Własne środowisko - sprawozdanie

Szymon Frączek, Piotr Gąsiorek

Instalacja środowiska

Instalujemy wymagane biblioteki jak gymnasium, numpy, matplotlib.

In [ ]:
```
import gym
from gym import spaces
import numpy as np
import pygame
import pickle
import matplotlib.pyplot as plt
import random
```

Implementacja algorytmu

Środowisko reprezentuje labirynt jako dwuwymiarową tablicę NumPy, gdzie różne symbole oznaczają start ('S'), cel ('G'), ściany ('#'), puste przestrzenie ('.'), ruchome przeszkody wraz z kierunkiem ruchu ('X^>') oraz ściany pojawiające się randomowo ($).

Agent w tym środowisku ma możliwość poruszania się w czterech kierunkach (góra, dół, lewo, prawo) i otrzymuje nagrodę w wysokości 1.0, kiedy dotrze do celu. Jeśli nie osiągnie celu, otrzymuje nagrodę negatywną w każdym kroku, zachęcając do szybkiego dotarcia do celu.

Klasa `MazeGameEnv` zawiera następujące kluczowe metody:

- `__init__` : Inicjalizacja środowiska, określenie stanu początkowego, celu i konfiguracja przestrzeni akcji i obserwacji.
- `reset` : Resetuje pozycję agenta do pozycji początkowej i inicjuje ruchome przeszkody.
- `locate_X` : Znajduje i przechowuje pozycje ruchomych przeszkód 'X' w labiryncie.
- `step` : Aktualizuje pozycję agenta na podstawie wykonanej akcji i zwraca nowy stan, nagrodę, informację o zakończeniu i dodatkowe informacje.
- `_update_Xes` : Aktualizuje pozycje ruchomych przeszkód.
- `_is_valid_position` : Sprawdza, czy nowa pozycja jest prawidłowa, tzn. nie jest ścianą ani poza granicami labiryntu.

- `_translate_action` : Przetłumaczenie numeru akcji na konkretny ruch w labiryncie.
- `_pos_to_index` : Konwertuje pozycję (x, y) na unikalny indeks używany w tabeli Q-learning.
- `render` : Wizualizuje aktualny stan labiryntu, rysując odpowiednie elementy w oknie Pygame.

Środowisko obsługuje również ruchome przeszkody 'X', które mogą zmieniać swoje położenie w labiryncie. Przy każdym wywołaniu metody `step` , przeszkody te mogą przemieścić się o jedną komórkę w losowym kierunku, pod warunkiem, że nowa pozycja nie jest ścianą, inną ruchomą przeszkodą lub pozycją agenta. Ta funkcjonalność dodaje dynamiki do problemu labiryntu, ponieważ agent musi nie tylko nauczyć się statycznego rozwiązania, ale także reagować na zmiany w środowisku.

Wybraliśmy algorytm Q-learning, będący bezmodelową metodą uczenia ze wzmocnieniem, pozwalającą agentowi ocenić wartość każdej z możliwych akcji bez konieczności posiadania modelu środowiska. Użyto zmiennego współczynnika uczenia i współczynnika, który pomaga określić, jak bardzo cenić przyszłe nagrody.

