CS 4641 - Assignment 4

By: Sean Chua (schua8)

Introduction and Overview

Note: We will be using the CMU Reinforcement Learning Simulator throughout this analysis.

Overview of Markov Decision Processes (MDPs)

Markov decision processes represent a convenient mathematical framework for modeling real world processes and decision making, particularly in situations where the outcomes are influenced by some form of probability and are also partially controlled by some decision maker (i.e. an agent).

MDPs are defined to have the following characteristics:

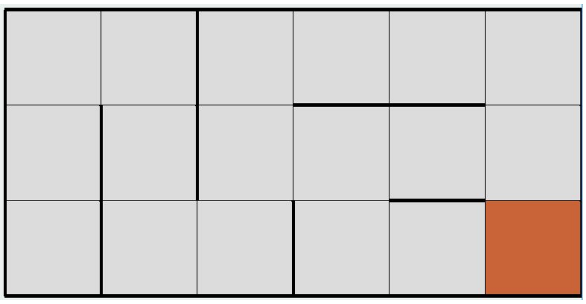
- **S** - a finite set of states, from which actions can be performed
- **A** - a finite set of actions, representing the actions that can be performed over the set of states
- **T(s,a,s') = Pr(s' | s, a)** - the transition function that represents the probability of an action a leading to a state s' from a given state s; this model encapsulates the underlying stochastic process (i.e. the randomness)
- **R(s)** - the reward function for the immediate value of being in a particular state
- **π(s)** - the policy of the decision maker, specifying which action to take in particular state

Note that the utility of an MDP in solving a decision making problem is dependent on the Markov property (Markov assumption), which states that the conditional probability of future states is dependent only on the present state of the system, and not on any previous states. Another assumption we need to include for MDPs is the assumption of stationarity, which says that the only parameter to our policy function is the given state; we are not limited by other constraints such as time, number of steps, etc, which in real life processes may change our choice of action in a given state.

For this MDP model, we are looking for the optimal policy π*, which provides maximum total reward.

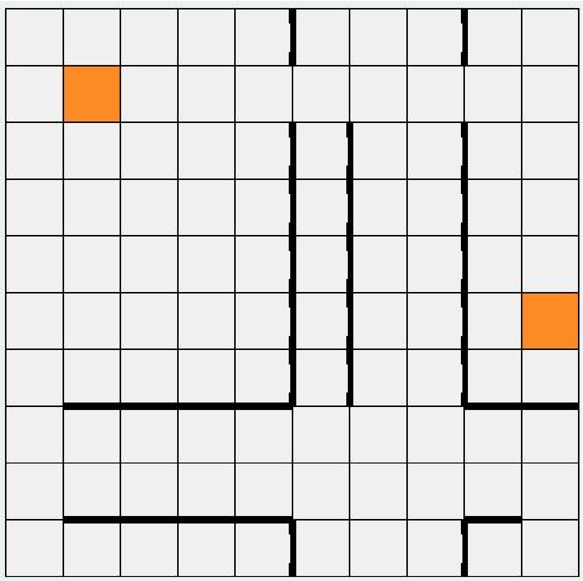
Grid Worlds

To get a good idea of the potential performance performance differences between value iteration, policy iteration, and Q-learning, we will apply them to an easy GridWorld and a more difficult one with more states and see how the algorithms perform in each case. The easy Gridworld is down below:



Our easy GridWorld is laid out like a simple maze, so that we can understand how algorithms navigates around walls and chooses the shortest paths in order to get to the goal. In the middle of the route, there are three possible routes to take, each separated by a wall. When first creating this GridWorld, I was very curious as to the differences in policies from each algorithm depending on how much living penalty and discounting I gave it.

This is the next GridWorld that is more difficult:



This next maze is more interesting, because I roughly modeled it after driving down a street. The goal is to find a restaurant to eat at, which is represented by the red squares. One of the goals only has one entrance, so I'm curious to see how many states will have a policy pointing towards that goal. This problem is also interesting because this could also be expanded to act the same way that a GPS works. Essentially, based on whatever a person's position is, a person could see what policy to follow in order to get to maximize their utility (i.e. find the nearest goal).

