Flappy Bird Reinforcement Learning Based on Unity

By: Yujie Zang 26/11/2024

1 Introduction

1.1 Overview of the project

The project focuses on applying **reinforcement learning** to a classic arcade game, **Flappy Bird**, to enable an AI agent to autonomously learn and improve its gameplay. The game's mechanics provide a simplified, yet effective, environment for training an agent to **maximize its survival in a dynamic, real-time system**.

1.2 Significance of the Project(what you can learn by this project)

The project can help enhance the valuable skills(Unity 2D, Reinforcement Learning) and insights gained through the project and how they contribute to broader AI applications.

1.2.1 Learning and Mastering Unity 2D

This project provides an opportunity to learn and master **Unity 2D**, a powerful game development engine that offers **intuitive tools and interactive components** for creating visually appealing and engaging game environments. Unity's flexibility and ease of use make it **an ideal platform for prototyping and experimenting with AI applications**, particularly in reinforcement learning. By using Unity, you gain hands-on experience with its extensive functionality, from scene creation and object manipulation to scripting and asset management.

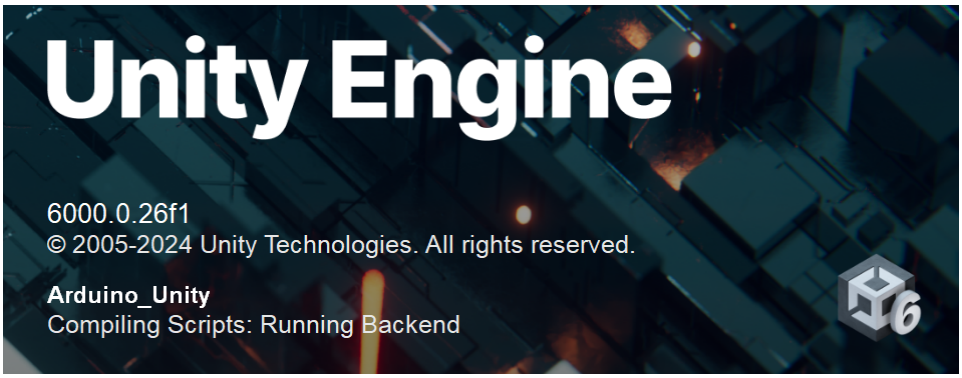


Fig 1: Unity Engine Startup

1.2.2 Learning, Understanding, and Applying Reinforcement Learning

The project deepens the understanding of **reinforcement learning (RL)** principles, such as how agents interact with their environments and learn optimal behaviors through rewards and penalties. By implementing RL algorithms like **Deep Q-Networks (DQN)**, you gain a solid understanding of key concepts like state-action-reward systems, **exploration-exploitation trade-offs, value functions, and policy optimization.** This knowledge is foundational for

applying RL techniques to more complex, real-world problems, where agents must make decisions in dynamic environments.

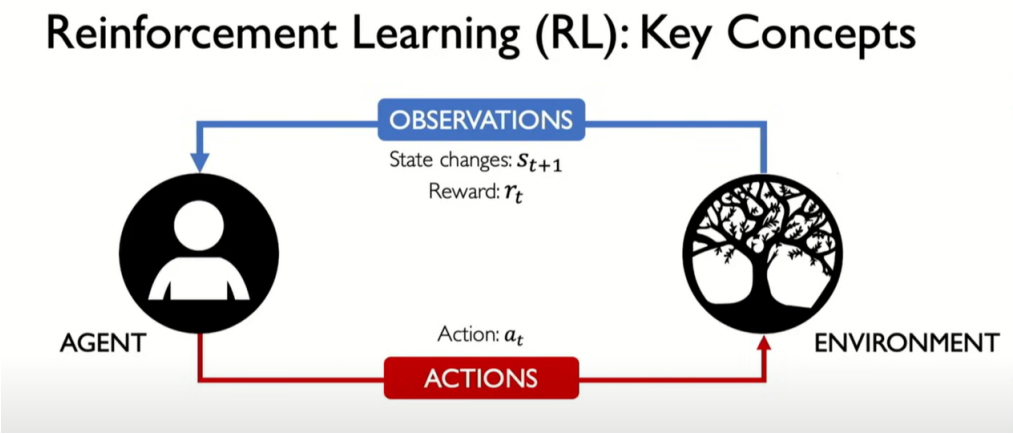


Fig 2: RL Lecture By MIT

1.2.3 Enhancing Conceptual Understanding of RL Applications

Through this project, you deepen your understanding of RL and its real-world applications, which can extend to more advanced technologies, such as **end-to-end autonomous driving systems**. Reinforcement learning is a core technique in training autonomous agents, where decision-making in complex, unpredictable environments is crucial. By training an RL agent in a **game-like environment**, you **gain a conceptual understanding of how such systems can be extended to handle real-world challenges**, providing a foundation for tackling similar problems in autonomous driving, robotics, and other AI-driven technologies in the future.

Fig 3: Autonomous Vehices using End-to-End By MIT

1.3 Structure of the Report:

The report is structured as follows:

- **Introduction:** Introduction to the project and significance.
- **Methodology:** Detailed explanation of how the game environment was created, how Python-Unity integration was achieved, and the implementation of the RL algorithm.
- **Results:** A summary of the training results, including reward curves and agent performance.
- **Discussion:** Insights gained during the project, challenges encountered, and suggested improvements.
- **Conclusion:** A summary of the project's findings and its potential applications.

- 26/11/2024(Tuesday): Do the Framework and improve the environment game created in Lecture Interactive Technology
- 27/11/2024: Do the Report of game development in unity, explore the Reinforcement Learning fundamentals and write the report
- 28/11/2024: do python-unity integration and create the flow of training and testing
- 29: evaluate and write the rest
- 30: release

2 Methodology

2.1 Game Development in Unity

Objective: Create the Flappy Bird game environment in Unity.

2.1.1 Game Mechanics Design

The core mechanics of the Flappy Bird game revolve around simulating realistic physics and player input. Here's a breakdown of the primary mechanics involved in creating the game:

(a) Gravity and Jump Force:

In the Flappy Bird game, the bird experiences **gravity**, which causes it to fall unless the player presses a **button (space key)**. The jump force counteracts gravity, enabling the bird to move upward when the jump action is triggered.

- **Gravity**: Gravity is handled by Unity's built-in physics engine. You can adjust the gravity scale to control how fast the bird falls. This is done through the **Rigidbody2D.gravityScale** property.

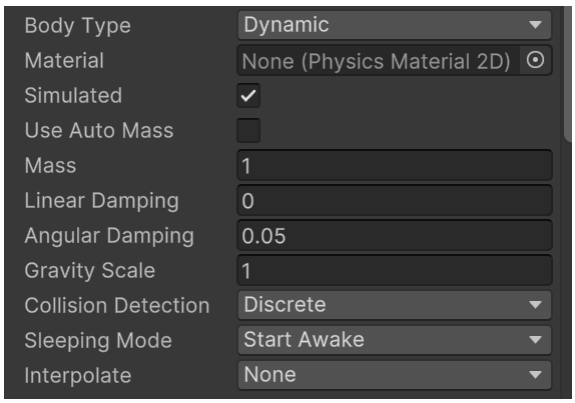


Fig 4: Rigidbody2D Set in Unity

- **Jump Force**: When the bird "flaps," we apply an upward force to the Rigidbody2D. This force counteracts gravity and makes the bird move upward.

```
// Apply upward force when the player presses the space key
if(Input.GetKeyDown(KeyCode.Space) && birdisAlive) {
    myRigidbody.linearVelocity = Vector2.up * fly_strength;
    }
```

(b) Collision Detection and Game Over Conditions

Collision detection is crucial in determining whether the bird collides with pipes, the ground, or the ceiling. Unity handles collision detection using **Collider2D** components.

- **Pipe Collision**: Similarly, the pipes are assigned **BoxCollider2D** components to detect when the bird passes through or hits them. Pipes are dynamically generated and moved from right to left across the screen.

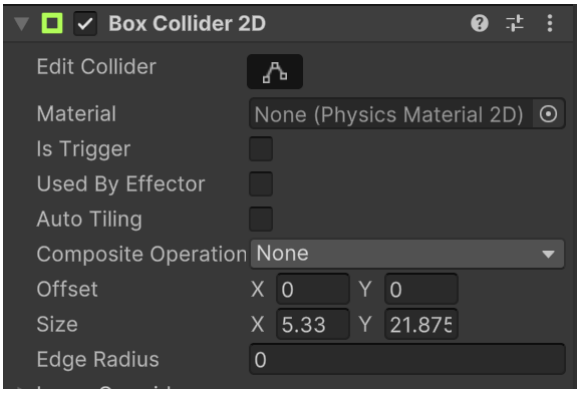


Fig 5: BoxCollider 2D Set in Unity

- **Bird Collision**: The bird is assigned a **BoxCollider2D** to detect when it collides with **obstacles**.

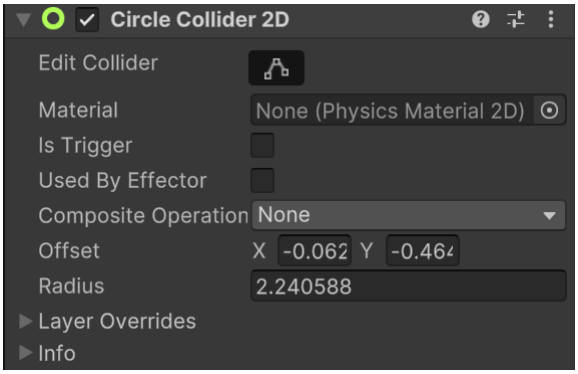


Fig 6: Circle Collider 2D Set in Unity

(c) Game Over Conditions

The game should end when the bird **collides with a pipe or falls out of bounds**.

```
// Obstacles(ceiling and ground)
if (transform.position.y > topBoundary || transform.position.y < bottomBoundary)
{
    logic.GameOver();
    birdisAlive = false;
    }
// Obstacles(Pipes)
private void OnCollisionEnter2D(Collision2D collision) {
    logic.GameOver();
    birdisAlive = false;
    }
```

2.1.2 Object Manipulation

Unity makes it easy to manipulate objects using the **Transform** component which allows us to move, rotate, or scale objects as needed.

(a) Bird Object Manipulation

- The bird is typically a **sprite** (a bird image) attached to a **GameObject**.

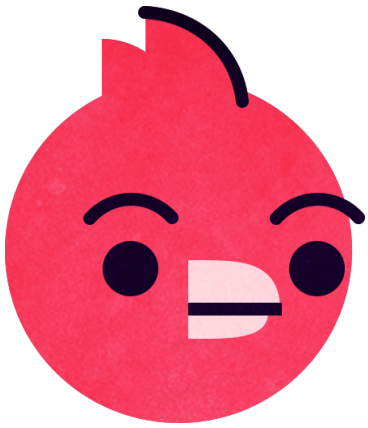


Fig 7: Bird Body in Unity

- To animate the bird's flap, an **Animator** component is used, where a simple animation of the bird flapping its wings can be triggered when the bird jumps.

```
// Trigger flap animation
myAnimator.SetTrigger("WingFlap");
```

Fig 8: Bird Wing Down in Unity

(b) Pipe Object Manipulation

- Pipes are created as **Prefab** and moved across the screen at a fixed rate.

```
// Create a Pipe Prefab Class
public class PipeMove : MonoBehaviour
{
    public float PipeVelocity = 5.0f; // Set the PipeVelocity as bird move
relatively
    public float deadPlace = -12.0f; // Set the dealplace to save the source

    // Update is called once per frame
    void Update()
    {
        transform.position = transform.position + (Vector3.left * PipeVelocity)
* Time.deltaTime;
        if(transform.position.x < deadPlace) {
           Debug.Log("Pipe is Deleted!");
           Destroy(gameObject);
        }
    }
}
```

- The pipes are instantiated in the game scene at random heights to create a challenge for the player/agent.

```
public class PipeGenerate : MonoBehaviour
{
```

```
    public GameObject Pipe;
    public float generate_rate = 5.0f;
    private float timer = 0.0f;
    public float offset = 1.0f;

    // Update is called once per frame
    void Update()
    {
        if(generate_rate > 0 && timer > generate_rate) {
            generatePipe();
            timer = 0.0f;
        }
        else {
            timer += Time.deltaTime;
        }
    }

    void generatePipe() {
        if (Pipe != null) {
            float highestPoint = transform.position.y + offset;
            float lowestPoint = transform.position.y - offset
            float randomY = Random.Range(lowestPoint, highestPoint);

            Instantiate(Pipe, new Vector3(transform.position.x, randomY*20, 0),
transform.rotation);
        } else {
            Debug.LogError("Pipe prefab is not assigned in the inspector!");
        }
    }

}
```

2.1.3 UI Design and Scene

In Unity, the UI system is based on **Canvas**. All UI elements like buttons, text, and images are placed inside a **Canvas** GameObject. And it is why I use unity engine to design the game; just for its powerful interactive UI design

(a) Home Scene

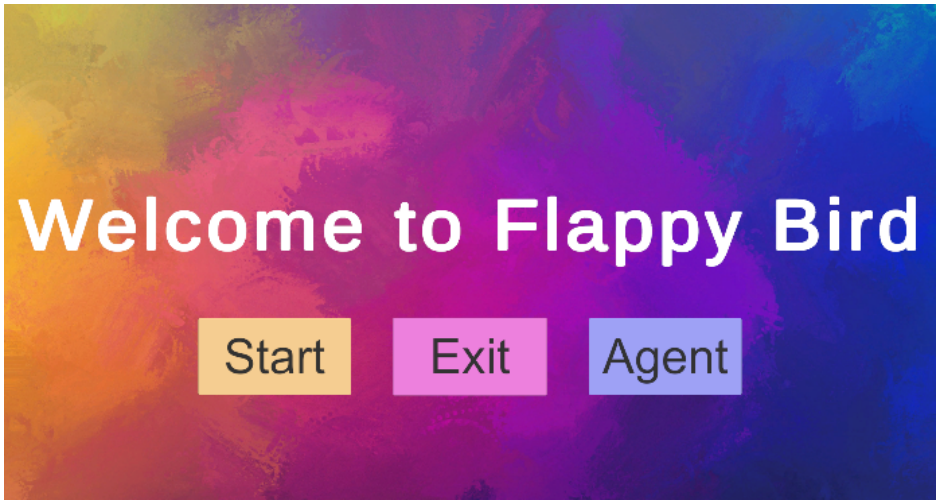


Fig 9: Home Scene

- **Buttons for Start and Exit**: Start the game by loading the in-game scene or Close the game or returns to the main menu

```
public class LogicStartScript : MonoBehaviour
{
    public void StartGame(){
        SceneManager.LoadScene("SampleScene");
        Debug.Log("Load SampleScene successfully");
    }

    public void quit(){
        Application.Quit();
        Debug.Log("Quit");
    }
}
```

- **Button for Agent**: trigger the agent's learning process or start an automated run.

(b) In-Game Scene

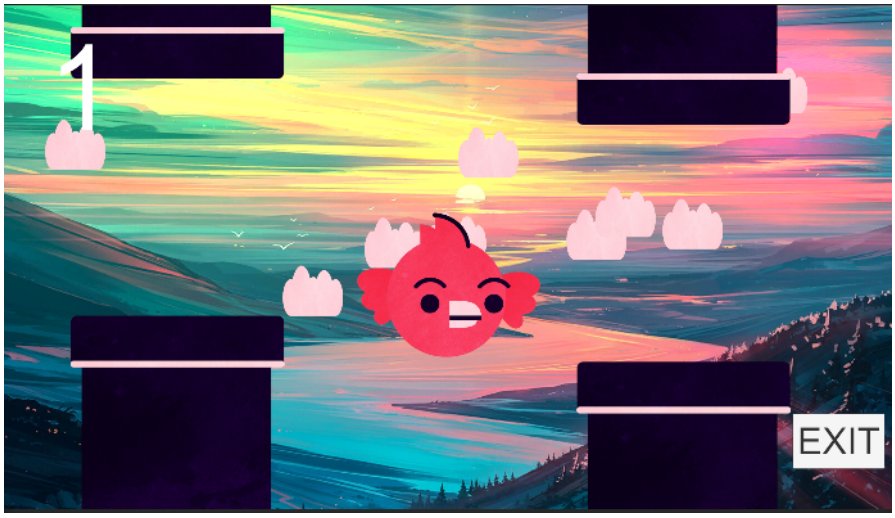


Fig 10: In-Game Scene

- **Score Text**: The score should be visible at the top-left of the screen. Update the score each time the player successfully passes a pipe.
- **Pink Cloud**: This could be a UI decoration or effect to enhance the game's atmosphere.
- **Exit Button**: If the player wants to quit during the game, they should be able to return to the Home Scene.

(c) GameOver Scene

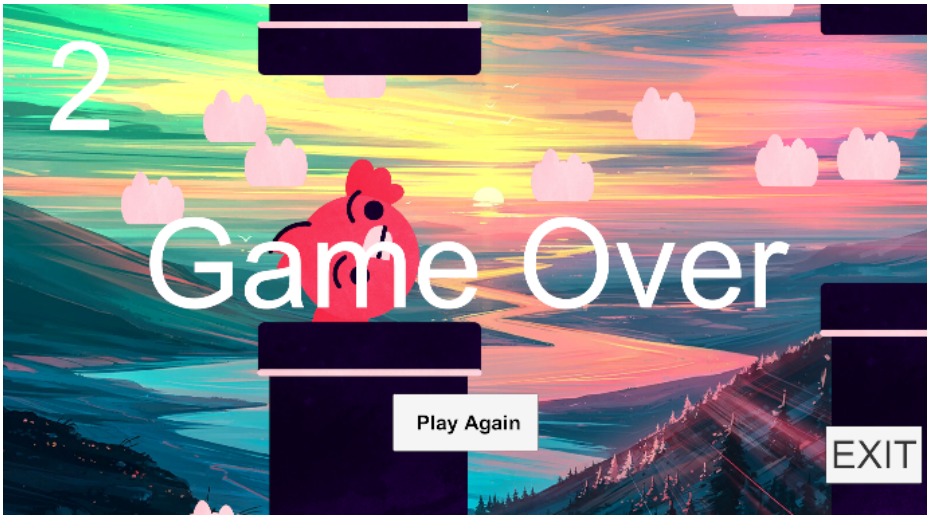


Fig 11: GameOver Scene

- **Game Over Text**: A large **Game Over** message in the center of the screen.
- **Play Again Button**: Do not be Upset and Retry again!
- **Exit Button**: Allows the player to return to the Home Scene or quit the game.

2.2 Reinforcement Learning Setup

Objective: Apply reinforcement learning algorithms** (DQN)** to train an agent (the bird) to play the game.

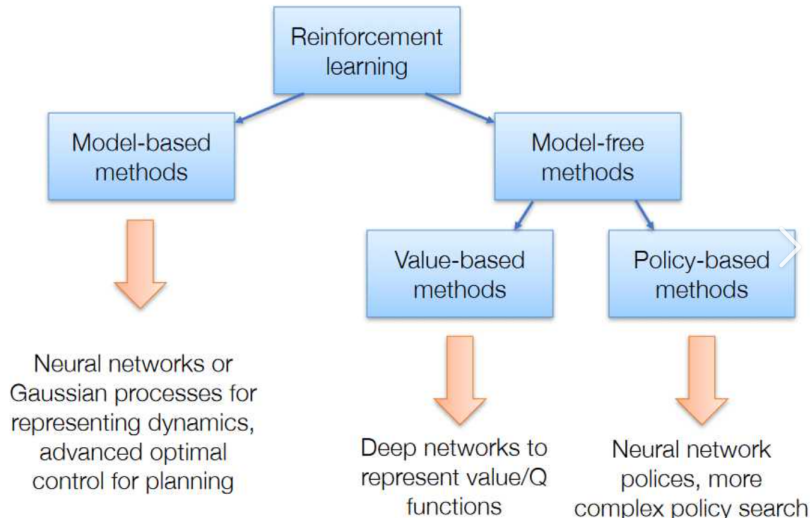


Fig 11: Reinforcement Learning Framework

2.2.1 Mathematical Concepts: Markov Chains, Markov Decision Processes (MDPs) , and Bellman Equation

This section deals with foundational concepts used in **decision-making under uncertainty**, particularly focusing on **Markov Chains**, **Markov Decision Processes (MDPs)**, and the **Bellman Equation**.

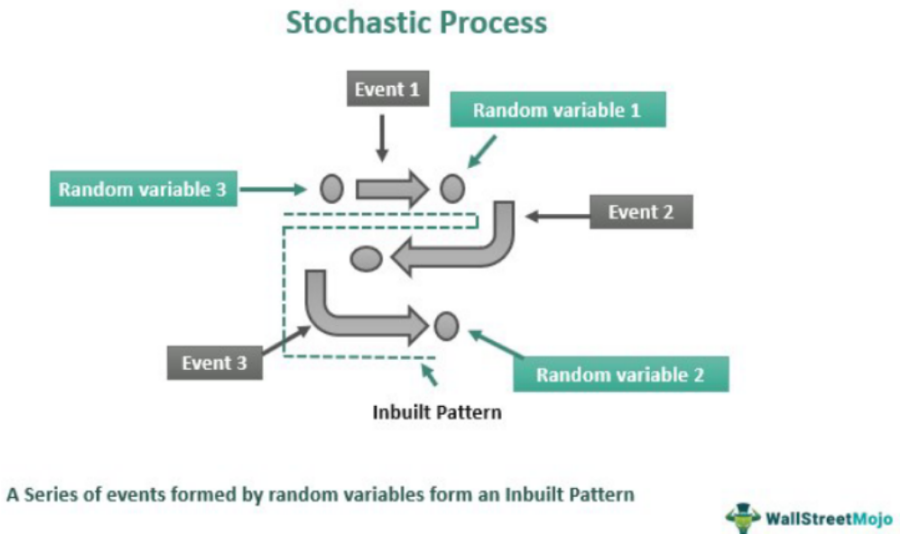


Fig 11: Stochastic Processes

A **stochastic process** is a collection of random variables indexed by time (or another parameter), representing a system or process that evolves over time in a **probabilistic manner**. **Unlike deterministic processes**, where future outcomes are fully determined by the initial conditions, stochastic processes incorporate randomness and uncertainty into their evolution.

(a) Markov Chains

(1) Brief Introduction

Markov Chains are a fundamental part of **stochastic processes**, which use probabilistic models to analyze and predict random systems that evolve over time.

A **Markov Chain** is a mathematical model used to describe a system that transitions from one state to another over time in a probabilistic manner. The key feature of a Markov Chain is the **Markov Property**, which means that the future state of the system depends only on the current state and **not** on the sequence of events that preceded it.

$$\begin{bmatrix} 0.3 & 0.4 & 0.3 & 0 \\ 0.3 & 0.1 & 0.4 & 0.2 \\ 0 & 0 & 0.5 & 0.5 \\ 0.3 & 0.5 & 0.2 & 0 \end{bmatrix}$$

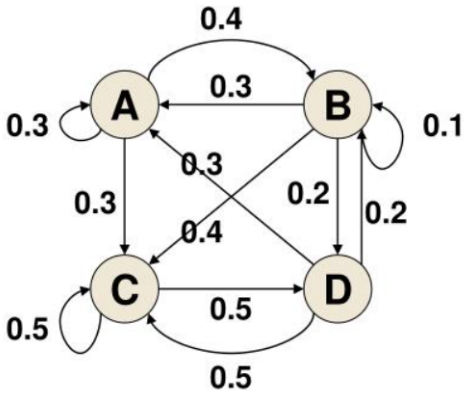


Fig 11: Markov Chain and Transition Matrix

(2) Transition Probabilities

Transition Probability refers to the probability of moving from one state to another in a Markov Chain. Denoted by P(i,j), where P(i,j) is the probability of transitioning from state s_i to state s_j.

$$\sum_j P(i,j) = 1$$

(3) Markov Property (Memoryless Property)

To simplify the model, we only need to track the current state(Reasonable).

The **Markov Property** states that the **future** state of the system depends **only on the current state**, not on the sequence of states that led to the current state:

$$P(X_{t+1} = s_j \mid X_t = s_i, X_{t-1} = s_{i-1}, \ldots, X_0 = s_0) = P(X_{t+1} = s_j \mid X_t = s_i)$$

This is also called the **memoryless property**, as the process doesn't "remember" past states—only the current state matters for predicting the future.

(5) steady-state distribution(Eigen_Function)

A **steady-state distribution** is the distribution of states that the Markov Chain tends to over a long period of time, when it reaches equilibrium. In other words, it represents the long-term behavior of the system.

For a Markov Chain with a **transition matrix**, the steady-state distribution satisfies the equation:

$$\pi P = \pi(Also_lambda)$$
π is the steady-state probability vectorstate.
P is the transition matrix of the system.

(6) Demo

```
import random

# Transition Matrix for Weather (Sunny, Rainy)
P = {
    'Sunny': {'Sunny': 0.8, 'Rainy': 0.2},
    'Rainy': {'Sunny': 0.4, 'Rainy': 0.6}
}

num_days = 2000000

# Function to simulate weather transitions for N days
def simulate_weather(days=10, initial_state='Sunny'):
    states = ['Sunny', 'Rainy']
    current_state = initial_state
    weather_sequence = [current_state]

    for _ in range(days - 1):
        next_state = random.choices(states, weights=[P[current_state]['Sunny'],
P[current_state]['Rainy']])[0]
        weather_sequence.append(next_state)
        current_state = next_state

    return weather_sequence

# Simulate for 10 days starting with Sunny weather
weather_sequence = simulate_weather(num_days, initial_state='Sunny')

# Count the occurrences of Sunny and Rainy days
sunny_days = weather_sequence.count('Sunny')
rainy_days = weather_sequence.count('Rainy')

# Calculate the proportions
sunny_proportion = sunny_days / len(weather_sequence)
rainy_proportion = rainy_days / len(weather_sequence)
```

```
# Print the result
#print("Weather sequence for days:", weather_sequence)
print(f"Proportion of Sunny days: {sunny_proportion:.2f}")
print(f"Proportion of Rainy days: {rainy_proportion:.2f}")
```

When num_days get larger, the result will converge to steady state:

```
(base) PS E:\Unity_PROJECT\Learn_2D\Code> & D:/anaconda3/python.exe "e:/Unity_PROJECT/Learn_2D/Code/Markov Chain Demo.py"
Proportion of Sunny days: 0.66
Proportion of Rainy days: 0.34
(base) PS E:\Unity_PROJECT\Learn_2D\Code> & D:/anaconda3/python.exe "e:/Unity_PROJECT/Learn_2D/Code/Markov Chain Demo.py"
Proportion of Sunny days: 0.67
Proportion of Rainy days: 0.33
(base) PS E:\Unity_PROJECT\Learn_2D\Code> & D:/anaconda3/python.exe "e:/Unity_PROJECT/Learn_2D/Code/Markov Chain Demo.py"
Proportion of Sunny days: 0.67
Proportion of Rainy days: 0.33
```

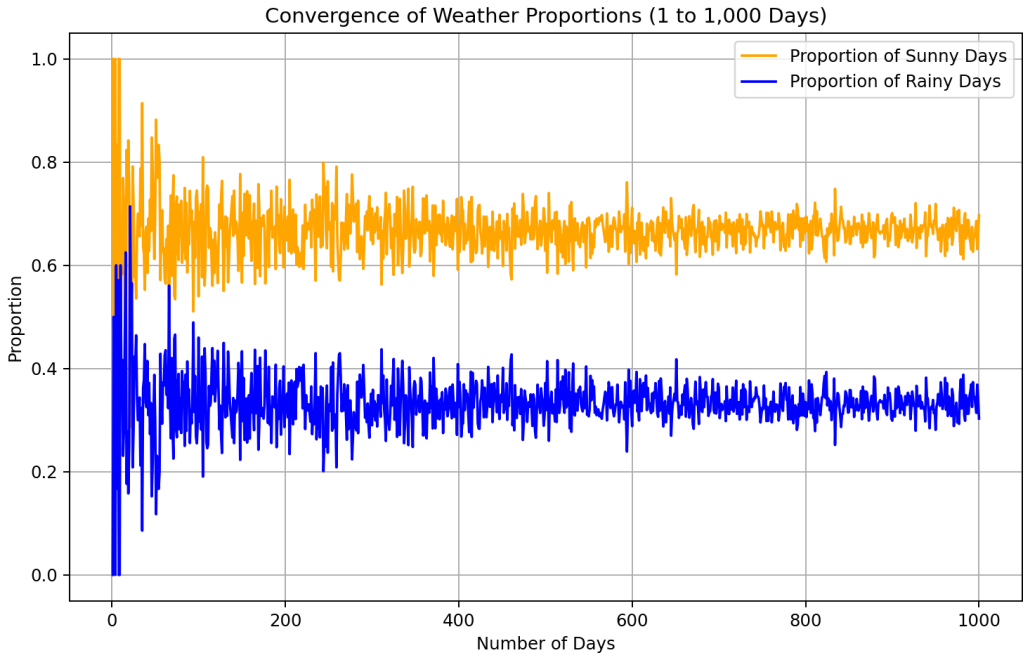


Fig 11: Results for (1,1000), 20000, and 2000000 times

(b) Markov Decision Process(MDPs)

(1) Brief Introduction

A **Markov Decision Process (MDP)**, also stochastic dynamic process, is an extension of the Markov Chain model where **an agent interacts with the environment, making decisions that influence the outcome.** MDPs are used to model sequential decision-making problems where the goal is to find an optimal strategy or policy.

The MDP framework is designed to provide a simplified representation of key elements of AI challenges.

(2) MDPs Framework

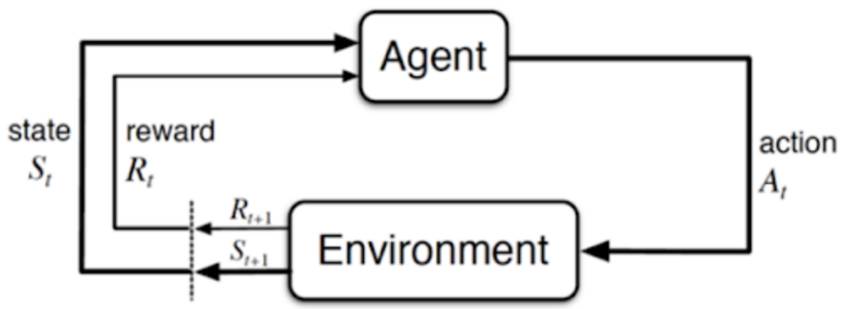


Fig 11: MDPs Framework

- **States** (S): The set of all possible states the environment can be in.
- **Actions** (a∈A): The set of all possible actions the agent can take and a is the specific action
- **Transition Model** (P(s'|s,a)): The probability of transitioning from states s to state s' by taking action a.
- **Reward Function** (R(s,a,s')): The reward the agent receives for transitioning from state s to state s' by taking action a.
- **Discounting Factor** (γ): A factor between 0 and 1 that discounts future rewards, making them less significant than immediate rewards.

Process:

At each step, the agent observes the **current state s_t**, takes an **action a_t**, receives a reward **R(s_t,a_t)**, and transitions to a new **state s_t+1** according to the transition model **P(s'|s,a)**.

The agent's goal is to maximize the **total accumulated reward** over time, which is formalized using the **value function** and **Bellman Equation**.

(c) Value Function and Bellman Equation

(1) State-Value Function(Vπ(s)): (evaluate current s <- future reward in policy π)

State-Value Function reflects how good or bad a state is, or **how much reward you can expect to get in the long run if you follow a policy from a state.** To **evaluate how good a strategy** is. If a state s, The value of s $V\pi(s)$ is high, indicating that starting from this state, it is expected to receive more reward, so this state is "valuable"

The state value function under policy π, denoted Vπ(s), represents the expected cumulative reward starting from state s and following policy π:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s\right]$$

- **R_t** is the reward at time step t.

(2) Bellman Equation for the Value Function

As State-Value Function is simply a **summary statistic** that tells you how "valuable" a state is, based on future expected rewards. The **Bellman equation** provides the **recursive relationship** to compute Vπ(s), breaking it down into immediate rewards and the discounted future value of reachable states.

It essentially breaks down the value of a state into the **immediate reward**(for taking an action from that state), plus the **expected value of the next state**, considering the probabilities of transitioning to different states and the expected future rewards from there.

The **Bellman equation** for the value function Vπ(s) under a policy π is:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\left[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^{\pi}(s')\right]$$

- **R(s,a)** : the immediate reward when taking action a in state s.
- **P(s'|s,a)**: the transition **probability** to state s' after taking action a in state s.
- **Vπ(s')**: the value-state of the next state s', given that the agent follows policy π.

