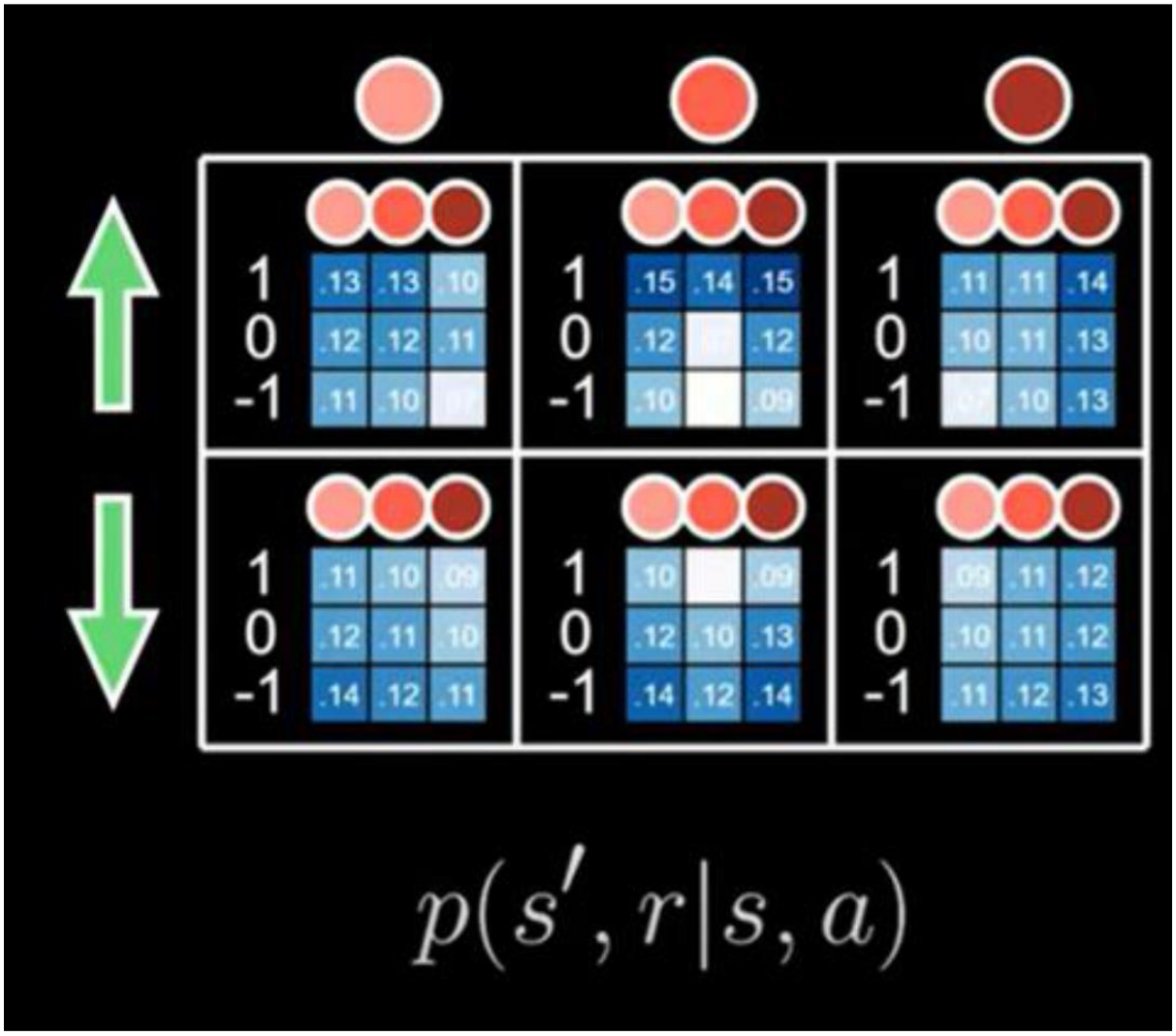
MDP Assignment

Assignment 1

1.1 Reflect on how versions 1 and 2 can deal with this case.
See course slides for detail on the MDP.



States:

$$S \in \{Color1, Color2, Color3\}$$

Where Color1, Color2, and Color3 are discrete variables that represent 3 colors.

Actions

$$A \in \{\uparrow, \downarrow\}$$

Where $\uparrow, \downarrow$ are discrete actions for each state.

Rewards

$$R \in \{-1,0,1\}$$

Where -1,0,1 are discrete rewards.

