Value Iteration and Q-Learning: SPML Assignment 4

Mantas Makelis (1007870) and David Leeftink (4496612)

Search, Planning & Machine Learning: BKI212A
Dr. P.A. Kamsteeg, Dr. J.H.P. Kwisthout
Radboud University, Nijmegen
January 12, 2019

Abstract

In this paper we present an implementation of the Value Iteration and Q-learning algorithms. Both implementations are made in a grid world, where the agent can discover positive rewards, negative rewards and obstacles that cannot be visited. The code for both algorithms was realized in Java programming language. In this paper, both algorithms and their implementations are explained along with the different observed outcomes by changing various parameters.

1 Introduction

The present report consists of two algorithms, namely value iteration, which can be found in section 2, and q-learning, section 3. The report covers the processes of both algorithms and their implementation in Java programming language. In both implementations, an agent is placed in a grid world consisting of $n \times m$ squares, where every square represents one state. The grid world representation and the framework the Markov Decision Problem is build on is made by Jered Vroon.[1] Additionally, the tests and their results will be reported and discussed.

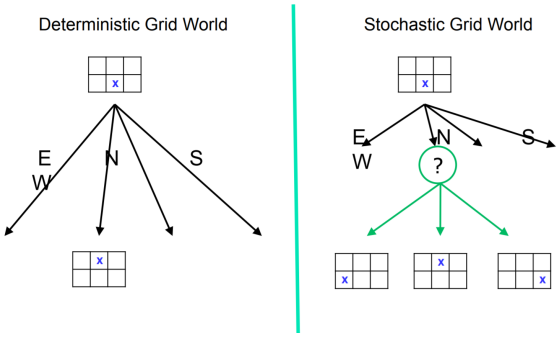

Figure 1: The difference between deterministic and stochastic approaches in the grid world. The graph is created by Berkeley University.[2]

The grid world domain can be approached in two different ways: fully deterministic or stochastic. The deterministic approach means that any action that the agent attempts will be carried out as intended. The stochastic approach contains a certain amount of randomness: the action that is selected by the agent is not guaranteed to result in this action. Figure 1 visualizes both approaches. In the deterministic approach, an action north will bring the agent to the desired square. However, in the stochastic approach an action north might instead lead to any other position that was reachable from the previous position.

To experiment with the performance of both algorithms, we have used several grid block worlds to test for the behaviour of the algorithms. A visualization of two examples of such worlds can be found in figure 2. The worlds are designed to vary in size and in difficulty.

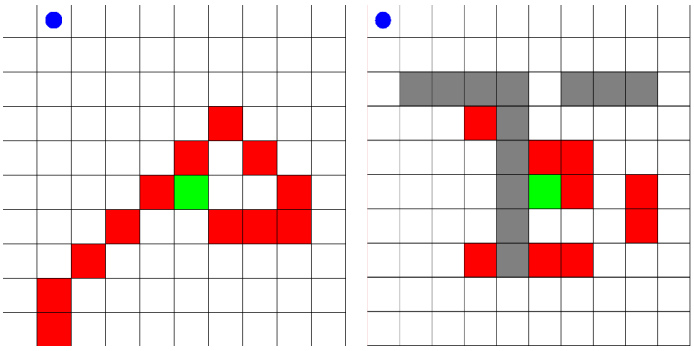

Figure 2: Two example visualization of grid worlds used for measuring the performance of the Value Iteration algorithm and the Q-learning algorithm. The green blocks represent reward states, while the red blocks represent negative reward states. The grey blocks are obstacles and cannot be visited. The second visualization is designed by Stanford University.[3]

2 Value Iteration

2.1 Methods

2.1.1 Algorithm

The value iteration algorithm takes a grid world and calculates the best policy π for each state. The policy is considered to be the most valuable action in a given state. In simple words, the policy determines which action should be executed to achieve the most optimal result. To attain it, the algorithm needs an array of all possible states S and an array of all possible actions A. Then at run-time the algorithm uses a transition function $P(s'|s, a)$ which specifies the

new state s' after the action a was taken in state s. Furthermore, the reward function $R(s, a)$ is necessary to determine the value of an action a in state s.[4]

