Capstone #3: Unleashing Strategy in a Puzzle Game with Reinforcement Learning

Zeliha Ural Merpez – M2M Tech

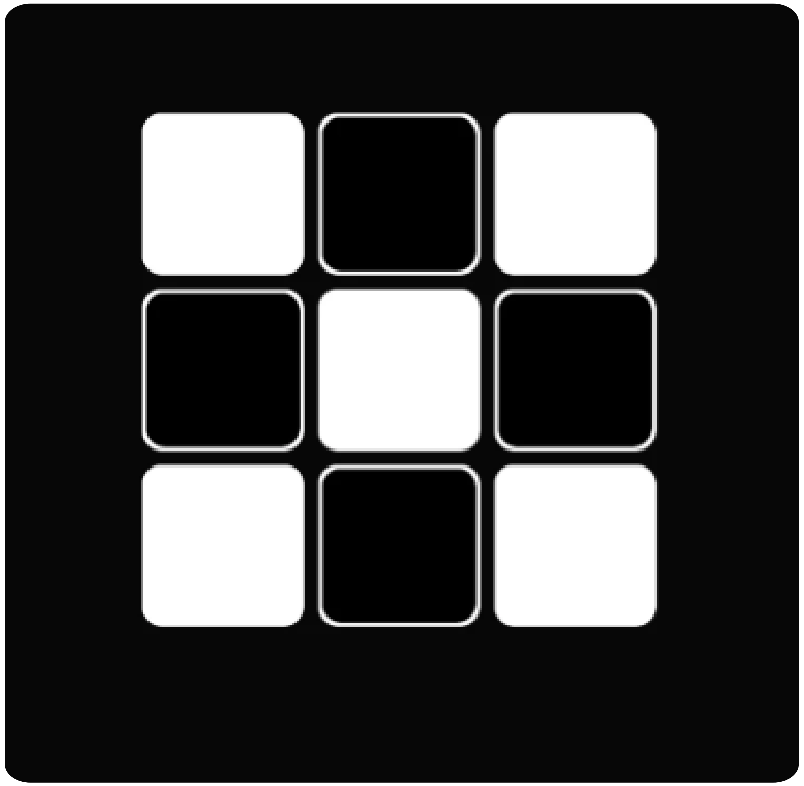
- Imagine a playing field filled with interconnected squares, each initially set to either white or black. The goal is simple: turn every square black.

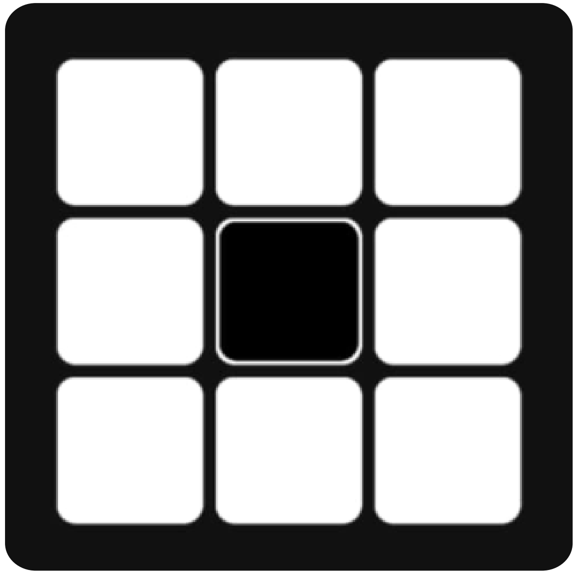
- However, there's a twist. When you press a square, not only does it change color, but all of its neighboring squares also change their color.

- But what if you could leave the puzzle-solving to an intelligent agent capable of learning the best strategies? That's where **reinforcement learning (RL)** comes in.

- In this capstone project, we will explore how RL, specifically Q-learning, can be applied to solve this intriguing puzzle game.

Dark – Zeliha Ural Merpez





The Math Behind the Puzzle: Understanding the Problem Structure

- **Inversion of Actions**: Each action is its own inverse. We can treat the color of each square as a binary value (0 for white, 1 for black).
- **Commutativity of Actions**: The order of actions doesn’t affect the resulting configuration which significantly reduces the complexity.
- **Optimization of Actions**: The minimum number of actions to solve the puzzle, initially found through brute force, can now be efficiently solved by reinforcement learning.
- **Solvability**: While not every state is solvable, all solvable states can be transformed into one another.

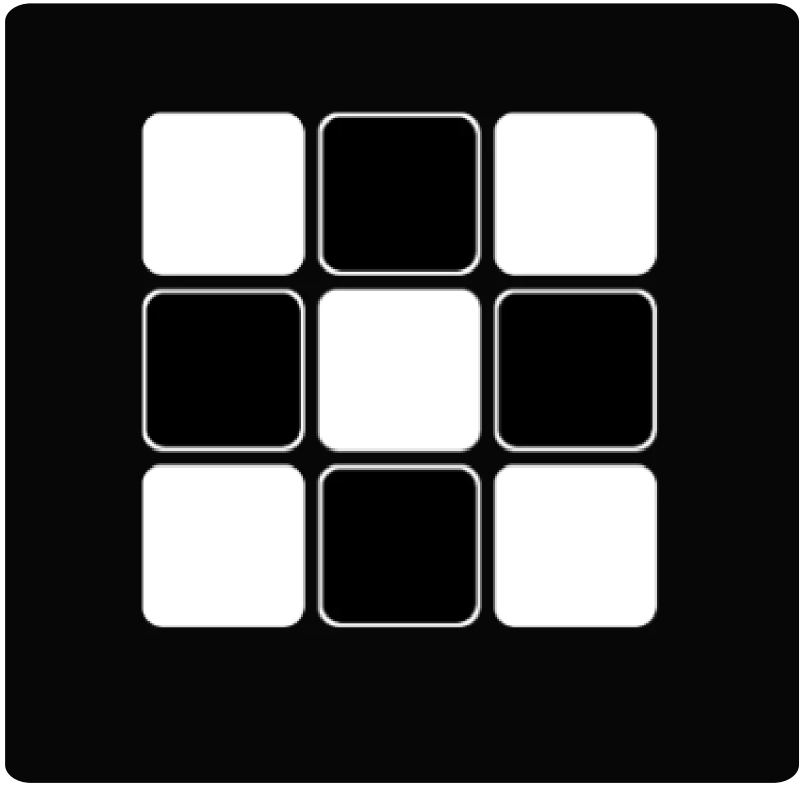
Applying Reinforcement Learning to the Puzzle

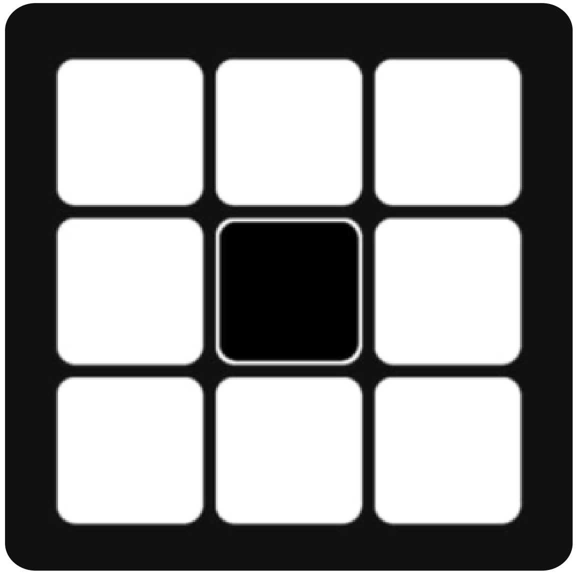
- Reinforcement learning is a branch of machine learning where an agent learns how to achieve a goal by interacting with an environment.

- The agent receives feedback in the form of rewards based on its actions, which helps it learn the most effective strategies over time.

- In our case, the environment consists of the grid of squares, and the goal is to toggle them all to black.

- The agent learns through trial and error, selecting actions (pressing squares) and receiving rewards or penalties based on the state of the grid after each action.





The Graph Environment

- Each square is represented as a node in a graph, and the edges between them represent their adjacency.

