HUST

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY.

INTRODUCTION TO AI

IT3160E

INSTRUCTOR: PROF. DR HAI VAN PHAM

Class ID: 147838

Comparative Analysis of Breadth-first search and Reinforcement Learning Algorithms for Automating Pacman Gameplay

---

Group Members:
Le Van Hau - 20226038
Nguyen Trung Hieu - 20225971
Phan Gia Do - 20226026
Nguyen Cong Dat - 20226022
Tran Gia Khanh - 20226048

Abstract

In this article, we used two methods to automate playing the Pacman game: Breadth-first search (BFS) and Reinforcement Learning (RL). Each method was implemented and trained over a long period and on a large scale to ensure the best results. After a series of experiments, we evaluated the performance of the two algorithms based on different criteria such as: Winning time, winning rate, adaptability and pathfinding ability of Pacman. From the results obtained , we conclude that each method has its own advantages and disadvantages depending on each case. The advantage of BFS is that it is easy to implement and can be considered an optimal solution in a simple environment. If in a complex environment, BFS will become ineffective and time-limited. In contrast, RL exhibits superior performance in a complex environment thanks to its ability to learn so that it can adapt to the current environment.

1 Introduction

Breadth-first search (BFS) and Reinforcement Learning (RL) are two classic algorithms in AI and have numerous applications across most domains of life. Their applications have been validated over decades, including automating gameplay. We chose the Pacman game because of its simple design, easy-to-understand rules, and familiarity to most people, making it convenient for us to implement the algorithms. We will redesign the Pacman game environment from Python's Pygame library, from graphics to gameplay. After having the Pacman game environment, the above two algorithms will be implemented, trained and tested to obtain results. We evaluate the results based on different criteria such as: Winning time, winning rate, Pacman's ability to adapt and find paths,... to draw conclusions about the performance of the two algorithms. Through this article, we hope readers can better understand how both algorithms work and understand their advantages and disadvantages in different situations.

2 The game of Pac-Man

2.1 What is Pac-Man ?

Figure 1: Pac-Man game

"Pac-Man" is a video game that was released by Namco Bandai Games (then called Namco) in 1980. Over the last 30 years, Pac-Man has become an international smash hit, gaining fame far beyond the borders of Japan. It is one of the best-selling Japanese video games of all time, and its characters are still loved by people around the world.

"Pac-Man" combines simple rules with deep gameplay, colorful and cute characters, lively music, unique sound effects and humorous intermissions (known as "coffee breaks") between certain game levels. This innovative combination has captured the hearts of gamers everywhere, including the many women and children to whom Pac-Man was the first video game they ever played.

The game's popularity in North America is especially notable as the arcade machine sold more than 100,000 units in its first year of production, then an unheard of number of sales. Incredibly, over 400 different types of Pac-Man-related merchandise were also released. The popularity of "Pac-Man" in the United States made him a social phenomenon, resulting in both a popular animated television series and a hit disco record ("Pac-Man Fever" by Buckner & Garcia).

Further, in 2005 "Pac-Man" received the Guinness World Records award for being the "Most Successful Coin-Operated Game".

After the release of the 1980 arcade game, "Pac-Man" was ported to various video game platform including home gaming consoles, such as the Nintendo Entertainment System and PlayStation, handheld systems, and even on mobile phones. Additionally, over the years, numerous new titles and series sequels have also been released, including "Super Pac-Man" and "PAC-LAND". Still today, 30 years after its release, the game can be played on present day gaming systems and brand new mobile phone devices, and continues to be loved by people around the world.

2.2 Gameplay

Pac-Man is an action maze chase video game; the player controls the eponymous character through an enclosed maze. The objective of the game is to eat all of the dots placed in the maze while avoiding four colored ghosts—Blinky (red), Pinky (pink), Inky (cyan), and Clyde (orange)—who chase Pac-Man. When Pac-Man eats all of the dots, the player advances to the next level. Levels are indicated by fruit icons at the bottom of the screen. In between levels are short cutscenes featuring Pac-Man and Blinky in humorous, comical situations.

If Pac-Man is caught by a ghost, he loses a life; the game ends when all lives are lost. Each of the four ghosts has its own unique artificial intelligence (A.I.), or "personality": Blinky gives direct chase to Pac-Man; Pinky and Inky try to position themselves in front of Pac-Man, usually by cornering him; and Clyde switches between chasing Pac-Man and fleeing from him.