In [ ]:
```
class MazeGameEnv(gym.Env):

    def __init__(self, maze):
        super().__init__()
        self.maze = np.array(maze) # 2d array
        self.start_pos = (np.where(self.maze == 'S')[0][0], np.where(self
        self.goal_pos = (np.where(self.maze == 'G')[0][0], np.where(self.
        self.current_pos = self.start_pos
        self.num_rows, self.num_cols = self.maze.shape
        self.X_pos = self.locate_X()
        self.move_count = 0

        self.action_space = spaces.Discrete(4)
        self.observation_space = gym.spaces.Discrete(self.num_rows * self

        # print(self.X_pos)

        pygame.init()
        self.cell_size = 125
        self.screen = pygame.display.set_mode((self.num_cols * self.cell_

    def reset(self, seed=None, options=None):
        """
            Reset agent position to start position
        """
        super().reset(seed=seed)
        self._init_temp_walls()
        self.current_pos = self.start_pos
```

```
        state_index = self._pos_to_index(self.current_pos)
        self.move_count = 0
        return state_index, {}

    def locate_X(self):
        temp_X_pos = []
        for row in range(self.num_rows):
            for col in range(self.num_cols):
                if self.maze[row, col].startswith('X'):
                    temp_X_pos.append((row, col, self.maze[row, col]))
        return temp_X_pos

    def step(self, action):
        """
            Updates agent's position according to the action taken and pr
        """
        self._update_Xes()
        moved = False

        new_pos = np.array(self.current_pos)
        direction = self._translate_action(action)
        new_pos = new_pos[0] + direction[0], new_pos[1] + direction[1]

        if self._is_valid_position(new_pos):
            self.current_pos = new_pos
            moved = True

        if np.array_equal(np.array(self.current_pos), np.array(self.goal_
            reward = 1.0
            done = True
        # elif self.move_count > self.num_rows * self.num_cols * 5:
        #     reward =
        #     done = True
        else:
            reward = -0.1/(self.num_rows * self.num_cols)
            done = False

        state_index = self._pos_to_index(self.current_pos)
        self.move_count += 1
        return state_index, reward, done, moved, {},

    def _init_temp_walls(self):
        for row in range(self.num_rows):
            for col in range(self.num_cols):
                if self.maze[row, col].endswith('$'):
                    rand = random.randint(0, 1)
                    if rand == 0:
                        self.maze[row, col] = '.$'
                    else:
                        self.maze[row, col] = '#$'

    def _is_valid_position(self, pos):
        row, col = pos
```

```
        if row < 0 or col < 0 or row >= self.num_rows or col >= self.num_
            return False

        if self.maze[row, col].startswith('#'):
            return False

        if self.maze[row, col].startswith('X'):
            return False

        return True

    def _translate_action(self, action):
        if action == 0:
            return (-1, 0) # Up
        if action == 1:
            return (1, 0) # Down
        if action == 2:
            return (0, -1) # Left
        if action == 3:
            return (0, 1) # Right
        else:
            return (0, 0)

    def _is_valid_X_position(self, pos):
        row, col, _ = pos

        if row < 0 or col < 0 or row >= self.num_rows or col >= self.num_
            return False

        if row == self.current_pos[0] and col == self.current_pos[1]:
            return False

        if self.maze[row, col].startswith('#') or self.maze[row, col].sta
            return False

        return True

    def _update_Xes(self):
        for X_pos in self.X_pos:
            valid = False
            new_X_pos = X_pos
            counter = 0
            while valid == False:
                rand_move = 1
                if X_pos[2].endswith('^'):
                    rand_dir = random.randint(0, 1)
                elif X_pos[2].endswith('>'):
                    rand_dir = random.randint(2, 3)
                else :
                    rand_dir = random.randint(0, 3)

                direction = self._translate_action(rand_dir)
```

```
                new_X_pos = X_pos[0] + direction[0] * rand_move, X_pos[1]
                if self._is_valid_X_position(new_X_pos):
                    valid = True
                    self.maze[X_pos[0], X_pos[1]] = '.'
                    self.maze[new_X_pos[0], new_X_pos[1]] = X_pos[2]

                    self.X_pos.remove(X_pos)
                    self.X_pos.append(new_X_pos)
                else:
                    counter += 1
                    if counter > 5:
                        break



    def _pos_to_index(self, pos):
        return pos[0] * self.num_cols + pos[1]

    def render(self):
        """
            Render game environment using pygame by drawing elements for
            You can simply print the maze grid as well, no necessary requ
        """
        self.screen.fill((255, 255, 255))

        for row in range(self.num_rows):
            for col in range(self.num_cols):
                cell_left = col * self.cell_size
                cell_top = row * self.cell_size

                # try:
                #     print(np.array(self.current_pos)==np.array([row,col
                # except Exception as e:
                #     print('Initial state')

                if self.maze[row, col].startswith('#'):
                    pygame.draw.rect(self.screen, (0, 0, 0), (cell_left,
                elif self.maze[row, col] == 'S':
                    pygame.draw.rect(self.screen, (0, 255, 0), (cell_left
                elif self.maze[row, col] == 'G':
                    pygame.draw.rect(self.screen, (255, 0, 0), (cell_left
                elif self.maze[row, col].startswith('X'):
                    pygame.draw.rect(self.screen, (125, 125, 125), (cell_

                if np.array_equal(np.array(self.current_pos), np.array([r
                    pygame.draw.rect(self.screen, (0, 0, 255), (cell_left

        pygame.display.update()
```

Rejestracja środowiska

In [ ]:
```
gym.register(
    id='MazeGame-v0',
    entry_point=MazeGameEnv,
    kwargs={'maze': None}
)
```

Funkcja do trenowania oraz testowania agenta

Kod przedstawia funkcję run, która służy do uruchomienia sekwencji epizodów w środowisku labiryntu w celu trenowania agenta za pomocą algorytmu Q-learning. Proces może odbywać się w trybie treningowym (is_training=True), gdzie agent aktualizuje tabelę wartości Q w oparciu o otrzymane nagrody i eksploruje środowisko, lub w trybie używania wcześniej nauczonego modelu (is_training=False), gdzie agent wykorzystuje istniejącą tabelę Q do nawigacji po labiryncie.

Środowisko:

Tworzone jest środowisko gry labiryntowej, w której agent musi znaleźć drogę od punktu startowego 'S' do celu 'G', unikając ścian '#' i dynamicznie poruszających się przeszkód 'X'. Dodatkowo, niektóre puste miejsca '.' mają możliwość tymczasowego stania się ścianami, co dodaje do gry element stochastyczności.

Tabela Q:

Inicjalizowana jest tabela Q, będąca dwuwymiarową tablicą, gdzie wiersze odpowiadają stanom środowiska, a kolumny — możliwym akcjom.

Parametry uczenia:

Ustawiane są hiperparametry uczenia takie jak współczynnik uczenia (learning_rate_a) i czynnik dyskontujący (discount_factor_g). Epsilon steruje równowagą między eksploracją a eksploatacją podczas procesu uczenia.

Przebieg epizodu:

Każdy epizod rozpoczyna się od resetowania środowiska. Następnie, w pętli, agent wybiera akcję, wykonuje krok w środowisku, otrzymuje nagrodę i aktualizuje tabelę Q, jeżeli jest w trybie treningowym. Jeśli render jest True, stan gry jest wyświetlany po każdym ruchu, co pozwala na wizualną obserwację postępów agenta.

Tryb nie-treningowy:

W trybie nie-treningowym, gdy is_training=False, zamiast losowego wyboru akcji, agent wybiera najlepszą akcję według tabeli Q. Jeśli akcja ta nie powoduje ruchu (agent wchodzi w ścianę lub przeszkodę), wybierana jest kolejna najlepsza akcja, aż do wykonania skutecznego ruchu.

Wyniki:

Nagrody z każdego epizodu są zapisywane, a po zakończeniu wszystkich epizodów są tworzone wykresy średniej i sumarycznej nagrody, które są zapisywane do plików obrazowych.

