Design of Pac-Man Strategies with Embedded Markov Decision Process in a Dynamic, Non-Deterministic, Fully Observable Environment

Jiachang (Ernest) Xu

***Abstract*— This paper iterates the design of Pac-Man strategies, whose decision-making protocol is solely based on Markov Decision Process, without the support of pathfinding algorithms nor heuristic functions, in a dynamic, non-deterministic, fully observable environment. This project provides the rare opportunity to refine the understanding of, and practice the application of Markov Decision Process in a classic arcade game setting. After significant number of hours of parameter tuning, my design achieved a win rate ranging between 50% and 60%. With the proven effectiveness of embedded Markov Decision Process, a spin-off Pac-Man AI project that incorporates the advantages of pathfinding algorithms, heuristic functions, and Markov Decision Process shall be on the agenda.**

I. Introduction

Imagine that you can see into the future. Would this be valuable for your decision-making process at the moment? My answer is "yes". With the embedded Markov Decision Process, the Pac-Man agent of my design, named `MDPAgent`, is able to predict the utility of taking an action in a dynamic, non-deterministic, fully observable environment. In other word, the `MDPAgent` quantifies the usefulness of moving to a neighbor location by adapting the Bellman's Equation to this project. This project refines my understanding and application of the Markov Decision Process, and value iteration of Bellman's Equation, and in turn proves the effectiveness of Markov Decision Process in decision-making tasks in a non-deterministic environment. This paper will provide a detailed discussion with visual aids about (1) properties of Pac-Man environment in Section II, (2) adaptation of Markov Decision Process to this project in Section III, (3) my design of strategies and workflow of `MDPAgent` in Section IV, and (4) how I used creativity to optimize and evaluate the effectiveness of my design in Section V.

II. Properties of Environment

This section will discuss the properties of the environment, dynamic, non-deterministic, and fully observable, that this project works within in the three following subsections.

A. Dynamic Environment

The environment of this project is dynamic, because the ghosts are always on the move. One major difference between the game state at timestamp t and $t+1$ is the location of the ghost. For example (Fig. 1), the ghost can moves either east or west.

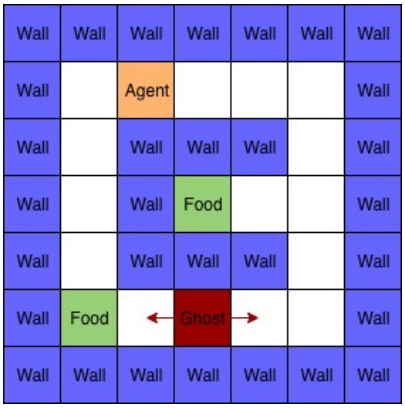

Fig. 1. Example of possible movements of the ghost

B. Non-Deterministic Environment

The environment of this project is non-deterministic, because the agent is not guaranteed to move in the direction in which it intends to move. More precisely, the probability for the agent to actually move in the direction in which it intends to move is 80%. There is a 10% probability each for the agent to move in the direction that is either left or right of its intended direction. Additionally, if the direction in which the agent actually moves leads to a wall, the agent stops at the original location.

C. Fully Observable Environment

The environment of this project is fully observable, because the version of `api.py` that this project works with provides information of every component in the maze.

III. Markov Decision Process

Markov Decision Process is the core technique that this project uses to design Pac-Man strategies in a dynamic, non-deterministic, fully observable environment. According to the book *Artificial Intelligence: A Modern Approach* (Third Edition) by Russel and Norvig, generalized Markov Decision Process consists of four key features:

- S: a set of states
- $A(s)$: a set of actions available at state s
- $\Pr(s'|s,a)$: a transition model that dictates that probability that action a from state s at timestamp t leads to state s' at timestamp $t+1$
- $R(s)$: a reward function

This section will discuss how this project adapts Bellman's Equation to the Pac-Man problem in Subsection III-A, and how to perform value iteration on Bellman's Equation

through Subsection III-B to III-D. The discussion of value iteration will include (1) how to initialize utility values for different components in the maze under different circumstances, (2) how to update utility values for all the non-convergent free spaces in the maze, and (3) definition of convergence.

