**Computational Modeling of Intelligent Behavior**

**by**

**Lindsey Brianna Harris**

**An Honors Capstone**

**submitted in partial fulfillment of the requirements**

**for the Honors Diploma**

**to**

**The Honors College**

**of**

**The University of Alabama in Huntsville**

**Nov 11, 2016**

**Honors Capstone Director: Dr. Daniel Rochowiak**

**Associate Dean, College of Science**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Student Date**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Director Date**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Department Chair Date**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Honors College Dean Date**

**Table of Contents**

Dedication……………………………………………………………..………………….. 2

Abstract……………………………………………………………….…………………… 3

Introduction……………………………………………………………………………….. 4

Project Description……………………………………………………………………….. x

Iteration 0: Preliminary Structures……….…………………………………………….. x

Iteration 1: The Simple Case..………………………………………………………….. x

Iteration 2: One-Step Memory………....…..……..…………………………………..… x

Iteration 3: Story Generator Application……………………………………………….. x

Demo………………………………………………………………………………………. x

Conclusion………………………………………..………………………………………. x

References…………………….…………………...……………………………………. x

Appendix A…………………………………………………………………………………x

**Dedication:**

This Capstone Project is dedicated to my mother, Angela Harris, who inspired me to become a Computer Science major and to explore programming and other such computer science skills.

I dedicated this also to the love of my life, Matthew Pittenger, who inspires me to work hard and solve problems on a regular basis. Without whom, my problem-solving skills would not be nearly as cultivated. Thank you for solving so many challenges alongside me.

Lastly, I dedicate this to my fellow Computer Science majors, who hopefully will find this matter to be useful and/or inspirational as they continue in their studies.

**Abstract** //To Be Redone

The problem of modeling intelligent behavior deals with varied probabilities of actions as well as patterns of action distribution in behavioral trees. It is simple to model a perfectly rational entity, which makes the rationally correct choice all of the time. However, as humans do not live perfectly rational lives, a perfectly rational model is not very realistic. Modeling human-like behavior accurately, therefore, must also take into account the irrational things that humans do that still seem to make sense. Some of the “irrationalities” are based on social norms, while others are based on habits. Utilizing a number of iterative steps, the branch of action that the model takes could change based on monitoring that occurs during the previous steps that the model has taken. With that in mind, the purpose of this project is to research the probabilities and behavior structures necessary to produce a reusable software component that realistically models human-like, artificial intelligence.

That is pretty good. I think you have all of the major points. Let’s see how far you go in developing it.

**Introduction**

//To be done after the rest of the thesis is written (below)

//also temporarily to be used as a todo list

// Make sure to Discuss language somewhere in the document: i.e. “choice” vs. “option” and add in the notions of habit and aversion

**Project Description**

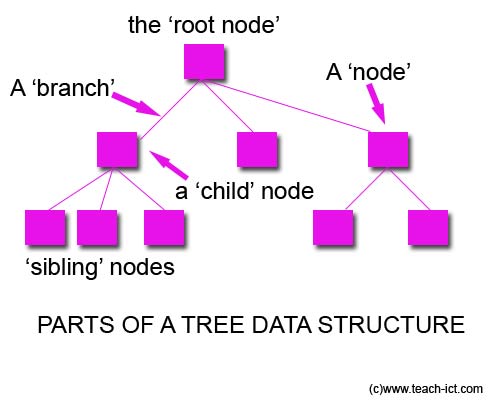
The purpose of this project is to explore the usability of a probability tree structure for behavioral AI and to demonstrate the structure’s effectiveness in an artificial intelligence setting. This will be accomplished through a series of iterative program designs that will build on each other and will eventually result in a finalized proof of concept program that will demonstrate the probability tree structure’s usability. As a proof of concept, a random story generator program, which uses the probability tree structure to select options from a given story line, will be designed and implemented. The iterations of this project will include a preliminary overview and exploration of the structures to be used (iteration 0), an implementation of the “simple case”, which will be a time-independent implementation that will select options from the same choice repeatedly in order to test the basic structure (iteration 1), and an implementation which will include a one-step memory (iteration 2). Finally, there will be a demo story tree that can be used by the generator to demonstrate the iterations of the project. This project will be programmed in Java, however, it must be noted that the implementation will be covered within the appendixes. The goal of this document is not to discuss the implementation of the project, but rather to share the concepts which others can build upon.

