Author: Neil Acharya

CS 4641 Project 4 Markov Decision Processes

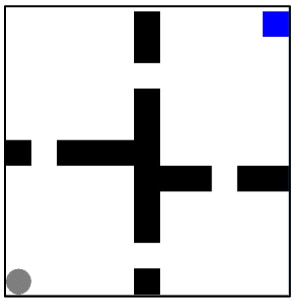
In this project, we are to explore how to define a **M**arkov **D**ecision **P**rocess (MDP) to help an agent act in an environment. We will use planning algorithms like Value Iteration and Policy Iteration to solve our MDPs, meaning achieve an optimal policy, and then use the learning algorithms Q-Learning and SARSA to learn an optimal policy.

What is an MDP?

An MDP is a way to define the decision-making engine of an agent in a world. There are four parts to a MDP: the set of states *S*, the set of available actions *A*, a model or transition function *T*, and a reward function *R*. A state is defined as all the necessary information relevant for the agent to make a decision in the world at the current time. An action is an action that the agent can take that may or may not change the state of the world. A model is a function in the form of $T(s, a, s') = \Pr(s'|s, a)$, or given a state and action how likely is the world to arrive at the new state *s'*. A reward function tells the agent the reward for arriving in a certain state, which can help the agent determine if their actions benefit them or hurt them. These four members can help completely define a MDP, which can then be solved to create a policy, or a mapping of state to action for our agent to maximize their rewards. The ultimate goal of defining an MDP is to either extensively plan out the optimal policy using Value Iteration or Policy Iteration when all these members are defined, or to learn the policy with a learning algorithm when the transition function and reward function are unknown.

My MDPs

For the first MDP, I wanted to keep things simple to help get used to our MDP framework BURLAP. As such, I went with their predefined "GridWorld" problem, where a single agent finds itself in a world of discrete row and column grid structures, where it can only move in the four cardinal directions, with the goal of reaching some goal location. The states here are defined simply by the XY-grid location of the agent. Its an 11x11 grid, meaning 121 possible states in the case where there are no obstacles. However, in our case we have the world split into four different "rooms" with one opening to each adjacent room, resulting in only 104 reachable states, as seen below:



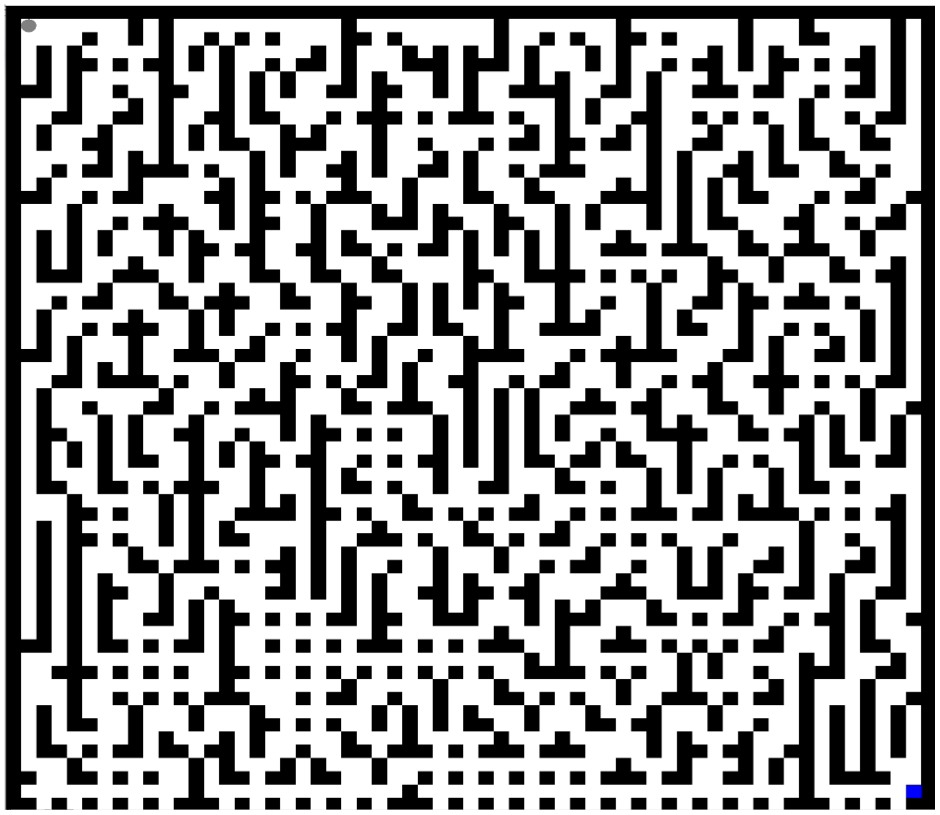
Explicitly defined, a state *s* is defined by the (x,y) coordinate of the agent.

The actions the agent can take are: NORTH, SOUTH, EAST, and WEST.

The agent receives a small negative reward of -0.04 for each move, except when it arrives in the blue square, where it receives 1.

The transitions for each action are stochastic, with an 30% chance to take an action successfully, while going in a random other direction in every other case.

The main change at this problem has is its highly stochastic, with only a 30% chance of success to move in the agent's desired direction. I expect this to change the optimal policy from the deterministic case dramatically.



My second MDP is a larger GridWorld with a more maze-like structure, having several possible dead-ends. I generated this grid world randomly using a DFS-based maze generation algorithm that can be found in *maze_maker.py*. The grid is 61x61, and therefore has a much more elaborate state space than the 11x11 situation from before, having 2241 reachable states. The definitions, however, remain the same as above, with the exception that all actions in this case are deterministic (because if they were stochastic this would take forever).

Planning Algorithms

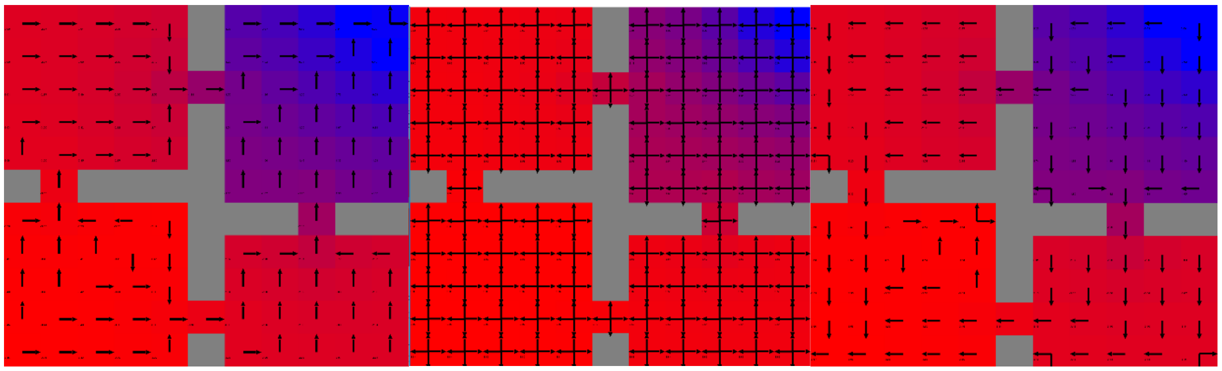
As the name suggests, planning algorithms actually plan a policy that optimally solves an MDP, or at least maximizes the sequence of delayed rewards of the MDP. These sequences of rewards can also be called the utility of a state, and is defined as the reward for our current state plus the discounted utility of the best adjacent state (adjacent here meaning reachable in one action). There exists a set of equations that explicitly solve for the value of the utility of every state called the Bellman Equations, and for any given MDP of n states, there will be n equations in n unknowns. Explicitly, the utility of a single state of an MDP is: $U(s) = R(s) + \gamma \max_{a} T(s, a, s')U(s')$. By solving these equations, you get the true utility of each state, and thereby arrive at an optimal policy.

Value Iteration

Solving the Bellman Equations would be easy since there are n equations and n unknowns; but the caveat is that these equations are not linear. The max function inside these equations means that they cannot be solved via a series of linear equations. Instead, to solve for these utilities, we use an algorithm known as Value Iteration. In value iteration, we initialize the utility of the states arbitrarily, then at each time step of the algorithm we update them based on the actual reward of the state and the utility of its neighbors. Once the algorithm converges and we arrive at the true utilities of the states, a true optimal policy can be derived.

