

Implementation and Analysis of an MDP agent for a stochastic version of Pac-Man

Danilo Del Busso
1745819

Department of Informatics, Kings College London
Strand, London, WC2R 2LS, United Kingdom

Introduction

This work shows the implementation and statistical analysis of an AI agent capable of winning the arcade game of Pac-Man using an MDP solver that follows a policy based on Value Iteration.

The Game

Pac-man (or Pacman) is a video game developed by Namco in 1980 for arcades. The player needs to control a yellow spherical creature, called Pac-Man, making it eat all the numerous circles spread across the labyrinth (food), while avoiding four coloured ghosts. If Pac-Man finds itself in the same cell as one of the ghosts, it loses the game. It is also able to eat special power pills (capsules) that temporarily allow him to eat ghosts and remove the losing states. In order to win the game, the player must make Pac-Man eat all the food present in the labyrinth.

Description of the Problem

The agent described in this work is constrained by the aforementioned rules and winning conditions. The game itself is also modelled as a stochastic variation of the Pac-Man game, meaning that some transitions are probabilistic. In the context of the Pac-Man game, the agent has an 80% probability of going in the direction specified by the policy, and a 10% change of going to either direction perpendicular to that (Fig. 1). If the agent hits a wall, it will not move.

To better reason about the problem we can identify it as being a Markovian Decision Process (MDP), which describes the problem mathematically as:

- A set of states $s \in S$ with an initial state s_0
- A set of actions $A(s)$ in each state
- A transition model $P(s'|s, a)$, which, given the payoff of a state and the action that is applied to it, describes the probability distribution of the next state
- A reward function $R(s)$, that returns a reward for a given state

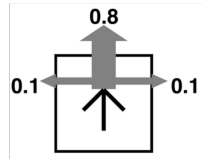


Figure 1: Diagram showing the probabilistic dependency of Pac-Man's movement

Since the game actions are not deterministic, the probability that a solution (policy) π will be applied as intended after n steps is 0.8^n , which results in exponentially smaller probabilities as the number of steps needed to reach the goal state increases. This is the reason that an optimal policy π^* , described as a sequence of actions, needs to consider the stochastic nature of the actions available to the agent.

Another consideration worth mentioning is that utilities in Pac-Man are non stationary, since the rewards change depending on the overall state of the game. However, for the purposes of this work, we will only consider some of the rewards as mutable, to simplify the result. An extension of the agent implementation as described in this work could start by removing this artificial restriction.

Solution

The Bellman Approach In order to find a reasonably efficient policy π for the MDP described in the previous section, the agent will compute the utility of each state using the Bellman equation (BELLMAN 1957):

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s')$$

where $U(s)$ is the utility for the given state, $R(s)$ is the reward function, γ is the discount factor, $P(s'|s, a)$ is the transition model, $A(s)$ is the set of possible actions, and s' is the previous state.

The solution takes the action a with the maximum expected utility as the action to be added to π .

Bellman's Inefficiency and Value Iteration The method above will solve for n states a set of equations with n variables. This will run into performance issues, reason being that the use of the \max operation makes the system non-linear and therefore inefficient.

In order to solve this issue, the solution uses an iterative approach, called Value Iteration (BELLMAN 1957). This approach starts by setting arbitrary values for each state and then applies the following value update to all states in parallel:

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

This assignment operation is applied until the value of $U_{i+1}(s)$ converges.

Implementation

Environment and Documentation

The implementation of the solution was done using the standard Python 2.7 distribution, and is fully found in the *mdpAgents.py* file.

All the functions used are documented using the Epytext Markup Language (Loper 2008).

High Level Description

The implementation of the solution was written using pure functions, in order to keep the logic compartmentalised and easy to debug.

Every function but the `registerInitialState` and `getAction` methods are pure.

The main issue with this approach, however, is the fact that calling functions of the *api* module results in significantly slow running times. For this reason, the calls to this module are kept to a minimum, instead opting for an approach similar to the dependency injection pattern used in object oriented programming in order to access needed information about the state.