Placed near the four corners of the maze are large flashing "energizers" or "power pellets". When Pac-Man eats one, the ghosts turn blue with a dizzied expression and reverse direction. Pac-Man can eat blue ghosts for bonus points; when a ghost is eaten, its eyes make their way back to the center box in the maze, where the ghost "regenerates" and resumes its normal activity. Eating multiple blue ghosts in succession increases their point value. After a certain amount

of time, blue-colored ghosts flash white before turning back into their normal forms. Eating a certain number of dots in a level causes a bonus item—usually a fruit—to appear underneath the center box; the item can be eaten for bonus points. To the sides of the maze are two "warp tunnels", which allow Pac-Man and the ghosts to travel to the opposite side of the screen. Ghosts become slower when entering and exiting these tunnels.

The game increases in difficulty as the player progresses: the ghosts become faster, and the energizers' effect decreases in duration, eventually disappearing entirely. An integer overflow causes the 256th level to load improperly, rendering it impossible to complete.

However, in this project, for the convenience of implementing the algorithm, we will slightly modify the gameplay compared to the original version. Instead of having multiple levels with different stages, we will have only one single level. Therefore, when Pac-Man eats all the food, the screen will display a victory message, and conversely, when Pac-Man gets caught, the screen will display a game-over message.

3 Inputs / Outputs

3.1 Inputs

3.1.1 Pac-Man game environment:

Using the Python Pygame library, we have redesigned the Pac-Man game based on the original version but with a few changes as mentioned above, including tasks such as:

- Designing a Tile-based Board
- Drawing Each Tile Type onto the board
- Drawing and Animating the Player onto the screen!
- Move the Player Using Arrow Keys
- Check for Player Collisions with walls and allowable turns!
- Scoring and 'Eating' The dots and powerups!
- Setting up powerup active timer
- Loading ghost images and setting up the Ghost Class!
- Get ghosts to pass through 'Ghost Cage' Door
- Colliding with ghosts to eat them and lose lives!
- Reset if Ghost is dead when they enter ghost box
- Scoring points when eating ghosts

- Game Over and Game Won Restart Conditions!

The results are shown in the image below. We can play normally just like in the original game. Based on this result, we began developing AI algorithms to enable the computer to autonomously control the Pac-Man character to find food and achieve victory without human intervention.

