CSC 584 : Homework III writeup

Anubhab Majumdar

1 Environment

The environment used for all the three subproblems of the assignment is shown in Figure 1. The environment has three parts:

1. **Room** - The top-left corner of the environment is the room. Its boundaries are demarcated with walls. It has two points of entry (doors). The length and width of the room is 190 pixels each (excluding the boundary width).
2. **Outside Room** - The area immediately below the room is considered outside-room. Its length is exactly same as the room, 190 pixels, but its width extends till the end of environment. The boundary on the right is marked with red line.
3. **Corridor** - Any area of the environment that is neither room or outside-room is corridor. This is the largest area of the environment. Its left boundary is marked with the red line.

To be clear, the red lines don't appear in the actual environment. For the assignment, the environment looks like Figure 2.

2 Characters

Three characters are used for the assignment:

1. Lego Superman (Figure 3) [1]
2. Red Monster (Figure 4) [2]
3. Blue Monster (Figure 5) [3]

3 First Steps

The pathfinding and pathfollowing algorithms coded for assignment 2 works perfectly for the new environment, proving that they are generic enough for any new environment.

4 Decision Trees

The decision tree crafted for the assignment is shown in Figure 6. Check the decision tree in action here - https://youtu.be/l7y9ZwMPyEI.

4.1 Description

4.1.1 Internal Nodes

1. **Inside Room** - This node checks whether the character is in the area demarcated as "Room" in Figure 1. This is done by checking whether the position attribute of the steering variable representing the character (both the X and Y coordinate) is between [0, 190]. It returns true if the character is inside the room; false otherwise.

2. **Max Rotation** - This node checks if the current rotation value is greater than the maximum rotation value set for the character. This is done by checking the *rotation* and *maxRot* attribute of steering object associated with the character. It returns true if the rotation value is greater than maxRot; false otherwise.

3. **Outside Room** - This node checks whether the character is in the area demarcated as "Outside Room" in Figure 1. This is done by checking the X coordinate of the position attribute of the steering variable representing the character:

   - X coordinate should be less than $width/3$ (where width means the width of the environment)

   It returns true if the character is inside the "Outside Room" area; false otherwise.

4. **Near Boundary** - Boundary is defined as a narrow area all along the four edges of the environment, 50 pixels in width. If the character in inside this area, then it is considered to be near the boundary of the environment. It is checked using the position attribute of the steering variable representing the character. It returns true if the character is in the vicinity of the boundary; false otherwise.

4.1.2 Leaf Nodes

1. **Pathfind** - This leaf finds a path for the character to traverse to a particular location from its current position. This leaf is re-purposed into three different leaf actions:

   - Pathfind (500, 300) - Pathfind from your current location to coordinate (500, 300), which is in the corridor and is the starting point for the character.

- Pathfind (50, 50) - Pathfind from your current location to coordinate (50, 50), which is inside the room.
- Pathfind ($width/2$, $height/2$) - Pathfind from your current location to the center of the environment.

The leaf sets the path in an instance variable named "path".

2. **Stop & Rotate** - This leaf sets the current velocity and acceleration of the character as zero. Then an angular acceleration of 0.001 is set to the $angularAccelaration$ attribute of the character's steering variable. This makes the character rotate till it reaches maximum rotation.

3. **Random Pathfind** - This leaf first selects a random coordinate within the boundaries of the environment. Then the pathfinding algorithm is called to find a path to that point from the current position. The leaf sets the path in an instance variable named "path". If a path doesn't exist, "path" variable is set as an empty ArrayList.

4.2 Implementation

The decision tree is coded following the decorator design pattern discussed in class and taught in CSC 517 (OOP Design and Development). The following interface and classes are used for constructing the decision tree:

1. **NodeInterface** - This is the interface which the classes implement. It has one method - $boolean\ evaluate(SteeringClass\ steeringClass)$. It returns a boolean value on completion.

2. **InternalNode** - This is the *base* class required to implement the decorator pattern (Banas, 2012) [4]. It implements NodeInterface and always returns false in evaluate method.