Version1

Version 1 is well designed for This MDP scenario because the **environment** defines the stochastic model for the next **state** and **reward** based on the **current state** and **action**.

$$p(s', r \mid s, a)$$

This means that the table in Figure defines the probability of getting to any **state** and receiving the **reward**.

Version2

Version 2 is not that suitable for this scenario. In such a version we have

$$p(s' \mid s, a)$$

Which represents the probability of the **next state** based on the **current state** and **taken action**. To calculate these probabilities based on Table:

$$p(s' \mid s, a) = \sum_{r \in R} p(s', r \mid s, a)$$

In this version the **reward** is defined as a deterministic function:

$$r = f(s, a, s')$$

But we can't define this **reward** in a deterministic way for this **environment**.
Hence this case couldn't be represented with this version.

1.2 Implement a generic stochastic MDP in Python (version 2).

A **stochastic Markov Decision Process (MDP)** is a mathematical model used to make decision-making in environments where outcomes depend on actions with some uncertainty.

MDP is defined in mathematical terms by a set of states S, a set of actions A, and $P(s', r \mid s, a)$ (or transition probabilities $P(s' \mid s, a)$, and a reward function $f(s, a, s')$).
Here, $P(s', r \mid s, a)$, denotes the probability of transitioning to state s' from state s after taking action a, and getting reward r.

The objective in an MDP is to find a policy π, a function from states to actions, that maximizes the Value function(V_π) with policy π. the Value function(V_π) is typically calculated as the expected return G_t. The return G_t is sum of all the future rewards, formalized as:

$$G_t = \sum_{i=t+1}^{T} \gamma^{i-t-1} R_t$$

or

$$G_t = R_t + \gamma G_{t+1}$$

Where $0 \leq \gamma \leq 1$ is the discount factor, which shows how we should weight the future rewards. The nearest rewards will be more significant for the model.

The transition probabilities determine the likelihood of moving from one state to another and receive reward given a specific action and state. The objective of an MDP is to find a policy that maximizes the expected sum of rewards over time at each state:

$$V(s) = \mathbb{E}[G_t \mid s]$$

Initialization and Configuration

The **MDP** class is initialized with five main components: states, actions, transitions, rewards, and current state.

```
class MDP:
    def __init__(self, states, terminal_states, transitions, current_state=None):
        self.states = states
        self.terminal_states = terminal_states
        self.actions = {state: list(action) for state, action in transitions.items()}
        self.transitions = transitions
        if current_state is None:
            self.current_state = random.choice([s for s in states if s not in self.terminal_states])
        else:
            self.current_state = current_state
```

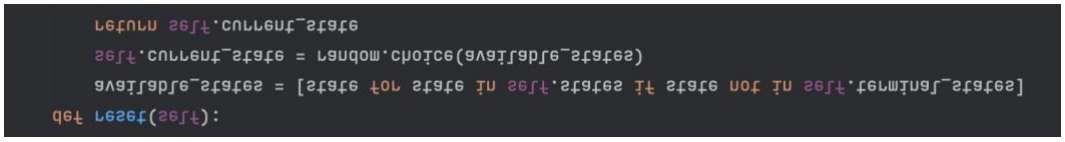
1. **States**: Represents the different conditions that can exist within the environment as a list.

2. **Terminal States**: Represents the terminal (end) state of the Model.

3. **Transitions**: Defines actions and it's the probability of moving from one state to another and receiving a reward given a specific action.

Methods

- **reset()**: Resets the MDP to a random initial state within not terminal.



- **step(action)**: Takes an action, determines the next state from current state based on the transition probabilities, updates the current state, and returns the new state, reward, and whether the state is terminal.

```python
def step(self, action):
    if self.current_state in self.terminal_states:
        raise Exception("Already in a terminal state")
    if action not in self.actions[self.current_state]:
        raise ValueError("Invalid action")

    outcomes = self.transitions[self.current_state][action]
    if not outcomes:
        print(f"No transitions available from this state({self.current_state}).")
        self.current_state = None
        return self.current_state, 0, True

    possible_states = list(outcomes.keys())
    probabilities = [outcomes[state][0] for state in possible_states]

    next_state = random.choices(possible_states, weights=probabilities)[0]
    reward = outcomes[next_state][1]

    self.current_state = next_state

    done = self.current_state in self.terminal_states or not self.get_available_actions()
    return next_state, reward, done
```