The algorithm starts by assigning a value of zero for every combination of action and state $Q(s, a)$. Since in principle there is no ending to the algorithm, value iteration requires an additional parameter k which indicates how many time the values of each $Q(s, a)$ will be updated. This is necessary for the termination of the algorithm. If this is not specified the algorithm continues into the infinity. The recursive update of the value is carried out by the following equation:

$$Q^*(s, a) = R(s, a) + \sum_{s'} P(s'|s, a) \cdot \gamma V^*(s') \tag{1}$$

In this formula the γ is a a constant and very important parameter of the value iteration. It regulates the importance of the future reward. The higher the gamma is set the more important the future state value becomes. After calculation of values for each pair $Q(s, a)$, the most valuable action is selected from all pairs containing the same state s. This is solved using the following formula:

$$V^*(s) = \max_a Q^*(s, a) \tag{2}$$

The most desirable outcome of the value iteration is for values to converge into an equilibrium which means that values no longer change no matter the amount of calculations. In such a case the true optimal policy for each state can be retrieved by finding the action to which the highest value belongs:

$$\pi^*(s) = arg \max_a Q^*(s, a) \tag{3}$$

2.1.2 Implementation

The following Java implementation of the algorithm uses two two-dimensional double arrays to represent the values and rewards of the states. When the algorithm is initialized the rewards are set according to the grid world. The implementation starts with iterating over all states for $k = 1000$ times and updates value V corresponding to the state-action pair.

To reach each state-action pair, all possible actions are retrieved. Then the iteration over state-actions pairs begins and calculates the value using equation 1. This part is the most gruesome part of the algorithm implementation since using non-deterministic approach in value iteration each action can result in any other action (right picture of figure 2) depending on the probabilities set in the Markov Decision Problem. For this reason every different action type (i.e. up, right, down, left) might lead to any other with some probability. Therefore, the possibility of resulting in any other state than expected must be accounted for.

To account for this, another iteration over all probable outcomes begins. First the algorithm checks if the possible move will not put the agent out of the bounds of the map, and also if the next state is not an obstacle (a barrier). If these conditions are met then the equation 1 is applied. Otherwise, if the state where the action leads is not an obstacle but out of range then in equation 1 $V^*(s')$ is simply replaced by -1.

When all the possibilities are accounted for and all actions in a state have a value calculated, formula 2 is used to fetch the best value of the corresponding state. When the k value hits the targeted 1000^{th} iteration the calculations are finished. As a result of the high amount of iterations, the whole landscape is evaluated and an (nearly) optimal policy is defined for every state.

The value iteration algorithm is concluded by assigning a policy to each state. The policy is retrieved with formula 3. This takes place directly after the calculations finish and is executed by again going over each state and picking the action associated with the highest value. Accordingly, the new two-dimensional array is created containing optimal policies.

2.1.3 Measurements of performance

The value iteration algorithm is measured by the total amount of $Q(s, a)$ values calculated. Such measurement provides an insight into the complexity of the implementation because it measures the amount of smallest calculations made by the algorithm.

2.2 Results

2.2.1 General performance

The performance of the algorithm was really good. The computation feels pretty instant for the 10×10 grid worlds which in total has 100 states. For example, the number of calculation for the replication of the Stanford map (right picture in figure 2) resulted in a round number of 1.560.000. As for the other map (left picture in figure 2) the algorithm computed 1.800.000 $Q(s, a)$ values. The end measurement of the complexity between same size maps depends on how many obstacles the map has.

2.2.2 Different discount factors

The change of γ (discount factor) was quite dramatic. If the discount was set to any value below .86 the algorithm could not converge to anything close to equilibrium, therefore, not making an optimal path to the reward state. Not only that but also the agent was always going away from the any negative reward tiles and getting stuck in one corner. Such behaviour was especially observed in bigger 10×10 maps with narrow passages. Any γ value above .86 did not

seem to influence the end result and the resulting optimal policies seemed to be equal or highly similar.

