Viet Nam National University, Ho Chi Minh City

University Of Science

Faculty Of Information Technology

Project 2

Wumpus World

Trần Đại Chí	18127070
Thái Nhật Tân	18127204

Lecture

Dr. Nguyễn Hải Minh

Dr. Nguyễn Ngọc Thảo

Ms. Lê Ngọc Thành

Course: Introduction to artificial intelligence

1. **Detail information**:

Student ID	**Full name**
18127070	Trần Đại Chí
18127204	Thái Nhật Tân

2. **Assignment Plan**:

Name	**Job**	**Evaluation**
Trần Đại Chí	Logic, Qlearning, Graphic game	100%
Thái Nhật Tân	Logic, Qlearning, Graphic game	100%

3. **Environment to complile and run the program**: pycharm
4. **Estimating the degree of completion level for each requirement**: 90% (making an agent can shoot an arrow hadn't been done)
5. **Input/Output**:
   - Input: 5 file map1.txt, map2.txt, map3.txt, map4.txt, map5.txt correspond to 5 maps of game
   - Output: point of agent and state of game such as: agent is staying in breeze/stench cells, fall into the pit or being eaten by wumpus and path of agent fully printed to the console screen
6. **Structure file for Wumpus Qlearning**:

main.py: main file of program where we put all functions into it.

- class MainProcess(tk.Canvas): main process for game such as: set GUI, button, map, image, handle movement, check collision…

+ def StartButtonQLearning(self): use this button to run agent with mode Qlearning (it often takes you about 5 minutes to train the again)

+ def InitImageGame(self, wallMap, breezeImg, agentImg, wumpusImg, floorImg, pitImg, goldImg, stenchImg): set all images for game. We used Image.thumbnail() from library PIL to make image when we import is not larger than the given size and Image.ANTILAS to get a high-quality downsampling fliter

+ def Create_tiles(self, x, y): create a image of a cell for drawing map game

+ def DrawTiles(self): draw all cells for map 10x10. First, self.Create_tiles(10, 10) will draw a cell in position (0, 0). Next, we use two loop to draw all cells in row and column. j in range(0,361,40) will get values (0,40,80,120,160,200,240,280,320,360) and i in range(0,321,40) will get values (0,40,80,120,160,200,240,280,320). Because our image cell thumbnail(40, 40), so row and column will be calculated: row = 10 + 40 * row, column = 10 + 40 * column, row and column in range (0,9), multiply by its thumbnail(40) then plus for a distance error value (10, this value use to control distance between objects when drawing). With this formula, min row/column will be: 10 + 40*0 = 10 and max row/column: 10 + 40*9 = 370, thus self.Create_tiles(50+i, 10+j) will get interval in range(10…370) when drawing

+ def Inizaline_Map(self): set image for game, draw tiles and create map for game

+ def StartWumpusQlearning(self): main process for running Qlearning agent and check agent is win or lose the game

+ def IsGameOver(self): if current state of game is losing, print image game over

+ def IsAgentWin(self): if current state of game is winning, print image winner

+ def AnimateGifWin(self): animate .gif image winner when agent win the the game

+ def AnimateGifLose(self): animate .gif image game over when agent lose the game

+ def ConvertLocationToCoordinate(self, column, row): convert current row/column of agent (in range(0,9)), as we wrote above, row will be: 10 + 40 *row, column: 10 + 40 * column

+ def Move(self, x, y): first we change the current row/column of agent correspond to coordinate (10…370), then check whether init random position of agent collision pit/wumpus or not then handle movement of agent. If self.x(current position of agent in column) = x(next step moving of agent to next column), if self.y(current position of agent in row) > y(next step moving of agent to next row), make agent go top, else go down. Sometimes, self.x – x or self.y – y didn't equal to 40(distance image between cells), so always check and make it equal to 40 before moving

top or down, go left or right also make similarly as go top and down. Moreover, in some cases both self.x != x and self.y != y, we will make it equal to 40 first, then check if self.x < self.y and x < y, take min value pair (self.x, x) or if self.x > self.y and x > y, take min value pair (self.y, y) for handling next moving step

+ def Left(self, x, y): handle when agent moving left and check collision if left direction has pit/wumpus/breeze/stench

