6CCS3AIN Coursework: Pacman in a hard(er) world

Kim-Anh Vu
1707295
`kim-anh.vu@kcl.ac.uk`

November 2019

Introduction

By implementing the MDP-solver and discovering the optimum values to set specific variables, the task was to be able to guide Pacman safely around the grid and win the game by consuming all of the food that is available.

Implementation

Board Class

When the getAction() function in the MDPAgent class is called, the necessary information (current position, corners, food, ghosts, walls, legal, capsules) are assigned to local variables from the API provided.

Firstly, to get the width and height of the layout, the corners list was used to find the maximum x and y value from the list of tuples and increment each value by 1. This is a better way of calculating the height and width than finding out which coordinates had the maximum y value and accessing the value directly using indexing, as order of the coordinates could change, so height and width values would not be accurate.

Then, creating a separate Board class, which width, height and the reward value for non-terminal states (which will be the default reward value for each position on the board) is passed to. Then, set the reward of positions of the food, ghosts and capsules - replicating the layout of the board used. Also, rewards of positions surrounding the ghosts will have a value that is more than the reward of ghosts but lower than the reward value for non-terminal states. Walls do not get a reward, thus 'x' is assigned to each wall position.

Public functions were also created in the Board class to prevent other classes from accessing the board and its attributes directly, contributing to achieving loose coupling. Examples of functions that are in the Board class are setting values of each position and converting the y coordinates, as the board (represented as a nested list) has row 0 as the top row, whereas the actual layout of the game's rows started from the bottom and the y coordinate increases as you go up. Therefore, you need a function that will be able to convert y coordinates of one board/layout to another.

To obtain the updated board with each state containing an utility value, the function calls on the value iteration function.

Value Iteration

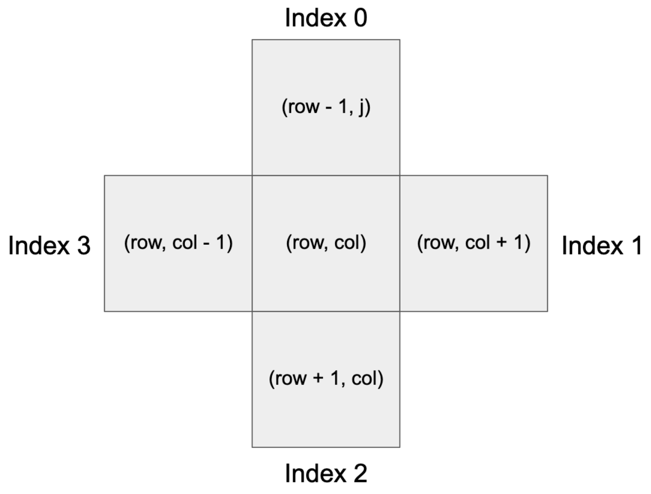
'board' will be used to obtain the original rewards for each position.
'board_copy' contains the deepcopy of the board that has been passed into the value iteration function and is the board that is used to update the utility for each state.
'U' is the previous state of 'board_copy'

> Example:
> You have updated the value at (x, y) in 'board_copy' and you needed the value at (x, y + 1) to do so. Now you need to update the value at (x, y + 1), which will need the old value of (x,

y) which will be stored in 'U'.

This function iterates through 'board_copy' for a maximum of 14 times, resets the variable total_difference (total difference between current board and future board as value iteration eventually converges) to zero and checks whether or not the cell is anything but a wall or a ghost. This is done by confirming if the position is not in 'protected_pos' which is a cocantenation of the list of ghost positions and list of wall positions. If the condition is satisfied 'calculate_expected_utility' is invoked.

'calculate_expected_utility' is a function that generates the potential next position of Pacman in a list called 'coords'. All lists in this function's ordering is always the same, with the position or utility associated with going north always comes first, then east, south and finally west. The 'coord' list which initially contains positions is looped through and any position that is a currently occupied by a wall is replaced with the current position. Due to this strategy, there is no need to check if the index is out of bounds. Then, pair each position with its corresponding reward/utility. The list is then looped through to find the positions that are at a right angle to each coordinate.



