

CE/CZ4046 – Intelligent Agents

Assignment 1: Agent Decision Making

Name - Garg Astha Matriculation No. - U1923971H

Table of Contents

1. Problem Statement	3
2. Organization of the code	4
3. Assumption	ϵ
4. Setting up the Grid	ϵ
5. Value Iteration	ϵ
6. Policy Iteration	14
7. Complex Grid Environment	21
8. Appendix	30

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.

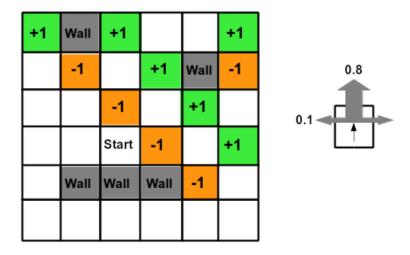


Figure 1. Question (from the Assignment document)

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.

2. 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.

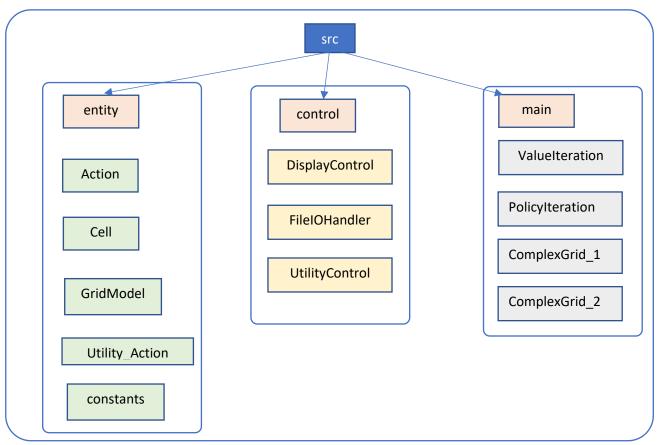


Figure 2. Code organization

Entity package:

- 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.

3. 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.

4. 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.

5. 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]:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$
.

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 $U_i(s)$ be

the utility value for state 's' at the i^{th} iteration. The iteration step is called Bellman Update, the equation for which is [1]:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

Equation 2. Bellman Update Equation [1]

```
\begin{array}{l} \textbf{function Value-Iteration}(mdp,\epsilon) \ \textbf{returns} \ \text{a utility function} \\ \textbf{inputs}: \ mdp, \ \text{an MDP with states} \ S, \ \text{actions} \ A(s), \ \text{transition model} \ P(s'\mid s,a), \\ \text{rewards} \ R(s), \ \text{discount} \ \gamma \\ \epsilon, \ \text{the maximum error allowed in the utility of any state} \\ \textbf{local variables}: \ U, \ U', \ \text{vectors of utilities for states in} \ S, \ \text{initially zero} \\ \delta, \ \text{the maximum change in the utility of any state in an iteration} \\ \textbf{repeat} \\ U \leftarrow U'; \ \delta \leftarrow 0 \\ \textbf{for each state} \ s \ \textbf{in} \ S \ \textbf{do} \\ U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'\mid s,a) \ U[s'] \\ \textbf{if} \ |U'[s] - U[s]| > \delta \ \textbf{then} \ \delta \leftarrow |U'[s] - U[s]| \\ \textbf{until} \ \delta < \epsilon(1-\gamma)/\gamma \\ \textbf{return} \ U \\ \end{array}
```

Figure 3. Value Iteration Algorithm [1]

a. 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.

b. 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*).

The code snippets are in the appendix – Code 1, Code 2.

c. Results

i. Plot of optimal policy

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

Util: Max I Const Epsi:	ount Fac ity Uppe Reward(R tant 'c' lon Valu ergence	r Bound max) e(c * Rr	: : max) :	1.6 20 20	0.00 0 .0	Utili Max F Const Epsil	ount Fact ity Upper Reward(Rm cant 'c' Lon Value ergence T	Bound ax)	•	1.0 1.0 1.0	.00	Util Max Cons Epsi	count Factity Upper Reward(Ri Stant 'c' Lon Valu Vergence	r Bound max) e(c * Rm		0.99 100. 1.0 0.1 0.1 0.00	. 00
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^	Wall < < Wall <	< < < < Wall	<	< Wall < ^ ^	^ ^ ^	^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^	Wall < < Wall <	< < < < Wall	< < ^ Wall	<pre> </pre> <pre> Wall </pre> <pre> ^ ^ ^ ^</pre>	^	^ ^ ^	Wall < < < Wall <	<	< < ^ Wall	<pre> Vall < ^ ^ ^ ^</pre>	^