Results

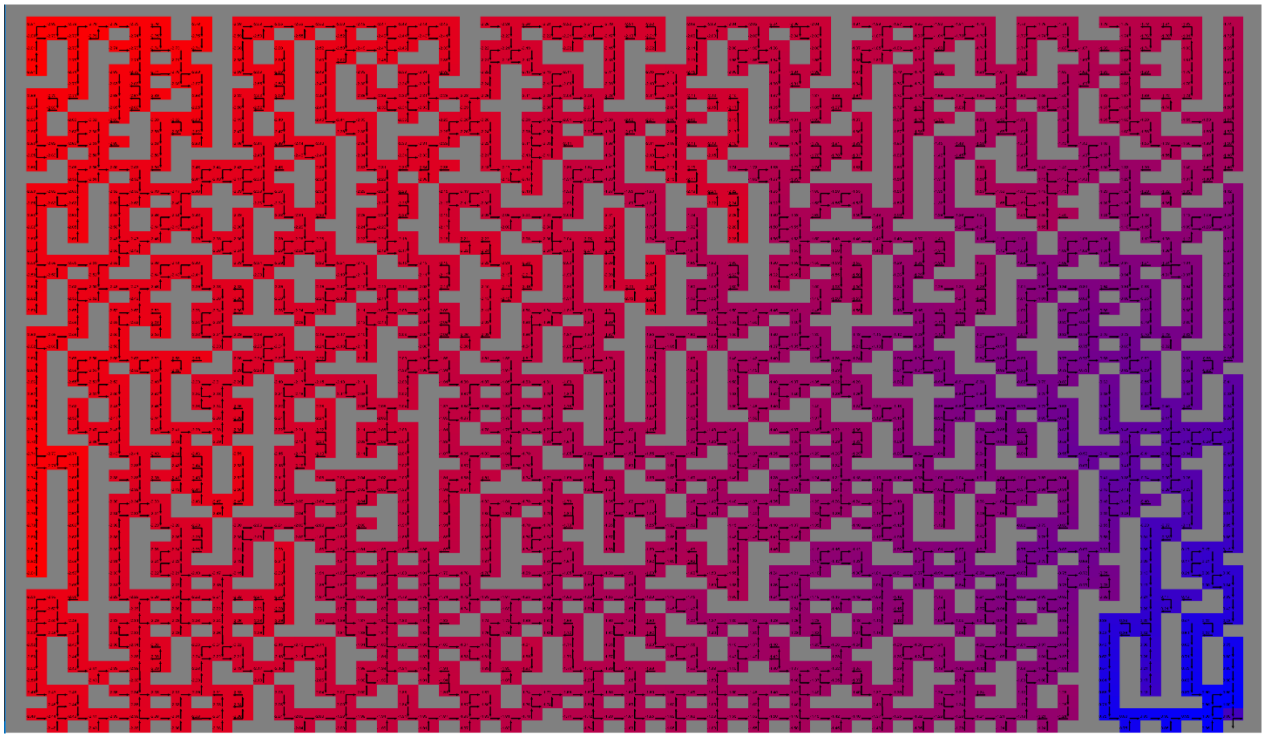
For the small grid problem, VI was run using a discount factor of .99, max change per iteration of 0.001, and 1000 max iterations. After running VI on the small grid problem at different success probabilities, we arrive at these policies:



The policies show a 30%, 25%, and 20% success rate respectively. For the 30% case it took 193 passes, for the 25% case it took 236 passes, for the 20% case it took 206 passes.

The completely random case took the longest to converge most likely because it had to propagate until all action-utility values were equalized while the other two cases could stop a bit earlier since some options for actions were clearly worse.

In the grid maze problem, VI was run with the same parameters to compare results. After running VI, the algorithm arrived at this policy:



This was run in 123 passes, which is lower than the small maze. I believe that this is due to the stochasticity of the world: the more stochastic it is, the more iterations are required to propagate the true reward to each state and its neighbors, meaning the policy will take longer to update in the long run. Interestingly, although the clock run time of the algorithm took longer for the grid maze problem, the number of states doesn't seem to influence the iteration count of the algorithms.

Policy Iteration

Although calculating state utility will eventually arrive at the right answer, it is more information than needed to arrive at the optimal policy. Compared to Value Iteration updating the utility of states, Policy Iteration (PI) arbitrarily initializes a policy and iterated and update that policy instead. This allows the algorithm to use this form of function approximation: $U(s) = R(s) + \gamma T(s, \Pi(s), s')U(s')$, where instead of picking the max from the actions, it instead uses its suggested policy. Each iteration updates these suggested utilities in an internal value iteration loop until the policy is changed. This usually results in fewer overall iterations, but each iteration costs more.

Results

In the small grid world, PI was run with a discount factor of 0.99, max change of 0.0001, max inner value iteration count of 100, and maximum policy iteration count of 100. These policies were all identical to those found by VI before.

For the 30% success rate MDP, PI had 5 iterations, with 305 internal iterations of VI. The 25% run also had 5 PI iterations, with 363 internal VI loops. The 20% case had 5 PI loops and 300 VI loops.

So, even though PI had fewer overall iterations, it did more policy evaluation than VI to figure out the optimal utilities. This is because instead of finding the optimal utilities, it only cares about its proposed policy and the changes there.

In the grid maze, PI was run with the same parameters. PI also arrived at the same policy as VI in this case, but looking at the episodes for each, they found two different sequences with the same final cumulative reward.

Comparison

Column1	Iterations (avg)	Internal Iter.	Time (ns)	Time (s)
VI(SmallGrid)	211.6666667	N/A	623086954.7	0.623087
PI (SmallGrid)	5	322.6666667	822037786.3	0.822038
VI(Maze)	123	N/A	1526022981	1.526023
PI(Maze)	73	1851	19374982377	19.37498

In both MDPs both algorithms arrived at the optimal policy, but take different amounts of iterations and clock time to run. The most interesting result here is the fact that for VI, the larger state space did not seem to make a difference in the iteration count for maximization of utility. This is probably because the maximization of utility is dependent not only on the state space, but the action space for maximization, and the action space between both MDPs is identical and small.

Learning Algorithm

Compared to using VI and PI to solve a fully defined MDP, learning algorithms don't have access to all the definitions and functions of the MDP and must achieve a policy through exploration. An agent that is interacting with their environment using a learning algorithm only knows the state it is currently in and the actions available to it; it knows nothing of the transition model or reward function of the MDP. Instead, the agent is put into a simulator of the environment where it samples the model in episodes, receiving rewards as it goes along, updating what it believes its policy is. This allows an agent to explore its environment/ approximate a solution to an MDP. The two learning algorithms taught in class are Q-Learning and SARSA.

Q-Learning

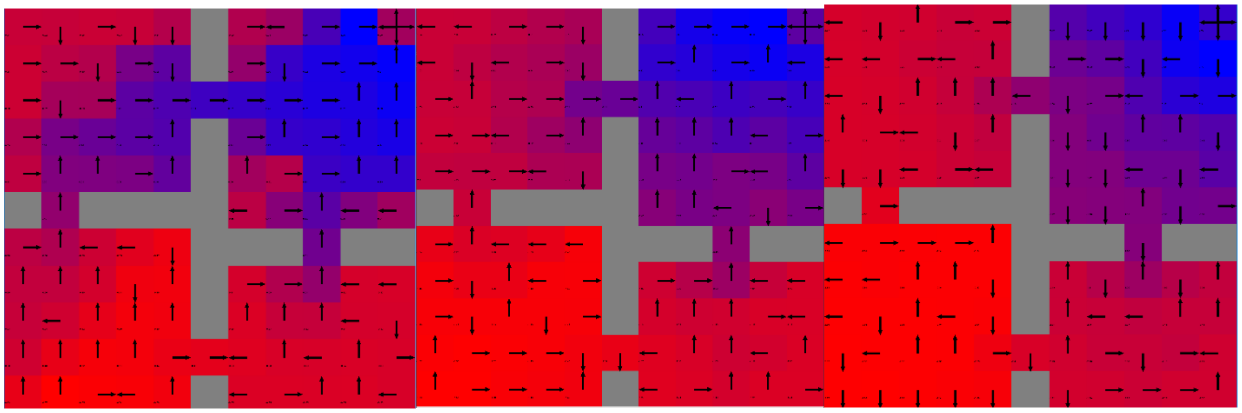
Of the two learning algorithms, Q-learning is like value iteration in that it tries to approximate utility rather than directly creating a policy. However, the actual utility can't be calculated since the transition model and reward function are unknown, so instead we calculate a function *Q* that approximates utility. *Q* is parameterized by state-action pairs rather than just states and updates similarly to value iteration. Specifically:

$$Q(s_t, a_t) \overset{\alpha}{\leftarrow} R_t + \gamma \max_a Q(s_{t+1}, a)$$

where α is the learning rate and γ is the discount factor. Given infinite episodes, and updating Q each step of each episode, Q will converge to the optimal policy.

