[AI Modelling: Behaviour Trees]

Anders Tankred Holm & Mads Bøgeskov

AALBORG UNIVERSITET

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
Telephone (+45) 9635 8080
http://www.cs.aau.dk

Title:
AI Modelling: Behaviour Trees

Project theme:
AI Modelling

Project period:
DAT6,
February 1st 2010 -
June 3rd 2010

Project group:
d615a

Participants:
Anders Tankred Holm
Mads Bøgeskov

Supervisor:
Zeng Yifeng

Print run:
4

Pages:
108

Appendices (number, type):
1 DVD with source and binaries

Abstract:

The report documents the development of a set of tools and techniques, which goal is to aid the process of defining probability values in a behaviour tree. A formal model of behaviour trees is presented that forms the basis for the problem. The report presents measures for diversity and agent outcome for behaviour trees. An optimal selector probability distribution guideline is achieved by utilising linear programming using these measures, that then afterwards is used in the tool extensions. The project is then evaluated by a series of test subjects that concludes the presented techniques is beneficial.

The contents of the report are freely available, however publishing (with source) must happen only by agreement with the authors.

Preface

This report has been written during the DAT6-project period, by group D615A at Aalborg University. The main theme of the project is Artificial Intelligence modelling, with focus on development the creation of behaviour trees for an agent in a computer game. This report is addressed to other students, supervisors, and anyone else who might be interested in the subject. To read and understand the report correctly, it is necessary to have knowledge equalling a bachelor in computer science and basic knowledge concerning Artificial Intelligence. A resume of the report has been written and can be found in Appendix C on page 107.

The entire report is written in British English and no translation will be accessible. Abbreviations and acronyms will at first appearance be written in parenthesis, to avoid breaking the reading stream. Specification of gender in the report is not to be understood as suppression or any other form of political/religious position. The gender is only specified to simplify the process of writing for the authors.

References to sources is marked by [#], where # refers to the related literature in the bibliography at the end of the report.

Attached to the report is a DVD, which contains the source code created for the project, a PDF version of the report, and also the contents of the Subversion repository used.

The report is written in LaTeX and is accessible as a PDF-document, which can be read with Adobe Acrobat Reader on Microsoft Windows, or Preview on Apple Mac OS X.

Project group d615a

Anders Tankred Holm

Mads Bøgeskov

CONTENTS

CHAPTER

1

Introduction

The objective of the project is to research behaviour trees and especially the development of these in order to further ease the process of creating Artificial Intelligences (AIs) for realtime games. The report focuses on the design of tools to improve behaviour tree development, and takes it basis in a behaviour tree editing tool created by the same authors in their DAT5 semester. This tool is called the Behaviour Editor and is designed for games created using the Unity3D engine[1]. However all techniques found in the project may be implemented in any other behaviour tree editing tool as well.

The report uses the concept of an agent to describe the in-game object, e.g. Non-Player Character (NPC), that has a AI component assigned, which in this case it based upon behaviour trees. The report furthermore uses the concept of an AI designer to describe the person in the game development team assigned to creating these agents. Such an AI designer will often be a person with limited knowledge of programming, but who works in close correlation to the game programmers and game designers. However because of this the tools must be usable by a person with limited technical skills. The AI designer knows the game context such as the game conditions, goals, behaviours of other agents and so on, which is used whenever a new agent is created.

The basis of the project is that the AI designer often is in a situation where he has to assign probability values to the behaviour tree, which can be challenging assignment with more complex agents. The AI designer must find the correct balance between achieving a certain agent outcome and a level of diversity in the agent behaviours. It is the goal of the report to create tools which can aid the AI designer in defining these probability values while at the same time maximise the diversity and secure a certain agent outcome.

The report starts with a presentation of a formal model for behaviour trees, which is useful as a reference when developing generic tools. The report moves to a brief description of the Behaviour Editor tool and concludes with a description of the problem which the tools should help to solve.

The diversity chapter explains what diversity for an agent is, and why it is desirable in a game. It also presents the Shannon entropy formulae which is capable of calculating the diversity or entropy for an agent based upon the probability values in the behaviour tree.

The agent outcome chapter describes what the outcome of an agent is and how it can used to specify the final behaviour of the agent. The chapter introduces the concept of utility and explains how the expected utility of an agent can be calculated.

The next chapter describes how linear programming can be used to create a guideline for the AI designer, which maximise the entropy and maintains a defined utility value of the behaviour tree. The chapter also presents how the AI designer can be helped further by auto generating probability values, and recommendation of nodes to define.

The tool extensions chapter shows how these techniques can be implemented into an existing behaviour tree tool, by utilising them in new tool extensions that can aid the AI designer in defining the probability values.

The evaluation chapter evaluates on the presented tools and the formal model by presenting it and a demonstration game to a set of test subjects. These test subjects is used to provide constructive feedback on the problem itself, the presented tools, and the formal model. The demonstration game is used to evaluate upon the behaviour of an agent created using the presented techniques.

The last chapter of the report presents the reflection and conclusion of the project and also a section regarding any potential future work of the project.

CHAPTER

2

Behaviour Tree

This chapter presents a formal model for behaviour trees which is necessary in order to help the AI designer to create good agents for games. Using the formal model the next section introduces the Behaviour Editor which is a behaviour tree editing tool used in the project as both inspiration but also helping to design any of the tool extensions necessary to help the AI designer. The last section introduces the problem that will be researched through the rest of the report.

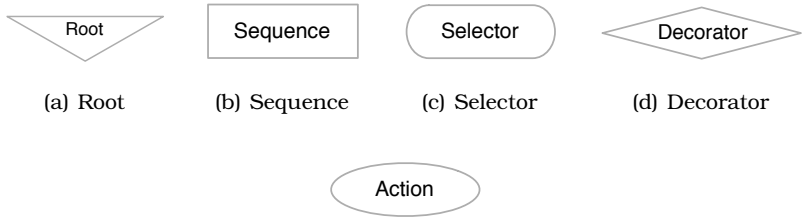
2.1 Formal Model

The concept of behaviour trees has been created to ease the collaboration between AI designers and programmers by creating a common abstraction layer. The AI designer knows the game and the purpose of the agent so he is able to create the overall behaviour scheme, and the programmer is capable of creating the actual implementation of the agents behaviours in-game. Behaviour trees uses this separation by introducing a graphical representation of the behaviours consisting of a tree structure with a small set of nodes available to create the behaviour scheme. The tree has a root node which serves as the main entry point, and it is then possible for the AI designer to expand the tree with multiple sub-trees representing certain behaviours capable of handling certain situations, and create a selection scheme that triggers these behaviours whenever a certain situation arises. The nodes available in the behaviour tree are decorators, sequences, selectors, and actions, which each adds a certain functionality to the tree structure. Most behaviour trees also contains some extra functionality to handle sudden events in the game world, e.g. when the agent is being shot at. Solutions include having multiple behaviour trees running in parallel to the main tree, which handles such events, or by using an interrupt system as was introduced in the Behaviour Editor. However this is tightly coupled with the platform of which the behaviour tree should run so this is let to the developers to chose a sufficient solution.

Both the graphical and the runtime tree structure is a connected acyclic graph, so there are certain limitations upon the design of the tree such as nodes that cannot connect to themselves and no node can have more than one parent. The behaviour tree works by first entering the node attached to the root node. This node will often start a sub-tree designed to find the appropriate behaviour for the current situation which then is executed. When the behaviour has finished executing the behaviour tree starts over and chooses a new behaviour to execute. It is important to realise that the behaviour tree does not model game states, such as win state, but only behaviour states of the agent, e.g. defend base. There is not a single way of building behaviour trees as this changes for each type of agent.

2.1.1 The Nodes

The behaviour tree model contains four different types of nodes used to create the behaviours for the agent. This section introduces the different nodes as is seen on Figure 2.1 with regard to their purpose in the behaviour tree. The different nodes can be connected by the use of the arcs shown in Figure 2.2 on the facing page.



(a) Root (b) Sequence (c) Selector (d) Decorator

(e) Action

Figure 2.1: The graphical representation of the different nodes available in the behaviour tree

(a) Standard Solid Arc

(b) Arc with Value

Figure 2.2: The two types of arcs used to denote the connections between the nodes of a behaviour tree

2.1.1.1 Root

The root node serves as the main entry point into a behaviour tree, and any behaviour tree must therefore have a root node assigned as the first node in the tree. This node cannot be deleted without deleting the entire tree. The root node can only have one child which will be the first node to be executed in the tree. The root node is illustrated on Figure 2.1(a) on the preceding page, and its connection to its child node is illustrated by a standard solid arc as seen on Figure 2.2(a).

2.1.1.2 Sequence

The sequence is capable of executing a series of sub-behaviours which in a specific order is set to perform one single grand behaviour. Figure 2.1(b) on the preceding page shows the sequence node as it is illustrated graphically in a behaviour tree editor. The node is a rectangle which can contain a word that describes the purpose of the node, e.g. Defend Base, if the sequence started a series of actions needed to defend a base. Each sub-behaviour assigned to a sequence will have an ordinal number assigned to indicate its place in the queue in the sequence.

The sequence node can have any node as its parent and also have an unlimited amount of child nodes attached of the types decorators, actions, selectors, and other sequences. The relationship is illustrated by the use of an value arc as seen on Figure 2.2(b) on the preceding page, where the value on the arc represents the ordinal number of the child node.

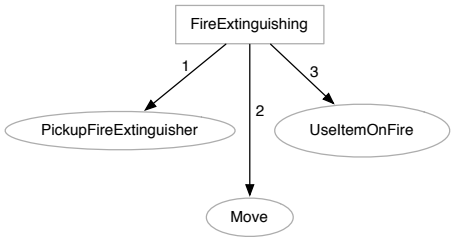


Figure 2.3: An example of the use of a sequence making an agent capable of putting out a fire

An example of the use of a sequence node would be to model the behaviour necessary to be able to put out a fire. The sequence shown on Figure 2.3 shows how this can be dissected into three sub-behaviours each handling a part of the grand behaviour. The first child node to be executed is the action node PickupFireExtinguisher which is capable of picking up a fire extinguisher. The next action node can move the agent near the fire, and the last utilises the fire extinguisher to put out the fire.

2.1.1.3 Selector

The purpose of the selector is to make the agent capable of making a decision by selecting one of its child nodes for execution depending upon the current game state. This selection is made by a selection scheme method assigned assigned to the node, such as random selection, probability selection, or priority selection. There are no finite set of selection scheme methods and the choice depends upon the situation in the behaviour tree.

As illustrated on Figure 2.1(c) on page 12 the selector is denoted by a squashed rectangle, and the word in the rectangle can contain the name of the selection scheme chosen for the node. It utilises the value arc which is illustrated on Figure 2.2(b) on the previous page to indicate a connection to a child node. The selector can have unlimited decorators, actions, sequences, and other selectors as child nodes, and any node type as its parent. The selection method chosen for the selector node depends upon the situation and the value on the arc is therefore also dependent upon this choice.

2.1.1.4 Decorator

The decorator changes or adds functionality to behaviour subtrees. The decorator is necessary to make it possible to fit generic behaviours into other locations in the behaviour tree, and also for general conditioning of execution of a behaviour. The functionality of the decorator is defined by the AI designer and there are therefore no limitations upon what it can do. The decorator is denoted by a diamond shape as shown on Figure 2.1(d) on page 12, and can contain a word that briefly explains the purpose of the node. The decorator can only have one child, which can be a action, sequence, selector, or another decorator. The decorator can only have one parent, but this can be of any node type. The decorator indicates a connection to a child node with the standard solid arc.

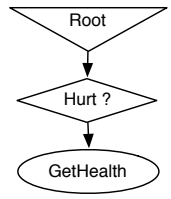


Figure 2.4: The decorator adds extra functionality to existing behaviours

An example of the use of a decorator is where an agent has a behaviour able to gather health. This behaviour should only be executed in the situation where the agent is hurt and the decorator can then be added on top of the sub-behaviour to limit access to it unless the hurt condition is filled, as illustrated on Figure 2.4.

2.1.1.5 Action

The action node represents the actual action performed by the agent, e.g. a behaviour capable of getting health will have two actions: one that finds the health source and the second that moves the agent to this. The actions are made by the programmers and then utilised by the AI designer.

The action is represented by a circle shape as seen on Figure 2.1(e) on page 12, and can contain the name of the action it performs. The action node cannot have any children as it serves as the leaf node of the behaviour tree, but it can have any node as its parent.

2.1.2 Termination Codes

The execution of the behaviour tree works by iterating down through the tree where each node will report a termination code back to its parent node. The parent node can then process the termination code and handle it according to some predefined setting. The termination code can be: Success, Failure, or Clean Failure. Success indicates that the sub-behaviour completed its execution without complications, while failure describes that the sub-behaviour did not succeeded and it has changed the starting assumptions that lead it to this sub-behaviour. The clean failure also describes that the execution of the sub-behaviour failed however the start assumptions were not compromised. An example of how this can be implemented within the different nodes can be seen in our previous report AISuite: Behaviour Editor[4].

2.1.3 Path

A path in a behaviour tree describes a route from the root of the behaviour tree and as far down as possible in the tree. That is each path in the tree represents one possible sequence of choices leading to the decision to perform a certain action, so the complete set of paths of a behaviour tree will represent all the possible decisions an agent can execute.

In the case of Figure 2.5 on the next page there are two possible paths, where one is leading to the Action 5 node through Selector 1 and Decorator 3. So this sequence of choices leads to the execution of the Action 5 action.

In certain cases a path may contain several actions, which whenever executed in a certain order will lead to the completion of a grand action. This is the case whenever one of the two paths leading to Action 4 is chosen. One of these paths are illustrated in Figure 2.6 on the facing page and the other is very similar in that it chooses Action 1 instead of Action 2. This also illustrates the case where the grand action of a sequence of actions is changed by choosing another action.

Probability

The probability of a path is a sub-problem of calculating the probability of reaching a certain node in the behaviour tree. This section will start by finding the path to the node, and then calculate the probability of this path.

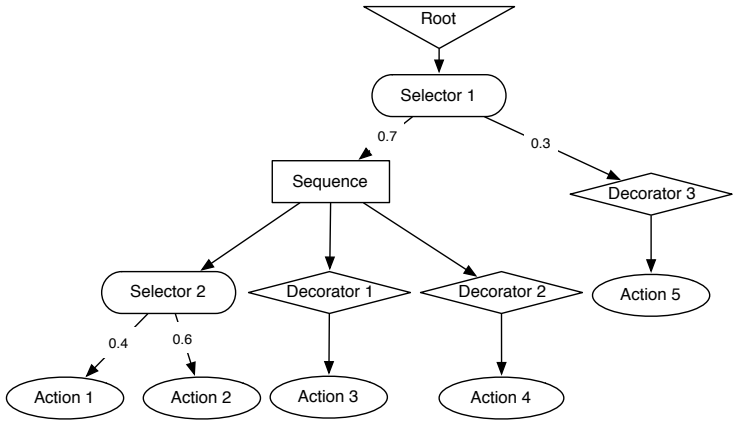


Figure 2.5: An example of a behaviour tree with three possible paths

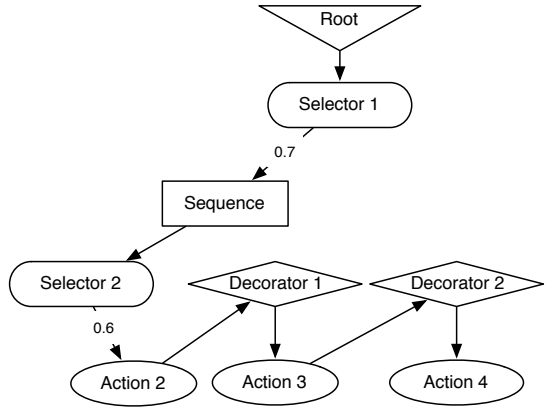


Figure 2.6: One of the possible paths in the behaviour tree shown in Figure 2.5

The probability of reaching a certain node in a behaviour tree can be calculated by finding the total probability of the path to the node. Since there can be several paths to the same node it is necessary to decide whether one needs the most or the least probable path, e.g. in the example of Figure 2.5 the probability of reaching the node Action 4 depends upon the choice taken in Selector 2 where one would chose

the right branch if the highest probable path was required, and the left otherwise. When this is decided the path to the selected node must be extracted, which following the previous example would look as illustrated on Figure 2.6 on the previous page. It is interesting to notice how the path goes through any previous branches of a sequence. The reason for this is that the probability of reaching Decorator 1 after executing, e.g. Action 2, depends upon the amount of times the Action 2 node succeeds or fails, i.e. how often the sequence continues to the next branch, and this is therefore indicated on the figure by making the path go from any leaf node of a sequence branch to the first node in the next branch.

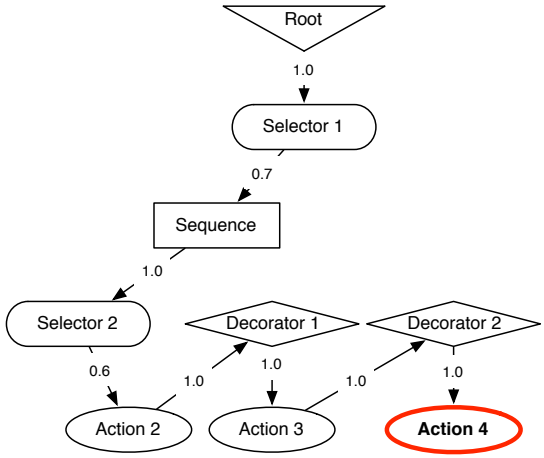


Figure 2.7: The path to the Action 4 node with all probability values assigned

As seen on Figure 2.6 on the preceding page only some of the probability values are known, and the rest must therefore be provided before the probability of reaching a node can be calculated. Two solutions exists to this problem, one being online sampling, i.e. running the tree in a test environment and record the probabilities which is time consuming or second merely define a set of preliminary probability values for each type of node which is more fit for quick prototyping.

The preliminary values is set according to how the behaviour tree will execute without any knowledge of the individual settings of the nodes. It is known that the root node always will execute its child so the probability

between the root and its child is therefore 1.0. In the sequence node there is a probability for how often a branch will lead on to execute the next branch in the sequence and it can be presumed that this will happen often since a sequence often defines a single behaviour, so the probability of reaching the next branch is 1.0. The decorator is a multi-purpose node and one of its possible uses is to limit the access to a subtree, but there is no way of knowing its purpose so the probability would also have to be set to 1.0. Continuing the example with the extracted path from Figure 2.6 on page 17 the new preliminary probability values can now be inserted into the path which is illustrated on Figure 2.7 on the facing page. The next goal is then to take the product of each probability value and calculate the probability of reaching the selected node Action 4:

$$1.0 * 0.7 * 1.0 * 0.6 * 1.0 * 1.0 * 1.0 * 1.0 = 0.42$$

Following this it can be concluded that the probability of reaching Action 4 is 42%.

2.2 Behaviour Editor

During the previous semester the authors developed a tool called the Behaviour Editor[4] which was capable of creating AI components for agents in games created using the Unity3D engine. The Behaviour Editor was based upon an implementation of behaviour trees, however was rewritten to fit the formal model in this semester. The Behaviour Editor can be seen in Figure 2.8 on the next page.

As mentioned in the formal model the Behaviour Editor includes support for interrupts to handle sudden events in the game world. Interrupts is sent by the agent to the AI component, and makes the behaviour tree switch to the corresponding interrupt tree. When the interrupt tree is done executing it will return control back to the main behaviour tree.

The Behaviour Editor is created so it fits into the existing environment of Unity3D and therefore created so it should be intuitive for existing Unity developers to use. The tool also includes functionalities expected by users of the Unity3D engine, such as an integrated inspector capable of assigning values to variables in the scripts assigned to the nodes. Scripts can be created for decorators, selectors, and action nodes so the behaviour tree can behave to the wishes of the programmer and AI designer.