3. **InternalNodeInterface** - Wrapper class for the internal nodes and leaf nodes. It implements NodeInterface and has three instance variables:

   - NodeInterface nodeInterface - This object contains the parameter checking logic for the internal nodes and actions for leaf nodes. The logic is coded in evaluate method of the object.
   - NodeInterface left - Points to the left node, which is another NodeInterface object.
   - NodeInterface right - Points to the right node, which is another NodeInterface object.

   InternalNodeInterface calls the evaluate() method of the nodeInterface instance variable inside its own evaluate() method.

4. **Specialized classes** - Various classes were defined to implement the logic for the internal and leaf nodes. The classes are:

- RandomPathFollowingLeaf - Implements “Random Pathfind”.
- PathFollowingLeaf - Implements “Pathfind”.
- highSpeedRotLeaf - Implements “Stop & Rotate”.
- NearWallCheckNode - Implements “Near Boundary”.
- OutsideRoomCheck - Implements “Outside Room”.
- MaxRotationCheckNode - Implements “Max Rotation”.
- InsideRoomCheck - Implements “Inside Room”.

The decision tree is then constructed using the code shown in Figure 7.

Once the tree is constructed, it can be traversed simply. We start with the root node and continue till we reach a leaf node. For each node, we call the evaluate() method. If the method returns true, we move on to the left node associated with the current node; else we move to the right node. The code is shown in Figure 8.

Pathfollowing for Superman is called in the draw() method of the driver program, using the “path” variable sent by the traverseDT() method, not in the leaf node of decision tree.

5 Behavior Trees

The behavior tree constructed to guide the monster is shown in Figure 9. Watch the behavior tree in action here - https://youtu.be/plYTG3KJh9w.

5.1 Composites

Three different composites are used:

1. Selector
2. Sequence
3. Random Selector

5.2 Checks & Actions

1. **Outside Room** - This checks whether Superman is inside the room or not and returns the negation of what it determines. It returns false if Superman is inside the “Room”; true otherwise.

2. **Pathfind Superman** - The character in our assignment is a Lego Superman as shown in Figure 3. The monster tries to eat (catch up) Superman. This action finds a path to the current location of Superman from the current location of the monster. The steering variable representing Superman is available to the behavior tree, so that this method can find the path successfully. The action sets the path in an instance variable named

"path". If a path doesn't exist, "path" variable is set as an empty ArrayList. Returns true if it can find path to Superman's current position; else returns false.

3. **Stop & Rotate** - Same as that of the decision tree. This leaf sets the current velocity and acceleration of the monster as zero. Then an angular acceleration of 0.001 is set to the $angularAccelaration$ attribute of the monster's steering variable. This makes the monster rotate till it reaches maximum rotation. Returns true if it can set the required values of the steering object; else returns false.

4. **Change Monster** - This leaf changes the *avatar* of the monster. The two possible $avatars$ are shown in Figure 4 and 5. Returns true after setting a new image to the $CustomShape$ object representing the monster; else returns false.

5.3 Description

The behavior tree shown in Figure 9 encapsulates the following behavior for the monster:

1. Check if Superman is outside the area marked "Room" in Figure 1.

2. If Superman is outside "Room", find a path to Superman's current location.

3. If Superman is inside "Room", choose randomly between actions Stop & Rotate and Change Monster.

The monster behaved exactly as I expected. Good monster!

5.4 Implementation

The decorator pattern used for constructing decision tree is re-used for making the behavior tree. Three classes are created to represent the composites:

1. **Selector** - This class extends InternalNodeInterface. In the evaluate() method of Selector class, we call the evaluate() method of left and right child and return the logical OR of both the results.

2. **Sequence** - This class extends InternalNodeInterface. In the evaluate() method of Sequence class, we call the evaluate() method of both its left and right child and return the logical AND of both the results.

3. **Random Selector** - This class extends InternalNodeInterface. In evaluate() method, we first generate a random number between [0, 1]. If the random number is greater than 0.5, we return
$(left.evaluate(steeringClass)||right.evaluate(steeringClass))$;
else we return
$(right.evaluate(steeringClass)||left.evaluate(steeringClass))$.