ii. Utilities of all states

Table 2. Utilities of all states

c = 20.0	c = 1.0	c = 0.1
Iterations: 161	Iterations: 459	Iterations: 688
(0, 0): 79.972297	(0, 0): 98.997881	(0, 0): 99.899682
(0, 1): 78.365659	(0, 1): 97.391243	(0, 1): 98.293044
(0, 2): 76.920797	(0, 2): 95.946382	(0, 2): 96.848182
(0, 3): 75.526136	(0, 3): 94.551720	(0, 3): 95.453521
(0, 4): 74.284817	(0, 4): 93.310401	(0, 4): 94.212201
(0, 5): 72.909772	(0, 5): 91.935356	(0, 5): 92.837156
(1, 1): 75.855315	(1, 1): 94.880899	(1, 1): 95.782699
(1, 2): 75.558725	(1, 2): 94.584309	(1, 2): 95.486110
(1, 3): 74.424791	(1, 3): 93.450375	(1, 3): 94.352176
(1, 5): 71.701075	(1, 5): 90.726659	(1, 5): 91.628460
(2, 0): 75.017763	(2, 0): 94.043339	(2, 0): 94.945139
(2, 1): 74.517297	(2, 1): 93.542880	(2, 1): 94.444680
(2, 2): 73.266725	(2, 2): 92.292309	(2, 2): 93.194110
(2, 3): 73.204843	(2, 3): 92.230427	(2, 3): 93.132227
(2, 5): 70.507449	(2, 5): 89.533033	(2, 5): 90.434834
(3, 0): 73.847307	(3, 0): 92.872882	(3, 0): 93.774683
(3, 1): 74.370014	(3, 1): 93.395596	(3, 1): 94.297397
(3, 2): 73.148572	(3, 2): 92.174154 (3, 3): 90.113138	(3, 2): 93.075955
(3, 3): 71.087556 (3, 5): 69.328707	(3, 5): 88.354291	(3, 3): 91.014939
(4, 0): 72.626922	(4, 0): 91.652496	(3, 5): 89.256091 (4, 0): 92.554296
(4, 2): 73.074669	(4, 2): 92.100250	(4, 2): 93.002051
(4, 3): 71.786709	(4, 3): 90.812288	(4, 3): 91.714089
(4, 4): 69.520717	(4, 4): 88.546294	(4, 4): 89.448095
(4, 5): 68.541404	(4, 5): 87.566980	(4, 5): 88.468781
(5, 0): 73.300897	(5, 0): 92.326384	(5, 0): 93.228185
(5, 1): 70.890329	(5, 1): 89.915805	(5, 1): 90.817605
(5, 2): 71.767184	(5, 2): 90.792752	(5, 2): 91.694553
(5, 3): 71.860398	(5, 3): 90.885966	(5, 3): 91.787767
(5, 4): 70.539080	(5, 4): 89.564647	(5, 4): 90.466448
(5, 5): 69.270006	(5, 5): 88.295572	(5, 5): 89.197373
		i

The results are stored as follows:

• Configuration, utility values and plot of optimal policy

- \circ C = 0.1, CZ4046-
 - $Assignment 1 \ Cz4046-Assignment 1 \ Cz404$
- \circ C = 1.0, CZ4046-
 - $Assignment 1 \ Cz4046-Assignment 1 \ Cz404$
- $\label{eq:config} $$ C = 20.0, CZ4046-$$ 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
 - \circ C = 0.1, CZ4046-Assignment1\GridWorld\results\images\value_c_01.png
 - \circ C = 1.0, CZ4046-Assignment1\GridWorld\results\images\value_c_1.png
 - \circ C = 20.0, CZ4046-Assignment1\GridWorld\results\images\value_c_20.png

iii. 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.

d. Plot of utility estimates as function of number of iterations

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

c_20 (Value Iteration)

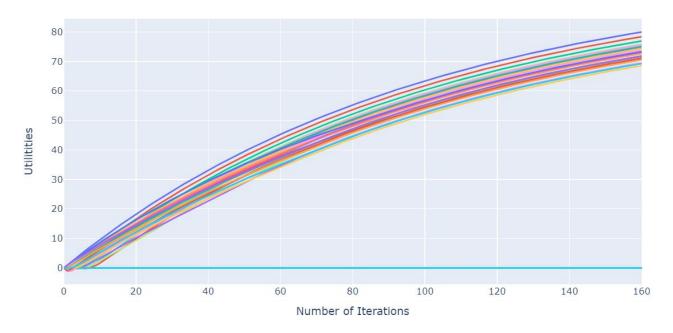


Figure 4. Value iteration for c = 20

c_1 (Value Iteration)

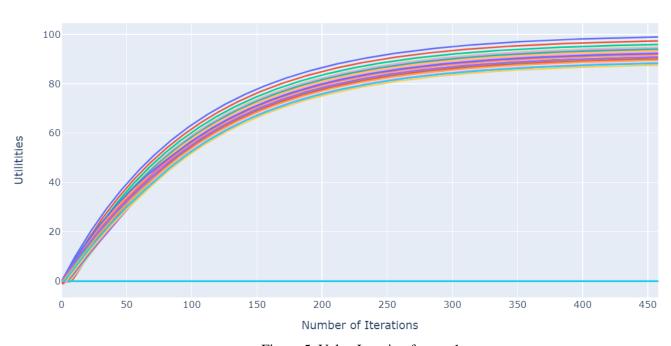


Figure 5. Value Iteration for c = 1

c_01 (Value Iteration)

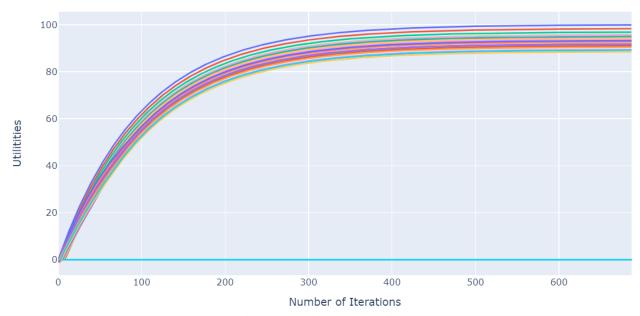


Figure 6. Value Iteration for c = 0.1

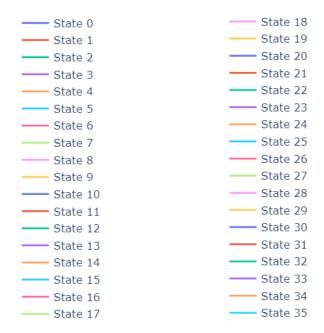


Figure 7. Legend for the plots

6. 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 π_0 [1]:

- **Policy evaluation**: given a policy π_i , calculate $Ui = U \pi_i$, the utility of each state if π_i were to be executed.
- **Policy improvement**: Calculate a new MEU policy π_{i+1} , using one-step look-ahead based on U_i .