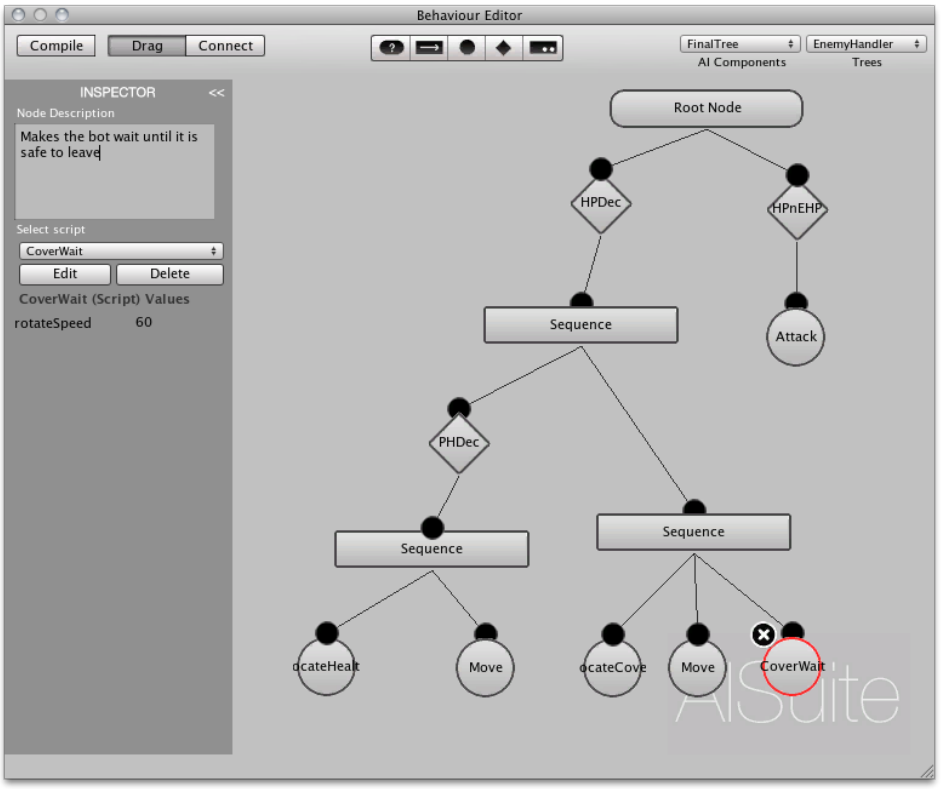


Figure 2.8: The Behaviour Editor

2.3 Problem

When the behaviour tree of an agent has been created the AI designer has to assign probability values to the probability selectors. As previously said the selectors can use other selection schemes than probability, however this report focuses upon optimising the use of probability selectors. Each of the probability value determines the final behaviour of the agent and is thereby able to shift the final agent outcome. This means that in order to achieve the wanted behaviour the AI designer has to perform a lot of time consuming tests to figure out how a certain change in probability value also changes the final outcome of the agent. It follows logically from this that the more advanced behaviour, the more immense the behaviour

tree will become which also leads to more probability values that must be assigned. In an immense behaviour tree even the slightest change in a probability value could shift the whole behaviour into another direction than wanted so the AI designer will often end up in a situation where one change created a ripple effect onto many other selector nodes in the tree. The wanted behaviour is furthermore a balance between the agent trying to fulfil its goal, but also to do it in such a way that it does not choose the same action over and over again but tries other actions to secure a certain level of diversity. This level of diversity is again a balance between keeping the final outcome of the agent and to try new tactics once in a while. Because of this it would be beneficial to help the AI designer to assign these probability values by providing a set of extensions to a behaviour tree editing tool that in combination with a set of guidelines would ease the process.

As said the specification of probability values depend upon a combination of the outcome of the agent and a desired level of diversity. Since no formal measure of diversity for behaviour trees exists as well as a measure of agent outcome these has to be defined. With a definition of these measures it would be possible to calculate an optimal probability distribution for the paths of the behaviour tree, which again also can serve as a benchmark or guideline for the AI designer. These benchmarks can be included into a behaviour tree editing tool where it in combination with a graphical representation of the behaviour tree, and other tool extensions, can help the AI designer to assign the probability values.

Summary

This chapter has presented the formal model of behaviours trees, which includes a presentation of the nodes in the tree, and some concepts necessary to know when working with behaviour trees such as path and probability calculation. The chapter ends in a presentation of one of the major hurdles when creating a good agent, which is to define good probability values for the selector with the probability selection scheme assigned in the behaviour tree.

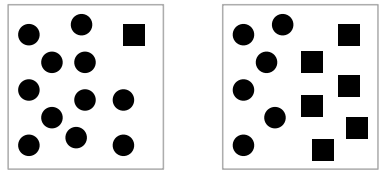
CHAPTER

3

Agent Diversity

This chapter introduces the concept of diversity for behaviour trees by finding a known measure tested to work in other situations. This measure is then tested to see if it can be adapted to be used with behaviour trees. In order to do so the first section presents the concept of entropy normally used to describe diversity in other areas such as information theory. This leads to another section that introduces the correlation between diversity and entropy and makes it possible to in the last section to show how to use this measure to calculate the level of diversity in a behaviour tree. The sources for this chapter is an article on Entropy from Wikipedia [8] and a text from MIT called Principle of Maximum Entropy: Simple Form [6].

The chapter uses an example throughout the sections to illustrate the concept of diversity. The example represents the set of possible behaviours a given agent can execute by shapes, which are grouped in a box where the quantity of each depends upon the probability of these being chosen in a random pass of the assigned agent type. Figure 3 illustrates the example by having two boxes which each represents a different agent, but where both has two types of behaviours available. It can be observed that the diversity in Figure 3.1(a) is very limited as there only is a small probability that one of the available behaviours will be executed, opposite Figure 3.1(b) where the diversity is high as the probability of executing either behaviour is equal.



(a) Less diverse (b) Very diverse

Figure 3.1: The boxes illustrates the concept of diversity for two different agents

3.1 Entropy

This section introduces the concept of entropy which is used to measure the uncertainty of a given environment, and has long been used in the fields of thermodynamics and information theory.

Entropy is used to quantify the uncertainty related to the expected value of a variable. In the example of a text string only consisting of A's the entropy would be very low as it is easy to predict the next character in the string, but e.g. a completely random string would have a high level of entropy.

A common measure of entropy is the Shannon entropy which is used to express the entropy of an environment based upon the information available regarding this environment. The Shannon entropy is denoted by an H and the diversity of an environment is written $H(p_1, p_2, p_3.., p_m)$ where p is the probability of a certain action in the environment is executed.

So to follow the previous example of the agent in Figure 3.1(a) on the previous page the environment would be expressed as $H(\frac{1}{12}, \frac{11}{12})$, and the environment for the agent in Figure 3.1(b) on the preceding page would be $H(\frac{6}{12}, \frac{6}{12})$. It is obvious that the maximum entropy can be denoted as the uniform distribution of probabilities as $M = \frac{1}{M}$ as this always provides the highest level of uncertainty.

$$H(X) = -\sum_{i=1}^{M} (p_i \cdot log_2(p_i)) \tag{3.1}$$

The Shannon entropy formulae is shown in Equation (3.1). This formulae calculates the binary entropy, but could be changed to the natural entropy or digit by changing the log_2 to either log_e or log_{10}. The difference between these is merely the range of the the diversity.

3.2 Entropy to Represent Diversity

This section presents the analogy between the two concepts entropy and diversity. Consider an agent with two possible actions available, each with probability $p_1 = 0.75$ and $p_2 = 0.25$. The Shannon entropy of this agent can be calculated according to Equation (3.1)

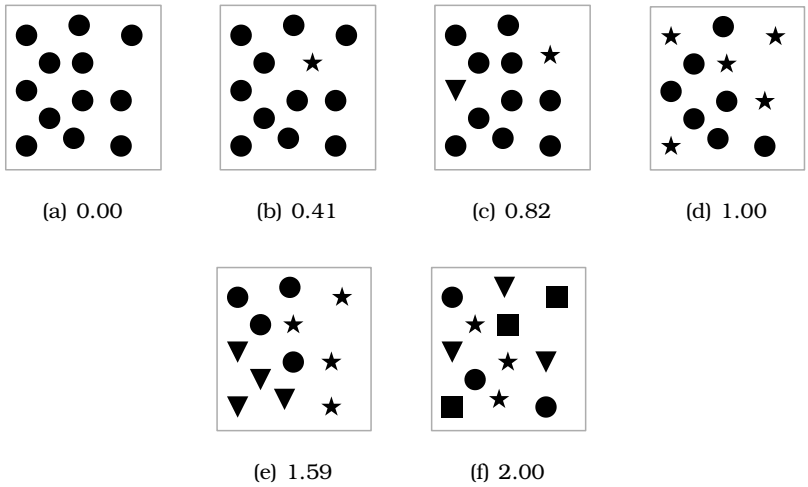
$$
\begin{aligned}
p_1 &= 0.75 \\
p_2 &= 0.25 \\
H(R) &= -\sum_{i=1}^{2} p_i \cdot log_2(p_i) \\
&= -((p_1 \cdot log_2(p_1)) + (p_2 \cdot log_2(p_2))) \\
&= -((0.75 \cdot log_2(0.75)) + (0.25 \cdot log_2(0.25))) \\
&= 0.811
\end{aligned}
$$

The Shannon entropy of this agent is 0.811 which means the agent has a low level of entropy which fits with the fact that the agent has a low level of diversity, seen by the fact that one of the actions always would be chosen 75% of the time.

It is obvious that the entropy of an homogeneous group is trivial, which is illustrated by the following example with an agent that only has one possible action that by the definition has a probability of one:

$$
\begin{aligned}
p_1 &= 1 \\
H(R) &= -\sum_{i=1}^{1} p_i \cdot log_2(p_i) \\
&= -(p_1 \cdot log_2(p_1)) \\
&= -(1 \cdot log_2(1)) \\
&= 0
\end{aligned}
$$

An entropy of zero means that it is possible always to predict the outcome of the environment in question, and it follows from this that the diversity of the agent is non-existent. Figure 3.2 on the next page illustrates six more examples of entropy calculations on different sets of agent behaviour distributions.



(a) 0.00 (b) 0.41 (c) 0.82 (d) 1.00

(e) 1.59 (f) 2.00

Figure 3.2: Example of how the entropy value shows the diversity of different sets of agent behaviour distributions

From these examples it is possible to state that H is minimised in homogeneous situations like Figure 3.2(a). Opposite this H is maximised when there are an equal probability of reaching behaviours, as seen on (f) in Figure 3.2.

3.3 Diversity in Behaviour Trees

The diversity was in the previous section set to be dependent upon the probability of a certain action being selected, however behaviour trees does not have the concept of a single action since it often has multiple action nodes that goes together to perform one grand action. Therefore it is necessary to employ the concept of paths to represent the actions in behaviour trees, and use the probability of each path as the basis for the definition of diversity. As was discussed in Section 2.1.3 on page 16 a path in the behaviour tree represents one possible sequence of choices and can therefore be analogous to an action as used in the previous sections.

Based upon this the Shannon entropy formulae from Equation (3.1) can be used to calculate the level of diversity of a behaviour tree based upon the probabilities of each possible path of the tree. Figure 3.3 illustrates the behaviour tree which is used as the example.

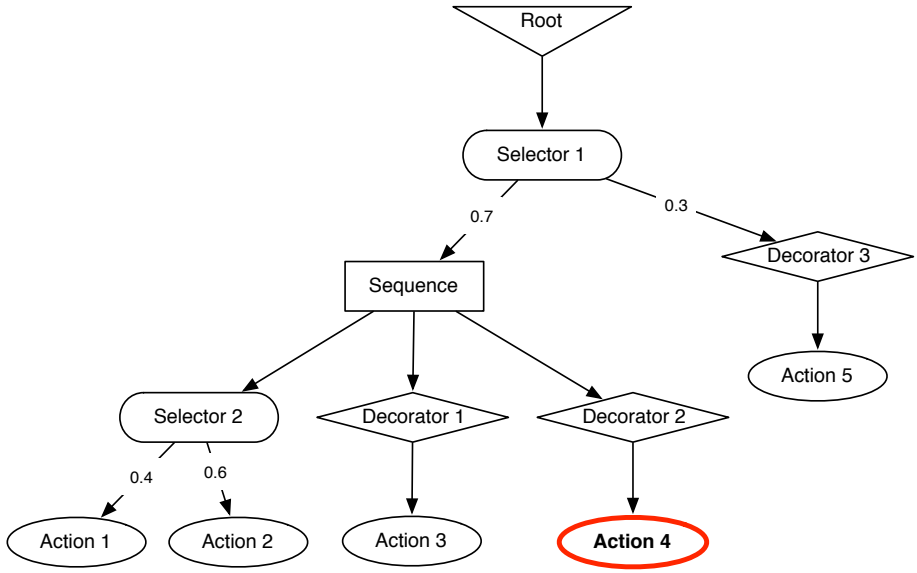


Figure 3.3: The behaviour tree used to illustrate how to calculate the diversity of a behaviour tree based upon entropy

The behaviour tree illustrated on Figure 3.3 has three paths shown in Table 3.1.

Path	**Probability**
Root, S1, D3, A5	0.30
Root, S1, S, S2, A1, D1, A3, D2, A4	0.28
Root, S1, S, S2, A2, D1, A3, D2, A4	0.42

Table 3.1: The paths from Figure 3.3 and their probability value

These probability values can then be inserted into Equation (3.1):

$$\begin{aligned} H(R) &= -\sum_{i=1}^{5} p_i \cdot log_2(p_i) \\ &= -((p_1 \cdot log_2(p_1)) + (p_2 \cdot log_2(p_2)) + (p_3 \cdot log_2(p_3))) \\ &= -((0.30 \cdot log_2(0.30)) + (0.28 \cdot log_2(0.28)) + (0.42 \cdot log_2(0.42))) \\ &= 1.561 \end{aligned}$$

The result is that the entropy of this behaviour tree is 1.561, which means it has a high level of entropy. The root starts with a selector that chooses the left branch 70% of the times, and the second selector in the left branch is nearly uniform distributed with a 40/60 distribution, hence a high diversity. The right branch of the tree has one path, so the distribution between the paths are close, which fits with the calculated entropy value.

By changing the probability values of Selector 1 to increase the probability of selecting the right branch the behaviour tree should provide a lower entropy value as the diversity decreases. In this example the probability of Selector 1 is set to 0.2 for the left branch and 0.8 for the right branch which changes the probability values of the entire behaviour tree to as shown in Table 3.2.

Path	**Probability**
Root, S1, D3, A5	0.80
Root, S1, S, S2, A1, D1, A3, D2, A4	0.08
Root, S1, S, S2, A2, D1, A3, D2, A4	0.12

Table 3.2: The paths from Figure 3.3 on the previous page after having changed probability values

These values can again be inserted into (3.1) to calculate the entropy:

$$\begin{aligned} H(R) &= -\sum_{i=1}^{5} p_i \cdot log_2(p_i) \\ &= -((p_1 \cdot log_2(p_1)) + (p_2 \cdot log_2(p_2)) + (p_3 \cdot log_2(p_3))) \\ &= -((0.80 \cdot log_2(0.80)) + (0.08 \cdot log_2(0.08)) + (0.12 \cdot log_2(0.12))) \\ &= 0.916 \end{aligned}$$

The entropy is now 0.916 which as expected is significantly lower than in the previous case which indicates that the diversity of the behaviour tree has lowered.

Summary

This chapter introduced the concept of entropy and illustrated its use in behaviour trees. The definition of diversity for behaviour trees is based upon paths to represent the concept of actions for behaviour trees and it is therefore an equal probability distribution of paths that would provide the highest amount of diversity for an agent using behaviour trees. The diversity of a behaviour tree is then calculated using the Shannon entropy formulae based upon the probability distribution of the paths in the behaviour tree.

CHAPTER

4

Agent Outcome

The outcome of an agent can be described as the impact the agent has on the game world, e.g. the outcome of a killer-agent would be a game world with possibly less alive agents. So the goal when creating a new agent is to define the outcome of the agent based upon its purpose in the game world. This outcome depends upon which actions of the behaviour tree are executed when and thereby it depends upon the probability values of the selector nodes, where a change could mean that the agent loses more often, acts less intelligently, or maybe becomes to difficult for the player. The outcome is measured using a concept known as utility values which is described in the first section. The next section introduces utility value types which is a property to which the AI designer can relate when assigning the utility values, and the last section shows how to calculate the expected utility for a behaviour tree. This chapter uses a section on utility from the book Artificial Intelligence: A Modern Approach [7] as its source.

4.1 Utility

Utility is a measure normally used to quantify the concept of "the quality of being useful" for an agent in a certain state. This concept can also be used as the basis for a measure of the outcome of an agent or just a single action. The utility value must be assigned to each path and quantifies some property e.g. the cost of the path. However it can be difficult to find the correlation between a certain path and the context of the game so to let the AI designer assign utility values to each path would be difficult. A better solution would be for the AI designer to assign utility values to each leaf node instead, as they are easier to fit into the context of the game, and thereby also easier to quantify. The utility value for the path would instead be the summation of the utility values of its action nodes.

Figure 4.1 illustrates how the utility values can be assigned to a behaviour tree where the utility value type is set to cost. With this it is possible to find the utility values of the paths as seen in Table 4.1.

Table 4.1: The paths and their utility values from Figure 4.2 on page 34

Path	**Utility Value**
Root, S1, D3, A5	5
Root, S1, S, S2, A1, D1, A3, D2, A4	40
Root, S1, S, S2, A2, D1, A3, D2, A4	50

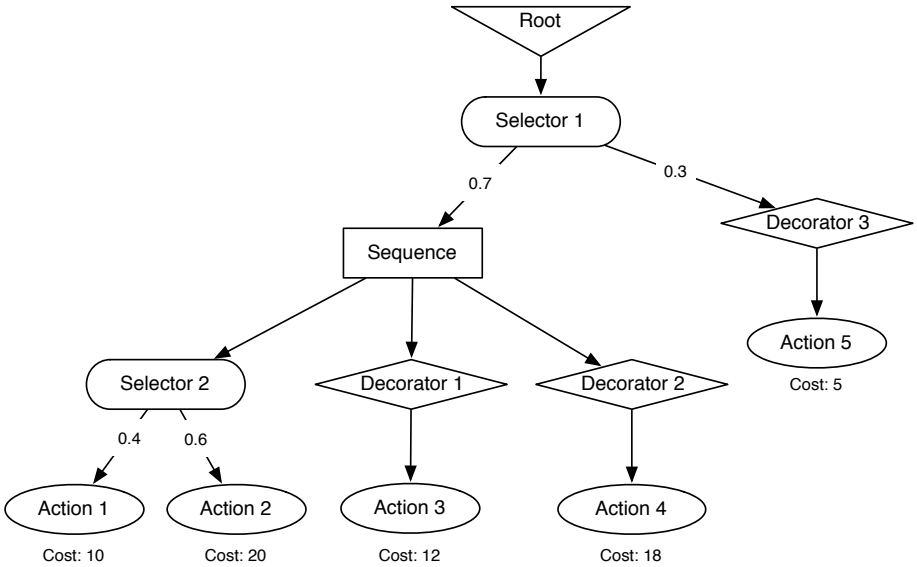


Figure 4.1: A behaviour tree with the utility value type cost and with utility values assigned to each action node

4.2 Utility Value Type

Before the utility values can be applied to the behaviour tree the AI designer must specify some property to which the outcome of the agent can relate to. However behaviour trees can be used to construct any type of agent with any type of purpose so there do not exist one common fitting utility value type. Instead it is up to the AI designer to analyse the wanted outcome of the agent in order to extract some property that he finds describes this outcome. This property can then be used as the utility value type of the agent.

Below is series of different utility value types that can be used as inspiration when the AI designer has to choose a utility value type. However these are only suggestions made by the authors and should only serve to illustrate the meaning behind creating custom utility value types.

4.2.1 Aggressiveness

In combat games such as first person shooters, the opponent agents could be defined by the amount of their aggressiveness towards the player. For instance an agent could have an high level of aggressiveness and could therefore often choose to storm the player, where a less aggressive player could choose to take a more defensive position.

4.2.2 Pace

In sport games the AI designer could choose a utility value type such as the pace of the agents on the field. The AI designer could then specify how much the pace is increased or decreased by the different actions taken.

4.2.3 Experience Points

In role playing games the terminology experience points is a well known concept, and this kind of value type would make sense for these kinds of games. This value could be used by the game logic in order to increase the experience points for the agent and therefore increase the level. This could also provide the AI designer with an overview on the expected amount of experience points the agent would receive by executing a certain sub-tree.

4.2.4 Cost

The utility value type cost requires the AI designer to consider what the cost of performing an action might be. The advantage of using the cost value type is that it can be fitted to almost any type of agent, but the disadvantage is that it also can be difficult to assign a value. The reason for this difficulty is that it is not always obvious what the ratio of cost between two actions might be. An example of the use of the cost utility value is illustrated on Figure 4.2. The goal of this selector is to choose a way to get from point A to point B, and the cost of doing so by e.g. walking is more expensive than running since that takes a longer time. However this could also be the other way around since running requires more energy than walking.