Q&A:

1.What is the significance of the Bellman Equation?

The Bellman Equation provides a **recursive way** to compute the value of a state based on **immediate rewards and future state values.** It forms the foundation for dynamic programming and reinforcement learning algorithms by allowing us to evaluate policies and find optimal solutions through **policy evaluation** and **policy improvement**. It helps in computing the expected long-term rewards and optimal strategies for an agent.

2.Why doesn't the State-Value Function involve actions, while the Bellman Equation does?

The State-Value Function Vπ(s) only measures the expected cumulative reward starting from a state s, given a specific policy π, without considering the actions taken. It evaluates the overall "goodness" of a state under the policy.
The Bellman Equation, on the other hand, involves actions because it calculates the expected reward and value recursively, considering the immediate reward from an action and the future value of resulting states.

3.Why does the Bellman Equation consider only the next state and not further states

The Bellman Equation is recursive, relying on the **Markov Property**, which states that the value of a state only depends on **the current state and action, not on previous states.** It models the decision-making process step-by-step, where future rewards are discounted over time. It focuses on the immediate next state to simplify the decision process, as future states are indirectly accounted for through the recursive process.

(3) Bellman Optimality Equation

The **Bellman optimality equation** gives the value of state s under the **optimal policy** π*, where the agent chooses the best possible action to maximize expected future rewards:

$$V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s') \right)$$

(4) Bellman Equation for Policy Evaluation

The **Bellman equation for policy evaluation** is used to compute the value function Vπ(s) for a given policy π:

$$V^\pi(s) = \sum_a \pi(a \mid s) \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s') \right)$$

- π(a|s) is the probability of taking action a in state s according to policy π\piπ.
- The equation computes the expected value of taking action a in state s, considering both immediate rewards and future rewards.

(5) Value Iteration

Value iteration is the process of iteratively updating the value function V(s) using the Bellman optimality equation until it converges. Once the value function converges, the optimal policy can be derived.

Update rule for value Iteration:

$$V(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right)$$

After convergence, the optimal policy π*(s) is given by:

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s') \right)$$

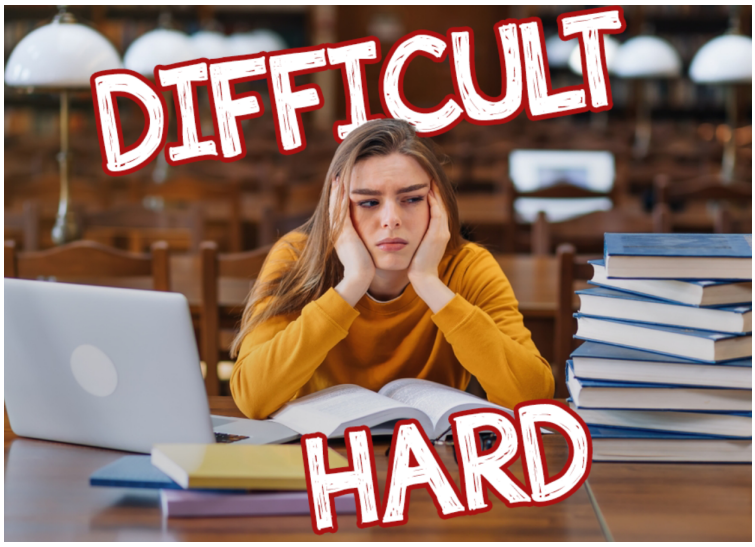


Fig 17: Oh No!!!

2.2.2 Preloading and Fundamentals: Problem-Solving Algorithms

Understanding problem-solving algorithms is crucial for designing efficient reinforcement learning models. Several algorithms play an important role in RL and can be applied to Flappy Bird:

(a) Greedy Algorithms: Local optimal solution -> Global optimal solution

> **Picking the shiniest coin you see on the ground, hoping it's the most valuable, but sometimes missing the treasure chest just a few steps ahead.**

> **Only children make a single choice, adults want everything at every steps**

Fig 17: Hahaha

(1) Brief Introduction

A **greedy algorithm** is a problem-solving strategy that makes a sequence of choices, each of which appears to be the **best option at that moment**. It makes these decisions based on **local optimality**, meaning it chooses the option that seems best for the current situation, without considering the bigger picture or how future decisions might affect the outcome.

The key idea behind a greedy algorithm is the assumption that **locally optimal choices** will lead to a **globally optimal solution**. However, this is not always the case, and while greedy algorithms can be fast and simple, they may **miss the best solution** by focusing on short-term gains rather than long-term benefits.

Fig 17: Greedy Algorithm

(2) General Structure

- **Local Optimality**: The algorithm picks the best immediate option, hoping that this choice leads to the best overall result.
- **Greedy Choice Property**: The choice made at each step must be locally optimal, and this approach works only if these local choices can lead to a global optimum.
- **No Backtracking**: Once a decision is made, it's final, and the algorithm doesn't revisit previous decisions. This makes it faster but potentially less accurate in finding the optimal solution.
- **No Long-Term Planning**: It doesn't always look ahead to the future, and sometimes this lack of foresight can lead to suboptimal solutions (depending on the problem).

(3) Classical Greedy Algorithm Problems

1 **Lemonade Change**

> At a lemonade stand, each lemonade costs `$5`. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a `$5`, `$10`, or `$20` bill. You must provide the correct change to each customer so that the net transaction is that the customer pays `$5`.
>
> Note that you do not have any change in hand at first.
>
> Given an integer array `bills` where `bills[i]` is the bill the `ith` customer pays, return `true` *if you can provide every customer with the correct change, or* `false` *otherwise*.
>
> **Example 1:**

```
Input: bills = [5,5,5,10,20]
Output: true
Explanation:
From the first 3 customers, we collect three $5 bills in order.
From the fourth customer, we collect a $10 bill and give back a $5.
From the fifth customer, we give a $10 bill and a $5 bill.
Since all customers got correct change, we output true.
```

Example 2:

```
Input: bills = [5,5,10,10,20]
Output: false
Explanation:
From the first two customers in order, we collect two $5 bills.
For the next two customers in order, we collect a $10 bill and give back a
$5 bill.
For the last customer, we can not give the change of $15 back because we
only have two $10 bills.
Since not every customer received the correct change, the answer is false.
```

Solution

Greedy(Local Optimazation):N5 > N10 > N20

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int N5 = 0, N10 = 0, N20 = 0;
        for(int i = 0; i < bills.size(); i++) {
            if(bills[i] == 5) N5++;
            else if(bills[i] == 10) {
                if(--N5 < 0) return false;
                N10++;
            }
            else {
                if(N10 > 0) {
                    if(--N5 < 0) return false;
                    N10--;
                }
                else {
                    N5 -= 3;
                    if(N5 < 0) return false;
                }
            }
        }
        return true;
    }
};
```

** 2 Assign Cookies**

> Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

> Each child `i` has a greed factor `g[i]`, which is the minimum size of a cookie that the child will be content with; and each cookie `j` has a size `s[j]`. If `s[j] >= g[i]`, we can assign the cookie `j` to the child `i`, and the child `i` will be content. Your goal is to maximize the number of your content children and output the maximum number.
>
> **Example 1:**
>
> ```
> Input: g = [1,2,3], s = [1,1]
> Output: 1
> Explanation: You have 3 children and 2 cookies. The greed factors of 3
> children are 1, 2, 3.
> And even though you have 2 cookies, since their size is both 1, you could
> only make the child whose greed factor is 1 content.
> You need to output 1.
> ```
>
> **Example 2:**
>
> ```
> Input: g = [1,2], s = [1,2,3]
> Output: 2
> Explanation: You have 2 children and 3 cookies. The greed factors of 2
> children are 1, 2.
> You have 3 cookies and their sizes are big enough to gratify all of the
> children,
> ```

Solution

Greedy(Local Optimization): Give priority to children with small stomachs

```cpp
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end()); sort(s.begin(), s.end());
        int child_num = 0, cookie_num = 0;
        while(child_num < g.size() && cookie_num < s.size()) {
            if(g[child_num] <= s[cookie_num]){
                child_num++;
                cookie_num++;
            }
            else cookie_num++;
        }
        return child_num;
    }
};
```

3 Non-overlapping Intervals

> Given an array of intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.
>
> **Note** that intervals which only touch at a point are **non-overlapping**. For example, `[1, 2]` and `[2, 3]` are non-overlapping.
>
> **Example 1:**

```
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-
overlapping.
```

Example 2:

```
Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals
non-overlapping.
```

Example 3:

```
Input: intervals = [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're
already non-overlapping.
```

Solution

Greedy(Local Optimization): choose the min_Length if overlapping, thereby minimizing overlaps and maximizing the number of non-overlapping intervals.

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());
        int n = intervals.size();
        int base = 0, checker = 1, ans = 0;
        while(checker < n) {
            if(intervals[base][1] <= intervals[checker][0]) {
                base = checker;
                checker++;
            }
            else {
                base = intervals[base][1] < intervals[checker][1] ? base :
checker;
                checker++;
                ans++;
            }
        }
        return ans;
    }
};
```

4 Jump Game: Interesting and more versions

55. Jump Game	6763	43.5%
2297. Jump Game VIII	20	51.3%
1871. Jump Game VII	207	29.4%
1696. Jump Game VI	302	44.3%
1340. Jump Game V	148	60.6%
1345. Jump Game IV	332	45.3%
1306. Jump Game III	507	59.0%
45. Jump Game II	4760	44.5%

Fig 17: Jump Game Problems

> You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.
>
> Return `true` *if you can reach the last index, or* `false` *otherwise*.
>
> **Example 1:**
>
> ```
> Input: nums = [2,3,1,1,4]
> Output: true
> Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
> ```
>
> **Example 2:**
>
> ```
> Input: nums = [3,2,1,0,4]
> Output: false
> Explanation: You will always arrive at index 3 no matter what. Its maximum
> jump length is 0, which makes it impossible to reach the last index.
> ```

Solution

Greedy(Local Optimization): At each index, instead of simply jumping the maximum possible distance from that index, we aim to maximize the reachable index

```cpp
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n = nums.size(), step = 0, max_zone = 0;
        while(step < n) {
            if(max_zone < step) break;
            max_zone = max(nums[step] + step, max_zone);
            step++;
        }
        return step == n? true : false;
    }
};
```

(4) Conclusion on Greedy Algorithm

1.Greedy algorithms are a powerful tool for **solving optimization problems** where making a locally optimal choice at each step leads to a globally optimal solution.

2.While they are efficient and easy to implement, they may not always provide the optimal solution for every problem.

