Modelling Markov Decision Processes for Solving Pac-Man in a Non-Deterministic, Fully Observable Environment with Random Agents

Oluwafemi Oladipo

I. Introduction

This paper evaluates the effectiveness of solving Pac-Man using solely a Markov Decision Process (MDP). Meaning Pac-Man's decision to make a move at any point (choose a policy) in time is solely determined by the best policy suggested by the MDP. This paper implements the MDP in python (version 2.7), with all the logic pertaining to the MDP contained within a single class MDPAgent. Finally, a few helper classes are also created to help represent the environment while keeping the code clean[4] and DRY[3].

II. Environment

Representing the environment was the first priority in order to implement an efficient MDP solver. I achieved this using three classes, a Grid Class, a Point Class and a States Class.

A. States Class

Enum of states for the point class, being: empty, pacman, food, capsule, ghost_hostile, ghost_edible, ghost_neighbour. Going forwards all three ghost states will be referred to as ghost for simplicity. However, whenever a specific sub category needs to be referenced its full name will be used.

B. Grid Class

Contains a representation of the 2D world Pac-Man operates in.

C. Point Class

Represents a single position on the grid. Each point has a type representing it's state, a utility representing it's current utility value, and a dynamic reward.

III. Markov Decision Process

The MDP in this paper is modelled using the bellman update[] formula, because simultaneous nature of the standard bellman equation proves unwieldy to solve. The MDPAgent class has various methods that help achieve different all the calculations, figure 1 depicts the hierarchy of calls.

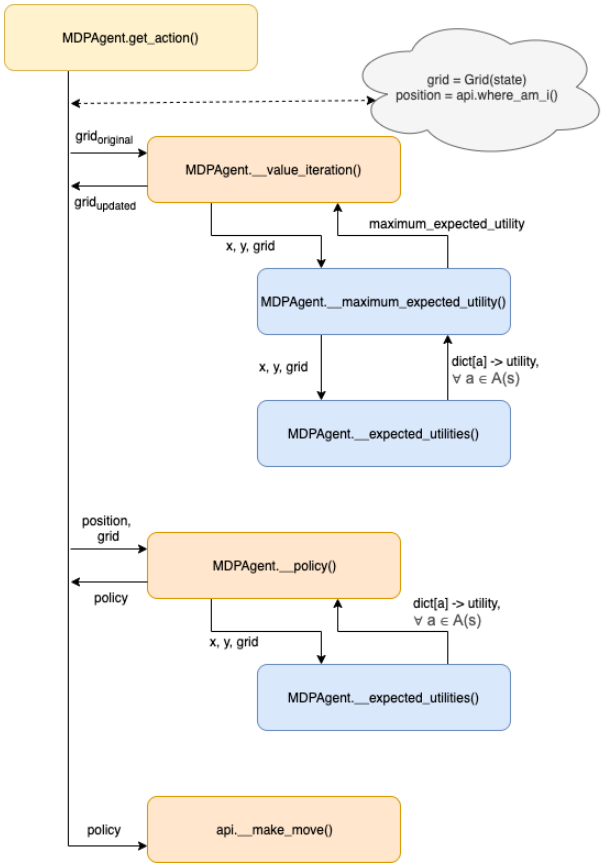


Fig. 1. $MDPAgent.get\_action()$ call stack.

- **MDPAgent.get_action()** - creates a new grid object from the current state, and finds pacman current position. Then uses this information to call subsequent methods, to; perform

value iteration on the grid, and find the optimum policy on the updated grid from current position.

- **MDPAgent.__value_iteration()** - performs the value iteration algorithm[1]. It also applies the bellman update formula at each point.

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \epsilon A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

The reward value $R(s)$ is calculated in the Point class. With the remaining of the formula being calculated in MDPAgent.__maximum_expected_utility().

- **MDPAgent.__policy()** - Finds the optimum policy for the next move. Using the following formula:

$$\pi_{i+1} = \operatorname*{argmax}_{a \epsilon A(s)} \sum_{s'} P(s'|s,a) U(s')$$

Policy however only performs the argmax operation, while MDPAgent.__expected_utilities() performs the calculation to find the scaled probability of the utility of each next state.

- **MDPAgent.__maximum_expected_utility()** - Calculates the maximum expected utility at a point by taking the maximum utility of its surrounding points. Using the following formula:

$$\max_{a \epsilon A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

However, only the max operation is performed here, the rest of the formula is performed in MDPAgent.__expected_utilities().

