CSC 584: Building Game AI
Homework 4

Name: Jayesh Bhagyesh Gajbhar(jgajbha)

First Steps:

For this assignment, I have used the environment created in the "putting it all together" part of Homework 3. Although the environment is mostly the same, there is a few noticeable changes.

Firstly, diagonal movements have been allowed in this implementation, as compared to the last homework, and the path finding algorithms have been appropriately accommodated to achieve the same.

Secondly, obstacles in the diagonal direction have been removed so that the pathfinding algorithm doesn't misinterpret the environment and produce wrong paths.

Decision Trees:

For building the decision tree, I have implemented the following Tree Node structure(inspired from decision trees in a market basket analysis):

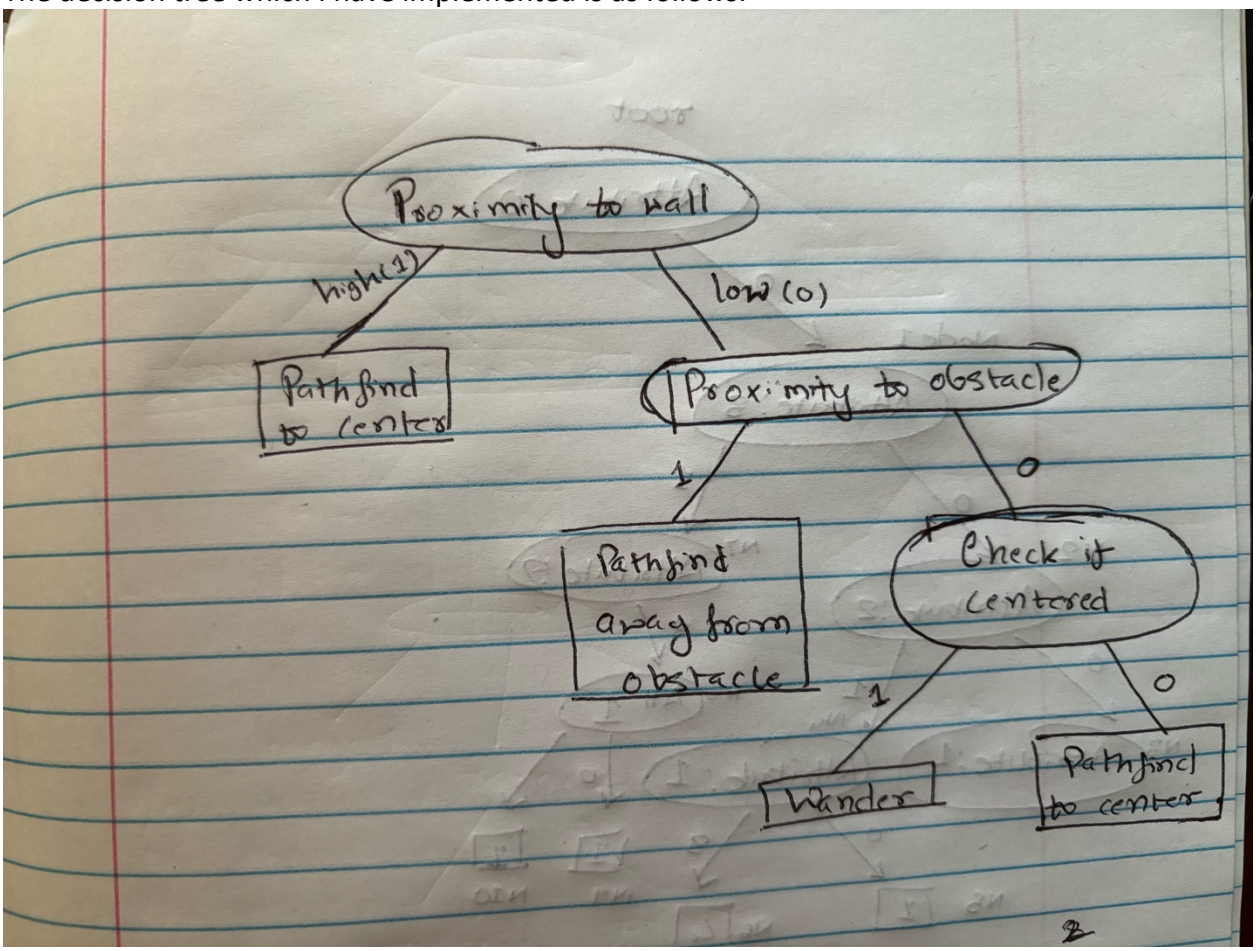
```
struct TreeNode
{
    int attribute;
    std::unordered_map<int, TreeNode *> branches;
    int classification;
};
```

