# PREVENT THIEVERY ARTIFICIAL INTELLIGENCE PROJECT REPORT

MAHMUT ALPEREN ÇAVUŞ 210408044

## Abstract:

This report explores the implementation and functionality of a Q-learning based agent within a grid world environment using the pygame library. The agent's task involves navigating through the grid, collecting a key, unlocking a chest, and retrieving a diamond while evading hazards like spikes, holes, and a chasing guard. Reinforcement learning principles, particularly Q-learning, is used for the agent's decision-making process, allowing it to learn optimal strategies through successive iterations. The report explains the game mechanics, the Q-learning algorithm, the agent's decision-making process, and the overall codebase structure. Moreover, it delves into the different difficulty levels of the game, each presenting progressively challenging environments for the agent to navigate.

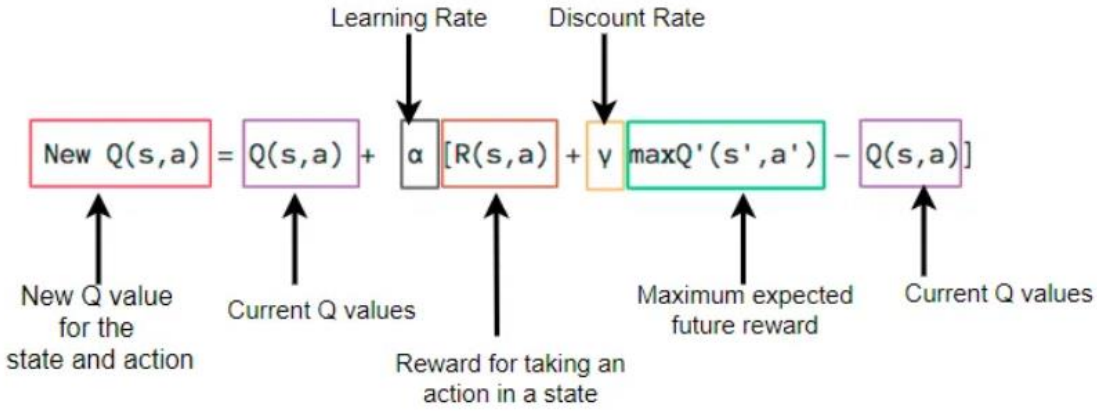
## Introduction:

Reinforcement learning encompasses algorithms and techniques enabling agents to learn optimal decision-making strategies by interacting with their environment. A prominent algorithm in this domain is Q-learning, which involves learning through trial and error to determine the best action in a given state. This report examines Q-learning's application in a grid world environment, where the agent must navigate obstacles, collect rewards, and avoid penalties to reach a specific goal.

## Q-Learning:

Q-learning is a model-free reinforcement learning algorithm enabling an agent to learn an optimal policy for decision-making in a Markov decision process (MDP) environment. It involves learning Q-values for state-action pairs, representing the expected cumulative reward of taking an action in a state and following the optimal policy

thereafter. Through iterative updates using the Bellman equation, the agent refines its Q-values to converge towards an optimal policy.




Bellman Equation Explanation for the episodes

## Code Functionality:

The code implements a grid world environment where an agent operates within a 10x10 grid. It encounters various elements like chests, keys, spikes, holes, and a chasing guard. The agent's objective is to collect the key, unlock the chest, and retrieve the diamond while avoiding hazards and the guard. The code includes Q-learning algorithm, allowing the agent to learn from experience and converge towards an optimal policy.

## Detailed Logic of the Project:

The game unfolds in a grid world where the agent navigates obstacles to achieve its goal. Through exploration, the agent gathers experience and updates its Q-values iteratively. The decision-making process balances exploration and exploitation using an epsilon-greedy strategy. Various elements like hazards and a patrolling guard pose challenges, shaping the agent's learning process.

## Agent Class

### __init__(self, position=(0, 0))

Initializes the agent at a given position, defaulting to the top-left corner of the grid. The agent starts without a key.

### choose_action(self, state, epsilon)

Chooses an action based on the ε-greedy strategy. The agent either explores by selecting a random action or exploits by choosing the action with the highest Q-value for the current state.