2.2.3 Different State Penalties

It is worth noting that changing values of the reward and penalty states had a major influence to how the value iteration algorithm performed. The most bizarre behaviour was observed when the value of the reward was very high, much higher than the penalty reward. Since the implementation did not take the termination state into consideration, the agent thought the high reward can compensate for stepping into a negative state. Which of course would be a logical thing to do if the penalty state was not a termination state. As a result, the agent was most rational in an opposite situation, when the penalty was set higher than the reward.

2.2.4 Different Transition Probabilities

The increase in transition probability, which refers to the right picture in figure 1, resulted in agent making random steps when following the calculated policy. This meant that instead of reaching the reward state agent sometimes made foolish steps straight into the penalty state. The non-determinism in value iteration seems to be irrelevant which only obstructs and hinders the execution of optimal policy moves.

2.3 Conclusion/Discussion

The overall performance of the implemented algorithm is very good. The modern processor can easily handle the amount of required calculations to find an optimal path in the 10×10 grid world and completes it in seconds. The implementation could be further improved by scraping unnecessary iterations of probable actions for zero probabilities. Mathematically it does not impact the algorithm but it definitely impacts the complexity although only by a little margin in small grid worlds. Of course this scales exponentially. Another improvement that could be made to the algorithm is to restructure it and use a class to represent the values instead of two-dimensional arrays for decreased complexity. This change could improve the algorithm quite significantly.

3 Q-learning

3.1 Methods

3.1.1 Algorithm

Q-learning is a reinforcement learning algorithm where an agent tries to learn the optimal policy from its history of interaction with the environment.[4] Instead of learning from data, the agent learns of its own actions and the rewards

of them in the environment it is placed in. Q-learning uses the Temporal Differences between the experiences to estimate the value of an action in a state.[4] The agent maintains a table of values for each action in each state, that gets continuously updated with the Temporal Difference equation.[4]

$$Q[s,a] = Q[s,a] + \alpha(r + \gamma maxQ[s',a'] - Q[s,a]) \tag{4}$$

By evaluating the reward of the state in an experience, the agent will learn after many iterations which actions are good and which actions are bad. This allows the Q-learning algorithm to learn optimal policies by itself, by observing the reward of each action and state.

Since the algorithm does not specify exactly what the agent should do, the agent is likely to stick with the reward values it will find early even though it might be possible that this is not the optimal policy. To counter this, a distinction is made between two considerations for an agent: exploitation and exploration. In exploitation it performs the action that returns the highest value, while in exploration it selects a random action in a different direction. The rate that this happens is represented by the variable ϵ. Often, there is a trade-off between exploitation and exploration. Too much exploitation will prevent the agent from finding the optimal policy, while too much exploration will prevent the agent of benefiting if the optimal path was already found. Another important value that is used in Q-learning is α, often referred to as the learning rate. This is the weight that is given to the new experiences. Lastly, Q-learning also uses the variable γ as is explained in section 2.

An important difficulty in Q-learning is the *blame attribution problem*, the problem of determining which action was responsible for which outcome.[4] It is uncertain for the algorithm at which time the action that was responsible for the right or the wrong move happened. This becomes especially difficult when it is a combination of actions that leads to for instance a high positive reward. The algorithm can only find out by running through the environment a large number of times.

3.1.2 Implementation

The implementation of Q-learning is written in Java. It follows the main outline of the algorithm, but differs in some details in the implementation. Every grid in the grid environment is represented by a so called 'Qfield'. This is a class that represents all the possible actions that an agent can perform and the possible rewards for these actions. Initially, these rewards are assigned arbitrarily. At every iteration, the agent selects the action with the highest expected reward out of the list of possible actions. The success rate of executing this action is defined by ϵ. Whenever the agent visits a new state, the pair of state and action gets updated within the 'Qfield' object. All the logic takes therefore place within this class, including the TD-formula (formula 4).

