EE449 Computational Intelligence Homework 3 Report

Deniz Soysal 230533

May 27, 2024

1 Basic Questions

None of the terms below has a direct counterpart in supervised learning. However; they can be related to some terms related to supervised learning in terms of their functionality.

- Agent is the decision-maker that learns from the process which makes it analogous to the model in supervised learning
- Environment is the external system that the agent interacts and learns from which can be related to the dataset that is used to train the model in the supervised learning
- Reward is the feedback that the agent gets after making a decision that has similar functionality to the result of the loss function that is used in backpropagation(in neural networks) to train the model.
- Policy is the mapping from states to the actions which can be loosely related to the decision boundaries of supervised learning
- Exploration is the learning process in which the agent develops the best strategy by taking action. In supervised learning, this can be loosely related to the training process in which the model makes decisions on the dataset and gets feedback from the loss function.

2 Experimental Work

Effect of alpha in TD learning

(a) utility values (b) policies

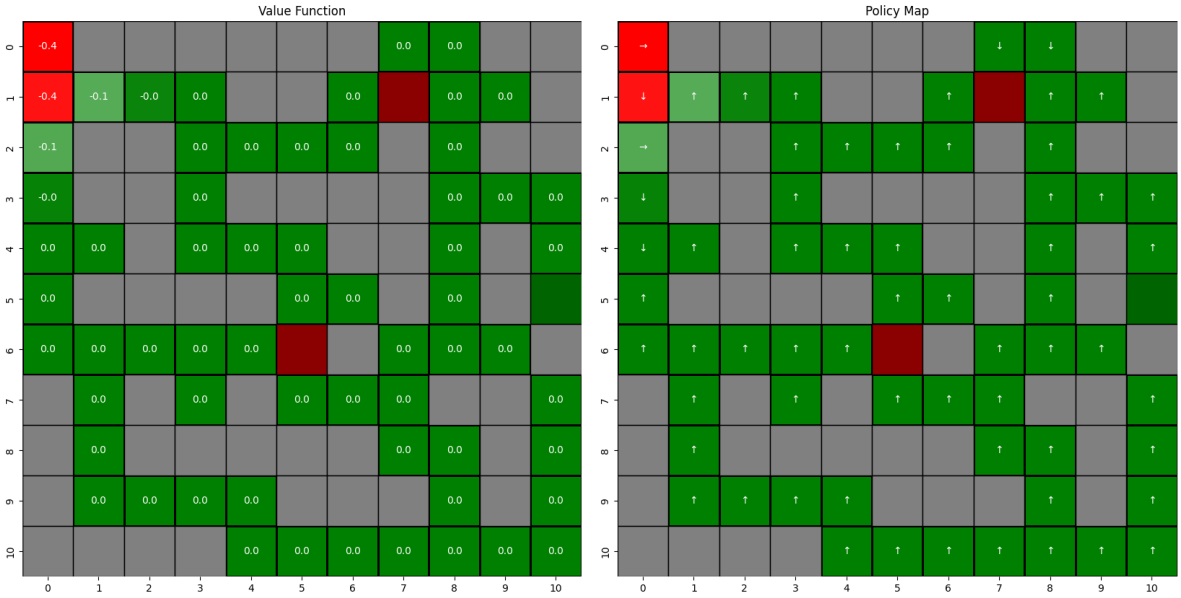


Figure 1: alpha = 0.001 for 1 episode

(a) utility values (b) policies

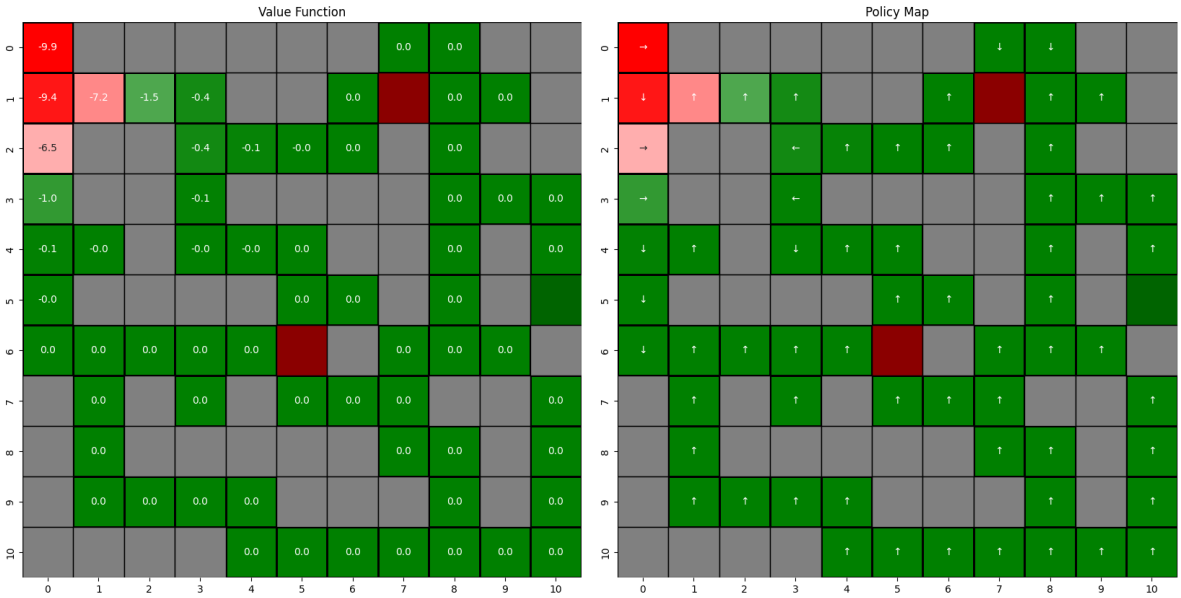


Figure 2: alpha = 0.001 for 50 episode

(a) utility values (b) policies

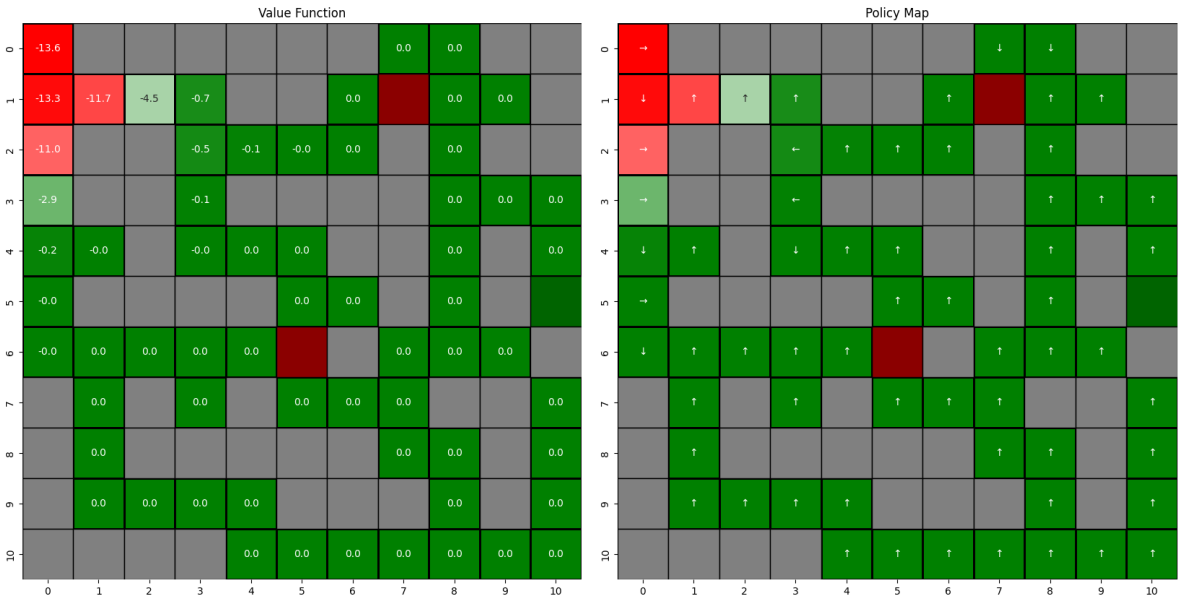


Figure 3: alpha = 0.001 for 100 episode

(a) utility values (b) policies

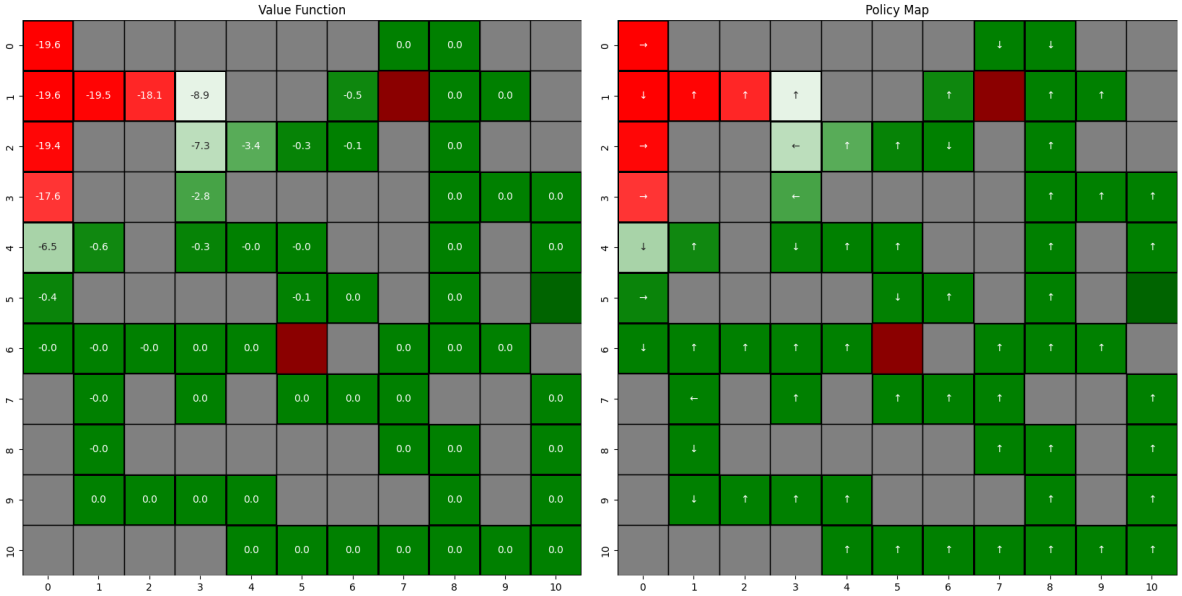


Figure 4: alpha = 0.001 for 1000 episode

(a) utility values (b) policies

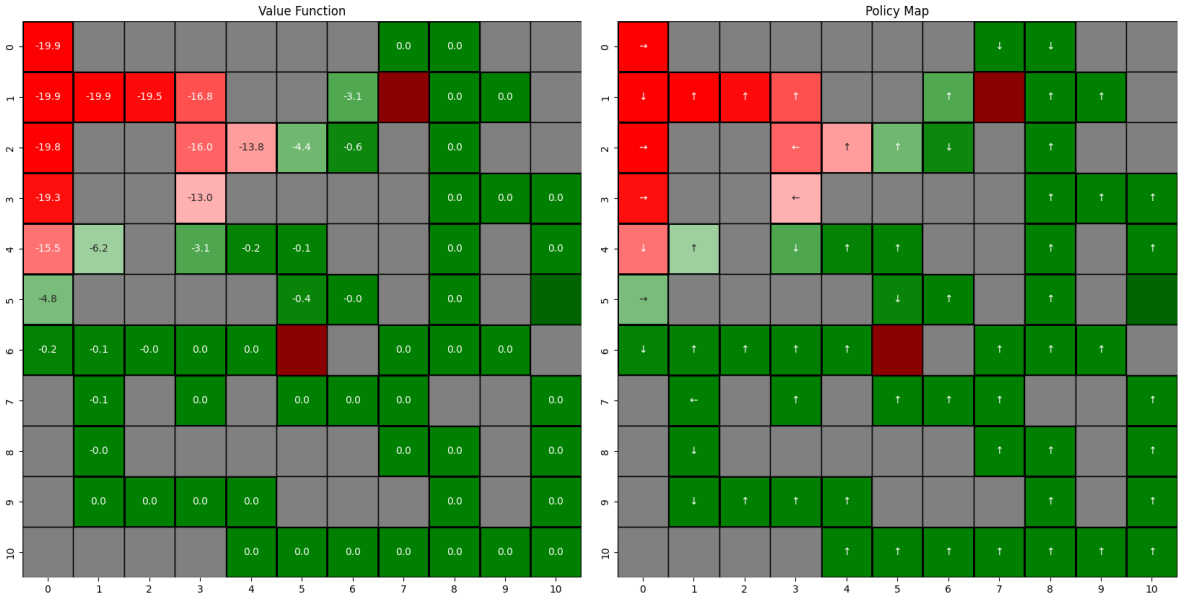


Figure 5: alpha = 0.001 for 5000 episode

(a) utility values (b) policies

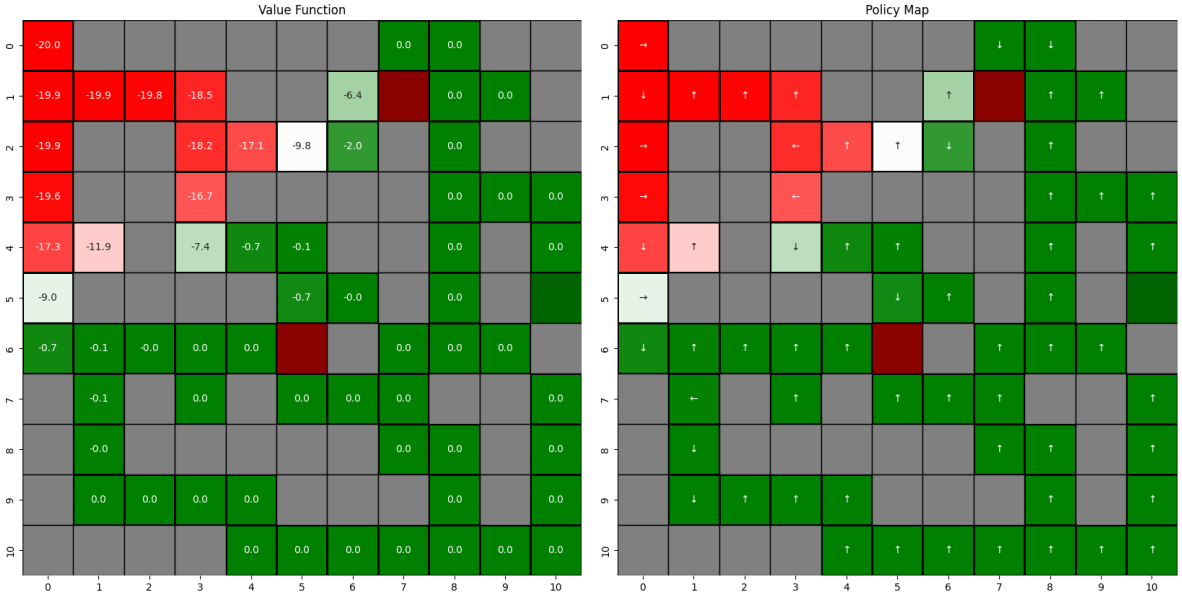


Figure 6: alpha = 0.001 for 10000 episode

(a) utility values (b) policies

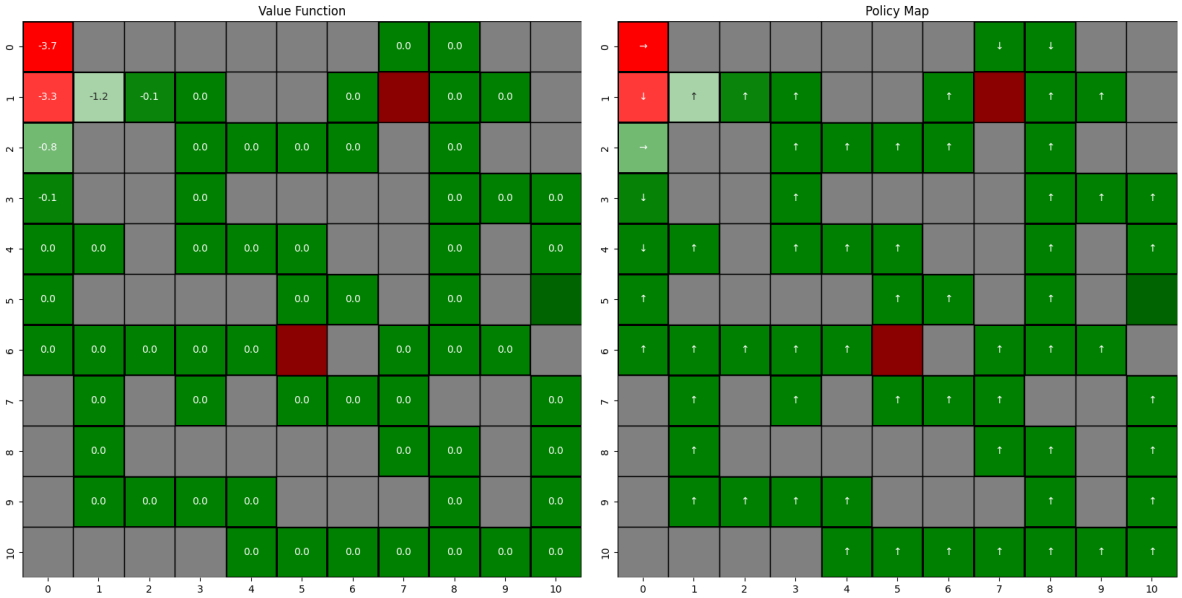


Figure 7: alpha = 0.01 for 1 episode

(a) utility values (b) policies

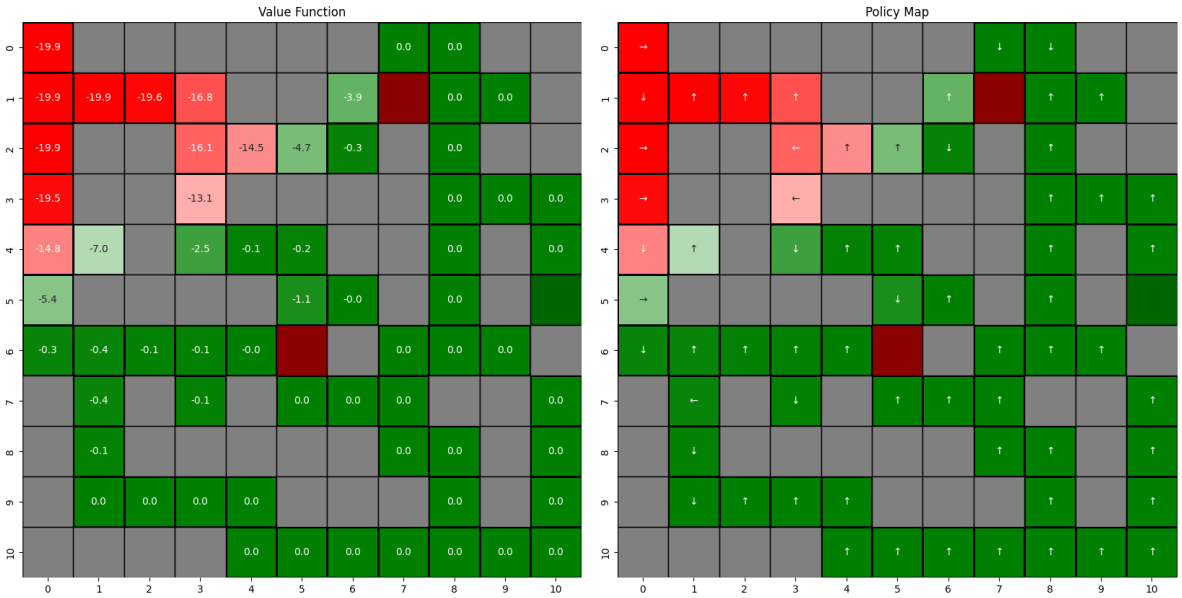


Figure 8: alpha = 0.01 for 50 episode

(a) utility values (b) policies

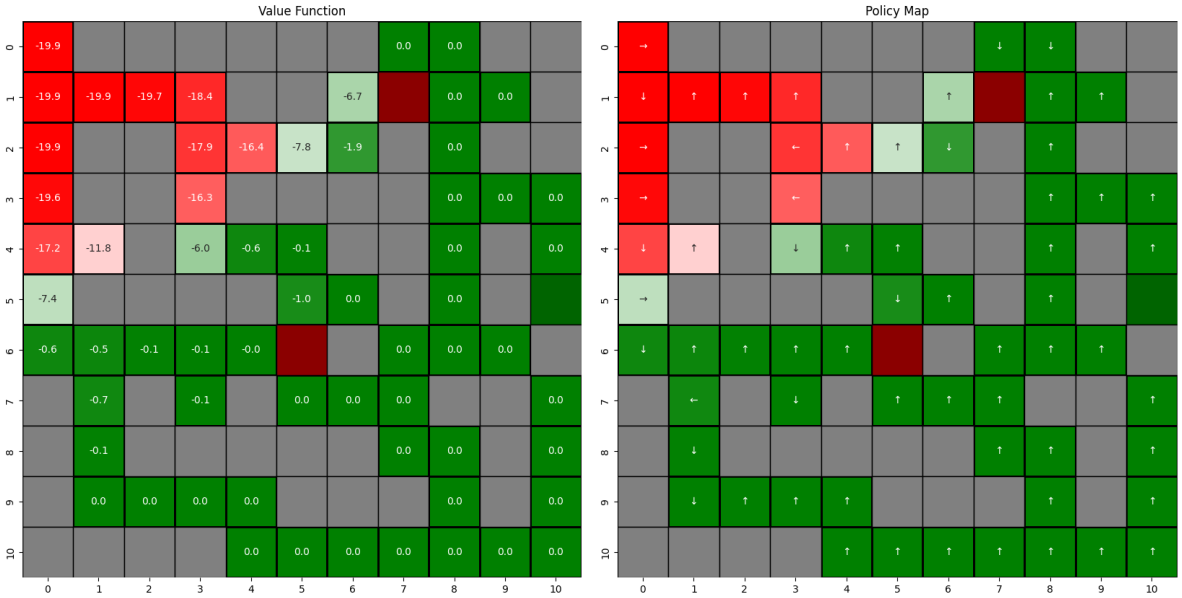


Figure 9: alpha = 0.01 for 100 episode

(a) utility values (b) policies

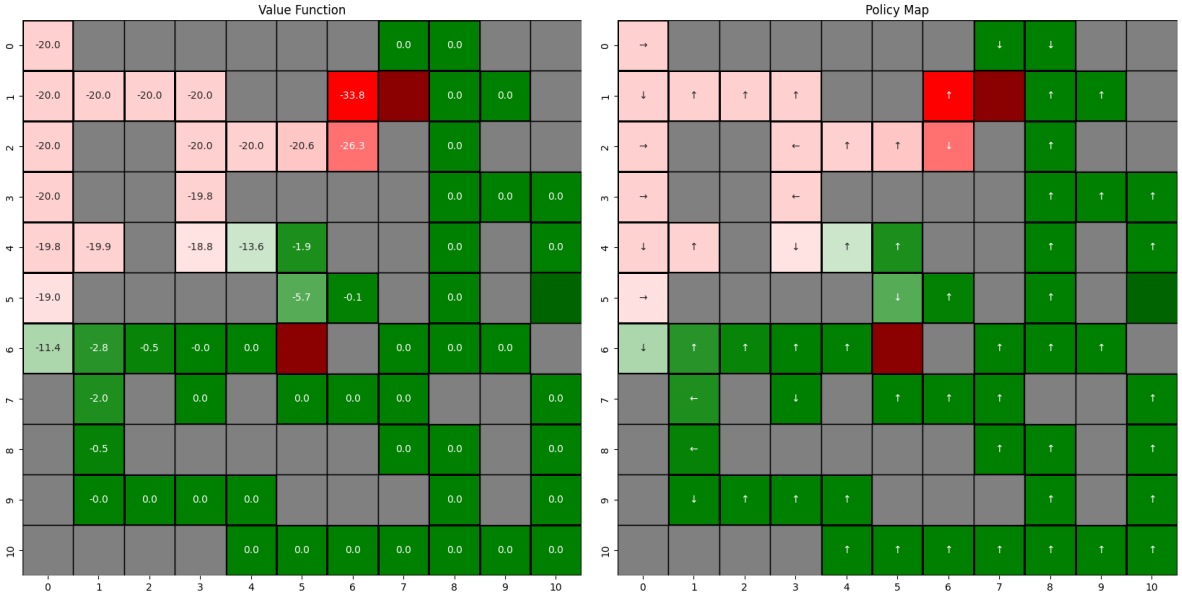


Figure 10: alpha = 0.01 for 1000 episode

(a) utility values (b) policies

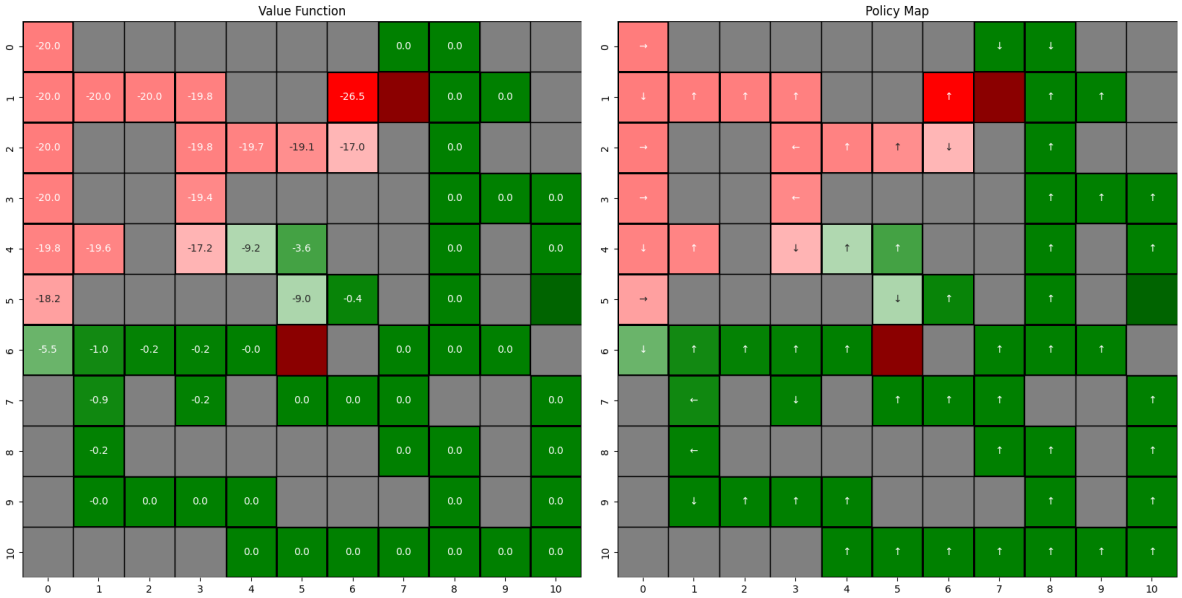


Figure 11: alpha = 0.01 for 5000 episode

(a) utility values (b) policies

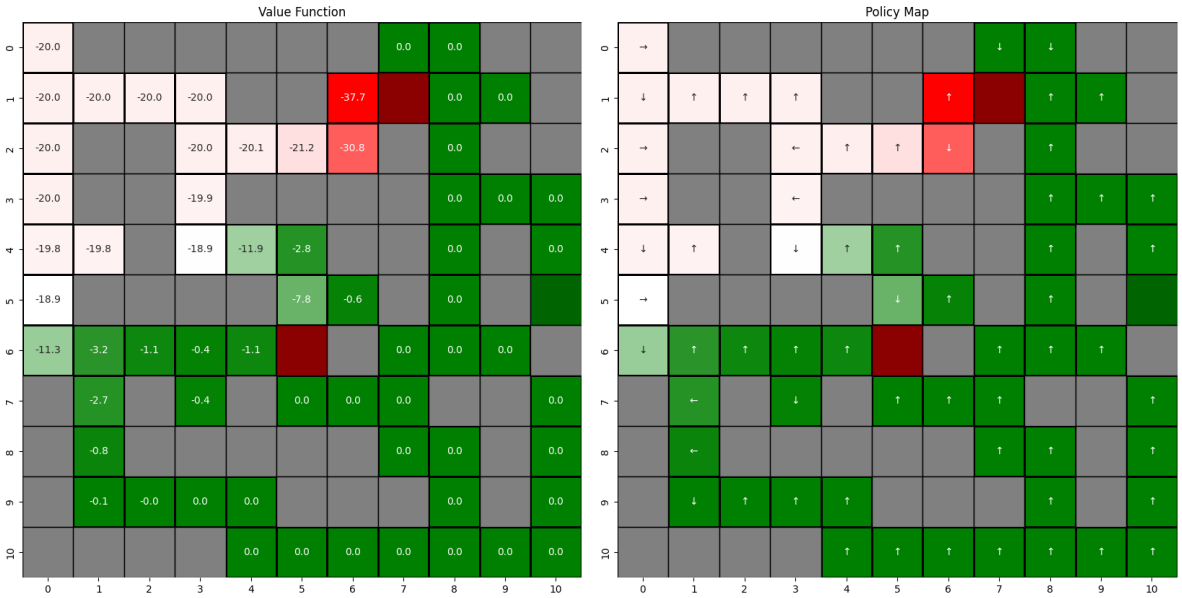


Figure 12: alpha = 0.01 for 10000 episode

(a) utility values (b) policies

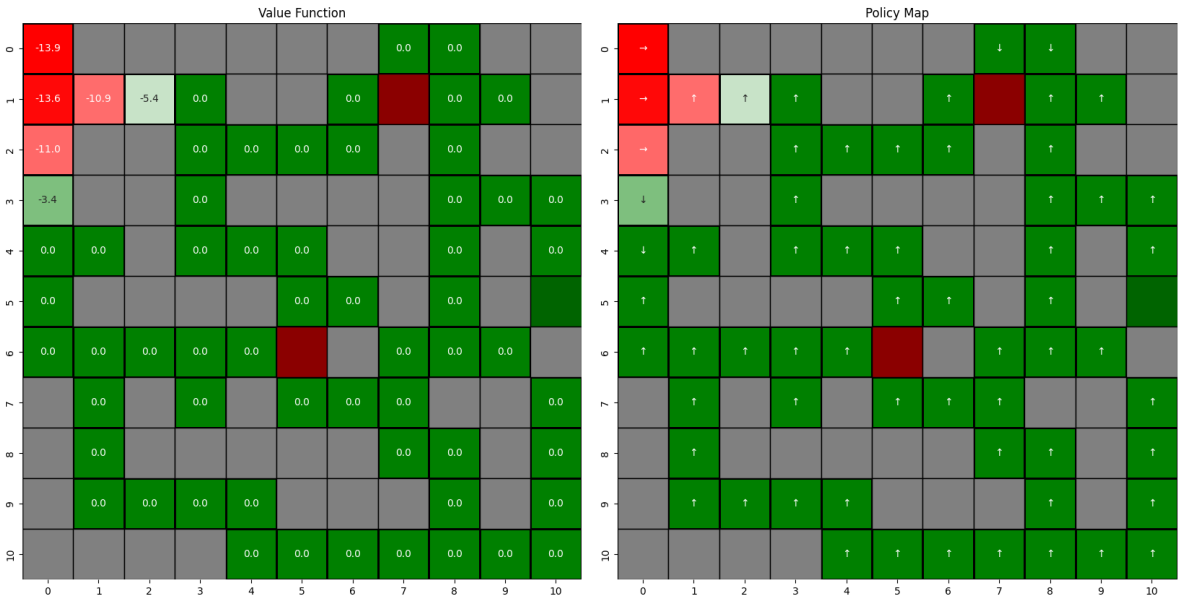


Figure 13: alpha = 0.1 for 1 episode

(a) utility values (b) policies

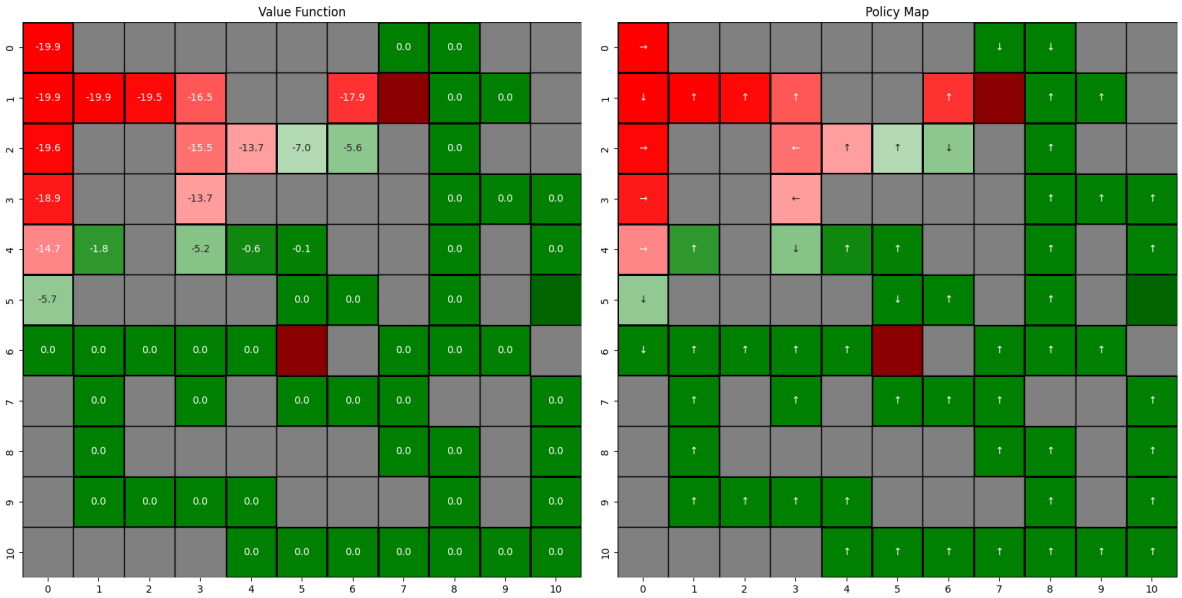


Figure 14: alpha = 0.1 for 50 episode

(a) utility values (b) policies

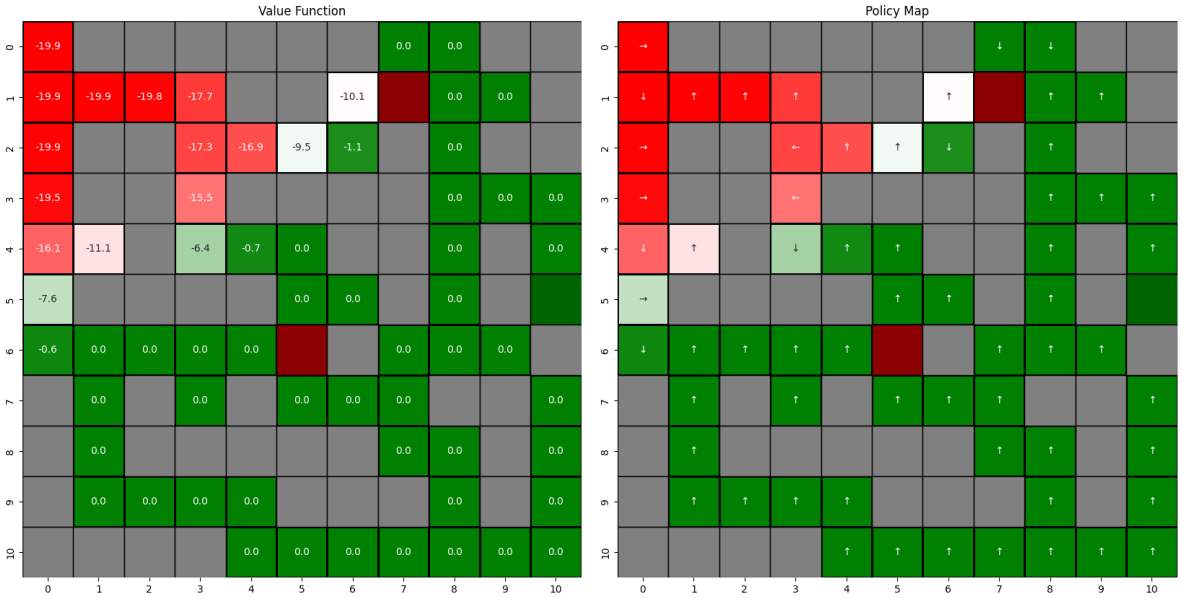


Figure 15: alpha = 0.1 for 100 episode

(a) utility values (b) policies

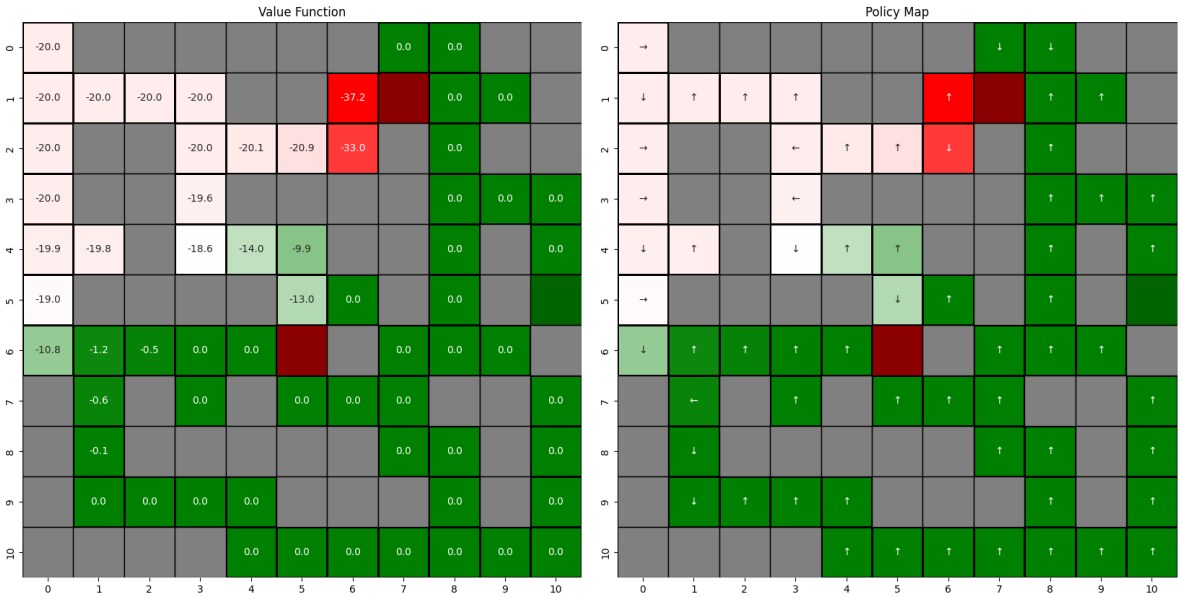
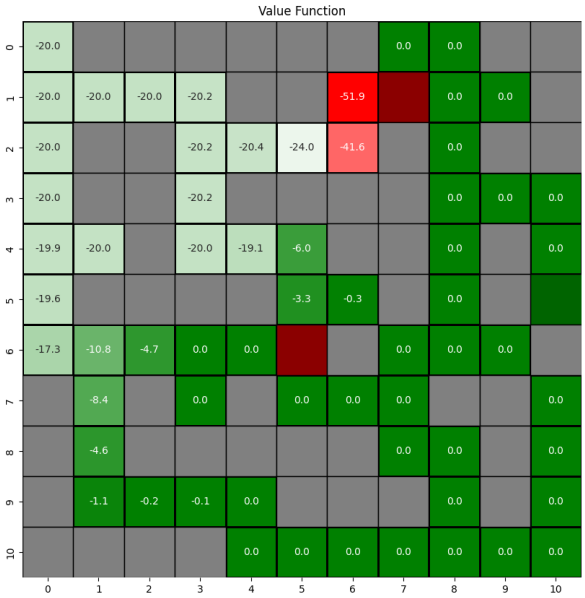


Figure 16: alpha = 0.1 for 1000 episode

(a) utility values



(b) policies

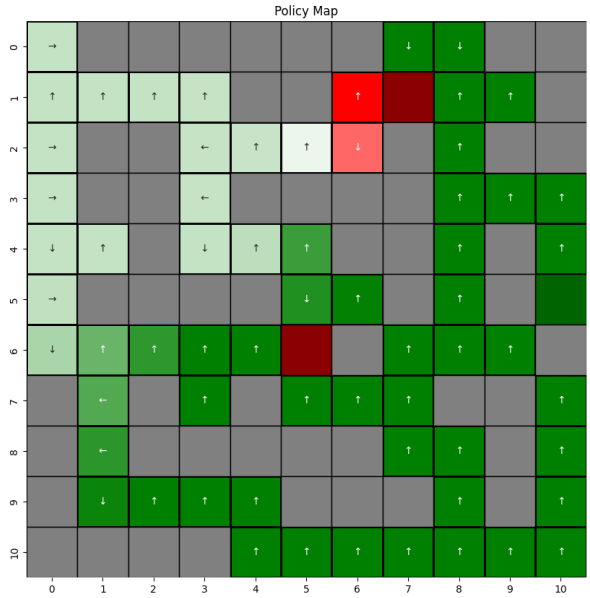
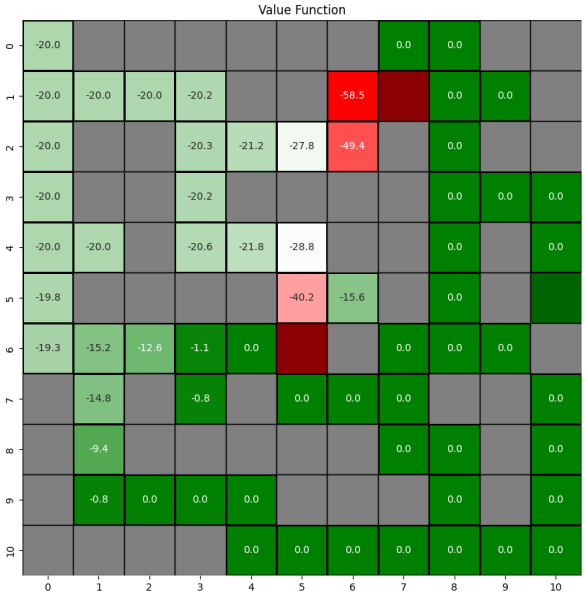


Figure 17: alpha = 0.1 for 5000 episode

(a) utility values



(b) policies

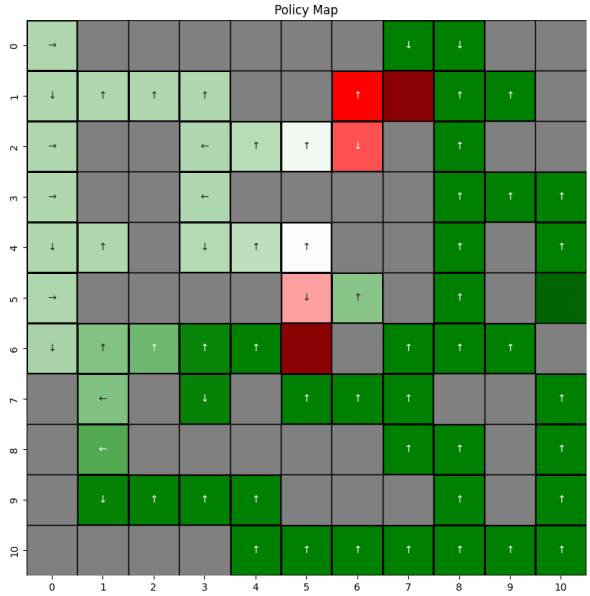


Figure 18: alpha = 0.1 for 10000 episode

(a) utility values (b) policies

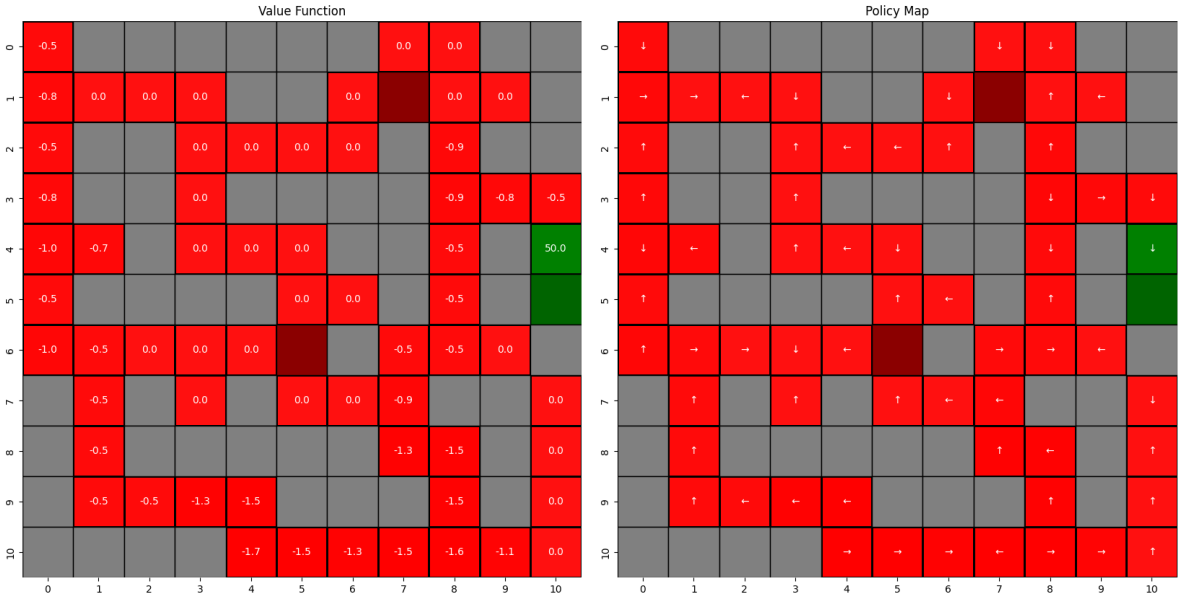


Figure 19: alpha = 0.5 for 1 episode

(a) utility values (b) policies

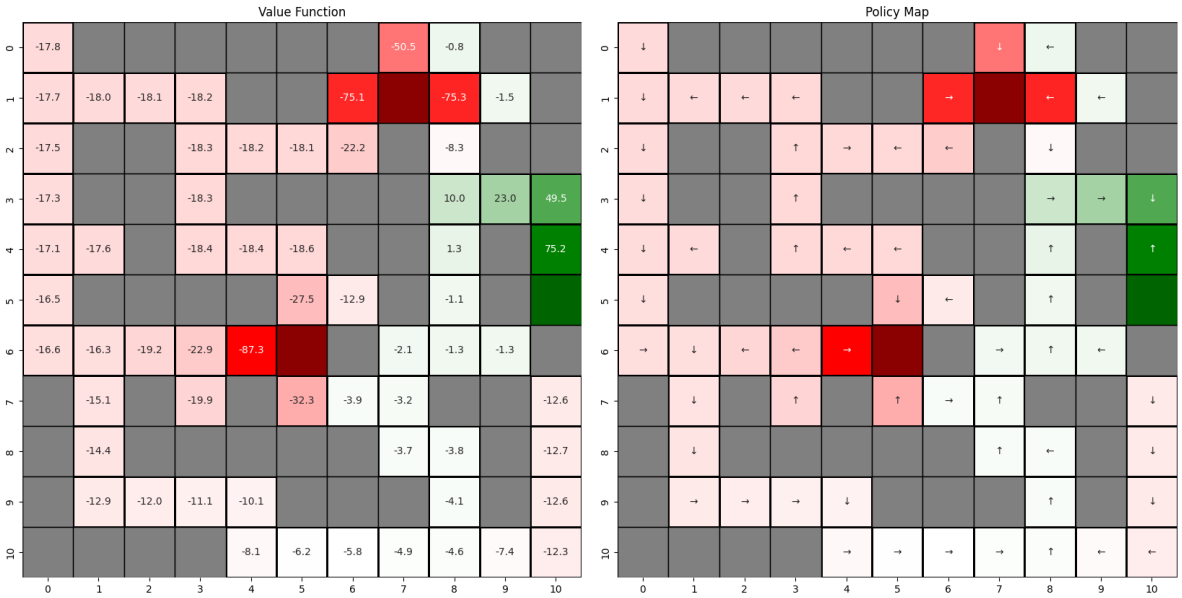
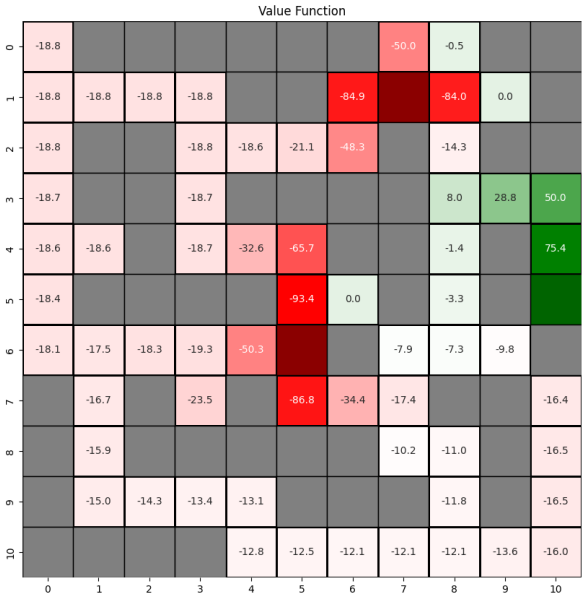


Figure 20: alpha = 0.5 for 50 episode

(a) utility values



(b) policies

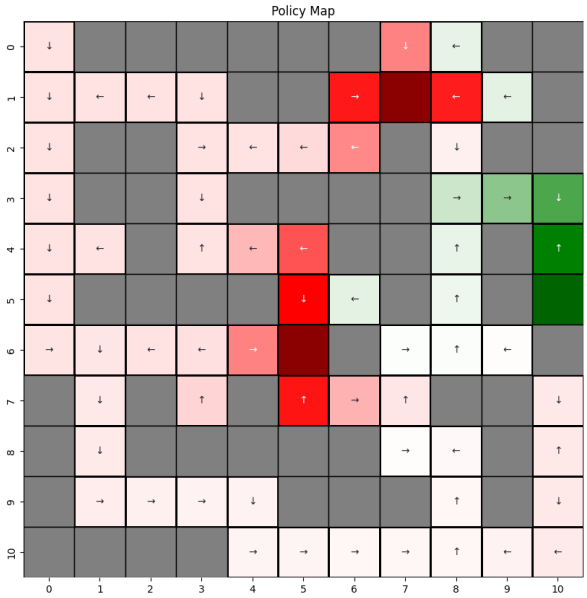
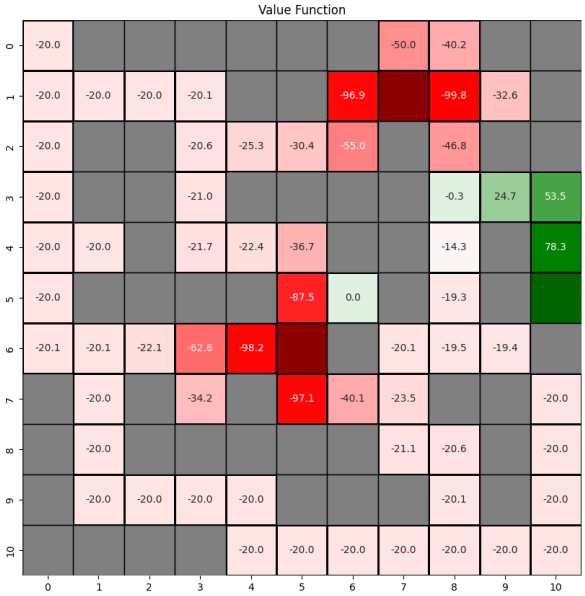


