Behaviour Trees: What They Are and Why They are Useful

Aaron Salo, 2020

Abstract

Behaviour Trees (or BTs) were originally developed in the computer game industry to increase modularity in the control structures of Non-Player Characters (Colledanchise and Ögren 4). In this paper we discuss the benefits of BTs over similar control architectures, in an effort to illustrate places where the architecture particularly excels. BTs are a relatively new topic (especially comparing to similar architectures, such as Finite State Machines), but are currently implemented in a wide range of fields such as games, brain surgery robotics (Colledanchise and Ögren 5), and much more.

Index

Introduction

Behaviour Trees (or BTs), are a new emerging Control Architecture for AI Agents. Their ease of use, and comprehensability, among others, give them a unique edge over other architectures, such as Finite State Machines (FSMs). In a complex FSM, we generally would have one state for each behaviour, and a transition that leads us from one state to another. The problem with this is when states are frequently transitioned to, it may become harder to interpret the way the AI will deal with a given situation. BTs however flow one way down the tree to the leaves, following a “thought process” that is easy to understand. Once implemented, other designers that may not have any computer science knowledge could use a Behaviour Tree to construct a complex AI system, given a set of behaviours. Nodes and their transitions can be managed through a simple GUI, providing an intuitive interface for behaviour tree assembly which will be illustrated in the programming section of this paper.

Behaviour Trees: An Overview

So how do behaviour trees actually work, and why should anyone use them over other AI systems such as FSMs, or goal oriented action planning (GOAP). The key reason they have an edge over other systems is the custoimizability, explicitness, and variability. They are customizable, as they are broken up into small sub-components to make up larger behaviours. Replacing these or customizing these sub components allow high customizability of sub-behaviours, and as a result, entire behaviours. BTs are explicit because everything an agent does is represented by a node, clearly showing what will be run and under what circumstances. Variability is given by the control

structures, the things that direct the agent which actions to perform, given a particular situation, and that they are also represented as nodes. (Francis, 2017, 116). Another strong benefit of behaviour trees is the reusability of behaviours. For example a GoTo action could be used in many contexts, just with the destination changed, or an action that would pick up an item could be used across behaviours, but in separate contexts. We can take these sub-tasks and use them as building blocks for other behaviours, rearranging them as needed. (Ghzouli et al., 2020, 205)

How They Work

Formally, a Behaviour tree is an acyclic directed graph consisting of nodes, and transitions. A node that has a transition to another node, is a parent of that node, and the node being transitioned to is called a child. Nodes may only have one parent, but can have as many children as they want. The root node, is starting point of the tree and has no parents. The internal nodes are any node that has at least one child, and are all control nodes. The leaf nodes, or nodes with no children are all execution nodes.

The BT starts by generating a tick from the root. They then flow down the tree to their children, each node handling themselves making local decisions, or in other words; decisions that are not affected by any other node in the tree. Each of these local rules are solved within each node, and combine to solve a global problem. Similar behaviour is seen in particle systems, and other AI algorithms such as a boid flocking algorithm. When we solve these smaller problems locally, everything is simplified, and we can focus on one small problem at a time, such as the act of making coffee. When we are grinding coffee for example, we do not need to worry about anything else in the process, and we only worry about completing our assigned task, and letting the function

who called us know whether we were successful or not. The ticks that flow down the tree can be further directed with control nodes. Rather than performing an action, these nodes direct the ticks down the tree in specific ways based on what kind of functionality we are looking for. We can use these control nodes to perform actions in a specific order, or just finish the first action and move on, as well as having the functionality to change the results of action nodes (Failed actions get changed to success, and vise versa for example).

Individual behaviours can, and probably should be made up of many smaller "sub-behaviours". Lets go back to our *Making Coffee* example. First, we need to fill the coffee pot with water, place in the coffee grinds, and turn on the coffee maker. Each of these actions are sub-behaviours, that make up the whole behaviour of "making coffee". Suppose, however, the beans have not been ground yet; we then can easily add on another sub-behaviour to grind the coffee pot, or later to pour the coffee, each adding more detail and complexity to the behaviour *without* effecting the rest of the behaviour. Ideally, we can change and add new behaviours without even touching anything else. Additionally, the larger behaviour "making coffee" has no need to know the details of its sub-behaviours either, all it knows is that they must be executed in a specified order. Similarly each sub-behaviour does not need to know (or care) about previous or subsequent sub-behaviours (Colledanchise and Ögren 6). By creating these sub behaviours, rather than a larger single behaviour, we have the potential to use those sub behaviours somewhere else. In the coffee example, perhaps something specific like grinding coffee would be exclusive to making coffee, but a pour action could be used in other actions, such as pouring juice or something similar.

