Markov Decision Processes Report

Adrienne S. Miller and Anaïs C. Sarrazin

Computer Science Department, Bowdoin College, Brunswick, Maine, USA
{amiller5, acsarraz}@bowdoin.edu

1 Introduction

Individuals are faced with making decisions everyday that require choosing from a number of different actions. The best choice of action requires understanding the tradeoffs that need to be made between the clear immediate rewards and the uncertain future gains, in order to yield the best possible solution. Markov Decision Processes (MDPs) provide an approach to determining the optimal behavior in maximizing a model's ability to attain different states that define an environment.

To calculate the optimal behavior, MDPs consider the possible reward of taking different actions in sequences. The benefits of these choices can be measured by calculating the utility of each state given the potential for future rewards. The ultimate goal of this problem is to find the optimal policy for each state, or which policy yields the highest utility out of all possible choices. Several algorithms exist to solve an MDP but we will be focusing on describing and comparing our implementation of Value Iteration (VI) and Policy Iteration (PI). Both VI and PI rely on two primary functions, the reward function, $R(s)$, and transition function, $T(s, a, s')$, where s is the current state, a is an action, and s' is a next state. $R(s)$ measures the reward given upon entering a state, and $T(s, a, s')$ measures the probability of arriving in the state s' when action a is taken from state s. Value Iteration iteratively calculates the true utility of states, performing *Bellman Updates* on each state until the state utility values reaches equilibrium and converges. On the other hand, Policy Iteration iteratively performs two steps: *policy evaluation*, which calculates the utility values of each state given a current policy, and *policy improvement*, which updates the current policy by performing a single Bellman Update. Once there are no more changes to the policy, the algorithm terminates.

In Section 2, we first present a more complicated version of the well-known Grid World problem and environment in which an agent navigates the environment and chooses actions that maximize the final reward received. We then investigate and describe our implementation of the algorithms. In Section 3, we explore the performance of VI and PI through this Grid World problem. We analyze the experimental results of the solutions run on the problem and discuss what we found. These experiments test the comparative performance of value and policy iteration, and the impact of changing factors in the environment on the performance of value iteration and the optimal policy selected. In Section 4, we discuss what we could do in the future to improve our results. Finally,

we conclude with a section that provides an overview of the MDP problem and algorithms used to solve it, and the results of our experimentation.

2 Description of Algorithms

2.1 Overview of Problem

An agent operates in a fully observable environment where it chooses an action, a, from a set of actions, A, in each state s based on the *Maximum Expected Utility Principle*, which says that decisions should be made to maximize the expected utility. Since the effects of each state transition are uncertain, a transition function, $T(s, a, s')$, is used to map each state and action pair to a probability distribution over states. This is defined as $P(s'|s, a)$, which shows the probability of reaching the next state s', while the agent is in state s and the action a is taken. When moving states, an agent can either gain reward or "receive a punishment"(Majercik). The reward function $R(s)$ measures the reward given upon the agent's entry to a particular state, s. A policy function π maps states to actions and ultimately recommends an action for the current state $s : \pi(s) = a$ (Russell et al., 647). The expected utility $U[s, a]$ depends on the probability of taking an action a in state s, as well as the utility of possible next-states. The end goal of an MDP is to find the optimal policy π^*, that maximizes the expected utility in every state s, by using the state utility.

