

|  |
| --- |
| Assignment 1 : Agent Decision Making |
| **CZ4046 –****INTELLIGENT AGENTS** (Semester 2, AY 2019/2020)  WILSON THURMAN TENG U1820540H |

# 

# 1. Overview

## 1.1 Problem Description

In this assignment, two algorithms, namely “Value Iteration” & “Policy Iteration” are introduced to find an optimal policy for a given gridworld environment.

The environment is a 6x6 gridworld with reward, penalty, wall, as well as start tiles. The transition model depicts that the agent will have a 0.8 probability of going in it’s intended direction and 0.1 probability of going in the clockwise as well as anti-clockwise direction.

To conclude, the environment has a discrete state space and is stochastic in nature.

## 1.2 Optimal Policy

In this problem, the optimal policy would be to get to state (0,0) as fast as possible and move upwards. Moving upwards will result in the agent remaining stationary as the clockwise and anti-clockwise directions, left and right respectively when the agent is in (0,0) is unvisitable. This will cause the agent to stay in (0,0) indefinitely and hence accumulate the most rewards.

## 1.3 File Organisation

Important files to take note of:

* Src folder :   
  > Contains the code implementation to solve the given problem description.
* Results folder :   
  > Contains the outputs obtained from running the executable files in the src folder.
* *Interactive Graph Results.html* :   
  > Contains interactive graph plots of the state utility values against number of iterations.  
  > Toggle the *“Compare Data on hover”* as well as the *“Toggle spike lines”* options for a more interactive session with the graph.
* Experiments.pdf

> Contains experimentation on discount values to obtain optimal policy.

* *CZ4046 Assignment 1 Class Diagram.jpg* :   
  > Simplified Class diagram of code implementation.

## 1.4 Code Implementation



The java code is split into 3 main packages:

* Utils
* Classes
* Main

### 1.4.1 Utils Package

The **Utils package** contains 4 files.

1. *FileIO.java* manages the saving of State Utility values.
2. *ApplicationInput.java* stores constant values required by the two aforementioned algorithms. (E.g. Discount, Rmax, C, K, etc)
3. *CommonlyUsedMethods.java* contains common methods. (E.g. Generating random integers values, etc)
4. *DisplayHelper.java* under AgentHelperFunction Package contains useful methods to help evaluate the policies generated.

### 1.4.2 Classes Package

The **Classes Package** contains the bulk of the agent’s logic and keeps track of the utility values of states. To do this efficiently, it is further divided into 5 separate parts:

1. States Package:
   1. Constraints GridWorld.java to a finite number of state types.
2. *Action.java*:
   1. Keeps track of the transition state probabilities as well as the actions available to the agent.
3. *ActionUtilityPair.java*:
   1. Derived from States and Action.java. Used to keep track of the policy as well as utility value of a given state.
4. Agents Package:
   1. Contains the logic for the 2 algorithms which will be explored further in later sections.
5. *GridWorld.java*
   1. Stipulates the environment that the agent operates in.

### 1.4.3 Main Package

The **Main Package** contains 3 files:

1. *Main.java*
   1. The executable file for the original gridworld in this assignment.
2. *ComplexGridWorldExperiment1.java*
3. *ComplexGridWorldExperiment2.java*

The two ComplexGridWorldExperiment.java files will be explored further in the *BonusQuestion.java* section.

# 2. Value Iteration

## 2.1 Description

A set of constants are first instantiated in *ApplicationInput.java*. The important constants to take note of are **c** and **Rmax**. These constants are used to calculate **Epsilon** for Value Iteration (Epsilon = c\*Rmax). Epsilon can be interpreted as the maximum error allowable for a given state before accounting for the discount factor.

