Nonogram Solver

Setup and Imports

In [2]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
from tqdm.notebook import tqdm
from torch.utils.tensorboard import SummaryWriter
import os
import glob

# Configure device to use GPU if available, otherwise use CPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

Helper Functions

Nonogram Generation

In [3]:
```python
def generate_unique_nonogram(grid_size, batch_size, existing_solutions=set()):
    """
    Generate unique Nonogram puzzles with corresponding clues.

    Parameters:
    - grid_size (int): Size of the Nonogram grid.
    - batch_size (int): Number of Nonogram puzzles to generate.
    - existing_solutions (set): Set of existing solutions to avoid duplicates.

    Returns:
    - solutions (ndarray): Generated Nonogram solutions.
    - row_clues (list): Row clues for the Nonogram puzzles.
    - col_clues (list): Column clues for the Nonogram puzzles.
    - existing_solutions (set): Updated set of existing solutions.
    """
    solutions = []
    while len(solutions) < batch_size:
        new_solutions = np.random.randint(2, size=(batch_size, grid_size, grid_size))
        for solution in new_solutions:
            solution_tuple = tuple(map(tuple, solution))
            if solution_tuple not in existing_solutions:
                solutions.append(solution)
                existing_solutions.add(solution_tuple)
            if len(solutions) == batch_size:
                break
    solutions = np.array(solutions)
    row_clues = [[list(map(len, ''.join(map(str, row)).split('0'))) for row in solution] for solu
    col_clues = [[list(map(len, ''.join(map(str, col)).split('0'))) for col in solution.T] for so
    row_clues = [[[clue for clue in clues if clue > 0] or [0] for clues in row] for row in row_c
    col_clues = [[[clue for clue in clues if clue > 0] or [0] for clues in col] for col in col_c

    return solutions, row_clues, col_clues, existing_solutions
```

Clue Padding

In [4]:
```python
def pad_clues(clues, max_len):
    """
    Pad clues to the maximum length.

    Parameters:
    - clues (list): List of clues to pad.
    - max_len (int): Maximum length to pad the clues to.

    Returns:
    - padded_clues (list): Padded clues.
    """
    return [clue + [0] * (max_len - len(clue)) for clue in clues]
```

Correct Guess Calculation

In [5]:
```python
def calculate_correct_guesses(states, solutions):
    """
    Calculate the number of correct guesses.

    Parameters:
    - states (ndarray): Current states of the Nonogram grids.
    - solutions (ndarray): Solution grids of the Nonogram puzzles.

    Returns:
    - correct_guesses (ndarray): Number of correct guesses.
    """
    return np.sum(states == solutions, axis=(1, 2))
```

Checkpoint Saving

In [6]:
```python
def save_checkpoint(agent, optimizer, episode, reward_list, correct_guess_percent_list, clue_max_
    """
    Save the training checkpoint.

    Parameters:
    - agent (NonogramAgent): The agent being trained.
    - optimizer (torch.optim.Optimizer): Optimizer used for training.
    - episode (int): Current episode number.
    - reward_list (list): List of rewards.
    - correct_guess_percent_list (list): List of correct guess percentages.
    - clue_max_len (int): Maximum length of the clues.
    - clue_dim (int): Dimensionality of the clues.
    - directory (str): Directory to save the checkpoint.
    """
    if not os.path.exists(directory):
        os.makedirs(directory)

    checkpoint_path = os.path.join(directory, f'checkpoint_{episode}.pth')

    torch.save({
        'episode': episode,
        'model_state_dict': agent.policy_net.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'reward_list': reward_list,
        'correct_guess_percent_list': correct_guess_percent_list,
        'clue_max_len': clue_max_len,
```

```
        'clue_dim': clue_dim
    }, checkpoint_path)
```