Few implementation details to make things look cool:

- The maximum speed of monster is set to 0.99 as compared to 1.0 of Superman. This draws out the chase between monster and Superman and we can test the behavior tree better.
- For similar reason, the behavior tree is traversed every 700 millisecond while the decision tree controlling Superman is traversed every 300 millisecond. This means the monster is always following an old location of Superman. This delays the monster in catching up to Superman.
- Pathfollowing for both monster and Superman is called in the driver program, not in the leaf of decision tree or action nodes of behavior tree.
- When the monster "eats" the character, we reset the position of Superman and the monster. This is handled in the draw() method of driver class. The reset() method checks the distance between Superman and monster (using their respective Steering objects), and if it is less than 30 pixels, then their positions are set at their respective start positions.

The code to create the behavior tree is shown in Figure 10.

6 Decision Tree Learning

6.1 Data Collection

For learning the behavior of the monster, we need to record data about the monster and Superman. The following data is collected:

1. character_posX - The X-coordinate of Superman's position attribute.
2. character_posY - The Y-coordinate of Superman's position attribute.
3. character_velX - The X-component of Superman's velocity attribute.
4. character_velY - The Y-component of Superman's velocity attribute.
5. character_accX - The X-component of Superman's acceleration attribute.
6. character_accY - The Y-component of Superman's acceleration attribute.
7. character_orientation - The orientation attribute of Superman's steering variable.
8. character_rotation - The rotation attribute of Superman's steering variable.
9. character_angAcc - The angular acceleration attribute of Superman's steering variable.
10. monster_posX - The X-coordinate of monster's position attribute.

11. monster_posY - The Y-coordinate of monster's position attribute.
12. monster_velX - The X-component of monster's velocity attribute.
13. monster_velY - The Y-component of monster's velocity attribute.
14. monster_accX - The X-component of monster's acceleration attribute.
15. monster_accY - The Y-component of monster's acceleration attribute.
16. monster_orientation - The orientation attribute of monster's steering variable.
17. monster_rotation - The rotation attribute of monster's steering variable.
18. monster_angAcc - The angular acceleration attribute of monster's steering variable.
19. monster_image - The current *avatar* of the monster.
20. action - The current action of the monster. The possible values are Pathfollow, Rotation, ChangeMonster and Reset.

Initially I recorded 10,000 instances, but the decision tree returned by the learning algorithm was less precise than I anticipated. Then I collected data for 50,000 instances and the decision tree returned by the algorithm exactly encapsulated the behavior tree it was mimicking. The data can be seen in $trainData2.csv$.

6.2 Feature Extraction

The data collected previously can be termed as raw data. However, for superior performance of any learning algorithm, we need to extract "features" from raw data. The features are supposed to be a better and more accurate representation of the data and filters any useless information in data. This make it easier for the learning algorithm to decipher patterns hidden in the data.

For the current assignment, I have extracted two features from raw data:

1. **Distance Between Objects** - This attribute records the euclidean distance (in pixels) between Superman and the monster. It does so by using character_posX, character_posY, monster_posX and monster_posY. If the distance is less than 30.0, we categorize it as "close", else "far". This changes the continuous variable to discrete. This is necessary for learning.
2. **In Room** - This records whether Superman is inside the "Room" or not. It checks the character_posX and character_posY variable to determine this. The feature can be either "true" or "false".

The actions corresponding to all the 50,000 records are kept exact from the raw data. The features and actions can be found in $features.csv$.

6.3 Decision Tree Learning Algorithm

The decision tree learning algorithm is coded in the file *DTLearning.java*. It is coded in the same way as the algorithm was discussed in the class. The algorithm is recursive in nature with the base cases being:

1. Only one action for all the instances dataset.
2. No more features left to explore.

If neither of the base cases are met, we choose a feature to split the current dataset. The metric used to choose this feature is information gain. Once split, we filter our dataset into two sets based on the value of the just selected feature. The algorithm is then called recursively for both the datasets.

The algorithm outputs the tree in a text format. It is shown in Figure 11.

For leaf nodes, the algorithm either outputs one action or the probabilities of the multiple actions that has remained. While constructing the tree, I do the following:

- Remove all actions whose probability is less than 10%.
- From the remaining actions, choose top two actions with highest probabilities. When the particular leaf is visited in the decision tree, select the actions weighted by the probabilities.

