IT University of Copenhagen, 2022

Foundations of Game AI

Exercise Session 5

Welcome to all. Today we will implement a GOAP system, as described in Chapter 5.7.5 of the book. It is fundamental that you do the reading to have a complete understanding of the implementation.

Sit tight, this is going to be a challenging one! :)

Exercise 1

For this exercise session, we will need the pellets and super-pellets that we skipped in the first session. Their implementation is explained in pacmancode.com's "Pellets" and "Eating Pellets" sections.

As a warm-up exercise, follow the explanation on the website and check that you understand how they work. In the provided code, they will both be already implemented.

Exercise 2

Let's start implementing the GOAP system.

We will do so in a top-down fashion: starting from the GOAP class and working our way down.

Navigate to the "GOAP.py" file. Scroll down and find the "planAction()" method. This is the core algorithm used for making the planning. This is explained in detail in Chapter 5.7.5 "Overall Utility GOAP", and it is fundamental that you read the chapter before attempting to implement it.

Exercise 3

After having done so, use the pseudo-code illustrated in the chapter (in the "Pseudo-code" section, right after "The Algorithm") to implement the method.

[NOTE: on line 4 and line 5 of the pseudo-code we are defining of two empty lists which have size maxDepth+1 and maxDepth respectively. In Python, this can be done with:

```
list = [None] * desiredLength
```

Furthermore, note that due to minor differences in how different programming languages handle data structures, we will need to use number of all possible actions in stead of the maxDepth. For avoiding confusion, these two lines of code are already implemented for you. Don't worry that "self.actions" is not defined, you will define it later.]

Hint: to solve the problem of Python not performing a copy when assigning "models[currentDepth+1] = models[currentDepth]", you can import the library "copy" which has a function "copy.deepcopy(____)" which can solve the issue.

Exercise 4

Good job on implementing the previous method!

You are probably wondering what all the objects are that use used in this method are. Let's implement them and you will understand.

Let's start by "Goal" class.

Navigate to "goalAndActions.py" file. We will follow the same structure used in the book in the earlier Chapter 5.7.2.

Implement the "Goal" class, which has as instance variables:

- name
- value

Additionally, implement the methods:

- "getDiscontentment()" which returns the discontentment for the current world model (as explained in the book, we will use the square of the highest goal's priority value as the discontentment for the current world model);
- "updateValue()" which takes a newValue as parameter and sets it as the new value of the "value" instance variable defined in the constructor. It doesn't return anything.

Exercise 5

Now implement the "Action" class. As shown in the book, it also has two instance variables:

- name
- value

And one method:

- "getGoalChange()": leave empty for now, it will be overwritten by Action's subclasses later.

Exercise 6

Now let's talk design.

The way we will implement our GOAP is this:

- 2 goals:
    a. Kill the enemy
    b. Eat pellets
2. 3 actions for each:
    a. actions for Kill the enemy:
        i. following the path to the target
        ii. go in the same quadrant as the target
        iii. accelerate
    b. actions for Eat pellets:
        i. go to a different quadrant than the one currently in
        ii. wander
        iii. go to the closest corner

We will have an Action class for each of our possible actions, which will extend the Abstract class we just implemented. Each of these classes will be initialised with a value. Additionally, based on which goal is the highest, each action updates the goal's priority value, as explained in the algorithm in the book. For your convenience, the following classes have been already implemented:

- "FollowPathToTarget()"
- "GoInSameQuadrant()"
- "Accelerate()"
- "VisitAnotherQuadrant()"
- "Wander()"
- "GoClosestCorner()"
- "Dummy()"

Make sure you agree with their implementation and go to the next exercise.

[Note: the "Dummy" class is needed to make the algorithm run correctly. More on this later]

Exercise 7

Now we will implement the WorldModel class. Navigate to "world.py".

Implement the "__init__()" method, where you will define the instance variables needed to describe the world model at a particular point in time; these are:

1. goals: a list of all the possible goals;
2. actions: a list of the available actions in a specific world model;
3. highestGoal: the goal with the highest priority value (more on this soon)

You will most likely need to define more, as needed by your implementation decisions of the next exercises.

Now implement the "setHighestGoal()" method, which (using the list of goals), finds the goal with the highest priority value. It doesn't return anything, but instead it sets the highestGoal instance variable with the right value.

Exercise 8

Next, let's implement the methods of the WorldModel class that are used in the "planAction()" method we just implemented earlier.

Implement the "calculateDiscontentment()" method. It returns the discontentment of the highest goal. You can use the "getDiscontentment()" method implemented in the Goal class.

