Markov Decision Processes: Policy Optimization Utilizing Value and Policy Iteration

Chad Carrera and Jake Adicoff
CSCI 3420: Optimization and Uncertainty
Bowdoin College, Brunswick, Maine

Keywords: utility; value iteration; policy iteration; optimal policy;

Abstract: In a Markov Decision Process (MDP), there is an agent that is making decisions in a probabilistically uncertain environment. Referred to as the grid-world, the environment is navigated by the agent according to the maximum expected utility principle, giving the solution to the MDP, called an optimal policy. We generate this policy by implementing one of two solution techniques, called Value Iteration and Policy Iteration. We describe *Value Iteration* (VI) as an algorithm that, given an MDP, will iterate through each initial state, action, and then sum over the transition states to calculate the maximum expected utility. *Policy Iteration* (PI) is an algorithm that, when given an MDP, will initially try to improve current policy by setting up and solving a linear matrix equation with n equations and n unknowns. A similar summation to the one in VI is then performed, however this summation over transition states is compared to a summation over the transition states using the current policy, and the policy is updated accordingly. In experimentation, the goal of the tests was to show not only the differences between the two solution methods, but also what effects the parameters given to the MDP will have on the agent and the calculation of the optimal policy.

1. Introduction

A Markov Decision Process (MDP) is a model that is used for decision-making in an environment where future states are probabilistically uncertain. The agent navigates the environment, by making a selection of actions, of which there are four: Go-North (Up), Go-South (Down), Go-East (Right), and Go-West (Left), each with equal uncertainty in their respective outcomes.

Because of the uncertainty, the rational agent will aim to maximize the expected utility that it will receive given its current state, the action, and the states that the agent can transition to. This is called the maximum expected utility principle. When combined with a MDP problem, we are able to operate in an environment where the agent has the opportunity to gain rewards (positive utility), and punishments (negative utility). The resulting set of actions for all of the states is a solution to the MDP called a policy. There are many different policies, however, and the goal when finding one is optimality. Since the rational agent will greedily make selections through application of the maximum expected utility principle, the end result is what is called the optimal policy. The optimal policy for the MDP is calculated using one of two algorithms: Value Iteration (VI) or Policy Iteration (PI).

In VI, the algorithm must be given an MDP with states S, actions A(s), the transition model $T(s' \mid a, s)$, rewards $R(s)$, discount factor γ, and the maximum error allowed in the utility of any state ϵ. With these parameters, VI will iterate through each initial state, action, and then each transition state, summing the maximum expected utility for the current state. By calculating new utilities for all states with each iteration, VI is synchronous, and upon using these new utilities as they become available via Bellman backups, it is in-place as well.

PI differs quite a bit from VI, because a very direct method of solving the MDP is used in PI. The parameters necessary for execution are an MDP with states S, actions A(s), and a transition function $T(s' \mid a, s)$. The very first calculation in PI is an evaluation of the current policy using a linear matrix equation, with n equations and n unknowns. The coefficients for each state's utility are put into a matrix A, and the constants, which are all of the reward values for each state, are put into matrix B. The equation Ax = B is then solved, and the utilities for all states are updated accordingly. Then, PI iterates through each initial state, action, and transition state, and sums the maximum utility given an action. This is then compared with utilities calculated using the linear equation and the current policy for that state, and the policy is updated accordingly. The algorithm will execute until no changes are made to the policy, meaning it has been fully optimized.

For experimentation, there were a total of 5 main tests that were completed. The implicit goal of the first three experiments was to show the fundamental differences between the implementation of the two algorithms. These differences were displayed by recording and analyzing the number of iterations that both VI and PI take to solve the grid-world MDP problem, the time that either algorithm took to do this, as well as the **actual** number of iterations that were needed by each algorithm to generate the optimal policy of the agent.

The last two experiments narrowed the scope a little, and focused on the manipulation of a single parameter at a time, and the effect that it has on the overall calculation of the optimal policy. These tests differ slightly in the fact that they are executed only on the VI algorithm. Data collected from each combines to give a picture as to how each parameter shapes the amount of iterations VI takes to solve the MDP, as well as what the agent does given these manipulations.

In Section 2, a more formal description of a MDP will be given, and both the Value Iteration and Policy Iteration algorithms will be presented in detail. Section 3 is an overview of the experimental process that was executed for the algorithms. The results of these experiments will be explored and displayed in Section 4. Potential for further work in MDPs, VI, and PI will be outlined in Section 5, and the summarization of our work done with MDPs will be discussed in the last portion, Section 6.

2. Description of Algorithms

To analyze the differences in running time and number of iterations to reach convergence given certain parameters, we implemented two algorithms, Value Iteration and Policy Iteration, to solve the MDP.

2.1 The Markov Decision Process (MDP)

Before understanding the algorithms used to solve the MDP, we must go over the nature of an MDP. An MDP starts with an agent in an environment that has sets of states and associated actions and rewards. As previously stated, to solve an MDP is to find a policy, and the optimal action in each state, to maximize the rewards of the agent. The structure is as follows:

- First there is a set of states *S*, Each state has an associated reward, and there is a rewards function *R(s)* for all *s* that gives the reward for being in state *s*.
- There is a set of actions *A* that can be taken in each state (although, for our test case, all 4 previously defined actions can be taken form any state).
- There is a state transition probability function defined as *T(s' | a, s)*. this function gives the probability that state *s'* is reached by taking action *a* from state *s*.
- There is a policy vector $\pi(s)$ that holds the current policy of the MDP. This vector holds a state with a single associated action.
- There are two special parameters, a discount factor, γ, and a maximum state utility error, ε. The purpose of these will be discussed in each algorithm.

As stated before, the policy vector is what we aim to optimize in solving an MDP problem. With an optimal policy, the agent knows which action to take from each state that will maximize the total rewards the agent will receive in the environment. In Value Iteration and Policy Iteration we have two different strategies for finding the optimal policy vector for the MDP.

