A picture containing logo

Description automatically generated

**CE/CZ4046 – Intelligent Agents**

**Assignment 1: Agent Decision Making**

**Name - Garg Astha**

**Matriculation No. - U1923971H**

**Table of Contents**

1. Problem Statement
2. Organization of the code
3. Assumption
4. Setting up the Grid
5. Value Iteration
6. Policy Iteration
7. Complex Grid Environment
8. **Problem Statement**

We have been given a maze environment. The transition model for which is as follows: the intended outcome occurs with probability 0.8, and with probability 0.1 the agent moves at either right angle to the intended direction. If the move would make the agent walk into a wall, the agent stays in the same place as before. The rewards for the white squares are -0.04, for the green squares are +1, and for the brown squares are -1. Note that there are no terminal states; the agent’s state sequence is infinite.

Diagram

Description automatically generated with medium confidence

Figure 1. Question (from the Assignment document)

With the given transition model and the reward function, the optimal policy, and the utilities of all the (non-wall) states using both value iteration and policy iteration must be calculated. In addition to this, the optimal policy, and the utilities of all the states are to be displayed, and the utility estimates as a function of the number of iterations are to be plotted. The discount factor of 0.990 is to be used for the purpose of this question.

1. **Organization of the code**
2. **Assumption**

For the purpose of this assignment, it has been assumed that the agent can intend to move towards a wall or the boundary of gird. In these cases, it will try to move towards the wall (or boundary) with probability of 0.8 and to the left or right with probabilities of 0.1 each.

1. **Setting up the Grid**

The grid is set up as given in the question. For this purpose, an entity Cell has been defined which has a reward and a property to identify if it is a wall. The rewards are assigned as given in the question, with white squares (default value), having a value of -0.04, green squares a value of +1.0 and brown squares with -1.0. This initialization is common for both the value and policy iteration methods.

The GridModel is defined as a 2-D array of the type of Cell. They are first initialized with white squares by default. The cells with brown and green rewards are updated and then the wall cells are also updated.

1. **Value Iteration**

*“The utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.”.* That is, the utility of a state is given by the Bellman Equation as follows:

A picture containing text, watch

Description automatically generated

The Bellman equation is the basis of the value iteration algorithm for solving Markov Decision Problems. There is one Bellman Equation for each state, thus, in the case of n possible states, then there are n Bellman equations.

The value iteration algorithm is implemented using an *iterative* approach, summarized in the figure 2. We start with arbitrary initial values for the utilities, calculate the right-hand side of the equation, and plug it into the left-hand side thereby updating the utility of each state from the utilities of its neighbours. We repeat this until we reach an equilibrium. Let Ui(s) be the utility value for state s at the ith iteration. The iteration step is called Bellman Update, the equation for which is:

A picture containing text, watch

Description automatically generated

Text, letter

Description automatically generated

Figure 2. Value Iteration Algorithm

* 1. **Set up**

Some constants that will be used for the implementation are as mentioned below:

C = [20.00, 1.000, 0.100]

R max = 1.000

Epsilon = [20.00, 1.000, 0.100]

Discount = 0.990

Convergence threshold = [0.202, 0.010, 0.0010]

A difference in these constant values can produce different result for utilities and the policy, which will be explained and investigated in more detail later.

* 1. **Implementing Value Iteration**

The code implementation of value iteration is based on the algorithm as in figure 2. To start off the code implementation, firstly, a 2-D array for storing the utilities is created. The array is initialized with value of 0.0 for the utility and the action is set as null for all the cells in the grid.

To keep a track of the change in utility value, a variable delta is initialized with the least value possible (Double.MIN\_VALUE).

Next, we loop over every cell in the grid and find the best action to take when the agent is in that cell such that the utility of the next state will be maximized. This is done for all the non-wall states. This is achieved by finding the utility for all four actions (up, down, left, and right) using the Bellman’s equation. With the action given, utility is calculated and then the action with the maximum utility is selected. For every cell, we find the difference in the previous utility vs the updated utility. If this difference is greater than the delta initialized earlier then value of delta is updated to the new difference. The variable delta is updated to record the maximum difference in the utility for each iteration. The count for iterations is then increased by 1, as the whole grid has been processed once. The utility and action in the iteration are recorded in an array to be accessed later on.