This implementation allows flexibility to have more than two child nodes for each node as compared to my first implementation which is as follows:

```
struct TreeNode
{
    int featureIndex; // Index of the feature to split on
    double threshold; // Threshold for the split
    std::unique_ptr<TreeNode> left;   // Pointer to the left child node
    std::unique_ptr<TreeNode> right;  // Pointer to the right child node
    int label;        // Label or action associated with this node

    // Constructor to initialize the node
    TreeNode(int featureIndex, double threshold, std::unique_ptr<TreeNode> left,
std::unique_ptr<TreeNode> right, int label = -1)
```

```
        : featureIndex(featureIndex), threshold(threshold), left(std::move(left)),
right(std::move(right)), label(label) {}
};
```

The decision tree which I have implemented is as follows:



The attributes it checks for are as follows:

Proximity to wall(edges of the window) : 1 if the character is close to a wall and 0 otherwise.
Proximity to obstacle : 1 if the character is close to an obstacle, 0 otherwise.
Check if centered: 1 if the character has reached the center, 0 otherwise.

The decision tree can output 3 decisions, depending on the input, which are encoded as follows:

```
enum ActionType
{
    WANDER,
    PATHFIND_TO_CENTER, // to center of screen
    PATHFIND_AWAY_FROM_OBSTACLE
};
```

PATHFIND_AWAY_FROM_OBSTACLE does exactly what it says and finds a target location farther away from an obstacle in an opposite direction to the obstacle.

The attributes are extracted from the game environment using the following piece of code and are stored in a vector called features, which is passed to the decision tree every frame.

```
std::vector<int> extractFeatures()
{
    std::vector<int> features;
    float dw = calculateDistanceFromEdges(character.position, windowSize);
    if (dw < 20.f)
        features.push_back(1);
    else
        features.push_back(0);

    bool po = checkObstacleProximity(character.position, obstaclePositions, 20.f);
    if (po)
        features.push_back(1);
    else
        features.push_back(0);

    float dc = calculateDistance(character.position, sf::Vector2f(250.f, 250.f));
    if (dc <= 0.f)
        features.push_back(1);
    else
        features.push_back(0);

    return features;
}
```

Therefore, the structure of the features vector is as follows:

features = {Proximity_to_wall, Proximity_to_obstacle, Check_if_centered}

Please note that although the decision tree is called at every frame, the actions are performed at intervals of 1 seconds, to not overwhelm the CPU by calling the pathfinding algorithm in every frame. This approach will be consistent throughout the next two parts of the assignment as well.

This decision tree may force the character to be stuck in one position for a while and juggle between pathfinding to the center and pathfinding away from an obstacle. Eventually, when the character does reach the center and tries to wander, it gets stuck pathfinding back to the center, as the tolerance assigned to being at the center is very low.
(Please note that the center point as been hardcoded in all the subsequent parts of the assignment)

Admittedly, this implementation is not flawless, but the decision tree works as expected.

The images of the implementation have been attached to the appendix(Part1).

Behavior Trees:

(The following implementation of Behavior Trees has been inspired from the following video: https://www.youtube.com/watch?v=F-3nxJ2ANXg)