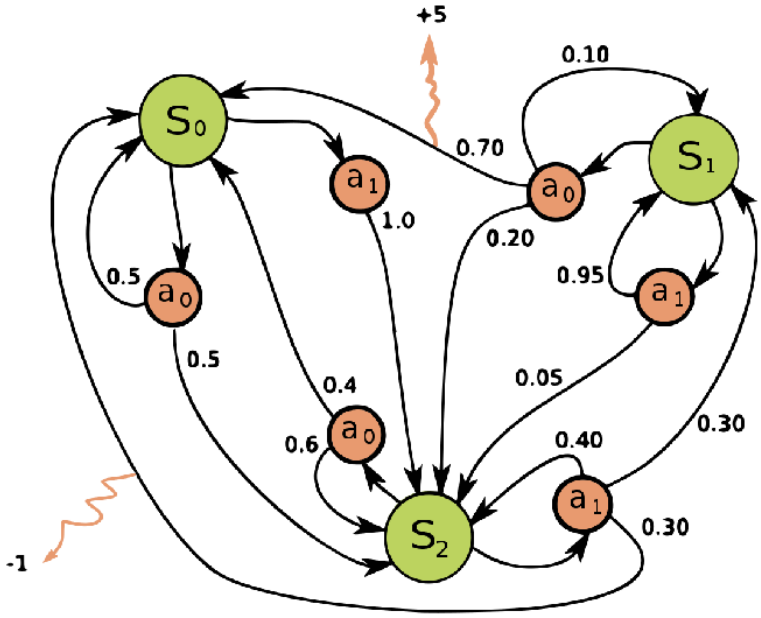
- **get_available_actions()**: Returns a list of valid actions that can be taken from the current state.

```python
def get_available_actions(self):
    return self.actions[self.current_state]
```

https://chat.openai.com/share/3ebc8fd4-23fe-46de-b68c-3e9bf261b4a1

ChatGPT was mostly used to debug issues and implement mostly Python features and functionality, the MDP logic was done with knowledge from lectures and presentations.

1.3 Implement the MDP in the image



States:

$$S \in \{S_0, S_1, S_2\}$$

Where S_0, S_1 *and* S_2 are discrete variables that represent 3 states.

```
states = [
    'S0',
    'S1',
    'S2'
]
```

Actions

$$A \in \{ a_0 , a_1 \}$$

Where a_0 , a_1 are discrete actions for each state.

Rewards

$$R \in \{-1,5\}$$

Where -1,5 are discrete rewards.

```
transitions = {
    'S0': {
        'a0': {'S0': [0.5, 0], 'S2': [0.5, 0]},
        'a1': {'S2': [1, 0]}
    },
    'S1': {
        'a0': {'S0': [0.7, 5], 'S2': [0.2, 0], 'S1': [0.1, 0]},
        'a1': {'S1': [0.95, 0], 'S2': [0.05, 0]}
    },
    'S2': {
        'a1': {'S0': [0.3, -1], 'S2': [0.4, 0], 'S1': [0.3, 0]},
        'a0': {'S0': [0.4, 0], 'S2': [0.6, 0]}
    }
}
```

Here transitions table define $p(s', r \mid s, a)$

MDP Test Run

```
mdp1_3 = MDP(states1_3, [], transitions1_3)
mdp1_3.reset()

for i in range(10):
    current_state = mdp1_3.current_state
    available_actions = mdp1_3.get_available_actions()
    action = random.choice(available_actions)
    new_state, reward, done = mdp1_3.step(action)

    print(f"{current_state} -> {action} -> {new_state} | Reward: {reward}")

    if done:
        print("Reached a terminal state.")
        break
```

Executed at 2024.05.14 13:01:19 in 15ms

```
S2 -> a0 -> S0 | Reward: 0
S0 -> a0 -> S2 | Reward: 0
S2 -> a1 -> S2 | Reward: 0
S2 -> a1 -> S2 | Reward: 0
S2 -> a1 -> S2 | Reward: 0
S2 -> a1 -> S1 | Reward: 0
S1 -> a1 -> S1 | Reward: 0
S1 -> a0 -> S0 | Reward: 5
S0 -> a0 -> S2 | Reward: 0
S2 -> a1 -> S0 | Reward: -1
```

Assignment 2

Assignment 2, MDP 2

Assignment 2, MDP 2

Allow the user to control the probability of slipping $0 \leq p < 1$.