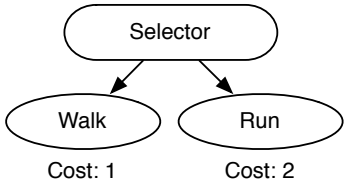


Figure 4.2: An example behaviour tree where the two actions nodes has had a cost utility value assigned

4.2.5 Difficulty

The difficulty utility value specifies a value which represents the amount of difficulty an action supplies to the environment. Figure 4.3 shows an example of the use of the difficulty utility selector, where the agent must choose between using either a knife or a rifle to attack an enemy. The knife will most likely provide a lower difficult to the environment, whereas the rifle will provide a higher amount.

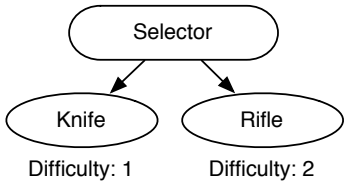


Figure 4.3: An example behaviour tree where the difficulty utility value has been assigned to each action

4.3 Expected Value

When all of the utility values of the agent has been defined it is possible to calculate the expected utility of the whole behaviour tree. The expected value can be a useful indicator to see how much a certain behaviour tree averagely provides in the defined amount of utility. This is useful when modelling an agent as this provides continuous information regarding the agent outcome. For instance if the utility value type was difficulty the expected utility would show how difficult the agent is, and each change in an utility value or probability value would change this.
The expected value for utility values is defined by

$$E(X) = \sum_{x \in \Omega} x \cdot m(x) \tag{4.1}$$

where Ω is the sample space and $m(x)$ is the distribution function. This formula can also be used to calculate the expected utility value of a behaviour tree based agent based upon a AI designers specified utility values. For instance the expected utility value for Figure 4.1 on page 32 would be calculated like this:

$$\begin{aligned} E(X) &= \sum_{x \in \Omega} x \cdot m(x) \\ &= (0.8 \cdot 5) + (0.08 \cdot 40) + (0.12 \cdot 50) \\ &= 13.2 \end{aligned}$$

The expected cost of this behaviour tree is 13.2, but if one of the probability values were changed it would also change the expected cost value, so e.g. by altering the values it could be possible to minimise the in-game cost of executing the agent.

Summary

This chapter introduced a measure of the agent outcome by the use of utility values. The chapter explained how to assign these values by first finding some property to be the utility value type and provided a set of utility value types as inspiration for the AI designer. Lastly it showed how to calculate the expected utility of a behaviour tree, which is useful when assigning probability values.

CHAPTER

5

Probability Value Guideline

The diversity of behaviour trees was in Chapter 3 on page 23 defined to be dependent upon the probability distribution of the paths, which represents the full set of possible actions to execute in a behaviour tree. So if it is possible to find the probability distribution of the paths where the Shannon entropy formulae would return a result larger than any other probability distribution, and make the tree conform to this distribution one would achieve the most diverse behaviour selection as possible. However the probability distribution should also ensure that the amount of the AI designer specified utility, as defined in Chapter 4 on page 31, is inside a specified range to ensure a certain outcome of the agent. A probability distribution that conforms to these requirements would provide the AI designer with a guideline when assigning probability values to the selectors.

The problem described above can be solved using a mathematical technique called linear programming, which is explained in Appendix A on page 79. The first section presents the linear program needed to solve this problem, but also shows why this not is a trivial linear program. The second section explains how this probability distribution is a step in helping the designer to assign the probability values of a behaviour tree. The last section uses sensitivity analysis to recommend a node to the AI designer, that if defined would have the largest impact on the path equations than any other probability value would have.

5.1 The Diversity-Outcome Linear Program

Based upon the description of linear programming from Appendix A on page 79 it is possible to outline a linear program capable of finding the maximum diversity and outcome probability distribution of a behaviour tree. The objective of the linear program is to achieve the maximum diversity based upon constraints set by the AI designer so the objective function is set to be the Shannon entropy formulae from Equation (3.1). The first constraint of the linear program would be to ensure the AI designer

specified outcome of the agent. To ensure this the linear program must have two constraints that defines the expected utility of the behaviour tree to be above the minimum range value V_1 and below the maximum range value V_2 both set by the AI designer. The linear program should also have constraints that ensure the standard constraints of probability, such as the probabilities total is one and that each probability value is above zero and below one. All of this is put together represents the linear program capable of finding the probability distribution that maximises diversity while keeping the agent outcome:

$$
\begin{aligned}
\text{Maximise} \quad & -\sum_{i=0}^{n} (p_i \cdot log_2(p_i)) \\
\text{subject to} \quad & \sum_{i=0}^{n} (p_i \cdot U_i) \geq V_1 \\
& \sum_{i=0}^{n} (p_i \cdot U_i) \leq V_2 \\
& \sum_{i=0}^{n} p_i = 1 \\
& p_0, ..., p_n \leq 1 \\
& p_0, ..., p_n \geq 0
\end{aligned}
$$

Where U_i is the utility value of the path. This linear program does not conform to any of the linear programming canonical forms but could easily be converted to do so.

As mentioned in the introduction this problem cannot be trivially applied to linear programming as the objective function is logarithmic and not linear. However since the constraints of the linear program are linear it is possible to use a method from the article Maximum Entropy Signal Restoration with Linear Programming [5]. This article presents a solution to a problem equivalent to this by making a piecewise linear approximation of the non-linear objective function. The approximation is illustrated in Figure 5.1 on the facing page where it is obvious that this would not create an exact solution however this is not required for this problem. This solution is therefore also applicable for this case by replacing the logarithmic objective function with the linear approximation. However we refer to the article for a detailed walkthrough of how to create this approximation as it is not the focus of the project.

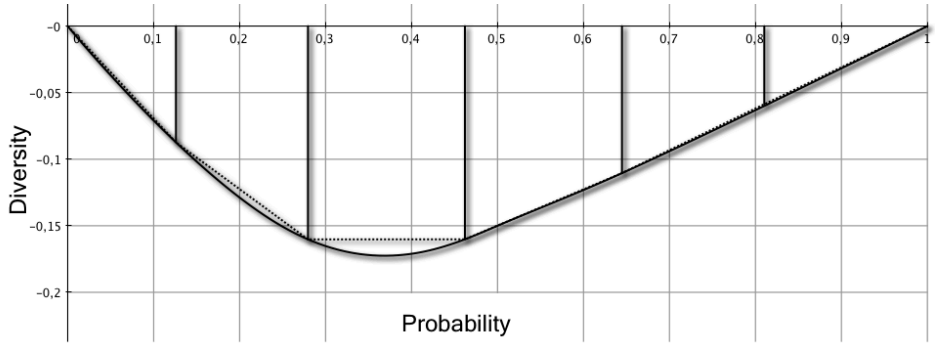


Figure 5.1: Illustrates how the piecewise approximation could look on the Shannon entropy formulae on a behaviour tree with only one path. This is of course a trivial example as this path always would have a probability of 1.0. The dotted line is the approximated piecewise linear function of the logarithmic solid line

5.2 Probability Distribution

By solving the linear program described in Section 5.1 on page 37 with the AI designers agent outcome requirements it is possible to get the optimal probability distribution for the behaviour tree with regard to diversity and agent outcome. The next step is to use this information to find the probability values for the selector nodes in the tree. Given that the behaviour tree can become quite immense the equation for calculating these probability values becomes increasingly complicated based upon the vast amount of unknown variables. To calculate the probability distribution would be time consuming and the result would be difficult for the AI designer to understand and make it harder to tweak the values of the behaviour tree. A solution is to involve the AI designer in the process by allowing him to assign probability values to the selector nodes in the behaviour tree. This would allow him to apply some of his own knowledge regarding how the behaviours in the behaviour tree should be selected which should be retained by the final result.

$$\begin{aligned} P_1 &= P_{11} \cdot P_{12} \\ P_2 &= P_{11} \cdot P_{13} \\ P_3 &= P_{22} \cdot P_{21} \\ P_4 &= P_{22} \cdot P_{31} \\ P_5 &= P_{22} \cdot P_{32} \end{aligned} \tag{5.1}$$

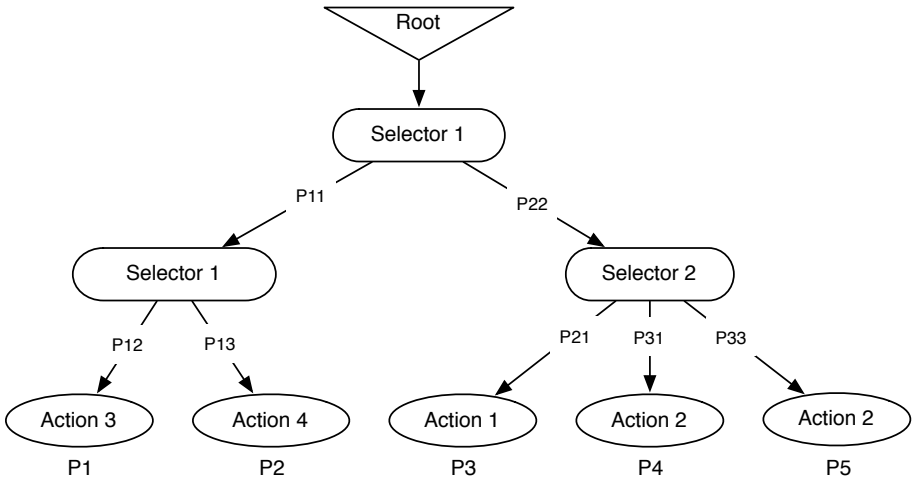


Figure 5.2: A behaviour tree where the probability distribution of the paths P1, P2, P3, P4, P5 has been found

Based upon the behaviour tree on Figure 5.2 it is possible to create the equations shown in 5.1 on the previous page. The left side of the equations are the path probabilities values received by the linear program, and the right side are the probability values for the selector nodes in their respective paths. It is obvious that it would be trivial to calculate the rest of the selector probabilities if only P_{11} and P_{22} were given by the AI designer, so the goal is to get him to assign enough probability values to make it possible to calculate the rest.

The AI designer will often have a crude idea about what nodes should be executed more often and their relationship in order to keep the behaviour of the agent in a certain way. In the case of Figure 5.3 on the facing page the designer could require that the get flag action is called more often in order to get a more aggressive behaviour, and since the AI designer knows this he could deduct a fitting probability value for the first selector node. The tool could then utilise this information to calculate the rest of the probabilities.

Based upon this the goal is to get the AI designer to help with the assignment of some of the probability values. The first solution to get his input would be to present a list of equations like the one on 5.1 on page 39, which shows the result of the linear program and then a list of unknown probability value variables. The AI designer could then specify the probability values he is able to deduct. As soon as a sufficient amount of probability values is specified the tool could calculate the rest of the probability values. This is however not an optimal solution as the behaviour tree quickly becomes quite immense, and the AI designer would have a difficult time finding the analogy between the equations and the game. A better way would be to introduce it in a more graphically pleasing way, and still use the AI designers knowledge about the game to solve the equations. The probability distribution for the paths could be illustrated in the behaviour tree editing tool as a guideline for the AI designer when he is applying probability values to the behaviour tree, and thereby help the tool to find rest of the probability values in the behaviour tree.

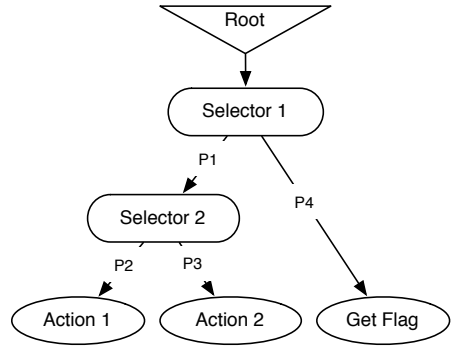


Figure 5.3: By providing any probability value it would be possible to calculate the rest of the probability values

5.3 Recommended Node

The previous section described how the AI designer is able to define probability values and thereby able to solve the set of equations representing the paths in the behaviour tree given the path probability distribution by the diversity-outcome linear program. However the AI designer does not know all of the probability values so it would be beneficial to get the tool to recommend a node that if defined would have the largest impact on the system of equations with regard to the path probability distribution.

The problem of finding this node can be solved by using a mathematical technique called sensitivity analysis. Sensitivity analysis is a useful technique for computer modellers and is among other things used to support decision making of recommendation for decision makers. There are several possible methods available to perform sensitivity analysis where the most used are; sampling, screening and local methods. The source for this section is the book Practical Optimization [2], that introduces sensitivity analysis and how the technique is used to determine how different independent values can impact a particular dependent variable.

The local method has been chosen to solve the problem of finding the node that impacts the equations the most. The local method provides the solution by finding the derivative of the dependent output factor Y with respect to an independent input factor X_i:

$$| \frac{\partial Y}{\partial X_i} | \tag{5.2}$$

The output and input factors must be specified in order to adopt the local method to the world of behaviour trees. The left side of the equations are as said the probability value of the path provided by the linear program and the right side are the probability values of the selectors in each path, so the path probability value is in this case a dependent variable and can therefore be used as the output variable in the local method. The input factor is then set to be the right side of the equations which contains the independent probability values of the path.

To illustrate how this technique can be used in behaviour trees to aid the AI designer in selecting a probability value to define an example must be presented. When the tool has completed the diversity-outcome linear program and the AI designer has defined some initial probability values in the behaviour tree he can ask the tool to show him the recommended node. The tool then utilises the result from the linear program and any probability values defined by the AI designer to create the equations. Figure 5.4 on the facing page illustrates a behaviour tree where each probability value has a variable that denotes its value. In this particular example the AI designer might have a good idea on the probability distribution for $P21$ and $P22$ and has therefore defined these. The AI designer is afterwards uncertain of what probability values to define next and therefore needs the tool to recommend a node. The problem can be set up as the following equations:

$$
\begin{aligned}
P1 &= P11 \cdot P21 \cdot P31 \\
P2 &= P11 \cdot P21 \cdot P32 \\
P3 &= P11 \cdot P22 \cdot P41 \\
P4 &= P11 \cdot P22 \cdot P42 \\
P5 &= P11 \cdot P22 \cdot P43 \\
P6 &= P12
\end{aligned}
$$

In this example it is not needed to calculate and compare all of the probability factors but instead it is sufficient to take a subset of them. The AI designer is in doubt of which subtree of Selector 2 to define. Because of this it is only required to compare factors from each subtree, and in this case $P1$ is chosen from the left branch and $P3$ from the right branch. These variables can be chosen randomly from each subtree as the comparison is made between the subtrees and not the children in the

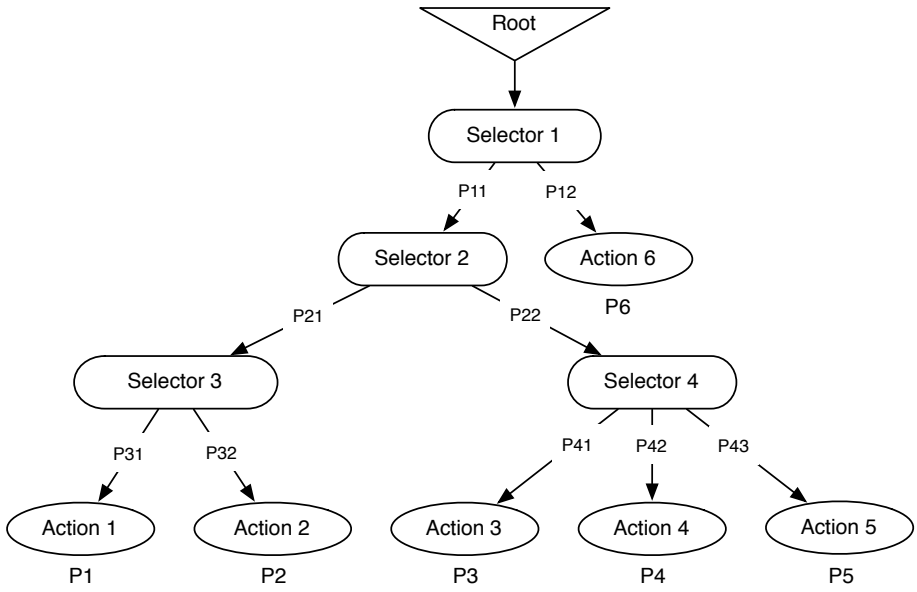


Figure 5.4: An illustration of a behaviour tree used as an example for the sensitivity analysis

subtree. The output factor for the sensitivity analysis derivation is the path probability, and the input factor is the probability factor for which the influence needs to be analysed.

$$\begin{aligned} \frac{\partial P1}{\partial P31} &= P11 \cdot P21 \\ \frac{\partial P3}{\partial P41} &= P11 \cdot P22 \end{aligned}$$

The result from sensitivity analysis can be used to compare the two probability factors and determine which one impacts the most. Both $P1$ and $P3$ is influenced by $P11$ but each of them is also influenced by a second factor. It can therefore be said that if

$$P21 > P22 \tag{5.3}$$

then $P41$ impacts $P3$ less than $P31$ impacts $P1$. Since the AI designer as previously mentioned has defined $P21$ and $P22$ the tool can suggest that the recommended node is Selector 3 and the AI designer can then choose to define it.

In this example the AI designer would have defined enough variables in order to calculate the rest of the tree but the technique scales to larger trees. In a larger tree the tool would use all the defined variables by the AI designer to find the node that impacts the behaviour tree the most. It it however not always possible to recommend a node for a AI designer and given the amount of defined variables and the structure of the tree, there could exist multiple solutions for a recommended node. One of these nodes can then chosen to be presented to the AI designer.

Summary

This chapter showed how the diversity-outcome linear program could find the path probability distribution that maximises diversity while keeping the agent outcome according to the AI designers wishes. By combining the path probability distribution with the game context from the AI designer it was shown how it would be possible to calculate probability values based upon this. Lastly it was shown how sensitivity analysis can be used to recommend a selector node that if defined impacts the system of paths the most.

CHAPTER

6

Tool Extensions

The path probability distribution, expected utility, and diversity provides information about the behaviour tree that combined with the knowledge of the game context can help the the AI designer to create the optimal behaviour selection scheme. This combined information can serve as the basis for tool extensions for a behaviour tree editing tool such as the Behaviour Editor [4], that serves to guide the AI designer to create the optimal behaviour selection scheme with regard to agent outcome and diversity.

The section begins with an introduction into benchmark values that provides the AI designer with hard numbers about the behaviour tree. It then continues to a section that suggests highlighting paths in a behavior tree to help the AI designer to keep focus. The next section shows how probability values in the behaviour tree can be locked to distinguish final probability values, which then is used in the following section that extends the principle of autogeneration probablity values from Section 5.2 on page 39 to include this. The next section presents a solution to the problem where the AI designer assigns probability values to the behaviour tree that breaks the path probability distribution. The following section introduces the concept of packages in behaviour trees to abstract generic behaviours into reusable nodes. The last section presents a workflow that utilises all of the tool extensions in the process of creating a behaviour tree.

6.1 Benchmarks

Benchmark values in this context means values that provide hard information about a behaviour tree, e.g. statistical information like the number of nodes in a behaviour tree. The benchmark values can be combined with the AI designers knowledge about the game context to provide an estimate about how the behaviour tree is constructed and in what areas, e.g. the diversity of the behaviour tree, that should be looked into for improvement.

Statistics

Statistical information about the behaviour tree such as the number of nodes and paths provides the AI designer with structural information. This information can be utilised to provide an estimate of the current level of complexity in the behaviour tree used when building the behaviour tree.

Diversity

Presenting the diversity value of the behaviour tree would allow the AI designer to first and foremost get an idea about the current level of diversity in the behaviour tree, but also allow him to see how a change in a probability value would change the diversity. For instance the AI designer could change one probability value in the behaviour tree and observe that this decreased the diversity of the agent, however he also knows that this change would shift the intended behaviour in the wanted direction. The AI designer could then allow this decrease or try another solution to see if that keeps the level of diversity while improving the agent behaviour. There are cases where the diversity of the agent not equals the maximum possible diversity, but where the AI designer concludes that the achieved behaviour selection scheme is diverse enough for the purpose and stops.

Agent Outcome

The outcome of the behaviour tree or any selected single node would be useful to the AI designer as it provides an idea about the current outcome of the agent, e.g. if the utility value type is aggressiveness and the expected utility is large the AI designer would know that this agent is very aggressive. This value of course depends upon the AI designer having chosen a utility value type and defined utility and probability values for the whole tree. This benchmark can also be used by the AI designer to deduct a range for the expected outcome condition needed to get the path probability distribution. This can also be used if the AI designer wants agents with the same basic behaviour tree, but with different levels of the utility value type, e.g. if the utility value type was difficulty the AI designer could use this benchmark to create agents with different difficulty levels.

Path Probability Distribution

The path probability distribution is the optimal goal for the AI designer to achieve based upon his requirements of the agent outcome. The path probability distribution should be written at the end of each path such that it represents a goal to achieve. The AI designer can then try to approximate the path probability distribution when assigning the probability values to the selectors.