+ def Right(self, x, y): handle when agent moving right and check collision if right direction has pit/wumpus/breeze/stench

+ def Top(self, x, y): handle when agent moving top and check collision if top direction has pit/wumpus/breeze/stench

+ def Down(self, x, y): handle when agent moving down and check collision if down direction has pit/wumpus/breeze/stench

+ def Player_location(self, x, y): get current position of agent for drawing image each step agent moving

+ def addWumpus(self, x, y): draw image wumpus to map game

+ def addPit(self, x, y): draw image pit to map game

+ def addGold(self, x, y): draw image gold to map game

+ def addBreeze(self, x, y): draw image breeze to map game

+ def addStench(self, x, y): draw image stench to map game

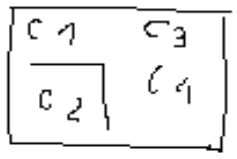
+ def create_map(self): when read file, we will convert 5 objects of map to number (character 0: empty cell, character 1: pit, character 2: breeze, character 3: wumpus, character 4: stench, character 5: gold) then use 5 functions above to draw 5 objects to map game. Purpose of converting our map is that we can use these number for evaluating our reward table

+ def checkCollision(self): we'll use function find_overlapping from tkinter canvas for checking collision. If collision wumpus/pit: game over, collision gold: add 100 points for agent and remove that gold from map, else print notification to console screen if collision breeze or stench)

+ performanceMeasure: a dictionary use for calculating point of reward table

+ def DictMapScore(scoreMap): a dictionary use for calculating point of agent correspond to each state (-1 point for moving to empty/breeze/stench cell, -10000 points for falling into pit or being eaten by Wumpus, +100 points for picking up a gold)

+ def initRewardTable(): first we will create a matrix for q_table with size 100x100 and all values are 0 (because our size of map is 10 row * 10 column = 100). Then we will get 4 neighbor cells of cell that this cell is directly reachable from previous cell and give a reward of -1 (because each cell agent move at next step will decrease -1 point), else if a location is not directly reachable from a particular location, give a reward of 0



	c1	c2	c3	c4
c1	0	0	-1	0
c2	0	0	0	0
c3	-1	0	0	-1
c4	0	0	-1	0

The image above is an example that show the result from function initRewardTable()

+ def updateRewardTable(q_table, row, column, new_state_qtable): update new value state for reward table from dictionary performanceMeasure at each step agent moving

+ def RewardTableToMap(mapGame, q_table): update reward of map from dictionary DictMapScore to q_table and use this q_table for training agent to maximize its reward

- class QAgent: main process for q_learning algorithm

+ def adjust_discount_factor(self, discount_factor): set discount_factor for agent, discount factor(gamma) is used to balance immediate and future reward. Typically this value can range anywhere from 0.8 to 0.99

+ def adjust_learning_rate(self, learning_rate): set learning_rate for agent, learning rate(alpha) can simply be defined as how much we accept the new value with the old value

+ def adjust_epsilon(self, epsilon): set epsilon for agent, agent use epsilon for balancing exploration, example epsilon = 0.1 it means its greed is 10%

+ def adjust_episode(self, episode): number of game play, update and store Q-value after an episode, when the episode initially starts, every Q-value is 0

+ def move(self, row, column, new_state_qtable): update reward table and q_table for each step agent moving, calculate new reward of agent and check if less than old reward that will lead to decrease 1 episode and start training again

+ def get_position(self): get current position of agent

+ def eval_actions(self): assume current position of agent is (0, 1), new_row = row * 10 + column = 0 * 10 + 1 = 1, new_column in range (0, 99), check reward_table[1][0.99] if any value != 0, apply that new_column position to a list available_actions, then we take q_table[1][0…99 != 0](*) compare to q_table[1][available_actions[0]](**), if (*) > (**), update max_action where max_action = available_action[0] because on each step, the agent selects maximum value over all the action for state s' (maxQ(s', a')), else if (*) = (**), we'll random a number in range(0, 1) and compare to a value epsilon for updating max_action, then update current state with max_action and return max_action for agent

+ def get_action(self): first define state_action = max_action, then row and column will be calculate: row = int(state_action / 10), column = int(state_action % 10) (example if state_action is an odd number such as 15, row = 15/10 = 1, column = 15%10 = 5 → row * 10 + column = 1 * 10 + 5 = 15 = state_action)

