Introductory information: This project uses the A* pathfinding algorithm from the previous project along with the environment to demonstrate the behaviors. To observe the behaviors of my code, choose one of the following in the setup() method to uncomment while keeping the others commented: decisionTree(), behaviorTree(), or treeLearning(). Keep in mind the treeLearning() method will not work if the id3Data.txt is not generated. You can generate a new id3Data.txt file by running behaviorTree() for a while. There will be a monster and a player entity on the field. The player field is controlled via mouse, while the monster will act based on whatever method is defining the tree to be active.

Decision Tree (Part 1): For this portion of the project, I decided to expose the following information to the AI for decision making: whether the target is within a range around the AI, how long the AI has been within range, and how long the AI has been out of range. Using the example code provided to us, I had the game state be defined every frame by finding the distance from between the monster and player and setting up a basic way of determining the time spent inside or outside of the range. I decided to incorporate the monster here so that the behavior tree could also use the “monster”.

Once the state has been established, the state would be sent to the tree to evaluate for a decision to be made. To follow through with a decision, Action has been introduced as a new class that keeps track of the behavior(s) and the identity (in this case, a string) it is associated with. The three actions available for this tree are “charge”, “slow”, and “wander”. The charge action that I have provided provides a path finding algorithm to the target with a high speed. The slow action is the same as charge, but with a significantly slower speed to allow the player time to escape. The wander action lets the monster go in a random direction until the state of the game allows it to take a different action. The actions are added along with decisions as nodes to a tree. The tree is created from bottom up (leaves to the root). All the leaves of the tree are actions that

are to be performed (so if the decision tree reaches a leaf, then a decision has been made).

The statement about the decision tree itself not being complicated within the project statement was fairly accurate. It was easy to understand the how to use and any issues I might encounter with the decision tree implementation provided by the sample code. The problem was trying to integrate the decision tree into the code I had already created. There was a little bit of refactoring to keep track of the action nodes of the tree. This was primarily because thew way I implemented the wander movement did not also determine a good orientation of the entity. To solve this issue, I could have made it so that the wander movement had an improved orientation determination. Instead, I chose to have the actions keep track of an ArrayList of the Movement type. This way an action can execute several types of Movement (since stuff determining orientation were also part of 'Movement'). This was also in case I wanted to pull some shenanigans with the behavior tree using a list of movements.

Description of decision tree behavior: The AI will first look at how far the player is from the monster. If it is outside the defined range, then the time within range is reset and AI will choose to wander. If it has been wandering outside the range for at least 10 seconds, the AI will then begin to charge to the location of the player. When the player gets within range of the AI, the time out of range is reset and the AI will switch to slow. When the player stays within range of the AI for 7.5 seconds, the AI will then charge at the player.

Behavior Tree (Part 2): There are 3 composite tasks that I have used in my behavior tree implementation. The first two are standard – sequence and selector. The last one was one that I implemented myself – random. The random composite task will select a random node among its children to run. I used this in conjunction with a sequence to have a behavior tree that will execute two actions at the same time. This

did complicate things later when I was working with the id3 algorithm to implement a decision tree learning algorithm, but it was something interesting I thought would work with this portion of the project. The RandomNode class I implemented will return a status of success regardless of what the child returns to allow a secondary action to complete. The purpose of this was to select a random way of orientating the monster while the game was running. In the behavior tree, I had two ways of orientating the monster – one way was to have the monster use the normal method of looking where it was heading, while the other method was to just not have anything – which made it have a slightly janky way of looking around. The determination of orientation came before the decision of movement in the implementation – Swapping the order would make it so that the added orientation would not be determined.

The overall behavior of the monster is the same as the decision tree I implemented before – it only has an extra layer that will decide a second action of the orientation method. The actual implementation of the behavior tree was not that difficult – I was basically translating from one type of tree to another and was thus able to get the same behavior from my behavior tree as decision tree in terms of movement only. The most difficult part for me was trying to determine the third type of composite node I wanted to use. As stated earlier, I ended up choosing one that would not overcomplicate things but still made things interesting.