- **MDPAgent.__expected_utilities()** - Creates a dictionary mapping all possible policies to their respective utilities.

$$\operatorname{dict}[\mathrm{a}] \leftarrow \sum_{s'} P(s'|s,a) U_i(s')$$

IV. Methodology & Creativity

A. Code Implementation

Various design choices have been made in order to make the code a readable, and flexible as possible. This subsection details a few of these decisions.

1) Effective Use of Data Models: The grid class makes effective use of three python data model methods, __getitem__, __iter__ and __contains__.

__getitem__ is used so that points on instances of the grid class can be indexed by:

```
g = Grid()
point = g[x,y]
```

As opposed to, defining a get method to be used as:

```
g = Grid()
point = g.get(x,y)
```

In addition, with the absence of a __setitem__ method, this prevents inadvertent assigning values to points in the grid. Meaning that following is invalid

```
g = Grid()
g[x,y] = N
```

Which would invalidate the data structure, as all coordinates should index an instance of the point class and not arbitrary values. Or worse defining, a set method to be used as:

```
g = Grid()
g.set(x,y,N)
```

Which would in validate the data structure, and create a lengthy syntax to be remembered. Instead, points can be updated only using the syntax:

```
g = Grid()
g[x,y].utility = N
```

__iter__ is used to iterate through the underlying coordinates, and points on the grid. In addition, this allows easy iteration through all points in the value iteration algorithm. While also making it easier to skip wall coordinates by omitting them in the grid.

__contains__ is used to check if a coordinate is valid grid non-wall grid position.

2) Adhering to PEP8: The base project is written using primarily camel casing, however python standards (PEP8) dictates that snake casing should be used. In order to interface with the base project, while using snaking casing, a series of function were created that aliased all imported functions as their a snake cased variant. They were then called on the imports:

```
snake_case(util)

snake_case(api)
```

Allowing the use of functions in the form:

```
api.legal_actions(state)

util.manhattan_distance(xy1, xy2)
```

Then all the methods in MDPAgent claas were aliased as their camel case variant, using a decorator.

```
@camel_case
class MDPAgent(Agent):
```

Allowing core methods to be defined as:

```
def get_action(state):

def register_initial_state(state):
```

B. Strategy

Pac-Man makes his decisions using solely the output of the MDP model. However, adjusting parts of the input has shown to improve the performance (win rate) of Pac-Man overall. In addition, some optimisations have also been made in terms of implementation in order to bring down the running time of the algorithm.

1) Dynamic Reward Shaping: Reward shaping is process by a reward $r(s)$, is altered in order to cause faster convergence or better performance[2]. The point class handles the reward calculation and does so by employing subclass of reward shaping, dynamic reward shaping. That is, it employs a formula in each state to the final resulting reward. In the code accompanying this paper, this is done in the reward @property field in the point class.

$$R(s) = \begin{cases} food, capsule & r(s) \cdot f_\phi \div f_\Delta \\ ghost, empty, pacman & r(s) \cdot f_\Delta \end{cases}$$

The cell five states: food, capsules, ghost, pacman and empty. Each cell state has a default reward value $r(s)$. For the purposes of calculations, the default reward values are in two classes $+ve$ and $-ve$ values. This paper assigns $+ve$ default reward values to food & capsules only, and $-ve$ default reward values to everything else (ghosts, pacman and empty cells). The goal is then to make the resulting reward $R(s)$, either more or less $+ve$ or $-ve$ compared to the original $r(s)$.

Positive Rewards

$$R(s) = \frac{r(s) \cdot f_\phi}{f_\Delta}$$

where

s ϵ {food, capsule}

In the positive default reward value class (food & capsules) the goal is for the values to become less positive the closer they are to ghosts and more positive the further they are away. To ensure this default reward $r(s)$ is divided by a value between 1 and e (f_Δ). Where again items closer to ghosts are divided by values closer to e, therefore when divided the resulting value is less positive and less favourable. In addition, there is a competing multiplier (f_ϕ) that increases foods reward value when there is less food available. The goal here is make Pac-Man seek out food more aggressively near the end of the game when food items are few and sparse.

