CS747 Programming Assignment 2 Report

Tezan Sahu — 170100035

October 22, 2020

---

1 Task 1

In the `planner.py` code, we define an `MDPPlanning` class, which is initialized with the path to the mdp text file. Data from this file is read into the class & used to initialize the state of the MDP.

The lines `transition s a s' reward probability` are used to initialize the transition probability tensor `T` and reward tensor `R`, each of size $n(S) \times n(A) \times n(S)$, where S & A are the set of states and actions respectively.

Based on the value received as the `--algorithm`, the corresponding function is used to obtain V^* and π^* for the required MDP.

1.1 Value Iteration

- The `valueIteration(...)` function in `MDPPlanning` class implements the Value Iteration algorithm.
- We use a `threshold` parameter to control the convergence of the algorithm. For an MDP, we claim that this algorithm has converged during iteration t if the following condition holds:

  $$max(|V(s)^t - V(s)^{t-1}|) < threshold\ ;\ s \in S$$

  Here, we check that the maxiumum absolute difference of the value functions across all states, computed between 2 consecutive iterations should be less than the `threshold`. This means that the difference of value functions for all other states is between these 2 iterations is definitely smaller than the `threshold`.
- For our implementation, we set `threshold` $= 10^{-12}$

1.2 Howard's Policy Iteration

- The `howardsPolicyIteration(...)` function in `MDPPlanning` class implements the Howard's Policy Iteration algorithm.
- We divide each iteration of the algorithm into 2 stages:

- The internal function `__policyEvaluation(...)` implements the Policy Evaluation stage. *[This has been implemented using steps similar to value Iteration]*
- The internal function `__policyImprovement(...)` implements the Policy Improvement stage. *[We improve **all** improvable states by uniformly sampling an action available (using `numpy.random.choice(...)`) within the set of improving actions for the corresponding state.]*

- Similar to the Value Iteration implementation, we set `threshold` $= 10^{-12}$ to signal the convergence of each of the above 2 stages for every iteration.

1.3 Linear Programming

- We use the `PuLP` library to solve the MDPPlanning problem, posed as a linear programming problem, by trying to **minimize** the objective function given by:

$$\sum_{s \in S} V(s)$$

subject to constraints:

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

- We use the V^* to obtain Q^* and eventually π^*

2 Task 2

In the `encoder.py` code, we create a `MazeEncoder` class, initialized using the path to the maze grid text file in order to encode the Maze M as an MDP as follows:

- Maintain `num_states` $= length(M) \times breadth(M)$
- Maintain `num_actions` $= 4$ [for North (0), East (1), South (2) and West (3)].
- Maintain 2 $num\_states \times num\_actions \times num\_states$ dimensional tensors to store the transition probabilities `T` & rewards `R`.
- Each cell M_{ij} in the Maze M is denoted by s_n, where $n = (i \times length(M)) + j$.
- We ensure that there are no transitions fom the `end` state (marked with 3).
- For all cells NOT marked as 1, i.e., for all cells in which the agent can exist, we do the following:
  - If the value of the cell is 2, we set it as the `start` state
  - For each of the 4 actions ($a \in 0, 1, 2, 3$), we set the corresponding `T[s, a, s']` $= 1$ (because the actions lead to deterministic transitions)

- Accordingly, we set `R[s, a, s']` $= -1$ (since we want to find the shortest path from `start` to `end`, we penalize every step taken by the agent. This would allow the agent to choose actions that will lead to the *least negative reward*, and hence, the shortest path)
- If `s'` is invalid (either it causes the agent to land on a cell labelled 1, or causes the agent to leave the boundaries of the maze), we set `s' = s` and still apply the same values of `T` and `R` as mentioned previously

In the `decoder.py` code, we create a `MazeDecoder` class, initialized using the path to the maze grid text file and the file containing the values of V^* and π^* for each state (generated by running `planner.py`) in order to decode the shortest path as follows:

- We first find the `start` and `ends` of the Maze M
- Using the values of $\pi^*(s)$, starting from $s =$ `start`, we traverse the states that the corresponding transitions lead to, while maintaining a string (`path`) of the actions taken, where the actions:
  - $0 \Rightarrow$ North (N)
  - $1 \Rightarrow$ East (E)
  - $2 \Rightarrow$ South (S)
  - $3 \Rightarrow$ West (W)
- We stop & break from this path traversal once we reach any of the states present in `ends`.
- the `path` string represents the decoded shortest path