Value Iteration

The concept behind value iteration is that if we could assign a value or utility to every state taht represented the usefulness of that state to the solution as a whole, then determining the optimal policy for a given process is simple. Given a state, we just need to simply perform the action that takes us to the state with the maximum utility from our given options. The problem with this is that because this is applied to MDP problems, there is a degree of randomness which affects the expected value of a state getting a certain reward from a certain action. Value iteration also introduces the notion of time-limited values and uses those values to calculate values for each state. Below is the full equation:

$$V_{k+1}(s) \leftarrow \underline{\max_a} \sum_{\underline{s'}} T(s,a,s') \left[\underline{R(s,a,s')} + \gamma \underline{V_k}(s')\right]$$

Where $V_{k+1}(s)$ is the optimal value of s if the game ends in k more time steps. We are also taking each action and summing up the total expected value and getting the maximum value in order to find the optimal value. We repeat this value calculation with each time step until convergence.

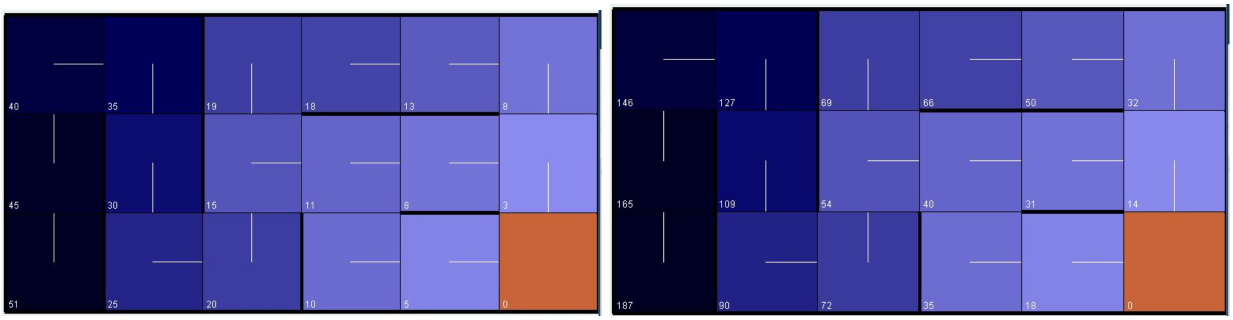
Small maze:

PJOG	**Precision**	**Number of Steps**	**Time taken (ms)**
0.1	0.001	24	2

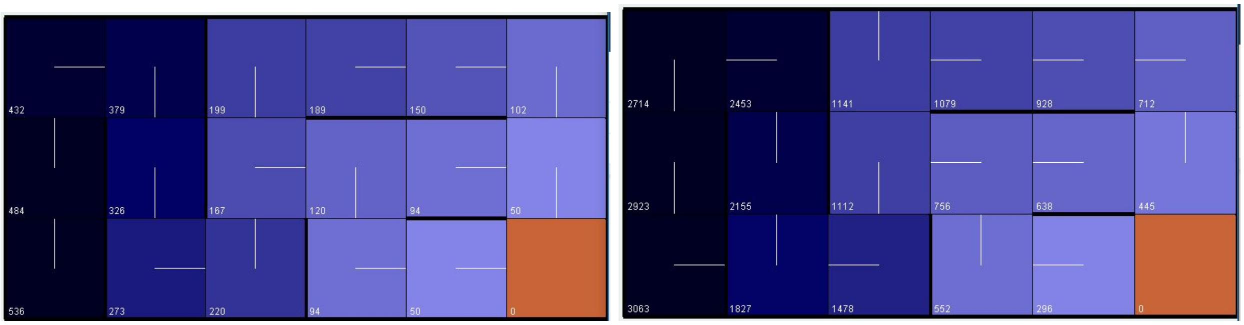
0.1	0.01	21	0
0.1	0.1	18	0
0.3	0.001	48	19
0.3	0.01	41	4
0.3	0.1	33	0
0.5	0.001	115	21
0.5	0.01	94	4
0.5	0.1	74	2
0.8	0.001	677	84
0.8	0.01	538	18
0.8	0.1	399	6

As we can see, the number of steps and the total time taken seems to increase when we increase PJOG. This makes sense, as PJOG indicates the amount of randomness in our system. This means that although an agent may want to move in a certain way, there is a high probability that the agent may not move according to the initial action made. This makes it more difficult for value iteration to converge on a policy that produces the optimal utility because there is so much variability. Decreasing the value of the precision parameter actually increases the precision and requires the algorithm to be much more accurate when converging.

