**DAC619**

**AI Assessment**

**Kevin Buchholz**

Table of Contents

[1 Behaviour Tree 3](#_Toc26454743)

[1.1 Justification 3](#_Toc26454744)

[1.2 Advantages 3](#_Toc26454745)

[1.3 Disadvantages 4](#_Toc26454746)

[2 Comparison 4](#_Toc26454747)

[2.1 Decision Tree 4](#_Toc26454748)

[2.2 Utility AI 5](#_Toc26454749)

[3 Analysis and Design 5](#_Toc26454750)

[4 Conclusion 6](#_Toc26454751)

[4.1 Strengths of Behaviour Tree 6](#_Toc26454752)

[4.2 Weaknesses of Behaviour Tree 7](#_Toc26454753)

[4.3 Improvements to the AI 7](#_Toc26454754)

[5 Appendices 7](#_Toc26454755)

[5.1 Behaviour Tree Chart 7](#_Toc26454756)

[5.2 Pseudo Code 7](#_Toc26454757)

[5.3 Test Plan 7](#_Toc26454758)

[6 References 7](#_Toc26454759)

# Behaviour Tree

## Justification

For this AI creation task, we have many different techniques on hand. We can use these following methods:

* Switch Case
* State Machine
* Decision Tree
* Behaviour Tree
* Utility AI

I choose the Behaviour Tree as my approach for the AI since it is combining many basic approaches we use in the other techniques and it is the most common approach for AI creation in today’s game industry. Likewise, it will be very useful for usage in future applications. The Switch Case and State Machine are in my opinion far too easy to implement and not the greatest option to pick here. I want to have a little challenge and learn new things while working on this assessment. The Decision Tree is a more simplified version of the Behaviour Tree without the versatile options it has to offer, so we do not choose that either. The Utility AI is maybe a more suitable option for the future of video games but currently it is too sophisticated for this project in my opinion. A single Behaviour Tree for each agent will suffice for this project.

## Advantages

Behaviour Trees offer highly reasonable and flexible options for creating more complex AI patterns. We have two core modules known as Sequence and Selector. Based on this we can build easily every pattern we want to implement for the AI. Of course, there are more optional components like the Parallel module, but we will look at this later if we need to implement this. If the Tree is clever designed, the AI can decide between different branches easily and always achieve the expected and best result. For current data context we will use a sort of cache called Blackboard. With this we can save current data contexts for each agent and easily access it later, if we need to use it in the current situation. Additionally, we are able to create a visually impressive and very understandable tree by using a graphical user interface for it. Then even non-programmers can use this method for creating an AI architecture.

## Disadvantages

On the other hand, a single Behaviour Tree can get really complicated over time when building a more and more complicated AI structure. At that point it can be very difficult to debug the AI in its entirety without breaking down each branch or action of the tree. Furthermore, even a single AI agent can be a huge performance impact on our system, if the tree is not performance friendly programmed.

# Comparison

## Decision Tree

A Decision Tree can offer many viable options in creating an AI, but its core mechanism is limited to binary operations. This means the tree can create branches by evaluating the data as “Yes” or “No” only. We have just two options for each branch here. So, each node just relies on a false or true result. In addition, we must be careful by not creating too many branches without planning it first. Even a single input change can influence the whole tree. (M. Bennet, 2019) It has to be balanced on each side of the tree to make it more stable. Instead for the Behaviour Tree we can create as many branches as we want without any particular order. We have to look how we utilise the Composite Nodes to produce an understandable and convenient process. However, a Decision Tree is highly more understandable at first glance because it does not have those complex patterns and follows an easier creation process. Therefore, it is easier to debug than the Behaviour Tree because we can determine the node, we are currently evaluating more easily than by the complex design patterns of a Behaviour Tree.

## Utility AI

The Utility AI is a very modern approach for creating complex and coherent structures. It does not follow a tree creation process. Instead the AI relies on determined desires. We set different goals for the AI that it must reach by increasing or reducing these desires. Thus, this approach can easily blend between different actions if the AI needs it in the current situation. It can look at more conditions at the same time than the Behaviour Tree. The order in which the inputs get evaluated does not matter. The Utility AI can perform any action by just looking on the given score. Hence, the AI can make decisions without any predefined routes it has to follow. The Behaviour Tree pursues the exact opposite procedure. There you have to plan every step the AI can possibly make. However, the Utility AI could evaluate actions the player does not have foreseen.

# Analysis and Design

Our AI plays the famous game mode “Capture the flag” against each other. Basically, there are three main paths the AI must follow:

* First, try to steal the flag from the enemy base
* Second, bring the stolen flag to his own base
* Third, if own flag gets stolen, chase down / attack the enemy and recover flag

We build these three paths but have to consider possible tactics and different approaches the AI can evaluate along these paths. For this we will carefully build reliable branches within the Behaviour Tree where the AI can easily switch back and forth.

We need to program its core modules which will be:

* Blackboard
* BaseNode
* CompositeNode
* LeafNode
* Decorator

As mentioned in section 1.2 the Blackboard will save contextual data, e.g. the “Enemy flag” from the perspective of each agent for later use in action nodes. We can use a Dictionary in this case, which will store a string and object pair.