**Preliminary Structures**

To understand the implementation of this project, it is crucial to first understand the structures that are being discussed throughout the text, namely the object of our study: the probability tree.

**The Basic Tree**

Anyone who has much experience with programming data structures should be aware of the tree data structure. Trees are hierarchical structures consisting of branches and nodes, where a node is a structure that holds data and a branch is a connection between two nodes. The “root” node is the base of the tree, and all of the other “leaf” nodes derive from it. The nodes directly connected to the root node are called the root’s children, and each of these “child” nodes can have child nodes of their own. A node with children connected to it is called a “parent” node. For a visual example of what a tree looks like, see Figure 1 below.



**Figure 1 (Google Images)**

Trees are used in a variety of applications, and are often used to store data that can be sorted in some hierarchical format. Often times, this makes it much easier for programmers to search for specific pieces of data more quickly, and it also allows users to make varying paths from the root to an ending leaf node. The path application is what will be used here to create the different stories in the story generator program. Each parent node will be a “choice” that the program can make, while the child nodes will each be an option that results in another choice. While this is quite useful, it is not enough to simply have a basic tree of choices to create a believable AI. To add some variability that will make the AI more believable, a special type of tree called a probability tree will be used for the story generator program.

**The Probability Tree**

A probability tree is a tree with multiple children whose nodes contain probability data. This probability data can be utilized by the probability tree’s extra functionality to determine what percentage of the time a given child node ought to be selected. For example, let us look at a simple tree where parent node P has branches to its child nodes A, B, and C. Let us also say that the user wants to randomly select one of the children. Using a normal tree, random selection would have to be implemented outside of the tree structure, and would likely give equal opportunity to each of the child nodes because there is nothing in the tree that conveys any sort of priority. In a probability tree, however, the functionality allows the user to select nodes based on the priority (probability) of the given node.

For example, looking at a tree with parent node P, let us say that node P is the choice of what to do when you are inside of a burning building and you see someone trapped behind a beam that you are capable of moving. Let us say that choice A is to help the person get out from behind the beam and then leave, choice B is to leave the building and to tell a firefighter that you saw a person trapped and where you saw them, and choice C is to leave the building and say nothing because the firefighters have it handled. If attempting to design an AI that selects one of these options using a normal tree structure, there is no way to determine which path should be chosen if one should be chosen more often than another. The only way to implement the selection process with a normal tree is to have a selector implemented outside of the tree. This selector then has two options: it can either select randomly, giving equal opportunity to each of the child nodes, which is highly unpredictable, OR it can select unevenly, giving weight to each of the options. However, the weighted approach would have to be based on the order of the nodes, and could not be based in the value of the node itself. This is because the weights are not implemented within the nodes themselves, but within the selection process. In other words, a weighted implementation might select node A 50% of the time, node B 30% of the time, and node C 20% of the time, but it would give those same probabilities to every choice that was made in that order, because the weights are tied to the selection process instead of to the individual option nodes.