A. Bellman's Equation

1) Generalized Bellman's Equation: Bellman's Equation is the center piece of the embedded Markov Decision Process in my design of Pac-Man strategies. The generalized Bellman's Equation (Formula 1) updates the utility value of state s based on the action a, which belongs to available actions $A(s)$ at state s and results in maximum expected utility. The discount factor γ dictates how much Bellman's Equation sees into the future, which is a parameter that can be fine-tuned to maximize win rate through experimentation (tested in Paragraph V-C.1).

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} \Pr(s'|s,a)U(s') \tag{1}$$

2) Abstract Adaption of Bellman's Equation: To adapt the generalized Bellman's Equation to the Pac-Man problem, Formula 2 updates the utility value for each non-convergent free-space location l at iteration i based on the maximum expected utility of its eastern, western, northern, and southern neighbors at iteration $i-1$. The eastern, western, northern, and southern neighbor locations of a location l are denoted as e_l, w_l, n_l, and s_l.

$$\begin{aligned} U_i(l) = R(l) + \gamma \max[&EU_{i-1}(e_l), \\ &EU_{i-1}(w_l), \\ &EU_{i-1}(n_l), \\ &EU_{i-1}(s_l)] \end{aligned} \tag{2}$$

3) Detailed Adaptation of Bellman's Equation: Because of the non-deterministic nature of the environment (defined in Subsection II-B), by expanding the expression of expected utility of each neighbor location of a location l, Formula 3 gives the specific adaption of Bellman's Equation to the Pac-Man problem. In case that a neighbor location of a location l is a wall, the utility of the neighbor location of the location l at iteration $i-1$ (i.e. $U_{i-1}(e_l)$, $U_{i-1}(w_l)$, $U_{i-1}(n_l)$, or $U_{i-1}(s_l)$) shall be substituted by the utility of the location l itself (i.e. $U_{i-1}(l)$).

$$\begin{aligned} &U_i(l) = R(l) + \\ &\gamma \max[0.8U_{i-1}(e_l) + 0.1U_{i-1}(n_l) + 0.1U_{i-1}(s_l), \\ &\quad 0.8U_{i-1}(w_l) + 0.1U_{i-1}(s_l) + 0.1U_{i-1}(n_l), \\ &\quad 0.8U_{i-1}(n_l) + 0.1U_{i-1}(w_l) + 0.1U_{i-1}(e_l), \\ &\quad 0.8U_{i-1}(s_l) + 0.1U_{i-1}(e_l) + 0.1U_{i-1}(s_l)] \end{aligned} \tag{3}$$

B. Initialize Utilities

The first step of value iteration on Bellman's Equation is to initialize utilities for different locations in the maze. The initialized utilities assigned to locations of different objects are listed in Table I. To counter the problem of surprise attack, I invented an early warning mechanism to preemptively evade from the ghosts that are hiding behind the foods (defined later in Subsection V-A). An example of utility initialization is provided (Fig. 2).

TABLE I
INITIALIZED UTILITY VALUES FOR DIFFERENT OBJECTS

Type of object	Initialized utility value
Capsule	+50.0
Food	+10.0
Free space	0.0
Edible ghost	+400.0
Hostile ghost	-500.0

Wall	Wall	Wall	Wall	Wall	Wall	Wall
Wall	Free 0.0	Agent 0.0	Free 0.0	Free 0.0	Free 0.0	Wall
Wall	Free 0.0	Wall	Wall	Wall	Free 0.0	Wall
Wall	Free 0.0	Wall	Food +10.0	Free 0.0	Free 0.0	Wall
Wall	Free 0.0	Wall	Wall	Wall	Free 0.0	Wall
Wall	Food -300.0	Free -400.0	Ghost -500.0	Free -400.0	Free -300.0	Wall
Wall	Wall	Wall	Wall	Wall	Wall	Wall

Fig. 2. Initialized utilities of `smallGrid` layout at timestamp 0

Because the version of `api.py` that this project works with provides the edible-or-hostile state of each ghost, my design incorporates 3 different ghostbuster modes to experiment with to maximize win rate in the `mediumClassic` layout. The three following paragraphs will iterate in details about how different choices of the parameter ghostbuster mode initialize their corresponding utilities differently. The key features of different ghostbuster modes are listed in Table II. The parameter ghostbuster mode is one that can be fine-tuned to maximize win rate through experimentation (tested in Paragraph V-C.2).