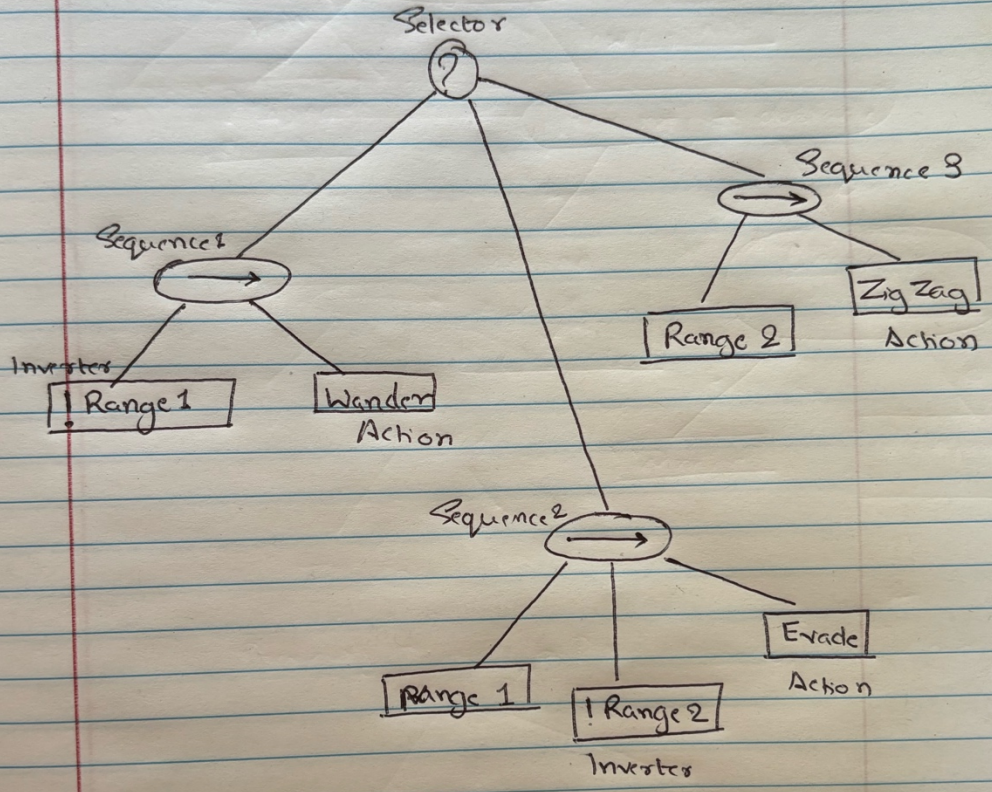
I have implemented three types of nodes, namely Sequence, Selector and Inverter (Part2/BehaviorTreeNode.hpp – lines 24 to 157), based on the following base class:

```
class Node
{
protected:
    NodeState _nodeState;

public:
    NodeState nodeState() const
    {
        return _nodeState;
    }

    virtual NodeState Evaluate() = 0;

    virtual ~Node() = default;
};
```

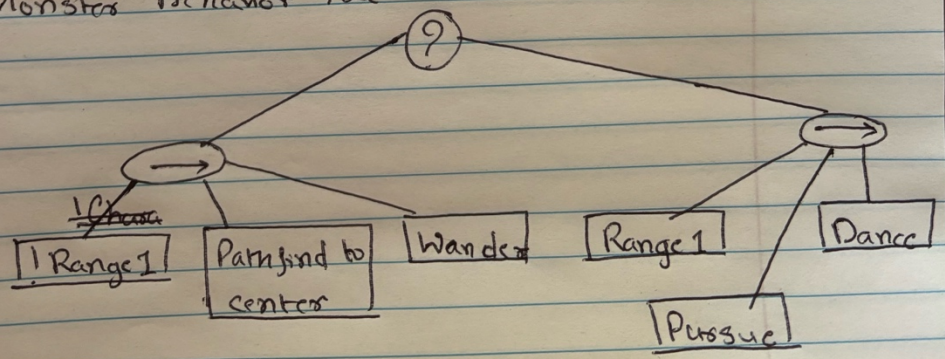
In this part of the assignment, both character and monster use the behavior trees to make decisions.

The trees are as follows:

Character Behavior Tree:



Monster Behavior Tree:



The actions are encoded as follows:

```
enum ActionType
{
    WANDER,
    PATHFIND_TO_CENTER, // to center of screen
    ZIGZAG,
    EVADE,
    PURSUE,
    DANCE
};
```

It is worth noting that PATHFIND_TO_CENTER is a misnomer as it doesn't pathfind to center, but rather, it finds a path to a point within 30 units of the character. Also, when this action is active, the character vibrates rapidly, to represent a rogue monster.

Range1 checks whether the characters are within 50 to 10 units of each other and Range2 checks whether the characters are within 10 units of each other.