In [ ]:
```
from copy import deepcopy
def run(episodes, is_training=True, render=False):
    """
    Run the maze problem

    :param episodes: number of episodes to run
    :param is_training: if True, the agent will learn, otherwise it will
    :return: None
    """

    maze = [
        ['S', '$', '.',  'X>', '.',  '.',  '.',  '.'],
        ['.', '#', '.',  '#',  'X^',  '#',  '.',  '.'],
        ['.', '#', '.',  '#',  '.',  '#',  '#',  '$'],
        ['.', '.', '.',  'X>',  '.',  '.',  '.',  '.'],
        ['.', '.', '.',  '$',  '.',  '$',  'X^', '#'],
        ['.', '$', 'X^', '$',  '.',  '$',  '.',  '.'],
        ['.', '.', '.',  '.',  'X>',  '.', '.',  'G'],
        ['$', '$', '$',  '$',  '$',  '$',  '$',  '#']
    ]

    env = gym.make('MazeGame-v0', maze=maze)

    if(is_training):
        q = np.zeros((len(maze) * len(maze[0]), 4))
    else:
        f = open('maze_game.pkl', 'rb')
        q = pickle.load(f)
        f.close()

    learning_rate_a = 0.9
    discount_factor_g = 0.9

    epsilon = 1
    epsilon_decay_rate = 0.0001
    rng = np.random.default_rng()

    rewards_per_episode = np.zeros(episodes)

    for i in range(episodes):
        state = env.reset()[0]
        terminated = False
        truncated = False
        rewards = 0
        not_moved_counter = 0
        while (not terminated):

            if render:
```

```
                pygame.event.get()

            if is_training and rng.random() < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q[state, :])

            if is_training:
                new_state, reward, terminated, moved, _ = env.step(action
            else:
                moved = False
                temp = deepcopy(q)
                while not moved:
                    action = np.argmax(temp[state, :])
                    new_state, reward, terminated, moved, _ = env.step(ac
                    temp[state, action] = -np.inf

            if render:
                env.render()
                pygame.time.wait(200)

            if is_training:
                q[state, action] = q[state, action] + learning_rate_a * (
                    reward + discount_factor_g * np.max(q[new_state, :])
                )

            state = new_state
            rewards += reward

        epsilon = max(epsilon - epsilon_decay_rate, 0)

        if epsilon == 0:
            learning_rate_a = 0.0001

        rewards_per_episode[i] = rewards
        if i % 100 == 0 or render:
            print(f'Episode {i+1}/{episodes}, rewards: {rewards}')

    env.close()

    if is_training:
        f = open('maze_game.pkl','wb')
        pickle.dump(q, f)
        f.close()

    mean_rewards = np.zeros(episodes)
    for t in range(episodes):
        mean_rewards[t] = np.mean(rewards_per_episode[max(0, t-100):(t+1)
    plt.plot(mean_rewards)
    plt.savefig(f'maze_game_mean.png')

    sum_rewards = np.zeros(episodes)
    for t in range(episodes):
```

