

# MARKOV DECISION PROCESSES

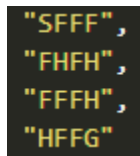
## Requirements

OpenAI gym, pymdptoolbox with some modifications, numpy, python 3.6

## 1. Problem Description

### 1.1 Frozen Lake

Frozen Lake is a classical grid-world problem. For a 4x4 frozen lake, which can be represented by the following letter strings:



```
"SFFF",  
"FHFH",  
"FFFF",  
"HFFG"
```

Figure 1.1 A 4x4 Frozen Lake visualization

Each letter represents a certain state. In each epoch the player starts from the upper left corner (S), and the goal is to reach the lower right corner (G, where player will be given a reward of 1). The player is allowed to move UP, DOWN, RIGHT, and LEFT. If the player is at F (Frozen), it is given a reward of 0; if the player is at H (Hole), the game is reset and starts over. The purpose is to find a path from S to G without falling into H.

The action at each state can be deterministic or probabilistic. That is, when player decides to move to a specific neighbor, deterministic action allows the player to do so, while probabilistic action assign a probability to the action. For simplicity, the probability to move to the desired neighbor is set to be  $1/3$ , while the probability of 'slipping' into the neighbors on the perpendicular direction is  $1/3$  each.

The problem has many advantages for reinforcement learning starter. First, it is easily scalable: if more states are needed, the map can be easily re-created with more grids. Second, visualization is easy. Thirds, straightforward sanity check on fewer states can be performed before moving into complex states.

### 1.2 Forest Management

Forest management is a simple MDP. Each year, the forest is represented by a state S. There are two actions that can happen to a state: KEEP (0) and CUT (1). If the KEEP is performed, a reward of R1 is given; on the other hand, if CUT is performed, a reward of R2 is given. Each year, there is a probability of P that the forest is burned. If the forest is burned, it returns to state 0.

The interesting part of this problem is the rewards for KEEP and CUT, and the probability of forest fire. For example, if the probability of forest fire is very high, to maximize total reward the forest will need to be cut every year no matter what. On the other hand, if the probability of forest fire is somewhat high, but the reward to keep forest is much higher than to cut down the forest, we might want to take a risk and wait. Different combination of rewards and probability affect the final policy, also algorithm performance.

## 2. Frozen Lake

For Frozen Lake, sanity check is performed on a 4x4 frozen lake (Figure 1) using Value Iteration. Then a more complex Frozen Lakes (20x20) are used for both Value Iteration and Policy Iteration to compare these 2 algorithms. Four actions (UP, DOWN, RIGHT, LEFT) are mapped as follows:

LEFT: 0, DOWN: 1, RIGHT: 2, UP: 3

### 2.1 Sanity Check

To avoid confusion, only deterministic action is allowed for sanity check. Two 4x4 maps are evaluated, and policy maps are shown below.



Figure 2.1 4x4 Frozen Lake map (left) and corresponding policy map (right)

In Figure 2.1, red boxes represent holes on the map. For the first map, the policy suggests that when we start from S, we should move down (1) twice, then move right (2) once, then move down again (1) once and then move right (2) twice to get to the goal. However, since there are multiple path to reach the goal, the player has other options if he/she is at other states. For instance, if the player happens to be at the state indicated by the blue arrow, the optimal policy is to move to the right (2) first, and then move down.

The first map is then modified by deliberately blocking all paths to the goal but one. As can be seen from the second policy map, all states on the upper right corner now have no choice but to move left (0) to the start point. At start S, the only path is to move down (1) to the lower left corner, and then move right (2) to get to the goal.

Sanity check on two small maps shows that algorithm is able to find the best policy from start to goal for every single state.

### 2.2 Value Iteration

Value iteration iterates utility to get the best value function for each state. To check convergence, utilities from the previous iteration step are compared with the current utilities, and if the difference is smaller than epsilon, iteration is complete.

One key parameter to determine how fast reward populates in each state is decay factor. The decay factor varies between 0 and 1, where 0 means current state utility is not affected by its neighboring utilities, 1 means current state utility is affected equally by all other state utilities, in other words, there is no influence decay due to neighbor distance. In this assignment, different decay values are tested and compared.

