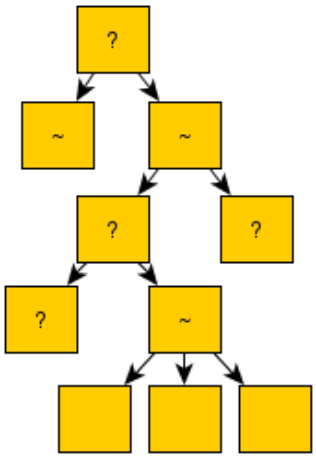
Behavior trees for AI: How they work

Chris Simpson

Jul 17, 2014

1 Introduction

While there are plenty of behaviour tree tutorials and guides around the internet, when exploring whether they would be right for use in Project Zomboid, I ran into the same problem again and again. Many of the guides I read focused very heavily on the actual code implementations of behaviour trees, or focused purely on the flow of generic contextless nodes without any real applicable examples, with diagrams like so:



While they were invaluable in helping me understand the core principles of Behaviour Trees, I found myself in a situation where despite knowing how a behaviour tree operated, I didn't really have any real-world context as to what sort of nodes I should be creating for the game, or what an actual fully developed behaviour tree would look like.

I've spent a ton of time experimenting (for the record since Project Zomboid is in Java I'm using the fantastic JBT[1] so didn't have to concern myself with the actual code implementation. However there are plenty of tutorials out there focusing on this, as well as implementations in many commonly used game engines.

It's possible some of the more specific decorator node types I detail here are actually native to JBT instead of general behaviour tree concepts, but I've found them to be integral to the way PZ behaviour trees work, so they are worth considering for implementation if your particular behaviour tree does not support them.

I'm not professing to be an expert on the subject, however over the development of the Project Zomboid NPCs I've found the results I've had to be pretty solid, so thought I'd bash out a few things that if I'd known would have made my first attempts go a lot more smoothly, or at least opened my eyes to what I could accomplish with behaviour trees. I'm not going to dig into the implementation but just give a few abstracted examples that were used in Project Zomboid.

2 Basics

So the clue is in the name. Unlike a Finite State Machine, or other systems used for AI programming, a behaviour tree is a tree of hierarchical nodes that control the flow of decision making of an AI entity. At the extents of the tree, the leaves, are the actual commands that control the AI entity, and forming the branches are various types of utility nodes that control the AI's walk down the trees to reach the sequences of commands best suited to the situation.

The trees can be extremely deep, with nodes calling sub-trees which perform particular functions, allowing for the developer to create libraries of

[1] http://sourceforge.net/projects/jbt/

behaviours that can be chained together to provide very convincing AI behaviour. Development is highly iterable, where you can start by forming a basic behaviour, then create new branches to deal with alternate methods of achieving goals, with branches ordered by their desirability, allowing for the AI to have fallback tactics should a particular behaviour fail. This is where they really shine.

3 Data Driven vs Code Driven

This distinction has little relevance to this guide, however it should be noted that there are many different possible implementations of behaviour trees. A main distinction is whether the trees are defined externally to the codebase, perhaps in XML or a proprietary format and manipulated with an external editor, or whether the structure of the trees is defined directly in code via nested class instances.

JBT uses a strange hybrid of these two, where an editor is provided to allow you to visually construct your behaviour tree, however an exporter command line tool actually generates java code to represent the behaviour trees in the code-base.

Whatever the implementation, the leaf nodes, the nodes that actually do the game specific business and control your character or check the character's situation or surroundings, are something you need to define yourself in code. Be that in the native language or using a scripting language such as Lua or Python. These can then be leveraged by your trees to provide complex behaviours. It is quite how expressive these nodes can be, sometimes operating more as a standard library to manipulate data within the tree itself, than just simply character commands, that really make behaviour trees exciting to me.

