ASSIGNMENT 2

SHUBHAM LOHIYA
18D100020

22nd October 2020

—

CS 747

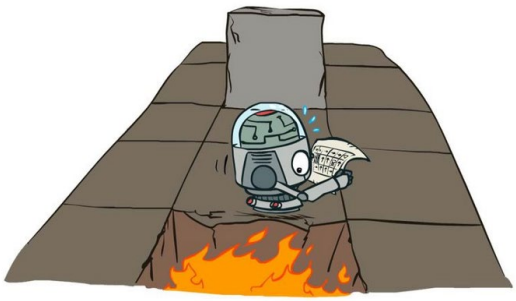
—

Shivaram Kalyanakrishnan

INTRODUCTION

This document is a report for the second assignment of the CS 747 – Fundamentals of Intelligent and Learning Agents course offered at IIT Bombay.
This assignment is based on Markov's Decision Processes (MDP) and explores various algorithms to arrive at the optimal policy and aims to choose the one which works the fastest

Markov Decision Processes



TASK 1

SOLVING MDPs

Value Iteration:

Implemented the Value Iteration algorithm like the following pseudocode:

$V_0 \leftarrow$ Arbitrary, element-wise bounded, n-length vector.
$t \leftarrow 0$.
Repeat:
  For $s \in S$:
    $V_{t+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V_t(s')\right)$.
    $t \leftarrow t + 1$.
Until $V_t \approx V_{t-1}$ (up to machine precision).

The state value array was initialized with zero before beginning iteration. For termination, it was checked if all elements of the value array from the previous iteration and the corresponding elements from the new improved vector are close within a certain tolerance: `np.allclose(V, V_old, rtol=0, atol=1e-11)`

Howard Policy Iteration:

Implemented HPI by estimating the value function on the current policy by solving the system of linear equations. Then, using this value function, policy is improved by picking the best action corresponding to any state. Here is the idea of the code used.

```
while (1):
    T_pi = mdp["T"][np.arange(mdp["ns"]), pi_old]
    R_pi = mdp["R"][np.arange(mdp["ns"]), pi_old]
    V = np.squeeze(np.linalg.inv(np.eye(mdp["ns"]) - mdp["gamma"] *
T_pi) @ np.sum(T_pi * R_pi, axis=-1, keepdims=True))
    pi = np.argmax(np.sum(mdp["T"] * (mdp["R"] + mdp["gamma"] * V),
axis=-1), axis=-1)
    if np.array_equal(pi, pi_old):
        break
    pi_old = pi
```

The policy was initialized randomly as follows:
`pi = np.random.randint(low=0, high=mdp["na"], size=mdp["ns"])`
We terminate when there are no improvable states, i.e. pi same as p_old

Linear Programming:

This algorithm was implemented by directly solving the linear programming problem with an objective function, n variables (the n state values) and n*k constraints where n is the number of states and k is the number of actions. The LP problem was formulated as follows:

$$\text{Maximise} \left(-\sum_{s \in S} V(s) \right)$$

subject to

$$V(s) \geq \sum_{s' \in S} T(s, a, s')\{R(s, a, s') + \gamma V(s')\}, \forall s \in S, a \in A.$$

Implementation using PULP solver was as follows:

```
prob = p.LpProblem('MDP', p.LpMinimize)
V = np.array(list(p.LpVariable.dicts("V", [i for i in range(mdp["ns"])]).values()))
prob += p.lpSum(V)  # Objective function
# Constraints
for s in range(mdp["ns"]):
    for a in range(mdp["na"]):
        prob += V[s] >= p.lpSum(mdp["T"][s, a] * (mdp["R"][s, a] + mdp["gamma"] * V))

prob.solve(p.apis.PULP_CBC_CMD(msg=0))
V = np.array(list(map(p.value, V)))
policy = np.argmax(np.sum(mdp["T"] * (mdp["R"] + mdp["gamma"] * V), axis=-1), axis=-1)
```

Observations:

- Value Iteration performs best on larger size MDPs.
- Both HPI and LP are faster than VI on smaller MDPs but things get slower for larger MDPs because for HPI, cost of inverting our big 2D matrix to solve for V increases a lot $O(n^3)$ where nxn is the size of the matrix.
  For larger MDPs (especially mazes formulated as MDPs with lot's of states), LP becomes slow mainly because the process of adding so many constraints to the solver takes the most time.
- These problems stem from the use of sparse 3-D arrays for representing transition probabilities and transition rewards as most of the entries would be zero. This can be solved by using dictionaries to formulate the given MDP problem. I have included the code for this (commented out) in my submission as I couldn't complete the algorithms for this type of formulation before the deadline. This representation is bound to give us a speedup in all algorithms for large MDPs.