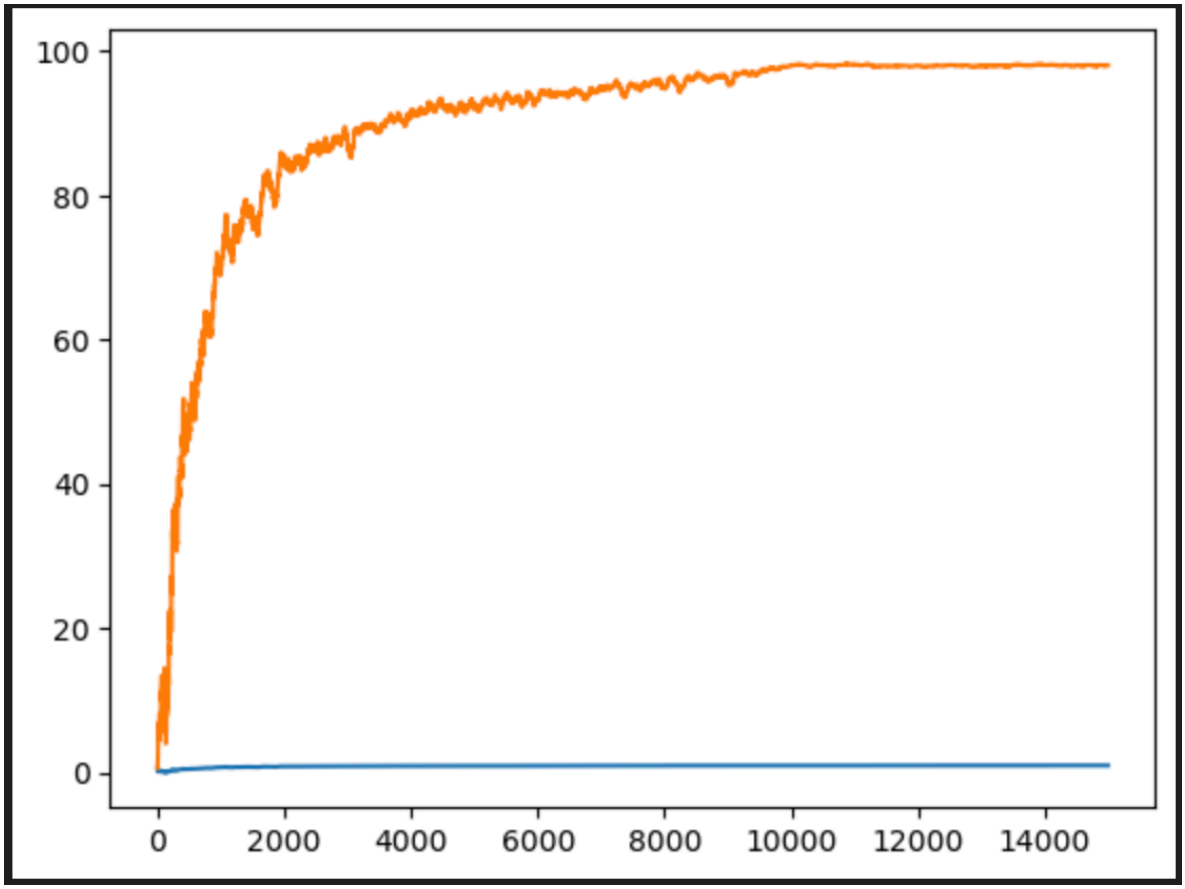
```
        sum_rewards[t] = np.sum(rewards_per_episode[max(0, t-100):(t+1)])
    plt.plot(sum_rewards)
    plt.savefig(f'maze_game_sum.png')
```

Trening agenta

Agent jest trenowany w środowisku labiryntu przez 15000 epizodów.

In [ ]:
```
run(15000, is_training=True, render=False)
```

```
Episode 1/15000, rewards: 0.632812499999999
Episode 101/15000, rewards: 0.17968749999999256
Episode 201/15000, rewards: 0.8718750000000002
Episode 301/15000, rewards: 0.7515625000000006
Episode 401/15000, rewards: 0.4156249999999959
Episode 501/15000, rewards: 0.5124999999999973
Episode 601/15000, rewards: 0.7984375000000005
Episode 701/15000, rewards: 0.9140625
Episode 801/15000, rewards: 0.8765625000000001
Episode 901/15000, rewards: 0.8218750000000004
Episode 1001/15000, rewards: 0.39843749999999567
Episode 1101/15000, rewards: -0.6984374999999703
Episode 1201/15000, rewards: 0.7859375000000005
Episode 1301/15000, rewards: 0.6656249999999995
Episode 1401/15000, rewards: 0.7984375000000005
Episode 1501/15000, rewards: 0.7796875000000005
Episode 1601/15000, rewards: 0.921875
Episode 1701/15000, rewards: 0.6640624999999994
Episode 1801/15000, rewards: 0.8562500000000003
Episode 1901/15000, rewards: 0.9546875
Episode 2001/15000, rewards: 0.7203125000000002
Episode 2101/15000, rewards: 0.8921875000000001
Episode 2201/15000, rewards: 0.7437500000000006
Episode 2301/15000, rewards: 0.8218750000000004
Episode 2401/15000, rewards: 0.8234375000000004
...
Episode 14601/15000, rewards: 0.9734375
Episode 14701/15000, rewards: 0.9734375
Episode 14801/15000, rewards: 0.9562499999999999
Episode 14901/15000, rewards: 0.9703125
```