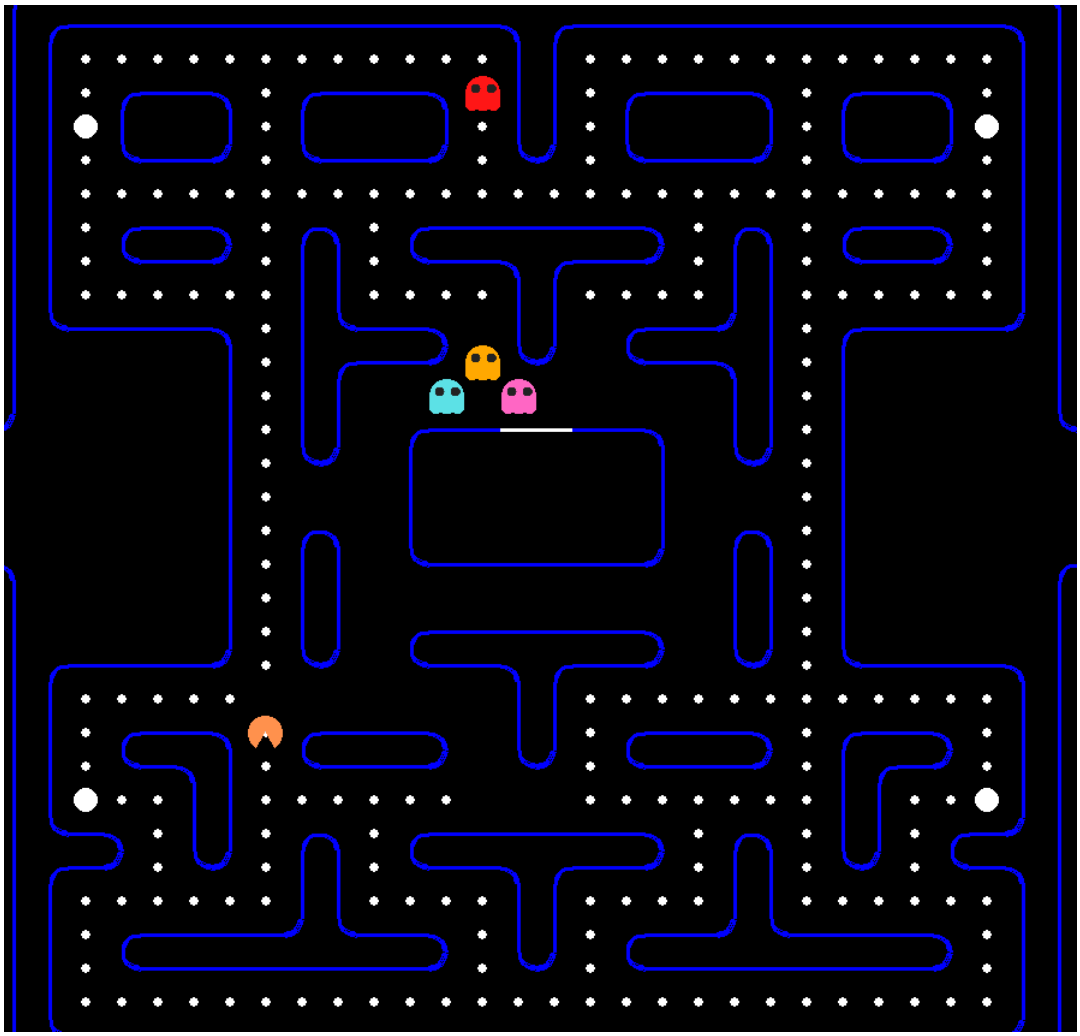


Figure 2: Maze Redesigned

Game Rules: The game rules, including how Pacman moves, how Ghosts move, and the winning and losing conditions (eating all dots or being caught by a Ghost).

3.1.2 BFS and Reinforcement Learning Algorithms:

- BFS: Implementation of the Breadth-First Search algorithm, requiring the maze map and initial state.
- Reinforcement Learning: Implementation of the Q-learning algorithm or another RL method, requiring the Pacman simulation environment and training parameters (learning rate, discount factor, exploration rate).

3.2 Outputs

We can see that the game will have two outcomes: Winning or Losing. Winning occurs when Pac-Man eats all the food pellets, and losing happens when Pac-Man loses all its lives. In both cases, the result will be displayed on the screen as shown in the images below.

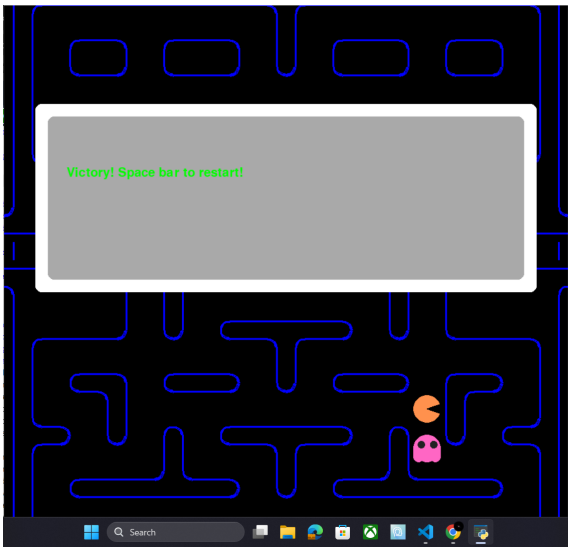


Figure 3: Win screen

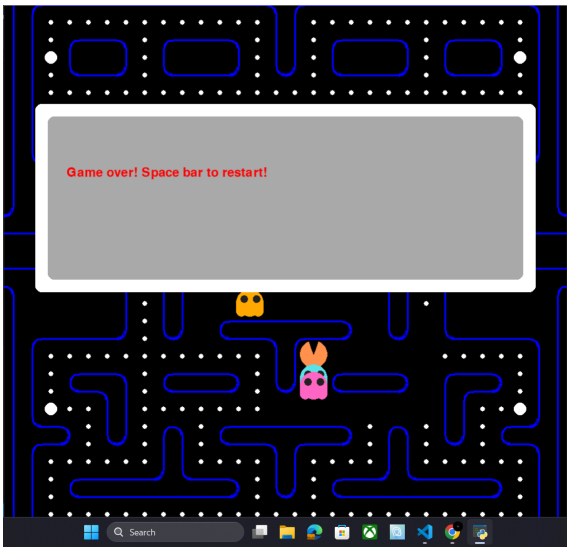


Figure 4: Lose screen

In addition, we have other outputs such as:

- Win Time: The time each win game.
- Win Rate: The percentage of games the algorithm can win after a certain number of iterations.
- Performance in Complex Environments: The results of the algorithm when applied to maze maps of different sizes and complexities.
- Path Quality: Evaluation of the quality of the paths generated by the algorithm, including path length and efficiency in avoiding Ghosts and eating pellets.
- Optimality of Solutions: Measurement of how optimal the solutions found by each algorithm are, including maximizing score and minimizing playtime.

