COMP3702 Tutorial 7

Matt Choy | matthew.choy@uq.edu.au

COMP3702 Tutorial 7

Matt Choy | matthew.choy@uq.edu.au

https://github.com/mattpchoy/comp3702-tutorials

Markov Decision Processes

Markov Decision Process is a framework we can use to determine the best action that an agent should perform in a environment that is:

- Stochastic (non-deterministic)
- Discrete-time
- Discrete-transition (well defined transitions between states)

MDPs are the basis for Reinforcement Learning, and can be used to solve games like Tic Tac Toe, Chess, Go and other games in which we consider the other player.

MDP Components

A set of states (S)

A set of actions (A)

Transition Function $P(S_{t+1} \mid S_t, A_t)$

Reward Function $R(S_t, A_t, S_{t+1})$

Discount Factor γ

Objective of MDP

The objective of an MDP is to maximise the expected value of the rewards

$$\mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \, R(s_t, a_t)\right]$$

MDPs – World State

World State:

If the agent knew the world state (information contained within the world state),

no information about the past is relevant to the future – "Markovian Property"

MDPs – World State

World State:

If the agent knew the world state (information contained within the world state),

no information about the past is relevant to the future – "Markovian Property"

Given that S_k denotes a state at time k, and action A_k is the action performed at time k,

$$P(S_{t+1} | S_0, A_0, \cdots, S_t, A_t) = P(S_{t+1} | S_t, A_t)$$

MDPs – World State

World State:

If the agent knew the world state (information contained within the world state),

no information about the past is relevant to the future – "Markovian Property"

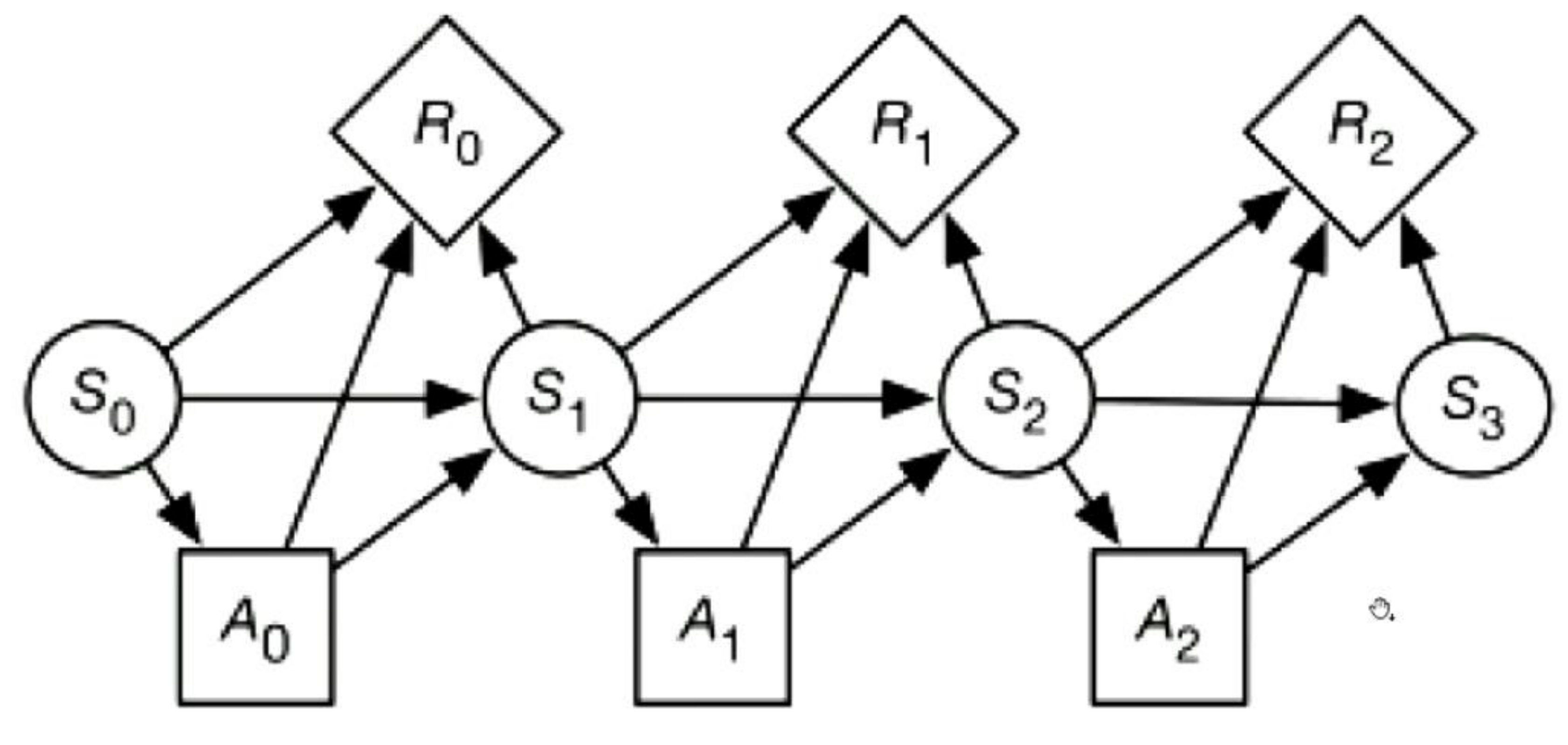
Given that S_k denotes a state at time k, and action A_k is the action performed at time k,

$$P(S_{t+1} | S_0, A_0, \cdots, S_t, A_t) = P(S_{t+1} | S_t, A_t)$$

$$P(s' | s, a)$$

MDPs vs Markov Chains

A MDP augments a Markov Chain with actions and values.



Planning Horizon & Information Availability

Planning Horizon: How far ahead the planner / agent looks forward to make a decision

Information Availability: What information is available when the agent decides what to do:

- **Fully Observable MDP (FOMDP)** The agent gets to observe S_t when deciding on action A_t
- **Partially Observable MDP (POMDP)** The agent mas some noisy / imperfect sensor of the state

Policies

A policy is a sequence of actions, taken to move from each state to the next state over the whole time horizon.

A **Stationary Policy** is a function or map. Given a state s, the policy $\pi(s)$ specifies the action takes.

Policies

A policy is a sequence of actions, taken to move from each state to the next state over the whole time horizon.

A **Stationary Policy** is a function or map. Given a state s, the policy $\pi(s)$ specifies the action takes.

An **Optimal Policy**, typically denoted as π^* is one with the maximum expected discount reward.

$$\max_{\pi} \mathbb{E}\left[\sum_{t=0} \gamma^t R\big(s_t, \pi(s_t)\big)\right]$$

Given that $\pi(t)$ is the action taken at time t *

Value of a Policy

- We first approach MDPs using a recursive reformulation of the objective called a `value function`
  - The value function of an MDP, $V^\pi(s)$ is the expected future reward of following an (arbitrary) policy π starting from state s, given by:

    $$V^\pi(s) = \sum_{s' \in S} P(s'|\pi(s), s)[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

    Where the policy $\pi(s)$ determines the action taken in state s

    $P(s'|\pi(s), s)$ is our transition function for our stochastic world.