Checkpoint Loading

In [7]:
```
def load_checkpoint(agent, optimizer, directory='models'):
    """
    Load the latest training checkpoint.

    Parameters:
    - agent (NonogramAgent): The agent being trained.
    - optimizer (torch.optim.Optimizer): Optimizer used for training.
    - directory (str): Directory to load the checkpoint from.

    Returns:
    - episode (int): Episode number to resume from.
    - reward_list (list): List of rewards.
    - correct_guess_percent_list (list): List of correct guess percentages.
    - clue_max_len (int): Maximum length of the clues.
    - clue_dim (int): Dimensionality of the clues.
    """
    checkpoints = sorted(glob.glob(os.path.join(directory, 'checkpoint_*.pth')), key=lambda x: in
    if checkpoints:
        checkpoint = torch.load(checkpoints[0])
        agent.policy_net.load_state_dict(checkpoint['model_state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        return checkpoint['episode'], checkpoint['reward_list'], checkpoint['correct_guess_perce

    return 0, [], [], None, None
```

Reward Discounting

In [8]:
```
def discount_rewards(rewards, gamma=0.995):
    """
    Compute discounted rewards.

    Parameters:
    - rewards (list): List of rewards.
    - gamma (float): Discount factor.

    Returns:
    - discounted_rewards (list): List of discounted rewards.
    """
    discounted_rewards = []
    for reward in rewards:
        cumulative_rewards = 0
        discounted = []
        for r in reversed(reward):
            cumulative_rewards = r + gamma * cumulative_rewards
            discounted.insert(0, cumulative_rewards)
        discounted_rewards.append(torch.tensor(discounted, dtype=torch.float32).to(device))
    return discounted_rewards
```

Divide and Round Up

In [9]:
```
def divide_and_round_up(n):
    """
    Divide and round up the number.
```

```
    Parameters:
    - n (int): Number to divide and round up.

    Returns:
    - result (int): Result of the division and rounding up.
    """
    return (n + 1) // 2 if n % 2 != 0 else n // 2
```

Visualize Nonogram

In [10]:

```
def visualize_nonogram(board):
    """
    Visualize the Nonogram board.

    Parameters:
    - board (ndarray): The current state of the Nonogram board.
    """
    grid_size = len(board)
    for row in range(grid_size):
        print(" ".join(str(cell) if cell != -1 else "?" for cell in board[row]))
```

Visualize Clues

In [11]:

```
def visualize_clues(clues):
    """
    Visualize the clues for the Nonogram puzzle.

    Parameters:
    - clues (list): List of clues for the puzzle.
    """
    for clue in clues:
        clue = [c for c in clue if c != 0]  # Remove padding zeros
        if not clue:  # If no clues, add a single zero
            clue = [0]
        print(clue)
```

Update Puzzle State

In [12]:

```
def update_puzzle_state(agent, env, states, row_clues, col_clues, solutions):
    """
    Update the puzzle state based on the agent's actions.

    Parameters:
    - agent (NonogramAgent): The agent solving the puzzle.
    - env (NonogramEnvironment): The environment of the Nonogram puzzle.
    - states (ndarray): The current states of the puzzles.
    - row_clues (list): The row clues for the puzzles.
    - col_clues (list): The column clues for the puzzles.
    - solutions (ndarray): The solutions for the puzzles.
    """
    move_counter = 0
    done = False
    while not done:
        actions, _ = agent.select_actions(states, row_clues, col_clues)
        states, rewards, done = env.step(actions)
        done = done[0]  # Since batch_size is 1
```

```
        move = actions[0]
        row, col, state_value = move
        move_counter += 1
        print(f"\nMove: {move_counter}, Guess: ({row + 1}, {col + 1}), State: {'1' if state_valu

        # Visualize current state
        print("Current Puzzle State:")
        visualize_nonogram(states[0])

        # Comparison with actual solution
        current_state = states[0]
        actual_state = solutions[0]

        # Calculate mismatches excluding cells with -1
        mismatches = np.argwhere((current_state != actual_state) & (current_state != -1) & (actu
        if mismatches.size > 0:
            print("Mismatches:")
            for (i, j) in mismatches:
                print(f"  Mismatch at ({i}, {j}): Algorithm State = {current_state[i][j]}, Actua

    print("Puzzle Solved!")
```