The output generated by the learning algorithm is shown in Figure 11. The tree constructed based on it is shown in Figure 12. One interesting thing to observe is that for one leaf node the actions are almost evenly distributed (ChangeMonster - 44.17% and StopAndRotate - 48.50%). This is because we used a random selector in our behavior tree. This is a clear indication that our features and learning algorithm is working as expected. Watch the learned decision tree in action here - https://youtu.be/YxytHHU-fl8.

6.4 Comparison

If we look closely at the generated tree (Figure 12) and the original behavior tree (Figure 9), we will see the former is pretty much the exact translation of the later. What I mean is, if someone decided to mimic the monster's behavior through a decision tree instead of the behavior tree, the output would look pretty much like the tree generated by the code. Checking the tree by hand confirms this. The decision tree should drive the monster same as the behavior tree.

Running the behavior tree and the decision tree side-by-side, I noticed no visual difference. The monsters' behavior when Superman is in room and outside the room is similar to that of the behavior tree.

For some concrete numbers, I counted the number of times the monster successfully eats Superman. Table 1 shows the result for both the behavior tree and learned decision tree. The count is quite comparable given the fact that Superman mostly wanders to random point and the monster is hamstrung by slow speed and rate of decision making (to give an edge to Superman).

Iterations	Behavior Tree	Learned Decision Tree
10,000	11	17
15,000	22	25
20,000	31	32

Table 1: Performance comparison of behavior tree and learned decision tree

7 Conclusion

The major takeaway from the assignment are:

- **Decision tress and behavior trees** - How to control a character using behavior and decision tree.
- **Learning algorithms** - Decision tree learning algorithm implementation.
- **Data collection and feature selection**
- **Hierarchical software development** - Build on top of previous assignments.
- **Decorator design pattern**

Appendix A Environment

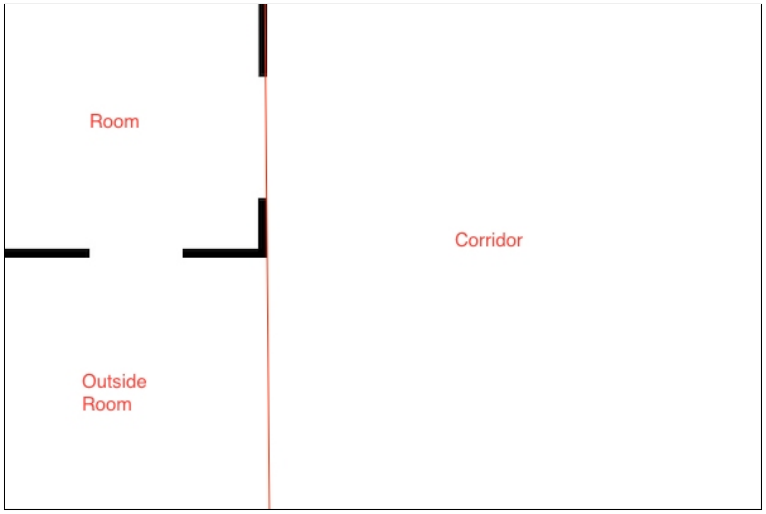

Figure 1: The components of the environment

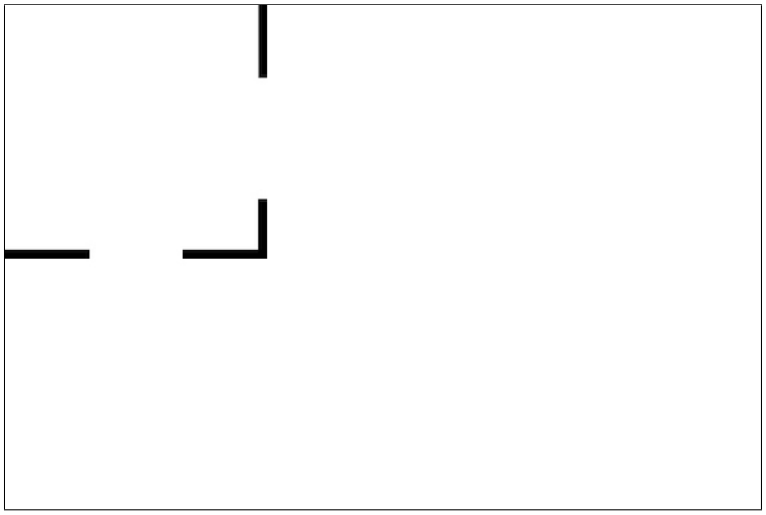

Figure 2: The environment used for the assignment

Appendix B Character

Figure 3: Lego superman (Image courtesy - [1])