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 π_i) to the utilities of its neighbours [1]:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi_i(s)) U_i(s')$$
.

Equation 3. Simplified Bellman Equation

```
\begin{array}{c} \textbf{function Policy-Iteration}(mdp) \ \textbf{returns} \ \textbf{a} \ \textbf{policy} \\ \textbf{inputs}: \ mdp, \ \textbf{an MDP} \ \textbf{with states} \ S, \ \textbf{actions} \ A(s), \ \textbf{transition model} \ P(s' \mid s, a) \\ \textbf{local variables}: \ U, \ \textbf{a} \ \textbf{vector of utilities} \ \textbf{for states} \ \textbf{in} \ S, \ \textbf{initially zero} \\ \pi, \ \textbf{a} \ \textbf{policy vector indexed} \ \textbf{by state}, \ \textbf{initially random} \\ \textbf{repeat} \\ U \leftarrow \text{Policy-Evaluation}(\pi, U, mdp) \\ unchanged? \leftarrow \text{true} \\ \textbf{for each state} \ s \ \textbf{in} \ S \ \textbf{do} \\ \textbf{if} \ \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \ U[s'] \ > \sum_{s'} P(s' \mid s, \pi[s]) \ U[s'] \ \textbf{then do} \\ \pi[s] \leftarrow \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) \ U[s'] \\ unchanged? \leftarrow \text{false} \\ \textbf{until} \ unchanged? \\ \textbf{return} \ \pi \\ \end{array}
```

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].

a. Set up

The k values used for the policy iteration are as follows (constants.java): k = [10, 30, 50]

b. 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.

The code snippets are in the appendix – Code 3, Code 4.

c. Results

i. Plot of optimal policy

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

K =	10					K =	= 30					K =	= 50				
^ ^ ^	Wall < < Wall <	<	< < ^ ^ Wall <	< Wall < ^ ^ ^	^	^ ^ ^	Wall < < Wall <	<	< < ^ ^ Wall	< Wall < ^ ^ ^ ^	^	^ ^ ^	Wall < < < Wall <	<	< < ^ ^ Wall	< Wall < ^ ^	^

ii. Utilities of all states

Table 4. utilities of all states

The results are stored as follows:

- Configuration, utility values and plot of optimal policy
 - \circ k = 10, CZ4046-

 $Assignment 1 \ Cz4046-Assignment 1 \ Cz404$

- k = 30, CZ4046 Assignment1\GridWorld\results\k_30\config_results_policy.txt
 CZ4046-Assignment1\GridWorld\results\k_30\policy_iteration_utilities.csv
- Plot of utilities of the states against the number of iterations
 - \circ K = 10, CZ4046-Assignment1\GridWorld\results\images\policy_k_10.png
 - \circ K = 30, CZ4046-Assignment1\GridWorld\results\images\policy_k_30.png
 - \circ K = 50, CZ4046-Assignment1\GridWorld\results\images\policy_k_50.png

iii. 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.

d. Plot of utility estimates as function of number of iterations

k_10 (Policy Iteration)

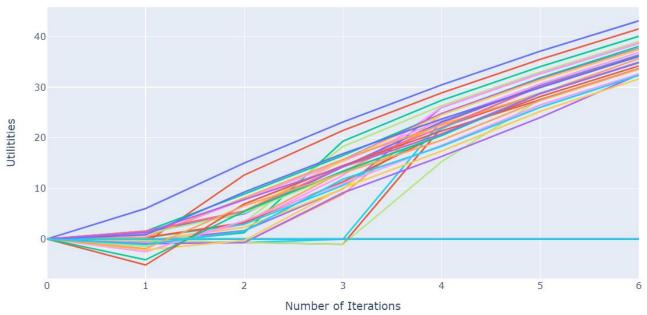


Figure 9. Policy Iteration for k = 10

k_30 (Policy Iteration)

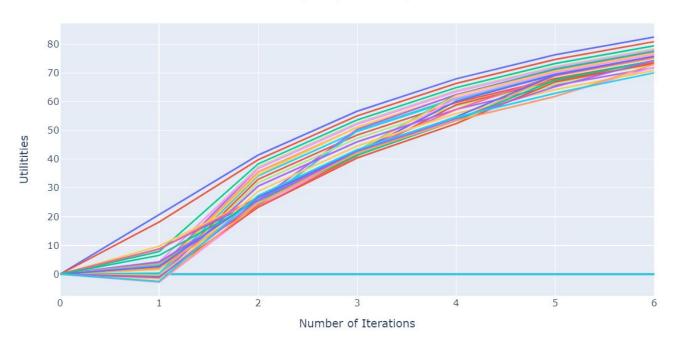


Figure 10. Policy Iteration for k = 30

k_50 (Policy Iteration)

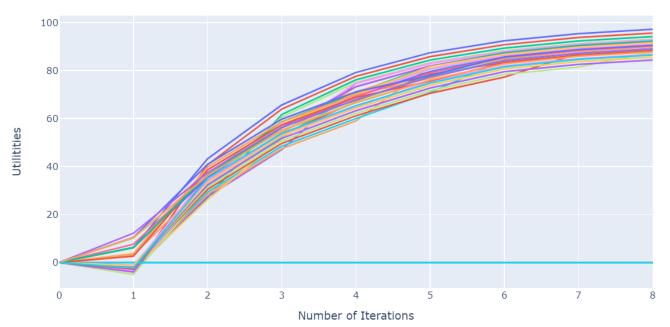


Figure 11. Policy Iteration for k = 50

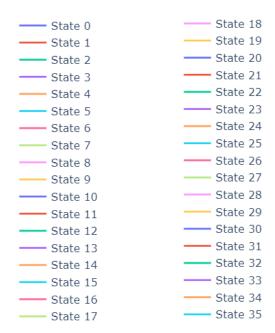


Figure 12. Legend for the plots

7. 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.

```
Exception in thread "main" java.lang.<u>OutOfMemoryError</u> Create breakpoint: Java heap space at main.ComplexGrid_2.initializeUtilities(<a href="ComplexGrid_2.java:107">ComplexGrid_2.java:107</a>) at main.ComplexGrid_2.runValueIteration(<a href="ComplexGrid_2.java:60">ComplexGrid_2.java:60</a>) at main.ComplexGrid_2.main(<a href="ComplexGrid_2.java:23">ComplexGrid_2.java:23</a>)

Process finished with exit code 1
```