TABLE II
COMPARISON OF DIFFERENT GHOSTBUSTER MODE

Ghostbuster mode:	Inactive	Defensive	Offensive
Evade from hostile ghosts?	Yes	Yes	Yes
Aim to eat edible ghosts	No	Yes	Yes
Aim to eat capsules?	No	No	Yes

1) Inactive Ghostbuster Mode: The inactive ghostbuster mode always runs away from the ghosts, no matter they are edible or hostile, and doesn't intentionally aim to eat the capsules. Therefore, the inactive ghostbuster mode only initializes utilities for foods, free spaces, and ghosts: +10.0 for each food; 0.0 for each free space; -500.0 for each ghost and decaying threats for its surrounding spaces.

2) Defensive Ghostbuster Mode: The defensive ghostbuster mode always aims to eat edible ghosts as a priority, and runs away from hostile ghosts, but doesn't intentionally aim to eat the capsules. Therefore, the defensive ghostbuster mode initializes utilities for free spaces, and hostile ghosts: 0.0 for each free space; -500.00 for each hostile ghost and decaying threats for its surrounding spaces. The defensive ghostbuster mode also initializes utilities for either edible ghosts or foods, but never for both at the same time: +400.0 for each edible ghost; +10.0 for each food.

3) Offensive Ghostbuster Mode: The offensive ghostbuster mode always aims to eat edible ghosts as a priority, and runs away from hostile ghosts, and aggressively aims to eat the capsules. Therefore, the offensive ghostbuster mode initializes utilities for free spaces, and hostile ghosts: 0.0 for each free space; -500.00 for each hostile ghost and decaying threats for its surrounding spaces. The offensive ghostbuster mode also initializes utilities for either capsules, or edible ghosts, or foods, but never for more than one type of them at the same time: +50.0 for each capsule; +400.0 for each edible ghost; +10.0 for each food.

C. Update Utilities

The second step of value iteration on Bellman's equation is to update utilities for all the non-convergent free spaces until convergence. Each iteration makes a deep copy of the `utilities` data structure, named `previous_utilities`. It uses the adapted Bellman's Equation (Formula 3) to update the `utilities` (iteration i) data structure, based on the `previous_utilities` (iteration $i-1$) data structure.

The technique of value iteration repeats the above process of updating utilities until convergence. An example of utility convergence is provided (Fig. 3). The next subsection will provide a computationally efficient definition of convergence.

Wall	Wall	Wall	Wall	Wall	Wall	Wall
Wall	Free -1.357	Agent +0.049	Free +1.299	Free +2.549	Free +3.799	Wall
Wall	Free -2.607	Wall	Wall	Wall	Free +5.205	Wall
Wall	Free -3.857	Wall	Food +10.0	Free +8.750	Free +7.106	Wall
Wall	Free -5.107	Wall	Wall	Wall	Free +5.856	Wall
Wall	Food -300.0	Free -400.0	Ghost -500.0	Free -400.0	Free -300.0	Wall
Wall	Wall	Wall	Wall	Wall	Wall	Wall

Fig. 3. Convergent utilities of `smallGrid` layout at timestamp 0

D. Definition of Convergence

To make the value iteration work in a computational efficient way, we need a definition of convergence. Because all the utility values assigned are decimal, to reach exact convergence, meaning the difference of the utility value of a location l at iteration i and that at iteration $i-1$ for each location equals exactly 0, is very computationally costly. Therefore, I came up with two techniques, (1) convergence tolerance and (2) early stopping point, to reduce computational cost.

1) Convergence Tolerance: The convergence tolerance is an arbitrary parameter, whose value is of the user's choice, to balance the trade-off between the exactness of convergence and the computational cost of value iteration on Bellman's Equation. For each location, if the difference of the utility value at iteration i and that at iteration $i-1$ is below the convergence tolerance, we mark that the utility of this location has converged. The small the convergence tolerance is, the closer the final convergent utility value is to the theoretical convergent utility value. However, there is always a trade-off between exact convergence and computational cost as stated in the previous paragraph. The convergence tolerance of my choice is 0.0001.

