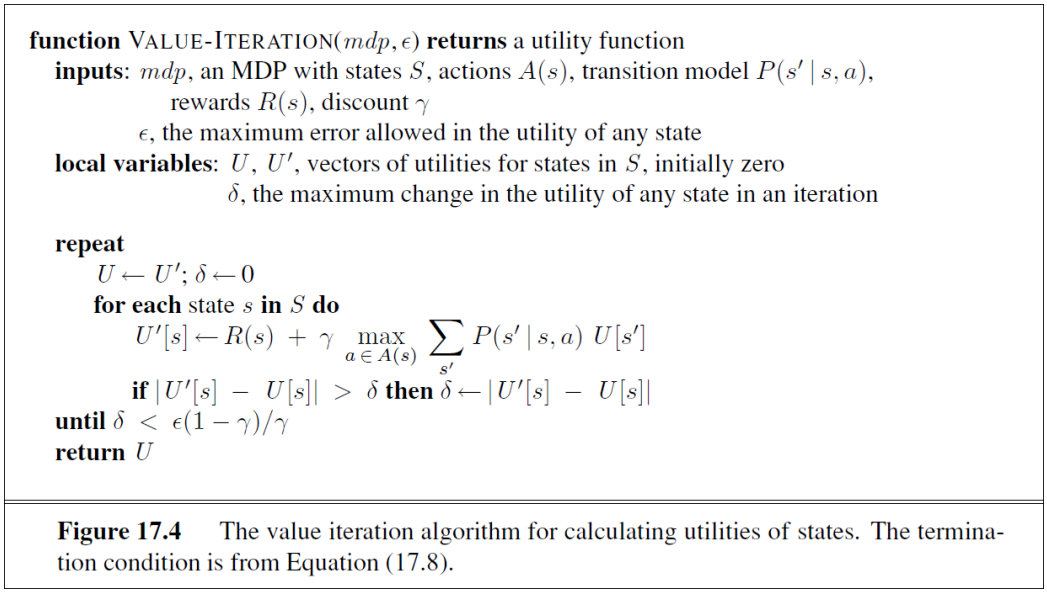
ITCS 3153: Introduction to Artificial Intelligence

Value Iteration for a Markov Decision Process

Implement the Value Iteration algorithm shown in the textbook on Figure 17.4. Use the 4x3 grid world shown in Figure 17.1 as an example problem.

Here is a screenshot of the pseudocode:



function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

Specifically, you need to implement a function called ‘valueIteration’ that takes in the following parameters:

A set of states
A set of actions
A transition model matrix
A set of rewards
A discount factor

You may use different parameters if you need/want to. The function should return an array of utility values that represents the utility of each state. A function header may look like the following:

def valueIteration(S, A, P, R_states, discount, terminal_index_reward_pairs):

S - list of states
A - list of actions
P - transition matrix
R_states - rewards for each state
discount - discount factor
terminal_index_reward_pairs is a list that specifies the rewards for the terminal states. For instance, in the grid world below position (4,3) is a terminal state and it has a reward of 1. The utility of this state should be set to the reward, and that utility will not change during the value iteration algorithm. *You can simply hardcode the terminal states until the algorithm works, and then put this back in later.*

When the non-terminal state rewards are -0.04 and the discount factor is 1, the utilities for each state are shown in the following figure:

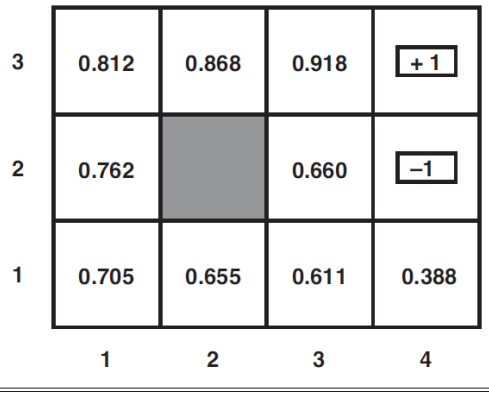


Figure 17.3 The utilities of the states in the 4×3 world, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

Use this to check your algorithm. Given the same reward and discount factor, your function should output utilities that are nearly identical to the ones in the figure above. You should use the numpy library for this assignment instead of lists whenever you are working with the utilities.
You will be given a transition model matrix to use for this assignment in the vi_util.py file on Canvas. This transition model will be a 4x11x11 numpy array of floats. Pass this to your valueIteration function.

Recommended steps: (Updated Nov. 16th, 2021)

Group:

Initialization:

Create a class for a State that has members ‘x’ and ‘y’.
Create a 1D list of State objects for the environment. Don’t forget to **exclude** the state for position (2,2) because that position is an obstacle. The states should be in **row-major order**, so [(1,1), (1,2), (1,3), etc.]
Create a list of actions. This can simply be A=[‘u’, ‘r’, ‘d’, ‘l’]. You just need four elements. The order of actions should be [up, right, down, left] in order to properly index the transition model (step 5).
Create a numpy array containing the rewards for each state. This can also be a 1D array that’s in row-major order. Make sure your terminal states have the correct rewards.
Create your transition model matrix **(this is given to you in vi_util.py).**
Set up the valueIteration function header and return statement (this can return nothing meaningful for now).

Implementing valueIteration:

Create a numpy array called U to hold a utility value for each state.
Set the utilities for the terminal states in U.
Create a numpy array called U_prime to hold updated utility values for each non-terminal state.

Work on function: getExpectedUtility()

Needed: index of action, index of state, transition model, utility vector/list/array (the utility value for each state), set of states

Individual

Set up the Bellman equation for a single state and action.

This will require several things to do.

Set U' for a single state.
Calculate U' for all states.
Set up the main loop in valueIteration

Calculate delta, which is the maximum difference between U' and U in an iteration.

Terminate loop once greatest change is smaller than threshold

The value iteration algorithm returns a set of optimal utility values, but it doesn't return the policy. The policy is obtained by determining the best action at each state, which will be the action with the highest expected utility.

After you have obtained a set of utility values, you should call the function getPolicyForGrid to obtain the policy for your utilities. Then, call printPolicyForGrid to print out the policy. **These two functions are provided for you in vi_util.py file.**

Run your code with the following [discount, reward] pairs and report the output for each:

[1.0, -0.04], [1.0, -0.25], [1.0, -0.01],
[0.5, -0.04], [0.5, -0.25], [0.5, -0.01]

Your output for one [discount, reward] pair should look like the following:

Discount = 1.0 Reward = -0.04
Utilities:
[[0.70530822]
 [0.65530822]
 [0.61141553]
 [0.38792491]
 [0.76155822]
 [0.66027397]
 [-1.   ]
 [0.81155822]
 [0.86780822]
 [0.91780822]
 [1.   ]]

Policy: ['u', 'l', 'l', 'l', 'u', 'u', 'T', 'r', 'r', 'r', 'T']
['r', 'r', 'r', 'T']
['u', '0', 'u', 'T']
['u', 'l', 'l', 'l']
Submit your code and output (in a text file) to Canvas.