2.2 Value Iteration (VI)

Value Iteration in its most basic form updates a state utility vector with each iteration of the algorithm until the values in that state utility vector converge. It is within each iteration that the important computation occurs. Essentially, in one iteration, we iterate through each state in the state set and update its state utility. To do this update of a state utility, we find the optimal action, *a,* given the current state utilities. To do this, we take the maximum sum,

$$\max(a\sum T(s' \mid a, s)U(s'))$$

for all actions that are legal to take from state *s*. After computing this maximum sum, we can make an update to the utility vector using the sum in the Bellman Equation:

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

You can see where the equation for the maximum action summed over possible state transitions comes into this equation. The γ in the equation is the discount factor of the MDP. It works to discount the rewards in neighboring states and discounts the rewards more for states that are farther from the current state.

For each iteration, we compute a new state utility for each state and update it in the utility vector. After this update we check to see if the state utility vector values have converged and if convergence is reached, value iteration terminates and we are left with the optimal policy for the MDP. Convergence is measured by first finding the maximum change in the utility vectors *U* and *U'*, where *U* is the utility vector before we update state utilities in an iteration and *U'* is the vector after the update. Finding the maximum change in the vector is to say that we look for the single state utility that has the largest absolute change from *U(s) to U'(s)*. We write this difference as $\|U - U'\|$. We then check to if this value has fallen below the specified error bound error bound using ε (ε specifies a max state utility error). To calculate a proper bound we calculate $\varepsilon(1-\gamma)/\gamma$. Now we look at the conditional:

$$\varepsilon(1-\gamma)/\gamma > \|U - U'\|$$

When this evaluates to true, we deem any further changes to the state utility vector unnecessary because we have essentially reached convergence in the utility vector. At this point, VI terminates, and all that is left is policy extraction. The utility vector is used to construct the optimal policy vector, which we do by choosing greedily the action in each state that maximizes the utility gained by taking that action. To do this we use the same maximum sum as before, but we store the action *a* in our policy vector. This looks like:

$$\pi(s) = \text{argument}(\max(a\sum T(s' \mid a, s)U(s')))$$

This guarantees the optimal policy for the agent in the environment and the MDP is solved.

2.3 Policy Iteration (PI)

Policy iteration is a two-step process in which we first set up a matrix equation with n equations and n unknowns and solve it to update a state utility vector *U*. This is referred to as policy evaluation. After this we test whether the policy is optimal. If the policy is not optimal, the policy vector, π, is updated, and the process is repeated until the policy vector does not change throughout an iteration of the algorithm.

To set up the equation, we again look to the bellman equation, although unlike in VI, we do not need to find the maximum action utility. We use a slightly altered version of the equation that sums over the state transition possibilities given the specified policy, so we are only calculating the sum for a single action from every state. The equation is as follows:

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

You can see that, unlike in value iteration, we are using the current policy (which is initially random) to set up the equation. Also, for this equation, we do not use the current utility vector in calculations and utilities are the unknowns in the linear matrix. We rearrange this Bellman equation to have the constants or rewards *R(s)* set equal to the linear equation. This rearranged equation takes the form:

$$R(s) = U(s) - \gamma(a\sum T(s' \mid \pi(s), s)U(s'))$$

Now we have a constant reward and coefficients of *U(s)* and *U(s'),* which are 1 and $-\gamma(a\sum T(s' \mid \pi(s), s)$, respectively. We iterate through every state, setting up the linear equation for each state and putting it into the matrix, call it A. We use the rewards function to create a vector, call it B, and we solve the equation Ax = B for x. The array x now holds all the state utilities given a specified policy, so we set *U* = x, and step one of the algorithm is complete.

Now after extracting the state utilities given a policy, we check to see if the policy is optimal. A policy is optimal when the policy vector π(*s)* does not change for any *s*. To do this, we iterate through all states and for each state we find the action that is optimal with the same method that is used VI. The equation is again:

$$\max(a\sum T(s' \mid a, s)U(s'))$$

We then make a comparison to the utility of the action specified by the current policy. The comparison is as follows:

$$\sum T(s' \mid \pi(s), s)U(s') < \max(a\sum T(s' \mid a, s)U(s'))$$

If this conditional evaluates to true we must update the policy vector with:

$$\pi(s) = \text{argument}(\max(a\sum T(s' \mid a, s)U(s')))$$

We make all necessary updates to the policy vector then, with the new policy, another iteration of PI is executed.

If the previous conditional evaluates to false, we do not update the policy, and if that conditional evaluates to false for every state, we know that the policy vector has not changed throughout the iteration and thus the policy has converged. We know that the policy has converged because there will only be strict improvements to the policy in each iteration of the algorithm. When the policy converges, the algorithm breaks and we are left with the optimal policy.

3. Experimental Methodology

There were a total of five primary experiments that were run on the VI and PI algorithms and the MDP grid-world problem. The following will be referred to as the **default parameters** for all of the experiments, which are as follows:

```
java MDP 0.999999 1e-6 0.5 1 -1 -0.04 t
```

These arguments execute the MDP program with the grid-world problem, and specify a discount rate of 0.999999, a maximum state utility error of 0.000001, a key loss probability of 0.5, a positive terminal state reward of +1, a negative terminal state reward of -1, a step cost of -0.04, and t represents the chosen technique, **either v** (Value Iteration) **or p** (Policy Iteration). If any of the parameters were changed in the carrying out of a particular experiment, it will be noted in the corresponding paragraph with the description of the experiment. Otherwise, assume that all of the values are the same as the default arguments (aside from the listed solution method).