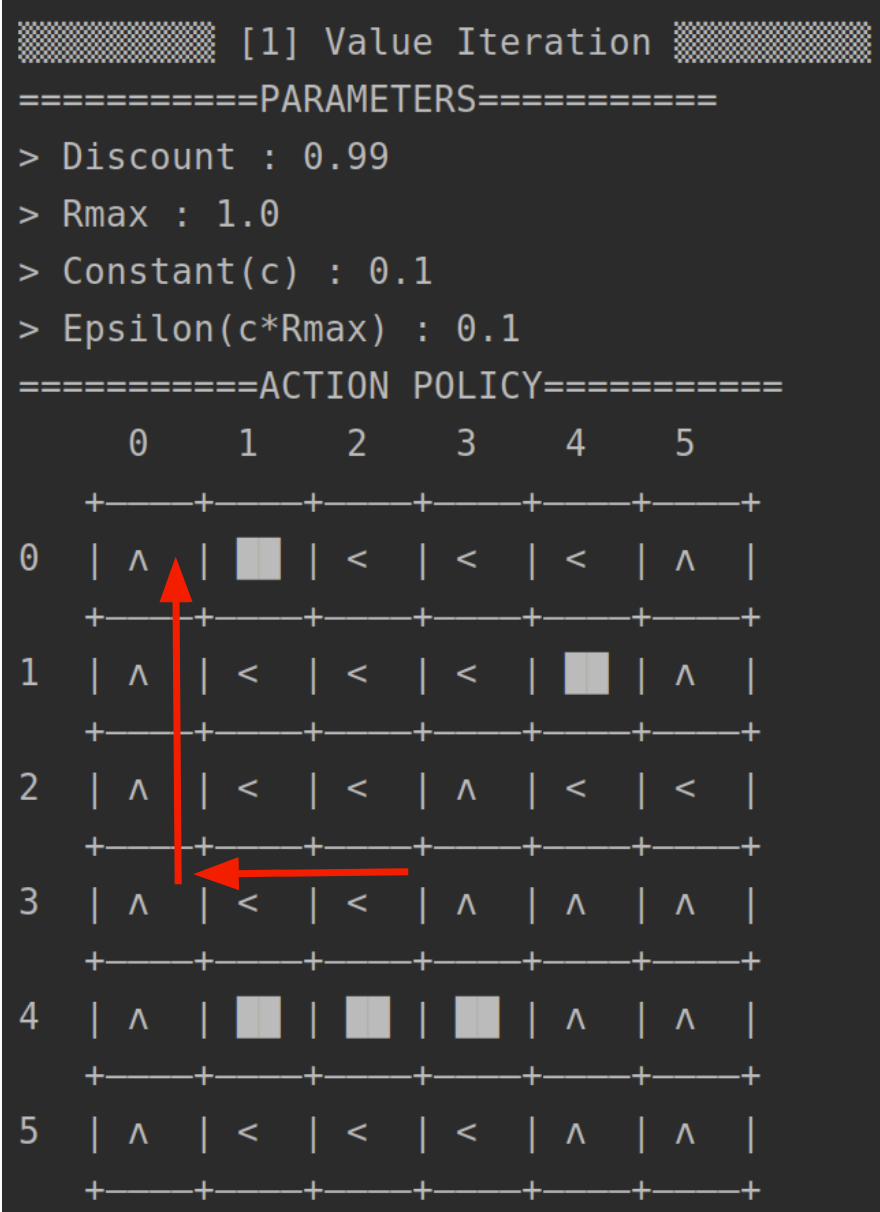
A GridWorld is then instantiated with states which contain reward values, as well as whether the state is visitable or not. (E.g. Wall is not visitable).

The *ValueIteration.java* class is called and the MDP representation, the GridWorld object are passed together with the epsilon value and discount value. Every state is then replaced with a StateActionUtilityPair. This is to keep track of the action with the highest utility for a given state.

An initial policy with no actions and 0 utility for every state is then initialised and Bellman Update will then be performed on state utility values. A bellman update is useful as it allows the value of a state to be represented by the values of other states. This bellman update continues until the maximum change in utility values amongst all the states in that particular iteration is less than the convergence criteria. The convergence criteria is calculated by: Epsilon \* ((1-discount) / discount).

Finally, an array of StateActionUtilityPairs where the state’s highest utility action along with its utility value is obtained.

