Task 3.1D: Exact policy iteration implementation for MDPs

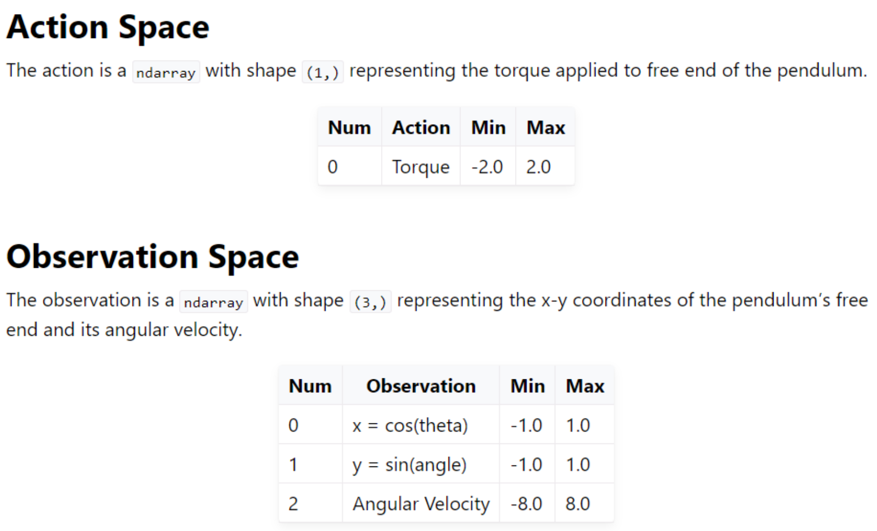
Introduction

In this task, we are to implement a discretized counterpart of the environment we have seen in Task 1.1P. In fact, the environment we previously discussed was the **'Pendulum-v0'** (Refer to this link: Pendulum - Gym Documentation (gymlibrary.dev)).

The following report aims to talk in detail about the environment and the results we have received from the environment we have created from scratch. This report has the devised environment on the following page and the Jupyter Notebook attached (The GitHub link is also provided).

About our Environment

Our environment consists of actions and observations that were Continuous in nature, as seen in the picture below. On the next page, we have made an environment where we have four states, each showing the positioning of the pendulum. Essentially, we have discretized both the action and the observation space of the environment from the image below. As we can see in the below picture, since the torque applied is usually in the range of -2 Nm to 2 Nm, our MDP discretized the Torque applied into four separate actions, which were derived based on both Magnitude and Direction. Likewise, the observation space consists of both the Cartesian coordinates and the Angular Velocity. For our case, we ignore the Angular velocity and instead derive the positions of the pendulum i.e., Bottom, Left, Right, and Top. Both spaces are derived in the next picture on the next page.



Action Space

The action is a `ndarray` with shape `(1,)` representing the torque applied to free end of the pendulum.

Num	Action	Min	Max
0	Torque	-2.0	2.0

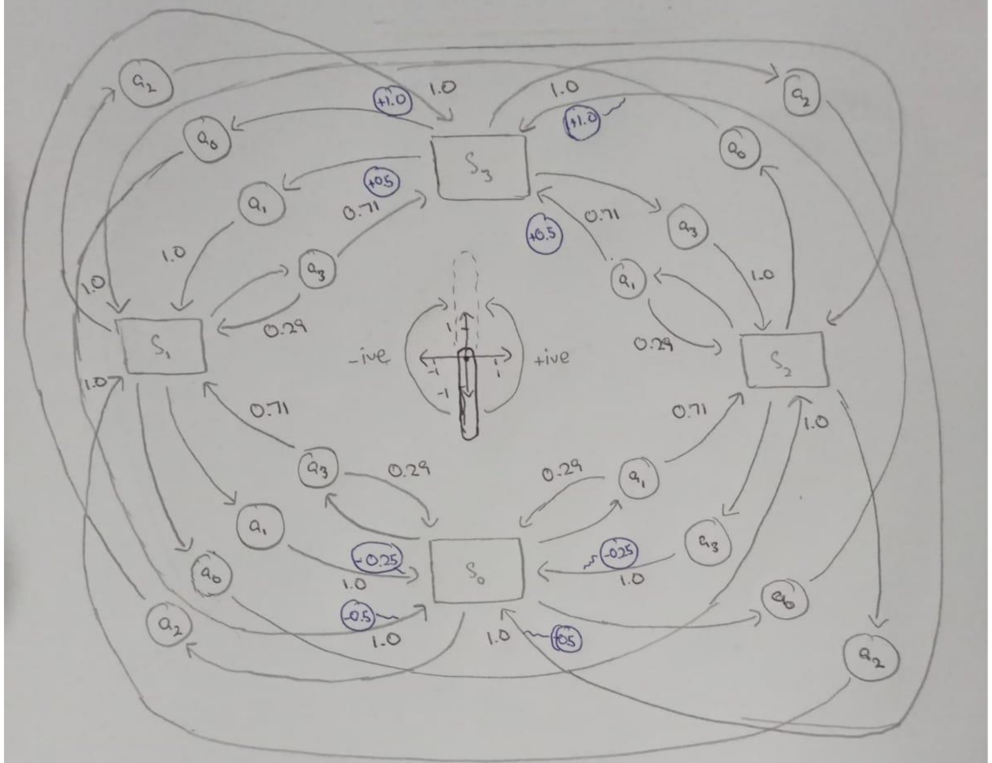
Observation Space

The observation is a `ndarray` with shape `(3,)` representing the x-y coordinates of the pendulum's free end and its angular velocity.

Num	Observation	Min	Max
0	x = cos(theta)	-1.0	1.0
1	y = sin(angle)	-1.0	1.0
2	Angular Velocity	-8.0	8.0

Since the goal of the environment is to reach the Topmost position, we have made a model where the agent is rewarded whenever it reaches the Top position, and likewise punished whenever it reaches the Bottom position, regardless of how the torque is applied. This way, the agent will learn to always maintain the topmost position.

Pendulum -v0 (MDP Implementation):



- Transition Probabilities : (Mentioned in Diagram!)
- Rewards : (Mentioned in Diagram!)
- Actions :
  - a_0 : 2 Nm ; Counter-Clockwise (+ive)
  - a_1 : 1 Nm ; +ive
  - a_2 : 2 Nm ; Clockwise (-ive)
  - a_3 : 1 Nm ; -ive

States :
- S_0 : Bottom Position ; (0,-1)
- S_1 : Left " ; (-1,0)
- S_2 : Right " ; (+1,0)
- S_3 : Top " ; (0,+1)

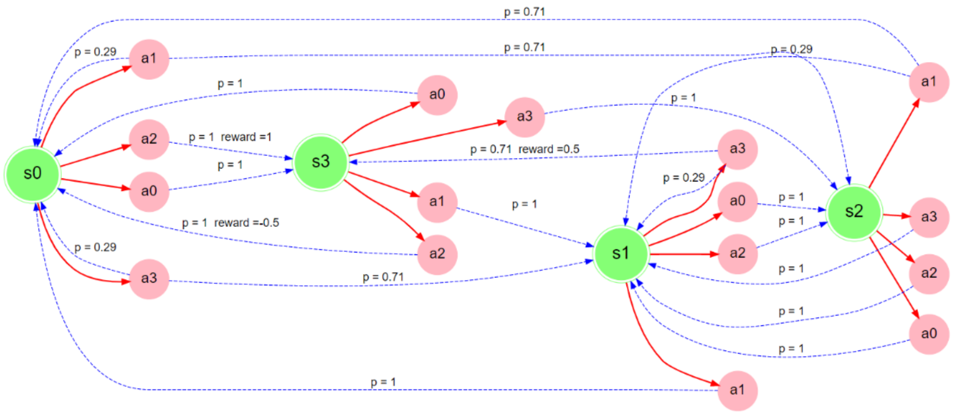
Based on the knowledge of the implementation of the Policy Iteration and the MDPs, we have also altered the reward system of the environment. The end goal of the environment from Task 1.1P was initially zero since the system was based on the Angular Velocity of the Pendulum and the Torque applied to it. Now the reward system is based solely on the positioning of the pendulum, and the transitions associated with the movement of the pendulum.

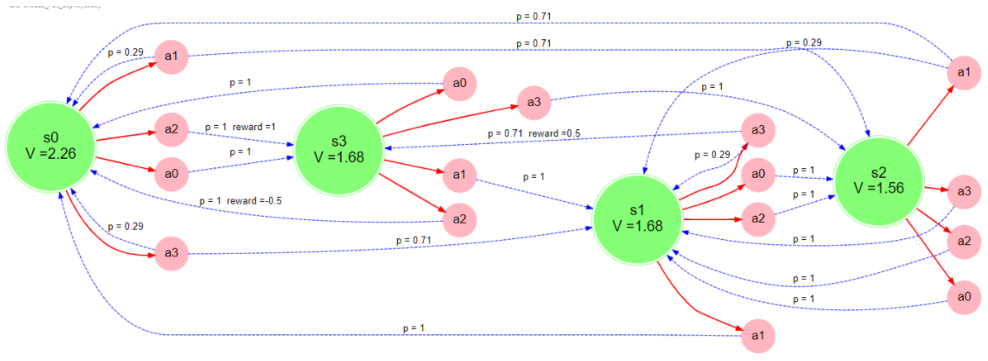
Please also note that you can also find the same Notebook in the GitHub link as shown below as well:

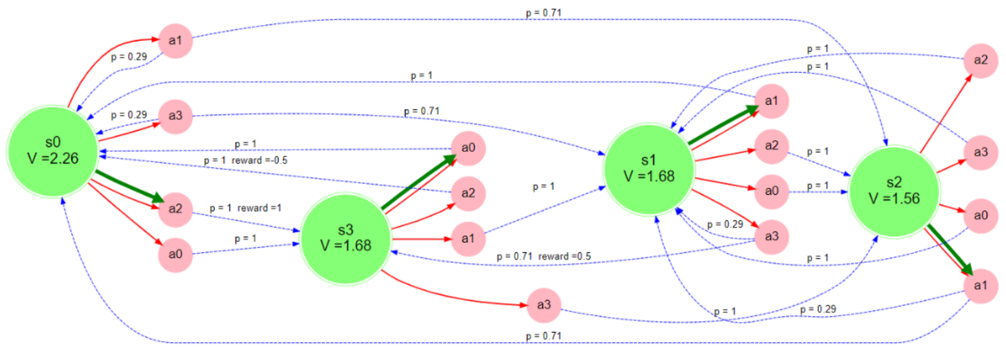
GitHub Link: M-S-Kashif/SIT788_Task_3.1D: Discretized Simulation of ‘Pendulum-v0’ (github.com)

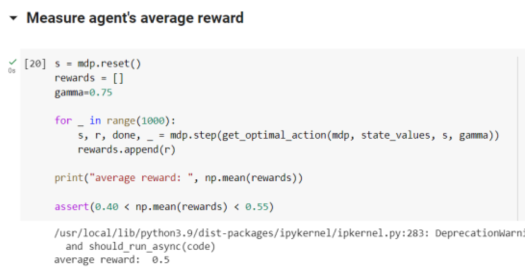
Results

The following are some of the results we get from our environment. The following images below contain representations of the environment as MDPs and how the implementation achieved some of the optimal actions. We also have an image that shows the mean average reward received by the agent over 1000 episodes, which in fact is true according to our assertion.









Measure agent's average reward

```
[20] s = mdp.reset()
     rewards = []
     gamma=0.75

     for _ in range(1000):
         s, r, done, _ = mdp.step(get_optimal_action(mdp, state_values, s, gamma))
         rewards.append(r)

     print("average reward: ", np.mean(rewards))

     assert(0.40 < np.mean(rewards) < 0.55)
```

```
/usr/local/lib/python3.9/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarni
  and should_run_async(code)
average reward:  0.5
```

References

- Pendulum - Gym Documentation (gymlibrary.dev)
- Classic Control - Gym Documentation (gymlibrary.dev)

Task 3.1D: Exact policy iteration implementation for MDPs

Objective: " Your implementation should employ a linear system (while referring Task 1.1P) to find the exact solution for a policy that can achieve the goal. Your report, in the other hand, should refer to the relevant pseudocode in the your Jupyter Notebook. You are also expected to show a graphical model similar to that used in Week 3 Practical corresponding to your state and transition probabilities. Also include the average reward result over 1000 steps."

"From Task 1.1P, we have successfully managed to model the same environment as from before and this is shown below:"

Part 1: Initializing the Gym environment

```
!pip install mdp
```

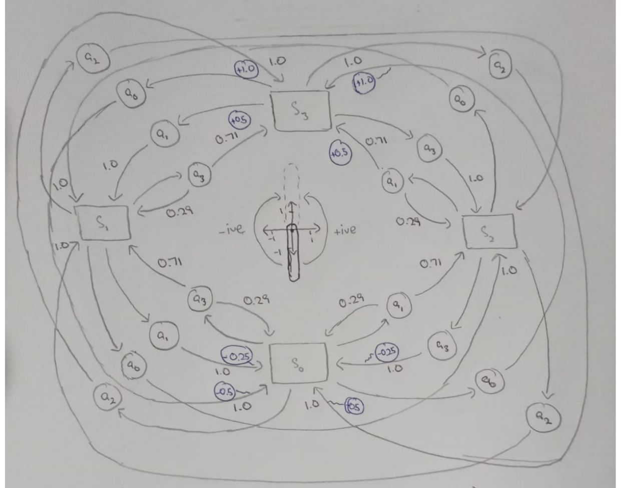
```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting mdp
  Downloading MDP-3.6-py2.py3-none-any.whl (454 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 454.9/454.9 KB 8.7 MB/s eta 0:00:00
Requirement already satisfied: future in /usr/local/lib/python3.9/dist-packages (from mdp) (0.18.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from mdp) (1.22.4)
Installing collected packages: mdp
Successfully installed mdp-3.6
```

```
from google.colab import drive
import sys
drive.mount('/content/drive')
sys.path.insert(0,'/content/drive/MyDrive/Colab Notebooks/')
```

```
    Mounted at /content/drive
```

"The following Gym Environment consists of an inverted pendulum, based on the problem presented in Task 1.1P. We have scaled this problem into an exact policy iteration implementation as shown in the below diagram. Based of the diagram, we have mentioned the tranistion probabilities as well as the rewards. "

Pendulum -v0 (MDP Implementation):



- Transition Probabilities : (Mentioned in Diagram!)
- Rewards : (Mentioned in Diagram!)
- Actions :
  - a_0 : 2 Nm ; Counter-Clockwise (+ive)
  - a_1 : 1 Nm ; +ive
  - a_2 : 2 Nm ; Clockwise (-ive)
  - a_3 : 1 Nm ; -ive

States :
- S_0 : Bottom Position ; (0,-1)
- S_1 : Left " ; (-1,0)
- S_2 : Right " ; (+1,0)
- S_3 : Top " ; (0,+1)

```
transition_probs = {
    's0': {
        'a0': {'s3': 1},
        'a1': {'s2': 0.71, 's0': 0.29},
        'a2': {'s3': 1},
        'a3': {'s1': 0.71, 's0': 0.29}
    },
    's1': {
        'a0': {'s2': 1},
        'a1': {'s0': 1},
        'a2': {'s2': 1},
        'a3': {'s3': 0.71, 's1': 0.29}
    },
    's2': {
        'a0': {'s1': 1},
        'a1': {'s0': 0.71, 's1': 0.29},
        'a2': {'s1': 1},
        'a3': {'s1': 1}
    },
    's3': {
        'a0': {'s0': 1},
        'a1': {'s1': 1},
        'a2': {'s0': 1},
        'a3': {'s2': 1}
    }
}
rewards = {
    's0': {'a0': {'s3': +1}},
    's0': {'a2': {'s3': +1}},
    's1': {'a1': {'s0': -0.25}},
    's1': {'a3': {'s3': +0.5}},
    's2': {'a1': {'s3': +0.5}},
    's2': {'a3': {'s0': -0.25}},
    's3': {'a0': {'s0': -0.5}},
    's3': {'a2': {'s0': -0.5}},
}
```