Experiment 1 is a simple test, and is just a comparison between the recorded numbers of iterations that VI and PI need for the optimal policy to be generated using the default parameters. This test only needed to be run once, as the number of iterations will always be the same given the same parameters. To do this, we simply run the program, and the iteration counter within each of the algorithms will print out the number of iterations upon completion of the algorithm. This experiment serves to offer a comparison between the solution techniques implemented in each algorithm, highlighting the indirect nature of VI, and the very direct solution method of PI. This is important to note due to the implications that this has for the data collected from both Experiments 2 and 3.

Experiment 2 is another basic comparison between VI and PI, however this time the focus is on the amount of time that each algorithm takes to find the optimal policy using the default parameters. Unlike Experiment 1, however, it was necessary here to run the same experiment multiple times, because there were pretty substantial fluctuations in the amount of time taken between any given pair of executions of the algorithm. Because of this, ten different times were recorded for the execution of both VI and PI, and these were averaged to generate the time that either algorithm took to solve the problem. The times were gathered by simply executing the default parameters and specified algorithm ten times. The time was printed out with the statistics from the given execution, and was then recorded. This experiment shows that although PI takes a fewer number of iterations than VI, it is not necessarily better (which could potentially be a conclusion drawn from Experiment 1). Experiment 2 shows that the time is an extremely important factor to be considered when choosing a method to solve a MDP. By timing PI, the experiment highlights the time inefficiency of having to solve a linear equation with every iteration of the algorithm.

Experiment 3 is another comparison between the two solution algorithms. Now, we are focused on the number of iterations that VI or PI **actually** needs to find the optimal policy, using the default parameters. This test only needs to be run once, as the number of iterations will not change given the same parameters. To do this, we simply modified our code to make a clone of the policy vector in the beginning of each iteration and test at the end of each iteration, after the policy has the opportunity to be updated, to see if the new policy and the old policy were the same. When the policy vector and the policy vector clone were the same, we printed the current number of iterations which was computed within the algorithm execution. The final number to be printed would reveal the number of iterations of each algorithm **actually** needed to find an optimal policy. This test is important to understanding VI in that it displays a discrepancy in the number of iterations required for the utility vector and the policy vector to converge. This part of the code is removed in our final code as it has potential to slow down execution times of the algorithms unevenly.

Experiment 4 is the first experiment where there are moving parts. In this experiment alone, there are five distinct parameters that will be manipulated and tested:

a) discount factor
b) step cost
c) negative terminal reward
d) positive terminal reward
e) key loss probability

However, there will only be one parameter manipulated at a time. For instance, provided I am changing the negative terminal reward, the rest of the parameters will remain the same as they are in the default parameters, such as the following:

```
java MDP 0.999999 1e-6 0.5 1 -50 v
```

As any one parameter is changed, the number of iterations that VI takes to find the optimal policy will be recorded. Prior to discussing the experiment, it must also be mentioned that the only algorithm that Experiment 4 will be performed on is VI. It is also worth noting that the tests run in Experiment 4, and the data recorded, will relate very closely with those of Experiment 5.

For part (a) of Experiment 4, the discount factor was the parameter that was being manipulated. The value was modified so that we can record what the effect on the number of iterations is when the future rewards are valued at varying levels relative to the current rewards. Initially, the first two manipulations were set to the lower and upper bounds of possible values for the discount factor, which are 0 and 1 respectively. After these two tests converged, small incremental changes were made from 1, subtracting tenths, hundredths, and even thousandths at a time. In all, a total of 23 different values were tested in the range from 0 to 1. This ensures that we receive a broad distribution of values collected, giving a complete picture as to what the effect of the discount factor is on the number of iterations for value iteration. We were unable to move out of this range, however, because the algorithm is not guaranteed to converge, and will potentially run infinitely.

In part (b) of this experiment, the step cost was the value to be adjusted. The step cost, the reward (positive OR negative) given at every step, has very strong influence over the agent's decisions, and the optimal policy that is calculated. The number of iterations that the algorithm took to find the solution was recorded with each different step cost value tested. To begin, the step cost was tested at what we decided would be our bounds, -100 and +100. We then began to work our way outward from 0 to both of the bounds. However, as the step cost approaches and crosses 0 (from negative to positive), increasingly smaller increments were necessary. The reason for this is that the increases in iterations were much more extreme as the step cost approached 0 and then entered the positive domain. As a result, the increments when the step cost is close to 0 were microscopic. This incrementing process was then repeated until the changes seen in the numbers of iterations started to plateau (relative to the changes seen closer to 0), at which point the step cost was both increased and decreased by larger increments, until the bounds were reached. In total, 37 different step costs were tested in the range of -100 to 100. Not only does this give a full range of values, but the experiment is also necessary to give insight as to what the agent is doing when the step cost is altered, for example to a large negative number, or a massive positive number. Intuitively, it would make sense that the more negative the step cost is, the fewer steps the agent would want to take, and so the optimal policy will be reached sooner (fewer iterations). The opposite is also true: when the step cost is a large positive number, the agent will try to take as many steps as possible, in order to collect as much utility as it can (more iterations).

Parts (c) and (d) of Experiment 4 test the effect of the negative and positive terminal rewards on the number of iterations of VI. The reasoning for this test is to gather quantitative data regarding the number of iterations needed for VI to converge while manipulating (raising or lowering) rewards in terminal states.

For testing negative terminal rewards, we tested 23 values beginning at -.00001 and ranging to -100, and recorded the number of iterations executed by the algorithm. We used increments of larger orders in places where we didn't see much change in the number of iterations, and we chose increments of very small size in places where the number of iterations executed by the code seemed to increase or decrease greatly. By sampling more in areas of larger absolute change, we are better able to visualize and determine the effects of manipulating the negative terminal rewards.

For positive terminal rewards, we, not surprisingly, used the same methodology. We sampled 31 different values of positive terminal reward, ranging from 0.00001 up to 1000. Again we were careful to notice places where the number of iterations jumped by large margins and chose to take sample more values in those key areas. Again, for building visualizable models with our data, we decided this sampling technique would lend itself well to displaying accurate information.

