Kiem Nguyen
6CCS3AIN Coursework 1
Value-iteration for pacman

Introduction

In this coursework, I will attempt to implement a pacman AI in a stochastic environment utilising the Markov Decision Process and the Value-Iteration algorithm.

Initialising

When the pacman is initialised, all of the relevant parameters are set:

- Variable values:
  - **direcProb** = 0.8
  - **emptyReward** = -0.04
  - **discountFactor** = 1
- Rewards:
  - **foodReward** = 1
  - **capsuleReward**= 1
  - **ghostReward** = -1
- Lists containing state info:
  - Fixed:
    - **Whole** = List of tuples with coords of the whole map
    - **Walls** = List of tuples with coords of walls (using api)
  - Updated:
    - **Food** = List of tuples with coords of food (using api)
    - **Capsules** = List of tuples with coords of capsules (using api)
    - **Ghosts** = List of tuples with coords of ghosts (using api)

This is done so that these values can easily be referenced in methods and changed easily without having to search the code.

The **registerInitialState** method initializes the lists with fixed values such as **'Whole'** and **'Walls'**.

The other lists will have to be constantly updated throughout the pacman run as the game states of **'Food'**, **'Capsules'**, and **'Ghosts'** are constantly changing as pacman moves around, therefore it is updated using the api in the **getAction** method.

Functions

The **wholeMap** function:

- **Params:** *self*.
- **Returns:** List of tuples representing coordinates of the whole map.

Returns a list of tuples containing coordinates of the whole map layout by getting the last coordinate of the list of walls, and then iterating from coordinate (0, 0) to that last value. There was a possibility of using the **api.corners** method to create the whole map, however since I already created a list of walls beforehand, I could simply use that list instead of calling another api method, which could affect the performance of the program.

The **mapValues** function:

- **Params:** *self*, *state*, *map*.
- **Returns:** Dictionary with keys representing map coords and values as utilities of coords.

Assigns a value to each of the coordinates on the map depending on the entity occupying it. **Food** = 1, **Capsules** = 1, **Ghosts** = 1 (adjusted later), **Walls** = 0, Empty = Calculated with the Bellman update.
Uses the map created from the **wholeMap** function to assign utilities.

The **findMax** function:

- **Params:** *self*, *coord*, *dictMap*
- **Returns:** The largest utility out of the four possible directions from the current coord.

Creates an initial dictionary of direction-utility pairs for each of the 4 directions with 0.0.
Finds the utilities of each of the four adjacent cells from the current coordinate by iteratively calling **setUtil** below for each pair, assigns to a dictionary and returns the highest value.

The **setUtil** function:

- **Params:** *self*, *direc*, *coord*, *dictMap*.
- **Returns:** the utility value of the direction (direc) adjacent to the current coord.

Calculates the utility by first mapping the coordinates of each of the 4 adjacent cells from the current coordinate (e.g. **[key]:** NORTH = **[value]:** current *coord* + 1 to y *coord*) in a dictionary. Checks to see if the current direction is in the list of wall locations or not and if not, sets the utility to the direction probability (0.8) multiplied by the utility of that adjacent location.
The two perpendicular directions are also checked if they are walls, and the utility of adjacent positions multiplied by (1 - direction probability) / 2. These two directions are totaled with the first utility of the current direction, where it is returned.

The **valueIteration** function:

- **Params:** *self*, *dictMap*

Performs value iteration on the dictionary of coord-utility key-values by first copying the dictionary and storing it as the ‘old’ dictionary (**oldMap**). Then iterates through every coordinate in the dictionary, using the list of coordinates created with the **wholeMap** function and checks if the current coordinate is not a **wall**, **food**, **ghost**, **capsule** or within ghost **avoidRadius** i.e. an empty space. If the coordinate is an empty space, assigns a new utility value to the coord key using:

- *reward* + (*discount* * **findMax**(coord, oldMap))

The **getAction** function:

- **Params:** *self*, *state*

Gets all the current legal actions pacman can make, and removes the “STOP” action. This is so that pacman does not voluntarily stop at any point in the game.
Calls the api for **food**, **capsules**, **ghosts** and **pacman** itself to get their new state, where it then calls the **mapValues** function to update all of the utilities of cells in the map.
The **valueIteration** function is then called with the map from **mapValues** to converge all of the utility values, where **api.makeMove** is then called with the direction key that has the highest utility value from the **findMax** function.

The **gridPrint** function:

- **Params:** *self*, *state*, *map*

This function is purely used for debugging purposes by printing a visual representation of the pacman map environment, however with utilities in each space instead of entities.

Creativity

For added creativity, I added a way for pacman to see ghosts better and decide whether or not it is worth it to chase the ghost based on the changing state, or run away from them.
This is done by updating areas around ghosts proportionally to its current state i.e. scared timer.