"Now we will import the MDP Package that was already provided to us for this task..."

```
import mdp
from  W3_mdp import MDP
mdp = MDP(transition_probs, rewards, initial_state='s0')
```

"Now this is the part where we test our environment's working."

```
print('initial state =', mdp.reset())
next_state, reward, done, info = mdp.step('a1')
print('next_state = %s, reward = %s, done = %s' % (next_state, reward, done))

    initial state = s0
    next_state = s0, reward = 0.0, done = False
    /usr/local/lib/python3.9/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
      and should_run_async(code)
```

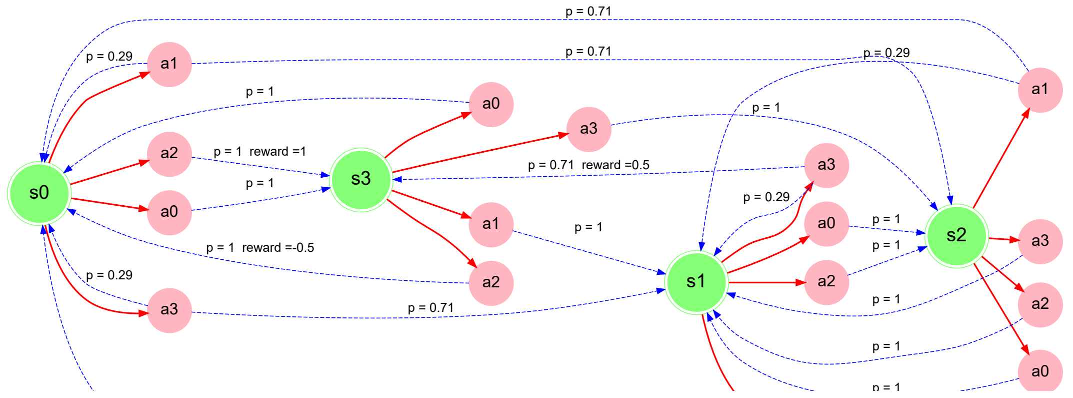
```
print("mdp.get_all_states =", mdp.get_all_states())
print("mdp.get_possible_actions('s1') = ", mdp.get_possible_actions('s1'))
print("mdp.get_next_states('s1', 'a0') = ", mdp.get_next_states('s1', 'a0'))
print("mdp.get_reward('s1', 'a0', 's0') = ", mdp.get_reward('s1', 'a0', 's0'))
print("mdp.get_transition_prob('s2', 'a0', 's3') = ", mdp.get_transition_prob('s2', 'a0', 's3'))

    mdp.get_all_states = ('s0', 's1', 's2', 's3')
    mdp.get_possible_actions('s1') =  ('a0', 'a1', 'a2', 'a3')
    mdp.get_next_states('s1', 'a0') =  {'s2': 1}
    mdp.get_reward('s1', 'a0', 's0') =  0.0
    mdp.get_transition_prob('s2', 'a0', 's3') =  0.0
```

Part 2: Visualizing MDPs

"Now let's visualize our MDP in this section and observe whether the model implemented is the same as the above picture."

```
from W3_mdp import has_graphviz
from IPython.display import display
print("Graphviz available:", has_graphviz)

    Graphviz available: True
```

```
if has_graphviz:
    from W3_mdp import plot_graph, plot_graph_with_state_values, plot_graph_optimal_strategy_and_state_values
    display(plot_graph(mdp, graph_size='30,30'))
```



"As you can see, our visualization of the model has been successful. Proceeding to the next part of our expiriment."