Figure 4: Red monster (Image courtesy - [2])

Figure 5: Blue monster (Image courtesy - [3])

Appendix C Decision Tree

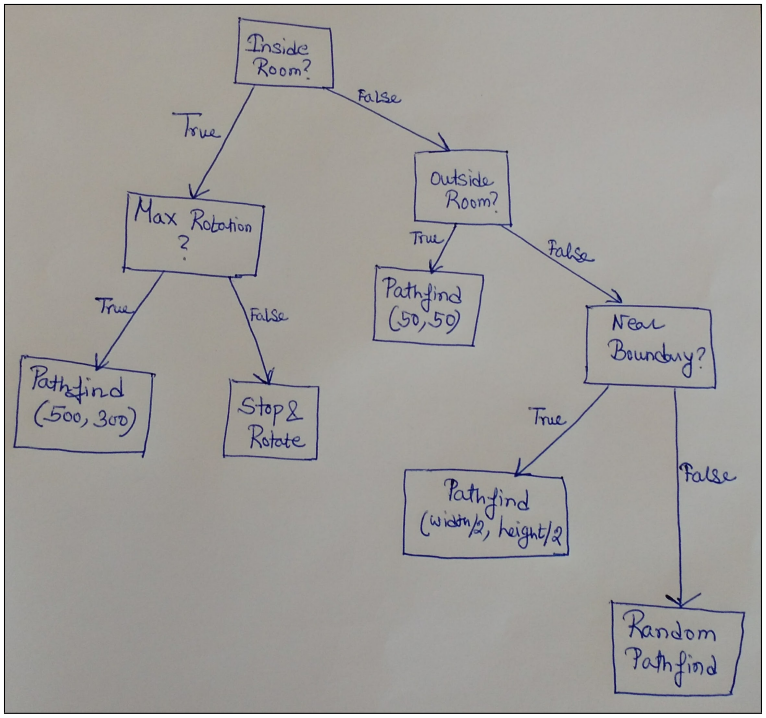

Figure 6: Decision Tree

```
public NodeInterface makeTree()
{
    // get leaves
    NodeInterface leaf1 = new RandomPathFollowingLeaf(new InternalNode(), left: null, right: null);

    NodeInterface leaf2 = new PathFollowingLeaf(new InternalNode(), left: null, right: null);
    ((PathFollowingLeaf)leaf2).setX(S1_x);
    ((PathFollowingLeaf)leaf2).setY(S1_y);

    NodeInterface leaf3 = new PathFollowingLeaf(new InternalNode(), left: null, right: null);
    ((PathFollowingLeaf)leaf3).setX(R1_x);
    ((PathFollowingLeaf)leaf3).setY(R1_y);

    NodeInterface leaf4 = new highSpeedRotLeaf(new InternalNode(), left: null, right: null);

    NodeInterface leaf5 = new PathFollowingLeaf(new InternalNode(), left: null, right: null);
    ((PathFollowingLeaf)leaf5).setX(500);
    ((PathFollowingLeaf)leaf5).setY(300);


    // get internal nodes
    NodeInterface internalNode1 = new NearWallCheckNode(new InternalNode(), leaf2, leaf1);
    NodeInterface internalNode2 = new OutsideRoomCheck(new InternalNode(), leaf3, internalNode1);
    NodeInterface internalNode3 = new MaxRotationCheckNode(new InternalNode(), leaf5, leaf4);
    NodeInterface root = new InsideRoomCheck(new InternalNode(), internalNode3, internalNode2);

    return root;
}
```

Figure 7: Decision Tree construction code

```
NodeInterface temp = root;
while (temp != null)
{
    flag = temp.evaluate(steeringClass);
    if (flag)
        temp = ((InternalNodeInterface)temp).getLeft();
    else
        temp = ((InternalNodeInterface)temp).getRight();
}
```