    $R(s, \pi(s), s')$ is our reward function

    $\gamma V^\pi(s')$ is our discounted future value function
  - Here, we have dropped the time index as it is redundant, but note that $a_t = \pi(s_t)$
  - Note that this is a recursive definition - the value of $V^\pi(s)$ depends on the value of $V^\pi(s')$

Value of a Policy

- We first approach MDPs using a recursive reformulation of the objective called a `value function`
  - The value function of an MDP, $V^\pi(s)$ is the expected future reward of following an (arbitrary) policy π starting from state s, given by:

    $$V^\pi(s) = \sum_{s' \in S} P(s'|\pi(s), s)[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

    Where the policy $\pi(s)$ determines the action taken in state s

    $P(s'|\pi(s), s)$ is our transition function for our stochastic world.

    $R(s, \pi(s), s')$ is our reward function

    $\gamma V^\pi(s')$ is our discounted future value function
  - Here, we have dropped the time index as it is redundant, but note that $a_t = \pi(s_t)$
  - Note that this is a recursive definition - the value of $V^\pi(s)$ depends on the value of $V^\pi(s')$

Value of a Policy

- Given a policy π
  - The Q-function represents the value of choosing an action and then following policy π in every subserquent state.
  - $Q^\pi(s, a)$, where a is an action and s is a state, is the expected value of doing a in state s, then following policy π
- Q^π and V^π can be defined mutually recursively:

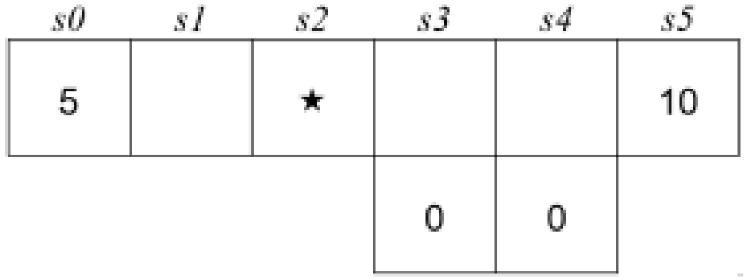
$$Q^\pi(s, a) = \sum_{s'} P(s'|a, s)(R(s, a, s') + \gamma V^\pi(s'))$$
$$V^\pi(s) = Q(s, \pi(s))$$

- `Note`: When computing the `future value`, choose the best action using the policy π instead of for some arbitrary action a

Exercise 7.1

$$V^{\pi}(s) = \sum_{s' \in S} P(s'|\pi(s), s)[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Exercise 7.1. Consider the gridworld below:

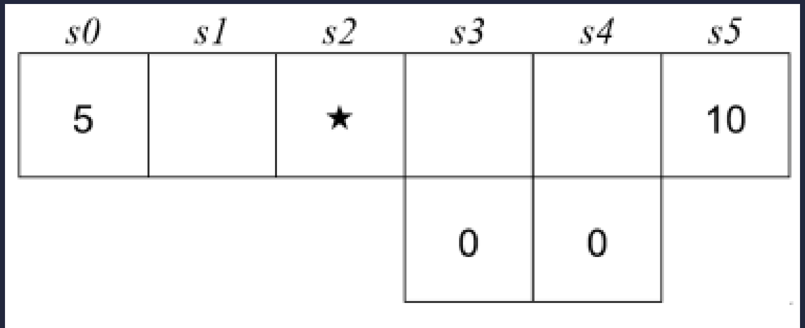


An agent is currently on grid cell s_2, as indicated by the star, and would like to collect the rewards that lie on both sides of it. If the agent is on a numbered square (0, 5 or 10), the instance terminates and the agent receives a reward equal to the number on the square. On any other (non-numbered) square, its available actions are to move Left and Right. Note that Up and Down are never available actions. If the agent is in a square with an adjacent square below it, it does not always move successfully: when the agent is in one of these squares and takes a move action, it will only succeed with probability p. With probability $1 - p$, the move action will fail and the agent will instead fall downwards into a trap. If the agent is not in a square with an adjacent space below, it will always move successfully.

a) Consider the policy π_R, which is to always move right when possible. For each state $s \in \{s_1, s_2, s_3, s_4\}$ in the diagram above, give the value function V^{π_R} in terms of $\gamma \in [0, 1]$ and $p \in [0, 1]$.

b) Consider the policy π_L, which is to always move left when possible. For each state $s \in \{s_1, s_2, s_3, s_4\}$ in the diagram above, give the value function V^{π_L} in terms of γ and p.

Exercise 7.2a



We first recall that we're computing the value of the value function at each state, for policy π_R. Under this policy, the optimal action in each state is to move right.

We recall the definition of a value function of a MDP.

$$V^{\pi}(s) = \sum_{s' \in S} P(s'|\pi(s), s)[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

And thus, our values are:

$V^{\pi_R}(s_4) = p[0 + \gamma 10] = 10p\gamma$ Probability of not falling is p. Next state s' is s_5 and it has value of 10

$V^{\pi_R}(s_3) = p[0 + 10p\gamma^2] = 10p^2\gamma^2$ Probability of not falling is p. Next state s' is s_4 and it has value of $10p\gamma$

$V^{\pi_R}(s_2) = 1[0 + 10p^2\gamma^3] = 10p^2\gamma^3$ Probability of not falling is 1. Next state s' is s_3 and it has value of $10p^2\gamma^2$

$V^{\pi_R}(s_1) = 1[0 + 10p^2\gamma^4] = 10p^2\gamma^4$ Probability of not falling is 1. Next state s' is s_2 and it has value of $10p^2\gamma^3$

Exercise 7.2a

s0	s1	s2	s3	s4	s5
5		★			10
			0	0	

And thus, our values are:

$V^{\pi_R}(s_4) = p[0 + \gamma 10] = 10p\gamma$ Probability of not falling is p. Next state s’ is s_5 and it has value of 10

$V^{\pi_R}(s_3) = p[0 + 10p\gamma^2] = 10p^2\gamma^2$ Probability of not falling is p. Next state s’ is s_4 and it has value of $10p\gamma$

$V^{\pi_R}(s_2) = 1[0 + 10p^2\gamma^3] = 10p^2\gamma^3$ Probability of not falling is 1. Next state s’ is s_3 and it has value of $10p^2\gamma^2$

$V^{\pi_R}(s_1) = 1[0 + 10p^2\gamma^4] = 10p^2\gamma^4$ Probability of not falling is 1. Next state s’ is s_2 and it has value of $10p^2\gamma^3$

s0	s1	s2	s3	s4	s5
5	$10\gamma^4p^2$	$10\gamma^3p^2$	$10\gamma^2p^2$	$10\gamma p$	10
			0	0	