The **ghostRadius** function:

- **Params:** *self*

Updates the new list variable **radiusList** which is a list of all coordinates within the new **avoidRadius** range that are not wall coordinates. Iterates through the list of ghosts in the game, then iterates through a range that is **avoidRadius’s** away from the ghost position in both the x and y plane i.e. if ghost = (2, 3), then a valid coord in the range would be within (2 +/- **avoidRadius**, 3 +/- **avoidRadius** + 1).
The **avoidRadius** variable is also calculated based on the size of the map. The smaller the map, the smaller the radius. This is because if the radius is fixed, the map might be too small for the radius if the radius is set to a larger number.
The **ghostRadius** function is used because the basic implementation does not allow pacman to see ghosts behind a line of food, as the utilities of any cell with food in it is always set to **foodReward** (default to 1). This function removes this problem by setting the utility values of food within a certain radius of ghosts to have a lower utility value than 1.
This list is then used by the **mapValues** function where it sets all of the values in the **radiusList** to a different value (which is calculated by the **ghostValue** function below.)

The **ghostValue** function:

- **Params:** *self*, *state*

Utility values of ghosts and cells close to ghosts are calculated based on the current state of the ghost. First, the list of ghosts and state times are collected and stored in the **api.stateTimes** variable, where it is then iterated through and the state time is checked for each ghost. If the

time is 0, the default reward value of the ghost is set, otherwise the utility is calculated by subtracting half (20) of the initial scared time (40) from the remaining scared time, then dividing all that by 2.5. This gives a range of utility values from 8 to -8, which is the default value of the **ghostReward**.

If pacman has just eaten a capsule (40 ticks left), it is not risky at all to pursue and chase a ghost, hence the ghost's utility value is high.

If a small amount of time has passed since pacman has eaten a capsule (i.e. 20 ticks left), it's utility is less.

And if it has been a long time (i.e. 10 ticks left), then the ghost is close to turning back to normal, therefore it would be very risky to try to eat the ghosts, hence it will have a negative utility.

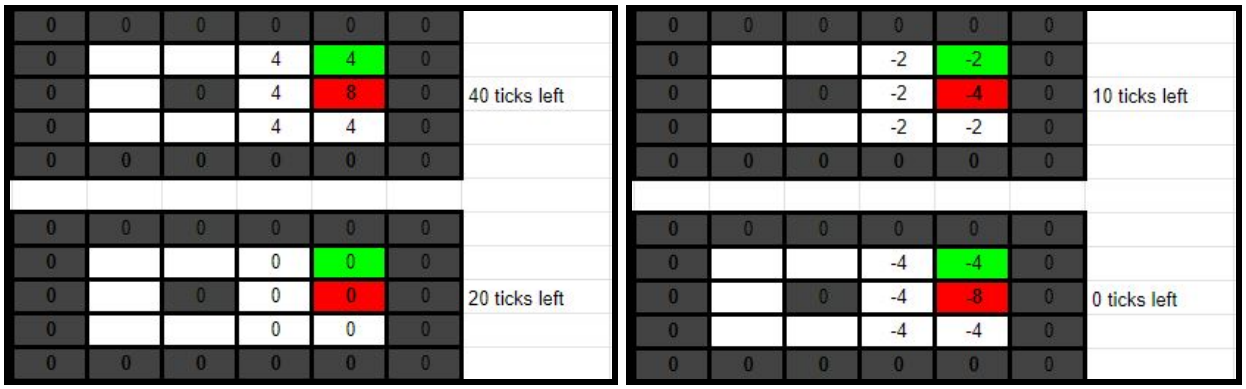

Figure 1: Surrounding utility values proportionally change depending on ghost utility. Red = ghost, Green = food, White = empty.

Testing of **avoidRadius** values:

smallGrid		
avoidRadius	1	2
Test	Wins	Wins
1	5	0
2	5	0
3	4	0
4	6	0
5	6	0
6	5	0
7	7	0
8	7	0
9	5	0
10	5	0
AVERAGE	5.5	0

mediumClassic		
avoidRadius	1	2
Test	Wins	Wins
1	0	1
2	2	2
3	1	4
4	4	0
5	2	1
6	2	4
7	4	2
8	4	2
9	1	3
10	1	1
AVERAGE	2.1	2

Figure 2: Tables representing average wins based on ghost avoidRadius

Here Figure 2 shows that in the **smallGrid** map, having a radius greater than 1 will result in 0 wins in total, however in **mediumClassic**, the number of wins are roughly the same. Therefore it is more beneficial to keep the **avoidRadius** at 1, instead of calculating it based on the size of the map.

Evaluation

Here I will test out my implementation using different parameters such as changing the discount factors and rewards for entities.

This is the testing done for the **smallGrid** map. Here I changed only the **ghostReward** and **discountFactor** parameters between each batch of 10 tests and recorded the average number of wins for each batch for a total of 500 games, 100 games for each value test.