Figure 13. Exception in java

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

1	1				1												
1.0	1.0	1.0	Wall	Wall	Wall	1.0	1.0	1.0							1.0	1.0	1.0
1.0	1.0	1.0	Wall	 Wall	Wall	1.0	1.0	1.0				 			1.0	1.0	1.0
1.0	1.0	1.0	 Wall	 Wall	Wall	1.0	1.0	1.0							1.0	1.0	1.0
			-1.0	-1.0	-1.0			 	1.0	1.0	1.0	Wall	Wall	Wall	-1.0	-1.0	-1.0
			-1.0	-1.0	-1.0			 	1.0	1.0	1.0	Wall	Wall	Wall	-1.0	-1.0	-1.0
			-1.0	-1.0	-1.0			 	1.0	1.0	1.0	Wall	Wall	Wall	-1.0	-1.0	-1.0
			 	 	 	-1.0	-1.0	-1.0				1.0	1.0	1.0			
			 	 	 	-1.0	-1.0	-1.0				1.0	1.0	1.0			
			 			-1.0	-1.0	-1.0				1.0	1.0	1.0			
			 	 		Start		 	-1.0	-1.0	-1.0				1.0	1.0	1.0
			 	 	 			 	-1.0	-1.0	-1.0			 	1.0	1.0	1.0
			 	 	 			 	-1.0	-1.0	-1.0			 	1.0	1.0	1.0
			Wall	 Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	-1.0	-1.0	-1.0			
			Wall	 Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	-1.0	-1.0	-1.0			
			Wall	 Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	-1.0	-1.0	-1.0			

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	

>	>	^	Wall	Wall	Wall	٨	<	<	<	<	<	>	>	>	>	>	٨
>	>	^	Wall	Wall	Wall	٨	<	<	<	<	v	>	>	>	>	>	٨
^	^	^	Wall	Wall	Wall	۸	۸	۸	<	v	v	<	>	>	۸	٨	^
^	^	^	<	<	>	^	^	۸	>	V	V	Wall	Wall	Wall	۸	۸	۸
^	^	^	<	>	>	۸	۸	>	>	>	>	Wall	Wall	Wall	۸	٨	^
^	^	^	<	<	>	۸	>	>	>	^	^	Wall	Wall	Wall	V	V	V
^	^	^	<	<	^	۸	۸	۸	٨	۸	۸	>	۸	<	<	<	V
^	^	^	<	<	^	۸	^	>	۸	^	>	>	^	<	<	V	V
^	^	^	۸	۸	^	۸	>	>	٨	>	>	۸	۸	۸	V	V	V
^	^	^	^	۸	^	>	>	۸	۸	۸	^	^	^	۸	>	V	V
^	^	^	^	^	^	>	٨	^	^	>	>	^	^	>	>	>	>
^	^	^				> Wall			> Wall	> Wall	> Wall	^	>	>	>	^	^
^	^	^	Wall Wall	Wall Wall	Wall Wall	Wall	Wall Wall	Wall Wall	Wall	Wall	Wall	^	^	>	^	^	^
^	^	^	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	Wall	>	>	>	^	٨	^
^	۸	۸	Wall	Wall	> wall	> wall	> wall	> wall	> wall	> wall	wall	>	>	>	۸	٨	٨
^	٨	^	ζ.	>	>	>	>	>	>	>	>	>	>	>	۸	٨	^
٨	٨	٨	۸	>	>	>	>	>	>	>	>	>	>	>	۸	٨	۸
99.986	99.986	99.986	00.000	00.000	00.000	99.983	99.983	99.981	98.683	97.416	96.260	96.144	97.401	98.685	99.984	99.986	99.986
99.986	99.986	99.986	00.000	00.000	00.000	99.983	99.982	99.968	98.674	97.518	97.148	96.211	97.373	98.661	99.969	99.984	99.986
99.986	99.986	99.986	00.000	00.000	00.000	99.981	99.968	99.841	98.683	98.628	98.497	97.108	97.280	98.548	99.828	99.969	99.984
144140000000000000000000000000000000000																	
98.686	98.660	98.396	95.757	93.172	95.752	98.390	98.644	98.680	99.808	99.933	99.948	00.000	00.000	00.000	97.351	97.464	97.488
97.398	97.332	96.873	94.419	92.002	94.421	96.879	97.437	98.585	99.914	99.946	99.950	00.000	00.000	00.000	94.901	94.994	95.022
96.119	96.007	95.404	93.090	90.880	93.067	95.478	96.731	98.217	99.758	99.928	99.948	00.000	00.000	00.000	95.731	94.484	93.834
133000000000000000000000000000000000000																	
94.852	94.702	94.078	92.751	91.363	91.926	93.056	94.312	95.848	98.219	98.605	98.775	99.947	99.949	99.948	98.358	96.865	96.200
93.597	93.415	92.784	91.576	90.377	90.701	90.789	92.019	94.296	96.727	97.393	98.605	99.928	99.946	99.933	98.610	97.485	97.383
92.355	92.148	91.516	90.393	89.341	89.424	88.733	90.656	92.926	95.343	96.726	98.219	99.758	99.914	99.804	98.651	98.639	98.650