Part 3: Value Iteration

From Practical 3, the pseudo-code for Value Iteration is as follows:

```
Step 1: Initialize $V^{(0)}(s) = 0$, for all $s$
Step 2: For $i = 0, 1, 2, \dots$
Step 3:     $V_{(i+1)}(s) = \max_a \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')]$, for all $s$
```

Step 1: Function to compute the state-action value function $Q(\pi)$, defined as follows

$$Q_i(s,a) = \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')]$$

Step 2: Using $Q(s,a)$ we can now define the "next" V(s) for value iteration.

$$V_{i+1}(s) = \max_a \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')] = \max_a Q_i(s,a)$$

Step 3: Using those $V_{i+1}(s)$ to find optimal actions in each state

$$\pi^*(s) = argmax_a \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')] = argmax_a Q_i(s,a)$$

The only difference vs V(s) is that here we take not max but argmax: find action such with maximum Q(s,a).

Step 1: Function to compute the state-action value function

```
def get_action_value(mdp, state_values, present_state, action, gamma):

    # Initialise Q
    Q = 0
    for s in mdp.get_all_states():
      # Compute Q using the equation above
      Q = Q + mdp.get_transition_prob(present_state, action, s)*(mdp.get_reward(present_state, action, s) + gamma*state_values[s])
    return Q
```

```
/usr/local/lib/python3.9/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
#Testing the function for Step 1
import numpy as np
test_Vs = {s: i for i, s in enumerate(sorted(mdp.get_all_states()))}
print(test_Vs)
print(get_action_value(mdp, test_Vs, 's2', 'a0', 0.75))
#assert np.isclose(get_action_value(mdp, test_Vs, 's2', 'a0', 0.9), 2.34)
print(get_action_value(mdp, test_Vs, 's3', 'a0', 0.75))
#assert np.isclose(get_action_value(mdp, test_Vs, 's3', 'a0', 0.9), 0.9)
```

```
{'s0': 0, 's1': 1, 's2': 2, 's3': 3}
0.75
0.0
```