4 Tree Traversal

A core aspect of Behavior Trees is that unlike a method within your codebase, a particular node or branch in the tree may take many ticks of the game to complete. In the basic implementation of behaviour trees, the system will traverse down from the root of the tree every single frame, testing each node down the tree to see which is active, rechecking any nodes along the way, until it reaches the currently active node to tick it again.

This isn't a very efficient way to do things, especially when the behaviour tree gets deeper as its developed and expanded during development. I'd say its a must that any behaviour tree you implement should store any currently processing nodes so they can be ticked directly within the behaviour tree engine rather than per tick traversal of the entire tree. Thankfully JBT fits into this category.

5 Flow

A behaviour tree is made up of several types of nodes, however some core functionality is common to any type of node in a behaviour tree. This is that they can return one of three statuses. (Depending on the implementation of the behaviour tree, there may be more than three return statuses, however I've yet to use one of these in practice and they are not pertinent to any introduction to the subject) The three common statuses are as follows:

Success Failure Running

The first two, as their names suggest, inform their parent that their operation was a success or a failure. The third means that success or failure is not yet determined, and the node is still running. The node will be ticked again next time the tree is ticked, at which point it will again have the opportunity to succeed, fail or continue running.

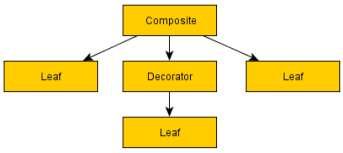
This functionality is key to the power of behaviour trees, since it allows a node's processing to persist for many ticks of the game. For example a Walk node would offer up the Running status during the time it attempts to calculate a path, as well as the time it takes the character to walk to the specified location. If the pathfinding failed for whatever reason, or some other complication arisen during the walk to stop the character reaching the target location, then the node returns failure to the parent. If at any point the character's current location equals

the target location, then it returns success indicating the Walk command executed successfully.

This means that this node in isolation has a cast iron contract defined for success and failure, and any tree utilizing this node can be assured of the result it received from this node. These statuses then propagate and define the flow of the tree, to provide a sequence of events and different execution paths down the tree to make sure the AI behaves as desired.

With this shared functionality in common, there are three main archetypes of behaviour tree node:

- Composite
- Decorator
- Leaf



5.1 Composite

A composite node is a node that can have one or more children. They will process one or more of these children in either a first to last sequence or random order depending on the particular composite node in question, and at some stage will consider their processing complete and pass either success or failure to their parent, often determined by the success or failure of the child nodes. During the time they are processing children, they will continue to return Running to the parent.

The most commonly used composite node is the Sequence, which simply runs each child in sequence, returning failure at the point any of the children fail, and returning success if every child returned a successful status.

5.2 Decorator

A decorator node, like a composite node, can have a child node. Unlike a composite node, they can specifically only have a single child. Their function is either to transform the result they receive from their child node's status, to terminate the child, or repeat processing of the child, depending on the type of decorator node.

A commonly used example of a decorator is the Inverter, which will simply invert the result of the child. A child fails and it will return success to its parent, or a child succeeds and it will return failure to the parent.

5.3 Leaf

These are the lowest level node type, and are incapable of having any children.

Leafs are however the most powerful of node types, as these will be defined and implemented by your game to do the game specific or character specific tests or actions required to make your tree actually do useful stuff.

An example of this, as used above, would be Walk. A Walk leaf node would make a character walk to a specific point on the map, and return success or failure depending on the result.

Since you can define what leaf nodes are yourself (often with very minimal code), they can be very expressive when layered on top of composite and decorators, and allow for you to make pretty powerful behavior trees capable of quite complicated layered and intelligently prioritized behaviour.

In an analogy of game code, think of composites and decorators as functions, if statements and while loops and other language constructs for defining flow of your code, and leaf nodes as game specific function calls that actually do the business for your AI characters or test their state or situation.

These nodes can be defined with parameters. For example the Walk leaf node may have a coordinate for the character to walk to.