3.It's important to analyze the problem carefully and ensure that the **greedy choice property** holds before relying on this approach. **But, it is always hard to prove but try over and over again**

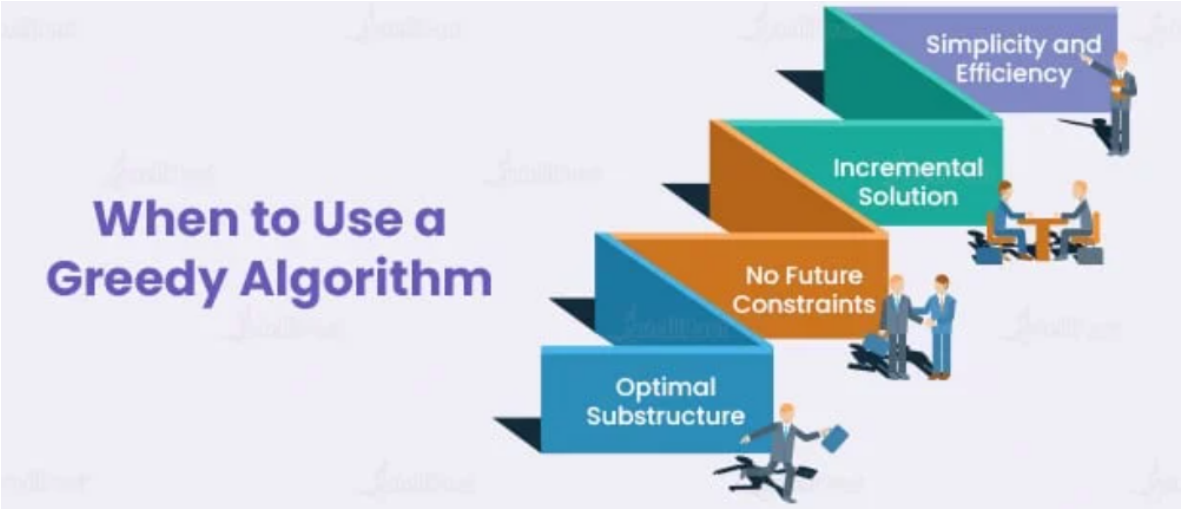


Fig 17: Greedy Algorithm

Strengths

- **Efficiency**: Greedy algorithms are often **fast and simple**, especially for problems where the greedy choice leads to a correct solution.
- **Simplicity**: They typically do not require complex structures or recursion, making them easier to implement compared to dynamic programming or other advanced techniques.

Limitations

- **Not Always Optimal**: The main limitation of greedy algorithms is that they **do not guarantee** an optimal solution in all cases. The assumption that making the locally optimal choice leads to a globally optimal one doesn't always hold true.
- **Greedy failure**: Problems like **Interval Scheduling** or **Fractional Knapsack** may fail with a greedy approach if the local choices don't account for the entire problem's context.

(5) GA and RL

In **Reinforcement Learning (RL)**, **greedy algorithms** are often used in decision-making strategies, specifically in the context of **policy improvement** and **exploration-exploitation trade-offs**

- **Greedy Policy in RL**: In Reinforcement Learning, an agent learns to make decisions by interacting with an environment to maximize cumulative reward over time. A **greedy policy** is one where the agent always selects the action that maximizes its immediate reward based on its current knowledge, i.e., the agent chooses the action that seems best in the current state without considering any future consequences.

  Example： Q-value

If the agent has a Q-value function (a measure of expected future reward for each state-action pair), the **greedy policy** would choose the action **a** that maximizes the Q-value in a given state **s**:

$$a = \arg\max_a Q(s, a)$$

In this case, the agent is being "greedy" by always choosing the action that seems to provide the highest reward according to its current estimates.

- **Exploration vs. Exploitation**: One of the key challenges in RL is balancing **exploration** (trying new actions to gather information) and **exploitation** (choosing the best-known action based on the current knowledge). A **greedy algorithm** tends to exploit the best-known action, potentially neglecting exploration of other actions that could lead to better long-term outcomes.

  Example： ε-Greedy Policy

  - With probability 1−ϵ, the agent selects the **greedy action** (exploitation).
  - With probability **ϵ**, the agent selects a **random action** (exploration).

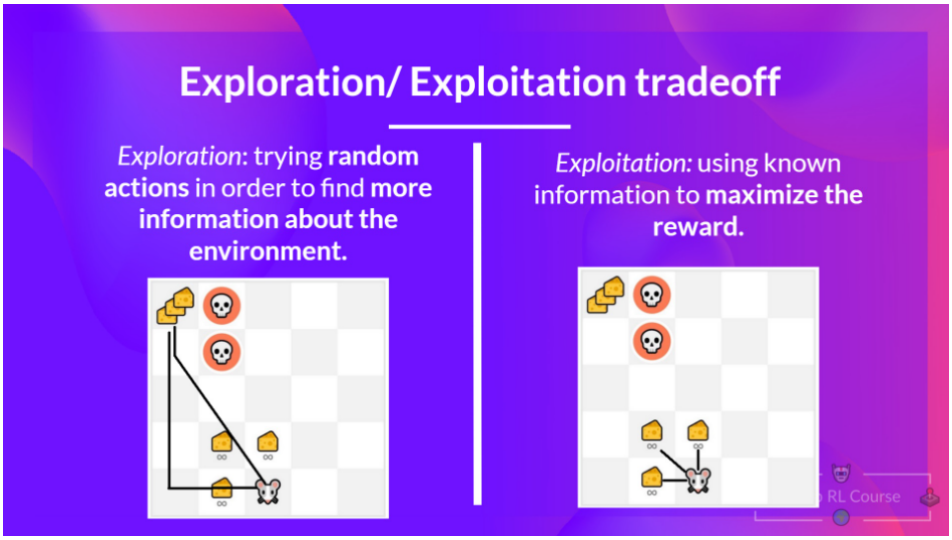
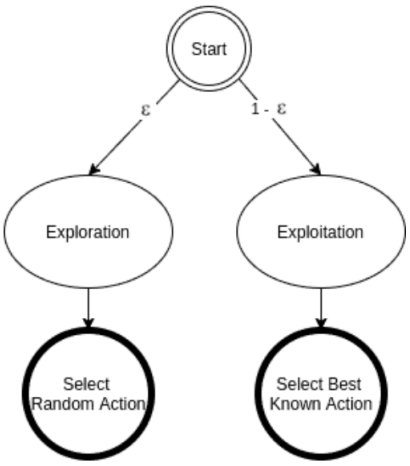


Fig 12: GA and Tradeoff

While **pure greedy algorithms** don't involve exploration, there are situations where greedy algorithms can be **modified** to introduce some form of exploration. This is often done to make the algorithm more flexible or to avoid getting **stuck in a suboptimal solution** (a local optimum).**A small epsilon** means the algorithm will mostly exploit the best-known option (greedy behavior), but with a small chance to explore new options.



(b) Dynamic Programming(DP): State Design, State Transition — Unknowns Derived from Known Values

> **"Those who forget the past are doomed to repeat it."**

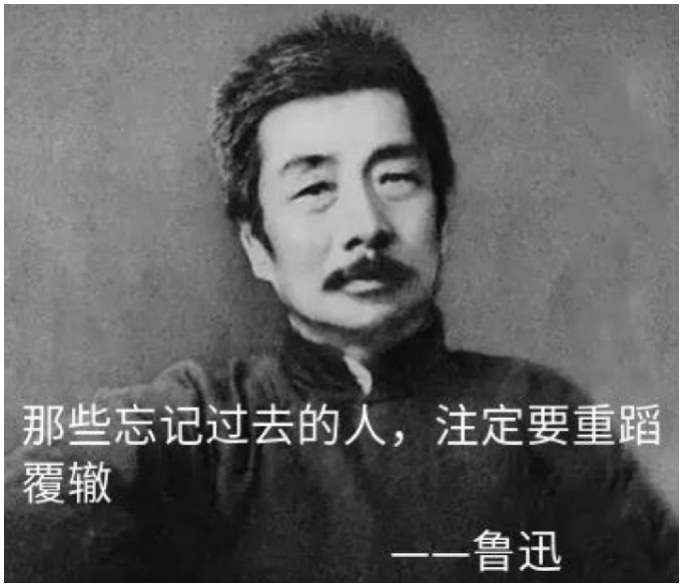


Fig 12: A Chinese man: Xun LU

(1) Brief Introduction

Greedy focuses on **local decisions** (good at the moment), whereas **DP** considers the entire problem for a **global optimal solution**. Dynamic Programming (DP) is a powerful optimization technique used to solve complex problems by **breaking them down into simpler subproblems**. It is particularly useful for problems that exhibit two key properties:

1. **Optimal Substructure**: The problem can be broken down into smaller subproblems, and the solution to the overall problem can be constructed from the solutions to these subproblems.

   Example

   > Consider the problem of **finding the minimum cost path in a weighted graph from a source node to a destination node.** We can break this problem down into smaller subproblems:
   >
   > - Find the minimum cost path from **the source node to each intermediate node**.
   > - Find the minimum cost path from each **intermediate node to the destination node**.
   >
   > -> The solution to the larger problem (finding the minimum cost path from the source node to the destination node) can be constructed from the solutions to these smaller subproblems.

2. **Overlapping Subproblems**: The subproblems recur multiple times, meaning that the same subproblem needs to be solved multiple times during the computation.

   Example

   > *Consider the problem of computing the ***Fibonacci series***. To compute the Fibonacci number at index ***n***, we need to compute the Fibonacci numbers at indices ***n-1*** and ***n-2***. This means that the subproblem of computing the Fibonacci number at index ***n-1*** is used twice in the solution to the larger problem of computing the Fibonacci number at index ***n***.*

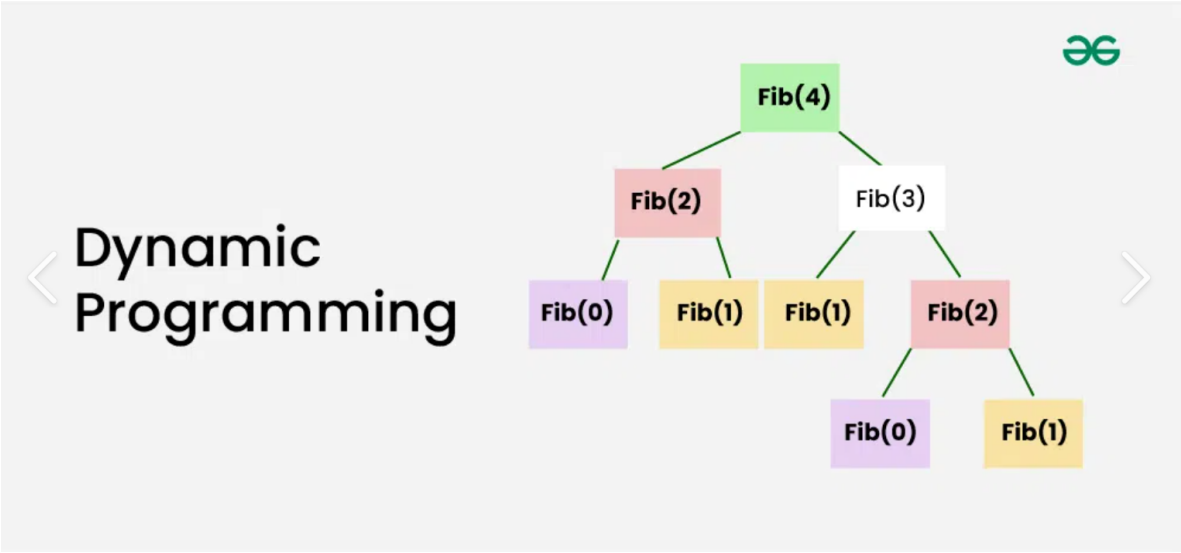


Fig 13: Dynamic Programming

(2) Two Thought and Structure

- **Memoization (Top-Down Approach)**: This involves solving the problem **recursively** but **storing the results of subproblems in a cache (usually a dictionary or array),** so each subproblem is solved only once.
- **Tabulation (Bottom-Up Approach)**: This approach solves all subproblems **iteratively**, starting with the smallest subproblem and **building up** to the solution of the original problem. It uses a table to store results and avoids recursion.