Intended Action	**90° Anti-Clockwise Action**	**90° Clockwise Action**
North (index 0)	West (index 3)	East (index 1)
East (index 1)	North (index 0)	South (index 2)
South (index 2)	East (index 1)	West (index 3)
West (index 3)	South (index 2)	North (index 0)

Table 1: Possible actions undertaken due to probability depending on intended action.

From looking at the diagram and the table, you can now work out any possible action from an intended action using indices and modular arithmetic - this removes the need to create many if statements. To get the index of the position which is 90° anti-clockwise to the intended action, you need to add 3 to the corresponding index to your action, mod 4 and to get the position which 90° clockwise to the intended action, you need to add 1 to the corresponding index to your action, mod 4.

Now that you know all coordinates that are associated when making a certain move, you multiply the value of the position that you will end up when making the intended action by 0.8 and the value of the positions if you make an action 90° to the intended action by 0.1. After, place the sum of the values in a tuple with the intended action and add it to the ‘prob’ list. When calculations are done for all possible actions, return the ‘prob’ list.

The ‘value_iteration’ only takes the first element of each tuple in the list returned by ‘calculate_expected_utility’, which is the expected utility for each intended action. The maximum expected utility is calculated and this value is inputted in the Bellman’s equation, in which the answer is then assigned as the utility of that position on the board.

In this case, value iteration will inevitably converge due to factors like the state and action space are finite and although the environment is continuous, the discount factor is less than 1 (0.9). However, convergence can take a long time, therefore a threshold is introduced to break the loop, hence reducing the running time, at a point where a long running time is not an acceptable trade-off for further iterations slightly improving the accuracy of the policy.

The total difference between the previous board and the new board in terms of utilities is calculated at the end of each iteration and if the total is less than the threshold, then the difference is so small that iterating through the board again is obsolete, therefore we break. However, if the total difference between the two boards is larger than the threshold, then we depreciate the variable ‘iterations’ by 1 and loop through the new board. When ‘iterations’ is equal to zero or when we break from the loop, the new board calculated is returned to ‘getAction’, where ‘getAction’ then calls on ‘calculate_expected_utility’ with the current position of the pacman and the new board obtained from ‘value_iteration’ as parameters. The list returned is filtered to only include actions that are legal and then returns ‘makeMove’ with the action with the maximum expected utility as the parameter.

The code was tested to return ‘makeMove’ with the action with the maximum expected utility without filtering actions that are not present in the ‘api.legalActions’ beforehand. The results were very poor (further elaborated in ‘Performance Analysis’), thus the decision to filter the list first, before finding the maximum expected utility was made.

Creativity

- Identifying positions that are at a right angle to the current position by manipulating indicies using modular arithmetic. This avoids the need to have many if statements for every action. In ‘calculate_expected_utility, only one if statement was used.
- Setting the reward of the positions that surround the ghosts to a value that is larger than the ghost reward but less than the non-terminal reward. This is essentially a warning to the pacman that going in that direction means they will be closer to the ghost.
- Used threshold and a limit to how many iterations you have to loop through the board. Prevents value_iteration from running infinitely and if the difference is small enough, does not waste computational power and time to iterate again.

Performance Analysis

Number of Iterations

No. of Iterations	Small Grid Win Rate (%)	Time Taken(s)	Medium Classic Win Rate (%)	Time Taken (s)
7	52	6.25	26	445.25
8	63	7.42	30	759.76
9	53	12.32	35	1110.07
10	60	16.22	41	1228.42
11	70	26.10	40	1660.70
12	63	30.46	37	2045.95
13	58	36.31	41	2200.12
14	64	46.89	44	2376.07
15	64	66.31	43	2495.31
16	59	50.44	38	2412.19
17	59	58.29	40	2499.86
18	69	68.32	42	2503.80
19	60	85.30	39	2366.64
20	56	64.38	38	1544.07

Table 2: Finding optimum number of iterations that results in highest win rate. For each iteration, the game was run 100 times, $\gamma = 0.9$, ghost reward = -3, non-terminal reward = -0.04, food reward = 1, capsule reward = 2.