Here are the policies that value iteration came up with:



Left: PJOG = 0.1; Right: PJOG = 0.3

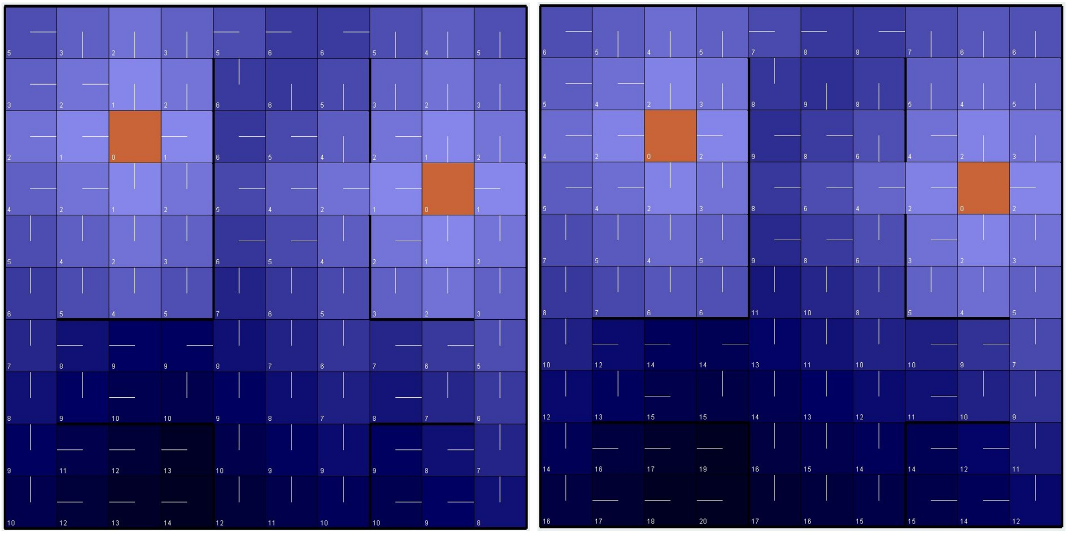


Left: PJOG = 0.5; Right: PJOG = 0.8

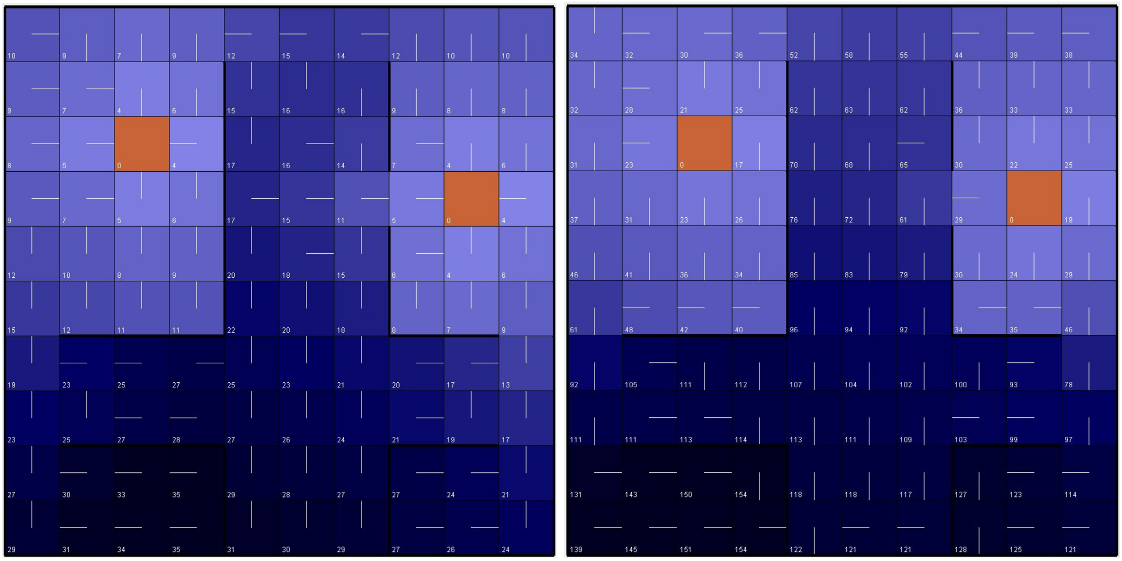
The pictures above show the results after each instance terminates with a precision of 0.001. The other ones are omitted because they have the same end result. The first two policies seem to behave as expected; however, with PJOG = 0.8, we see some interesting choices for policy. Overall, the policy seems to aim for almost the opposite direction in order to compensate for the very high probability that the action will deviate from what was originally intended. One example of this is in the bottom left corner, where the policy is actually pointing towards a wall. However, because it is most likely not going to take that action, it will probably go up or in a different direction instead.