Constants Early on in the development phase, it was clear that some variables used in the implementation had to be constant and initially defined arbitrarily. These constants are used to calculate the utility values and are the main object of analysis of the Analysis section of this work.

Rewards The implementation of the solution for this project starts by generating a Reward Map, defined as a bi-dimensional array with the same dimensions as the map Pac-Man navigates through. Additionally, it contains `None` values at the coordinates corresponding to in-game walls, and numeric values for all navigable cells. These numeric values are the result of the reward function $R(s)$ for a given state s .


None	None	None	None	None	None	None
None	-0.04	-0.04	-0.04	-0.04	-100	None
None	-0.04	None	None	None	-0.04	None
None	-0.04	None		10	-0.04	None
None	-0.04	None	None	None	-0.04	None
None	-0.04	-0.04	-0.04	-0.04	-0.04	None
None	None	None	None	None	None	None

Figure 2: A visualisation of a Reward Map for the *small-Grid* map, showing the values applied as rewards for each cell. -100 for the presence of a ghost, 10 for the presence of food, `None` for walls, and -0.04 for empty cells. The value assigned to empty cells is negative in order to push the agent to avoid empty cells and solve the MDP quicker.

The `reward_map` function is used to generate the Reward Map as described above. In order to perform this action, `reward_map` uses the *api* module to retrieve the positions of Pac-Man, the walls, the food, the capsules, and the ghosts on the map. It then assigns to each cell of the Reward Map its reward value.

Updating the Reward Map Initial exploratory tests of the implementation of the reward function as described above were found to be insufficient to reach the goal state in maps with the presence of ghost agents. The *malus* resulting from the presence of a ghost in a given cell would only allow the agent to react to ghosts that were at most two cells away from its position. This resulted in a very low win rate. The Pac-Man agent would try to follow the actions given to it, but the stochastic nature of the game made it so that it would occasionally fail to follow the best action given by the policy, and occasionally stop in its tracks when hitting a wall. This caused the ghost to be able to catch up to Pac-Man, ending the game in a loss.

In order to mitigate this issue, the reward function is updated after the initial values have been set, to take into consideration the relative distance of ghosts from Pac-Man.

After the initial values of the rewards have been assigned to the corresponding cells, the `reward_map` function passes the Reward Map and the position of Pac-Man to `add_ghost_distance_rewards`.

This function uses a queue based frontier algorithm (Red-BlobGames 2014) to find the cells closest to the agent. While exploring the frontier, the function keeps track of how many adjacent cells have been visited. If a ghost is found, all the cells that have been explored up until that point receive a reward *malus*, as specified by a constant. The closest a specific cell is to a ghost, the greater the *malus* that is applied to the reward value of the cell is.

The main issue that arose after applying this version of the `add_ghost_distance_rewards` was that an unbounded frontier would result in the agent concentrating its efforts into avoiding ghosts at all times, rather than to also collecting food and win the game, since the *malus* applied by the updated Reward Map to cells close to the ghosts was much greater than the *bonus* that cells with food received.

The final iteration of the `add_ghost_distance_rewards` function circumvents this issue by limiting the frontier to a specific maximum distance. This distance is expressed in terms of number of cells explored by the frontier algorithm and depends on a constant, `DANGER_ZONE_RATIO`. The number of cells n_{max} that the function visits before interrupting the frontier exploration is expressed as:

$$n_{max} = \frac{m_h * m_w}{R}$$

where m_h and m_w are the height and width of the map respectively, and R is the `DANGER_ZONE_RATIO`.