### update_position(self, action)

Updates the agent's position based on the chosen action. The agent can move up, down, left, or right, constrained by the grid boundaries.

## Guard Class:

The Guard class represents a mobile entity that patrols the grid world, pursuing the agent using an A* algorithm. It also, like the agent, avoids hazards and dynamically calculates paths to the agent, enhancing the environment's challenge and realism. The guard has a 30% of possibility to move randomly.

### __init__(self, position=(GRID_SIZE - 1, GRID_SIZE - 1), hole_positions=None, spike_positions=None):

Initializes the guard at a given position, defaulting to the bottom-right corner of the grid. The guard is aware of the positions of holes and spikes to avoid stepping on them.

### move(self, agent_position)

Moves the guard towards the agent using the A* pathfinding algorithm, ensuring the guard does not step on holes or spikes.

### astar_move(self, agent_position)

Implements the A* algorithm to find the shortest path to the agent's position, avoiding hazards. If the guard reaches a hole or spike, it will retry until a valid path is found.

### heuristic(self, a, b)

Calculates the Euclidean distance between two points, used as the heuristic in the A* algorithm.

### is_valid_position(self, position)

Checks if a given position is within the grid boundaries and not occupied by hazards.

### random_move(self)

Makes a random valid move, used as a fallback when the A* algorithm suggests an invalid move.

## GridWorld Class:

The GridWorld class encapsulates the grid layout and game elements, facilitating interactions between them. It manages game state, agent movement, guard pursuit, and rendering. The GridWorld class' functions are based on the OpenAI's Gymnasium framework. Gym also has functions like step, render, reset functions.

## step():

The step function decides what happens when the agent takes a step, what reward will be given to it. It includes various cases in it. The cases are: If the agent takes an empty step 'empty' reward is given. If the agent takes a step that converges itself to the chest it gets +100 reward for positive feedback. If the agent gets to the key position, it is rewarded with 'key' reward. Agent can't collect 'key' reward if it got the 'key' reward once in the same episode. If the agent gets to the chest without having the key, it is counted as an empty step. If the agent converges to the chest while having the key, agent is awarded with +200 points for positive feed-back. If the agent collects the chest while having the key, agent is awarded with 'chest' and 'finish' rewards. If the agent steps on a spike it is awarded with 'spike' reward. If the agent falls into a hole, it is awarded with 'hole' reward. If the agent gets caught by the guard it is awarded with 'guard' reward.Guard's movement is also included in this function.

## render():

The render function renders everything. Agent and guard's positions is rendered and updated according to their moves. The traps are rendered according to the map generation functions.

## reset():

The reset function resets the environment when the episode ends. It puts agent and the guard to their initial positions.

### is_valid_position(self, position)

Checks if a given position is within the grid boundaries and not occupied by hazards.

### get_state(self)

Returns the agent's current state as a flattened array of a 5x5 observation area centered on the agent, encoding the positions of various elements.

### check_game_over(self)

Checks if the game is over based on the agent's position. Returns a boolean indicating game over and the associated reward.

## Utility Functions

### load_experience(filename)

Loads the Q-table from a JSON file.

### save_experience(filename, experience)

Saves the Q-table to a JSON file.

### generate_easy_map()

Generates an easy map configuration with predefined positions for the chest, key, spikes, and holes.