Large maze:

PJOG	**Precision**	**Number of Steps**	**Time taken (ms)**
0.1	0.001	24	20
0.1	0.01	21	15
0.1	0.1	17	36
0.3	0.001	47	44
0.3	0.01	38	34
0.3	0.1	28	27
0.5	0.001	112	105
0.5	0.01	85	76
0.5	0.1	57	80
0.8	0.001	680	608
0.8	0.01	483	407
0.8	0.1	284	250



Left: PJOG = 0.1; Right: PJOG = 0.3



Left: PJOG = 0.5; Right: PJOG = 0.8

Again, we see the same trends as with the smaller maze, where as we increase PJOG and increase the precision to which we want our algorithm to converge to, the amount of time and number of steps also increases. We can also see that this maze takes longer to converge than the smaller one; because it has more states to calculate for each V_k per time step, it takes much longer to converge.

Unfortunately, we were unable to model discounting, terminal states (which provided negative utility) and living rewards with the software used, but some experimentation with this would be interesting to see. For example, having a very high living penalty usually incentivizes the agent to find the nearest terminal state, regardless of whether that terminal state yields positive or negative values. Even with the addition of negative terminal states, agents may take completely different routes to avoid them. Because of the random nature of MDPs, being in an adjacent space to a negative terminal state could be dangerous, as an action intended to move forward to a positive terminal state could have the agent end up in the negative terminal state instead.

Policy Iteration

Value iteration is rather useful, but it suffers from a few weaknesses. One is that it can be incredibly slow to converge for some scenarios, despite the fact that the underlying policy it is computing remains the same, and that it only indirectly finds the optimal policy. Value iteration is finding the long-term value or utility of each state, and then using that to find the optimal policy. Thus, we can move past this and try to just find the optimal policy directly, just as policy iteration does.

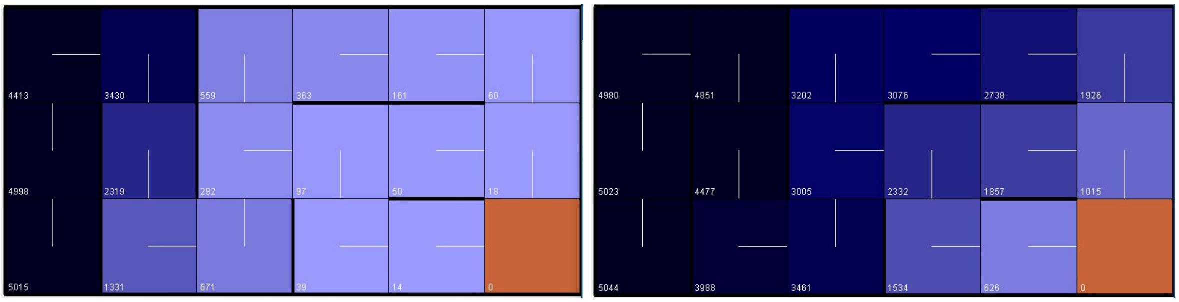
With a simple modification to value iteration, we can iterate over policies instead of states, by selecting a random policy, computing the utility of each state, and then selecting an optimal policy from the already computed policies:

- Generate an initial random policy, being a permutation of actions for all state in the MDP
- Loop until no action in our policy changes:
    - Compute the utility for each state in **S** relative to the current policy
    - Update the utilities for each state
    - Select the new optimal actions for each state, changing the current policy in the process

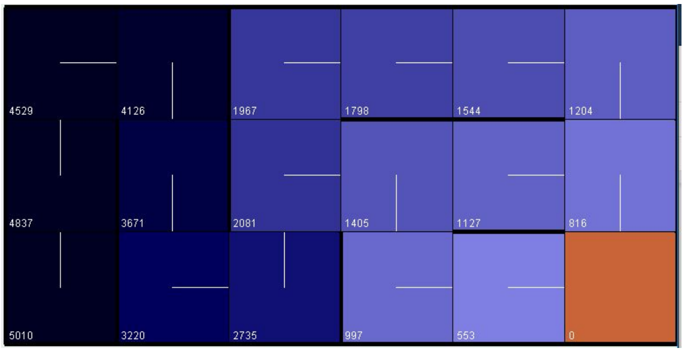
Policy iteration also makes use of Bellman equations, similar to value iteration. With each iteration, the new policy will be better, or else the algorithm will converge.

Small maze:

PJOG	**Precision**	**Number of Steps**	**Time taken (ms)**
0.2	0.001	3	1
0.2	0.01	3	5
0.2	0.1	3	10
0.5	0.001	3	2
0.5	0.01	3	2
0.5	0.1	3	6
0.8	0.001	2	4
0.8	0.01	2	7
0.8	0.1	2	6



Left: PJOG = 0.2; Right: PJOG = 0.5

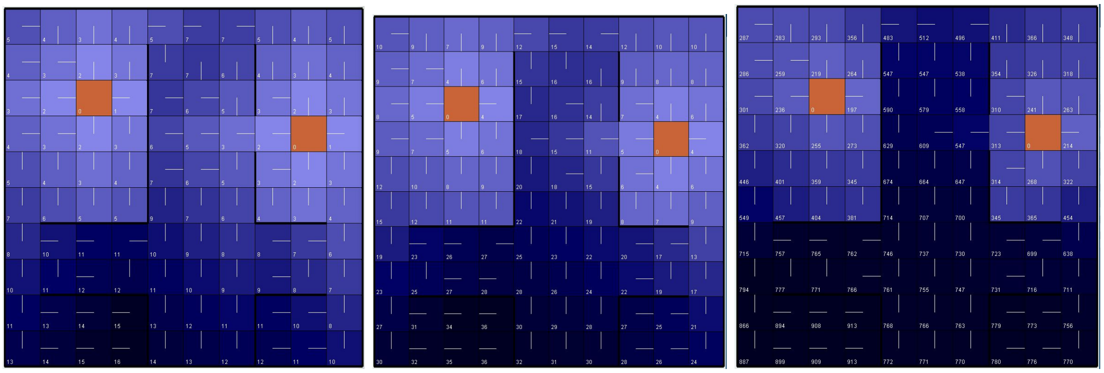


PJOG = 0.8

In comparison to the results from value iteration, we seem to have somewhat differing results. The actual policy for this algorithm seems to be more constant, even with increasing PJOG. The number of steps and amount of time for the algorithm to converge also seem pretty consistent. This is interesting, as it seems to find the optimal policy much more quickly than with value iteration. This can be attributed to how simple the maze is. There are only a few routes to choose from, so the algorithm didn't have much to go through, and didn't need to be constantly updating for each iteration.

Large maze:

PJOG	**Precision**	**Number of Steps**	**Time taken (ms)**
0.2	0.001	6	253
0.2	0.01	6	244
0.2	0.1	6	218
0.5	0.001	4	66
0.5	0.01	4	70
0.5	0.1	5	32
0.8	0.001	3	458
0.8	0.01	3	283
0.8	0.1	3	280



5	4	3	4	5	7	7	5	4	5
4	3	2	3	7	7	6	4	3	4
3	2	0	1	7	6	5	3	2	3
4	3	2	3	6	5	3	2	0	1
5	4	3	4	7	6	5	3	2	3
7	6	5	5	9	7	6	4	3	4
8	10	11	11	10	9	8	8	7	6
10	11	12	12	11	10	9	9	8	7
11	13	14	15	13	12	11	11	10	8
13	14	15	16	14	13	12	12	11	10

10	9	7	9	12	15	14	12	10	10
9	7	4	6	15	16	16	9	8	8
8	5	0	4	17	16	14	7	4	6
9	7	5	6	18	15	11	5	0	4
12	10	8	9	20	18	15	6	4	6
15	12	11	11	22	21	19	8	7	9
19	23	26	27	25	23	22	20	17	13
23	25	27	28	28	26	24	22	19	17
27	31	34	36	30	29	28	27	25	21
30	32	35	36	32	31	30	28	26	24