Part (e) of the experiment was designed to look at fluctuations in the number of iterations of VI as we manipulated the value of the key loss probability. Here we did not expect to see much change in the number of iterations, but we did not want to discount the effects that the key loss probability could have on the MDP. For this test, we chose incremental values between 0 and 1. Since this is a probability, any other numbers would yield unusable results. We incremented our probabilities by a relatively constant factor, but as in parts (c) and (d) we did do a bit more sampling where we noticed greater changes in output values of the code.

Our 5th experiment differs from the others in that it was purely for gain of qualitative information. In solving an MDP, the true goal is to determine an optimal policy, so it would be remiss to not look at the effects on the policy with the manipulation of input values into the MDP. This test is similar to our 4th in that it looks at effects when changing the same variables, but here we are not observing the number of iterations, but instead at the behavior of the agent in in an environment that we specify with input parameters. Again, we are looking at the results of manipulating the following:

a) discount factor
b) step cost
c) negative terminal reward
d) positive terminal reward
e) key loss probability

We take into account exactly one parameter at a time while holding all other input values constant. Again the default parameters in the function call are same as in Experiment 4.

In all 5 parts of this experiment, we tested a small number of inputs, 5-10, and looked for patterns in the agent's behavior as values were manipulated. Because the grid-world is small and well defined, we were able to develop a very specific set of patterns to look for while running these tests. Specifically, we took note of:

- Risk – was the policy 'safe,' or did it specify actions that could lead to negative terminal rewards?
- Direction – was there apparent direction to the policy?
- Key Grabs – did the policy specify an attempt to grab the key? From what states was it beneficial to attempt to grab the key?

And we worked to fit trends in these behaviors to increases and decreases in each respective parameter we manipulated.

4. Results

The results from our first experiment are simple, yet remarkable. In measuring the number of iterations it takes for Value Iteration and Policy Iteration to execute, we found the following: It took a total of 64 iterations of VI for the algorithm to terminate, while for PI it only took 8, as seen in Figure 1. This shows how impressive of an improvement PI is in solving an MDP. It highlights the relatively large magnitude of the change in state utility vectors from one iteration to the next in PI, as compared to VI. It also shows how ineffective VI can be in converging on an optimal state utility vector, and how slowly rewards propagate through all the states in an MDP.

Experiment 2 also yielded some very interesting results. The time taken for each algorithm to execute the default command averaged over 10 tests were as follows, as seen in Figure 2:

Value Iteration: 0.007559758 seconds
Policy Iteration: 0.018152875 seconds

The 10 averaged times can be seen in figure 2 as well. These times are particularly interesting because they do not match intuition we held based on results from experiment 1. Even though PI iterates one-eighth the number of times as VI, it is about 2.3 times slower.