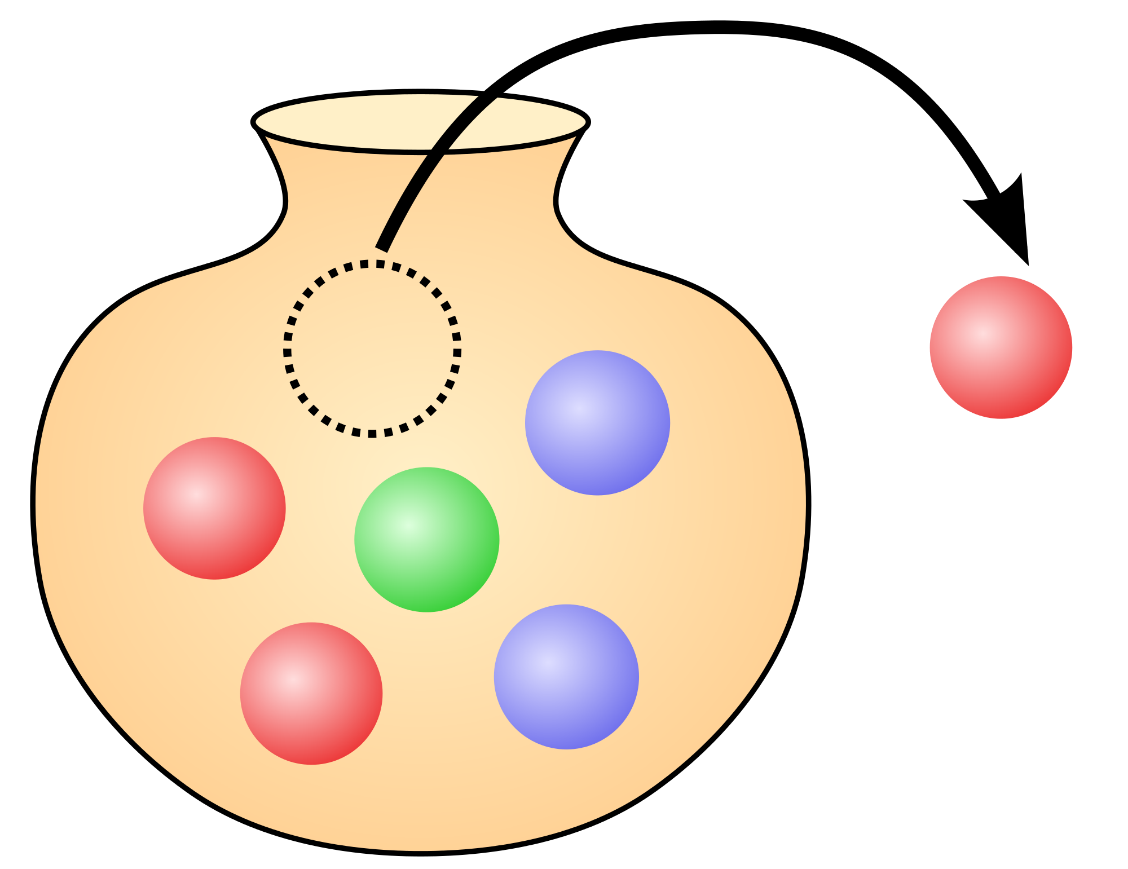
With a probability tree, we can say that node A (choice A) has the probability value of 0.7, node B (choice B) has the probability value of 0.2, and node C (choice C) has the probability value of 0.1. When the user goes to select a child of P, now there is data that can be used such that you rescue the person 70% of the time, leave and tell a firefighter about the person 20% of the time, and ignore the person only 10% of the time. Given another choice, you would be able to set node A’s probability value to 15%, node B’s probability value to 40%, and node C’s value to 45%. As you can see, this enables the tree to determine the value of each choice, and simplifies implementing a useful and adaptable artificial intelligence. Other methods, such as completely random selection or weighted selection carry with them simplicity as well, but they are not nearly as flexible or realistic as the probability tree implementation. Case in point, with the firefighter example, a normal tree would make all characters utilizing the tree approximately the same, whereas with the probability tree, you could have some characters more likely to be a hero, or some to be more of an oblivious person. Therefore, a probability tree can be used to add personality to different AI characters, which helps make the characters realistic. This added realism makes the probability tree structure ideal over some of the other AI implementations. For probability tree implementation details, see Appendix A.

**The Simple Case**

One of the simplest examples of using a probability tree is the case where a single choice is made repeatedly. Selecting one option within a choice has no effect on selecting an option the next time that same choice is made. Mathematically, this can be equated to an example commonly used in Probability and Statistics courses, the urn problem.

**The Urn Problem**

An urn problem is a probability problem that uses an urn (hence the name) and a selection of different colored marbles. The marbles are all put into the urn and one marble is randomly selected out of it, as can be seen in Figure 2 below.



**Figure 2 (Google Images)**

If the marble is put back into the urn before another marble is selected, it is said that this is an urn problem with replacement. With replacement, it is equally likely to pull a marble of the same color out of the urn as before. Our simple case works much like the urn problem with replacement. When we have a choice with options to select, every option has the same likelihood of being chosen each time the choice is made. In this case, the choice is the urn and each option is a different color of marble. The probability of each option correlates to the number of each type of marble in the urn.