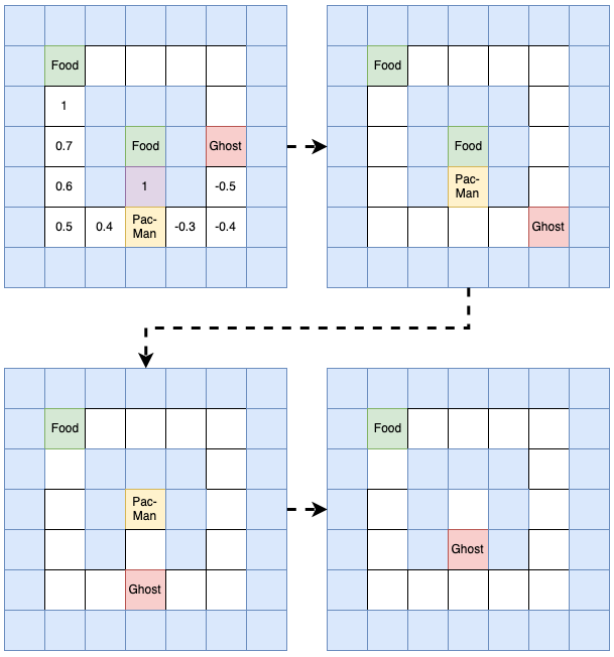


Fig. 2. Diagram depicting optimum utility is north (purple), from the starting position. Then, how the resulting sequence of actions lead to Pac-Man's ultimate death.

Figure 2 shows how scaling the reward value food items can change optimum policy for the

better. In this case, if Pac-Man goes directly north as suggest by the optimum policy (purple) he will often die in the sequence of events the figure shows. However, by employing the scaling technique, considering distance to ghosts, Pac-Man's optimum policy changes as shown in figure 3. In this scenario, if Pac-Man later goes for the food in the centre he'll survive as the game would have ended. The high level reasoning for this is that places nearer to a ghost are more risky, however when there are fewer pieces remaining the risk is decreased as Pac-Man should hopefully complete the game before being eaten.

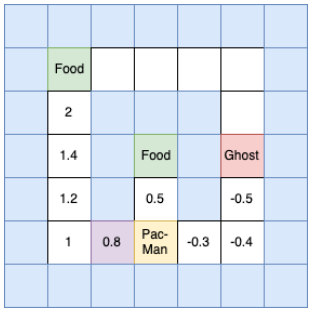


Fig. 3. Diagram depicting optimum policy (purple), is now east.

Negative Rewards

$$R(s) = r(s) \cdot f_{\Delta}$$

where

s ϵ {empty, ghost, pacman}

The default reward of an empty cell is -0.04. However if this cell is near a ghost, then Pac-Man should avoid this cell more than an empty cell far away from any ghosts. In this case, the value needs to be more negative the closer it is to a ghost and less negative the further away. In order to achieve this the default reward is multiplied by a scalar between 1 and e (f_{Δ}), with empty cells closer to a ghost being multiplied by values closer to e. Hence making them more negative and less favourable. Likewise, the same happens for all other default rewards in the $-ve$ value class.

Peaking inside the functions

Each function f_{Δ} and f_{ϕ} rely on a constant C derived from the environment. They then model the environment by raising e to the power of C (e^C).

$$f_{\Delta} = e^{\frac{D-d}{D}}$$

where:

D	is maximum manhattan distance of two points on grid
d	is minimum manhattan distance between point and any ghost

Starting with the fraction, the top part $D - d$ will give values between 0 and D. Therefore when divided by D the result will between 0 and 1. Tending to 0 the closer d is to D (the point is far from any ghost), tending to D when d is nearly 0 (the point is adjacent to a ghost). Resulting in f_{Δ} tending to e when point is far from any ghost, and tending to 1 when the point is adjacent to a ghost.

$$f_{\phi} = e^{\frac{s}{S}}$$

where:

S	is the total number of spaces on the grid
s	is the numbers of spaces that are empty

$\frac{s}{S}$ will range between 0 and 1 inclusive. Tending to 0 when when s is close to 0 (there is a lot of food remaining), tending to 1 when s is close to S (there is hardly any food remaining). f_{ϕ} will range between 1 and e, tending to 1 when there is a lot of food remaining and tending to e when there is hardly any food remaining.

2) Variable Discount Factor:

$$\gamma(x) = A + \frac{K - A}{1 + e^{-B(x-M)}}$$

where:

K	is the upper asymptote (1)
A	is the lower asymptote (0.5)
B	is the growth rate from left to right (-0.1)
M	is the midpoint of the growth area (5)
x	is the number of remaining food + capsules

In this environment the game ends either when, all the pieces are eaten or Pac-Man is eaten by a ghost. With the favourable of the two states

being that all pieces are eaten. Therefore all pieces being eaten can be conceptualised as the goal state. Assuming this, initially the grid has a near infinite horizon[6], so additive rewards will be inadmissible and discounted rewards are required. However the closer we are to the end (the fewer food pieces remaining) the more definite the route to the goal state. At the extreme, when there is one food piece remaining meaning we can use actual additive rewards. As such gamma has been modelled to reflect this. Varying between 0.6 and 1, increasing to values closer to one with fewer food pieces remaining. In order to model this shape Richard's Curve[5], a flexible/malleable sigmoid curve has been used.

