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 | 3 |
| 1. Organization of the code | 4 |
| 1. Assumption | 6 |
| 1. Setting up the Grid | 6 |
| 1. Value Iteration | 6 |
| 1. Policy Iteration | 14 |
| 1. Complex Grid Environment | 21 |
| 1. Appendix | 29 |

1. **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**

The code has been written in java language. The figure below displays the organization of the code files in different folders.

Figure 2. Code organization

ComplexGrid\_2

ComplexGrid\_1

PolicyIteration

ValueIteration

FileIOHandler

UtilityControl

DisplayControl

constants

Utility\_Action

GridModel

Cell

Action

main

control

entity

src

**Entity package:**

1. Action – This class defines the four actions and enumerates them with symbolic representations.
2. Cell – This class will be used to create the grid later on. It has properties for reward for the cell and if the cell is a wall state.
3. GridModel – This creates a 2-D array of type cell and initializes the cells with white, brown and green rewards.
4. Utility\_Action – This class helps in recording the utility and the corresponding action that the agent should take in a specific state.
5. Constants – This file defines the constants that are used in different files for various purposes. This includes the configuration for the value and policy iterations.

**Control package:**

1. DisplayControl – This class contains functions to display the grid, optimal policy and utilities of all states in a readable format.
2. FileIOHandler – This class contains functions to create csv with the utility values and save the optimal policy and utility values to a txt file.
3. UtilityControl – This class defines the different functions that are needed during the execution of the 2 iteration algorithms. It includes functions to estimate utilities and calculate and update utilities based on Bellman Equation etc.

**Main package:**

1. ValueIteration – This runs the value iteration algorithm by making use of the functions and values defined in other packages.
2. PolicyIteration - This runs the policy iteration algorithm by making use of the functions and values defined in other packages.
3. ComplexGrid\_1 - This runs the value and policy iteration algorithm by making use of the functions and values defined in other packages for scale value of 3.
4. ComplexGrid\_2 - This runs the value and policy iteration algorithm by making use of the functions and values defined in other packages for scale value of 6.

**Other Files and Folders:**

**Plots.ipynb –** This file in python is used to plot the graphs using the csv files obtained that store the utilities of the states.

**Results –** This folder contains subfolders to store all the csv, txt files and plot images. The subfolders “c\_01”, “c\_1”, “c\_20” contain the results of value iteration algorithm for c values of 0.1, 1 and 20 respectively. The subfolders “k\_10”, “k\_30” and “k\_50” contain the results for policy iteration algorithm for k values of 10, 30 and 50 respectively.

The subfolder “complex grid results” contains the txt and csv files for the bonus question as well as the graphs.

The subfolder “images” contains the screenshots of plots for utility against iterations as obtained from the Plots.ipynb notebook.

1. **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 grid. 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. To achieve this, 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.

This set up of the grid is used for executing value iteration, policy iteration of the given grid as well as running the algorithms on complex grid models.

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 [1]:

A picture containing text, watch

Description automatically generated

Equation 1. Bellman Equation [1]

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 3. 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 [1]:

A picture containing text, watch

Description automatically generated

Equation 2. Bellman Update Equation [1]

Text, letter

Description automatically generated

Figure 3. Value Iteration Algorithm [1]

* 1. **Set up**

Some constants that will be used for the implementation are as mentioned below and defined in the *constants.java* file:

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 3 and is executed from the *ValueIteration.java* file. 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. The functions for this are defined in *UtilityControl.java* file. 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, when the whole grid has been processed once. The utility and action found 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.

In the end, the array of the utilities is saved in a csv (*FileIOHandler.java*) form to be plotted. The optimal policy and utilities of all states are saved to a .txt file (*FileIOHandler.java* and *DisplayControl.java*).