This solution allowed the agent to reach its goal of eating all the food on the map without being constantly affected by the vicinity of the ghosts. The choice of using a ratio instead of a specific constant for n_{max} is that this method allows

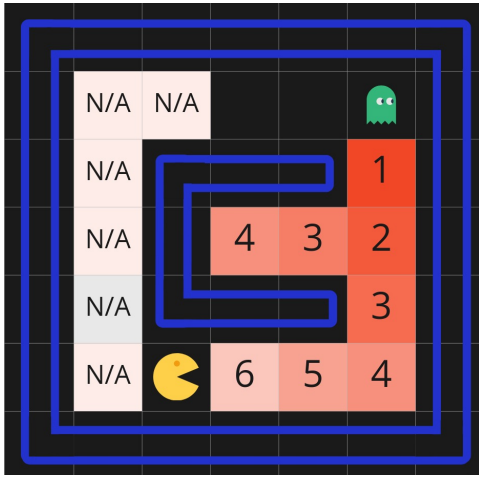


Figure 3: A visualisation of the implementation of the frontier algorithm in *smallGrid*. It shows its influence on the value of rewards for each cell. The redder a cell is, the more *malus* is applied to it. The numbers show the distance from the ghost, for which the *malus* is divided by

one constant to be used for both maps used in the testing phase, *smallGrid* and *mediumClassic*, without implementing *ad hoc* logic.

Bellman Assignment and Value Iteration After obtaining the Reward Map, the implementation of the solution generates the new utility values by using the Bellman Assignment as described in the Bellman’s Inefficiency and Value Iteration section.

The transition model $P(s'|s, a)$ is based on the known stochastic consequences of the agent’s actions, as described in the Description of the Problem section.

Using the utility map of the current state s' , the `bellman` function is applied to each cell on the map from left to right, top to bottom. This function returns the updated utility value $U_{i+1}(s)$. This value is inserted into an empty copy of the map, generated using the `get_empty_map` function.

The `bellman` function is also given the cell’s reward value, based on the Reward Map as described in the two previous sections.

The process of calculating the result of the Bellman equation as described above is repeated a certain amount of times, equal to value of the `ITERATIONS` constant. The choice of using a constant instead of waiting for the value of $U_{i+1}(s)$ to converge was made because relying on convergence does not result in constant performance in terms of running time. Furthermore, the use of the `ITERATIONS` constant did not affect the win rate of the agent by a significant factor, and its variability was taken into consideration during the analysis phase of the work.

During each iteration $i \in \mathbb{N} : 0 < i \leq \text{ITERATIONS}$ of the `bellman` function, the values of $U_{i+1}(s)$ and $U_i(s)$ are stored in separate bi-dimensional arrays. This allows for the parallel calculations required by the use of Value Iteration.

Choosing an Action The result of the process of Value Iteration is stored in a bi-dimensional array and then used within the `getAction` method to tell the agent which action to perform.

In order to do this, the `getAction` method fetches the utility values of the four cells adjacent to Pac-Man. It then filters out the cells that Pac-Man is unable to reach. Finally, the agent is told to follow the action for which he would reach the cell with the highest utility value, as returned by the `value_iteration` function.

Analysis

Testing Environment

In order to run multiple concurrent test sessions, an extension to the *pacman.py* was written, allowing for the constants mentioned in the Constants section to be set using custom arguments. Table 1 shows all available constants, the arguments used to set them, and their default values.

Argument	Constant	Default
--ELR	EMPTY_LOCATION_REWARD	-0.04
--FR	FOOD_REWARD	10
--CR	CAPSULE_REWARD	100
--GR	GHOST_REWARD	-1000
--GA	GAMMA	0.3
--DZR	DANGER_ZONE_RATIO	3
--DG	DANGER	500
--IT	ITERATIONS	100

Table 1: All arguments that can be used to change the constants used to calculate the utility values

More information is available in the documentation for the extended version of *pacman.py* (Del Busso 2020). Further scripts were written in order to run the extended version of *pacman.py* with multiple values and collecting them in .csv files for analysis. An example of this can be found in the aforementioned repository, in the *iterations_medium.py* file, which runs the MDP agent with several values for the `ITERATIONS` constant and collects the data in a .csv file. These scripts were run on a virtual machine running on the [Google Cloud Platform](#). The used configuration was as shown in Table 2.