These parameters can be taken from variables stored within the context of the AI character processing the tree. So for example a location to walk to could be determined by a 'GetSafeLocation' node, stored in a variable, and then a 'Walk' node could use that variable stored in the context to

define the destination. It's through using a shared context between nodes for storing and altering of arbitrary persistent data during processing of a tree that makes behaviour trees immensely powerful.

Another integral type of Leaf node is one that calls another behaviour tree, passing the existing tree's data context through to the called tree.

These are key as they allow you to modularise the trees heavily to create behaviour trees that can be reused in countless places, perhaps using a specific variable name within the context to operate on. For example a 'Break into Building' behaviour may expect a 'targetBuilding' variable with which to operate on, so parent trees can set this variable in the context, then call the sub-tree via a sub-tree Leaf node.

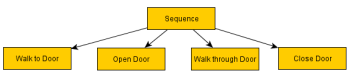
6 Composite Nodes

Here we will talk about the most common composite nodes found within behaviour trees. There are others, but we will cover the basics that should see you on your way to writing some pretty complex behaviour trees in their own right.

6.1 Sequences

The simplest composite node found within behaviour trees, their name says it all. A sequence will visit each child in order, starting with the first, and when that succeeds will call the second, and so on down the list of children. If any child fails it will immediately return failure to the parent. If the last child in the sequence succeeds, then the sequence will return success to its parent.

It's important to make clear that the node types in behaviour trees have quite a wide range of applications. The most obvious usage of sequences is to define a sequence of tasks that must be completed in entirety, and where failure of one means further processing of that sequence of tasks becomes redundant. For example:



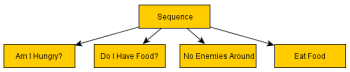
This sequence, as is probably clear, will make the given character walk through a door, closing it behind them. In truth, these nodes would likely be more abstracted and use parameters in a production environment. Walk (location), Open (openable), Walk (location), Close (openable)

The processing order is thus:

Sequence -¿ Walk to Door (success) -¿ Sequence (running) -¿ Open Door (success) -¿ Sequence (running) -¿ Walk through Door (success) -¿ Sequence (running) -¿ Close Door (success) -¿ Sequence (success) -¿ at which point the sequence returns success to its own parent.

If a character fails to walk to the door, perhaps because the way is blocked, then it is no longer relevant to try opening the door, or walking through it. The sequence returns failure at the moment the walk fails, and the parent of the sequence can then deal with the failure gracefully.

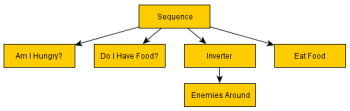
The fact that sequences naturally lend themselves to sequences of character actions, and since AI behaviour trees tend to suggest this is their only use, it may not be clear that there are several different ways to leverage sequences beyond making a character do a sequential list of 'things'. Consider this:



In the above example, we have not a list of actions but a list of tests. The child nodes check if the character is hungry, if they have food on their person, if they are in a safe location, and only if all of these return success to the sequence parent, will the character then eat food. Using sequences like this allow you to test one or more conditions before carrying out an action. Analogous to if statements in code, and to an AND gate in circuitry. Since all children need to succeed, and those children could be any combination of composite, decorator or leaf nodes, it allows for pretty powerful conditional checking within your AI brain.

Consider for example the Inverter decorator mentioned in the above section:

Functionally identical to the previous example, here we show how you can use inverters to negate



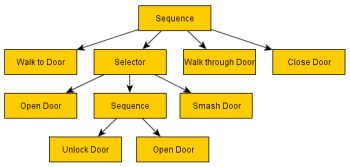
any test and therefore give you a NOT gate. This means you can drastically cut the amount of nodes you will need for testing the conditions of your character or game world.

6.2 Selector

Selectors are the yin to the sequence's yang. Where a sequence is an AND, requiring all children to succeed to return a success, a selector will return a success if any of its children succeed and not process any further children. It will process the first child, and if it fails will process the second, and if that fails will process the third, until a success is reached, at which point it will instantly return success. It will fail if all children fail. This means a selector is analagous with an OR gate, and as a conditional statement can be used to check multiple conditions to see if any one of them is true.