Figure 21: alpha = 0.5 for 100 episode

(a) utility values



(b) policies

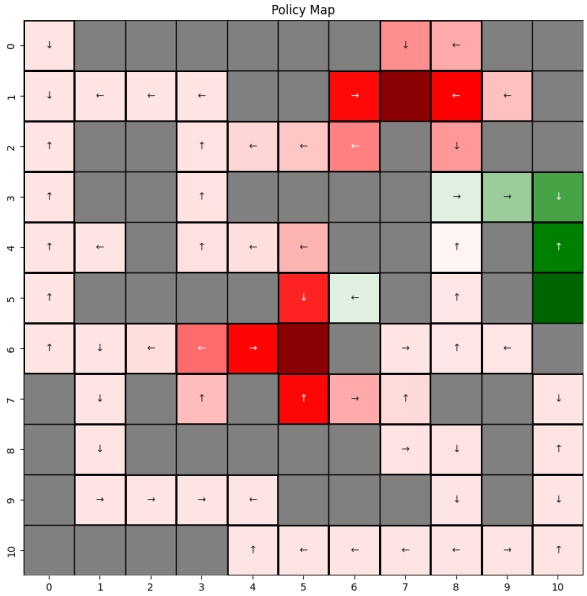
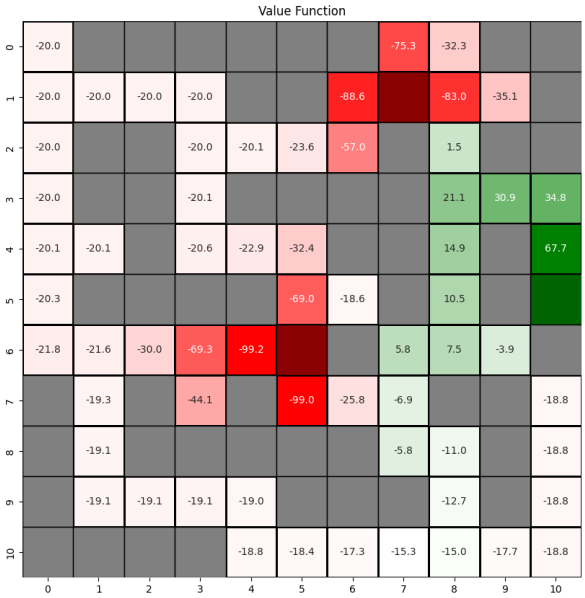


Figure 22: alpha = 0.5 for 1000 episode

(a) utility values



(b) policies

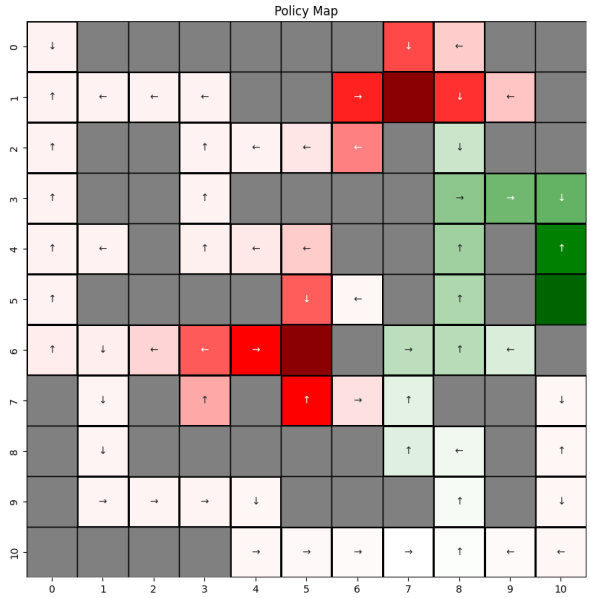
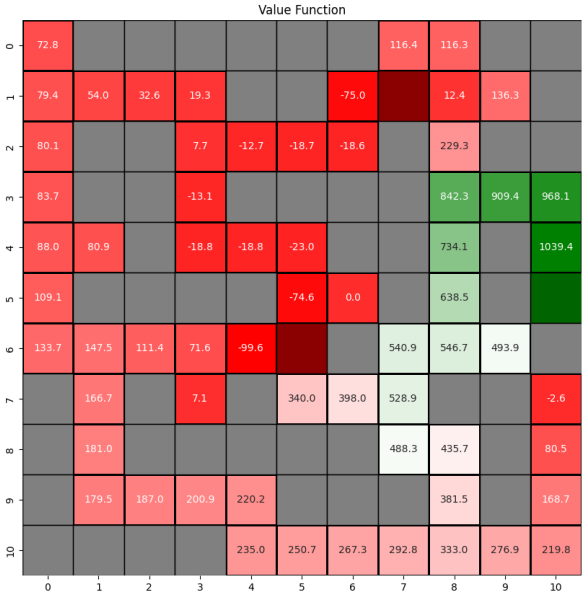


Figure 23: alpha = 0.5 for 5000 episode

(a) utility values



(b) policies

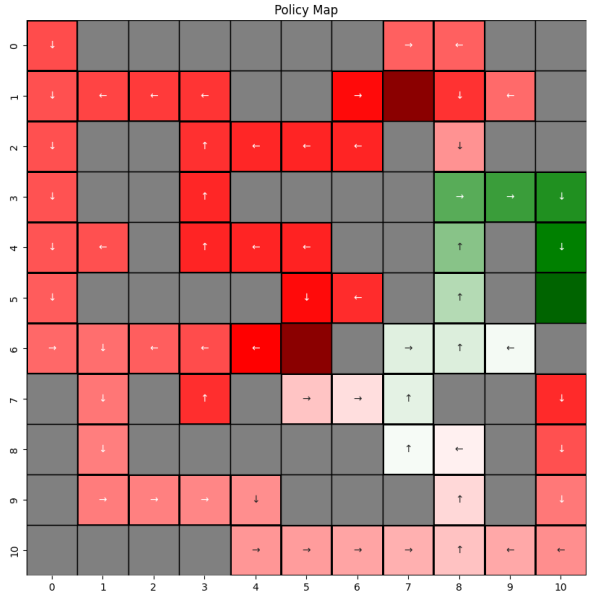


Figure 24: alpha = 0.5 for 10000 episode

(a) utility values (b) policies

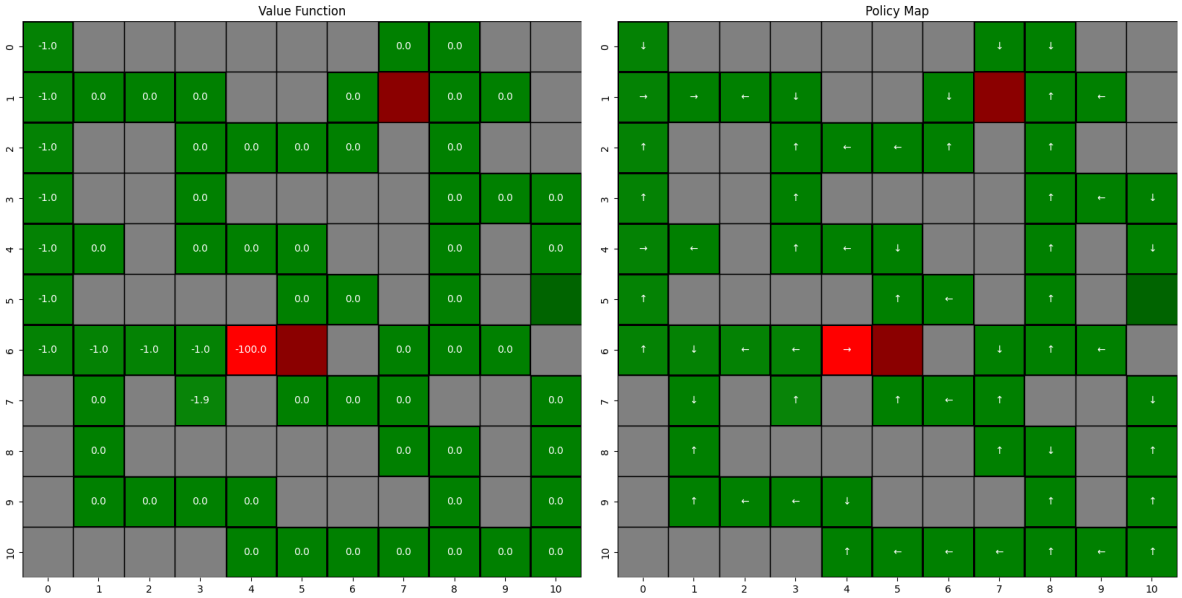


Figure 25: alpha = 1 for 1 episode

(a) utility values (b) policies

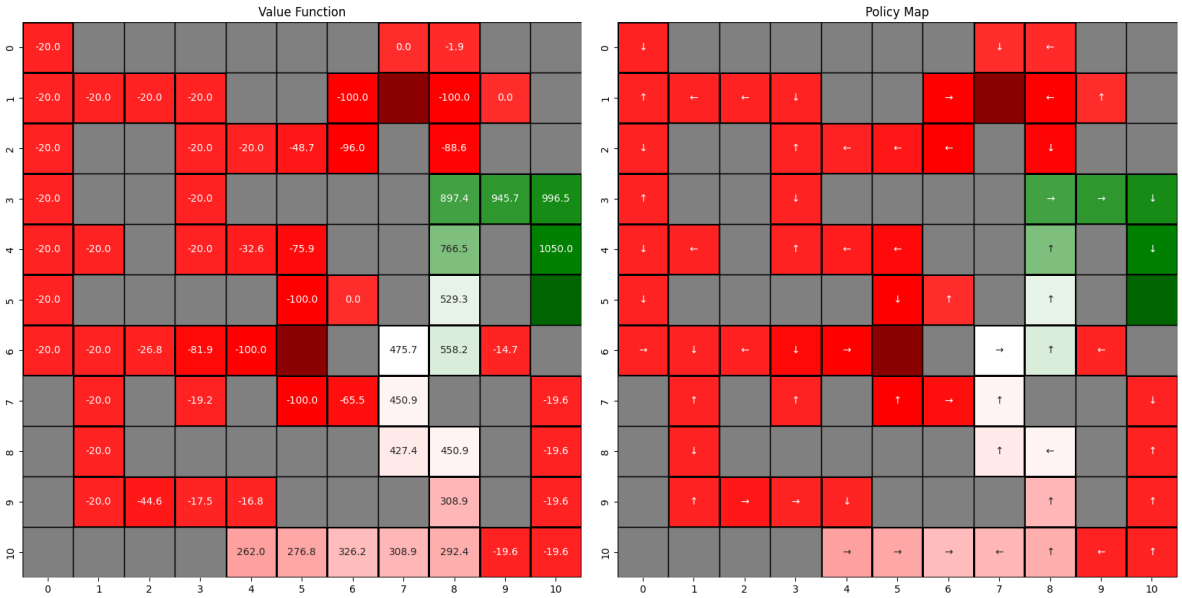
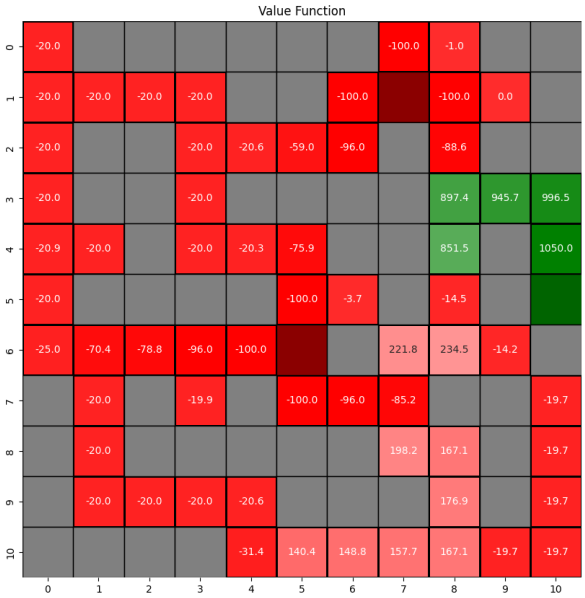


Figure 26: alpha = 1 for 50 episode

(a) utility values



(b) policies

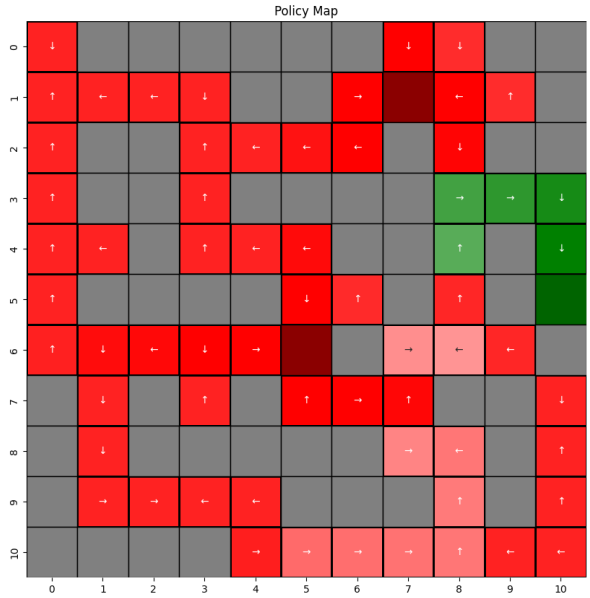
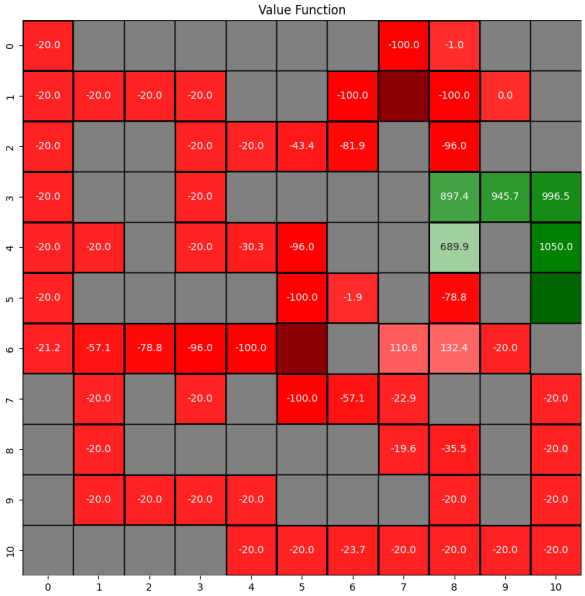


Figure 27: alpha = 1 for 100 episode

(a) utility values



(b) policies

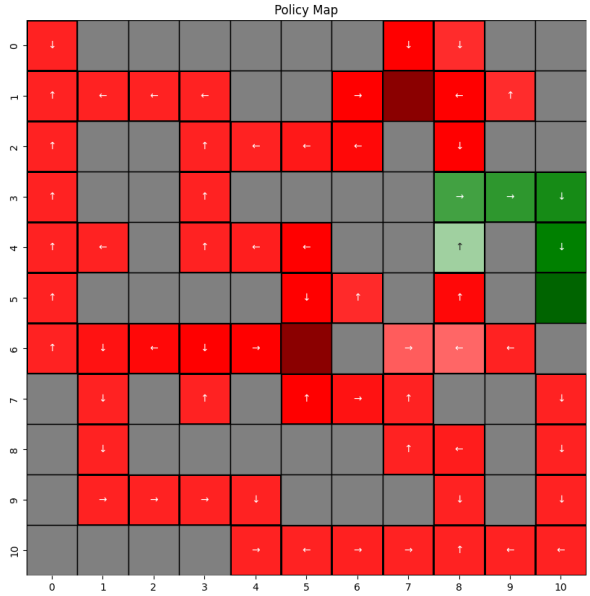
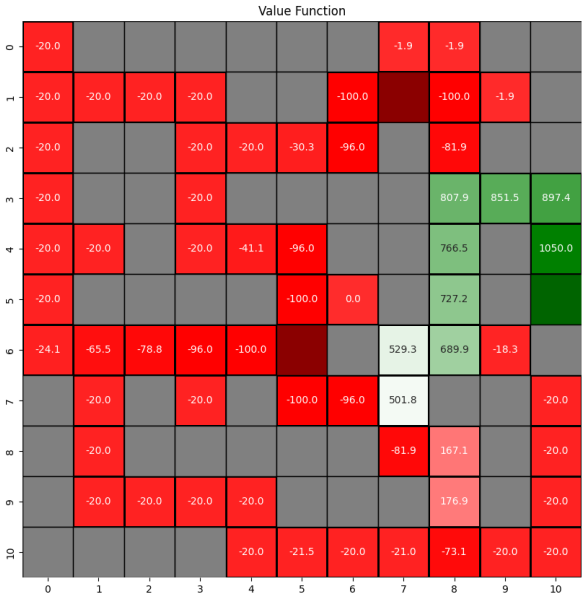


Figure 28: alpha = 1 for 1000 episode

(a) utility values



(b) policies

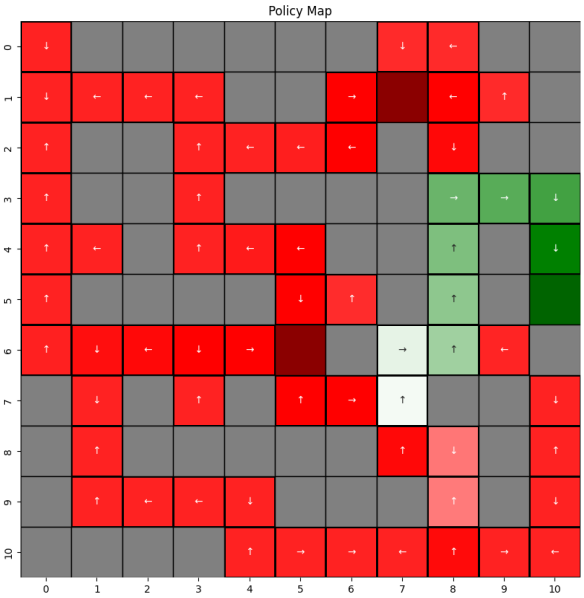
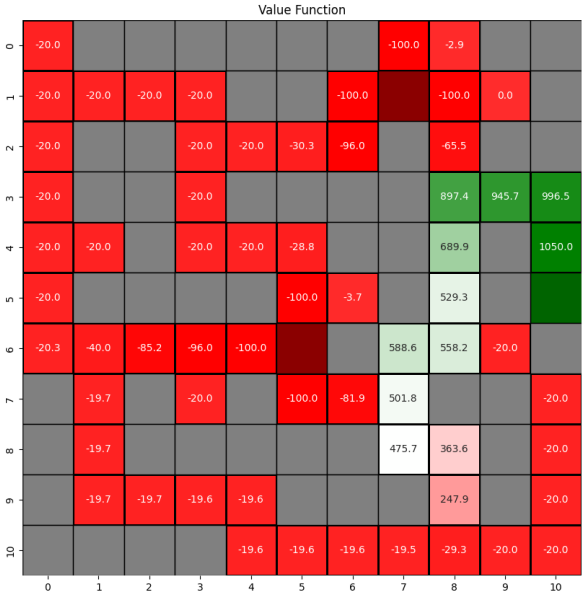


Figure 29: alpha = 1 for 5000 episode

(a) utility values



(b) policies

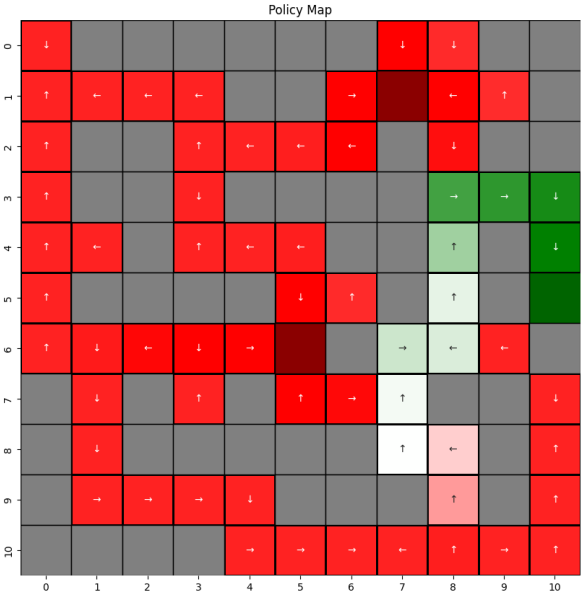
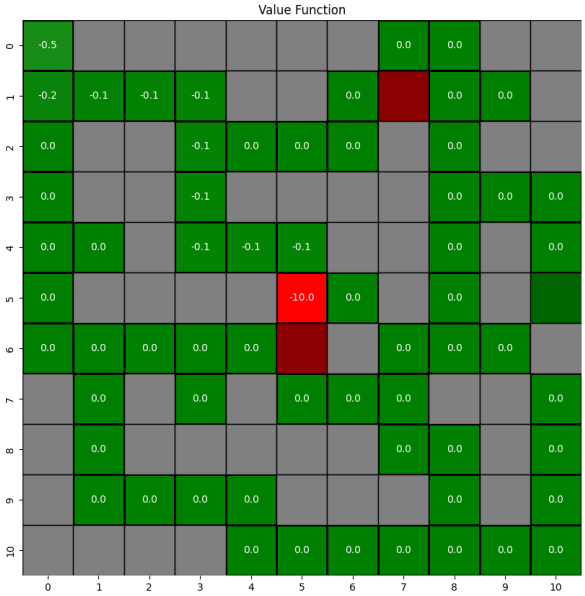


Figure 30: alpha = 1 for 10000 episode

In this section, simulations yielded successful results for 0.5 alpha at 50, 100, and 10000 episodes. First of all, in this section, design modifications have been made to optimize the system. obstacles have been given a -100 utility value, also the goal state has been given a 1000 utility value. Other than modifications alpha value can be considered as the learning rate even after modifications increased learning rate has been observed to be detrimental after some point as it can lead to harmful convergence where the goal has little effect on the far-away states while their utility value quickly converge so that it traps the agent.

Effect of epsilon in TD learning

(a) utility values



(b) policies

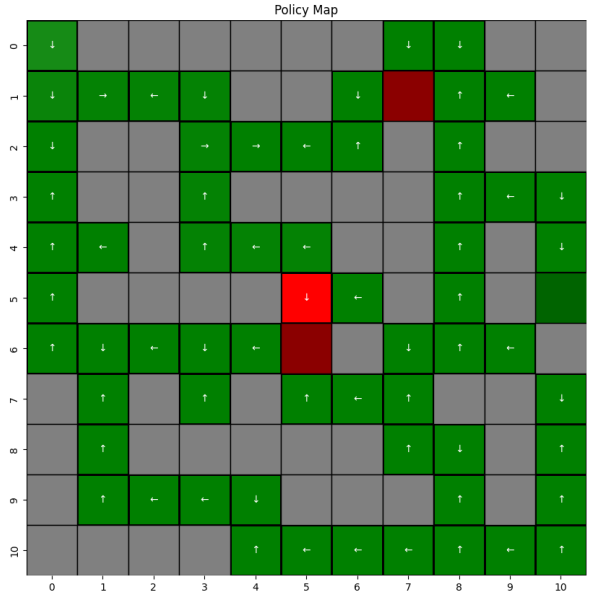
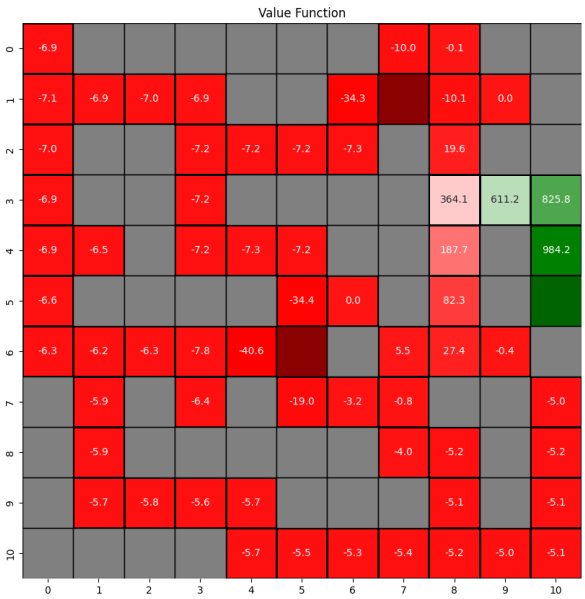


Figure 31: epsilon = 0 for 1 episode

(a) utility values



(b) policies

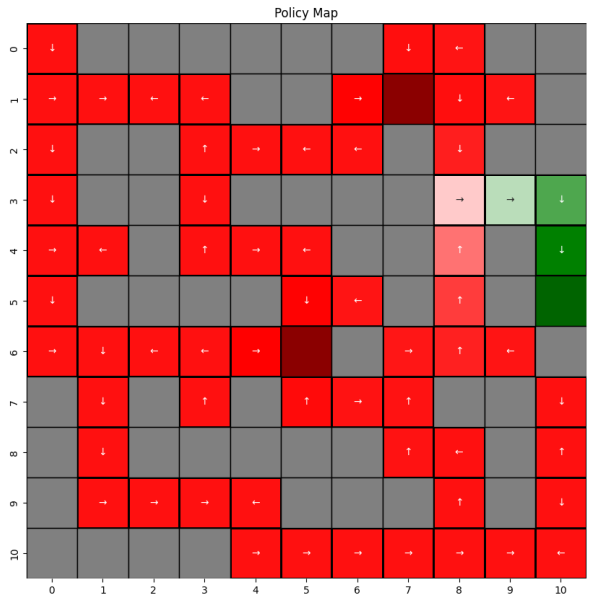


Figure 32: epsilon = 0 for 50 episode

(a) utility values (b) policies

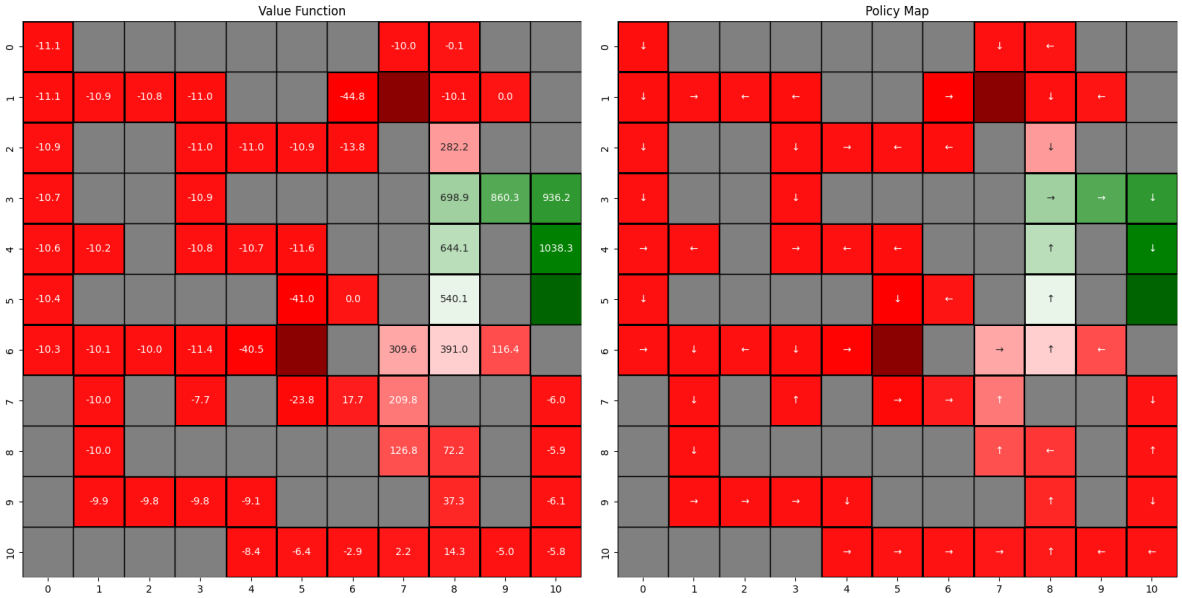


Figure 33: epsilon = 0 for 100 episode

(a) utility values (b) policies

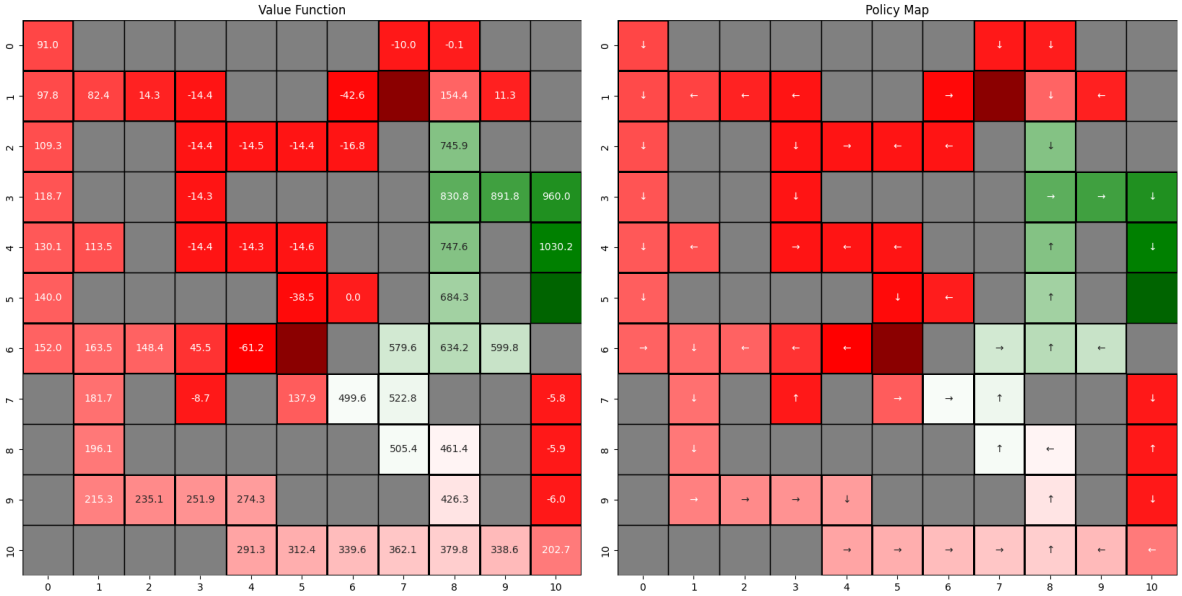
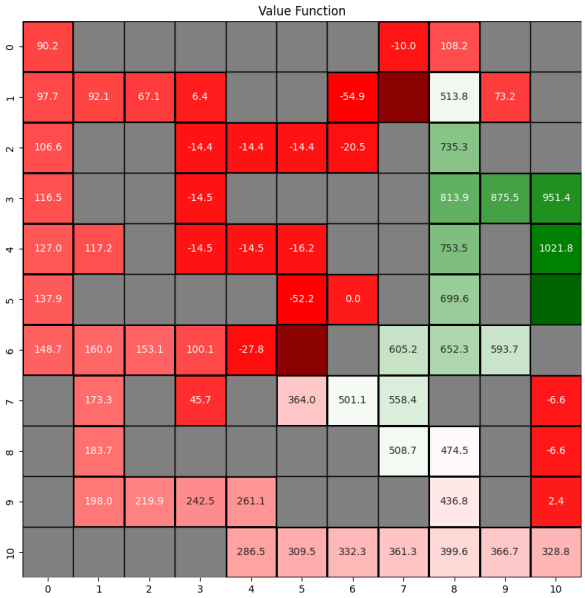


Figure 34: epsilon = 0 for 1000 episode

(a) utility values



(b) policies

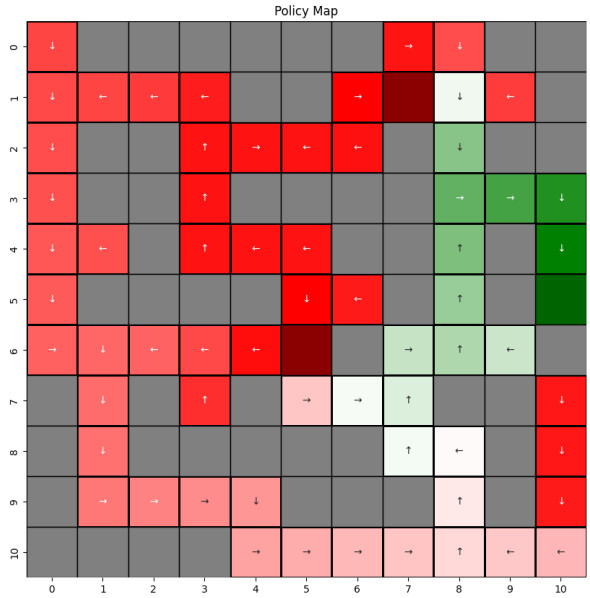
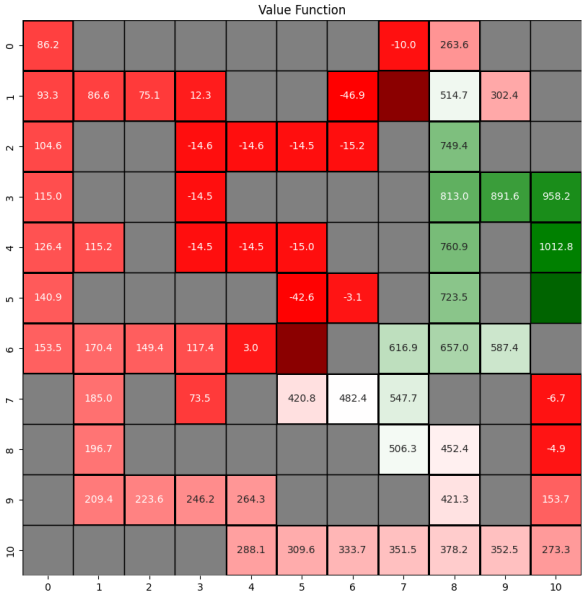


Figure 35: epsilon = 0 for 5000 episode

(a) utility values



(b) policies

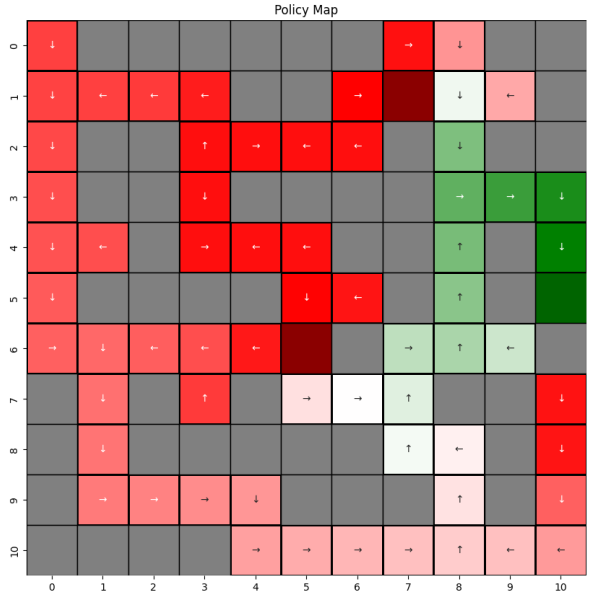
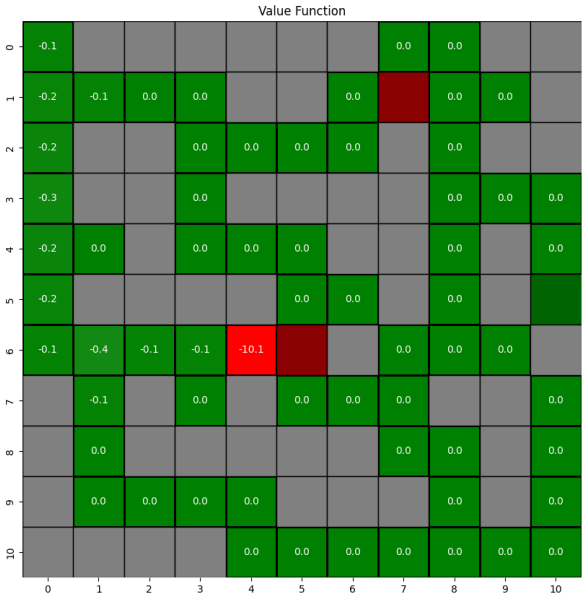


Figure 36: epsilon = 0 for 10000 episode

(a) utility values



(b) policies

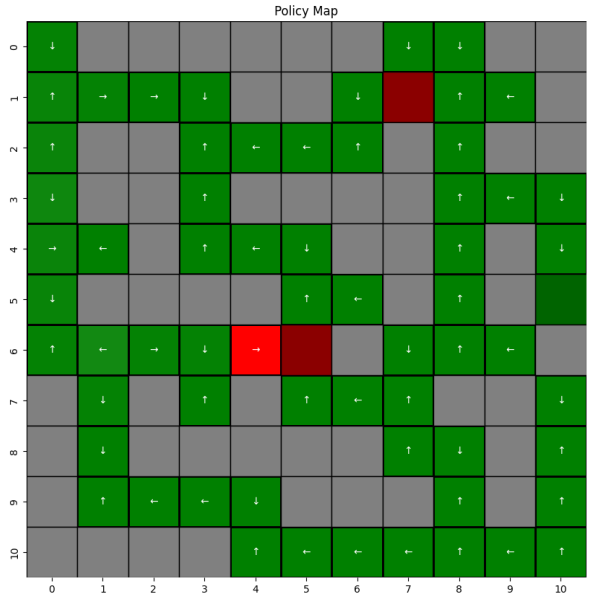
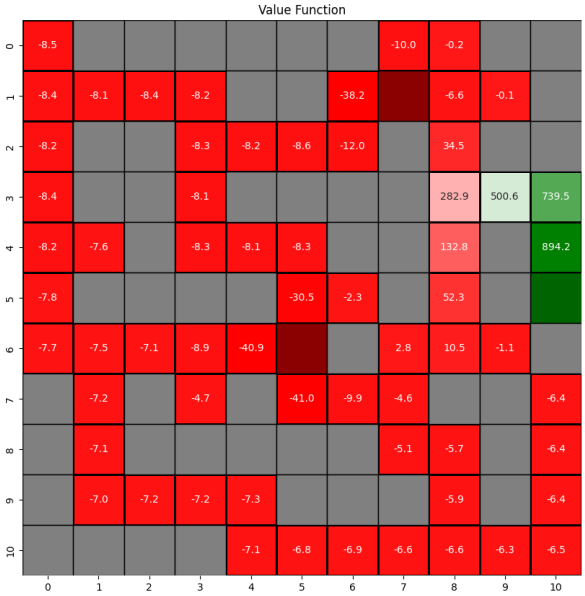


Figure 37: epsilon = 0.2 for 1 episode

(a) utility values



(b) policies

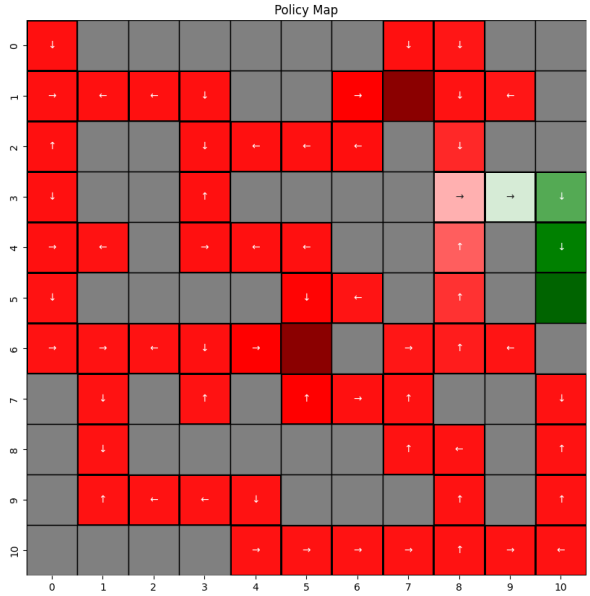
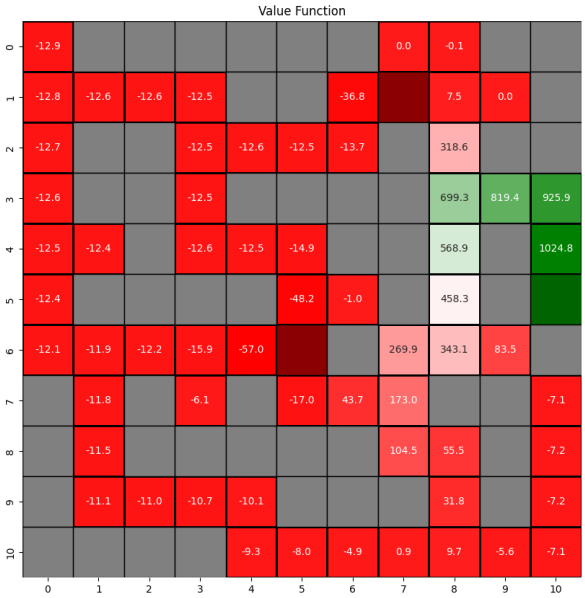


Figure 38: epsilon = 0.2 for 50 episode

(a) utility values



(b) policies

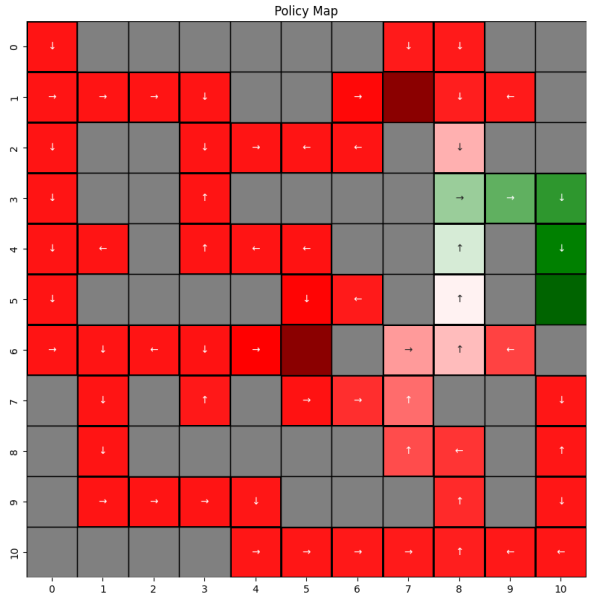
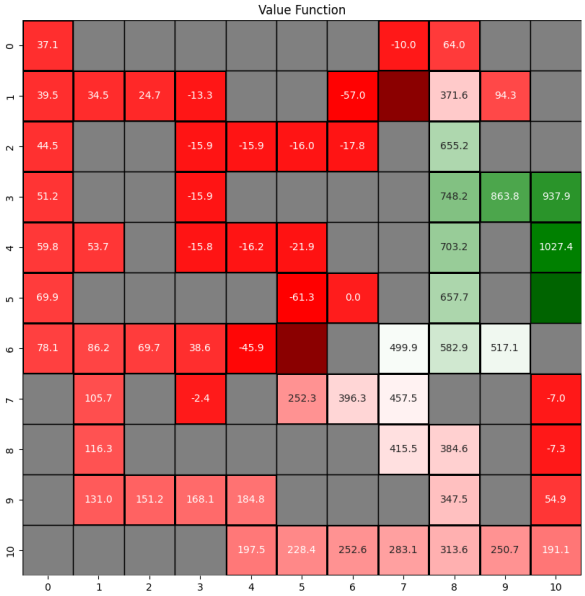


Figure 39: epsilon = 0.2 for 100 episode

(a) utility values



(b) policies

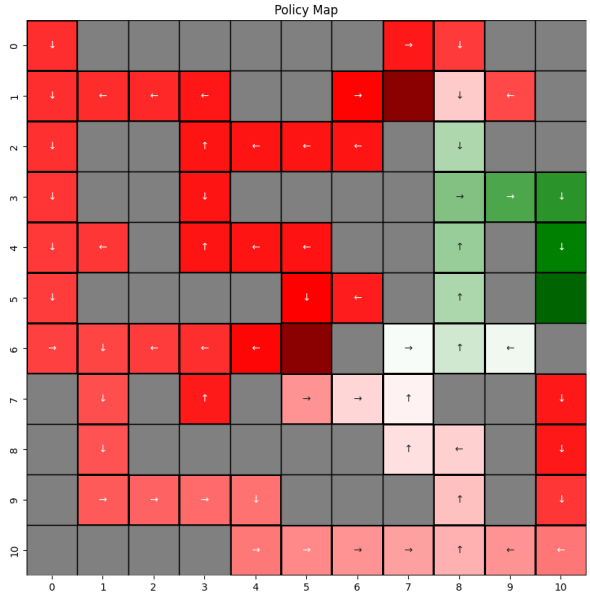
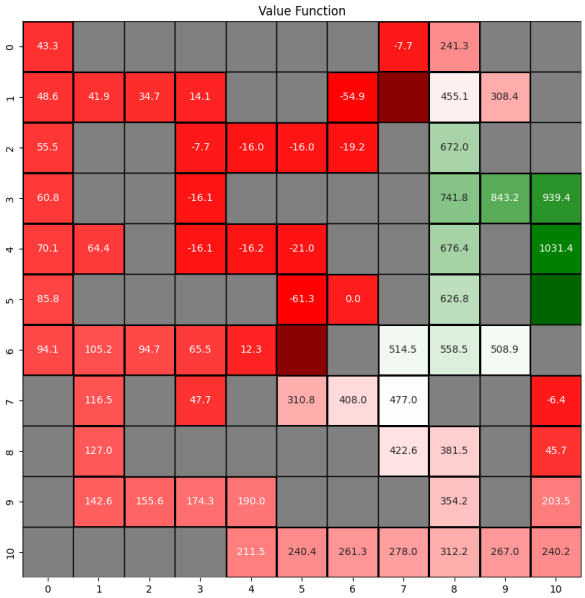


Figure 40: epsilon = 0.2 for 1000 episode

(a) utility values



(b) policies

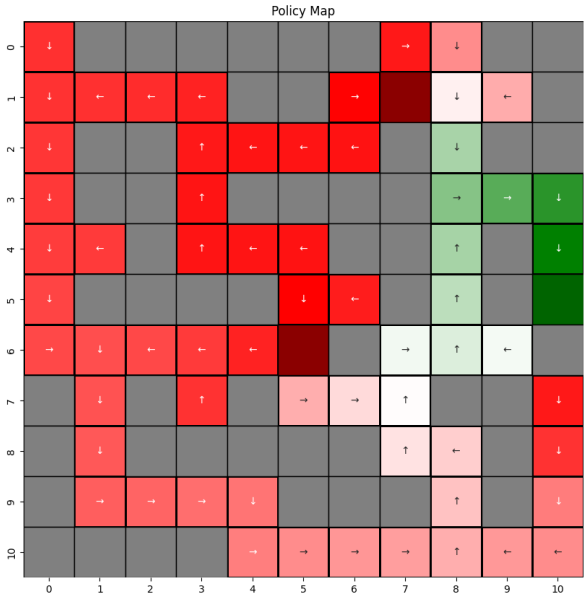
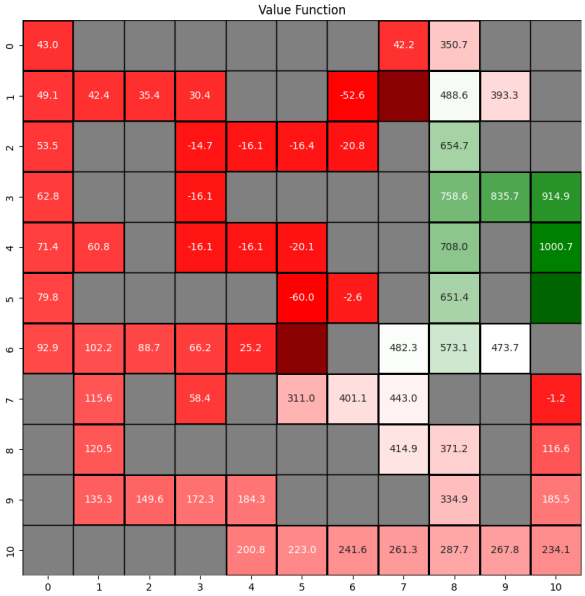


Figure 41: epsilon = 0.2 for 5000 episode

(a) utility values



(b) policies

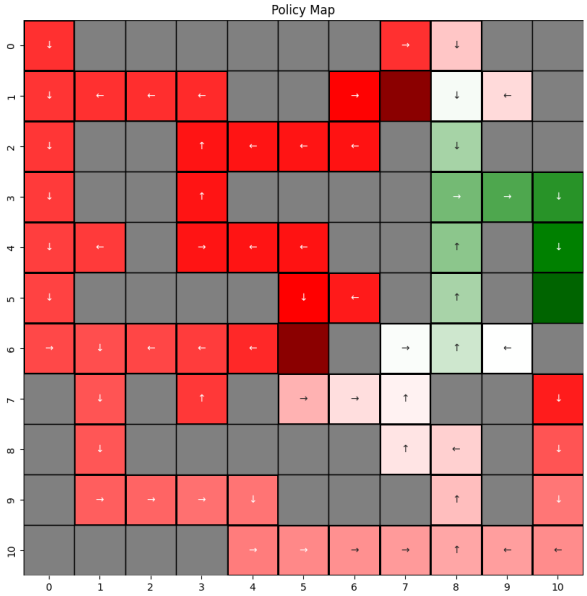
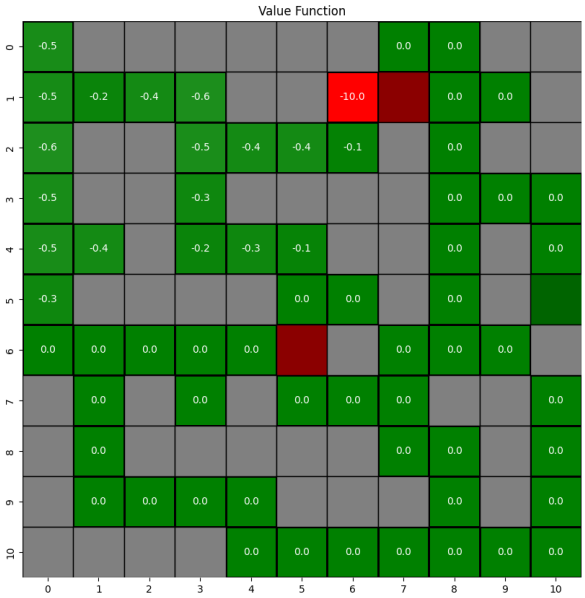


