IT University of Copenhagen, 2022

Foundations of Game AI

Exercise Session 4

Welcome to all. Today we will work with Finite State Machines, as seen in the lecture and as per described in Chapter 5.3 "State Machines" of the book.

Exercise 1

Right now, we only have two possible states for the characters: SEEK and FLEE. To make the system more interesting, let's implement the WANDER behaviour we saw in the steering behaviour lecture. In the "entity.py" file, find the "wanderRandom()" method.

Given the list of possible valid directions as argument ("directions"), randomly pick one of the directions and return it.

[Hint 1: we already implemented a "randomDirection()" method]

[Hint 2: to select this method as the character's behaviour, assign it to the instance variable "self.directionMethod = self.wanderRandom"]

Exercise 2

That was easy. But the resulting behaviour is a bit TOO random. Let's add a bias, and create the method "wanderBiased()".

Given the list of possible valid directions as argument ("directions"), the character has a 50% probability of going in the same direction that it is already traveling (if possible), while only having a 25% probability of selecting the other two options. If the previous direction is not possible (i.e. we hit a wall with only left or right as directions), then a random direction should be returned.

[Recall: when we implemented the code in our first exercise session, we imposed as a rule that the characters cannot reverse its direction. Hence there only being 3 total possible directions]

[Hint: if you are unsure of Python syntax, Google is your best friend e.g. "python how to get random int", "python how to access element from a list at specific index", …]

Exercise 3

We now have 3 states: let's start doing more interesting things. Find the "FSMstateChecker()" method. We want to ensure that the right direction method is used for each state without us having to manually modify the code, but rather press a key ("S" for seek, "F" for flee and "W" for wander) and have the code take care of it. We will deal with the key press functionality in the next exercise.

For now, open the "constants.py" file and make sure there is a variable for each state. We are doing this so that we can simply write SEEK or FLEE rather than having to use a string.

In the "entity.py" file, create new instance variables in the "__init__()" method:

- a variable "self.states" that holds a list of the newly created constants,
- a variable "self.myState" that keeps track of the character's state (initialise with zero).

Navigate now to the "FSMstateChecker()" method. Using a sequence of if-statements, change the behaviour of the characters so that the functions used in each of the three states are the following:

- SEEK → A* algorithm implemented last week;
- FLEE → the fleeing behaviour used for the ghost in "goalDirection()" method from the first exercise session (make sure that the order of the subtraction is: position vector – goal vector and that the "max" magnitude is selected rather than the "min");
- WANDER → the "wanderBiased()" method we just created;
- ELSE → pick a random state from the "states" list.

[Notice: this function doesn't return anything, it should only change the state of "self.myState" variable. Thus, delete the "return" statement (it is present now for avoid the code from crash).]

[Hint: the behaviour of a character is changed by setting the "self.directionMethod" variable to the wanted method.]

Exercise 4

Now we will make the state of our characters change based on keyboard inputs. Navigate to "run.py" file and find "checkEvents()" method. This is the function that we created in the first session to detect whether the player clicks on the "X" to exit the game. We will use it to check for key presses.

For each of the three key presses (K_s, K_f, K_w, corresponding to "S", "F" and "W" keys respectively), assign the respective state of the **ghost** character.

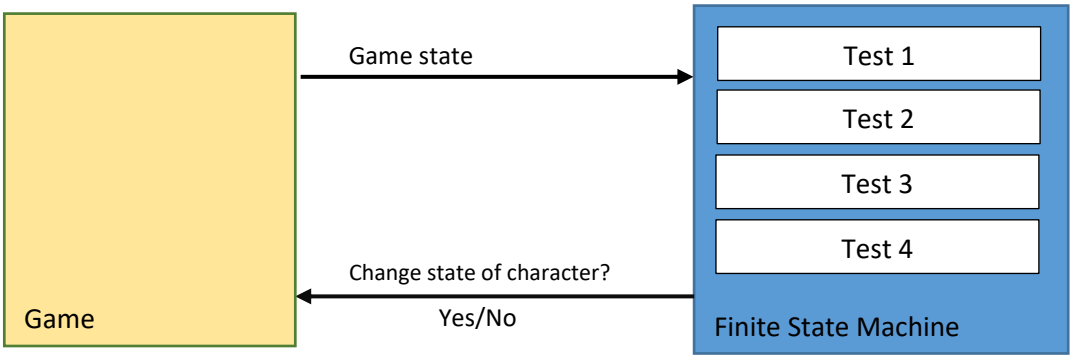
When executing the game, you should see the behaviour of the ghost changing with each key press.