Decay factor	0.99	0.75	0.6
--------------	------	------	-----

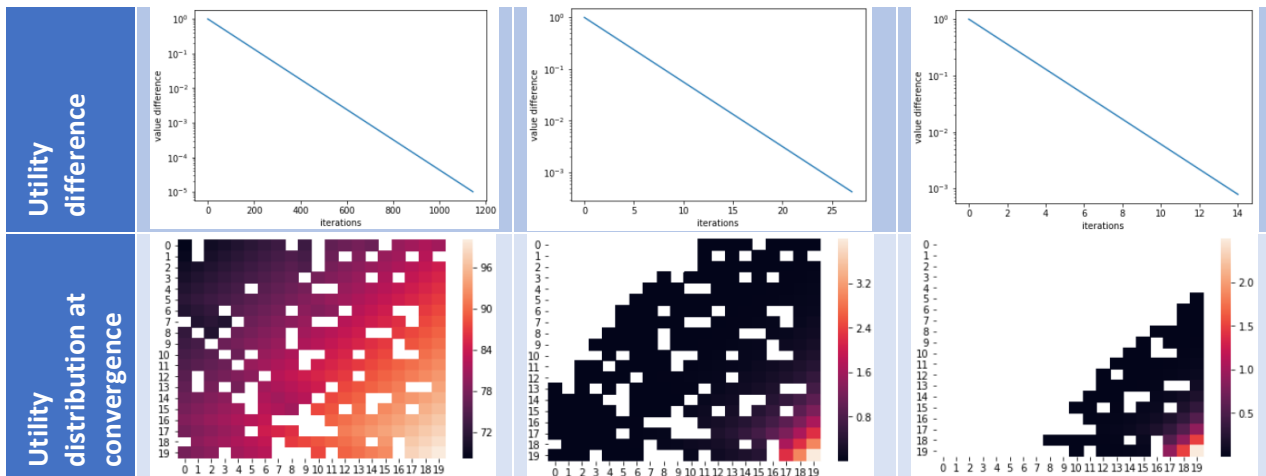


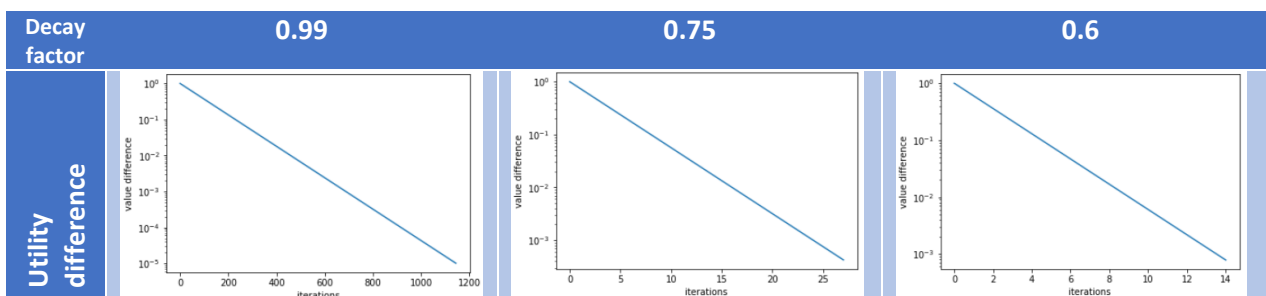
Figure 2.2 Utility difference vs. iteration and utility distribution at convergence, deterministic action

Figure 2.2 shows the results with deterministic action environment. As decay factor decreases, value iteration converges with fewer iterations. However, convergence with lower decay factor could potentially lead to incorrect policy. This will be addressed in the next paragraph.

Heat maps in Figure 2.2 show utility distribution at convergence for different decay factors. The isolated missing boxes are holes on Frozen Lake, and the upper left corner where heat map is missing is due to utility of that grid being 0. For decay factor = 0.99, reward manages propagate from goal (lower right corner) to the entire map. At convergence, the entire map has a general upward gradient from upper left (start) to lower right (goal). It makes logical sense since to extract policy from utility, a state has to compare utilities for all its neighboring states and find out the largest one, and then assign an action towards that neighboring state. Only an upward gradient from start to goal would allow action at each state to point to the goal.

As decay factor decreases, however, a big portion of the heatmap remains 0. This means 1) the reward from goal does not propagate to those states, and 2) without a gradient, there is no way to get a correct policy for those states. Even though when decay factor = 0.75 or 0.6 iteration still manages to converge, that doesn't mean a good policy can be extracted.

There is tradeoff between the selection of decay factor and iteration efficiency. Higher decay factors (close to 1) ensures the reward at goal can propagate to the rest of the map, but requires higher number of iterations; lower decay factor (close to 0) allows iteration to converge much faster but might suffer from the missing gradient problem in Figure 2.2.