Resource	Configuration
Processor	4 Intel Xeon vCPUs
Memory	16 GB
Machine Type	c2-standard-4
Operating System	Debian GNU/Linux, 10 (buster)

Table 2: Configuration of the testing machine

Motivation & Goal

The default values as shown in Table 1 are somewhat arbitrary, and were found by trying a few initial values and manually adjusting them in order to increase the win rate. The

motivation for the analysis was to optimise the Constants used by the MDP agent in order to increase its win rate in the *smallGrid* and *mediumClassic* maps.

Optimising the Constants

In order to achieve the goal of increasing the agent's win rate, the default values were used as control values; additionally, scripts were written to calculate a statistically significant win rate when changing one constant individually. For instance, 10000 games were run using all the default values of the constants as shown in Table 1, except for the GAMMA constant. Every hundredth game, the win rate of the past one hundred games would be stored, and the GAMMA constant increased by 0.01. This resulted in statistically significant win rates for all the values of GAMMA between 0 and 1, with 0.01 increments. This was done for both the *smallGrid* and *mediumClassic* maps, resulting in a total of 20000 games, and 200 win rates to compare. Finally, the results were plotted in a line chart to visually spot sharp changes in the win rate. For the aforementioned example, the win rate for the *smallGrid* map showed a sharp increase in win rate for values of GAMMA above ca. 0.5, as shown in Fig. 4.

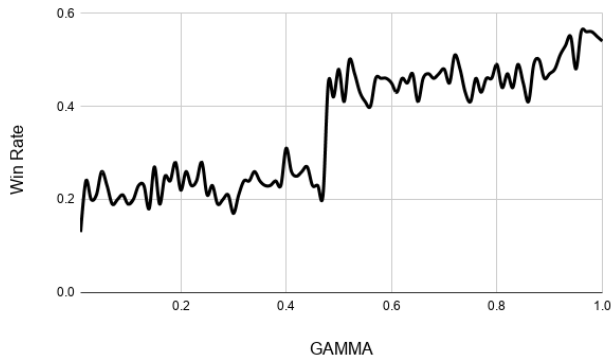


Figure 4: Change of the win rate vs GAMMA value in *smallGrid*

For the reasons described above, the value of GAMMA was set to 0.9.

Analysis of the Influence of Constants on the Win Rate

While all constants were evaluated as described in the previous section, this work will only report the ones which have significantly influenced the agent's win rate, as well as one which did not.

ITERATIONS The ITERATIONS constant was initially set to 100. However, initial exploratory tests showed that the utility values converged much earlier. Therefore, the processing time spent in the Value Iteration segment of the computation was much higher than needed.

In order to arrive at a good compromise between running time and number of iterations, the chart in Fig. 5 was produced using the method highlighted in the Optimising the Constants section.

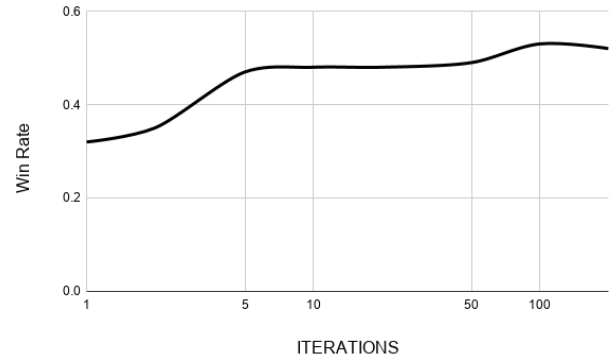


Figure 5: Change of the win rate vs ITERATION value in *smallGrid*

This chart shows a plateau that is reached in the win rate between the values 10 and 50. Therefore, the conclusion was made that 10 iterations of the Value Iteration method were sufficient to have a relatively high win rate. The chart produced for the *mediumClassic* map showed a similar pattern and is not shown.