**Simple Case in Action**

Let's apply this simple case to a sample probability tree to see what this type of selection process looks like. Take, for instance, a choice that involves traveling down a path. When you come to a split in the path, this case consists of 3 options: turn left, turn right, or go straight. The left path is broad and flat, the right path leads into a questionable part of a dark forest, and the center path is incredibly steep, but all pathways lead to your destination. The left pathway is the farthest distance, the steep pathway is the middle distance, and the dark woods are the shortest distance. Let us also say that you have difficulty with steep hills and are slightly afraid of the dark. On the average day, you are 50% likely to take the path on the left, 30% likely to go straight up the steep path, and 20% likely to venture into the dark wood.



**Figure 3 (Google Images)**

If you come to this route on 10,000 different days, the chances of taking each path does not change from day to day. So over the course of the 10,000 days, you should take the left path ~5,000 times, the straight path ~3,000 times, and the right path ~2,000 times. This is great for simple choice-making, and makes characters slightly more believable than complete random selection when applied to artificial intelligence. However, this method would make all characters using the same probability look approximately the same. To give characters a little more personality, it is necessary to allow them to build habits, which will then differentiate them from other characters using the same probability tree. To do this, we need to examine the second iteration of the project, the one-step memory.

**One-Step Memory**

Conceptually, the one-step memory problem is exactly what it sounds like. With this case, when making a choice, selecting an option will affect the probabilities of the options the next time the choice is made. The goal of the one-step memory is to simulate making decisions based on previous decisions. This could be applied to forming a habit or avoiding unpleasant scenarios.

**Forming a Habit**

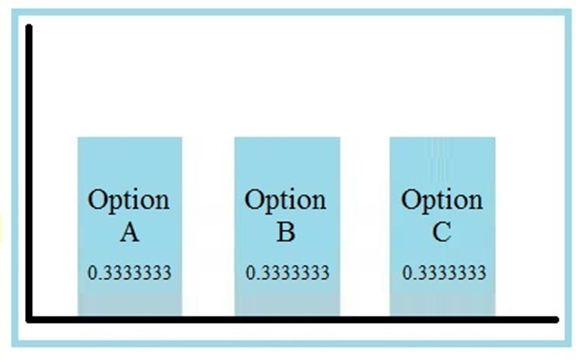
Human beings are often said to be creatures of habit, a habit being an act that is repeated frequently over the course of time. For example, washing your hands after going to the bathroom is a habit. Oftentimes, people see an individual's habits or mannerisms as a part of that individual's personality. With washing one’s hands, people see it as a good habit and may look down upon any person that does not wash their hands when they leave the restroom. The more often you wash your hands, the more likely you are to continue washing your hands. However, if a person does not wash their hands in the restroom regularly, they are less likely to do so regularly in the future. Therefore, when trying to construct a believable AI, it is important to be able to incorporate habits into the decision-making algorithm. In order to accomplish this, when one option of a choice is selected one time, it should be more likely for that option to be selected again the next time that choice is made.

**Avoiding Unpleasant Scenarios**

Sometimes there are choices that end badly. Humans do not repeat every single choice that they make—especially if selecting an option results in an unpleasant outcome. In these cases, a habit could be formed, but oftentimes people are less likely to select an option that gave them a previously bad experience. For example, if a person gets food poisoning at a restaurant, the likelihood of them returning to that restaurant the next time they go out to eat is significantly smaller than if they had not received food poisoning. (Might be a good place for a reference? You might use the following in various places) Similar to the concept of forming habits, it may be desirable to apply this concept to our AI to make it more believable. To do this, when an undesirable option is selected, then the next time that choice is made, it should be less likely for that option to be selected again.

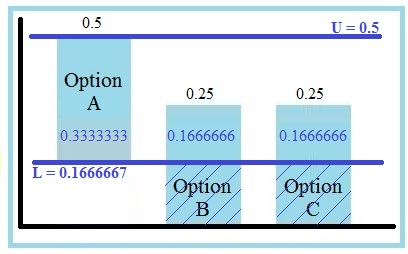
**Bounding Method**

Separately, it is simple to create a habit-building and an avoidance algorithm, but how do we effectively combine these concepts to work together when one is adding to the probability of the event and one is subtracting from it? One method of altering probabilities is bounding. Using upper and lower bounds can enable a shift in the dynamic of the selection process. The lower bound can set the lower limit for the selected option the next time a choice is made, and the upper bound can set off the block for the new probabilities. For example, say you have a choice with three options, A, B, and C, as laid out in Figure 3 below.