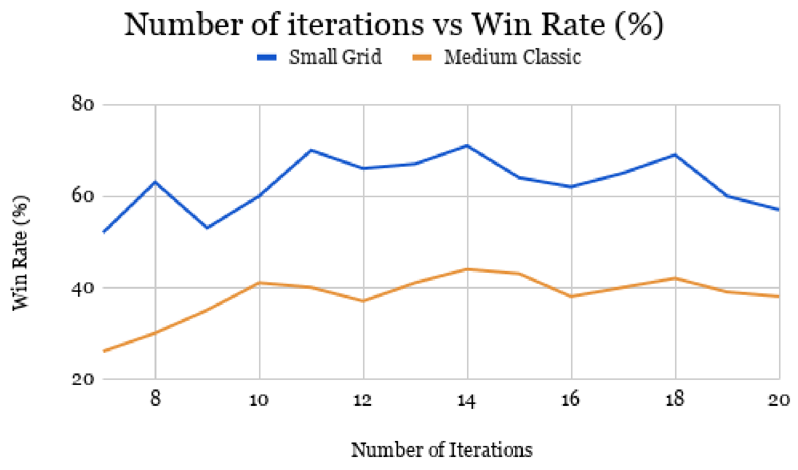


Figure 1: Graph showing relationship between number of iterations and win rate for each layout

Starting point for number of iterations was initially 1, however the average win rate for mediumClassic was 14% and after leaving smallGrid to run for an hour, the first game was still ongoing. Therefore, it was realised that the starting point needs to be considerably higher, thus 7 was randomly chosen as the official starting point. It was observed that as the number of iterations positively diverged from 7, the higher the win rate for both layouts. However, as it reaches 14 for both layouts, the win rate plateaus. There are fluctuations in win rate from 14 onwards, but it may be due to the non-deterministic nature of the game, therefore the optimal iteration value chosen was 14.

Threshold

Threshold	Small Grid Win Rate (%)	Time Taken(s)	Medium Classic Win Rate (%)	Time Taken (s)
0.1	64	125.85	51	7057.32
0.2	65	110.23	33	7011.25
0.3	62	98.34	39	6843.77
0.4	60	72.67	40	6305.70
0.5	62	54.66	42	6506.80
0.6	58	42.23	34	3479.87
0.7	61	46.75	39	3134.97
0.8	56	30.63	31	2423.78
0.9	61	28.95	37	2664.63
1.0	61	25.10.66	39	2709.44

Table 3: Finding optimum threshold that results in highest win rate. For each iteration, the game was run 100 times, $\gamma = 0.9$, ghost reward = -3, non-terminal reward = -0.04, food reward = 1, capsule reward = 2, range = (0.1, 1.0)

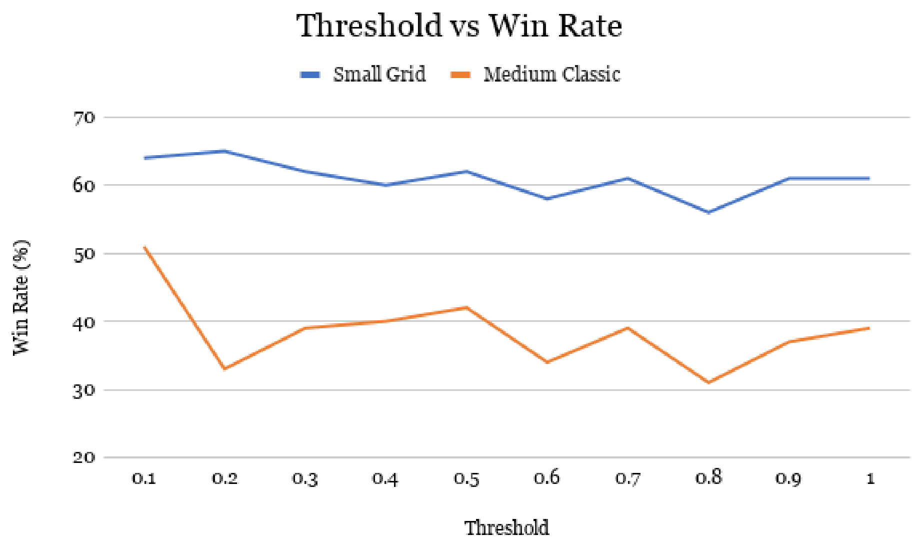


Figure 2: Graph showing relationship between threshold and win rate for each layout