The bar charts show the average number of wins, minimum and maximum wins as well as the discount factor based on the parameters **ghostReward** and **discount** in **smallGrid**:

ghostReward	-1	-2	-4	-4	-8
Discount	1	1	1	0.5	0.5
Test	Wins	Wins	Wins	Wins	Wins
1	6	5	3	5	5
2	4	7	3	4	5
3	4	5	3	5	7
4	4	8	6	6	3
5	6	9	6	5	4
6	3	7	5	6	3
7	5	5	4	3	4
8	7	4	5	4	8
9	5	5	4	5	3
10	5	4	7	4	5
AVERAGE	4.9	5.9	4.6	4.7	4.7
MIN	3	4	3	3	3
MAX	7	9	7	6	8

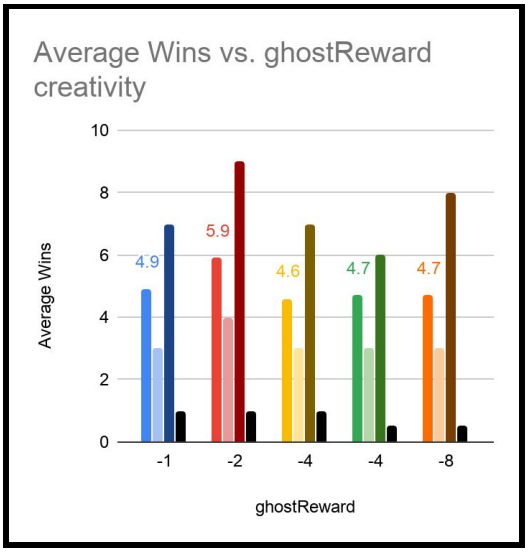

Figure 3: Table and graph of wins based on parameters in smallGrid

Based on the information provided by these graphs, in the **smallGrid** map, having **ghostReward** at -2 and **discount** at 1 seems to have the highest average number of wins as well as the highest minimum and highest maximum wins.

ghostReward	-1	-2	-4	-4	-8
Discount	1	1	1	0.5	0.5
Test	Wins	Wins	Wins	Wins	Wins
1	5	3	3	4	1
2	1	1	3	4	4
3	0	1	2	2	2
4	3	2	3	3	1
5	2	4	6	2	2
6	5	0	2	4	2
7	4	3	0	1	2
8	3	1	2	1	3
9	0	2	2	4	0
10	6	2	3	3	1
AVERAGE	2.9	1.9	2.6	2.8	1.8
MIN	0	0	0	1	0
MAX	6	4	6	4	4

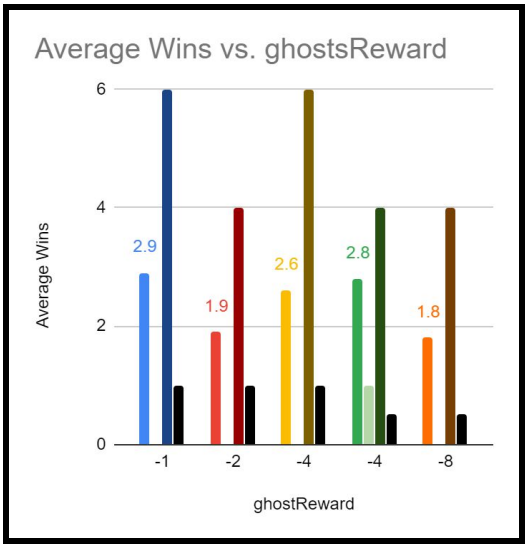

Figure 4: Table and graph of wins based on parameters in mediumClassic

Here the testing done on mediumClassic shows that having **ghostReward** at -4 and **discount** at 0.5 yields the highest minimum wins, whereas having them at -1 and 1 respectively yields the highest average and maximum wins.

Further Testing and Conclusion

avoidRadius	1	1	1	1	1	1
ghostReward	-1	-2	-4	-1	-2	-4
Discount	1	1	1	0.5	0.5	0.5
Wins	50	59	57	58	57	58

avoidRadius	1	1	1	1	1	1
ghostReward	-1	-2	-4	-1	-2	-4
Discount	1	1	1	0.5	0.5	0.5
Wins	11	21	15	19	28	12

Figure 5: smallGrid and mediumClassic tested for 100 games

In **smallGrid**, all of the tests resulted in pacman winning more than 50% of its games, passing the 40% mark. However in **mediumClassic**, only 2 tests resulted in pacman winning more than 20% of its games: having **ghostReward** at -2 for both and **discounts** at 1 and 0.5 respectively. Therefore, it would be best to set the **ghostReward** to -2 and **discount** to 0.5, as this achieves good results for both maps: 57% for **smallGrid** and 28% for **mediumClassic**, with both beating their respective pass marks of 40% and 20%.

	smallGrid	mediumClassic
avoidRadius	1	1
ghostReward	-2	-2
Discount	0.5	0.5
Wins	272/500	140/500

Further testing with 500 consecutive games yielded a 54.4% win rate in **smallGrid** and 28% win rate in **mediumClassic**.