DANGER_ZONE_RATIO The DANGER_ZONE_RATIO constant, as described in the Updating the Reward Map section, was crucial to increasing the win rate. While its initial value was 3, this proved to be inefficient when plotted on a chart. The ratio increased the win rate sharply up until reaching the value of 6, after which it stalled and flattened out. Plotting the win rate and the value of DANGER_ZONE_RATIO on a line chart with a logarithmic scale applied to the x axis (Fig. 6) shows that after the value of ca. 6 the DANGER_ZONE_RATIO does not influence the win rate in a significant manner. Therefore, the value for DANGER_ZONE_RATIO was set to 6.

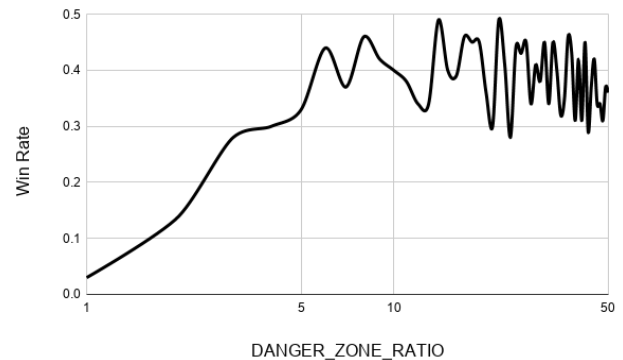


Figure 6: Change of the win rate vs DANGER_ZONE_RATIO value in *mediumClassic*

DANGER The analysis of the DANGER constant revealed that its influence on the win rate was minimal. Fig. 7 shows how its value plateaued since the start and does not show a

statistically significant trend with the win rate. This can be explained by the fact that the `DANGER` value is only used as the amount of *malus* to apply to the reward values of cells near ghosts. Any positive *malus* will discourage the agent from going towards a direction that is near a ghost, since it will always result in lower reward values than for cells that are not in the vicinity of a ghost. Therefore, the value has been set to an arbitrary 500.

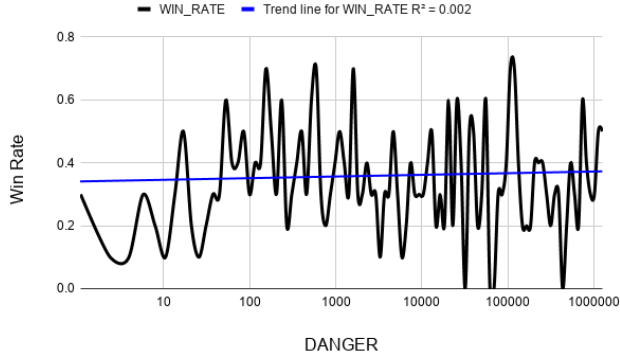


Figure 7: Change of the win rate vs `DANGER` value in *mediumClassic*

Results

After applying the results from the statistical analysis as shown in the Results & Analysis section, the constants as shown in Table 3 have been changed from the default values shown in Table 1 in order to achieve a higher win rate for the solution as described in this work.

Constant	Default Value	Updated Value
GAMMA	0.3	0.9
DANGER_ZONE_RATIO	3	6
ITERATIONS	100	10

Table 3: The default and updated values based on the performed analysis

To obtain a significant value for the win rate of the final configuration of constants, two runs of the game were run, with the following configurations and results:

Map	Number of Games	Win Rate
<i>mediumClassic</i>	100	0.42
<i>smallGrid</i>	100	0.53

References

- [BELLMAN 1957] BELLMAN, R. 1957. A markovian decision process. *Journal of Mathematics and Mechanics* 6(5):679–684.
- [Del Busso 2020] Del Busso, D. 2020. Pacman extension repository. <https://github.com/danilo-delbusso/pacman-cw-scripts>. [Online; accessed 13-Dec-2020].

[Loper 2008] Loper, E. 2008. The Epytext Markup Language. <http://epydoc.sourceforge.net/epytext.html>. [Online; accessed 13-Dec-2020].

[RedBlobGames 2014] RedBlobGames. 2014. Introduction to the A* Algorithm. <https://www.redblobgames.com/pathfinding/a-star/introduction.html>. [Online; accessed 13-Dec-2020].