Step 2: Using the state-action value (from Step 1) for the next state-value

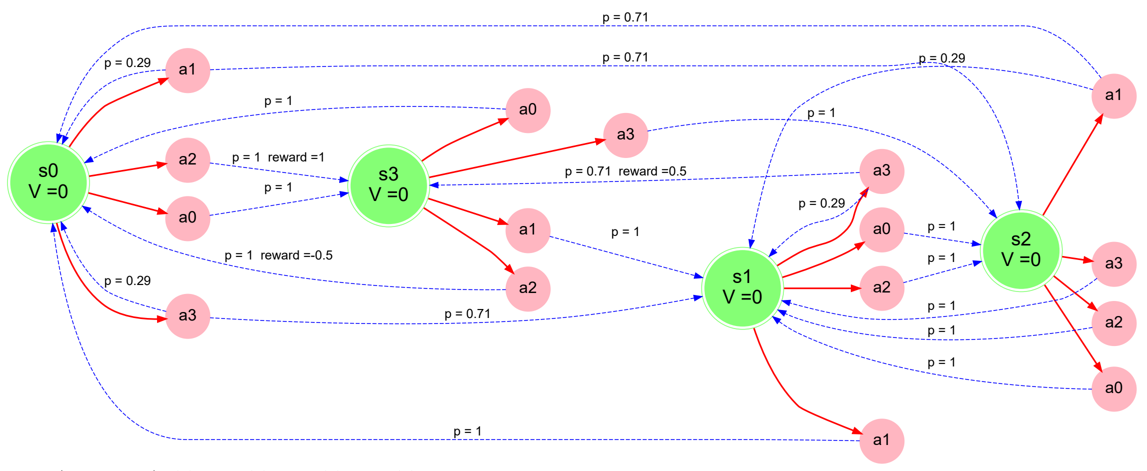
```
def get_new_state_value(mdp, state_values, state, gamma):
    import numpy as np

    # Computes next V(s) as in formula above. Do not change state_values in process.
    if mdp.is_terminal(state):
        return 0

    # Initialise the dict
    A = [a for a in mdp.get_possible_actions(state)]
    v = np.zeros(len(mdp.get_possible_actions(state)))
    i = 0

    # Compute all possible options
    for a in mdp.get_possible_actions(state):
      v[i] = get_action_value(mdp, state_values, state, a, gamma)
      A[i] = a
      i = i+1

    # Recover V(s) and π*(s) as per the formula above
    V = {A[np.argmax(v)]:v[np.argmax(v)]}

    return V
```

```
test_Vs_copy = dict(test_Vs)
V = get_new_state_value(mdp, test_Vs, 's0', 0.9)
print(test_Vs)
a = list(V)[0]
v = V[a]
print(a, v)
# assert np.isclose(v, 1.4)
#assert test_Vs == test_Vs_copy, "Please do not change state_values in get_new_state_value"
```

```
{'s0': 0, 's1': 1, 's2': 2, 's3': 3}
a2 3.7
```

```
# parameters
gamma = 0.75           # discount for MDP
num_iter = 100         # maximum iterations, excluding initialization
# stop VI if new values are this close to old values (or closer)
min_difference = 0.02

# initialize V(s)
state_values = {s: 0 for s in mdp.get_all_states()}

if has_graphviz:
    display(plot_graph_with_state_values(mdp, state_values))

for i in range(num_iter):

    # Compute new state values using the functions defined above.
    # It must be a dict {state : float V_new(state)}
    new_state_values = {}
    for s in mdp.get_all_states():
      nsv = get_new_state_value(mdp, state_values, s, gamma)
      a = list(nsv)[0]
      v = nsv[a]
      new_state_values[s] = v

    #assert isinstance(new_state_values, dict)

    # Compute difference
    diff = max(abs(new_state_values[s] - state_values[s]) for s in mdp.get_all_states())
    print("iter %4i   |   diff: %6.5f   |   " % (i, diff), end="")
    print('   '.join("V(%s) = %.3f" % (s, v) for s, v in state_values.items()))
    state_values = new_state_values

    if diff < min_difference:
        print("Terminated")
        break
```