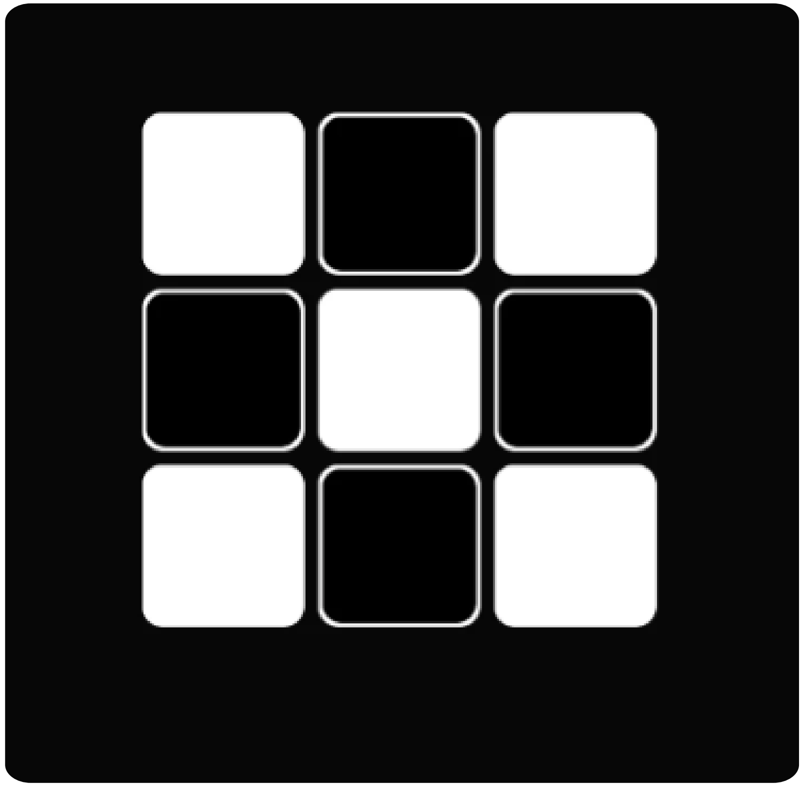
- The environment tracks the state of each square (whether it is black or white) and updates the state whenever a square is pressed.

```
class GraphEnvironment:
    def __init__(self, graph, goal_product=1):
        self.graph = graph
        self.goal_product = goal_product
        self.state = self.get_state()
        self.initial_state = self.get_state()
```

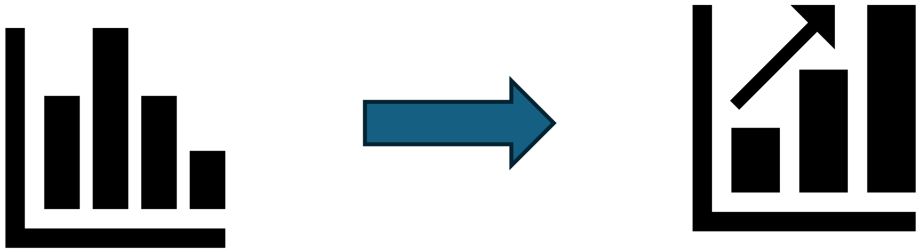
```
def get_state(self):
    return {str(node):  self.graph.nodes[node]['color'] for node in self.graph.nodes}
```

Cont. The Graph Environment

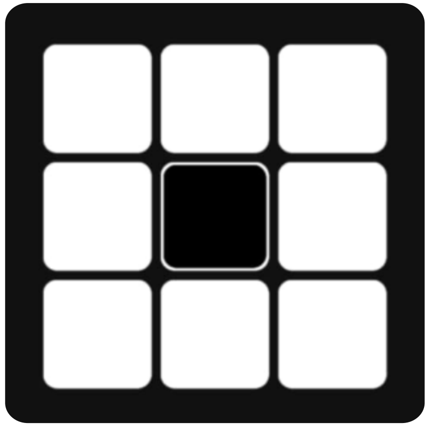
```python
def set_state(self, target_state):
    self.state = target_state
    for node in self.graph.nodes:
        self.graph.nodes[node]['color'] = target_state[str(node)]

def change_color(self, node):
    self.graph.nodes[node]['color'] = 1 - self.graph.nodes[node]['color']

def apply_action(self, action):
    """ Apply an action (modify node weights) and return the new state. """
    self.change_color(action)
    for node in self.graph.neighbors(action):
        self.change_color(node)
    self.state = self.get_state()
    return self.state
```