The main class 'Qlearning' represents the decisions the agent makes. It contains the Markov Decision Problem and keeps track of all the fields that can be visited. These fields also return the possible actions and their values. The main procedure is the following: the agent selects an action, that depending on the outcome of ϵ results in the highest-rated action. After this, the rewards of the new state are obtained and the TD-formula (formula 4) is used to update the value of the given action and state. The current state then switches, and the iteration is complete.

3.1.3 Measurements of performance

A reinforcement learning algorithm is measured by the reward it obtained. This is done by measuring the accumulative reward in a given world, i.e. the sum of all rewards. In the beginning this is likely to be negative since the agent will take a certain amount of trials to find the positive rewards and stabilize. Once the values for each action in each state get closer to the optimal policy $\pi*$, the reward will become higher. Since all the rewards are summed up, this performance will always eventually go to infinity. Therefore, each attempt stops when the accumulative reward is over 100.000. This way, the performance of the Q-learning agents can be compared not only in the minimum of the curve and slope, but also in terms of the amount of steps that is needed to achieve the necessary accumulative reward.

3.2 Results

3.2.1 General performance

In this paragraph the performance of the algorithm with default values for the agent are compared in three different grid worlds. These settings are a learning rate (α) of 0.1, a discount rate (γ) of 0.9 and an epsilon (ϵ) of 0.04.

When the agent is acting in a larger world, such as visualized in the left side of figure 2, the accumulative reward shows a pattern where it takes relatively long to reach the minimum of the curve. Figure 3 shows the performance of three Q-learning agents in this environment. After the minimum has been reached however, the accumulative reward rises very quickly. This also matches the behaviour of the agent: once the right value has been found, the negative states get avoided very quickly and the optimal route is achieved fast.

Figure 4 shows the performance of the Q-learning algorithm on a 10×10 grid world where the agent has to maneuver through a narrow passage that is six blocks long. This passage is only at the end of the world, to make it difficult for the agent to find the cause of the reward it gets. The performance shows an even stronger effect of negative values, since it takes the algorithm a substantial amount of time to solve such large and difficult map. However, once the positive rewards have been found the strong paths are established quickly and the limit

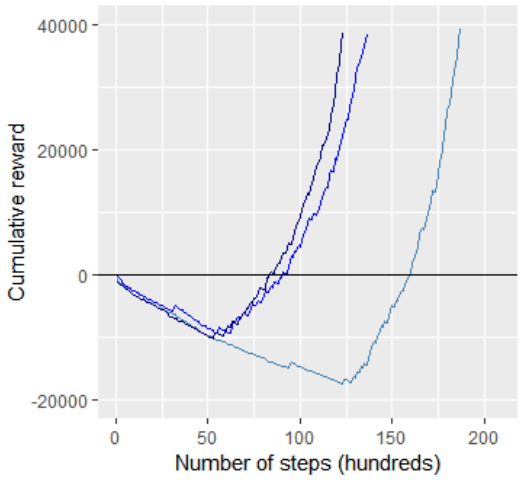

Figure 3: Cumulative reward as a function of the number of steps. Measured in a 10×10 grid world with multiple negative reward states.

of 100.000 is quickly reached.

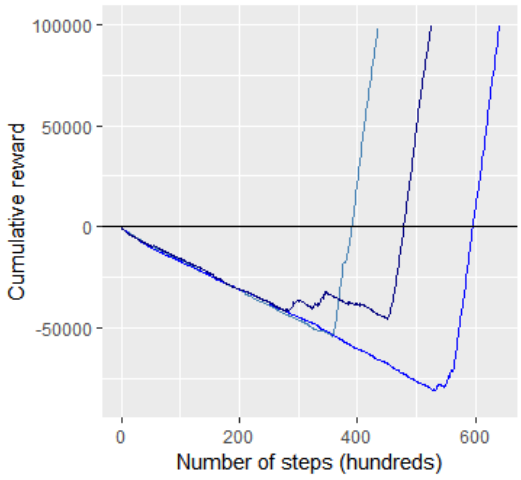

Figure 4: Cumulative reward measured in a 10×10 grid world where a combination of moves is required.