91.125	90.899	90.273	89.229	88.297	88.342	89.058	90.399	91.706	92.926	94.296	95.848	98.217	98.581	98.645	99.804	99.932	99.948
89.909	89.670	89.054	88.084	87.253	87.283	88.058	89.223	90.399	90.656	92.017	94.308	96.724	97.371	98.581	99.913	99.946	99.949
88.707	88.458	87.856	86.955	86.210	86.239	87.036	88.057	89.053	88.677	90.812	93.071	95.354	96.723	98.216	99.757	99.928	99.947
87.521	87.274	86.773	00.000	00.000	00.000	00.000	00.000	00.000	00.000	00.000	00.000	93.071	94.307	95.847	98.217	98.592	98.644
87.521	61.214	80.773	00.000	99.999	99.999	00.000	99.99	00.00	00.000	00.000	00.000	93.0/1	94.30/	95.84/	98.21/	98.592	98.044
86.350	86.113	85.693	00.000	00.000	00.000	00.000	00.000	00.000	00.000	00.000	00.000	90.812	92.016	94.285	96.710	97.257	97.353
85.196	84.973	84.618	00.000	00.000	00.000	00.000	00.000	00.000	00.000	00.000	00.000	88.758	90.752	92.943	95.253	95.928	96.071
000000000000000000000000000000000000000																	
84.057	83.838	83.415	82.199	81.024	81.536	82.656	83.795	84.955	86.136	87.340	88.567	89.818	91.229	92.612	93.936	94.620	94.801
82.931	82.712	82.250	81.178	80.214	81.219	82.296	83.388	84.494	85.615	86.751	87.904	89.073	90.261	91.452	92.651	93.332	93.543

Scale 3 (Policy Iteration)

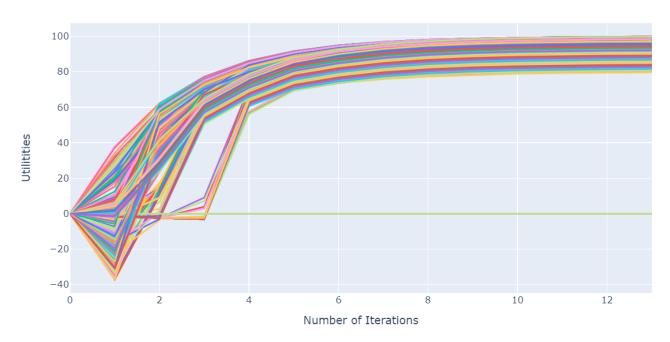


Figure 15. Plot of utilities against iterations for 18*18 = 324 states for Policy Iteration