+ def update_qtable(self, row, column): first, we also check reward_table and appy new_column position to a list available_actions as function eval_actions(), then we use formula to calculate Q_value

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha \left(R(s,a) + \gamma \max_{a'} Q\left(s', a'\right) - Q_{t-1}(s,a) \right)$$

As we can see above, this formula is same to formula of double q-learning rather than q-learning, but I changed a bit from q-learning to double q-learning for better correclation. Next, we'll start defining the function by initializing the Q-values (a numpy array 100x100) to be all zeros. For convenience, we'll copy the reward matrix to new matrix.

[100, 100] as we have a total of 100 locations. So, we define two variables MIN_INF = -999, MAX_INF = 100 for taking a starting location and ending location of total reward. Then, we iterate through the rewards matrix to get the states that are directly rachable from the randomly choosen and assign those state to a list avai_actions. We continue to random a number in range(0,1) and compare to a value epsilon. If value random < epsilon, we pick an action randomly from the list avai_action which lead us to next state, else we will compare and update next_state = max_action as we did in function eval_actions(). Finally, we compute Q_value as formula above and update both total_reward and q_table.

+ def training(self, map): all steps in training function was done similarly to function update_qtable, except we add a function rewardTableToMap() to take score game from dictionary dictMapScore and update to our reward table for training process and because learning is a continuous, so we let the agent to explore the environment for a while and

we simply loop it through 10000 times instead of 100 times in function update_qtable() and then we'll pick a state randomly from the set of states

+ def GenerateMap(mapFile): read input file for creating map. First we read the input file, then example in first line, we count character '.' From first character '.' to character 'P' we have total 4 character '.', so we have first character 'P' in index [0][4] and we also do it similarly to character 'W' and 'G' and update these position with values correspond to dictionary dictMapScore (a dictionary store score of each state of game) and dictionary performanceMeasure (a dictionary update new value state of reward table). Finally, random position for agent in range row(0,10), column(0,10) and check to avoid these positions collision with pit, wumpus or gold

+ def SetCells(map, row, column, value): take 4 neighbor cells of a position and add correspond value

+ def GetCells(row, column): get 4 neighbor cells of a position and use positions for reward table in function initRewardTable()

7. **Structure file for Wuspus Logic**:

- class KB: using to add KB clause .Including senven list.
  + Self.unknown: using to contain suspected KB is Pit or Wumpus
  + Self.pit: using to contain KB is Pit
  + Self.wumpus: using to contain KB is Wumpus
  + Self.stench: using to contain KB is Stench
  + Self.breeze: using to contain KB is Breeze
  + Self.visited: using to contain KB is visited
  + Self.safe: using to contain KB is safe
- def PossibleMove(self,str): using to scan surrounding cells that are walkable
- def addStench(self,str): using to add into stench
- def addBreeze(self,str): using to add into breeze
- def inferePit(self,str): using to add into pit if locating it as pit
- def infereWunpus(self,str): using to add into pit if locating it as Wumpus
- class Process: using to handle clause KB
  + def FindNearest(self,cur): to find safe position nearest agent
  + def CalculateMove(self): to handle each step of agent

8. **Reflection and comments**:
   + In wumpus use q-learning, we had tested with many maps and agent perform its action to utilize food and avoid pit/wumpus very well, but our agent sometimes doesn't find shortest path to food for utilizing its score, so I think if we have more time, we should make some optimize for our Qlearning algorithm
   + In wumpus use logic, agent took a lot of time to find the path rather than q-learning because in each cell of map agent move to, agent need to add new state of that cell to KB and keep its inference to know which path is safe for it. So if map is larger than 10x10, it won't provide us a good performance as q-learning does
   + We also have another issues with wumpus use logic is that in some case if agent don't find any more path safe to move, agent will come back to the cave to end the game but agent is sometimes stucked that can lead to an endless loop or agent just stay at last position and don't move anymore, this is a problem that our group didn't have enough time to fix it
9. **Run the program**: you just need to install the library pygame and open main.py file to run it (no more packages are required)

References Qlearning:

https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/

https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/