2.2 Value Iteration

2.2.1 How does it work?

In-Place Value Iteration can be used to find the true utility of each state, enabling us to make *greedy choices* to determine the optimal policy. The true utility of each state is determined by the *Bellman Equation*:

$$U(s) = R(s) + \gamma \, max \sum T(s, a, s')U(s')$$

where $U(s)$ is the true utility of a state, $R(s)$ is the reward upon entering that state, and $T(s, a, s')$ is the probability of entering s' when action a is taken from state s. VI begins by assigning all states an arbitrary utility value, usually 0. Then, VI performs *Bellman Updates* on each state, updating the optimal policy and estimated utility value for that state. One complete Bellman Update comprises of the following:

- For each state, s, in the grid-world, each possible next action, a, is evaluated. This is done through the calculation of a weighted probability for each action. The weighted probability for each action is essentially the average reward expected across all possible next states when a given action is taken, weighted to account for the probability that the agent will arrive in any given s'. Then, the action that has the maximum weighted probability is chosen as

the current policy for that state. The maximum weighted probability can be seen in the Bellman equations as follows: $max \sum T(s, a, s')U(s')$. This value, along with the discount factor and reward function is also used to update the utility value for s.

As we can see, each Bellman Update results in an improvement upon the policy for each state, and a more accurate utility value for each state. So, to find the true utility values of each state, we run Bellman updates until the equation converges, and we reach a point where the maximum error is small enough. This is done by calculating the maximum change in utility in a single update, across all states. While this maximum change in utility is greater than $\varepsilon((1-\gamma)/\gamma)$, where ε is the maximum error and γ is the discount factor, we continue to perform Bellman updates. Once the maximum change in utility has passed this threshold, we know that the policy is optimal, and the maximum *loss* in utility from not continuing to make Bellman updates is equal to $2\varepsilon(\gamma/(1-\gamma))$.

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Fig. 1: Pseudocode for Value Iteration

2.2.2 Our Implementation?

Our implementation of the VI algorithm directly follows the above pseudocode, iteratively improving the current policy by running Bellman updates until the change in utility from one iteration to the next was small enough to ensure a maximum error. The parameters of this method included a 3D array holding the transition matrix created in the `TransitionFunction` class, an array holding the reward values collected from the `RewardFunction` class, a discount factor provided by the user, and a maximum allowable error, also specified by the user.

First, we initialized a policy array with arbitrary values. From there, we enter a while loop, performing Bellman Updates until the maximum error is as small as specified by the user. One Bellman update consists of the iteration through all 65 states. For each state, the weighted probability of each particular action,

given the utilities of possible next-states, is calculated. The action that results in the highest probability is taken as the current optimal policy, and is used to update the policy array and the state utility array. One Bellman update takes the form of three nested for loops, iterating over all possible current states, all possible actions, and all possible next states. Once the maximum change in utility across all states from one iteration to the next reaches a minimum threshold, VI terminates.

2.3 Policy Iteration

2.3.1 How does it work?

As opposed to finding the optimal value of the function like in VI, the Policy Iteration algorithm iteratively improves the current policy by alternating between evaluating the current policy and improving the policy as such:

- **Policy Evaluation:** Calculate the utility, $U_i = U^{\pi_i}$, where π_i is a given policy executed.
- **Policy Improvement:** Perform a single Bellman Update by iterating through all states. Determines the optimal policy implied by the utilities calculated in Policy Evaluation.

PI begins with an arbitrary policy π_0 and seeks to find the utility of π in each state s during Policy Evaluation: $U_i = U^{\pi_i}$, where the policy π_i designates the action $\pi_i(s)$ in state s at the ith iteration. Since policy implies a relationship among state utilities, then we know that if we have n states, then we have n equations with n unknowns(Majercik). By using a simplified version of Bellman Equation and the current policy, we can go through the *Policy Evaluation* step by setting up a system of linear equations and solving each utility using the following equation:

$$U_i(s) = R(s) + \gamma \sum T(s, a, s')U_i(s')$$

The main difference between this and the Bellman Equation used in VI is that the *max* operator has been removed, which shows that these equations are linear. This system of equations is solved to find the current utilities of each state, given the current policy.

