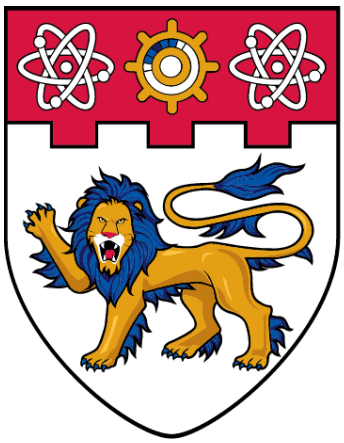


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Nanyang Technological University

School of Computer Science and Engineering

SC4003 – Intelligent Agents

Assignment 1 Report

Cholakov Kristiyan Kamenov

U2123543B

Table of Contents

1. Introduction

This report presents the findings of the first assignment for the course **SC4003 Intelligent Agents** at Nanyang Technological University, focusing on agent decision making. The first assignment covers the first half of the course – Single Intelligent Agents. The assignment's primary objective is to analyze agent behavior and decision-making processes within a simulated maze environment, drawing on concepts from Module 3: Agent Decision Making, and specifically referencing Chapters 16 and 17 of "Artificial Intelligence: A Modern Approach" by S. Russell and P. Norvig.

The solution presented in this uses a functional interactive maze environment (mini-game) for visualizing the maze environments, their utilities and the policies for them. The interactive maze environment (mini-game) is implemented via PyGame and is benefitial for better graphic representation of the results of the algorithms compared to terminal/console text output.

Components of the interactive environment:

- Blue cell – The initial position of the agent
- White/Emplty cells – Blank cells
- Green cells – Reward cells
- Orange cells – Penalty/Hole cells
- Dark grey cells – Wall cells
- Purple avatar – The current position of the agent

All the mentioned components and functionalities can be seen below in Figure 1.

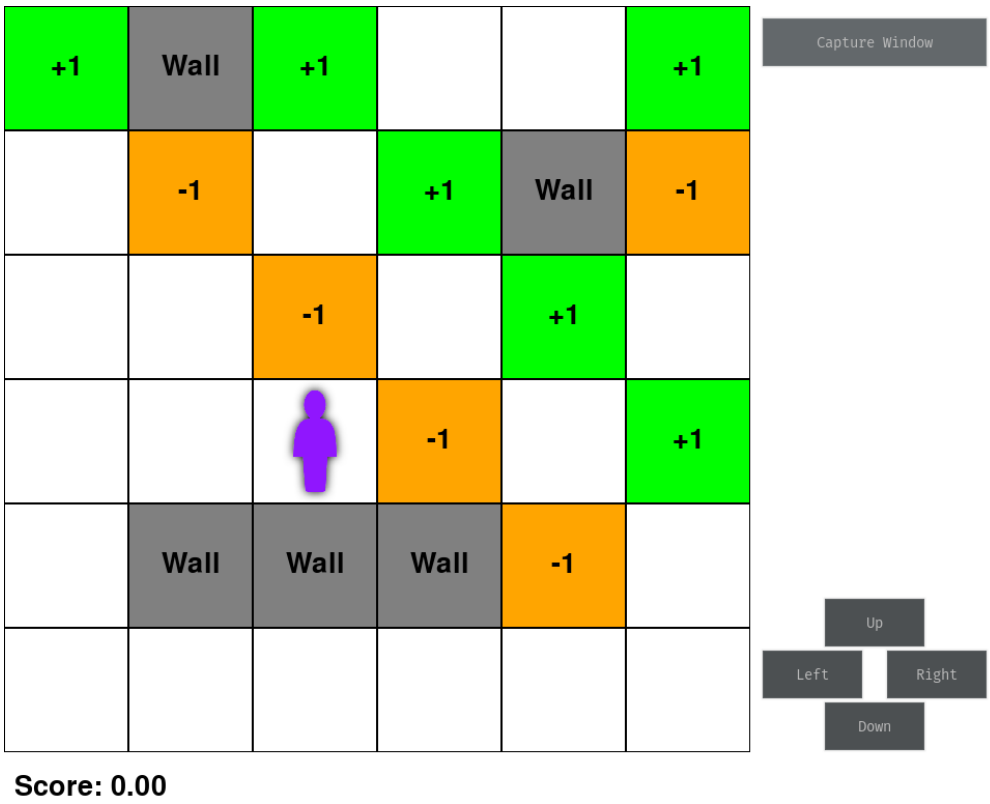


Figure 1. Screenshot of the interactive environment

2. Problem Description

2.1. Maze Environment (Environment States)

The single intelligent agent is supposed to be placed inside a predefined maze environment. The environment is a 6x6 maze with 4 different types of cells/tiles:

- Rewards – Cells on which the agent receives a reward (+1)
- Holes/Penalties – Cells on which the agent is penalised (-1)
- Walls – Cells on which the agent cannot go
- Empty Cells – Cells on which the agent is penalised slightly (-0.04)

Also, the environment has the following rules:

- The agent cannot enter the wall cells and go out of bounds
- The environment does not have a termination state (the agent can execute indefinitely in the environment)
- The agent can perform only 1 out of 4 possible actions at once

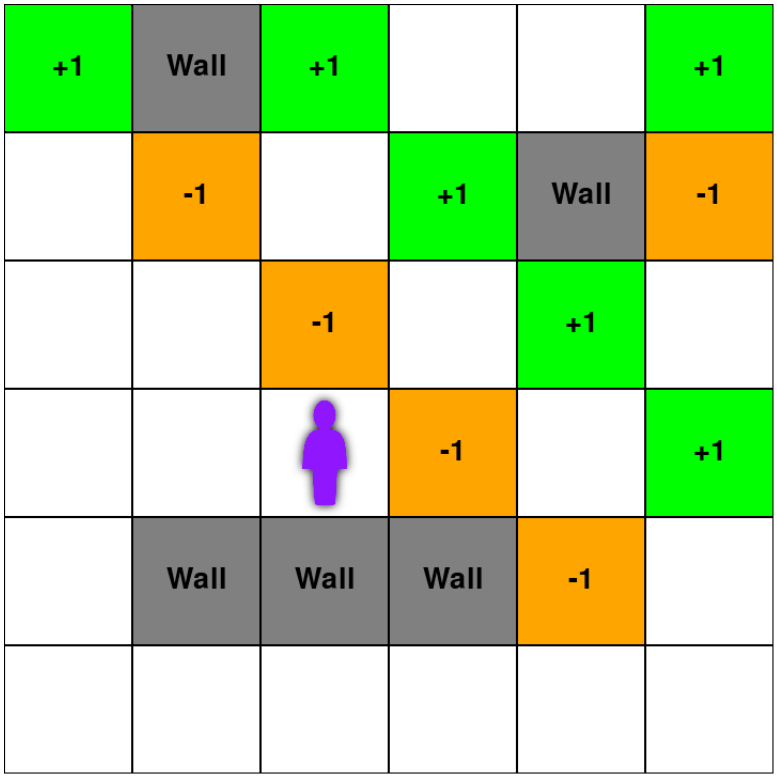


Figure 2. Maze Environment (given in the assignment)

Figure 2. is a visualization of the given environment in the assignment.

2.2. Possible Action

The actions that the agent can perform are going in any of the 4 direction (up, down, left, right) with a slight probability of moving incorrectly in either the left or right direction in terms of the intended direction.

2.3. Transition Model

The transition model for the agen of making the intended action is as follows:

- Probability of 0.8 of the agent going in the intended direction
- Probability of 0.1 of the agent going in the right direction relative to the intended direction
- Probability of 0.1 of the agent going in the left direction relative to the intended direction

If the agent attempts to move into a wall or outside the bounds of the maze, it remains in its current location. This model captures the uncertainty and challenges faced by agents in real-world environments where actions do not always lead to the expected outcomes.

2.4. Discount Factor

We are given the discount factor of 0.99 for the scope of the assignment.

2.5. Objectives

The assignment consists of 2 primary objectives: Task 1 and Task 2 (Bonus Question)

2.5.1. Task 1

Assuming the known transition model and reward function listed above, find the optimal policy and the utilities of all the (non-wall) states using both value iteration and policy iteration. Display the optimal policy and the utilities of all the states, and plot utility estimates as a function of the number of iterations. In this question, use a discount factor of 0.99.

2.5.2. Task 2 (Bonus Question)

Design a more complicated maze environment of your own and re-run the algorithms designed for Part 1 on it. How does the number of states and the complexity of the environment affect convergence? How complex can you make the environment and still be able to learn the right policy?

3. Solution Project Structure

config.py – A file to hold all the predefined constants and variables for the solution

maze_configs – A directory with .py files that represent the different maze environments

- **base.py** – A file containing all the needed data for constructing the base maze environment (given in the assignment)
- **blockages.py** - A file containing all the needed data for constructing the blockages maze environment (used for Task 2)
- **increased_size.py** - A file containing all the needed data for constructing the increased-size maze environment (used for Task 2)
- **labyrinth.py** - A file containing all the needed data for constructing the labyrinth maze environment (used for Task 2)

main.ipynb – A Jupyter notebook for retrieving the results used in the report

main.py – A file used for running the interactive environment (running the mini-game)

grid.py – A file used for handling the graphics of the mini-game

utils.py – A file to store the common functions used across the project

window_capture – A directory to hold all the screenshots from the mini-game or the visualization of the environment, utilities or policies

results – A directory to hold all the csv tables with the results from the Value Iteration and the Policy Iteration for the different maze environments

plots – A directory to hold all the plots from the Value Iteration and the Policy Iteration for the different maze environments

assets – A directory to hold all visual assets used for visualizing or playing the mini-game

algorithms – A directory to hold the python files used for implementing the Value Iteration and Policy Iteration algorithms, also, storing the file with the visualization function for the environment, utilities and policies

- **value_iteration.py** – A file implementing the Value Iteration algorithm
- **policy_iteration.py** – A file implementing the Policy Iteration algorithm
- **algorithm_utils.py** – A file storing all the functions used across the algorithms and the visualization functions for the environment, utilities and policies

4. Task 1

4.1. Value Iteration

Value iteration is a dynamic programming algorithm used for determining the optimal policy in a Markov Decision Process (MDP). It iteratively computes the utility of each state until the utility values converge to a stable set of values. At each iteration, the algorithm updates the utility of every state based on the expected utility of taking the best action, considering both the immediate reward and the discounted future rewards. This process continues until the change in utility values between iterations is below a small threshold, indicating convergence. The resulting utility values are then used to derive the optimal policy, which specifies the best action to take in each state to maximize the agent's expected reward over time.

4.1.1. Theory

4.1.1.1. Markov Decision Process (MDP)

An MDP is defined by a set of states (S), a set of actions (A), a transition model (T), and a reward function (R). The transition model $T(a)$ describes the probability of performing the current action and the probabilities of performing the incorrect actions. The reward function $R(s)$ specifies the immediate reward received after transitioning from state s to state s' by taking action a.

4.1.1.2. The Principle of Optimality

The Value Iteration algorithm is based on the Bellman Equation, which expresses the principle of optimality. For any policy to be optimal, the expected utility of taking action a in state s (and thereafter following the optimal policy) must be equal to the expected utility of the state s. The Bellman Equation for a state s can be formulated as:

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s,a)\, U(s')$$

where $U(s)$ is the utility of state s, γ is a discount factor between 0 and 1 that models the present value of future rewards, and the summation is over all possible next states s'.

4.1.1.3. Algorithm Overview

Value Iteration proceeds by repeatedly updating the value of each state using the Bellman Equation until the values converge. Specifically, for each state, it calculates the expected utility of taking each possible action and then updates the value of the state to be the maximum of these expected utilities. The algorithm follows these steps:

1. **Initialization**: Start with arbitrary utility values for all states, often initialized to zero.
2. **Iteration**: For each state, calculate the expected utility of taking each action $EU(s,a) = \sum_{s'} P(s'|s,a)\, U(s')$ and update the value of the state to the maximum of these expected utilities.
3. **Convergence Check**: After each full iteration over all states, check if the maximum change in the value of any state is less than the tolerance threshold $Tolerance = \epsilon \frac{1-\gamma}{\gamma}$ whre ϵ (epsilon) is defined by $\epsilon = C\, R_{max}$ (C – tolerance constant, R_{max} – the maximum reward). If not, repeat step 2.