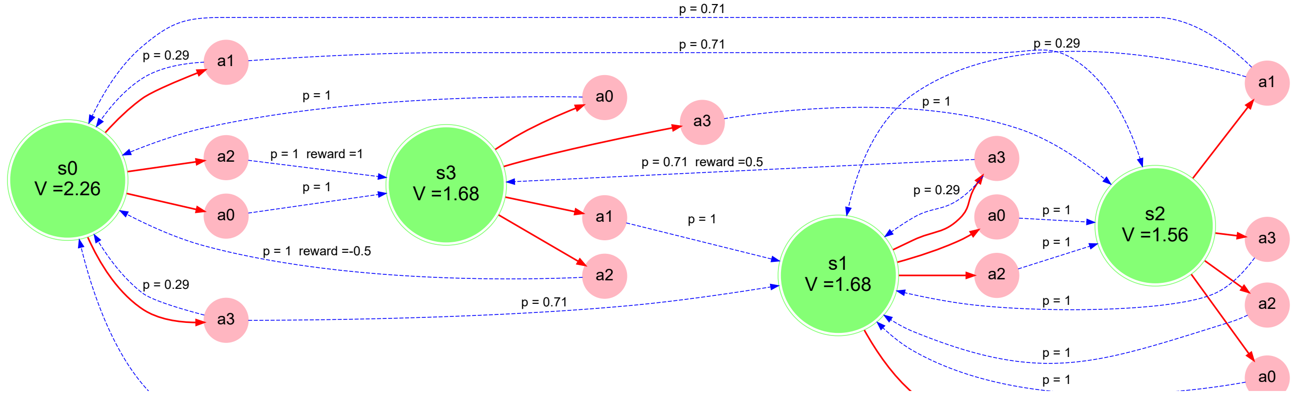
Remove Ads
TOTAL Adblock



```
iter    0   |   diff: 1.00000   |   V(s0) = 0.000   V(s1) = 0.000   V(s2) = 0.000   V(s3) = 0.000
iter    1   |   diff: 0.75000   |   V(s0) = 1.000   V(s1) = 0.355   V(s2) = 0.000   V(s3) = 0.000
iter    2   |   diff: 0.56250   |   V(s0) = 1.000   V(s1) = 0.750   V(s2) = 0.610   V(s3) = 0.750
iter    3   |   diff: 0.42188   |   V(s0) = 1.562   V(s1) = 0.917   V(s2) = 0.696   V(s3) = 0.750
iter    4   |   diff: 0.31641   |   V(s0) = 1.562   V(s1) = 1.172   V(s2) = 1.032   V(s3) = 1.172
iter    5   |   diff: 0.23730   |   V(s0) = 1.879   V(s1) = 1.234   V(s2) = 1.087   V(s3) = 1.172
```

```
if has_graphviz:
    display(plot_graph_with_state_values(mdp, state_values))
```

```
/usr/local/lib/python3.9/dist-packages/ipykernel/ipykernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

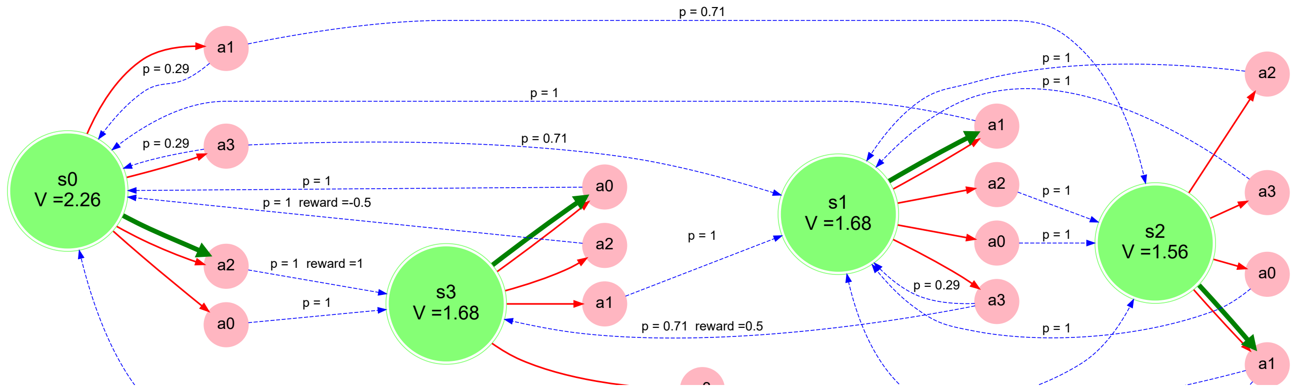


```
print("Final state values:", state_values)

#assert abs(state_values['s0'] - 3.781) < 0.01
#assert abs(state_values['s1'] - 7.294) < 0.01
#assert abs(state_values['s2'] - 4.202) < 0.01
```

```
Final state values: {'s0': 2.2628054954111576, 's1': 1.6837406605482101, 's2': 1.5616694626584648, 's3': 1.6837406605482101}
/usr/local/lib/python3.9/dist-packages/ipykernel/ipykernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Step 3: Using the next states (from Step 2) to find the optimal actions

```
def get_optimal_action(mdp, state_values, state, gamma):
    # Finds optimal action using formula above.
    if mdp.is_terminal(state):
        return None

    nsv = get_new_state_value(mdp, state_values, state, gamma)
    a = list(nsv)[0]

    return a


if has_graphviz:
    display(plot_graph_optimal_strategy_and_state_values(mdp, state_values, get_action_value))
```



Measure agent's average reward

```
s = mdp.reset()
rewards = []
gamma=0.75

for _ in range(1000):
    s, r, done, _ = mdp.step(get_optimal_action(mdp, state_values, s, gamma))
    rewards.append(r)

print("average reward: ", np.mean(rewards))

assert(0.40 < np.mean(rewards) < 0.55)
```

```
/usr/local/lib/python3.9/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
average reward:  0.5
```

✓ 0s completed at 8:25 AM