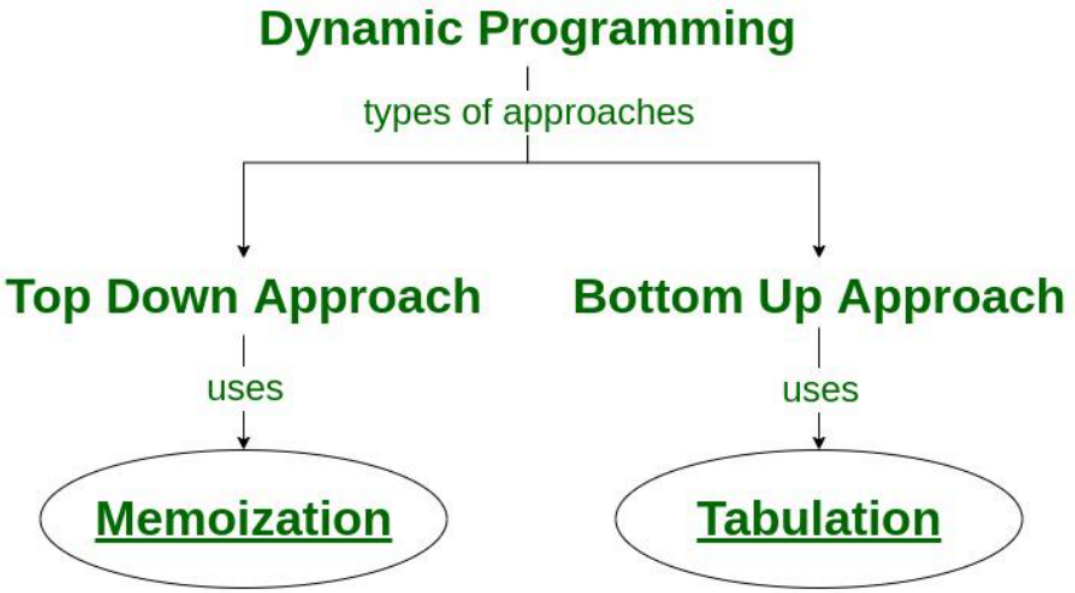


Fig 13: Two Structure of DP

(3) Classical DP Problems

1 [Contiguous Sequence LCCI]

> You are given an array of integers (both positive and negative). Find the contiguous sequence with the largest sum. Return the sum.
>
> **Example:**

```
Input:  [-2,1,-3,4,-1,2,1,-5,4]
Output:  6
Explanation:  [4,-1,2,1] has the largest sum 6.
```

> **Follow Up:**
>
> If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution:

$$\text{State Transition:}$$
$$f(i) = max\{f(i-1) + nums[i], nums[i]\}$$

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int contain = nums[0], ans = nums[0];
        for(int i = 1; i < nums.size(); i++) {
            contain = max(nums[i], contain + nums[i]);
            ans = max(contain, ans);
        }
        return ans;
    }
};
```

2 House Robber : Interesting and more update version

> You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.
>
> Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.
>
> **Example 1:**
>
> ```
> Input: nums = [1,2,3,1]
> Output: 4
> Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
> Total amount you can rob = 1 + 3 = 4.
> ```
>
> **Example 2:**
>
> ```
> Input: nums = [2,7,9,3,1]
> Output: 12
> Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house
> 5 (money = 1).
> Total amount you can rob = 2 + 9 + 1 = 12.
> ```

Solution:

$$\text{State Transition:}$$
$$f(i) = max\{f(i-2) + nums[i], f(i-1)\}$$

```cpp
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if(n == 1) return nums[0];
        vector<int> dp(n+1, 0);
        dp[0] = nums[0]; dp[1] = max(nums[0],nums[1]);
        for(int i = 2; i < nums.size(); i++) {
            dp[i] = max(dp[i-2] + nums[i], dp[i-1]);
        }
        return dp[n-1];
    }
};
```

3 Unique Paths

> There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.
>
> Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.
>
> The test cases are generated so that the answer will be less than or equal to `2 * 109`.
>
> **Example 1:**
>
> 
>
>
> ```
> Input: m = 3, n = 7
> Output: 28
> ```
>
> **Example 2:**
>
> ```
> Input: m = 3, n = 2
> Output: 3
> Explanation: From the top-left corner, there are a total of 3 ways to reach
> the bottom-right corner:
> 1. Right -> Down -> Down
> 2. Down -> Down -> Right
> 3. Down -> Right -> Down
> ```
>
> **Constraints:**
>
> - `1 <= m, n <= 100

Solutions:

$$\text{State Transition:}$$
$$dp[i][j] = dp[i][j-1] + dp[i-1][j]$$

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        if(m == 1 || n == 1) return 1;
        vector<vector<int>> dp(m, vector<int>(n, 1));
        for(int i = 1; i < m; i++) {
            for(int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1];
    }
};
```

```
## permutation and combination
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        return int(math.factorial(m+n-2)/math.factorial(m-1)/math.factorial(n-1))
```

4 Longest Common Subsequence

> Given two strings `text1` and `text2`, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return `0`.
>
> A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.
>
> - For example, `"ace"` is a subsequence of `"abcde"`.
>
> A **common subsequence** of two strings is a subsequence that is common to both strings.
>
> **Example 1:**
>
> ```
> Input: text1 = "abcde", text2 = "ace"
> Output: 3
> Explanation: The longest common subsequence is "ace" and its length is 3.
> ```
>
> **Example 2:**
>
> ```
> Input: text1 = "abc", text2 = "abc"
> Output: 3
> Explanation: The longest common subsequence is "abc" and its length is 3.
> ```
>
> **Example 3:**
>
> ```
> Input: text1 = "abc", text2 = "def"
> Output: 0
> Explanation: There is no such common subsequence, so the result is 0.
> ```

> **Constraints:**
>
> - `1 <= text1.length, text2.length <= 1000`
> - `text1` and `text2` consist of only lowercase English characters.

Solution:

State Definition: dp[i][j]=the length of the longest common subsequence of text 1[0:i] and text2[0:j]
State Transition:$dp[i][j] = dp[i-1][j-1] + 1$when text1[i−1]=text2[j−1]
$dp[i][j] = max\{dp[i-1][j], dp[i][j-1]\}$when text1[i−1] != text2[j−1]

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int len1 = text1.size(), len2 = text2.size();
        vector<vector<int>> dp(len1, vector<int>(len2));
        dp[0][0] = text1[0] == text2[0];
        for(int i = 1; i < len1; i++) {
            if(text1[i] == text2[0]) dp[i][0] = 1;
            else dp[i][0] = dp[i - 1][0];
        }
        for(int i = 1; i < len2; i++) {
            if(text1[0] == text2[i]) dp[0][i] = 1;
            else dp[0][i] = dp[0][i - 1];
        }