4. **Policy Extraction**: Once the utility values have converged, extract the optimal policy by choosing for each state the action that maximizes the expected utility based on the final utility values. The optimal action for each state is chosen based on the following equation: $Best\_Action = \underset{a \in A}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)\, U(s')$

4.1.2. Implementation

The implementation of the Value Iteration algorithm is done in the **value_iteration.py** and **algorithm_utils.py** files in the **algorithms** directory.

```
# Function to initialize the value iteration maze environment
kristiyancholakov
def init_vi_env():
    # Create a get_grid_size_h() x get_grid_size_w() grid matrix filled with zeros
    vi_env = [[0 for _ in range(get_grid_size_w())] for _ in range(get_grid_size_h())]
    return vi_env
```

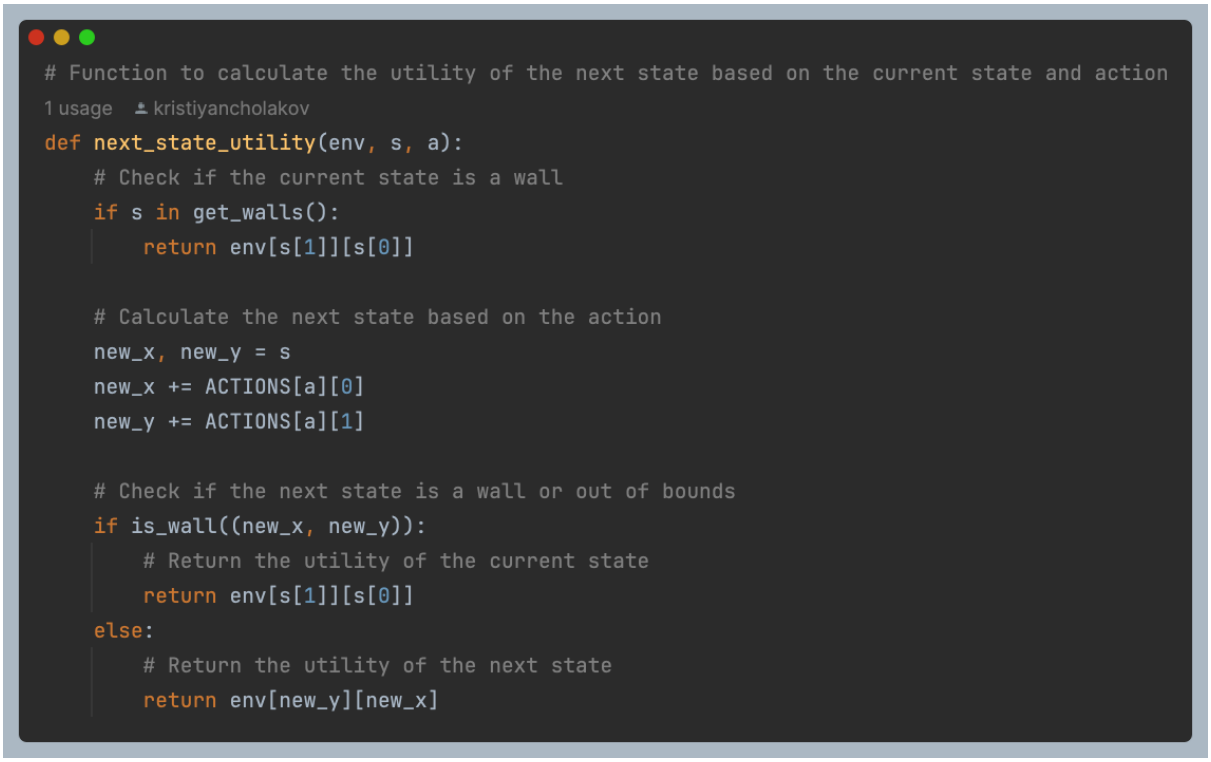
Figure 3. init_vi_env() implementation

Figure 3. shows a function that creates a blank environment (all 0s) matrix with sizes based on the predefined maze environment. Matrices initialized with this function are used to store the utilities of the states.

```
# Function to check if the state is a wall or out of bounds
1 usage  kristiyancholakov
def is_wall(s):
    return s in get_walls() or s[0] < 0 or s[0] >= get_grid_size_w() or s[1] < 0 or s[1] >= get_grid_size_h()
```

Figure 4. is_wall(s) implementation

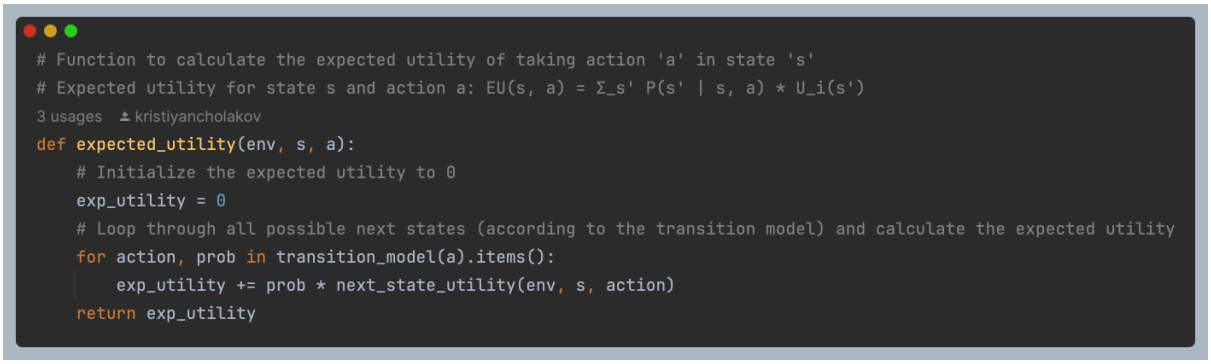
Figure 4. shows a function to check if the state is a wall cell or is out of bounds.



```python
# Function to calculate the utility of the next state based on the current state and action
1 usage  kristiyancholakov
def next_state_utility(env, s, a):
    # Check if the current state is a wall
    if s in get_walls():
        return env[s[1]][s[0]]

    # Calculate the next state based on the action
    new_x, new_y = s
    new_x += ACTIONS[a][0]
    new_y += ACTIONS[a][1]

    # Check if the next state is a wall or out of bounds
    if is_wall((new_x, new_y)):
        # Return the utility of the current state
        return env[s[1]][s[0]]
    else:
        # Return the utility of the next state
        return env[new_y][new_x]
```

Figure 5. next_state_utility(env, s, a) implementation

Figure 5. shows a function that calculates the utility of the next state $U(s')$ determined by the current one and the taken action. First, we are checking if the current state is not a wall cell, if it is not, we are calculating the next state. If the next state is a wall or out of bounds, we return the current state utility as the agent will stay in the same place, if the next state is feasible, we return its utility.



```python
# Function to calculate the expected utility of taking action 'a' in state 's'
# Expected utility for state s and action a: EU(s, a) = Σ_s' P(s' | s, a) * U_i(s')
3 usages  kristiyancholakov
def expected_utility(env, s, a):
    # Initialize the expected utility to 0
    exp_utility = 0
    # Loop through all possible next states (according to the transition model) and calculate the expected utility
    for action, prob in transition_model(a).items():
        exp_utility += prob * next_state_utility(env, s, action)
    return exp_utility
```

Figure 6. expected_utility(env, s, a) implementation

Figure 6. shows a function that calculates the expected utility for the current state and a taken action. $EU(s,a) = \sum_{s'} P(s'|s,a)\, U(s')$ is calculated by summing the next possible states' utilities multiplied by thir probability.

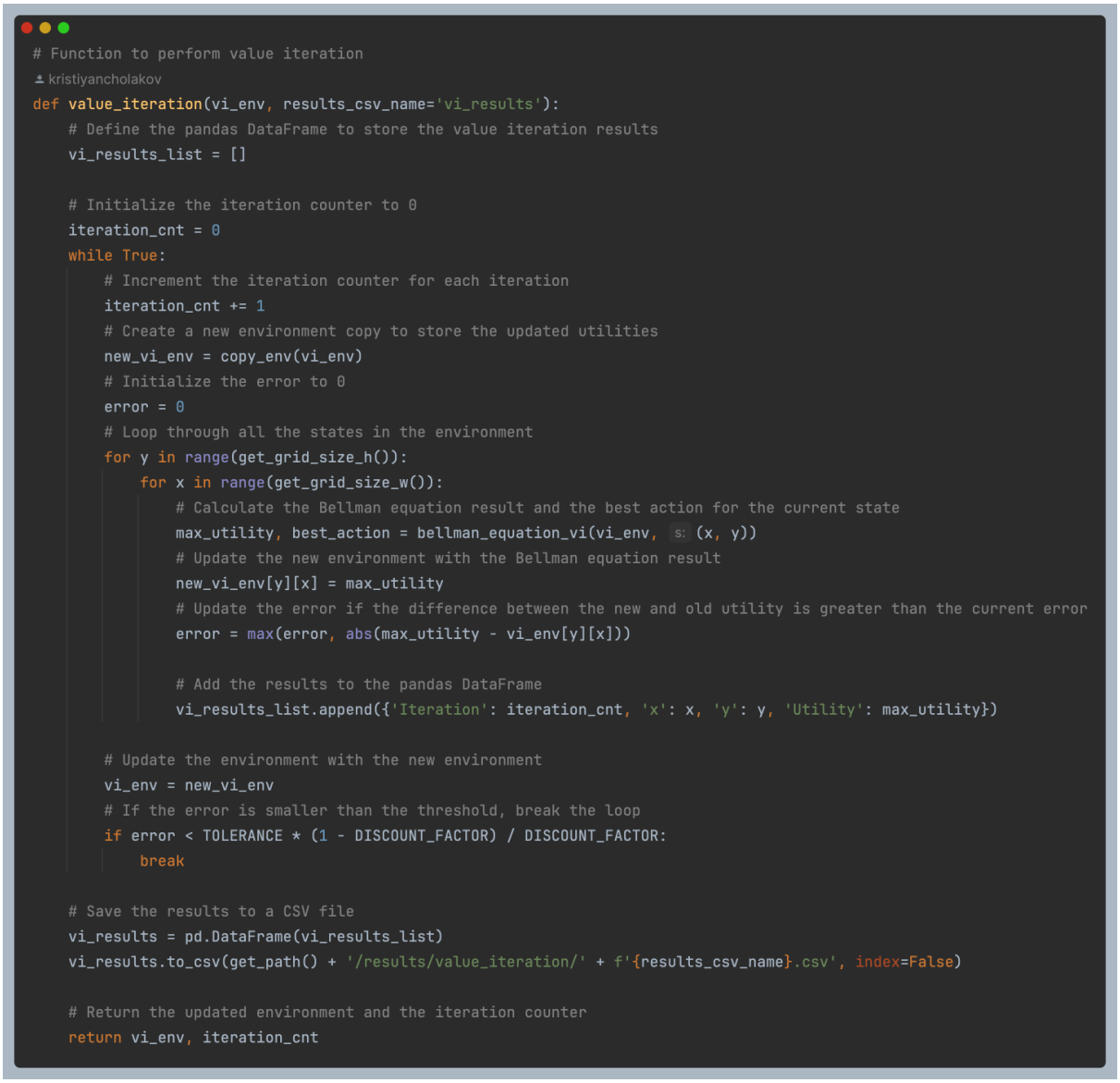
```python
# Function for the Bellman equation used for value iteration
# Bellman equation: U_(i+1)(s) = R(s) + γ * max_(a ∈ A(s)) ∑_(s') P(s' | s, a) * U_i(s')
1 usage  kristiyancholakov
def bellman_equation_vi(vi_env, s):
    # Check if the current state is a reward, hole, wall or empty field and assign the corresponding reward
    if s in get_rewards():  # If the current state is a reward
        reward = get_rewards()[s]
    elif s in get_holes():  # If the current state is a hole
        reward = get_holes()[s]
    elif s in get_walls():  #
        reward = 0
    else:  # If the current state is an empty field
        reward = get_empty_reward()

    # Defining the maximum utility --> max_(a ∈ A(s)) ∑_(s') P(s' | s, a) * U_i(s')
    max_utility = float('-inf')
    # Defining the best action --> argmax_(a ∈ A(s)) ∑_(s') P(s' | s, a) * U_i(s')
    best_action = None
    # Loop through all possible actions
    for action in ACTIONS:
        # Calculate the expected utility of taking action the current action in the current state
        utility = expected_utility(vi_env, s, action)
        # If the utility is greater than the maximum utility, update the maximum utility and the best action
        if utility > max_utility:
            max_utility = utility
            best_action = action

    # Return the Bellman equation result and the best action
    return (reward + DISCOUNT_FACTOR * max_utility), best_action
```

Figure 7. bellman_equation_vi(vi_env, s)

Figure 7. shows a function that implements the Bellman's equation:

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s,a)\, U(s')$$

The function returns the utility for the current state as the sum of the current state's reward and the expected utility for the best action in the current state.



```python
# Function to perform value iteration
kristiyancholakov
def value_iteration(vi_env, results_csv_name='vi_results'):
    # Define the pandas DataFrame to store the value iteration results
    vi_results_list = []

    # Initialize the iteration counter to 0
    iteration_cnt = 0
    while True:
        # Increment the iteration counter for each iteration
        iteration_cnt += 1
        # Create a new environment copy to store the updated utilities
        new_vi_env = copy_env(vi_env)
        # Initialize the error to 0
        error = 0
        # Loop through all the states in the environment
        for y in range(get_grid_size_h()):
            for x in range(get_grid_size_w()):
                # Calculate the Bellman equation result and the best action for the current state
                max_utility, best_action = bellman_equation_vi(vi_env, s: (x, y))
                # Update the new environment with the Bellman equation result
                new_vi_env[y][x] = max_utility
                # Update the error if the difference between the new and old utility is greater than the current error
                error = max(error, abs(max_utility - vi_env[y][x]))

                # Add the results to the pandas DataFrame
                vi_results_list.append({'Iteration': iteration_cnt, 'x': x, 'y': y, 'Utility': max_utility})

        # Update the environment with the new environment
        vi_env = new_vi_env
        # If the error is smaller than the threshold, break the loop
        if error < TOLERANCE * (1 - DISCOUNT_FACTOR) / DISCOUNT_FACTOR:
            break

    # Save the results to a CSV file
    vi_results = pd.DataFrame(vi_results_list)
    vi_results.to_csv(get_path() + '/results/value_iteration/' + f'{results_csv_name}.csv', index=False)

    # Return the updated environment and the iteration counter
    return vi_env, iteration_cnt
```

Figure 8. value_iteration(vi_env) implementation

Figure 8. shows the function that implements the whole Value Iteration algorithm. The algorithm loops through all the states of the environment and calculates their utilities until a convergence is reached. The convergence is reached when the maximum error in the environment is less than the tolerance equation threshold. As a result, the environment matrix with the utility values and the number of iterations are returned.

```python
# Function to generate the optimal policy based on the value iteration results
kristiyancholakov
def generate_policy(vi_env):
    # Create a get_grid_size_h() x get_grid_size_w() grid matrix to store the optimal policies for each state
    policy = [[None for _ in range(get_grid_size_w())] for _ in range(get_grid_size_h())]
    # Loop through all the states in the environment
    for y in range(get_grid_size_h()):
        for x in range(get_grid_size_w()):
            # Initialize the maximum utility to negative infinity and the best action to None
            max_utility = float('-inf')
            best_action = None
            # Loop through all the possible actions
            for action in ACTIONS:
                # Calculate the expected utility of taking the current action in the current state
                utility = expected_utility(vi_env, s: (x, y), action)
                # If the utility is greater than the maximum utility, update the maximum utility and the best action
                if utility > max_utility:
                    max_utility = utility
                    best_action = action
            # Update the policy with the best action for the current state
            policy[y][x] = best_action
    # Return the optimal policy
    return policy
```

Figure 9. generate_policy(vi_env) implementation

Figure 9. shows the function used to generate the policy after the Value Iteration has passed. The policy is generated by choosing the action that will result in the highest expected utility for each state. The result from the function is a matrix with the size of the predefined maze environment with the best action for each state.

4.1.3. Results

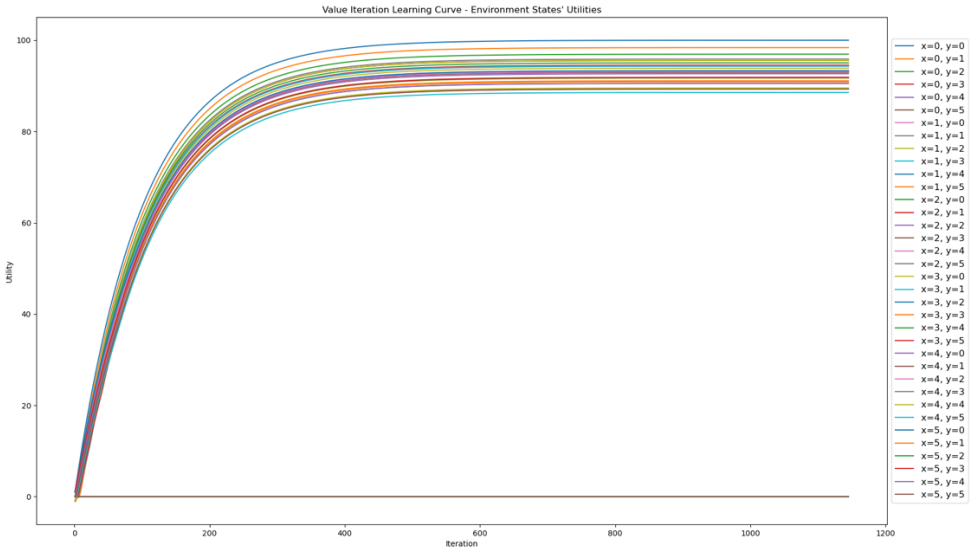


Figure 10. Value Iteration Learning Curve – Environment States' Utilities

Figure 10. shows a plot of the environment states' utilities over the number of iterations the Value Iteration algorithm was running for. This plot basically represents the learning curve of the algorithm.

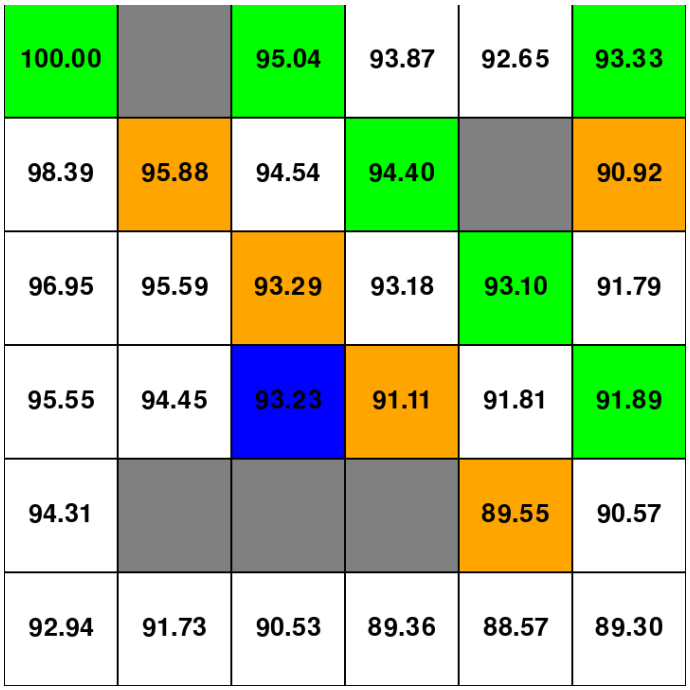


Figure 11. Environment states' utilities (Value Iteration for Base Maze)

Figure 11. shows the final environment states' utilities after the final iteration of the Value Iteration algorithm. Using these utilities the **generate_policy(vi_env)** function will generate the optimal policy for the environment.

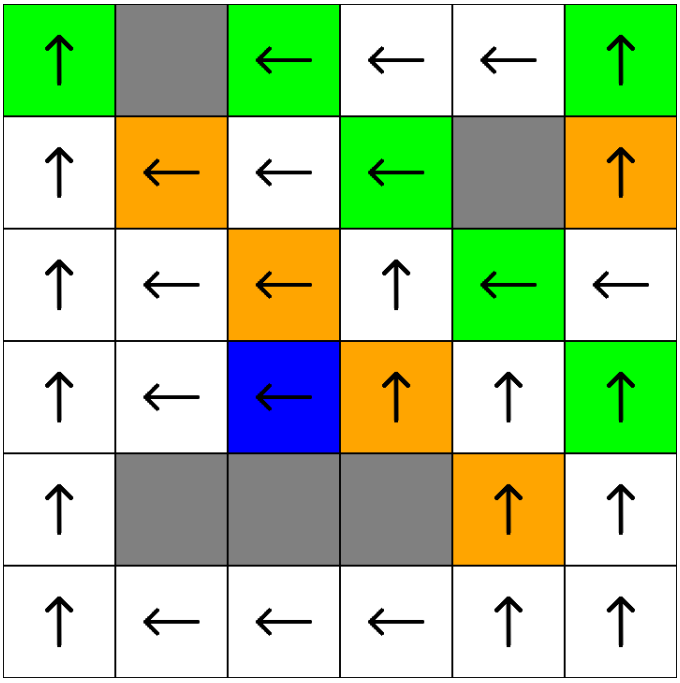


Figure 12. Optimal policy (Value Iteration for Base Maze)

Figure 12. shows the generated optimal policy by the **generate_policy(vi_env)** function after running the Value Iteration algorithm for the Base Maze (given by the assignment) environment.

4.1.4. Results Analysis

For performing the Value Iteration we have chosen the following error tolerance parameters:

- $C = 0.1$
- $\epsilon = R_{max}\, C = 0.1$
- $Tolerance = \epsilon \frac{1-\gamma}{\gamma} = \frac{0.1(1-0.99)}{0.99} = \frac{1}{990}$

In Figure 10. we can see the plotted utilities for all the environment states over the 1145 iterations of the algorithm. In the beginning of the Value Iteration, we can spot a steep increase in the utilities of all states, this is due to the fact that the algorithm has started learning and the utilities' initial 0 values are improving. In the later iterations (>500) the curve becomes flatter as we have set the error tolerance threshold low. The lower the tolerance is the longer it will take for the algorithm to stop (converge based on the tolerance).

In Figure 11. we can observe the final version of the environment states' utilities. We can see that the utilities are increasing towards the green cells (the reward states), this is what was expected as the rewards increase the utilities of the states. The cell with the highest utility is the top left corner state (0, 0) with a utility around 100 (the utility values are rounded up to the second decimal place). This means that the (0, 0) state is the desired one.

In Figure 12. our observation for the (0, 0) state is confirmed, as most of the policies direct towards it. Now, we can also see that starting from the given initial position and following the policies will take the agent to the top left corner. This is easy explainable because the (0, 0) position is surrounded with a wall to the right side and is bounded to the left and up side. This means that if the agent reaches the (0, 0) state, he can just try to go up forever, his possible transitions according to the transition model are infeasible and thus the

agent will stay in the (0, 0) state and will gain +1 reward every time. This strategy is the best and is possible because of the infinitive and static type of the environment.

4.2. Policy Iteration

Policy iteration is a method used in Markov Decision Processes (MDP) for finding the optimal policy that maximizes the expected reward for an agent over time. Unlike value iteration, which directly computes the utility of states, policy iteration alternates between two main steps: policy evaluation and policy improvement. In the policy evaluation step, the algorithm calculates the utility of each state under the current policy by solving a set of linear equations, essentially determining how good it is to follow the current policy. Then, in the policy improvement step, the policy is updated by choosing actions in each state that maximize the expected utility based on the current utility estimates. This process repeats, with the policy being refined in each iteration, until the policy remains unchanged between iterations, indicating that the optimal policy has been found. This method efficiently converges to the optimal policy by iteratively improving the policy based on the calculated utilities.

4.2.1. Theory

4.2.1.1. Markov Decision Process (MDP) and The Principle of Optimality

The Markov Decision Process (MDP) and the Principle of Optimality serve as the foundational concepts for both Value Iteration and Policy Iteration algorithms, with each approach applying these principles slightly differently. While Value Iteration directly computes the utility of states to derive the optimal policy, Policy Iteration iteratively refines a policy based on current utility estimates, underscoring a nuanced distinction in their application of the same core principles.

4.2.1.2. Algorithm Overview

Policy Iteration alternates between evaluating the current policy and improving it to produce a new policy that is guaranteed to be better or equally good. The process is as follows:

1. **Initialization**: Begin with an arbitrary policy π, where $\pi(s)$ specifies an action to be taken in each state s.

2. **Policy Evaluation**: Calculate the utility of each state if the agent were to follow the current policy π to the letter. This step involves solving the set of linear equations formed by the Bellman Equation for policy evaluation in terms of s and $\pi(s)$ $U(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))\, U(s')$, where $U(s')$ is the utility of state s under policy π, and γ is the discount factor.
3. **Policy Improvement**: Update the policy by choosing the action in each state that maximizes the expected utility based on the current utility estimates. This produces a new policy that is as good as or better than the previous one. The policy is updated according to: $\pi(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s'} P(s'|s, a)\, U(s')$,where $\pi(s)$ is the action chosen under the new policy for state s.
4. **Convergence Check**: The algorithm iterates between policy evaluation and policy improvement steps until the policy stabilizes and no longer changes, indicating that the optimal policy has been found. The policy is considered stabilized if there has not been a change in any state policy for the current iteration.
5. **Efficiency and Convergence**: Policy Iteration is particularly efficient because it often requires fewer iterations than Value Iteration to converge to the optimal policy. The policy improvement theorem guarantees that the policy iteration process converges to the optimal policy, as each iteration produces a policy that is strictly better than the preceding one until the optimal policy is achieved.

