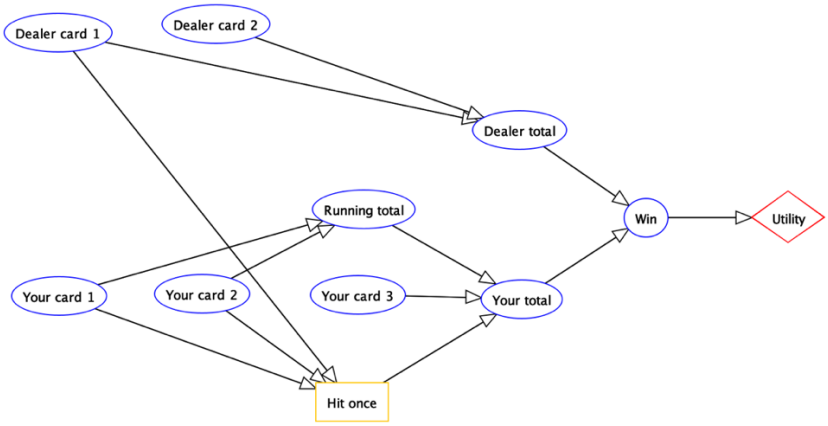
Question 1

a)

- Value of information for Dealer card 1 is the expected utility when knowing the value of Dealer card 1 (i.e., the network with an arc from Dealer Card 1 to the Hit Once) minus the expected utility when not knowing the value of Dealer card 1(i.e., the original network).

- The formula to calculate this is:
  EU(knowing Dealer card 1) - EU(not knowing Dealer card 1)
  = -0.23437 - (-0.25)
  = 0.01563

- To calculate this, we change the network by adding an arc from Dealer card 1 (random variable) to Hit Once (decision), which is shown in the picture below.

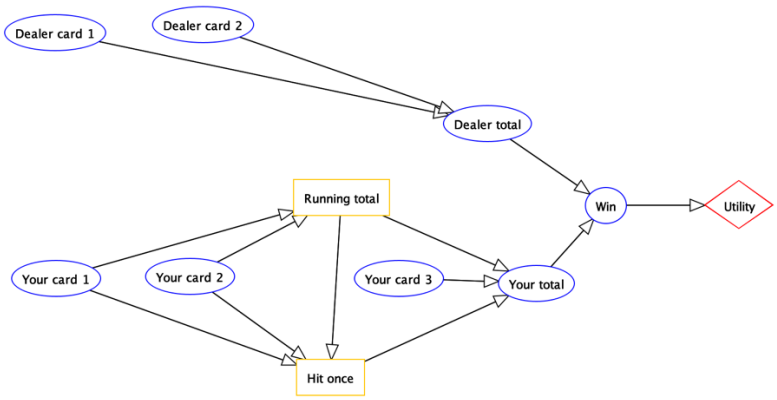


b)

- The value of control for Running Total is the expected utility of the network when we make Running Total a decision variable (i.e., the network with Running Total as a decision variable and with an arc from it to Hit Once) minus the expected utility when Running Total is a random variable (i.e., the original network).

- The formula to calculate this is:
  EU(controlling Running Total) - EU(not controlling Running Total)
  = 0.5 - (-0.25)
  = 0.75

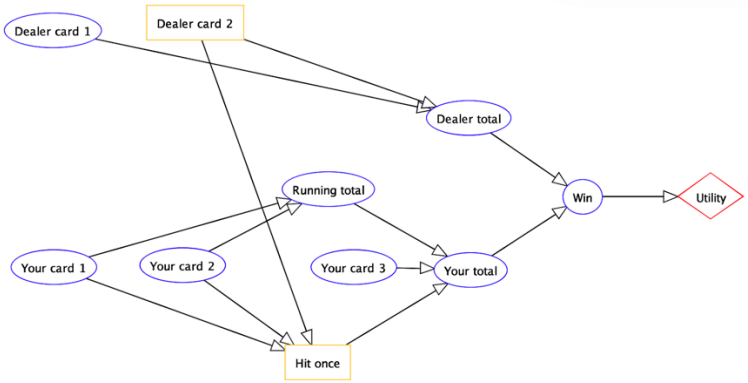
- To calculate this, we change the network by making Running Total a decision variable and then adding an arc from Running Total (decision) to Hit Once (decision), which is shown in the picture below.



c)

- The value of control for Dealer Card 2 is the expected utility of the network when we make Dealer Card 2 a decision variable (i.e., the network with Dealer Card 2 as a decision variable and with an arc from it to Hit Once) minus the expected utility when Dealer Card 2 is a random variable (i.e., the original network).