Results

For small grid world, our q-learning agent was run with a discount factor of 0.99, all Q-values initialized at 0, a learning rate of 0.5. The Q-Learning agent had an epsilon greedy policy with $\epsilon = 0.1$, meaning 90% of the time it would follow its policy, while 10% of the time is would pick a random action. This time, I also tested how stochasticity effected the policy using 80%, 50%, and 20% success rate respectively for each policy below:

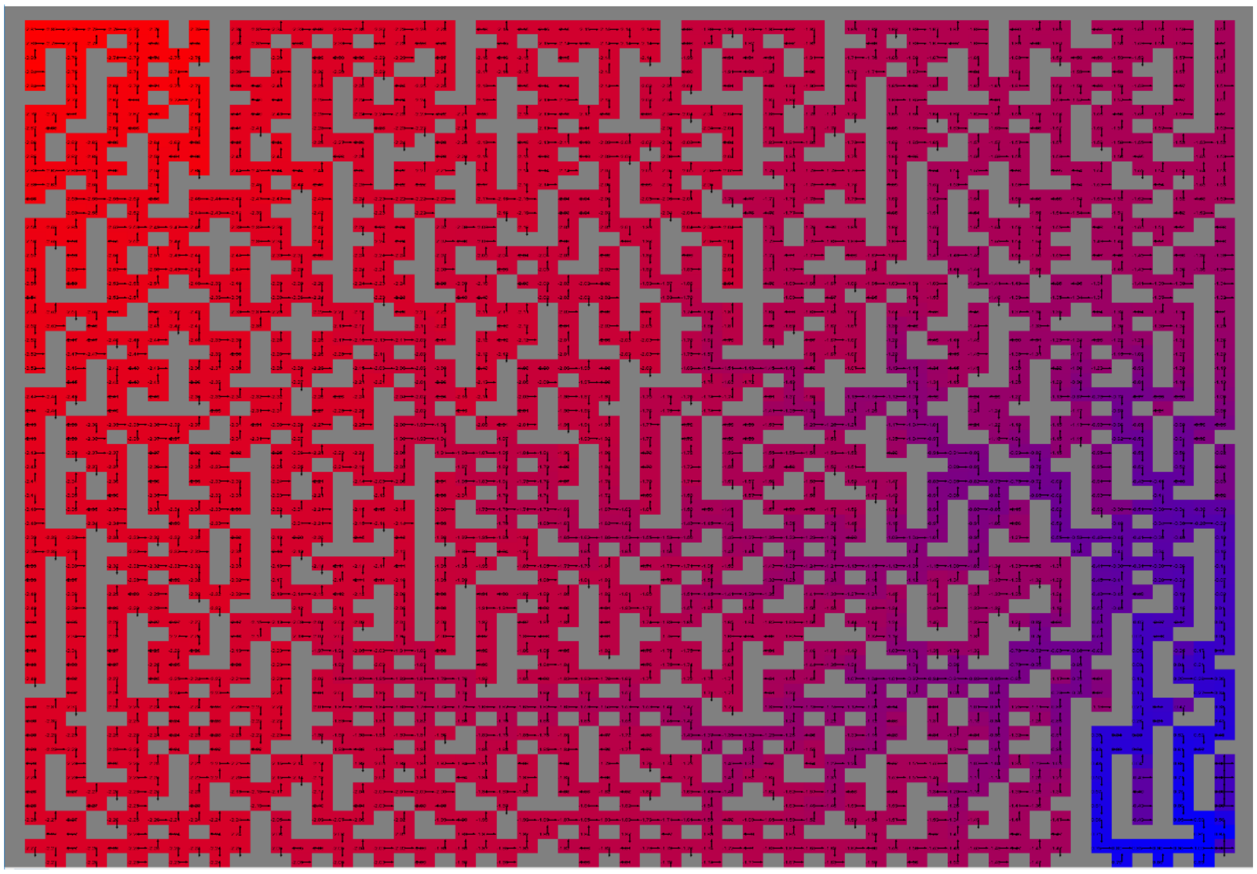


The 80% case ran in 1.1907 seconds, 50% case ran in 1.69 seconds, and the 20% case ran in 4.16 seconds. Each learner ran for 100 episodes.