4 Method

4.1 Maze encoding

For the convenience of developing the algorithm, we have constructed the maze in a grid format and encoded the different components in the maze as distinct numbers. When moving, Pacman will move through each cell in the grid sequentially. This way, our algorithms can easily distinguish between the components. The details of our encoding are as follows:

Table 1: Maze Encoding Table

Component	Encoding Value	Component	Encoding Value
Void region	0	Top right edge of arc	5
Small dots	1	Top left edge of arc	6
Big dots	2	Bottom left edge of arc	7
Vertical straight line	3	Bottom right edge of arc	8
Horizontal straight line	4	Special straight line	9

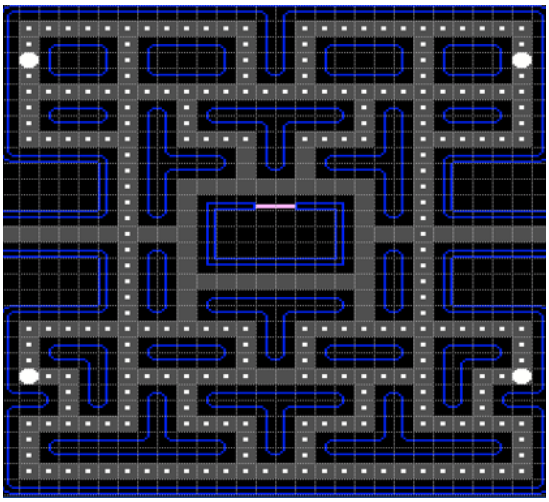


Figure 5: Grid Maze

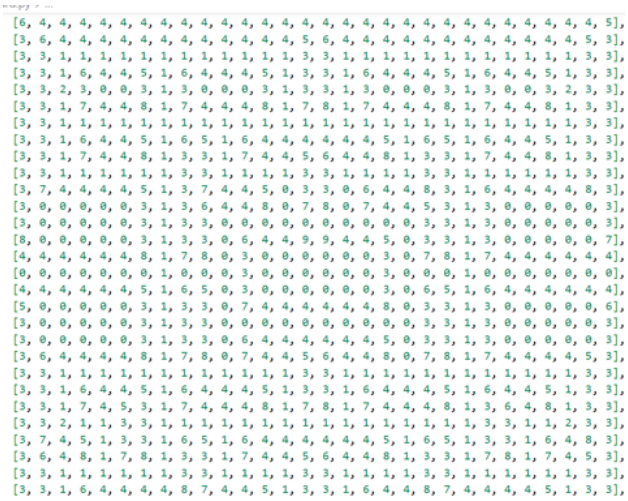


Figure 6: Grid Maze Encoded

4.2 Breadth-first search - BFS

4.2.1 Definition

Breadth-first search (BFS) is one of the classical graph theory algorithms, typically expressed under the imperative style. However, it has applications in other domains such as artificial intelligence. All known algorithms of BFS use iteration. In addition, BFS is one of the oldest and most fundamental graph traversal algorithms, infuencing many other graph algorithms.

BFS is used to search a single connected graph component. It describes a deterministic way of exploring the vertices of the graph component. The standard definition of BFS involves an iterative loop over a queue of vertices.

Exploration of a graph component is achieved by dequeuing a vertex to discover unexplored vertices which are subsequently enqueued. This kind of manipulation produces graph exploration which can be viewed as progressive levels of discovered vertices.

A BFS algorithm implicitly computes the shortest path distances from a given source vertex to all other reachable vertices in a single graph component $Gc = (Vc; Ec)$. This computation is achieved by traversing a particular spanning tree rooted at the source vertex in a deterministic order. Let the spanning tree be known as the BFS spanning tree.

The traversal of the BFS spanning tree starts at level 0 and continues to vertices at each level of the spanning tree from level 1 to level h. A level corresponds to the distance along a path from the source vertex to a vertex in the spanning tree. Level 0 contains the source vertex. The value h is the height of the BFS spanning tree (the largest level number). Based on these concepts, a mathematical description of the traversal of the BFS spanning tree is defined.

Definition 4.1. *A sequence $L = L_0...L_h$ represents the traversal of a BFS spanning tree. Each $L_i = v_{i1}...v_{i*}$ (called a level sequence) is the sequence of vertices traversed at level i of the tree, where v_{ij} represents the j^{th} vertex traversed and vi is the last vertex traversed.*

The source vertex of the BFS spanning tree is v_{01}, and L_0 only contains v_{01}. The distinction between each L_i indicates the separation of vertices at each level of the BFS spanning tree. Vertices in $L_0...L_{h-1}$ are internal nodes of the spanning tree, and vertices in L_h are the leaf (external) nodes of the spanning tree. The concatenation of $L_0...L_h$ is the overall BFS traversal sequence. The relationship between the BFS spanning tree and the level sequences can be further elaborated by the following statements:

1. All vertices in the graph component (and hence, the BFS spanning tree) are found in the union of level sequences,

$$V_c = \bigcup_{k \in 0...h} L_k$$

2. Vertices and corresponding adjacent vertices in the BFS spanning tree occupy adjacent level sequences, for $i \in 0...(h-1)$

$$L_{i+1} = \{v | u \in L_i, (u,v) \in E_c, v \in V_c - \bigcup_{k \in 0...i} L_k\}$$

3. A vertex only appears once in one of the level sequences.

Definition 4.2. *For $i \in 0...h$, $s \in V_c$, and $L_0 = s$, $clos(L_0)$ generates the sequence $L = L_0...L_h$, where the clos recurrences is defined as,*

$$clos(L_0...L_i) \rightarrow \begin{cases} L_0...L_i & if \quad L_{i+1} = \lambda \\ clos(L_0...L_{i+1}) & otherwise \end{cases}$$

and

$$L_{i+1} = \{v | u \in L_i, (u,v) \in E_c, v \in V_c - \bigcup_{k \in 0...i} L_k\}$$

Where $L_{i+1} = \lambda$ means level $(i+1)$ does not exist in the BFS spanning tree, and the calculation of L_{i+1} generates a sequence. The recurrence *clos* is so named because the calculation is similar to transitive closure. Definition 2.2 leads directly to the following lemma.

Lemma 4.1. *The clos recurrence of Definition 2.2 is a valid recurrence, and* $clos(L_0)$ *calculates the sequence of level sequences* $L_0...L_h$

Level sequences are not obvious in the standard imperative BFS algorithm. Vertices are collected individually, with no emphasis on the distinction between level sequences. However, it is possible to define recursive and iterative algorithms to compute BFS with respect to the level sequences, and to derive the standard BFS algorithm.

Let υ be a global array of booleans associated with vertices ($\upsilon : V \rightarrow \{False, True\}$).

Before any of the algorithms are executed, assume that the boolean array is initialized ($\forall v \in V_c; \upsilon[v] \leftarrow False$). Let Adj be the global adjacency array representing E_c. BFS algorithms (such as AI algorithms) typically produce the ordered traversal of vertices as a result. The algorithms here produce an ordered set of vertices.

From these conditions, it is no longer necessary to keep track of the entire sequence $L_0..L_i$ from the *clos* recurrence, only L_i and L_{i+1} are necessary. Algorithm 3.1 calculates $clos(L_0)$ with the initial call $recBFS(L_0)$, where $L_0 = s$. The algorithmic call *CONCAT* is defined as follows:

$$CONCAT(a_1...a_*, b_1...b_*) = a_1...a_*b_1...b_*$$

Algorithm 3.1 A recursive BFS algorithm.

recBFS(L_i)

```
(1)   foreach v ∈ L_i do
(2)      ϑ[v] ← True
(3)   endfor
(4)   L_{i+1} ← λ
(5)   foreach u ∈ L_i do
(6)      foreach v ∈ Adj[u] do
(7)         if ϑ[v] = False then
(8)            L_{i+1} ← CONCAT(L_{i+1}, v)
(9)         endif
(10)     endfor
(11)  endfor
(12)  if L_{i+1} ≠ λ then
(13)     return CONCAT(L_i, recBFS(L_{i+1}))
(14)  else
(15)     return L_i
(16)  endif
```

$recBFS(L_0)$ produces the call sequence:

$$CONCAT(L_0, CONCAT(L1, ...CONCAT(L_{h-1}, L_h)...))$$

This sequence is tail-recursive. After unfolding a recursive call, no further changes are made (no further recursion). For example, with $h = 3$ the call sequence is:

$$\begin{aligned} recBFS(L_0) &\rightarrow CONCAT(L_0, recBFS(L_1)) \\ &\rightarrow CONCAT(L_0, CONCAT(L_1, recBFS(L_2)) \\ &\rightarrow CONCAT(L_0, CONCAT(L_1, L_2) \\ &\rightarrow CONCAT(L_0, L_1L_2) \\ &\rightarrow L_0L_1L_2 \end{aligned}$$

4.2.2 BFS algorithm

We use the Breadth-First Search (BFS) algorithm for the Pacman game to find the optimal path from Pacman's current position to the nearest food while avoiding threats from the Ghosts. The algorithm begins by defining the possible movement directions for Pacman: *Right, Left, Up, and Down*. The BFS function takes Pacman's current coordinates as input and returns the direction that leads to the food. Another function, *"Check ghost"*, checks the coordinates of both the Ghosts and Pacman to discover if any Ghost is in the four directions around Pacman.

During execution, if Pacman's movement direction coincides with a Ghost's direction, the algorithm will check other directions to find an escape route. Otherwise, Pacman will follow the direction towards the food. This method ensures that Pacman always finds the safest and most efficient path to collect food while avoiding danger from the Ghosts.