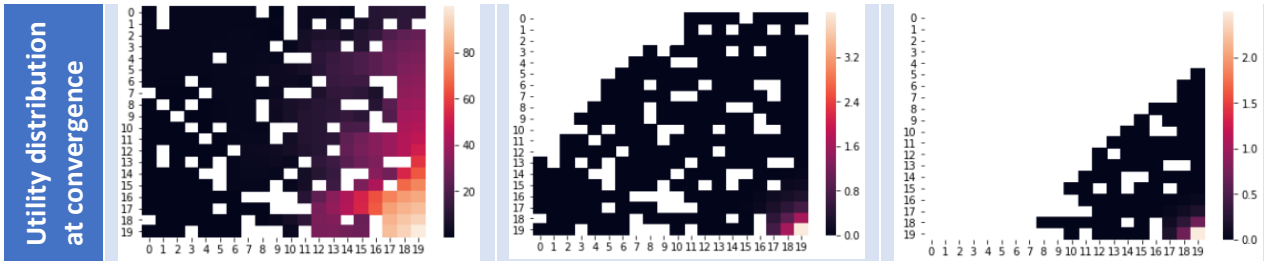


Figure 2.3 Utility difference vs. iteration and utility distribution at convergence, probabilistic action

Figure 2.3 shows the same value iteration result, but with probabilistic action environment. The utility difference curves are very similar to Figure 2.2, where lower decay factor corresponds to faster convergence. The value range on the heatmap is much larger in probabilistic action environment than deterministic. This is attributed to the Bellman equation solved in iteration. Since probability is a multiplier to utilities, a lower probability (for deterministic probability is 1, and for probabilistic probability is 1/3), a higher discount of the reward at goal is expected as it propagates to the rest of map.

A similar behavior is observed with lower decay factor, where values for the upper left states remain 0s. As discussed before, faster decay only allow reward at goal to propagate to part of the map. If a state does not ‘see’ reward at goal, no correct policy can be extracted to guide action.

## 2.3 Policy Iteration

Policy iteration computes utility, and uses utility to update policy at each state. It is considered converged if there is no policy change between two consecutive iterations. Decay factor is also the key parameter in policy iteration. Different decay factors are tested and results are shown in Figure 2.4.

The initial state of policy affects convergence speed as well. For Frozen Lake, if the initial policy is all ‘LEFT’, that is, every state the action is to go left, it takes many more iterations to converge. Setting initial state to be all ‘RIGHT’ facilitates convergence, and it becomes the default initial policy in this calculation.