Figure 42: epsilon = 0.2 for 10000 episode

(a) utility values



(b) policies

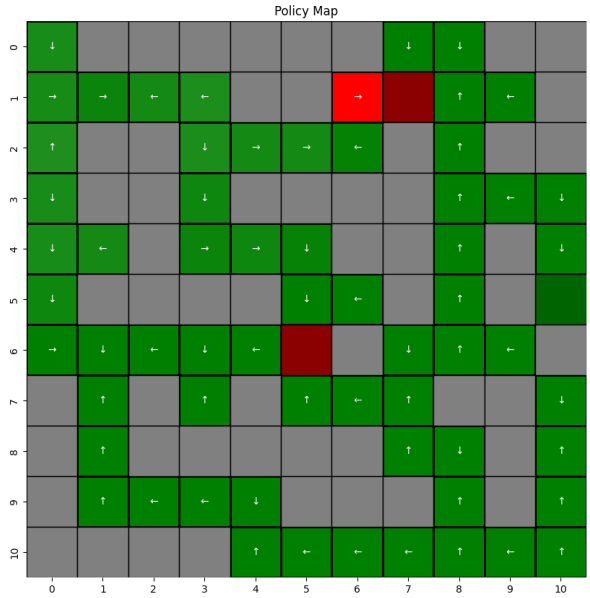
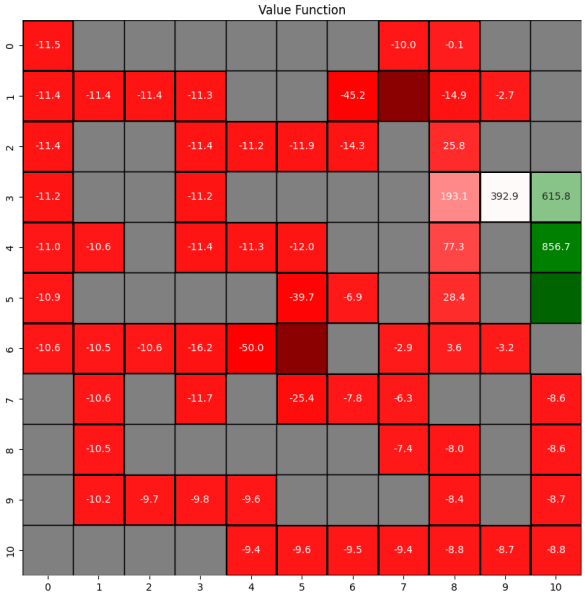


Figure 43: epsilon = 0.5 for 1 episode

(a) utility values



(b) policies

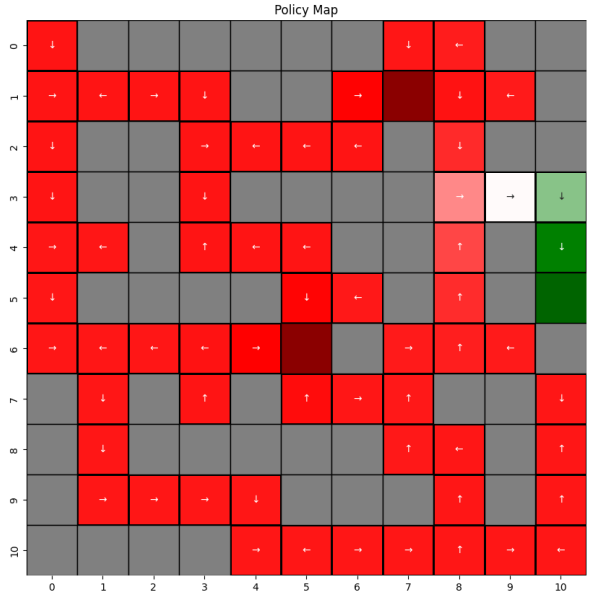


Figure 44: epsilon = 0.5 for 50 episode

(a) utility values (b) policies

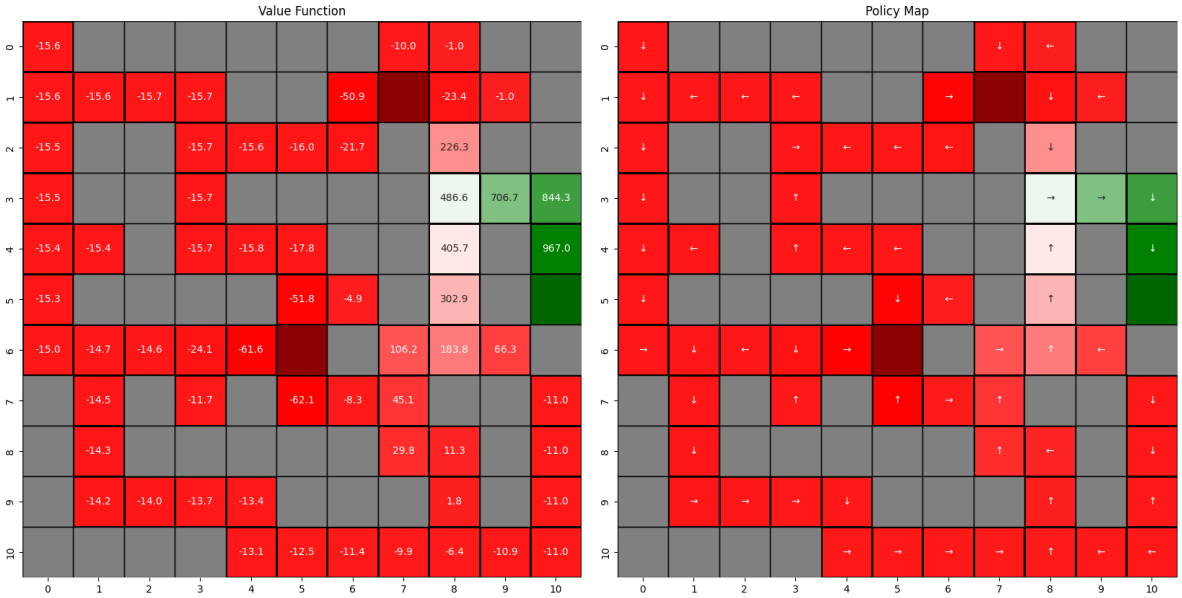


Figure 45: epsilon = 0.5 for 100 episode

(a) utility values (b) policies

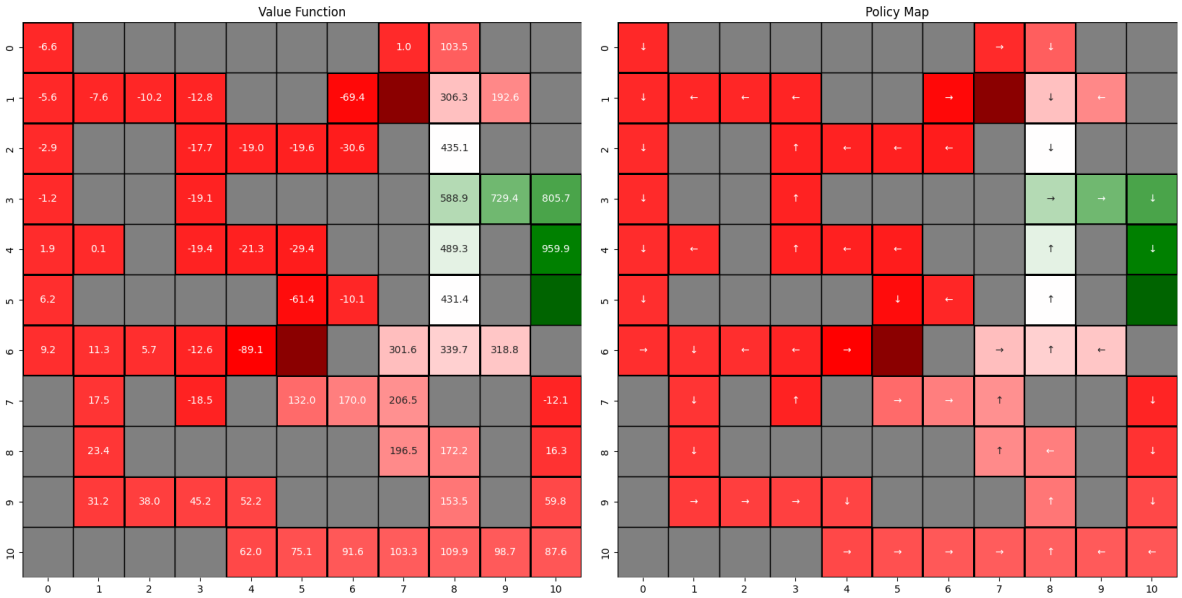


Figure 46: epsilon = 0.5 for 1000 episode

(a) utility values (b) policies

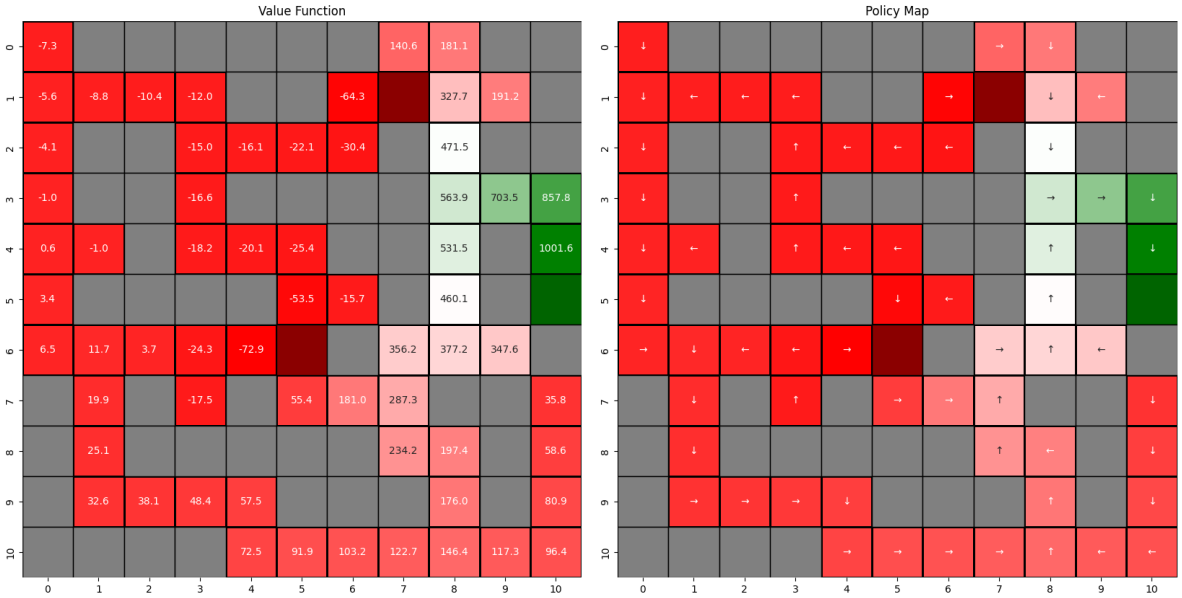


Figure 47: epsilon = 0.5 for 5000 episode

(a) utility values (b) policies

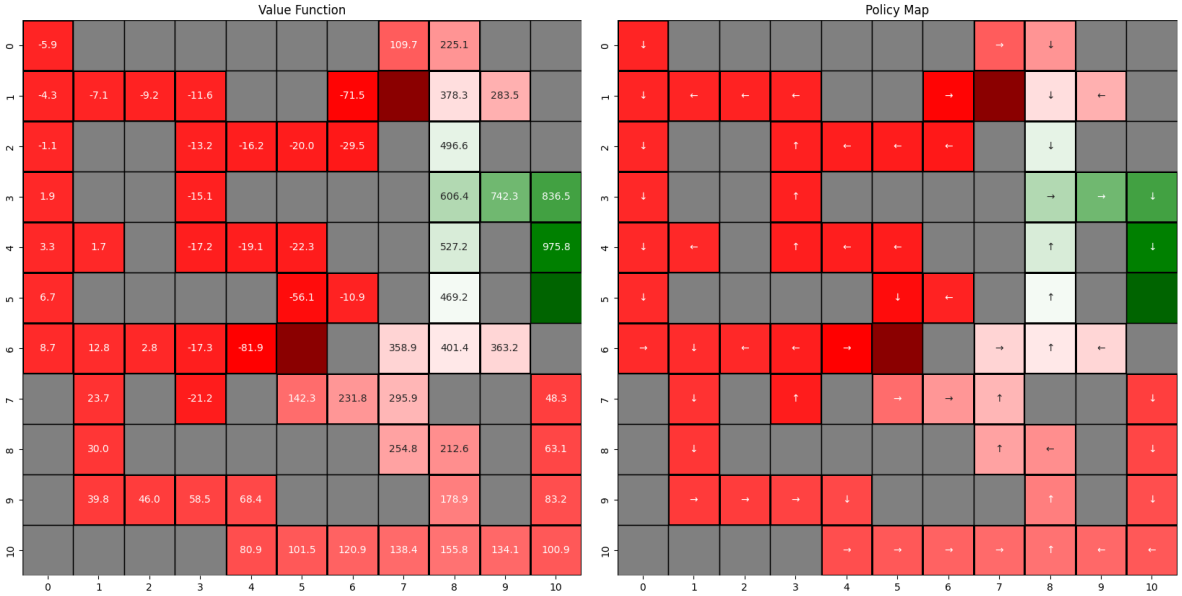
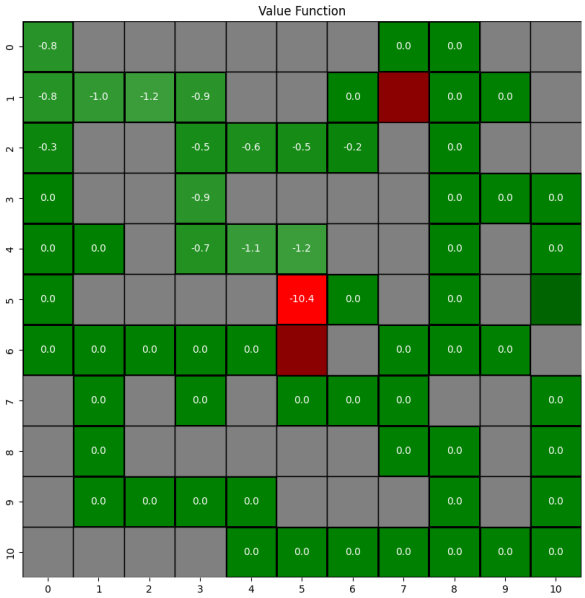


Figure 48: epsilon = 0.5 for 10000 episode

(a) utility values



(b) policies

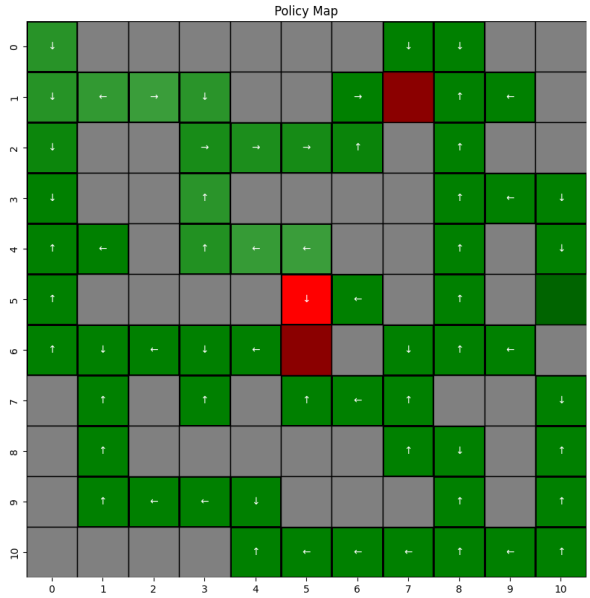
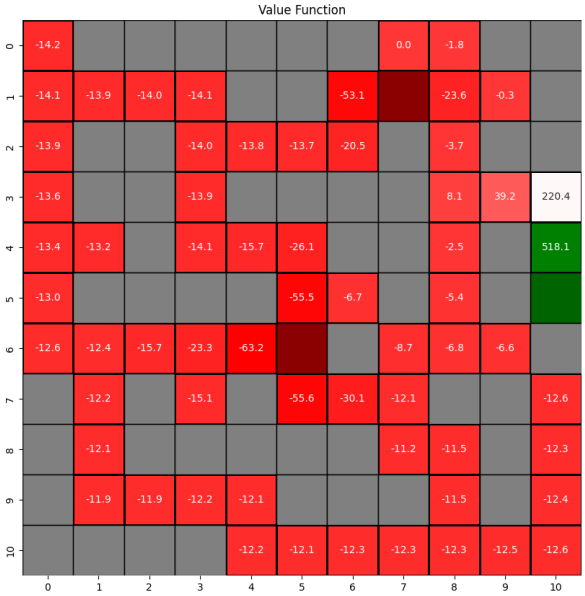


Figure 49: epsilon = 0.8 for 1 episode

(a) utility values



(b) policies

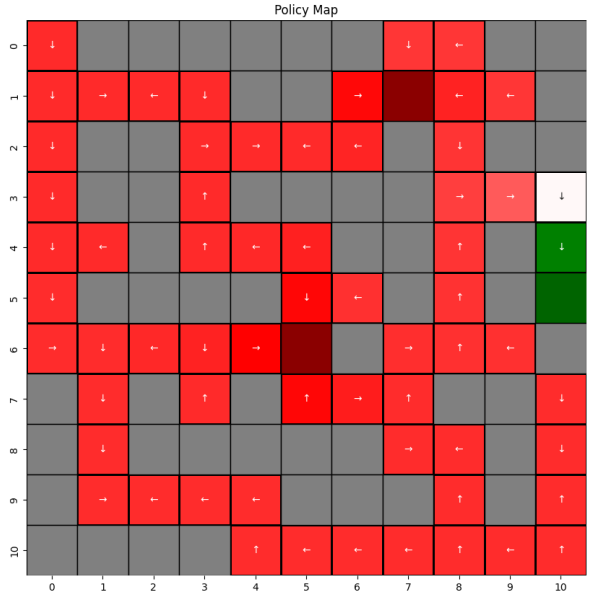
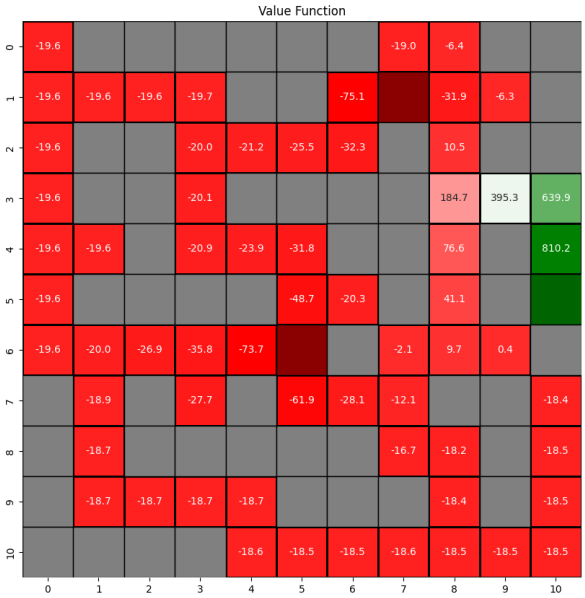


Figure 50: epsilon = 0.8 for 50 episode

(a) utility values



(b) policies

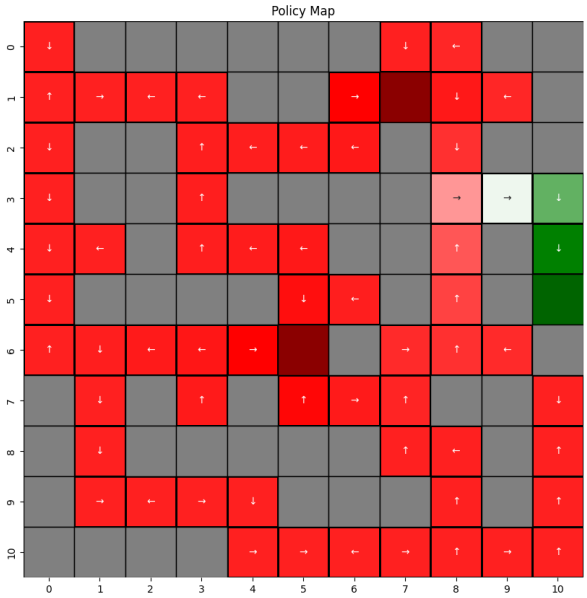
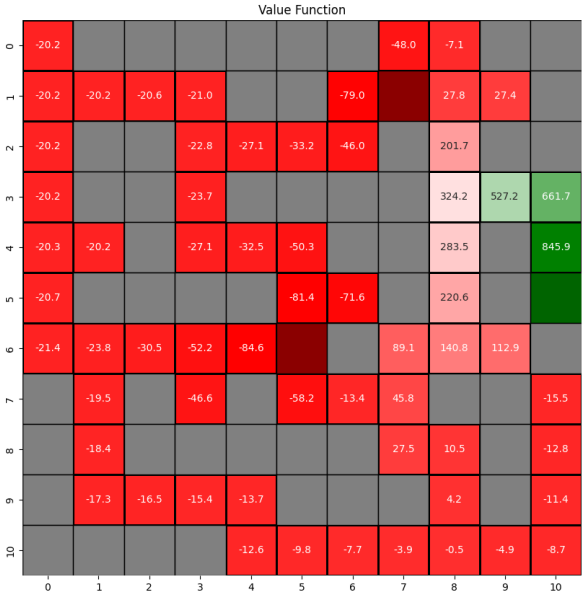


Figure 51: epsilon = 0.8 for 100 episode

(a) utility values



(b) policies

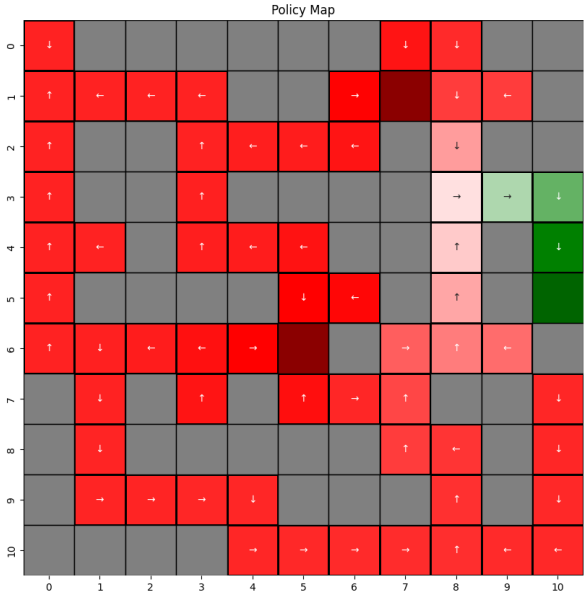
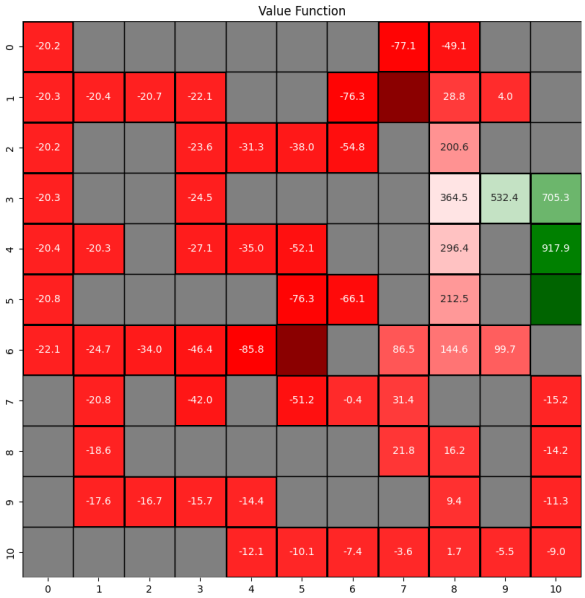


Figure 52: epsilon = 0.8 for 1000 episode

(a) utility values



(b) policies

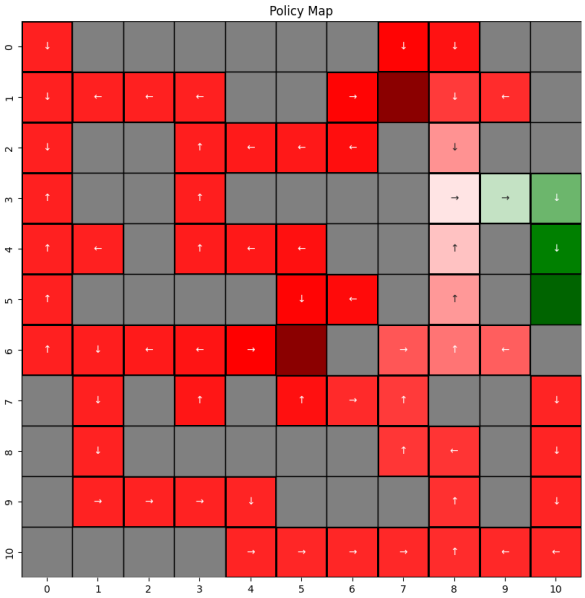
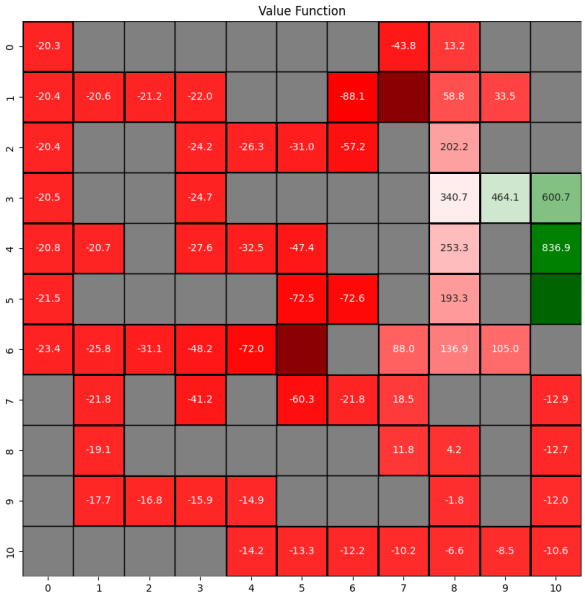


Figure 53: epsilon = 0.8 for 5000 episode

(a) utility values



(b) policies

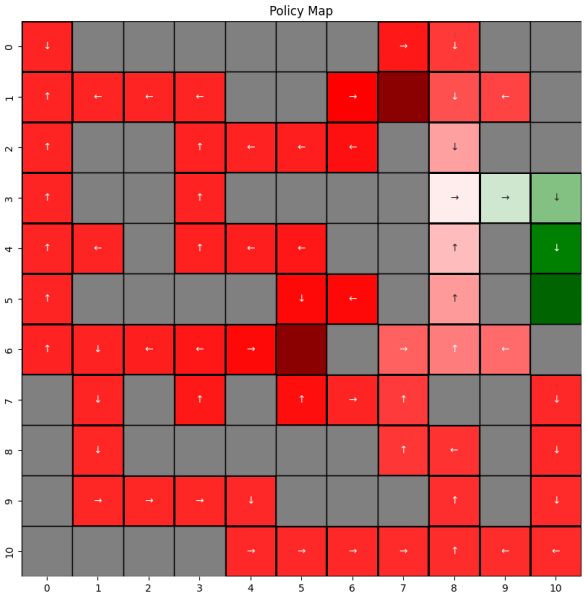
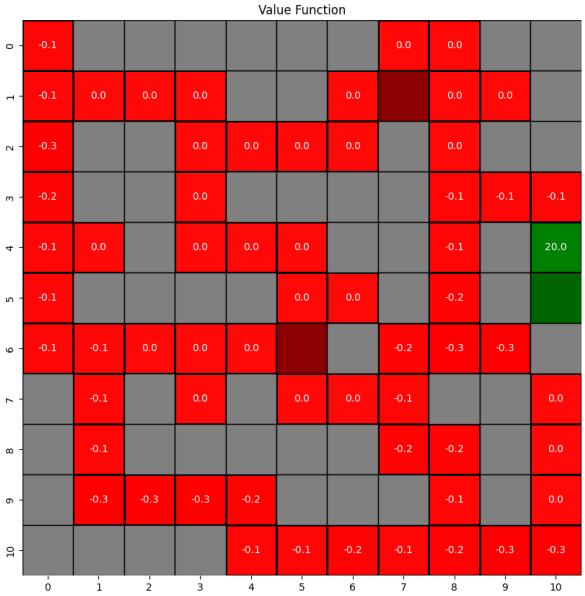


Figure 54: epsilon = 0.8 for 10000 episode

I have obtained successful results for 0 epsilon and 1000, 5000, and 10000 episodes; for 0.2 epsilon this time too, 1000, 5000, and 10000 episodes were successful, and finally, for 0.5 episodes, 100, 1000, 5000, and 10000 episodes were successful. Epsilon is the rate of random choice-making over utility-based choice-making. As can be seen here, very high rates of random choice led to a decrease in agent performance; however, for 0.5 epsilon, the agent showed success after only 100 episodes. This might be interpreted as the benefit of the exploration. This was probably due to the structure of the maze in which traps are closer to the starting point, so higher epsilon might've led to escaping from their effect earlier.

Effect of gamma in TD learning

(a) utility values



(b) policies

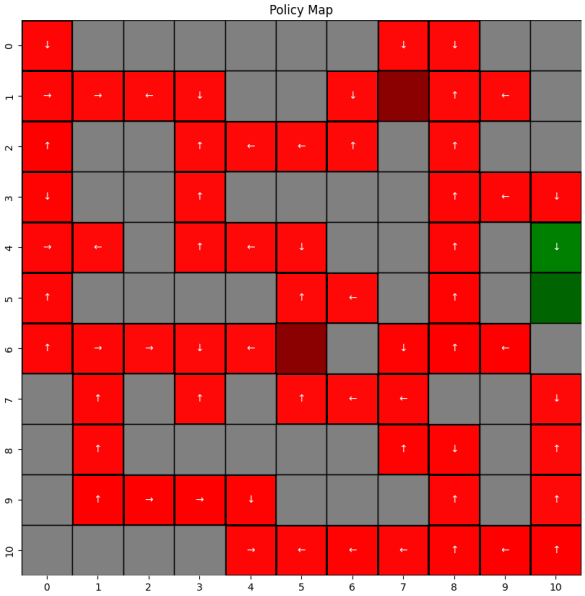
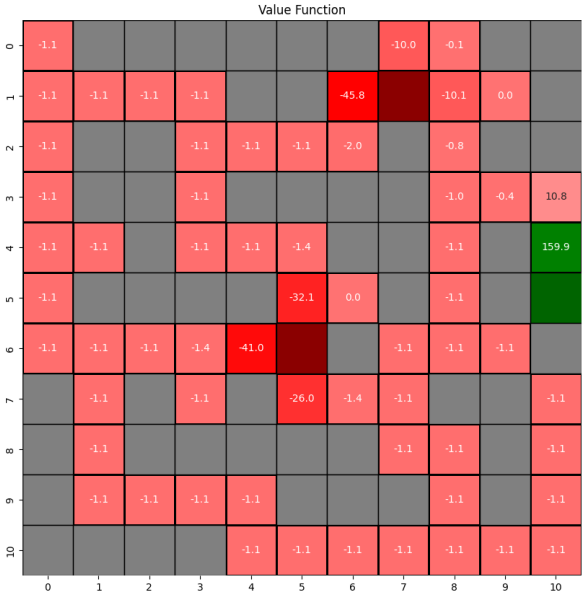


Figure 55: gamma = 0.1 for 1 episode

(a) utility values



(b) policies

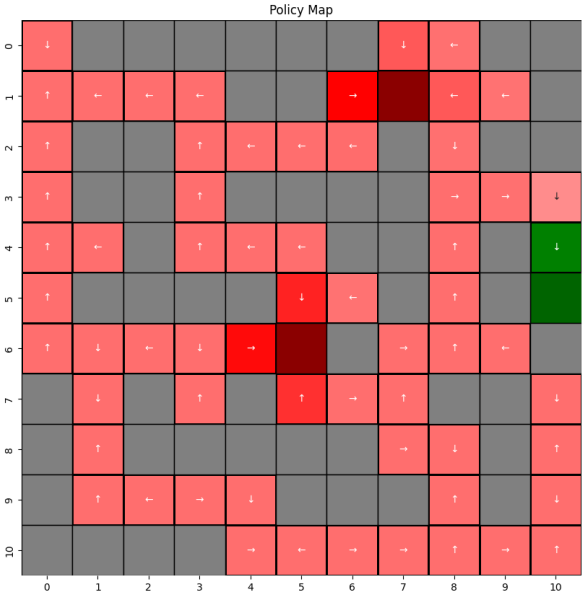


Figure 56: gamma = 0.1 for 50 episode

(a) utility values (b) policies

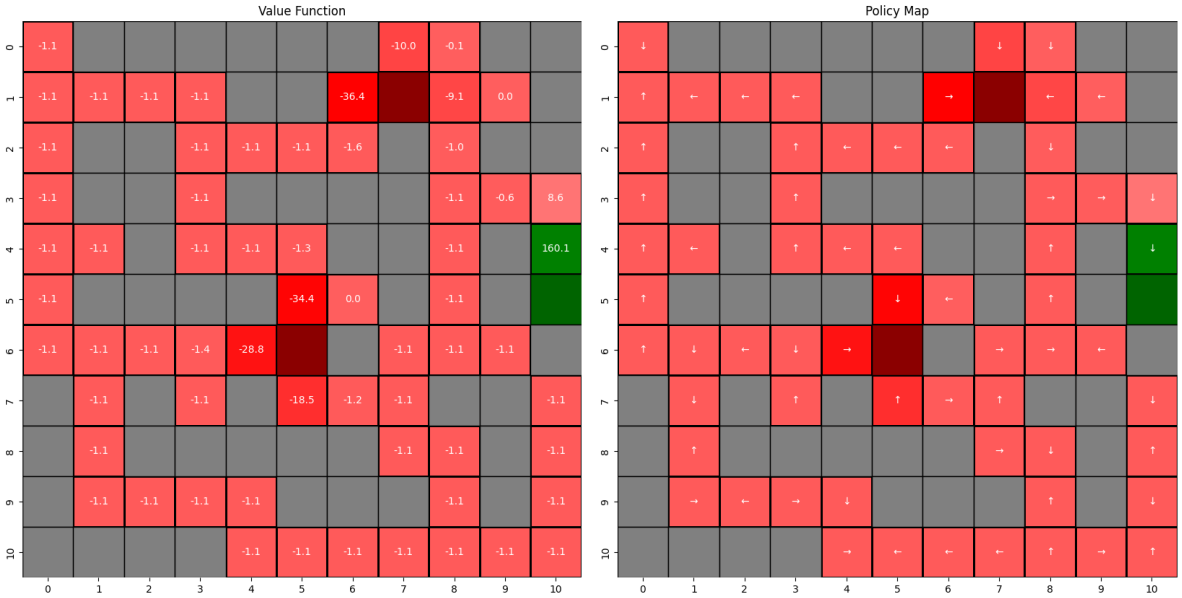


Figure 57: gamma = 0.1 for 100 episode

(a) utility values (b) policies

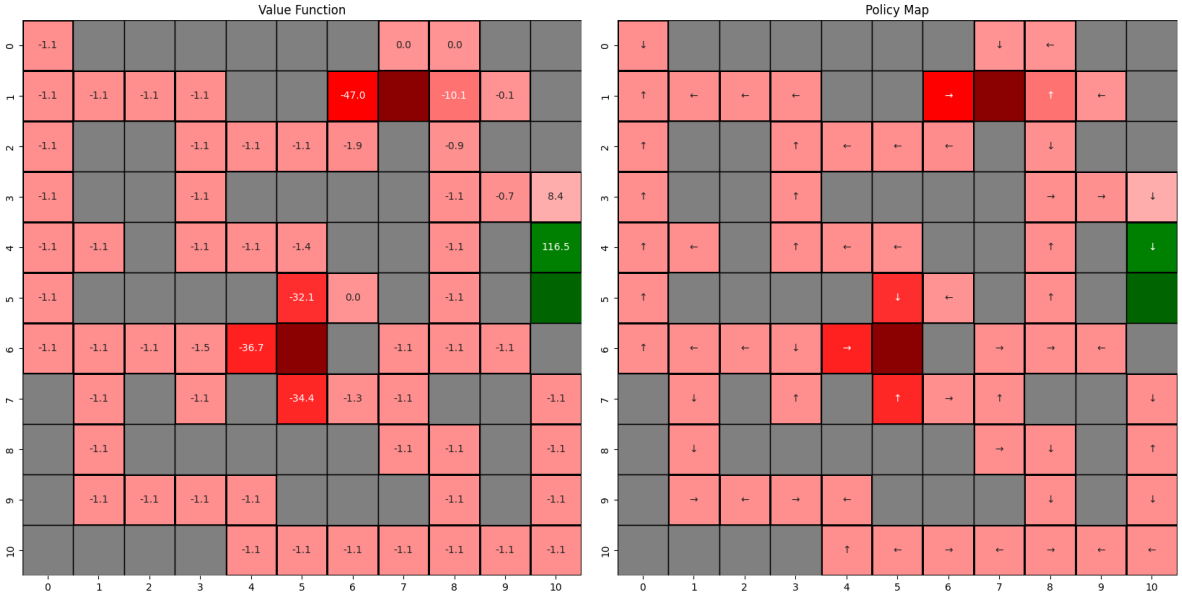
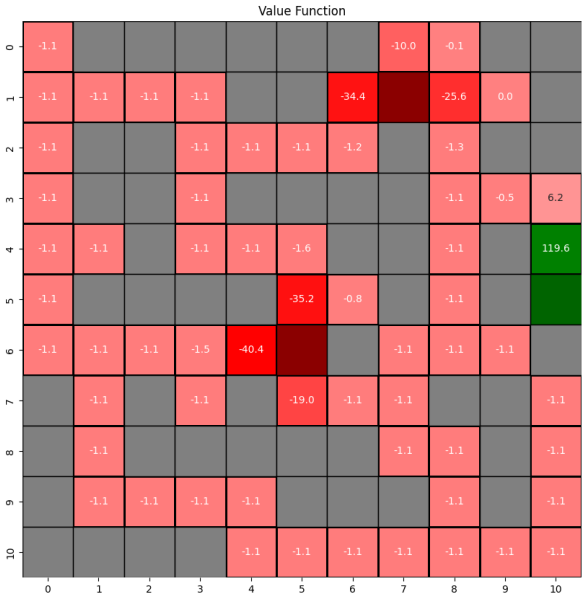


Figure 58: gamma = 0.1 for 1000 episode

(a) utility values



(b) policies

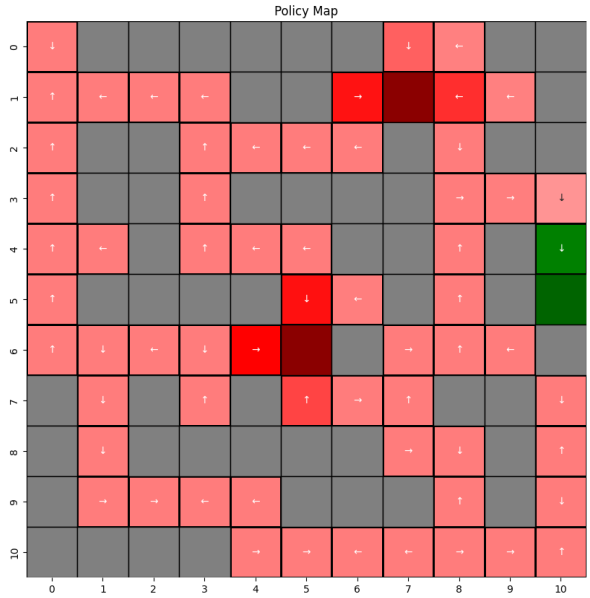
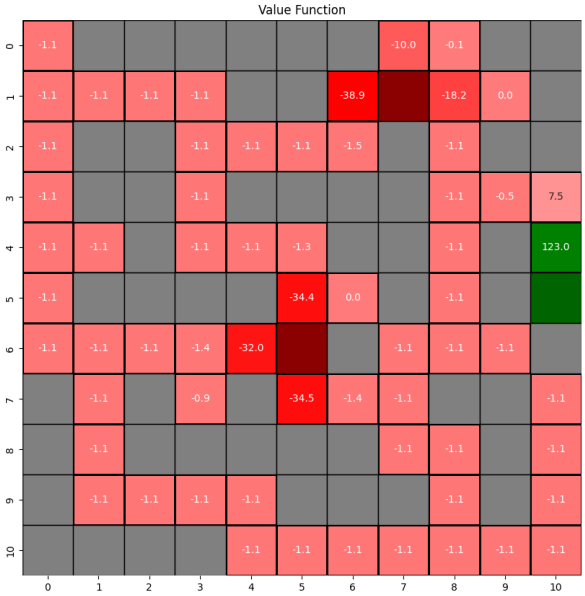


Figure 59: gamma = 0.1 for 5000 episode

(a) utility values



(b) policies

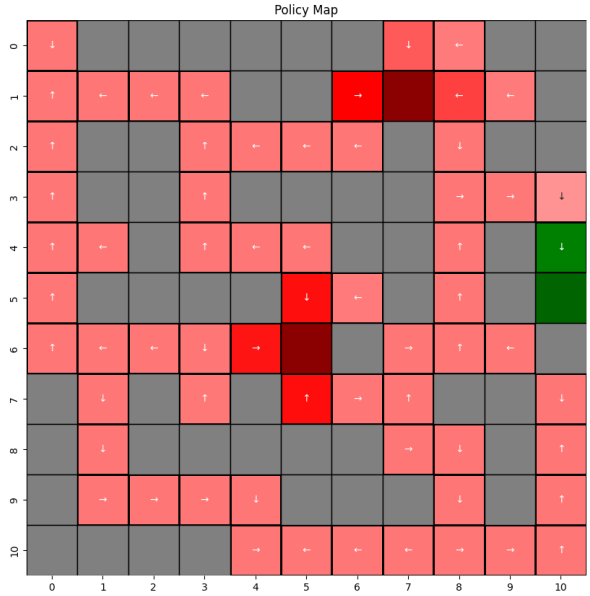


Figure 60: gamma = 0.1 for 10000 episode

(a) utility values (b) policies

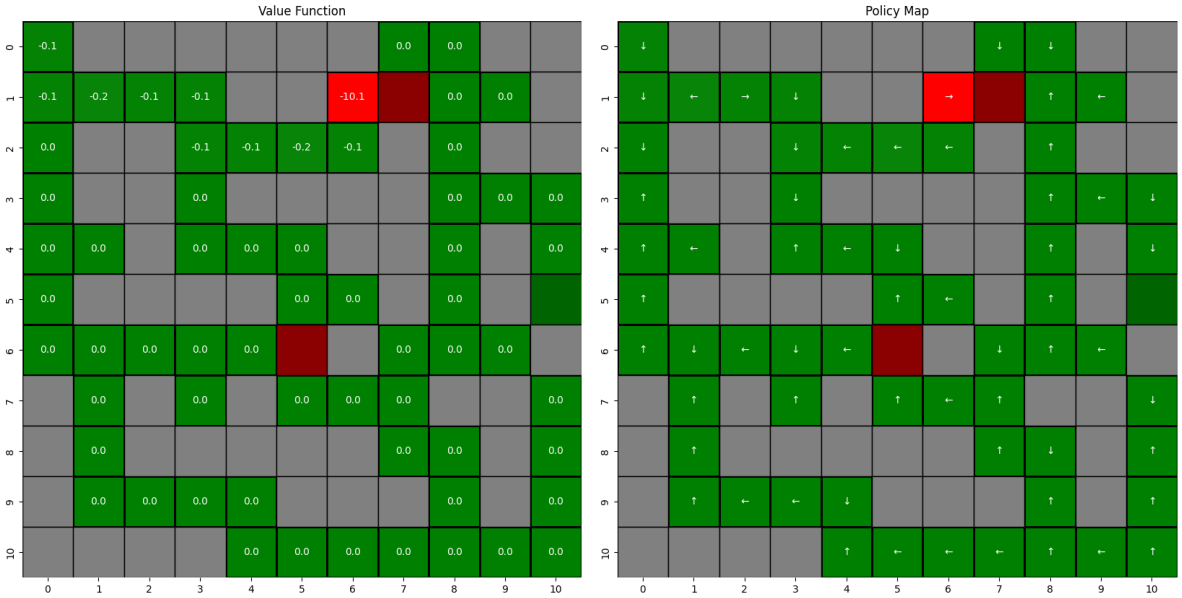


Figure 61: gamma = 0.25 for 1 episode

(a) utility values (b) policies

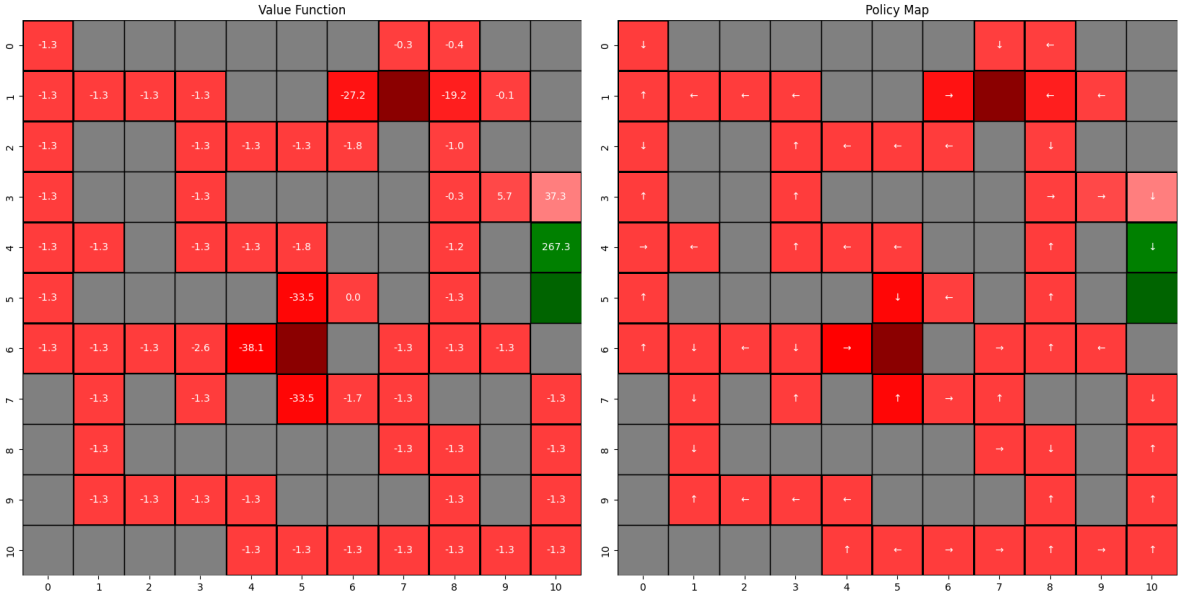
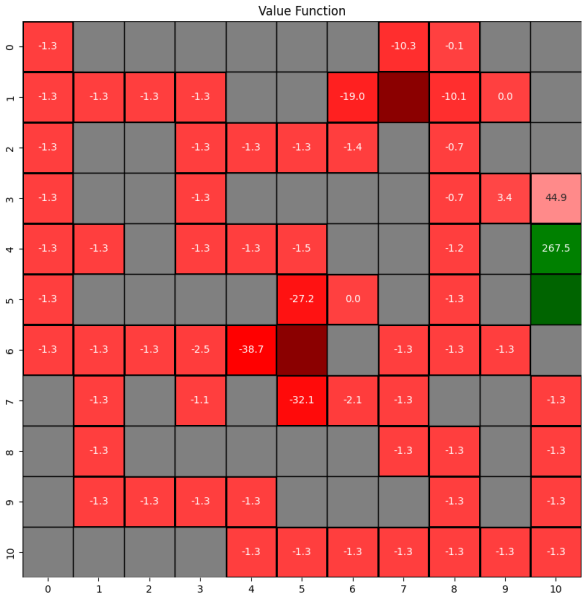


Figure 62: gamma = 0.25 for 50 episode

(a) utility values



(b) policies

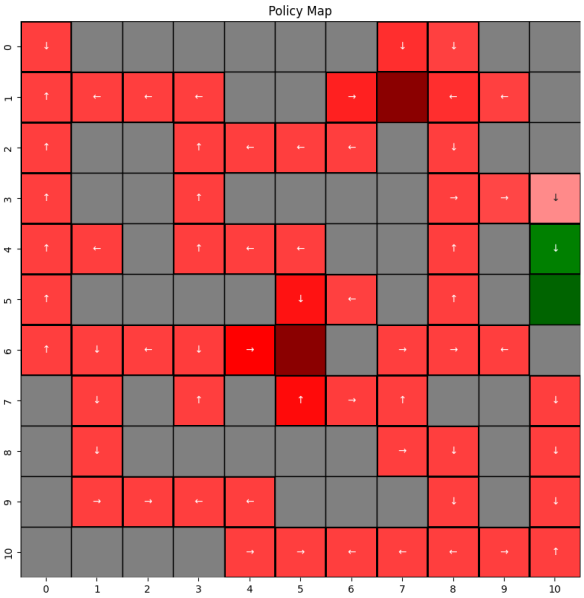
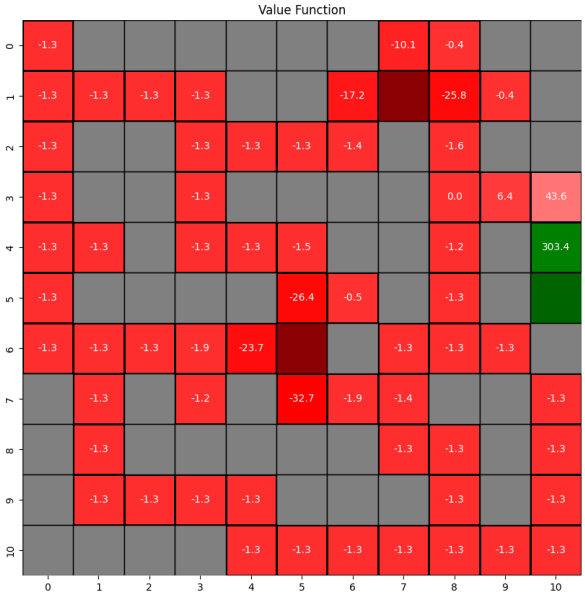


Figure 63: gamma = 0.25 for 100 episode

(a) utility values



(b) policies

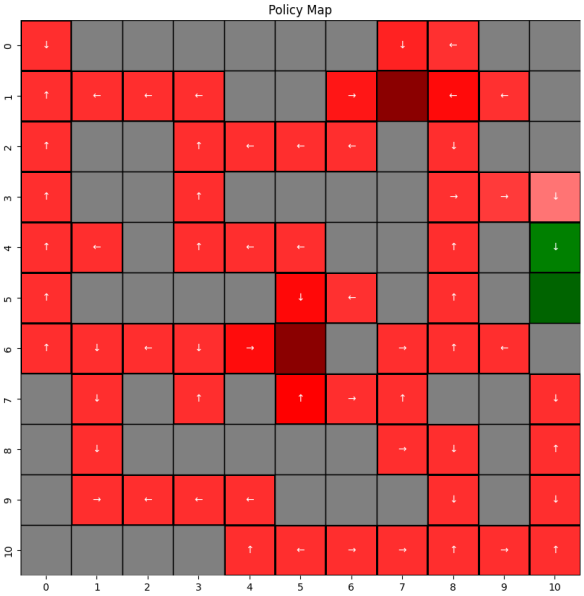
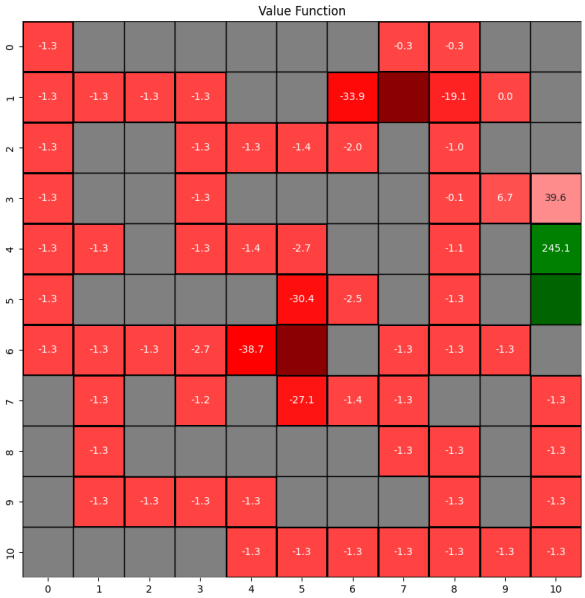


Figure 64: gamma = 0.25 for 1000 episode

(a) utility values



(b) policies

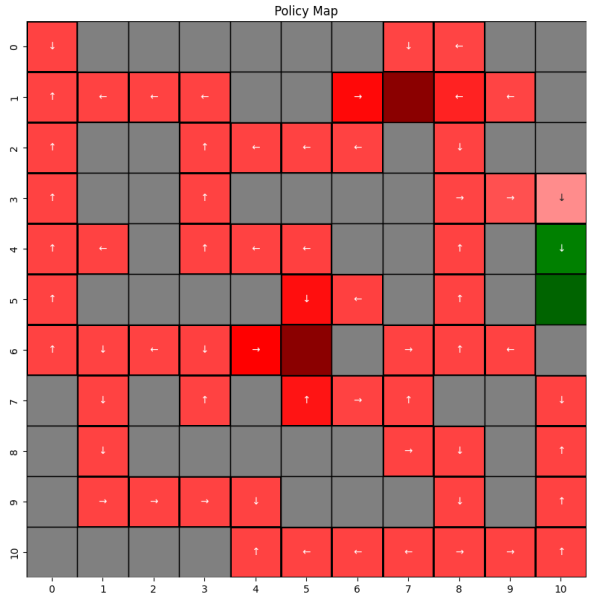
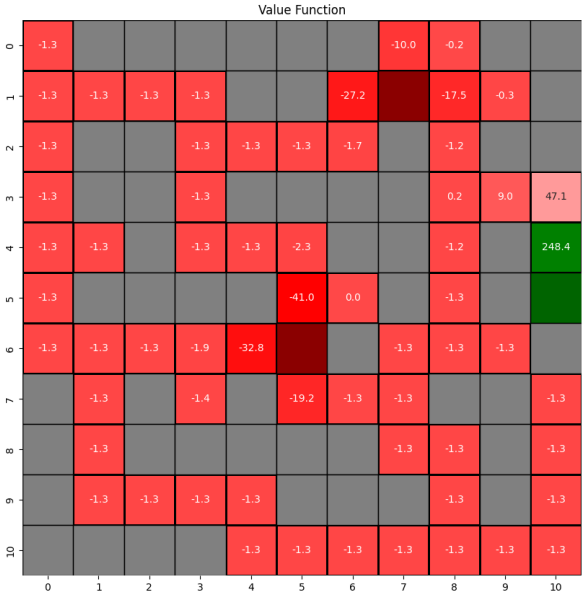


Figure 65: gamma = 0.25 for 5000 episode

(a) utility values



(b) policies

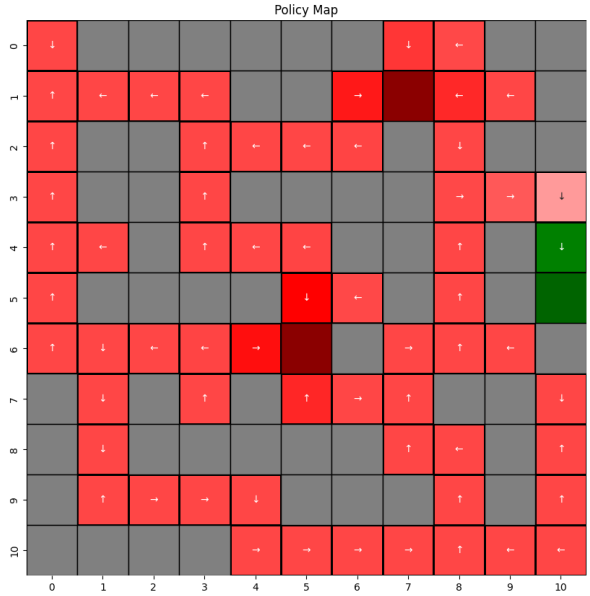


Figure 66: gamma = 0.25 for 10000 episode

(a) utility values (b) policies

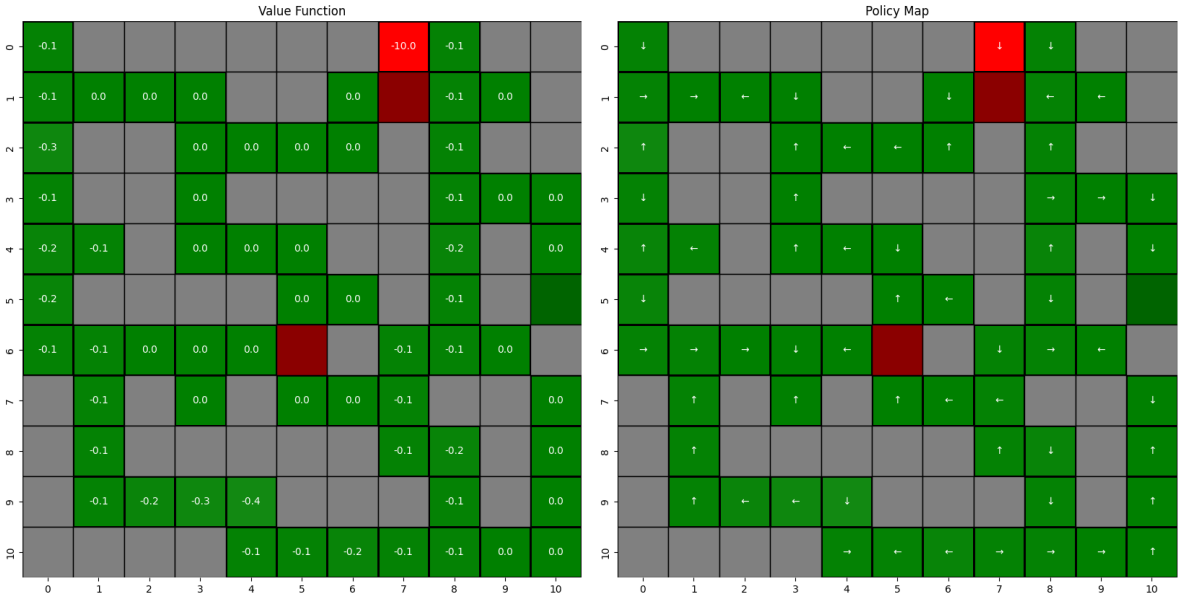


Figure 67: gamma = 0.5 for 1 episode

(a) utility values (b) policies

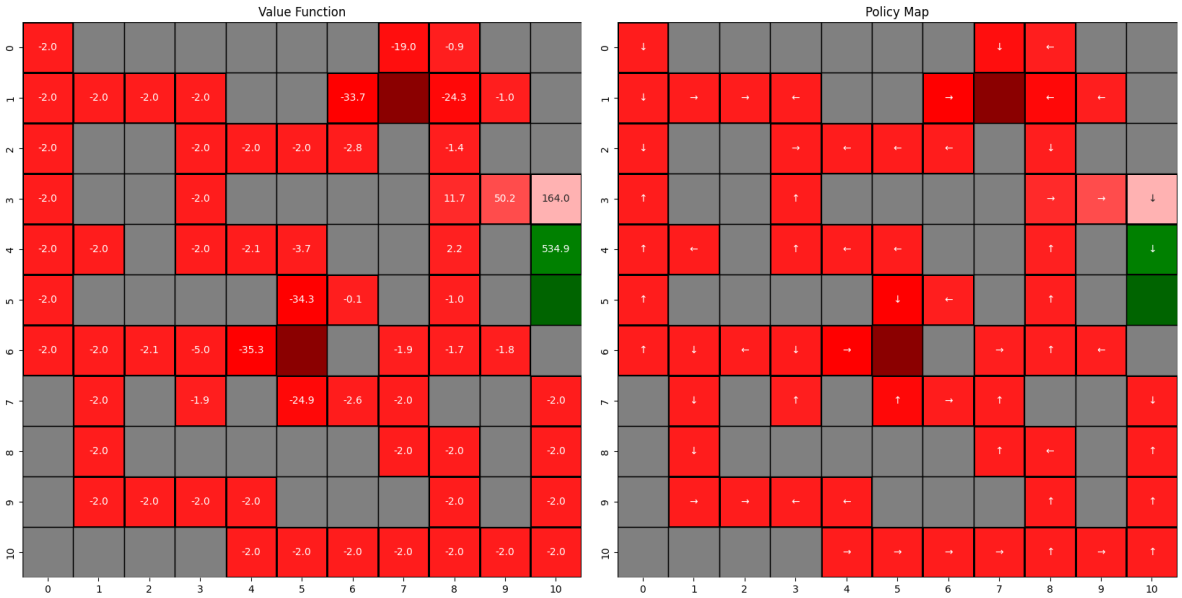
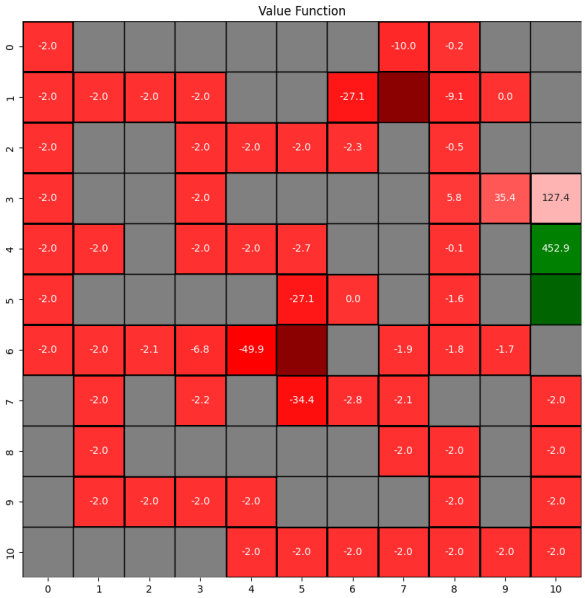


Figure 68: gamma = 0.5 for 50 episode

(a) utility values



(b) policies

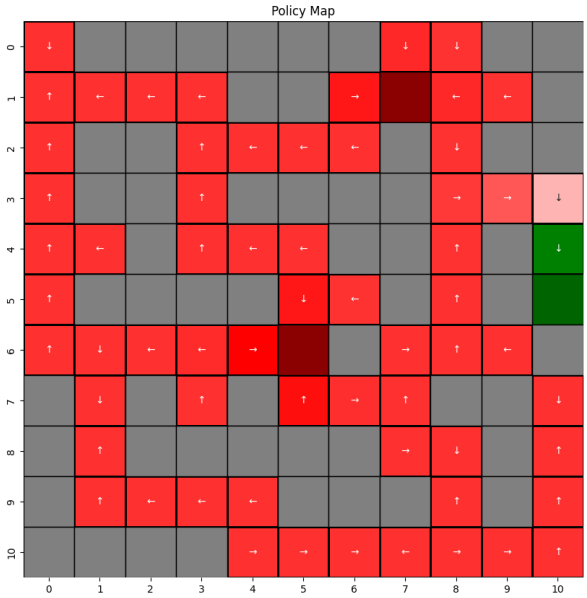
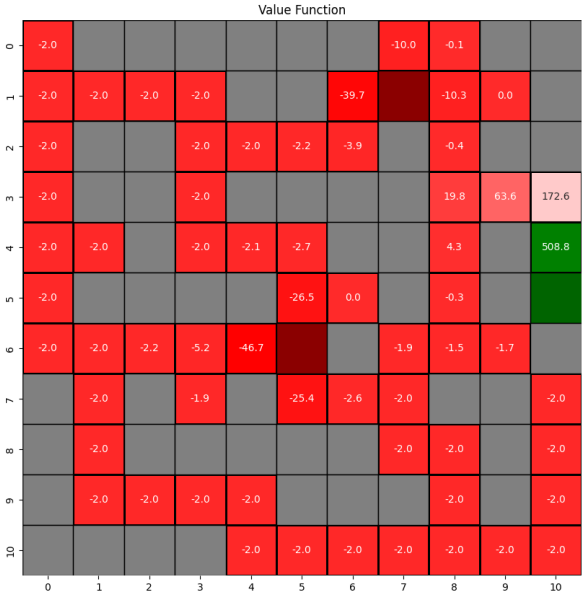


Figure 69: gamma = 0.5 for 100 episode

(a) utility values



(b) policies

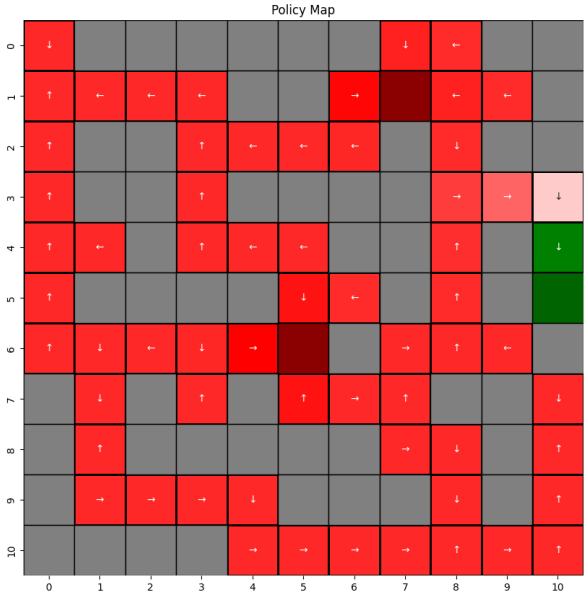
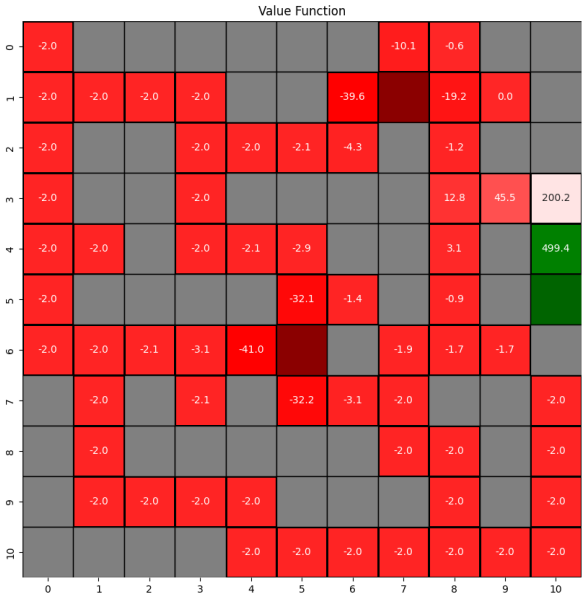


Figure 70: gamma = 0.5 for 1000 episode

(a) utility values



(b) policies

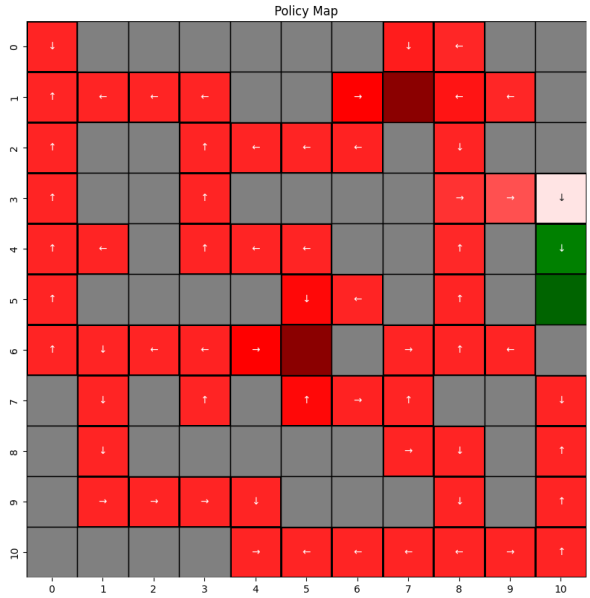
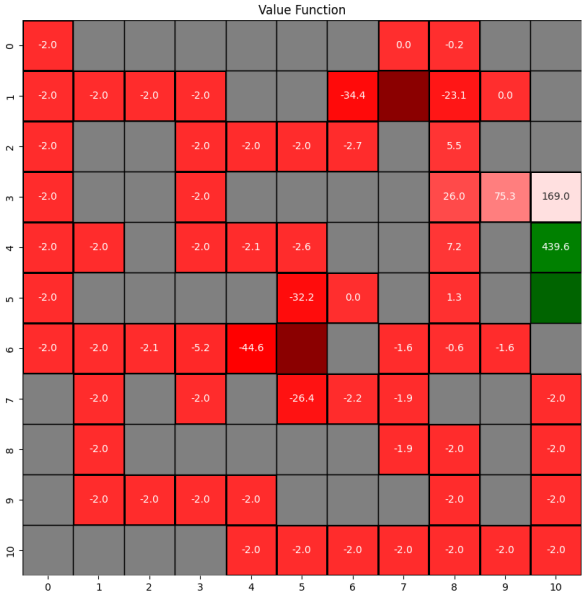


Figure 71: gamma = 0.5 for 5000 episode

(a) utility values



(b) policies

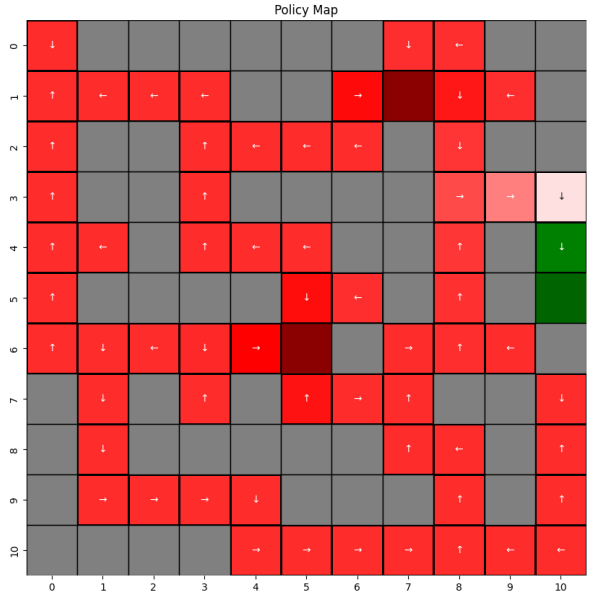
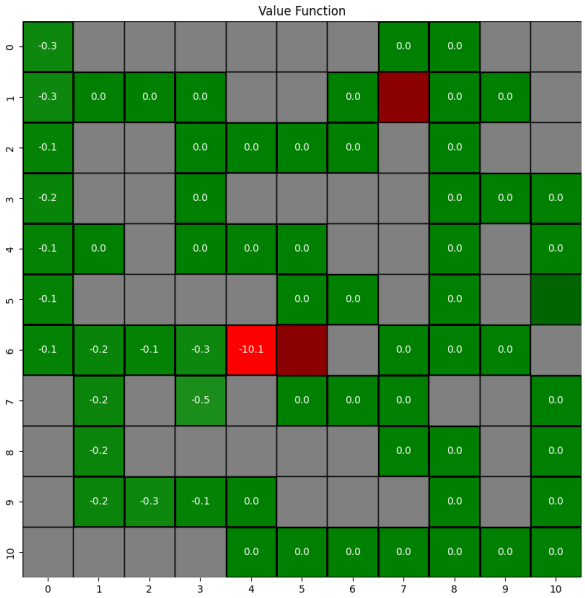


Figure 72: gamma = 0.5 for 10000 episode

(a) utility values



(b) policies

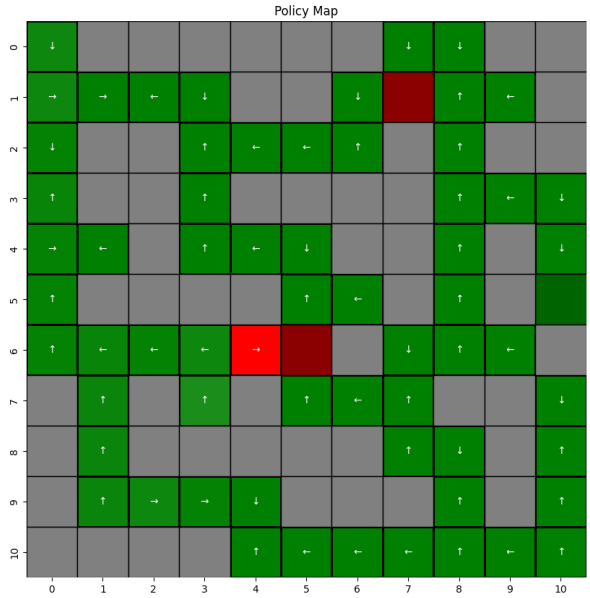
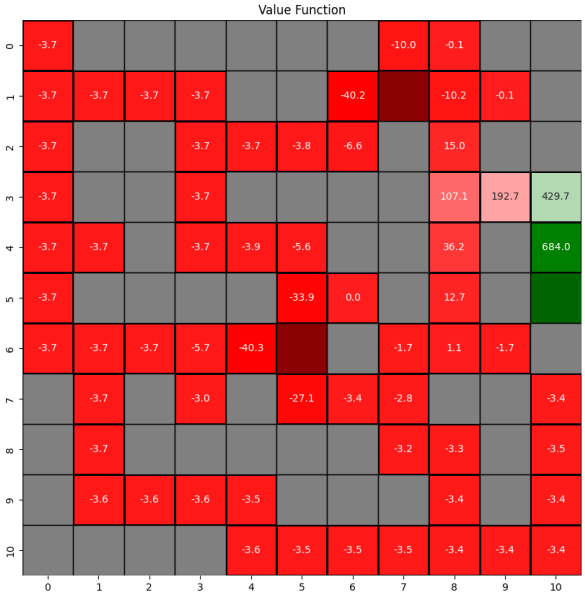


Figure 73: gamma = 0.75 for 1 episode

(a) utility values



(b) policies

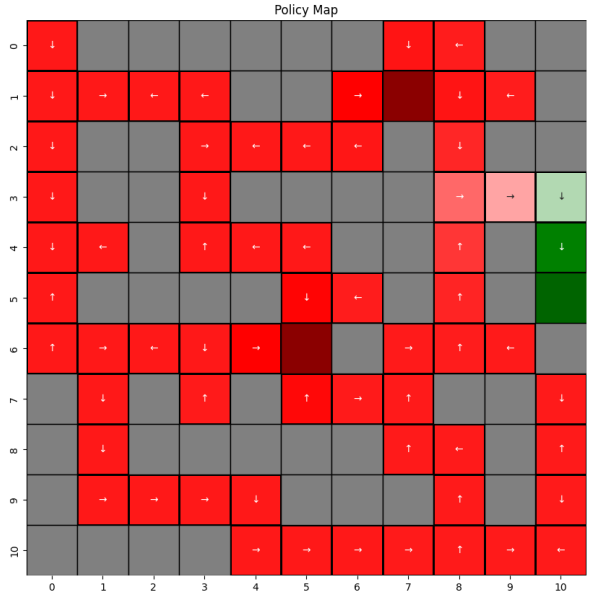
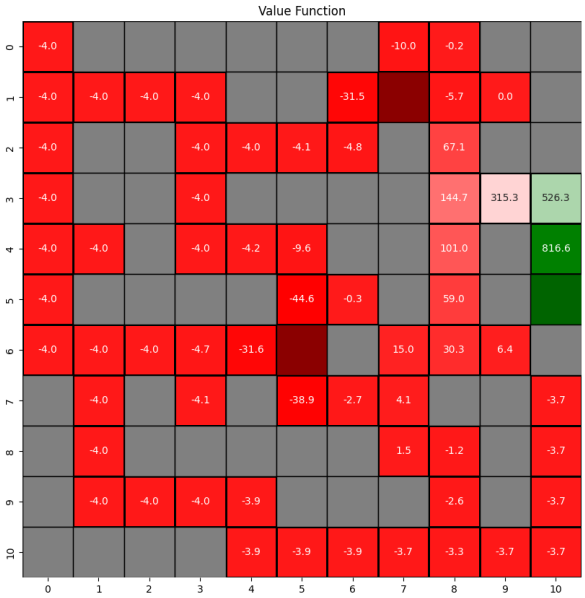


Figure 74: gamma = 0.75 for 50 episode

(a) utility values



(b) policies

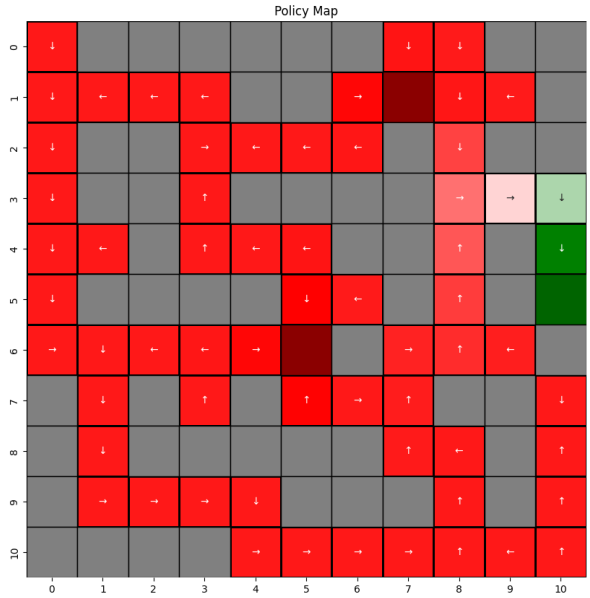
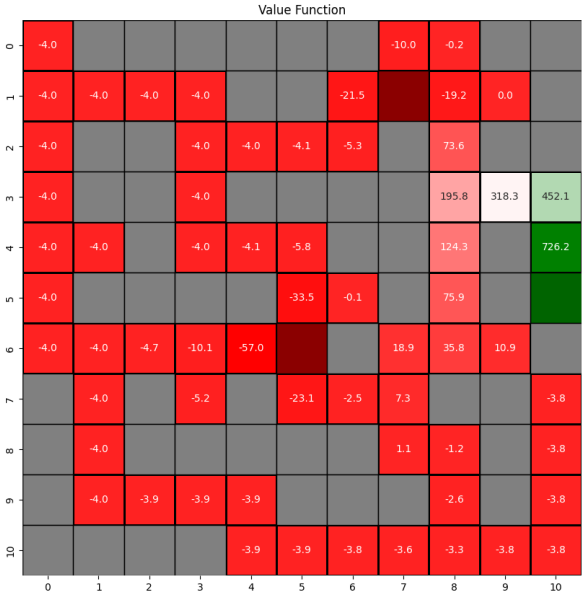


Figure 75: gamma = 0.75 for 100 episode

(a) utility values



(b) policies

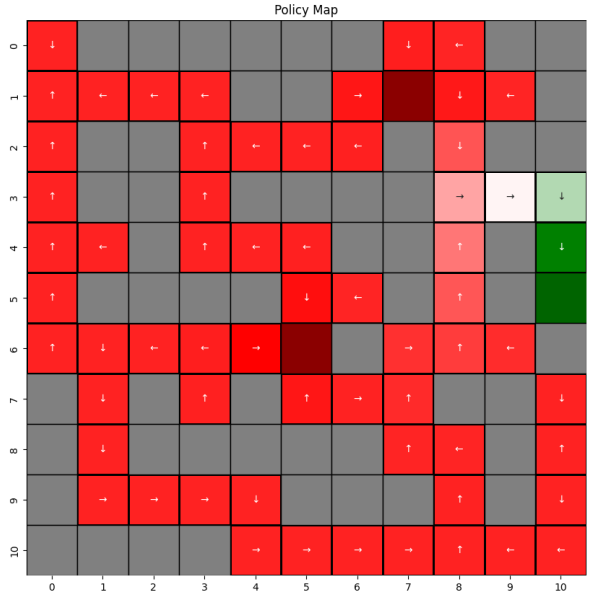
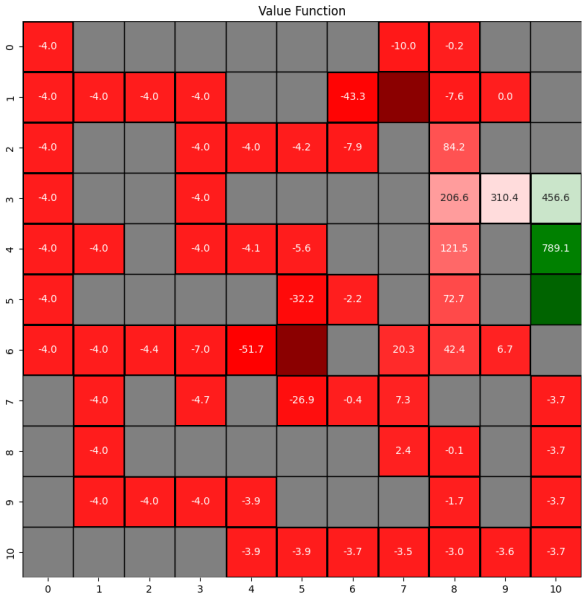


Figure 76: gamma = 0.75 for 1000 episode

(a) utility values



(b) policies

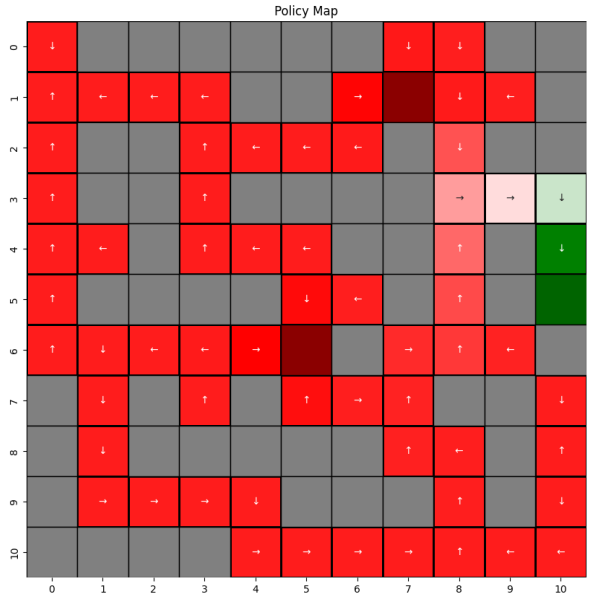
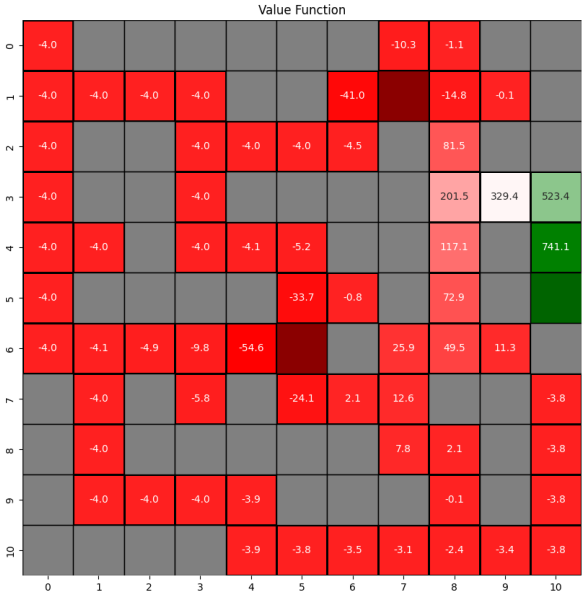


Figure 77: gamma = 0.75 for 5000 episode

(a) utility values



(b) policies

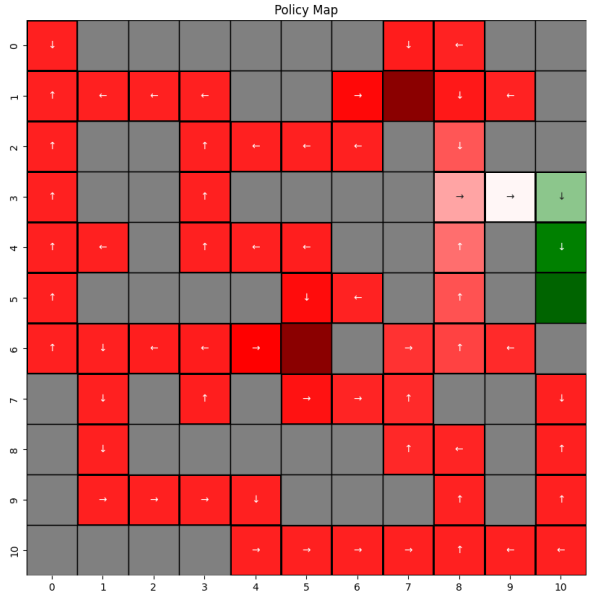
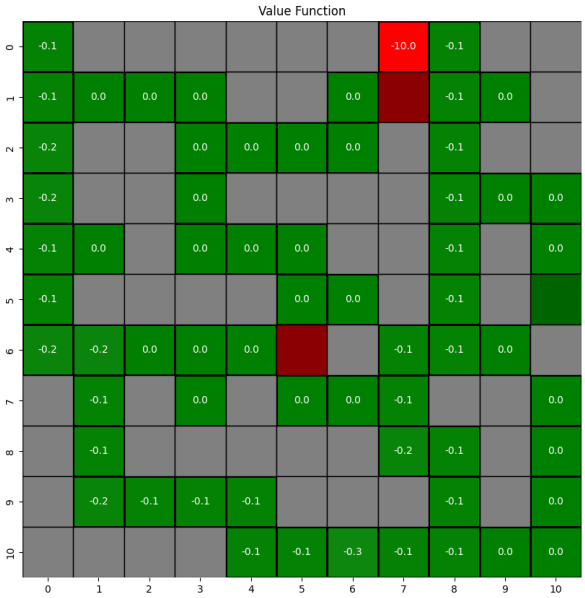


Figure 78: gamma = 0.75 for 10000 episode

(a) utility values



(b) policies

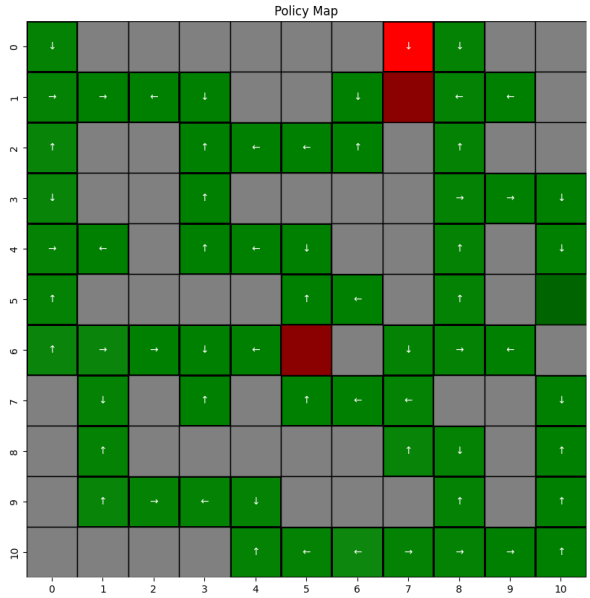
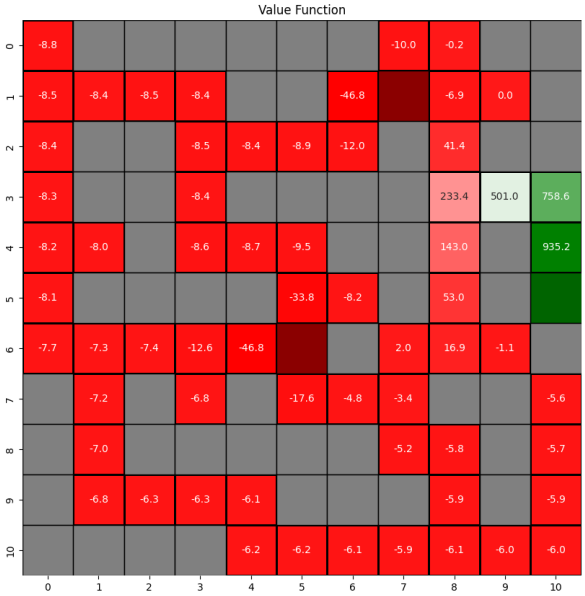


Figure 79: gamma = 1 for 1 episode

(a) utility values



(b) policies

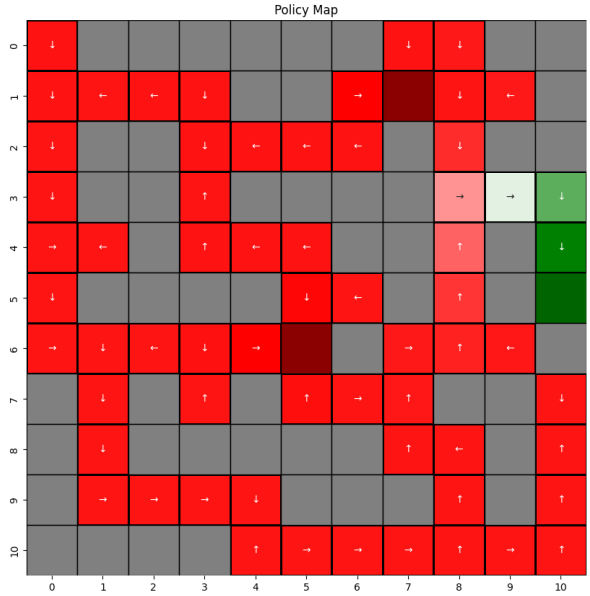
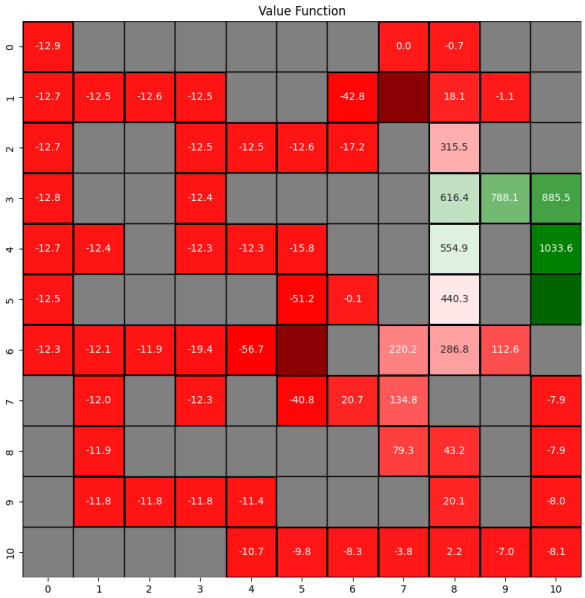


Figure 80: gamma = 1 for 50 episode

(a) utility values



(b) policies

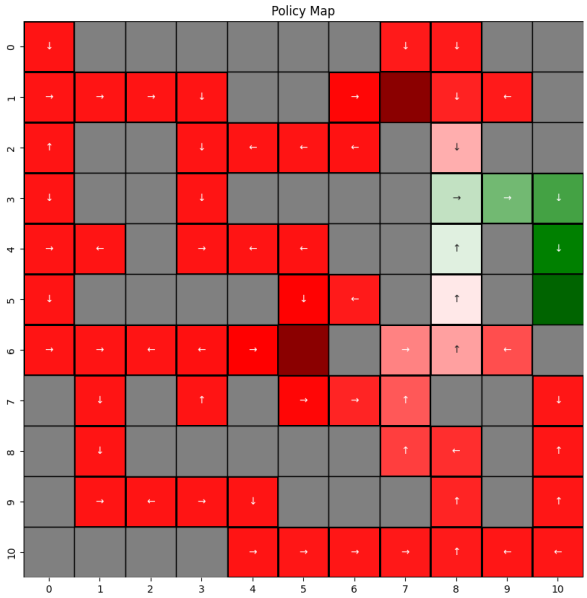
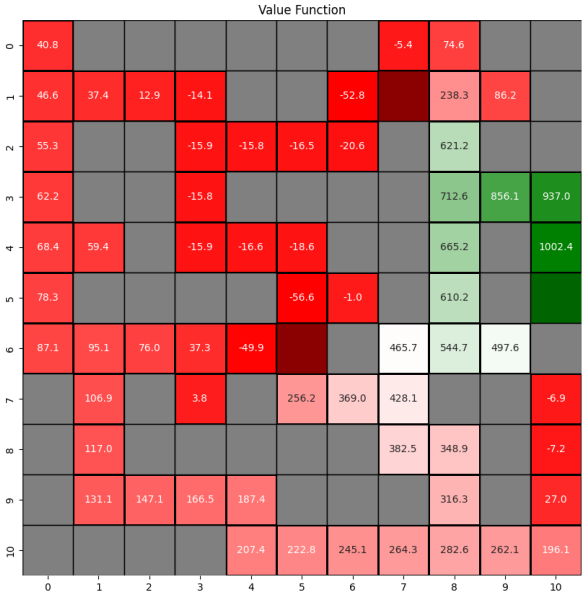


Figure 81: gamma = 1 for 100 episode

(a) utility values



(b) policies

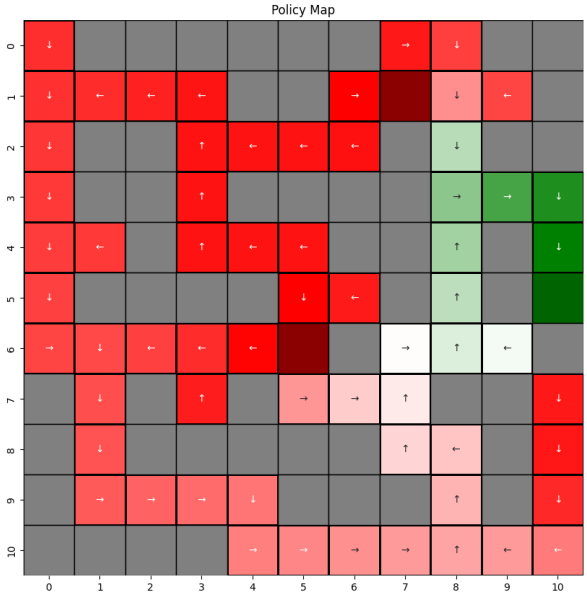


Figure 82: gamma = 1 for 1000 episode

(a) utility values (b) policies

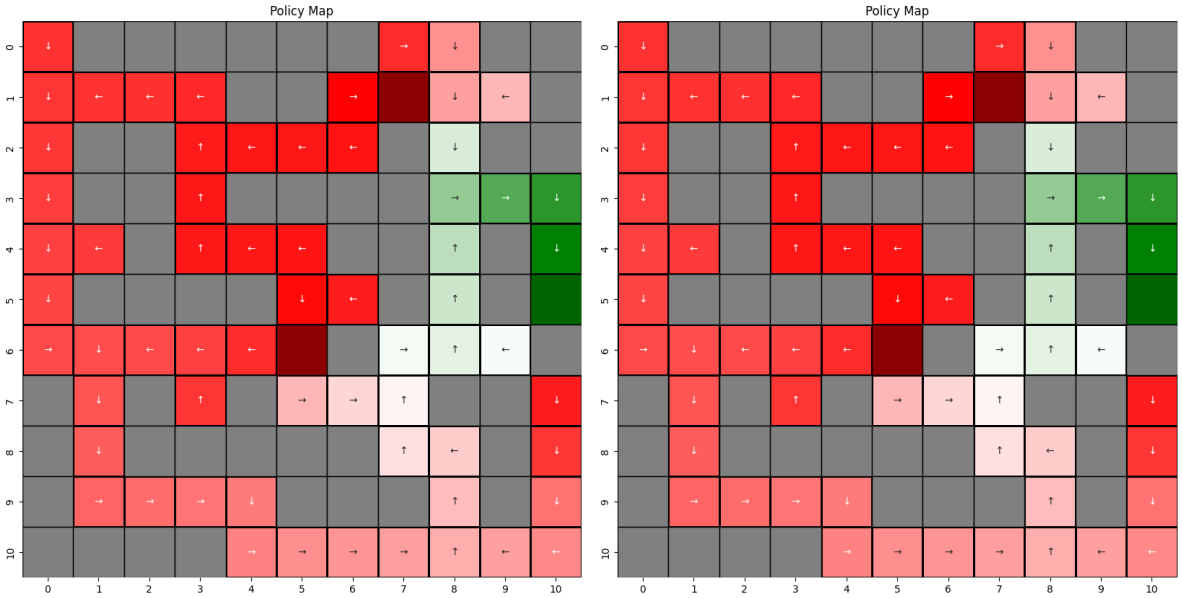


Figure 83: gamma = 1 for 5000 episode

(a) utility values (b) policies

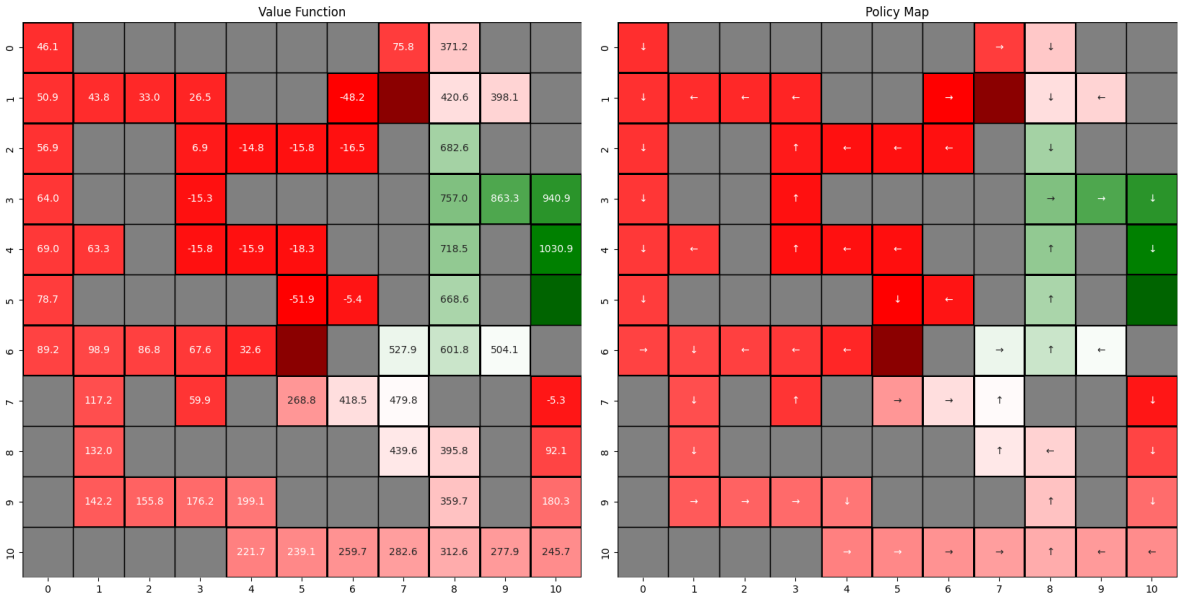
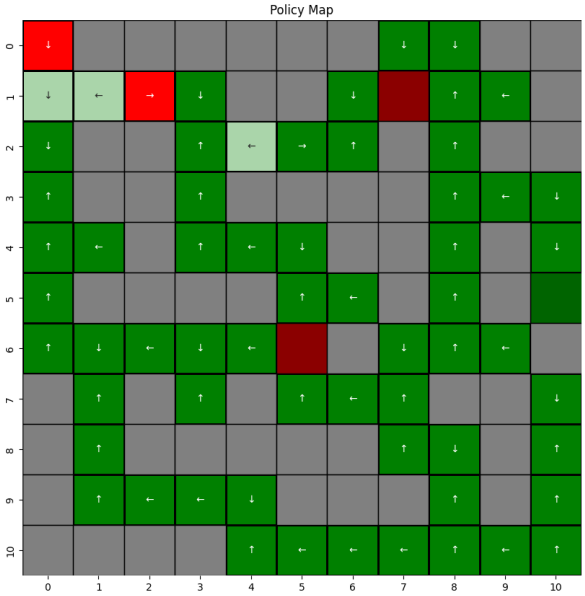


Figure 84: gamma = 0.95 for 10000 episode

For the differing values of gamma, my agent showed success in 0.75 gamma with 100 episodes and 0.95 gamma with 100, 1000, 5000, and 10000 episodes. gamma value increases the effect of the difference between the utility values of changed states. low values of gamma led to fast convergence to small absolute values of utilities, which led to the agent being stuck in the beginning states. plus higher gamma values led to an increase in the effect of the utility of traps and goals, which overrides the detrimental convergence that can hinder the agent.