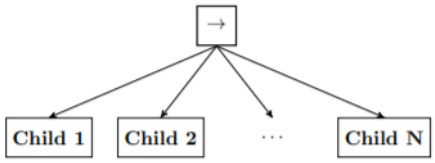
Behaviour Tree Components

There are four different categories of control flow nodes in a behaviour tree; Sequence, fallback/selector, parallel, and decorator, as well as two different execution nodes; Action, and condition.

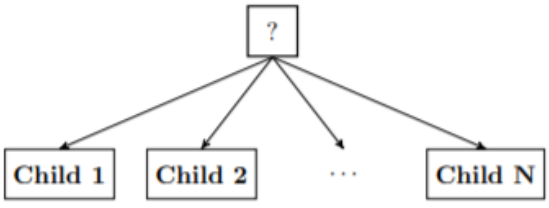
Control Flow Nodes

Control Flow Nodes, well, they control the flow of the tree. They decide where to send ticks based on the results from its children, or modify those results based on a simple set of rules.

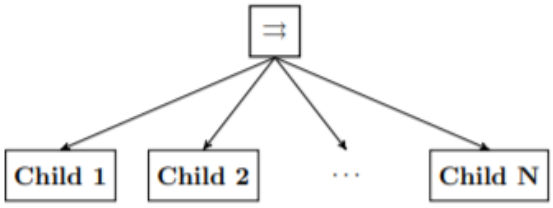
A sequence node sends ticks to its children from left to right, continuing until it receives a running, or failure tick, skipping over nodes that returned success. The sequence node itself returns success if and only if all of its children return success. It is sometimes easier to think of a sequence node as an *AND* node, and is denoted with a box containing a → (Colledanchise and Ögren 6).



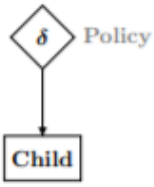
A fallback/selector node sends ticks to its children from left to right, until it finds a child that returns a success, or running and returns it immediately. If all of its children fail, it will return failure. It is sometimes easier to think of a fallback node as an *OR* node, and is denoted with a box containing the symbol “?” (Colledanchise and Ögren 7)



A parallel node sends ticks to all of its children and returns success if a certain number of its children succeed. If less that this specified number of children succeed, then the node will fail. A parallel node is denoted with the symbol "⇉".



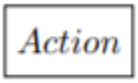
The decorator is a control flow node with a single child that manipulates its child's return status according to a user-defined rule. It also ticks its child according to some rule. The diamond's label also describes the policy the node takes. For example, if the node inverts the result of its child, it might be denoted "inverter".



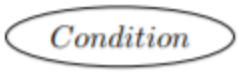
Execution Nodes

Execution nodes are the leaves of the tree, and actually hold implementation of behaviours, or checks that a certain state is true or not.

An action node will execute an action, returning success if it has been completed, running if it is running, or failure if it has failed. The boxed label describes the action the node does.



A condition node checks a proposition, returning success or failure, depending on if the proposition holds or not. As with the action node, the condition nodes' round label describes the condition being checked.



Behaviour Trees versus Finite State Machines

BTs are very similar to FSMs, and actually can be converted to an FSM that completes the same tasks. We can still have this modularity of behaviour, where we can model a behaviour with several smaller sub-behaviours, but there are clear differences. The seemingly most obvious difference is the cyclical nature of FSMs, versus the acyclical BT. Another core difference is the use of control structures such as a sequence or fallback, as nodes themselves.

While behaviour trees and FSMs can both complete the same tasks, FSMs are harder to modify and add to without analyzing and re-analyzing the structure of the FSM, which is prone to error. Many existing transitions must be re-evaluated. Some FSMs may require many states to be reused, and many transitions to and from other states which can make understanding the machine difficult. In contrast, the directed flow and ability to reuse states allow BTs higher readability and understanding even for users unfamiliar with the data structure (Colledanchise and Ögren 23 - 28).

Connecting to a Knowledge Base

Often, agents will need to know the status of something before they are to act. An agent should not try to open a door that is already opened, or shoot a gun before picking one

up, for instance. In many cases, the state of these objects will change, and evolve, and more often than not, other entities will be making those changes. So how do we ensure our agents behaving correctly in these situations? We need some kind of way to quickly and efficiently make these checks, as well as update the state in case we, or other agents need to check on the state subsequent to our actions.

A common solution to this is using a Blackboard. This is a structure that agents can ask things, as well as update other states. This Blackboard stores these values and allows us to make sure that things are going correctly, and in a way, communicate between agents and the world (Francis, 2017, 123).

Disadvantages

Although BTs are great at a lot of things, they can be difficult/complex to implement compared to their counterparts. In some systems that have somewhat straight-forward environments, a BT could be considered overly complex, and a simpler architecture would perform just as well.

Additionally, comparing to other methods BTs have a lack of study compared to more traditional methods such as FSMs, which are heavily researched, and have no lack of software and tools available for them. (Colledanchise & Ögren, 2017, 42) Online support for development of BTs is also less available.