The WANDER action selects a random point on the grid and pathfinds to it.
The ZIGZAG action executes a zigzag motion to confuse the monster's PURSUE action.
The EVADE action calculates the monster's future position and pathfinds to a point away from it. Similarly, the PURSUE action calculates the character's future position and pathfinds to it in order to catch it.
Both the PURSUE and EVADE actions predict 5 seconds into the future.
Finally, the DANCE action executes a dance sequence (left-right-left-right-up-down-up-down (the controls to activate cheats, almost)).

The zigzag pattern is a hit or miss as it may or may not execute before the monster "eats" the character. Perhaps, a better calibration of Range1 and Range2 might help in this case.

The problem of calibrating Range1 and Range2 affects the next part of the assignment as well.

The behavior tree inputs are not encapsulated in a manner that has been done for the decision tree, instead, the behavior tree takes Steering Data from both the character and monster as input.

Finally, the screenshots of this part of the assignment have been attached to the appendix(Part2).

Decision Tree Learning:

i. Data collection:

The following data was collected from 4-5 runs of Part2, with each run having a different starting point for both the character and monster. The data collection script runs in every frame.

float distanceToCharacter – distance of monster from character
int characterAction – action taken by the character
int monsterAction – action taken by the monster
bool isDanceComplete – to check if dance is complete or not
float distanceToCenter – distance of the monster from the center of the screen
bool obstacleProximity – to check if the monster is close to an obstacle

The data was collected in a csv file, an excerpt from which is depicted below:

39.3806,0,0,*0,*2.96295,*0*
39.4119,0,0,*0,*2.85998,*0*
39.4438,0,0,*0,*2.75524,*0*
39.4764,0,0,*0,*2.64889,*0*
39.5097,0,0,*0,*2.54099,*0*
39.5435,0,0,*0,*2.43165,*0*
39.5766,0,0,*0,*2.32577,*0*
39.6117,0,0,*0,*2.21376,*0*
39.6474,0,0,*0,*2.10058,*0*

The final dataset has about 100,000 datapoints

ii. Data preprocessing:

The data was then preprocessed to be nominal, using a python script (learning/preprocess4.py)

If the distanceToCharacter >50, distanceToCharacter = 0
If the 10<= distanceToCharacter <=50, distanceToCharacter = 1
If the distanceToCharacter <10, distanceToCharacter = 2

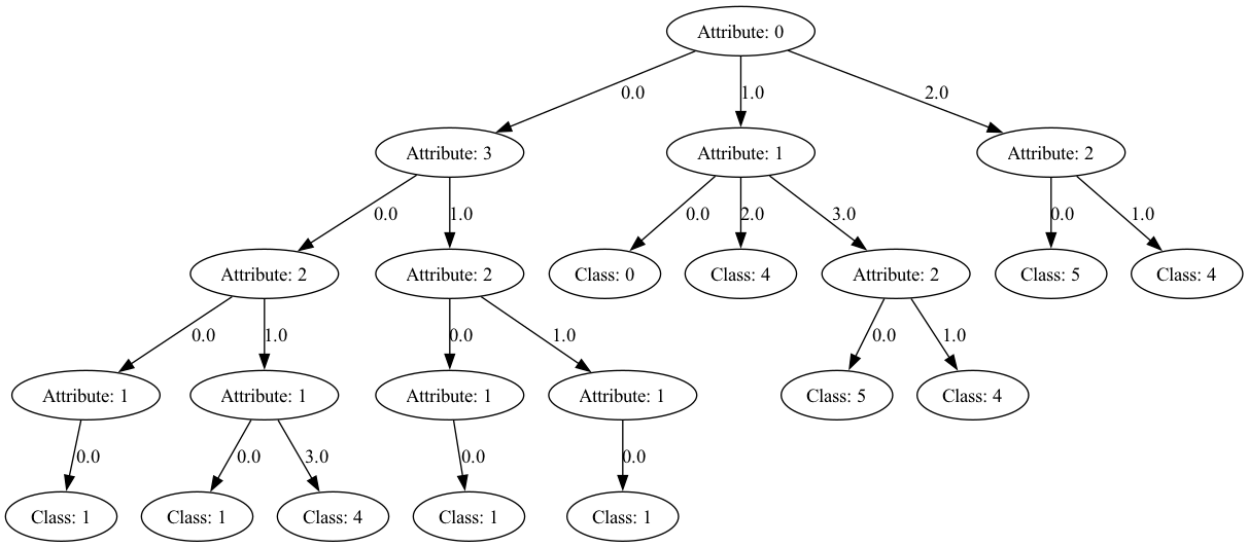
The distanceToCenter data was dropped, as it didn't provide much information gain.