Exercise 7.2

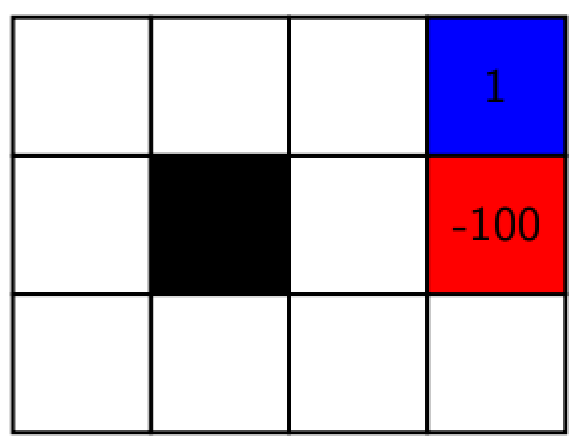
Exercise 7.2.

a) Implement the `attempt_move` and `get_transition_probabilities` methods in the `Grid` class in `grid_world_starter.py` according to their docstrings. The aim is to have functions that can be used on an arbitrary gridworld (i.e. do not hard-code your function just for this problem instance!).

b) What is the one-step probability of arriving in each state s' when starting from [0,0] for each a, i.e what is $P(s'|a,[0,0])\ \forall\ a, s'$?

c) What is the one-step probability of arriving in state [1,0] from each state s after taking action a, i.e what is $P([1,0]|a,s)\ \forall\ (a,s)$?

1. Download grid_world_starter.py off Blackboard or GitHub and implement two functions:
    1. attempt_move(self, state, action)
    2. get_transition_probabilities(self, state, action)



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXIT_STATE”

$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

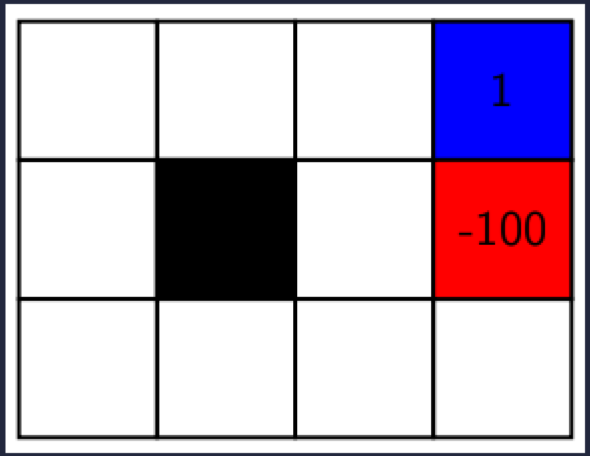
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

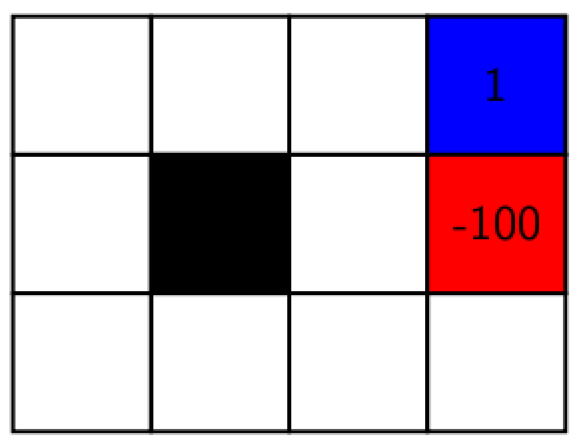
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

self.states = list((x, y) for x in range(self.x_size) for y in range(self.y_size))
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”

$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

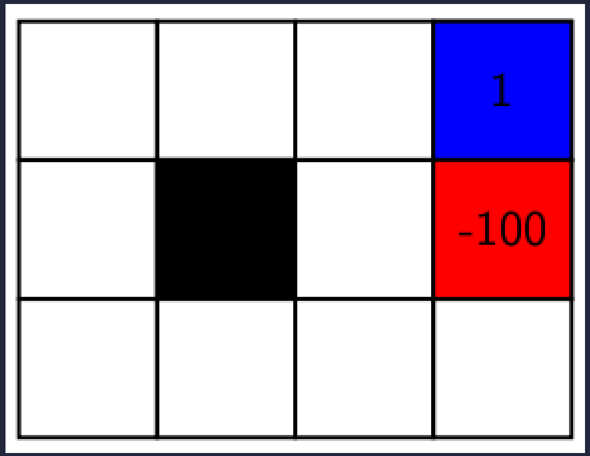
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

def attempt_move(self, s, a):
    x, y = s
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

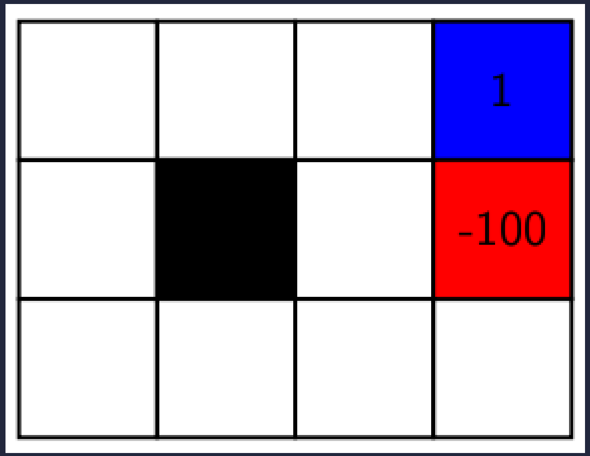
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT']

def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called "EXITED"
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

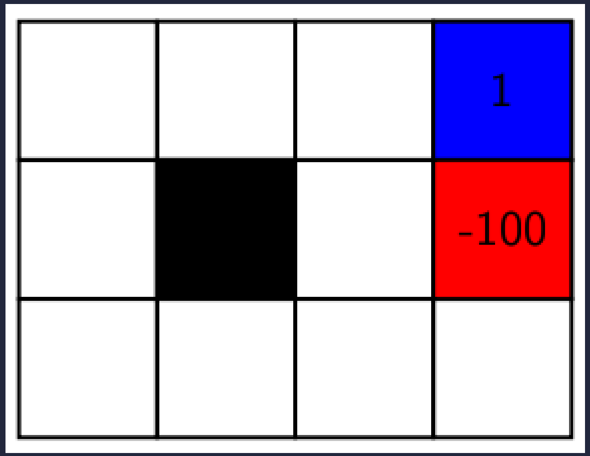
R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
    if s in self.rewards.keys():
        return EXIT_STATE
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

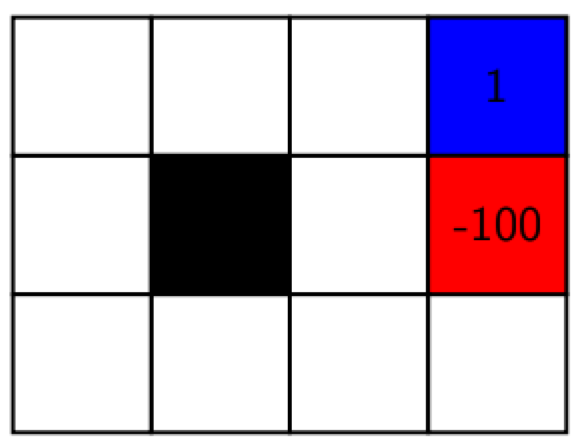
R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
    if s in self.rewards.keys():
        return EXIT_STATE