Cont. The Graph Environment



```python
def get_reward(self, state):
    """ Reward is based on how close the product of node weights is to the goal. """
    product = 1
    for color in state.values():
        product *= color
    if product == self.goal_product:
        return 100
    elif product == 0:
        return -100
    else:
        return -10
```



The Q-Learning Agent

Q-learning is an off-policy RL algorithm that helps an agent learn the best actions by estimating the long-term value (reward) of each action in each state.

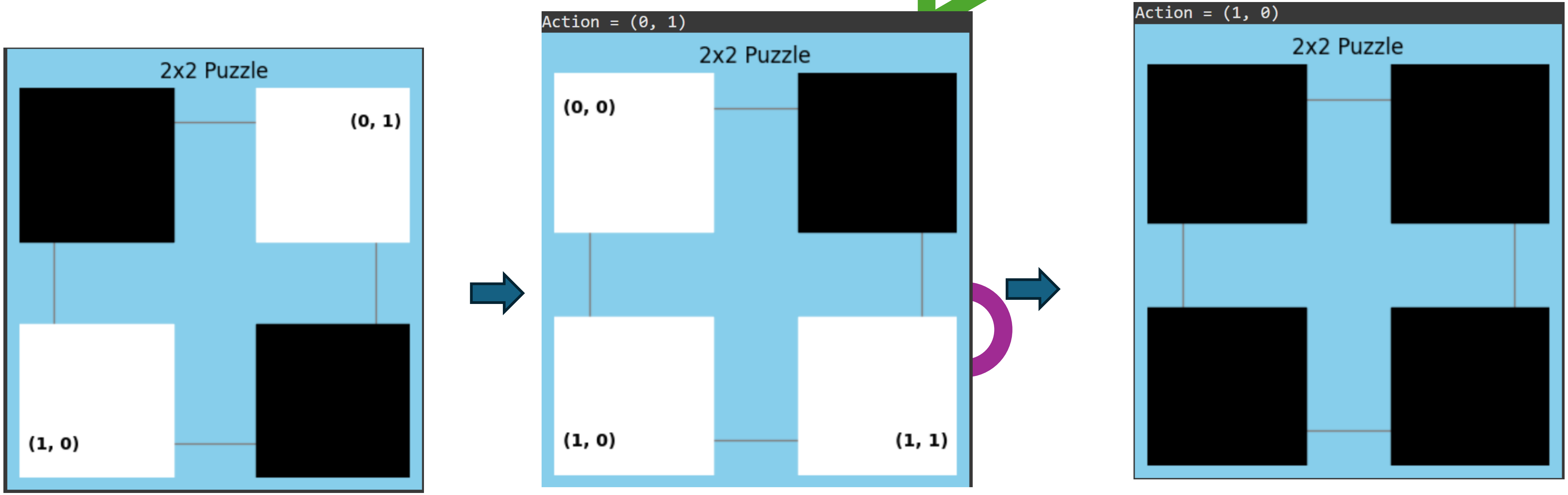
The Q-learning algorithm maintains a **Q-table** that stores the expected future rewards for each state-action pair. During training, the agent explores different actions and updates the Q-table based on the rewards it receives.

```python
def choose_action(self):
    """ Choose an action using epsilon-greedy approach. """
    state = self.env.get_state()
    state_tuple = tuple(sorted(state.items()))
    if random.random() < self.epsilon:
        return random.choice(self.env.get_possible_actions())
    else:
        if state_tuple not in self.q_table:
            self.q_table[state_tuple] = {action: 0 for action in self.env.get_possible_actions()}
        return max(self.q_table[state_tuple], key=self.q_table[state_tuple].get)
```