Finally, the csv file was restructured to resemble to following structure:
distance_to_character, character_action, is_dance_complete, obstacle_proximity, monster_action

iii. Decision Tree Learning:

Finally, the python script named learn.py (in the learn folder) was used to create a decision tree.

Python's graphviz library was used to visualize the resultant tree, which is as follows:



The attributes are as follows:
0: distance_to_character
1: character_action
2: is_dance_complete
3: obstacle_proximity
output_class: monster_action

The actions for both monster and character are encoded as follows:

```
enum ActionType
{
   WANDER,
   PATHFIND_TO_CENTER,
   ZIGZAG,
   EVADE,
   PURSUE,
   DANCE
};
```

iv. Performance evaluation:

The data was collected by running the decision tree in the same format as in the first step and preprocessed in the same manner as the second step.

To evaluate the performance, I used the sklearn.metrics library in python.

Since the data collection and preprocessing was done in the same manner, the two datasets could be compared directly.

The results were as follows:

Accuracy: 0.4106793639996599
Precision: 0.38924742809938556
Recall: 0.4106793639996599
F1-score: 0.2836202886209185

The most notable metric here is the accuracy, it means that the decision tree replicates the behavior of the behavior tree with an accuracy of 41%.

This low accuracy, in my opinion, is due to the calibration of the distance_to_character data, which builds on top of the Range1 and Range2 nodes in the implementation of Part2. The monster starts dancing even when the character has not been caught, but is very close, in some cases and fails to execute the pursue action almost consistently.

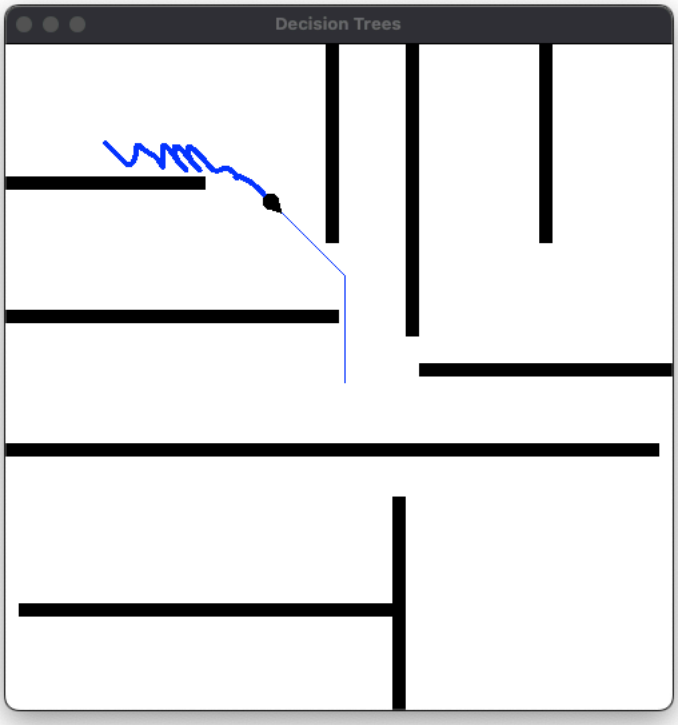
The accuracy could also be improved by increasing the size of the dataset as this dataset seems insufficient for learning the behavior tree model.

Even with this low accuracy, the monster successfully pathfinds to a point close to the character, and manages to catch the character by “luck”.