Allow for a parametrized cost of living.

This scenario including Slipping and Cost of living require extension to generic MDP Code:

The slippery and cost of living is implemented inside enviornment and could be defined as hyperparameter

- Add new parameters is_slippery, slippery_factor, and cost_of_living to Initialization

```
class MDP:
    def __init__(self, states, terminal_states, transitions, current_state=None, slippery_factor = 0.1, is_slippery = False, cost_of_living = 0.01 ):
        self.states = states
        self.terminal_states = terminal_states
        self.actions = {state: list(action) for state, action in transitions.items()}
        self.transitions = transitions
        self.observation_space = len(states)
        self.action_space = len(self.actions)
        self.is_slippery = is_slippery
        self.slippery_factor = slippery_factor
        self.cost_of_living = cost_of_living
        if current_state is None:
            self.current_state = random.choice([s for s in states if s not in self.terminal_states])
        else:
            self.current_state = current_state
```

- Add logic inside step function to take random step from avaliable $A(s)$ is Slippery is enabled with defined probability

```
if self.is_slippery and random_number_generator.random() < self.slippery_factor:
    action = random.choice(self.get_available_actions())
    print(f"Slipped")

outcomes = self.transitions[self.current_state][action]
```

- Add logic inside step function to decrement reward by cost 0f living at each step

```
reward = outcomes[next_state][1] - self.cost_of_living
```

States:

$$S \in \{1, 2, 3,4,5\}$$

Where $1, 2,3,4\ and\ 5$ are discrete states inside the grid(starting from left).

Actions

$$A \in \{ \leftarrow, \rightarrow \}$$

Where $\leftarrow, \rightarrow$ are discrete actions for each state, except bound left and bound right.

Rewards

$$R \in \{-1,1\}$$

Where -1,1 are discrete rewards.

Transition table is defined with function:

```
def create_transitions(num_states):
    transitions = {}
    for state in range(1, num_states + 1):
        state_str = str(state)
        transitions[state_str] = {}
        if state < num_states:
            transitions[state_str]['r'] = {str(state + 1): [1, 1 if state == num_states - 1 else 0]}
        if state > 1:
            transitions[state_str]['l'] = {str(state - 1): [1, -1 if state == 2 else 0]}
    return transitions

num_states = 5
transitions2_1 = create_transitions(num_states)
```

Or manually coded looks like:

```
# transitions2_1 = {
#     '1' : {
#         'r' : {'2' : [1, 0]}
#     },
#     '2' : {
#         'l' : {'1' : [1, -1]},
#         'r' : {'3' : [1, 0]}
#     },
#     '3' : {
#         'l' : {'2' : [1, 0]},
#         'r' : {'4' : [1, 0]}
#     },
#     '4' : {
#         'l' : {'3' : [1, 0]},
#         'r' : {'5' : [1, 1]}
#     },
#     '5' : {
#         'l' : {'4' : [1, 0]}
#     }
# }
```

MDP2.1 initialization and test run

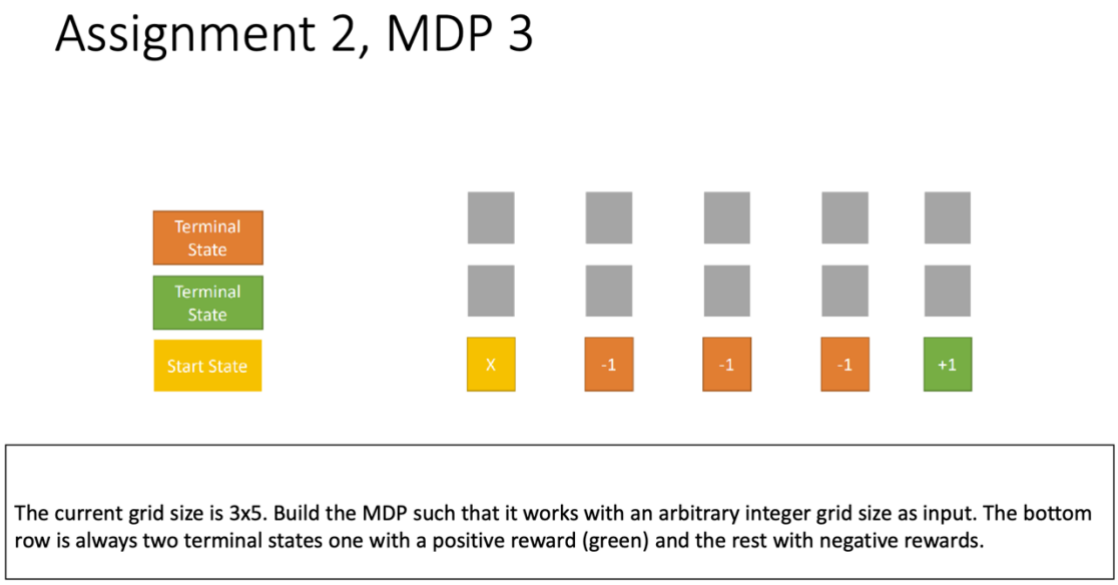
```
mdp2_1 = MDP(states2_1, terminal_states2_1, transitions2_1, slippery_factor=0.6, is_slippery=True, cost_of_living=0.1)
mdp2_1.reset()

while True:
    current_state = mdp2_1.current_state
    available_actions = mdp2_1.get_available_actions()
    action = random.choice(available_actions)
    print(f"Chosen action:", action)
    new_state, reward, done = mdp2_1.step(action)

    if done:
        print("Reached a terminal state.")
        break
```