Figures 1 and 3

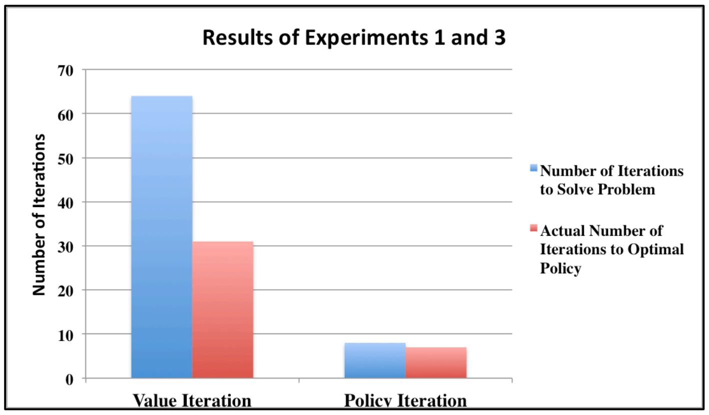


Figure 2

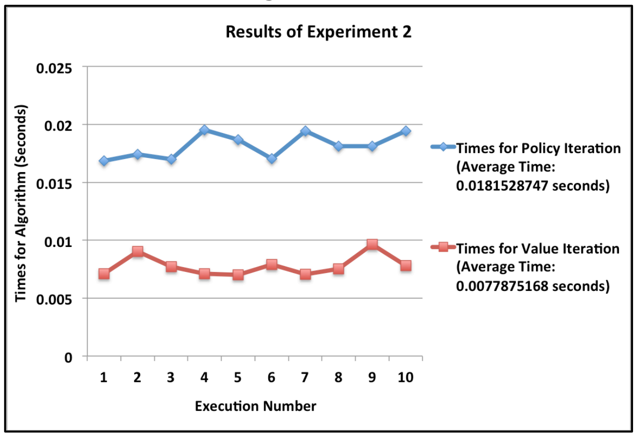


Figure 4a

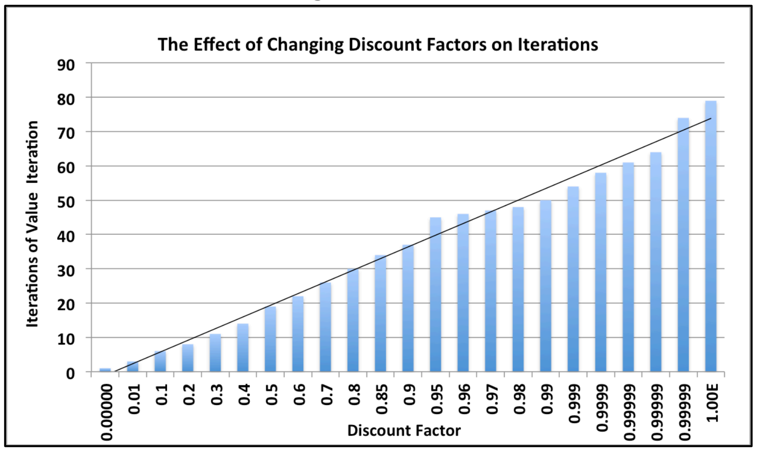


Figure 4b1

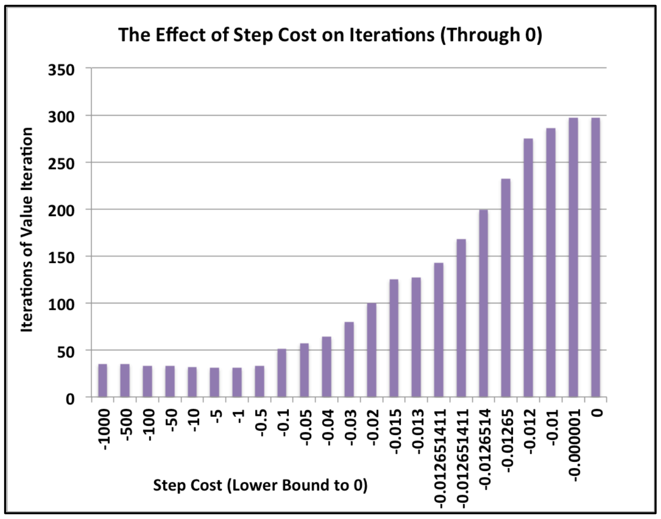
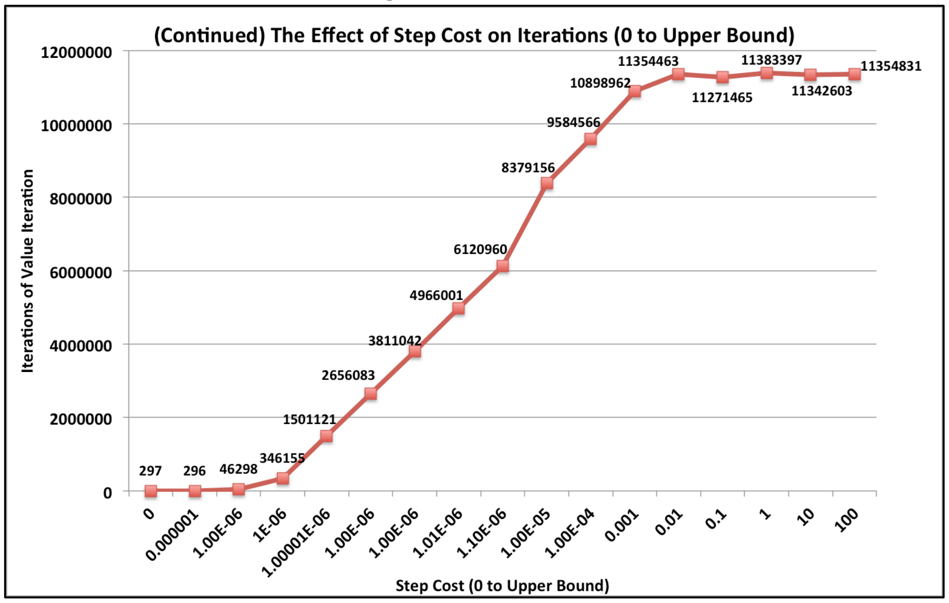


Figure 4b2



Tables 4b1 and 4b2

Step Cost	Iterations of VI
-1000	35
-500	35
-100	33
-50	33
-10	32
-5	31
-1	31
-0.5	33
-0.1	51
-0.05	57
-0.04	64
-0.03	80
-0.02	100
-0.015	125
-0.013	127
-0.012651411	143
-0.012651411	168
-0.0126514	199
-0.01265	232
-0.012	275
-0.01	286
-0.000001	297
0	297

Step Cost	Iterations of VI
0	297
0.000001	296
1.00E-06	46298
1E-06	346155
1.00001E-06	1501121
1.00E-06	2656083
1.00E-06	3811042
1.01E-06	4966001
1.10E-06	6120960
1.00E-05	8379156
1.00E-04	9584566
0.001	10898962
0.01	11354463
0.1	11271465
1	11383397
10	11342603
100	11354831