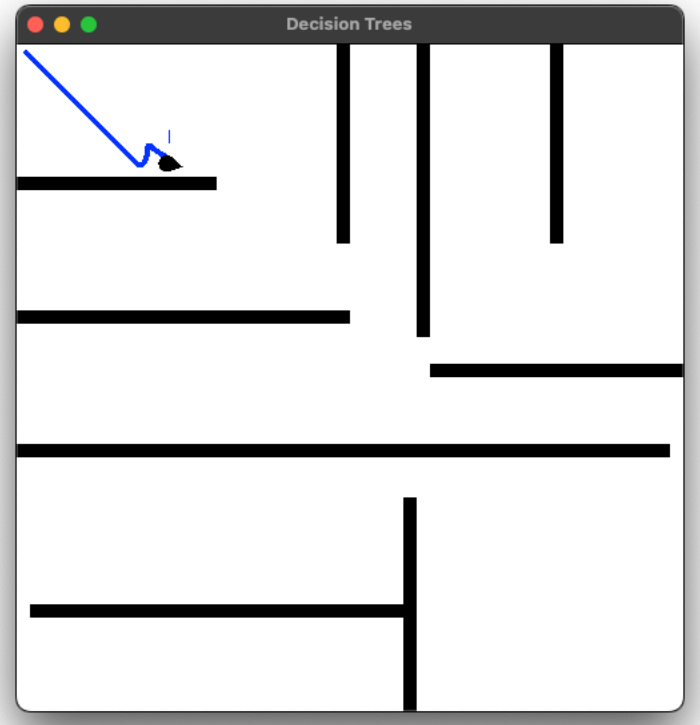
v. Screenshots: provided in the appendix (Part3)

APPENDIX:

Part1

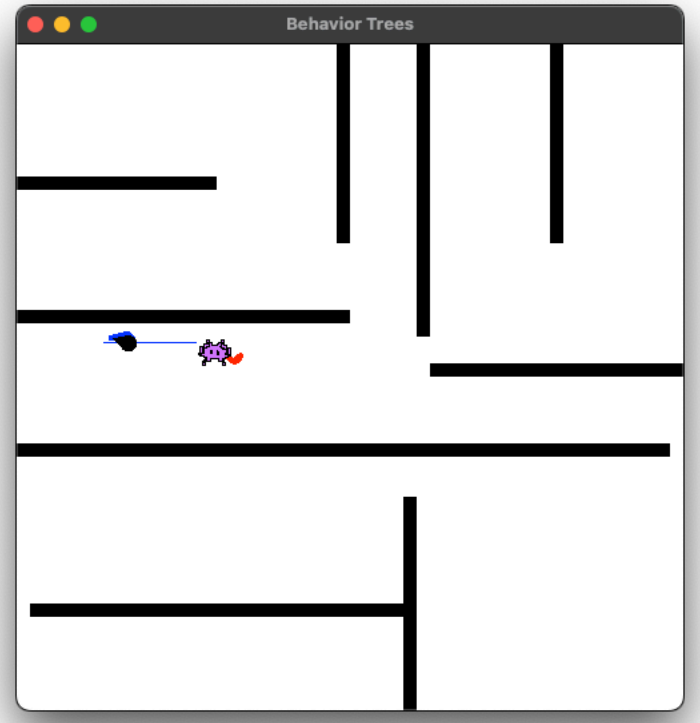


Pathfinding to center

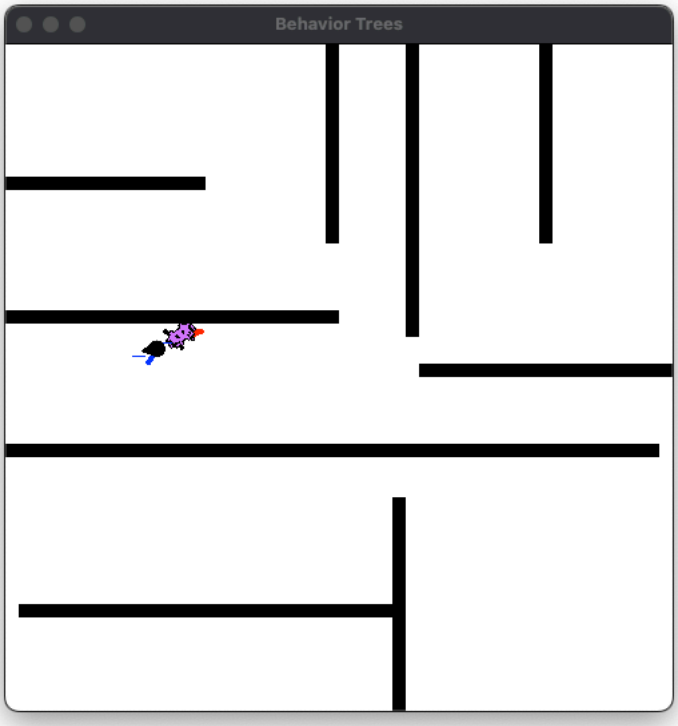


Pathfinding away from obstacle.

