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 was 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 all the tuples and add one to each value. 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 is passed to. Then, set the reward of positions of the food, ghosts and capsules - replicating the layout of the board used. 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.

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

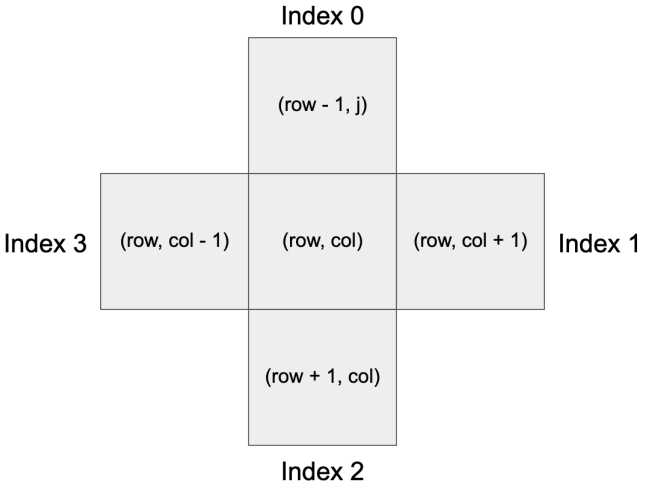
Value Iteration

'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' and checks whether or not the cell is anything but a wall or a ghost. This is done by confirming if the value stored at the cell is a number, as walls have a value of 'x' (a string) and if the current position is in the list of ghost positions ('ghosts'). 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. To make sure, there are more than one position for Pacman to move to, the algorithm checks if the set of the list (to remove duplicates) contains more than one element. If that condition is satisfied, the list is 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. 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.

Board	Win Rate	Average Win Rate (%)
SMALL GRID	377/500	75.4
MEDIUM CLASSIC	279/500	55.8

Table 2: Win rate for the two boards. Used set amount of iterations (10) & $\gamma = 0.9$.

Board	Win Rate	Average Win Rate (%)
SMALL GRID	382/500	76.4
MEDIUM CLASSIC	279/500	55.8

Table 3: Win rate for the two boards. Used a threshold of 0.01 & $\gamma = 0.9$.

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

Table 4: 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.

No. of Iterations	Small Grid Win Rate (%)	Medium Classic Win Rate (%)
10	75.4	55.8
14	74.8	0
15	78.6	57.6
16	77.8	0
17	74.4	0
20	75.2	61

Table 5: Finding optimum number of iterations which results in highest win rate. For each iteration, the game was run 500 times & $\gamma = 0.9$.

Performance Analysis

Conclusion

none