Exercise 5 *(OPTIONAL)*

Now implement the same mechanism for PacMan. Make sure you use different keys.

Exercise 6

What we just made is a very simple implementation of a Decision Tree. Let's now take an abstraction leap and implement a Finite State Machines. The concept will remain the same: we want to change the state of characters, but this time instead of checking conditions (if a button was pressed => change state), we will pass all the information about the game to an "external checker" (the FSM) which will run specific tests and return as a result whether a change in state should occur. Look at the image below for a visual representation:



Navigate to "FSM.py" file and observe the code structure. This structure is taken from the Pseudo-Code in chapter 5.3.3. Make sure you read Chapter 5.3 to understand FSM in complex games.

Exercise 7

Now that you understand the structure of FSMs in complex games, let's apply this knowledge to our much simpler PacMan game.

We will use this States diagram:

- When in WANDER:
  - If character is in the top-left quarter of the map within 2 to 5 seconds from when the wandering behaviour started, then switch to SEEK.
  - If 5 seconds pass and character is still in wander behaviour, switch to FLEE.
- When in SEEK state:
  - If enemy is less than 2 nodes away, switch to WANDER.
- When in FLEE state:
  - If character hits one of the four corner nodes, switch to SEEK.

Now, draw the described States diagram. Make sure to specify the triggering condition for each transition on the arrow that connects one state to another.

Exercise 8

Let's implement the FSM. We will start by creating the Transition class, which will describe a transition from one state to another: these start and target states are passed as attributes during initialization (start_state, target_state).

The first methods we will implement are the tests for checking whether a transition is triggered.

Using the state diagram description from the previous exercise, implement the test method for the transition from the SEEK state (i.e. "seek2wander()" method).

Return True if the transition is triggered, False otherwise.

[Hint: you might want to (you should, actually) pass some parameters to the method, such as the path that is returned from the A* algorithm that leads the character to its goal]

Now implement the tests for the transitions starting from the WANDER state (i.e. "wander2seek()" and "wander2flee()"). Return True if the transitions are triggered, False otherwise.

[Hint: again, you should pass some parameters -i.e. the game timer (dt) and the character's coordinates]

Finally, implement the tests for the FLEE state transitions (i.e. "flee2sek()"). Return True if the transition is triggered, False otherwise.

Exercise 9

Now let's implement the "isTriggered()" method, which acts as a "driver": simply put, it runs the right tests based on which transition is being tested.

Implement this method so that, depending on what the start_state and target_state are, the appropriate test (which we have just implemented in the previous exercise) is run, and its result (True or False) returned.

[Note: Since this is the method that calls all the other tests-methods, it should take as parameters all the same parameters that you used for the smaller tests e.g. game timer, path, coordinates, etc.]

Exercise 10

Finally, let's implement the "getTargetState()" method. It should simply return the correct target_state.

Exercise 11

We have now finished implementing the Transition class. The attentive eyes might have noticed that, unlike the book's Pseudo-code implementation (Chapt 5.3.4, page 313), we have not implemented the "getAction()" methods.

- Why do we not need a "getAction()" method?

Exercise 12

Let's now move on to implement the State class. When initialising the State object, we want to pass as parameter which one of the 3 states (SEEK, WANDER or FLEE) we are creating the object for. Therefore, create an instance variable "self.state" that saves this information.