Effect of alpha in Q learning

(a) utility values



(b) policies

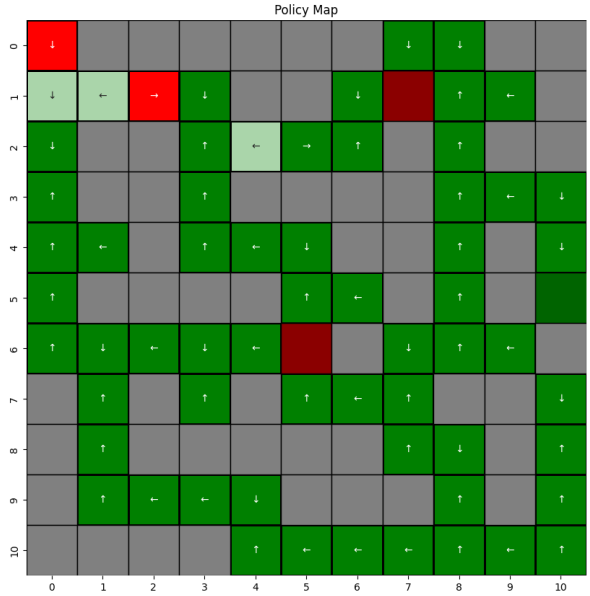
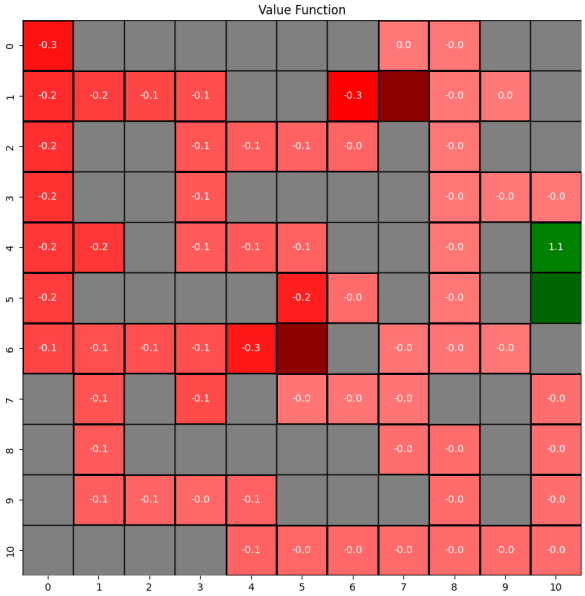


Figure 85: alpha = 0.001 for 1 episode

(a) utility values



(b) policies

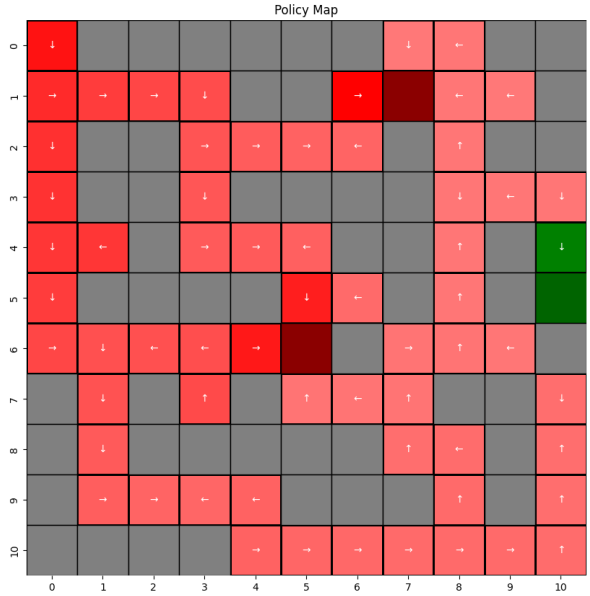


Figure 86: alpha = 0.001 for 50 episode

(a) utility values (b) policies

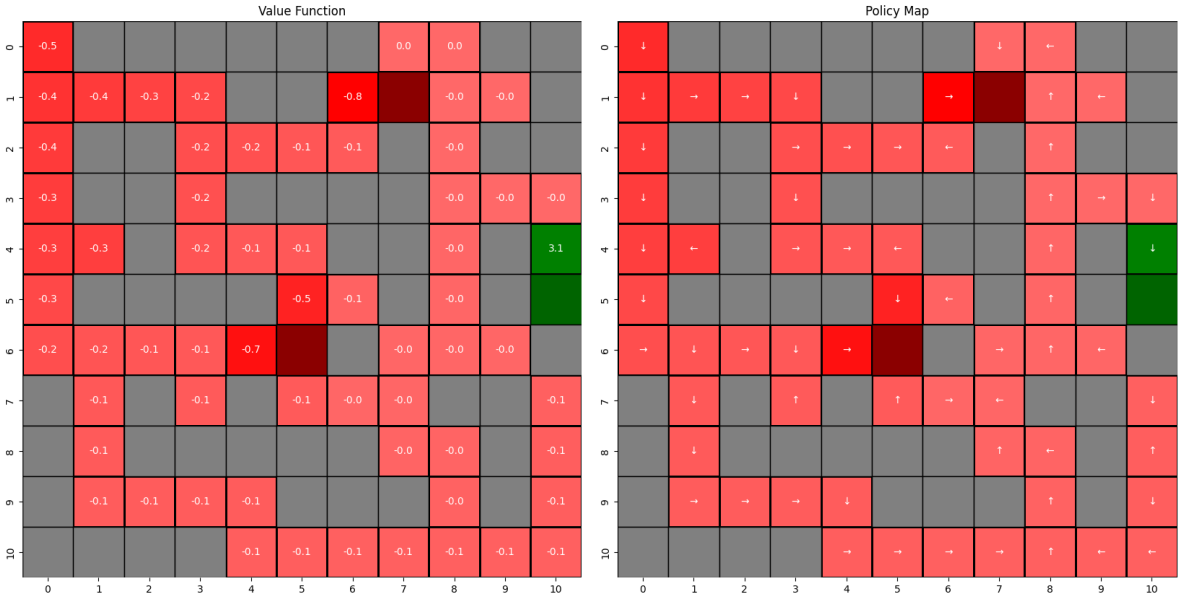


Figure 87: alpha = 0.001 for 100 episode

(a) utility values (b) policies

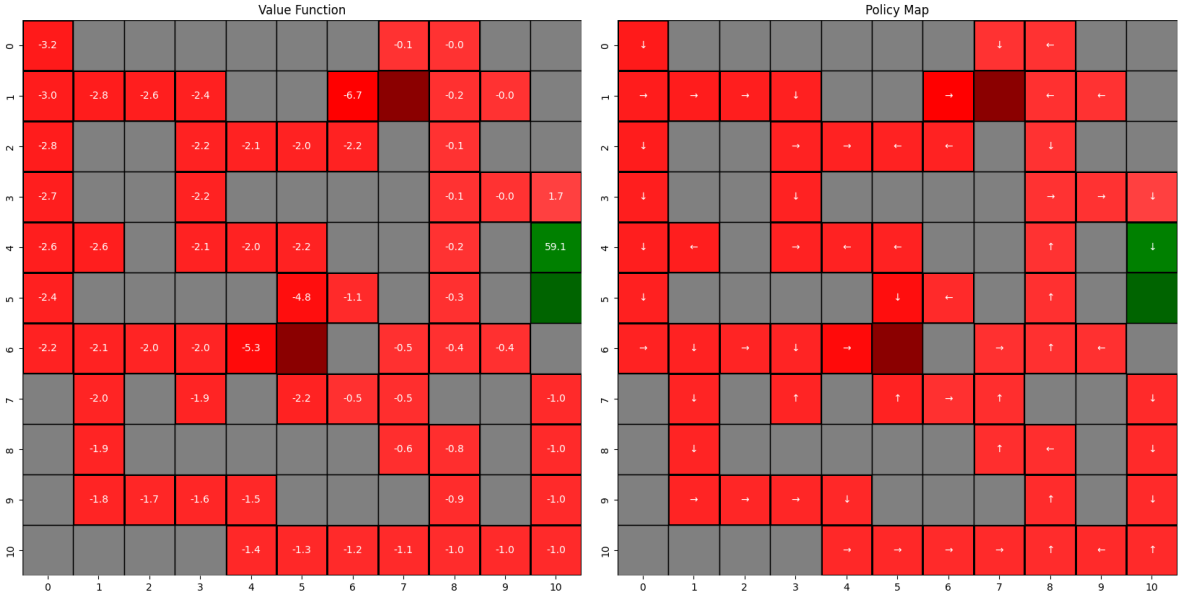
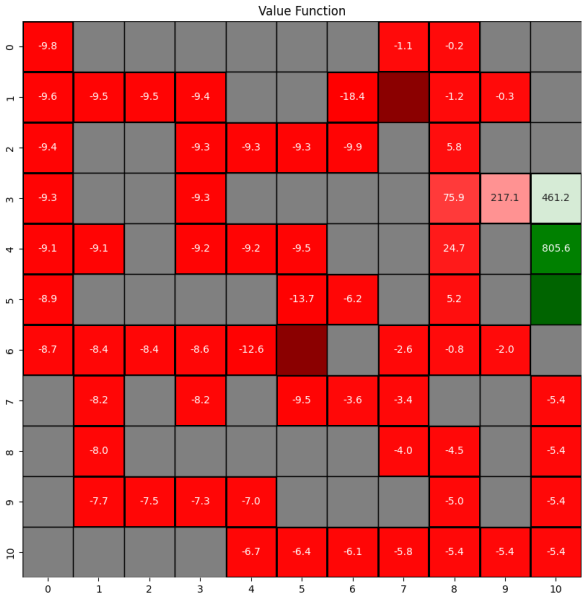


Figure 88: alpha = 0.001 for 1000 episode

(a) utility values



(b) policies

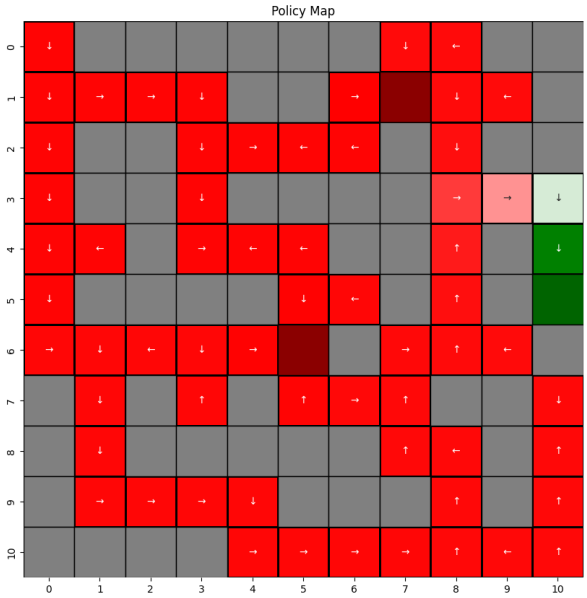
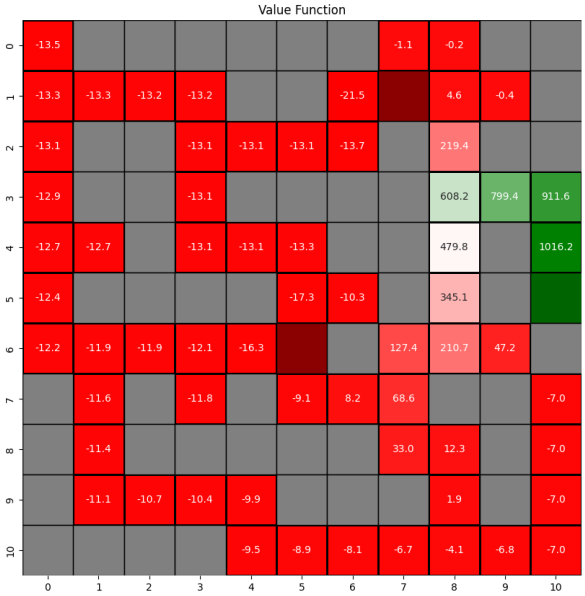


Figure 89: alpha = 0.001 for 5000 episode

(a) utility values



(b) policies

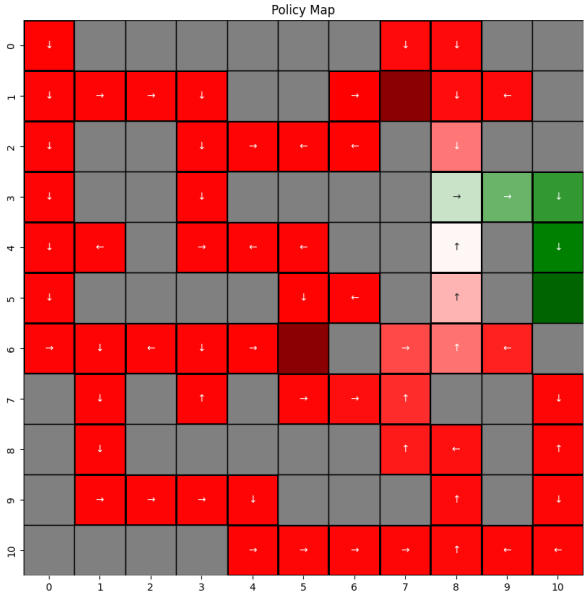


Figure 90: alpha = 0.001 for 10000 episode

(a) utility values (b) policies

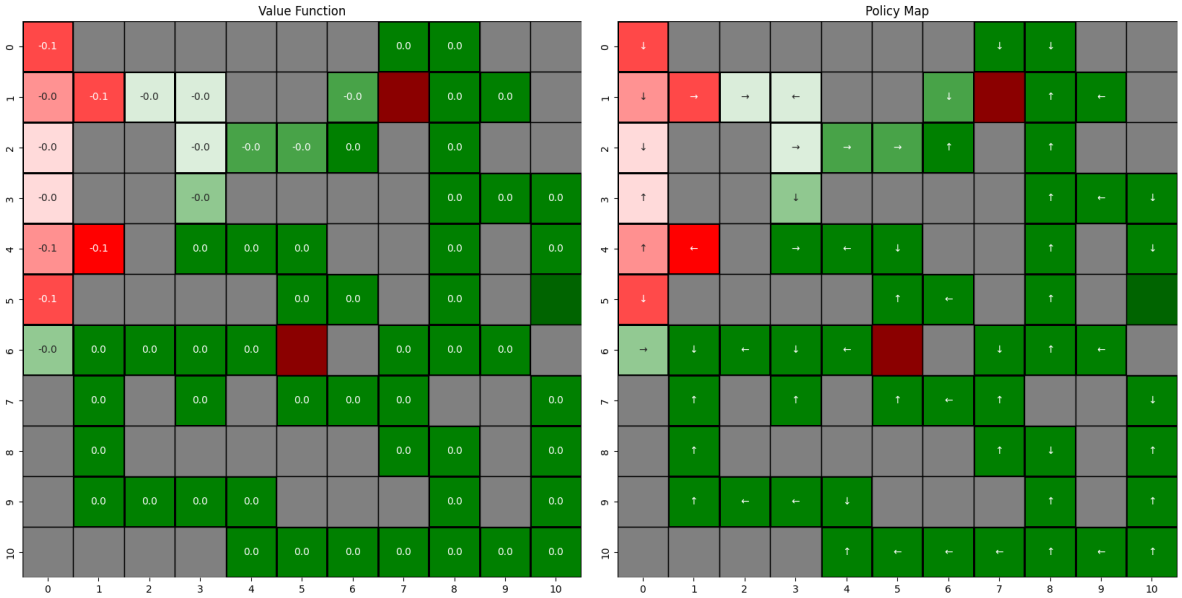


Figure 91: alpha = 0.01 for 1 episode

(a) utility values (b) policies

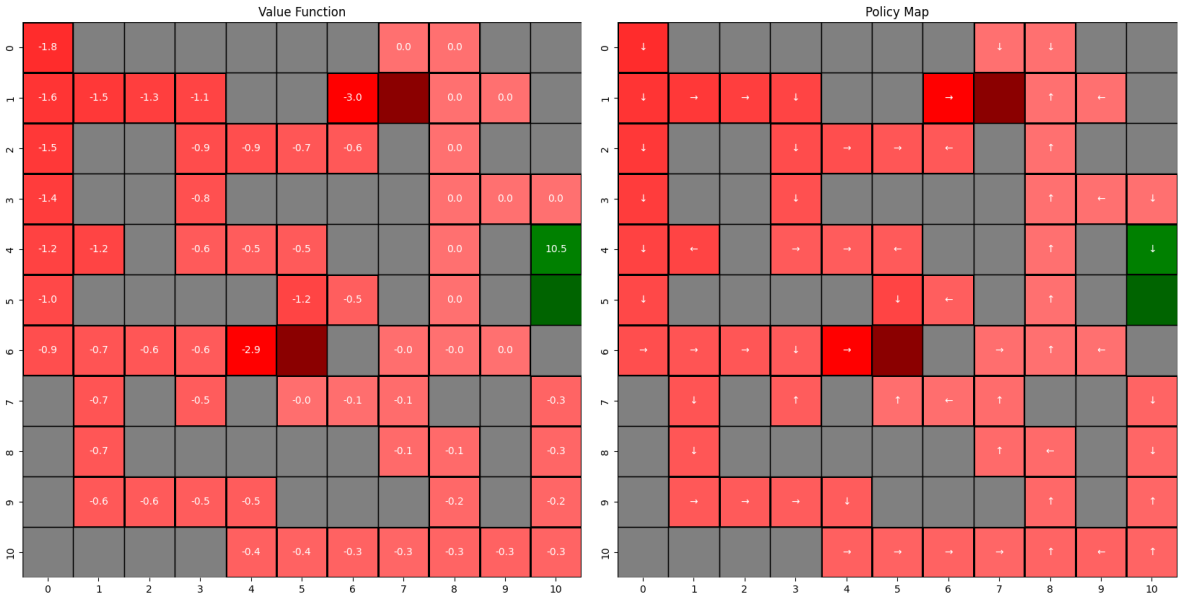


Figure 92: alpha = 0.01 for 50 episode

(a) utility values (b) policies

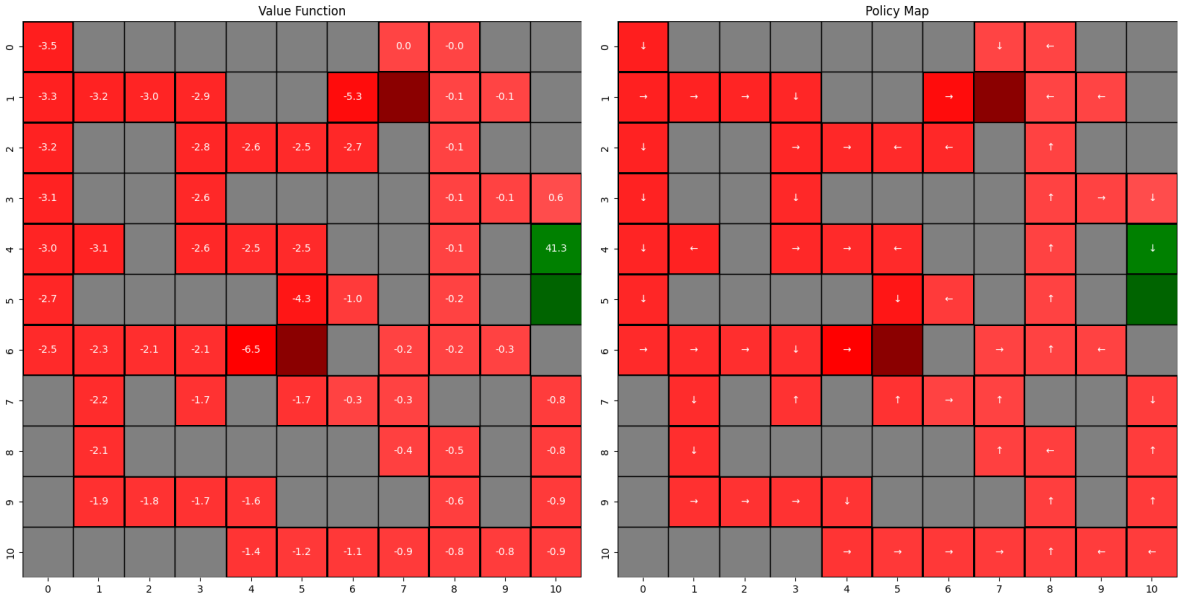


Figure 93: alpha = 0.01 for 100 episode

(a) utility values (b) policies

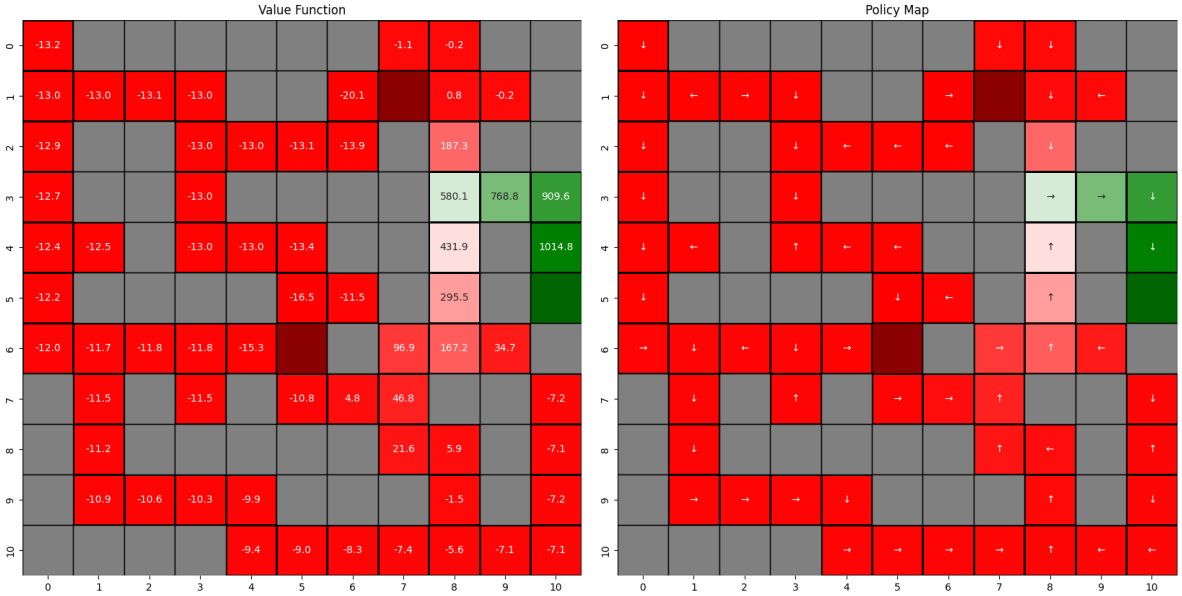
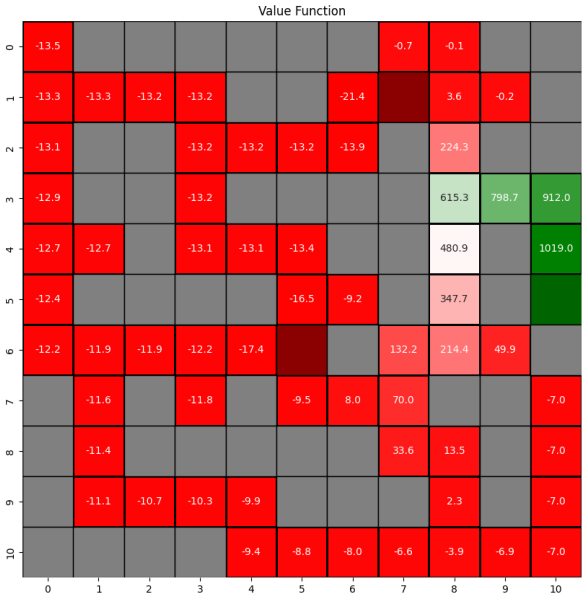


Figure 94: alpha = 0.01 for 1000 episode

(a) utility values



(b) policies

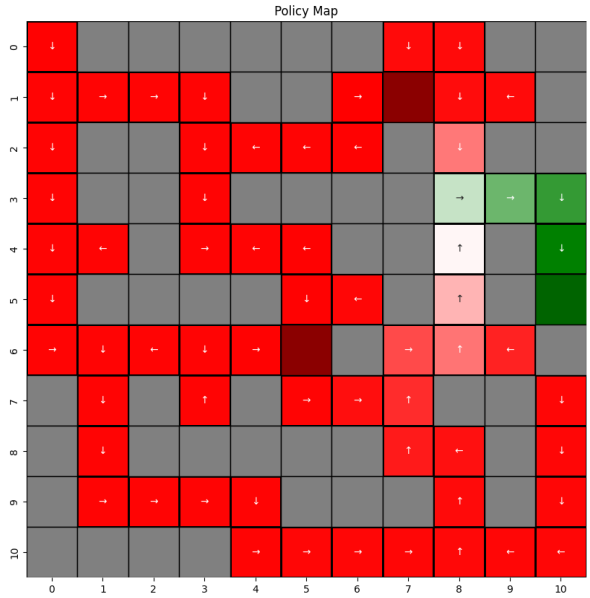
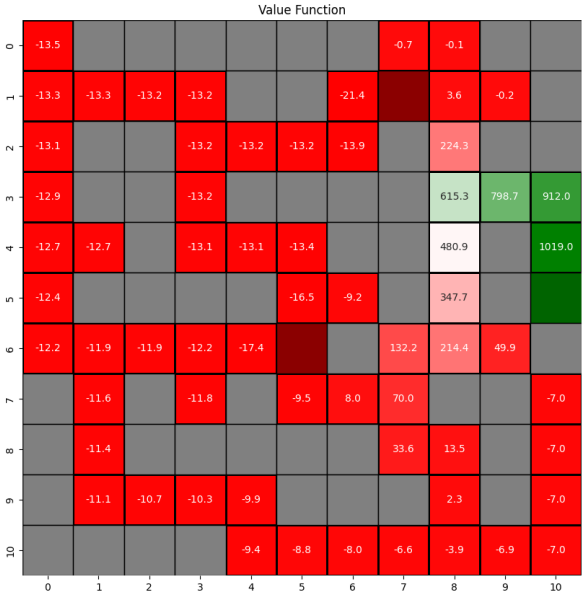


Figure 95: alpha = 0.01 for 5000 episode

(a) utility values



(b) policies

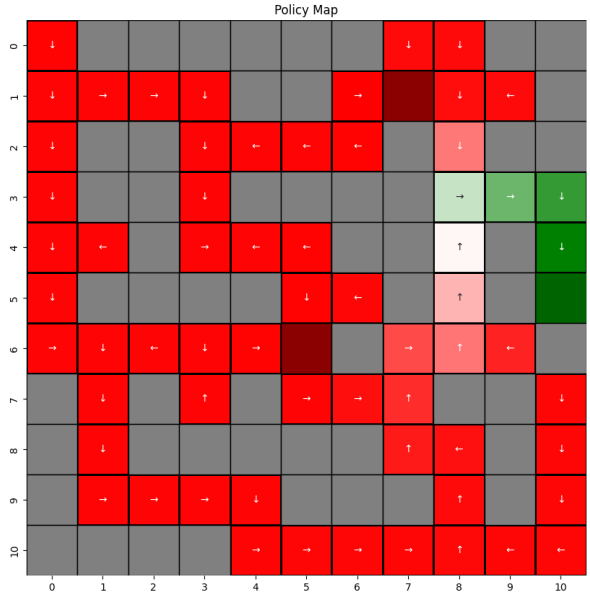


Figure 96: alpha = 0.01 for 10000 episode

(a) utility values (b) policies

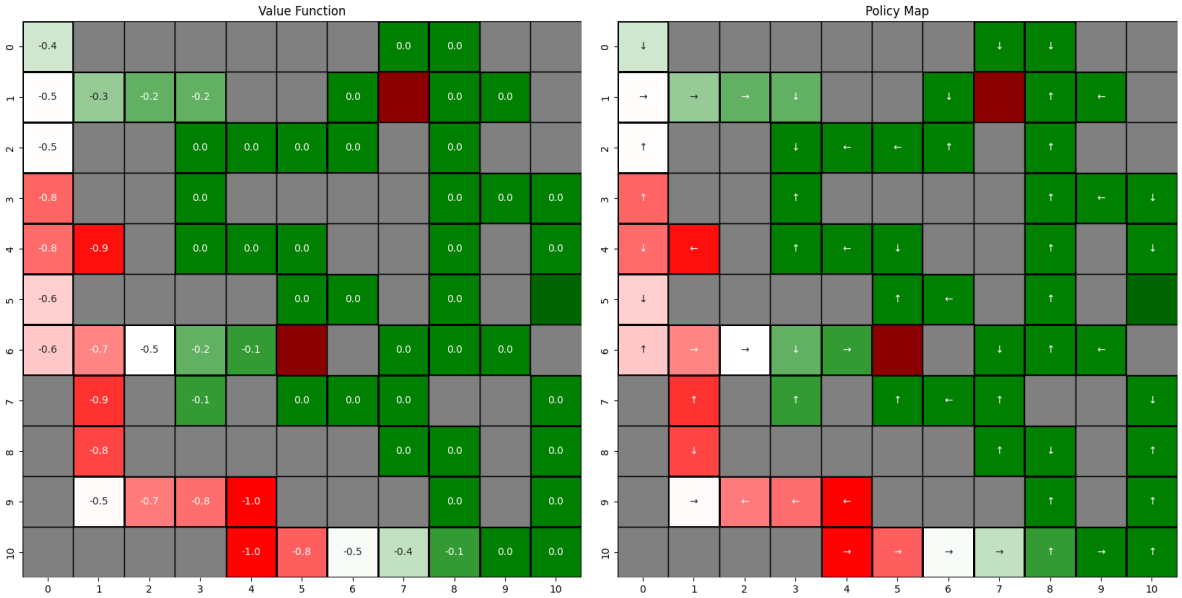


Figure 97: alpha = 0.1 for 1 episode

(a) utility values (b) policies

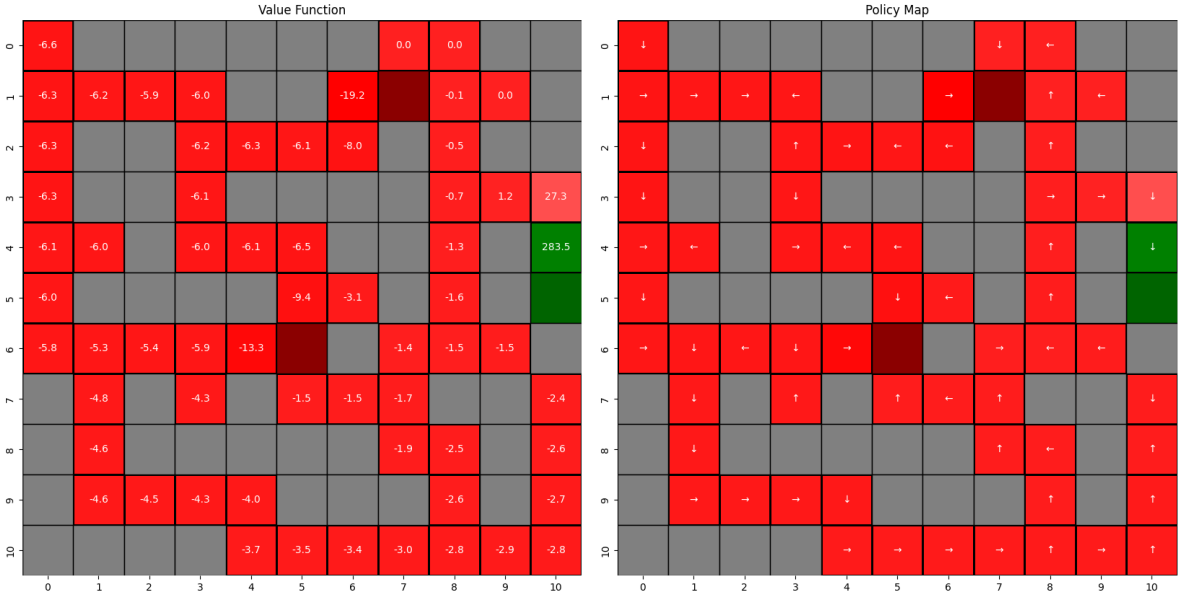
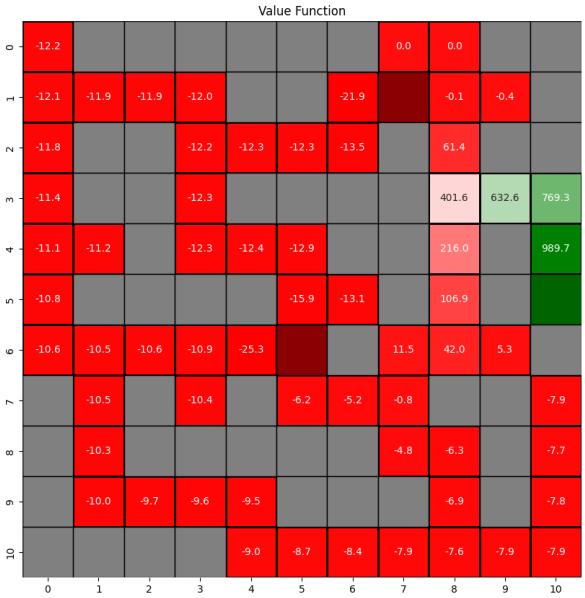


Figure 98: alpha = 0.1 for 50 episode

(a) utility values



(b) policies

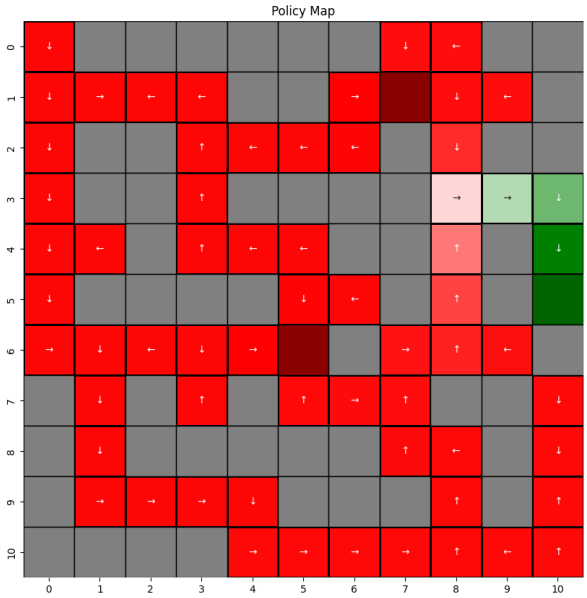
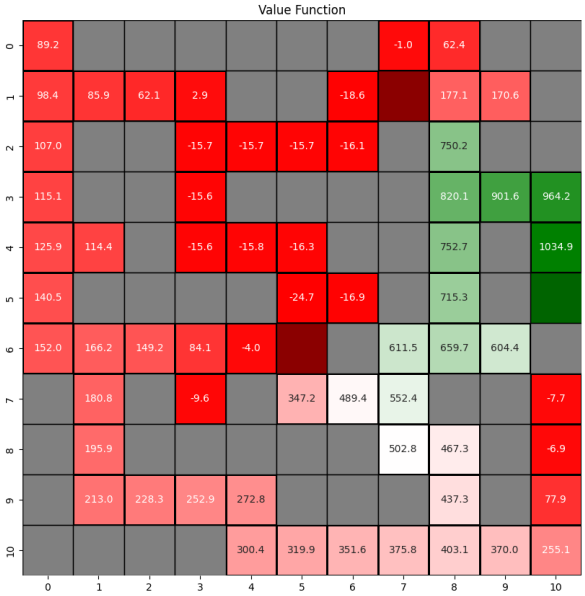


Figure 99: alpha = 0.1 for 100 episode

(a) utility values



(b) policies

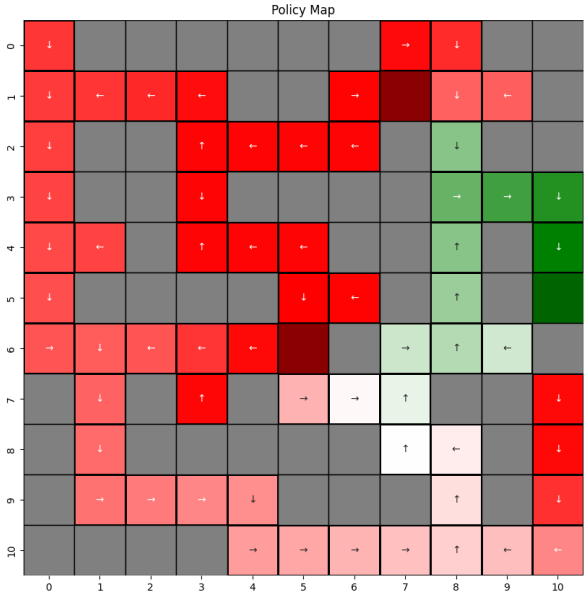
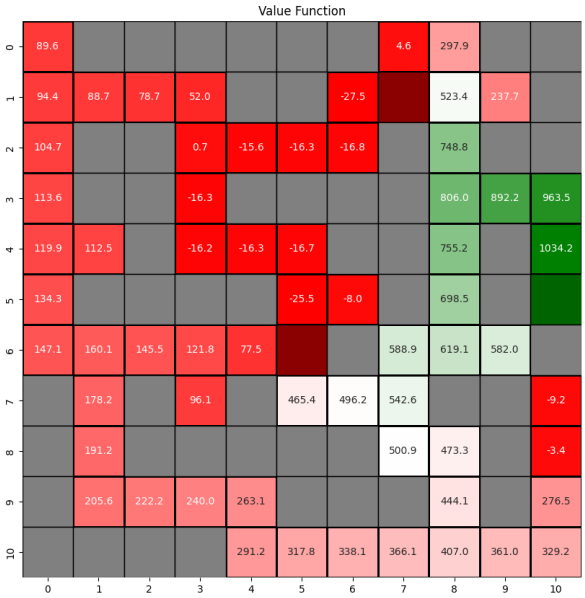


Figure 100: alpha = 0.1 for 1000 episode

(a) utility values



(b) policies

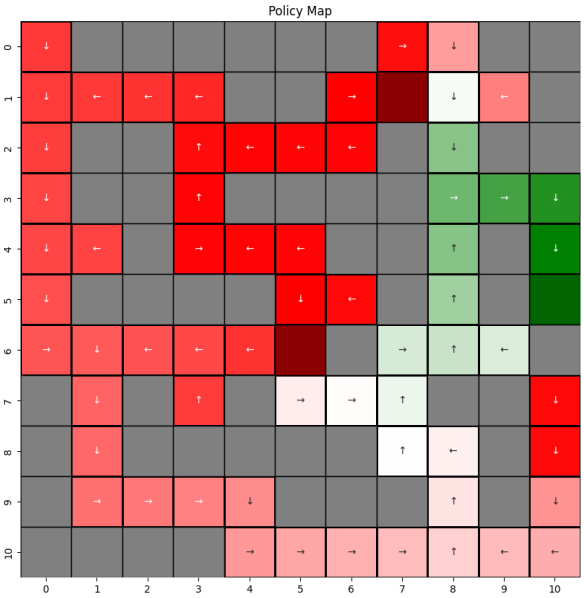
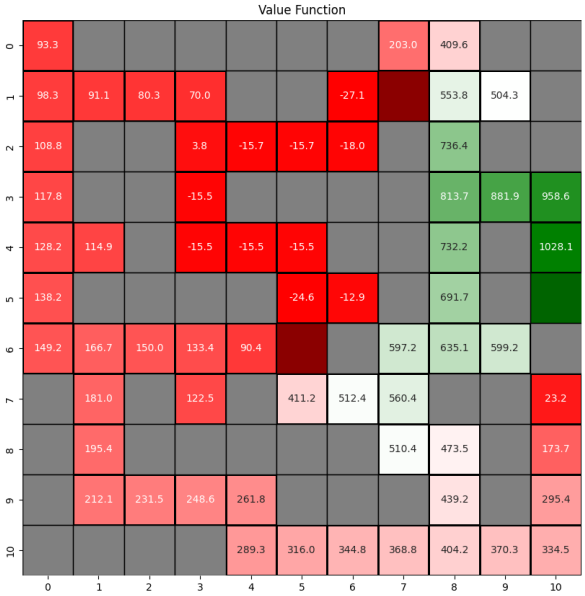


Figure 101: alpha = 0.1 for 5000 episode

(a) utility values



(b) policies

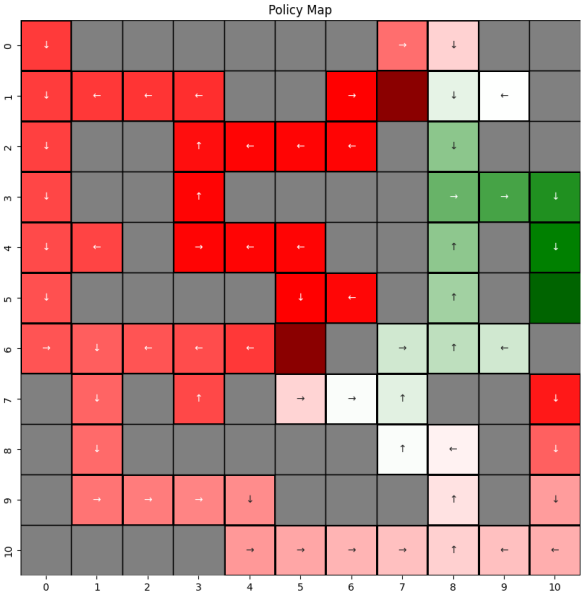
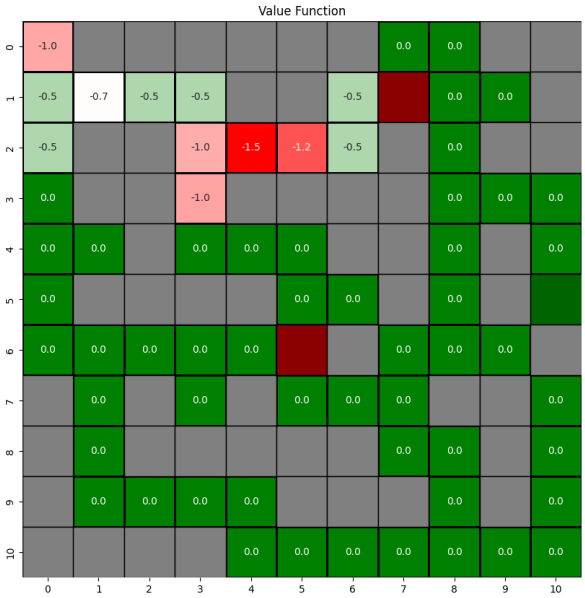


Figure 102: alpha = 0.1 for 10000 episode

(a) utility values



(b) policies

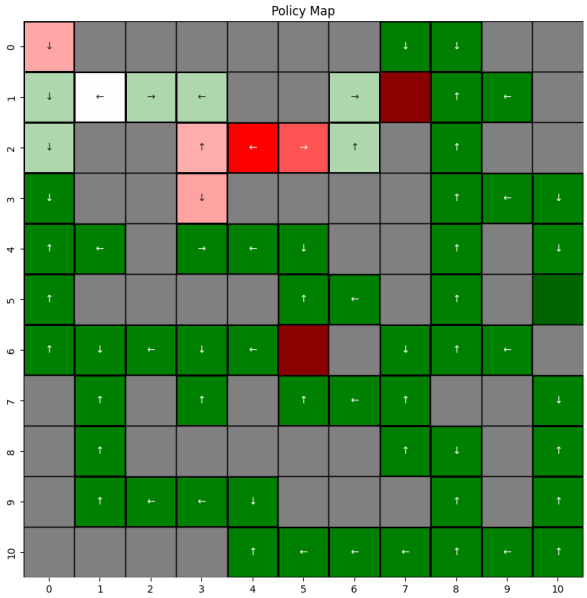
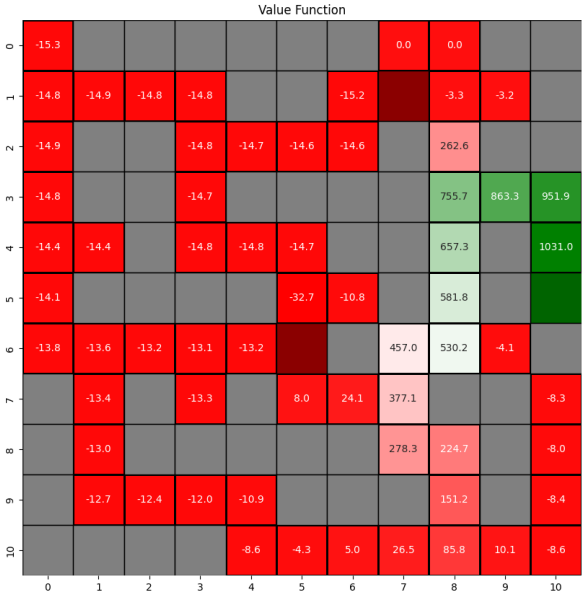


Figure 103: alpha = 0.5 for 1 episode

(a) utility values



(b) policies

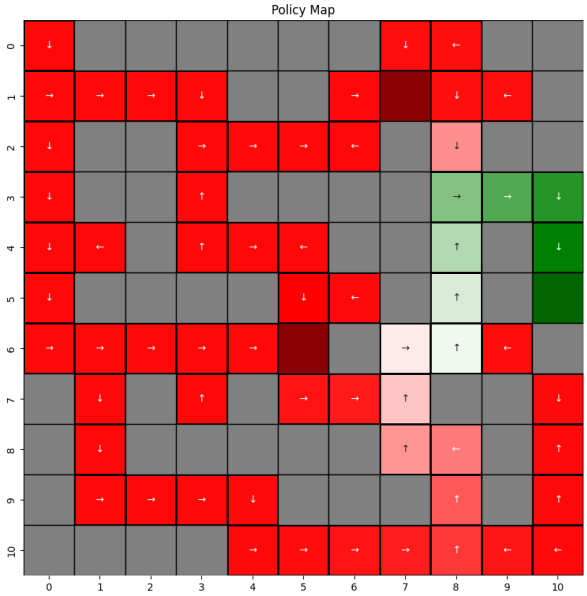
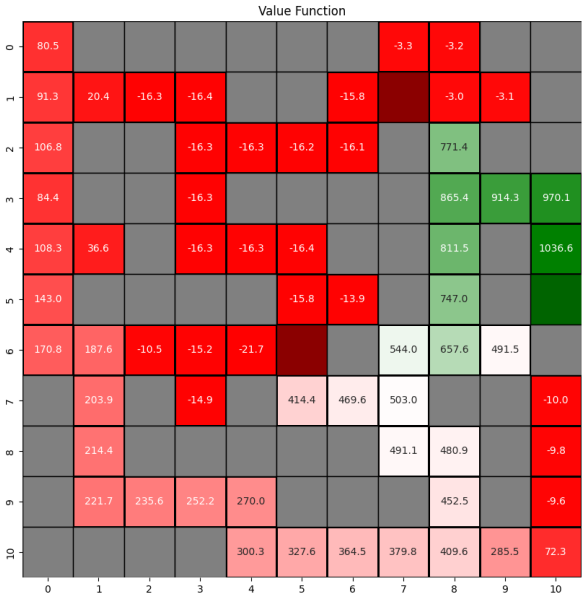


Figure 104: alpha = 0.5 for 50 episode

(a) utility values



(b) policies

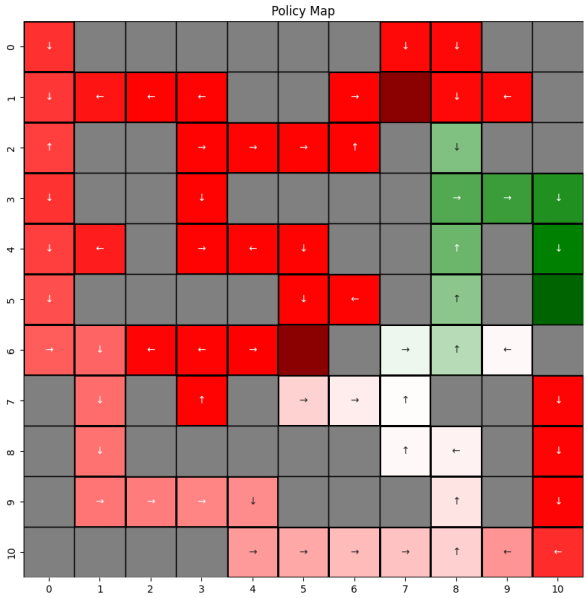
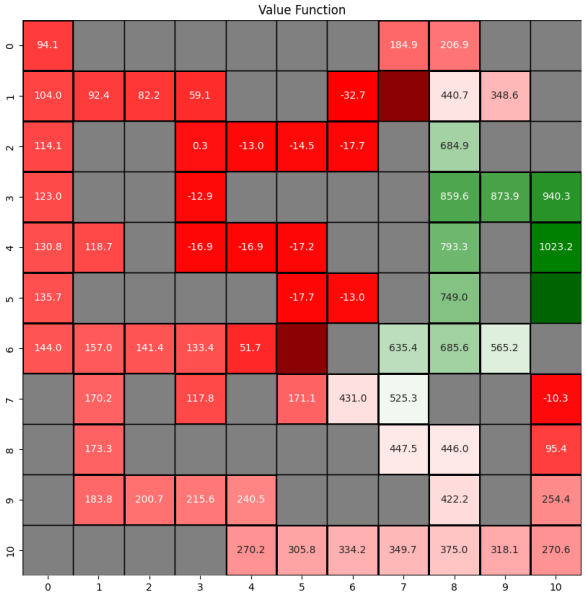


Figure 105: alpha = 0.5 for 100 episode

(a) utility values



(b) policies

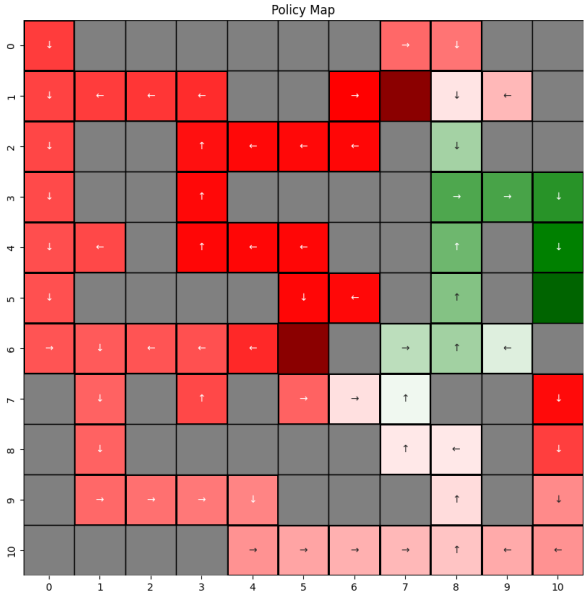
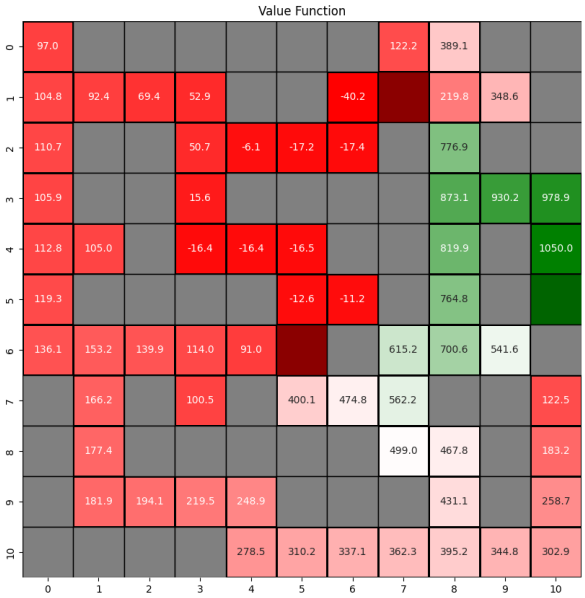


Figure 106: alpha = 0.5 for 1000 episode

(a) utility values



(b) policies

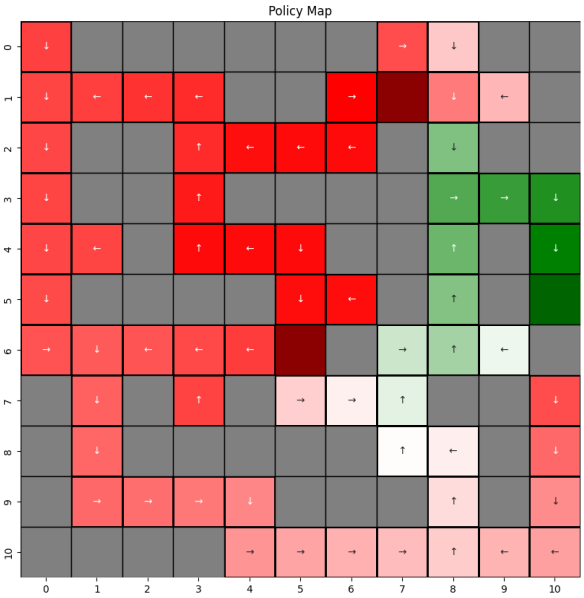
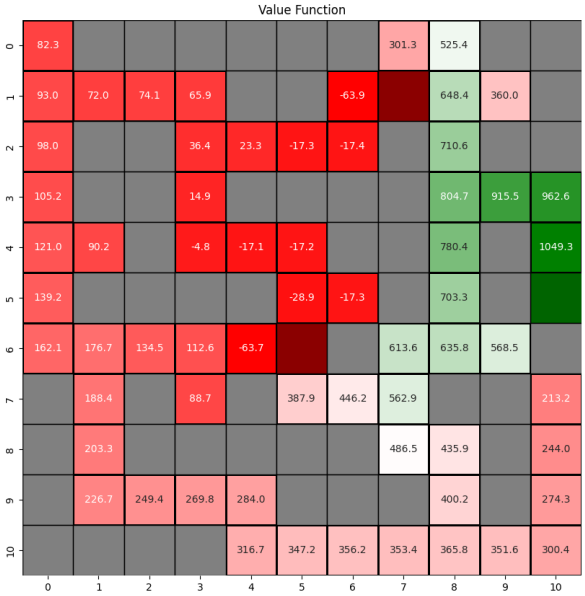