3) Terminal States: Since in this environment the game ends either when, all the pieces are eaten or Pac-Man is eaten by a ghost. The ghosts are therefore also an end state, however an undesirable one. So similar to increasing gamma to more resemble additive rewards for fewer and fewer food items. Ghosts are treated as terminal states, meaning there utility values never change. This has the effect of creating a well around them, ensuring the surrounding values are negative. In addition it ensures the the ghosts utility value doesn't become positive when surrounded by only food. [1].

4) Convergence:

$$\lim_{iteration} = 2 * \left\lceil \sqrt{height * width} \right\rceil$$

The number of iterations used to converge the utility values has a big affect on the success of the MDP. However, although generally increasing the number of iterations leads to higher success the running time of the program also increases and very quickly becomes unwieldy. In addition, through testing larger boards require more iterations. This is because, with more points on the board values have a further distance to propagate to.

5) Hyper-Parameters: To increase the ability for the utility values to accurately model the environment, ghost neighbours were introduced. These points have a negative default reward value ($r(s) \in \mathbb{R}^-$), which is less than that of an empty space. This was model by setting a maximum manhattan distance (radius) around any ghosts and point within were deemed ghost neighbours, except if they were walls or other ghosts. In addition, as ghosts move twice as fast as Pac-Man when they are edible the could become hostile again and eat Pac-Man quickly. However, to avoid this Pac-Man avoids ghosts when their remaining timer is less than a set value.

V. Results

To come up with the final version of MDPAgent, various design decisions had to be made such as; whether or not to use a variable discount factor, whether or not to use terminal states and which states should they be, and how to design the dynamic reward shaping functions, all of this in addition to initialising various hyper-parameters. All of which will be discussed in this section, with statistical evidence to back up decisions made. All experiments were repeated 10 times, with 10 iterations in each run. The statically relevant stat, used is then the average of all 100 iterations. For reference hence forth performance will, will refer to as the win percentage out of 100 iterations. Finally, most experiments have been conducted on both the smallGrid and mediumClassic layouts, however the grid/s used are always stated.

A. Dynamic Reward Shaping

To test the effect of including dynamic reward shaping, the first step was to test the condition that inspired it's creation. As detailed in methodology, figure 2 shows the starting position to which a high proportion of losses on the small grid were attributed to. Therefore a new starting layout was created in the this configuration which will be named smallGridCustom.

TABLE I
Dynamic Reward Shaping on smallGridCustom

Dynamic Reward Shaping	Win Rate (%)
	smallGridCustom
True	43
False	25

This demonstrated dynamic reward shaping works in the niche that inspired it, however testing what was then required was to show whether this performance increase continues in other situations.

This demonstrates the general usefulness of dynamic reward shaping and that they increased

TABLE II

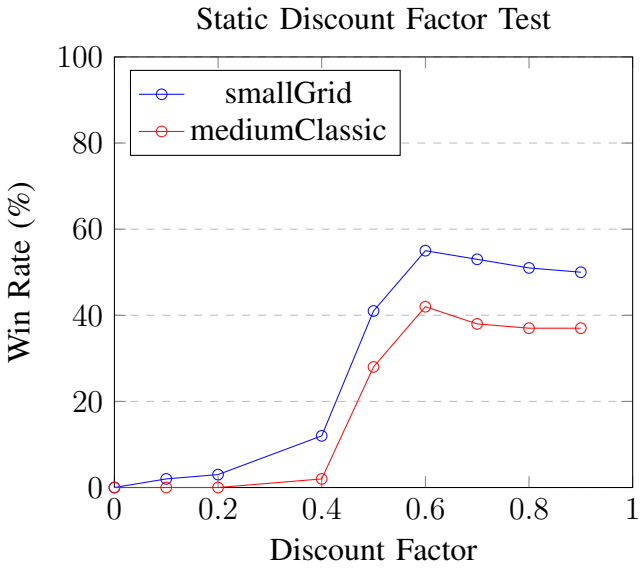
DYNAMIC REWARD SHAPING TEST

Dynamic Reward Shaping	Win Rate (%)	
	smallGrid	mediumClassic
True	57	38
False	41	18