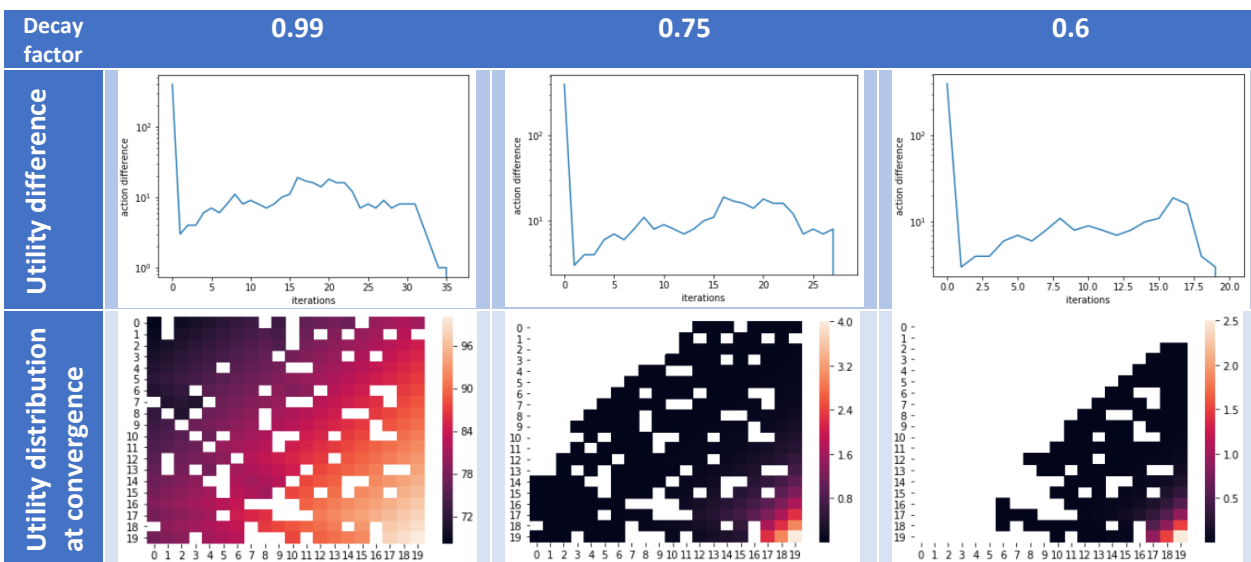


Figure 2.4 Policy difference vs. iteration and utility distribution at convergence, deterministic action

As decay factor decreases, Policy Iteration tends to converge faster, similar to Value Iteration.

The utility heatmaps at convergence for Policy Iteration show similar patterns as Value Iteration. At high decay factor, the reward at goal can propagate to the entire map, with an upward gradient from upper left (start) to lower right (goal). As decay factor decreases, the propagation range shrinks. Without a gradient, it is impossible to extract a correct policy.

The results for probabilistic action environment are shown in Figure 2.5. Action difference continues to decrease as number of iterations increases for all decay factors, which differs from the deterministic case in Figure 2.4. Because probability is a multiplier to update utilities, the gradient in heatmap in general is greater than that in Figure 2.4: for decay factor = 0.99 in Figure 2.5, the utility at start is close to 0 and at goal is close to 100, while utilities at the same states are 70 and 100 in Figure 2.4.

A similar missing value problem is observed in Figure 2.5 as well. With lower decay factor, reward at goal does not reach the upper left corner during iteration, and the initial policy is never updated. Policy extracted at convergence should therefore not be used as it is unreliable.

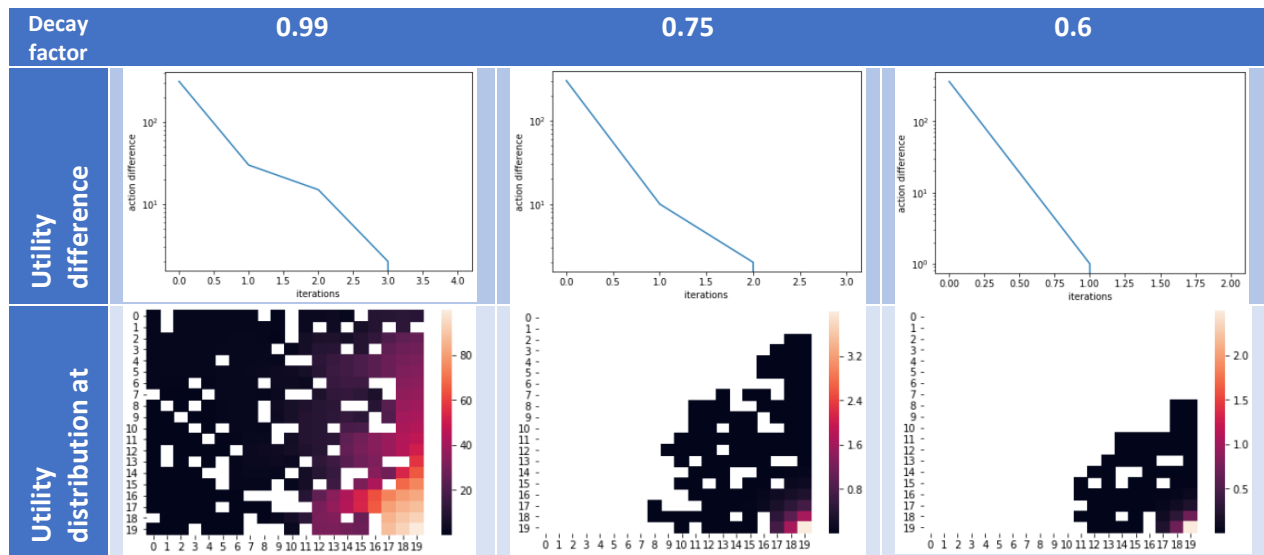
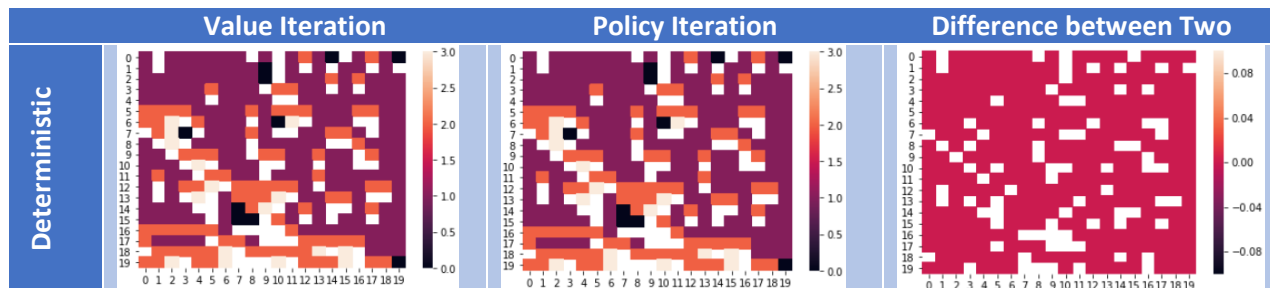


Figure 2.5 Policy difference vs. iteration and utility distribution at convergence, probabilistic action

## 2.4 Algorithm Comparison

The policy distributions are shown for both Value Iteration and Policy Iteration with decay factor = 0.99 in Figure 2.6.



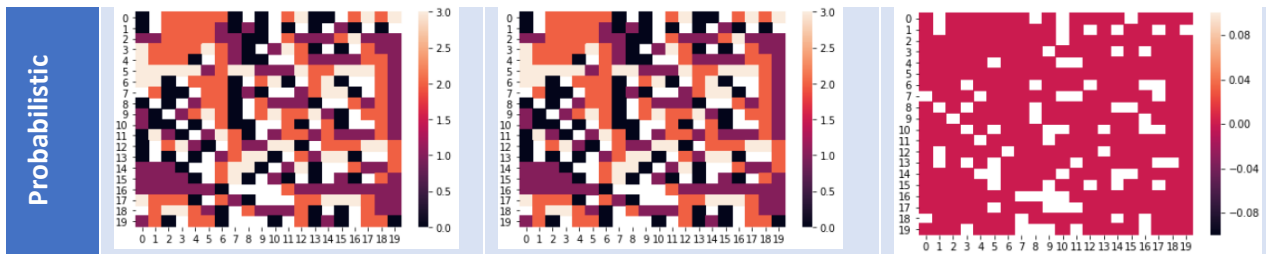


Figure 2.5 policy comparison between Value Iteration and Policy Iteration

Both algorithms manage to converge to the identical state after certain number of iterations, for both deterministic and probabilistic action environments. For deterministic action environment, most of the actions are 1 (DOWN) and 2 (RIGHT). Given that we need to move from the upper left corner to lower right corner, moving down and right makes logical sense. There are some cases where action is 3 (UP), which are states blocked by holes to their bottom, and 0 (LEFT), which are states blocked by holes to their right.

For probabilistic action environment it is more difficult to track the action for each state. But if we focus on states with holes to their left, the safest move for those states are moving right (with possibility of slipping up or down) rather than moving down (with possibility of slipping left and into the holes, and right). Most of states with that settings follow this logic, with some exceptions due to other influencing factors.

The convergence speed for 2 algorithms are tabulated below. Value Iteration uses much larger number of iterations to converge. However, every iteration only takes about 0.002 seconds. On the other hand, Policy Iteration uses much fewer number of iterations but every iteration takes much longer (~2 seconds). Value Iteration turns out to be a much more efficient algorithm than Policy Iteration, in terms of total time to converge.

		Number of iterations	Total time (s)	Time per iteration (s)
<b>Deterministic</b>	Value Iteration	1145	2.35	0.002
	Policy Iteration	37	86.00	2.324
<b>Probabilistic</b>	Value Iteration	1145	2.10	0.002
	Policy Iteration	5	10.15	2.03

The difference in algorithm performance roots in their individual ways to handle iteration. In Value Iteration, each iteration involves two steps: find utility and update policy. These two steps happens concurrently: once a new utility function is obtained, the policy at each state can be easily calculated by taking the highest utility of neighboring states.

Each step of Policy Iteration also has two steps: evaluate policy and improve policy. To evaluate policy it is required to iterate over the policy distribution until utility is converged, and then use the updated utility to improve policy. Therefore, there is a nested iteration in each iteration steps. At each iteration step, Policy Iteration can move towards the global optimal solution faster than Value Iteration. However, because of the nested iteration to evaluate policy, each iteration step for Policy Iteration takes much longer than Value Iteration.

