Name: Kuangyao Mai

Unity ID: kmai

CSC 484 HW4

For the first part of the assignment, the requirement is to incorporate decision-making into the steering behavior and pathfinding system that was implemented in previous homework. Based on the last assignment I did for pathfinding, I redesigned the environment and made it more appropriate for the character's decision. To prevent the stop-and-move behavior that I had observed in the previous homework and improve the movement of the boid, I started by decreasing the size of each tile. This adds a lot more nodes to the graph, but I discovered that the movement became much smoother. Then, I divided the map into five different areas: the main room, bathroom, yellow farm, red farm, and green farm. Similar to the previous assignment, each of these areas had walls separating them to mimic the effects of obstacle avoidance. Then, I created circle shapes of different colors to represent the various items that would spawn in each area. There are four items: toys that would spawn in the big room, bananas that would spawn in the yellow farm, watermelons that would spawn in the red farm, and fruit salads that would spawn in the green farm. Based on the items and their spawning locations, I defined the expected behavior of the character, which will look like this: Initially, the character would spawn in the big room with a food level of 10 and begin to wander around to find toys. Once it collides with a toy, the character's food level decreases. If the food level drops below 4, the character would randomly select a location from the three farms and wander around to look for food. The character's food level will increase when it collides with a food item. Once the food level reaches 10, the character would path follow to the bathroom and return to the main room, and the process would repeat. In addition, the character has a limited visual range and will not randomly wander into the items. Instead, if an item is within the character's line of sight, it will path follow the item. Before implementing these behaviors into a decision tree, I had to modify the character's wandering behavior. Initially, I used the wandering behavior from homework 2, but it did not produce the result that I wanted. The problem was that it did not take into account obstacle avoidance, and when I observed the character's movement, it often moved in a circular pattern. Due to this, I decided to let the character randomly select a vertex and use the pathfinding algorithm to find the path to it. This approach allows the character to avoid the walls and circular patterns but with the downside of producing a less natural wander movement. To implement the decision tree, I created a set of classes to represent the tree's nodes. There is an interface class that has a pure virtual function called "evaluate." The two derived classes, ActionNode and ConditionNode, will override this function. ActionNode represented a leaf node and contained an action that the character would perform, while ConditionNode represented an internal node and contained a condition that determined which branch of the tree to traverse. To determine which child node to traverse, the ConditionNode in the decision tree uses a function that is stored in a function type. This function type is defined when the ConditionNode object is constructed, and a custom condition function is passed as the parameter. The ActionNode also uses evaluate, but since it is a leaf node so it would just return an action. Then, in each iteration of the game loop,

the root node will traverse itself and reach an action node which will determine what the character will do. For the current decision tree implementation, there are several conditions that the tree evaluates to determine which action to take next. These conditions include checking if the character is close to an item, determining which area the character is in, and checking the character's food level to ensure that it can still perform actions. After implementing the decision tree and observing the resulting movement of the character, I was pleased to see that the character's behavior matched my expectations. The use of different colored areas, pathfinding to different locations, and food levels as a consideration provided the character with enough decision-making capabilities to navigate the environment effectively. Despite seeing the desired result, I still noticed that the current wandering behavior could make the character less likely to explore specific areas of the map. For instance, even if the character randomly chooses any location on the map, the yellow area is always where it will find food. This is because the yellow area is located in the center of the map, and as a result, the character often abandons its original path to explore other areas, such as the red or green farm, since it finds food in the yellow area first.

For the second part, the requirement is to build a behavior tree for a monster and let it move around the environment and collide with other characters. The environment for this part is the same as the previous section, with areas being separated by walls. In contrast to the previous section, the monster character in this part does not need to maintain a food level; instead, it has a stamina level that decreases over time when it tries to path follow other characters. Additionally, all items are simply referred to as food. The goal of the behavior tree is for the monster to actively search for nearby characters and path follow them if they are within the line of sight. If the monster collides with another character, both characters will reset their position. However, if the monster's stamina level falls below a threshold of 4, it will stop following the character and wander around the map searching for food. Similarly, If the monster is close to a food item, it will follow its path to replenish its stamina. To implement the behavior tree, I followed the similar data structure that I used in the first part. In addition to the ConditionNode and ActionNode, I also incorporated Selector Node and InverterNode. Since each node in the behavior tree can have more than two children, I changed the structure to use a vector to store them. Moreover, to further the functionality of the sequence node, I had to make changes to the ActionNode. Previously, it simply returned an integer representing the action to be performed. However, to enable a sequence of actions to be performed, I modified it to store a function type, which is similar to ConditionNode. This function is then called when the behavior tree traverses to the ActionNode. This allows the monster to perform actions in a specific order. For the monster's behavior tree, there is a root node which is a SelectorNode. The root node will have four SequenceNode as children, and they are evaluated in order until one of them returns a success status. The first sequence node determines if the monster will attack and reset the position. The second sequence node checks if the monster is close to a character and sees if it can path follow to it. The third sequence node checks if the monster is close to a food item and sees if it can path follow to it. Finally, if all of the above conditions return false, the monster will just wander around the map. To note, the picture of the behavior tree I drew is not exactly the same as the one I actually implemented. This is because, during the implementation process, I had to make some adjustments to the order of sequence nodes and prioritize certain conditions, such as

checking the monster's stamina and adding an extra node to check nearby items for some of the SequenceNode. Upon observing the behaviors, the monster was able to perform the correct actions based on the state it was in. Overall, I found the implementation difficulty for the behavior tree to be comparable to the decision tree. However, I found it easier to author the behavior tree, possibly due to the reduced complexity of the monster's behavior compared to the first part.