Figure 107: alpha = 0.5 for 5000 episode

(a) utility values



(b) policies

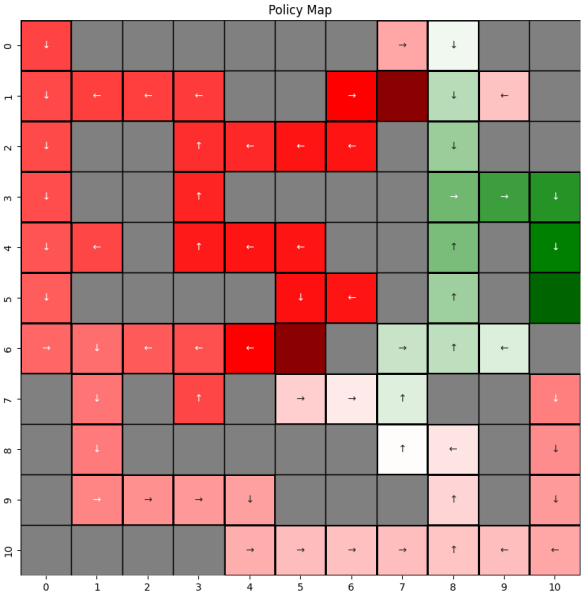
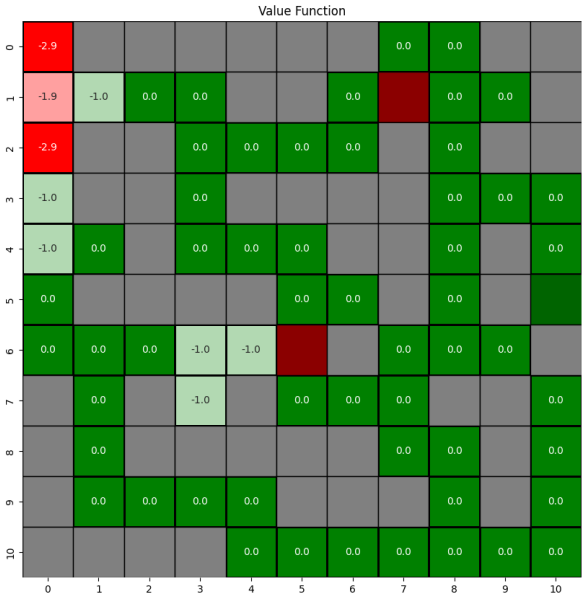


Figure 108: alpha = 0.5 for 10000 episode

(a) utility values



(b) policies

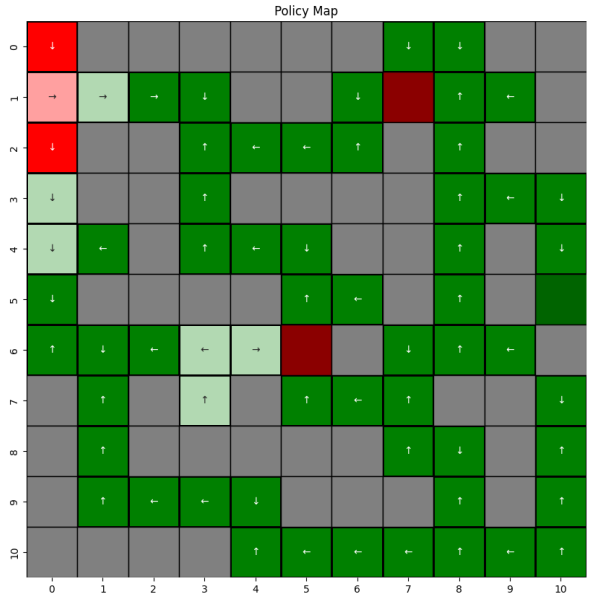
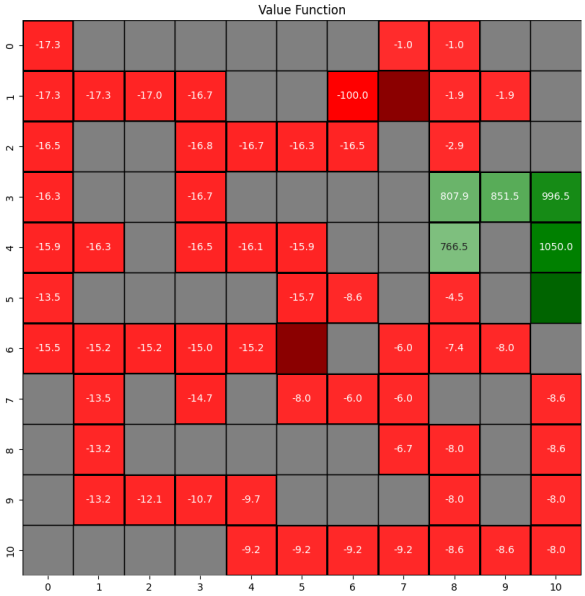


Figure 109: alpha = 1 for 1 episode

(a) utility values



(b) policies

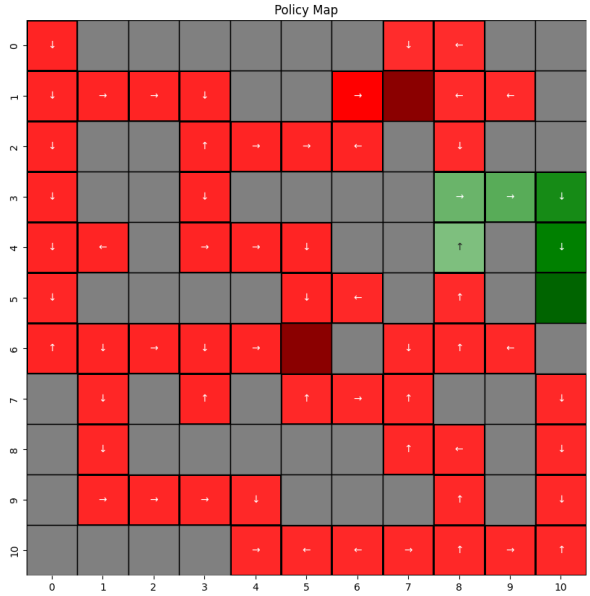
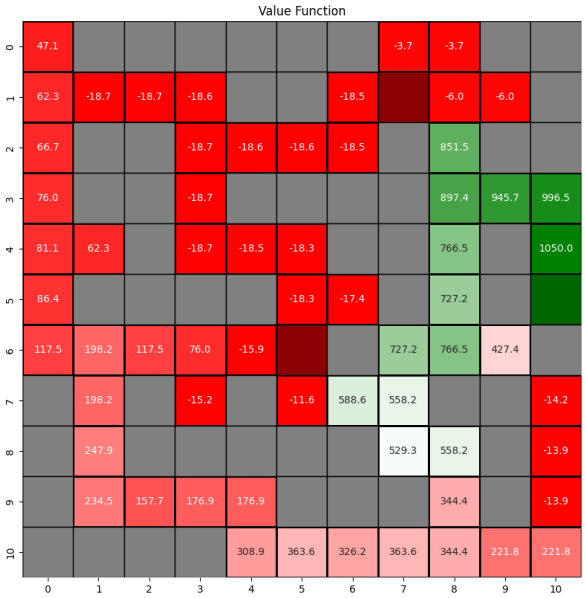


Figure 110: alpha = 1 for 50 episode

(a) utility values



(b) policies

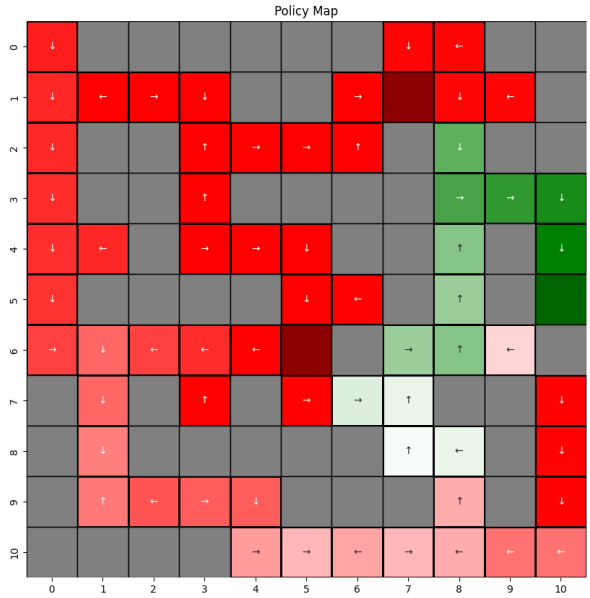
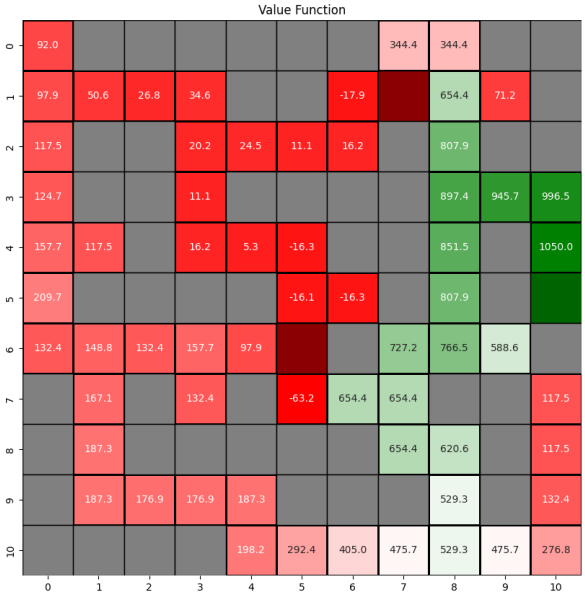


Figure 111: alpha = 1 for 100 episode

(a) utility values



(b) policies

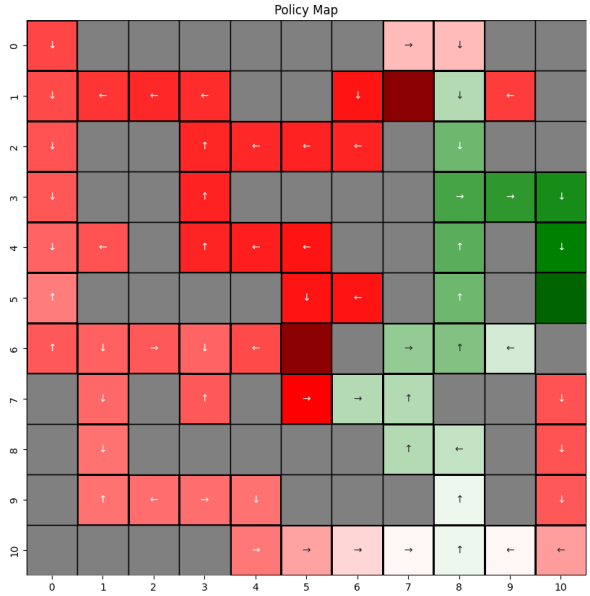
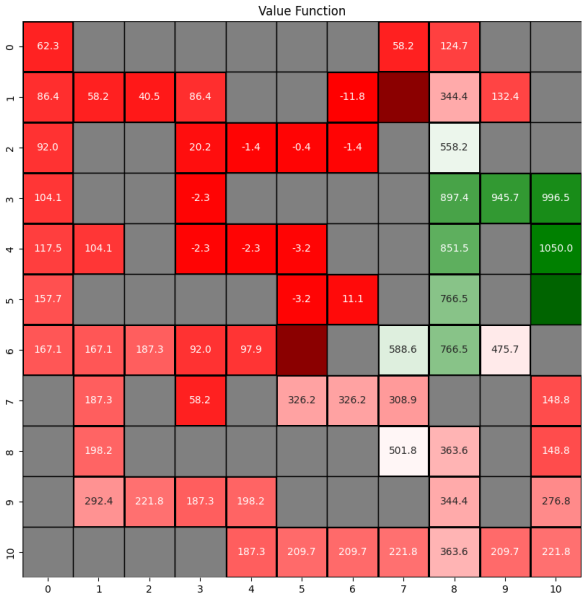


Figure 112: alpha = 1 for 1000 episode

(a) utility values



(b) policies

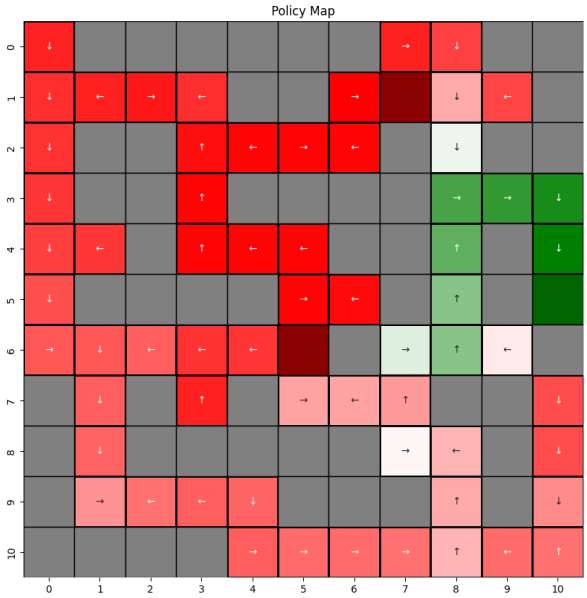
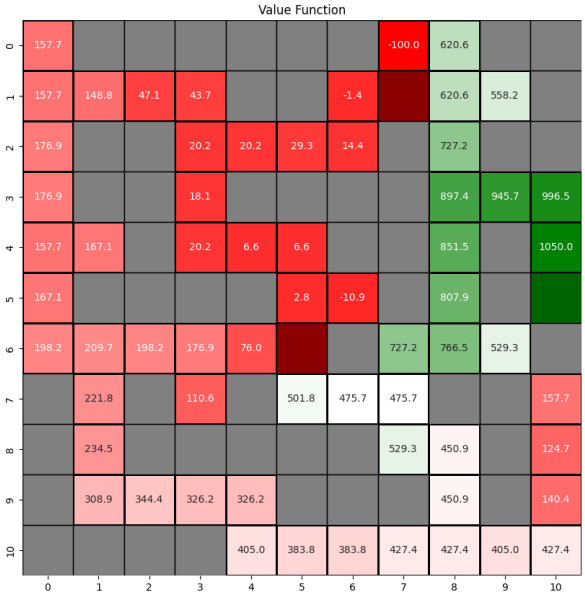


Figure 113: alpha = 1 for 5000 episode

(a) utility values



(b) policies

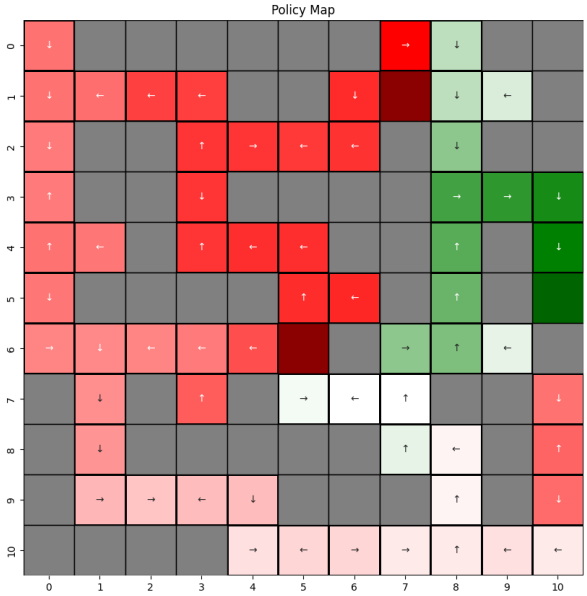
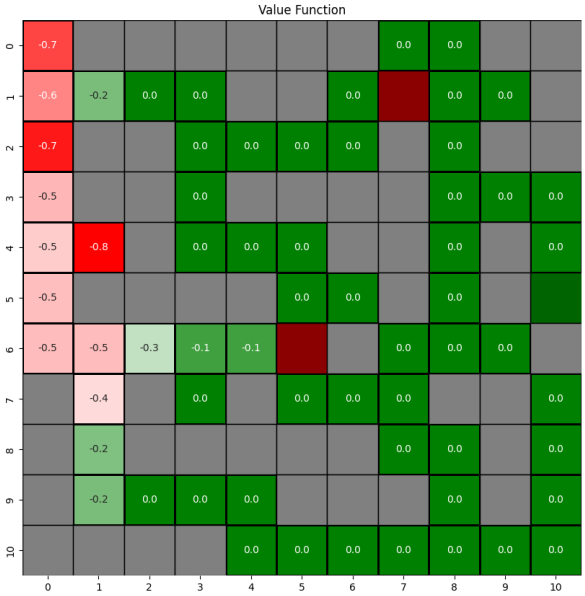


Figure 114: alpha = 1 for 10000 episode

For each value of alpha, there are many successful runs here, but there can be a few key takeaways spotted in this part. which are: higher alpha value increases the effect of the reward of the goal state reaching further; however, similar to TD learning, increased alpha can lead to fast convergence of the beginning states before the agent gains the ability to reliably find the goal state, which leads to agent getting stuck at first states.

Effect of epsilon in Q learning

(a) utility values



(b) policies

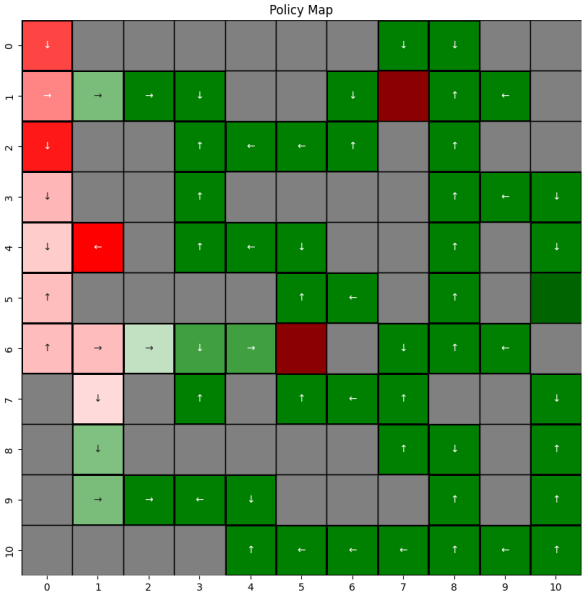
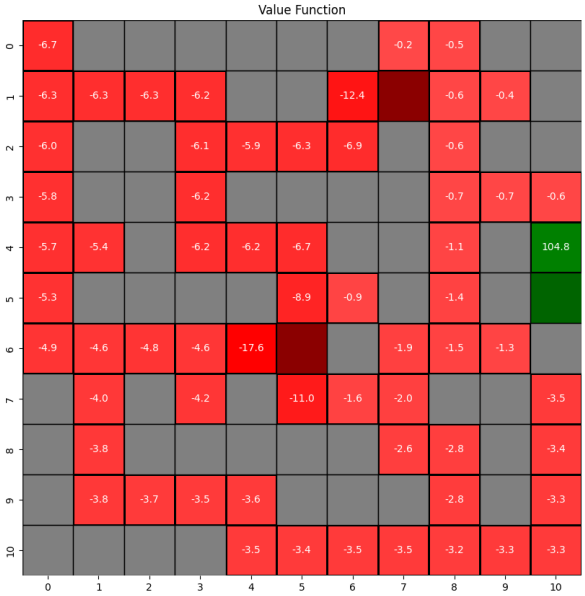


Figure 115: epsilon = 0 for 1 episode

(a) utility values



(b) policies

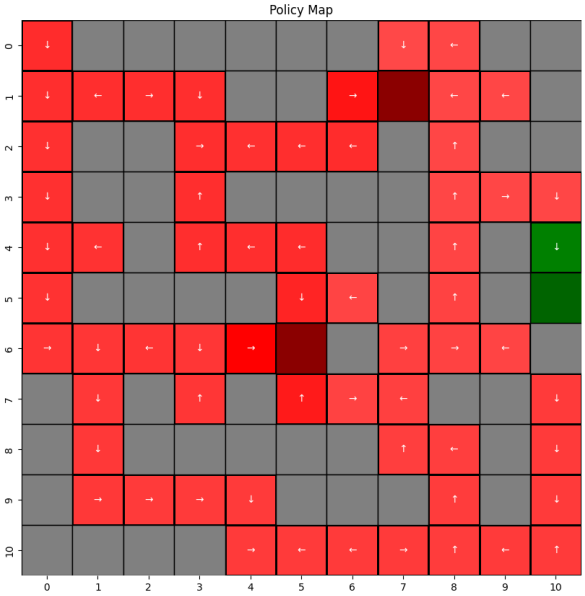
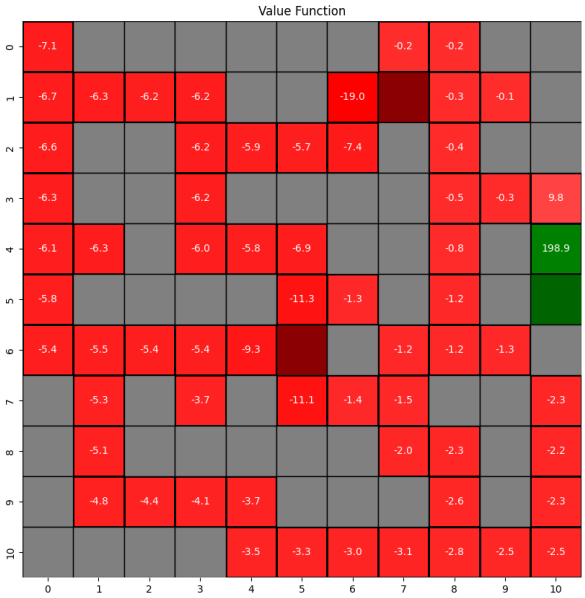


Figure 116: epsilon = 0 for 50 episode

(a) utility values



(b) policies

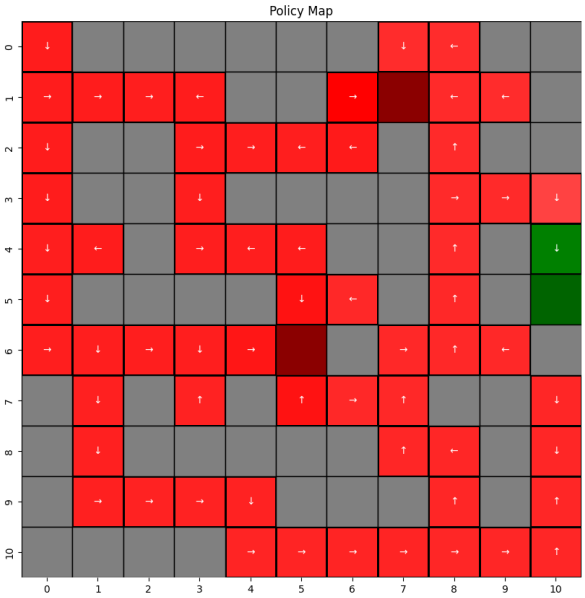
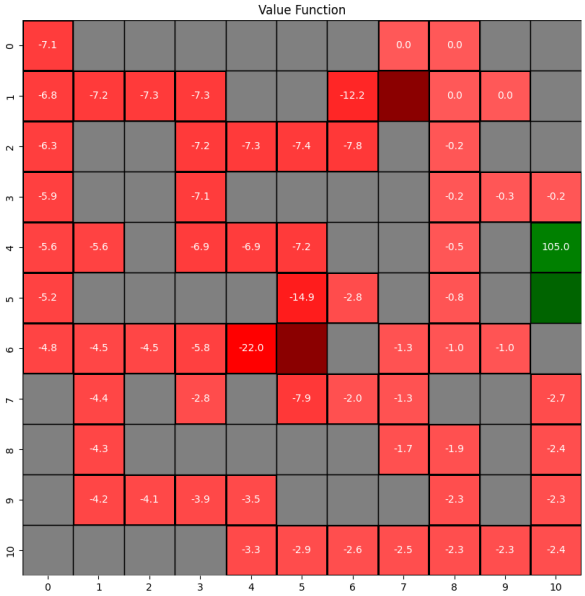


Figure 117: epsilon = 0 for 100 episode

(a) utility values



(b) policies

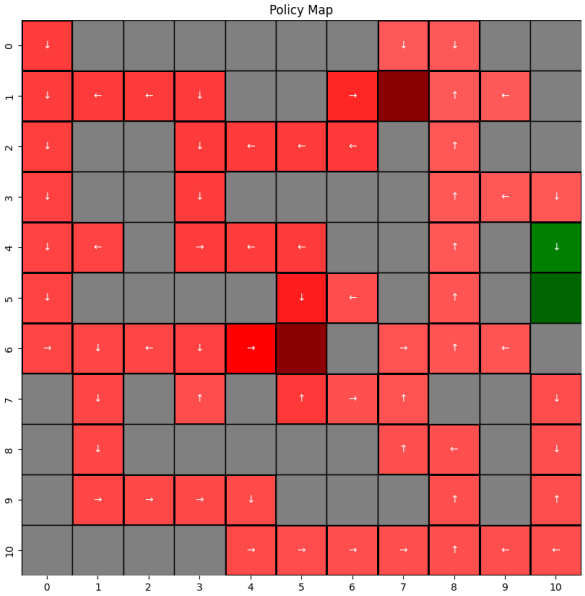
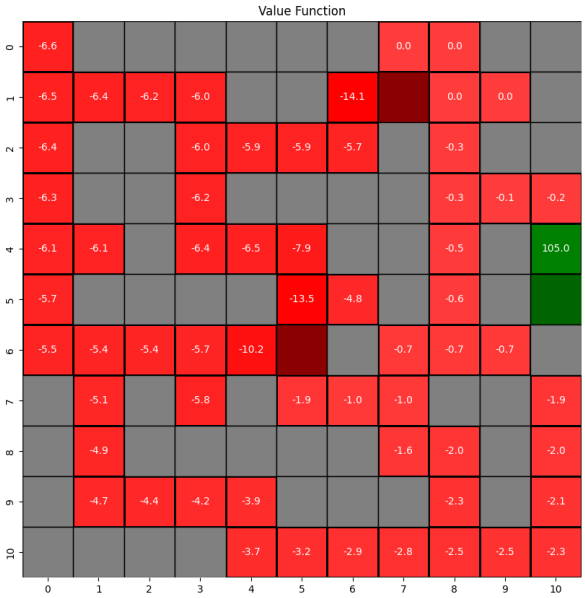


Figure 118: epsilon = 0 for 1000 episode

(a) utility values



(b) policies

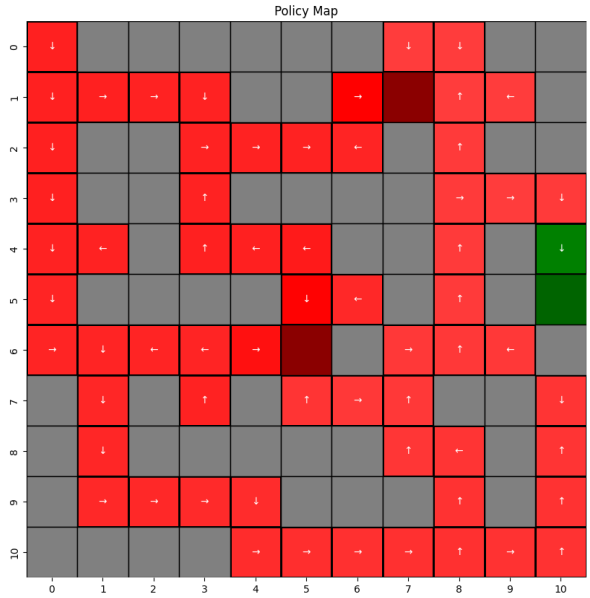
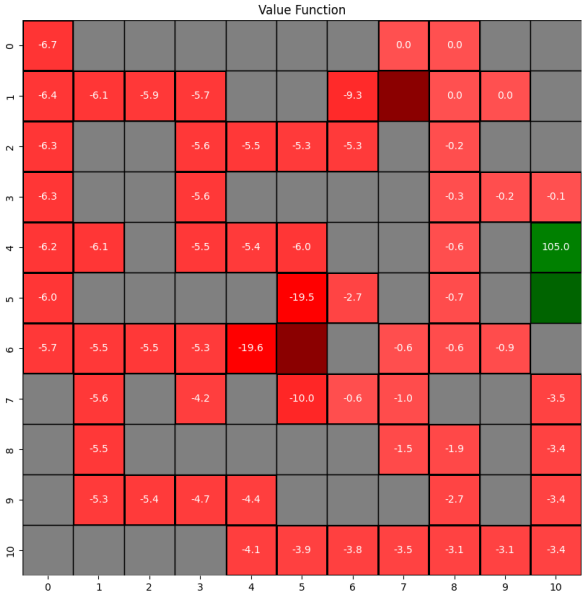


Figure 119: epsilon = 0 for 5000 episode

(a) utility values



(b) policies

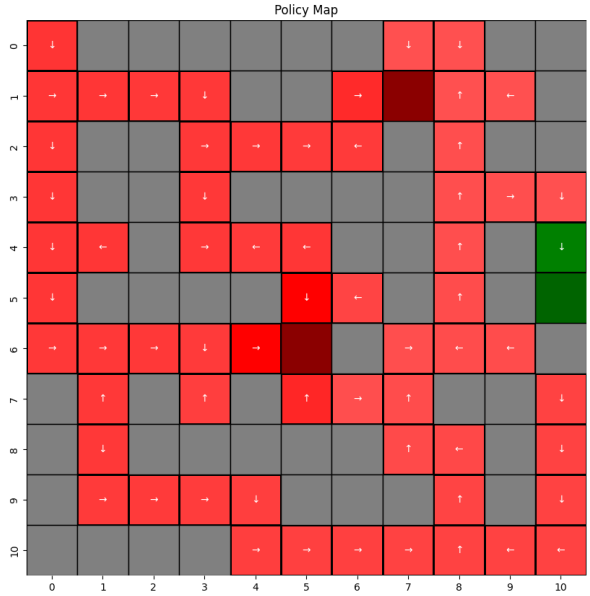
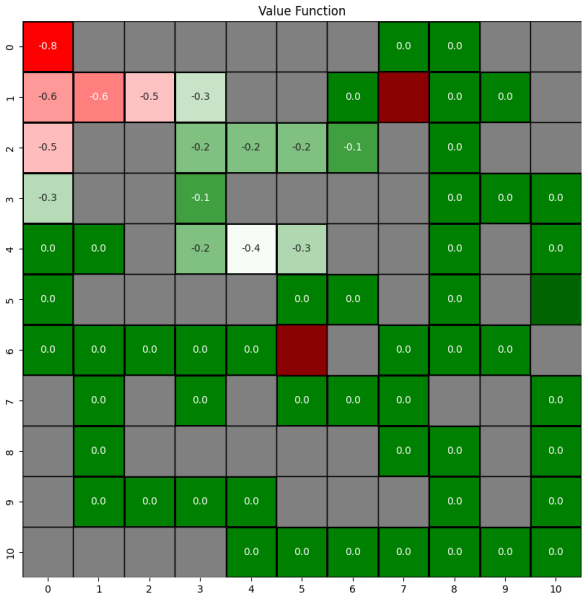


Figure 120: epsilon = 0 for 10000 episode

(a) utility values



(b) policies

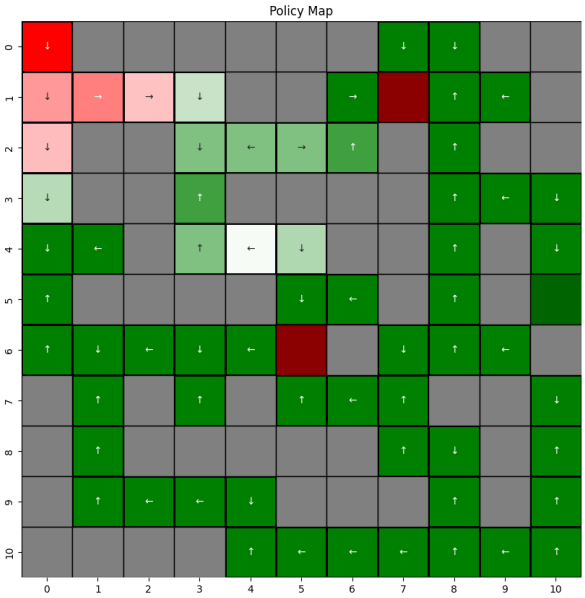
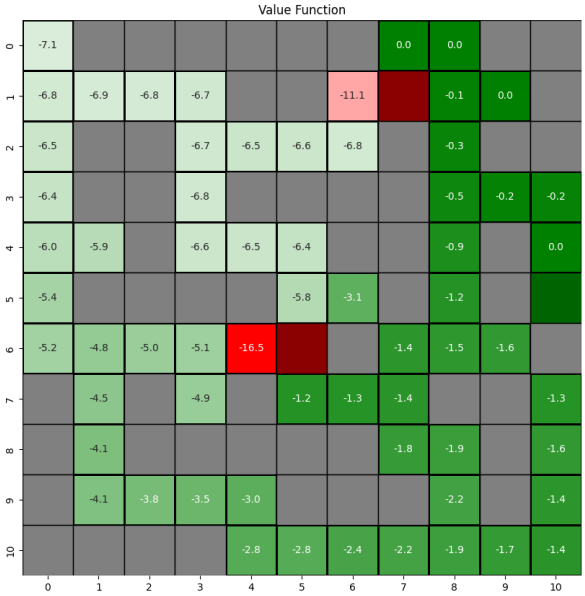


Figure 121: epsilon = 0.2 for 1 episode

(a) utility values



(b) policies

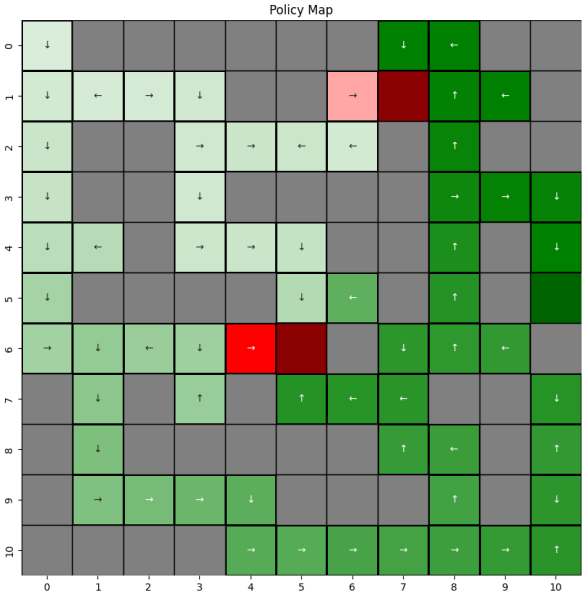


Figure 122: epsilon = 0.2 for 50 episode

(a) utility values (b) policies

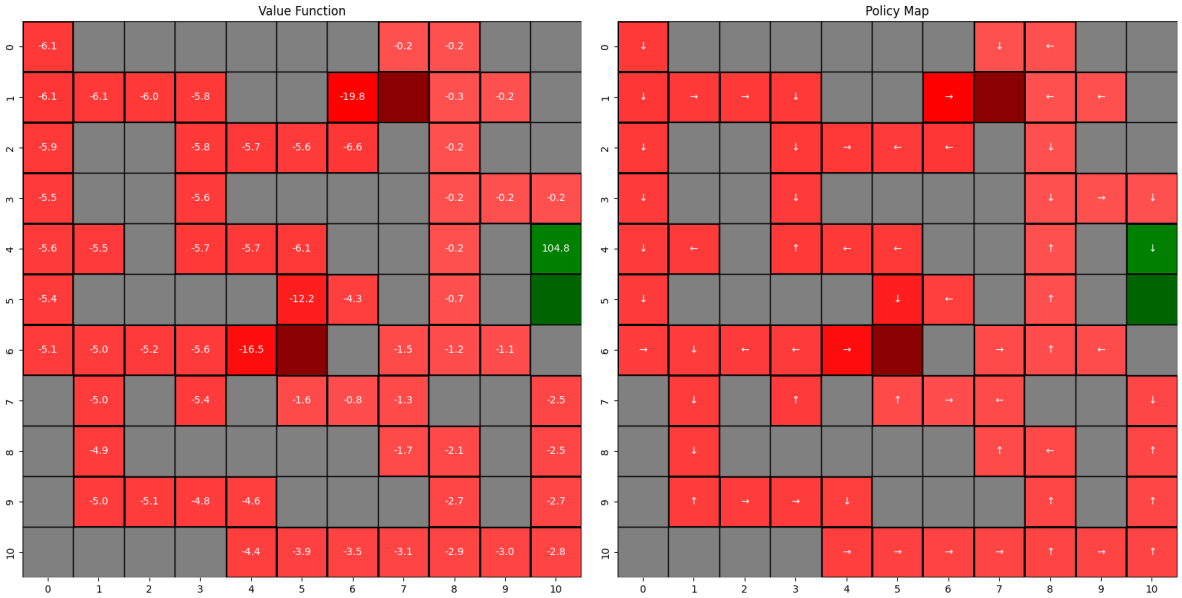


Figure 123: epsilon = 0.2 for 100 episode

(a) utility values (b) policies

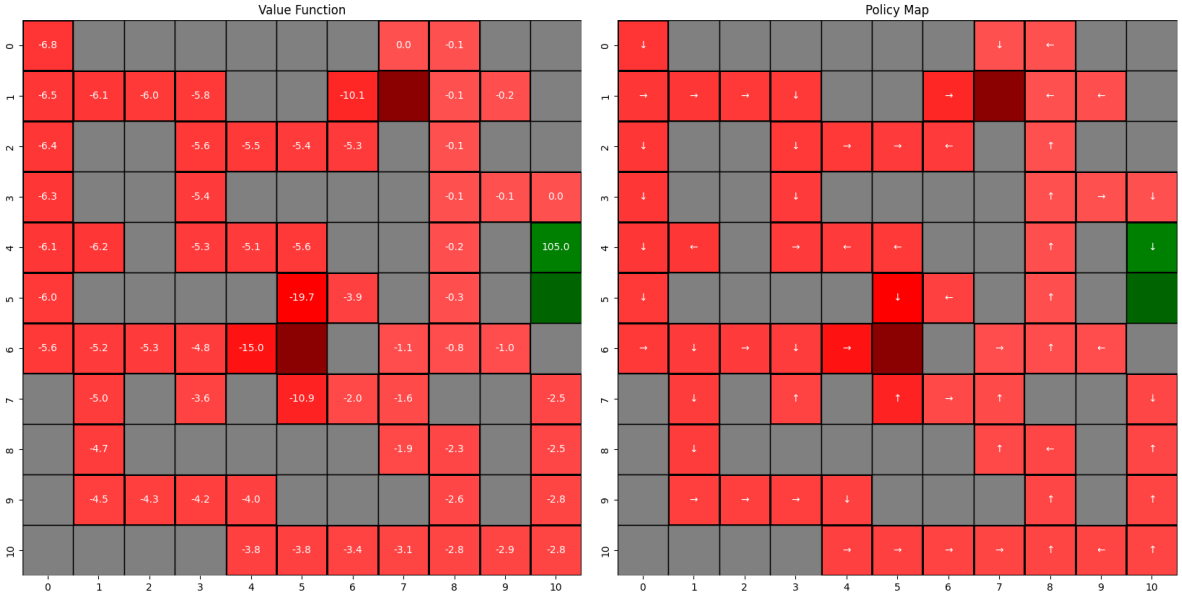
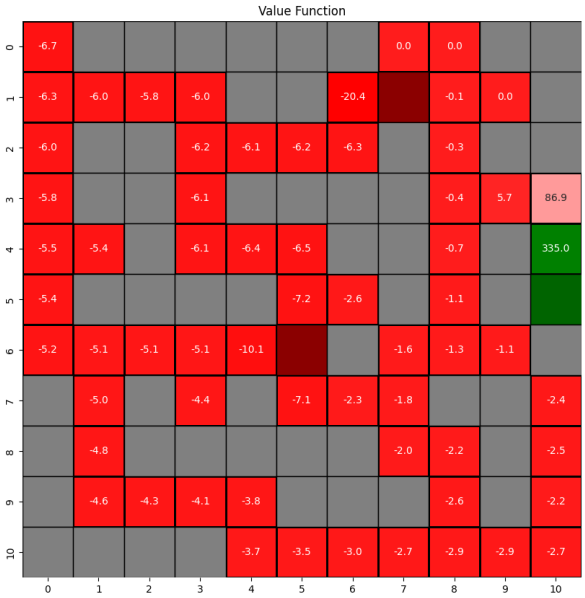


Figure 124: epsilon = 0.2 for 1000 episode

(a) utility values



(b) policies

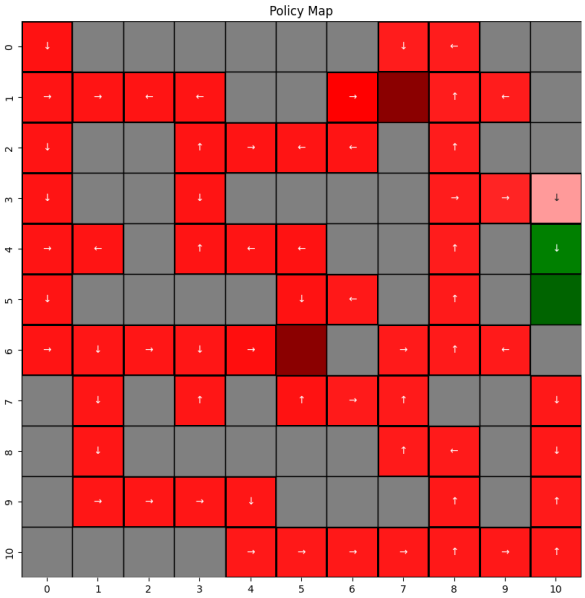
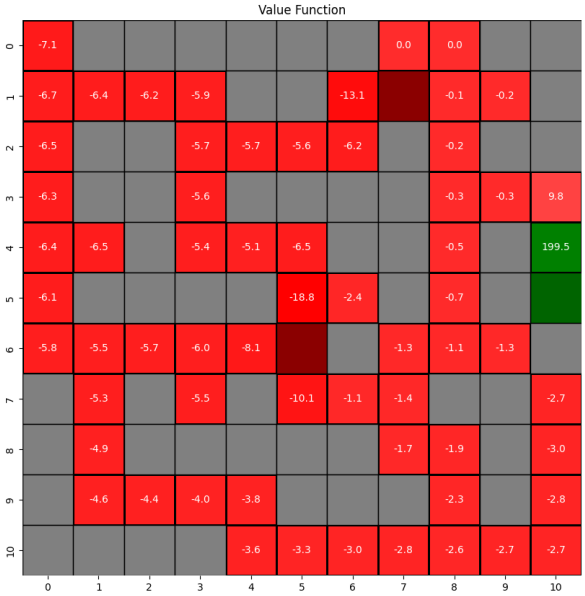


Figure 125: epsilon = 0.2 for 5000 episode

(a) utility values



(b) policies

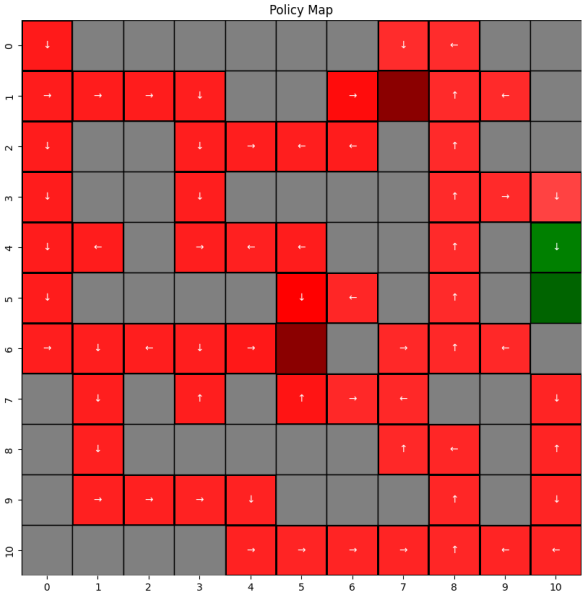
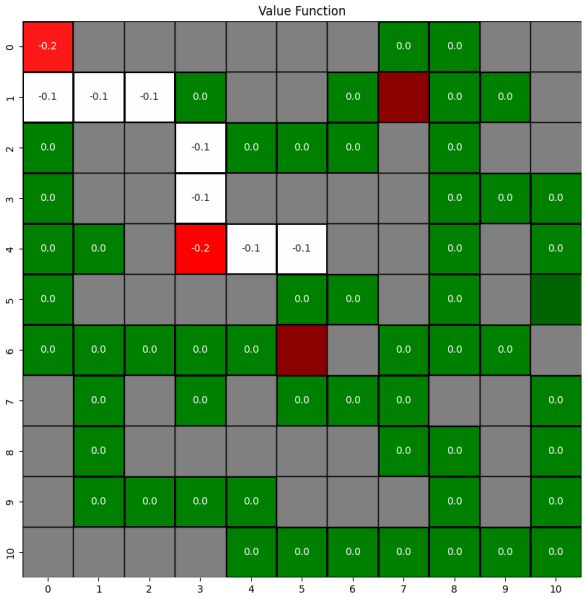


Figure 126: epsilon = 0.2 for 10000 episode

(a) utility values



(b) policies

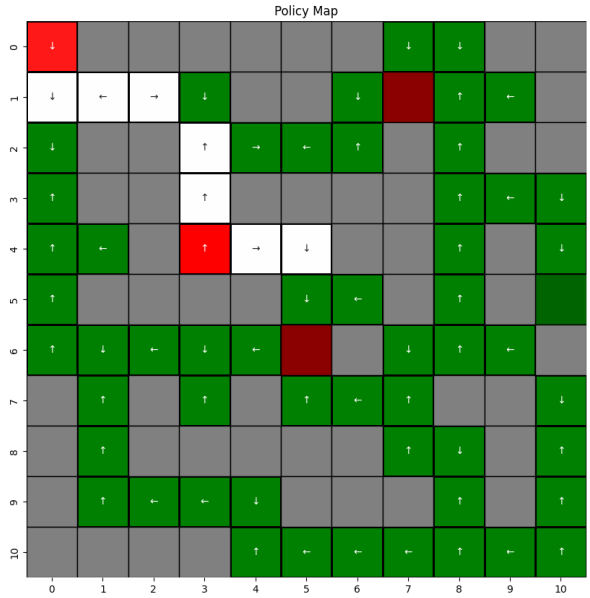
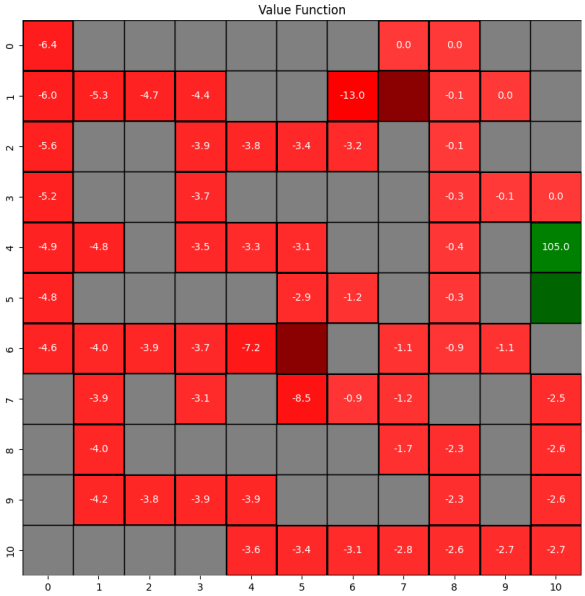


Figure 127: epsilon = 0.5 for 1 episode

(a) utility values



(b) policies

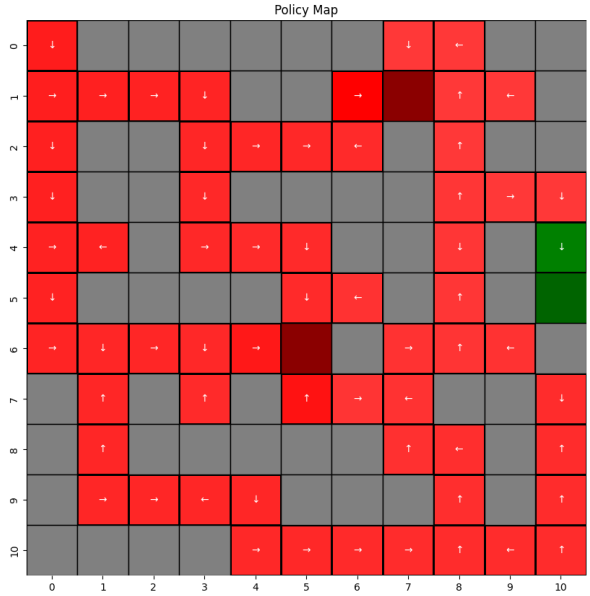
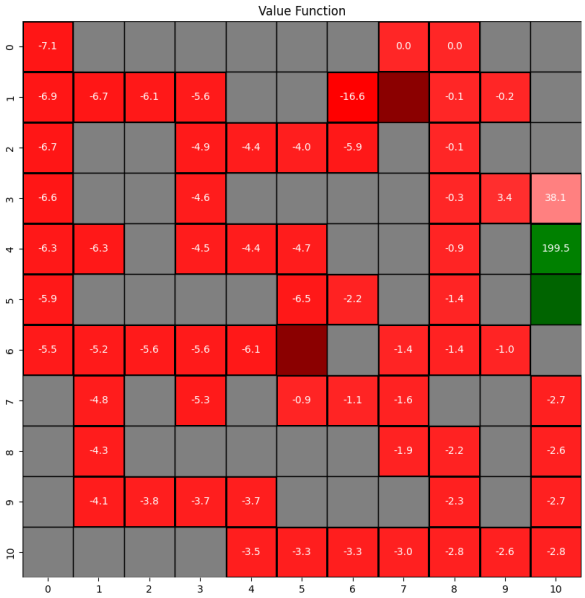