### 3. Forest Management

For this problem, the focus is to analyze the effect of number of states to iteration, which includes computational time and number of iterations.

### 3.1 Sanity Check

Here two extreme cases are tested: 1). the probability of forest fire is high, therefore the logical choice would be to cut down the forest every year. 2) the probability of forest fire is low, therefore whether to keep or cut down the forest depends on the rewards for these two actions. For both evaluation, 10 states are used, which corresponds to 10 years.

parameters	policy
<b>R1: 4, R2: 2, P:0.99</b>	[0,1,1,1,1,1,1,1,1,0]
<b>R1: 4, R2: 2, P:0.1</b>	[0,0,0,0,0,0,0,0,0,0]

For the first case, the probability of forest fire is 99%. As a result, it is logical to cut the forest and get a reward of 2 on each state. The last state does not matter since we have already reached the last state. In the second case, the probability of forest fire is set to be 10%. Since it is unlikely to get fire, and also R1 is higher than R2, it makes sense for each state to keep the forest instead of cutting it.

### 3.2 Value Iteration

Several parameter combinations are tested for the experiments. Figure 3.1 shows the Value Iteration statistics and Figure 3.2 shows the Policy Iteration statistics, with the same parameter settings.

	R1	R2	P	Number of States	Number of Iterations	Total Time
0	2	4	0.05	500	349	0.035934
1	2	4	0.05	2000	349	0.924424
2	2	4	0.05	5000	349	5.295082
3	2	4	0.10	500	186	0.020090
4	2	4	0.10	2000	186	0.444574
5	2	4	0.10	5000	186	3.069998
6	2	4	0.30	500	58	0.005984
7	2	4	0.30	2000	58	0.149604
8	2	4	0.30	5000	58	0.883718
9	4	2	0.05	500	361	0.034913
10	4	2	0.05	2000	361	0.877880
11	4	2	0.05	5000	361	5.555610
12	4	2	0.10	500	192	0.015957
13	4	2	0.10	2000	192	0.479182
14	4	2	0.10	5000	192	2.956577
15	4	2	0.30	500	61	0.006982
16	4	2	0.30	2000	61	0.158577
17	4	2	0.30	5000	61	0.923644

Figure 3.1 Value Iteration statistics

	R1	R2	P	Number of States	Number of Iterations	Total Time
0	2	4	0.05	500	23	0.893415
1	2	4	0.05	2000	23	38.455994
2	2	4	0.05	5000	23	247.292471
3	2	4	0.10	500	12	0.512675
4	2	4	0.10	2000	12	19.664859
5	2	4	0.10	5000	12	121.436780
6	2	4	0.30	500	4	0.168548
7	2	4	0.30	2000	4	5.919407
8	2	4	0.30	5000	4	40.011176
9	4	2	0.05	500	34	1.286557
10	4	2	0.05	2000	34	62.770796
11	4	2	0.05	5000	34	373.290112
12	4	2	0.10	500	18	0.793026
13	4	2	0.10	2000	18	29.835315
14	4	2	0.10	5000	18	195.903975
15	4	2	0.30	500	6	0.200760
16	4	2	0.30	2000	6	9.992666
17	4	2	0.30	5000	6	60.037288

Figure 3.2 Policy Iteration statistics

All experiments converge to the correct policy. As can be seen from Figure 3.1, the variation in number of states does not affect the number of iterations required to converge. But each iteration takes longer, and therefore resulting in longer total time. The key equation that Value Iteration and Policy Iteration solves is Bellman Equation, which updates policy and utility for each state in every iteration. With larger number of states, the transition matrix is bigger, and requires longer time to calculate Bellman Equation. In fact, since the size of transition matrix is number of states<sup>2</sup>, the time required for computation almost exponentially increases with the size of states. Therefore even the number of iteration does not change, the total time required for computation still increases.

The number of iterations varies with different parameter combinations. However, all experiments can complete within very short period of time. This is much more efficient than Policy Iteration, which is discussed below.

### 3.3 Policy Iteration

Similarly, here I only focus on algorithm performance on different parameter combinations and number of states.

Compared to Value Iteration, to reach convergence Policy Iteration uses much fewer number of iterations. For any row on Figures 3.2 and 3.1, Value Iteration requires more than 10 times more iterations to converge. However, Policy Iteration takes much longer to converge. On average, it takes about one order of magnitude longer for Policy Iteration to converge. As a result, each iteration in Value Iteration is much faster than Policy Iteration, by several orders of magnitudes.