The third part of the assignment is to implement decision-tree learning for the behavior tree. The first step I did was to identify the important attributes that can help determine the monster's action. Upon reviewing the behavior tree, I realized they essentially correspond to the conditions evaluated in the ConditionNode. Therefore, I selected four attributes: 1). check if characters are nearby, 2). check if the monster collides with a character, 3). check if the monster is at low stamina, 4). check if it is near a food item. The struct was then converted into a vector and outputted to a file where attributes are boolean, and action could just be represented as an integer. After the data preparation, I began implementing the ID3 algorithm for decision tree learning. The primary reference for the pseudocode was the lecture slide from CSC 411. The first step in the algorithm is to select the best attribute to split the dataset on (bestCriteria), which involves the calculation of both entropy and conditional entropy. The implementation ideas for these two functions were partly inspired by the pseudocode found on this website: https://www.cs.swarthmore.edu/~meeden/cs63/f05/id3.html. The first function considers the frequency of each action in the dataset to calculate entropy. The second one calculates the average entropy of the dataset after it has been split based on the values of a specific attribute. In this case, it would just split into 0 or 1 because attributes are boolean. Then bestCriteria finds the best attribute to split the dataset on, which is based on the information gained. It iterates through each attribute, calculates the conditional entropy for that attribute, and then subtracts from the base entropy. The attribute with the highest information gain is selected as the best attribute. After finding the first best attribute, it will split the dataset based on that and create a decision node with the best attribute. The function then recursively builds the decision tree for each subset of the split dataset and adds the resulting subtrees as children of the decision node. To support this, I had to edit the DecisionNode. The DecisionNode class now has a vector of child nodes, which are the tree branches that represent the chosen attribute's possible values. The evaluate function in the DecisionNode class now considers the input attributes and uses the chosen attribute index to determine which child node to evaluate further. So in every iteration of the game loop, I would have to use the MonsterState struct again to record the current status, pass it into the root node, and let it return an action. Upon observation, the decision tree implementation provided the same movement and behavior for the monster compared to the previous behavior tree approach when the same environment and action functions were used (slight modification on the increase and decrease of stamina). This suggests that the decision tree was successful in capturing the decision-making logic present in the behavior tree.

Demo for each part:

Part1: https://youtu.be/6TpTk88vNWQ

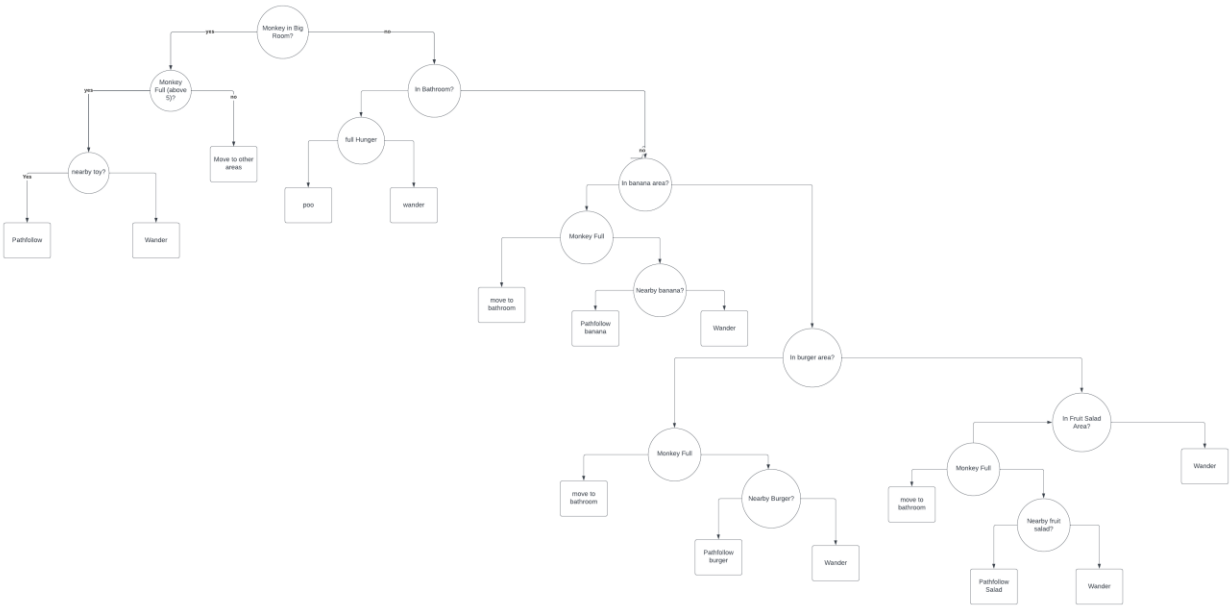
Part2: https://youtu.be/f5xFBIVKoVk

Part3: https://youtu.be/UhnUobcTBxM

ID3 Pseudocode

```
Algorithm BuildDecisionTree(samples)
   if samples are all the same class
      return Node(class)
   else
      C ← bestCriteria(samples)
      cSamples ← splitExamples(samples, C)
      node = Node(C)
      for childSample in cSamples
          node.branch.add(BuildDecisionTree(childSample))
      return node
```

Decision Tree:



Behavior Tree: