CS 747 - Foundations of Intelligent and Learning Agents
Assignment 2

Gagan Jain
180100043

October 23, 2020

Algorithms

This section is dedicated to making particular remarks about my implementation of all the algorithms. The three algorithms run over all the given planner instances takes around 5 seconds in total.

Value Iteration

In this algorithm, we iteratively compute the Value functions for all the states of the MDP using the recursive relation:

$$V(s) = max_a \Sigma_{s'} T(s,a,s^{'})(R(s,a,s^{'}) + \gamma V(s^{'})) \tag{1}$$

which necessarily converges to the optimal Value functions as discussed using the idea of Bellman Operator. We initialise the Value functions for all states with a uniform random distribution in $[0,1]$. It is important to note that irrespective of this initialisation, the value functions are still bound to converge to the same values. We vectorise the algorithm using numpy which significantly boosts the computations. Once we have the results converged (checked by ensuring that the value function at this and previous iteration do not differ by more than 10^{-9}), we compute the optimal action-value functions and the optimal policies using the following relations.

$$Q^*(s,a) = \Sigma_{s'} T(s,a,s^{'})(R(s,a,s^{'}) + \gamma V(s^{'})) \tag{2}$$

$$\pi^*(s) = argmax_a Q^*(s,a) \tag{3}$$

Howard Policy Iteration

The key idea behind policy iteration is to start with an arbitrary policy and keep updating it as long as there are improvable states. Howard policy iteration, specifically updates the policy for all the states for which there are improvable states. This is done by computing the action value function for each state and comparing it to the value function. In the implementation, in order to speed up convergence, we choose the policy with the greatest action-value function instead of any improvable policy. For the evaluation step, we need to solve the Bellman equations to find out the updated Value functions. In order to do this, we use the *PULP CBC CMD* solver. Note that the transition probabilities and rewards have been stored and looked up from a dictionary instead of the usual numpy array as we need to do element wise operations, which are slow when directly done on numpy. Dictionaries are faster for lookups as they are stored using hashing. The convergence condition used is same as the one used for Value Iteration, the value function at this and previous iteration should not differ by more than 10^{-9}.

Linear Programming

We formulate the problem of finding the optimal policy as a problem with linear constraints and use the standard pulp solver (*PULP CBC CMD*) to solve them. Again, we use dicts instead of numpy arrays for the same reason as mentioned in the previous section. To emphasize more on this, here are some numbers. When numpy arrays were used, the algorithm took 154 minutes to completely run on the checker file for Maze Output. When this was replaced by dicts, the time was reduced to 1.15 minutes! Once the optimal value functions have been computed, we find the optimal policy.

Maze to MDP

In order to formulate the maze as an MDP, we first see all the states where the agent can be present. To make use of the nice properties of the square grid, we keep a mapping from states to the grid locations. This is a significant optimization from taking the entire grid cells as states. We then mark the start and end points and define the four actions (N,W,E,S) that the agent can take. Now, in order to define the transitions, we take care of two things. In case we are at the end state, we're done and we do nothing. So, there are no transitions out of the end state. Now, if this is not the case, we check if the four neighbours of the state are valid states. We give a reward of -1 for every valid transition with a probability 1 for action in that direction. If the neighbour is not a state, we consider it to be a transition to the same state with a probability 1 and reward -1. The idea behind giving -1 reward is that since the mdp is episodic, we have end states. If we reach to the end states soon, we will accumulate less negative reward. If this is not the case, we will keep accumulating negative rewards. So, an optimal policy will try to minimise the penalties and move to the states closer to the end states. Now, clearly this is episodic and we set the discount factor to be 0.9 as it ensures fast convergence, as observed during experiments. In general, a smaller gamma will have a faster convergence but will have a smaller future lookup.