Figure 8: Decision Tree traversal

Appendix D Behavior Tree

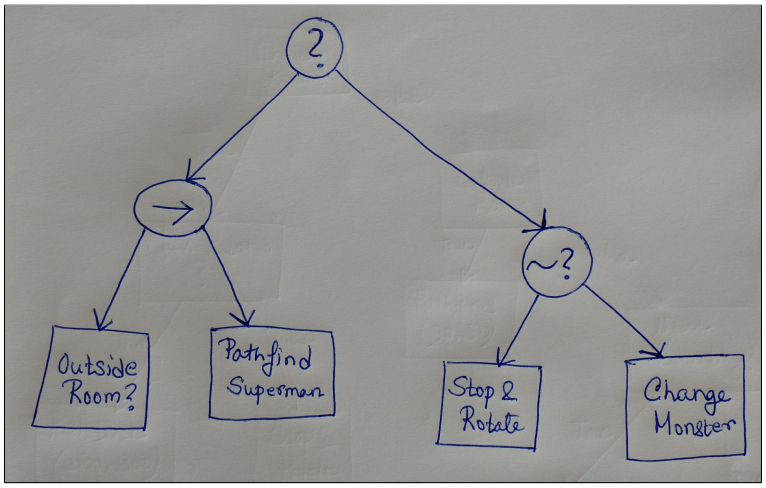

Figure 9: Behavior Tree

```
public NodeInterface makeTree()
{
    // get leaves
    NodeInterface outsideRoom = new OutsideRoomCheck(new InternalNode(), left: null, right: null);

    NodeInterface supermanFollowing = new SupermanFollowingLeaf(new InternalNode(), left: null, right: null);

    NodeInterface stopAndRotate = new highSpeedRotLeaf(new InternalNode(), left: null, right: null);

    //NodeInterface caught = new CaughtCheck(new InternalNode(), null, null);

    //NodeInterface materialize = new Materialize(new InternalNode(), null, null);

    NodeInterface changeMonster = new ChangeMonster(new InternalNode(), left: null, right: null);

    // get internal nodes
    //NodeInterface internalNode1 = new Sequence(new InternalNode(), caught, materialize);
    NodeInterface internalNode2 = new Sequence(new InternalNode(), outsideRoom, supermanFollowing);
    NodeInterface internalNode3 = new RandomSelector(new InternalNode(), stopAndRotate, changeMonster);
    NodeInterface internalNode4 = new Selector(new InternalNode(), internalNode2, internalNode3);
    //NodeInterface root = new Selector(new InternalNode(), internalNode1, internalNode4);

    return internalNode4;
}
```

