IT University of Copenhagen, 2022

Foundations of Game AI

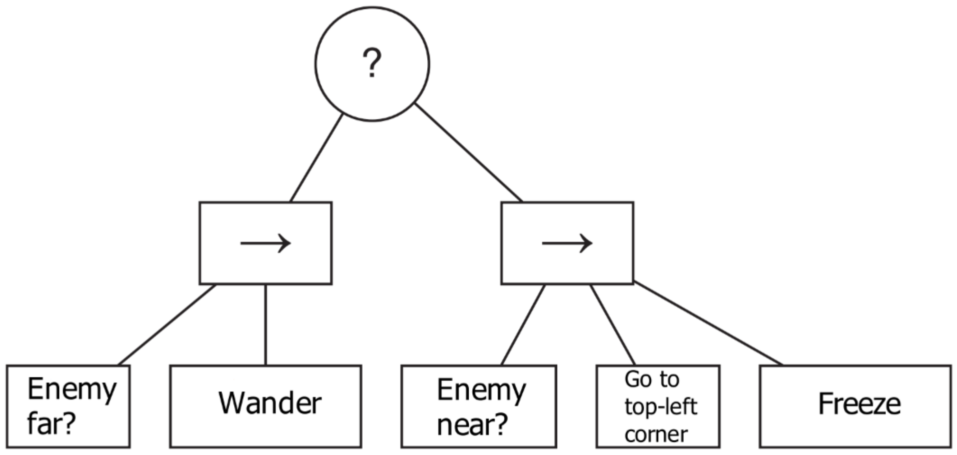
Exercise Session 5

Welcome to all. Today we will work with Behaviour Trees, as seen in the lecture and as per described in Chapter 5.4 of the book. Make sure to do the reading to have a complete understanding of the implementation.

Unlike last week, the book's implementation for the topic of this week is much easier to follow along and to implement in our game.

Exercise 1

We will try to implement the following simple tree:



Re-write this tree using Pseudo-code and if-statements (as done in page 338 of the book). Don't over-think this, it should be quite simple.

Exercise 2

Let's start implementing it: we will follow the implementation given in the book. The first class to introduce is the Task class, which every other class will inherit from.

Using the pseudo-code on chapter 5.4.2 (page 341) as a guide, implement the Task class.

[Hint: remember that when creating a class in Python, every class inherits from "object".]

Exercise 3

Next, we will need to implement the Selector task and the Sequence task, which are represented in the tree above as the "question mark node" and the "arrow node".

Using the pseudo-code from the book (page 342), implement the Selector and Sequence classes.

Are you able to explain how the different use of the if-statements defines their behaviour?

Exercise 4

Now it is time to implement the various specific tasks. Let’s start with the first one: “EnemyFar”.

Define a class that inherits from Task class which, given the distance to the enemy as a parameter, returns (in the “run()” method) True if the enemy is close, or False otherwise.

You are free to choose what the exact distance for the “enemy being close” should be: this will most likely depend on whether you use the coordinates of the two characters or the number of nodes in between them.

[Hint: remember that every Entity object has an instance variable “self.position” which is the position vector of the character, as well as a “self.target” which represents the closest node that the character is traveling to.]

[Hint 2: if you are stuck, the book has an implementation of “EnemyNear” class on page 341.]

Exercise 5

Next, implement “Wander” class. Given the character that is being controlled by the Behavioural Tree system as a parameter, it should simply set the wandering behaviour as the character’s behaviour. The “run()” method for this task should always return True.

Exercise 6

We now created all the necessary tasks for the first of the two Sequences. Before we move on to instantiate them, let’s finish defining all the necessary tasks i.e. the ones in the second Sequence.

Implement the “EnemyNear” class. It should return exactly the opposite of “EnemyFar”.

Exercise 7

Answer the following questions:

- is the “EnemyNear” class needed?

The answer to this question does not concern the fact that “EnemyNear” is the opposite of “EnemyFar”, but instead is more related to the way Sequence tasks and Selector tasks work.

Exercise 8

Now implement the “GoTopLeft” class, which, as the name suggests, makes the character go in the top-left corner of the node system. It might be useful knowing that this node is the first element of the list of nodes. It might also be useful knowing that the coordinates for this node are (16,64).

Just like the “Wander” class, the “run()” method for this class should always return True.

[Hint: when setting a new goal, remember to not simply pass the coordinates but to create a Vector2 instance].

Exercise 9

Finally, implement the "Freeze" class. We want this task to stop our character's movement once the top-left node is reached. Notice that there are different ways this can be implemented: give it a try, and check the Hint below if you get stuck.

[Hint: remember that it's the "update()" method in Entity class that updates the character's movement. Additionally, remember that because the Ghost class inherits from Entity, we can override any method... this could perhaps be used to stop the method from updating the position.]

Exercise 10

We have now defined all the tasks constituting the Behavioural Tree. Let's use it to modify the ghost's movement.