Figure 128: epsilon = 0.5 for 50 episode

(a) utility values



(b) policies

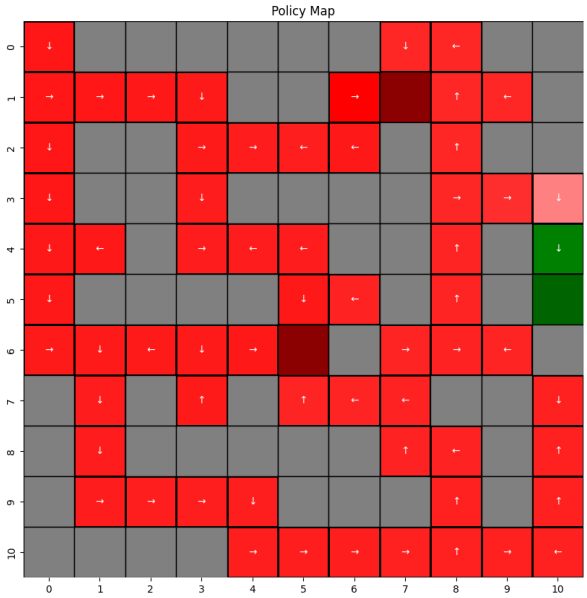
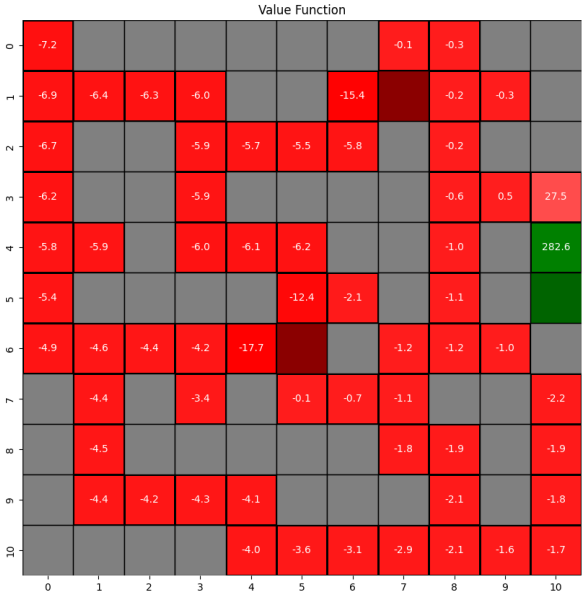


Figure 129: epsilon = 0.5 for 100 episode

(a) utility values



(b) policies

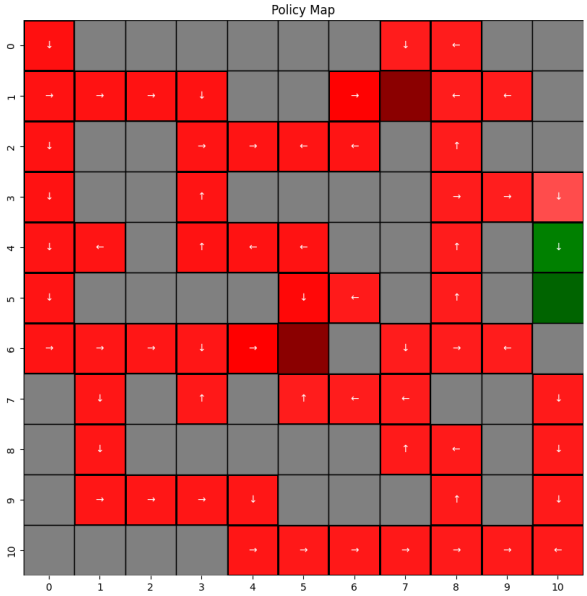


Figure 130: epsilon = 0.5 for 1000 episode

(a) utility values (b) policies

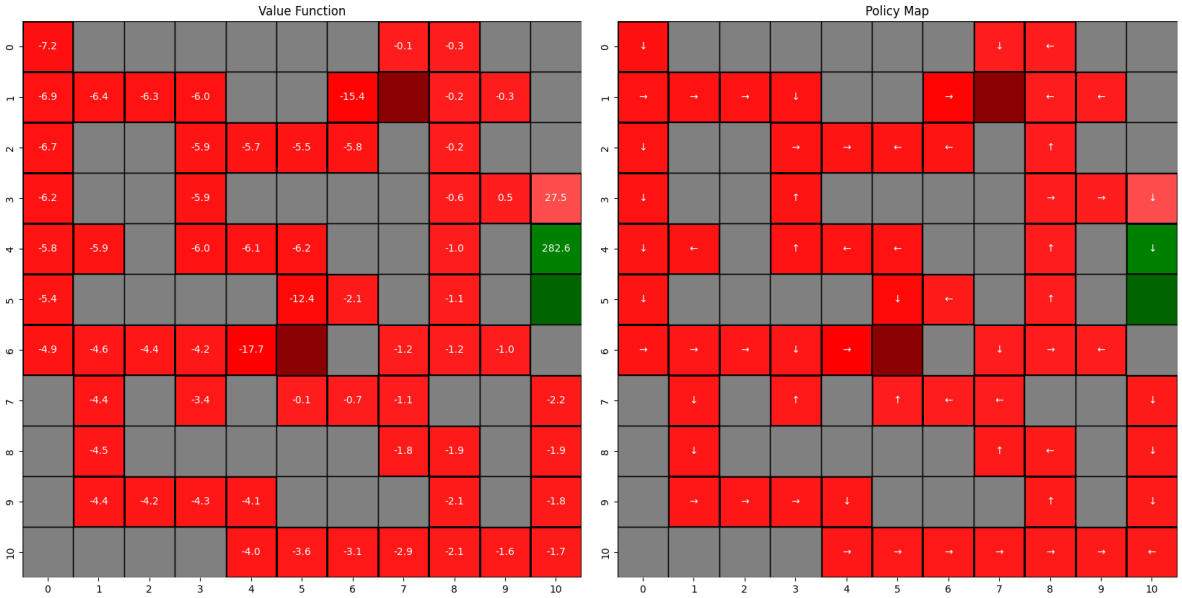


Figure 131: epsilon = 0.5 for 5000 episode

(a) utility values (b) policies

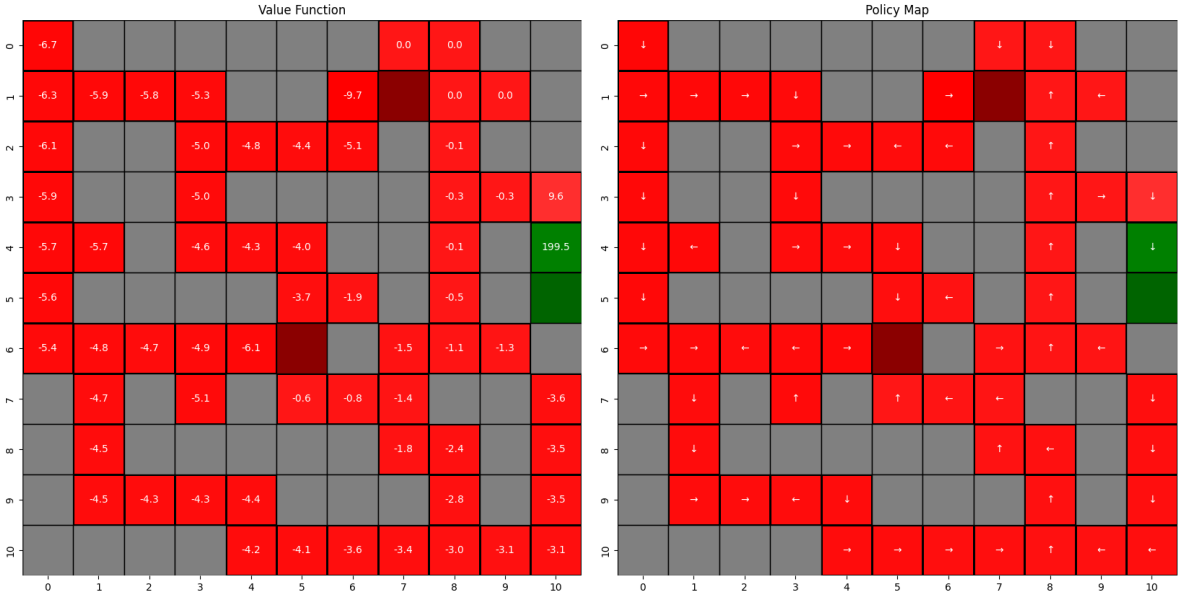


Figure 132: epsilon = 0.5 for 10000 episode

(a) utility values (b) policies

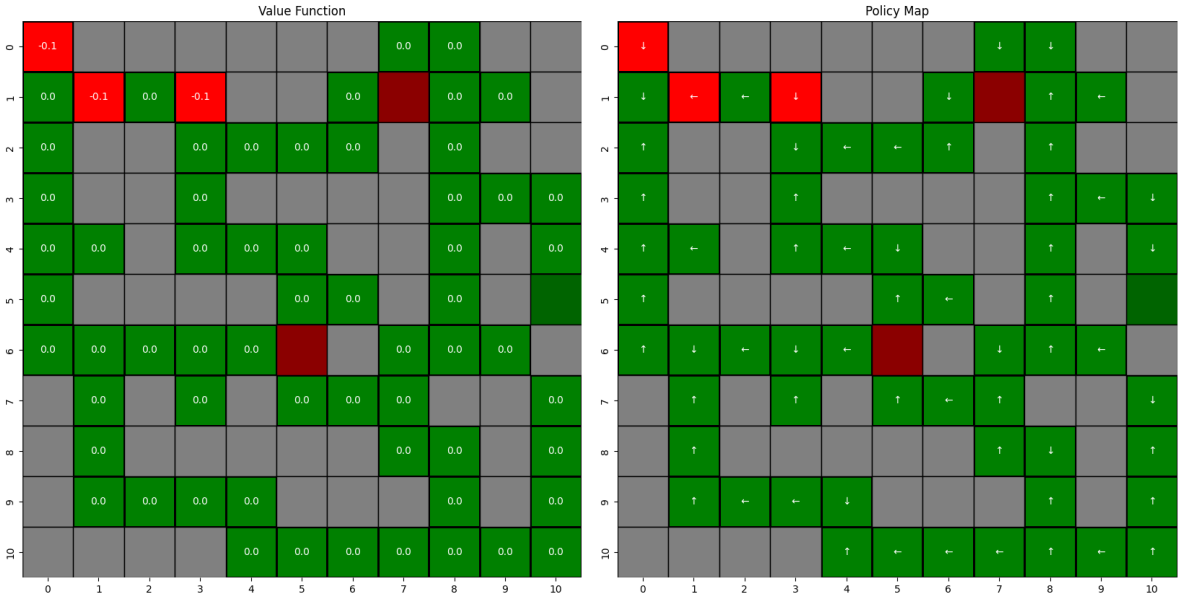


Figure 133: epsilon = 0.8 for 1 episode

(a) utility values (b) policies

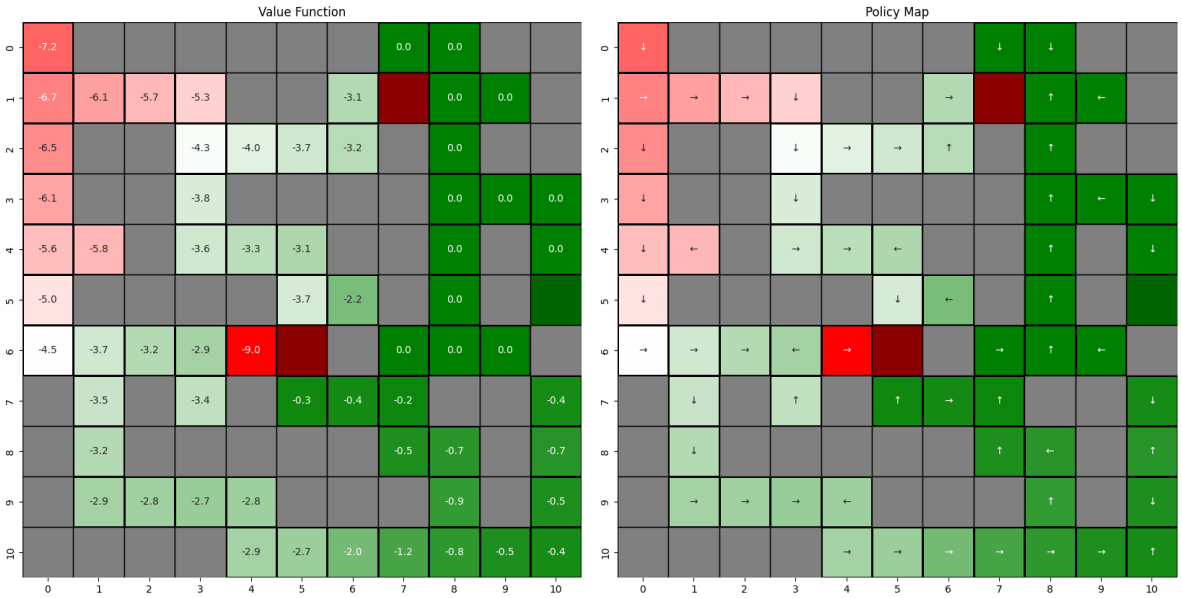
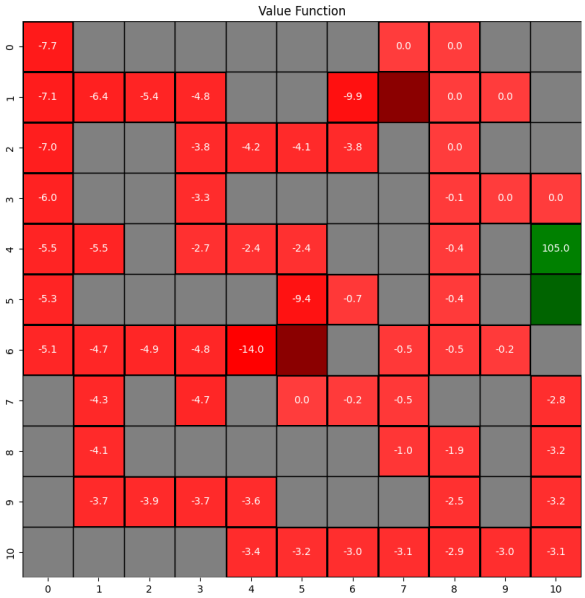


Figure 134: epsilon = 0.8 for 50 episode

(a) utility values



(b) policies

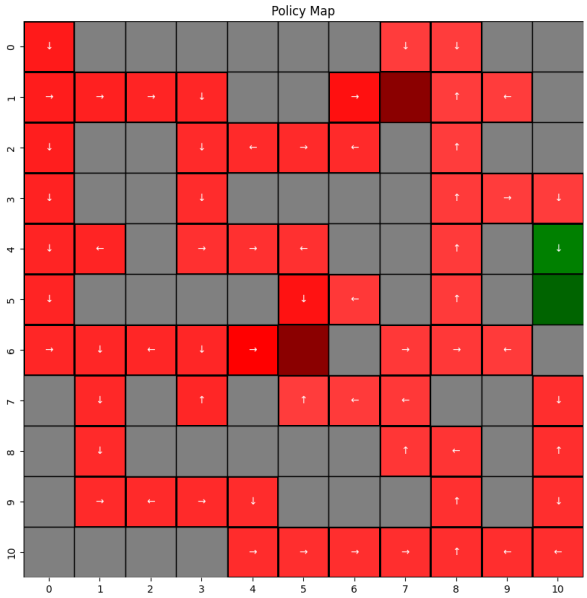
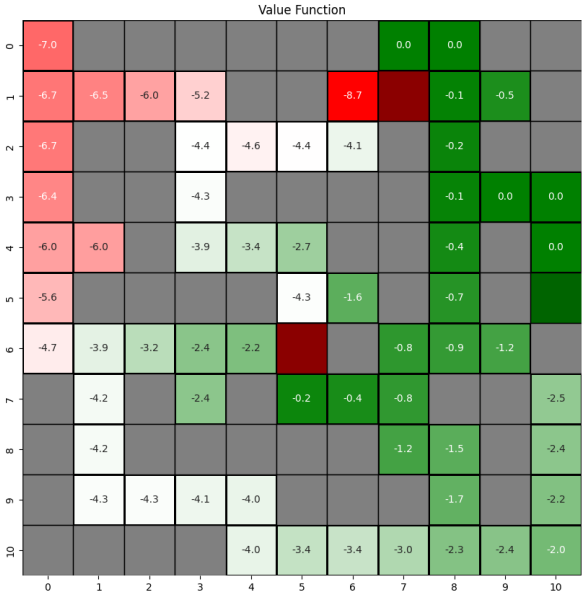


Figure 135: epsilon = 0.8 for 100 episode

(a) utility values



(b) policies

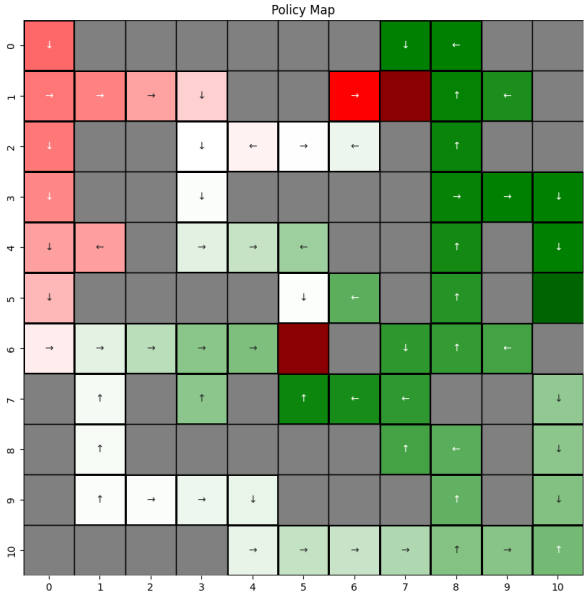
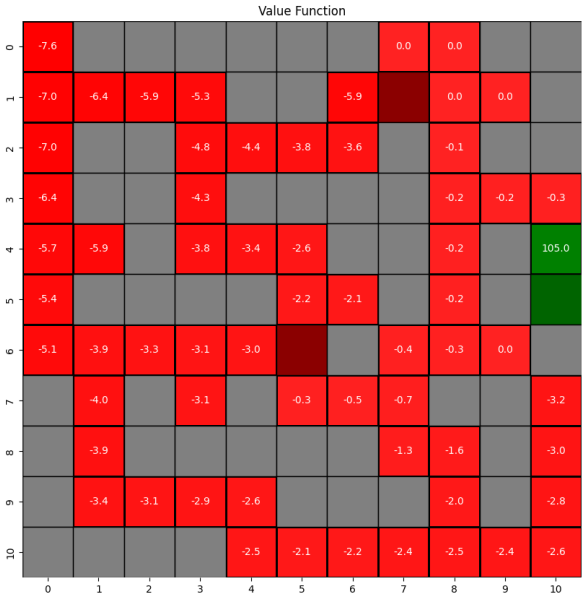


Figure 136: epsilon = 0.8 for 1000 episode

(a) utility values



(b) policies

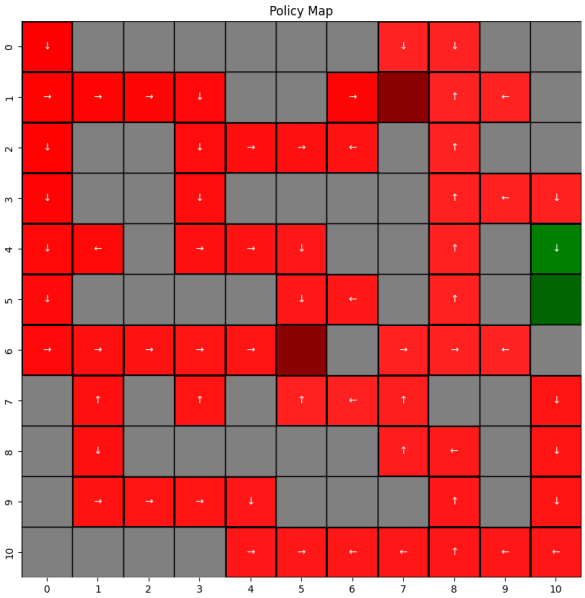
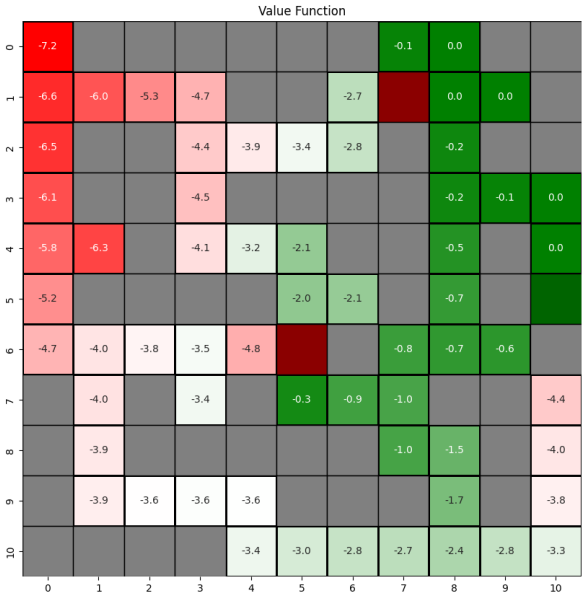


Figure 137: epsilon = 0.8 for 5000 episode

(a) utility values



(b) policies

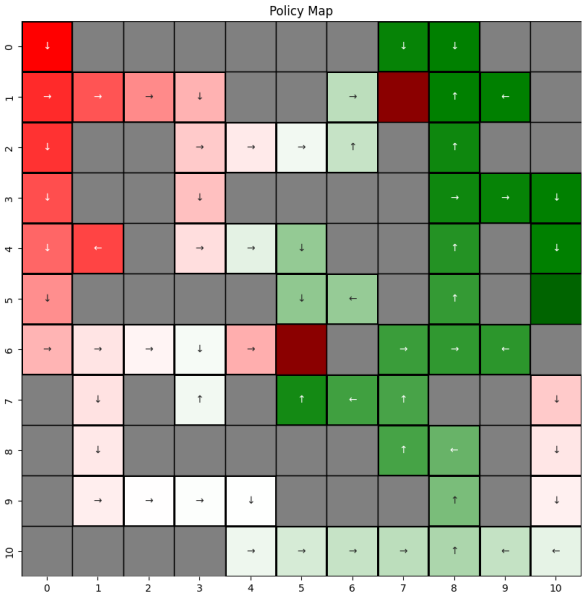
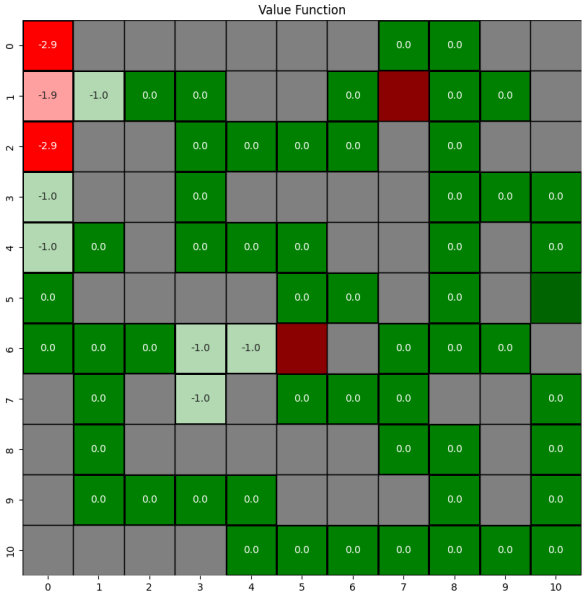


Figure 138: epsilon = 0.8 for 10000 episode

As in TD learning, epsilon increases the chance of randomized movement. While increased epsilon prevents the agent from getting stuck and detrimental fast convergence of states, it lowers the logical movement capability of the agent, which eventually becomes detrimental after some point.

Effect of gamma in Q learning

(a) utility values



(b) policies

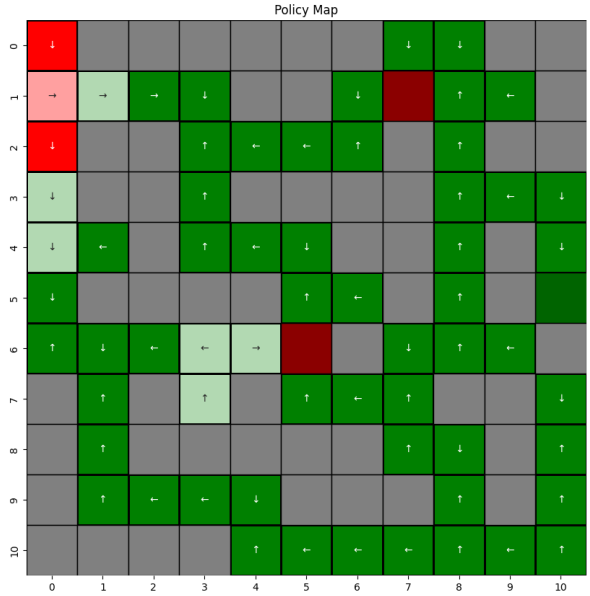
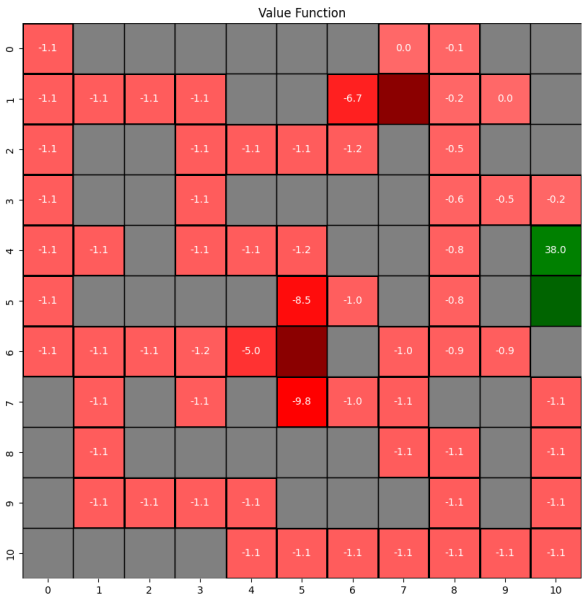


Figure 139: gamma = 0.1 for 1 episode

(a) utility values



(b) policies

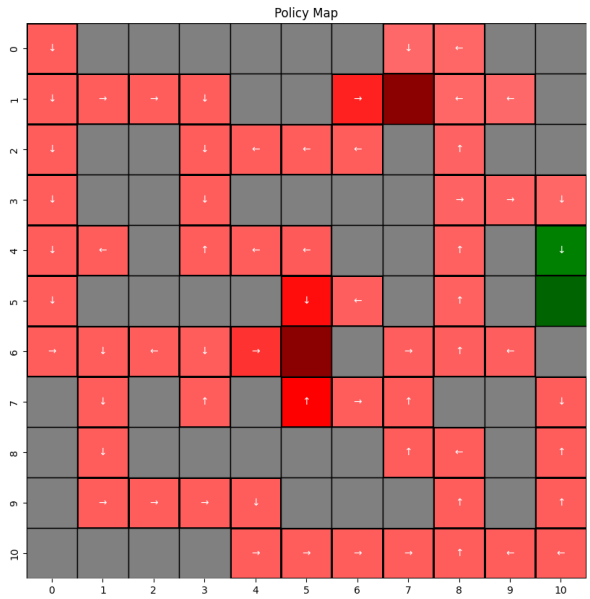
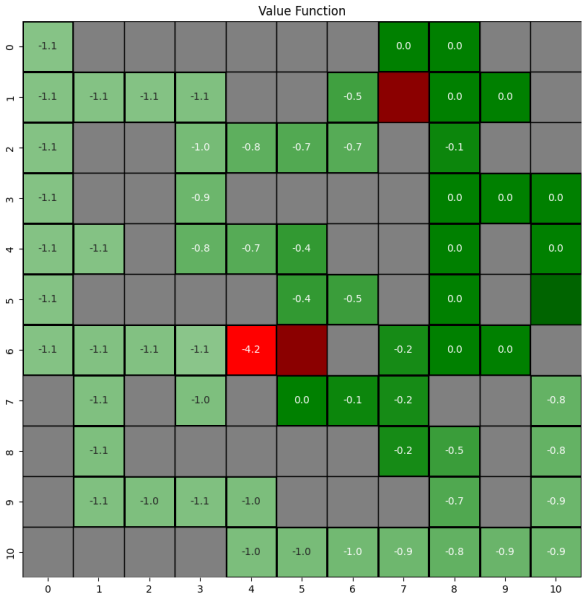


Figure 140: gamma = 0.1 for 50 episode

(a) utility values



(b) policies

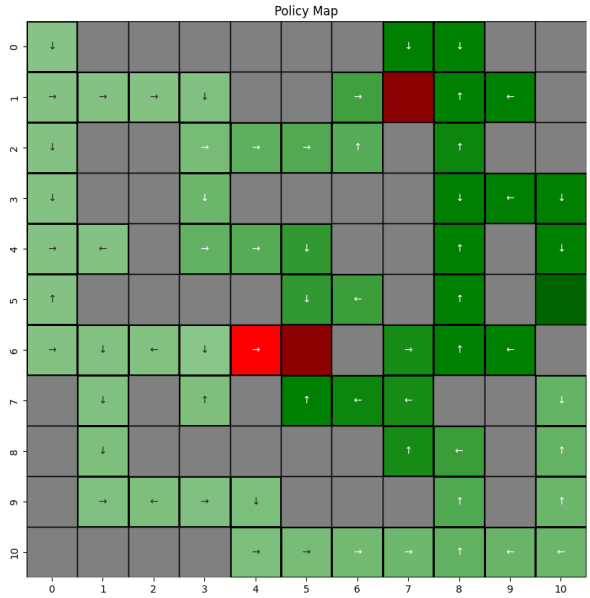
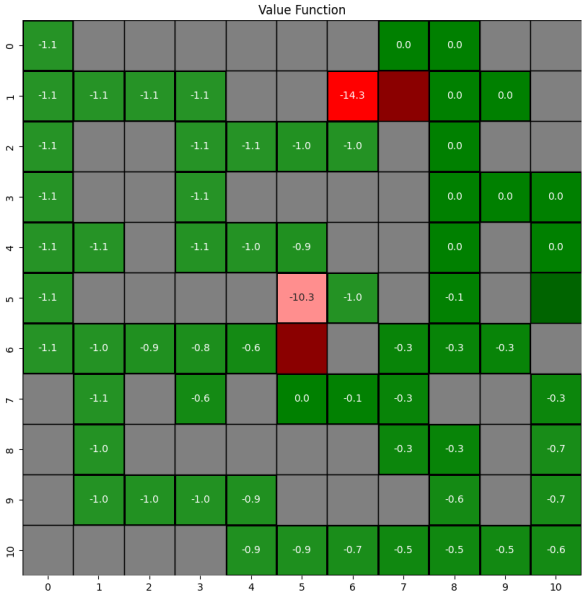


Figure 141: gamma = 0.1 for 100 episode

(a) utility values



(b) policies

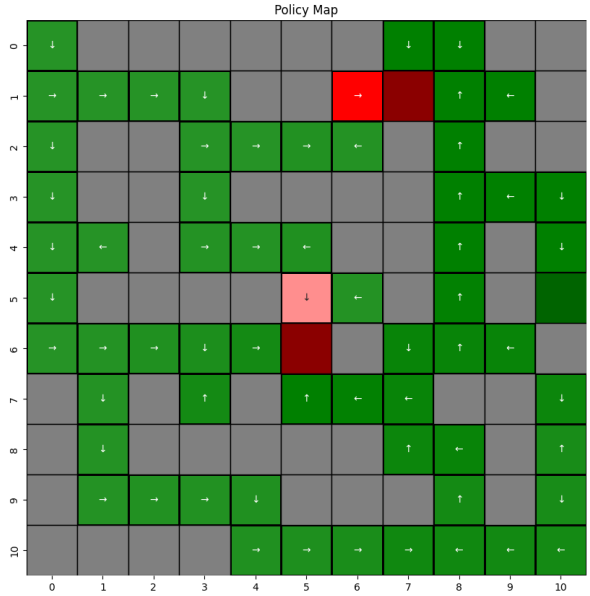
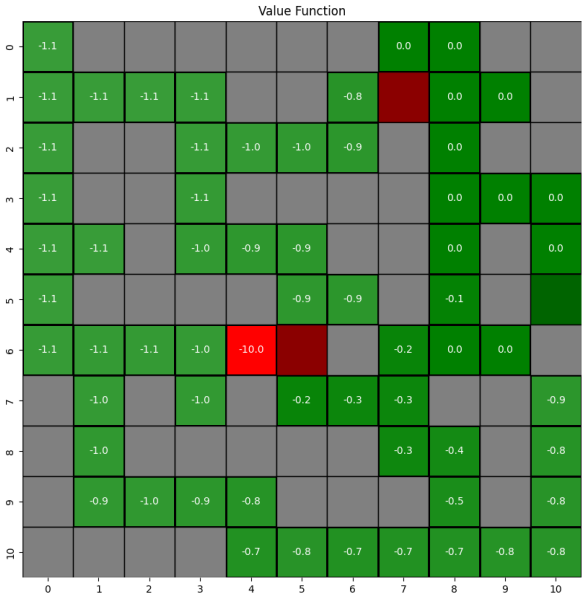


Figure 142: gamma = 0.1 for 1000 episode

(a) utility values



(b) policies

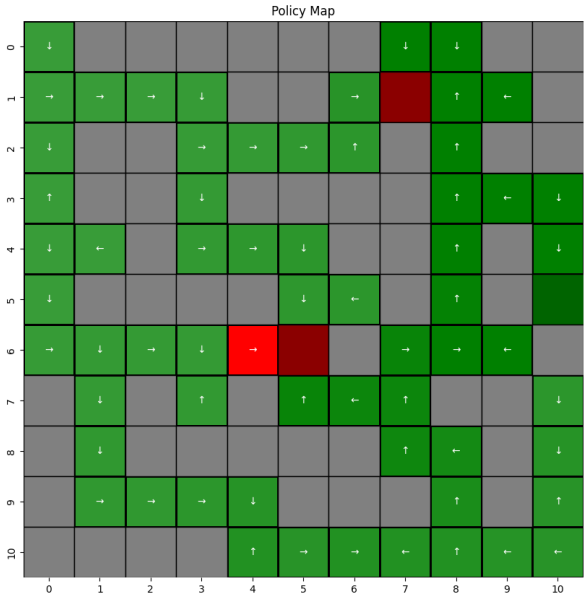
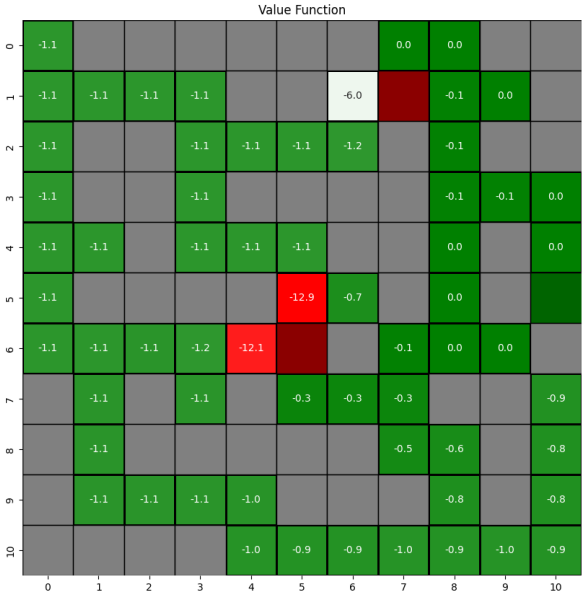


Figure 143: gamma = 0.1 for 5000 episode

(a) utility values



(b) policies

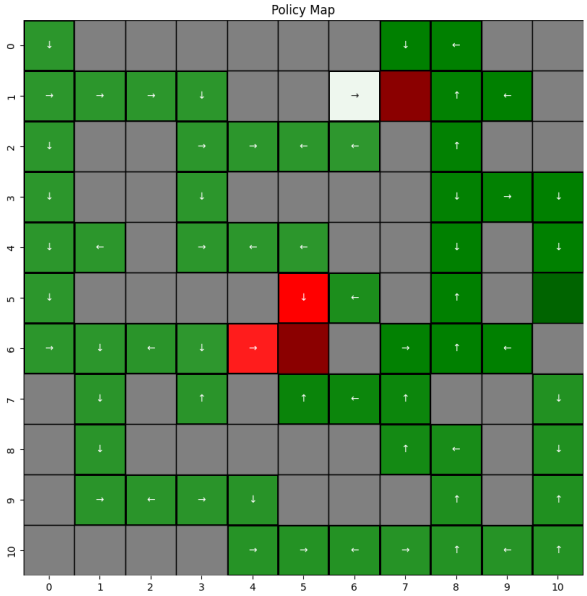


Figure 144: gamma = 0.1 for 10000 episode

(a) utility values (b) policies

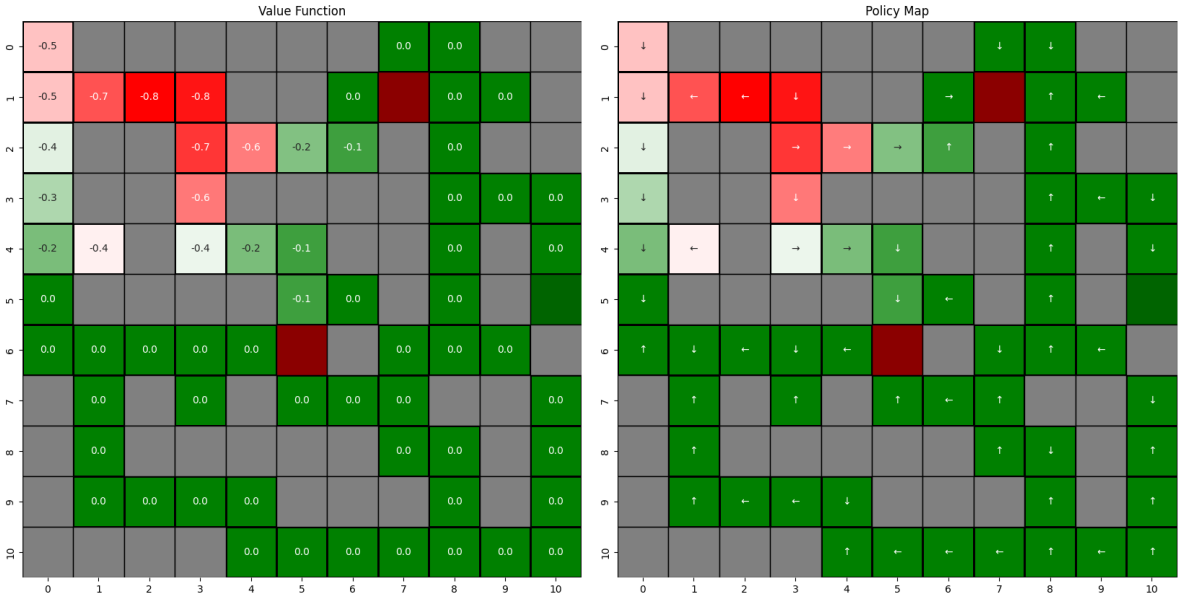


Figure 145: gamma = 0.25 for 1 episode

(a) utility values (b) policies

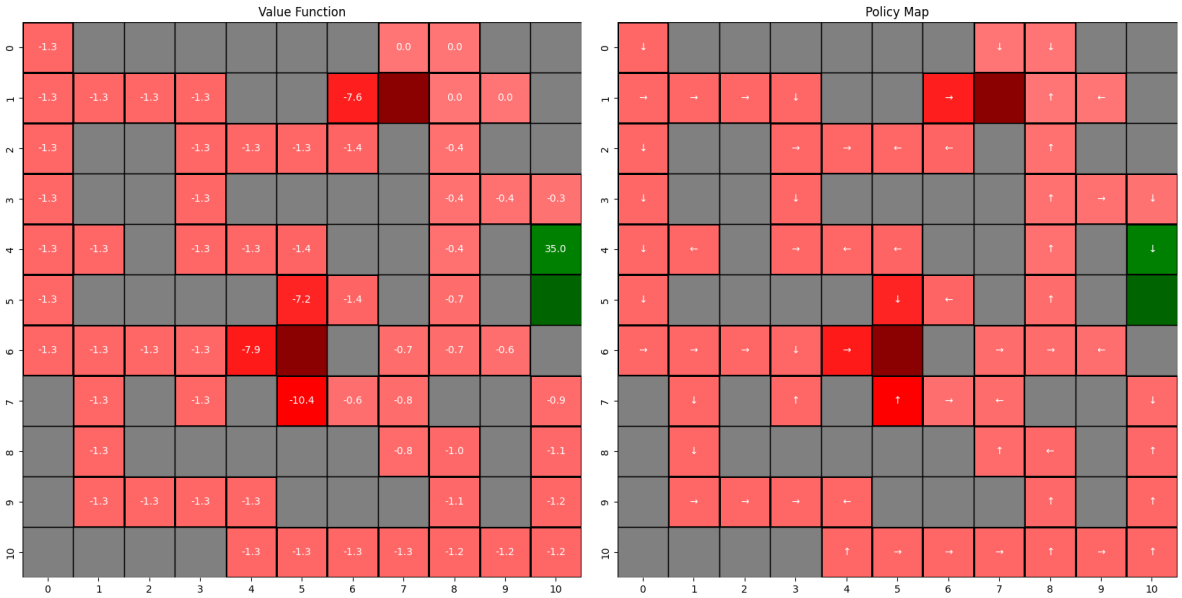


Figure 146: gamma = 0.25 for 50 episode

(a) utility values (b) policies

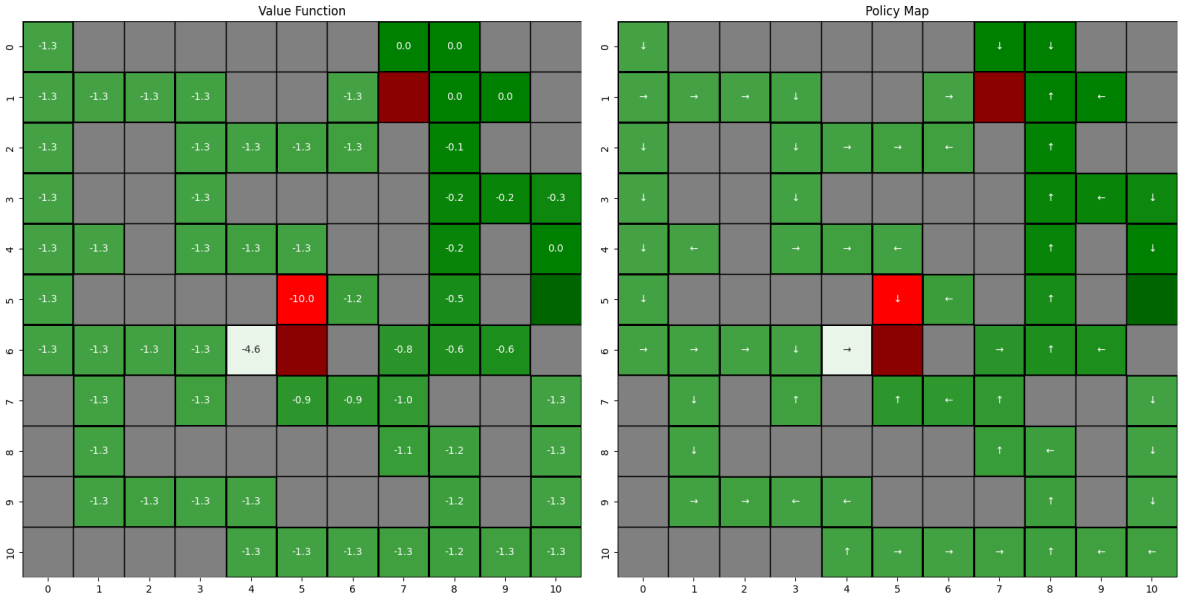


Figure 147: gamma = 0.25 for 100 episode

(a) utility values (b) policies

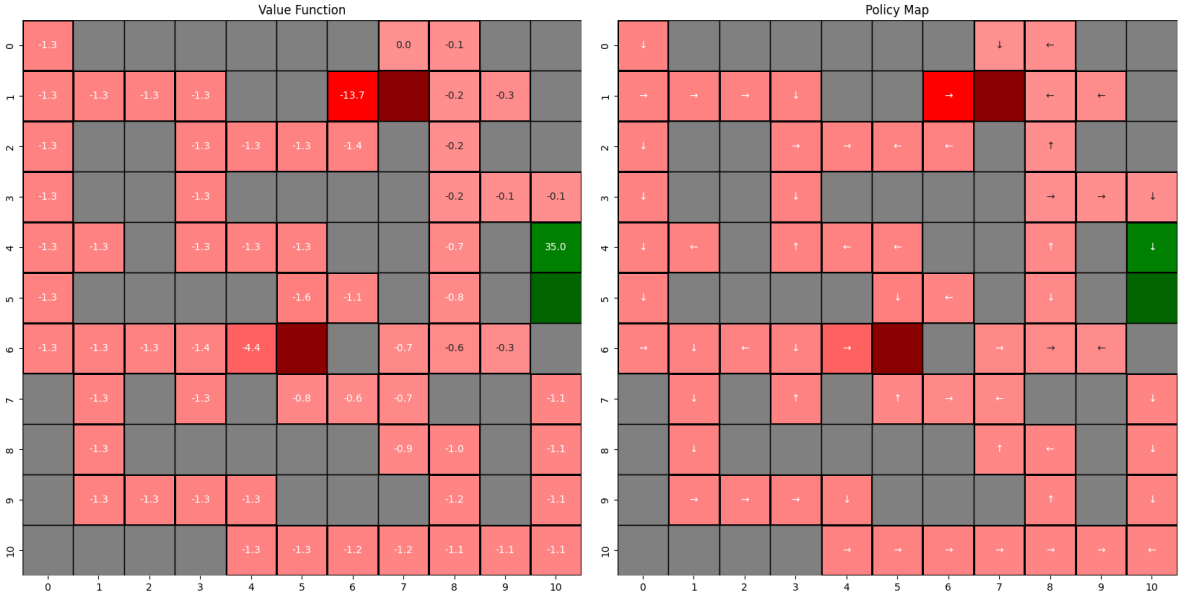
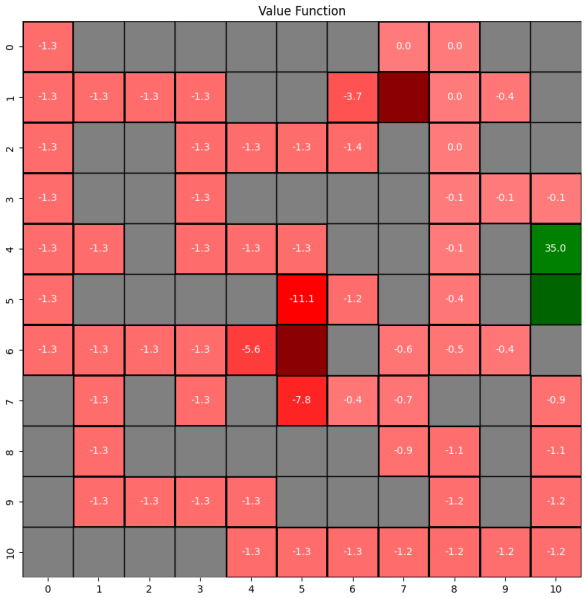


Figure 148: gamma = 0.25 for 1000 episode

(a) utility values



(b) policies

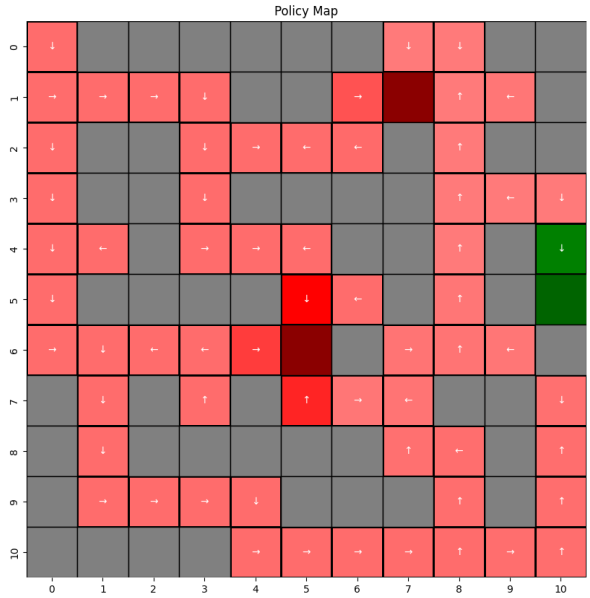
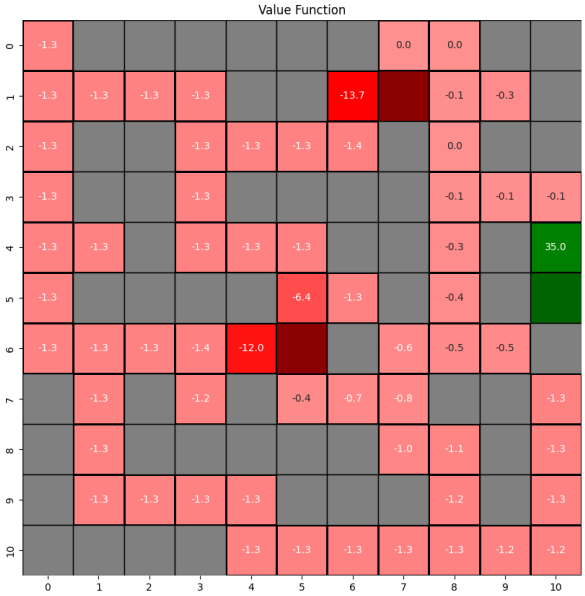