**Figure 3**

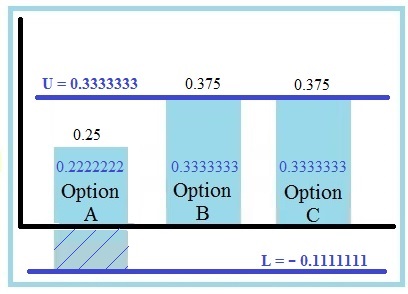
Each of these options has an equal probability of being selected. If we want to change the probabilities to build a habit or an aversion, then we introduce upper and lower bounds. Let Option A be the option that is selected. If the end goal is to build a habit, then the probability for Option A should increase. To do this, the lower bound is positive, and the upper bound is set greater than or equal to the probability of A plus the lower bound, as can be seen in Figure 4 below.



**Figure 4**

Here, we set the lower bound to 0.1666667, which cuts through the middle of options B and C. Option A builds on top of that, and here we set the upper bound equal to the probability of A plus the lower bound, which is 0.5. This leaves the probabilities of A, B, and C at 0.3333333, 0.1666666, and 0.1666666 respectively. As 0.3333333, 0.1666666, and 0.1666666 do not add up to 1, the probabilities are normalized, which brings the new probabilities to A = 0.5, B = 0.25, and C = 0.25. Because 0.5 is greater than A’s previous probability of 0.3333333, this effectively builds a habit when A is chosen.

If instead, we wanted to form an aversion to an option, we could set the lower bound as a negative value. Setting the lower bound as a negative value would subtract from probability of the selected option, and would then act as though the lower bound was zero for the remainder of the options in the choice. An example of this can be seen in Figure 5 below.



**Figure 5**

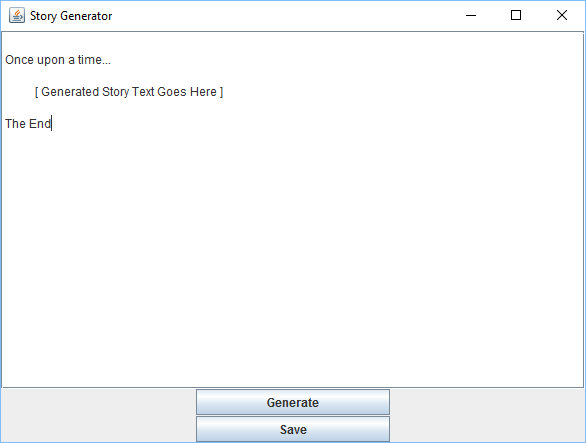
Here, we set the lower bound to -0.1111111, which subtracts 0.1111111 from probability A. By setting the upper bound at 0.3333333, this leaves the probabilities for A, B, and C at 0.2222222, 0.3333333, and 0.3333333 respectively. After normalizing, this results in final probabilities of A = 0.25, B = 0.375, and C = 0.375. This made the probability of A go down, and the probabilities of B and C go up, effectively creating an aversion to Option A. By giving each option a different upper and lower bound, this enables the user to create any number of combinations of habits and aversions that can be used within a character-building AI.

**Story Generator Application**

Now that we've spent some time using the probability tree structure and how we can apply it to artificial intelligence, let us examine the focal point of the project: the Story Generator Application. The primary goal of this project is to explore the probability tree structure and apply it to AI to create believable characters. The Story Generator Application seeks to fulfill this goal by utilizing the probability tree to generate varying stories based on the likelihood of certain events occurring. Ideally, this application will use the one-step memory to form believable habits for the characters involved.

**What it Looks Like**

The Story Generator Application from the user’s standpoint is fairly simple, consisting of a text field, a generate button, and a save button, which can be seen in Figure 6 below.



**Figure 6**

**How it Works**

The Story Generator Application uses a preset probability tree, which contains all of the story scene information. When the user clicks the “Generate” button, the application moves through the probability tree scenes and displays the scene text for each of the chosen scenes. This text will reset every time the user clicks the “Generate” button, and a new story will be generated. The generation algorithm will utilize the one step memory within the probability tree selection process to string scenes together for the story. The setup of the probability tree will determine how the one-step memory affects the outcome of the story. For implementation details, see Appendix B (add Appendix B).