Nonogram Environment

In [13]:

```
class NonogramEnvironment:
    def __init__(self, grid_size, batch_size, streak_cap=5):
        """
        Initialize the Nonogram environment.

        Parameters:
        - grid_size (int): Size of the Nonogram grid.
        - batch_size (int): Number of puzzles in a batch.
        - streak_cap (int): Maximum streak for unique guesses.
        """
        self.grid_size = grid_size
        self.batch_size = batch_size
        self.streak_cap = streak_cap
        self.solution, self.row_clues, self.col_clues = self.generate_initial_nonogram(grid_size
        self.state = np.full((batch_size, grid_size, grid_size), -1, dtype=int)
        self.steps = np.zeros(batch_size, dtype=int)
        self.chosen_cells = [set() for _ in range(batch_size)]
        self.correct_guesses = [set() for _ in range(batch_size)]
        self.unique_guesses_streak = np.zeros(batch_size, dtype=int)

    def generate_initial_nonogram(self, grid_size, batch_size):
        """
        Generate initial Nonogram puzzles.

        Parameters:
        - grid_size (int): Size of the Nonogram grid.
        - batch_size (int): Number of puzzles in a batch.

        Returns:
        - solution, row_clues, col_clues (tuple): Initial puzzles and their clues.
        """
        return generate_unique_nonogram(grid_size, batch_size)[0:3]

    def reset(self):
        """
        Reset the Nonogram environment to its initial state.
```

```python
        Returns:
        - state, row_clues, col_clues (tuple): Reset state and clues.
        """
        self.state = np.full((self.batch_size, self.grid_size, self.grid_size), -1, dtype=int)
        self.steps = np.zeros(self.batch_size, dtype=int)
        self.chosen_cells = [set() for _ in range(self.batch_size)]
        self.correct_guesses = [set() for _ in range(self.batch_size)]
        self.unique_guesses_streak = np.zeros(self.batch_size, dtype=int)
        return self.state, self.row_clues, self.col_clues

    def reset_with_solutions(self, solutions, row_clues, col_clues):
        """
        Reset the environment with specified solutions and clues.

        Parameters:
        - solutions (ndarray): Solution grids of the Nonogram puzzles.
        - row_clues (list): Row clues for the Nonogram puzzles.
        - col_clues (list): Column clues for the Nonogram puzzles.

        Returns:
        - state, row_clues, col_clues (tuple): Reset state and clues.
        """
        self.solution = solutions
        self.row_clues = row_clues
        self.col_clues = col_clues
        return self.reset()

    def step(self, actions):
        """
        Take a step in the Nonogram environment.

        Parameters:
        - actions (list): List of actions to take.

        Returns:
        - state, rewards, done (tuple): Updated state, rewards, and done flags.
        """
        rewards = np.zeros(self.batch_size, dtype=float)
        done = np.zeros(self.batch_size, dtype=bool)

        for i, action in enumerate(actions):
            row, col, value = action
            self.steps[i] += 1

            if (row, col) in self.chosen_cells[i]:
                rewards[i] = -5
                self.unique_guesses_streak[i] = 0
            else:
                self.chosen_cells[i].add((row, col))
                self.unique_guesses_streak[i] += 1
                rewards[i] = min(self.unique_guesses_streak[i], self.streak_cap)

                if self.solution[i, row, col] == value:
                    rewards[i] += 2
                    self.correct_guesses[i].add((row, col))
                else:
                    rewards[i] -= 2

                self.state[i, row, col] = self.solution[i, row, col]

                if all(self.state[i, row, c] != -1 for c in range(self.grid_size)) and \
```

```
                    all(self.state[i, row, c] == self.solution[i, row, c] for c in range(self.gri
                     rewards[i] += 10

                if all(self.state[i, r, col] != -1 for r in range(self.grid_size)) and \
                    all(self.state[i, r, col] == self.solution[i, r, col] for r in range(self.gri
                     rewards[i] += 10

                if all(self.state[i, r, c] != -1 for r in range(self.grid_size) for c in range(se
                    all(self.state[i, r, c] == self.solution[i, r, c] for r in range(self.grid_si
                     rewards[i] += 100

            done[i] = self._check_done(i)
        return self.state, rewards, done

    def _check_done(self, index):
        """
        Check if the puzzle is solved or maximum steps reached.

        Parameters:
        - index (int): Index of the puzzle.

        Returns:
        - done (bool): Whether the puzzle is solved or maximum steps reached.
        """
        return np.array_equal(self.state[index], self.solution[index]) or self.steps[index] >= se
```