2) Early Stopping Point: The early stopping point is another arbitrary parameter, whose value is of the user's choice, to balance the trade-off between the exactness of convergence and the computational cost of value iteration on Bellman's Equation. For an entire cycle of value iteration, even though the technique of convergence tolerance is in place, it might still possess the possibility of non-convergence within a reasonably short length of iterations. This phenomenon is especially frequent when the each round is close to an end, when most locations within the maze are free spaces on which value iteration needs to update utilities. Therefore, by putting a ceiling on the number of iterations, value iteration can stop early. The ceiling is called early stopping point. The early stopping point of my choice is 100 for non-sparse targets, such as foods, and 200 for sparse targets, such as capsules and edible ghosts.

IV. Design of Strategies

The Pac-Man agent I designed is named `MDPAgent`. With the Markov Decision Process embedded, the overall workflow of `MDPAgent`'s strategies is divided into three major functional components in sequence (Fig. 4): (1) mapping operation, (2) value iteration, and (3) decision-making based on maximum expected utility. The three following subsections will discuss these three major functional components with visual aid of UML-style diagrams.

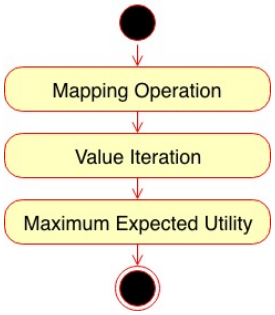

Fig. 4. Activity diagram of `MDPAgent.getAction()`

A. Mapping Operation

The first major functional component of `MDPAgent`'s workflow is the mapping operation. Although this project returns to the setting of a fully observable environment, maintaining a collection of internal memories about some aspects of information about the maze helps reduce computational redundancy. More specifically, that is to prevent from repeatedly calling `api.py` for information about the static or predictable components of the environment. The static components of the environment, such as locations of walls, corners, and navigable spaces, are those that are fixed once the maze is instantiated. the predictable components of the environment, such as locations of capsules and foods, are those that can be tracked as long as the initial record of these components and travel record of the `MDPAgent` are provided constantly.

The mapping operation (Fig. 5) works as follows. At the initial state of each round, the `MDPAgent` initializes the internal memories to store and track information about the static and predictable components of the environment by calling the relevant functions from `api.py`. Then, every time the `getAction()` method is invoked, the mapping operation logs the game state history, and finds the location of the agent. If the location of the agent belongs to the internal memories of available capsules or foods, the mapping operation removes this location from the corresponding internal memories.

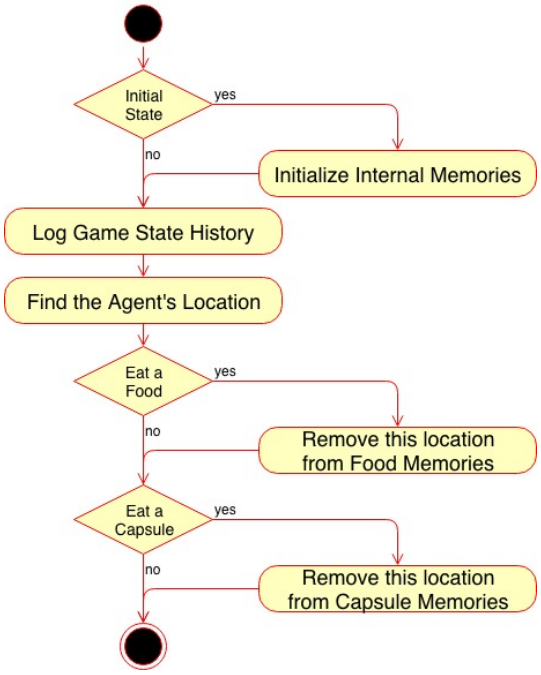

Fig. 5. Activity diagram of Mapping Operation

B. Value Iteration

The second major functional component of `MDPAgent`'s workflow is to apply value iteration on Bellman's Equation. To begin with, the `MDPAgent` initializes utility values for different locations in the maze (defined in Subsection III-B). Then, the `MDPAgent` updates utility values for all the free spaces in the maze (defined in Subsection III-C), until reaching either full convergence or early stopping point (defined in Subsection III-D).