Their main power comes from their ability to represent multiple different courses of action, in order of priority from most favorable to least favorable, and to return success if it managed to succeed at any course of action. The implications of this are huge, and you can very quickly develop pretty sophisticated AI behaviours through the use of selectors.

Let's revisit our door sequence example from earlier, adding a potential complication to it and a selector to solve it.



Yes, here we can deal with locked doors intelligently, with the use of only a handful of new nodes.

So what happens when this selector is processed?

First, it will process the Open Door node. The most preferable cause of action is to simply open the door. No messing. If that succeeds then the selector succeeds, knowing it was a job well done. There's no further need to explore any other child nodes of that selector.

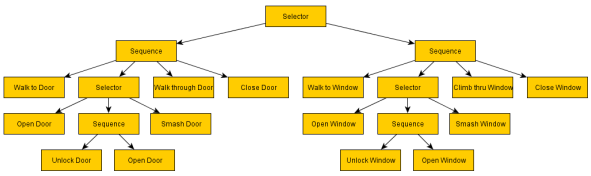
If, however, the door fails to open because some sod has locked it, then the open door node will fail, passing failure to the parent selector. At this point the selector will try the second node, or the second preferable cause of action, which is to attempt to unlock the door.

Here we've created another sequence (that must be completed in entirety to pass success back to the selector) where we first unlock the door, then attempt to open it.

If either step of unlocking the door fails (perhaps the AI doesn't have the key, or the required lock-picking skill, or perhaps they managed to pick the lock, but found the door was nailed shut when attempting to open it?) then it will return failure to the selector, which will then try the third course of action, smashing the door off its hinges!

If the character is not strong enough, then perhaps this fails. In this case there are no more courses of action left, and the the selector will fail, and this will in turn cause the selector's parent sequence to fail, abandoning the attempt to walk through the door.

To take this a step further, perhaps there is a selector above that which will then choose another course of action based on this sequence's failure?



Here we've expanded the tree with a topmost selector. On the left (most preferable side) we enter through the door, and if that fails we instead try to enter through the window. In truth the actual implementation would likely not look this way and its a bit of a simplification on what we did on Project Zomboid, but it illustrates the point. We'll get to a more generic and usable implementation later.

In short, we have here an 'Enter Building' be-

haviour that you can rely on to either get inside the building in question, or to inform its parent that it failed to. Perhaps there are no windows? In this case the topmost selector will fail, and perhaps a parent selector will tell the AI to head to another building?

A key factor in behaviour trees that has simplified AI development a huge deal for myself over previous attempts is that failure is no longer a critical full stop on whatever I'm trying to do (uhoh, the pathfind failed, WHAT NOW?), but just a natural and expected part of the decision making process that fits naturally in the paradigm of the AI system.

You can layer failsafes and alternate courses of action for every possible situation. An example with Project Zomboid would be the EnsureItemInInventory behaviour.

This behaviour takes in an inventory item type, and uses a selector to determine from several courses of action to ensure an item is in the NPC's inventory, including recursive calls to the same behaviour with different item parameters.

First it'll check if the item is already in the character's main top level inventory. This is the ideal situation as nothing needs to be done. If it is, then the selector succeeds and thus the entire behaviour succeeds. EnsureItemInInventory has succeeded, and the item is there for use.

If the item is not in the character's inventory, then they will check the contents of any bags or backpacks the character is carrying. If the item is found, then they will transfer the item from the bag into his top level inventory. This will then succeed, as the success criteria is met.

If THIS fails, then a third branch of the selector will determine of the item is located in the building the character is currently residing in. If it is, then the character will travel to the location of the container holding the item and take it from the container. Again the criteria is met, so success!

If THIS fails, then there is one more trick up the NPCs sleeve. It will then iterate a list of crafting recipes that result in the item they desire, and for each of these recipes it will iterate through each ingredient item, and will recursively call the EnsureItemInInventory behaviour for each of those items in turn. If each of these succeeds, then we know for a fact that the NPC now carries every ingredient required to craft their desired item. The character will then craft the item from those ingredients, before returning success as the criteria of having the item is met.