The path probability distribution can also provide the AI designer with some initial ideas about how to assign the probability values of the selectors before any of them has been assigned. An example is a path which according to the probability distribution should be 70%, where the AI designer then would know that the whole subtree containing the path at least would require 70% probability of execution to adhere to the path probability distribution.

The path probability distribution depends upon the correct assignment of utility values to achieve it, but by combining the knowledge of the game context and the probability distribution it would be possible for the AI designer to deduct if the distribution fits with his idea. For instance the AI designer could notice that an important path has a low probability value, which thereby makes it possible for him to conclude that some of the utility values in that branch might be poorly assigned. After having updated the utility values the AI designer could then ask the tool to recalculate the path probability distribution and continue the cycle until he is satisfied with the result.

6.2 Path Markup

It is difficult to find a direct correlation between the path probability value and the game context so it is not sufficient to show the path probability distribution to the AI designer. However it is possible to use the path probability distribution as a guideline for the AI designer when a probability value on a path is to be changed. As Figure 6.1 on the following page shows, the behaviour tree can illustrate the paths going away from the probability node, and thereby highlight which of the paths this probability value affects, which in effect helps the AI designer to approximate the path probability distribution.

The AI designer will in some cases be able to deduct the probability values of a path and would therefore be beneficial to highlight the path to help him to keep focus on the task at hand. Figure 6.2 on page 49

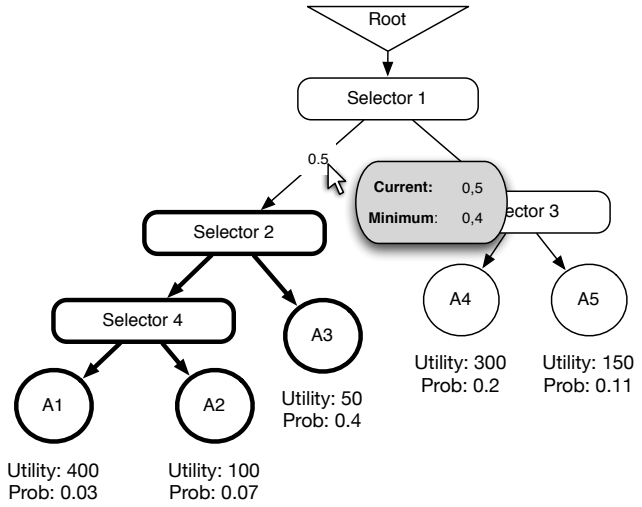


Figure 6.1: An example of a behaviour tree with three possible paths

illustrates this where the AI designer in (a) has selected one path, and in (b) has selected another. After having selected this path the AI designer can set any probability values to the path and try to approximate the path probability.

6.3 Probability Value Lock

When assigning probability values to the behaviour tree the AI designer can in some cases deduct one or more probability values based upon his knowledge of the game context. In this case the AI designer needs a way to indicate that this setting is final and should not be changed e.g. when he is tweaking the behaviour tree. Such settings can be locked to indicate that the value is final, and the AI designer can write a note in the tool that explains why this value is locked. The use is illustrated in Figure 6.3 on the facing page, where the first probability value is locked, and the rest are not, so in this case the AI designer has deducted that the leftmost branch must be executed 30% of the times but are uncertain about the rest. If the AI designer at some point changes his mind the probability value can be unlocked to allow changes to the variable.

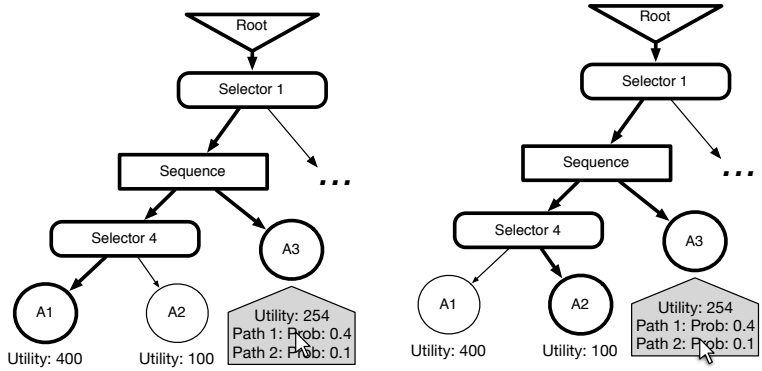


Figure 6.2: An illustration of how values can be automatically calculated in the behaviour tree.

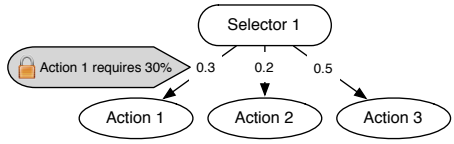


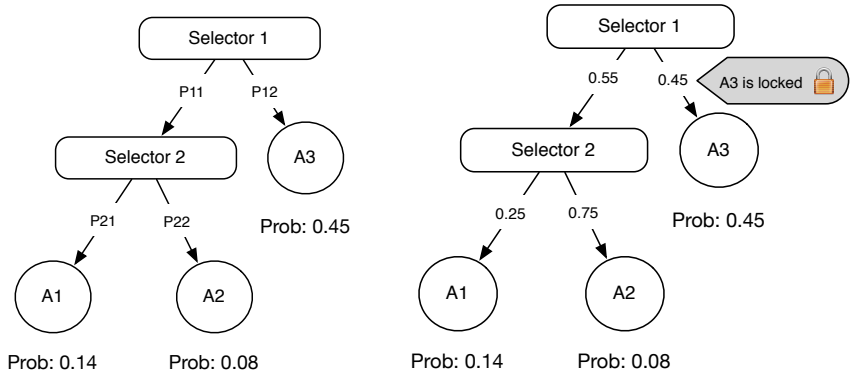
Figure 6.3: Probability values in the behaviour can be locked and a note can be written that describes why

6.4 Autogenerate Trivial Probability Values

When the AI designer locks a probability value in the behaviour tree he indicates that this setting is final and should not be changed, however the same cannot be said for the rest of the unlocked variables. The tool can test if any probability values can be calculated based upon the locked variables. This test is performed every time a new variable is locked as described in Section 5.2 on page 39. This auto generation of probability values removes some of the complexity of the behaviour tree for the AI designer and makes it possible for him to concentrate on assigning only a subset of the variables.

The equations illustrated in Section 5.2 on page 39 can be set up for a behaviour tree and the path probability distribution and the locked variables can be inserted into the equations. The tool can then test if it is possible to find the values of any of the unlocked variables based upon this. Figure 6.4 illustrates this where (a) is the empty tree with the unknown variables and (b) is after the AI designer has defined and locked a probability value. As is illustrated in the equations in 6.1 this complies with the requirement of the right branch, and makes it possible for the tool to calculate suggested values for the rest of the variables in this example.

$$\begin{aligned} 0.14 &= P11 \cdot P21 \\ 0.08 &= P11 \cdot P22 \\ 0.45 &= P12 \end{aligned} \tag{6.1}$$



(a) The behaviour tree before any probability value is defined

(b) The behaviour tree after a probability value has been defined and locked

Figure 6.4: This behaviour tree only requires one lock to make it possible to calculate suggested values for the rest

6.5 Probability Value Constraint

The AI designer is able to insert any probability value in a selector as long as the sum of the values is 1.0 and that each value p complies with $0.0 \leq p \leq 1.0$. This makes it possible for the AI designer to assign probability values that renders the path probability distribution impossible. This is possible since the AI designer might know that some variable should be set to a certain value in order to acheive the wanted behaviour, however that value makes it impossible to comply with the path probability distribution. The problem is illustrated in Figure 6.5 where the AI designer has assigned the probability distribution {0.1, 0.9} to Selector 1 which makes it impossible to achieve the path probability distribution.

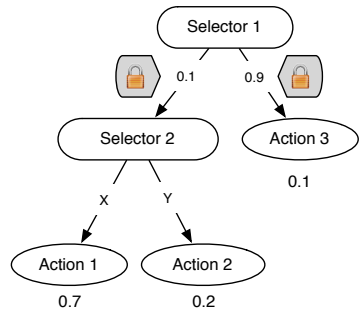


Figure 6.5: The values assigned by the AI designer might not comply with the path probability distribution

The solution to the problem is that the tool after the AI designer has locked a variable tests if the newly defined variable breaks the path probability distribution. This can be done by setting up the equation for the path and check if it is possible to solve. If this check fails and the defined variable renders the path probability distribution impossible to achieve the AI designer will be presented with a choice. Either he changes the value to one that fits with the path probability distribution or he sets the setting as a constraint. If the AI designer chooses not to change the value the tool must calculate a maximum possible probability value of each path which the probability value affects. The pseudo code to calculate the value for one path is shown in Listing 6.1.

Listing 6.1: This function calculates the maximum possible probability value of a path

```
1 var maximum_path_prob = 1
2 foreach probability value p in the path
3     if p is locked
4         maximum_path_prob *= p
5 return maximum_path_prob;
```

These values can then be added as constraints to the diversity-outcome linear program, so when it is run again the linear program will either output a new path probability distribution that complies with the current settings or tells the AI designer that no solution is available. In the example of Figure 6.5 on the preceding page the linear program would look like below.

$$
\begin{aligned}
\text{Maximise} \quad & -(p_1 \cdot log_2(p_1) + p_2 \cdot log_2(p_2) + p_3 \cdot log_2(p_3)) \\
\text{subject to} \quad & p_1 \cdot U_1 + p_2 \cdot U_2 + p_3 \cdot U_3 \geq V_1 \\
& p_1 \cdot U_1 + p_2 \cdot U_2 + p_3 \cdot U_3 \leq V_2 \\
& p_1 + p_2 + p_3 = 1 \\
& p_1 \leq 0.1 \\
& p_2 \leq 0.1 \\
& p_3 \leq 0.9 \\
& p_1, p_2, p_3 \geq 0
\end{aligned}
$$

The path probability distribution can be rendered impossible to achieve for instance if the AI designer has chosen difficulty as utility type and the action node which reflects the goal of the game, for instance capture the flag, has a high difficulty value. The AI designer sets the agent to have a low difficulty level, i.e. set a low range for the utility constraint, and the tool will then calculate the path probability distribution. However the result of this might be a probability distribution which rarely executes the goal of the game, and the AI designer knows the probability of executing this particular action and therefore sets the probability value to a high amount. This breaks the path probability distribution, but the AI designer is aware of this and locks the value, and thereby forces the tool to calculate a new probability distribution that fits with the setting.

The AI designer is able to use as many of the defined variable as constraints as needed, but at some point it might not be possible to solve the linear program given the constraints. If that is the case the tool provides the AI designer with two options again. First to loose some of the defined variables to satisfy the utility range constraint or secondly change the utility range as it is not possible to solve given the other constraints.

6.6 Behaviour Packages

Often the behaviour tree consists of a series of generic behaviours which not necessarily has any probability correlation to the rest of the behaviour tree. This means that no probability value in this generic behaviour should depend upon other probability values in the rest of the behaviour tree. An example could be a behaviour that follows some leader agent. This behaviour would be identical in all agents sharing this behaviour and should therefore have the same probability values among the agents. This subtree can then be put into a package and the AI designer can see this as a single node with only one utility value and no path branching. So even though this node after compilation might branch into several paths it will only be seen as one in the main tree. The main tree then becomes smaller and thereby makes it easier to assign the rest of the probability values with the rest of the tools available. This is also illustrated on Figure 6.6 where (a) shows a behaviour where the sub-tree beginning in node Selector 2 is a generic behaviour, which then in (b) is replaced by a package node making the behaviour tree smaller and easier to assign probability values to.

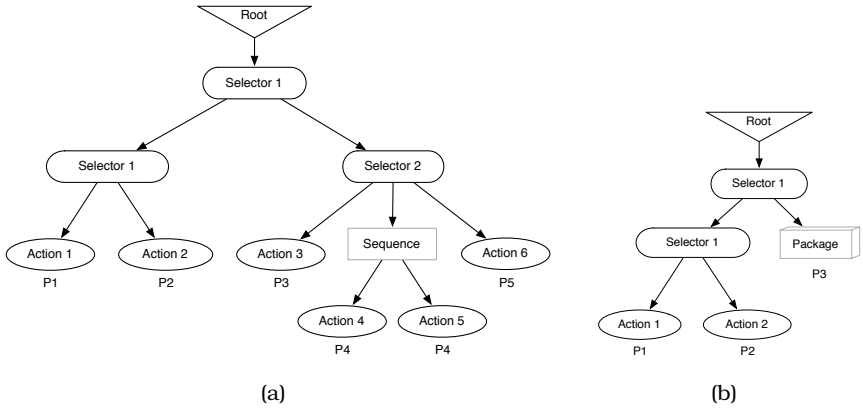


Figure 6.6: Shows how a behaviour tree after having put generic behaviours into packages can make it smaller and thereby remove some of the complexity of probability values assignments

6.7 Workflow

This section describes the workflow of creating an agent for a game with the help of the new tool extensions. The workflow is based upon mockup images of how the tool extensions would look in the Behaviour Editor as presented in Section 2.2 on page 19, however any of the tools can be implemented in any type of behaviour tree editing tool.

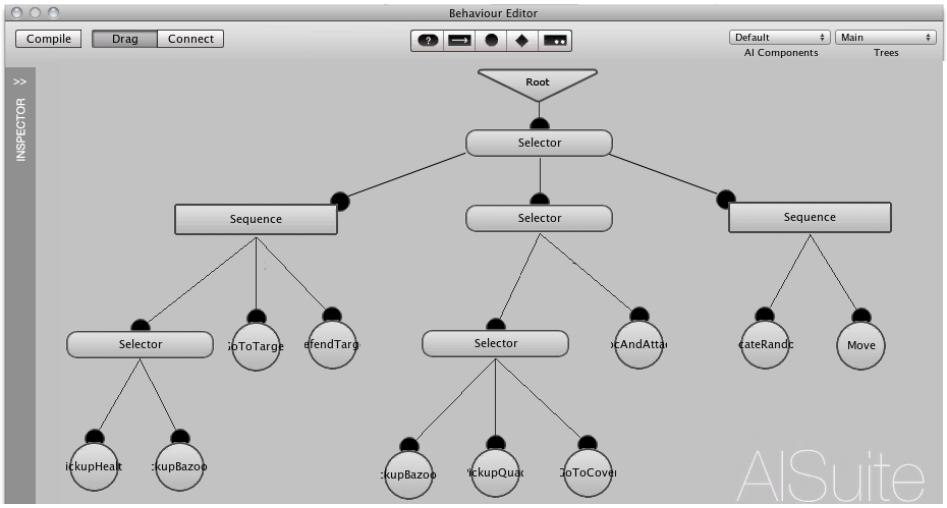


Figure 6.7: A behaviour tree as it looks in the Behaviour Editor

When the AI designer opens the Behaviour Editor it will only show an empty canvas, and the AI designer can create a new AI component. When the AI designer has created this he is able to draw the behaviour trees required by the agent using the same notation as presented in the formal model. Figure 6.7 shows a behaviour tree drawn in the Behaviour Editor. The name on the action nodes are automatically set to be the name of the action script, which performs the action on the agent, assigned to the node. In this behaviour tree the AI designer knows that the third subtree of the topmost selector performs the generic behaviour for walking randomly around the level. The AI designer needs the behaviour in another agent so he chooses, as shown in Figure 6.8 on the facing page to turn this subtree into a package to simplify the behaviour tree. The AI de-

signer is at this point able to compile the behaviour tree and attach it to an agent in a game and run it. However the selectors are using the random selection scheme, and it is therefore unlikely that the outcome of the agent represents the desirable agent outcome and the AI designer must therefore switch some of the selectors to other selection methods such as probability to tweak the behaviour. When a selector scheme is switched to probability the children of the selector will automatically received an equal share of probability.

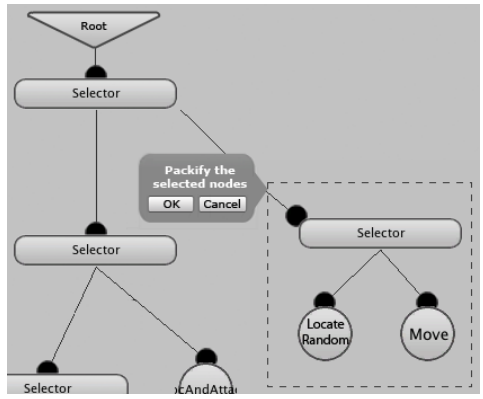


Figure 6.8: A subtree is recognised as containing a generic behaviour and is therefore converted into a package

The AI designer is often not able to define all probability values in the behaviour tree, but knows some of them from the game context. The AI designer is encouraged to define as many probability values as possible as this makes it easier to assign the last probability values later. In the case of Figure 6.9 on the next page the AI designer knows that the agent only should move random around the level 10% of the times and has therefore set this value and locked it. The figure also illustrates how path markup helps to see which nodes are affected when the probability values of the selector is changed. It is at this point possible to see the diversity level of the agent, which is available in the inspector on the left.

To assign the last probability values the AI designer can use the probability value guideline tool extension. First the AI designer specifies some utility type that represents the agent outcome. The AI designer should then assign utility values to each leaf node of the behaviour tree, as is illustrated in Figure 6.10 on page 57. The utility type can be seen in

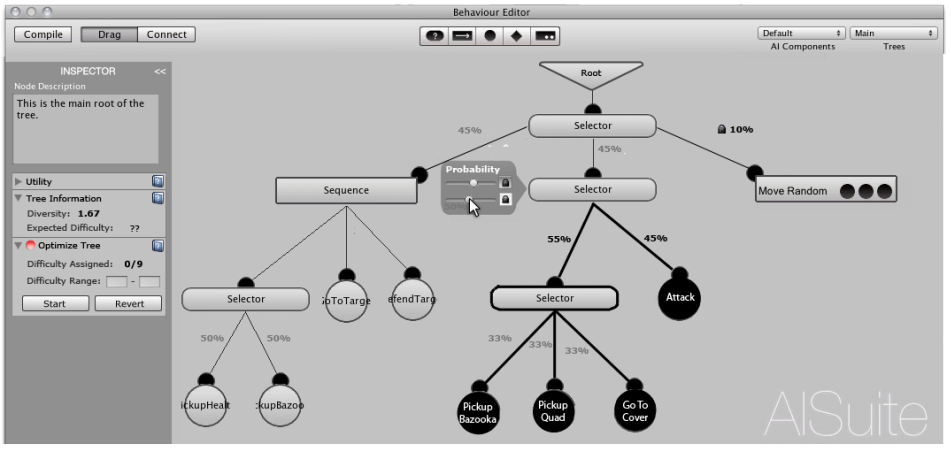


Figure 6.9: Probability values has been defined by the AI designer

the inspector window on the left, and the utility values can be seen in a box beneath a leaf node by selecting the blue icon. The inspector also shows how many leaf nodes that lack utility values before it is possible to continue. When the AI designer has specified all utility values the tool will calculate the path probability distribution based upon the locked probability values, the utility values, and the utility value range. The probability value of each path is then shown beneath each leaf node as seen in Figure 6.11 on page 58, and the AI designer can then define the missing probability values by using these as a guideline. If the AI designer does not find the path probability distribution to fit with his idea about the agent design he can change any of its constraints, e.g. utility values, and allow the tool to recalculate a new distribution. Each locked probability value makes the tool calculate other trivial probability values as described in Section 6.4 on page 49.

The AI designer might know the importance of a certain leaf node but because the behaviour tree is immense he is in doubt of which selector nodes that influences it. The AI designer can then select the node and one by one select the paths going to the node to see them highlighted as illustrated on Figure 6.12 on page 58. There are also situations where the AI designer does not know what node to define next, and can then use the node recommendation system to find the node that if defined would affect the final outcome of the behaviour tree the most.