Now we have in our hands a software design problem. Think about this: imagine a hostel room where different tourist share a room with multiple beds.

- the first tourist arrives and goes to sleep, thinking that he is the only tourist in the hostel;
- the second tourist arrives, and the receptionist of the hostel informs him that there is already another tourist, and gives all the details of the first tourist to the second tourist (not very GDPR friendly…). The second tourist then goes to sleep, knowing there's 2 of them;
- finally, the third tourist arrives and the receptionist informs him of the other 2 tourists. The third tourist takes all the information and goes to sleep, thinking there's 3 of them.

Now, the idea here is that unless the receptionist goes and wakes up the first tourist, he will never know anything about the other 2 tourists. He won't even know they are there.

The same concept applies to creating objects. The first object you create will not have any information passed at initialisation (when the tourist arrives at the hostel), while you can pass the information of the first object (first tourist) to the second object (second tourist) by passing it as parameter during initialisation (at the reception).

The solution to this is to not pass any parameter at initialisation, and instead create a "getOtherStates()" method which, after all the States have been created, lets all States know about each other.

Implement the "getOtherStates()" method which takes "state1" and "state2" as parameters and, depending on the state of the object being created, creates the instance variables for the remaining 2 states, as well as the appropriate transitions (using Transition class) for the state being created.

e.g. if for the SEEK State object you call "self.seek.getOtherStates(self.wander, self.flee)" in the "StateMachine" class, then you want to initialise the Seek object so that

- "self.wander = state1",
- "self.flee = state2",

- "self.seek2wander = Transition(self.state, self.wander)".

The method shouldn't return anything, thus delete the return command after implementing.

Exercise 13

In the Book, there are 3 (!!!) methods for specifying the list of actions to complete when **entering** the state, when **into** the state and when **exiting** the state ("getAction()", "getEntryAction()", "getExitAction()"). For this simple implementation, however, we only want to change the state without making any action at all.

We will therefore only implement "getTransition()" method. Implement it so that it returns a list containing the transitions for the state it is being called on.

Exercise 14

Now that we have the Transition and State classes, let's finally implement the StateMachine class. This is the object class we will call to initialise the FSM and to pass information about the game to.

In the "__init__" method, create the instance variables for the three states using the State class. Then, for each of them, pass the information about the other two remaining states.

Finally, create two instance variables for defining the initial state, and for the current state.

The method shouldn't return anything, thus delete the return command after implementing.

Exercise 15 *(CHALLENGING)*

Using the Pseudo-code in Chapter 5.3.3 (page 311), implement the "updateState()" method (in the book it is called "update()"). Can you see which lines from the pseudo-code are not needed in our implementation?

[Hint: as said before, the book's implementation returns a list of actions to execute when transitioning, whereas we only want to change state…]

Exercise 16 *(CHALLENGING)*

Now that we have our FSM, let's use it for changing the Ghost's behaviour. Navigate to "ghost.py" file and implement the "advancedFSM()" method. Using an FSM, it should update the behaviour of the ghost according to the transition conditions we used in the FSM.

The method shouldn't return anything, thus delete the return command after implementing.

Exercise 17 (*CHALLENGING, OPTIONAL*)

Now, implement the FSM in the "entity.py" file, making it so that both PacMan and the Ghost use it. Depending on how you implemented the "advancedFSM()" method, you might have to change some lines of code.

Exercise 18 (*CHALLENGING, OPTIONAL*)

Modify the FSM so that the new State diagram is the following:

- In WANDER state:
  - After 5 turns to the RIGHT direction, switch to FLEE state;
  - If hasn't been in the proximity (less than 2 nodes away) of the enemy in the last 15 seconds, switch to SEEK state;
- In SEEK state:
  - If enemy less than 2 nodes away, switch to FLEE state;
- In FLEE state:
  - If enemy is in the opposite side of the map, switch to WANDER2 state;
- In WANDER2 state:
  - After 4 seconds, switch to WANDER state

where WANDER2 state is the same behaviour of WANDER but with speed set to 300.