I also compared the final results for both algorithms. Given the same parameter settings and number of states, both algorithms converge to the same result. Given the great difference in algorithm efficiency, one might need to use Value Iteration for complex problems with lots of states. For both algorithms, time required to converge increases exponentially with the number of states, and therefore Policy Iteration is not suitable for complex problems.

## 4. Q-Learning

### 4.1 Frozen Lake

It turns out that Q-Learning really takes a long time to converge to the optimal solution. For instance, for an 8x8 frozen lake problem, it takes 1 million iterations. In order to finish the experiment within a reasonable time, here a simpler frozen lake (15x15) is used. Figure 4.1 shows the policy comparison between Q-Learning and Value Iteration.

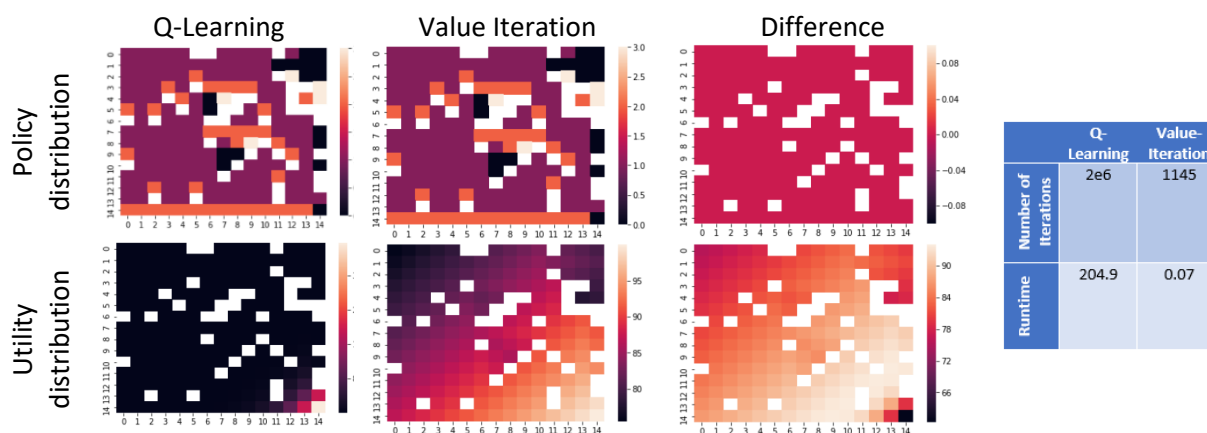


Figure 4.1 comparison between Q-Learning and Value Iteration

There is no difference in policy between two algorithms. Both algorithms converge to the correct policy. Utility distribution, on the other hand, show quite significant difference. The utility map for Value Iteration exhibits a much bigger gradient from start to goal than that of Q-Learning, suggesting that the reward at goal is not well populated throughout the map for Q-Learning. That explains why it takes so many iterations for Q-Learning to converge: as reward propagation is so slow, small number of iterations will be insufficient to establish correct utility distribution as to find the correct policy for each state.

The performance difference is quite significant. Value Iteration outperforms Q-Learning in both number of iterations and total runtime. It takes Q-Learning 2 million steps to converge (this is an arbitrary number to terminate iteration, as there is no good metrics to determine convergence. See next section for discussion), while it only takes Value Iteration 1145 steps. When it comes to total runtime, Q-Learning is about 3000 times slower than Value-Iteration.

There are three key parameters in Q-Learning to determine algorithm performance: decay factor, learning rate, and initial state. Decay factor has the same effect as in Value Iteration and Policy Iteration. Learning rate determines how fast the Q value is updated. Learning rate is between 0 and 1, and for each iteration,  $new\ Q = (1 - learning\ rate) \times$



$Q + \text{learning rate} \times \text{update}$ . For a deterministic environment such as this one, learning rate can be close to 1. Learning rate should be close to 1 for a probabilistic environment.

Initial state has the biggest effect on convergence for this experiment. Since the correct moving direction is from upper left corner (start) to lower right corner (goal), having an initial state of 1 (moving down) or 2 (moving up) would facilitate convergence. The result shown in Figure 4.1 is based on an initial state of all 1s for the entire map. If the initial state is 0, no convergence can be achieved in this experiment.