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

	۸	٨	Wall	Wall	Wall	٨	<	<	<	<	<	>	>	>	>	>	′
	۸	٨	Wall	Wall	Wall	^	^	<	<	<	V	>	>	>	>	^	
	^	^	Wall	Wall	Wall	٨	۸	^	<	V	V	<	>	>	^	٨	
	۸	٨	<	<	>	٨	٨	^	>	V	V	Wall	Wall	Wall	^	٨	
`	۸	۸	<	>	>	٨	۸	>	>	>	>	Wall	Wall	Wall	^	٨	
`	۸	^	<	<	>	۸	>	>	>	۸	^	Wall	Wall	Wall	V	V	
	^	^	<	<	^	^	^	^	^	^	^	>	^	<	<	<	
	٨	^	<	<	۸	٨	٨	>	٨	٨	>	>	۸	<	<	V	
	۸	^	۸	۸	۸	٨	>	>	٨	>	>	۸	۸	۸	V	V	
	۸	^	۸	^	^	>	>	^	۸	۸	^	۸	۸	۸	>	V	
	^	٨	^	^	^	>	^	^	^	>	>	٨	۸	>	>	>	
`	٨	٨	٨	٨	^	>	٨	^	>	>	>	٨	>	>	>	٨	
١	۸	٨	Wall	Wall	Wall	٨	۸	^	^	٨							
	^	٨	Wall	Wall	Wall	۸	۸	>	^	^							
	^	۸	Wall	Wall	Wall	>	>	>	^	^							
	^	^	<	<	>	>	>	>	>	>	>	>	>	>	۸	^	
`	^	^	<	>	>	>	>	>	>	>	>	>	>	>	۸	^	
	^	^	۸	>	>	>	>	>	>	>	>	>	>	>	۸	^	
9.900	99.900	99.900	00.000	00.000	00.000	99.898	99.898	99.896	98.598	97.330	96.175	96.056	97.312	98.596	99.896	99.898	99
9.900	99.900	99.900	00.000	00.000	00.000	99.898	99.896	99.882	98.588	97.432	97.064	96.123	97.285	98.573	99.881	99.896	99
9.900	99.900	99.900	00.000	00.000	00.000	99.896	99.882	99.755	98.598	98.544	98.413	97.023	97.192	98.459	99.740	99.881	99
8.600	98.574	98.310	95.671	93.085	95.667	98.305	98.559	98.595	99.724	99.849	99.864	00.000	00.000	00.000	97.262	97.376	97
7.311				000000000			00000000000	220000000000		2000000000			50000000000				
	97.245	96.786	94.333	91.917	94.336	96.794	97.352	98.501	99.830	99.862	99.866	00.000	00.000	00.000	94.813	94.906	94
	95.921	95.317	93.003	90.794	92.982	95.393	96.647	98.133	99.830 99.674	99.844	99.864	00.000 00.000	00.000	00.000 00.000	94.813 95.647	94.906 94.399	94 93
4.766	95.921 94.615	95.317 93.991	93.003 92.665	90.794 91.277	92.982 91.841	95.393 92.971	96.647 94.228	98.133 95.765	99.830 99.674 98.135	99.844 98.521	99.864 98.691	00.000 00.000 99.863	99.865	00.000 00.000 99.864	94.813 95.647 98.274	94.906 94.399 96.781	94 93 96
4.766 3.510	95.921	95.317	93.003	90.794	92.982	95.393	96.647	98.133	99.830 99.674	99.844	99.864	00.000 00.000	00.000	00.000 00.000	94.813 95.647	94.906 94.399	94 93 96 97
4.766 3.510 2.268	95.921 94.615 93.329	95.317 93.991 92.697	93.003 92.665 91.489	90.794 91.277 90.291	92.982 91.841 90.616	95.393 92.971 90.705	96.647 94.228 91.934	98.133 95.765 94.212	99.830 99.674 98.135 96.643	99.844 98.521 97.309	99.864 98.691 98.521	00.000 00.000 99.863 99.844	99.865 99.862	00.000 00.000 99.864 99.848	94.813 95.647 98.274 98.526	94.906 94.399 96.781 97.401	94 93 96 97 98
4.766 3.510 2.268 1.038	95.921 94.615 93.329 92.061	95.317 93.991 92.697 91.429	93.003 92.665 91.489 90.306	90.794 91.277 90.291 89.254	92.982 91.841 90.616 89.339	95.393 92.971 90.705 88.648	96.647 94.228 91.934 90.572	98.133 95.765 94.212 92.842	99.830 99.674 98.135 96.643 95.259	99.844 98.521 97.309 96.642	99.864 98.691 98.521 98.135	00.000 00.000 99.863 99.844 99.674	99.865 99.862 99.830	00.000 00.000 99.864 99.848 99.720	94.813 95.647 98.274 98.526 98.567	94.906 94.399 96.781 97.401 98.555	94 93 96 97 98
4.766 3.510 2.268 1.038 9.823	95.921 94.615 93.329 92.061 90.813	95.317 93.991 92.697 91.429 90.187	93.003 92.665 91.489 90.306 89.143	90.794 91.277 90.291 89.254 88.211	92.982 91.841 90.616 89.339 88.257	95.393 92.971 90.705 88.648 88.974	96.647 94.228 91.934 90.572 90.315	98.133 95.765 94.212 92.842 91.622	99.830 99.674 98.135 96.643 95.259 92.842	99.844 98.521 97.309 96.642 94.212	99.864 98.691 98.521 98.135 95.764	00.000 00.000 99.863 99.844 99.674 98.132	99.865 99.862 99.830 98.497	99.864 99.848 99.720 98.561	94.813 95.647 98.274 98.526 98.567 99.720	94.906 94.399 96.781 97.401 98.555 99.848	94
4.766 3.510 2.268 1.038 9.823 8.621	95.921 94.615 93.329 92.061 90.813 89.583	95.317 93.991 92.697 91.429 90.187 88.967	93.003 92.665 91.489 90.306 89.143 87.997	90.794 91.277 90.291 89.254 88.211 87.166	92.982 91.841 90.616 89.339 88.257 87.197	95.393 92.971 90.705 88.648 88.974 87.974	96.647 94.228 91.934 90.572 90.315 89.139	98.133 95.765 94.212 92.842 91.622 90.315	99.830 99.674 98.135 96.643 95.259 92.842 90.572	99.844 98.521 97.309 96.642 94.212 91.933	99.864 98.691 98.521 98.135 95.764 94.224	99.863 99.844 99.674 98.132	99.865 99.862 99.830 98.497 97.287	90.000 90.000 99.864 99.848 99.720 98.561 98.497	94.813 95.647 98.274 98.526 98.567 99.720 99.830	94.906 94.399 96.781 97.401 98.555 99.848 99.862	94 93 96 97 98 99
4.766 3.510 2.268 1.038 9.823 8.621 7.434	95.921 94.615 93.329 92.061 90.813 89.583 88.371	95.317 93.991 92.697 91.429 90.187 88.967 87.770	93.003 92.665 91.489 90.306 89.143 87.997 86.868	90.794 91.277 90.291 89.254 88.211 87.166 86.124	92.982 91.841 98.616 89.339 88.257 87.197	95.393 92.971 96.705 88.648 88.974 87.974	96.647 94.228 91.934 90.572 90.315 89.139 87.973	98.133 95.765 94.212 92.842 91.622 90.315 88.969	99.830 99.674 98.135 96.643 95.259 92.842 90.572 88.593	99.844 98.521 97.309 96.642 94.212 91.933 90.728	99.864 98.691 98.521 98.135 95.764 94.224 92.987	99.863 99.844 99.674 98.132 96.639	99.865 99.862 99.830 98.497 97.287 96.639	99.864 99.848 99.720 98.561 98.497	94.813 95.647 98.274 98.526 98.567 99.720 99.830 99.674	94.906 94.399 96.781 97.401 98.555 99.848 99.862 99.844	94 93 96 97 98 99
4.766 3.510 2.268 1.038 9.823 8.621 7.434 6.263	95.921 94.615 93.329 92.061 90.813 89.583 88.371 87.187	95.317 93.991 92.697 91.429 90.187 88.967 87.770 86.686	93.003 92.665 91.489 90.306 89.143 87.997 86.868 00.000	90.794 91.277 90.291 89.254 88.211 87.166 86.124 90.000	92.982 91.841 90.616 89.339 88.257 87.197 86.154	95.393 92.971 90.705 88.648 88.974 87.974 86.952 00.000	96.647 94.228 91.934 90.572 90.315 89.139 87.973 00.000	98.133 95.765 94.212 92.842 91.622 90.315 88.969 00.000	99.830 99.674 98.135 96.643 95.259 92.842 90.572 88.593 00.000	99.844 98.521 97.309 96.642 94.212 91.933 90.728	99.864 98.691 98.521 98.135 95.764 94.224 92.987 00.000	99.863 99.844 99.674 98.132 96.639 95.270	99.865 99.862 99.830 98.497 97.287 96.639	00.000 00.000 99.864 99.720 98.561 98.497 98.132 95.763	94.813 95.647 98.274 98.526 98.567 99.720 99.830 99.674	94.996 94.399 96.781 97.401 98.555 99.848 99.862 99.844 98.508	94 93 96 97 98 99 99 98 97
4.766 3.510 2.268 1.038 9.823 8.621 7.434 6.263 5.109 3.970	95.921 94.615 93.329 92.061 90.813 89.583 88.371 87.187 86.026 84.886	95.317 93.991 92.697 91.429 90.187 88.967 87.770 86.686 85.606 84.531 83.328	93.003 92.665 91.489 90.306 89.143 87.997 86.868 00.000 00.000	90.794 91.277 90.291 89.254 88.211 87.166 86.124 90.000	92.982 91.841 90.616 89.339 88.257 87.197 86.154 00.000	95.393 92.971 90.705 88.648 88.974 87.974 86.952 90.000	96.647 94.228 91.934 90.572 90.315 89.139 87.973 00.000	98.133 95.765 94.212 92.842 91.622 90.315 88.969 00.000	99.830 99.674 98.135 96.643 95.259 92.842 90.572 88.593 90.000 90.000 86.052	99.844 98.521 97.309 96.642 94.212 91.933 90.728 90.000	99.864 98.691 98.521 98.135 95.764 94.224 92.987 00.000 00.000 88.483	90.000 99.863 99.844 99.674 98.132 96.639 95.270 92.987 90.728 88.674 89.735	99.865 99.862 99.830 98.497 97.287 96.639 94.224 91.932 99.668 91.145	00.000 00.000 99.864 99.720 98.561 98.497 98.132 95.763 94.201 92.859 92.528	94.813 95.647 98.274 98.526 98.567 99.720 99.830 99.674 98.133 96.627 95.169 93.852	94.996 94.399 96.781 97.491 98.555 99.848 99.862 99.844 98.508 97.173 95.844 94.536	94 93 96 97 98 99 99 99 98 97 95 94
14.766 13.510 12.268 11.038 19.823 18.621 17.434 16.263 15.109 13.970 12.845 11.733	95.921 94.615 93.329 92.061 90.813 89.583 88.371 87.187 86.026	95.317 93.991 92.697 91.429 90.187 88.967 87.770 86.686 85.606 84.531	93.003 92.665 91.489 90.306 89.143 87.997 86.868 90.000 90.000	90.794 91.277 90.291 89.254 88.211 87.166 86.124 90.000 90.000	92.982 91.841 90.616 89.339 88.257 87.197 86.154 00.000 00.000	95.393 92.971 90.705 88.648 88.974 87.974 86.952 00.000 00.000	96.647 94.228 91.934 90.572 90.315 89.139 87.973 00.000 00.000	98.133 95.765 94.212 92.842 91.622 90.315 88.969 00.000 00.000	99.830 99.674 98.135 96.643 95.259 92.842 90.572 88.593 00.000 00.000	99.844 98.521 97.309 96.642 94.212 91.933 90.728 00.000 00.000	99.864 98.691 98.521 98.135 95.764 94.224 92.987 00.000 00.000	90.000 99.863 99.844 99.674 98.132 96.639 95.270 92.987 90.728 88.674	99.865 99.862 99.830 98.497 97.287 96.639 94.224 91.932	00.000 00.000 99.864 99.720 98.561 98.497 98.132 95.763 94.201 92.859	94.813 95.647 98.274 98.526 98.567 99.720 99.830 99.674 98.133 96.627 95.169	94.996 94.399 96.781 97.481 98.555 99.848 99.862 99.844 98.588 97.173	94 93 96 97 98 99 99

