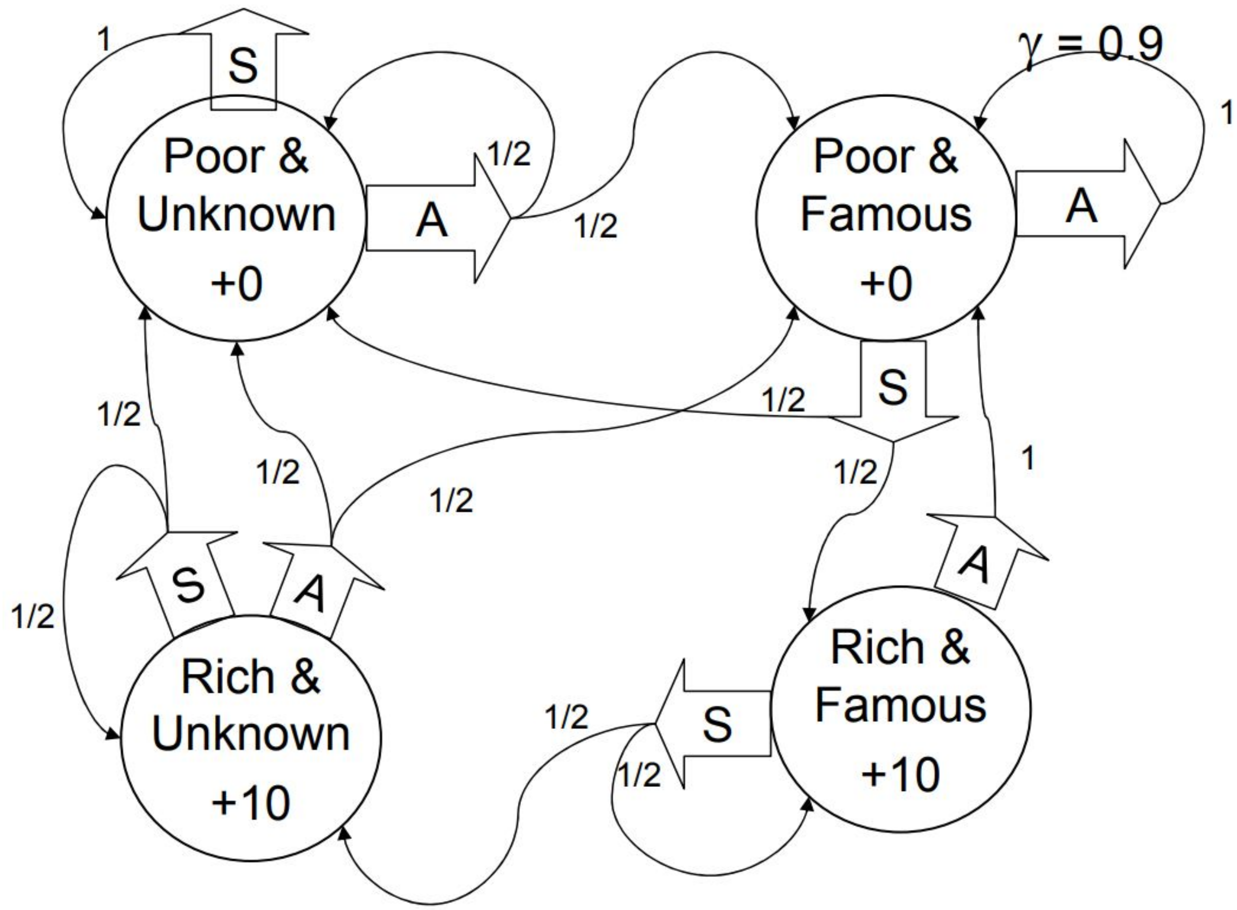

Markov Decision Process

Value Iteration

23.01.2020

Nivedita Rufus
2019702002

INTRODUCTION

An MDP is a Markov Reward Process with decisions, it's an environment in which all states are Markov. This is what we want to solve. An MDP is a tuple (S, A, P, R, γ), where S is our state space, A is a finite set of actions, P is the state transition probability function,

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

R is the reward function,

$$\mathcal{R}_{s}^{a} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$

and γ is a discount factor $\gamma \in [0, 1]$.