Cont. The Q-Learning Agent

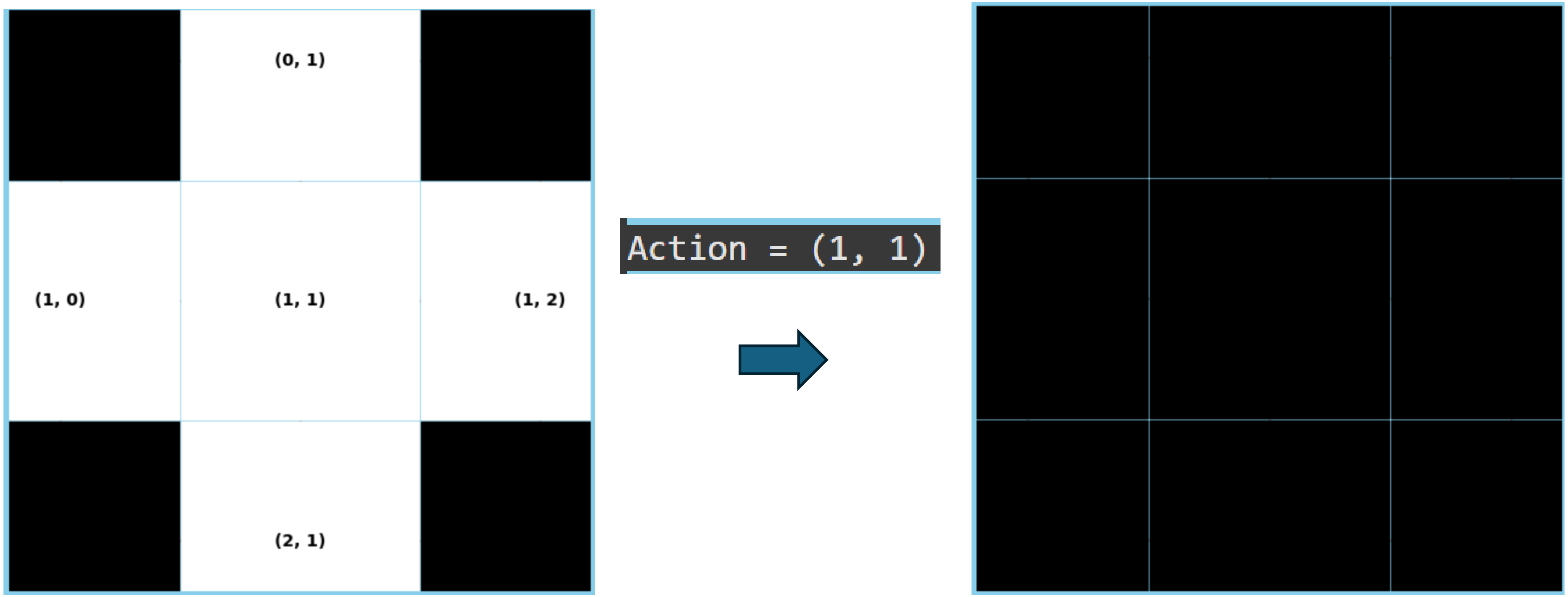
```
def learn(self, action, reward, next_state):
    """ Update the Q-table based on the action taken and the received reward. """
    state = self.env.get_state()
    state_tuple = tuple(sorted(state.items()))
    next_state_tuple = tuple(sorted(next_state.items()))
    if state_tuple not in self.q_table:
        self.q_table[state_tuple] = {action: 0 for action in self.env.get_possible_actions()}
    if next_state_tuple not in self.q_table:
        self.q_table[next_state_tuple] = {action: 0 for action in self.env.get_possible_actions()}

    max_next_q_value = max(self.q_table[next_state_tuple].values())  # Maximum Q-value for next sta

    # Update Q-value using the Q-learning formula
    self.q_table[state_tuple][action] = self.q_table[state_tuple][action] + self.alpha * (
        reward + self.gamma * max_next_q_value - self.q_table[state_tuple][action])
```