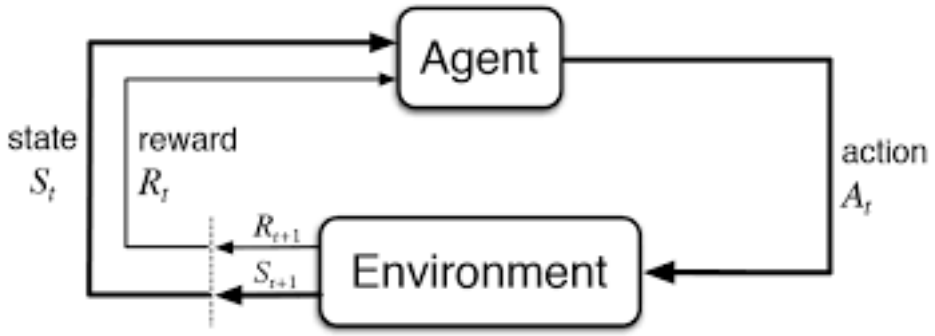
```
Directions = [RIGHT, LEFT, UP, DOWN]

def BFS(Pacman's coordinate):
    return direction to food

def Check_ghost(Ghost's coordinate, Pacman's coordinate):
    return check if ghost is in any of the four directions

if Pacman's direction == ghost's direction:
    Check other directions to escape
else:
    Pacman takes the direction to food
```

Figure 7: BFS Pseudocode

4.3 Reinforcement Learing - RL

4.3.1 Definition

Reinforcement learning is one of three basic machine learning paradigms alongside supervised learning and unsupervised learning. It is the science of decision making. It is about learning the optimal behavior in an environment to obtain maximum reward. This optimal behavior is learned through interactions with the environment and observations of how it responds, similar to children exploring the world around them and learning the actions that help them achieve a goal.



An excellent way for describing a RL task is through the use of Markov Decision Processes. A Markov Decision Process (MDP) can be supposed as a tuple (S, A, P, R, γ), where S is set of states; A a set of actions; P : S × A × S → [0, 1] is a Markovian transition model that specifies

the probability, P(s, a, s'), of transition to a state s' when taken an action a in state s; R : S × A $\rightarrow$ **R** is the reward function for a state-action pair; and $\gamma \in (0, 1)$ is the discount factor for future rewards. A stationary policy, π : S $\rightarrow$ A, for a MDP is a mapping from states to actions and denotes a mechanism for choosing actions. An episode can be supposed as a sequence of state transitions: $< s_1, s_2, ..., s_T >$. An agent repeatedly chooses actions until the current episode terminates, followed by a reset to a starting state.

The notion of value function is of central interest in reinforcement learning tasks. Given a policy π, the value $V^{\pi}(s)$ of a state s is defined as the expected discounted returns obtained when starting from this state until the current episode terminates following policy π:

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s, \pi\right]$$

As it is well-known, the value function must obey the Bellman's equation:

$$V^{\pi}(s) = E_{\pi}\left[R(s_t) + \gamma V^{\pi}(s_{t+1}) | s_t = s\right],$$

which expresses a relationship between the values of successive states in the same episode. In the same way, the state-action value function (Q-function), Q(s, a), denotes the expected cumulative reward as received by taking action a in state s and then following the policy π,

$$Q^{\pi}(s,a) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s, a_0 = a\right]$$

In this study, we will focus on the Q functions dealing with state-action pairs (s, a). The objective of RL problems is to estimate an optimal policy π^* by choosing actions that yields the optimal action-state value function Q^*:

$$\pi^*(s) = arg(\max_a Q^*(s,a)).$$

Learning a policy therefore means updating the Q-function to make it more accurate. To account for potential inaccuracies in the Q-function, it must perform occasional exploratory actions. A common strategy is the ε-greedy exploration, where with a small probability ε, the agent chooses a random action. In an environment with a capable (reasonably small) number of states, the Q-function can simply be represented with a table of values, one entry for each state-action pair. Thus, basic algorithmic RL schemes make updates to individual Q-value entries in this table.

4.3.2 Q learning algorithms

Q-learning is a model-free, value-based, off-policy algorithm that will find the best series of actions based on the agent's current state. The "Q" stands for quality. Quality represents how

valuable the action is in maximizing future rewards. The core of the Q learning is based on Q values ,which represents the expected utility (or cumulative reward) of taking a given action in a given state, and then following the optimal policy thereafter. The maximum predicted Q value of the new state is used to calculate an improved estimate for the Q value of the previous state-action pair.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta_t,$$

where

$$\delta_t = (r_t + \max_a \gamma Q(s_{t+1}, a) - Q(s_t, a_t))$$

is known as the one step temporal-difference (TD) error. The term α is the learning rate which is initially set to a large value and can be frequently decreased to small values (e.g. α = 0.1) during the learning process.