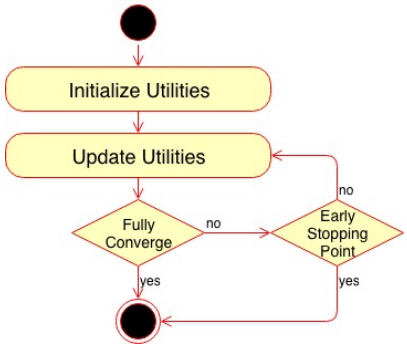

Fig. 6. Activity diagram of Value Iteration

C. Decision-Making based on Maximum Expected Utility

The third major functional component of `MDPAgent`'s workflow is the decision-making process based on maximum expected utility. After the cycle of value iteration on the current game state, which means reaching either full convergence on all the free spaces or early stopping point, the `MDPAgent` uses the rule of maximum expected utility to decide which neighbor location to move towards (Fig. 7). More specifically, the `MDPAgent` looks at all of its neighbor locations, to which all the legal actions (the stop option removed) lead to, and chooses the neighbor location with the highest value of expected utility calculated by value iteration. Then, the `MDPAgent` returns the legal action that leads to that neighbor location.

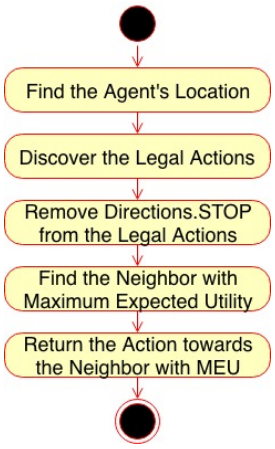

Fig. 7. Activity diagram of Maximum Expected Utility

V. Creativity, Optimization and Evaluation

Due to non-deterministic nature of this project, the first version of my design had a catastrophic win rate of less than 20%. Therefore, I used my creativity to optimize my design,

and analyzed the effectiveness of these new features based on the data generated by running thousands of rounds of games. The following subsections will discuss (1) the early warning system to evade from ghosts, (2) modular programming of object-oriented design, and (3) how I fine-tuned the arbitrary parameter setting of my `MDPAgent` strategies.

A. Early Warning System

In the very early stage of this project, I simply initialized utility values to different components of the environment as follows: +10.0 for each; 0.0 for each free space; -500.0 for each ghost. However, this naive way of initializing utilities only resulted in a stabilized win rate of 20%. After analyzing the printed utility values of the entire maze for those failed rounds, I discovered that the main cause was the surprise attack by the ghosts, and I provided a countermeasure to surprise attack, called decaying threat. The two following paragraphs will give the definitions of surprise attack and decaying threat.

1) Surprise Attack: Because the technique of value iteration on Bellman's Equation only updates utilities on free spaces, if the ghosts are hiding inside a cluster of foods (Fig. 8), it is impossible for the information of the ghosts to penetrate the barrier of foods and reach the agent. Therefore, the agent is not aware of the ghosts until it is too close to the ghosts to evade from them. To solve this problem, the agent needs an early warning system to preemptively evade from the ghosts.

Wall	Free < +10	Free < +10	Free < +10	Free < +10	Wall
Wall	Free < +10	Wall	Wall	Wall	Wall
Wall	Free < +10	Food +10	Ghost -500	Food +10	Free < +10
Wall	Free < +10	Wall	Food +10	Wall	Free < +10
Wall	Free < +10	Wall	Agent	Wall	Free < +10

Fig. 8. Surprise Attack: ghosts hiding behind foods

2) Decaying Threat: I created an early warning system by initializing decaying threats for each ghost's surrounding spaces. Starting with -500.0 for each ghost, the early warning system recursively initializes negative utility values at an arbitrary decay rate as the radius to the ghost grows. The early warning system stops the decaying threats at a safety distance (a parameter that needs to be fine-tuned through experimentation, tested in Paragraph V-C.3). For example (Fig. 9), the decay rate is 100.0; the safety distance is 5 steps. The agent knows from which direction the ghost will attack it at that moment, at the maximum distance of 5. Therefore, in this example, the agent definitely would not move to east, because it is more dangerous (i.e. more negative utility).

3) Effectiveness of Early Warning System: Table III proves that the early warning system of my design, which ueses the solution of decaying threats, increases the stabilized win rate by 30 percentage point.