4.2.2. Implementation

The implementation of the Policy Iteration algorithm is done in the **policy_iteration.py** and **algorithm_utils.py** files in the **algorithms** directory.

```
# Function to initialize the policy iteration maze environment
kristiyancholakov
def init_pi_env():
    # Create a get_grid_size_h() x get_grid_size_w() grid matrix filled with zeros
    pi_env = [[0 for _ in range(get_grid_size_w())] for _ in range(get_grid_size_h())]
    # Create a get_grid_size_h() x get_grid_size_w() grid matrix filled with random actions (Initialize the policy randomly)
    pi_policy = [[random.choice(list(ACTIONS.keys())) for _ in range(get_grid_size_w())] for _ in range(get_grid_size_h())]
    # Return the environment and the policy matrices
    return pi_env, pi_policy
```

Figure 13. init_pi_env() implementation

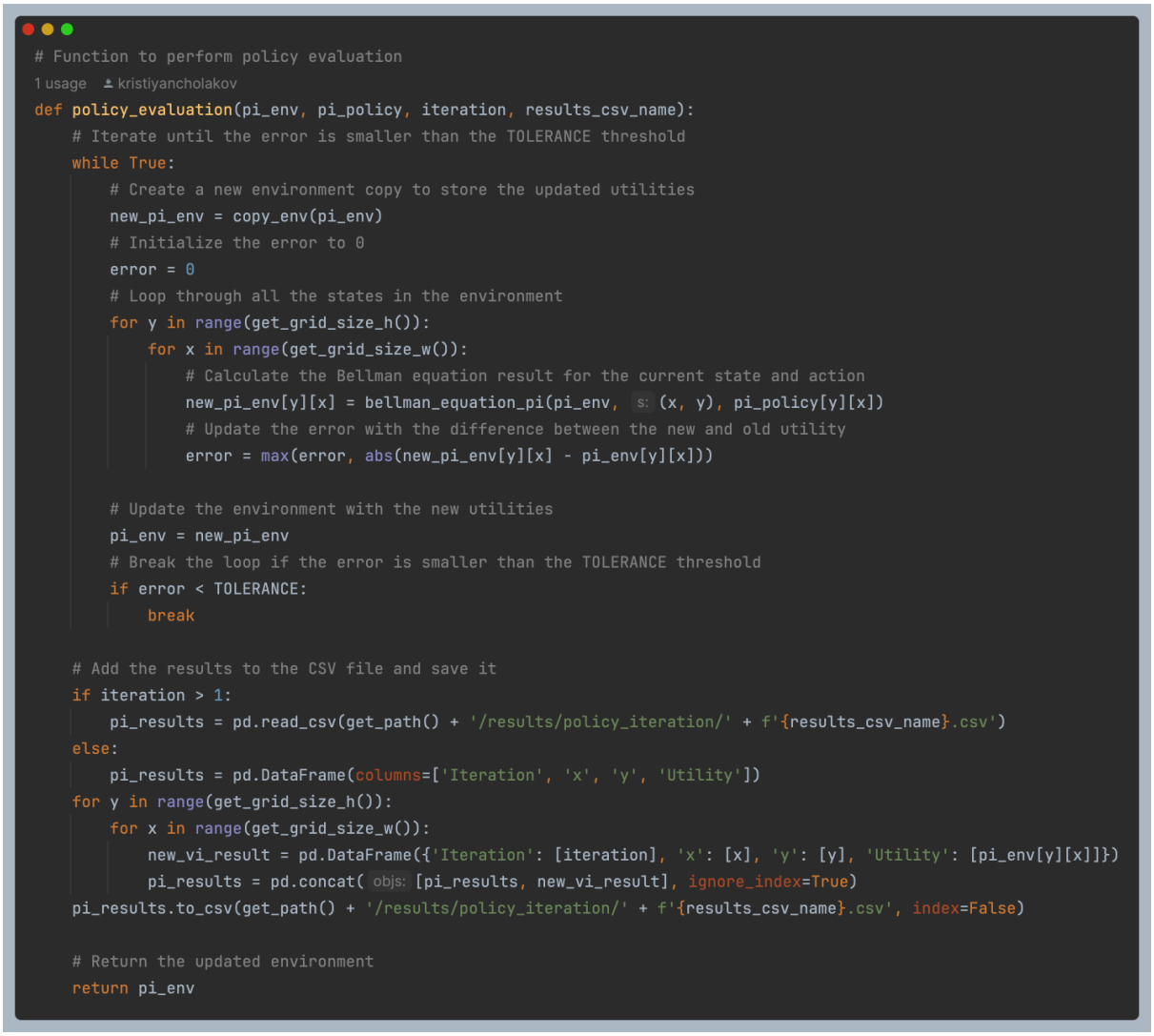
Figure 13. shows the implementation for the initialization of the environment and the initial policy for the Policy Iteration algorithm. The environment is created in similar way

to the one for Value Iteration just create a matrix with the size of the maze and all 0 values. As for the initial policy, it is created by picking a random action for each state in the environment.

```python
# Function for the Bellman equation used for policy iteration
# Bellman equation: U_(i+1)(s) = R(s) + γ * Σ_(s') P(s' | s, π(s)) * U_i(s')
2 usages  kristiyancholakov
def bellman_equation_pi(vi_env, s, action):
    # Check if the current state is a reward, hole, wall or empty field and assign the corresponding reward
    if s in get_rewards():  # If the current state is a reward
        reward = get_rewards()[s]
    elif s in get_holes():  # If the current state is a hole
        reward = get_holes()[s]
    elif s in get_walls():  # If the current state is a wall
        reward = 0
    else:  # If the current state is an empty field
        reward = get_empty_reward()

    # Calculate the expected utility of taking action 'action' in state 's'
    utility = expected_utility(vi_env, s, action)

    # Return the Bellman equation result
    return reward + (DISCOUNT_FACTOR * utility)
```

Figure 14. bellman_equation_pi(vi_env, s, action)

Figure 14. presents the implementation of the Bellman equation for the Policy Iteration algorithm: $U(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s))\, U(s')$. The difference in the Policy Iteration version is that we do not search through all the possible actions, we just compute the expected utility in terms of s and $\pi(s)$ (passed through the parameter **a**). The **expected_utility(env, s, a)** and **next_state_utility(env, s, a)** functions are reused as the environment and its handling are done in the same way as for Value Iteration. That is why these two function are moved to the **algorithm_utils.py** file.



```python
# Function to perform policy evaluation
1 usage  kristiyancholakov
def policy_evaluation(pi_env, pi_policy, iteration, results_csv_name):
    # Iterate until the error is smaller than the TOLERANCE threshold
    while True:
        # Create a new environment copy to store the updated utilities
        new_pi_env = copy_env(pi_env)
        # Initialize the error to 0
        error = 0
        # Loop through all the states in the environment
        for y in range(get_grid_size_h()):
            for x in range(get_grid_size_w()):
                # Calculate the Bellman equation result for the current state and action
                new_pi_env[y][x] = bellman_equation_pi(pi_env, s: (x, y), pi_policy[y][x])
                # Update the error with the difference between the new and old utility
                error = max(error, abs(new_pi_env[y][x] - pi_env[y][x]))

        # Update the environment with the new utilities
        pi_env = new_pi_env
        # Break the loop if the error is smaller than the TOLERANCE threshold
        if error < TOLERANCE:
            break

    # Add the results to the CSV file and save it
    if iteration > 1:
        pi_results = pd.read_csv(get_path() + '/results/policy_iteration/' + f'{results_csv_name}.csv')
    else:
        pi_results = pd.DataFrame(columns=['Iteration', 'x', 'y', 'Utility'])
    for y in range(get_grid_size_h()):
        for x in range(get_grid_size_w()):
            new_vi_result = pd.DataFrame({'Iteration': [iteration], 'x': [x], 'y': [y], 'Utility': [pi_env[y][x]]})
            pi_results = pd.concat( objs: [pi_results, new_vi_result], ignore_index=True)
    pi_results.to_csv(get_path() + '/results/policy_iteration/' + f'{results_csv_name}.csv', index=False)

    # Return the updated environment
    return pi_env
```

Figure 15. policy_evaluation(pi_env, pi_policy, iteration) implementation

Figure 15. shows the implementation of the policy evaluation (performed in every Policy Iteration algorithm iteration). This function evaluates the policy for the environment and calculates the utilities for all environment states.

```python
# Function to perform policy iteration
# kristiyancholakov *
def policy_iteration(pi_env, pi_policy, results_csv_name='pi_results'):
    # Initialize the iteration counter to 0
    iteration_cnt = 0
    while True:
        # Increment the iteration counter for each iteration
        iteration_cnt += 1
        # Perform policy evaluation
        new_pi_env = policy_evaluation(pi_env, pi_policy, iteration_cnt, results_csv_name)
        # Initialize the policy_stable flag to True (used to check if the policy has converged)
        policy_stable = True
        # Loop through all the states in the environment
        for y in range(get_grid_size_h()):
            for x in range(get_grid_size_w()):
                # Store the old action
                old_action = pi_policy[y][x]
                # Defining the maximum utility --> max_(a ∈ A(s)) ∑_(s') P(s' | s, a) * U_i(s')
                max_utility = float('-inf')
                # Defining the best action --> argmax_(a ∈ A(s)) ∑_(s') P(s' | s, a) * U_i(s')
                best_action = None
                # Loop through all possible actions
                for action in ACTIONS:
                    # Calculate the Bellman equation result for the current state and action
                    utility = bellman_equation_pi(new_pi_env, s: (x, y), action)
                    # If the utility is greater than the maximum utility, update the maximum utility and the best action
                    if utility > max_utility:
                        max_utility = utility
                        best_action = action
                # Update the environment with the Bellman equation result
                pi_env[y][x] = max_utility
                # Update the policy with the best action
                pi_policy[y][x] = best_action

                # If the old action is different from the best action, set the policy_stable flag to False
                if old_action != best_action:
                    policy_stable = False

        # Break the loop if the policy has converged
        if policy_stable:
            break

    # Return the updated environment, policy and the iteration counter
    return pi_env, pi_policy, iteration_cnt
```

Figure 16. policy_iteration(pi_env, pi_policy) implementation

Figure 16. shows the implementation of the Policy Iteration algorithm. The algorithm iterates and improves the environment policy until no environment state policy can be improved, then it is considered that the algorithm has converged. For each iteration, the algorithm calculates the utilities for all possible actions for each environment state and chooses the action with the maximum utility. If the action is different that the previously assigned one, then the policy has changed and the model will go for another iteration until no state policy can be improved (changed).

4.2.3. Results

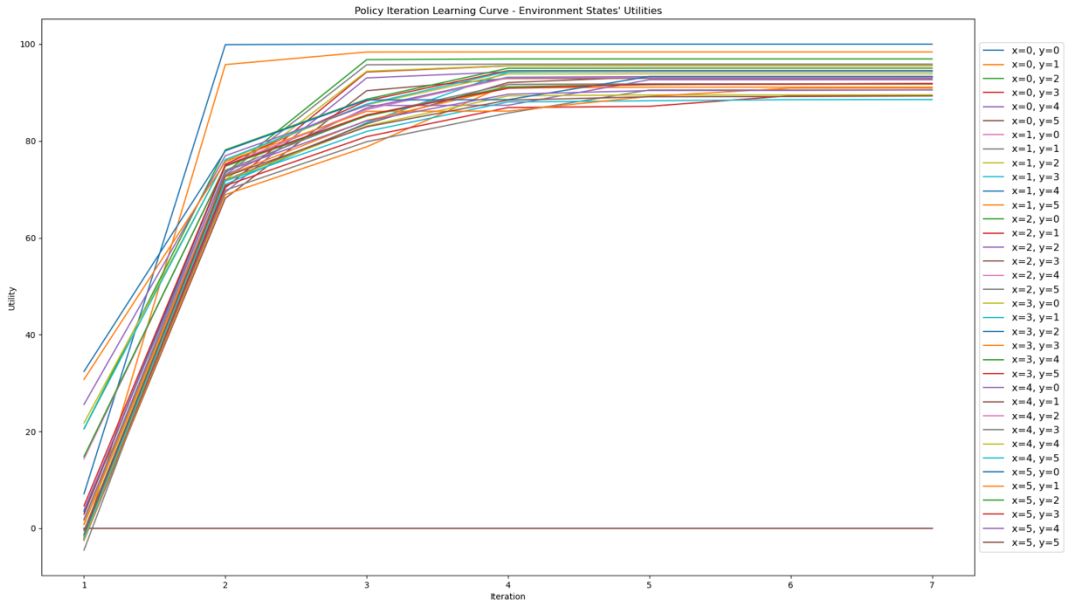


Figure 17. Policy Iteration Learning Curve – Environment States' Utilities

Figure 17. shows a plot of the environment states' utilities over the number of iterations the Policy Iteration algorithm was running for. This plot basically represents the learning curve of the algorithm.