In the "ghost.py" file, implement a method "behaviouralTree()" in which we will create the various Task objects using the classes we just defined. Specifically create the following:

- an instance of EnemyFar;
- an instance of Wander;
- an instance of Sequence, which will have the instances of EnemyFar and Wander that you just created as children;

- an instance of EnemyNear;
- an instance of GoTopLeft;
- an instance of Freeze;
- a second instance of Sequence, which will have the instances of EnemyNear, GoTopLeft and Freeze that you just created as children;
-
- an instance of Selector, which will have the two instances of Sequence that you just created as children.

Remember to pass the necessary parameters (the ones that you used in each of the various "run()" methods) when initialising the various objects.

Finally, you want to call the Selector's "run.py" method to execute the tasks. Make sure that you call "behaviouralTree()"in the "update" method, or else the character will not be subject to the results of the Tree.

IMPORTANT: if you get no errors but the behaviours is incorrect (i.e. the ghost constantly wanders, even after colliding with the enemy), this is normal; go to the next exercise.

Exercise 11

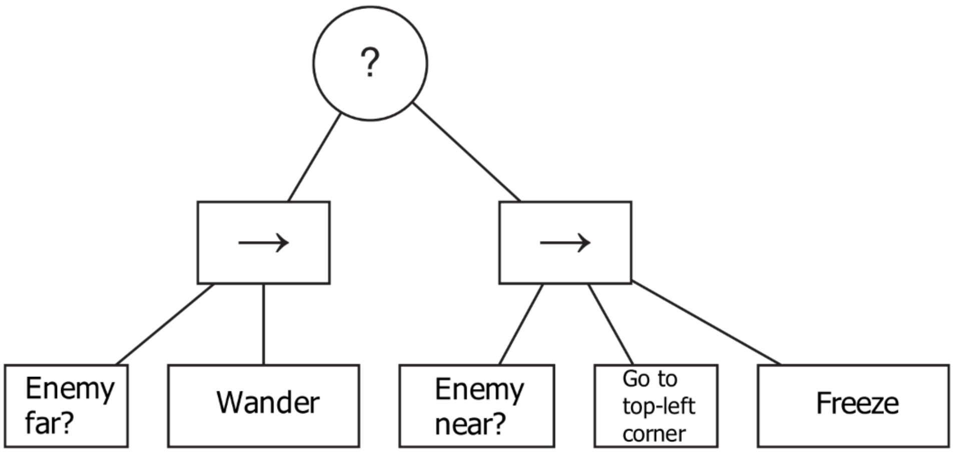
You will notice that there is something wrong: even when correctly implemented (code-wise), the behaviour will not exactly represent what we designed in the Tree diagram at the top of this PDF.

Before I tell you the reason, can you understand what the problem is, and why it happens?
Look at the diagram: at every iteration of the game loop, the tree is traversed. Why does this create a problem?

Answer:

The problem lies in what happens after the two characters collide. When they collide, our ghost's "self.goal" and "self.directionMethod" get updated to the top-left corner and to whatever path-finding algorithm you chose. This remains true for as long as the two characters are within the range that you chose in the test for EnemyFar task.

Looking at the diagram (which I pasted here below), this means that we go down the left-most branch, evaluate the EnemyFar task, it returns False (since the characters are within the defined range), then we go back to the "?" node (the Selector node), and go down the branch that leads to "EnemyNear", which evaluates to true: as a consequence, GoTopLeft and Freeze are successfully executed.

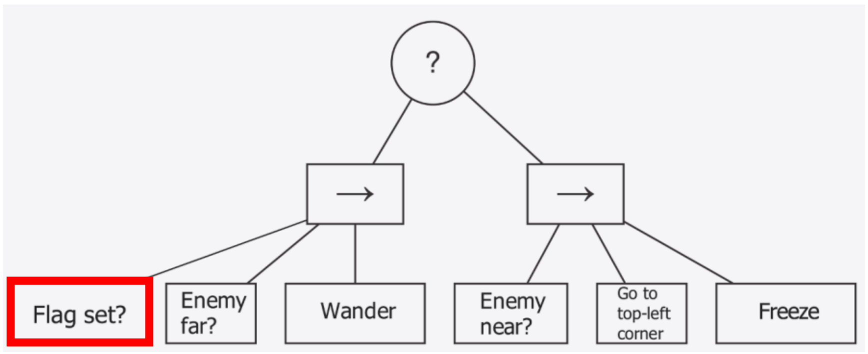


The issue with this is that as soon as the characters exceed that range (i.e. are apart from each other again), the EnemyFar will again return True, effectively setting the ghost's behaviour back to Wander.

Can you think of how this problem can be solved?

Exercise 12

One possible solution for this is to use a flag. Whenever we want to stop the Tree from evaluating again the EnemyFar task, we set the flag to True: a new Task , "IsFlagSet" can then be inserted before EnemyFar, and if the flag is set to True, the Task returns False, essentially skipping the EnemyFar and Wander Tasks. The new diagram would looks as follows:



Implement "IsFlagSet", which checks if the above-described flag is set, returning False if it is.

[Note: you have to implement the flag too. A simple instance variable should do.]