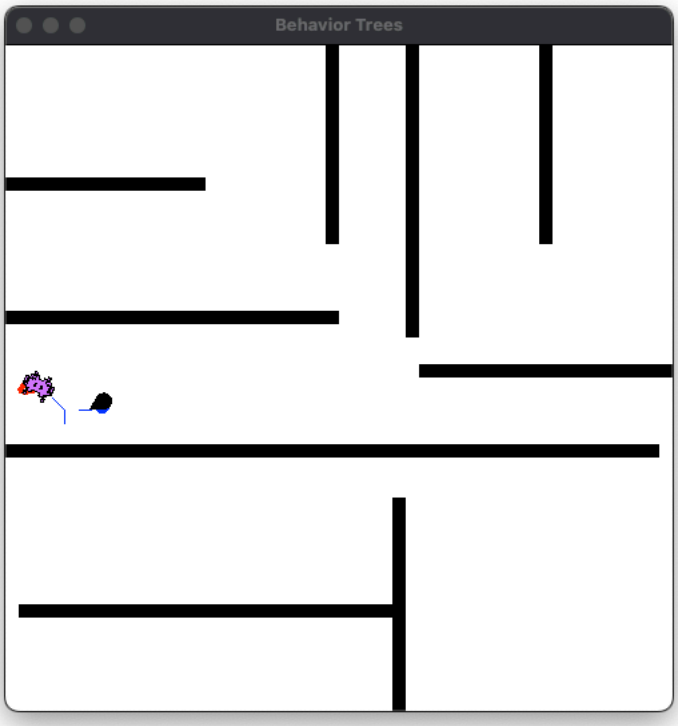
Part2



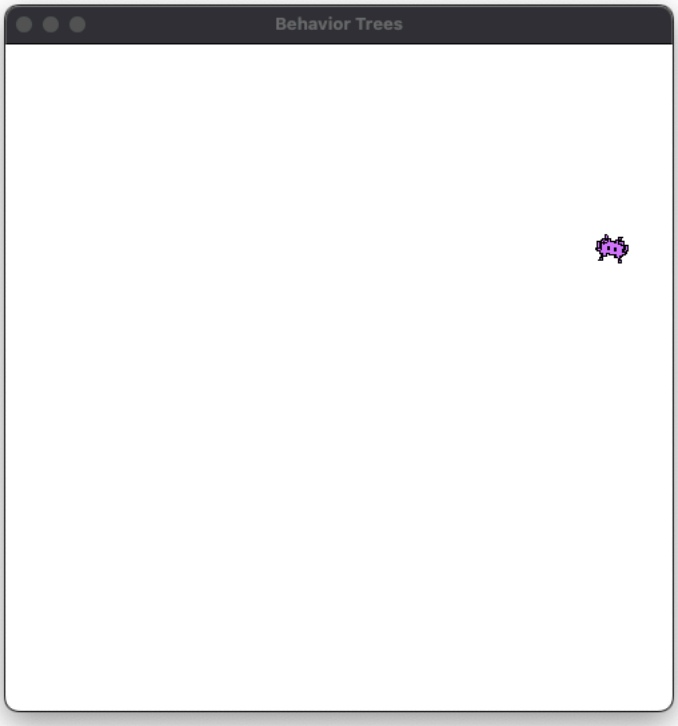
Monster pathfinding to a point near the character and character wandering