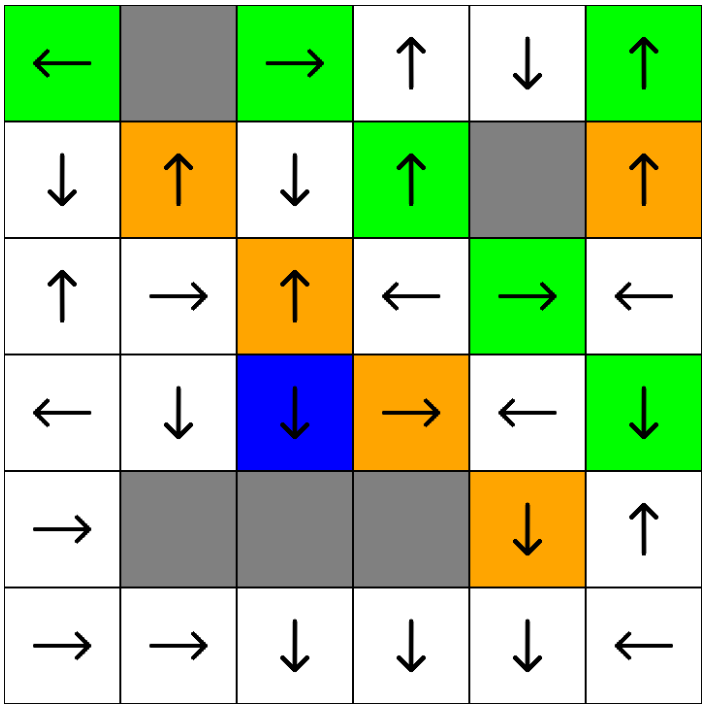


Figure 18. Initial policy (Policy Iteration for Base Maze)

Figure 18. shows the initial arbitrary policy for Policy Iteration algorithm for the Base Maze environment.

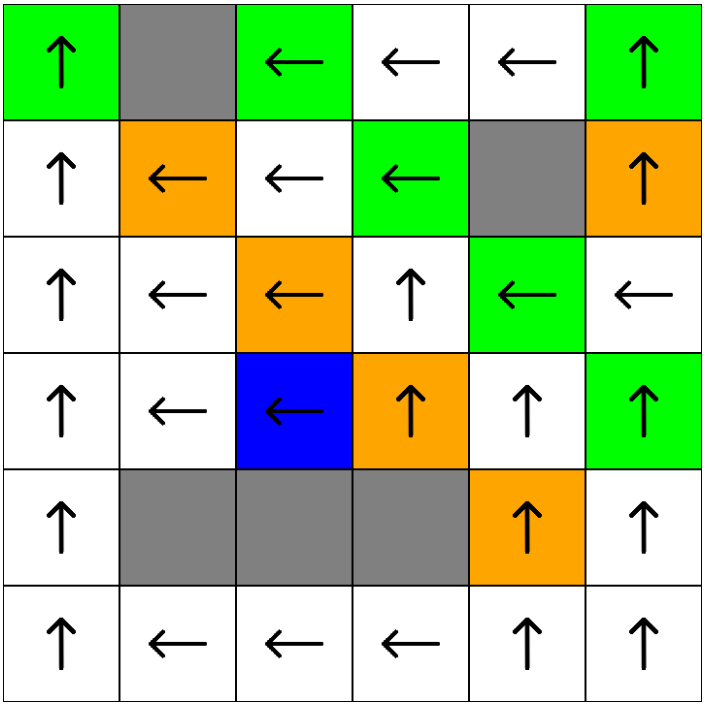


Figure 19. Optimal Policy (Policy Iteration for Base Maze)

Figure 19. shows the optimal policy, the result after running the Policy Iteration algorithm for the Base Maze (given by the assignment) environment.

100.00		95.05	93.87	92.65	93.33
98.39	95.88	94.54	94.40		90.92
96.95	95.59	93.29	93.18	93.10	91.79
95.55	94.45	93.23	91.12	91.81	91.89
94.31				89.55	90.57
92.94	91.73	90.53	89.36	88.57	89.30

Figure 20. Environment states' utilities (Policy Iteration for Base Maze)

Figure 20. shows the environment states' utilities after running the Policy Iteration algorithm for the Base Maze environment (defined in the assignment).

4.2.4. Results Analysis

The tolerance constants are not changed for producing these results. However, the condition for convergence is changed due to the inherent mechanics of each algorithm. Value Iteration's convergence is influenced by the discount factor, necessitating an adjusted threshold to accurately reflect the diminishing impact of future rewards. Policy Iteration, focusing on policy stability rather than direct utility convergence, uses a simpler condition, as its convergence is determined by when the policy ceases to change, not solely on the utilities' numerical precision.

Figure 17. shows the significant difference between the Policy Iteration and the Value Iteration algorithms. In comparison to the Value Iteration algorithm that needed more than 1400 iterations to conver, the Policy Iteration algorithm converges in less than 10. The number of iterations required for Policy Iteration may vary but in most cases is less than 10, this is due to that the initial policy is initialised randomly every time. The closer the initial policy is to the optimal one, the faster it will converge. In conclusion, comparing the graphs we can confirm that Policy Iteration needs less iterations to converge.
Comparing Figure 18. and Figure 19. we can observe how the policy changed from the beginning of the algorithm when it was randomly initialized to when the algorithm converged.

Comparing Figure 19. and Figure 12. we can spot that there is no difference between the results that the Value Iteration and Policy Iteration algorithms gave. In both case the policy directs the agent to the upper left corner state (0, 0). As we mentioned previously, this is due to the characteristics of the environment given in the assignment.

Also, comparing Figure 20. and Figure 11. we can notice that the difference in the final utilty values for the environment states for the Value Iteration and Policy Iteration is minimal. They differ in only 2 states (2, 0) and (3, 3):

$$95.05 - 95.04 = 91.12 - 91.11 = 0.01$$

4.3. Conclusion

In the context of the Base Maze environment (defined in the assignment), Policy Iteration and Value Iteration gave optimal policy results that are exactly the same. However, the number of iterations they took to converge differs significantly. Thus, Policy Iteration performed the same result but in much less iterations and we can consider it the better algorithm for this specific task and environment.

5. Task 2

Designing a more complex environment and re-running the Value Iteration and Policy Iteration algorithms on it.

5.1. Increased-Size Maze

One way to design a more complex environment is to increase the size of the maze. This will increase the number of states in the environment.

5.1.1. Proposed Maze Environment

Thus, we will consider the new size of 10x10 and the environment configuration in Figure 21.

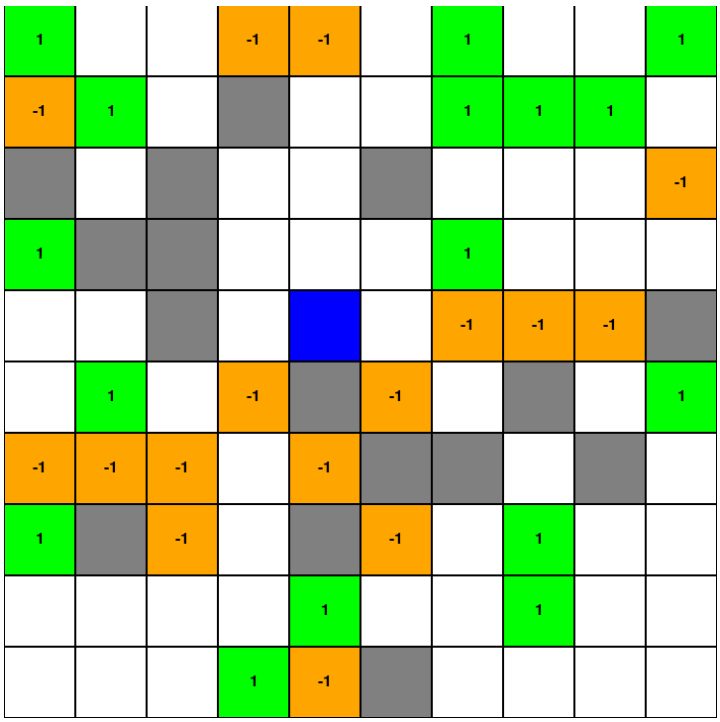


Figure 21. New increased-size maze environment (10x10)

Figure 21. shows the new environment proposed for increasing the complexity by increasing the size of the maze.

5.1.2. Value Iteration Results

88.46	87.30	86.15	83.83	84.54	86.39	87.52	86.67	87.57	88.71
86.23	87.20	86.07		86.73	86.46	87.61	87.60	87.70	87.57
	86.06		88.83	87.91		86.56	86.46	86.43	85.36
100.00			90.12	88.96	87.88	87.71	86.33	85.31	84.36
98.54	97.25		91.45	90.12	88.56	86.26	84.30	84.74	
97.25	97.11	95.55	92.82		86.10	85.24		87.12	88.57
94.76	94.51	93.13	91.88	89.49			85.55		87.42
94.82		90.71	90.69		85.83	85.76	86.69	85.55	86.20
93.42	92.08	90.77	89.74	89.65	88.19	86.77	86.77	85.52	85.12
92.08	91.01	89.93	90.02	87.87		85.63	85.53	84.54	84.07

Figure 22. . Environment states' utilities (Value Iteration for Increased-Size Maze)

Figure 22. shows the environment states' utilities after running the Value Iteration algorithm for the Increased-Size Maze environment.

↑	←	←	←	↓	→	↓	↓	→	↑
↑	↑	←		↓	→	→	→	↑	↑
	↑		↓	↓		↓	↑	↑	↑
↑			↓	←	←	←	←	↑	↑
↑	←		↓	←	←	←	↑	↓	
↑	↑	←	←		↑	↑		→	→
↑	↑	↑	←	←			↓		↑
↑		↑	↑		↓	↓	↓	←	↑
↑	←	←	←	←	←	←	←	←	↑
↑	←	←	←	←		↑	↑	←	↑

Figure 23. Optimal Policy (Value Iteration for Increase-Size Maze)

Figure 23. shows the optimal policy, the result after running the Value Iteration algorithm for the Increase-Size Maze.

5.1.3. Policy Iteration Results

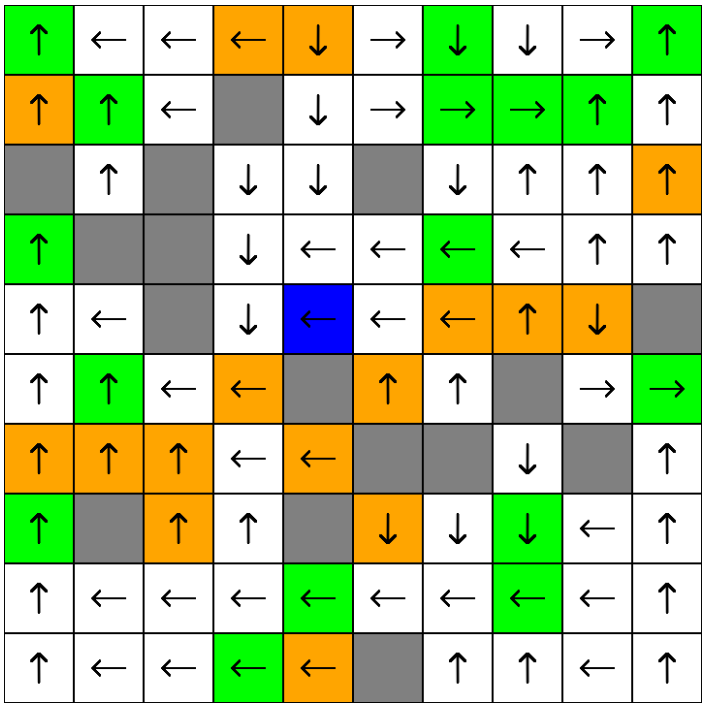


Figure 24. Optimal Policy (Policy Iteration for Increase-Size Maze)

Figure 24. shows the optimal policy, the result after running the Policy Iteration algorithm for the Increase-Size Maze.