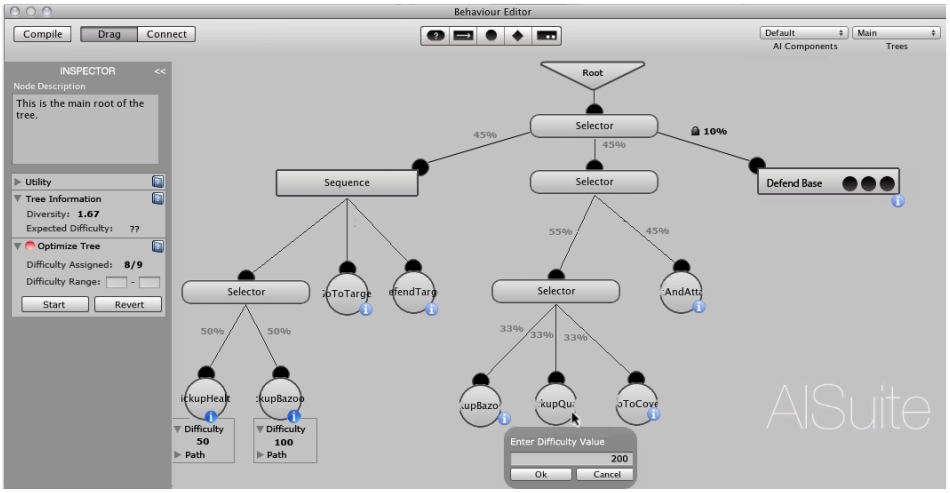


Figure 6.10: The inspector shows the benchmark values such as diversity and expected utility, and also makes it possible to change the utility type and create the path probability distribution. The main canvas shows the utility values as defined by the AI designer

The AI designer can define a probability value which will make it impossible to achieve the path probability distribution. In this case the AI designer is presented with a dialog box asking him if he wants to keep the value. In the case he does the tool must recalculate the probability distribution with the new constraints, otherwise the AI designer must assign a probability value that adheres to the probability distribution. As previously mentioned each locked probability value will make the tool recalculate trivial probability values, however before they are set the AI designer must approve them as illustrated on Figure 6.13.

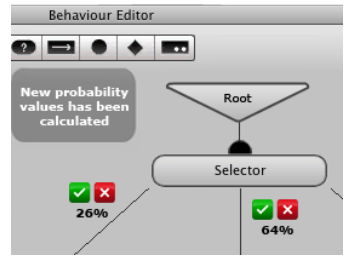


Figure 6.13: The tool has found some new trivial probability values and they are then presented for approval

When all of the probability values are defined the behaviour tree is complete, and the maximum level of diversity based upon the constraints set by the AI designer are achieved.

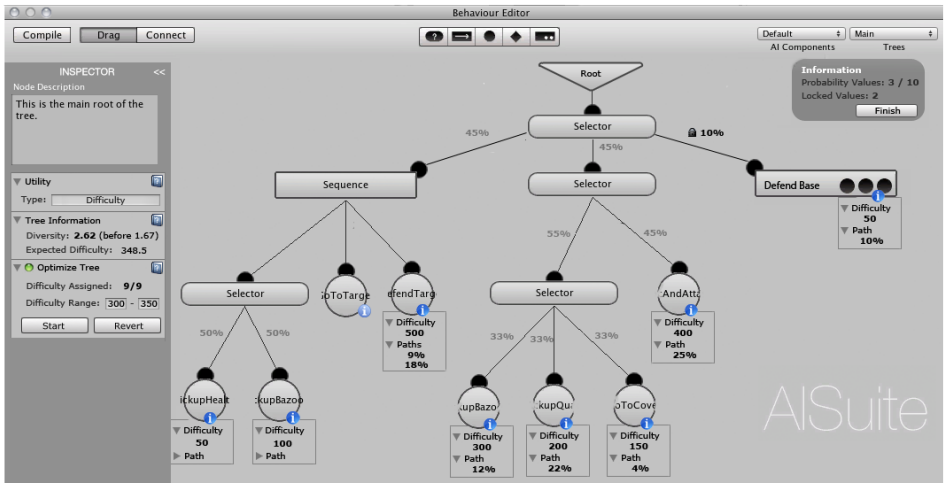


Figure 6.11: The probability distribution for each path has been calculated

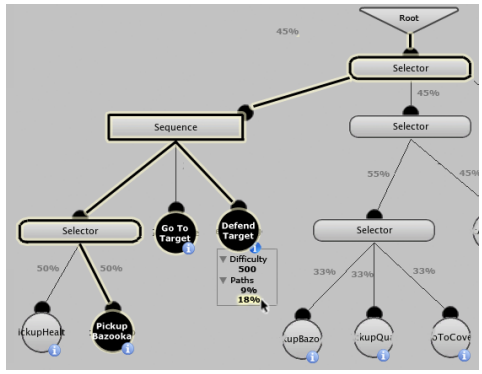


Figure 6.12: By selecting a path in the drop down menu it will highlight all of the nodes in the path

Summary

This chapter presented a series of tool extensions that utilised the concepts presented in the previous chapters to help the AI designer create a good behaviour selection scheme for an agent. The tool extensions covered hard numbers about the behaviour tree, how to help the AI designer to keep focus, locking and autogeneration of trivial probability values, and packaging generic behaviours. One section also covered how to handle probability values in the behaviour tree that does not comply with the path probability distribution. The chapter ended in a section that presented a suggested workflow for the new set of tool extensions.

CHAPTER

7

Evaluation

To review the contribution made by the formal model, the Behaviour Editor, and the tool extensions, it is necessary to evaluate these. This is done by presenting the project to a series of different test subjects in order to get their subjective opinion.

Two different types of test subjects are required, one who has experience creating agents but no direct experience using behaviour trees, and another with behaviour tree experience. This allows for two types of views on the project, as the one with behaviour tree experience can provide additional feedback about the project and the unexperienced provides some fresh insight into behaviour trees. The evaluation is the presentation of the different parts of the project and works as a discussion between the test subject and the tester, in such a way that the tester will ask into different areas to allow the test subject to provide his own opinion. Because of this there are no locked set of questions, but more an agenda containing the different areas which must be covered. The agenda can be found in Appendix B.1 on page 89. Besides having test subjects evaluate the techniques a demonstration game is developed in order to evaluate upon an agent created using the proposed techniques.

The evaluation is split up into four discussion areas where the first discusses the formal modal, second the Behaviour Editor, three the tool extensions, and fourth a demonstration game using different agents.

7.1 The Formal Model

The formal model that was presented in Section 2.1 on page 11 is largely responsible for how easy the process of building a good agent is and the comprehensibility of this agent. If the formal model is incomprehensible the advantage of using it would be severely limited, and it is therefore important to get an evaluation of it.

The formal model is to be presented to the test subjects in the form of pre-made behaviour trees as seen in Appendix B.3 on page 98. The tester should then start a discussion regarding the readability of the formal model to uncover if the test subject understands the behaviour trees and thereby the formal model. After this the tester must ask the test subject if he sees it possible to create all types of agents, or if something is lacking in the formal model.

7.2 The Behaviour Editor

The Behaviour Editor, as presented in Section 2.2 on page 19, is an example of a behaviour tree editing tool. The Behaviour Editor has implemented the formal model, and thereby makes it possible to draw behaviour trees according to the model. The Behaviour Editor is also used as the basis for the presentation of the tool extensions for the test subjects and it is therefore necessary to have an evaluation of the tool itself. The tool is evaluated by taking the test subjects though the process of building a standard behaviour tree in the Behaviour Editor, so they can provide continous feedback on both the tool and the workflow.

7.3 Tool Extensions

After having run through a standard workflow in the Behaviour Editor the test subjects is presented with the tool extensions. Each tool extension is illustrated in a mockup that bases its graphical appearance on the look of the Behaviour Editor in order for the test subjects to have a known tool to relate to.

The first tool extension to be evaluated is the probability value guideline which is the main tool extension capable of helping the AI designer to assign probability values to the selector nodes. The evaluation begin with a discussion of how important diversity is to the test subject when creating an agent. The tester then start a discussion about utility types and utility values to see if the test subject would be able to relate to such terms and if how hard it would be for him to assign such values to a behaviour tree after this. The tester must engage in a discussion regarding the automatic assignment of probability values and how the node recommendation system would help.

Secondly the other tool extensions, such as benchmarks, path markup, and locking is presented using the mockups, and it must then be discussed how much they each would help in the process of creating the final behaviour tree.

7.4 Agent Comparison

The formal model, the tool, and the tool extensions can be combined into a common solution for creating agents for realtime games. The best way to get the solution evaluated is to create an agent using the new techniques, and then compare it against other agents created using different techniques. The agents must then be compared with regard to diversity and the required agent outcome. All of the agents must use the same behaviour tree, but with different probability distributions on the selectors to ensure that it is not the behaviour tree itself that improves the agent behaviour.

7.4.1 The Demonstration Game

The demonstration game is based upon a game mode known as King of the Hill, where the goal of the players is to stand on the "Hill" to receive points. The game is described in detail in AISuite: Behaviour Editor [4].

The Behaviour Editor uses the concept of interrupt trees to enable the agents to react to sudden events in the game world, e.g. if the agent is being shot at, and thereby overwrite the main behaviour until the situation has been handled.

As mentioned each of the agents has been implemented using different selection methods, where the first agent uses a selection algorithm based upon random. Random selection should create an agent with the highest amount of diversity but where the difficulty of the agent will vary. The second agent is created using probability selectors, but without using the new techniques. The probability values is therefore defined by an AI designer and it is difficult to measure the level of difficulty the agent provides to the game. The third agent is implemented using the new proposed techniques for defining probability values. Using this technique the AI designer is able to control the difficulty of the agent and at the same time maximise the diversity of the agent.

The agents uses interrupt trees, but only the probability values on the main tree differs between the agents. The main tree, as seen on Figure 7.1, controls the overall behaviour of an agent, and the interrupt trees handles the behaviours responsible for situations such when the agent is being shot at, sees an enemy, or is near the target. The utility type of the agents is set to difficulty, and the utility values is assigned to each action node in the behaviour tree.

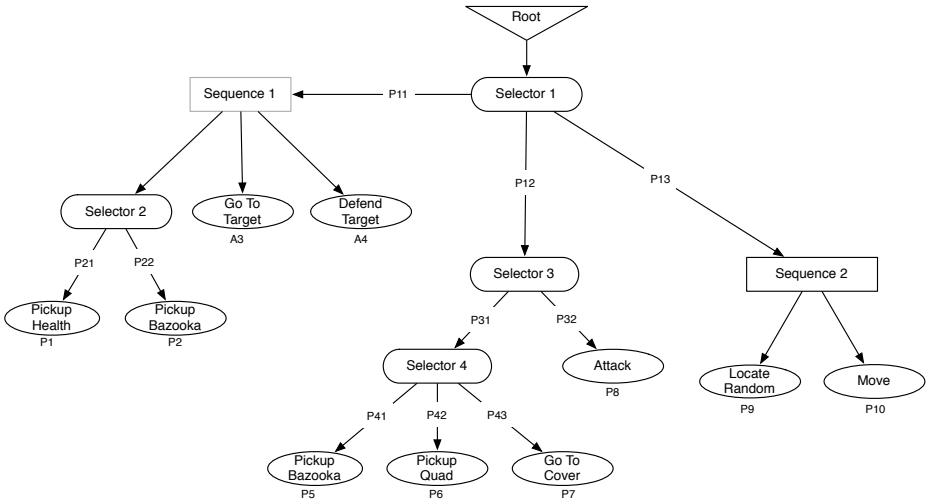
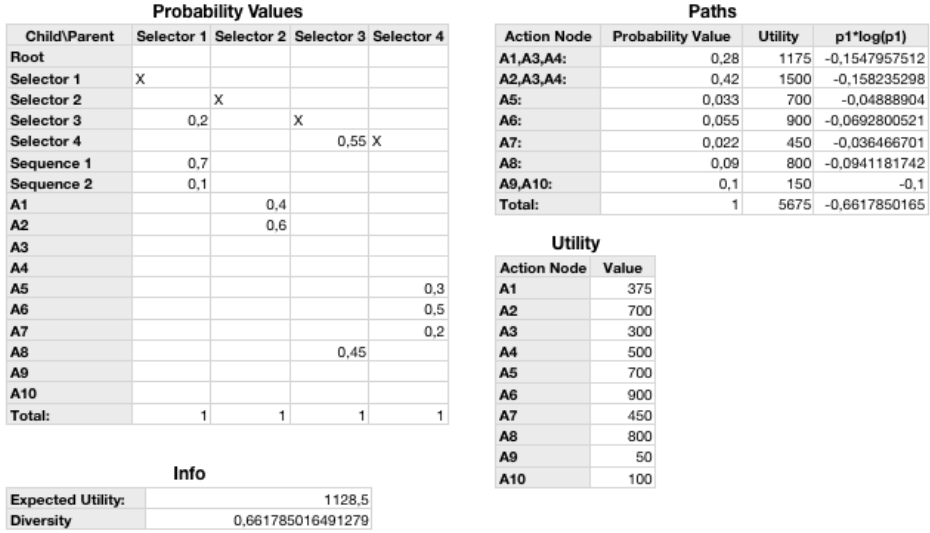


Figure 7.1: The main behaviour tree used by the agents in the demonstration game

As there exists no implementation of the diversity-outcome linear program with piecewise approximation the probability values has been found by approximating the optimal configuration of the behaviour tree in a spreadsheet capable of calculating the diversity and the expected utility based upon the probability values of the nodes. The spreadsheet is shown in Figure 7.2 on the facing page.

Figure 7.3 on page 66 illustrates the main behaviour tree of the agent that has been implemented using the new proposed techniques. The utility condition was set to high and the agent should therefore provide a high amount of difficulty in the game. The settings of the two other agents can be seen in Appendix B.5 on page 105.

Each of the agents is then placed in the demonstration game and it



Probability Values

Child\Parent	Selector 1	Selector 2	Selector 3	Selector 4
Root				
Selector 1	X			
Selector 2		X		
Selector 3	0,2		X	
Selector 4			0,55	X
Sequence 1	0,7			
Sequence 2	0,1			
A1		0,4		
A2		0,6		
A3				
A4				
A5				0,3
A6				0,5
A7				0,2
A8			0,45	
A9				
A10				
Total:	1	1	1	1

Paths

Action Node	Probability Value	Utility	p1*log(p1)
A1,A3,A4:	0,28	1175	-0,1547957512
A2,A3,A4:	0,42	1500	-0,158235298
A5:	0,033	700	-0,04888904
A6:	0,055	900	-0,0692800521
A7:	0,022	450	-0,036466701
A8:	0,09	800	-0,0941181742
A9,A10:	0,1	150	-0,1
Total:	1	5675	-0,6617850165

Utility

Action Node	Value
A1	375
A2	700
A3	300
A4	500
A5	700
A6	900
A7	450
A8	800
A9	50
A10	100

Info

Expected Utility:	1128,5
Diversity	0,661785016491279

Figure 7.2: The spreadsheet capable of calculating diversity and expected utility based upon the probability values and the utility settings

Table 7.1: Theoretical diversity and utility levels for each agent

	Sir Manual	**Sir Random**	**Sir Tool**
Difficulty:	309,975	736,74	1128,5
Diversity:	0,348	0,754	0,661

is possible to see the scores and their diversity for each agent in the game. The agents has a theoretical diversity and utility level as seen in Table 7.1. The score can be used to benchmark each of the agents performance and the agent implemented using the new techniques must have a higher amount of difficulty according to the utility values and should therefore often win. The diversity for each agent is calculated while the game is executed by sampling the behaviours executed in the game. The calculated diversity can later be compared with the theoretical diversity in order to ensure the agent implemented using the new techniques has a high amount of diversity. Figure 7.4 on page 67 illustrates the running game with three different agents.

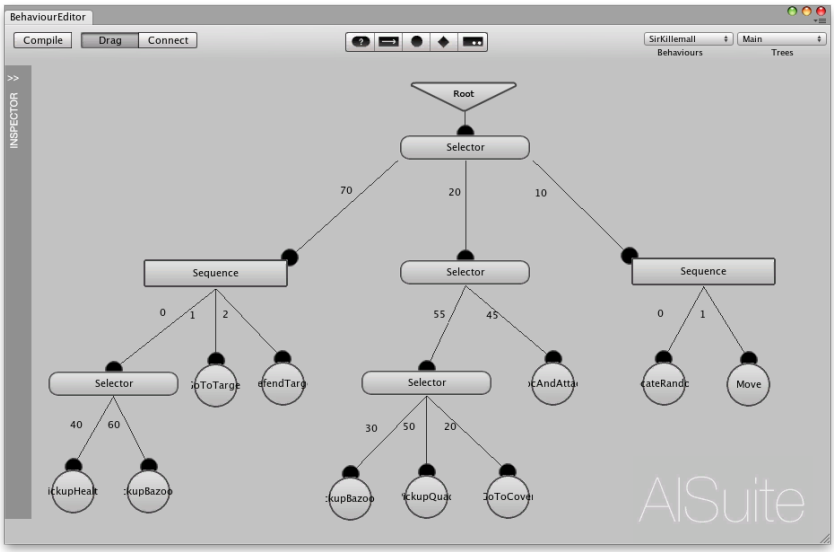


Figure 7.3: The main tree used by a agent in the game

7.4.2 Evaluation method

As said the agent created with the use of the presented solutions should be evaluated by comparing it to other agents. This comparison can be accomplished by running the game multiple times and let the tool sample the scores and diversity for each agent. The sampled data can afterwards be analysed and compared with the theoretical data in order to show the new techniques ensures an agent with a high amount of diversity and an agent outcome with in the range defined by the AI designer. The sampled data should also show that the agent implemented using the random selection technique has the highest amount of diversity but that the difficulty of the agent would vary.

It is not sufficient to calculate the agent diversity and scores in the game. Therefore it is required to analyse upon the agents behaviours by letting the test subjects provide subjective feedback by showing them the running demonstration. The demonstration game therefore features the three different agents competing in a game of King of the Hill. The tester then interviews the test subjects during the execution of the game regarding the behaviours of the agents.

Figure 7.4: The game running with three different agents

7.5 Results

Four test subjects were interviewed and the interview notes can be found in Appendix B.2 on page 91. Two of the test subjects had previous experience with behaviour trees, and the other two had experience in the field of AI but no direct experience with behaviour trees. The following sections is a summary of the four interviews, which is divided into each of the discussion areas.

Formal Model

Since two of the test subjects has no previous experience with behaviour trees it is important if they can understand the model and see how to use it. They both found it to be easy to read and understand, and as long as the scripting was let to the programmers it should be easy for them to use the model. The test subjects with behaviour tree experience also noted that the formal model clearly illustrates how behaviour trees is constructed. The test subjects was also inquired regarding if they could find any lacks in the formal model that would need to be fixed before the formal model could be used to create an agent, however none of the test subjects could notice any such limitation. It should be noted that the non-experienced with behaviour trees only had limited time to understand the formal model which did not provide enough time for an in depth analysis of the structure, however the experienced had previously created agents with behaviour trees that had the same set of nodes without any obvious limitations. The test subjects all understood the principle of path and how to calculate the probability of this, which was important for the later parts of the evaluation.

One of the test subjects had concerns regarding what would happen if the behaviour tree got immense, however the authors feel that if such a behaviour tree is required to achieve the wanted behaviour it is a sign that the behaviour tree should either be redesigned or perhaps behaviour trees is not the right structure for that type of agent. A test subject also requested some way of reusing subtrees of the behaviour tree, and it was noted that even though this is introduced later as a tool extension this should be added as a part of the formal model as well.

Behaviour Editor

The test subjects played around with the Behaviour Editor tool and created small behaviour trees in order to get first hand experience both with the tool and also the interactive version of the formal model used in the Behaviour Editor. There was a general consensus between the test subjects that there was a clear correlation between the formal model and the model used in the tool, however they did note that the graphical representation of the root node and the arrows could be better implemented to better fit with the formal model. The root node and the connection arrow as used in the Behaviour Editor is illustrated on Figure 7.5 on the facing page.

The tool was written for Unity so it is important that it does fit into the existing infrastructure and the test subjects all agreed that the overall look and feel of the Behaviour Editor fitted well into Unity. The test subjects also thought the inspector, and the fact that most nodes in the behaviour tree can be scripted to the wishes of the programmer and designer was clear advantages of the tool. There was some discussion if the inspector should be in a window for itself to fit a bit more into the Unity experience. The test subjects also found the add node to canvas and the node connection utility to be unintuitive.

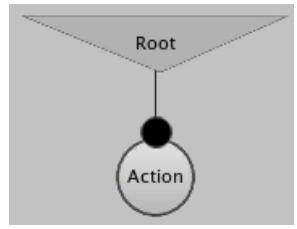


Figure 7.5: The root node does not look like the rest of the nodes, and the connection indicator in the tool ends in a circle instead of arrow

Some of the test subject was concerned regarding the use of interrupt trees as they felt it could be difficult to understand and use. The example used by one of the test subjects was whether or not the main tree should be reset after an interrupt tree had run, which is a consideration needed to be made. One test subject also wished to have built in support for debug information shown when executing the tree. The last part of the Behaviour Editor was the workflow required to create an agent for a game which all of the test subject found to be the proper way to construct an agent and in overall find the tool to be useful.

Tool Extensions

The test subjects was inquired regarding how important diversity was for agents, and the general consensus was that as long as it did not mean that the agent would perform an unintelligent action and thereby disrupt the goal of the agent it was desired. The test subjects was then told about the concept of agent outcome and again it was agreed that it would be beneficial to define a level of outcome and then try achieve the highest possible amount of diversity while keeping the outcome.