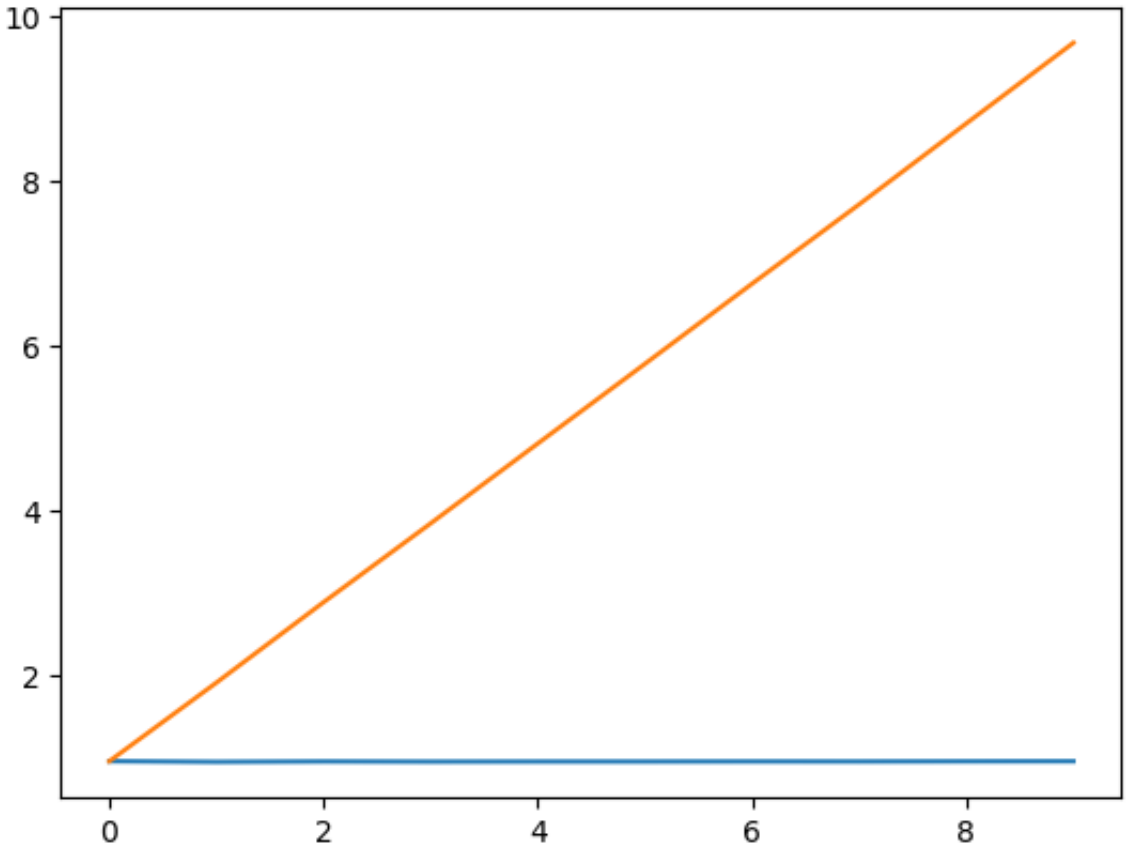


Testowanie agenta

Agent jest testowany w środowisku labiryntu przez 10 epizodów.