    # Default: no movement
    result = s
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

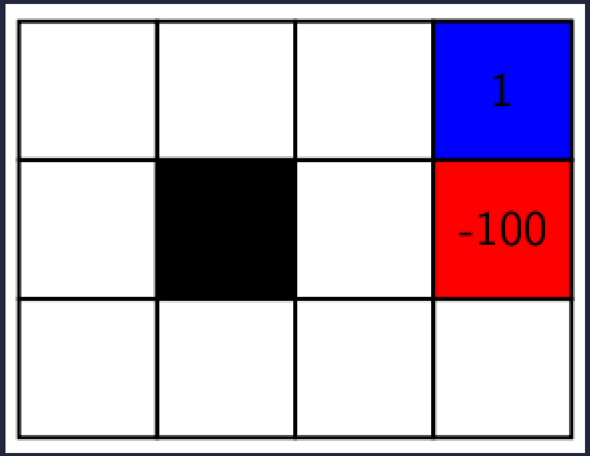
R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
    if s in self.rewards.keys():
        return EXIT_STATE

    # Default: no movement
    result = s

    if a == LEFT and x > 0:
        result = (x - 1, y)
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

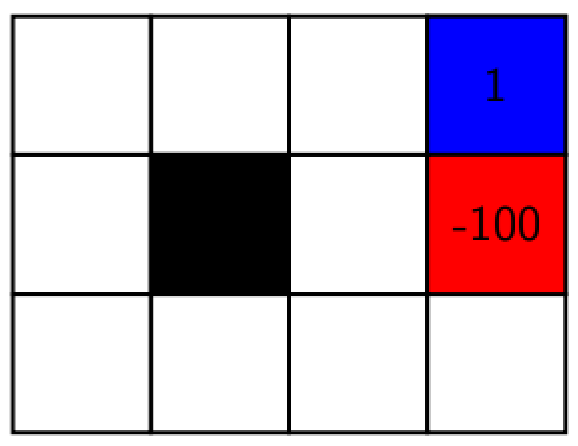
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```python
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
    if s in self.rewards.keys():
        return EXIT_STATE

    # Default: no movement
    result = s

    if a == LEFT and x > 0:
        result = (x - 1, y)
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”

$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

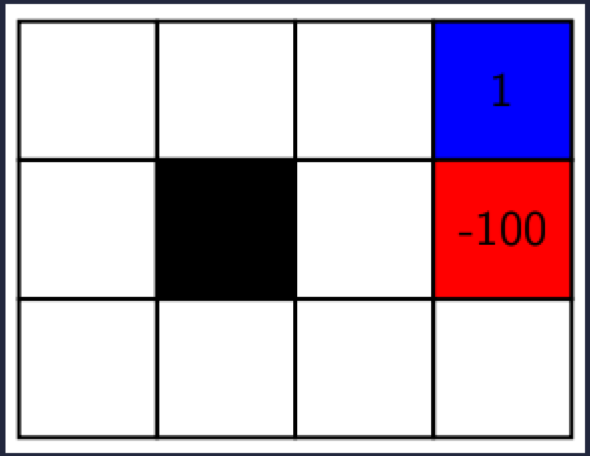
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
ACTION_NAMES = ['UP', 'DOWN', 'LEFT', 'RIGHT’]

def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
    if s in self.rewards.keys():
        return EXIT_STATE

    # Default: no movement
    result = s

    if a == LEFT and x > 0:
        result = (x - 1, y)
    if a == RIGHT and x < self.x_size - 1:
        result = (x + 1, y)
    if a == UP and y < self.y_size - 1:
        result = (x, y + 1)
    if a == DOWN and y > 0:
        result = (x, y - 1)
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”

$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

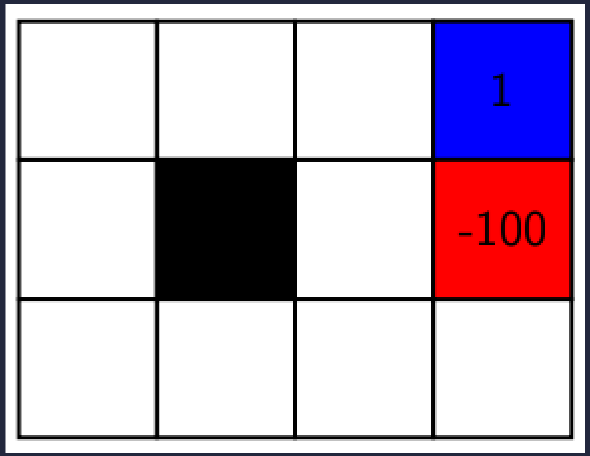
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – attempt_move

```
def attempt_move(self, s, a):
    x, y = s

    # Check absorbing state
    if s == EXIT_STATE:
        return s
    if s in self.rewards.keys():
        return EXIT_STATE

    # Default: no movement
    result = s

    if a == LEFT and x > 0:
        result = (x - 1, y)
    if a == RIGHT and x < self.x_size - 1:
        result = (x + 1, y)
    if a == UP and y < self.y_size - 1:
        result = (x, y + 1)
    if a == DOWN and y > 0:
        result = (x, y - 1)

    # Check obstacle cells
    if result in OBSTACLES:
        return s

    return result
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXITED”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

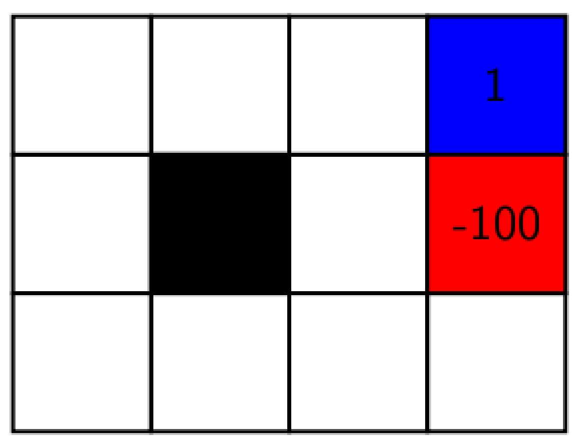
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – transition prob

```python
def stoch_action(self, a):
    """ Returns the probabilities with which each action will actually occur,
        given that action a was requested.

    Parameters:
        a: The action requested by the agent.

