Rushabh Kanadiya
180070026

Task 1

Howard Policy Iteration

- For choosing the Improving Action, simply choose the action that gives the best improvemt in terms of the Q function.
- For the Policy Evaluation Step, I tried solving for V using these methods: `numpy.linalg.pinv`, `numpy.linalg.solve`, `numpy.linalg.lstsq`, `Bellman Operator`, which are based on, respectively, Solving Using Matrix Inverse, Numpy's Linear Equation Solver, Least Square Solution if Inverse doesn't exist and application of *Bellman Operator* until convergence.
- Out of all these, Bellman Operator seems to be the most promising and is guaranteed to converge, given that Bellman Operator is a Contraction Mapping in the Value Function Space.

Linear Programming

- Initially Linear Programming seemed to be the slowest one, however when code was optimzed, it worked faster than the other two almost all the times.
- Optimzied code using this method

Task 2

MDP Design

- Initially, I encoded each cell as a separate state, thus having $rows * cols$ number of different states, the walls and the destination, would be encoded as end states, that is with no outward edge in the state diagram.
- As for the transition function, from each of the non-ending states, each action would lead to a state deterministically (because physically N, W, E and S can only take you a single step in that direction). For the actions that take you out of the maze area, I introduced one extra end-state state, so that all such invalid actions lead to that end-state.
- For the reward system, since we want the shortest path, I gave each valid action a reward of -1. Since there are many end states (destination and walls), and we want the MDP to chose a path to the destination, we will give a very high positive reward to the actions that lead to the destination and a similar negative reward to the invalid actions that lead to one of the end-states (walls and one other state introduced in the above point)
- However, this encoding with $rows * cols + 1$ states, would take up a lot of memory and would take a lot of time to run.
- Observing carefully, we can notice that all the end-states are very much the same in the sense that they have no outward edge, difference, if any, should only be in the Reward function. In this MDP, reward depends only on the current state and the action taken from the current state, because the next state is determined deterministically from the current state and the direction of the action, so these end states; don't even differ in terms of Reward function, because the reward of incoming edges doesn't depend on them. Not that it matters, but but for the walls even the reward on the incoming edges is the same. So merging these states perfectly makes sense. So I merged all the end-states to a single end-state, including the destination, walls and the extra-end state added as in point 2.
- This optimization significantly reduces memory requirements and execution time.
- After all the optimization, the following Reward Function suits our purpose: each valid action except those leading directly to the destination cell, would be rewarded -1(since we want it to find the shortest path). And the action directly leading to destination cell would be assigned a high positive reward($\sim 10^5$). Since we do not want the our planner get stuck at a wall or stop after taking an invalid action, any negative reward should do for an invalid action, however on a safer side we assign such action high negative rewards ($\sim -10^5$).
- As for the discount factor, we keep it 0.9, just to make sure it converges faster when using *Howard Policy Iteration*. However, as per my test runs, the discount factor of 1 works just as good with *Value Iteration* and *Linear Problem Solver*.

Algorithm

- For solving the MDP, Value Iteration does reasonably well, taking around 1.5min for `MazeVerifyOutput.py` provided.
- However, *Linear Problem Solver* outperformed both other methods, taking the least amount of time, after optimizing the code, under 20sec for `MazeVerifyOutput.py`.
- Howard Policy Iteration was the slowest, 5.5min even after a small tweaking the discount factor, which was required to make it converge faster.
- With the reward scheme chosen, these algorithms should choose the shortest path to any destination cell, given the grid size is reasonably small (basically the path to destination cell should be smaller than the large reward assigned on reaching it)