a way to force the reset button to be false and disabled when connected to the root node.

12. TEST AND FIX ISSUES WHEN DELETING NODES

While working on a separate project to stress test how this plugin works, I encountered some issues that eventually resulted in my loosing all of the nodes I had saved in a tree. I am not exactly sure what caused this, so I want to spend some time exploring with the plugin to see if I can identify any situations that can cause issues and work to address them.