Then, for *Policy Improvement*, we determine whether or not the value could be improved by changing the first action that was taken. The process of Policy Improvement is essentially performing a single Bellman Update on each state. For each state, s, we calculate the weighted probability for each action, and determine the most advantageous action for that state, given current utility values. We choose to improve or not improve based on the utility values that are calculated:

$$\pi_0 \rightarrow U^{\pi_0} \rightarrow \pi_1 \rightarrow U^{\pi_1} \rightarrow ... \rightarrow \pi_* \rightarrow U^{\pi_*}$$

If the value can be improved, then we take a note that this policy should take this action when found in this given situation. This guarantees improvements in

the performance of the policy. After improving the policy, we return to Policy Evaluation, solving again for the current utility values given the new policy. We continue this process of Policy Evaluation and Improvement until the policy can no longer be improved. If the policy can't be improved, then the policy is guaranteed to be optimal since a finite number of policies guarantees a convergence. Each iteration also guarantees a better policy until you hit equilibrium, which is when the policy improvement step stops producing changes in the utilities. As a result, when the algorithm terminates, we know we have found the optimal policy at π_i. We can see the pseudocode for the PI algorithm below:

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ **true**
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
        $unchanged? \leftarrow$ **false**
  until $unchanged?$
  return π

Fig. 2: Pseudocode for Policy Iteration

So, for n states, we have n linear equations with n unknown values. Thus, if we are solving this using linear algebra, then PI can be solved in $O(n^3)$.

2.3.2 Our Implementation?

Our implementation of the policy iteration algorithm directly follows the above pseudocode, iteratively improving the current policy using a round of Bellman update and returning an array holding the policy values for each state. The parameters of this method included a 3D array holding the transition matrix created in the `TransitionFunction` class, an array holding the reward values collected from the `RewardFunction` class, and a discount factor provided by the user.

First, we initialized a policy array with arbitrary values and a rewards matrix. Then, we entered into a while loop that iterates until the given policy has changed in the Bellman update. Within the while loop, a transition matrix is created and a utility matrix is solved and converted into array form. By using this array of utility values, we then ran one round of Bellman updates and calculated the weighted probabilities. In our Bellman update, we held a `max` variable that would be compared to the `weightProb` to update the policy to what the Bellman

update would suggest(Policy Improvement). The method ran until the policy no longer changed during the policy improvement phase of the algorithm.

3 Experimental Results

Optimal policies can be found using a variety of different solution techniques. We investigated these techniques by considering a more complicated version of the simple Grid World problem presented in Russell et al.'s Textbook (2009). The environment is described by a grid shown in Figure 3.

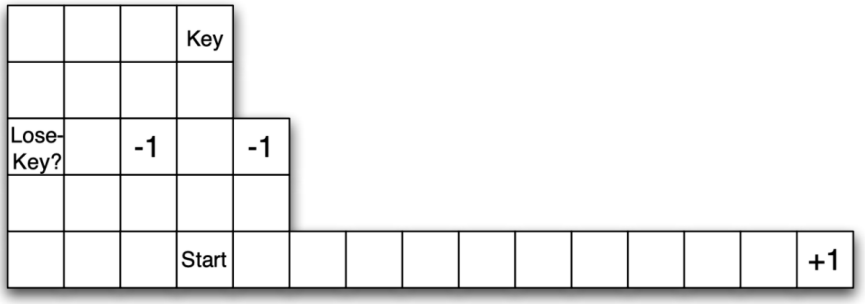

Fig. 3: The Environment Exhibited in the Problem

The agent always starts from the `Start` square and moves from one square to another until it reaches a terminal state. In each square, the agent has four available actions that can move the agent one square in the intended direction with a probability of 0.8: `GoNorth, Go-East, Go-South, Go-West`. Otherwise, the action goes in a direction 90 degree clockwise or counterclockwise with a probability of 0.1. For example, if the agent chooses to `GoNorth`, there is a probability of 0.8 that the agent will move north, and a probability of 0.1 that the agent will move east or west. If an agent's action takes the agent into a wall, the agent doesn't move, but the cost associated with a step is still incurred.