287	283	293	356	483	512	496	411	366	348
286	259	219	264	547	547	538	354	326	318
301	236	0	197	590	579	558	310	241	263
362	320	255	273	629	609	547	313	0	214
446	401	359	345	674	664	647	314	268	322
549	457	404	381	714	707	700	345	365	454
715	757	765	762	746	737	730	723	699	638
794	777	771	766	761	755	747	731	716	711
866	894	908	913	768	766	763	779	773	756
887	899	909	913	772	771	770	780	776	770

Left: PJOG = 0.2; Middle: PJOG = 0.5; Right: PJOG = 0.8

Similar to value iteration, policy iteration took more time to converge for the maze with more states. However, this required much fewer steps to do so. Thus, it seems that the number of steps stays relatively constant for policy iteration. This is a bit surprising given that this maze is still a bit more difficult. Yet, although value iteration and policy iteration produced such similar results, policy iteration consistently did better in terms of time and number of steps. Again, it would have been interesting to see if we could have added more parameters like discounting, living penalties, etc.

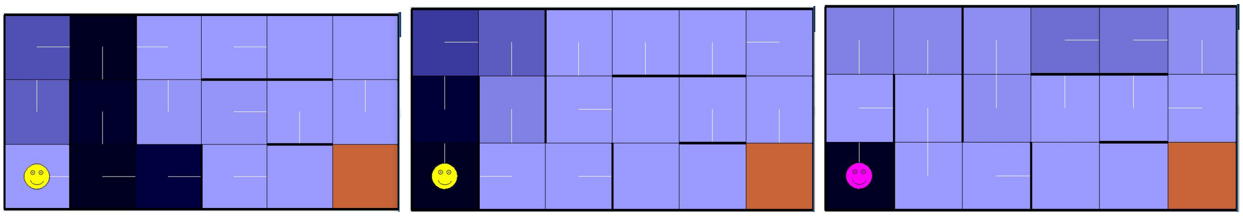
Overall, while both value iteration and policy produce similar results, there are some fundamental differences between the two. Value iteration focuses on calculating the value of every state and only converges when the change in each value over each iteration is less than the threshold. Policy iteration focuses more on the optimal action and only converges when there is no change in any of the optimal action. Because of this, policy iteration terminates much more quickly, while value iteration can get stuck in some situations where some of the value changes are not below the threshold, yet the policy is still the same.

Q-learning

Both value iteration and policy iteration are incredibly useful for determining the optimal policy for a given MDP, but they do so by making a strong assumption that the decision maker has a substantial amount of knowledge about the underlying stochastic process, namely that the transition model is known along with the rewards for all possible states. In many real world applications, we may not have access to all of this information, so we'll have to explore other means of solving this problem.

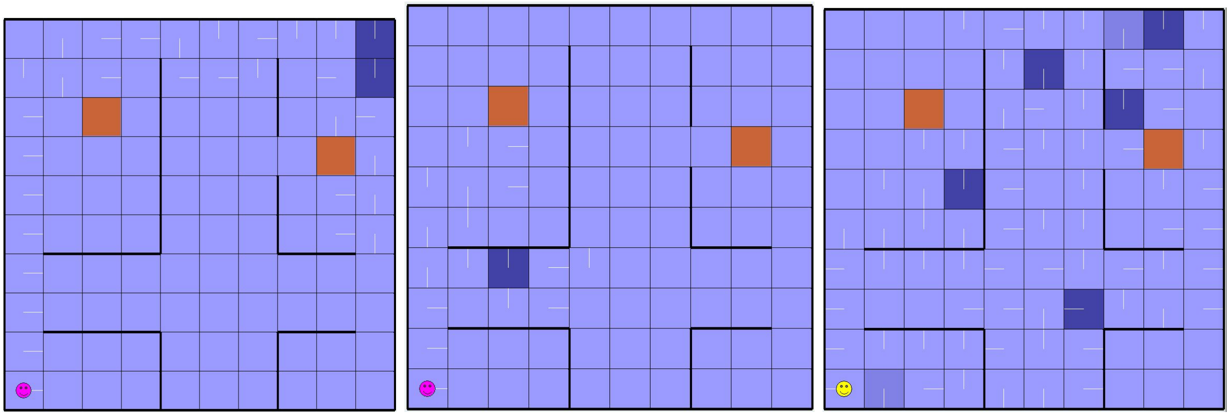
Because of their reliance on having an MDP, neither value iteration nor policy iteration are true reinforcement learning algorithms, since the optimal policy can be effectively computed almost deterministically. Q-learning is very different in that it learns about its environment through exploration and exploitation. Exploration in this case refers to taking a non-optimal action (which could lead to maximum utility), while exploitation refers to using the knowledge that has already been built into the policy and picking the best action. This is similar to how simulated annealing works, as it also makes "bad" decisions to avoid being stuck at local optima. In the CMU Reinforcement Learning Simulator, this value is represented through "Epsilon", the probability that the algorithm acts randomly. Another important parameter, learning rate, is a value between 0 and 1 and represents how much the new policy changes the existing policy.