Scale 3 (Value Iteration)

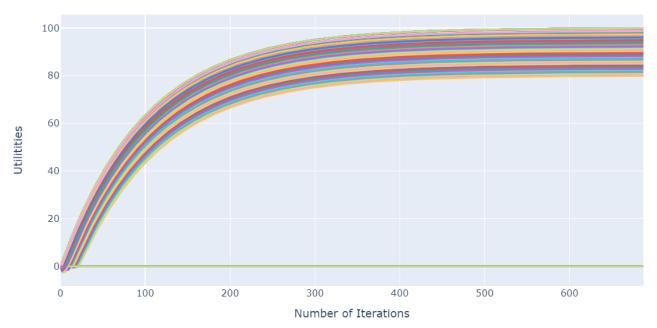


Figure 16. Plot of utilities against iterations for 18*18 = 324 states for Value Iteration

The optimal policy and the utility values can be found in the text files "CZ4046-Assignment1\GridWorld\results\complex grid results\config_results_value_3.txt" and "CZ4046-Assignment1\GridWorld\results\complex grid results\value_iteration_utilities_scale_3.csv" (for value iteration) and "CZ4046-Assignment1\GridWorld\results\complex grid results\config_results_policy_3.txt" and "CZ4046-Assignment1\GridWorld\results\complex grid results\complex grid results\policy_iteration_utilities_scale_3.csv" (for policy 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.