        for(int i = 1; i < len1; i++) {
            for(int j = 1; j < len2; j++) {
                if(text1[i] == text2[j]) dp[i][j] = dp[i - 1][j - 1] + 1;
                else {
                  dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[len1 - 1][len2 - 1];
    }
};
```

(4) Conclusion on Dynamic Programming

Difference between Greedy and Dynamic Programming

- **1). Greedy algorithm works when the problem has **Greedy Choice Property** and **Optimal Substructure**, Dynamic programming also works when a problem has optimal substructure but it also requires **Overlapping Subproblems****.
- 2). In greedy algorithm each local decision leads to an optimal solution for the entire problem whereas in dynamic programming solution to the main problem ***depends*** on the ***overlapping subproblems***.

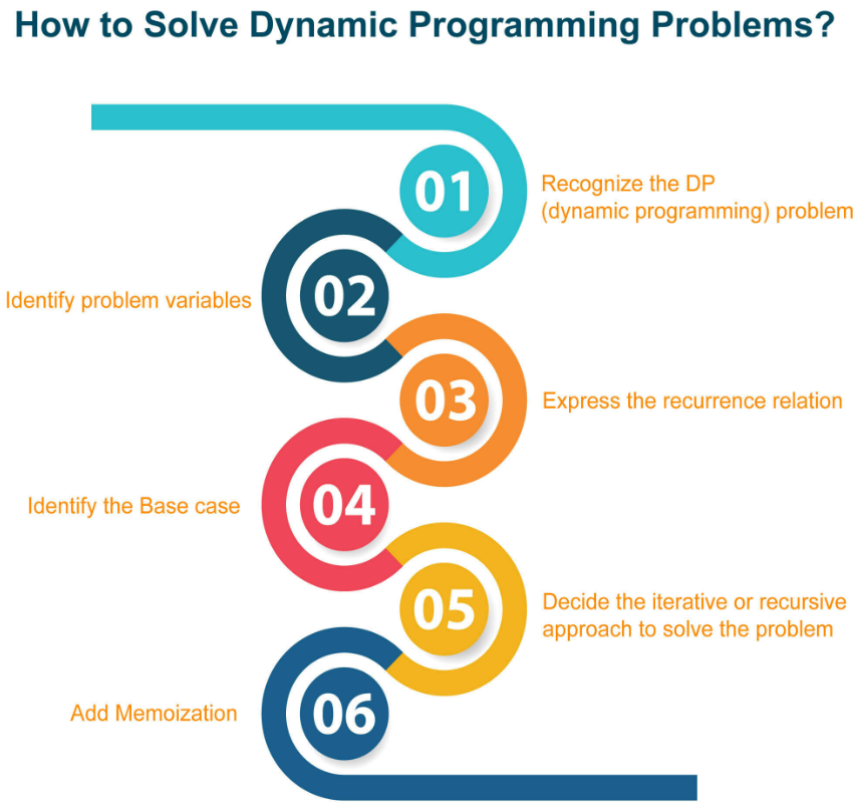


Fig 15: How to solove Dynamic Programming Problems

How to Design States and State Transitions in DP

The key to solving DP problems is designing the **state**, which represents a subproblem or a specific point in the problem-solving process.

- **State Definition**: Each state corresponds to a decision or a combination of variables that influence the outcome of the problem. The challenge is to decide what parameters (variables) will fully capture the necessary information for the current subproblem.
    - **Example in "House Robber" Problem**: The state could represent the maximum money robbed up to house iii. This means you must keep track of the value of each house and whether it is robbed or skipped.
- **State Transition**: The transition depends on previously computed solutions (known values) to derive the solution for the current subproblem (unknown). This is often done using a recurrence relation or a decision rule that links the current state to past states.
    - **Example in "Longest Common Subsequence"**: If characters in both strings match at position i and j, the state transition equation is:

      $$dp[i][j] = dp[i-1][j-1] + 1$$

      Otherwise, it takes the maximum of either ignoring the current character in `text1` or `text2`:

      $$dp[i][j] = max(dp[i-1][j], dp[i][j-1])$$

Dimensions of Array

- **1D Array (Space Optimization)**: In many DP problems, especially those where each state depends only on the immediately preceding state (e.g., in problems like "House Robber"), it's possible to optimize the space complexity by using a **1D array** rather than a 2D array.
- **2D Array (Full State Storage)**: If each state depends on multiple previous states, you'll need a **2D array** (or higher dimensions, depending on the problem). For example, in problems involving multiple dimensions like in the "Longest Common Subsequence" or "Unique Paths" problems, a 2D array is essential to store the solutions for all subproblems.

(5) DP and RL

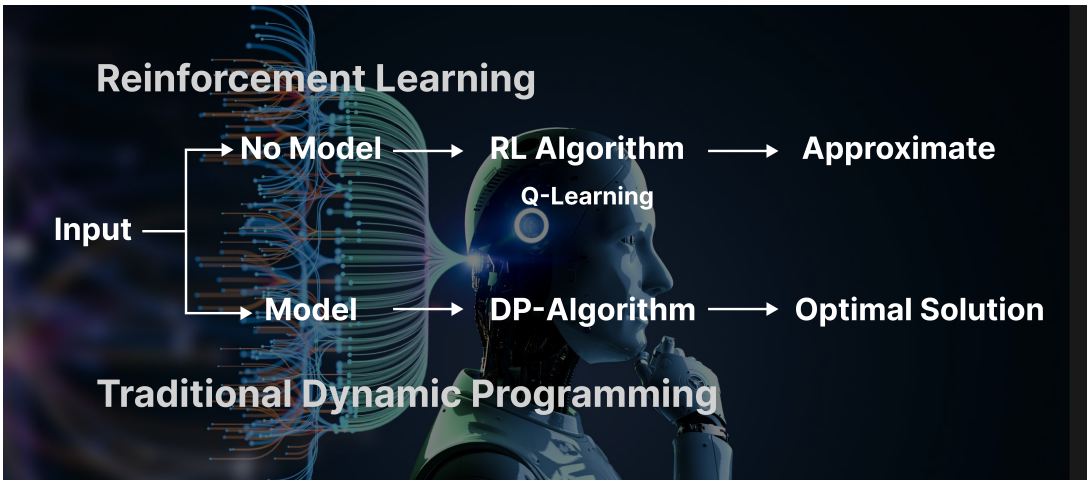


Fig 16: RL and DP

Both are rooted in **optimal decision-making**. Dynamic Programming (DP) shares fundamental concepts with **Reinforcement Learning (RL)**.

- **State Representation**: In both DP and RL, we must design an appropriate **state space**. In RL, the state represents the agent's current situation in the environment, while in DP, the state represents partial solutions to the problem.
- **Optimal Substructure**: Both DP and RL problems exhibit **optimal substructure**, meaning that the optimal solution to a problem can be constructed by combining optimal solutions to subproblems.
- **Model-based RL learnt by DP**: DP techniques, especially those involving **Bellman equations** (like Value Iteration and Policy Iteration), form the **core** of many **model-based RL algorithms**. The principles of **state value optimization** and **recurrence relations** in DP provide the mathematical foundation for designing RL algorithms.
- **Key Difference**: DP assumes a known model (transitions and rewards), while RL often works without one.

(c) Monte Carlo Methods

(d) Temporal Difference (TD) Learning

2.2.3 Basics of Reinforcement Learning (Q-Learning and DQN)

2.2.4 Problem Definition and Goal of the RL Agent for Flappy Bird

When applying **Reinforcement Learning** to **Flappy Bird**, the key is to define the problem in terms of states, actions, rewards, and the environment in a way that allows the agent to learn a policy for playing the game effectively. This section will focus on how to define the **RL problem** for Flappy Bird, including the **state space**, **action space**, **reward structure**, and the **goal of the RL agent**.

(1) State Space Definition

The **state space** is the set of all possible states the agent can be in at any given time during the game. In the case of Flappy Bird, the state of the agent (the bird) is typically a combination of variables that describe the bird’s current condition and its environment.

- **Bird's Vertical Position (Y-coordinate/h_bird)**: The vertical position of the bird is important because it determines how close the bird is to the ground or how much room it has to fly upwards.
- **Bird's Vertical Velocity(v_bird)**: The vertical velocity (speed of ascent or descent) tells the agent whether the bird is currently going up or down and how fast.
- **Distance to Next Pipe (X-coordinate/d_pipe)**: The horizontal distance between the bird and the next pipe is crucial because it gives the agent an idea of how much time it has before it needs to make a decision (e.g., flap or wait).
- **Height of the Next Pipe(h_pipe)**: The height of the next pipe relative to the bird's current position tells the agent if it will hit the pipe if it doesn't flap, and it helps decide when to flap.

State Space:

$$State = (h_{bird}, v_{bird}, d_{pipe}, h_{pipe})$$

The **state space** will consist of all possible combinations of these variables, which could be **discretized**.

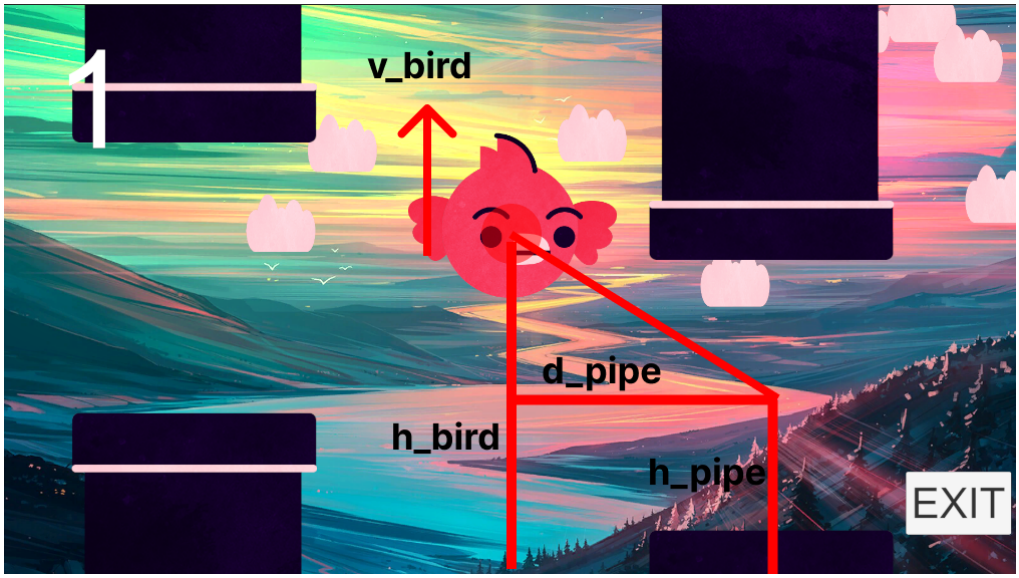


Fig 16: State Space in game

(2) Action Space Definition

The **action space** consists of the set of all actions that the agent can take at any given time. In Flappy Bird, the action space is typically quite simple, consisting of two discrete actions:

- **Flap**: The agent makes the bird flap its wings, which causes the bird to ascend (again altitude).
- **Do Nothing**: The agent does nothing, allowing gravity to pull the bird down.

Action Space:

$$\mathrm{a} \in \mathrm{A}=\{\mathrm{Flap}, \mathrm{Do\ Nothing}\}$$

This is a **discrete action space** because there are only two possible actions at each time step.

(3) Reward Function Define

The **reward function** defines how the agent is rewarded or penalized for each action taken in a given state. The goal of the agent is to **maximize the total reward** over the course of the game.

- **Survival Reward**: The agent should be rewarded for surviving longer for successfully avoiding pipes and not falling to the ceiling and ground. This can be a **small positive**

reward for each frame survived or for passing a pipe.

- **Frame Survive**: The agent should also consider adding **progressive rewards** for surviving longer or getting more creative with penalty functions to avoid the agent from just passing pipes quickly without learning good behavior.
- **Collision Penalty**: The agent should be penalized for colliding with a pipe or the ground. This penalty can be **negative**, with a large negative reward for a collision.
- **Flap Penalty**: A **small negative reward** for flapping to encourage the agent to use flapping wisely, only when it's necessary. This can help prevent the agent from flapping too frequently without good reason.

Reward Function:

$$\text{Reward} = \begin{cases} +10 & \text{if the bird successfully passes a pipe} \\ +0.1 & \text{for each frame survived} \\ -100 & \text{if the bird collides with a pipe or the ground/ceiling} \\ -0.1 & \text{if the bird flaps (optional, to encourage sparing use of flap)} \\ 0 & \text{otherwise} \end{cases}$$

The reward function could be adjusted depending on how you want the agent to behave. For example, a higher reward could be given for passing pipes or a larger penalty for collisions to make the agent more cautious.

(4) Goal of the RL Agent

The **goal of the RL agent** is to learn a policy that maximizes its total expected reward over time, which in the case of Flappy Bird means learning to **keep the bird alive** and **pass as many pipes as possible**. The agent must learn:

- **When to flap**: The agent needs to learn when it's best to flap (i.e., when to gain altitude) and when to let the bird fall due to gravity.
- **Avoiding collisions**: The agent should learn to navigate the space between pipes without colliding with them or hitting the ground.
- **Efficient action selection**: The agent needs to understand the best time to flap based on the bird's position and velocity relative to the next pipe.

(5) State Transition

The **state transition** defines how the environment responds to the agent's actions. In the case of Flappy Bird, the state transitions are determined by the dynamics of the bird's motion, gravity, and the position of the pipes. The state will evolve based on the agent's action, and the environment will compute the new state and the associated reward. State Transition is:

1. If the agent takes the **"Flap"** action, the bird's vertical velocity will change, and the bird's position will be updated accordingly. The reward will depend on whether the bird survives or collides with an obstacle.
2. If the agent takes the **"Do Nothing"** action, the bird will continue falling due to gravity, and the game state will evolve based on the bird's new position and velocity.

These transitions follow the basic **physics of the game** and are deterministic, meaning the same action in the same state will always lead to the same result.

2.2.5 Algorithm Implementation and Code

Based on the problem definition and goal of the RL agent for **Flappy Bird** outlined in section 2.2.4, the next step is to implement the **Reinforcement Learning algorithm** that allows the agent to learn how to play the game effectively. In this case, we will use a **(DQN)**.

- **1.Initialization**

- **Environment Setup:**
  - Initialize the **Flappy Bird** environment E.
  - Define the **state space** SSS and the **action space** A.
- **Q-Network Setup:**
  - Initialize the Q-Network Q(s,a;θ) with random weights θ.
  - Initialize the **target network** Q(s,a;θ-) with the same weights θ- =θ.
- **Replay Memory Setup:**
  - Initialize an empty **replay buffer** D to store experiences.
  - Set the **maximum buffer size** |D|=N.
- **Hyperparameters:**
  - Set the **discount factor** γ.
  - Set the **exploration rate** ϵ, **decay rate** for ϵ, and the **minimum epsilon** ϵ_min.
  - Set **learning rate** α, **batch size** B, and **target network update frequency** T.

- **2.Algorithm Execution**
  - **For each episode e∈{1,2,...,M}:**
    - Initialize the initial state ***s0*** by resetting the environment: ***s0=env.reset()***
    - Set ***done = False and total_reward = 0***.
    - For each time step ***t*** within the episode:
      - 1.**Action Selection**: Select an action a_t using **epsilon-greedy strategy**

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon, \\ \arg\max_{a'} Q(s_t, a'; \theta) & \text{with probability } 1-\epsilon. \end{cases}$$

      - 2.**Take Action and Observe Reward**:
        - Take action ***a_t***, observe the reward ***r_t*** and next state ***s_{t+1}***, and whether the episode has ended ***done***.
        - Store the experience ***(s_t, a_t, r_t, s_{t+1}, done)*** in the **replay buffer** D.
        - Update ***total_reward*** by adding ***r_t***.
      - 3.**Experience Replay and Training**:
        - If |D|≥ B (sufficient experiences in the buffer):
          - Sample a random minibatch of size B from D.
          - For each sample ***(s_t, a_t, r_t, s_{t+1}, done)***:
            - Calculate the target:

$$y_t = \begin{cases} r_t & \text{if done} = \text{True} \\ r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) & \text{if done} = \text{False} \end{cases}$$

            - Calculate the loss function:

$$L(\theta) = \frac{1}{B} \sum_{i=1}^{B} \left[Q(s_t^{(i)}, a_t^{(i)}; \theta) - y_t^{(i)} \right]^2$$

            - Perform gradient descent on the loss to update the Q-Network:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

      - 4.**Target Network Update**:
        - Periodically (every T episodes), update the target network:

$$\hat{Q}(s, a; \theta^-) \leftarrow Q(s, a; \theta)$$

      - 5.**Decay Exploration Rate**:

$$\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}})$$

        - End of episode: Print out the ***total_reward***.

- 3.**Repeat until convergence or max episodes**:
    - Continue this process for all M episodes or until the agent learns to play optimally.

Key Concepts

- **Exploration vs. Exploitation**: The algorithm uses an epsilon-greedy strategy where the agent explores randomly with probability ϵ, and exploits the learned policy with probability 1−ϵ.
- **Replay Memory**: The experience replay buffer helps break correlations between consecutive experiences, allowing for more stable training by sampling random batches.
- **Target Network**: The target network helps to stabilize learning by providing a fixed target for the Q-value updates, reducing oscillations and divergence.
- **Q-Network Update**: The agent updates its policy by approximating the Q-values through the loss function, using the Bellman equation to update the Q-values.
- **Training Loop**: The agent trains over multiple episodes, gradually improving its policy by balancing exploration and exploitation while using the replay buffer and target network to stabilize learning.

Code

```
import random
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque
import gym

# Define the Q-Network (Neural Network)
class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, output_dim)

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        q_values = self.fc3(x)
        return q_values

# Hyperparameters
gamma = 0.9  # Discount factor
epsilon = 1.0  # Exploration rate
epsilon_min = 0.01  # Minimum epsilon value
epsilon_decay = 0.995  # Decay rate for epsilon
learning_rate = 0.001  # Learning rate
batch_size = 32  # Batch size for experience replay
buffer_size = 10000  # Maximum size of the replay buffer
update_frequency = 4  # How often to update the target network
target_update_freq = 1000  # How often to update the target network
max_episodes = 1000  # Max number of episodes
```

```
# Set up the environment (Assume Flappy Bird environment is available in OpenAI Gym)
env = gym.make('FlappyBird-v0')  # This assumes you have a custom Flappy Bird Gym environment

# Initialize the Q-network and target network
state_dim = env.observation_space.shape[0]  # The dimensionality of the state space
action_dim = env.action_space.n  # The number of possible actions
q_network = QNetwork(state_dim, action_dim)
target_network = QNetwork(state_dim, action_dim)
target_network.load_state_dict(q_network.state_dict())  # Initialize target network with same weights

# Initialize optimizer
optimizer = optim.Adam(q_network.parameters(), lr=learning_rate)

# Replay buffer to store experiences
replay_buffer = deque(maxlen=buffer_size)

# Function to select an action
def select_action(state, epsilon):
    if random.random() < epsilon:
        return env.action_space.sample()  # Random action (exploration)
    else:
        state = torch.tensor(state, dtype=torch.float32).unsqueeze(0)  # Add batch dimension
        q_values = q_network(state)
        return torch.argmax(q_values, dim=1).item()  # Select action with max Q-value

# Function to update the Q-network
def update_q_network(batch):
    states, actions, rewards, next_states, dones = batch

    # Convert everything to tensors
    states = torch.tensor(states, dtype=torch.float32)
    next_states = torch.tensor(next_states, dtype=torch.float32)
    actions = torch.tensor(actions, dtype=torch.long)
    rewards = torch.tensor(rewards, dtype=torch.float32)
    dones = torch.tensor(dones, dtype=torch.bool)

    # Get Q-values for current states
    q_values = q_network(states)
    q_values_next = target_network(next_states)

    # Calculate the target Q-values (using Bellman equation)
    max_next_q_values = q_values_next.max(dim=1)[0]
    targets = rewards + gamma * max_next_q_values * ~dones  # Use ~dones to mask out final step (no next state)

    # Get Q-values for the actions taken
    q_values_for_actions = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)

    # Compute loss (Mean Squared Error)
    loss = nn.MSELoss()(q_values_for_actions, targets)
```

```
    # Backpropagate and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Function to train the agent
def train():
    global epsilon
    for episode in range(max_episodes):
        state = env.reset()  # Reset the environment
        done = False
        total_reward = 0

        while not done:
            action = select_action(state, epsilon)  # Select action
            next_state, reward, done, _ = env.step(action)  # Take the action
and observe the result

            # Store the experience in the replay buffer
            replay_buffer.append((state, action, reward, next_state, done))

            state = next_state
            total_reward += reward

            # If enough experiences are in the buffer, sample a batch and train
the Q-network
            if len(replay_buffer) >= batch_size:
                batch = random.sample(replay_buffer, batch_size)
                update_q_network(batch)

        # Decay epsilon (exploration vs. exploitation)
        if epsilon > epsilon_min:
            epsilon *= epsilon_decay

        print(f"Episode {episode+1}, Total Reward: {total_reward}")

        # Update target network every few episodes
        if episode % target_update_freq == 0:
            target_network.load_state_dict(q_network.state_dict())

    print("Training complete!")

# Run the training
train()
```

2.3 Python-Unity Integration

Objective: Establish communication between the Unity game and Python for training the RL agent.

1. Configuring a virtual environment
2. Install the Unity ML-Agents Toolkit
3.

2.4 Training the Agent

Objective: Train the RL agent to play the game using the selected reinforcement learning algorithm.

1.

2.5 Testing and Optimization

Objective: Test and optimize the performance of the trained agent in the game environment.

3 Results Analysis

Objective: Analyze the training results and draw insights from the agent's performance.

4 Discussion

5 Conclusion

6 Reference

1. Dynamic Programming (DP) Introduction - GeeksforGeeks
2. 题库 - 力扣 (LeetCode) 全球极客挚爱的技术成长平台

3. Joshi, A.V. (2023). Dynamic Programming and Reinforcement Learning. In: Machine Learning and Artificial Intelligence. Springer, Cham. https://doi.org/10.1007/978-3-031-12282-8_10
4. What is a Greedy Algorithm? Examples of Greedy Algorithms (freecodecamp.org)
5. Greedy Algorithms - GeeksforGeeks
6. Epsilon-Greedy Q-learning | Baeldung on Computer Science
7. 强化学习（一）- 强化学习介绍、Markov决策过程和贝尔曼期望方程_强化学习gamma的意思-CSDN博客
8. https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-markov-decision-process/
9. https://en.wikipedia.org/wiki/Markov_decision_process
10. 简述马尔可夫链【通俗易懂】 - 知乎 (zhihu.com)
11. Z. He, "Analysis on Deep Reinforcement Learning with Flappy Brid Gameplay," *2022 5th International Conference on Information and Computer Technologies (ICICT)*, New York, NY, USA, 2022, pp. 95-99, doi: 10.1109/ICICT55905.2022.00025.
12.

7 Project Schedule

Day 1: 26/11/2024 (Tuesday) - Game Environment Setup and Basic Mechanics Design

- **Morning:**
  - Conceptualize the project and design the **framework** for the Flappy Bird game.
  - Define core game mechanics and how the environment should function.
- **Afternoon:**
  - Write the **Introduction** section of the report.
- **Evening:**
  - Finalize the **Unity environment setup** for the Flappy Bird game, implementing basic mechanics like physics, gravity, and jump force to establish the foundational game dynamics.

Day 2: 27/11/2024 (Wednesday) - Report Writing and RL Theory Exploration

- **Morning:**
  - Write the **Game Development in Unity** section of the report.
- **Afternoon:**
  - Study the fundamentals of **Deep Q-Networks (DQN)** through educational resources on Bilibili.
  - Explore how **reinforcement learning algorithms (such as DQN)** can be effectively applied to games like Flappy Bird to enhance gameplay through agent learning and decision-making.
- **Evening:**
  - Write the **Reinforcement Learning Setup** sections of the report.

Day 3: 28/11/2024 (Thursday) - Python-Unity Integration

- Morning:
    - Implement Python-Unity communication using **Unity ML-Agents** or custom API calls.
    - Set up the agent's learning environment in Unity (using agent sensors for observation and actions).
- Afternoon:
    - Start integrating the Q-learning/DQN algorithm with the game.
    - Define states (bird's position, velocity, distance to pipes, etc.), actions (flap or do nothing), and rewards (score increment or game over).

Day 4: 29/11/2024 (Friday) - Testing, Optimization, and Report Writing

- Morning:
    - Test the training process and evaluate the agent's performance in Unity.
    - Debug and optimize the Python-Unity integration.
- Afternoon:
    - Begin analyzing the results (reward curves, agent's learning curve, performance over time).
    - Write the **Results** section of the report, focusing on agent performance.

Day 5: 30/11/2024 (Saturday) - Final Evaluation and Report Completion

- **Morning:**
    - Conduct final testing and assess the performance of the agent in the game environment.
    - Explore additional improvements to the agent's behavior (e.g., more complex exploration/exploitation strategies).
- **Afternoon:**
    - Finalize the **Discussion** and **Conclusion** sections of the report.
    - Ensure the project meets all requirements and objectives.
    - Submit the report and release the project.