Model Definition

Clue Transformer

In [14]:

```
class ClueTransformer(nn.Module):
    def __init__(self, grid_size, clue_max_len, clue_dim, num_heads, num_layers, model_dim):
        """
        Initialize the Clue Transformer.

        Parameters:
        - grid_size (int): Size of the Nonogram grid.
        - clue_max_len (int): Maximum length of the clues.
        - clue_dim (int): Dimensionality of the clues.
        - num_heads (int): Number of attention heads.
        - num_layers (int): Number of transformer layers.
        - model_dim (int): Dimensionality of the model.
        """
        super(ClueTransformer, self).__init__()
        self.grid_size = grid_size
        self.embedding = nn.Embedding(clue_dim + 1, model_dim)
        self.model_dim = model_dim
        self.positional_encoding = nn.Parameter(torch.randn(1, clue_max_len*grid_size, model_dim
        encoder_layer = nn.TransformerEncoderLayer(d_model=model_dim, nhead=num_heads, batch_firs
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)

    def forward(self, clues):
        """
        Forward pass of the Clue Transformer.

        Parameters:
        - clues (Tensor): Clues for the Nonogram puzzles.

        Returns:
```

```
        - transformer_output (Tensor): Output of the transformer.
        """
        batch_size, num_clues, clue_len = clues.size()
        clues = clues.view(batch_size, -1)

        # Ensure clues are within the valid range
        assert torch.max(clues) <= self.embedding.num_embeddings - 1, f"Index {torch.max(clues)}

        embedded_clues = self.embedding(clues)

        max_len = embedded_clues.size(1)
        if self.positional_encoding.size(1) < max_len:
            self.positional_encoding = nn.Parameter(torch.randn(1, max_len, self.model_dim).to(em

        embedded_clues = embedded_clues + self.positional_encoding[:, :embedded_clues.size(1), :
        transformer_output = self.transformer(embedded_clues)
        return transformer_output
```