In [ ]:
```
run(10, is_training=False, render=True)
```

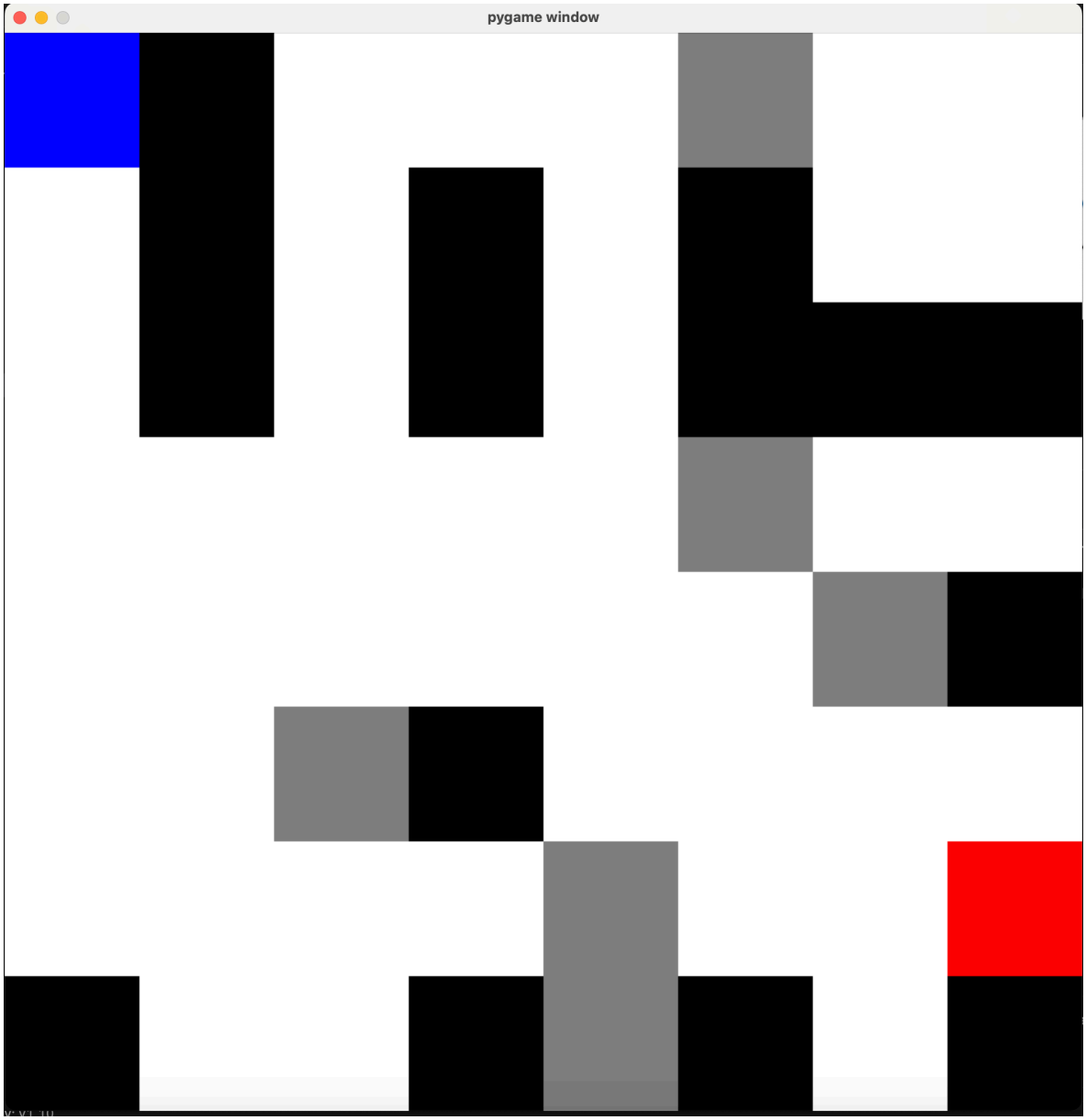
```
Episode 1/10, rewards: 0.96875
Episode 2/10, rewards: 0.953125
Episode 3/10, rewards: 0.975
Episode 4/10, rewards: 0.9562499999999999
Episode 5/10, rewards: 0.96875
Episode 6/10, rewards: 0.965625
Episode 7/10, rewards: 0.971875
Episode 8/10, rewards: 0.9625
Episode 9/10, rewards: 0.98125
Episode 10/10, rewards: 0.978125
```