After having described utility as used in the project to the test subjects they were inquired if they would find it difficult to assign such values. The test subjects did here provide some thoughts on how they would do it, e.g. one would assign a common value to all leaf nodes where there was doubt of the value and thereby create a basis for the rest of the values. But in general the test subjects did not find it to be a difficult task. The test subjects found the idea of custom utility types to be intriguing and one

found that e.g. agents in a real time strategy game also could be modelled by using a utility type called level of exploration.

They all agreed that to assign probability values to the behaviour tree that keeps the balance of outcome and diversity could provide a challenge, and that tools helping with this would be well received. The test subjects was presented to the whole suite of tool extensions, and generally liked how they could add their knowledge from the game by e.g. locking values in the behaviour tree, while still getting help to achieve an optimal distribution. It was important for the test subjects that they should keep control, e.g. automatically completing the behaviour tree for them would not be beneficial as they then would have lost both control and overview. However they did like how it could calculate trivial probability values as they had to be approved before use. They also liked that the tool could help them by recommending a node to define next.

The test subjects had some general improvements for the look and feel of the new extensions however the consensus was that they would be helpful in developing an agent.

Agent Comparison

Most of the test subjects could easily see how each agent behaved differently in the game, and they found it easy to spot the agent which has been implemented using the new techniques. However one of the test subjects pointed out that in order to get a complete grip on the difference in behaviour the game must run for a longer period of time.

The demonstration game was also run ten times where the scores and the diversity of the different agents was calculated. Table 7.2 on the next page illustrates the calculated data from ten game executions.

The game winner was Sir Tool which also was the agent created using the new techniques. The diversity values for each agent is close to the theoretical calculated value. The agent with the largest amount of diversity is Sir Random closely followed by Sir Tool. The agent with the lowest amount of diversity is Sir Manual. Sir Manual has a quite small level of diversity compared to the rest because his probability values are poorly defined. Based upon these observations it is apparent that the use of the new techniques allows one to create an agent that has a certain agent outcome while maximising the diversity. Sir Random does provides the

Table 7.2: The calculated data for each agent from ten game executions.

	Sir Manual		**Sir Random**		**Sir Tool**	
	Score	Diversity	Score	Diversity	Score	Diversity
1	0	0,293	468	0,731	2000	0,652
2	51	0,258	36	0,692	2000	0,584
3	396	0,307	1573	0,703	2000	0,639
4	720	0,388	856	0,706	2000	0,641
5	0	0,169	2000	0,678	1792	0,655
6	571	0,243	1102	0.685	2000	0,701
7	922	0,237	398	0.726	2000	0,608
8	-187	0,192	85	0.711	2000	0,572
9	-100	0,415	2000	0,748	1861	0,589
10	287	0,264	1295	0.736	2000	0,622
Average	266	0,277	981,3	0,712	1965,3	0,626
Wins	0		2		8	

highest level of diversity as expected, but does not keep the agent outcome as well as Sir Tool. Also just assigning probability values in the behaviour tree as done with Sir Manual requires large amounts of testing before a good result can be achieved.

Summary

The evaluation covered all the main areas of the project to ensure that the result would be beneficial for potential users. The formal modal was seen as a readable and easy structure to create agents for games. The Behaviour Editor had its flaws but generally it provided the users with the functionality they would require of such a tool. The test subjects also liked the idea of having a tool capable of helping them to assign probability values to the behaviour tree, and they felt that the set of proposed tool extensions would cover this. The demonstration game illustrated how an agent quickly could be set to be difficult and have its diversity maximised when using the new techniques, which was further proved by letting it fight against a baseline agent created manually, and a random agent.

CHAPTER

8

Epilogue

The following sections describes the reflection, the conclusion and the future development of the proposed techniques and the behaviour editing tool Behaviour Editor. The first section reflects upon why a formal model has to be created and also the final experiences with using the new proposed tool extensions for Behaviour Editor. This is then followed by a project conclusion, and lastly the future development section that e.g. presents suggestions for how the tool can be improved and researched in greater detail.

8.1 Reflection

The formal model of behaviour trees had to be created before any research into easing the development of behaviour trees could be created as no formal model previously existed. The formal model makes it possible to have the same graphical appearance as well as the same behaviours for the nodes and the same definitions for paths in all tools following the model. This also makes it easier for users to make the transition to other tools but also to implement the features presented in the report if they follow the model. The formal model also made it possible to define diversity and agent outcome measures for behaviour trees and thereby effectively making it possible for the authors to create a tool based upon linear programming to help the users define probability values for the behaviour tree. All in all the formal model was a great aid through the project to keep the ideas on track and constantly having a model to check the theories towards.

In the beginning it was chosen to utilise standard linear programming for calculating the path probability distribution as this seemed to be able to produce a proper solution. Linear programming was then researched in great detail in order to fully understand the mathematical technique, but late in the project the authors realised that the objective function based upon entropy was logarithmic and since linear programming re-

quires a linear objective function another solution had to be found. An article was found which presented a problem nearly identical to this, that proposed a solution called piecewise approximation of the logarithmic objective function. After having read the article it was concluded to be applicable for this case as well, however in order to evaluate the performance and the results of utilising this technique it has to be analysed and implemented.

The evaluation of the Behaviour Editor and the new tool extensions proved that the proposed techniques is a valuable aid for AI designers developing agents for games. The two introduced measures diversity and agent outcome was found to be understandable and usable when distributing probability values to the behaviour tree, but also a great factor when comparing agents among each other.

A demonstration game were made where the agent created with the use of the presented techniques was evaluated by comparing it again two other agents. The results was positive as the measured diversity values for each agent was close to the theoretical value. This shows that using the new techniques makes the AI designer able to create an agent who's diversity level is lower but close to the agent implemented using the random selection technique. However the amount of games won shows that the agents difficulty level is the highest which fits with the fact that it was designed to have this in the behaviour tree as well. The test also showed that the agent implemented by manually assigned probability values without the new techniques has the lowest diversity and the lowest overall score. This shows that by designing an agent without using any tools to aid the process it is hard to ensure a high level of diversity and agent outcome.

Besides the verification of the tool to be beneficial for an AI designer, the evaluation also provided new possible improvements which would increase the ease of use of the tool. It is however necessary to implement the tool extensions for the Behaviour Editor in order to conduct tests which could provide more detailed results. By having this implementation it would be possible to have an AI designer actually use the tool to create an agent. The test should prove that it is difficult to assign proper probability values which maximises the diversity and retains a defined outcome of the agent without the help of a tool.

8.2 Conclusion

The goal of the project was to ease the process of defining probability values in a behaviour tree. To fulfil this the authors produced a formal model for behaviour trees, that can be used as a model for others to create behaviour tree tools. The formal model was also used for the development of the new benchmarking measures for behaviour trees.

Two benchmark measures was proposed and was found to be valuable measures of an agents performance and behaviour. These measures could then be used together with a mathematical technique known as linear programming. The linear program created from this is able to calculate a probability distribution for each path in a behaviour tree which can be used both as a guideline when defining probability values but also for automatic generation of probability values. These extensions then serve as the basis for a set of proposed tool extensions for use in a behaviour tree editing tool such as the Behaviour Editor.

An evaluation of the formal model and the tool extensions were made and proved that the formal model was understandable and that the new extensions to the tool is a valuable aid in the process of defining probability values. Lastly an evaluation of an agent that was developed using the new proposed techniques showed that the agent behaved as desired, and that using the techniques made it easier to find out what to change in order to switch the agent behaviour in a certain direction.

8.3 Further Development

As previously mentioned linear programming can not directly be used to solve the problem for this project. A piecewise approximation technique is proposed but has not been researched in greater detail nor implemented so in order to evaluate the performance and the results of using this technique, future work would include researching how the technique works and how it can be implemented in the Behaviour Editor.

During the evaluation of the tool extensions the test subjects observed improvements for the tool. An important suggestion for improvement are regarding the graphical representation of a behaviour tree in the tool where the test subjects pointed out that the graphical representation did not fit completely. Other issues included the tool that draws connections between nodes in the canvas and the add node functionality which the

test subjects did not find easy to use. The last issue with the tool is the implementation of interrupt trees, where test subjects was uncertain whether or not an AI designer would understand the system and thereby be able to use it. These suggestions has be taken into consideration of the next version of the Behaviour Editor to have a more functional and powerful behaviour tree tool.

Lastly it would be required to perform more tests such as a functional test of the tool, but also a practical test of the workflow by letting a designer create an agent for a specific game.

BIBLIOGRAPHY

[1] Unity3d. http://www.unity3d.com. 9

[2] John W. Chinneck. Practical Optimization: a Gentle Introduction. 2000. 42

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2 edition, 2005. 79, 87

[4] Anders Tankred Holm and Mads Bøgeskov. Aisuite: Behaviour editor. https://services.cs.aau.dk/cs/tools/library/details.php?id=1262691382, January 2010. 16, 19, 45, 63

[5] Gary A. Mastin and Richard J. Hanson. Maximum entropy signal restoration with linear programming. 05 1988. 38

[6] Jr. Paul Penfield. Principle of maximum entropy: Simple form, March 2010. [Online; accessed 23-March-2010]. 23

[7] Stuart Russel and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 2 edition, 2003. 31

[8] Wikipedia. Entropy (information theory) — wikipedia, the free encyclopedia, 2010. [Online; accessed 19-March-2010]. 23

APPENDIX

A

Linear Programming

Linear programming is a mathematical concept capable of solving problems where the goal is to maximise or minimise the variables of a linear function, which somehow are limited, e.g. in the case of resources or by competing constraints. The source for this section is Introduction to Algorithms [3], which is recommend for a more thorough exposition of the subject. Such problems is called linear programming problems, and is constructed as a linear program. A linear program consist of an objective function, which is the linear function who's variables needs to be maximised or minimised. The constraints of the linear program are called linear constraints and are represented as equalities or inequalities on the variables.

A feasible solution to a linear program is any setting of the variables in the objective function which satisfies all the constraints, while a infeasible solution is where at least one constraint is violated. A setting of the variables in the objective function is denoted by $\bar{x}$, and the result of the objective function with a certain setting applied is called the objective value. A feasible solution $\bar{x}$ whose objective value is larger than any other feasible solution is called the optimal solution, and the objective value is called the optimal objective value. A linear program is feasible if there exist a feasible solution to it, otherwise it is infeasible. If there are no finite optimal solution the linear program is unbounded.

The section starts with an introduction of the different canonical forms in which a linear program can be presented, and continues on to discuss the simplex algorithm which is capable of solving linear programming problems.

A.1 Canonical Forms

A linear program can be represented in two types of canonical forms called standard and slack. The standard canonical form is the max-

imisation of a linear function subject to a set of linear inequalities, and the slack form is the same except it requires linear equality constraints. Standard is a more convenient and readable way to express a linear program, however the simplex algorithm works internally on slack forms so in order to explain it it is presented. It is possible to convert a linear program into standard form and then from that into the slack form.

A.1.1 Standard Form

The first form to be presented is the standard form which is presented below:

$$\begin{aligned} \text{Maximise} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\ & x_j \geq 0 \end{aligned}$$

where c is the coefficients of the objective function, b is constants in the constraints, and a is the mn matrix containing all the coefficients in the constraints. x is the set of variables that must be maximised. From this it is possible to find four conditions that must be fulfilled before a linear program can be said to be in standard form:

1. The objective must be to maximise the objective value
2. All variables must have a non-negativity constraint
3. Equality constraints with equal signs must be converted to use less-than-or-equal sign
4. Inequality constraints with greater-than-or-equal sign must be converted to use less-than-or-equal sign

A.1.2 Slack Form

The slack form is similar to the standard form with the exception of only allowing one inequality constraint which is the non-negativity constraint

and the rest must be equality constraints. However an inequality constraint, as seen in Equation (A.1) can be converted to an equality constraint by introducting the notion of a slack variable that measures the difference between the two sides of expression. This slack variable can be seen in Equation (A.2) and Equation (A.3), and it is obvious that both must hold before the original expression is true.

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \tag{A.1}$$

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j \tag{A.2}$$

$$s \geq 0 \tag{A.3}$$

The result is that the standard form rewritten to conform to the slack conditions looks like this:

$$\begin{aligned} \text{Maximise} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & b_i - \sum_{j=1}^{n} a_{ij}x_j = x_{n+1} \\ & x_0 ... x_{n+1} \geq 0 \end{aligned}$$

where x_{n+1} is the slack variable. However slack form is a more concise structure than this, as it omits the "maximise" and "subject to" and represents the objective value by z:

$$\begin{aligned} z &= v + \sum_{j \in N} c_j x_j \\ x_i &= b_i - \sum_{j \in N} a_{ij}x_j \text{ for } i \in B \end{aligned}$$

The variables on the left side of the constraint are called basic variables, and the variables on the right side are called non-basic variables. This representation makes it possible to represent the linear program by a tuple: (N, B, A, b, c, v). Where N and B is the respective sets of indicies for

the non-basic and basic variables, A is the mn matrix containing the coefficients of the constraint variables, c is a vector containing the coefficients of the variables in the objective function, and b and v are the constants of the constraints and objective function respectively.

A.2 The Simplex Algorithm

The simplex algorithm is an algorithm capable of solving linear programming problems. Simplex takes a linear program as input represented in the concise form as shown in Section A.1.2 on page 80. This section explains how the algorithm works, and utilises the linear program shown in A.4 as an example to illustrate certain points of the algorithm.

$$\begin{array}{rcrcrcrcr} z & = & 2 & + & 3x_1 & + & x_2 & + & 2x_3 \\ x_4 & = & 23 & - & 2x_1 & - & x_2 & - & 3x_3 \\ x_5 & = & 53 & - & x_1 & - & 2x_2 & - & 2x_3 \\ x_6 & = & 10 & - & 2x_1 & - & x_2 & - & 2x_3 \end{array} \tag{A.4}$$

Simplex works by converting the provided slack form into an equivalent slack form, where the objective value of the basic solution is either equivalent or better. This is done until all the coefficients in the objective function are negative, which indicates that the optimal solution has been found. A basic solution is a setting where all the non-basic variables is set to zero so the basic variables can be calculated, and it is obvious that this is equivalent to setting the basic variables to the same value as the constant in their respective equations. In A.4 the basic solution would be $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_6) = (0, 0, 0, 23, 53, 10)$, and the objective value of this would be 2. The algorithm is split up into three parts called main, initialise simplex, and pivot.

A.2.1 The Main Function

The main function is the entry point of the algorithm, and the pseudo code for the algorithm can be found in Listing A.1. This function is responsible for continuously finding an equivalent slack form with a hopefully higher objective value given the basic solution. This is done by finding a non-basic and a basic variable and then switching their place. The entering variable is found by finding the constraint that has the the tighest constraint on the entering variable, and then selecting the basic variable of that constraint as the leaving variable. That is the constraint

where it is possible to assign the largest value to the entering variable without violating the constraints. E.g. if x_1 was chosen to be the entering variable in the example, then the x_4 constraint would allow x_1 to be 11, x_5 53, and x_6 5, before the non-negativity constraint is violated. So in this case x_6 has the tightest constraint so this would be chosen to become the leaving variable.

When all the variables in the objective function has negative coefficients the system can return the basic solution for this slack form, which when inserted into the original slack form provides the maximum objective value.

Listing A.1: Pseudo code for the main part of the simplex algorithm

```
1  float[] Simplex( float[,] pConstraintCoef, float[] pConstraintConst, float[]
       pObjCoef )
2  {
3      SlackForm slackForm = InitialiseSimplex( pConstraintCoef, pConstraintConst,
           pObjCoef );
4      if ( slackForm == null )
5          return null;
6
7      while ( terminate( slackForm.NonBasicIndicies, slackForm.ObjCoef ) )
8      {
9          // Choose a non-basic variable which has a non negative coefficient in the
               objective function
10         int enteringVariable = ChooseEnteringVariable();
11
12         float[] limitation = new float[slackForm.BasicIndicies.Length];
13         for (int i = 0; i < slackForm.BasicIndicies; i++) {
14             int BasicIndex = slackForm.BasicIndicies[ i ];
15             if ( slackForm.ConstraintCoef[ BasicIndex, enteringVariable ] > 0 )
16                 limitation[i] = slackForm.ConstraintConst[BasicIndex] / slackForm.
                       ConstraintCoef[BasicIndex, enteringVariable];
17             else
18                 limitation[i] = Math.Infinity;
19         }
20
21         // Choose a basic leaving variable which allows the most severe increase
               without violating the constraints
22         int leavingVariable = ChooseLeavingVariable( limitation );
23
24         if ( limitation[ leavingVariable ] == Math.Infinity )
25             return null; // The solution is unbounded
26         else
27             slackForm = SimplexPivot ( slackForm );
28     }
29
30     float[] solution = new float[ slackForm.BasicIndicies.Length + slackForm.
           NonBasicIndicies.Length ];
31     for (int i = 0; i < solution.Length; i++) {
32         if ( slackForm.BasicIndicies.Contains ( i ) ) // Is this index a basic
               indicies ?
33             solution[i] = slackForm.ConstraintConst[i];
34         else
35             solution[i] = 0;
```

```
36      }
37
38      return solution;
39  }
```

Listing A.2: The termination condition for the simplex algorithm. As long as there are positive coefficients in the objective function this function returns true

```
1  bool terminate( int[] pNonBasicIndicies, float[] pObjCoef )
2  {
3      foreach (int NonBasic in pNonBasicIndicies)
4          if ( pObjCoef[NonBasic] > 0 )
5              return true;
6
7      return false;
8  }
```

A.2.2 The IntialiseSimplex Function

The slack form that originally is provided to the simplex algorithm is not guaranteed to have a feasible basic solution, or to be feasible at all, so the algorithm needs to investigate this before trying to find an optimal solution to the slack form.

The algorithm starts by testing if the basic solution for the original slack form is feasible, and then returns the original slack form. If not it must be determined if there are a feasible basic solution, and InitialiseSimplex does this by creating an auxiliary linear program based upon the provided slack form. This auxiliary linear program is guaranteed to have a feasible basic solution, which then can be used to determine if the original linear program is feasible, and if so provide a slack form which has a feasible basic solution back to the main simplex function.

The auxiliary program is created by converting the original linear program into this form:

$$\begin{aligned} \text{Maximise} \quad & -x_0 \\ \text{subject to} \quad & \sum_{j=1}^{n} (a_{ij}x_j - x_0) \leq -b_i \text{ for } i = 1, 2, ..m \\ & x_j \geq 0 \text{ for } j = 0, 1, ...n \end{aligned}$$

This corresponds to setting the objective function to $-x_0$, and adding $-x_0$ to each constraint. Based upon this it is possible to deduct that the

original linear program is feasible if and only if the optimal objective value of the auxiliary program is 0.

Listing A.3: This function determines if a linear program is feasible or not.

```
1  SlackForm InitialiseSimplex( float[,] pConstraintCoef, float[] pConstraintConst,
       float[] pObjCoef )
2  {
3      int minConstraintConstantIndex = Find index of the minimum constraint constant;
4
5      if ( pConstraintConst[ minConstraintConstantIndex ] > 0 )
6          return new SlackForm( new int[]{1, 2, ... , nNonBasic }, new int[]{ n + 1,
               ..., nNonBasic + nBasic }, pConstraintCoef, pConstraintConst, pObjCoef,
                0 );
7
8      SlackForm Auxiliary = CreateAuxiliaryLinearProgram();
9
10     leaving = minConstraintConstantIndex + nNonBasic;
11
12     SlackForm FeasibleAuxiliary = SimplexPivot ( Auxiliary, leaving, 0 );
13
14     // Run while loop from Simplex and extract the basic solution from the slack
           form
15     FeasibleAuxiliary = FindOptimalSolution( FeasibleAuxiliary );
16     float[] solution = GetBasicSolution( FeasibleAuxiliary );
17
18     if ( x0 in solution == 0 )
19     {
20         SlackForm final = Remove x0 from FeasibleAuxiliary and restore original
               objective function
21         return final;
22     } else
23         return null; // The slack form is infeasible
24 }
```

The psuedo code for the algorithm can be seen in Listing A.3. The function starts by finding the minimum constraint constant, and tests if this is above zero. If it is the basic solution is feasible and the slack form can be returned. In line eight the function creates the auxiliary program as described above, and finds the index of the basic variable which has the minimum constraint constant in its equation by adding the amount of non basic variables to its index in line ten. Since the original slack form basic solution was not feasible neither will the auxiliary basic solution be, so line 12 does a pivot with x_0 as entering, and $x_{leaving}$ as leaving, which is guaranteed to provide a slack form with a feasible basic solution. Line 15 then runs the while loop from Listing A.1 in order to find the optimal solution, which is extracted in line 16. As said above the original linear program is feasible if and only if x_0 is zero in the basic solution, so this is checked in line 18. If so, the final slack form from line 15 will have x_0 removed and its original objective function restored, and then returned,

otherwise the linear program was infeasible.