Executed at 2024.05.15 08:47:32 in 32ms

```
Chosen action: l
Slipped
3 -> r -> 4 | Reward: -0.1
Chosen action: l
4 -> l -> 3 | Reward: -0.1
Chosen action: l
Slipped
3 -> r -> 4 | Reward: -0.1
Chosen action: r
Slipped
4 -> r -> 5 | Reward: 0.9
Reached a terminal state.
```

Here it is possible to see when agent select (randomly) action l at the step 2 the action r is executed instead due to slippery factor.

Assignment 2, MDP 3



The Generic MDP class is capable of capturing such MDP model, we need to create staes and transitions functions $f(n, m)$, where n, m are size of the grid

States

The state is discrete number, the states mapped from top left increasing horizontally and starting from each new row.

$$s = j + (i - 1) * m$$

```
def create_states(h, w):
    states = []
    for i in range(h):
        for j in range(1, w + 1):
            states.append(str(j + w * i))
    return states
```

Initial State

$$s_0 \in \{1\}$$

Terminal State

$$s_t \in \{last\ index\ of\ states\ array\}$$

Actions

$$A \in \{\leftarrow, \uparrow, \rightarrow, \downarrow\}$$

Where $\leftarrow, \uparrow, \rightarrow, \downarrow$ are discrete actions.

Rewards

$$R \in \{-1,1\}$$

Where -1,1 are discrete rewards.

Transitions

Transitions are defined with function:

```
def create_transitions(h,w):
    transitions = {}
    for i in range(h):
        for j in range(1, w + 1):
            state_str = str(j + w * i)
            transitions[state_str] = {}
            if i != 0:
                transitions[state_str]['u'] = {str(j + w * (i-1) ): [1, 0]}
            if i != h - 1:
                reward = 1 if j == w and i == h - 2 else -1 if j != 1 and i == h - 2 else 0
                transitions[state_str]['d'] = {str(j + w * (i + 1) ): [1, reward]}
            if j != 1:
                reward = -1 if j != 2 and i == h - 1 else 0
                transitions[state_str]['l'] = {str((j - 1) + w * i ): [1, reward]}
            if j != w:
                reward = 1 if j == w - 1 and i == h - 1 else -1
                transitions[state_str]['r'] = {str((j + 1) + w * i ): [1, reward]}

    return transitions
```

MDP2.2 initialization and test run

```
h = 3
w = 5
states2_2 = create_states(h, w)
current_state2_2 = '1'
terminal_states2_2 = [states2_2[-1]]
transitions2_2 = create_transitions(h, w)


mdp2_2 = MDP(states2_2, terminal_states2_2,transitions2_2, current_state=current_state2_2, is_slippery=True, slippery_factor = 0.5, cost_of_living = 0.01)

for i in range(50):
    current_state = mdp2_2.current_state
    available_actions = mdp2_2.get_available_actions()
    action = random.choice(available_actions)
    print("Actions chosen:", action)
    new_state, reward, done = mdp2_2.step(action)

    if done:
        print("Reached a terminal state.")
        break
```