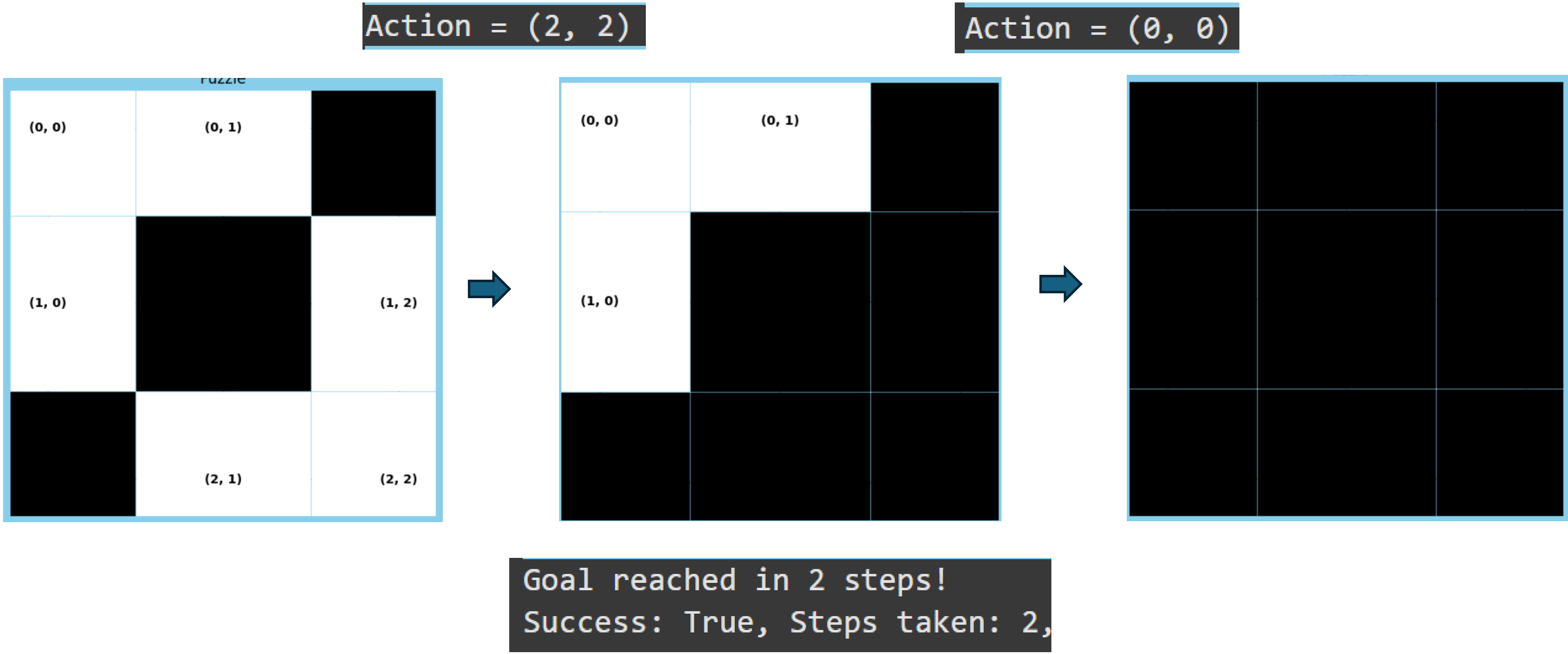
Testing 2x2 Puzzle




```
Goal reached in 2 steps!
Success: True, Steps taken: 2
```

Testing 3x3 Puzzle





Action = (2, 2)
Action = (0, 0)
(0, 0)
(0, 1)
(1, 0)
(1, 2)
(2, 1)
(2, 2)
(0, 0)
(0, 1)
(1, 0)
Goal reached in 2 steps!
Success: True, Steps taken: 2,

The Power of Reinforcement Learning in Puzzle Games

The results of the experiment show that the agent was able to effectively solve simple tasks requiring 1-2 steps.

However, as the complexity of the tasks increased, particularly in cases requiring more steps, the agent struggled to generate the desired solutions.

Potential Improvements:

•**Extended Training**: Training the agent for a longer period could help it better explore and adapt to complex tasks.

•**Adjusting the Reward Mechanism**: Tweaking the reward structure could better guide the agent's decision-making process.

Thanks!

Link for game:

https://play.google.com/store/apps/details?id=com.appercute.dark&pcampaignid=web_share

Link for code:

https://github.com/zmerpez/dark-RL

Zeliha Ural Merpez