Wall	Free < +10	Free < +10	Free < +10	Free < +10	Wall
Wall	Free < +10	Wall	Wall	Wall	Wall
Wall	Agent	Food -100	Food -200	Food -300	Food -400
Wall	Free < +10	Wall	Free -100	Wall	Ghost -500
Wall	Free < +10	Wall	Free < +10	Wall	Free -400

Fig. 9. Decaying Threat: a solution to counter surprise attack

TABLE III
EVALUATE THE EFFECTIVENESS OF EARLY WARNING SYSTEM

	No decaying threat	Decaying threat
Stabilized win rate	20%	50%

B. Object-Oriented Design

In this project, I further put modular programming into effect. This good practice of object-oriented design improves modularity and readability of my code structure (Fig. 10). I breaks down the `getAction()` method, a level-1 function, into three level-2 functions, which correspond to the three major functional components of the overall workflow of `MDPAgent`'s strategies (defined in Section IV). Furthermore, the `__value_iteration()` method, a level-2 function, is broken down into three level-3 functions, including two functional sub-components, which represent the core functionalities inside value iteration, and a helper function that print the `utilities` data structure on a user-friendly display. This design of function call hierarchy provides convenience for debug mode and log mode, which I will discuss in the two following paragraphs.

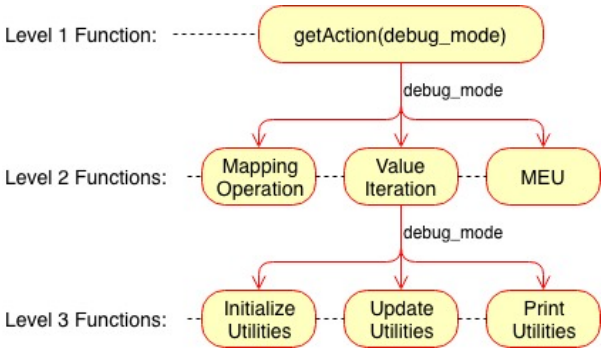

Fig. 10. Function call hierarchy of `MDPAgent.getAction()`

1) Debug Mode Master Switch: My design of the 3-level function call hierarchy (Fig. 10) allows me to embed a master switch for debug mode. I added an optional input parameter `debug_mode` for the `getAction()` method,

and a required input parameter `debug_mode` for every level-2 function. The level-1 setting of `debug_mode` therefore automatically flows to all the level-2 functions. By changing the default value of the level-1 input parameter `debug_mode`, the `MDPAgent` switches between active and inactive debug mode across all its member methods, like flipping a master switch. If the debug mode is active, the `getAction()` method automatically print every well-formatted debug message, including the user-friendly display of the `utilities` data structure. This design is more efficient and more elegant than spending time on toggling comments on all the `print()` lines.

2) Log Mode for Parameter Analysis: Because the game calls the `final()` method on `MDPAgent` at the end of each round, I added an optional input parameter `log_mode` for the `final()`. By changing the default value of the input parameter `log_mode`, the `MDPAgent` switches between active and inactive log mode. If the log mode is active, the `MDPAgent` logs the parameter (i.e. discount factor, convergence tolerance, ghostbuster mode, etc.) setting of this round, along with the result (i.e. win or loss) and score to a `log.csv` file on the append mode. The log file allows me to aggregate all the game result and analyze the performance of different parameter settings.

C. Parameter Tuning

This subsection analyzes the performance of various parameter settings. There are three key parameters in my design that needs fine-tuning to find their optimal values to maximize win rate in the `mediumClassic` layout through experimentation: (1) the discount factor inside Bellman's Equation, (2) the ghostbuster mode that dictates the agent's behavioral pattern, and (3) the safety distance to initialize utility values for a ghost's surrounding spaces. I wrote a separate `test_case_generator.py` code file to generate all the combinations of selected values of the three key parameters (Table IV). In total, there are 45 different combination of test cases. I ran 50 rounds of games on each test case, and logged the record of each round into a log file for parameter analysis. The total number of testing data entry is 2,250. The three following paragraphs will analyze each one of these key parameters above and discuss the findings.