88.42	87.26	86.11	83.79	84.54	86.38	87.51	86.66	87.56	88.70
86.19	87.16	86.03		86.73	86.45	87.60	87.59	87.69	87.56
	86.02		88.83	87.91		86.56	86.45	86.42	85.35
100.00			90.12	88.96	87.88	87.71	86.33	85.30	84.36
98.54	97.25		91.45	90.12	88.56	86.26	84.30	84.74	
97.25	97.11	95.55	92.82		86.10	85.24		87.12	88.57
94.76	94.51	93.13	91.88	89.49			85.56		87.42
94.82		90.71	90.69		85.83	85.76	86.69	85.55	86.20
93.42	92.08	90.77	89.74	89.65	88.19	86.77	86.77	85.52	85.12
92.08	91.01	89.93	90.02	87.87		85.63	85.53	84.54	84.07

Figure 25 . Environment states' utilities (Policy Iteration for Increased-Size Maze)

Figure 25. shows the environment states' utilities after running the Policy Iteration algorithm for the Increased-Size Maze environment.

5.1.4. Results Comparison and Conclusion

After inspection Figure 24. and Figure 23. we observe that the optimal policies give as results from the Value Iteration and the Policy Iteration algorithms are identical. Also, we observe similar insight for Figure 22. and Figure 25. the final utility values from the two algorithms are very similar with difference less than 0.10.

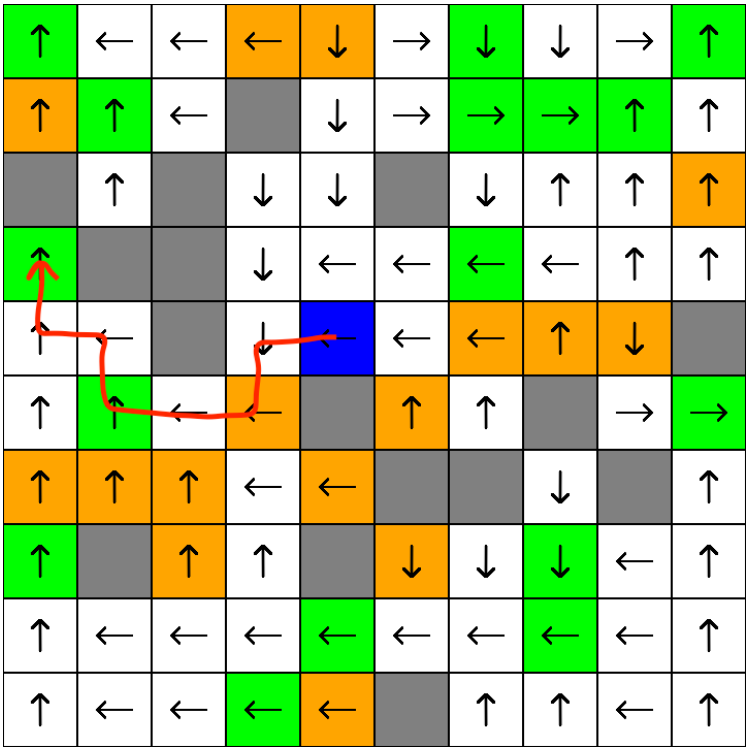


Figure 26. The correct path over the Figure 24.

In Figure 26. we mark the correct path on Figure 24. and we can see that it perfectly aligns with the policy from the two algorithms. The highlighted red path is the correct one because when the agent is located at (0, 3) it can try to go up and because the movement is infeasible in any possible direction, it will remain at the same state and gain a reward. The path goes through a penalty cell but in the long-term goal this is better because when it reaches (0, 3), the agent has 100% probability of keep getting rewards. This strategy is better than going to the upper right corner because the movements there are not bounded and may cause the agent to exit the reward state.

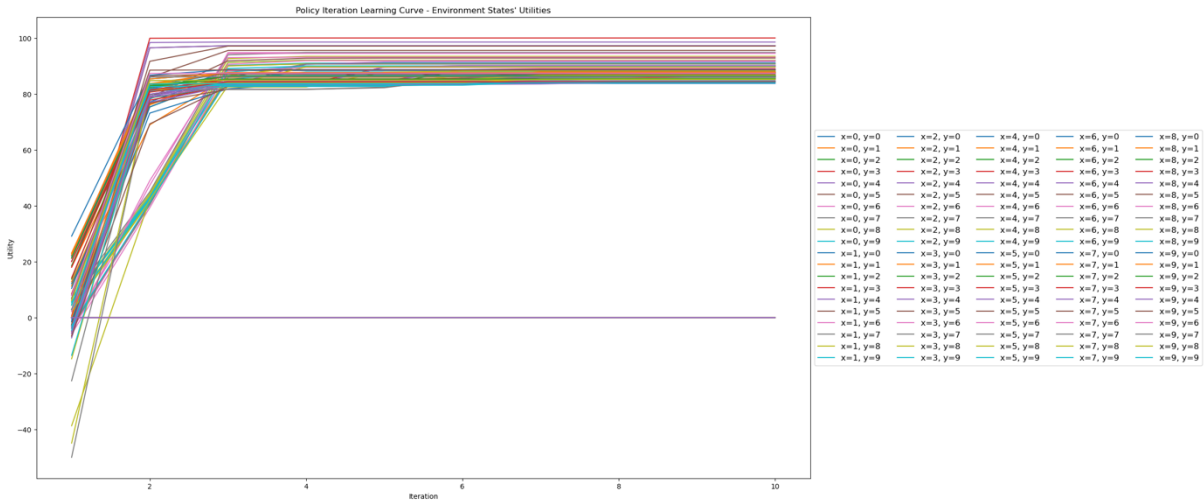


Figure 27. Policy Iteration Learning Curve – Environment States' Utilities (Increased-Size Maze)

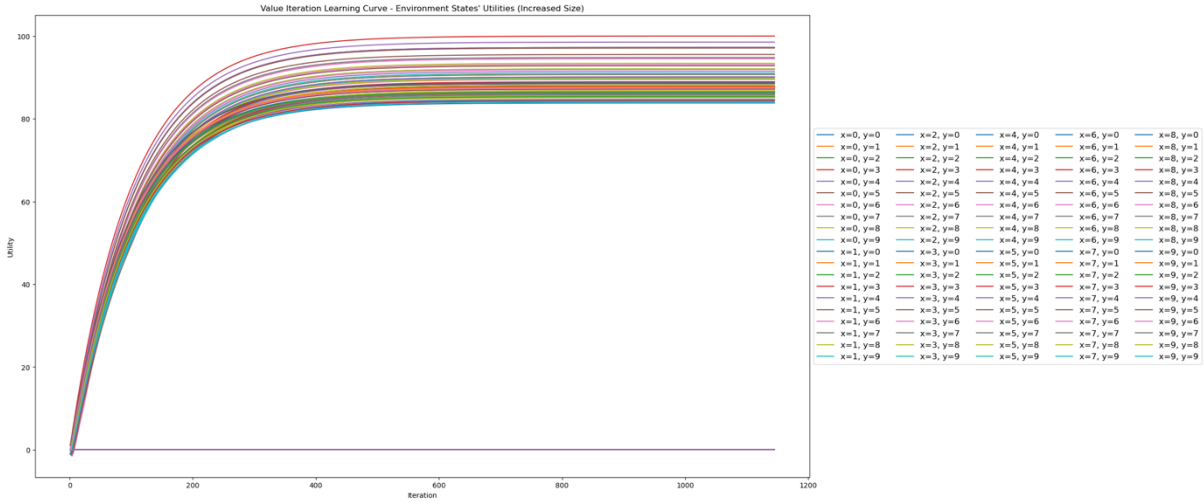


Figure 28. Value Iteration Learning Curve – Environment States' Utilities (Increased-Size Maze)

Comparing the plots from Figure 27. and Figure 28. to the ones in Task 1, we can observe that there is almost no difference in the number of iterations that the algorithms needed to converge.

5.2. Labyrinth Maze

Another way to design a more complex environment is to create a labyrinth maze environment and put penalty cells on the path to the rewards.

5.2.1. Proposed Maze Environment

Thus, we will consider the new size of 10x10 and the environment configuration in Figure 29.

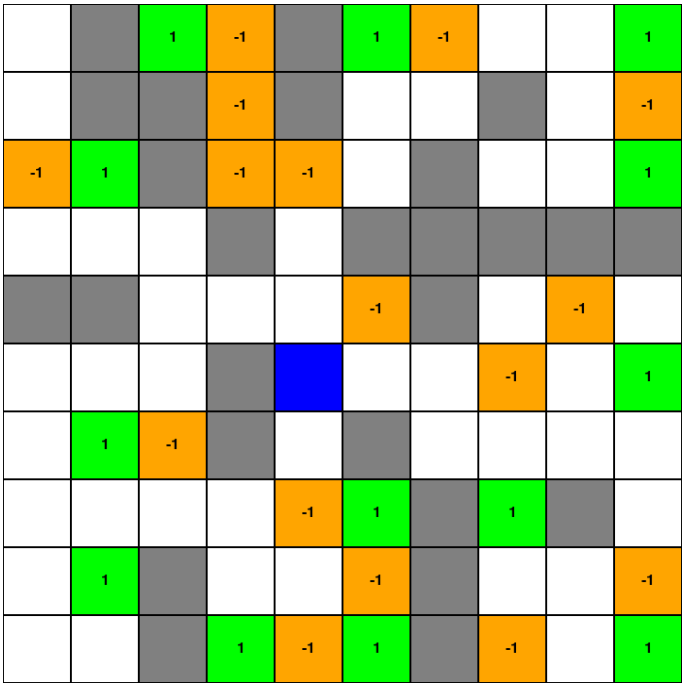


Figure 29. New labyrinth maze environment (10x10)

5.2.2. Value Iteration Results

81.59		100.00	97.20		88.06	85.71	84.96	86.08	87.36
82.67			94.74		86.85	85.72		84.96	85.01
83.77	86.23		92.00	89.46	88.14		84.96	86.08	87.36
83.74	84.84	83.79		88.30					
		84.26	85.59	86.72	84.39		79.83	79.62	81.62
80.90	81.96	83.03		85.45	84.34	83.11	80.91	81.61	82.95
80.99	82.06	80.98		84.34		82.10	82.66	81.58	82.55
81.06	82.10	83.13	84.50	83.42	85.47		83.96		83.77
81.42	82.66		85.92	84.67	84.00		83.64	84.71	84.88
80.51	81.46		87.21	84.85	86.24		83.62	85.94	87.23

Figure 30 . Environment states' utilities (Value Iteration for Labyrinth Maze)

Figure 30. shows the environment states' utilities after running the Value Iteration algorithm for the Labyrinth Maze environment.

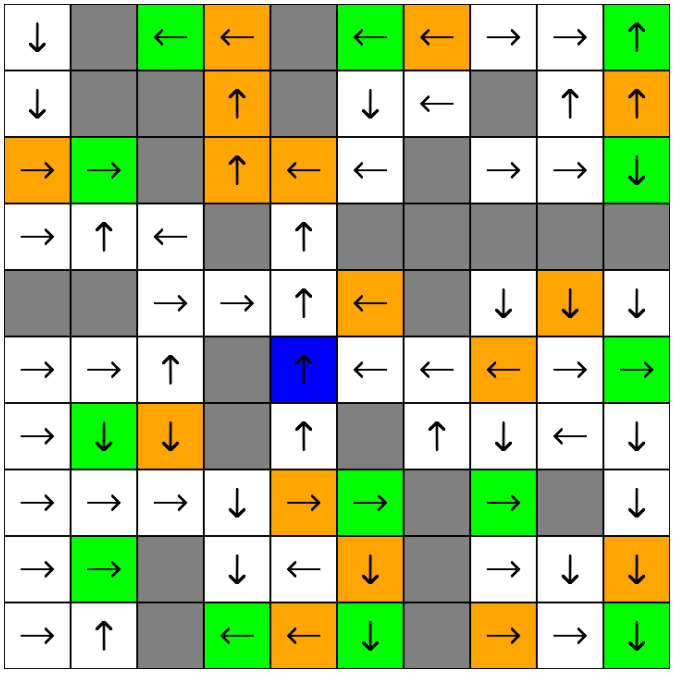


Figure 31. Optimal Policy (Value Iteration for Labyrinth Maze)

Figure 31. shows the optimal policy, the result after running the Value Iteration algorithm for the Labyrinth Maze.