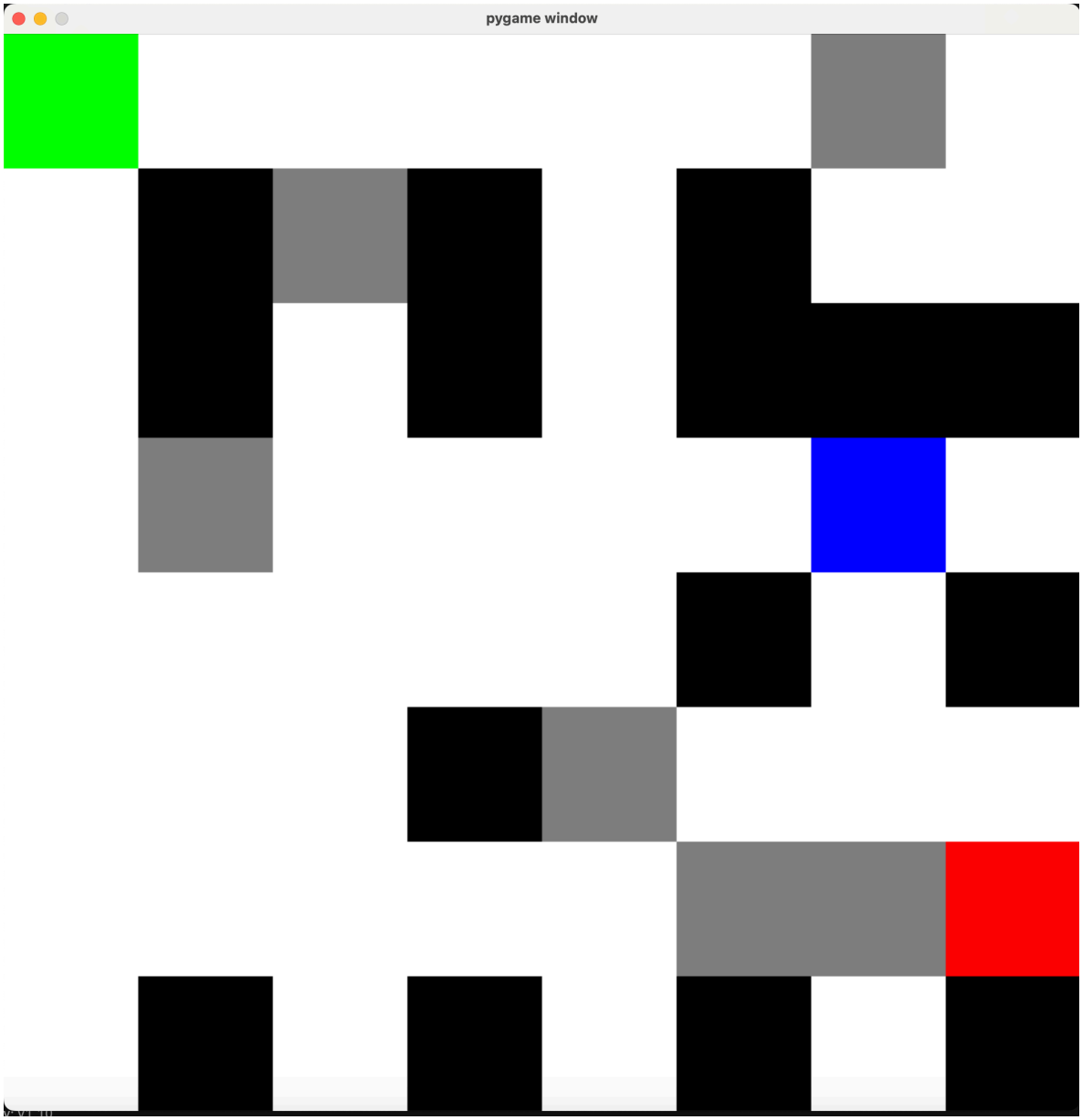


Opis wizuazlizacji

Niebieski kwadrat - agent, Zielony kwadrat - start, Czerwony kwadrat - cel, Czarne kwadraty - ściany, Szare kwadraty - ruchowe przeszkody, Biada kwadraty - puste przestrzenie,

Agent porusza się w czterech kierunkach: góra, dół, lewo, prawo. Celem jest dotarcie do czerwinego kwadratu, unikając ścian i przeszkód. Co kilka kroków, puste miejsca mogą zamienić się w ściany, co dodaje trudności do gry. Agent mając zablokowaną znajduje nową drogę do celu.


pygame window


pygame window