TABLE IV
SELECTED VALUES OF THE THREE KEY PARAMETERS

Key parameter	Selected values
Discount factor	1.0, 0.9, 0.8, 0.7, 0.6
Ghostbuster mode	inactive, defensive, offensive
Safety distance	4 steps, 3 steps, 2 steps

1) Discount Factor: The parameter discount factor (defined in Subsection III-A) dictates how much the agent sees into the future. The bigger the value of discount factor is, the more the agent sees into the future. According to data gathered through experimentation, discount factor being 1.0 results in the highest average win rate of 28.70%. This finding makes intuitive sense, because 1.0 is the largest possible value of discount factor, which grants the agent maximum level of insight into the future.

TABLE V
PARAMETER ANALYSIS OF DISCOUNT FACTOR

Discount factor	Average win rate
1.0	28.70%
0.9	27.42%
0.8	19.56%
0.7	12.61%
0.6	11.70%

2) Ghostbuster Mode: The parameter ghostbuster mode (defined in Subsection III-B) dictates how aggressively the agent goes after the ghosts. The inactive ghostbuster mode is the least aggressive; the defensive ghostbuster mode is in the middle; the offensive ghostbuster mode is the most aggressive. According to data gathered through experimentation, ghostbuster mode being inactive results in the highest average win rate of 46.25%. My first impression of ghostbuster mode is that the offensive ghostbuster mode should be the optimal choice, because it proactively destroys the ghosts. However, the experimentation tells the exactly opposite. One possible explanation for the optimal setting of ghostbuster mode being inactive is that, since eating all the foods wins the game, the inactive ghostbuster mode, which does not bother to hunt down capsules or ghosts, spends less time to win a game on average, decreasing the chance being eaten by the ghosts.

TABLE VI
PARAMETER ANALYSIS OF GHOSTBUSTER MODE

Ghostbuster mode	Average win rate
Inactive	46.25%
Defensive	29.07%
Offensive	24.68%

3) Safety Distance: The parameter safety distance (defined in Subsection V-A) dictates the range of the early warning system. The larger the value of safety distance is, the farther away the agent anticipates the incoming ghosts. According to data gathered through experimentation, safety distance being 4 steps results in the highest average win rate of 41.68%. This finding makes intuitive sense, because 4 steps is the largest among the selected values of safety distance, which grants the agent the maximum level of caution and better chance of survival.

TABLE VII
PARAMETER ANALYSIS OF SAFETY DISTANCE

Safety distance	Average win rate
4 steps	41.68%
3 steps	34.00%
2 steps	24.31%

4) Optimal Parameter Setting: According to the findings of separate analyses of the three key parameters, the optimal parameter setting is: discount factor being 1.0; ghostbuster mode being inactive; safety distance being 4 steps (Table

VIII). To evaluate the effectiveness of this optimal parameter setting, I ran 10,000 rounds on `smallGrid` layout plus 1,000 rounds on `mediumClassic` layout, achieving 58.04% and 55.90% win rate respectively.

TABLE VIII
SPECIFICATION OF OPTIMAL PARAMETER SETTING

Key parameter	Optimal value
Discount factor	1.0
Ghostbuster mode	Inactive
Safety distance	4 steps

TABLE IX
EVALUATION OF OPTIMAL PARAMETER SETTING

Simulation configuration	Win rate
10,000 rounds on `smallGrid` layout	58.04%
1,000 rounds on `mediumClassic` layout	55.90%

VI. CONCLUSION

This project provides the opportunities to apply Markov Decision Process in a non-deterministic environment. My design of Pac-Man strategies gives a practical sense of effectiveness of Markov Decision Process in decision-making process. In addition, this project highlights the importance of modular programming of object-oriented design, which rewarded me with robust code structures, and the necessary infrastructure in place to help debugging and logging. Finally, data through statistically significant counts of experiments provide indisputable evidence to any hypotheses and findings. Considering the advantages of both pathfinding algorithms from my last Pac-Man project, and Markov Decision Process from this project, I believe a spin-off project that incorporate the fast searching capability of pathfinding algorithms, and the look-into-future magic of Markov Decision Process should be on the agenda.

REFERENCES

[1] Russell, Stuart J.; Norvig, Peter (2010), Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Pearson Education, Inc., ISBN 0-13-604259-7, chpt. 17.

[2] Bellman, R. (1957). "A Markovian Decision Process". Journal of Mathematics and Mechanics. 6.