Figure 4c

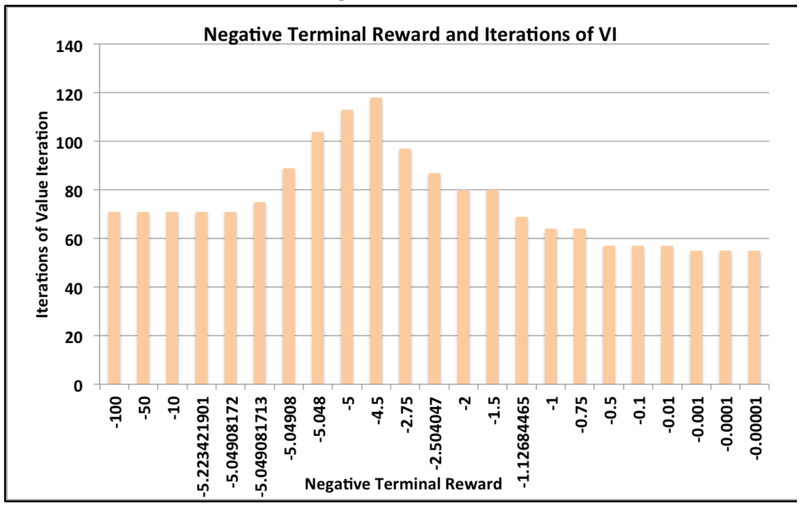


Figure 4d

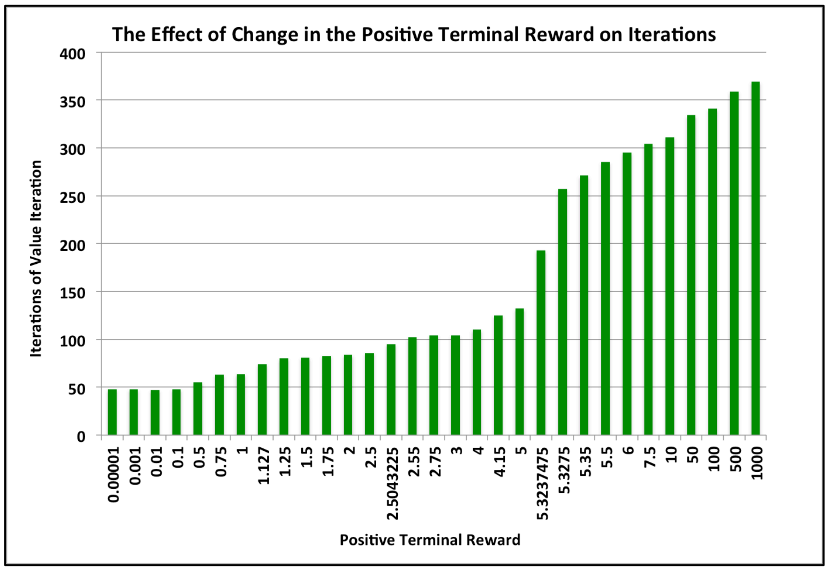


Figure 4e

Key Loss Probability	Iterations of VI
0	65
0.001	66
0.01	70
0.1	71
0.2	75
0.3	77
0.3157295	70
0.5	64
0.55	65
0.6	65
0.75	65
0.9	65

We did not expect this. We assumed there would be a very close correlation between the number of iterations of an algorithm and the time it would take to execute. In review of the code, however, we see that the code for checking if a policy is optimal (not the policy evaluation) is nearly identical to that of the algorithm used in VI. So it then becomes very apparent how costly solving a linear matrix equation on each iteration is. Noting that solving a matrix equation is an $O(n^3)$, and that in our case n is the 65 states of our grid-world, it seems more obvious that this is the slower algorithm. Further, if we look at the times for one iteration of each algorithm we see:

VI iteration: 0.00012168 seconds
PI iteration: 0.002269109 seconds

This means that a single iteration of Policy Iteration is 18.3 times slower than one of Value Iteration. That is an impressively bad factor.

In Experiment 3 we see yet another interesting characteristic of the algorithms we've implemented. We looked at the discrepancy between the number of iterations needed to find an optimal policy and the number of iterations it took to for the algorithm to terminate. These differences are illustrated in Figure 3. In Policy iteration, it took 7 iterations for the policy to become optimal. This was completely expected, since the algorithm terminates after finding an optimal policy. The reason it didn't take 8 iterations is because on the final iteration, the policy is not actually updated, and the iteration instead serves as a test to see if the policy has converged. Technically, we have the optimal policy after the 7th iteration of PI.

The more interesting results were in seen in Value Iteration. While the code takes 64 iterations for VI to break, it only takes 31 iterations to find the optimal policy. While this is surprising, it can be explained easily. We could eliminate the extra iterations by increasing the value of the ε, the maximum state utility error. If we set this bound higher, the values in the utility vector *U* would converge to a threshold we deem appropriate, sooner. If the correct value of ε is chosen, which would certainly take some effort to find, it would cut down the number of iterations to the absolute minimum possible while still maintaining the optimal policy.

It was in our 4th experiment that we really gained a more thorough understanding of Value Iteration. As we manipulated our input parameters, we were able to see definite relationships between those parameter values and the number of iterations VI needed to execute until convergence. We will discuss results in sections, by the parameter we were manipulating in each test.

In part (a) we were working with the discount factor. Starting with a discount factor of 0.0000001, VI took a total of 1 iteration to execute – not surprising; Rewards were discounted by so much in this case that values of state utilities almost the exact same value of the rewards in those respective states. As the factor increased, the iterations grew too. You can see this very close relationship in discount factor and iterations in Figure 4a. We saw steady growth in iterations up to our default value of 0.999999, where there were 64 iterations and further to a discount factor of 0.999999999. Here, and for values above, we saw iterations reach an apparent ceiling at 79. Because our grid-world is relatively small, VI was able to converge despite the high discount factors, which are very close to 1. Had the number of states been substantially larger, this may have not happened, because rewards far in the distance for a state would not be discounted enough, and utilities would grow in each iteration such that the utility vector would never converge.

In part (b) of this experiment, we looked at the effects of step cost on the number of iterations of VI. We tested negative step costs first. Starting form step cost -1000, we observed low iterations: 35 vs. the 64 iterations for our control. As values increased toward 0, we saw very little fluctuation. There was a slight dip to 31 iterations when the step cost was -1, but the iterations were largely held constant. On the interval from -1 to 0, we observed growth in the number of iterations. You can see the growth in Figure 4b1, and corresponding Table 4b1. As step costs approached 0, iterations were high but appeared to be leveling off. In the MDP if the negative step costs are low, the only rewards in the environment come from negative terminal states, so those rewards must propagate through the system and it takes a lot more iterations for the utilities to converge.

We also tested positive step costs. Here there was a very apparent pattern. At cost 0.000001, there were 296, but at all higher values, the number iterations became very large. In fact, growth was almost instantaneous at this threshold. At step cost 0.000001000001, the number of iterations was already 346155. As you can see from Figure 4b2, small increments in the step cost sent iterations into the millions. Table 4b2 also shows these changes using color to denote the size of value, with green being low, and red being high. It is not hard to understand what is happening here. If step costs are positive, the agent gains rewards from taking as many steps as possible. Therefore, the utility vector continues to increase because state utilities do not stop increasing through iterations of the algorithm.

In part (c), we focused on negative terminal rewards. Here, results were interesting. Figure 4c shows the general trend in iterations given negative terminal rewards. You can see that when negative rewards are either very high or very low, the numbers of iterations are low and somewhat constant. At a negative rewards between -2 and -5 however there were greater iterations. In the MDP, when rewards are very high or very low, the policy becomes very obvious. But at intermediate values, the policy is less obvious, and the algorithm takes more iterations for the state utilities to converge.