There are five *special* squares: the key square (`Key`), the key-loss square (`Lose-Key`), the positive terminal state (`+1`), and the two negative terminal states (`-1`). If the agent reaches the key square, it obtains the key necessary to get positive reward in a terminal state. If the agent enters the key-loss square, it will lose the key with a probability specified by the user. Once an agent enters a terminal state, it cannot move to any other state. The two negative terminal states incur a negative reward, specified by the user, on the agent. If the agent enters the positive terminal state, it will receive a positive reward, again specified by the user, but only if the agent has obtained the key. If the agent enters the positive terminal state with no key, the reward is 0. All states with the exception of the three terminal states have a reward upon entry, which can be viewed as a step cost. The states defined in this problem denote the location of the agent as well as whether or not the agent has a key. So, for every location, with the exception of the Key square, there are two available states in which that square

is the agent's location, one state in which the agent has a key and one state in which it does not (Majercik, Part 1 Problem). Because the agent always receives the key upon entering the key-square, there is no possible state in which the agent is in that square, but does not have a key.

In our experiment, we varied the values of the parameters discount factor, key-loss probability, step-cost, positive reward, and negative reward. We tested the impact of changing these values on the number of iterations taken to complete value iteration and the impact on optimal policy.

3.1 Baseline Values for Experiments

Unless otherwise specified, the default values for these experiments are as followed:

- Discount Factor = 0.999999
- Maximum Allowable Error = 1E-6
- Key-Loss Probability = 0.5
- Step-Cost = -0.04
- Positive Reward = 1.0
- Negative Reward = -1.0

3.2 Differences in Value Iteration and Policy Iteration

When our experiments were run with the default values discussed in Section 3.1, VI completes 64 iterations, while PI completes only 8. However, value iteration takes approximately 10 ms to complete, while policy iteration takes 52 ms. For each iteration in policy iteration, it gets closer to the optimal policy (relative to value iteration). However, it takes much longer to complete a single iteration in PI as for each iteration, a system of linear equations must be solved, which takes $O(n^3)$.

	Value Iteration	**Policy Iteration**
Number of Iterations	64	8
Time Elapsed (ms)	10	52

Table 1: Time Differences between Policy Iteration and Value Iteration

When only run until an optimal policy is reached (instead of until a maximum error in utility is reached), value and policy iteration take 31 and 7 iterations, respectively. Value iteration takes 64 iterations to complete, despite reaching the optimal policy at 31 iterations, because although an optimal policy is reached at that point, the error in utility for any state will still be above the maximum threshold at that point. Although policy iteration technically takes 8 iterations to complete, it reaches an optimal policy on the 7th iteration. Policy iteration terminates once the policy from one iteration to the next is unchanged. Thus, the policy after the 7th and 8th iterations will both be optimal, even though policy iteration does not terminate until after the 8th iteration is completed.

3.3 Impact of Changing Parameters on Value Iteration

3.3.1 Effect of Changing Discount Factor

Below, the table shows the effect of changing the discount factor on the number of iterations taken to complete value iteration.

Discount Factor	0.1	0.4	0.7	1.0
Number of Iterations	6	14	26	79

Table 2: Results of Different Step Cost Values

The change in discount factor has a significant impact on the optimal policy created by value iteration. With a low discount factor, such as 0.1, there seems to be no reason to the policy selected. The agent does not avoid negative rewards or seek out positive rewards, or even try to just reach a terminal state. This change in policy makes sense for the value; when rewards are so heavily discounted, it essentially does not matter what choice is made by the agent. No matter what actions are taken, the likelihood of a large positive or negative reward is highly unlikely. As the discount factor increases, the policy takes the positive and negative rewards into greater consideration.