Genetic Algorithms

In recent studies, researchers have been looking at the idea of using genetic algorithms to evolve the structure of behaviour trees. In their studies, they allowed the

algorithm to swap two random subtrees from two different parents, swapping a control node for a control node, and an execution node for an execution node (Berthling-Hansen & Morch, 22). Another study used a similar approach; swapping two nodes of similar type, but added the functionality to change variables in action nodes. This would slightly change how these actions were performed, and compared the different evolutions using fitness factors (Berthling-Hansen & Morch, 23).

These techniques are a natural extension to behaviour trees, as understanding what priority some actions should have can sometimes be rather ambiguous. They also can have the benefit of identifying ‘prunable’ branches of the tree, or branches we can remove from the tree entirely, improving performance and readability.

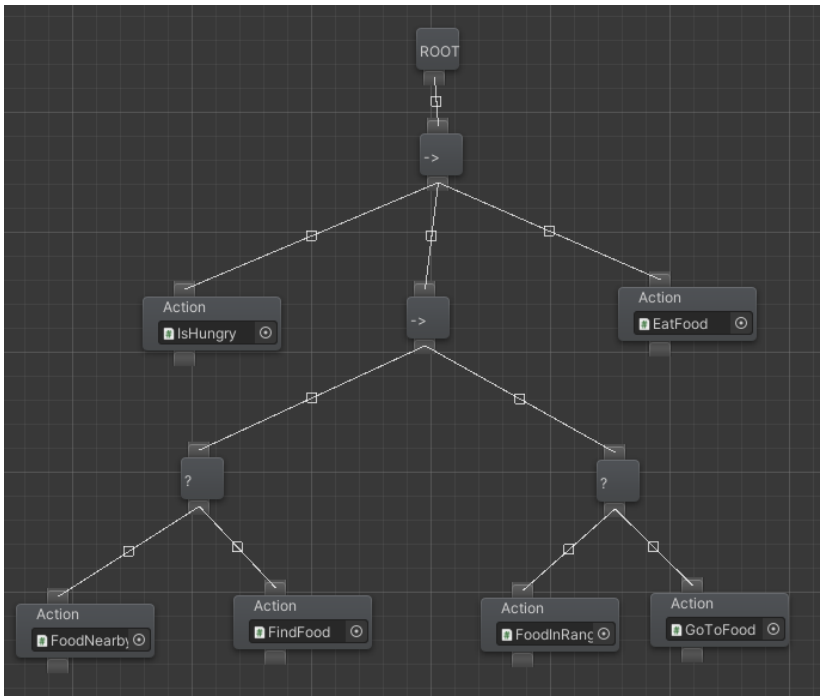
Behaviour Tree Demo Program

Program Overview

I have created a small demo program to illustrate the key benefits of using Behaviour trees. The main goal is to show how easy it is to create behaviours, as well as how easy they are to understand. Here is a link to a video walking through the creation of a tree, and its execution. In the video I have a simple task I want my agent to accomplish; eating food. I walk through different degrees of complexity, slowly adding behaviours and building up a more refined agent. In the beginning, he must be told what food item to eat before eating it, to being able to find the food on his own, to eventually being able to walk around and eat food himself. The project github can be found here. Code involving different nodes are here, and action code is here.

Program Goal

The goal of the demo was to show the ease of use and understanding of behaviour trees. I dont edit a single line of code during the video, but I am able to put together simple building blocks to create a behaviour that suits my needs. I also tried to illustrate the ability to reuse smaller subtrees for more complex actions. Although the program is simple, I believe it shows the potential behaviour trees have to create complex AI, using these simple building blocks.



- A snapshot of the final behaviour tree from the video

Bibliography

Berthling-Hansen, G., & Morch, E. (2018, July 1). *Automated Behaviour Modelling for Computer Generated Forces*. NTNU Open. Retrieved 12 06, 2020, from https://ntnuopen.ntnu.no/

Colledanchise, M. (2017). *Behaviour Trees in Robotics*. PhD thesis KTH Royal Institute of Technology. ISBN 978-91-7729-283-8

Colledanchise, M., & Ögren, P. (2017). *Behaviour Trees in Robotics and AI: An Introduction*. Chapman and Hall.

Francis, A. (2017). *Game AI Pro 360* (S. Rabin, Ed.). CRC Press. http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter09_Overcoming_Pitfalls_in_Behavior_Tree_Design.pdf

Ghzouli, R., Berger, T., Johnsen, E. B., Dragule, S., & Wasowski, A. (2020). Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. *Behavior Trees in Action: A Study of Robotics Applications*, 196-209. 10.1145/3426425.3426942

Program Sources

Character Art
Ground Tileset
Node editor adapted from