If THIS fails, then the EnsureItemInInventory behaviour will fail, with no more fallbacks, and the NPC will just add that item to a list of desired items to look out for during looting missions and live without the item.

The result of this is that the NPC is suddenly capable of crafting any item in the game they desire if they have the ingredients required, or those ingredients can be obtained from the building.

Due to the recursive nature of the behaviour, if they don't have the ingredients themselves, then they will even attempt to craft them from even baser level ingredients, hunting the building if necessary, crafting multiple stages of items to be able to craft the item they actually need.

Suddenly we have a quite complicated and impressive looking AI behaviour that actually boils down to relatively simple nodes layered on top of each other. The EnsureItemInInventory behaviour can then be used liberally throughout many other trees, whenever we need an NPC to ensure they have an item in their inventory.

I'm sure at some point during development we'll continue this further with another fallback, and allow the NPCs to actually go out specifically in search of items they critically desire, choosing a looting target that has the highest chance of containing that item.

Another failsafe that could be higher in the priority list may be to consider other items which may accomplish the same goal as the selected item. If one day we finally code in support for makeshift tools, then looking for less effective alternatives and hammering a nail in with a rock may trump sneaking across town into a zombie infested hardware store.

Due to the ease of extending the trees during development, its easy to create a simple behaviour that 'does the job', and then iteratively improve that NPC behaviour with extra branches via a se-

lector to cater for more solid failsafes and fallbacks to reduce the likelihood of the behaviour failing. The crafting fallback was added much later down the line, and just goes to further equip NPCs with behaviours to further aid them in achieving their goals.

Furthermore if prioritized carefully, these fallbacks, despite being essentially scripted behaviours, bestow the appearance of intelligent problem solving and natural decision making to the AI character.

6.3 Random Selectors / Sequences

I'm not going to dwell on these, as their behaviour will be obvious given the previous sections. Random sequences/selectors work identically to their namesakes, except the actual order the child nodes are processed is determined randomly. These can be used to add more unpredictability to an AI character in cases where there isn't a clear preferable order of execution of possible courses of action.

7 Decorator Nodes

7.1 Inverter

We've already covered this one. Simply put they will invert or negate the result of their child node. Success becomes failure, and failure becomes success. They are most often used in conditional tests.

7.2 Succeeder

A succeeder will always return success, irrespective of what the child node actually returned. These are useful in cases where you want to process a branch of a tree where a failure is expected or anticipated, but you don't want to abandon processing of a sequence that branch sits on. The opposite of this type of node is not required, as an inverter will turn a succeeder into a 'failer' if a failure is required for the parent.

7.3 Repeater

A repeater will reprocess its child node each time its child returns a result. These are often used at the very base of the tree, to make the tree to run continuously. Repeaters may optionally run their children a set number of times before returning to their parent.

7.4 Repeat Until Fail

Like a repeater, these decorators will continue to reprocess their child. That is until the child finally returns a failure, at which point the repeater will return success to its parent.

8 Data Context

The specifics of this are down to the actual implementation of the behaviour tree, the programming language used, and all manner of other things, so we'll keep this all rather abstract and conceptual.

When a behaviour tree is called on an AI entity, a data context is also created which acts as a storage for arbitrary variables that are interpreted and altered by the nodes (using string/object pair in a C# Dictionary or java HashMap, probably a C++ string/void* STL map, though its a long time since I've used C++ so there are probably better ways to handle this)

Nodes will be able to read or write into variables to provide nodes processed later with contextual data and allow the behaviour tree to act as a cohesive unit. As soon as you start exploiting this heavily, the flexibility and scope of behaviour trees becomes very impressive, and the true power at your fingertips becomes apparent. We'll get to this in a while when we revisit our doors and windows behaviour.