Monster pursuing and character evading

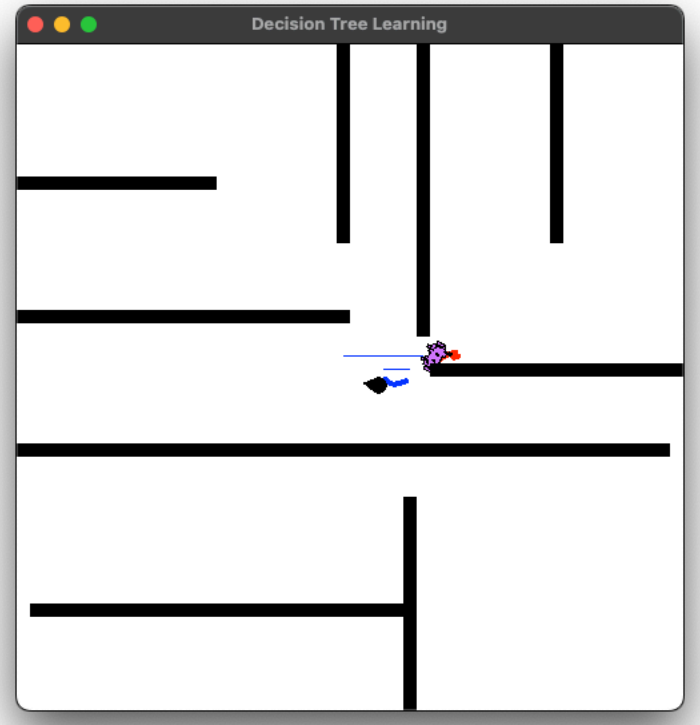


Monster and character wandering

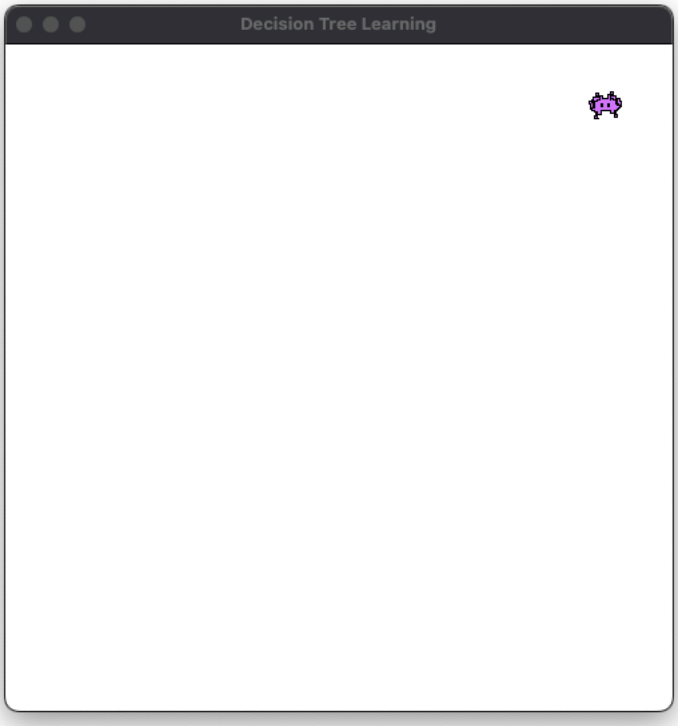


Monster dancing

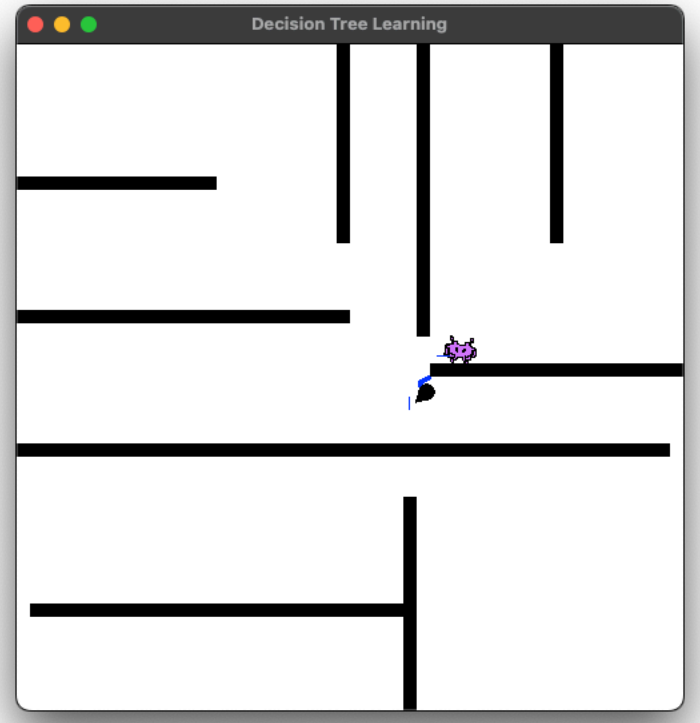
Part3



Monster pathfinding to a point near the character



Monster dancing



Monster and character wandering