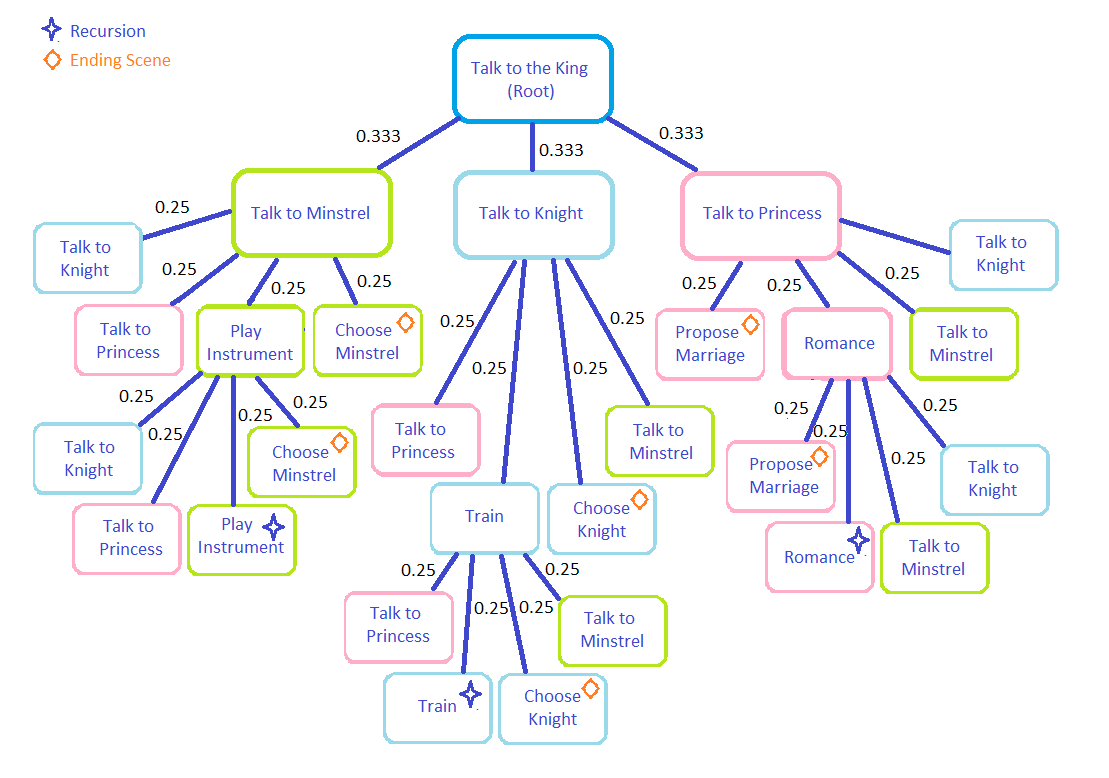
**Future Improvements**

The Story Generator application itself is a fairly simple application designed for the purpose of illustrating the probability tree structure with AI. Some suggested improvements of this application include:

* Loading a custom Probability Tree File
* Graphics for Story Illustrations

**Demo**

To demo the Story Generator Application, we will utilize the following probability tree:



**Figure 7**

This visualization includes the nodes and probabilities. There are 13 nodes used in this demo tree, and each starts with an equal chance of selection. Some of these nodes are recursive, which will allow the one-step memory to really expand bias towards certain choices. Other nodes are endpoints for the story. The upper and lower bounds applied to each node can be seen in Table 1 on the next page.

**Table 1. Demo Tree Bounds**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Choice Node*** | ***Initial Probability*** | ***Lower Bound*** | ***Upper Bound*** |
| Talk to the King (Root) | 1.0 | 0.0 | 1.0 |
| Talk to Minstrel  (1st level) | 0.3333333 | 0.03 | 1.0 |
| Talk to Knight  (1st level) | 0.3333333 | 0.03 | 1.0 |
| Talk to Princess  (1st level) | 0.3333333 | 0.03 | 1.0 |
| Talk to Minstrel  (2nd level) | 0.25 | 0.03 | 1.0 |
| Play Instrument | 0.25 | 0.03 | 1.0 |
| Choose Minstrel | 0.25 | 0.03 | 1.0 |
| Talk to Knight  (2nd level) | 0.25 | 0.03 | 1.0 |
| Train | 0.25 | 0.03 | 1.0 |
| Choose Knight | 0.25 | 0.03 | 1.0 |
| Talk to Princess  (2nd level) | 0.25 | 0.03 | 1.0 |
| Romance | 0.25 | 0.03 | 1.0 |
| Propose | 0.25 | 0.03 | 1.0 |

As can be seen in Table 1, the bounds provide a habit-building structure for the demo tree.