Value Function

Value of a state **s** under a certian policy **π**, denoted by $\mathbf{V_{\pi}(s)}$, is the value of the expected return when starting from state **s** and following **π** thereafter

Value Iteration

Here we compute the state-value function **Vπ** for a given policy **π**. It is possible to solve **N** Linear Equations in **N** variables, where **N** is the number of states in the MDP. In the given problem **N = 4** (PU,PF,RU,RF). The state values can be calculated by repeatedly applying Bellman's Expectation operation to an initially random state-value vector **V** , until convergence. But running till infinity is not computationally feasible, so we stop when the change in two consecutive iterations **||Vk+1 − Vk||∞** is within some acceptable error **(θ)**. In the problem we use convergence error to be 0.

Code:

```
def do_value_iter(Rewards, gamma, pss, curr_state):
      return (Rewards + gamma*(pss@curr_state))
```

Algorithm 1: Iterative Policy Evaluation

Function `PolicyEvaluation(`π, MDP`)`**:**
 p $\leftarrow$ MDP.transition_function
 r $\leftarrow$ MDP.reward_function
 $\theta \leftarrow$ The minimum value of L_∞ norm between two consecutive iterations
 $V \leftarrow$ Random vector $\in \Re^N$ where N is the number of state in the MDP
 do
  $\Delta = 0$
  forall $s \in S$ **do**
   $V_{old} \leftarrow V(s)$
   $V(s) \leftarrow \sum_{a \in A(s)} \left\{ \pi(a|s)(r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_{\pi_k}(s')) \right\}$
   $\Delta \leftarrow \max(\Delta, |V(s) - V_{old}|)$
  end
 while $\Delta > \theta$;
return V

Image courtesy: google.com

Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function **Vπ** for a policy **π**. One way to answer this question is to consider selecting **a** in **s** and thereafter following the existing policy, **π**. We try this for each possible a improve the policy. For each **s ∈ S**, choose:

$$\arg\max_k \left[r_i + \gamma \sum_j P_{ij}^k J^*(S_j) \right]$$

```
def get_P(new_state_a, new_state_s, P):
      for i in range(0,4):
            if(new_state_a[i] > new_state_s[i]):
                  P[i] = 'advertise'
            if (new_state_a[i] < new_state_s[i]):
                  P[i] = 'save'
      return P
```

Solving the MDP

For the given MDP we have

The state set is:

S = {PU, PF, RU, RF}

The action set is:

A = {advertise, save}

The probability transition matrix for action “advertise” is:

pss_a =[0.5, 0.5, 0, 0]

[0, 1, 0, 0]

[0.5, 0.5, 0, 0]

[0, 1, 0, 0]

The probability transition matrix for action “save” is:

Pss_s =[1, 0, 0, 0]

[0.5, 0, 0, 0.5]

[0.5, 0, 0.5, 0]

[0, 0, 0.5, 0.5]

Steps:

- We do Value Iteration using the following recursive equation
- **V(s) = R + 0.9*(PSS*V(s))** for all states and all actions.
- Once the updates become really small. It means that a policy has been stabilized and that is our optimal policy.
- The core part of the code which computes the value iteration can be vectorised for all the states in the following manner. Since there are only 2 states this can be written as:

```
while(True):

            V_curr = V_new

            V_new_a = do_value_iter(Rewards, gamma, PSS_a, V_curr)
            V_new_s = do_value_iter(Rewards, gamma, PSS_s, V_curr)

            V_new = np.maximum(V_new_a, V_new_s)
            Policy = get_policy(V_new_a, V_new_s, Policy)

            print('The current values for the states are', V_curr)
            print('The current policy is', Policy)

            if(np.array_equal(V_curr, V_new)):
                  break
```

Given the

The state set is:

S = {PU, PF, RU, RF}

And the action set is:

A = {advertise, save}

The optimal policy is

['advertise', 'save', 'save', 'save']

The converged Values for each of the states are

[31.58510431, 38.60401638, 44.02417625, 54.20159875]