The BaseNode will be our base class for every node we create. We need a node state (running, success, failure) and an evaluation method which will execute the nodes code.

The CompositeNode will be used for Sequences and Selectors. It stores a List of nodes which will be iterated consecutively. The Sequence evaluates every node until one node fails. The selector will do the opposite. It will evaluate all nodes until one success and stick to it.

The LeafNode represents our action that will be executed. Every action the AI takes is based on the LeafNode, which will return a success or failure state.

At last the Decorator holds one node in its class and modifies its outcome, like repeating or inverting the result of its given node.

For the general evaluation of each node we will use Coroutines, because they are more reliable than the usual update loop from Unity itself. We can stop and start them at will. Basically, we create our own controllable update loops. The tree’s main class will start a single coroutine that evaluates the root node. After that the Composite Node will evaluate each node in its list, which are also coroutines. From now on the tree will work its way down each branch.

In the agents AI class, we construct all node connections we need and start the root node afterwards, so the tree for each agent will be created and started.

# Conclusion

## Strengths of Behaviour Tree

The Behaviour Tree is in general a very strong tool to build different paths for the AI. It was a more or less easy process to design the tree structure for this assignment if you understood its core components. It has a fast work flow for creating branches and different paths. You can clearly set the direction in which the AI should go by setting conditions for each branch which made decisions pretty reliable by connecting different sequences with a selector.

## Weaknesses of Behaviour Tree

The most difficult part about using the Behaviour Tree was the debugging process. Whenever an agent does not execute what he was intended to do, you were not sure if the Composite Node or Action was broken. Most of the time the actions were causing errors because there were bugs in the code or the correct state was not returned. The whole process is very fragile if one action in the Composite Nodes does not function properly.

## Improvements to the AI

Due to time limitations many improvements like a role system could have been added to each agent. For each role a new branch in the tree could be created for different possible tactics the AI can choose from (for Example Flag Hunter, Supporter, Guard). Not all planned actions were implemented due to time limitations and different technical problems, so the full tree is not implemented yet.

# Appendices

## Behaviour Tree Chart

## Pseudo Code

BehaviourTree

public class BehaviourTree

{

private BaseNode rootNode;

private IEnumerator mainLoop;

public BehaviourTree(BaseNode rootNode)

{

this.rootNode = rootNode;

}

public void Traverse()

{

mainLoop = rootNode.Evaluate();

StartCoroutine(mainLoop);

}

public void StopTree()

{

StopCoroutine(mainLoop);

mainLoop = null;

}

}

BaseNode

public abstract class BaseNode

{

public enum NodeState

{

SUCCESS,

FAILURE,

RUNNING

};

public NodeState ReturnCurrentState();

public void SetCurrentState();

protected abstract IEnumerator Execute();

public IEnumerator Evaluate()

{

Assign new IEnumerator to reset it everytime;

return IEnumerator;

}

}

Composite Node

public abstract class CompositeNode : BaseNode

{

protected List<BaseNode> childNodes;

public void AddNodeToList();

}

Selector

public class Selector : CompositeNode

{

Public void Execute()

{

SetOwnStateToRunning();

for (every child in CompositeNode)

{

while (current child is running)

{

EvaluateCurrentNode();

}

if (current child successes)

{

SetOwnStateToSuccess();

Exit loop;

}

}

SetOwnStateToFailure();}

Sequence

public class Sequence : CompositeNode

{

public void Execute()

{

SetOwnStateToRunning();

for (every child in CompositeNode)

{

while (current child is running)

{

EvaluateCurrentNode();

}

if (current child fails)

{

SetOwnStateToFail();

Exit loop;

}

}

SetOwnStateToFailure();}

Decorator

public abstract class Decorator : BaseNode

{

protected BaseNode childNode;

public NodeState EvaluateChildNode();

}

Inverter

public class Inverter : Decorator

{

public Inverter(BaseNode childNode)

{

this.childNode = childNode;

}

public void Execute()

{

SetOwnStateToRunning();

EvaluateCurrentNode();

if (childNode state is failure)

{

SetOwnStateToSuccess;

}

else

{

SetOwnStateToFailure();

}

}

}

## Test Plan



# References

**GAMASUTRA, 2016. Are Behaviour Trees a Thing of the Past? [viewed 30 November 2019]. Available from:**<https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php>

**GAMASUTRA, 2014. Behaviour Trees for AI: How they work [viewed 30 November 2019]. Available from:**<https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php>

**M. BENNET, 2019. Topic 3 – Decision Trees [viewed 4 December 2019]. Available from:**<https://learn.solent.ac.uk/mod/resource/view.php?id=1350267>

**M. BENNET, 2019. Topic 5 – Behaviour Trees [viewed 4 December 2019]. Available from:**<https://learn.solent.ac.uk/mod/resource/view.php?id=1291935>

**M. BENNET, 2019. Topic 6 – Goal Oriented Behaviour [viewed 4 December 2019]. Available from:**<https://learn.solent.ac.uk/mod/resource/view.php?id=1291943>