The above step of looping over the grid is repeated until delta is less than or equal to the convergence threshold. Since there is no time or iteration limit as well as terminating state, when delta is less or equal to the convergence threshold established earlier, the iterations are stopped. This also means that the change in utility for any cell in the grid will not be more than the allowed value (convergence threshold). The iterations over the grid are continued till this condition is not satisfied.

* 1. **Results**
     1. **Plot of optimal policy**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

* + 1. **Utilities of all states**

|  |  |  |
| --- | --- | --- |
| c = 20.0  Iterations: 161 | c = 1.0  Iterations: 459 | c = 0.1  Iterations: 688 |
|  |  |  |

* + 1. **Findings**
  1. **Plot of utility estimates as function of number of iterations**

1. **Policy Iteration**

We saw in the result of the value iteration that it is possible to get an optimal policy even when the utility function estimate is inaccurate. If one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise. The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy π0:

• **Policy evaluation**: given a policy πi, calculate Ui =U πi , the utility of each state if πi

were to be executed.

• **Policy improvement**: Calculate a new MEU policy πi+1, using one-step look-ahead

based on Ui (as in Equation (17.4)).

The algorithm terminates when the policy improvement step yields no change in the utilities. The algorithm terminates when the policy improvement step yields no change in the utilities. This means that we have a simplified version of the Bellman equation (17.5) relating

the utility of s (under πi) to the utilities of its neighbours:



It is not necessary to do *exact* policy evaluation. Instead, we can perform some number of

simplified value iteration steps (simplified because the policy is fixed) to give a reasonably

good approximation of the utilities. The simplified Bellman update for this process is

Text, letter

Description automatically generated

Figure 3. Policy Iteration Algorithm

and this is repeated k times to produce the next utility estimate. The resulting algorithm is

called **modified policy iteration**. It is often much more efficient than standard policy iteration or value iteration.

* 1. **Set up**

The k values used for the policy iteration are as follows:

k = [10, 30, 50]

* 1. **Implementing policy iteration**

The code for implementation of policy iteration algorithm is the modified policy iteration method as described earlier. Firstly, a utility and action 2-D array is declared with the values set to random actions for the initial policy. A Boolean variable is initialized to keep track of when the policy stops changing. It is set to false before iterating over the cells of the grid. Another array is declared to save the utilities of the state after each complete iteration which will be used to plot the graph for later.

On the basis of the current utility array, new utility values are estimated by running the simplified Bellman Equation for all the cells of the grid k number of times. This is in accordance with the algorithm mentioned above to get a good approximation of the utilities.

For each of the k iterations, a utility array is created to hold the current policy and the utility of the states based on this policy.

After finding the utility estimates using the simplified bellman update equations, we find the best action that will maximize the utility for the subsequent states. This is done by finding the utility of all the four actions as done in Value Iteration. Then we choose the action that will maximize the utility.

Next, for all the non-wall cells in the grid, the utility of the action obtained from the current policy is compared with the utility of the best action found in the previous step. If the utility of the best action is greater than that given by the current policy, then the policy is updated, and the Boolean variable tracking change is updated.

This process is repeated till the policy for all the cells is unchanged. This simply implies that the taking the actions defined by the current policy bring no change in the utility as compared to the best action.

* 1. **Results**
     1. **Plot of optimal policy**

|  |  |  |
| --- | --- | --- |
| **K = 10** | **K = 30** | **K = 50** |
|  |  |  |

* + 1. **Utilities of all states**

|  |  |  |
| --- | --- | --- |
| **K = 10**  **Iterations: 7** | **K = 30**  **Iterations: 7** | **K = 50**  **Iterations: 9** |
|  |  |  |

* + 1. **Findings**
  1. **Plot of utility estimates as function of number of iterations**

1. **Complex grid environment**