With a discount factor of 0.4, the agent attempts to avoid the negative reward squares. However, the agent will not seek out the key, but just try to reach the positive terminal state. When the discount factor is increased to 0.7, the agent will seek out the key, but only if it is in a state relatively close to the key. The key-loss probability square is not avoided by the agent. When the discount factor is increased to 1, the policy given is virtually the same is the "default policy." As a general rule, when the discount factor increases, the agent "cares" more about positive and negative rewards because they will have more impact on them in the long run.

3.3.2 Effect of Changing Step Cost

Below, the table shows the effect of changing the step cost factor on the number of iterations taken to complete value iteration.

Step Cost	-10.0	-1.0	-0.01	0.0	0.01
Number of Iterations	2	31	286	297	11,354,463

Table 3: Results of Different Step Cost Values

When the step cost is as high as -10.0, the policy created does not make a lot of sense. While one might expect the agent to try to reach any terminal state as fast as possible, the agent just seems to pick default moves. It would make more sense for the agent to seek out any terminal state, but it's possible that every single policy is terrible when the step cost is so high. This coincides with the

low number of iterations for this test; value iteration terminates once the change from one policy to another is small enough. Thus, if there is no improvement from one policy to the next, value iteration will terminate, even if the current policy does not seem logical.

When the step cost is decreased to -1.0, the agent's only goal is to reach a terminal state as fast as possible. This policy makes intuitive sense; with a step cost as high as the maximum positive reward, reaching the positive terminal state is useless. Unlike the policy when the step cost is -10.0, the step cost is low enough that the policy chosen will have an impact on the expected utility. However, the step cost is still high enough that it's not worth it to seek out the positive terminal state.

With a step cost of -0.01, the policy is very similar to the default policy. The agent will attempt to get the key and reach the terminal. The agent avoids the negative terminal state, and is willing to go through the key-loss square and risk having to backtrack for the key. Since the step-cost is so low, the agent would rather take a significant number of steps than risk a high negative reward. The policy when the step cost is 0 is essentially the same as the above policy; the agent will always seek out the key, and avoid the negative terminal state at all costs.

With a positive step cost, the agent attempts to take as many steps as possible. This strategy includes avoiding all terminal states, and going towards a wall when possible. The agent does not care about the key, likely because if the agent reaches a terminal state, the reward will be so heavily discounted that it will essentially be 0. The general trend for changes in policy is that as the step cost decreases (becomes more positive), the agent becomes more risk averse. With a higher step cost, the agent will risk, or even seek out, a negative terminal state if it means less steps are taken. When step cost is low, or even positive, the agent will avoid negative states and all costs, and even take more steps in pursuit of a higher positive reward.

3.3.3 Effect of Changing Negative Reward

Below, the table shows the effect of changing the negative reward on the number of iterations taken to complete value iteration.

Negative Reward	-100.0	-10.0	-1
Number of Iterations	71	71	64

Table 4: Results of Different Negative Reward Values

The policy resulting from changing the negative reward to either -10.0 or -100.0 are effectively the same. Both these negative rewards are high enough that the agent's principle goal is to avoid the negative terminal states at all costs. The agent will still seek out the key in hopes of gaining a positive reward, but will go through the key-loss square to avoid negative terminal states. When the negative reward is -1.0, the baseline policy described above holds.

3.3.4 Effect of Changing Positive Reward

Below, the table shows the effect of changing the step cost factor on the number of iterations taken to complete value iteration.

Positive Reward	1.0	10.0	100.0
Number of Iterations	64	311	341

Table 5: Results of Different Positive Reward Values

The policy resulting from changing the positive reward to 10 or 100 is very similar. The agent will seek out the key no matter how far it is from the key square, and will avoid the negative terminal state at all costs, including going through the key loss square. Given the potential for such a high positive reward, step cost has little impact on the choices of the agent. When the reward is 1.0, the baseline policy holds.

3.3.5 Effect of Changing Key-Loss Probability

The below table shows the effect of changing the step cost factor on the number of iterations taken to complete value iteration.