When scale = 6, the grid environment is initialized as follows:

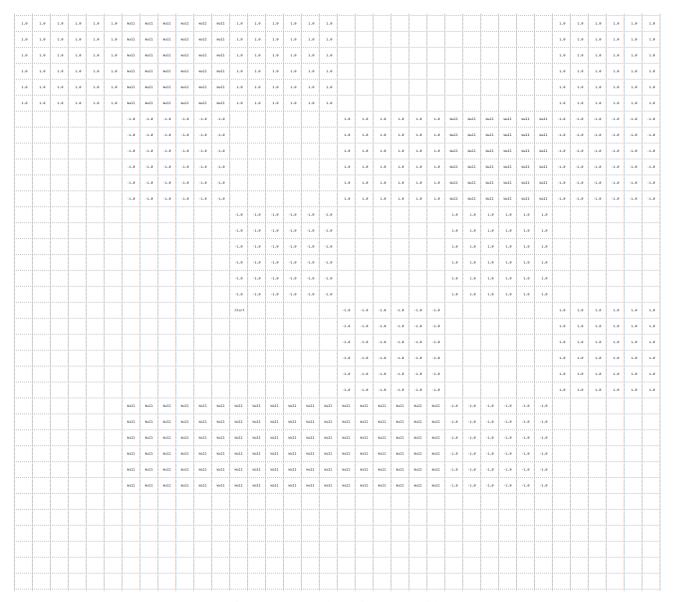


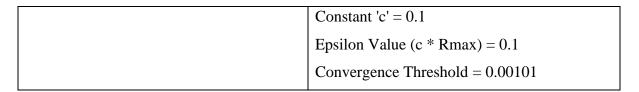
Figure 17. Complex grid model with scale = 6

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	Value Iteration
Scale = 6	Scale = 6
Discount Factor = 0.99	Discount Factor = 0.99
K = 50	Utility Upper Bound = 100.00
	Max Reward (Rmax) = 1.0



Scale 6 (Value Iteration)

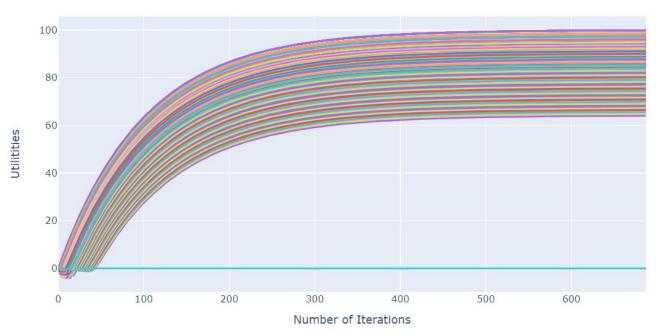


Figure 18. Plot of utilities against iterations for 36*36 = 1296 states for Value Iteration

Scale 6 (Policy Iteration)

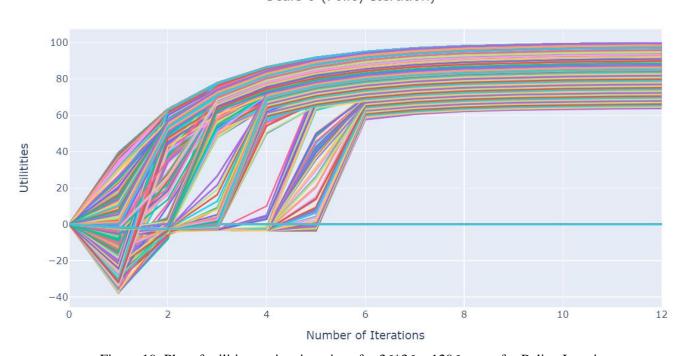


Figure 19. Plot of utilities against iterations for 36*36 = 1296 states for Policy Iteration

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\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.

8. Appendix

Code Snippets:

```
void runValueIteration(final Cell[][] grid) {
for (int col = 0; col < constants.NUM_COLS*SCALE; col++) {
   for (int row = 0; row < constants.NUM_ROWS*SCALE; row++) {</pre>
                          double updatedUtil = newUtilArr[col][row].getUtil();
double currentUtil = currUtilArr[col][row].getUtil();
```

Code 1. Value Iteration

```
public static Utility_Action getBestUtility(final int col, final int row, final Utility_Action[][] currUtilArr, final Cell[][] grid, int scale) {
   List<Utility_Action> utilities = new ArrayList<>();

   utilities.add(new Utility_Action(Action.UP, getActionUpUtility(col, row, currUtilArr, grid, scale)));
   utilities.add(new Utility_Action(Action.DOWN, getActionDownUtility(col, row, currUtilArr, grid, scale)));
   utilities.add(new Utility_Action(Action.LEFT, getActionLeftUtility(col, row, currUtilArr, grid, scale)));
   utilities.add(new Utility_Action(Action.RIGHT, getActionRightUtility(col, row, currUtilArr, grid, scale)));
   Collections.sort(utilities);
   return utilities.get(0);
}
```

Code 2. Function to get the action that maximizes subsequent utilities

```
lic static void runPolicyIteration(final Cell[][] grid) {
Utility_Action[][] currUtilArr = new Utility_Action[constants.NUM_COLS*SCALE][constants.NUM_ROWS*SCALE];
Utility_Action[][] newUtilArr = new Utility_Action[constants.NUM_COLS][constants.NUM_ROWS];
for (int col = 0; col < constants.NUM_COLS*SCALE; col++) {
   for (int row = 0; row < constants.NUM_ROWS*SCALE; row++) {</pre>
                                    case DDWW -> new Utility_Action(Action.DDWW, UtilityControl.getActionDownUtilityCol, row, newUtilArr, grid, SCALE));
case LEFT -> new Utility_Action(Action.LEFT, UtilityControl.getActionLeftUtility(col, row, newUtilArr, grid, SCALE));
```

Code 3. Policy Iteration

Code 4. Running simplified Bellman Equation k times

Reference:

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