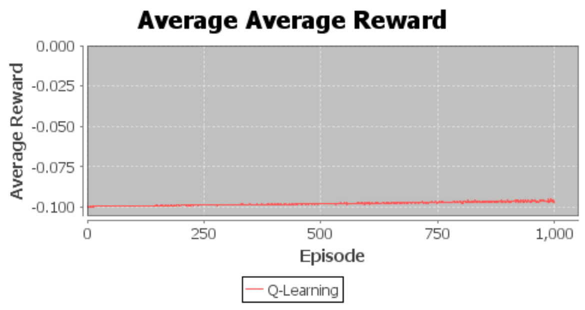
We can see from the apparent policies above that in the 80% case, we arrived at a reasonable policy that solves our MDP. The 50% case has a good policy for the upper right room, but seemingly random everywhere else. The 20% case is seemingly random throughout.

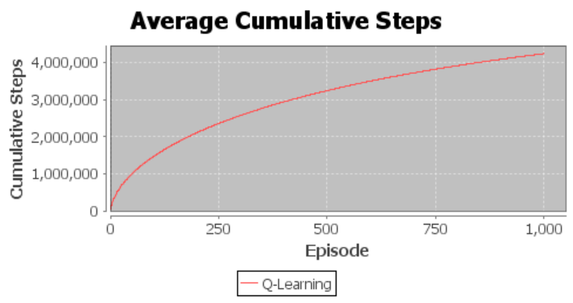
This leads to the conclusion that the more probabilistic/ non-deterministic an environment is, the slower q-learning will converge to an optimal solution. Not only that, but each individual episode the agent partakes in also increases in length the less reliable actions are. This also makes sense since each episode will last until the MDP reaches a terminal state.

In the Grid Maze Problem, the agent was initialized with a discount factor of 0.99, initial Q-values of 0, and learning rate of 0.3. This also used an Epsilon Greedy Policy with $\epsilon = 0.1$. In this case, however, the agent was run for 2000 episodes. This agent took 57.03 seconds to run.



After 2000 iterations, Q-learning does arrive at the same policy as Value Iteration in this case.





Here we see that as Q-learning runs, there is a steady increase in the amount of reward it picks up per episode, and a decrease in the number of steps each episode takes.

In general, Q-learning was able to arrive at a reasonable policy and solve the MDP for the larger state space given it had more episodes to learn with. This is because of the completely deterministic world compared to the variably stochastic world of the Small Grid problem. Comparing the two even further, the calculated run-time per episode in the maze versus the 20% success rate room gird is 0.028515 against 0.0416, meaning episodes in the small grid world took longer. From this we can see that stochasticity has a higher impact on Q-learning agents than the state space in regards to average episodic runtime, which makes sense.

Conclusion

Comparing MDP planning algorithms to learning algorithms in the above two cases, we can draw a variety of conclusions in regard to how to apply each. Despite our Small Grid world having a smaller state space, having that tradeoff in stochasticity really hurt the learning algorithms, while policy and value iteration seemed to remain unchanged since the model was known in those situations. Comparatively, for Grid Maze both planning and learning algorithms arrived at acceptable policies from the initial state.

From these conclusions, we can see that the Bellman Equations, although are parameterized by the number of states, are more importantly bounded by the max/argmax term with regards to the action space of the problem. The larger the action space is, the harder the computation is and the longer both VI, PI, and Q learning will take.

One aspect of learning algorithms I couldn't explore but is worth mentioning is the initial policy decision that the learner chooses to adapt. In all our cases, this was the ϵ-greedy policy iteration method with $\epsilon = 0.1$. In other words, by adopting this policy we address the exploration-exploitation dilemma by taking random actions 10% of them rather than exploit our policy completely. Either by changing ϵ or adopting a different strategy, the performance of Q-learning could change drastically by encouraging the algorithm to explore more or less.