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

Table 1. Configuration for value iteration and plot of optimal policy

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

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

Table 2. Utilities of all states

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

The results are stored as follows:

* Configuration, utility values and plot of optimal policy
  + C = 0.1, CZ4046-Assignment1\GridWorld\results\c\_01\config\_results\_values.txt

CZ4046-Assignment1\GridWorld\results\c\_01\value\_iteration\_utilities.csv

* + C = 1.0, CZ4046- Assignment1\GridWorld\results\c\_1\config\_results\_values.txt

CZ4046-Assignment1\GridWorld\results\c\_1\value\_iteration\_utilities.csv

* + C = 20.0, CZ4046- Assignment1\GridWorld\results\c\_20\config\_results\_values.txt

CZ4046-Assignment1\GridWorld\results\c\_20\value\_iteration\_utilities.csv

* Plot of utilities of the states against the number of iterations
  + C = 0.1, CZ4046-Assignment1\GridWorld\results\images\value\_c\_01.png
  + C = 1.0, CZ4046-Assignment1\GridWorld\results\images\value\_c\_1.png
  + C = 20.0, CZ4046-Assignment1\GridWorld\results\images\value\_c\_20.png
    1. **Findings**

Executing value iteration for different c values returns the same optimal policy. Although the utility values are different because of different number of iterations. Higher the value of c, higher is the convergence threshold which means that convergence is achieved with lesser number of iterations.

It can also be seen from the plots below (figures 4 - 6) that the change in utility is insignificant after about 250 iterations. It can be inferred that the optimal policy was learned by that number of iterations however, the iterations continued as the convergence threshold was not yet achieved.

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

The code for the plots is in the file Plots.ipynb.

**Chart, line chart

Description automatically generated**

Figure 4. Value iteration for c = 20

**Graphical user interface

Description automatically generated**

Figure 5. Value Iteration for c = 1

**Chart

Description automatically generated**Figure 6. Value Iteration for c = 0.1

A picture containing text

Description automatically generatedA picture containing diagram

Description automatically generated

Figure 7. Legend for the plots

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 [1]:

• **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.

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 relating

the utility of s (under πi) to the utilities of its neighbours [1]:



Equation 3. Simplified Bellman Equation

Text, letter

Description automatically generated

Figure 8. Policy Iteration Algorithm

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 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].

* 1. **Set up**

The k values used for the policy iteration are as follows (*constants.java*):

k = [10, 30, 50]

* 1. **Implementing policy iteration**

The code for implementation of policy iteration algorithm in file *PolicyIteration.java* 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 for the subsequent states in the grid. The functions used above are defined in *UtilityControl.java*.

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.

In the end, the array of the utilities is saved in a csv form to be plotted.

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

Table 3. Configuration of Policy Iteration and plot of optimal policy

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

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

Table 4. utilities of all states

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

The results are stored as follows:

* Configuration, utility values and plot of optimal policy
  + k = 10, CZ4046-Assignment1\GridWorld\results\k\_10\config\_results\_policy.txt

CZ4046-Assignment1\GridWorld\results\k\_10\policy\_iteration\_utilities.csv

* + k = 30, CZ4046- Assignment1\GridWorld\results\k\_30\config\_results\_policy.txt

CZ4046-Assignment1\GridWorld\results\k\_30\policy\_iteration\_utilities.csv

* + k = 50, CZ4046- Assignment1\GridWorld\results\k\_50\config\_results\_policy.txt

CZ4046-Assignment1\GridWorld\results\k\_50\policy\_iteration\_utilities.csv

* Plot of utilities of the states against the number of iterations
  + K = 10, CZ4046-Assignment1\GridWorld\results\images\policy\_k\_10.png
  + K = 30, CZ4046-Assignment1\GridWorld\results\images\policy\_k\_30.png
  + K = 50, CZ4046-Assignment1\GridWorld\results\images\policy\_k\_50.png
    1. **Findings**

In this case as well, the different values of k give the same optimal policy. The number of iterations is much lower than that needed by value iteration. The number of iterations remains less than 10. The results for the utility can vary slightly as they are dependent on the initial random policy.

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

Chart, line chart

Description automatically generatedFigure 9. Policy Iteration for k = 10

**Chart, line chart

Description automatically generated**

Figure 10. Policy Iteration for k = 30

Chart

Description automatically generatedFigure 11. Policy Iteration for k = 50

A picture containing text

Description automatically generatedA picture containing diagram

Description automatically generated

Figure 12. Legend for the plots

1. **Complex Grid Environment**

For constructing a more complex grid environment, the given grid is scaled up by factors of 3, 6 and 1000. This means the total number of states become 18\*18, 36\*36, 6000\*6000 respectively.