In part (d) we expected to see a very similar trend as in part (c), but were surprised to find a completely different trend. Here, we were looking at positive terminal rewards. Evidenced by Figure 4d, unlike the bell-curve shape we saw in testing negative terminal rewards, as positive terminal rewards grew from 0, the number of iterations experienced strict growth. This is perhaps due to the placement of the positive terminal state. Far to the side of our grid-world, it can take some time for rewards to propagate to far away states. If positive reward is large, the discount factor discounts a lot of the reward, and it takes longer for the full reward to propagate to far away states. Utilities converge slowly because each utility is growing by a large factor in each iteration.

In part (e) of our 4th experiment, we looked to see if key loss probability had any effect on Value Iteration. It ended up having very little effect overall. In all probabilities we tested, we only saw a maximum difference of 13 iterations. You can see this in the table presented by Figure 4e. Key loss probability has very little effect on the number of iterations because it very indirectly affects the state utility vector. The key loss state is also placed such that it easy to slightly alter a policy that avoids that state if the probability is too high, so it is not a huge factor in changing individual state utilities.

Our 5th experiment is the most qualitatively interesting. We were able to determine different patterns in the policy with a strong correlation to each separate function parameter we manipulated. To make sense of the results we'll discuss each test separately as we did for experiment 4. It is obvious that there is no true way to measure these changes quantitatively, and as a result they are strictly observations. Because of this, more description of these results will be presented in Section 7, called Comments, after the conclusion. The raw observations are on the following page, in a table labeled Figure 5.

In part (a) we tested a range of discount factors starting from 0.0000001 and ranging to 0.9999. For the low values we saw very little update to the policy from the initial policy which is set such that at each state, the policy is to choose the North action. This is because with a low discount factor, rewards from terminal states do not propagate to states that are far away on the infinite horizon. As we increased the discount factor by a small margin, we saw a very similar situation, but close to the negative terminal rewards, the policy was to move away from them, and near the positive terminal reward, the policy specified actions that would lead to that state. The more we increased the discount, the more we saw the policy align with the policy specified when the program was run with the default parameters. At a discount factor of .9, we start to see the policy change in the top few rows of the grid-world, such that the agent would make an attempt to grab the key before going to the positive terminal reward state. With a discount factor of .9999, we reached convergence with the default policy. Here the agent will attempt to grab the key if it is in the top 5 rows of the grid. It will choose the most direct path through the negative terminal rewards to reach the key state, and then it will take the most direct path back to the positive terminal reward state. If the agent is keyless and close to the positive terminal reward state, it will attempt to go there without getting the key first.

In part (b) of experiment, we looked at the effect of step cost manipulation on the policy. First we looked at step costs that were ≤ 0. With a step cost of 0, we saw what we classified as a very safe policy. From all keyless states (except the terminal states), the agent would attempt to grab the key, but in doing so it would choose a very indirect route, that avoided any possibility of ending in the negative terminal states. If the agent had the key, the policy specified the same indirect route, but in revers, leading to the positive terminal reward state. As the step cost increased toward -0.04, the default value, we saw a policy that became more and more risky. At a step cost of -.1 we saw that if the agent was keyless, it would move the closest terminal state.

At a value of -.2, the policy specified actions that lead to the closest terminal state, regardless of weather of not the agent had the key. It is very apparent that as negative step cost increased, the negative rewards of taking steps was far greater than the negative terminal rewards, so the agent would naturally attempt to reach a terminal state as fast as possible.

We also tested a few positive step costs. The pattern was very obvious; the agent would take any action that led it away from the terminal states. Because there are positive rewards for taking a step, the agent here is trying to take as many steps in the environment as possible.

In part (c), we manipulated the negative terminal rewards. If numbers were low (below -.1) the agent would take risky paths to and from the key state. As these values approach -1, the policy begins to converge to the default policy as expected. As values grow to -2, the agent starts to take less risky paths to the positive terminal state weather or not the agent has the key, and after -2, we don't see any further changes in the policy. This all stands to reason. If negative terminal rewards are high, the agent will pick any action that does not lead to the negative terminal states.

For part (d) we looked at positive terminal rewards. When rewards are low (close to 0) we saw the policy specify moves directly to the positive terminal state, without making an attempt at grabbing the key. As rewards increased to .5 we saw that in keyless states close to the key, the agent would make an attempt to grab the key. At values above 1, and approaching 2, we saw the agent go for the key and take very safe paths around the negative terminal states. For values greater than 2, we did not see any further change in the optimal policy. Here the agent is optimizing its rewards by going for the large positive rewards. To do this, it is making the safest choices possible, and even choosing to go through the key loss state when the positive rewards were high enough,

Finally, in part (e) we observed the effects of manipulating the key loss probability. This parameter had the least significant effects on the policy, but we still did find some subtle trends. When the probability was very low, the agent moved freely through the key loss state so as to avoid the negative terminal states. At around a probability of .25 however, this trend seemed to end and for probabilities greater than .5 the agent would no longer go through the key loss state. This is simply reflects the fact that the agent is far less likely to gain positive rewards and more likely to accumulate negative rewards from step costs if it loses the key. So if key loss probability is high it will not chose to move through the state.

5. Further Work

There are many places to go from here. Now that we have a thorough data set, and two basic algorithms implemented, we have the tools required to make improvements to the algorithms. There are a number of improvements to the Value Iteration and Policy Iteration algorithms that can decrease expected time complexity and number of total iterations. By tweaking our current algorithms and measuring data sets against each other, we will be able to see if changes to our code are actual improvements in solving the MDP. We will now propose and discuss some improvements to Value Iteration.

