11/28/2020

Artificial Intelligence Final Project Report

Dino Becaj

Table of Contents

[Preparation 2](#_Toc57446402)

[Default Position Change in Agent Class 2](#_Toc57446403)

[# I did this because the agent technically starts in (3, 0), not (0, 3) 2](#_Toc57446404)

[Formulating the Solution 2](#_Toc57446405)

[Conceptual (High-Level Explanation of Thought Proceess) 2](#_Toc57446406)

[Technical (How Components from the Conceptual Were Implemented) 3](#_Toc57446407)

# Preparation

This section covers small changes I made to the default code before actually beginning to work on the code.

## Default Position Change in Agent Class

When I opened up the project files, I noticed that *wwsim.py* noted locations as (*row*, *column*) while *wwagent.py* had locations as (*column*, *row*). Because the former is the more accurate representation within the code, I modified *wwagent.py* to remain consistent with *wwsim.py.*

self.position = (3, 0) # top is (0,0). The default was (0, 3), but I changed it so it would align with the actual simulation.

Because of this change, I also had to update the calculateNextPosition function as well.

def calculateNextPosition(self,action):

# I had to reverse the positions because in CS, the first number means the row and

# the second number means the column (row, column)

if self.facing=='up':

self.position = (max(0,self.position[0]-1), self.position[1])

elif self.facing =='down':

self.position = (min(self.max-1,self.position[0]+1), self.position[1])

elif self.facing =='right':

self.position = (self.position[0], min(self.max-1, self.position[1]+1))

elif self.facing =='left':

self.position = (self.position[0], max(0,self.position[1]-1))

return self.position

# Formulating the Solution

## Conceptual (High-Level Explanation of Thought Process)

I began my search for the solution in the text, *Artificial Intelligence: A Modern Approach Fourth Edition by Stuart Russell and Peter Norvig*. The specific portion that helped me the most was Chapter 13, Section 13.6: The Wumpus World Revisited.

This section helped me conceptualize how to use probability within the agent’s decision-making process in the Wumpus World.

A picture containing text

Description automatically generated

In the diagram, we see that as the agent explores the environment, it adds squares to the *Known* section. This section tells us which squares have no pits (it cannot tell us if there is a pit because if the agent stumbles upon a square with a pit, it dies and the game ends). The frontier consists of squares adjacent to squares we have visited. These squares can be thought of as the agent’s next possible moves. When trying to see where it can move next, the agent selects a square from the frontier as the *Query* and tries to calculate the probability that there is a *Pit* or *Wumpus* in that square.

If the agent does this for each of the squares in the frontier, it can generate probability estimates for which square is the safest. Obviously, a rational agent will move to the square that has the smallest probability of containing a *Pit* or a *Wumpus*. Once the agent moves to a new square, the game either ends (because it chanced upon a *Pit* or *Wumpus*), or it adds a new square to its *Known* list.

To calculate the probability that there is a *Pit* or *Wumpus* in square *ij*, we must use the *Known* squares combined with the sensor data (agent.map).

The following is an example formula for calculating the probability of there being a *Pit* in square (1, 3).

A picture containing text, letter

Description automatically generated

The logic for the inner portion is that you are calculating the probability that there would be breezes in the squares where you have identified breezes, given that there are no *Pits* in the *Known* squares, that there is (or isn’t) a *Pit* in square (1, 3), and that there are/aren’t pits in the squares in the frontier. This is the model generation portion of the code. You are generating models with various pit configurations of the frontier. For example, if there are 2 squares in the frontier, some possible configurations are (S1 = pit, S2 = pit), (S1 = no pit, S2 = pit), (S1 = no pit, S2 = no pit), (S1 = pit, S2 = no pit).

Obviously, some of these models cannot hold given the sensor data. For example, given that you have identified breezes, there is no possible way that (S1 = no pit, S2 = no pit) could be true. This means that the model is *false* given the sensor data and that the probability of obtaining the sensor data given that model is 0. This means that the entire probability of the first part of the inner formula is 0. For all models that are *true* given the sensor data (Knowledge Base), the probability is 1.

The P(frontier) is easily calculated. Because each square has a 20% chance of containing a pit, you assign a value of 0.2 to any square in the model frontier to which you have assigned a pit and a value of 0.8 for any square that does not have a pit. If you multiply all these values, you get the probability of that frontier configuration. If you sum up all those probabilities and then multiply them by P(P13), (which is 0.2 if you’re querying that the square has a pit and 0.8 for the opposite), you get a 2-value vector that, after normalization, gives you the probability that there is or isn’t a pit in square (1,3).

## Technical (How Components from the Conceptual Were Implemented)

First, I will discuss how I represented each of the following components of the screen capture in the previous section: *Known* and *Frontier*.

For the *Known* section, as soon as the agent visits a square, *self.position* is appended to a new class member variable, *self.known*. None of the locations in that variable contain a *Pit* or a *Wumpus.*

For the *Frontier*, once an agent visits a square, a new method (updateFrontier) is called that checks for adjacent squares above, below, to the left, and to the right of the agent’s position. If the adjacent square is valid (not beyond the board) and if it is not already in *self.known*, the square is added to *self.frontier*.

It is important to note that not just the location of adjacent squares is appended to *self.frontier*. A tuple of the following format is appended: *((row, column), pitStatus, wumpusStatus)*. Initially, *pitStatus* and *wumpusStatus* are 0. When we enumerate models, the values will remain 0 for no *Pit* or no *Wumpus* or 1 for the opposite.

The action function will call another function called calculateProbabilities. Within this function each square in *self.frontier* takes a turn being the query. We then create a list called *models* set equal to the return value of a function enumerateModels. From there, we check each model against the sensor data (knowledge base). If a model does not apply, we pop it from the list (because its value would be 0 anyway).

We don’t need to actually “plug” the valid models into the formula because their value is always 1. Instead, for each model in valid models, we calculate P(frontier) and sum them up.