A.2.3 The Pivot Function

The pivot operation is the basic operation in the simplex algorithm which handles switching a basic and non-basic variable in a slack form, and thereby create another equivalent slack form.

Listing A.4 shows the pseudo code behind this operation. The algorithm takes the slack form as input including the indices of the leaving and entering variables. The algorithm starts out by creating the set of variables necessary to represent the new slack form. This is, as seen on lines four to nine, based upon the original slack form.

The second step is to create the new equation where the entering variable is basis, i.e. on the left side of the equation. To continue the example of illustrated in Equation (A.4), the entering variable could be x_1, and the leaving would be x_6 as this has the tightest constraint, i.e. this is the equation where x_1 can be set to the lowest value without violating the non negativity constraint. So in the example x_1 should be converted into becoming a basis variable like this:

$$\begin{aligned} x_6 &= 10 - 2x_1 - x_2 - 2x_3 \\ x_1 &= 5 - \frac{1}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_6 \end{aligned}$$

This begins in line 13 where the constant of the new equation is calculated on basis of the old constant divided by the coefficient of the entering variable. Lines 16 to 21 then calculates the coefficients of the non-basic variables in the new equation, and line 24 calculates the coefficient of the leaving variable.

The third step is to replace any occurrence of the entering variable in the other constraints with the equation created in step two. In the example this would look like:

$$\begin{aligned} x_4 &= 23 - 2x_1 - x_2 - 3x_3 \\ x_4 &= 23 - 2(5 - \frac{1}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_6) - x_2 - 3x_3 \\ x_4 &= 13 - 0x_2 - 2x_3 + x_6 \end{aligned}$$

Line 27 iterates over the set of constraints, and ignores the one that was rewritten in step two. Lines 32 to 44 then performs the same operation as when it in the above example multiplies the coefficient of the entering variable with the equation for the entering variable, and then shortens the whole equation. The fourth step is to update the objective function which is done in lines 47 to 57, and this works in the same way as the previous. The last step is to update the sets of basic and non-basic variables, and then return the new slack form.

Listing A.4: The pivot algorithm used in the simplex algorithm. This is based upon the pseudocode from [3].

```
1  SlackForm SimplexPivot( int[] pNonBasicIndicies, int[] pBasicIndicies, float[,]
       pConstraintCoef, float[] pConstraintConst, float[] pObjCoef, float pObjConst,
       int pLeaving, int pEntering )
2  {
3      // 1. Create the variables for the new slack form
4      float[] newConstraintConst = pConstraintConst.Clone();
5      float[,] newConstraintCoef = pConstraintCoef.Clone();
6      float[] newObjCoef = pObjCoef.Clone();
7      float newObjConst = pObjConst;
8      int[] newBasicIndicies = pBasicIndicies.Clone();
9      int[] newNonBasicIndicies = pNonBasicIndicies.Clone();
10
11     // 2. Create a new equation with the entering variable as basis
12     // 2.1. The constant of the new equation is the constant of the leaving equation
           , divided with the coef of the entering
13     newConstraintConst[ pEntering ] = pConstraintConst[ pLeaving ] / pConstraintCoef
           [ pLeaving, pEntering ];
14
15     // 2.2. The coefs of the rest of the variables also requires a division with the
           coef of the entering
16     foreach (int NonBasicIndex in pNonBasicIndicies) {
17         if ( NonBasicIndex == pEntering )
18             continue;
19
20         newConstraintCoef[ pEntering, NonBasicIndex ] = pConstraintCoef[ pLeaving,
               NonBasicIndex ] / pConstraintCoef[ pLeaving, pEntering ];
21     }
22
23     // 2.3. The coef for the leaving in the new entering equation is 1 divided with
           the entering coef
24     newConstraintCoef = 1 / pConstraintCoef[ pLeaving, pEntering ];
25
26     // 3. Update the coeficients of the other constraints
27     foreach (int BasicIndex in pBasicIndicies) {
28         if ( BasicIndex == pLeaving ) // This is the equation that was fixed in 2.
29             continue;
30
31         // 3.1 Update the constant
32         newConstraintConst[ BasicIndex ] = pConstraintConst[ Basicindex ] -
               pConstraintCoef[ BasicIndex, pEntering ] * newConstraintConst[
               pEntering ];
33
34         foreach (int NonBasicIndex in pNonBasicIndicies) {
35             if ( NonBasicIndex == pEntering )
```

```
36                  continue;
37
38              // 3.2 Update the nonbasic coefficient
39              newConstraintCoef[ BasicIndex, NonBasicIndex ] = pConstraintCoef[
                     BasicIndex, NonBasicIndex ] - pConstraintCoef[ BasicIndex,
                     pEntering ] * newConstraintCoef[ pEntering, NonBasicIndex ];
40          }
41
42          // 3.3 Applies a constant to the leaving variable
43          newConstraintCoef[BasicIndex, pLeaving] = -1 * pConstraintCoef[BasicIndex,
                pEntering] * newConstraintCoef[ pEntering, pLeaving ];
44      }
45
46      // 4. Update the objective function
47      newObjConst = pObjConst + pObjCoef[ pEntering ] * pConstraintConst[ pEntering ];
48
49      foreach ( int NonBasicIndex in pNonBasicIndicies )
50      {
51          if ( NonBasicIndex == pEntering )
52              continue;
53
54          newObjCoef = pObjCoef[NonBasicIndex] - pObjCoef[pEntering] *
                newConstraintCoef[pEntering, NonBasicIndex];
55      }
56
57      newObjCoef[pLeaving] = -1 * pObjCoef[pEntering] * newConstraintCoef[ pEntering,
            pLeaving ];
58
59      // 6. Return to sender
60      // 6.1 Update the basic and nonbasic sets
61      newNonBasicIndicies.Remove( pEntering );
62      newNonBasicIndicies.Add( pLeaving );
63
64      newBasicIndicies.Remove( pLeaving );
65      newBasicIndicies.Add( pEntering );
66
67      // 6.2 Return the new slack form
68      return new SlackForm( newNonBasicIndicies, newBasicIndicies, newConstraintCoef,
            newConstraintConst, newObjCoef, newObjConst );
69  }
```

APPENDIX

B

Evaluation

B.1 Evaluation Agenda

The evaluation is divided into parts, where each parts should be discussed in detail and the following topics should be included.

B.1.1 The Formal Model

Present the user with pre-made behaviour trees, which include the different nodes available. Important to get feedback on the following:

- Remember to ask the test subjects background
- Discuss how each node in the tree is used
- Discuss the structure of behaviour trees, how each node is linked with another
- Describe how the execution of the tree is handled
- Present the idea behind paths in a tree, especially when sequences are involved
- Introduce probability values for each node in the tree, and how these values can be used to calculate the probability of reaching a path
- Does the formal model feel sufficient and flexible to any possible agent the test subject could think of

B.1.2 The Behaviour Editor

Present the user with the Behaviour Editor running inside Unity. The demo game is used as base for the discussion. Let the user play around with the tool and guide the user towards creating a simple tree and behaviour for the agent. Important to get feedback on the following:

- How does the tool feel compared with the formal model. Is all nodes and features denoted in the formal model possible in the tool
- Describe the implementation of interrupt trees and how they should be used
- Does the tool fit into the Unity environment, regarding the features and look and feel
- Is the tool intuitive to use, is it easy to draw trees in the canvas
- Is the inspector understandable, does it lack information regarding the tree
- How is the workflow of creating an agent for a game, could something be optimised

B.1.3 The Tool Extensions

Present the user with the new extensions to Behaviour Editor using mockups. The mockups is used to discuss the following:

- Discuss the idea behind agent diversity, and why it is important to maximise
- Discuss agent outcome, how the outcome can be presented. What could the different utility types be, the subject might have some to share
- How hard would it be to define utility types for a tree
- Discuss the idea behind ease the process with defining probability values using linear programming.
- Discuss why all the nodes is not automatic calculated
- Discuss the tool extensions, benchmark information in inspector, path markup and how the locking mechanism works
- Does these tools aid in the process of defining probability values
- Is it hard to define probability values, and find the proper place to define the values
- Introduce recommended node system

B.1.4 Agent Comparison

The user is presented with the demo game, where four different agents is competing against each other. The following is important to include in the evaluation:

- Describe the different agents for the user, and present the trees for each agent
- Present the game for the user, so the goal of the game is known
- Discuss how each agent perform in the game, how does they behave
- Is it possible to see the difference between the agents
- Is it beneficial to have a system, which can ease the process with creating agent with for instance different difficulty types.

B.2 Results

Four different test subjects where used for this interview, and the notes from each interview is presented below.

B.2.1 First Interview

Martin Midtgaard an SW10 student in the MI department of Aalborg University. Martin has no previous experience with behaviour trees or Unity3D.

Formal Model

- The behaviour seems easy enough for a designer to use, if the actions scripts is developed by programmers.
- The structure seems very static.
- It might be confusing when the tree is growing to big.
- The test subject understands the path and probability principle.
- Wants the tree to be reusable, often the same behaviours is used multiple times.

The Behaviour Editor

- The drag connection buttons does not make sense.
- Maybe use small boxes the connections can be pulled from.
- The tool is a 1 to 1 comparison with the formal model, except the graphical representation.
- Understands the idea behind the inspector.
- The interrupts might be confusing for some.
- The interrupts might invalidate the old action and it might be infeasible to execute the old action.
- In some conditions you might not want to interrupt the tree.
- The look and feel fits with Unity.
- The inspector could be placed in its own window, so it is it can be dragged.
- The presented workflow is how it should be done.
- He wants to be able to compile tree without scripts attached, so the tree can still be used even if it not done.
- Debug information regarding the runtime execution of the tree.

Tool Extensions

- It makes perfectly sense to discuss diversity for an agent.
- Utility values might be difficult to balance in a tree.
- If the utility value is mistyped the agents behaviour is wrong.
- It is impossible to define probability values by hand, so a tool would be needed.
- Is it always desirable to have a high amount of diversity?
- The package system is a nice addition to the tool.
- The locking system works, but it could also be nice to define the value when clicking on the arch.

- The utility and path information should also be present in the inspector.
- The information box in the canvas is nice.
- It is very nice to be able to execute the linear program again with new constraints.
- Wants a button to distribute the remaining values in the tree.
- The tool could be nice for a designer to use.

Agent Comparison

- It is a bit difficulty to see the different behaviour.
- The game should run a bit longer, then it is easier to see the difference.

B.2.2 Second Interview

Rasmus Kristensen a DAT6 student in the SSE department of Aalborg University. Rasmus has previous experience with behaviour trees or Unity3D.

Formal Model

- The model makes perfect sense, and describes how behavior trees is structured.
- The path principle makes sense and is a good addition.
- Preliminary values is understandable.
- The model is flexible and can be used for any thinkable agent.

The Behaviour Editor

- Drag and connect system is a bit broken.
- The tool and the formal model fits perfect together, so the tool is easy to use.
- The tool fits into Unity look and feel.

- The workflow seems to easy to be true, so it is kind of magic.
- The graphical representation should however fits more into the formal model.

Tool Extensions

- Diversity is important for him.
- It makes sense not to lock the designer into a specific utility type.
- It is not difficult to assign utility values, but it might be alternated by debugging.
- It is very needed with help to assign probability values.
- Makes sense not to auto generate all the values, as the information regarding the game world is important.
- The package system is sweet, removes generic behaviours and increase focus on agent specific tasks.
- The lock system make sense.
- Path markup is usable, especially in large unstructured trees.
- Expected utility is nice to compare agents.
- The path guideline system is understandable, and the recommended node system is very nice.
- The system is a nice addition as he is not required to define all probability values by hand.

Agent Comparison

- There is a clear difference in the agents behaviour.
- The random agent is truly random.

B.2.3 Third Interview

Allan Mørch Christensen a SW10 student in the MI department of Aalborg University. Allan Mørch Christensen has no previous experience with behaviour trees or Unity3D.

Formal Model

- The principle behind behaviour trees is understandable and feels nice for a designer.
- The path principle makes sense to introduce for behaviour trees.
- Wants to use the environment from the game to control the selection algorithm, which is possible.

The Behaviour Editor

- Needs tooltips for different nodes.
- Graphical representation is not 1 to 1 for the formal model.
- Connect and Drag system is a bit confusing.
- Beside the graphical representation the formal model and tool fits together.
- The inspector is great and clever.
- Besides the graphical glitches, the tool looks nice and fits with Unity.
- The interrupt system is important for the behaviour trees to be useful.
- Does find a bit troublesome with multiple trees.

Tool Extensions

- Diversity is a important factor for an agent.
- Does not feel it difficult to assign utility values, if in doubt just use a average value.
- It would be difficult to assign probability values in order to increase diversity and secure outcome.
- Would be clever to have a system that could be used after you have defined some values.
- The package system could be nice, but in doubt.

- The locking system seems to be working nice, but the markup is a bit to crazy.
- The numbers in the benchmarks window, might be confusing for a designer.
- Recommended node is a nice additions as long it is not required to define, it is not.
- The test subject find it very useful for a designer to have a system like this.

Agent Comparison

- It is possible to see the different behaviours.
- The red agent, the one implemented using the new system, is great and wins.

B.2.4 Fourth Interview

Anders Ejlersen a DAT6 student in the SSE department of Aalborg University. Anders Ejlersen has previous experience with behaviour trees or Unity3D.

Formal Model

- The graphical notation make sense, and behaviour trees is very intuitive to use.
- Path principle makes good sense.
- Find it hard to see any limitation of behaviour trees.

The Behaviour Editor

- The drag and connection system is a bit clunky.
- Why is there not and arrow on the arch.
- The tool and formal model fits together besides the graphical representation.

- The inspector might be placed in the opposite side, to fit more into Unity.
- The tool fits good together with Unity.
- The scripts system with public variables is a clever solution.
- The workflow seems quite simple and smart to use.

Tool Extensions

- Diversity makes sense to discuss about an agent.
- Can see the benefit from having the designer define his own utility type.
- Does not feel it should be difficult to assign utility values for all actions.
- It is is not difficult to assign probability values, but to define them so an outcome is secured is.
- It makes sense not to auto generate all values as the information about the game is lost.
- Package system is smart, but the designer might loose the overview of the tree.
- The graphical representation of the package node should be changed.
- The locking system makes sense and easy to use.
- Path markup feel like a great addition.
- The graphical representation of the new tool is very nice.
- Does not initially wants a recommended node, wants to be in control.
- Propose the same solution as us, when discussing the variable problem.
- The system feels like a great aid in defining probability values.

Agent Comparison

- The is a clear difference between the agents.
- Yes the red agents is clearly more as intended as the rest.

B.3 Behaviour Trees

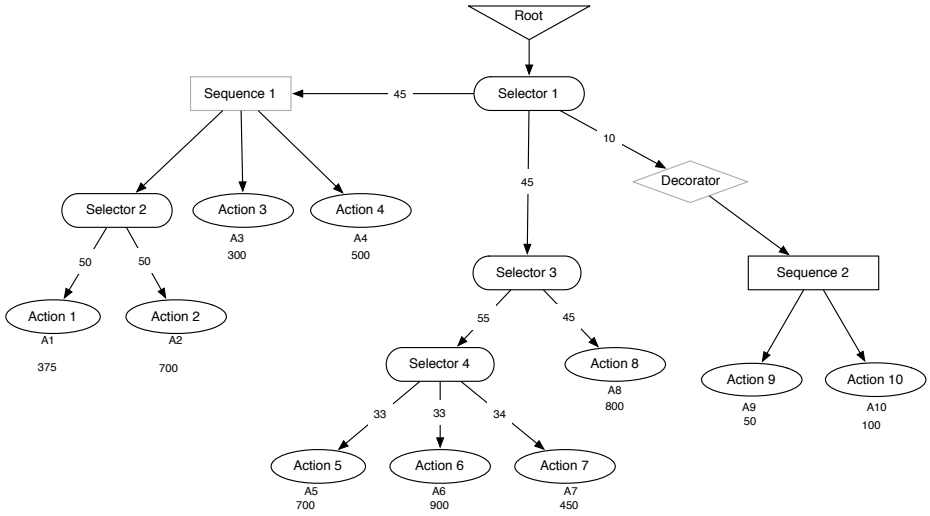


Figure B.1: An illustration of a behaviour tree

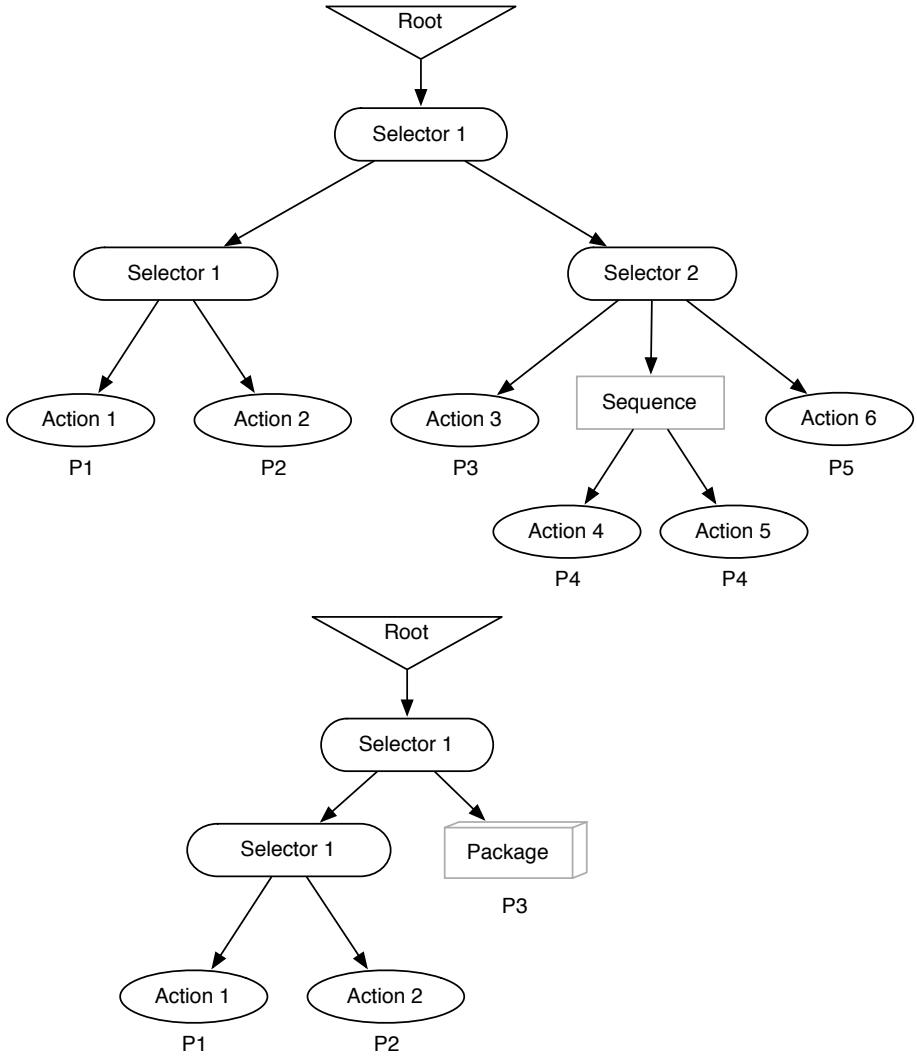


Figure B.2: An illustration on how the package node works

B.4 Mockups

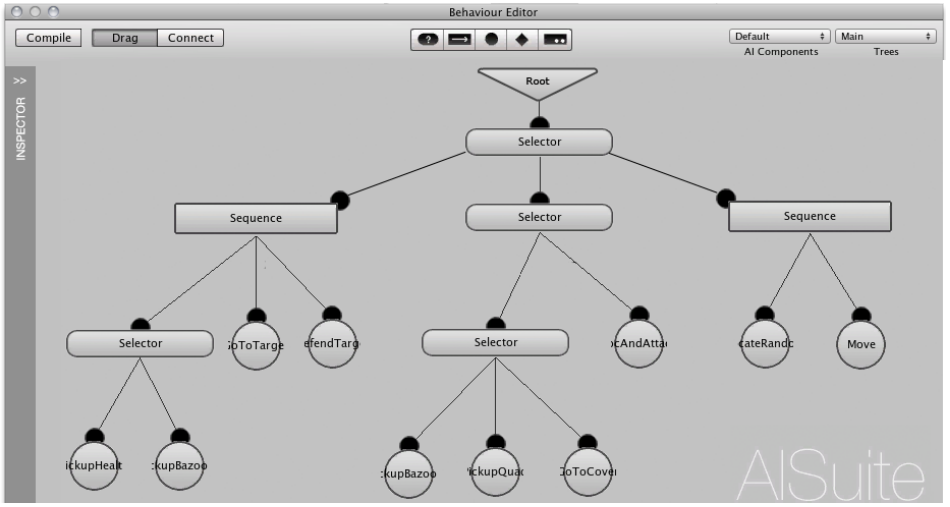


Figure B.3: A behaviour tree as it looks in the Behaviour Editor

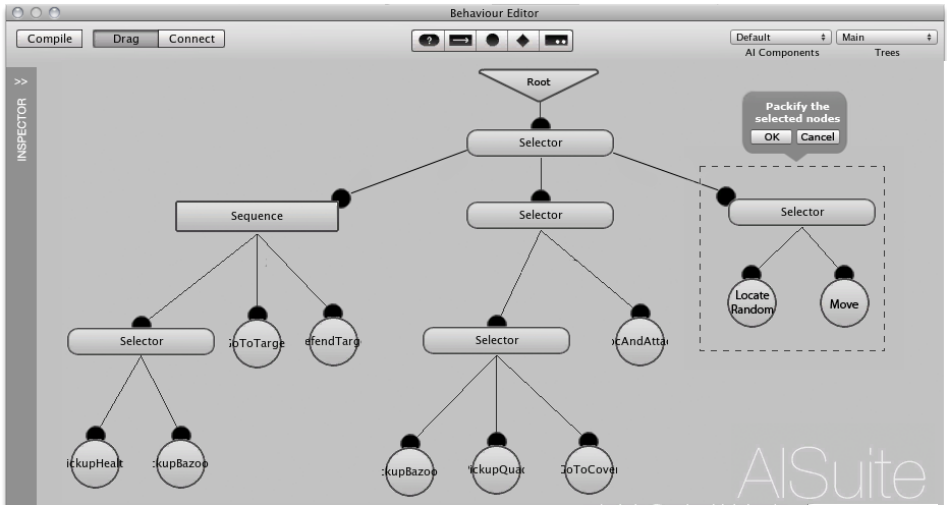


Figure B.4: A subtree is recognised as containing a generic behaviour and is therefore converted into a package.

