Computing Optimal Policies and Solving Mazes as Markov Decision Processes

CS747: Foundations of Intelligent and Learning Agents
Programming Assignment 02

Report submitted by: Aaron John Sabu

1 Introduction

The assignment deals with the implementation of different algorithms (Value Iteration, Linear Programming, Howard's Policy Iteration) for computing an optimal policy for a given Markov Decision Process. While the first part deals with the direct implementation of the algorithms, the second part involves the application of these algorithms in solving a real-life problem that computers can perform more optimally than humans - solving a maze. It has been observed that linear programming gives the fastest result for this problem in the conditions that the programs have been tested in.

2 Tasks

Task 1: Implementation of the algorithms for optimal policy

The basic algorithms - Value Iteration, Linear Programming, and Howard's Policy Iteration - have been implemented as *planner.py* running on Python 3.8. They have been evaluated for several MDPs:

Value Iteration

This algorithm initiates two vectors representing the value function, and based on iterating one with respect to the other, it terminates computation when the difference of the vectors is less than a tolerance value (1×10^{-12}). The optimal policy is also evaluated along with the computation of the value function.

Linear Programming

This algorithm directly implements linear programming using the *PuLP* module wherein the decision variables being the value function terms and the constraints on the decision variables are provided to the *PULP_CBC_CMD* solver. The optimal policy is evaluated from the optimal value function by comparison with the action value function. Due to the inaccuracy of the solution obtained from the solver, the value function is reevaluated using a generic value evaluation technique given the optimal policy.

Howard's Policy Iteration

The policy is initialized as a zero vector, and as long as possible (based on a *flag*), the policy is optimized by modifying the policy for a a state to the action with highest margin in action value function when compared to the state value function.

Task 2: Solving Mazes as Markov Decision Processes

In order to solve mazes, two additional scripts are developed: *encoder.py* to encode the maze into an MDP, and *decoder.py* to convert the policy into a path for the maze from a specified *start* point to an *end* point.

encoder.py

A *states* array stores all the valid maze cells as the states of the MDP and the index of the maze cell in *states* represents its state number. The *start* state number is specifically stored in another variable *start*, and the closest *end* state by Manhattan distance is stored in the *end* variable.
For each state, the transitions to neighboring maze cells is evaluated. Moving to the wall or the boundary causes the agents to return back to the state with a huge negative reward. In order to minimize the path length, a small negative reward is supplied for each transition to a valid state as well, apart from moving into the end state when a huge positive reward is supplied. At the end state, there is no transition to any of the neighboring maze cells. To decrease the path length even if the former method fails, the discount factor is set to 0.9. All evaluations are conducted assuming that the MDP is episodic.

decoder.py

A *states* array and the corresponding *start* and *end* states are generated similar to the method in *encoder.py*, and starting from the *start* state, the policy computed at *planner.py* is followed to generate the route until the present state reaches the *end* state.

3 Results

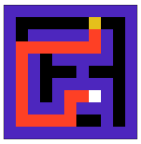
The two scripts provided for testing the planner and the maze solver are modified to display the time taken for evaluating the respective solutions. All results are positive and given below are the times taken for each algorithm in both cases:

Algorithm	Computation Time (s)	
	MDP Evaluation	Maze Solving
Value Iteration	24.36338	76.49915
Linear Programming	9.24100	39.59601
Howard's Policy Iteration	9.70000	820.71383

Table 1: Computation Time for Algorithms

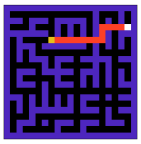
Also, given below are the unsolved vs. solved mazes for all 10 grids that have been provided for testing:

(a) Grid 10 - Unsolved

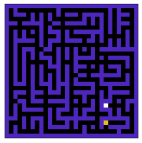


(b) Grid 10 - Solved

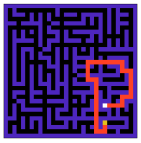
(c) Grid 20 - Unsolved



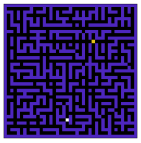
(d) Grid 20 - Solved



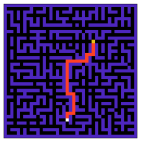
(e) Grid 30 - Unsolved



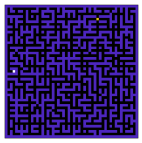
(f) Grid 30 - Solved



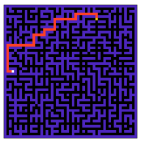
(g) Grid 40 - Unsolved



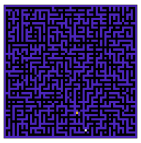
(h) Grid 40 - Solved



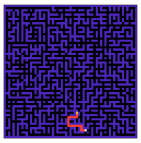
(i) Grid 50 - Unsolved



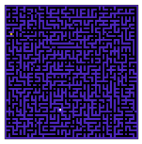
(j) Grid 50 - Solved



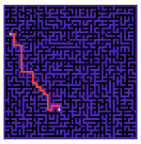
(k) Grid 60 - Unsolved



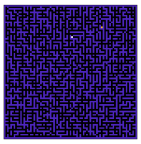
(l) Grid 60 - Solved



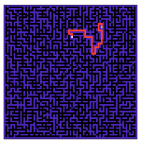
(m) Grid 70 - Unsolved



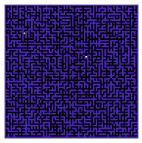
(n) Grid 70 - Solved



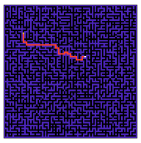
(o) Grid 80 - Unsolved



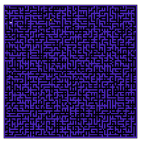
(p) Grid 80 - Solved



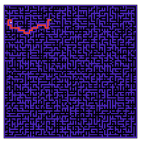
(q) Grid 90 - Unsolved



(r) Grid 90 - Solved



(s) Grid 100 - Unsolved



(t) Grid 100 - Solved

Figure 1: Unsolved vs. Solved Mazes