Figure 10: Behavior Tree construction code

Appendix E Decision Tree Learning

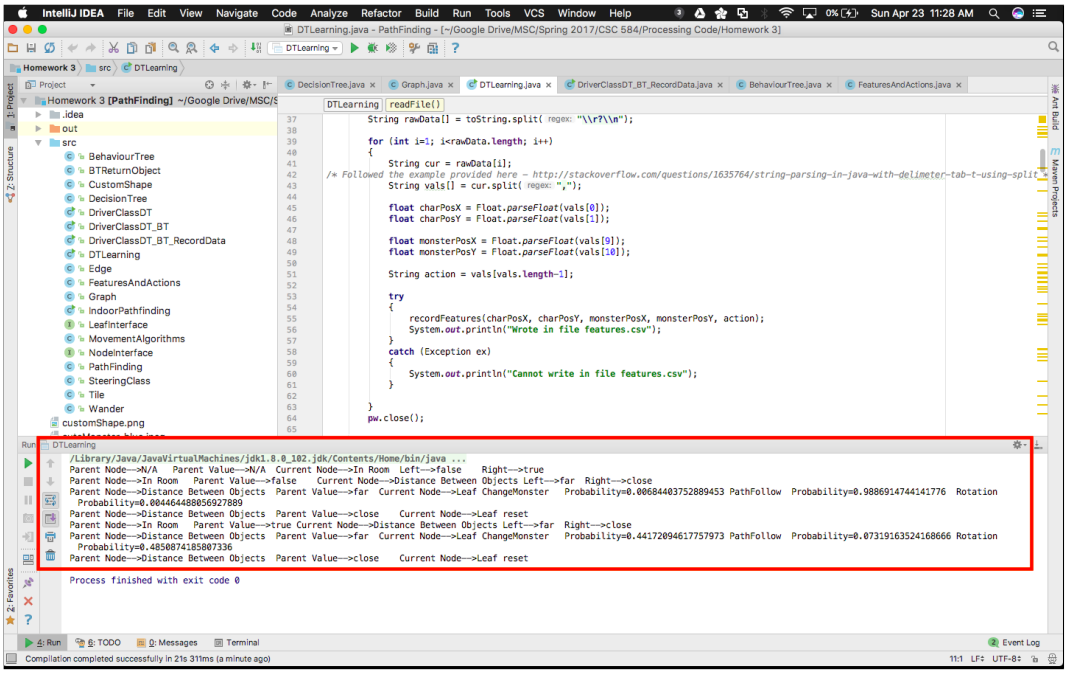

Figure 11: Decision Tree Learning Algorithm Output

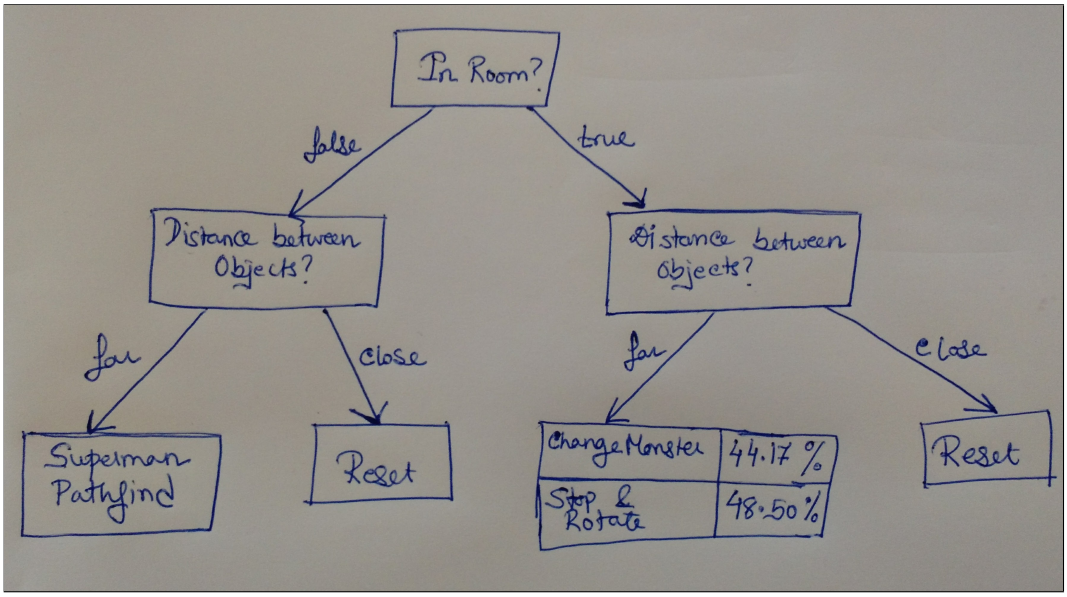

Figure 12: Decision Tree generated

References

[1] "Lego superman." Accessed on : 04/17/2017; https://lc-www-live-s.legocdn.com/r/www/r/catalogs/-/media/catalogs/characters/dc/fullsize/2014/76040-superman_360w_2x.png?l.r2=-2079447138.

[2] "Red monster." Accessed on : 04/17/2017; http://fsb.zedge.net/scale.php?img=MS83LzIvNS8xLTEwMTM5MDgwLTE3MjUxMzMuanBn&ctype=1&v=4&q=71&xs=300&ys=266&sig=f646cac1f807dad3d23cb9133120724f950c0178.

[3] "Blue monster." Accessed on : 04/17/2017; https://s-media-cache-ak0.pinimg.com/736x/b2/53/36/b25336fe10b2e2fbb709889f9b3e74c1.jpg.

[4] D. Banas, "Decorator design pattern," Sep 2012. Accessed on : 04/17/2017; https://www.youtube.com/watch?v=j40kRwSm4VE.

[5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[6] I. Millington and J. Funge, *Artificial Intelligence for Games, Second Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd ed., 2009.