Policy Network

In [15]:

```
class PolicyNetwork(nn.Module):
    def __init__(self, grid_size, clue_max_len, clue_dim):
        """
        Initialize the Policy Network.

        Parameters:
        - grid_size (int): Size of the Nonogram grid.
        - clue_max_len (int): Maximum length of the clues.
        - clue_dim (int): Dimensionality of the clues.
        """
        super(PolicyNetwork, self).__init__()
        self.grid_size = grid_size
        self.conv1 = nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(4, 8, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(8 * grid_size * grid_size, 16)
        self.row_clue_transformer = ClueTransformer(grid_size, clue_max_len, clue_dim, num_heads=
        self.col_clue_transformer = ClueTransformer(grid_size, clue_max_len, clue_dim, num_heads=
        self.fc2 = nn.Linear(16 * 2 + 16, 32)
        self.fc3 = nn.Linear(32, grid_size * grid_size * 2)

    def forward(self, state, row_clues, col_clues):
        """
        Forward pass of the Policy Network.

        Parameters:
        - state (Tensor): Current state of the Nonogram puzzles.
        - row_clues (Tensor): Row clues for the Nonogram puzzles.
        - col_clues (Tensor): Column clues for the Nonogram puzzles.

        Returns:
        - output (Tensor): Action logits for the Nonogram puzzles.
        """
        state = state.to(device)
        row_clues = row_clues.to(device)
        col_clues = col_clues.to(device)

        x = state.unsqueeze(1).float()
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = x.view(-1, 8 * self.grid_size * self.grid_size)
```

```python
        x = F.relu(self.fc1(x))

        row_clues[row_clues >= clue_dim + 1] = clue_dim
        col_clues[row_clues >= clue_dim + 1] = clue_dim

        row_clues = self.row_clue_transformer(row_clues).mean(dim=1)
        col_clues = self.col_clue_transformer(col_clues).mean(dim=1)

        clues = torch.cat((row_clues, col_clues), dim=1)

        x = torch.cat((x, clues), dim=1)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x.view(-1, self.grid_size * self.grid_size, 2)
```

Nonogram Agent

In [16]:

```python
class NonogramAgent:
    def __init__(self, grid_size, clue_max_len, clue_dim):
        """
        Initialize the Nonogram Agent.

        Parameters:
        - grid_size (int): Size of the Nonogram grid.
        - clue_max_len (int): Maximum length of the clues.
        - clue_dim (int): Dimensionality of the clues.
        """
        self.policy_net = PolicyNetwork(grid_size, clue_max_len, clue_dim).to(device)
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=0.001)
        self.grid_size = grid_size

    def select_actions(self, states, row_clues, col_clues):
        """
        Select actions based on the current state and clues.

        Parameters:
        - states (ndarray): Current states of the Nonogram puzzles.
        - row_clues (ndarray): Row clues for the Nonogram puzzles.
        - col_clues (ndarray): Column clues for the Nonogram puzzles.

        Returns:
        - actions (list): List of selected actions.
        - log_probs (Tensor): Log probabilities of the selected actions.
        """
        states = torch.tensor(states, dtype=torch.float32).to(device)
        row_clues = torch.tensor(row_clues, dtype=torch.long).to(device)
        col_clues = torch.tensor(col_clues, dtype=torch.long).to(device)
        logits = self.policy_net(states, row_clues, col_clues)

        action_probs = torch.softmax(logits.view(states.size(0), -1), dim=-1)

        action_dist = torch.distributions.Categorical(action_probs)
        flat_actions = action_dist.sample()
        log_probs = action_dist.log_prob(flat_actions)

        actions = []
        for flat_action in flat_actions:
            flat_action_idx = flat_action.item()
            position_idx = flat_action_idx // 2
            value = flat_action_idx % 2
```

```
            row = position_idx // self.grid_size
            col = position_idx % self.grid_size
            actions.append((row, col, value))

        return actions, log_probs

    def update_policy(self, log_probs, rewards):
        """
        Update the policy network based on the collected log probabilities and rewards.

        Parameters:
        - log_probs (list): Log probabilities of the selected actions.
        - rewards (list): Rewards obtained from the actions.
        """
        discounted_rewards = discount_rewards(rewards)

        discounted_rewards = torch.cat(discounted_rewards)
        discounted_rewards = (discounted_rewards - discounted_rewards.mean()) / (discounted_rewa

        log_probs = torch.cat([torch.stack(lps) for lps in log_probs])

        loss = -torch.sum(log_probs * discounted_rewards.to(device))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

Training Procedure

In [17]:

```
def train_agent(grid_size, clue_max_len, clue_dim, num_episodes, train_batch_size, val_batch_siz
    """
    Train the Nonogram Agent.