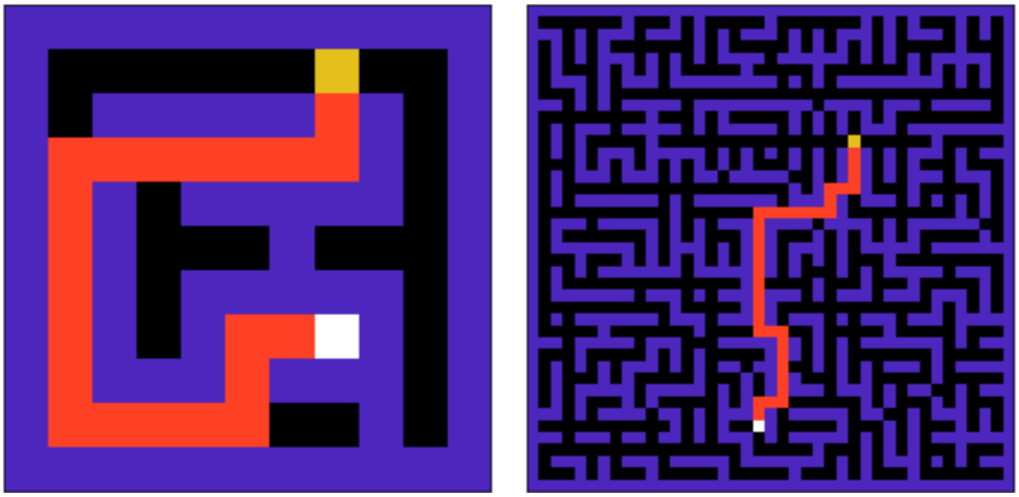
This MDP is passed to the planner and we choose the Linear Programming algorithm for solving the MDP, the reason being that it performs quickest among all the algorithms and gives the optimal solution as long as the linear algebra formulation of the problem is feasible. With a discount factor smaller than 1 and having the surety that there is always going to be at least one end state, we can say that this is always satisfied.

Now the decoder is given the optimal value function and policy and it just refers to the grid file and accordingly performs the transitions.

Some general observations about the algorithms are :

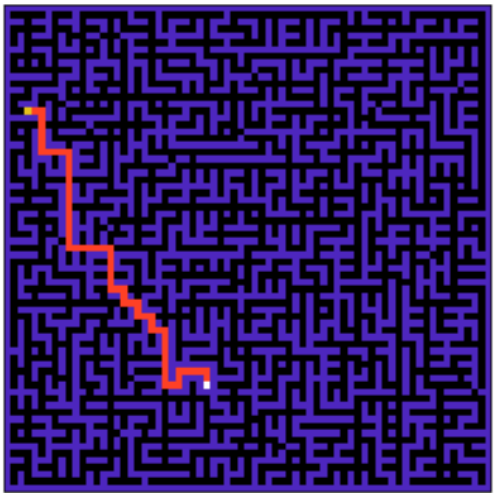
- Linear Programming takes around 1.5 minutes (total) to run on all instances of Mazes that have been given. The fact that once the problem has been formulated, the solver has to solve it only once and then there are only few computational steps before we get to the optimal policy. So, if the formulation is good (from a linear algebra point of view), the results are quick. This also depends on the solver's capabilities and pulp is good enough.
- Value Iteration, as the name suggests, reaches iteratively to the optimal value function and policy. So, as the number of states and actions increase, the time taken for convergence also increases. Another point that has been observed is that for discount factors close to 1, the convergence is very slow. Value Iteration takes around 5 minutes (total) to run on all the given instances.
- Howard Policy Evaluation needs to solve the Bellman equations at every iteration before convergence, and is thus slow for instances with high number of states and actions. For the given instances, we require around 35 minutes to run on all instances using this algorithm.

Here are some of the grid visualisations:

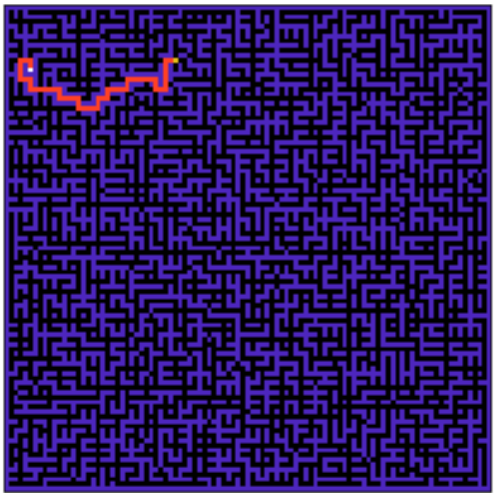


(a) Grid 10

(b) Grid 40



(c) Grid 70



(d) Grid 100

Figure 1: Shortest Paths