In case of scale = 1000 i.e., 36,000,000 states, the program was not able to execute as the heap memory run out. This means that another approach to implement the algorithm is required as the current approach is inadequate.

The code for implementation is in files *ComplexGrid\_1.java* (for scale value of 3) and *ComplexGrid\_2.java* (for scale value of 6). Like before, they use functions defined in the other control package classes to carry out value iteration and policy iteration.

A screenshot of a computer

Description automatically generated

Figure 13. Exception in java

**When scale = 3**, the grid environment is initialized as follows:

Calendar

Description automatically generated with low confidence

Figure 14. Complex grid model with scale = 3

In the above figure, the cells with +1.0 rewards are green squares, -1.0 are brown squares and white squares have -0.4 reward.

Policy iteration and value iteration is executed in the similar way as done earlier. The results for which are discussed as follows:

Table 5. Policy Iteration result for scale = 3

|  |
| --- |
| Policy Iteration  Scale = 3  Discount Factor = 0.990  K = 50 |
| Iterations = 19 |
| A picture containing text, light, several  Description automatically generated |
|  |

Chart

Description automatically generatedFigure 15. Plot of utilities against iterations for 18\*18 = 324 states for Policy Iteration

Table 6. Value Iteration results for scale = 3

|  |
| --- |
| Value Iteration  Scale = 3  Discount Factor = 0.990  Utility Upper Bound = 100.00  Max Reward (Rmax) = 1.0  Constant 'c' = 0.1  Epsilon Value (c \* Rmax) = 0.1  Convergence Threshold = 0.00101 |
| Iterations = 688 |
|  |
|  |

A picture containing graphical user interface

Description automatically generated

Figure 16. Plot of utilities against iterations for 18\*18 = 324 states for Value Iteration

The policies obtained from both the algorithms are similar. However, there are some differences. Regardless of these differences, they do not follow the optimal policy as it was found earlier with the grid environment given in the question. In that case, the agent made its way to the (0,0) state and stayed there to get +1.0 reward.

Since the number of states is considerably larger than in the question, using a higher discount factor can help in assigning more priority to future rewards. Another point to note is that increasing the discount factor will decrease the convergence threshold if we use other variables (c) with the same values. This will result in an increase in the number of iterations causing heap memory to run out.

Table

Description automatically generated**When scale = 6**, the grid environment is initialized as follows:

Figure 17. Complex grid model with scale = 6

In the above figure, the cells with +1.0 rewards are green squares, -1.0 are brown squares and white squares have -0.4 reward.

Policy iteration and value iteration is executed in the similar way as done earlier. The results for which are discussed as follows:

Table 7. Configuration for grid model with scale = 6

|  |  |
| --- | --- |
| Policy Iteration  Scale = 6  Discount Factor = 0.99  K = 50 | Value Iteration  Scale = 6  Discount Factor = 0.99  Utility Upper Bound = 100.00  Max Reward (Rmax) = 1.0  Constant 'c' = 0.1  Epsilon Value (c \* Rmax) = 0.1  Convergence Threshold = 0.00101 |

Chart

Description automatically generatedChart

Description automatically generatedFigure 18. Plot of utilities against iterations for 36\*36 = 1296 states for Value Iteration

Figure 19. Plot of utilities against iterations for 36\*36 = 1296 states for Policy Iteration

The optimal policy and the utility values can be found in the text files “CZ4046-Assignment1\GridWorld\results\complex grid results\config\_results\_value\_6.txt” and “CZ4046-Assignment1\GridWorld\results\complex grid results\value\_iteration\_utilities\_scale\_6.csv” (for value iteration) and “CZ4046-Assignment1\GridWorld\results\complex grid results\config\_results\_policy\_6.txt” and “CZ4046-Assignment1\GridWorld\results\complex grid results\policy\_iteration\_utilities\_scale\_6.csv” (for policy iteration).

Both the algorithms are not able to learn the optimal policy. This maybe because the number of states has been increased sufficiently to 36\*36. Therefore, the convergence is reached before the optimal policy can be learned. This can be avoided using a higher discount factor, which would enable the algorithm to run for more iterations.

1. **Appendix**

Reference:

[1] Michael Wooldridge, *An Introduction to MultiAgent Systems,* John Wiley&Sons, 2009.