Key-Loss Probability	0.0	0.5	1.0
Number of Iterations	65	64	65

Table 6: Results of Different Key-Loss Probability Values

The effect of changing key-loss probability has the least dramatic impact on policy out of all of these factors. The only impact this factor has on the baseline policy described above is whether the agent will choose to avoid the key-loss square and risk entering a negative terminal state, or go through the square. When the key-loss probability is 0, the agent goes through the key-loss square, avoiding the terminal state. When the key-loss probability is 1, the agent avoids the key-loss square, but there are no other notable changes in policy. In the baseline policy (key-loss probability is 0.5), the agent will risk entering both the key-loss square and the negative terminal state, but will not seek out either.

4 Further Work

To begin expanding on this work, it would be interesting to experiment with more algorithms for computing an optimal policy. To improve overall efficiencies in the algorithms that we had implemented, looking at modified versions of these algorithms, such as asynchronous value iteration and modified policy iteration, would certainly have been interesting to compare with the basic policy and value iteration algorithms. Additionally, we could have finished implementing our reinforcement learning algorithm, Q-Learning, which solved for MDP's by

learning how to interact with the environment, as opposed to using a transition function and reward function.

Another way in which we could expand this work is by testing the effects of different factors in greater depth by increasing the range of values used in experimentation, and testing policy iteration and reinforcement learning as well as value iteration. This could give us further insight to the strengths and weaknesses of the different methods to solving MDPs, and show in which conditions one algorithm may operate better than another. Although we gave a general overview of how changes in different parameters affected the optimal policy created, a more in depth analysis of the changes in policy would help us better understand both the grid world and the algorithms used to solve the MDP.

5 Conclusion

Uncertainty is highly prevalent in the everyday decisions we make. The MDP problem provides us with a method to analyze potential actions and consider future risks and possibilities for reward. Through the solution techniques, Policy Iteration and Value Iteration, we are able to solve the MDP problem, and determine an optimal policy for a given environment. VI iteratively improves the through Bellman Updates, updating utility values and greedily choosing optimal policy until the utility values converge. On the other hand, PI works in two phases, first evaluating the current policy by calculating true utility values for each state, then greedily improving the policy using the calculated utility values. This process is guaranteed to improve the policy on each iteration, and continues until the policy remains unchanged from one iteration to the next.

In our experimentation, we tested the comparative run time of VI and PI, and the impact that changing relevant factors in the environment had the performance of VI, and the optimal policy it ultimately produced. These tests showed that while PI takes only 8 iterations in comparison to VI's 64, the cost of policy evaluation in PI is great enough that VI runs considerably faster. We also discovered that while VI takes 31 iterations to reach optimal policy (but 64 to reach the minimum error), PI terminates upon reaching the optimal policy. In our tests on VI, we investigated the impact of discount factor, step cost, negative reward, positive reward, and key-loss probability. When the discount factor is very low, the rewards are so heavily discounted that the policy calculated effectively doesn't matter. As the discount factor increases, the agent becomes more aware of positive and negative rewards, and will make choices to optimize the final reward. Our tests on step-cost showed that with a very high step cost, the only goal of the agent was to reach a terminal step cost. As the step-cost decreases, and even becomes positive, the agent will become very risk averse, always seeking out the positive reward. Likewise, when negative and positive rewards are higher, the agent will avoid negative terminal states at all cost, in hopes of not incurring a large negative reward or obtaining the large positive reward. The last factor we tested, key-loss probability, had the smallest impact on policy. Changing this factor only impacts the way the agent interacts with

the key-loss square, and whether the agent will risk entering a negative terminal state to avoid losing the key. As we continue to be faced with uncertain decisions in our everyday life, it would be interesting to expand on this research by exploring other MDP methods and experiments that will make decision-making less stress-inducing.

6 Bibliography

Russel, R. J., Norvig, P., Davis, E. (2009). Artificial Intelligence: A Modern Approach. 3rd Edition Prentice Hall.