- The formula to calculate this is:
  EU(controlling Dealer Card 2) - EU(not controlling Dealer Card 2)
  = -0.0625 - (-0.25)
  = 0.1875



- Optimal policy: let the value of Dealer Card 2 be 4 and let Hit Once be false.

Question 2

a)
According to the textbook chapter 9.5.1, we have:

$$U^{(i+1)}(s) = \max_a \sum_{s'} P\,(s'|s,a)\left(R(s,a,s') + \gamma U^{(i)}(s')\right)$$

where i is non-negative integers.

Given $\gamma = 0.9$, i=0 and s=(10,8), we have:

$$U^{(1)}\big((10,8)\big) = \max_a \sum_{s'} P\,(s'|(10,8)\,,a)\left(R((10,8),a,s') + 0.9 \times U^{(0)}(s')\right)$$

When i=0, all of the states s have values $U^{(0)}(s) = 0$, so we have:

$$U^{(1)}\big((10,8)\big) = \max_a \sum_{s'} P\,(s'|(10,8)\,,a) \times R((10,8),a,s')$$

Since the agent gets the reward when it performs an action in that state, $R((10,8),a,s') = -1$ when it hits the wall, resulting in $s' = (10,8)$; otherwise, the reward is 0. Hence, we get:

$$U^{(1)}\big((10,8)\big) = \max_a \sum_{s'} P\left((10,8)\middle|(10,8),a\right) \times R\big((10,8),a,(10,8)\big)$$

$$= \max_a \sum_{s'} P\left((10,8)\middle|(10,8),a\right) \times (-1)$$

$$= \max_a \begin{vmatrix} P\big((10,8)\big|(10,8),\ LEFT\big) \times (-1) = 0.1 \times (-1) = -0.1 \\ P\big((10,8)\big|(10,8),\ RIGHT\big) \times (-1) = 0.7 \times (-1) = -0.7 \\ P\big((10,8)\big|(10,8),\ UP\big) \times (-1) = 0.1 \times (-1) = -0.1 \\ P\big((10,8)\big|(10,8),\ DOWN\big) \times (-1) = 0.1 \times (-1) = -0.1 \end{vmatrix}$$

$$= -0.1$$

Hence, $U^{(1)}\big((10,8)\big) = -0.1$ where actions can be LEFT, UP, or DOWN.

b)

1. $U^{(2)}((10,8)) = \max_a \sum_{s'} P\,(s'|(10,8),a)\left(R((10,8),a,s') + \gamma U^{(1)}(s')\right)$

$$= \max_a \begin{cases} \left(0.7 \times \left(0 + 0.9 \times U^{(1)}((10,7))\right) + 0.1 \times \left(0 + 0.9 \times U^{(1)}((9,8))\right)\right. \\ \qquad \left. + 0.1 \times \left(0 + 0.9 \times U^{(1)}((10,9))\right) + 0.1 \times \left(-1 + 0.9 \times U^{(1)}((10,8))\right)\right) & \text{UP} \\ \left(0.7 \times \left(0 + 0.9 \times U^{(1)}((10,9))\right) + 0.1 \times \left(0 + 0.9 \times U^{(1)}((9,8))\right)\right. \\ \qquad \left. + 0.1 \times \left(0 + 0.9 \times U^{(1)}((10,7))\right) + 0.1 \times \left(-1 + 0.9 \times U^{(1)}((10,8))\right)\right) & \text{DOWN} \\ \left(0.7 \times \left(0 + 0.9 \times U^{(1)}((9,8))\right) + 0.1 \times \left(0 + 0.9 \times U^{(1)}((10,9))\right)\right. \\ \qquad \left. + 0.1 \times \left(0 + 0.9 \times U^{(1)}((10,7))\right) + 0.1 \times \left(-1 + 0.9 \times U^{(1)}((10,8))\right)\right) & \text{LEFT} \\ \left(0.7 \times \left((-1) + 0.9 \times U^{(1)}((10,8))\right) + 0.1 \times \left(0 + 0.9 \times U^{(1)}((10,9))\right)\right. \\ \qquad \left. + 0.1 \times \left(0 + 0.9 \times U^{(1)}((10,7))\right) + 0.1 \times \left(0 + 0.9 \times U^{(1)}((9,8))\right)\right) & \text{RIGHT} \end{cases}$$