Figure 149: gamma = 0.25 for 5000 episode

(a) utility values



(b) policies

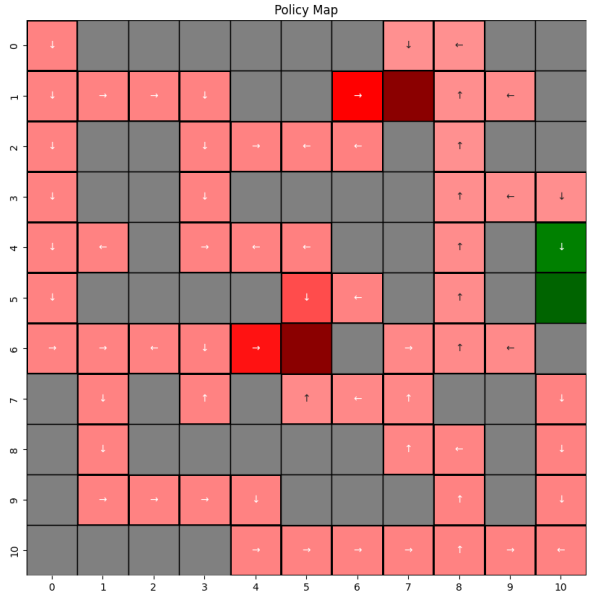


Figure 150: gamma = 0.25 for 10000 episode

(a) utility values (b) policies

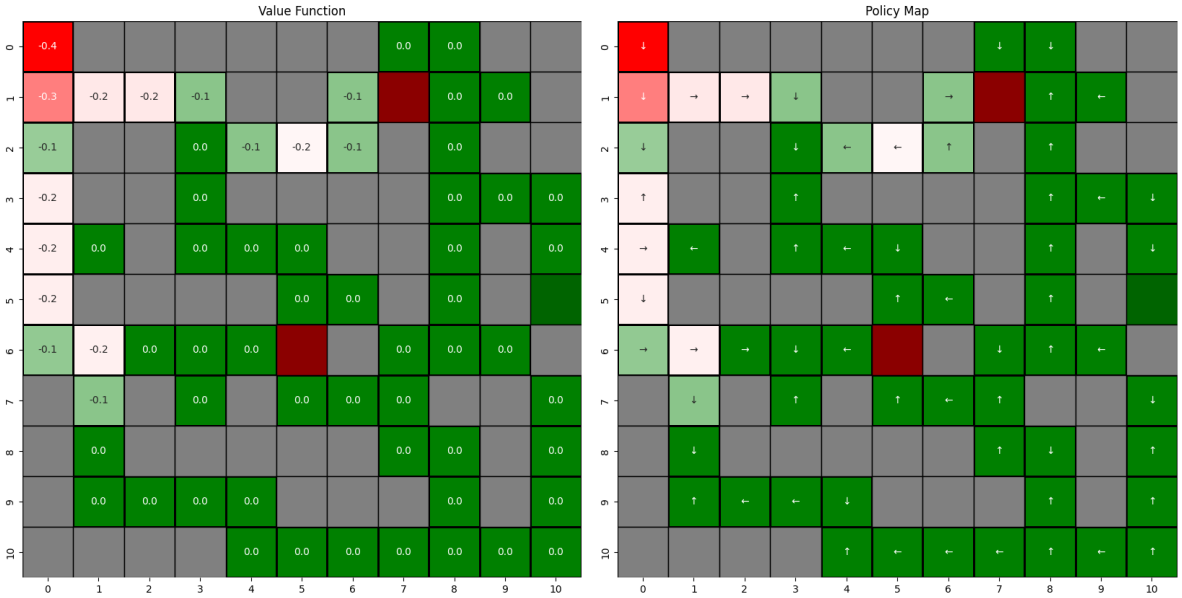


Figure 151: gamma = 0.5 for 1 episode

(a) utility values (b) policies

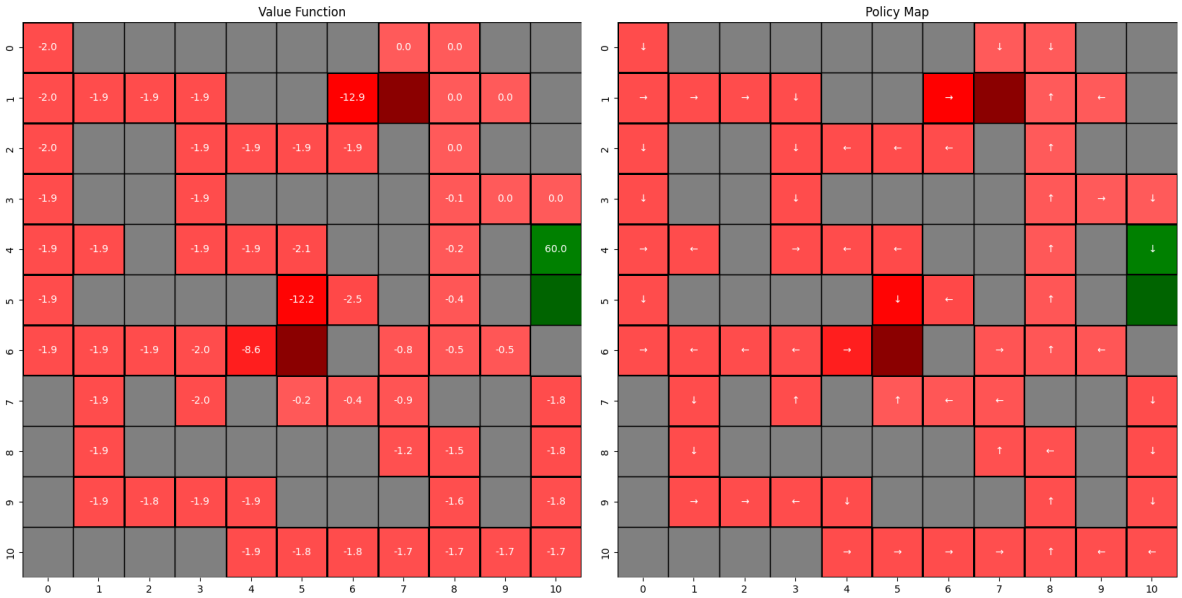
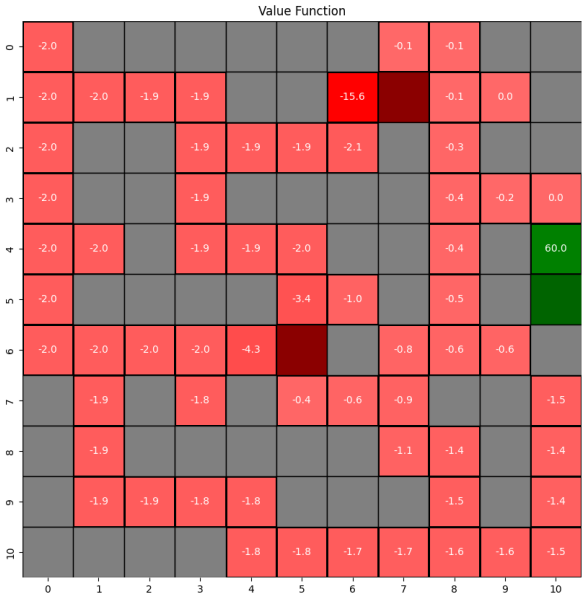


Figure 152: gamma = 0.5 for 50 episode

(a) utility values



(b) policies

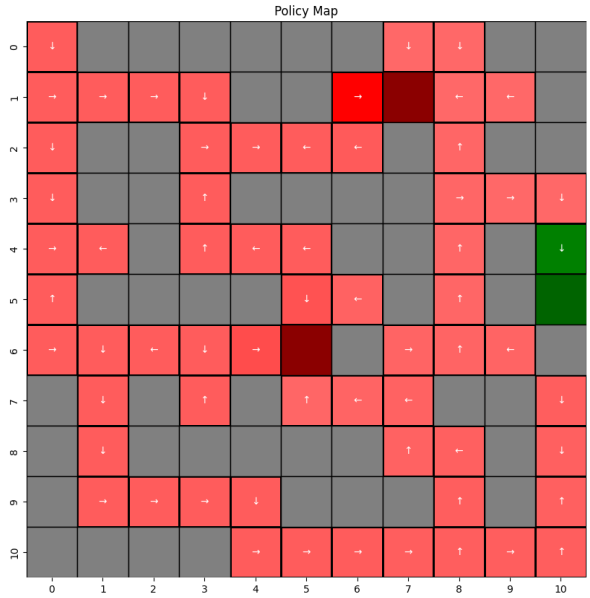
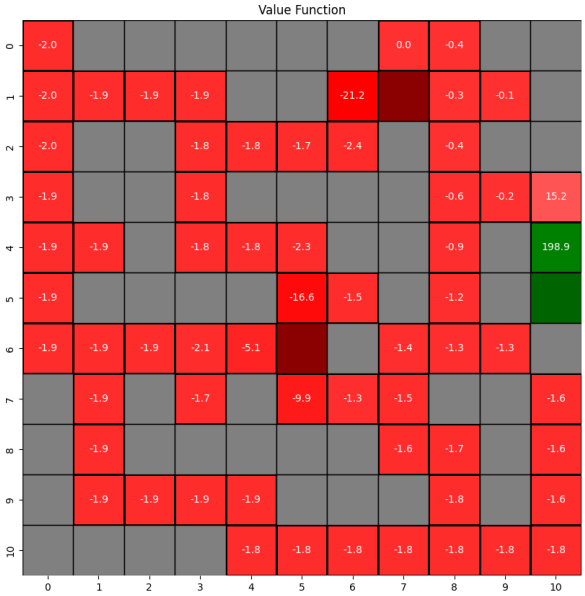


Figure 153: gamma = 0.5 for 100 episode

(a) utility values



(b) policies

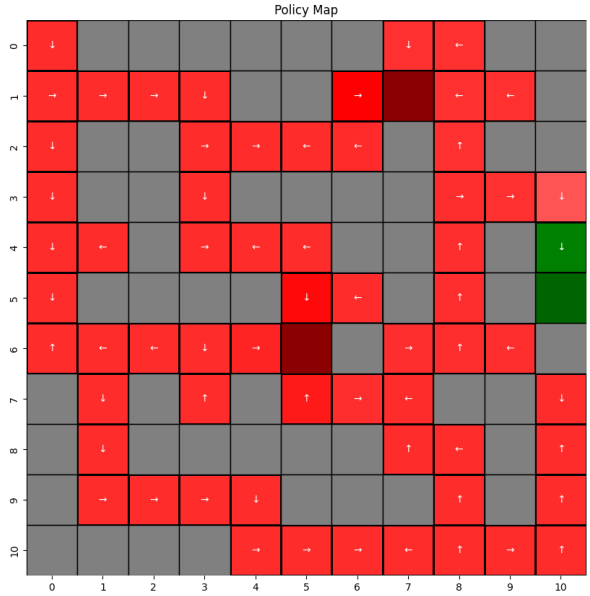
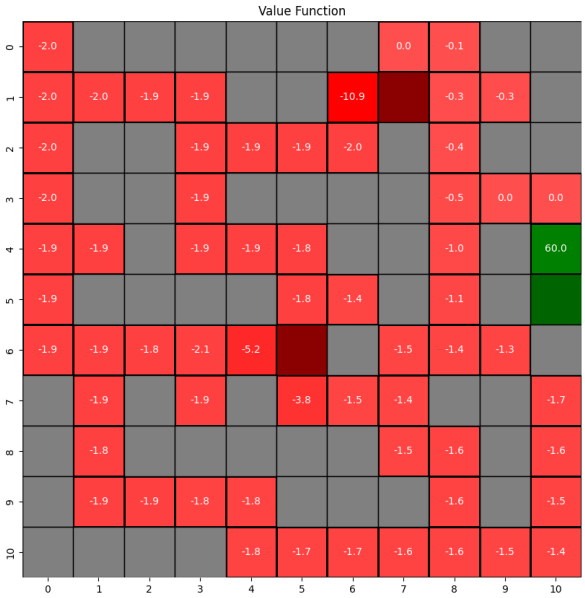


Figure 154: gamma = 0.5 for 1000 episode

(a) utility values



(b) policies

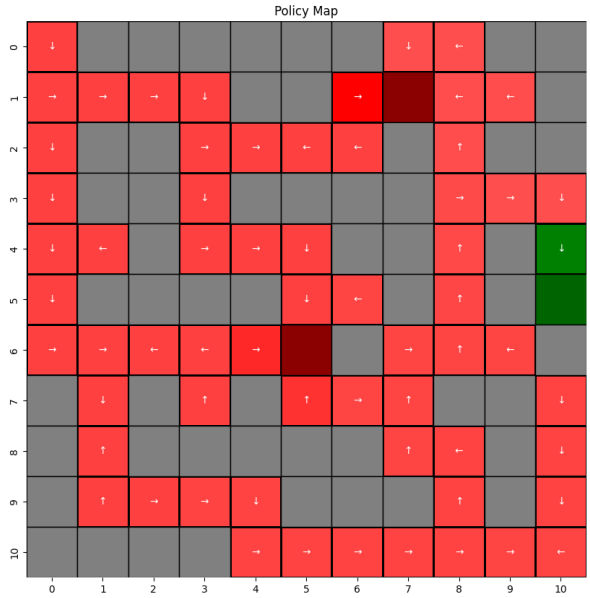
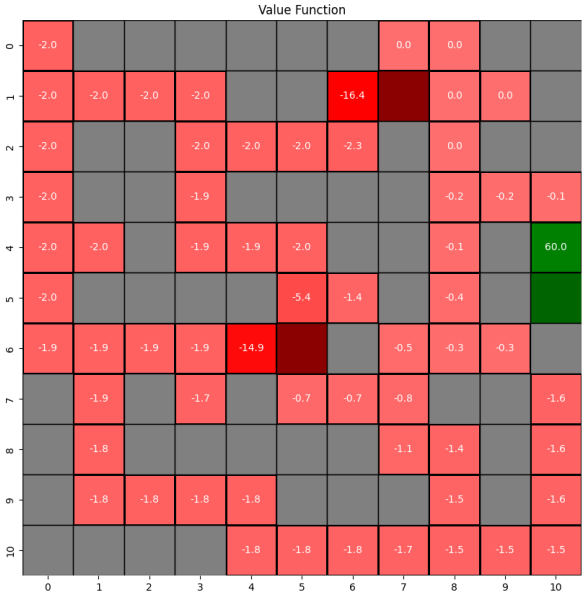


Figure 155: gamma = 0.5 for 5000 episode

(a) utility values



(b) policies

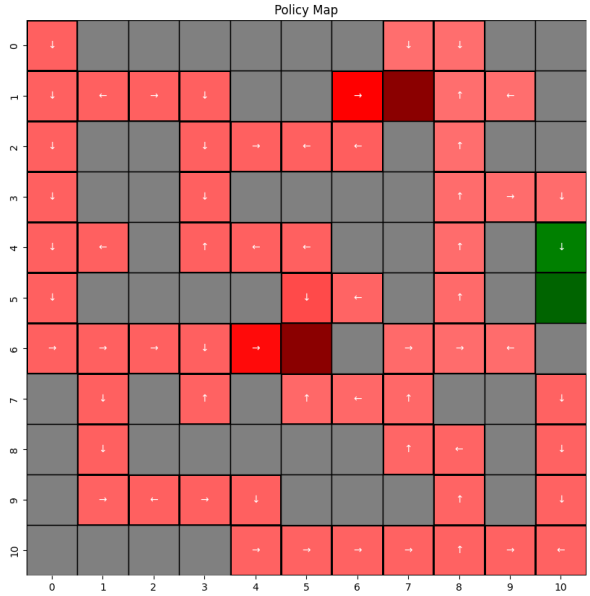


Figure 156: gamma = 0.5 for 10000 episode

(a) utility values (b) policies

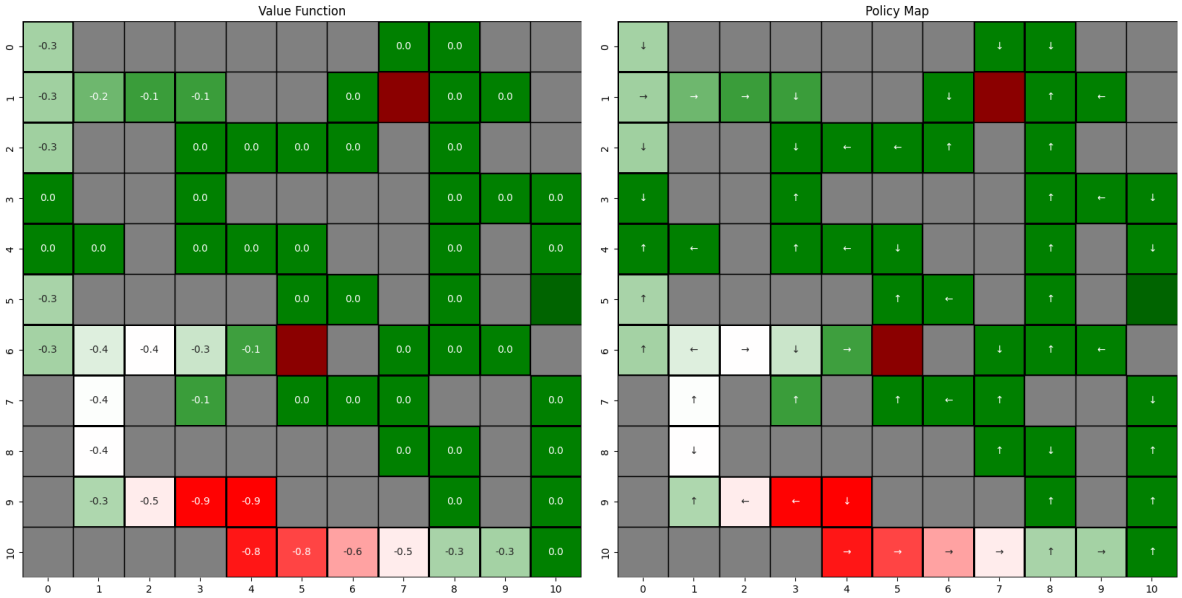


Figure 157: gamma = 0.75 for 1 episode

(a) utility values (b) policies

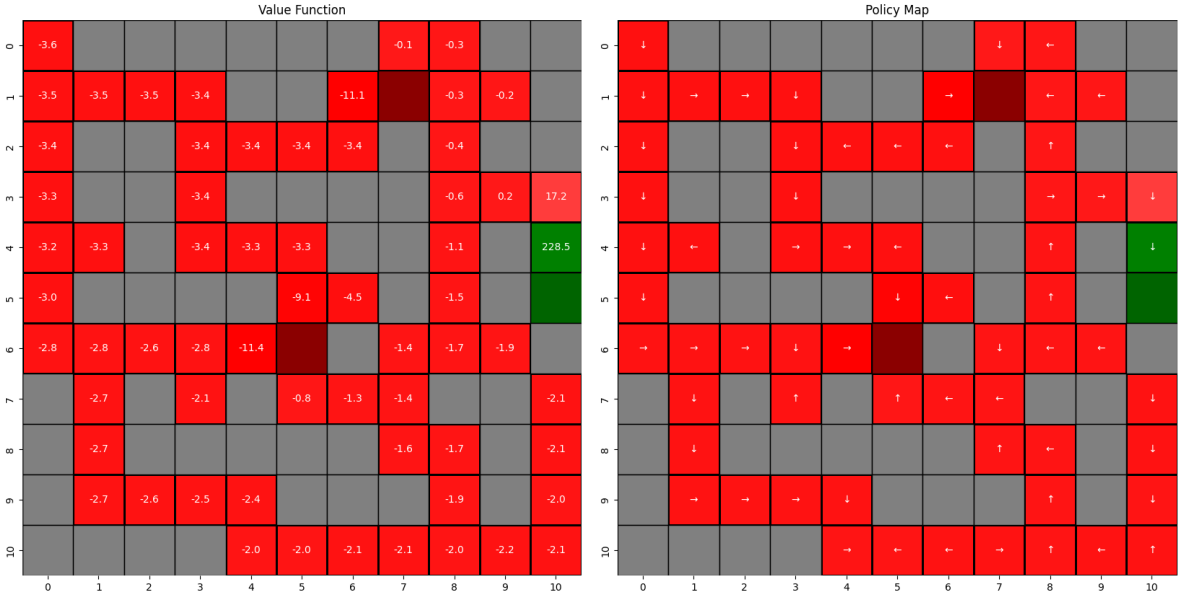
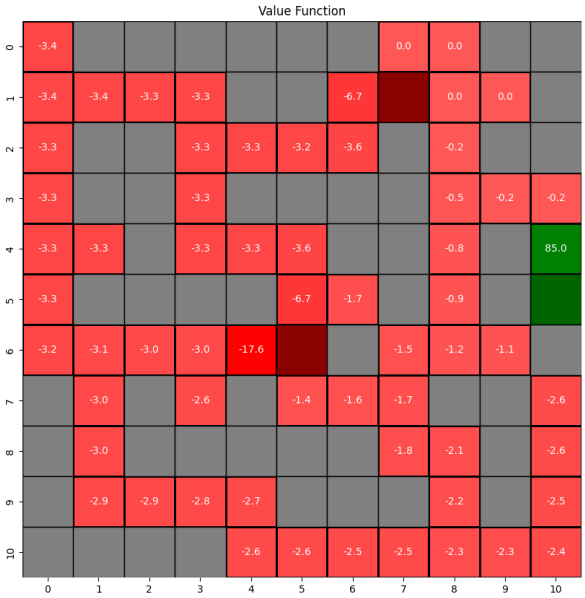


Figure 158: gamma = 0.75 for 50 episode

(a) utility values



(b) policies

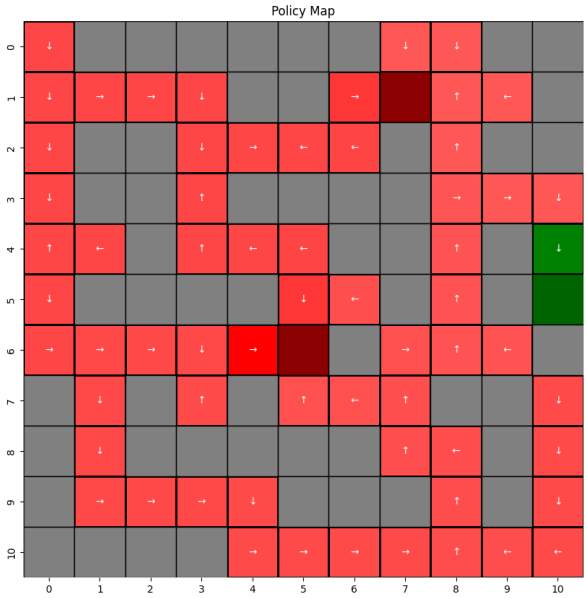
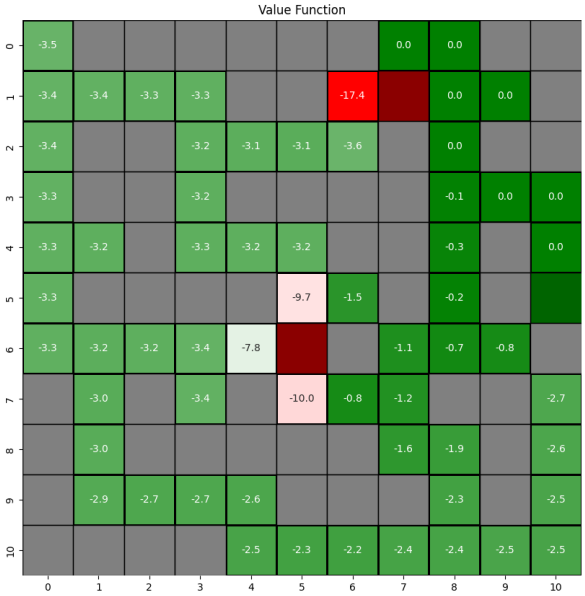


Figure 159: gamma = 0.75 for 100 episode

(a) utility values



(b) policies

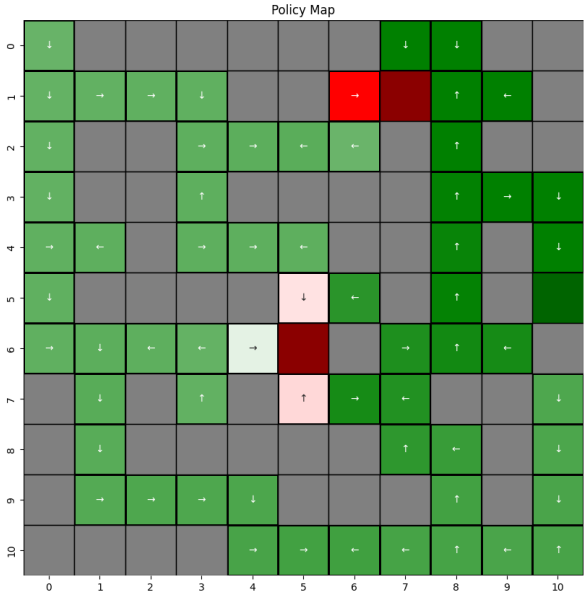


Figure 160: gamma = 0.75 for 1000 episode

(a) utility values (b) policies

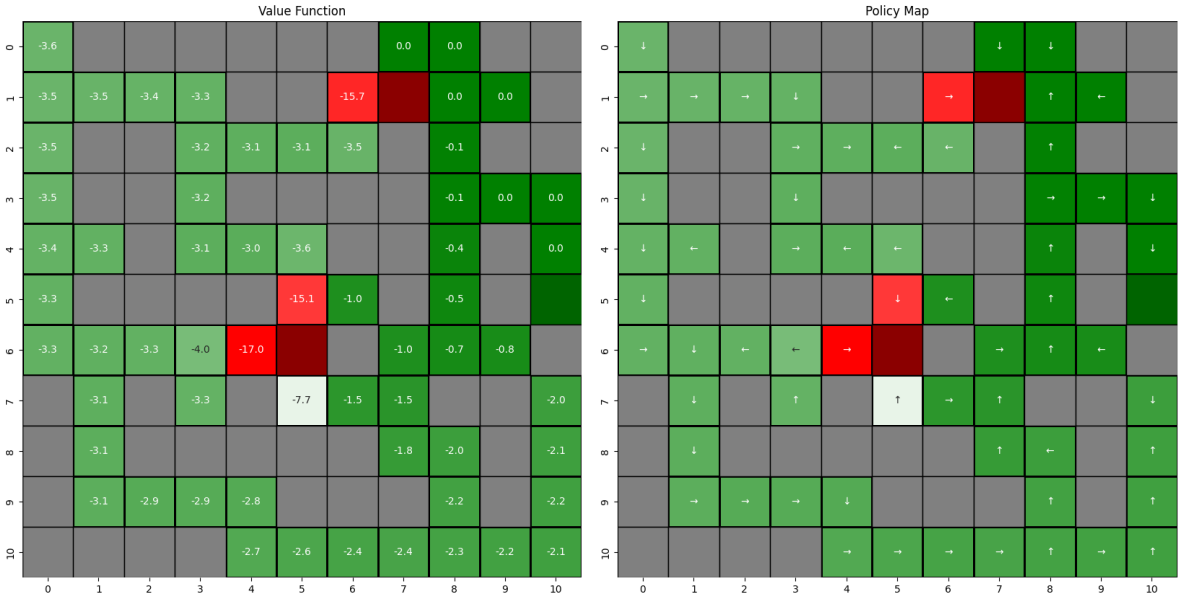


Figure 161: gamma = 0.75 for 5000 episode

(a) utility values (b) policies

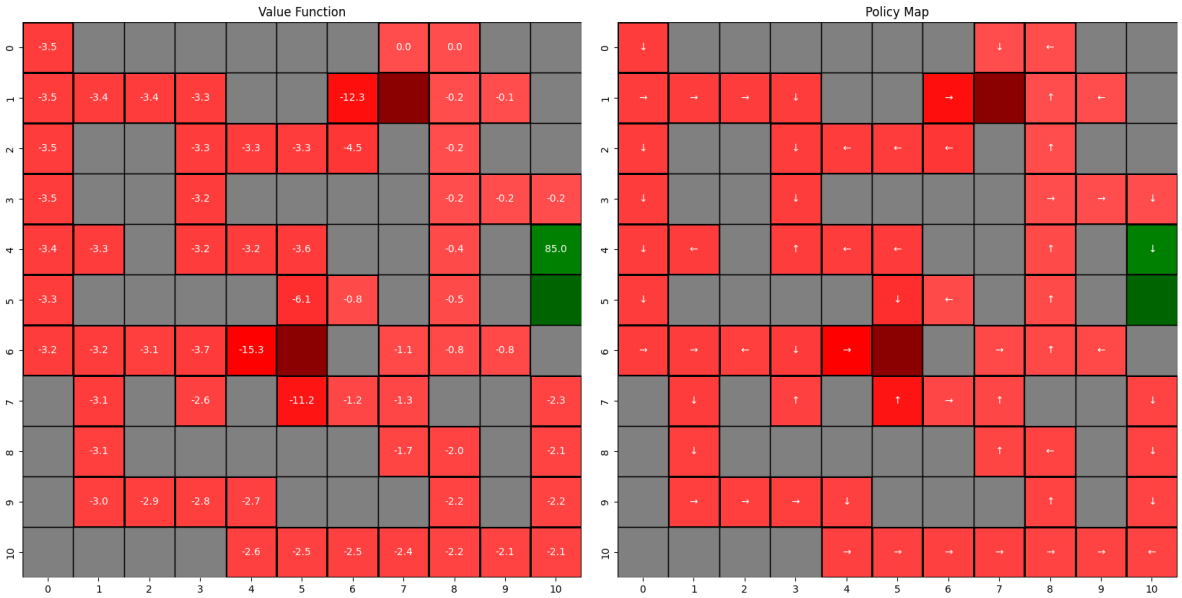


Figure 162: gamma = 0.75 for 10000 episode

(a) utility values (b) policies

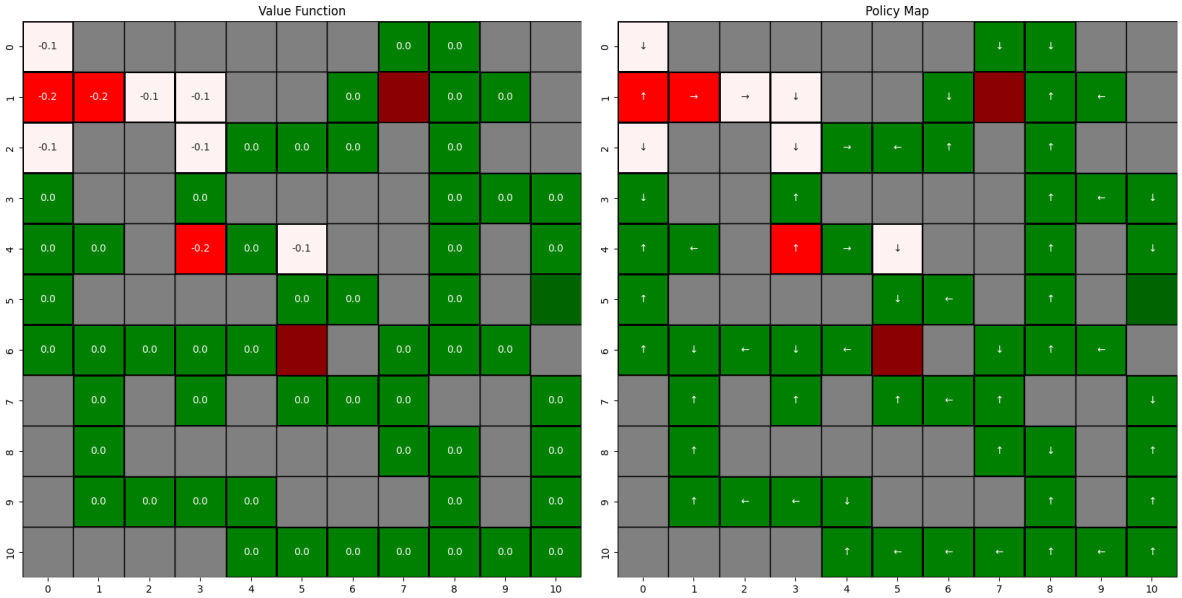


Figure 163: gamma = 0.95 for 1 episode

(a) utility values (b) policies

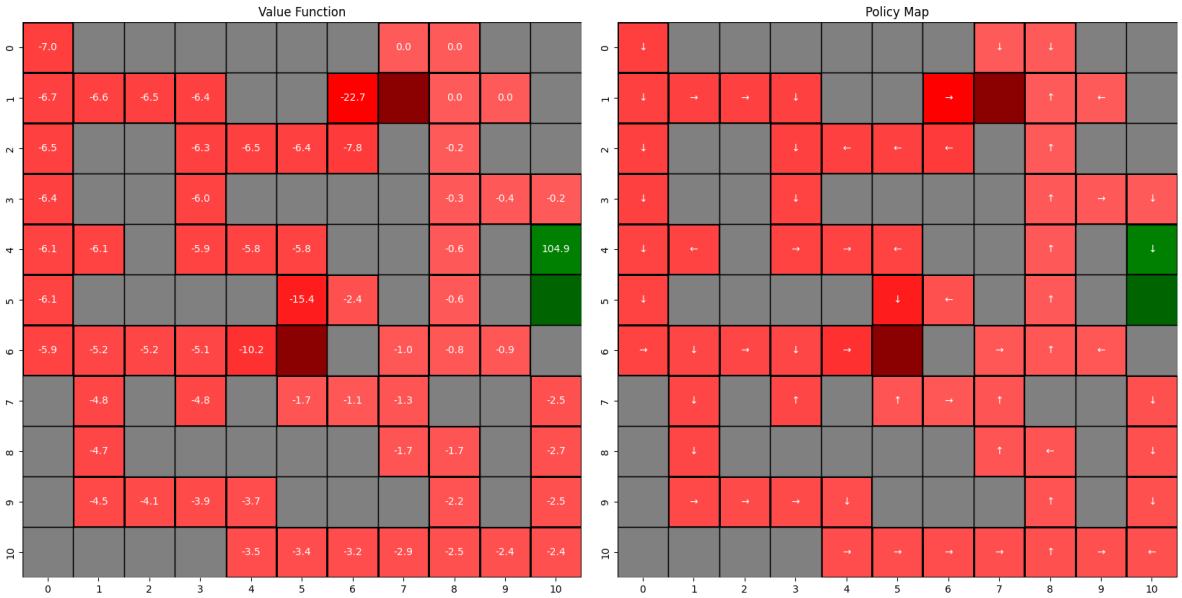
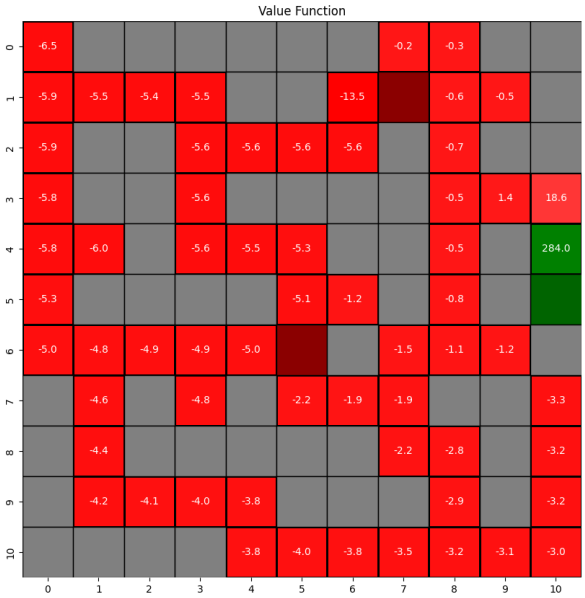


Figure 164: gamma = 0.95 for 50 episode

(a) utility values



(b) policies

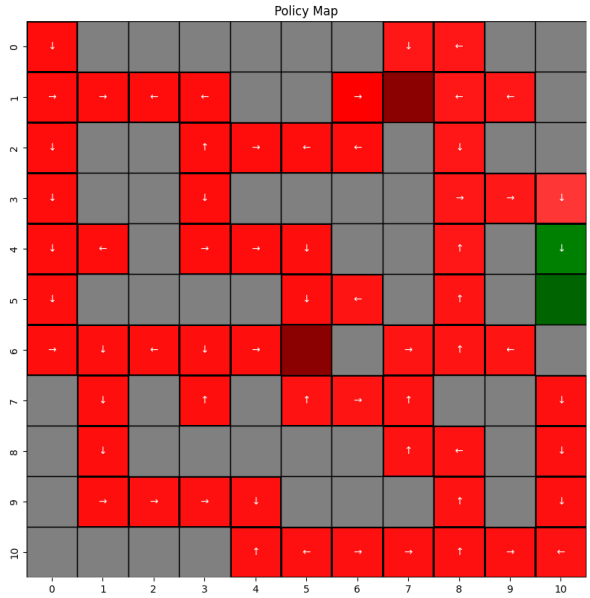
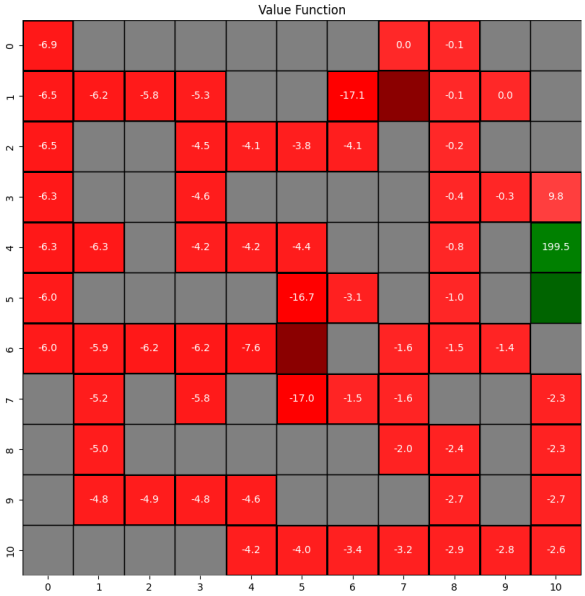


Figure 165: gamma = 0.95 for 100 episode

(a) utility values



(b) policies

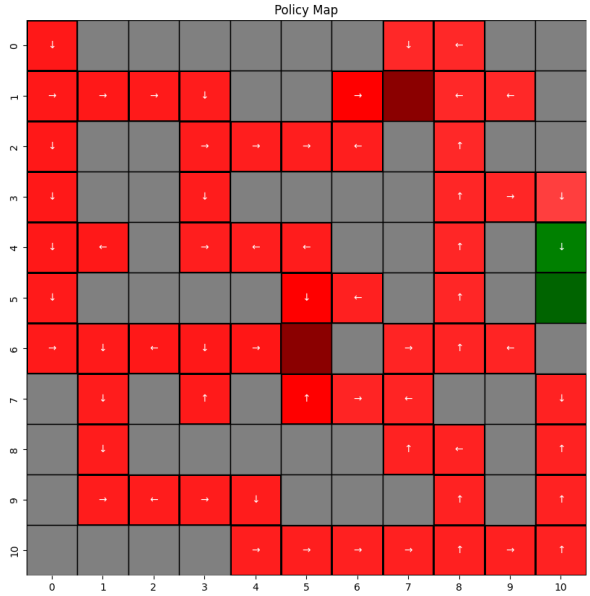
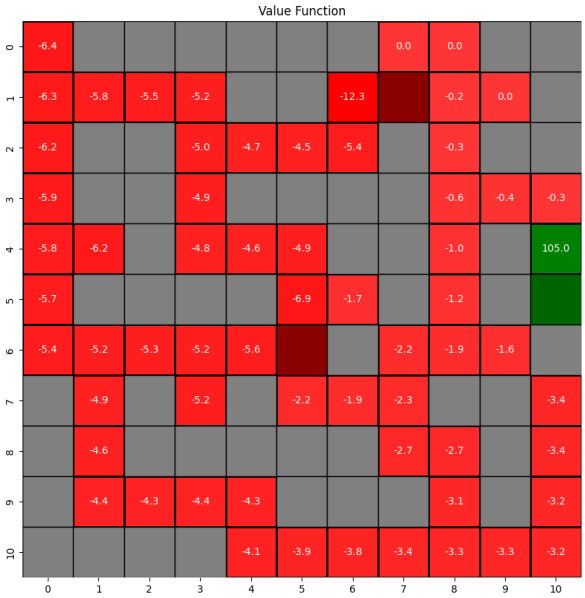


Figure 166: gamma = 0.95 for 1000 episode

(a) utility values



(b) policies

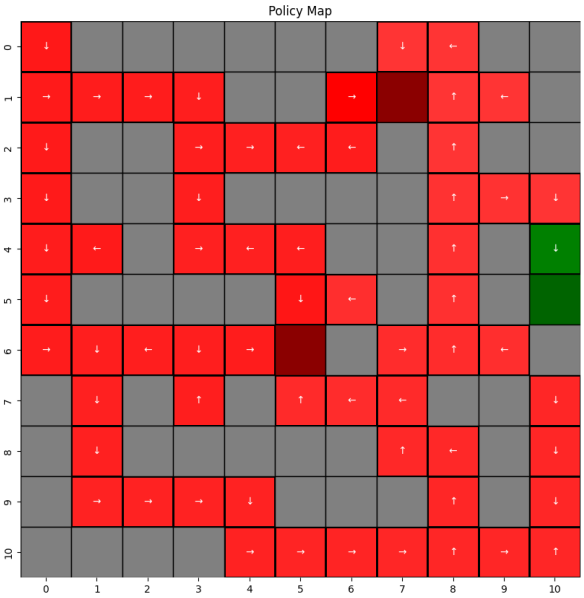
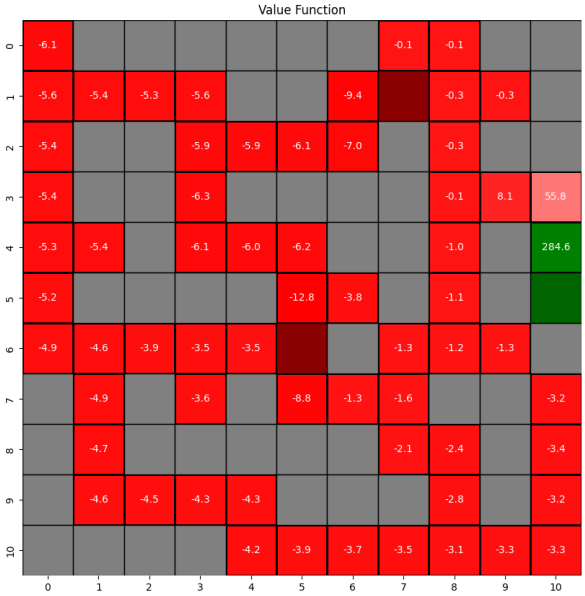


Figure 167: gamma = 0.95 for 5000 episode

(a) utility values



(b) policies

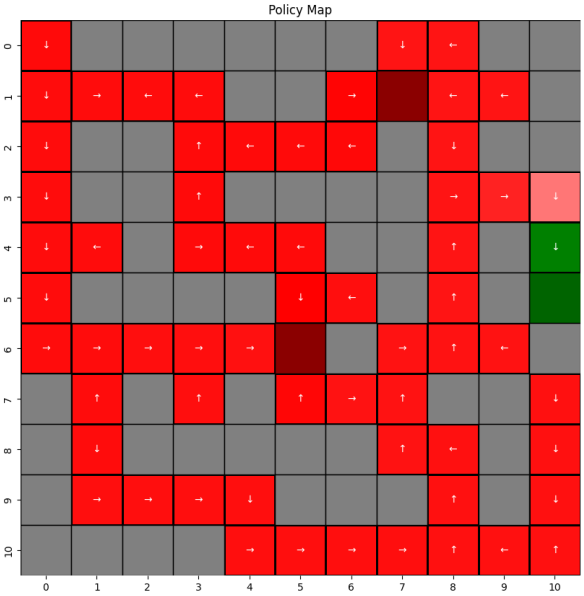


Figure 168: gamma = 0.95 for 10000 episode

In this section, there weren't many successful results, which were 0.1 gamma, 50 episodes, and 0.25 gamma, 10000 episodes. Gamma value increase increases the effect of the goal and trap states

1 Discussions

- Reward function assigns penalties to undesired states and rewards to the desired states. Implementing this with an appropriate action-choosing algorithm creates a logically moving agent. Transition probabilities give stochastic qualities to the agent. When these two combined and balance between exploitation and exploration achieved a functional agent is obtained.
- As the number of episodes increases state values converge to specific values under the effect of the trap and goal states high-absolute-value utilities and transition frequencies.
- For different transition probabilities and hyper-parameters convergence time has changed or sometimes never happened.
- TD learning was relatively sensitive to the changes in hyper-parameters epsilon has prevented the agent from stuck while also making convergence harder. alpha was also directly proportional to the convergence time, as it can be considered a learning state. but very high alpha diminished the effect of the goal state over other states. And gamma, too, was related to convergence speed.
- Assigning low utility to obstacles was a necessity. also, the traps being close to the beginning states led to the agent being stuck under some circumstances. For the cases where the agent gets stuck, I added a movement cap of 1000 in order to prevent a long training process.
- Q learning(without movement cap) tends to be faster than TD learning. This was due to Q-values based on the maximum future Q-value without estimating state utilities as in TD learning. and Q learning, in theory, should be less sensitive to hyperparameters.
- As discussed several times above, increasing epsilon puts more emphasis on the exploration, which prevents the agent from getting stuck but also has a negative effect on convergence after some point.
- TD and Q learning differ in their mechanisms and rules. Q learning directly estimates action values and learns an optimal action-value function. TD learning can be preferable when state utilities are more relevant. The choice between 2 methods depends on the implementation.
- Trap states being closer to the beginning states creates a risk for the agent being stuck, while the goal state being far away significantly increases the difficulty of the situation.
- multiple-step learning might increase the robustness.

2 codes

2.1 Helper Functions

```
def tuple_sum(a, b): # Element-wise summation of two tuples
  return tuple(map(sum, zip(a, b)))

def tuple_subtract(a, b):  # Element-wise subtraction of two tuples
  res = tuple(map(lambda i, j: i - j, a, b))
  return res

def tuple_swap(a): # swapping tuple elements
  res = tuple((a[1], a[0]))
  return res
```

2.2 Maze Implementation

```
  class MazeEnvironment:
def __init__(self):
  # Define the maze layout, rewards

  self.start_pos = (0,0) # Starting position of the agent
  self.current_pos = self.start_pos
  self.pre_pos1 = None
  self.pre_pos2 = None
  self.pre_pos3 = None
  self.pre_pos4 = None
  self.pre_pos5 = None
  self.state_penalty = -1
  self.trap_penalty = -100
  self.goal_reward = 100

  self.actions = {0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)}  # action space (up,
                                                down, left, right)

  self.maze_layout = np.array([ # Maze layout 0:open 1:obstacle 2:trap 3:goal
     [0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1],
     [0, 0, 0, 0, 1, 1, 0, 2, 0, 0, 1],
     [0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0],
     [0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0],
     [0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 3],
     [0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 1],
     [1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0],
     [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0],
     [1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0],
     [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
 ])

def calculate_reward(self, s_type): # for extracting reqard
  if s_type==0:
    return self.state_penalty
  elif s_type==2:
    return self.trap_penalty
  else:
    return self.goal_reward

def is_valid(self, pos):
      # Check if position is within the maze bounds and not an obstacle
      return (0 <= pos[0] < self.maze_layout.shape[0]) and (0 <= pos[1] < self.
                                                maze_layout.shape[1]) and self.
                                                maze_layout[pos[0], pos[1]] != 1

def reset(self): # resets for next episode
  self.current_pos = self.start_pos
  return self.current_pos

def step(self, action): # arranges new values of states for a taken action
  opposite = tuple_subtract((0,0), action)
  perp_1 = tuple_swap(action)
  perp_2 = tuple_subtract((0,0), perp_1)
  actions = {0:action, 1:opposite, 2:perp_1, 3:perp_2}

  choices = [0, 1, 2, 3]
  probs = [0.75, 0.05, 0.1, 0.1] # chosen direction, the inverse, left of the
                                              direction, right of the direction

  choice = np.random.choice(choices, p=probs)
  new_pos = tuple_sum(self.current_pos, actions[choice])

  if self.is_valid(new_pos):
          self.current_pos = new_pos


  reward = self.calculate_reward(self.maze_layout[self.current_pos])
```

```
    return self.current_pos, reward
```

2.3 TD learning

```
 class MazeTD0(MazeEnvironment): # Inherited from MazeEnvironment
def __init__(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=5000):
  super().__init__()
  self.maze = maze
  self.alpha = alpha #Learning Rate
  self.gamma = gamma #Discount factor
  self.epsilon = epsilon #Exploration Rate
  self.episodes = episodes
  self.utility = np.zeros_like(maze.maze_layout, dtype=float) # Encourage
                                                    exploration
  #self.utility = np.ones_like(maze.maze_layout, dtype=float) * 10
  for i in range(11):
    for j in range(11):
      if maze.maze_layout[i][j] == 1:
        self.utility[i][j] = -100
      elif maze.maze_layout[i][j] == 3:
        self.utility[i][j] = 1000


def choose_action(self, state):
  if np.random.rand() < self.epsilon:  # Exploration
      choice = np.random.choice(list(self.actions.keys()))
      return self.actions[choice]
  else:  # Exploitation
      # Calculate valid actions
      valid_actions = [action for action in self.actions.values() if self.is_valid(
                                                tuple(np.array(state) + np.array(
                                                action)))]

      # Find the action leading to the highest utility state
      best_action = max(valid_actions, key=lambda a: self.utility[tuple(np.array(
                                                state) + np.array(a))])

      #print(valid_actions)
      #print("chosen action: ", best_action)

      return best_action


def update_utility_value(self, current_state, reward, new_state):
  current_value = self.utility[current_state[0], current_state[1]]
  new_value = self.utility[new_state[0], new_state[1]]

  # TD(0) update formula
  self.utility[current_state[0], current_state[1]] += self.alpha * (reward + self.
                                                    gamma * new_value - current_value)

def run_episodes(self):
      for i in range(self.episodes):
          state = self.reset()
          done = False
          step_num = 0
          while not done and step_num < 1000:
          #while not done:
              #print("position: ", self.current_pos)
              #input("Press Enter to continue...")
              action = self.choose_action(state)
              new_state, reward = self.step(action)
              self.update_utility_value(state, reward, new_state)
              state = new_state

              if reward == self.goal_reward or reward == self.trap_penalty:  #
                                                              Episode ends
                  done = True
              step_num += 1
```

```
            #self.value_history.append(self.utility.copy())
            if i%100 == 0:
              print("episode: ", i, " is completed.")
              values = maze_td0.utility
              new_values = np.trunc(values*10**2)/(10**2)
              print(new_values.astype(np.int32))
        return self.utility
```

2.4 Q learning

```
class MazeQLearning(MazeEnvironment):
    def __init__(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=100):
        super().__init__()
        self.maze = maze
        self.alpha = alpha # Learning Rate
        self.gamma = gamma # Discount factor
        self.epsilon = epsilon # Exploration Rate
        self.episodes = episodes
        # Initialize Q-table with zeros
        self.q_table = np.zeros(self.maze.maze_layout.shape + (len(self.actions),))

    def choose_action(self, state):
        if np.random.rand() < self.epsilon:  # Exploration
            return np.random.choice(list(self.actions.keys()))
        else:  # Exploitation
            state_q_values = self.q_table[state[0], state[1], :]
            return np.argmax(state_q_values)

    def update_q_table(self, action, current_state, reward, new_state):
        current_q = self.q_table[current_state[0], current_state[1], action]
        max_future_q = np.max(self.q_table[new_state[0], new_state[1], :])
        new_q = current_q + self.alpha * (reward + self.gamma * max_future_q -
                                                    current_q)
        self.q_table[current_state[0], current_state[1], action] = new_q

    def run_episodes(self):
        for i in range(self.episodes):
            state = self.reset()
            done = False
            while not done:
                action = self.choose_action(state)
                new_state, reward = self.step(self.actions[action])
                self.update_q_table(action, state, reward, new_state)
                state = new_state

                if reward == self.goal_reward or reward == self.trap_penalty:  #
                                                                Episode ends
                    done = True

            if i % 100 == 0:
                print(f"Episode {i} completed.")

        return self.q_table
```

2.5 Running simulations

```
# running the td
maze_td0 = MazeTD0(maze, alpha=0.1, gamma=0.1, epsilon=0.2, episodes=10000)
final_values = maze_td0.run_episodes()

#running the q
maze_q = MazeQLearning(maze, alpha=0.1, gamma=0.1, epsilon=0.2, episodes=10000)
final_values = maze_q.run_episodes()
```

2.6 visualizations

```
# Extract the value function from the Q-table
value_function = np.max(q_values, axis=2)

# Plot the value function
plot_value_function(value_function, maze.maze_layout)
plot_policy(value_function, maze.maze_layout)
maze_td0.plot_convergence()
```

```
values = maze_td0.utility
plot_value_function(values, maze.maze_layout)
plot_policy(values, maze.maze_layout)
```