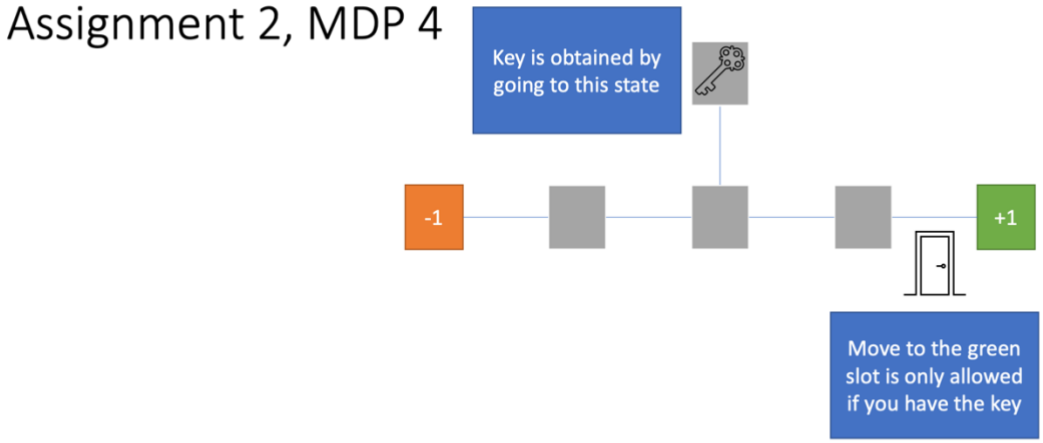
Here MDP is initialized where h, w are hyperparams to create grid, also here defined the slippery flag and slippery factor and cost of living.

Test Run

```
Actions chosen: r
1 -> r -> 2 | Reward: -1.01
Actions chosen: r
Slipped
2 -> d -> 7 | Reward: -0.01
Actions chosen: d
Slipped
7 -> u -> 2 | Reward: -0.01
Actions chosen: d
Slipped
2 -> r -> 3 | Reward: -1.01
Actions chosen: d
Slipped
3 -> l -> 2 | Reward: -0.01
Actions chosen: l
2 -> l -> 1 | Reward: -0.01
Actions chosen: r
Slipped
1 -> d -> 6 | Reward: -0.01
Actions chosen: r
Slipped
6 -> r -> 7 | Reward: -1.01
Actions chosen: r
Slipped
7 -> u -> 2 | Reward: -0.01
Actions chosen: d
2 -> d -> 7 | Reward: -0.01
Actions chosen: r
7 -> r -> 8 | Reward: -1.01
Actions chosen: u
Slipped
8 -> d -> 13 | Reward: -1.01
Actions chosen: r
13 -> r -> 14 | Reward: -1.01
Actions chosen: r
14 -> r -> 15 | Reward: 0.99
Reached a terminal state.
```

Here we could see that enviornment behave properly

Assignment 2, MDP 4



Your Generic MDP should be able to deal with this. Do not make a customized version for this environment. If needed fix your generic MDP.

Generic MDP is still capable of capturing this scenario. I will use twice more states to represent the problem, each state is represented as vector $s = \begin{bmatrix} position \\ key \end{bmatrix}$ where **key** is 0 if the key has not been obtained and 1 if it has.

States

The state is discrete number, the states mapped from top left increasing horizontally and starting from each new row.

$$s = \begin{bmatrix} position \\ key \end{bmatrix}$$

All positions except key and terminal positions will have 2 states. Positions mapped in the way that horizonal is represented as numbers from 1 to 5 and key position is 6.
States are created with function (used tupples instead vectors for python):

```
def create_states():
    states = []
    for i in range(1,6):
        if i != 5:
            states.append(str((i,0)))
        states.append(str((i,1)))
    states.append(str((6,1)))
    return states
```

```
['(1, 0)',
 '(1, 1)',
 '(2, 0)',
 '(2, 1)',
 '(3, 0)',
 '(3, 1)',
 '(4, 0)',
 '(4, 1)',
 '(5, 1)',
 '(6, 1)']
```

Terminal States

$$s_t \in \{(1,0),(1,1),(5,1)\}$$

Actions

$$A \in \{\leftarrow, \uparrow, \rightarrow, \downarrow\}$$

Where $\leftarrow, \uparrow, \rightarrow, \downarrow$ are discrete actions.

Rewards

$$R \in \{-1,1\}$$

Where -1,1 are discrete rewards.