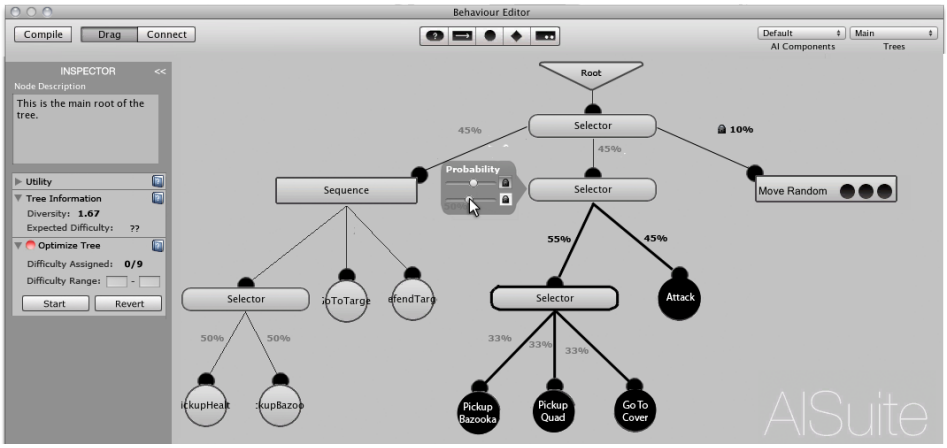


Figure B.5: Probability values has been defined by the AI designer

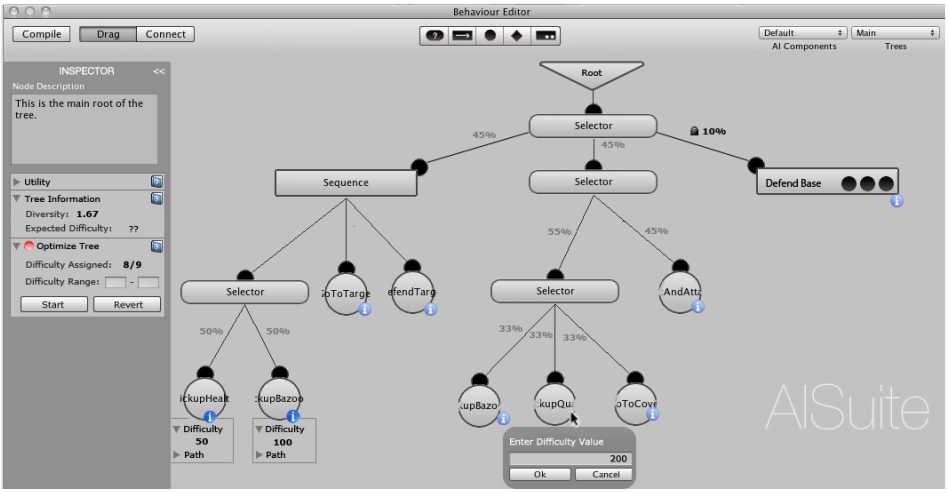


Figure B.6: The inspector shows the benchmark values such as diversity and expected utility, and also makes it possible to change the utility type and create the path probability distribution. The main canvas shows the utility values as defined by the AI designer.

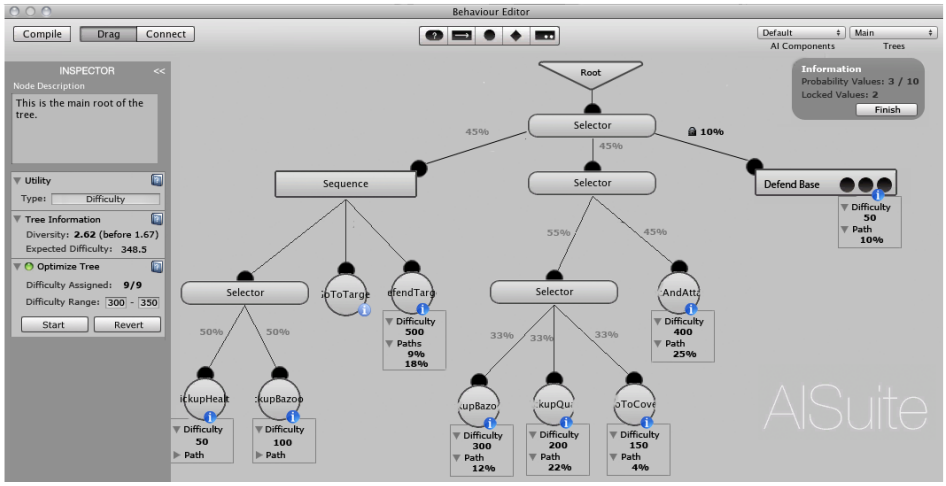


Figure B.7: The probability distribution for each path has been calculated

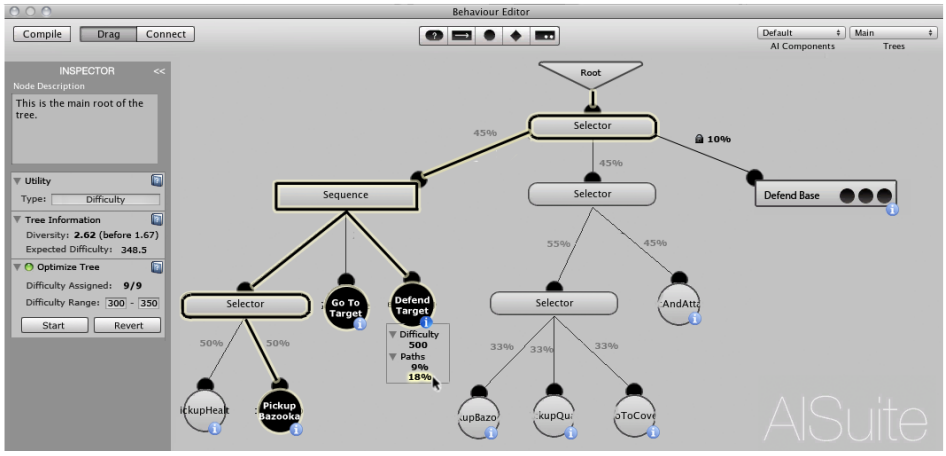


Figure B.8: By selecting a path in the drop down menu it will highlight all of the nodes in the path.

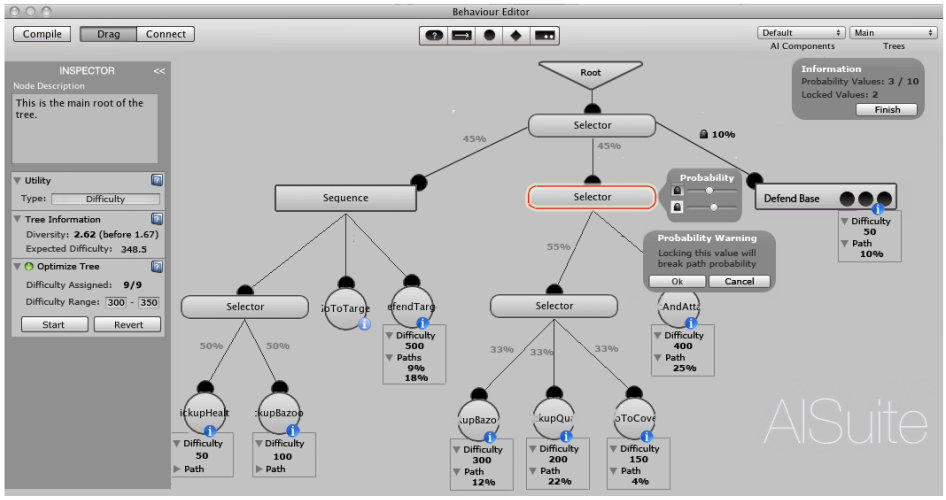


Figure B.9: Recommended node system and a warning due to poorly defined variable.

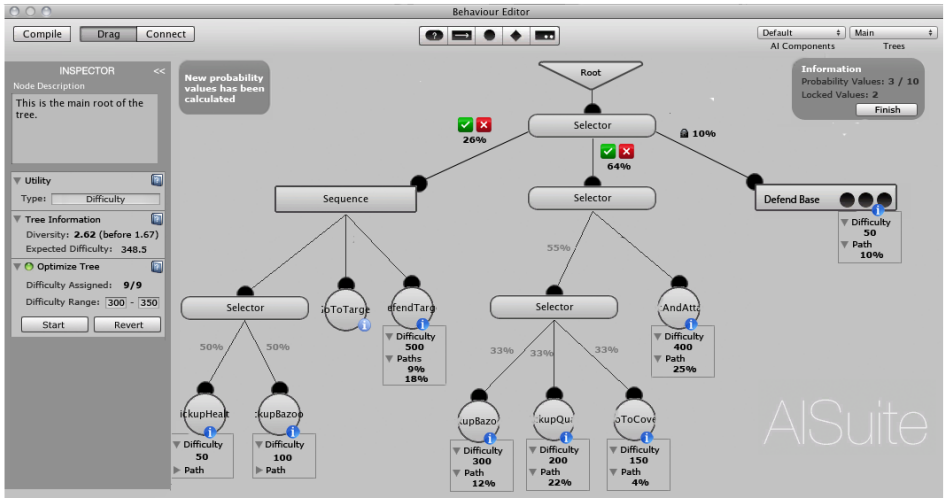


Figure B.10: The tool has found some new trivial probability values and they are then presented for approval.

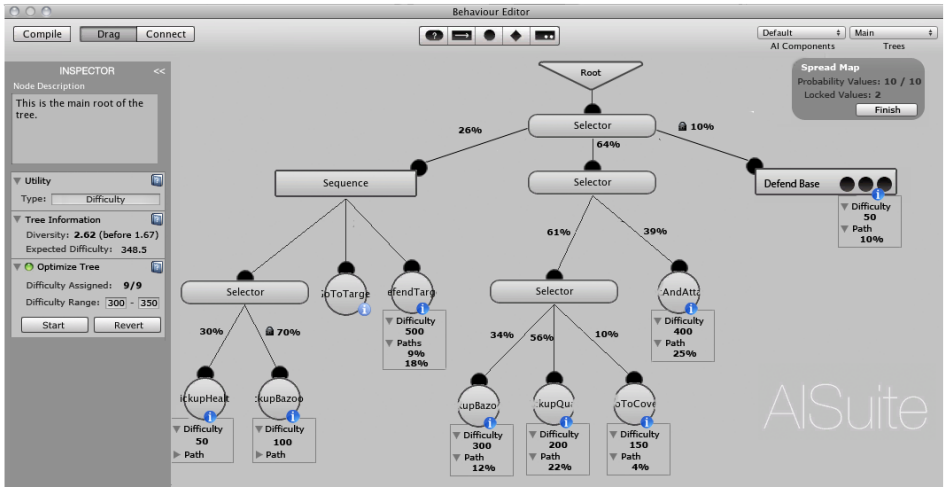


Figure B.11: The final tree, with all probability values defined.

B.5 Agents

B.5.1 Random Agent

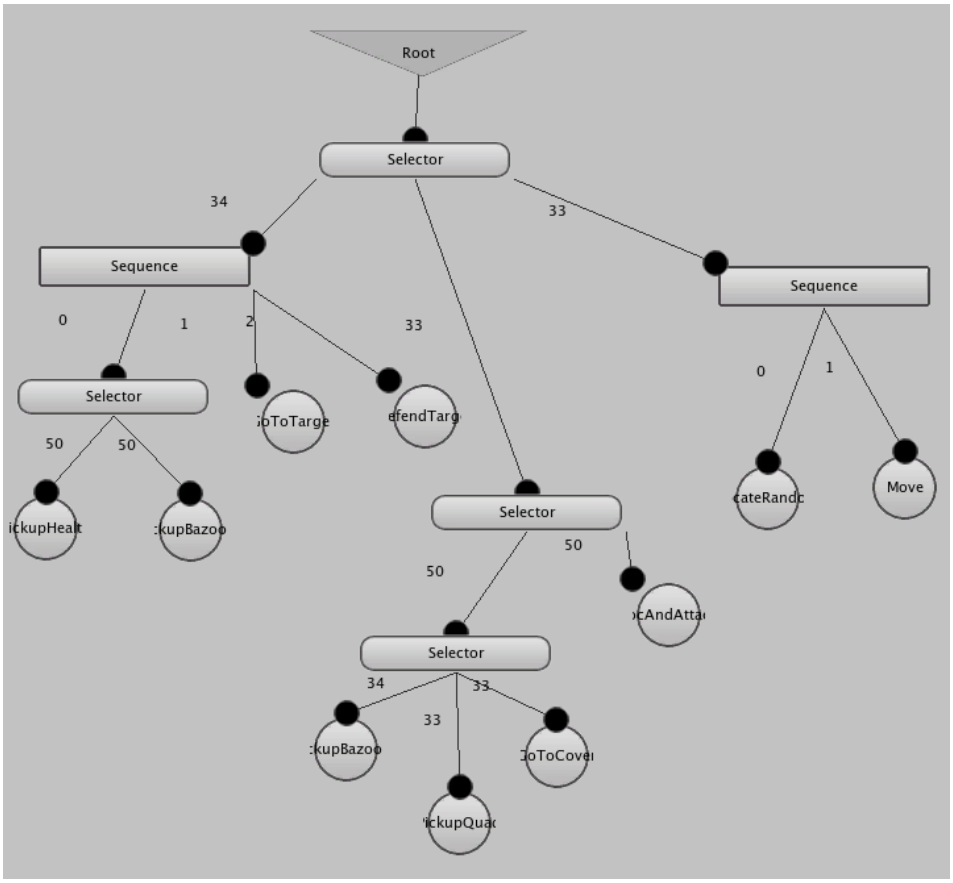


Figure B.12: The main tree used by the random agent in the demo game

B.5.2 Manually Assigned Probability Values

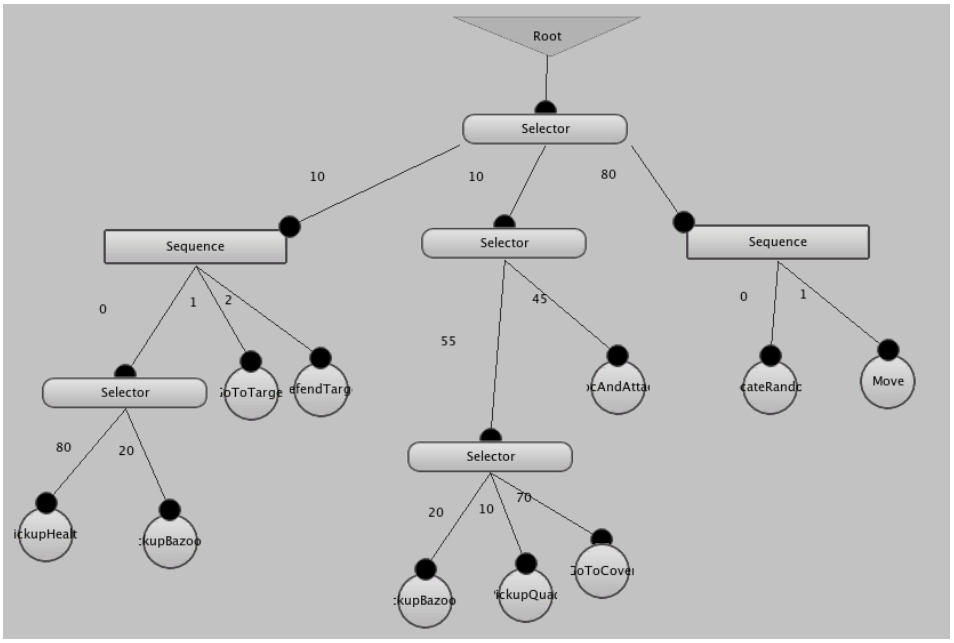


Figure B.13: The main tree used by the manually assigned probability value agent in the demo game

APPENDIX

C

Resume

The report documents the development of a set of tools and techniques designed to aid the process of creating agents using an AI modelling technique known as Behaviour Trees.

The reports starts with an formal model of behaviour trees. This presents the tree structure and the different nodes used in the model and thereby creates a common foundation that supports the new techniques. By using the formal model it would enable other developers to use the presented techniques as well. The formal model then creates basis for a presentation of the problem, which is to ease the development of agent using behaviour trees. This must be done by finding techniques to ease the assignment of probability values in the selector nodes that is set to use this selection scheme.

In the agent creation process the AI designer tries to achieve a certain agent outcome, while maximising diversity in its selection scheme. To create an technique that achieves this measures must be found for diversity and agent outcome that fits with behaviour trees.

The concept of entropy is researched and the formulae for Shannon entropy is tested to work with behaviour trees by utilising it to calculate the diversity of the paths in the behaviour tree. To measure the agent outcome the authors use the known concept of utility and presents how this can be defined by the AI designer. It is important to note that the agent outcome measures some property of the agent which must be set by the AI designer. The property is a characteristic of the agent, e.g. difficult or aggressive.

These measures are then used in correlation with the mathematical technique linear programming to create the optimal path probability distribution with regard to diversity and a AI designer specified level of agent outcome. Having this probability distribution allows the tool to automatically calculate probability values in the behaviour tree whenever enough probability values has been defined by the AI designer. A method for recommending nodes to the AI designer is also presented. This method is based upon sensitivity analysis which basically finds the selector node that if defined has the largest impact than any other undefined selector

node on the agents behaviour.

To utilise these techniques a series of tool extensions are presented. The first tool extension presents benchmark data regarding the behaviour tree, like diversity and expected utility. The next tool extension tries to help the AI designer to keep focus on the assignment of probability values of a certain path by highlighting it. To specify that certain variables in the behaviour tree are final configurations the AI designer can lock them. This is used to extend the principle of auto generating probability values to calculate the values of the unlocked variables using the locked variables and the path probability distribution. It is possible for the AI designer to specify probability values that renders it impossible to achieve the path probability distribution, when this happens the AI designer must either recalculate a new path probability distribution with the current settings as constraints in the linear program or redefine the probability values in the behaviour tree. The last tool extension shows how to divide and conquer the behaviour trees by placing generic behaviours in packages and thereby only see them as one in the main behaviour tree. Lastly a workflow is presented which takes all of the new tool extensions into consideration in the process of creating a behaviour tree.

The techniques and the tools are then evaluated by a series of test subjects who finds them to ease the development of agents using behaviour trees considerately. A demonstration game is created where three agents are competing against each other. One of these agents are created using the new techniques and after having performed a series of tests it is concluded that it performed as expected.