9 Defining Leaf Nodes

Again, the specifics of this are down to the actual implementation of the behaviour tree. In order to provide functionality to leaf nodes, to allow

for game specific functionality to be added into behaviour trees, most systems have two functions that will need to be implemented.

init —Called the first time a node is visited by its parent during its parents execution. For example a sequence will call this when its the node's turn to be processed. It will not be called again until the next time the parent node is fired after the parent has finished processing and returned a result to its parent. This function is used to initialise the node and start the action the node represents. Using our walk example, it will retrieve the parameters and perhaps initiate the pathfinding job.

process —This is called every tick of the behaviour tree while the node is processing. If this function returns Success or Failure, then its processing will end and the result passed to its parent. If it returns Running it will be reprocessed next tick, and again and again until it returns a Success or Failure. In the Walk example, it will return Running until the pathfinding either succeeds or fails.

Nodes can have properties associated with them, that may be explicitly passed literal parameters, or references to variables within the data context of the AI entity being controlled.

I'm not going to go into the specifics of implementation, as this is not only language dependent but also behaviour tree implementation dependent, but the concept of parameters and storage of arbitrary data within the behaviour tree instance are fairly universal.

So for example, we may describe a Walk node as such:

Walk (character, destination)

- *success*: Reached destination
- *failure*: Failed to reach destination
- *running*: En route

In this case Walk has two parameters, the character and the destination. While it may seem natural to always assume that the character who is running the AI behaviour is the subject of a node and therefore would not need to be passed explicitly as a parameter, it's best not to make this assumption, despite 'Walk' being a pretty safe bet. As too many times, particularly on conditional nodes, I've found myself having to recode nodes to cater for testing another characters state or interacting with them in some way. It's always best to go the extra mile and pass the character the command applies to even if you're fairly sure that only the AI running the behaviour would require it.

The passed location, as stated earlier, could be inputted manually with X, Y, Z coordinates. But more likely, the location would be stored in the context as a variable by another node, obtaining the location of some game object, or building, or perhaps calculating a safe place in cover in the NPCs vicinity.

10 Stacks

When first looking into behaviour trees, its natural to constrain the scope of the nodes they use to character actions, or conditional tests about the character or their environment. With this limitation it's sometimes difficult to see how powerful behaviour trees are.

It's when it occurred to me to implement stack operations as nodes that their utility really became apparent to me. So I added the following node implementations to the game:

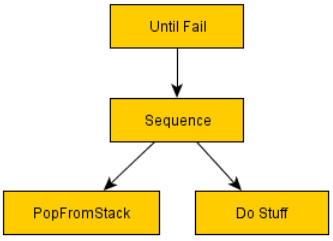
```
PushToStack(item, stackVar)
PopFromStack(stack, itemVar)
IsEmpty(stack)
```

That's it, just these three nodes. All they needed was init/process functions implemented to create and modify a standard library stack object with just a few lines of code, and they open up a whole host of possibilities.

For example `PushToStack` creates a new stack if one doesn't exist, and stores it in the passed variable name, and then pushes `item` object onto it.

Similarly pop pops an item off the stack, and stores it in the `itemVar` variable, failing if the stack is already empty, and `IsEmpty` checks if the stack passed is empty and returns success if it is, and failure if its not.

With these nodes, we now have the capacity to iterate through a stack of objects like this:



Using an Until Fail repeater, we can repeatedly pop an item from the stack and operate on it, until the point the stack is empty, at which point PopFromStack will return a fail and exit out of the Until Fail repeater.