For these trials, I used a PJOG of 0.2, a precision of 0.001, enabled the decaying learning rate option, and ran the algorithm for 1000 epsiodes, where one episode represents transitioning from the start state to another state repeatedly until the goal state is reached. Because decaying learning rate is enabled, each successive policy has a smaller influence on the overall policy. Therefore, the initial policies have the largest impact. Below are the results for Q-learning for the small maze:



Left: Epsilon = 0.2; Middle: Epsilon = 0.5; Right: Epsilon = 0.8

Initially, we tried to use a learning rate of 0.1, but many of the policies generated did not seem to work very well. This is because the algorithm would rely too much on the randomly generated policy at the beginning, which led to too much variability among different runs. Thus, we used a learning rate of 0.7 and changed the epsilon to see how that would impact our results. There were varying results among each algorithm, but it seems that having a balance of exploration and exploitation worked the best with an epsilon of 0.5. This makes sense as there are a few different routes that the agent can explore. Let's see how this compares with our larger maze:



Left: Epsilon = 0.2; Middle: Epsilon = 0.5; Right: Epsilon = 0.8

After running Q-learning multiple times, it seemed that there was a large amount of variability in the results. Most of the results pointed the agent towards the goal on the left, which makes sense as it is closer to the origin point. It was interesting to see that with epsilon of 0.8, this was the only one that ended up even exploring the right side of the board consistently. In comparison to the smaller board, Q-learning seems to not have explored as much of the board, due to having found a close goal already. It'd be interesting to play with the starting position of the agent, as this would probably affect how much it explored the entire board. Overall, it seems that with a larger state size, the impact of exploration versus exploitation increases. It seems that an agent will avoid more difficult goals (e.g. the one that's farther away) and prefers closer goal states, especially when the epsilon is low (i.e. low exploration). However, in the case of exploration, there is a greater possibility of a path leading to the more difficult goal that is farther away. Some other things that would be interesting to see would be the lowering epsilon over time or using exploration functions. Especially since the agent is attempting to learn more about the environment at first, this would help Q-learning by exploring areas whose “badness” is not yet established, so it can eventually stop exploring.

Conclusion

Throughout this analysis, we have learned about numerous reinforcement learning algorithms, specifically vlaue iteration, policy iteration, and Q-learning. While value iteration and policy iteration converged to similar solutions for both the big and small mazes, Q-learning performs pretty poorly on the big maze. For most runs on the big maze, the algorithm didn’t even evaluate some states, so it wasn’t as useful as expected. Value iteration is typically fast at computing the optimal policy but also needs a large number of iterations to converge, not to mention how it needs full knowledge of the state space and model. However, policy iteration requires far fewer iterations and runs at similar or shorter times than value iteration. On the other hand, Q-learning requires no prior knowledge of the state space or the model and introduces the notion of exploration vs exploitation. Exploration leads to the agent taking less optimal actions, but can result in finding a more optimal solution in the future. Exploitation leads to the agent utilizing actions that are already known to produce good results, but this increases the potential of a policy getting stuck at local optima. Finding a balance between exploration and exploitation is key to effectively finding solutions when using Q-learning. Q-learning takes exponential time compared to either value iteration and policy iteration, so in any situation where we can be reasonably certain about domain information, we should pick either of those methods. In reality, Q-learning is much more applicable because it is much more adaptable than either MDP algorithm. However, gaining any domain knowledge may allow us to create effective models of the environment, and in this situation, one of the MDP algorithms would fit accordingly.