    Parameters:
    - grid_size (int): Size of the Nonogram grid.
    - clue_max_len (int): Maximum length of the clues.
    - clue_dim (int): Dimensionality of the clues.
    - num_episodes (int): Number of training episodes.
    - train_batch_size (int): Batch size for training.
    - val_batch_size (int): Batch size for validation.
    - save_interval (int): Interval for saving checkpoints.
    """
    validation_solutions, validation_row_clues, validation_col_clues, existing_solutions = genera
    validation_row_clues = [pad_clues(rc, clue_max_len) for rc in validation_row_clues]
    validation_col_clues = [pad_clues(cc, clue_max_len) for cc in validation_col_clues]

    env = NonogramEnvironment(grid_size, train_batch_size)
    agent = NonogramAgent(grid_size, clue_max_len, clue_dim)
    optimizer = agent.optimizer

    writer = SummaryWriter('runs/nonogram_experiment')

    total_cells = grid_size * grid_size

    start_episode, reward_list, correct_guess_percent_list, saved_clue_max_len, saved_clue_dim =

    if saved_clue_max_len is not None:
        clue_max_len = saved_clue_max_len
    if saved_clue_dim is not None:
        clue_dim = saved_clue_dim
```

```
    progress_bar = tqdm(range(start_episode, num_episodes), desc="Training")

    for episode in progress_bar:
        train_solutions, train_row_clues, train_col_clues, existing_solutions = generate_unique_
        train_row_clues = [pad_clues(rc, clue_max_len) for rc in train_row_clues]
        train_col_clues = [pad_clues(cc, clue_max_len) for cc in train_col_clues]

        states, row_clues, col_clues = env.reset_with_solutions(train_solutions, train_row_clues

        log_probs = [[] for _ in range(train_batch_size)]
        rewards = [[] for _ in range(train_batch_size)]
        done = np.zeros(train_batch_size, dtype=bool)

        while not np.all(done):
            actions, log_prob = agent.select_actions(states, row_clues, col_clues)
            next_states, reward, done_step = env.step(actions)
            for i in range(train_batch_size):
                if not done[i]:
                    log_probs[i].append(log_prob[i])
                    rewards[i].append(reward[i])

            states = next_states
            done = np.logical_or(done, done_step)

        agent.update_policy(log_probs, rewards)
        correct_guesses = calculate_correct_guesses(states, env.solution)
        correct_guess_percent = correct_guesses / total_cells
        total_reward = np.mean([sum(r) for r in rewards])

        reward_list.append(total_reward)
        correct_guess_percent_list.append(correct_guess_percent.mean())

        writer.add_scalar('Train_Reward', total_reward, episode)
        writer.add_scalar('Train_Correct_Guess_Percent', correct_guess_percent.mean(), episode)

        if episode % save_interval == 0:
            save_checkpoint(agent, optimizer, episode, reward_list, correct_guess_percent_list, 

        with torch.no_grad():
            validation_env = NonogramEnvironment(grid_size, val_batch_size)
            validation_env.reset_with_solutions(validation_solutions, validation_row_clues, vali

            validation_states, validation_row_clues, validation_col_clues = validation_env.reset
            validation_row_clues = [pad_clues(rc, clue_max_len) for rc in validation_row_clues]
            validation_col_clues = [pad_clues(cc, clue_max_len) for cc in validation_col_clues]

            validation_done = np.zeros(val_batch_size, dtype=bool)
            validation_log_probs = [[] for _ in range(val_batch_size)]
            validation_rewards = [[] for _ in range(val_batch_size)]

            while not np.all(validation_done):
                validation_actions, validation_log_prob = agent.select_actions(
                    validation_states, validation_row_clues, validation_col_clues)
                validation_next_states, validation_reward, validation_done_step = validation_env
                for i in range(val_batch_size):
                    if not validation_done[i]:
                        validation_log_probs[i].append(validation_log_prob[i])
                        validation_rewards[i].append(validation_reward[i])

                validation_states = validation_next_states
                validation_done = np.logical_or(validation_done, validation_done_step)
```