5.2.3. Policy Iteration Results

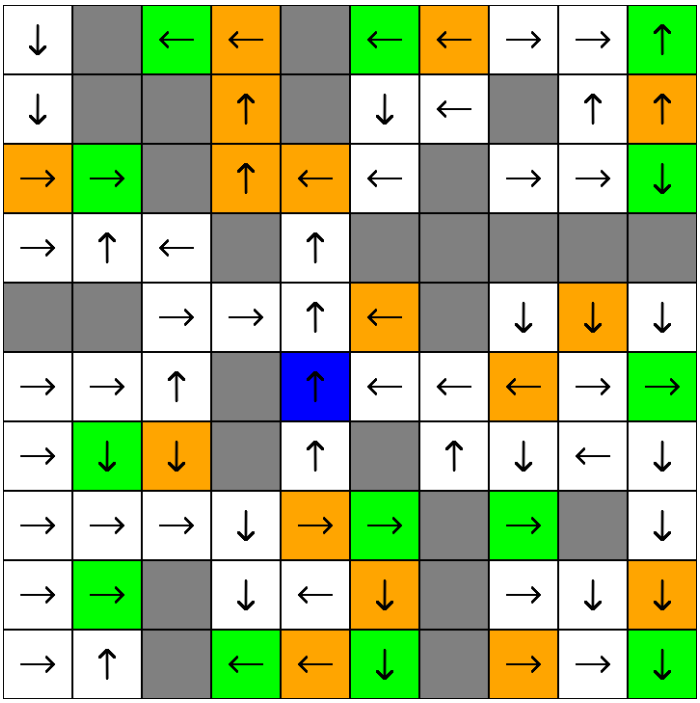


Figure 32. Optimal Policy (Policy Iteration for Labyrinth Maze)

Figure 32. shows the optimal policy, the result after running the Policy Iteration algorithm for the Labyrinth Maze.

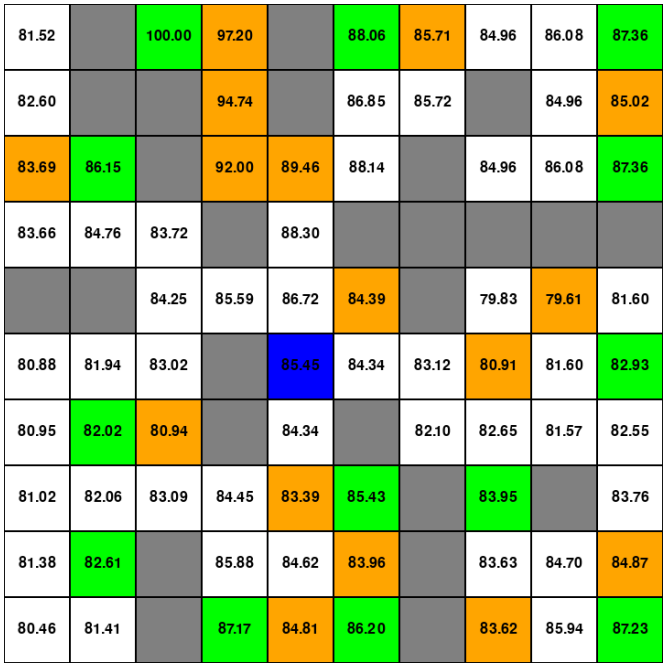


Figure 33 . Environment states' utilities (Policy Iteration for Labyrinth Maze)

Figure 33. shows the environment states' utilities after running the Policy Iteration algorithm for the Labyrinth Maze environment.

5.2.4. Results Comparison and Conclusion

Comparing Figure 31. And Figure 32 we again cannot spot any difference in the optimal policies given by the algorithms. As for Figure 30. And Figure 33 the differences are minor.

In Figure 34. we have marked the optimal path and can see that it matches the proposed ones from the Value Iteration and Policy Iteration algorithms (Figure 32. & Figure 31.). This path is the optimal one despite it goes through a lot of penalty cells and despite that there are reward cells with no penalties on their paths (like (1, 8)). The path highlighted in red is the correct one because it finds the only state in which the agent movement in one direction is bounded despite the transition model. Thus, this state will be the best one in the long term because the agent can stuck itself on a reward cell.

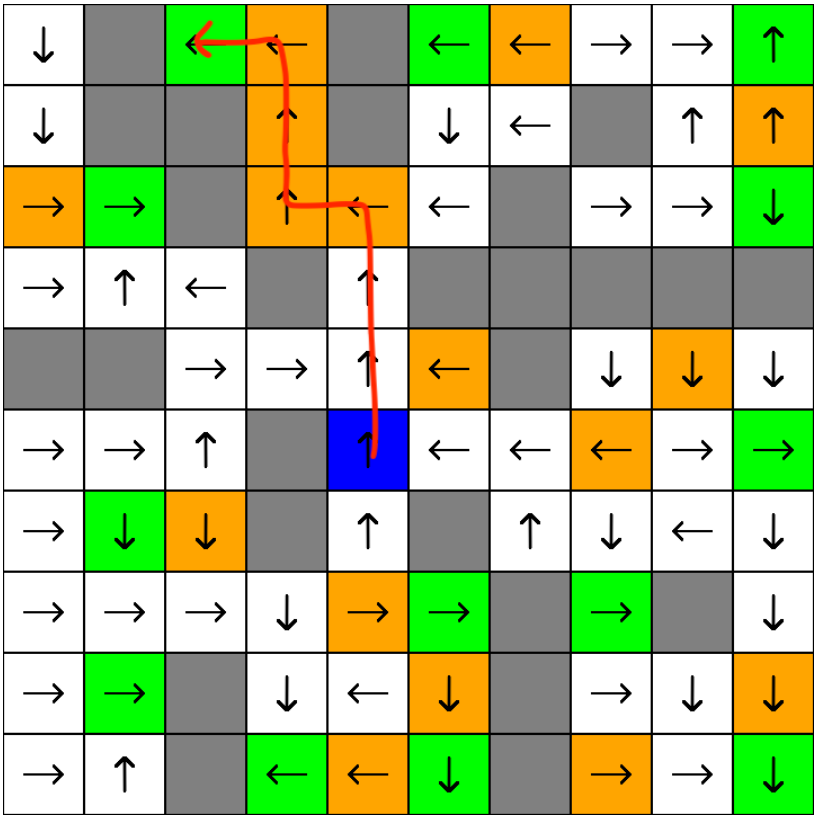


Figure 34. The correct path over the Figure 32.

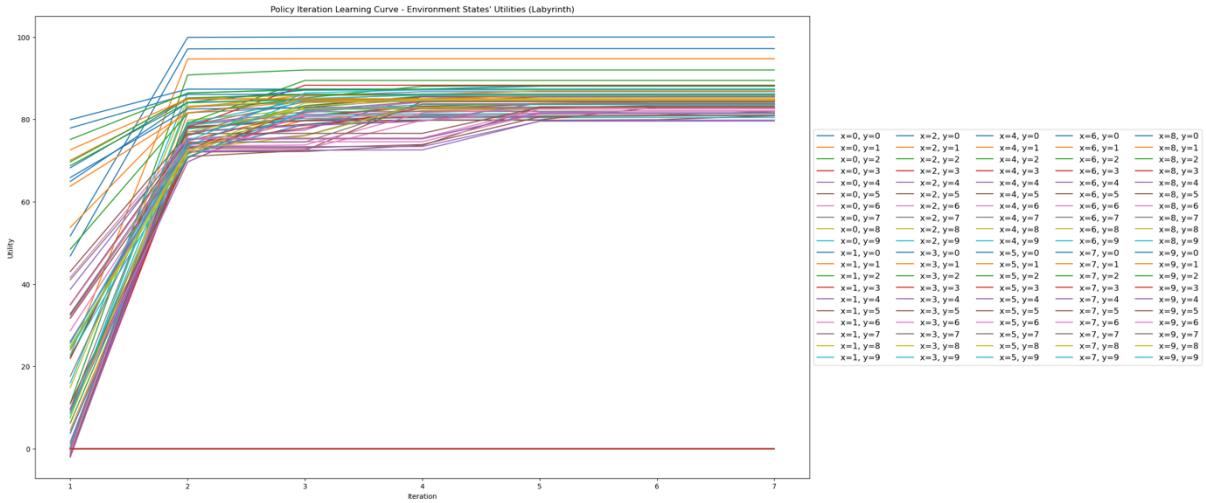


Figure 35. Policy Iteration Learning Curve – Environment States' Utilities (Labyrinth Maze)

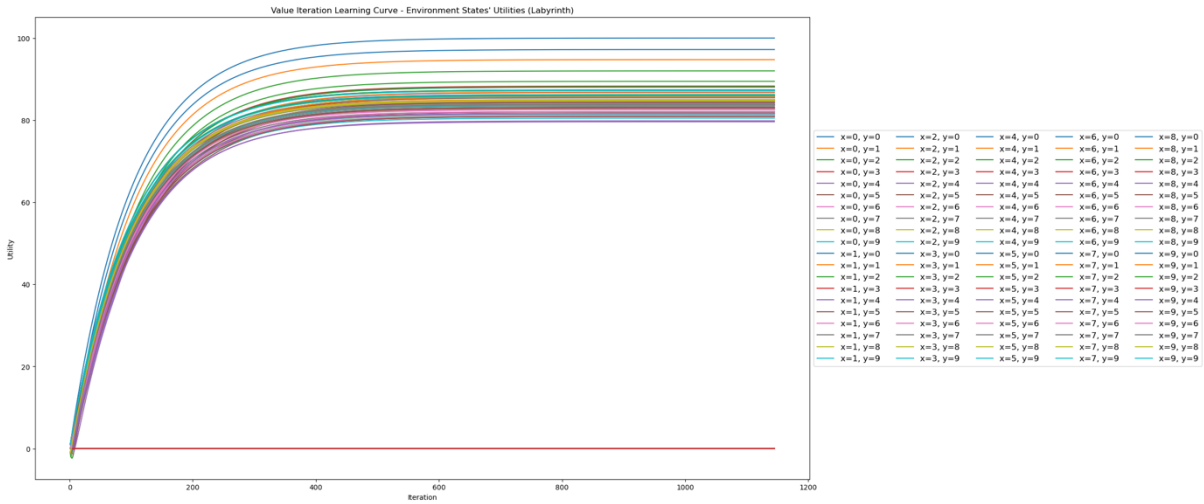


Figure 36. Value Iteration Learning Curve – Environment States' Utilities (Labyrinth Maze)

Comparing the plots from Figure 35. and Figure 36. to the ones in Task 1, we can observe that there is almost no difference in the number of iterations that the algorithms needed to converge.

5.3. Blockages Maze

Lastly, we will propose an environment where the agent starts from one side and has to reach the rewards on the other side while there are long wall sequences with some gates through which the agent go cross.

5.3.1. Proposed Maze Environment

Thus, we will consider the size of 10x10 and the environment configuration in Figure 37.

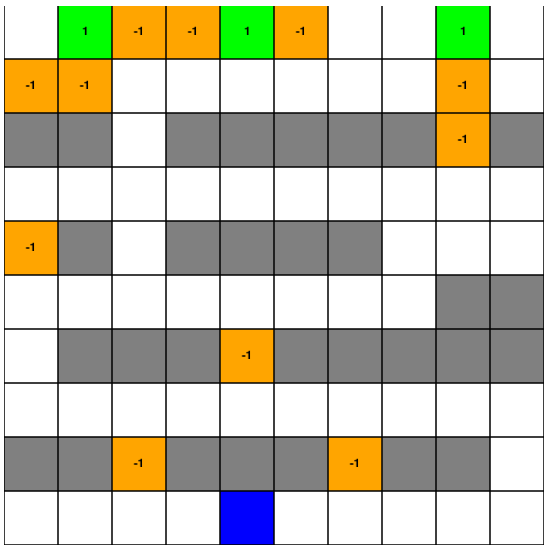


Figure 37. New blockages maze environment (10x10)

5.3.2. Value Iteration Results

69.88	71.06	69.31	71.33	73.47	72.93	75.08	76.19	77.34	76.20
67.89	68.96	70.32	71.50	72.47	73.20	74.21	75.09	75.13	75.19
		69.39						72.95	
66.21	67.35	68.25	67.35	67.92	68.83	69.75	70.68	71.73	70.68
64.13		67.35					69.84	70.60	69.84
64.37	65.37	66.25	65.37	65.86	67.00	67.90	68.81		
63.52				63.80					
62.57	61.74	60.84	61.91	62.75	61.91	60.84	60.04	59.24	58.35
		58.84				58.84			57.57
56.33	57.09	57.86	57.09	56.33	57.09	57.86	57.09	56.33	56.75

Figure 38 . Environment states' utilities (Value Iteration for Blockages Maze)

Figure 38. shows the environment states' utilities after running the Value Iteration algorithm for the Blockages Maze environment.