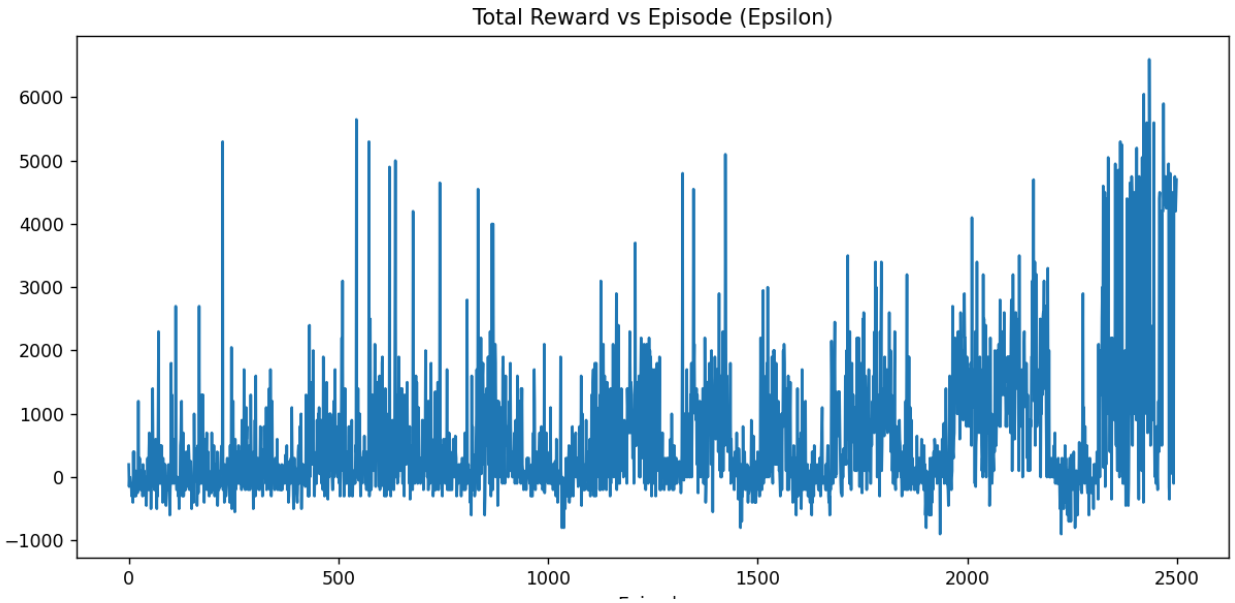
### generate_moderate_map()

Generates a moderate map configuration with more obstacles compared to the easy map.

### generate_hard_map()

Generates a hard map configuration with the most obstacles and complex layout.

## CHANGE OF THE REWARD VALUE BASED ON EPSILON VALUE




In the beginning epsilon is 1.0, with each episode epsilon is decaying (decreasing) by 0.001. There are 2500 episodes. The epsilon is used for epsilon greedy approach. The idea is to choose a random action with probability $\epsilon$ (exploration) and the action with the highest estimated reward with probability 1−ϵ (exploitation). This helps the agent to explore new actions and states while also exploiting the knowledge it has gained so far to maximize the reward. In this plot, the learning rate is 0.1. This is the best value for the maximizing the reward according to my experiments. The plot is wavy because there are too many factors for maximizing the reward. The inconsistency occurs because the guard moves randomly with a 30% probability. However, we can see that near to 2500 episodes the high reward rate is increased. Because the agent exploits.

## main(difficulty_level)

- Initializes the agent and grid world based on the specified difficulty level.
- Sets up Q-learning parameters and loads any existing experience.
- Runs the training loop for a specified number of episodes, where the agent interacts with the environment, updates the Q-table, and renders the grid world using Pygame.
- Decreases the exploration rate (epsilon) over time to balance exploration and exploitation.
- Saves the Q-table after each episode and plots the total rewards over episodes.

# CONCLUSION

In this report, we explored the implementation and effectiveness of the epsilon-greedy approach within the context of Q-learning, a fundamental reinforcement learning algorithm. By balancing exploration and exploitation, the epsilon-greedy strategy enables the agent to discover optimal actions in uncertain environments while gradually refining its policy based on accumulated experience.

Our implementation commenced with a high probability of exploration ($\epsilon$=1.0) allowing the agent to gather diverse experiences across the state-action space. As training progressed, the decay mechanism systematically reduced ϵ, encouraging the agent to increasingly exploit its knowledge by selecting actions with the highest estimated rewards. This transition from exploration to exploitation is pivotal for achieving robust and efficient learning, as it ensures comprehensive state-action space coverage and fine-tuning of the Q-values.

The results demonstrated that the agent relatively successfully learned to navigate and optimize its actions, achieving progressively higher rewards over time.