The final grid world is the grid world as designed by Stanford University, and features a 10×10 world with multiple negative reward and a single positive reward.[3] The world is, compared to the previously described worlds, relatively easy to solve for the agent. This shows in the accumulative reward in figure 5.

The black zero-line is relatively quickly crossed, indicating that it did not take the agent long to recover its costs of learning.

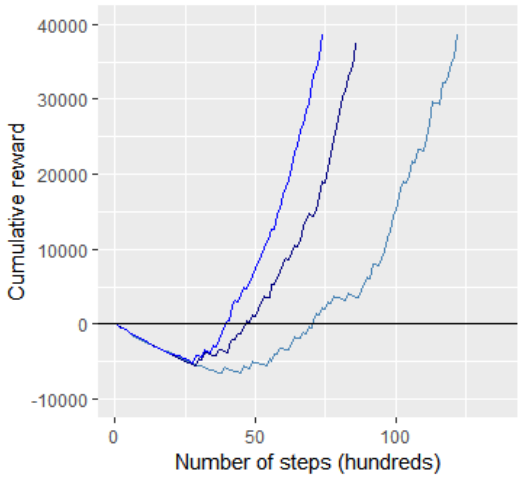

Figure 5: Cumulative reward measured in a 10×10 grid world as designed by the Stanford University Reinforcement Learning toolbox.[3]

3.2.2 Different learning rates and Epsilon values

The learning rate (α) determines to what extend new information from an experience is used to update the previous known information about the world. To test the change of learning rate and the change of exploration and exploitation, a test is setup where the performances with different values are compared to the optimal policy as determined by the Value Iteration algorithm. For every combination of high and low learning rate and high and low epsilon value, it is measured when it first reaches the same optimal policy as the Value Iteration algorithm determined. The results can be found in table 1.

Changing the learning rate had different effects depending on the environment: when ϵ was low a higher α showed better. When ϵ was high a lower alpha of 0.1 showed better. This suggests that in an environment where there is a lot of randomness a lower learning rate performs better, while in an environment that is more deterministic a higher learning rate is better. This can be explained, because in an environment that is deterministic exploitation is more wanted: the path will result in the expected reward. In a stochastic environment a lower learning rate performs better, since the danger of learning too much from mistakes is present at all times. Summarized, the more deterministic an environment is the more it will benefit from a high learning rate. This goes also the opposite way: the more uncertainty an environment has, the more it

will benefit from a low learning rate.

Iterations	$\alpha = 0.9$	$\alpha = 0.1$
$\epsilon = 0.5$	4683	747
$\epsilon = 0.2$	454	+10.000

Table 1: The effects of different learning rates in contrast to different epsilon values, measured in iterations needed to achieve the first optimal result.

3.3 Discussion/Conclusion

The Q-learning algorithm takes a high amount of steps before it can effectively work itself through the world. However, once it has crossed the zero line it quickly builds a cumulative reward that is far higher than the costs of training were. How long it takes to reach this point depends on the size and difficulty of the environment. This is related to the blame-attribution problem: when the world is increasingly complex, there are more possible actions that could have caused the reward.

The learning rate determines how quickly an agent should adopt new information about the environment. When the environment is stochastic, Q-learning performs best with a low learning rate. When the environment is deterministic, the algorithm performs better with a higher learning rate.

References

[1] Jered Vroon. Grid world implementation in java.

[2] UC Berkeley EECS Pieter Abbeel. Markov decision processes: Value iteration.

[3] Andrej Karpathy. Stanford university, reinforcement learning webtool.

[4] David L. Poole and Alan K. MacWorth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2011.