The pseudo code for the Q learning for the full process of training is described as follows:

Set values for learning rate α、 discount rate γ、 reward matrix R
Initialize $Q(s,a)$ to zeros
Repeat for each episode,do
    Select state s randomly
    Repeat for each step of episode,do
        Choose a from s using ε-greedy policy or Boltzmann policy
        Take action a obtain reward r from R, and next state s'
        Update $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        Set $s = s'$
        Until s is the terminal state
    End do
End do

Figure 8: Q learning algorithm

4.3.3 The proposed state space representation

The game of Pac-Man constitutes a challenging domain for building and testing intelligent agents. The state space representation is of central interest for an agent, since it plays a significant role in system modeling, identification, and adaptive control. At each time step, the agent has to make decisions according to its observations. The state space model should describe the physical dynamic system and the states must represent the internal behaviour of system by modeling an efficient relationship from inputs to actions. In particular, the description of the

state space in the Pac-Man domain should incorporate useful information about his position, the food (dots, scared ghosts) as well as the ghosts. An ideal state space representation for Pac-Man could incorporate all these information that included in a game snapshot, such as:

- the relative position of Pac-Man in the maze,
- the situation about the food (dots, power dots) around the agent,
- the condition of nearest ghosts.

Although the state space representation constitutes an integral part of the agent, only little effort has been paid in seeking a reasonable and informative state structure. As indicated in above, a full description of the state would include (a) whether the dots have been eaten, (b) the position and direction of Pac-Man, (c) the position and direction of the four ghosts, (d) whether the ghosts are edible (blue), and if so, for how long they remain in this situation. Despite its benefits, the adoption of such a detailed state space representation can bring several undesirable effects (e.g. high computational complexity, low convergence rate, resource demanding, e.t.c), that makes modeling of them to be a difficult task.

According to the above discussion, in our study we have chosen carefully an abstract space description that simultaneously incorporate all the necessary information for the construction of a competitive agent. More specifically, in our approach the state space is structured as a 10-dimensional feature vector, s = $(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10})$ with discrete values. Its detailed description is given below:

- The four (4) features $(s_2, ..., s_5)$ are binary and used to indicate the existence (1) or not (0) of the wall in the Pac-Man's four wind directions (up, left, down, right), respectively. Some characteristic examples are illustrated in Figure 8; state vector $(s_2 = 0, s_3 = 0, s_4 = 1, s_5 = 1)$ indicates that the Pac-Man is found in a corridor with horizontal walls, while state values $(s_2 = 1, s_3 = 1, s_4 = 0, s_5 = 0)$ means that Pac-Man is located between a north and south wall.
- The first feature s_1 suggests the direction of the nearest target where it is preferable for the Pac-Man to move. It takes four (4) values (from 0 to 3) that correspond to up, left, down or right direction, respectively. The desired target depends on the Pac-Man's position in terms of the four ghosts. In particular, when the Pac-Man is going to be trapped by the ghosts (i.e. at least one ghost with distance less than eight (3) steps is moving against Pac-Man), then the direction to the closest safer exit (escape direction) must be chosen. In all other cases this feature takes the direction leading to the closest dot . Note here that for calculating the distance as well as the direction between Pac-Man and target, we have used the known BFS search algorithm for finding the shortest path.
- The next four features $(s_6, ..., s_9)$ are binary and specify the situation of any direction (up, left, down, right) in terms of a direct ghost threat. When a ghost with distance less that three steps (3) is moving towards pac-man from a specific direction, then the corresponding direction is selected as the nearest safe way where there is no ghost. The neareast safe finding starts from the right direction ,and then next to the left,up, and down.

- The last feature s_{10} specifies the distance from the Pacman to the nearest dot. It is calculated through the BFS algorithm at the same time with the fifth feature. We have estimated the maximum possble distance between the Pacman and dots, and we concluded that the appropriate range for this feature is 50.

Table 2 summarizes the proposed state space. Obviously, its size is average, containing $4 \times 2^8 \times 50 = 51200$ states. However, this allows the construction of a computationally efficient RL agent without the need of any approximation scheme. Last but not least, the adopted reasonable state space combined with the small action space speed up the learning process and enables the agent to discover optimal policy solutions with sufficient generalization capabilities.