First and foremost is the huge discrepancy between the numbers of iterations in solving an MDP with VI and the number of iterations VI actually takes for the policy vector to converge. This seems like a trivial fix, but is perhaps a bit more intricate than we first thought. First, we thought to break the loop as soon as the policy from before the update of state utilities matched the policy from after the update. This would have been an easy fix, but it is incorrect. The policy vector in VI does not experience strict improvements as it does in PI. There are iterations where no change is made to the policy vector, but the policy is not yet optimal. So we must work on a heuristic that determines when the policy loss of breaking VI early is deemed small enough. This could take the form of deciding on a number of how many iterations without updates to the policy vector is enough for the conclusion to be that the policy good enough. This proposed solution, however, could be much more complicated.

Another improvement on VI that we have considered could potentially be a reduction in the running time. This would not affect the number of iterations, however. In value iteration, we update an entire vector of state utilities until the entire vector has converged. This is unnecessary. We can, in constant time, determine if a single state utility has converged, and if it has, it no longer needs to be updated in any further iterations of value iteration. It could be a bit more costly in space, as it would implementation of another data structure to keep track of converged state utilities, but it would be interesting to see the effects this method could have on running time.

6. Conclusions

In summary, we have given two solution techniques to MDP problems like the grid-world that we solved. These algorithms, called Value Iteration, and Policy Iteration, generate an optimal policy for the agent, which specifies the action it should take at each state in order to maximize the expected utility.

These algorithms differ fundamentally in the way that they go about calculating the optimal policy, though. Value Iteration takes a more indirect approach, calculating the utilities of states by backwards propagating the rewards from the terminal states.

We tested this algorithm by modifying the default parameters given to it. The five that were chosen were:

a) discount factor
b) step cost
c) negative terminal reward
d) positive terminal reward
e) key loss probability

These parameters were modified individually, and then we recorded the number of iterations VI takes to converge given the change of that parameter. The discount factor, step cost, and the positive terminal reward had the greatest influence over the numbers of iterations that the algorithm took. In terms of time, the algorithm executes very fast, however we see that it is not totally efficient. The optimal policy is actually reached at 31 iterations, so there is obviously room to improve. In order to do this, a heuristic would need to be developed that would essentially decide when the change in state utilities is negligible enough to terminate. This would take some toying with, as there is possibility to terminate much too soon.

Policy Iteration takes a much more direct approach to solving the MDP. With each iteration, we set up a matrix equation with n equations and n unknowns, and solve it to update a state utility vector U. This portion of the algorithm is the evaluation of the current policy. After this, we test whether the policy is optimal. If the policy is not optimal, the policy vector, π, is updated, and the process is repeated until the policy vector does not change throughout an iteration of the algorithm. The algorithm itself performs very well, solving the MDP problem in 8 iterations, as compared to the 64 that VI took. In addition, the optimal policy was reached at iteration 7, compared to 31 iterations of VI. The main difference in execution comes down to the running time of PI. A single iteration can take up to 18 times longer than one of VI, and overall, the algorithm took an average of 2.3 times longer than VI to solve the MDP given the default parameters.

7. Comments

In regards to Experiment 5, the results are essentially impossible to model numerically, and as a result our conclusions were drawn from our observations. Given the parameter that is changed, we executed the algorithm and reviewed the optimal policy generated. General trends were then recorded, which allowed us to piece together how the policy is changing. These are presented in the following Figure 5 on the following page, which is a table of our documentations of changes when executing the Value Iteration algorithm.

Figure 5

5a. Discount factors	agent behavior	5b. Step cost <=0	agent behavior	5b. Step cost >= 0	agent behavior
0.0000001	undeterminable, Will only go north or east	0	Chooses safest path, will go for key form any keyless state. Path with key loss state	0	Chooses safest path, will go for key form any keyless state. Path with key loss state
0.01	largely unchanged, too much discount to propigate	-0.0001	same as above	0.000000001	take safe path to key and then to positive terminal state
0.1	same as before, will avoid negative terminal rewards	-0.001	same as above	0.0001	take any action to avoid terminal state
0.5	still, rewards of termial states not fully propigating, will move toward positive terminal reward and away form negative, but will not grab key	-0.01	same as above		
0.7	agent starts making attempt at grabbing key if close. Avoides terminal rewards	-0.03	more risky around negative terminal states. Will still go for key from most states		
0.9	agent chooses to go for key if in top two rows, will choose	-0.04	even more risky, no longer goes through key loss state. Will go for key unless close to positive terminal reward state		
0.99	keyless agent goes for key unless close to positive terminal reward	-0.08	takes most direct path to key and then to positive terminal rewards		
0.9999	policy esssentially same as above. Converged	-0.1	if keyless will move toward any terminal state. It with key will avoid negative terminal and go to positive terminal		
		-0.2	essentially goes to closest terminal state.		
			same for all lesser values		

5c negative rewards	agent behavior	5d. Positive rewards	agent behavior	5e. Key loss probabil	agent behavior
-0.001	if no key, and below negative state, will north. Everything else is same as default	0	take most direct path to positive terminal without going through negative terminal	0	if with key near key loss stae, will move through key loss state
-0.1	takes risky direct paths to key and positive terminal state, will not move into negative terminal if without key	0.1	same as above	0.25	will do same as above
-1	default, avoid negative terminal	0.5	attemts grab at key if keyless and not too far from key state. Takes direct path from key to positive terminal	0.5	if above key loss state with key, will move through it
-2	will still take most direct path to positive terminal from key state	1	standard.	0.75	now, will not move through key loss state unless already keyless
-5	takes longer path around both negative terminal states instead of going between	2	will go for key unless very close to positive terminal, still takes direct path from key to positive terminal	1	same for all greater values
-10	will take path least likeley to end in the negative terminal state	5	takes safe path to key state and from key state to positive terminal		
	same for larger negative values		not changing after this		