Next, a couple of other vital utility nodes that I use regularly:

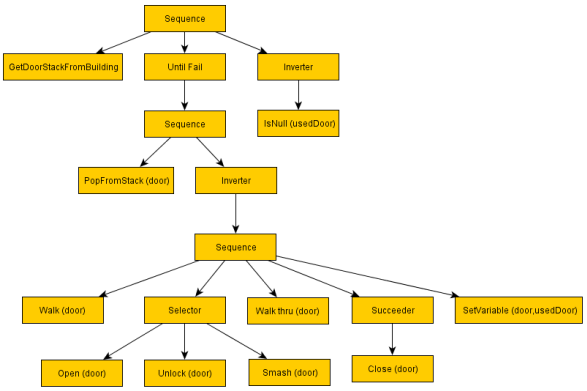
`SetVariable(varName, object)`
`IsNull(object)`

These allow us to set arbitrary variables throughout the behaviour tree in circumstances where the composites and decorators don't allow us enough granularity to get information up the tree we require. We'll hit a situation like this in a moment, though I don't doubt there's a way to organize it so it's not required.

Now supposing we added a node called `GetDoorStackFromBuilding`, where you passed a building object and it retrieved a list of exterior door objects in that building, newing and filling a Stack with the objects and setting the target variable. What could we do then using the things we've detailed above?

Eek. This has gotten a little more complicated, and at first glance it may seem a bit difficult to ascertain what's going on, but like any language eventually it becomes easier to read at a glance, and what you lose in readability you gain in flexibility.

So what does this do? Well it may be a little bit of a head mangler at first, but once you become familiar with the way the nodes operate and how the successes and failures transverse the tree, it becomes a lot easier to visualise. If necessary I may expand this section to show the walk through the tree, if my description proves insufficient.



In short, it is a behaviour that will retrieve and then try to enter every single door into a building, and return success if the character succeeded in getting in any of the doors, and it will return failure if they did not.

First up it grabs a stack containing every doorway into the building. Then it calls the Until Fail repeater node which will continue to reprocess its child until its child returns a failure.

That child, a sequence, will first pop a door from the stack, storing it in the door variable.

If the stack is empty because there are no doors, then this node will fail and break out of the Until Fail repeater with a success (Until Fail always succeeds), to continue the parent sequence, where we have an inverted `IsNull` check on `usedDoor`. This will fail if the `usedDoor` is null (which it will be, since it never got chance to set that variable), and this will cause the entire behaviour to fail.

If the stack does manage to grab a door, it then calls another sequence (with an inverter) which will attempt to walk to the door, open it and walk through it.

If the NPC fails to get through the door by any means available to him (the door is locked, and the NPC is too weak to break it down), then the selector will fail, and will return fail to the parent, which is the Inverter, which inverts the failure into a success, which means it doesn't escape the Until Fail repeater, which in turn repeats and freshly recalls its child sequence to pop the next door from the stack and the NPC will try the next door.

If the NPC succeeds in getting through a door, then it will set that door in the `usedDoor` variable, at which point the sequence will return success. This success will be inverted into a failure so we can escape the Until Fail repeater.

In this circumstance, we then fail in the `IsNull` check on `usedDoor`, since it's not null. This is inverted into a success, which causes the entire behaviour to succeed. The parent knows the NPC successfully found a door and got through it into the building.

If it failed, the same process could be repeated with a `GetWindowStackFromBuilding` node, to repeat the process with windows. Or with a little stack manipulation with a few more nodes, perhaps you could call `GetDoorStackFromBuilding` and `GetWindowStackFromBuilding` immediately after each other, and append the windows to the end of the door stack, and process all of them in the same Until Fail, assuming that Open, Unlock, Smash, Close operated on a generic base of doors and windows, or run-time type checked the object they were operating on.

Finally, you may notice I've added a Succeeder decorator parenting the close door node. This is because it occurred to me that if an NPC smashed the door, they would no doubt fail to close it.

Without the succeeder this would cause the sequence to fail before the usedDoor variable was set and move onto the next door. An alternate solution would be for Close Door to be designed to always succeed even if the door was smashed. However, we want to retain the ability to test success of closing a door (for example using the node within a 'Secure Safehouse' behaviour would deem a failure to close the door because it's no longer on its hinges as pretty pertinent to the situation!), so a Succeeder can ensure that the failure is ignored if that behaviour is required.