TASK 2

MAZE FORMULATION AS AN MDP

Encoder

I start of my reading the given maze and then making an array giving every valid tile a state name from 0 to numStates-1. For example, consider the 10x10 grid:

```
[[1 1 1 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 2 0 0 1]
 [1 0 1 1 1 1 1 0 1 0 1]
 [1 0 0 0 0 0 0 0 1 0 1]
 [1 0 1 0 1 1 1 1 1 0 1]
 [1 0 1 0 0 0 1 0 0 0 1]
 [1 0 1 0 1 1 1 1 1 0 1]
 [1 0 1 0 1 0 0 3 1 0 1]
 [1 0 1 1 1 0 1 1 1 0 1]
 [1 0 0 0 0 0 0 0 1 0 1]
 [1 1 1 1 1 1 1 1 1 1 1]]
```

Given Maze

```
[[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
 [-1  0  1  2  3  4  5  6  7  8 -1]
 [-1  9 -1 -1 -1 -1 -1 10 -1 11 -1]
 [-1 12 13 14 15 16 17 18 -1 19 -1]
 [-1 20 -1 21 -1 -1 -1 -1 -1 22 -1]
 [-1 23 -1 24 25 26 -1 27 28 29 -1]
 [-1 30 -1 31 -1 -1 -1 -1 -1 32 -1]
 [-1 33 -1 34 -1 35 36 37 -1 38 -1]
 [-1 39 -1 -1 -1 40 -1 -1 -1 41 -1]
 [-1 42 43 44 45 46 47 48 -1 49 -1]
 [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]]
```

Maze with State names given to valid tiles (!=1)

Thus, for the given 50 valid tiles, we have state names from 0-49. While giving the state names, if the value encountered was 2, that state name has been flagged as a start state and if the value function is 3, it has been flagged as one of the end states. All the initial data has been accumulated.
Now, I loop over the maze and for every valid tile which is not an end state, I get the possible transitions for that that state, i.e. on taking an action from that state, which state will I end up this. This is deterministic, as in, if I go north and the state to my north is the valid tile, that become my next state and I get a state, action, next_state pair. If it's invalid, the state does not change and state==next_state. The actions are labelled 0, 1 , 2 or 3 for N, S, E, and W repectively. Since any action deterministically leads to one next state, all transition probabilities have been set to 1.
Reward at each time step is -1 as this incentivizes our agent to get to an end state fast so as to maximize reward or minimize negative reward accumulation. The mdptype is episodic as we have been given end states. I gave taken discount factor (gamma) as 1 to give slightly more importance to sooner rewards instead of giving all rewards the same weight as this led to faster convergence.

Implementation: (Check source code for more details)

```
for i in range(1, a - 1):
    for j in range(1, b - 1):
        if maze[i, j] != 1:
            if maze[i, j] == 3:
                continue
            mdp += get_transitions(i, j, maze, state_names)
```

```
def get_transitions(i, j, maze, state_names):
    s = []
    a_values = []

    # North
    a_values.append(0)
    if maze[i-1, j]!=1:
        s.append(state_names[i-1, j])
    else:
        s.append(state_names[i, j])

    # South
    a_values.append(1)
    if maze[i+1, j]!=1:
        s.append(state_names[i+1, j])
    else:
        s.append(state_names[i, j])

    # East
    a_values.append(2)
    if maze[i, j+1]!=1:
        s.append(state_names[i, j+1])
        # s_values.append(maze[i, j+1])
    else:
        s.append(state_names[i, j])

    # West
    a_values.append(3)
    if maze[i, j-1]!=1:
        s.append(state_names[i, j-1])
    else:
        s.append(state_names[i, j])

    transitions = ""
    for t in range(4):
        transitions += f"transition {state_names[i,j]} {a_values[t]} {s[t]} -1.0 1.0\n"
    return transitions
```

The resulting mdpfile was sent to planner.py to generate optimal policy. Then the optimal policy was used by the decoder to simulate a path starting from the start state to an end state by following the optimal policy:

```
curr = start
path = ""
while curr not in end:
    path += get_action(pi[curr]) + " "
    curr = next_state(curr, pi[curr], state_names)
print(path.strip())
```

Observations:

- Value Iteration works best, especially for large maze. HPI is slower due to cost of inversion of such large matrices. LP is the slowest due to the large time consumed for adding constraints (5 mins later also it was adding constraints when I gave a keyboard interrupt)
- Gamma values slightly less than one gives a speedup for convergence.
  I took gamma = 0.9
- Here, the code can be made much more efficient and faster by using dictionaries instead of such sparse numpy arrays. This is because any state has transitions to at most 4 states but using numpy arrays, it uses values functions of all other states while evaluating bellman equations or while using the bellman contraction operator for iteration, albeit with transition probability 0 to most of these states in the mdp. Thus a lot of computational resources are wasted in working with so many zeros which don't contribute to the solution.
  The starter code for dictionaries has been included as commented out code in the source code. It's implementation is incomplete for algorithms due to lack of time before the deadline.
- Implementation with 3D numpy arrays is easier and also gives decent results.