```python
            validation_correct_guesses = calculate_correct_guesses(validation_states, validation_
            validation_correct_guess_percent = validation_correct_guesses / total_cells
            validation_total_reward = np.mean([sum(r) for r in validation_rewards])

            writer.add_scalar('Validation_Reward', validation_total_reward, episode)
            writer.add_scalar('Validation_Correct_Guess_Percent', validation_correct_guess_percen

        progress_bar.set_description(f"Train Reward: {total_reward:.2f}, Train Correct: {correct_

    writer.close()
```

Main Execution

In [ ]:

```python
if __name__ == "__main__":
    # Parameters for training
    grid_size = 5
    clue_max_len = divide_and_round_up(grid_size)
    clue_dim = grid_size
    num_episodes = 100000
    train_batch_size = 512
    val_batch_size = 128
    save_interval = 1000

    # Train the agent
    train_agent(grid_size, clue_max_len, clue_dim, num_episodes, train_batch_size, val_batch_siz
```

Testing Main Execution

In [18]:

```python
if __name__ == "__main__":
    # Load the pretrained model
    grid_size = 5
    clue_max_len = 3
    clue_dim = grid_size
    agent = NonogramAgent(grid_size, clue_max_len, clue_dim)
    optimizer = agent.optimizer
    _, _, _, clue_max_len, clue_dim = load_checkpoint(agent, optimizer)

    # Generate a random puzzle
    solutions, row_clues, col_clues, _ = generate_unique_nonogram(grid_size, 1)
    row_clues = [pad_clues(rc, clue_max_len) for rc in row_clues]
    col_clues = [pad_clues(cc, clue_max_len) for cc in col_clues]

    # Initialize the environment with the generated puzzle
    env = NonogramEnvironment(grid_size, 1)
    states, row_clues, col_clues = env.reset_with_solutions(solutions, row_clues, col_clues)

    # Display clues
    print("Row Clues:")
    visualize_clues(row_clues)
    print("Column Clues:")
    visualize_clues(col_clues)

    # Display initial puzzle state
    print("Initial Puzzle State:")
    visualize_nonogram(states[0])

    # Solve the puzzle
    update_puzzle_state(agent, env, states, row_clues, col_clues, solutions)
```

```
Row Clues:
[[1, 3, 0], [2, 1, 0], [2, 1, 0], [2, 2, 0], [2, 2, 0]]
Column Clues:
[[1, 2, 0], [4, 0, 0], [3, 0, 0], [1, 2, 0], [5, 0, 0]]
Initial Puzzle State:
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?

Move: 1, Guess: (4, 2), State: 0
Current Puzzle State:
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
? 1 ? ? ?
? ? ? ? ?

Move: 2, Guess: (4, 4), State: 0
Current Puzzle State:
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
? 1 ? 1 ?
? ? ? ? ?

Move: 3, Guess: (4, 3), State: 1
Current Puzzle State:
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
? 1 0 1 ?
? ? ? ? ?

Move: 4, Guess: (2, 4), State: 1
Current Puzzle State:
? ? ? ? ?
? ? ? 0 ?
? ? ? ? ?
? 1 0 1 ?
? ? ? ? ?

Move: 5, Guess: (3, 2), State: 0
Current Puzzle State:
? ? ? ? ?
? ? ? 0 ?
? 1 ? ? ?
? 1 0 1 ?
? ? ? ? ?

Move: 6, Guess: (4, 5), State: 1
Current Puzzle State:
? ? ? ? ?
? ? ? 0 ?
? 1 ? ? ?
? 1 0 1 1
? ? ? ? ?

Move: 7, Guess: (2, 2), State: 1
Current Puzzle State:
? ? ? ? ?
```

```
? 1 ? 0 ?
? 1 ? ? ?
? 1 0 1 1
? ? ? ? ?