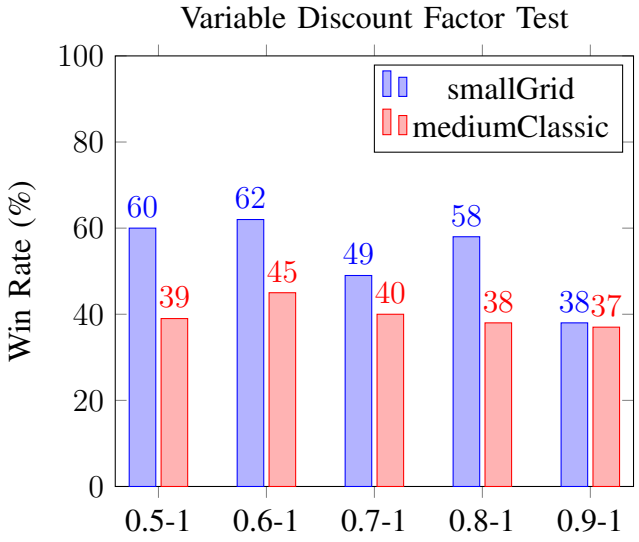
overall performance. This method was therefore used in the final model.

B. Variable Discount Factor

A range of discount factor were tested used on both grids, from 0.1 to 1.



The results showed a discount factor in the open interval (0.5, 1) were the most successful. However, smallGrid and mediumClassic had different peaks. This was due to them starting at different distances away from their end state (see section IV.B Variable Discount Factor). As such a final setting varying gamma, dependant on the remaining number of pieces, was tested. Always setting the upper bound to 1 (additive rewards), and tested different lower bounds within the optimal range.



The results showed varying gamma, always lead to higher performance than any static gamma value. How it also showed varying gamma in the open interval (0.6, 1) was the most optimal.

C. Terminal States

The aim of using terminal states was to reduce running time, while not affecting performance.

TABLE III

TERMINAL STATES TEST

Terminal States	Win Rate (%)	
	smallGrid	mediumClassic
none	55	34
ghosts only	57	39
food only	35	18
ghosts & food	40	22

TABLE IV

TERMINAL STATES (RUNNING TIME) TEST

Terminal States	Running Time (seconds)	
	smallGrid	mediumClassic
none	3	320
ghosts only	3	270
food only	2	250
ghosts & food	1	200

The results showed using only ghosts as terminal states was the most productive, when considering both running time and performance. As this was the only setting to reduce running time, without affecting performance.

D. Hyper-Parameters

Two hyper-parameters (ghost radius, and ghost safe time) were tested, with varied values against both layouts in-order to find their respective optimal values.

Ghost Radius - is the manahattan distance around ghosts that Pac-Man should avoid. The results showed having settings on smallGrid and mediumClassic would be . most optimal, having the values 0 and 3 respectively.

TABLE V

TABLE TO TEST CAPTIONS AND LABELS

Ghost Radius	Win Rate (%)	
	smallGrid	mediumClassic
0	55	25
1	54	25
2	34	38
3	0	42
4	0	40
5	0	20

Ghost Safe Time - is amount of time remaining, whereby edible ghosts should be consider as hostile as regular ghosts. This was only tested on the mediumClassic layout as smallGrid had no capsules, so ghosts could never become edible. The results showed a time of 3 was as optimal as anyway great time, therefore this was used in-order to keep the restiction low.

TABLE VI

TABLE TO TEST CAPTIONS AND LABELS

Ghost Safe Time	Win Rate (%)
	mediumClassic
0	25
1	25
2	24
3	41
4	40
5	41

VI. Conclusion

The the non-deterministic fact about the environment played a big factor in the final performance, as testing with the same final configuration but turning of the non-deterministic effect yielded a performance of 100% and 85% in smallGrid and mediumClassic respectively. Where as with the non-deterministic effect turned on yielded a performance of 60& and 40% on smallGrid and mediumClassic respectively. Overall, this demonstrates that an MDP model can accurately select the best policy in a Non-Deterministic, Fully Observable Environment with Random Agents. However, further research how can be done to see if combining this with ad-hoc heuristics such as avoiding ghosts at the last minute, and planning the shortest path to a food item will further increase the performance.

References

[1] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[2] Kyriakos Efthymiadis and Daniel Kudenko. A comparison of plan-based and abstract mdp reward shaping. *Connection Science*, 26(1):85–99, 2014.

[3] Deepak Karanth. Is your code dry or wet? - dzone devops, Apr 2016.

[4] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[5] F. J. Richards. A Flexible Growth Function for Empirical Use. *Journal of Experimental Botany*, 10(2):290–301, 06 1959.

[6] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.