The table shows the optimum for smallGrid is between 0.1 and 0.2 and the optimum for mediumClassic is 0.1. Furthermore, if you are to compare the results between using a fixed number of iterations (14) and a threshold (0.1), you can see the results for smallGrid are similar, however for mediumClassic, using a threshold slightly outperforms using a fixed number iterations alone. On the contrary, the running time difference between using threshold alone and using a fixed number of iterations is significant as on average it takes many more iterations to have a total difference that is less than 0.1. Due to evaluating the advantages and disadvantages of using different iterating strategies, it was decided to use mainly a fixed number of iterations as the win rate would be similar to using a threshold of 0.1, taking 46.81s on average faster for each game. Although, a threshold is implemented as well, so if there is an instance where the total difference is less than threshold and the program has ran less than 14 iterations, it will break the loop which reduces overall running time and computational power.

Discount Factor

Discount Factor	Small Grid Win Rate (%)	Time Taken(s)	Medium Classic Win Rate (%)	Time Taken (s)
0.0	6	1.28	0	48.19
0.1	1	1.75	0	129.14
0.2	5	1.62	14	181.35
0.3	7	2.41	17	233.43
0.4	12	3.31	31	406.39
0.5	64	5.06	34	511.38
0.6	66	6.62	35	504.32
0.7	69	9.44	43	626.83
0.8	66	9.47	46	572.78
0.9	75	9.52	50	511.64
1.0	59	9.25	37	484.08

Table 4: Finding optimum threshold that results in highest win rate. For each iteration, the game was run 100 times, threshold = 0.1, ghost reward = -3, non-terminal reward = -0.04, food reward = 1, capsule reward = 2. Number of iterations = 14.

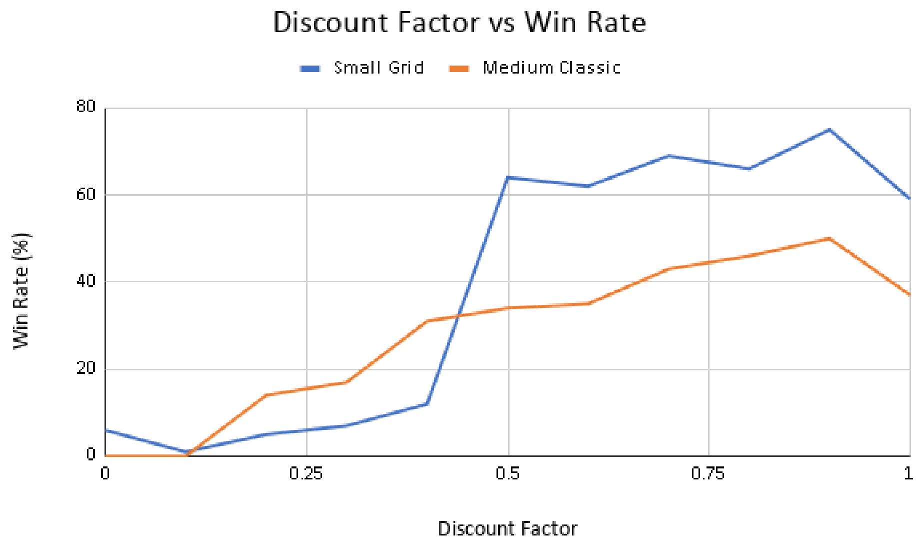


Figure 3: Graph showing relationship between discount factor and win rate for each layout

There is a common trend for both layouts where win rate increases from 0.0 to 0.9, however when the discount factor is 1.0, the win rate for both layouts dips drastically (21.3% and 26% decrease for smallGrid and mediumClassic respectively).

	Average Win Rate (%)
WITHOUT THRESHOLD	57.2
WITH THRESHOLD	60

Table 5: Win rate when causing ghosts' rewards to be -0.04 when scared and -3 when not scared. 10 iterations & $\gamma = 0.9$.

The choice was to keep the ghosts' rewards constant throughout the game was made due to the fact that it takes a shorter time to compute the final action to take and on average, the win rate is high compared to tailoring the ghosts' rewards based on if they were scared or not.

Conclusion

none