    Returns:
        The probability distribution over actual actions that may occur.
    """
    if a == RIGHT:
        return {RIGHT: self.p , UP: (1-self.p)/2, DOWN: (1-self.p)/2}
    elif a == UP:
        return {UP: self.p , LEFT: (1-self.p)/2, RIGHT: (1-self.p)/2}
    elif a == LEFT:
        return {LEFT: self.p , UP: (1-self.p)/2, DOWN: (1-self.p)/2}
    return {DOWN: self.p , LEFT: (1-self.p)/2, RIGHT: (1-self.p)/2}
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called "EXITED"

$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

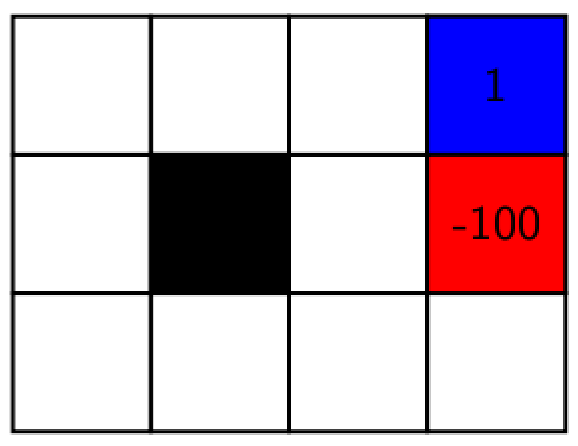
R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2 – transition prob

```python
def get_transition_probabilities(self, s, a):
    probabilities = {}
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called "EXITED"
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

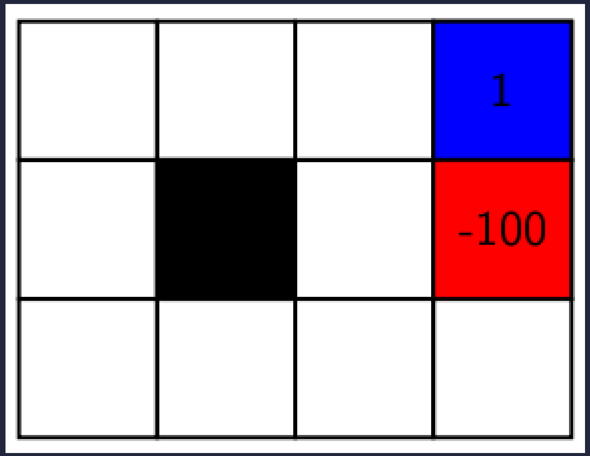
R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.2 – transition prob

```python
def get_transition_probabilities(self, s, a):
    probabilities = {}
    for action, probability in self.stoch_action(a).items():
        next_state = self.attempt_move(s, action)
        probabilities[next_state] = probabilities.get(next_state, 0) + probability
```



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called "EXITED"
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

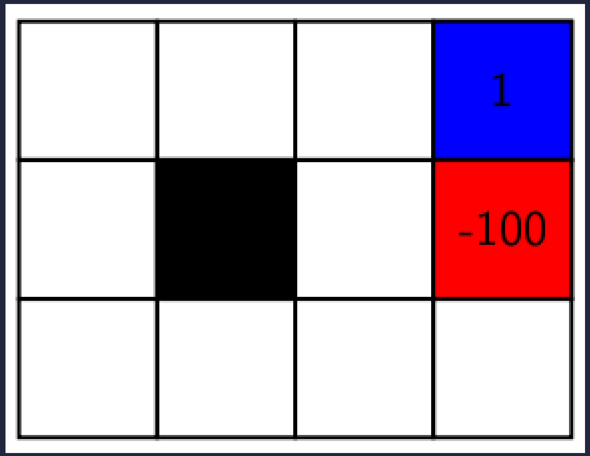
Rewards

R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.2 – transition prob




```
def get_transition_probabilities(self, s, a):
    probabilities = {}
    for action, probability in self.stoch_action(a).items():
        next_state = self.attempt_move(s, action)
        probabilities[next_state] = probabilities.get(next_state, 0) + probability
    return probabilities
```

States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called "EXITED"

$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor

$\gamma = 0.9$

Exercise 7.2b

b) What is the one-step probability of arriving in each state s' when starting from [0,0] for each a, i.e what is $P(s'|a,[0,0])\ \forall\ a, s'$?

We can determine the one-step probability of arriving in each state by calling our get_transition_probabilities function for each action.

UP = 0 DOWN = 1 LEFT = 2 RIGHT = 3

Action LEFT

{2: 0.8, 0: 0.09999999999999998, 1: 0.09999999999999998}

{(0, 0): 0.9, (0, 1): 0.09999999999999998}

Action RIGHT

{3: 0.8, 0: 0.09999999999999998, 1: 0.09999999999999998}

{(1, 0): 0.8, (0, 1): 0.09999999999999998, (0, 0): 0.09999999999999998}

Action UP

{0: 0.8, 2: 0.09999999999999998, 3: 0.09999999999999998}

{(0, 1): 0.8, (0, 0): 0.09999999999999998, (1, 0): 0.09999999999999998}

Action DOWN

{1: 0.8, 2: 0.09999999999999998, 3: 0.09999999999999998}

{(0, 0): 0.9, (1, 0): 0.09999999999999998}

Exercise 7.2b

b) What is the one-step probability of arriving in each state s' when starting from [0,0] for each a, i.e what is $P(s'|a,[0,0])\ \forall\ a, s'$?

We can determine the one-step probability of arriving in each state by calling our get_transition_probabilities function for each action.

P((1, 0)|right,(0, 0)) = 0.8
P((0, 1)|right,(0, 0)) = P((0, 0)|right,(0, 0)) = 0.1

P((0, 1)|up,(0, 0)) = 0.8
P((1, 0)|up,(0, 0)) = P((0, 0)|up,(0, 0)) = 0.1

P((0, 0)|left,(0, 0)) = 0.9
P((0, 1)|left,(0, 0)) = 0.1

P((0, 0)|down,(0, 0)) = 0.9
P((1, 0)|down,(0, 0)) = 0.1

Exercise 7.2c

c) What is the one-step probability of arriving in state [1,0] from each state s after taking action a, i.e what is $P([1,0]|a,s)\ \forall\ (a,s)$?

We can once again call the get_transition_probabilities function in a loop over all state and action pairs.

P((1, 0)|right,(0, 0)) = 0.8
P((1, 0)|up,(0, 0)) = 0.1
P((1, 0)|down,(0, 0)) = 0.1

P((1, 0)|left, [2, 0]) = 0.8
P((1, 0)|up, [2, 0]) = 0.1
P((1, 0)|down, [2, 0]) = 0.1

P((1, 0)|up,(1, 0)) = 0.8
P((1, 0)|down,(1, 0)) = 0.8
P((1, 0)|left,(1, 0)) = 0.2
P((1, 0)|right,(1, 0)) = 0.2

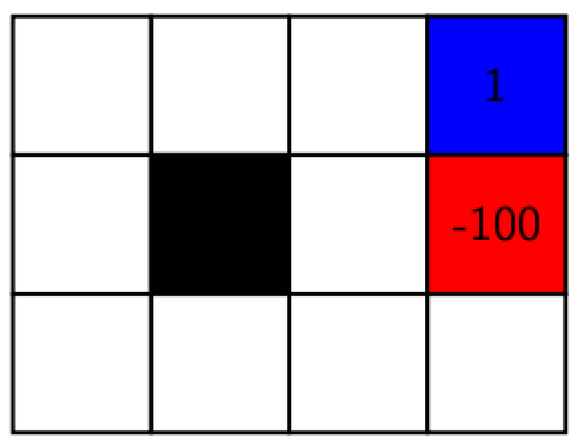
Exercise 7.3

Exercise 7.3. Implement VI for this problem, using: $V[s] \leftarrow \max_a \sum_{s'} P(s' \mid s, a)\,(R(s,a,s') + \gamma V[s'])$.

Note that $R(s,a,s') = R(s)$ is the reward for landing on a square, which is non-zero for only the red and blue squares at [3,1] and [3,2], respectively.

a) What is the value function estimate after 4 iterations? What is the policy according to the value function estimate after 4 iterations?

b) What is the value function estimate after 10 iterations? What is the policy according to the value function estimate after 10 iterations?



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXIT_STATE”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.3

The equation $V[s] \leftarrow \max\limits_{a} \sum_{s'} P(s'|a, s)[R(s, a, s') + \gamma V[s']\,]$ is a version of the Bellman update equation for Value Iteration.

To implement Value Iteration, we want to compute $\sum_{s'} P(s'|a, s)[R(s, a, s') + \gamma V[s']]$ for each possible next state, s'.

Exercise 7.3

The equation $V[s] \leftarrow \max_a \sum_{s'} P(s'|a,s)[R(s,a,s') + \gamma V[s']\,]$ is a version of the Bellman update equation for Value Iteration.

To implement Value Iteration, we want to compute $\sum_{s'} P(s'|a,s)[R(s,a,s') + \gamma V[s']]$ for each possible next state, s'.

We first define some functions and constants that help with the Value Iteration implementation.

```
EPSILON = 0.0001

def dict_argmax(d):
    max_value = max(d.values())
    for k, v in d.items():
        if v == max_value:
            return k
```

Exercise 7.3

We first create a class containing all of the information we want to keep track of in our Value Iteration implementation

```python
class ValueIteration:
    def __init__(self, grid):
        self.grid = grid
        self.values = {state: 0 for state in self.grid.states}
        self.policy = {state: RIGHT for state in self.grid.states}
        self.converged = False
        self.differences = []
```

Exercise 7.3

```
class ValueIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()
```

Exercise 7.3

```
class ValueIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()
        for s in self.grid.states:
            # Keep track of maximum value
            action_values = dict()
```

Exercise 7.3

```
class ValueIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()
        for s in self.grid.states:
            # Keep track of maximum value
            action_values = dict()
```

Exercise 7.3

$$V^{\pi}(s) = \sum_{s' \in S} P(s'|\pi(s), s)[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

```
class ValueIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()
        for s in self.grid.states:
            # Keep track of maximum value
            action_values = dict()
            for a in self.grid.actions:
                total = 0
                for stoch_action, p in self.grid.stoch_action(a).items():
                    # Apply action
                    s_next = self.grid.attempt_move(s, stoch_action)
                    total += p * (self.grid.get_reward(s)
                                  + (self.grid.discount * self.values[s_next]))
                action_values[a] = total
```

Exercise 7.3

```python
class ValueIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()
        for s in self.grid.states:
            # Keep track of maximum value
            action_values = dict()
            for a in self.grid.actions:
                total = 0
                for stoch_action, p in self.grid.stoch_action(a).items():
                    # Apply action
                    s_next = self.grid.attempt_move(s, stoch_action)
                    total += p * (self.grid.get_reward(s)
                                  + (self.grid.discount * self.values[s_next]))
                action_values[a] = total
            # Update state value with best action
            new_values[s] = max(action_values.values())
            new_policy[s] = dict_argmax(action_values)
```

Exercise 7.3

```python
class ValueIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()
        for s in self.grid.states:
            ...
            # Update state value with best action
            new_values[s] = max(action_values.values())
            new_policy[s] = dict_argmax(action_values)
        # Check convergence
        differences = [abs(self.values[s] - new_values[s]) for s in self.grid.states]
        max_diff = max(differences)
        self.differences.append(max_diff)

        if max_diff < EPSILON:
            self.converged = True

        # Update values
        self.values = new_values
        self.policy = new_policy
```

Exercise 7.3

```
def run_vi():
    grid = Grid()
    vi = ValueIteration(grid)

    start = time.time()
    print("Initial values:")
    vi.print_values()
    print()

    for i in range(MAX_ITER):
        vi.next_iteration()
        print("Values after iteration", i + 1)
        vi.print_values_and_policy()
        print()
        if vi.converged:
            break

    end = time.time()
    print("Time to complete", i + 1, "VI iterations")
    print(end - start)
```

Exercise 7.3a and b

```
Values and Policies after iteration 4
(0, 0)    UP       0.0
(0, 1)    UP       0.0
(0, 2)    RIGHT    0.3732480000000001
(1, 0)    UP       0.0
(1, 2)    RIGHT    0.6583680000000002
(2, 0)    UP       0.046656
(2, 1)    LEFT     0.11728799999999999
(2, 2)    RIGHT    0.7964640000000001
(3, 0)    DOWN     0.0
(3, 1)    UP       -100.0
(3, 2)    UP       1.0
(-1, -1)  UP       0.0
Converged: False
```

```
Values and Policies after iteration 10
(0, 0)    UP       0.4490637007006404
(0, 1)    UP       0.5362371998424762
(0, 2)    RIGHT    0.61632756154903
(1, 0)    LEFT     0.3679911227699528
(1, 2)    RIGHT    0.7155133495934718
(2, 0)    LEFT     0.28052219829783076
(2, 1)    LEFT     0.28600606577514903
(2, 2)    RIGHT    0.8174373191274608
(3, 0)    DOWN     0.05225467158005328
(3, 1)    UP       -100.0
(3, 2)    UP       1.0
(-1, -1)  UP       0.0
Converged: False
```

Exercise 7.3c

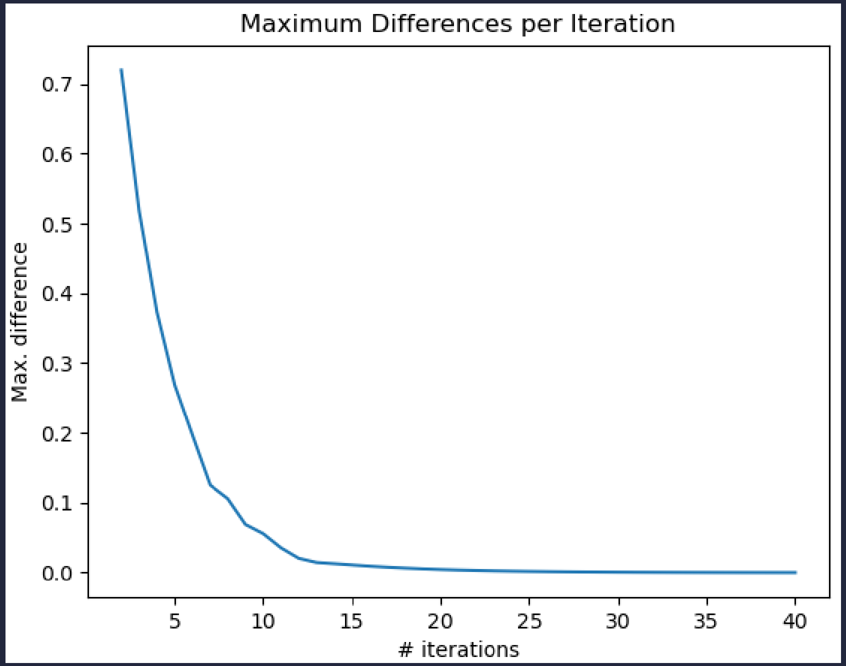
```
Values and Policies after iteration 4
(0, 0)    UP      0.0
(0, 1)    UP      0.0
(0, 2)    RIGHT   0.3732480000000001
(1, 0)    UP      0.0
(1, 2)    RIGHT   0.6583680000000002
(2, 0)    UP      0.046656
(2, 1)    LEFT    0.11728799999999999
(2, 2)    RIGHT   0.7964640000000001
(3, 0)    DOWN    0.0
(3, 1)    UP      -100.0
(3, 2)    UP      1.0
(-1, -1)  UP      0.0
Converged: False
```

```
Values and Policies after iteration 10
(0, 0)    UP      0.4490637007006404
(0, 1)    UP      0.5362371998424762
(0, 2)    RIGHT   0.61632756154903
(1, 0)    LEFT    0.3679911227699528
(1, 2)    RIGHT   0.7155133495934718
(2, 0)    LEFT    0.28052219829783076
(2, 1)    LEFT    0.28600606577514903
(2, 2)    RIGHT   0.8174373191274608
(3, 0)    DOWN    0.05225467158005328
(3, 1)    UP      -100.0
(3, 2)    UP      1.0
(-1, -1)  UP      0.0
Converged: False
```

```
Values and Policies after iteration 40
(0, 0)    UP      0.4800323382261456
(0, 1)    UP      0.5540265799556026
(0, 2)    RIGHT   0.6309786313152921
(1, 0)    LEFT    0.42148665011938496
(1, 2)    RIGHT   0.728236805418173
(2, 0)    LEFT    0.37165369571437096
(2, 1)    LEFT    0.38600516990982364
(2, 2)    RIGHT   0.8293834149435776
(3, 0)    DOWN    0.17564736007905382
(3, 1)    UP      -100.0
(3, 2)    UP      1.0
(-1, -1)  UP      0.0
Converged: True
```

Exercise 7.3d

```
MaxDiff: 100.0  iteration = 0
MaxDiff: 0.7200000000000001     iteration = 1
MaxDiff: 0.5184000000000001     iteration = 2
MaxDiff: 0.3732480000000001     iteration = 3
MaxDiff: 0.26873856000000007    iteration = 4
MaxDiff: 0.19651507200000007    iteration = 5
MaxDiff: 0.12516498432          iteration = 6
MaxDiff: 0.10572027144192       iteration = 7
MaxDiff: 0.0688670804422657     iteration = 8
MaxDiff: 0.05569379313807068 5   iteration = 9
MaxDiff: 0.03541895386310351    iteration = 10
MaxDiff: 0.020406488641122267   iteration = 11
MaxDiff: 0.014469165897111988   iteration = 12
MaxDiff: 0.01276617362193612    iteration = 13
MaxDiff: 0.010953912341080507   iteration = 14
MaxDiff: 0.009252095430991175   iteration = 15
MaxDiff: 0.0077449479888855866  iteration = 16
MaxDiff: 0.006452362030390624   iteration = 17
MaxDiff: 0.0053647123430298205  iteration = 18
MaxDiff: 0.004457891514759699   iteration = 19
MaxDiff: 0.003703988627437288   iteration = 20
MaxDiff: 0.0030769857665578215  iteration = 21
MaxDiff: 0.0025548243622560696  iteration = 22
MaxDiff: 0.0021195066796275974  iteration = 23
MaxDiff: 0.0017564643518459266  iteration = 24
MaxDiff: 0.0014538233682367119  iteration = 25
MaxDiff: 0.0012017914934342455  iteration = 26
MaxDiff: 0.0009922017781994474  iteration = 27
MaxDiff: 0.0008181866026194806  iteration = 28
MaxDiff: 0.0006739451222508019  iteration = 29
MaxDiff: 0.0005545718913637643  iteration = 30
MaxDiff: 0.0004559236087381957  iteration = 31
MaxDiff: 0.0003745096366224443  iteration = 32
MaxDiff: 0.00030739837798915426 iteration = 33
MaxDiff: 0.00025213556900344214 iteration = 34
MaxDiff: 0.00020667261021548033 iteration = 35
MaxDiff: 0.0001693039437246635  iteration = 36
MaxDiff: 0.0001386127553908434  iteration = 37
MaxDiff: 0.00011342429908361984 iteration = 38
MaxDiff: 9.27660922024065e-05   iteration = 39
```



Exercise 7.4

Exercise 7.4.

a) Using an iterative approach to policy evaluation, implement PI for this problem, following:

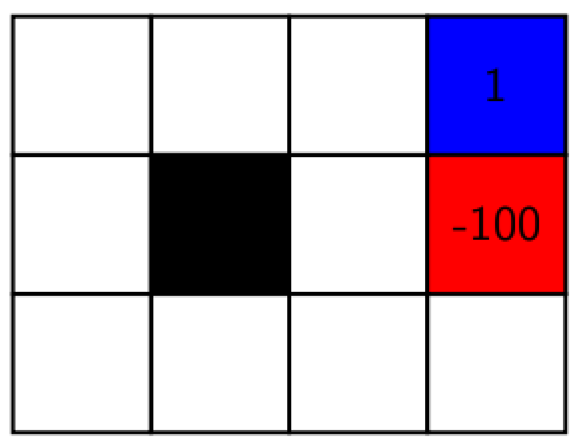
- Set $\pi(s) => \ \forall\ s \in S$, and let $iter = 0$
- Repeat:
  1. Policy evaluation - Solve for $V^{\pi_i}(s)$ (or $Q^{\pi_i}(s,a)$):

     $$V^{\pi_i}(s) = \sum_{s' \in \mathcal{S}} P(s' \mid \pi_i(s), s)\left[R(s, \pi_i(s), s') + \gamma V^{\pi_i}(s')\right] \quad \forall s \in S$$

  2. Policy improvement - Update policy:

     $$\pi_{i+1}(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}} P(s' \mid a, s)\left[R(s, a, s') + \gamma V^{\pi_i}(s')\right]$$

  3. $iter = iter + 1$
- until $\pi_i(s) = \pi_{i-1}(s)$



States – Positions on tiles.

Terminal States – Add a pseudo-extra absorbing state called “EXIT_STATE”
$T(exited, a, exited) = 1$

Actions = {U, D, L, R}

(non-deterministic)

Rewards

R([3,2], exit)=1

R([3,1], exit=-100

Discount Factor
$\gamma = 0.9$

Exercise 7.4a