Transitions

```python
def create_transitions(states):
    transitions = {}
    for state in states:
        transitions[state] = {}
        position = ast.literal_eval(state)[0]
        key = ast.literal_eval(state)[1]

        if 1 < position < 4:
            reward_left = -1 if position == 2 else 0
            transitions[state]['l'] = {str((position-1, key)) : [1, reward_left]}
            transitions[state]['r'] = {str((position+1, key)) : [1, 0]}

        if position == 3:
            transitions[state]['u'] = {'(6,1)' : [1, 0]}

        if position == 6:
            transitions[state]['d'] = {'(3,1)' : [1, 0]} # good idea to add small reward to this step

        if position == 4:
            transitions[state]['l'] = {str((position-1, key)) : [1, 0]}
            if key == 1:
                transitions[state]['r'] = {'(5,1)' : [1, 1]}


    return transitions
```

Transitions are defined with this function and result is:

```
{'(1, 0)': {},
 '(1, 1)': {},
 '(2, 0)': {'l': {'(1, 0)': [1, -1]}, 'r': {'(3, 0)': [1, 0]}},
 '(2, 1)': {'l': {'(1, 1)': [1, -1]}, 'r': {'(3, 1)': [1, 0]}},
 '(3, 0)': {'l': {'(2, 0)': [1, 0]},
  'r': {'(4, 0)': [1, 0]},
  'u': {'(6,1)': [1, 0]}},
 '(3, 1)': {'l': {'(2, 1)': [1, 0]},
  'r': {'(4, 1)': [1, 0]},
  'u': {'(6,1)': [1, 0]}},
 '(4, 0)': {'l': {'(3, 0)': [1, 0]}},
 '(4, 1)': {'l': {'(3, 1)': [1, 0]}, 'r': {'(5,1)': [1, 1]}},
 '(5, 1)': {},
 '(6, 1)': {'d': {'(3,1)': [1, 0]}}}
```

Also this is good idea to give some small reward to the key obtain, that will especially help with agent learning, so agent will be likely going for the key to go for some reward, faster, than randomly be the case where it take the key and then open the door and receive reward only in such scenarion. So agent will learn faster.

MDP2.4 initialization and test run

```python
states2_4 = create_states()

terminal_states2_4 = ['(1, 0)',
                      '(1, 1)',
                      '(5, 1']

initial_state2_4 = '(3, 0)'

transitions2_4 = create_transitions(states2_4)

mdp2_4 = MDP(states2_4, terminal_states2_4, transitions2_4, current_state=initial_state2_4, slippery_factor=0.3, is_slippery=True, cost_of_living=0.01)

for i in range(50):
    current_state = mdp2_4.current_state
    available_actions = mdp2_4.get_available_actions()
    action = random.choice(available_actions)
    print("Actions chosen:", action)
    new_state, reward, done = mdp2_4.step(action)

    if done:
        print("Reached a terminal state.")
        break
```

In this certain scenarion I decided to choose initial state to (3,0) as middle of the map without key, to check the behaviour of enviornment on practice.

Test Run

```
Actions chosen: r
(3, 0) -> r -> (4, 0) | Reward: -0.01
Actions chosen: l
(4, 0) -> l -> (3, 0) | Reward: -0.01
Actions chosen: r
(3, 0) -> r -> (4, 0) | Reward: -0.01
Actions chosen: l
(4, 0) -> l -> (3, 0) | Reward: -0.01
Actions chosen: u
(3, 0) -> u -> (6, 1) | Reward: -0.01
Actions chosen: d
Slipped
(6, 1) -> d -> (3, 1) | Reward: -0.01
Actions chosen: l
(3, 1) -> l -> (2, 1) | Reward: -0.01
Actions chosen: r
(2, 1) -> r -> (3, 1) | Reward: -0.01
Actions chosen: u
(3, 1) -> u -> (6, 1) | Reward: -0.01
Actions chosen: d
Slipped
(6, 1) -> d -> (3, 1) | Reward: -0.01
Actions chosen: u
(3, 1) -> u -> (6, 1) | Reward: -0.01
Actions chosen: d
(6, 1) -> d -> (3, 1) | Reward: -0.01
Actions chosen: r
Slipped
(3, 1) -> r -> (4, 1) | Reward: -0.01
Actions chosen: l
(4, 1) -> l -> (3, 1) | Reward: -0.01
Actions chosen: r
(3, 1) -> r -> (4, 1) | Reward: -0.01
Actions chosen: r
(4, 1) -> r -> (5, 1) | Reward: 0.99
Reached a terminal state.
```