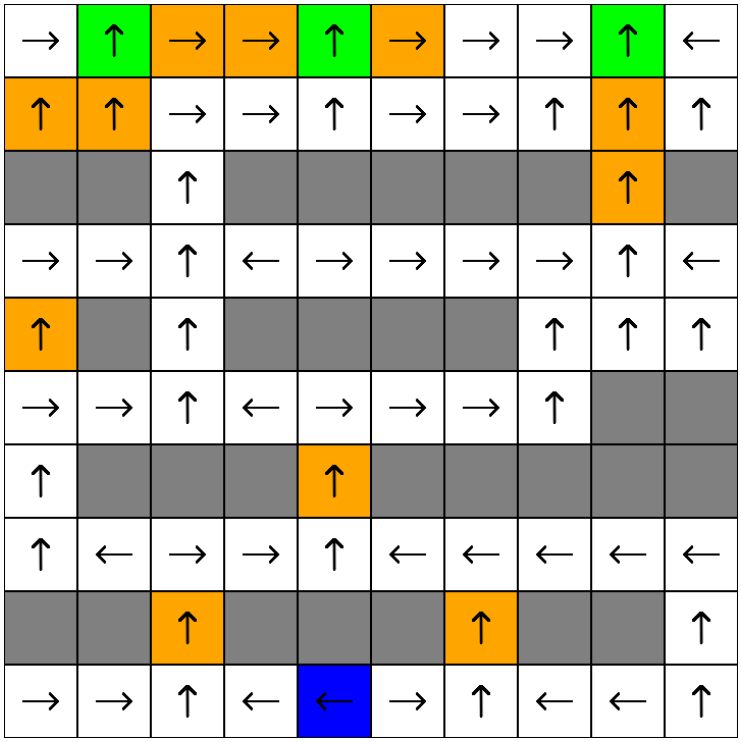


Figure 39. Optimal Policy (Value Iteration for Blockages Maze)

Figure 39. shows the optimal policy, the result after running the Value Iteration algorithm for the Blockages Maze.

5.3.3. Policy Iteration Results

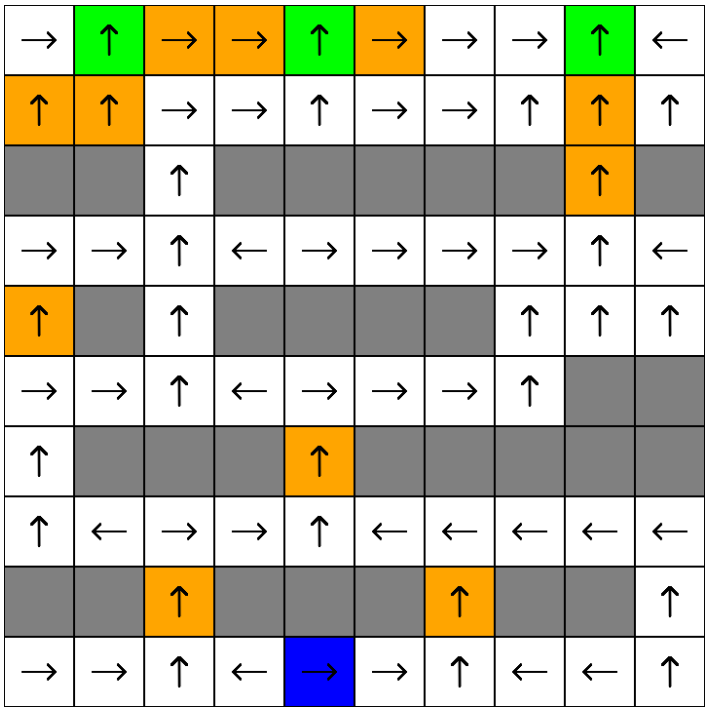


Figure 40. Optimal Policy (Policy Iteration for Blockages Maze)

Figure 40. shows the optimal policy, the result after running the Policy Iteration algorithm for the Blockages Maze.

69.84	71.02	69.28	71.29	73.44	72.90	75.05	76.16	77.31	76.17
67.86	68.93	70.29	71.47	72.44	73.17	74.18	75.05	75.10	75.16
		69.36						72.91	
66.17	67.32	68.22	67.32	67.89	68.80	69.72	70.65	71.70	70.65
64.10		67.32					69.81	70.57	69.81
64.34	65.34	66.21	65.34	65.83	66.97	67.87	68.78		
63.49				63.76					
62.54	61.71	60.81	61.88	62.71	61.88	60.81	60.00	59.20	58.32
		58.81				58.81			57.54
56.30	57.06	57.83	57.06	56.30	57.06	57.83	57.06	56.30	56.72

Figure 41. Environment states' utilities (Policy Iteration for Blockages Maze)

Figure 41. shows the environment states' utilities after running the Policy Iteration algorithm for the Blockages Maze environment.

5.3.4. Results Comparison and Conclusion

Comparing Figure 39. And Figure 40 we can spot difference in the provided optimal policies from the Value Iteration and Policy Iteration algorithms. The difference is that Value Iteration suggests 'left' action for the initial state, while Policy Iteration suggests 'right' action.

When we compare the environment states' utilities (Figure 41. & Figure 38.), we see that the values are slightly different but the relationship between the states is the same. Also, we can compare the states on the sides of the initial state, in both figures we can see the utilities are equal when rounded. This explains the difference in the policies. However, this time we do not have a state utility of 100. The maximum utility in both figures is around 77.

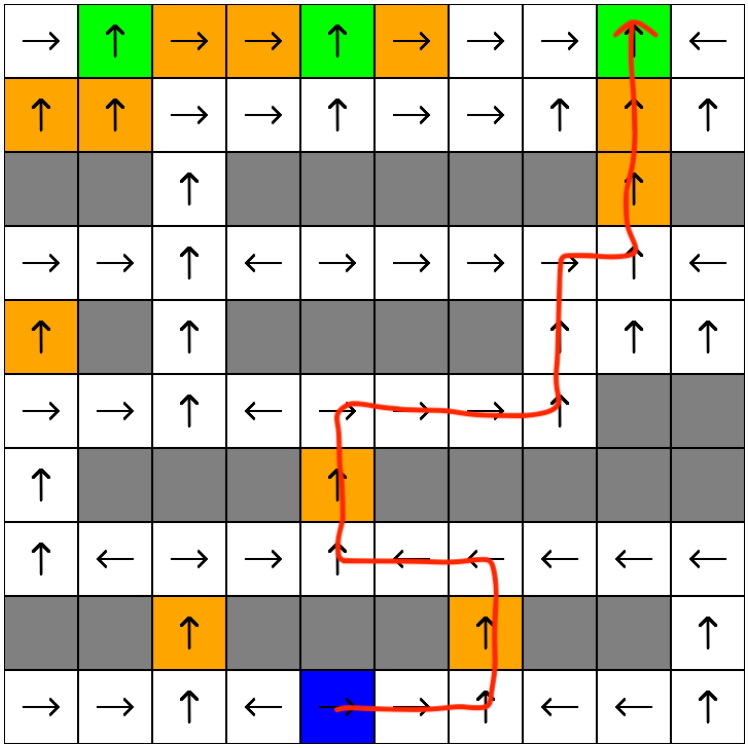


Figure 42. The correct path over the Figure 40.

In Figure 42. we are drawing the correct optimal path in red over the optimal policy from the Policy Iteration algorithm (Figure 40.). We can see that the policy is matching the

optimal path, thus the Policy Iteration algorithm is able to solve this Blockage Maze environment. After further thoughts, the policy from the Value Iteration is also optimal as the only difference will be that the agent will go through the left gate of the first wall sequence (symmetrical start). However, this Blockage Maze environment is more complex as it is the first one to disturb our algorithms and cause difference in the policies. Moreover, it is the first environment to decrease the state utilities so significantly.

Significant observation to be made is the fact that this is also the first environment where there is not a bounded state. In this environment, the agent will not stuck, it will continue to perform actions.

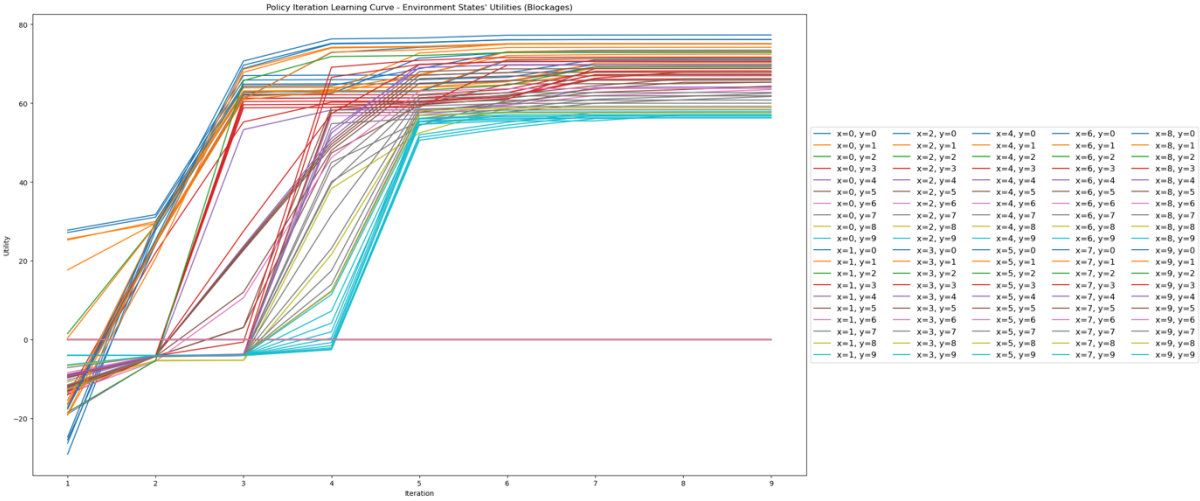


Figure 43. Policy Iteration Learning Curve – Environment States' Utilities (Blockage Maze)

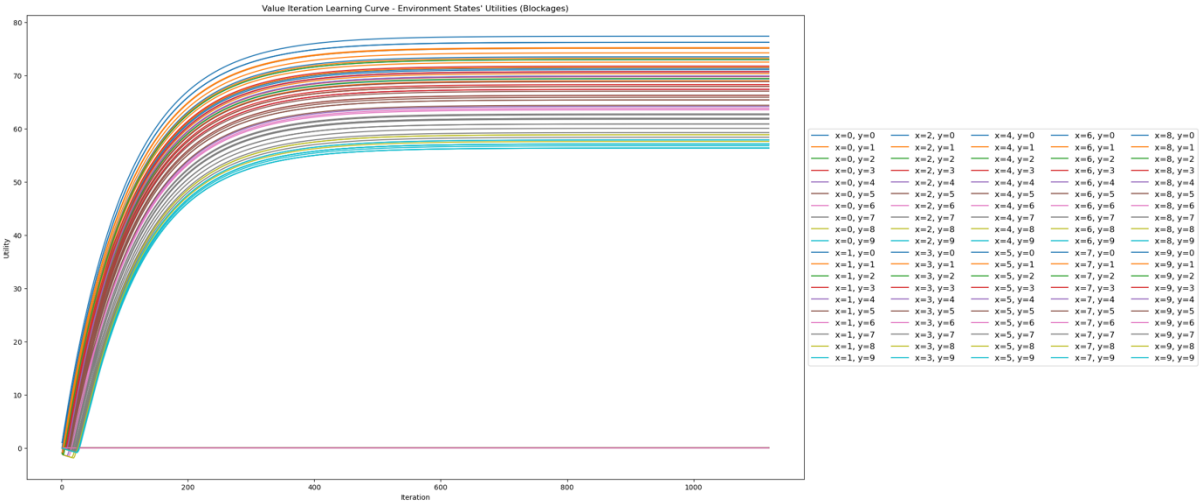


Figure 44. Value Iteration Learning Curve – Environment States' Utilities (Blockage Maze)

However, when we compare Figure 43. and Figure 44. we again see that the number of iterations needed for converging does not differ much from the number of iterations needed for the algorithms in Task 1. The converging for all these new environments is still similar to the one in Task 1 because the environments are still static (the cells do not change)

5.4. Dynamic Environment (Additional)

For all the previous environments we observed only static mazes where the cells do not change over time or after a certain action. The environment that will challenge the agent the most and will increase the converging time is a dynamic environment. In this environment, the rewards may disappear when collected or over time. This will eliminate any chance of the agent finding a state to be stuck in a reward cell or a region with several rewards and less penalties in which to move.

6. Interactive Environment – Mini-Game (Additional)

As this project implements visualization through drawing graphics using the PyGame Python library, an interactive environment (mini-game) where the user can control the agent was created. This environment simulates the maze environment, and a user can control the agent and test the maze capabilities. The environment also consists of a score counter that replicates the one defined in the assignment. The interface of the mini-game can be seen in Figure 1.