**Demo Story Outline**

The demo story follows a young man who has grown up in the king’s palace. The king has decided that it is time for his ward to decide what he wants to do with his life. Thus, the young man has to talk to the court minstrel and the captain of the guard to determine whether he would prefer to be an entertainer or to defend the kingdom. There is also a princess visiting from another kingdom who fancies his eye. If he marries her, he could take on the duties of a prince. He just has to decide what he wants by the end of the day. Given this scenario, there are a number of interactions that can occur, as outlined in Figure 7. Based on those interactions, eventually one of the given results of becoming a minstrel, becoming a knight, or marrying a princess will occur, and the story will come to an end.

**Demo Results**

**Conclusion**

[Did the probability tree work with the story generator? What worked? What didn't? What could be improved if someone was to expand this project? (a lot). Etc. Etc. Expound upon the details.]

**References**

Lockton, Dan. “Cognitive biases, heuristics and decision-making in design for behaviour change.” http://danlockton.co.uk. 2012. (Working Paper)

Mark, Dave. *Behavioral mathematics for game AI*. Boston, MA: Course Technology Cengage Learning, 2009.

Paliath, Vivin. "Generic (n-ary) Tree in Java." Vivin.net. N.p., 30 Jan. 2010. Web. 11 Nov. 2016. http://dspace.brunel.ac.uk/bitstream/2438/6706/2/Fultext.pdf

Treanor, Mike, et al. "AI-Based Game Design Patterns." University Georgia Institute Of Technology. 2015. Web. 11 Nov. 2016. http://www.cc.gatech.edu/~azook3/paper/treanor-fdg-2015.pdf

REFERENCED UNSITED:

http://www.vogella.com/tutorials/JavaIO/article.html

**Appendix A**

The probability tree code was adapted from Paliath’s code for generic n-ary trees, adding the probabilities and selection processes as follows.

**ProbabilityTree.java**

package storygenerator;

import java.util.List;

import java.util.LinkedList;

import java.util.Random;

/\*\*

\* @author Lindsey Harris

\*/

class ProbabilitySelector<T> // Straightforward Probability Selection

{

double sum;

Random randomGenerator;

public ProbabilitySelector()

{

super();

randomGenerator = new Random();

sum = 0.0;

}

public ProbabilityNode<T> selectOption(ProbabilityNode<T> choice)

{

if(!choice.hasChildren())

{

ProbabilityNode<T> nullNode = new ProbabilityNode();

return nullNode;

}

choice.normalize();

sum = choice.getChildAt(0).getProbability();

// Get number in range 0 - 999

int randomInt = randomGenerator.nextInt(1000);

if(randomInt <= sum\*1000)

{

return choice.getChildAt(0);

}

else

{

int i = 0;

sum = 0.0;

while (((sum \* 1000) < randomInt) &&

(i<choice.getNumChildren()))

{

sum += choice.getChildAt(i).getProbability();

i++;

}

return choice.getChildAt(i-1);

}

}

}

class AdvProbabilitySelector<T> // One-Step Memory

{

double sum;

int upperBound;

int lowerBound;

Random randomGenerator;

public AdvProbabilitySelector()

{

super();

this.randomGenerator = new Random();

this.sum = 0.0;

this.upperBound = 1;

this.lowerBound = 0;

}

public ProbabilityNode<T> selectOption(ProbabilityNode<T> choice)

{

// Make sure choice has options to choose from

if(!choice.hasChildren())

{

ProbabilityNode<T> nullNode = new ProbabilityNode();

return nullNode;

}

//Make sure choice probabilities are distributed properly

choice.normalize();

// Initialize randomInt and sum

int randomInt = randomGenerator.nextInt(1000);

sum = choice.getChildAt(0).getProbability();

ProbabilityNode<T> selectedNode;

if(randomInt <= sum\*1000) {

selectedNode = choice.getChildAt(0);

}

else

{

int i = 0;

sum = 0.0;

while (((sum \* 1000) < randomInt) &&

(i < choice.getNumChildren()))

{

sum += choice.getChildAt(i).getProbability();

i++;