One other factor that could potentially affect Q-Learning convergence is the probability to explore vs. exploit at each step. In this assignment, the following equation is used for such purpose.

$$1 - \frac{1}{\ln(n+2)} \quad (1)$$

Where n is the number of iterations. For each state at each iteration, a random number is generated between 0 and 1. If the random number is larger than the above equation, a random action is taken, otherwise the action is determined by the largest Q value. As the number of iterations increases, the above equation is getting close to 1, and therefore less likely to explore (random selection) and more likely to exploit (follow Q value). Such setting is similar to Simulated Annealing, which aims for global optimum rather than local optimum.

Multiple other equations are also tested, for example,  $1 - \frac{1}{e^{n/m}}$ , where m is a normalization factor that controls how fast the equation is approaching 1. With all the equations tested, there is no significant difference in terms of algorithm performance. For this frozen lake problem, the key determinants are the initial state and learning rate.

## 4.2 Forest Management

Similar to Section 3, the focus for Forest Management using Q-Learning is to test algorithm performance. However, the exact number of iterations to converge is difficult to quantify in Q-Learning. Q-Learning tracks Q value updates at each step, which can be used to update utility and policy. On one hand, For Forest Management utility is always increasing. At small number of iterations, it is close to a sigmoid function, and becomes linearly increasing afterwards. Therefore, utility cannot be used to determine whether the computation converges. On the other hand, it is also difficult to determine convergence by tracking policy difference between two iterations just as Policy Iteration, since policy is not updated every step in Q-Learning. As a result, it is a trial-and-error process to find out the convergence of Q-Learning. In this assignment, only a few of such experiments can be finished and results are tabulated below.

R1	R2	P	Number of States	Q-Learning		Value-Iteration	
				Number of Iterations	Total Time (s)	Number of Iterations	Total Time (s)
20	2	0.001	50	6e6	141	52	0.057
20	2	0.001	100	10e6	231	105	0.133
20	2	0.001	200	25e6	562	221	0.31

All experiments are configured to have a much higher R1 than R2, and very low P1. This allows an easy check on the final policy: since the probability of fire is so low and the reward of keeping forest is so high, the policy for each state should be 0 (KEEP) rather than (CUT).

For all three experiments, Q-Learning is orders of magnitudes slower than Value-Iteration. Even though it manages to converge in all cases, the resources required to compute such a simple MDP problem is quite significant.

For both Frozen Lake and Forest Management problems, Q-Learning requires much longer time to converge than Value Iteration and Policy Iteration. Given Q-Learning is a famous algorithm in Reinforcement Learning, a plausible explanation for such low performance is that current Q-Learning implementation is not optimized. This is something to be improved in the future.

## 5. Summary

In this report two MDP problems (Frozen Lake and Forest Management) are used to evaluate 3 different algorithms: Value Iteration, Policy Iteration, and Q-Learning. While the first two algorithms are model-based, Q-Learning is model-less algorithm that is widely used in Reinforcement Learning.

### **Frozen Lake**

Frozen Lake is a 2D, 4 action grid world problem that can be easily visualized. In this assignment, both deterministic and probabilistic action environment are considered. Of all algorithms, Value Iteration takes least time to converge, but with relatively high number of iterations. Every iteration of Policy Iteration takes much longer than Value Iteration, but to converge, Policy Iteration uses the least number of iterations.

Q-Learning, on the other hand, is the worst performer of all algorithms. To converge a 15x15 Frozen Lake problem, it takes Q-Learning 2 million steps, with initiate state of all 1s (moving down) or 2s (moving right). Q-Learning is not able to converge with initial state of all 0s (moving left) or 3s (moving up).

### **Forest Management**

Forest management is a 1D, 2 action problem. In this assignment, many experiments with different algorithm parameter combinations can be performed thanks to relatively fast algorithm implementation.

Similar to Frozen Lake, Value Iteration can converge to the correct policy much faster than Policy Iteration, for all parameter combinations tested. Even though Value Iteration requires many more iterations for each experiment, runtime for each iteration takes much shorter time than Policy Iteration. Policy Iteration, on the other hand, requires much fewer number of iterations to converge, but its total runtime is orders of magnitudes longer than Value Iteration.

In terms of computation time, the best performer of three algorithms is Value Iteration, which is much faster than the other two. Policy Iteration, on the other hand, is a clear winner in terms of number of iterations required to converge. Q-Learning implementation in this assignment is quite rudimentary, and requires additional work to improve its performance.