Now implement the "nextAction()" method. It returns the next unvisited action if present, else it returns None. There are several ways to implement this: one idea is to make a copy of the list of actions and pop the first element off each time the method is called, and return it. Feel free to make your own decisions.

Finally, implement the "applyAction()" method, which simulates the application of a chosen action by calling the "getGoalChange()" method implemented in the Goal class. It does not return anything.

Exercise 9

Now we are ready to implement the rest of the GOAP class. Navigate back to "GOAP.py".

In the "__init__()" method, define the instance variables needed to run our GOAP. These are:

- a depth instance variable representing the number of look-ahead moves we will analyse (see book for more explanation);
- a world State instance variable, which we will first initialise to "None", and will later update;
- an instance of the Goal class for each of the goals (initialise killGhost with value 100 and eatPellets with value 1)
- an instance of Action class for each of the 6 possible actions (each of which will use the appropriate class we implemented in exercise 6)
- a list containing all the defined actions (including the dummy action).

Exercise 10

Now we will implement the "updateWorldState()" method. This is the method that is called on every iteration of the game loop to create a new world to describe the new state of the game.

It does not return anything, but simply creates a new instance of the WorldModel class, and passes the needed parameters to it (e.g. goals, actions, whatever else you implemented).

Exercise 11

We are finally getting ready to assemble everything and make it work.

We will now implement the method that updates the values of the different actions, so that the character will be able to select a different action based on the game state.

You are free to decide how you want to design your game. In my implementation, I follow these rules:

- If the goal is Kill enemy:
  - If I am in the same quadrant as the enemy, use follow path to target action;
  - If I am in the quadrant diagonally opposite to enemy, go to same quadrant as enemy;
  - If I am in one of the two neighbouring quadrants to enemy, accelerate;
- If the goal is Eat pellets:
  - Rotate (with randomized start) between:
    - Go to closest corner (duration = 2 seconds)
    - Wander (duration = 4 seconds)
    - Go to different quadrant (duration = 5 seconds)
    - repeat

Implement the "updateActionValues()" method using either the rules I specified, or using your own rules.

[Note: "using a certain action" means that you set the value of that action to be considerably bigger than the values of the other actions. The way you achieve this is up to your judgment.]

Exercise 12

Next, implement the "updateGoalValues()" method. Similarly to the previously implemented "updateActionValues()", this method doesn't return anything: instead, it changes the values of the goals according to information from the game state.

In my implementation, I use a "killFlag" to check when the two characters collide: setting it to true triggers a change of the goal values that sets the eatPellets value to 100 and killEnemy to 1. After 7 seconds, the two values are reverted back to 1 and 100 respectively.

You are free to choose your own triggering conditions or to take inspiration from mine.

Exercise 13

Finally, implement the "run()" method, which takes as parameters all the information that you need to construct your world model and to test whether the actions' and goals' values should change.

This is the method that will be called in the character's update method on each game loop iteration.

In its body, implement the following method calls:

- "updateWorldState()",
- "updateGoalsValues()",
- "updateActionsValues()"
- "planAction()".

For the last method call, make sure you store the returned result in a variable: this is the next action that was computed by the GOAP, and that will be executed by your character.

As such, this "run()" method should return this action.

Exercise 14

Let's now implement everything in Pacman's class. Move to the "pacman.py" file. In the constructor (the "__init__()" method) initialise an instance variable for the GOAP object, as well as any other variable you need for the type of implementation you chose. In my case, I need instance variables such as killFlag, killedTimer, quadrant, enemyQuadrant, etc.

Then, scroll down to the "updateKillFlag()" and "updateQuadrant()" methods. If you are following my implementation design, implement these methods. The first method checks whether a collision happened, and sets the killFlag variable to true for 7 seconds. The second method returns the quadrant in which a character is in. For both of these methods the way you implement it does not matter, as long as they produce the correct behaviour.

Exercise 15

Almost there. Implement the "execGOAP()" method, which calls the "updateKillFlag()" and "updateQuadrant()" methods (and stores the result of the latter), and executes the "GOAP.run()" method defined in the GOAP class.

Effectively, this is what runs the GOAP action choosing mechanism. Don't forget to pass the necessary parameters, and to store the returned result (i.e. the new action).

Exercise 16

Finally, for each of the possible actions, create a method which executes the action. This will be then used to execute the action returned by the "execGOAP()" method.

Once again, the choice of how you implement this is entirely up to what you see fit.