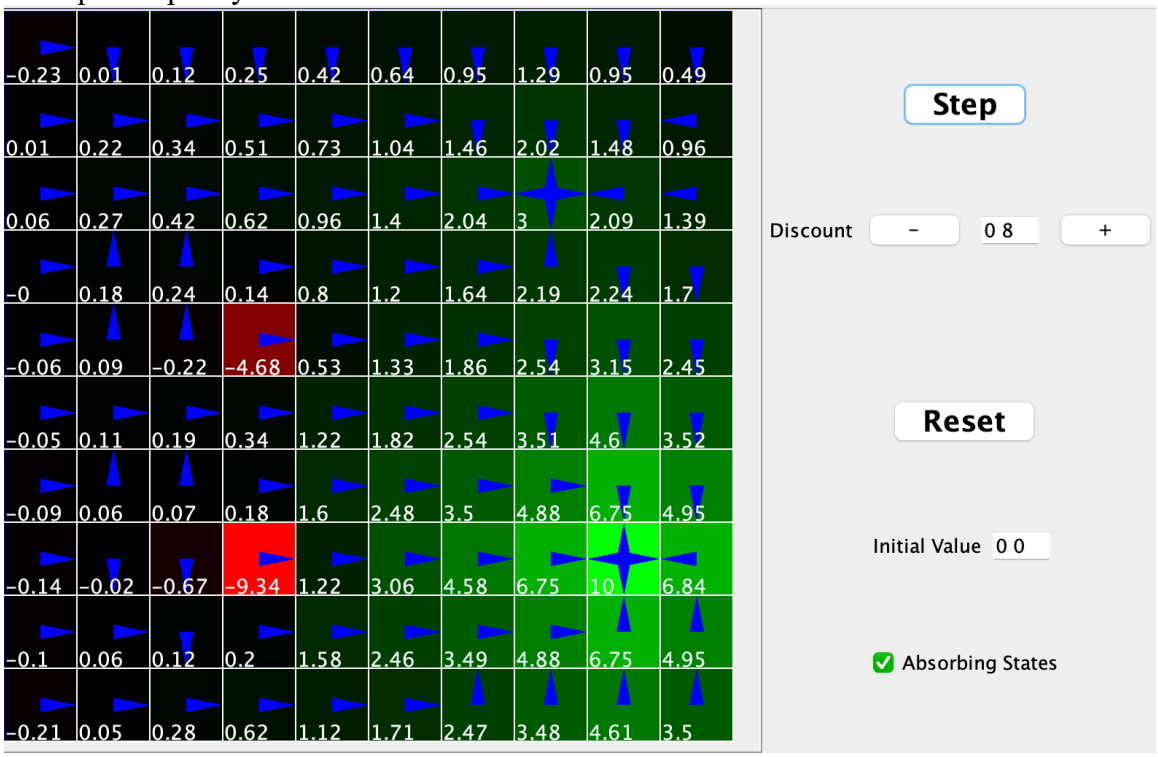
$$= \max_a \begin{cases} \left(0.7 \times (0 + 0.9 \times (-0.1)) + 0.1 \times (0 + 0.9 \times 10) + 0.1 \times (0 + 0.9 \times (-0.1))\right. \\ \qquad \left. + 0.1 \times (-1 + 0.9 \times (-0.1))\right) = 0.719 & \text{UP} \\ \left(0.7 \times (0 + 0.9 \times (-0.1)) + 0.1 \times (0 + 0.9 \times 10) + 0.1 \times (0 + 0.9 \times (-0.1))\right. \\ \qquad \left. + 0.1 \times (-1 + 0.9 \times (-0.1))\right) = 0.719 & \text{DOWN} \\ \left(0.7 \times (0 + 0.9 \times 10) + 0.1 \times (0 + 0.9 \times (-0.1)) + 0.1 \times (0 + 0.9 \times (-0.1))\right. \\ \qquad \left. + 0.1 \times (-1 + 0.9 \times (-0.1))\right) = 6.173 & \text{LEFT} \\ \left(0.7 \times ((-1) + 0.9 \times (-0.1)) + 0.1 \times (0 + 0.9 \times (-0.1))\right. \\ \qquad \left. + 0.1 \times (0 + 0.9 \times (-0.1)) + 0.1 \times (0 + 0.9 \times 10)\right) = 0.119 & \text{RIGHT} \end{cases}$$

Hence, $U^{(2)}((10,8)) = 6.173$ where the action is LEFT.

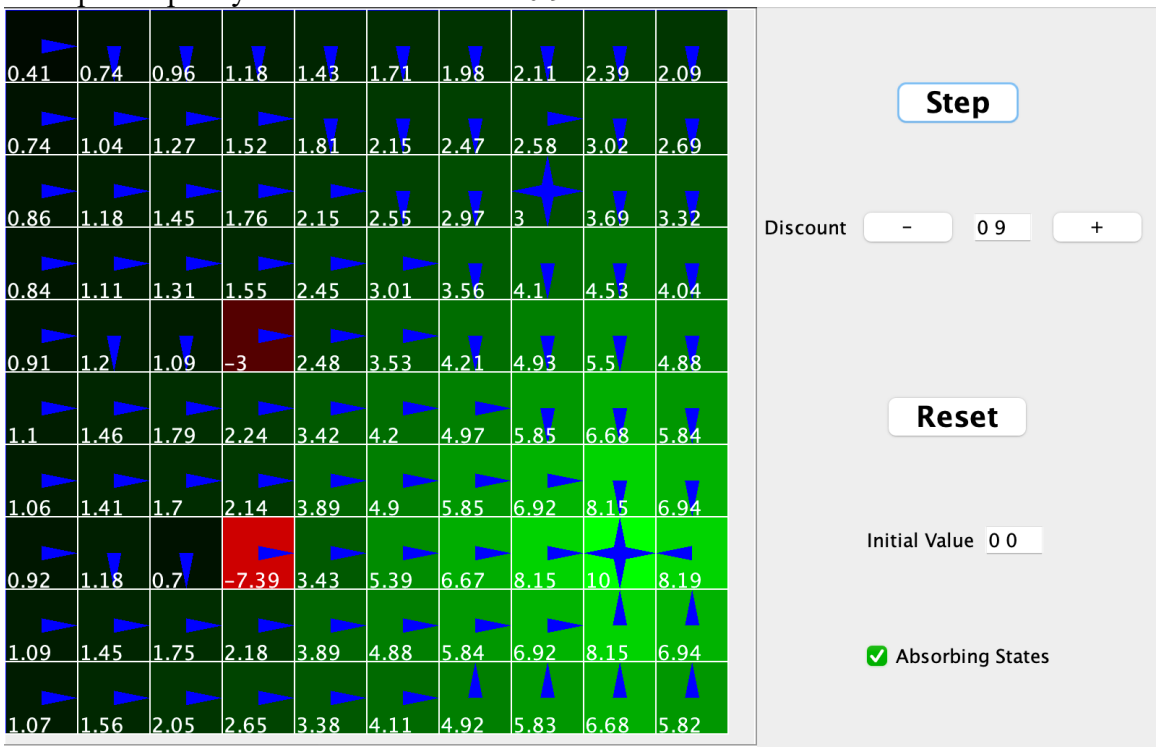
2. $U^{(2)}((9,9))$ is larger than $U^{(2)}((10,8))$ because (10,8) is adjacent to the wall, resulting in a reward of -1 if the action causes a bounding. However, at (9,9), the agent will always gain a reward = 0 since no matter what actions it takes, it will not hit the wall.

3. $U^{(2)}((9,7))$ is larger than $U^{(2)}((9,9))$ since the possible next states of (9,7) have the values of 0, 0, -0.1, 10, whereas the possible next states of (9,9) have the values of 0, -0.1, -0.1, 10. It is more likely for the agent to gain a negative value at (9,9) than at (9,7).

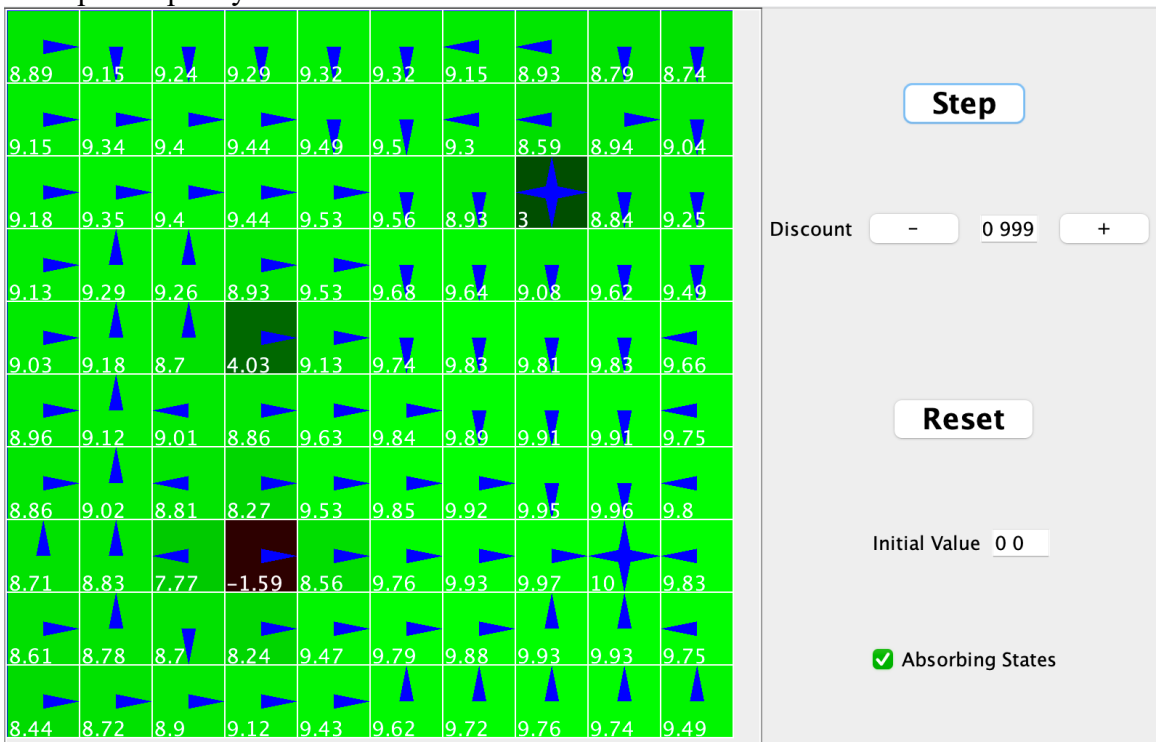
c)
1.
The optimal policy for discount factor = 0.8:



The optimal policy for discount factor = 0.9:



The optimal policy for discount factor = 0.999:



2.
When γ approaches to one, the values that the agent can gain in the future are discounted less. Therefore, compared with gaining the closer reward 3 at (8,3) within a relatively smaller number of actions, the agent changes the policies to perform more actions to reach the distant reward with value 10 at (9,8), which is more worthy.

3.
For the states (2,6) and (3,6), when γ = 0.8 or 0.9, the optimal policies are to move right since the values of each state gets discounted relatively more and the agent wants to reach (9,8) with reward 10 as soon as possible even if there are risks of entering the rewarding state (4,5) with a negative reward.

For the states (2,7) and (3,7), when γ = 0.8, the optimal policies are to move up to stay away from the rewarding state (4,8) with value = -9.34, a heavy price. When γ = 0.9, the optimal policies are to move right to reach (9,8) with reward 10 as soon as possible and value of (4,8) is -7.39, which is less risky.

When γ = 0.999, the values of each state are decreasing slower. In this case, the optimal policies at (2,6), (3,6), (2,7) and (3,7) change to stay away from the rewarding states with smaller rewards and would like to take more steps to obtain the distant reward 10 at (9,8).

Question 3

1.

- Belief state for (up, up, up), (2 walls, 2 walls, 2 walls):

  0.56837 0.22573 0.00313 0.00000
  0.03595 0.00000 0.00026 0.00000
  0.02016 0.14362 0.00201 0.00076

The probabilities that the agent is finally at (3,1), (3,2), or (1,2) are the greatest. When the agent moves up, the probability that it really moves up three times and observes 2 walls three times without errors is the largest. Therefore, it is most likely that the agent reaches the third row, except (3,3) which has only 1 wall, by moving up three times or stays at (1,2) by keeping hitting the wall above. The probability of being in terminal states is 0 since the last observation is not "end" without error.

- Belief state for (up, up, up) (1 wall,1 wall, 1 wall):

  0.00198 0.01354 0.96445 0.00000
  0.00012 0.00000 0.01834 0.00000
  0.00008 0.00065 0.00076 0.00009

Given the sequences of actions and observations, it is most likely for the agent to actually go up three times and reach states with 1 wall each time. Hence, the belief in terminal states is 0 (without error). Only states in the third column have 1 wall, among which (3, 3) has the greatest probability for the agent being in, because it can be reached through the states below it or itself by keeping hitting the wall.

- Belief state for (right, right, up) (1 wall, 1 wall, end) with S_0= (3,2):
  0.00000 0.00000 0.00000 0.55000
  0.00000 0.00000 0.00000 0.45000
  0.00000 0.00000 0.00000 0.00000

The last observation is "end", which has no probability of error, meaning that the agent has to reach one of terminal states. With 3 steps from (3,2), there is only one path to reach (2,4), while there are more ways to reach (3,4) since the agent could either hit the wall at (3,2) or (3,3).

- Belief state for (up, right, right, right) (2 walls, 2 walls, 1 wall, 1 wall) with S_0= (1,1):

  0.01626 0.01898 0.12473 0.00000
  0.11842 0.00000 0.17463 0.00000
  0.01628 0.02068 0.35309 0.15693

Based on the observations, the probabilities that the agent ends in states with 1 wall (i.e., in the third column) are greater, among which the belief in (1,3) is the greatest since it becomes positive earlier as it is located nearer to S_0. It is also likely for the agent to end up in (1,4) because there are three consecutive right actions, and the last observation is not "end". The agent performing all intended actions will reach (2,1) so the belief in this state is relatively high as well.

```
import sys


class State:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y


class Node:
    def __init__(self, state: State, e: list, b: float):
        self.state = state
        self.e = e
        self.b = b


def validX(x: int):
    return x is None and 0 < x <= 3


def validY(y: int):
    return y is None and 0 < y <= 4


def isNotBlock(s: State):
    return s.x != 2 or s.y != 2


def isNoneState(s: State):
    return s.x is None and s.y is None


def isSameState(s1: State, s2: State):
    return s1.x == s2.x and s1.y == s2.y


def isTerminal(s: State):
    if s.x == 2 and s.y == 4:
        return True
    elif s.x == 3 and s.y == 4:
        return True
    else:
        return False


def withoutNone(states: list):
    result = []
    for state in states:
        if not isNoneState(state):
            result.append(state)
    return result


def wallAtUp(s: State):
    if s.x == 3 or (s.x == 1 and s.y == 2):
        return 1
    else:
        return 0


def wallAtDown(s: State):
    if s.x == 1 or (s.x == 3 and s.y == 2):
        return 1
```

```
    else:
        return 0


def wallAtLeft(s: State):
    if s.y == 1 or (s.x == 2 and s.y == 3):
        return 1
    else:
        return 0


def wallAtRight(s: State):
    if s.y == 4 or (s.x == 2 and s.y == 1):
        return 1
    else:
        return 0


def possibleStateAtLeft(state_to_be_updated: State):
    curr_state_l = State(None, None)

    if state_to_be_updated.y == 1 or (state_to_be_updated.x == 2 and
state_to_be_updated.y == 3):
        curr_state_l = state_to_be_updated
    else:
        curr_state_l.x = state_to_be_updated.x
        curr_state_l.y = state_to_be_updated.y - 1

    if isTerminal(curr_state_l):
        curr_state_l = State(None, None)

    return curr_state_l


def possibleStateAtRight(state_to_be_updated: State):
    curr_state_r = State(None, None)

    if state_to_be_updated.y == 4 or (state_to_be_updated.x == 2 and
state_to_be_updated.y == 1):
        curr_state_r = state_to_be_updated
    else:
        curr_state_r.x = state_to_be_updated.x
        curr_state_r.y = state_to_be_updated.y + 1

    if isTerminal(curr_state_r):
        curr_state_r = State(None, None)

    return curr_state_r


def possibleStateAtDown(state_to_be_updated: State):
    curr_state_d = State(None, None)

    if state_to_be_updated.x == 1 or (state_to_be_updated.x == 3 and
state_to_be_updated.y == 2):
        curr_state_d = state_to_be_updated
    else:
        curr_state_d.x = state_to_be_updated.x - 1
        curr_state_d.y = state_to_be_updated.y

    if isTerminal(curr_state_d):
        curr_state_d = State(None, None)
```

```
    return curr_state_d


def possibleStateAtUp(state_to_be_updated: State):
    curr_state_u = State(None, None)

    if state_to_be_updated.x == 3 or (state_to_be_updated.x == 1 and
state_to_be_updated.y == 2):
        curr_state_u = state_to_be_updated
    else:
        curr_state_u.x = state_to_be_updated.x + 1
        curr_state_u.y = state_to_be_updated.y

    if isTerminal(curr_state_u):
        curr_state_u = State(None, None)

    return curr_state_u


def currForUp(state_to_be_updated: State):
    state_at_l = possibleStateAtLeft(state_to_be_updated)
    state_at_r = possibleStateAtRight(state_to_be_updated)
    state_at_u = possibleStateAtUp(state_to_be_updated)
    state_at_d = possibleStateAtDown(state_to_be_updated)

    if wallAtDown(state_to_be_updated):
        state_at_d = State(None, None)

    if not wallAtUp(state_to_be_updated):
        state_at_u = State(None, None)
    return [state_at_u, state_at_d, state_at_l, state_at_r]


def currForDown(state_to_be_updated: State):
    state_at_l = possibleStateAtLeft(state_to_be_updated)
    state_at_r = possibleStateAtRight(state_to_be_updated)
    state_at_u = possibleStateAtUp(state_to_be_updated)
    state_at_d = possibleStateAtDown(state_to_be_updated)

    if wallAtUp(state_to_be_updated):
        state_at_u = State(None, None)

    if not wallAtDown(state_to_be_updated):
        state_at_d = State(None, None)
    return [state_at_u, state_at_d, state_at_l, state_at_r]


def currForLeft(state_to_be_updated: State):
    state_at_l = possibleStateAtLeft(state_to_be_updated)
    state_at_r = possibleStateAtRight(state_to_be_updated)
    state_at_u = possibleStateAtUp(state_to_be_updated)
    state_at_d = possibleStateAtDown(state_to_be_updated)

    if wallAtRight(state_to_be_updated):
        state_at_r = State(None, None)

    if not wallAtLeft(state_to_be_updated):
        state_at_l = State(None, None)
    return [state_at_u, state_at_d, state_at_l, state_at_r]


def currForRight(state_to_be_updated: State):
    state_at_l = possibleStateAtLeft(state_to_be_updated)
```

```
    state_at_r = possibleStateAtRight(state_to_be_updated)
    state_at_u = possibleStateAtUp(state_to_be_updated)
    state_at_d = possibleStateAtDown(state_to_be_updated)

    if wallAtLeft(state_to_be_updated):
        state_at_l = State(None, None)

    if not wallAtRight(state_to_be_updated):
        state_at_r = State(None, None)
    return [state_at_u, state_at_d, state_at_l, state_at_r]


def possibleCurrStates(stateToBeUpdated: State, action: str):
    if isNotBlock(stateToBeUpdated):

        tmp = []

        if action == "up":
            tmp = currForUp(stateToBeUpdated)

        if action == "down":
            tmp = currForDown(stateToBeUpdated)

        if action == "left":
            tmp = currForLeft(stateToBeUpdated)

        if action == "right":
            tmp = currForRight(stateToBeUpdated)

        res = withoutNone(tmp)
        result = list(set(res))

        return result

    else:
        print("err")
        return []


def initializeObservations(bs):
    for row in bs:
        for node in row:
            if isTerminal(node.state):
                node.e = [0, 0, 1]
            elif node.state.y == 3:
                node.e = [0.9, 0.1, 0]
            else:
                node.e = [0.1, 0.9, 0]
    return bs


def initializeBelief(bs, init_state):
    if init_state is None:
        for row in bs:
            for node in row:
                if isTerminal(node.state):
                    node.b = 0.0
                else:
                    node.b = 1 / 9
    else:
        for row in bs:
            for node in row:
                if node.state.x == init_state.x and node.state.y == init_state.y:
```

```
                node.b = 1.0
            else:
                node.b = 0.0
    return bs


def initializeBeliefStates(init_state: State):
    matrix: Node = [[0 for y in range(4)] for x in range(3)]
    for x in range(3):
        for y in range(4):
            matrix[x][y] = Node(State(3 - x, y + 1), [0, 0, 0], 0)

    initializeObservations(matrix)
    initializeBelief(matrix, init_state)

    matrix[1][1] = Node(State(2, 2), [0, 0, 0], 0.0)

    return matrix


def atLeft(a: State, b: State):
    return a.y == b.y - 1 and a.x == b.x


def atUp(a: State, b: State):
    return a.x == b.x + 1 and a.y == b.y


def transModelUp(s: State, ps: State):
    if s.x == ps.x and s.y == ps.y:
        result = 0.8 * (wallAtUp(s)) + 0.1 * (wallAtLeft(s)) + 0.1 *
(wallAtRight(s))
    elif atUp(ps, s):
        result = 0
    elif atUp(s, ps):
        result = 0.8
    else:
        result = 0.1
    return result


def transModelDown(s: State, ps: State):
    if s.x == ps.x and s.y == ps.y:
        result = 0.8 * (wallAtDown(s)) + 0.1 * (wallAtLeft(s)) + 0.1 *
(wallAtRight(s))
    elif atUp(s, ps):
        result = 0
    elif atUp(ps, s):
        result = 0.8
    else:
        result = 0.1
    return result


def transModelLeft(s: State, ps: State):
    if s.x == ps.x and s.y == ps.y:
        result = 0.8 * (wallAtLeft(s)) + 0.1 * (wallAtUp(s)) + 0.1 *
(wallAtDown(s))
    elif atLeft(ps, s):
        result = 0
    elif atLeft(s, ps):
        result = 0.8
    else:
```

```
        result = 0.1
    return result


def transModelRight(s: State, ps: State):
    if s.x == ps.x and s.y == ps.y:
        result = 0.8 * (wallAtRight(s)) + 0.1 * (wallAtUp(s)) + 0.1 *
(wallAtDown(s))
    elif atLeft(s, ps):
        result = 0
    elif atLeft(ps, s):
        result = 0.8
    else:
        result = 0.1
    return result


def transModel(action: str, s: State, ps: State):
    result = 0
    if (s.x - ps.x) * (s.x - ps.x) >= 1 and (not (s.y == ps.y)):
        print("wrong ps")
    elif (s.y - ps.y) * (s.y - ps.y) >= 1 and (not (s.x == ps.x)):
        print("wrong ps")
    elif action == "up":
        result = transModelUp(s, ps)
    elif action == "down":
        result = transModelDown(s, ps)
    elif action == "left":
        result = transModelLeft(s, ps)
    elif action == "right":
        result = transModelRight(s, ps)
    return result


def updateBelief(bs, node, observation: str, action):
    total = 0
    possible_states = possibleCurrStates(node.state, action)
    for ps in possible_states:
        total += transModel(action, node.state, ps) * bs[-1 * (ps.x - 3)][ps.y -
1].b

    index = 0
    if observation == "1 wall":
        index = 0
    elif observation == "2 walls":
        index = 1
    elif observation == "end":
        index = 2
    else:
        print("invalid observation")

    newBelief = node.e[index] * total

    return newBelief


def normalize(bs):
    total = 0.0
    for row in bs:
        for node in row:
            if not (node.state.x == 2 and node.state.y == 2):
                total += node.b
```

```
    if total != 0:
        for row in bs:
            for node in row:
                if not (node.state.x == 2 and node.state.y == 2):
                    node.b = node.b / total

    return bs


def print_belief_states(matrix):
    for x in range(3):
        for y in range(4):
            print('{0:1.5f}'.format(matrix[x][y].b), end=' ')
        print()


def solvePOMDPs(init_state: State, sequence_of_actions: list,
sequence_of_observations: list):
    bs = initializeBeliefStates(init_state)
    # print("initial states")
    # printMatrix(bs)

    for i in range(len(sequence_of_actions)):
        tmp_bs: Node = [[0 for y in range(4)] for x in range(3)]

        for x in range(3):
            for y in range(4):
                tmp_bs[x][y] = 0

        for row in bs:
            for node in row:
                x = node.state.x
                y = node.state.y
                if not (x == 2 and y == 2):
                    tmp_bs[-1 * (x - 3)][y - 1] = updateBelief(bs, node,
sequence_of_observations[i],
                                                               sequence_of_actions[i])

        for row in bs:
            for node in row:
                x = node.state.x
                y = node.state.y
                if not (x == 2 and y == 2):
                    bs[-1 * (x - 3)][y - 1].b = tmp_bs[-1 * (x - 3)][y - 1]

    bs = normalize(bs)

    # print("after updated")
    print_belief_states(bs)


def solve(args):
    split_line = "-------------------------------------------"

    print("Belief state for sequence 1:\n")
    # S_0 is not specified
    solvePOMDPs(None, ["up", "up", "up"], ["2 walls", "2 walls", "2 walls"])

    print(split_line)
    print("Belief state for sequence 2:\n")
    # S_0 is not specified
    solvePOMDPs(None, ["up", "up", "up"], ["1 wall", "1 wall", "1 wall"])
```

```
    print(split_line)
    print("Belief state for sequence 3:\n")
    solvePOMDPs(State(3, 2), ["right", "right", "up"], ["1 wall", "1 wall",
"end"])

    print(split_line)
    print("Belief state for sequence 4:\n")
    solvePOMDPs(State(1, 1), ["up", "right", "right", "right"], ["2 walls", "2
walls", "1 wall", "1 wall"])


if __name__ == '__main__':
    solve(sys.argv)
```