Feature	Range	Source
s_1	{0, 1, 2, 3}	target direction
$[s_2 \ s_3 \ s_4 \ s_5]$	{0, 1}	wall checking
$[s_6 \ s_7 \ s_8 \ s_9]$	{0, 1}	ghost threat direction
s_{10}	{1,.., 50}	distance to the dot

Table 2: A summary of the proposed state space

Event	Reward	Description
Step	-0.5	Pac-Man move into the empty space
Lose	-100	Pac-Man is eaten by a ghost
Wall	-100	Pac-Man hit the wall
Power	+10	Pac-Man eat a normal dot
Power	+20	Pac-Man eat a big dot

Table 3: The reward function for different game events

5 Experiment

5.1 BFS

In the BFS algorithm, we consider two cases. Case 1 has only 2 ghosts on the map, while case 2 has 4 ghosts on the map. The results show that in case 1, Pacman can win with 447 steps. However, in case 2, Pacman encountered some issues and could not win.

From this, we observe some limitations of the BFS method:

- Not optimal in Changing Environments: BFS cannot quickly adjust or adapt to changes in the environment or game rules. It must restart the entire search process from scratch whenever any changes occur.
- Static Nature: BFS lacks the ability to adapt or learn from the environment. It methodically explores all states without leveraging previous experiences to improve performance.

- Unable to find the optimal path: Pacman's movement directions in the BFS algorithm are fixed within the game.

5.2 Reinforcement Learning

The process of training our RL algorithm follows a two-stage strategy: Exploration and Exploitation. Exploration phase: we had Pacman run randomly on the map for 200 consecutive games to learn about and become familiar with the current environment. It is done by setting up the high epsilon value (0.87 - 1). Exploitation phase: with the experience gained from the Exploration phase, Pacman can now adapt to the current environment and choose appropriate paths to achieve victory by lowering the epsilon value (0.1 - 0.2). Now, We will try training with a loop of 10 games in the Exploitation phase. (The Exploration phase has been previously conducted)

In the loop of 10 games, Pacman can win 5 out of 10 games, which is equivalent to 50%, and the minimum number of steps in the winning games is 356.

Similarly, we increased the number of loops to 50, 100, and 500, respectively, and obtained the following results

Iteration	Win rate	Min step
10	50%	356
50	54%	350
100	57%	344
500	63%	334

The results show that with the RL method, the higher the number of loops, the higher the win rate is, and the fewer steps needed to win.

The results show that RL allows the agent to learn from experience through interactions with the game environment. By optimizing the action policy based on the rewards received, the agent can improve game performance over time and thereby adapting to the current environment, solving situations, and selecting more optimal paths, thus giving better results.

However, despite of the above success, the reinforcement learning also possesses some limitations :

- Challenges in stage specification: To run the algorithm smoothly for a appropriate time, we need to take into account which features to include in a state.If the number of features are too large, the training process will need more time, or even need some other advanced techniques. On the other hand, if the state space is too small, the training is inefficient and the agent cannot win the game, even with long time training.
- Time taking for training: For the agent to play the game, it needs time of training, which can be hundreds, sometimes thousand of games.

- The unstable gameplay : The number of winnings differs dramatically in every different game. Sometimes, the agent might lose the game.

6 Conclusions and state of art

In this article, we used two methods to automate playing the Pacman game: Breadth-first search (BFS) and Reinforcement Learning (RL). Each method was implemented and trained over a long period and on a large scale to ensure the best results. After a series of experiments, we evaluated the performance of the two algorithms based on different criteria such as: Winning time, winning rate, adaptability and pathfinding ability of Pacman. From the results obtained, we conclude that each method has its own advantages and disadvantages depending on each case. The advantage of BFS is that it is easy to implement and can be considered an optimal solution in a simple environment. If in a complex environment, BFS will become ineffective and time-limited. In contrast, RL exhibits superior performance in a complex environment thanks to its ability to learn so that it can adapt to the current environment. However, RL also requires lots of time for training as well as the performance through every game is unstable.

There are many potential directions for future work. For example, in our approach the power dots are not included in the state structure. Intuitively thinking, moving towards the power dots can be seen gainful since it can increase the pacman's life as well as the score. However, there is a trade-off between searching (greedily for food) and defensive (avoiding the ghosts) abilities that must be taken into account. Another alternative is to investigate bootstrapping mechanisms, by restarting the learning process with previously learned policies, as well as to combine different policies that are trained simultaneously so as to achieve improved performance, especially in critical situations of the domain.

References

[1] Goldberg, Marty. *Pac-Man: The Phenomenon: Part 1*, 2022.

[2] Jason Holdsworth. *The Nature of Breadth-First Search*, 1999.

[3] Mnih et al. *Playing Atari with Deep Reinforcement Learning*, 2013.

[4] Nikolaos Tziortziotis, Konstantinos Tziortziotis, and Konstantinos Blekas. *Play Pac-Man using an advanced reinforcement learning agent*.

[5] Nguyen Thi Thuan . *Phuong phap hoc tang cuong* (Master's thesis, Hanoi University of science and technology), 2006