Decision Tree Learning (Part 3): For this portion of the project, I implemented it using the id3 algorithm for simplicity. It took a little bit of time for me to figure out where in the behavior tree I wanted to handle writing data to the file and how the data should be formatted. In the end, I settled with the following format to be written after each draw loop:
“attribute1_BOOLEAN,attribute2_BOOLEAN,attribute3_BOOLEAN action1,action2”

The id3 algorithm was largely based off the code provided. Adjustments have been made to compare Strings properly and for the logarithmic in entropy calculations to be done in base 2. Whenever I recorded data for my id3 algorithm to generate a tree, I had to record for around 30 seconds at least. This was so I could demonstrate the AI the behaviors I want it to imitate. If I do not give it enough time, there may be behaviors that the AI will not pick up. Once I demonstrate all behaviors, it should be good enough for the id3 algorithm to attempt imitating, so any longer would be unnecessary.

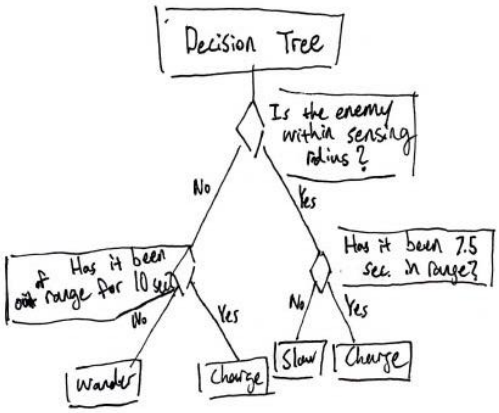
Once I record data, I have the id3 algorithm run and generate a decision tree based off my behavior tree. In order to simplify things, I ignored the secondary action recorded that determines the extra orientation and just made it part of the game specifications. Upon executing the decision tree generated by the id3 algorithm (assuming you've collected enough data demonstrating the behaviors), then the decision tree behaves in pretty much the same way as the behavior tree.

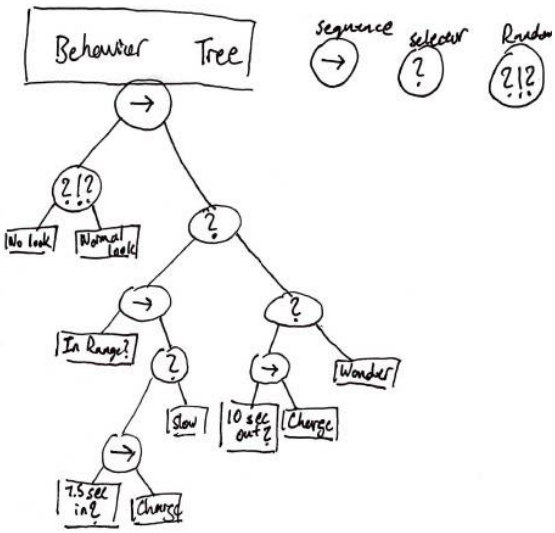
For the purposes of this section, I will be using the data stored in id3Data-COPYFORANALYSIS.txt to explain. In order to compare the behavior tree I implemented and the decision tree generated by the id3 algorithm, I had my algorithm print the tree to examine whether or not there were any wasteful representations of the nodes. If you refer to the decision tree generated by the id3 algorithm, you will notice that the tree generated is not as efficient as it can be – the tree has an extra level of height that is unnecessary. The decision tree that I implemented has a height of two while the learned tree has a height of three. If you examine the behavior tree I implemented, it only uses condition nodes three times. The learned tree, on the other hand, runs comparisons three times. The reason why I am ignoring height is because the only nodes on the tree that have a significant impact on performance would be the action nodes and the condition nodes. All the other nodes should be inconsequential. The outcomes that can be derived from all the trees (ignoring the look behavior) are 4. This is because in the learned tree, one of the actions cannot be achieved (the wander action under the 2nd child from the root).

The following are the actions and the attributes that the id3 algorithm has been given access to within the code (the same as the other trees more or less):

```
GameSpec spec = new GameSpec();

spec.addAction(new Action(wanderMove, "wander"));

spec.addAction(new Action(chargeMove, "charge"));

spec.addAction(new Action(slowMove, "slow"));



spec.addAttribute("timeInRange", "True");

spec.addAttribute("timeInRange", "False");

spec.addAttribute("timeOutRange", "True");

spec.addAttribute("timeOutRange", "False");

spec.addAttribute("withinRange", "True");

spec.addAttribute("withinRange", "False");
```

Check video in the folder to see a decision tree generated by the id3 algorithm.





Decision Tree produced by id3 observing behavior tree (movement actions only)
-NO orientation based actions taken into account