Move: 8, Guess: (1, 2), State: 0
Current Puzzle State:
? 0 ? ? ?
? 1 ? 0 ?
? 1 ? ? ?
? 1 0 1 1
? ? ? ? ?

Move: 9, Guess: (2, 5), State: 0
Current Puzzle State:
? 0 ? ? ?
? 1 ? 0 1
? 1 ? ? ?
? 1 0 1 1
? ? ? ? ?

Move: 10, Guess: (2, 3), State: 0
Current Puzzle State:
? 0 ? ? ?
? 1 1 0 1
? 1 ? ? ?
? 1 0 1 1
? ? ? ? ?

Move: 11, Guess: (5, 3), State: 1
Current Puzzle State:
? 0 ? ? ?
? 1 1 0 1
? 1 ? ? ?
? 1 0 1 1
? ? 0 ? ?

Move: 12, Guess: (1, 3), State: 1
Current Puzzle State:
? 0 1 ? ?
? 1 1 0 1
? 1 ? ? ?
? 1 0 1 1
? ? 0 ? ?

Move: 13, Guess: (3, 3), State: 0
Current Puzzle State:
? 0 1 ? ?
? 1 1 0 1
? 1 1 ? ?
? 1 0 1 1
? ? 0 ? ?

Move: 14, Guess: (3, 4), State: 0
Current Puzzle State:
? 0 1 ? ?
? 1 1 0 1
? 1 1 0 ?
? 1 0 1 1
? ? 0 ? ?

Move: 15, Guess: (4, 1), State: 1
```

```
Current Puzzle State:
? 0 1 ? ?
? 1 1 0 1
? 1 1 0 ?
1 1 0 1 1
? ? 0 ? ?

Move: 16, Guess: (1, 5), State: 1
Current Puzzle State:
? 0 1 ? 1
? 1 1 0 1
? 1 1 0 ?
1 1 0 1 1
? ? 0 ? ?

Move: 17, Guess: (1, 4), State: 1
Current Puzzle State:
? 0 1 1 1
? 1 1 0 1
? 1 1 0 ?
1 1 0 1 1
? ? 0 ? ?

Move: 18, Guess: (5, 2), State: 1
Current Puzzle State:
? 0 1 1 1
? 1 1 0 1
? 1 1 0 ?
1 1 0 1 1
? 1 0 ? ?

Move: 19, Guess: (3, 1), State: 0
Current Puzzle State:
? 0 1 1 1
? 1 1 0 1
0 1 1 0 ?
1 1 0 1 1
? 1 0 ? ?

Move: 20, Guess: (2, 1), State: 1
Current Puzzle State:
? 0 1 1 1
0 1 1 0 1
0 1 1 0 ?
1 1 0 1 1
? 1 0 ? ?

Move: 21, Guess: (1, 1), State: 0
Current Puzzle State:
1 0 1 1 1
0 1 1 0 1
0 1 1 0 ?
1 1 0 1 1
? 1 0 ? ?

Move: 22, Guess: (5, 4), State: 0
Current Puzzle State:
1 0 1 1 1
0 1 1 0 1
0 1 1 0 ?
1 1 0 1 1
? 1 0 1 ?
```

```
Move: 23, Guess: (5, 5), State: 1
Current Puzzle State:
1 0 1 1 1
0 1 1 0 1
0 1 1 0 ?
1 1 0 1 1
? 1 0 1 1

Move: 24, Guess: (3, 5), State: 0
Current Puzzle State:
1 0 1 1 1
0 1 1 0 1
0 1 1 0 1
1 1 0 1 1
? 1 0 1 1

Move: 25, Guess: (5, 1), State: 0
Current Puzzle State:
1 0 1 1 1
0 1 1 0 1
0 1 1 0 1
1 1 0 1 1
1 1 0 1 1
Puzzle Solved!
```

In [ ]: