An MDP solver for Pacman in a fully observable, non-deterministic environment

Adam Stanley - 11/12/2020 - k20110590 - Msc Artificial Intelligence

***Abstract*—The aim of this report was to design an effective Markov Decision Process (MDP) solver for pacman in a dynamic, fully observable environment where pacman's movement is non-deterministic. The sole focus of pacman is to win games, score is not taken into account when judging performance. This report concisely discusses and evaluates decisions which were made in the design process of this MDP in order to enhance performance.**

I. Introduction

In this environment pacman has a full picture of the world around him however his motion model is non deterministic. For every move taken, there is a probability of 0.8 that pacman moves in the intended direction and a probability of 0.1 that he moves in a direction perpendicular to that which was intended, for both respective perpendicular directions.

The particular method by which this agent updates its current understanding of its environment and surroundings is through value iteration. At every step in the game, an initial utility map is created in which utilities are just assigned to the rewards for their corresponding coordinates. This initial map is then run through the value iteration algorithm using the standard version of Bellman's equation:

$$U_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U_i(s')$$

until convergence is achieved. Pacman then uses this iterated map and selects the move with the maximum expected utility out of all possible moves.

II. Implementation & Methodology

A. Environment & Terminal States

Since pacman and the ghosts move with every time-step, the map is dynamic and constantly changing. For this reason, each step of the game can be viewed as an individual Markov decision process in which pacman need to decide where to move next. The only terminal state in the game is the last piece of food on the board. Hence at every time-step, pacman updates the entire utility map.

The original design of this MDP viewed food, capsules and ghosts as a terminal states (for each step) and therefore only updated the utility map in locations which were empty. This was mainly to reduce computation time at each step and also due to the fact that iterating over the entire map can be quite redundant. However it was advised that this technically isn't an MDP solver therefore the design was altered to iterate over the entire map and to treat food as non-terminal.

The maps `mediumClassic` and `smallGrid` are very contrasting in terms of size, amount of food available and number of ghosts in the environment. Hence it is wise to focus on each map individually and to build slightly altered MDP solvers for each map. Overall, the methodology used is identical for both map variations but for minor tweaks to certain aspects. In particular when it comes to the ghosts.

B. Rewards

The intuition behind the rewards assignment comes from the fact that we want pacman to be a good ghost avoider and at the same time able to win games by eating all the food. The initial reward specification which was assigned is seen in the table below. These rewards were to be tuned and optimised later on however the overall design of negative for ghosts vs. positive for food etc. remained throughout.

Food	Capsules	Ghost	Empty space
10	10	-10	0

TABLE I: Reward specification

In order to increase the priority and spread of food when there is little food left, food rewards are increase by a multiple of 5 or 2 depending on how many food pieces are remaining on the grid. This helps ensure that pacman can still successfully find the food even when it is very scarce or far away. For example:

No. of Food left	< 10	< 20	$20+$
Reward	50 ($\times 5$)	20 ($\times 2$)	10 (Standard)

TABLE II: Food scarcity reward

Regarding the reward for pacman's current position on the map, the initial implementation gave pacman a negative reward of -1 in order to encourage pacman to always move away from his current position. However this resulted in poor performance as sometimes pacman was so eager to move from his current position that he would turn in a direction towards a ghost and subsequently die. After some experimenting with different rewards it was found that keeping pacman's reward for current position the same as all other empty spaces (0) was optimal.

C. Ghost Avoidance

Further steps need to be taken to ensure pacman safely avoids ghosts. Due to the max operator in Bellman's equation for utility updates, the negative ghost utilities do not spread throughout the map. Ghosts can hide behind areas of food and pacman will not detect them until they are directly beside him.

In order to ensure early detection of ghosts, the reachable regions around the ghosts within a certain step limit were given

Fig. 1: Ghost Region example

suitably negative rewards decreasing in magnitude depending on how many steps from a ghost a coordinate is. Figure 1 illustrates this region. The size of these ghost regions is dependent on how afraid one wants pacman to be. Too small a region and pacman will carelessly move close to ghosts and risk death, whereas too large a region may cause pacman to focus too much on running from ghosts and result in slow food consumption. This ghost region is 4 steps in any direction

Ghost	GR1 (90%)	GR2 (70%)	GR3 (50%)	GR4 (20%)
-10	-9	-7	-5	-2

TABLE III: Ghost Region (GR) reward spread

around a ghost for the `mediumClassic` map and 2 steps in any direction around a ghost for the `smallGrid` map. Table III shows the gradual decrease in negative reward assigned the the various regions around the ghosts. The smaller ghost region of 2 steps is more suitable for the smaller map since a region of 4 steps in any direction would lead to it covering too large an area of the map.

Rather than finding all the points within a certain Manhattan distance from a ghost, using this number of *possible* steps method is far more efficient and less crude. When using a simple Manhattan distance calculation, pacman would tend to run from ghosts even if it were not possible for the ghost to reach him within 6 or 7 moves, leading to pacman fleeing unnecessarily.

Another important aspect to note is how pacman should view capsules. This ultimately comes down to what type of agent one wishes pacman to be. For this design, pacman is not interested in the final score of the game therefore there is no benefit in hunting ghosts and eating them. Hence the reward for capsules is specified to the same value as food. In other words, pacman is indifferent about eating food or a capsule. Early implementations of this solver instructed pacman to prioritise capsules however this resulted in pacman becoming quite careless and dying.

III. Code Breakdown

Figure 2 describes the functional flow of the `MDPAgent`. At the beginning of every step in the game, pacman checks the size of the map which he is playing in order to decide what reward specification he should assign. An initial utility map `u_map` is created in the form of dictionary. The map which is built is a dictionary of key value pairs in which the keys are all the possible coordinates of the two dimensional map and the values are the utilities associated with each coordinate.

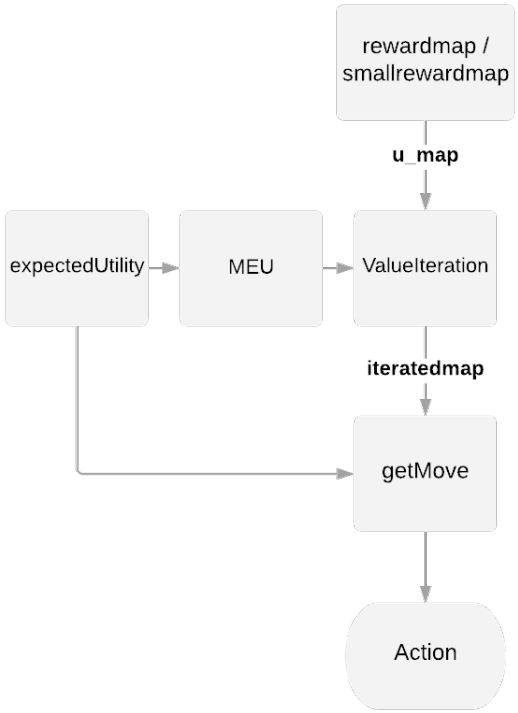


Fig. 2: MDP Algorithm flowchart

Initially, this utility map `u_map` is assigned as the rewards for the corresponding coordinates. Therefore, depending on what size of layout pacman is playing, it is the either the `rewardmap` or `smallrewardmap` function which actually creates the map for pacman.

This `u_map` is then fed to the `ValueIteration` algorithm to be updated. The `MEU` function which is used within the value iteration algorithm calculates the maximum expected utility for each coordinate via use of the `expectedUtility` function. Through this process the original `u_map` is updated until convergence is viewed to have been achieved. As for specifying the stopping criterion for the Bellman update, (Russell & Norvig 2010) prove that if the update is small

between iterations (i.e., no state's utility changes by much), then the error, compared with the true utility function, also is small. More precisely:

$$\|U_{i+1} - U_i\| < \frac{\epsilon(1-\gamma)}{\gamma} \implies \|U_{i+1} - U\| < \epsilon$$

Where U is the true utility function. This is the termination condition used in the `ValueIteration` algorithm in which ϵ is assigned a tolerance value of 0.01.

Finally, using this `iteratedmap`, pacman again calculates the expected utilities of every possible move and returns the move corresponding to the maximum expected utility via the `getMove` function.

This code is particularly efficient in the fact that it uses the `rewardmap` function to create both the initial utility map structure and the reward map itself. Initial versions of the `MDPagent` used separate functions for these processes however since initial utilities are arbitrary, we choose to assign them as the reward values and as result we can make use of the same function twice.

IV. Parameter Tuning & Results

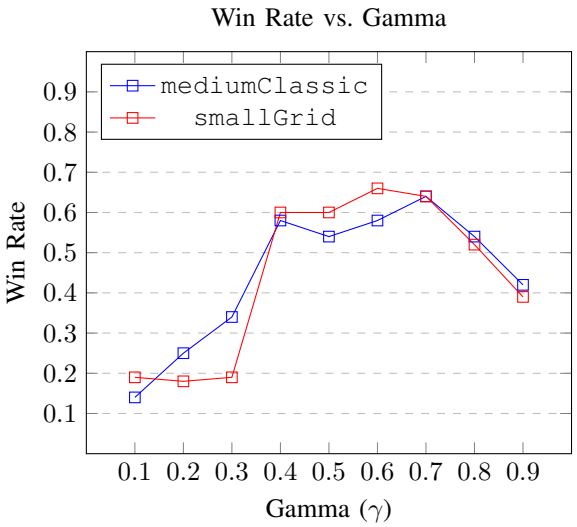
A. Gamma Tuning

To tune gamma, 100 games of `mediumClassic` and 500 `smallGrid` games were run for gamma values ranging from 0.1 to 0.9 while rewards were held constant as specified in Section II. Initial gamma testing revealed that for `mediumClassic`, a gamma value of 0.7 was most successful with a win rate of 64% whereas for `smallGrid`, a gamma value of 0.6 was most successful with a 66% win rate.

This similarity in gamma performance across both maps is relatively surprising given that one might have expecteed lower gamma values to be more effective on `smallGrid` due to the fact that gamma dictates how much the agent values future vs. immediate reward and since the map is much smaller, there could be less of an emphasis needed on future rewards. However for `mediumClassic` results were as expected, showing that suitably high gamma values produce optimal performance. In the larger map pacman may not always be near to a piece of food and higher gamma value results in better spread of the food utility which enables pacman to find it even when food is scarce or far from him.

It is also interesting to note that for both layouts, once gamma reaches approximately 0.7, the win rate decreases rapidly the more gamma increases. This could be explained by the fact that large gamma values result in slower convergence rates and thus hindering the agents performance.

It is important to note that due to the non deterministic nature of pacman, the same gamma value will yield different results for every test. However with a well designed system, one can limit this variation in results to an acceptable amount.



B. Reward Tuning

Reward tuning was performed through a process of elimination procedure until one could narrow down an effective configuration. This process is an efficient and non time consuming way of optimising parameters as one does not waste time running large amounts of tests on poor performing parameters (i.e. redundant tests). In order to reduce computation time, the gamma and reward parameters were tuned independently. The reward assignments were tuned using the optimal gamma values of 0.7 for `mediumClassic` and 0.6 for `smallGrid`. Table III illustrates the various reward assignments which were initially tested by running 25 `mediumClassic` games and 500 `smallGrid` games for each configuration.

Food	Ghost	`mediumClassic` Win Rate (25 games)	`smallGrid` Win Rate (500 games)
10	-10	0.48	0.66
10	-20	**0.60**	0.58
10	-50	**0.68**	0.63
20	-10	0.48	**0.68**
20	-50	**0.64**	0.64
50	-10	0.48	0.66
50	-20	0.44	0.65

TABLE IV: Food/Ghost reward testing

1) mediumClassic: Immediately a pattern can be recognised in the results above. The configurations which consistently perform better are those in which ghosts have strong negative values compared to food. With this knowledge, further games were ran under the following configurations to explore: Table V confirms that large negative values for the ghosts yield optimal results. A further 50 games were ran on the three configurations highlighted in bold in Table V. The optimal rewards were found to be 50 for food and -150 for ghosts.

2) smallGrid: Viewing Table IV of the initial reward testing for `smallGrid`, it is clear that this layout is more sensitive to the randomness and non deterministic nature of pacman.

Food	Ghost	mediumClassic Win Rate (25 games)
10	-100	0.60
20	-100	0.60
50	-100	**0.68**
10	-150	**0.68**
20	-150	0.64
50	-150	**0.68**
10	-200	0.56
20	-200	0.68
50	-200	0.56

TABLE V: Testing more extreme reward assignments

Performance is pretty level across the board and there are no clear patterns or trends are visible. The highest performing configuration was that for which food was set to 20 and ghosts -10.

V. Optimal Parameter Evaluation

Finally, under the following optimal configurations, 200 `mediumClassic` games and 1000 `smallGrid` games were ran to evaluate their performance.

Optimal Configurations

	Gamma	Food	Ghost	Win Rate
`mediumClassic`	0.7	50	-150	64%
`smallGrid`	0.6	20	-10	66%

The optimal configurations achieved a 64% win rate on `mediumClassic` and a 66% win rate on `smallGrid`. Due to the non-deterministic nature of the environment, it is wise to run several tests using these optimal configurations in order to get an idea of their win range over 25 games. Sets of 25 games were run repeatedly to check the mean number of games won on both layouts. Given the win rates of 64% and 66% calculated above, we expected these to be at around 16 wins for both maps. The following frequency distributions shows the results from running 100 rounds of 25 games on both `mediumClassic` and `smallGrid`.

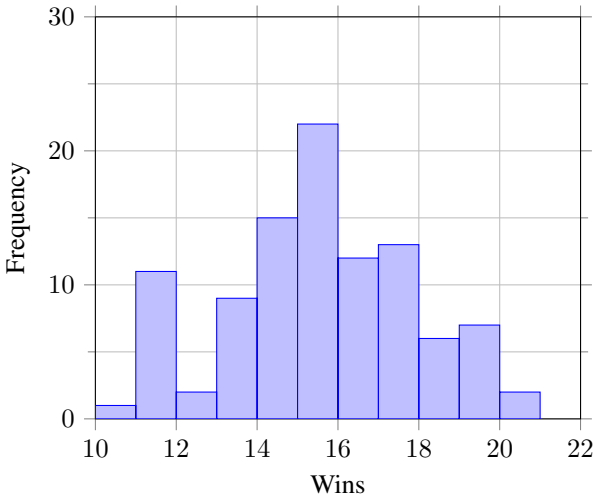
The mean wins observed in `mediumClassic` was 15. There was a standard deviation of 2.36 in number of the wins which is promising since the 95% confidence interval deduced from the data (10.4, 19.7) has a lower bound which is above the minimum threshold specified for judging the strength of the agent (10 wins).

As expected the mean wins of `smallGrid` is 16. The standard deviation is 2.3. One can deduce a 95% confidence interval for the number of wins achieved in 25 games. This interval is (11.6, 20.6) which similarly to `mediumClassic` has a relatively high spread of values. Further tuning and improvements are required to limit this variation however it was to be expected due to the nature of the environment.

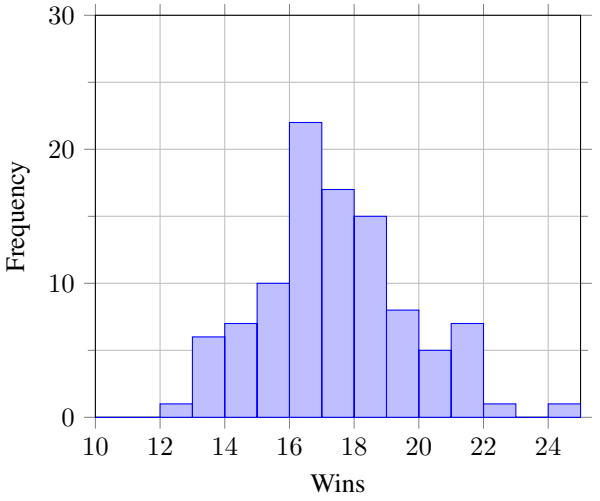
VI. Conclusion

Overall we have designed an efficient and successful generalised MDP agent which can operate on any size or style map layout and can win a respectable amount of games on both specific layouts in this report. If one wished to further improve this agent, a suggestion could be to further tune its specification of rewards, ghost region size or perhaps the reward assigned to empty coordinates. Due to time and computational constraints, not all of these factors could be considered entirely in this analysis. However, on the whole this agent is effective and manages well in its non deterministic environment.

Win frequency over 25 `mediumClassic` games



Win frequency over 25 `smallGrid` games



VII. References

- Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd. ed.). Prentice Hall Press, USA.