```
while not converged: # Convergence criteria - (new_policy == policy)
  # policy_evaluation uses linear algebra to derive a set of values
  # as shown above
  values <- policy_evaluation(policy)
  # compute a new policy using the previously computed set of values
  # Looks like value iteration, but only a single iteration is performed.
  new_policy <- policy_improvement(values)
  if new_policy == policy:
    # Convergence criteria matched, return the policy
    return policy
  # Convergence criteria not matched, set the policy and perform
  # another iteration
  policy <- new_policy
```

Exercise 7.4

```python
class PolicyIteration:
    def __init__(self, grid):
        self.grid = grid
        self.values = {state: 0 for state in self.grid.states}
        self.policy = {pi: RIGHT for pi in self.grid.states}
        self.r = [0 for s in self.grid.states]
        self.USE_LIN_ALG = False
        self.converged = False
```

Exercise 7.4

```
class PolicyIteration:
    def __init__(self, grid):
        self.grid = grid
        self.values = {state: 0 for state in self.grid.states}
        self.policy = {pi: RIGHT for pi in self.grid.states}
        self.r = [0 for s in self.grid.states]
        self.USE_LIN_ALG = False
        self.converged = False

        # Full transition matrix (P) of dimensionality |S|x|A|x|S| since its
        # not specific to any one policy. We'll slice out a |S|x|S| matrix
        # from it for each policy evaluation
        # t model (lin alg)
        self.t_model = np.zeros([len(self.grid.states), len(self.grid.actions),
                                 len(self.grid.states)])
        for i, s in enumerate(self.grid.states):
            for j, a in enumerate(self.grid.actions):
                transitions = self.grid.get_transition_probabilities(s, a)
                for next_state, prob in transitions.items():
                    self.t_model[i][j][self.grid.states.index(next_state)] = prob
```

Exercise 7.4

```
class PolicyIteration:
    def __init__(self, grid):

        ...

        # Reward vector
        r_model = np.zeros([len(self.grid.states)])
        for i, s in enumerate(self.grid.states):
            r_model[i] = self.grid.get_reward(s)
        self.r_model = r_model

        # lin alg policy
        la_policy = np.zeros([len(self.grid.states)], dtype=np.int64)
        for i, s in enumerate(self.grid.states):
            la_policy[i] = 3 # Allocate arbitrary initial policy
        self.la_policy = la_policy
```

Exercise 7.4

```python
class PolicyIteration:
    def next_iteration(self):
        new_values = dict()
        new_policy = dict()

        self.policy_evaluation()
        new_policy = self.policy_improvement()
        self.convergence_check(new_policy)
```

Exercise 7.4

```
class PolicyIteration:
    def policy_evaluation(self):
        if not self.USE_LIN_ALG:
            # use 'naive'/iterative policy evaluation
            value_converged = False
            while not value_converged:
                new_values = dict()
                for s in self.grid.states:
                    total = 0
                    for stoch_action, p in self.grid.stoch_action(self.policy[s]).items():
                        # Apply action
                        s_next = self.grid.attempt_move(s, stoch_action)
                        total += p * (self.grid.get_reward(s) + (self.grid.discount * self.values[s_next]))
                    # Update state value with best action
                    new_values[s] = total

                # Check convergence
                differences = [abs(self.values[s] - new_values[s]) for s in self.grid.states]
                if max(differences) < EPSILON:
                    value_converged = True

                # Update values and policy
                self.values = new_values
```

Exercise 7.4

```
class PolicyIteration:
    def policy_evaluation(self):
        else:
            # use linear algebra for policy evaluation
            # V^pi = R + gamma T^pi V^pi
            # (I - gamma * T^pi) V^pi = R
            # Ax = b; A = (I - gamma * T^pi),  b = R
            state_numbers = np.array(range(len(self.grid.states)))  # indices of every state
            t_pi = self.t_model[state_numbers, self.la_policy]
            values = np.linalg.solve(np.identity(len(self.grid.states)) - (self.grid.discount * t_pi),
                   self.r_model)
            self.values = {s: values[i] for i, s in enumerate(self.grid.states)}
```

Exercise 7.4

```
def policy_improvement(self):
    if self.USE_LIN_ALG:
        new_policy = {s: self.grid.actions[self.la_policy[i]] for i, s in enumerate(self.grid.states)}
    else:
        new_policy = {}

    for s in self.grid.states:
        # Keep track of maximum value
        action_values = dict()
        for a in self.grid.actions:
            total = 0
            for stoch_action, p in self.grid.stoch_action(a).items():
                # Apply action
                s_next = self.grid.attempt_move(s, stoch_action)
                total += p * (self.grid.get_reward(s) + (self.grid.discount * self.values[s_next]))
            action_values[a] = total
        # Update policy
        new_policy[s] = dict_argmax(action_values)
    return new_policy
```

Exercise 7.4

```python
def convergence_check(self, new_policy):
    if new_policy == self.policy:
        self.converged = True

    self.policy = new_policy
    if self.USE_LIN_ALG:
        for i, s in enumerate(self.grid.states):
            self.la_policy[i] = self.policy[s]
```

Exercise 7.4b

Let $P^\pi \in R^{|S|\times|S|}$ be a matrix containing probabilities for each transition under some policy π, where:

$$P^\pi_{ij} = P(s_{t+1} = j \mid s_t = i, a_t = \pi(s_t))$$

Calculate the size of P^π in this gridworld, when the special pseudo-state *exited* is included.

Excluding the obstacle at (1,1) as an unreacable state, but include the exit state, then the size of P^π is 12x12.

Including the obstacle brings the size to 13x13 but the transition probability to (1,1) is 0 from all states.

Exercise 7.4c

c) Set the policy to move *right* everywhere, $\pi(s) = \text{RIGHT}\ \forall\ s \in S$. Calculate the row of P^{RIGHT} corresponding to an initial state at the bottom left corner, (0,0) of the gridworld.

Let the state indices be ordered:

$$((0,0),(1,0),(2,0),(3,0),(0,1),(2,1),(3,1),(0,2),(1,2),(2,2),(3,2),\\ \text{EXIT\_STATE})$$

The row is $P^{\text{RIGHT}}_{((0,0),j)} = [0.1\ \ 0.8\ \ 0\ \ 0\ \ 0.1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0]$

i.e. The probability of starting in state (0,0) and moving to state j==s' when performing the action RIGHT has the probability ___.

$P((0,0)|(0,0),\text{RIGHT}) = 0.1$

The probability of starting in state (0,0) and moving to state (0,0) when action RIGHT is performed has probability 0.01 of occurring - this is the first perpendicular state.

$P((0,1)|(0,0),\text{RIGHT}) = 0.8$

The probability of starting in state (0,0) and moving to state (1,0) when action RIGHT is performed has probability of 0.8 occurring - this is the chosen direction

$P((0,0)|(0,0),\text{RIGHT}) = 0.1$

The probability of starting in state (0,0) and moving to state (2,0) when action RIGHT is performed has probability of 0.8 occurring - this is the second perpendicular state.

Exercise 7.4d

d) Write a function that calculate the row of P^{RIGHT} corresponding to an initial state at the bottom left corner, (0,0) of the gridworld for any action or deterministic $\pi((0,0))$.