}

if(i>0){ selectedNode = choice.getChildAt(i-1); }

else { selectedNode = choice.getChildAt(i);}

}

double lb = selectedNode.lowerBound;

double ub = selectedNode.upperBound;

selectedNode.setProbability(selectedNode.getProbability()+lb);

if(selectedNode.getProbability() < 0)

{

selectedNode.setProbability(0.0);

}

if(lb < 0)

{

lb = 0.0;

}

for(int i = 0; i < choice.getNumChildren(); i++)

{

ProbabilityNode<T> temp = choice.getChildAt(i);

if(temp.getProbability() > ub)

{

choice.getChildAt(i).setProbability(ub);

}

double newProb = temp.getProbability()-lb;

if(newProb < 0.0){newProb = 0.0;}

choice.getChildAt(i).setProbability(newProb);

}

choice.normalize();

return selectedNode;

}

}

class ProbabilityNode<T>

{

public T nodeObject;

double probability;

double upperBound;

double lowerBound;

public List<ProbabilityNode<T>> children;

public ProbabilityNode()

{

this.children = new LinkedList<ProbabilityNode<T>>();

this.upperBound = 1.0;

this.lowerBound = 0.0;

}

public ProbabilityNode(T object, double prob)

{

this.children = new LinkedList<ProbabilityNode<T>>();

this.nodeObject = object;

this.probability = prob;

this.upperBound = 1.0;

this.lowerBound = 0.0;

}

public ProbabilityNode(T object,

double prob,

LinkedList<ProbabilityNode<T>> childList

)

{

this.nodeObject = object;

this.probability = prob;

this.children = childList;

this.upperBound = 1.0;

this.lowerBound = 0.0;

}

public ProbabilityNode(T object, double prob,

double lowBnd, double upBnd)

{

this.children = new LinkedList<ProbabilityNode<T>>();

this.nodeObject = object;

this.probability = prob;

this.upperBound = upBnd;

this.lowerBound = lowBnd;

}

public ProbabilityNode(T object,

double prob,

LinkedList<ProbabilityNode<T>> childList,

double upBnd,

double lowBnd

)

{

this.nodeObject = object;

this.probability = prob;

this.children = childList;

this.upperBound = upBnd;

this.lowerBound = lowBnd;

}

public void setNodeObject(T object)

{

this.nodeObject = object;

}

public T getNodeObject()

{

return this.nodeObject;

}

public void setProbability(double prob)

{

this.probability = prob;

}

public double getProbability()

{

return this.probability;

}

public List<ProbabilityNode<T>> getChildren()

{

return children;

}

public void setChildren(List<ProbabilityNode<T>> children)

{

this.children = children;

}

public void addChild(ProbabilityNode<T> childNode)

{

this.children.add(childNode);

}

public void addChildAt(int index, ProbabilityNode<T> child)

throws IndexOutOfBoundsException

{

this.children.add(index, child);

}

public ProbabilityNode<T> getChildAt(int index)

throws IndexOutOfBoundsException

{

return this.children.get(index);

}

public void removeChildAt(int index)

throws IndexOutOfBoundsException

{

this.children.remove(index);

}

public int getNumChildren()

{

return this.children.size();

}

public boolean hasChildren()

{

return (getNumChildren() > 0);

}

public void normalize()

{

double newProbSum = 0;

for(int i = 0; i < this.getNumChildren(); i++)

{

newProbSum += this.getChildAt(i).getProbability();

}

for(int i = 0; i < this.getNumChildren(); i++)

{

double prob = this.getChildAt(i).getProbability();

this.getChildAt(i).setProbability(prob / newProbSum);

}

}

}

public class ProbabilityTree<T>

{

ProbabilityNode<T> root;

int iterator;

AdvProbabilitySelector<T> selector;

public ProbabilityTree()

{

super();

this.iterator = 0;

this.selector = new AdvProbabilitySelector();

}

public ProbabilityTree(ProbabilityNode<T> rootNode)

{

this.root = rootNode;

this.iterator = 0;

this.selector = new AdvProbabilitySelector();

}

public ProbabilityNode<T> selectNodeChild(ProbabilityNode<T> node)

{

return selector.selectOption(node);

}

}

**Appendix B**

The implementation for the Story Generator Application is comprised of two files: StoryGenerator.java and GUIWindow.java. The code for these files is as follows.

**StoryGenerator.java**

**Appendix C**