## 2.2 Plot of Optimal Policy

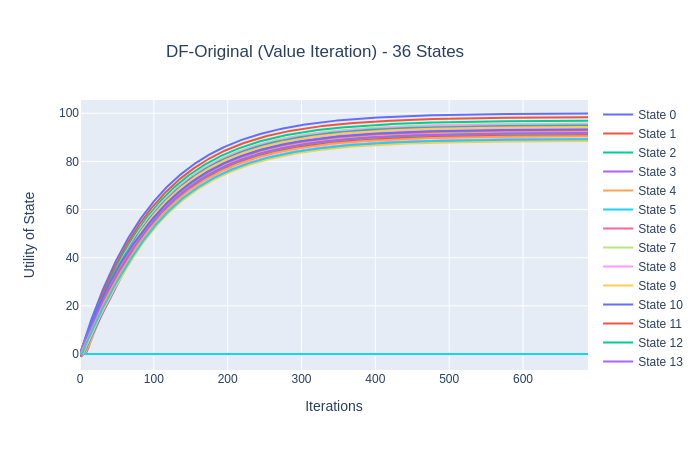


In the above image, it is clear that the agent has figured out the most optimal policy as explained in Section 1.2, which is to reach (0,0).

## 2.3 Utilities of all states

Refer to Appendix.

## 2.4 Plot of Utility Estimates as a function of the number of iterations



*Value Iteration Graph from “Interactive Graph Results.html”*



*Value Iteration Graph with Log(Number of Iterations) from “Interactive Graph Results.html”*

## 2.5 Value Iteration Findings

In this implementation, a total of 688 iterations are required to reach convergence.

Experimentation with c or Rmax parameters yielded no significant change in results. The optimal policy was still found, albeit at a faster or slower pace depending on whether epsilon was increased or decreased respectively.

In this graph, we observe that the relative positions in utility values of the different states stay the same after about iteration 200 which highlights that the optimal policy could have been reached much earlier but because of the epsilon value, the number of iterations is already pre-set.

Another interesting observation is that the utility values plotted after iteration 200 are very similar for most states and increase at roughly the same rate. This implies that the utility values for these states may be overestimated as the number of iterations increase.

# 3. Policy Iteration

## 3.1 Description

As discussed in the previous section on Value Iteration, it should be possible to achieve the optimal policy before the utility values reach convergence even if the utility values are inaccurate. Upon further inspection, this idea becomes intuitive. If an action for a particular state is clearly more optimal than the other options available, the exact value of the utility for that action holds little significance.

In this implementation, Modified Policy Iteration is used where a simplified form of Bellman Update calculates the approximate utility instead of the exact utility. This is to account for the growing state space in the bonus question’s implementation.

A set of constants are first instantiated in *ApplicationInput.java*. The important constant to take note of is K. K depicts the number of times simplified Bellman Update is performed on the policy during the policy evaluation step of the policy iteration algorithm for each iteration.

A GridWorld is then instantiated with states which contain reward values, as well as whether the state is visitable or not. (E.g. Wall is not visitable).

The *PolicyIteration.java* class is called and the MDP representation, the GridWorld object, is passed into the object. Every state is replaced with a StateActionUtilityPair. This is to keep track of the action with the highest utility for a given state.

An initial policy with random actions and 0 utility for every state is then initialised.

For each iteration, policy iteration is divided into 2 steps:

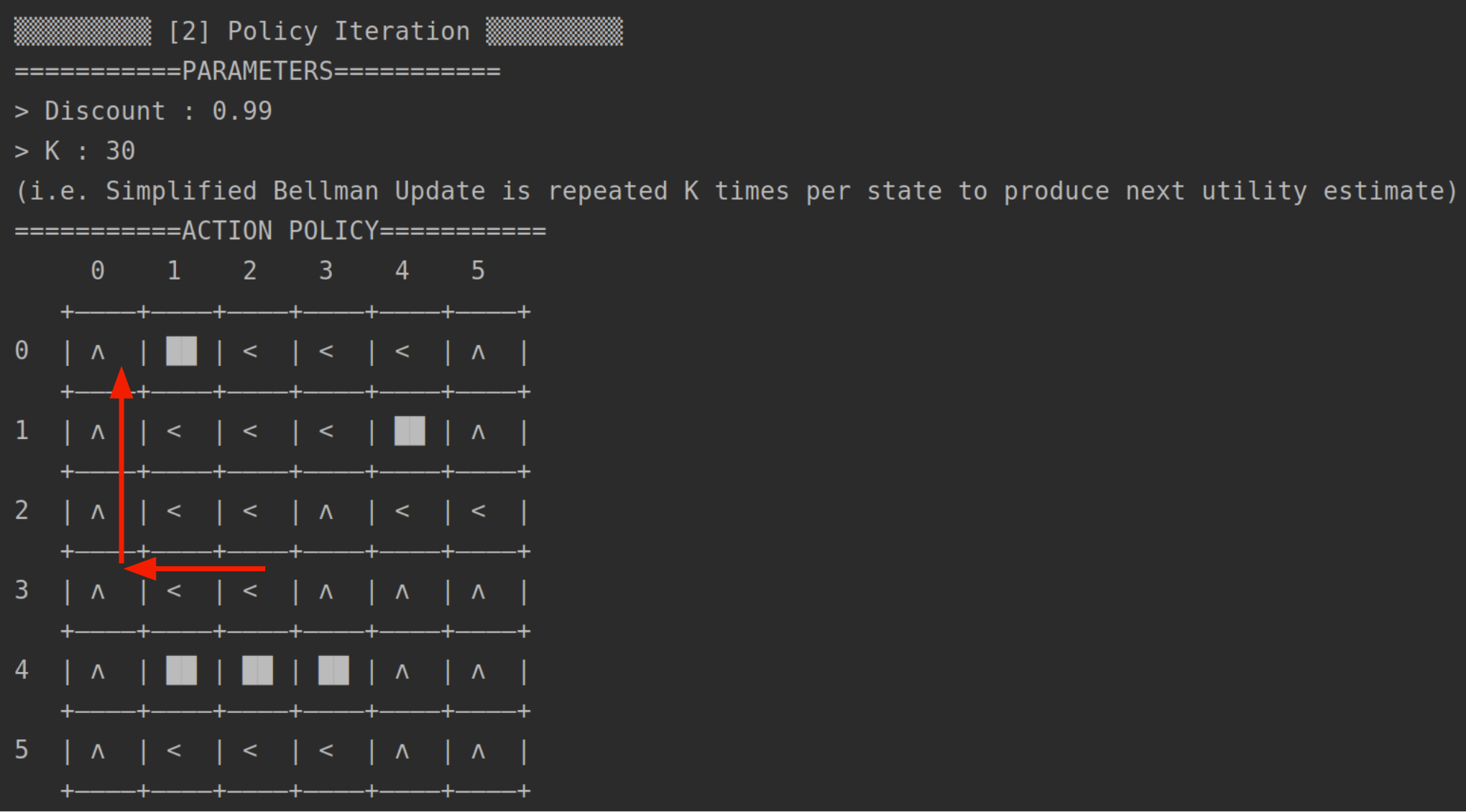
Step 1) Policy Evaluation : Perform simplified bellman update K times.

Step 2) Policy Improvement : Calculate new policy using one-step lookahead.

The algorithm continues until the policy is unchanged between two consective iterations.

Finally, an array of StateActionUtilityPairs where the state’s highest utility action along with its utility value is obtained.

## 3.2 Plot of Optimal Policy

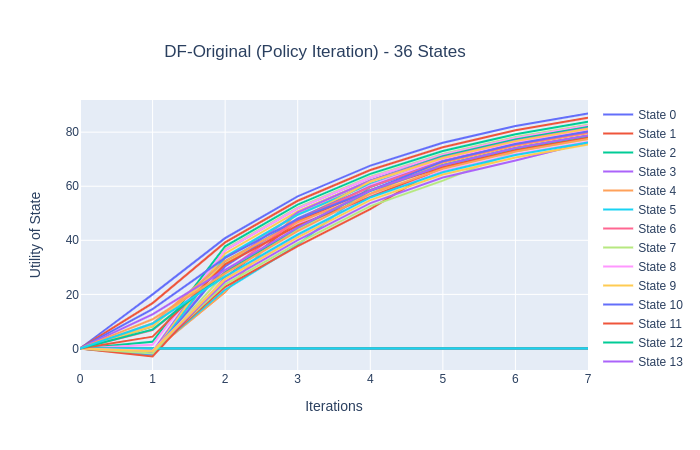


In the above image, it is clear that the agent has figured out the most optimal policy as explained in Section 1.2, which is to reach (0,0).

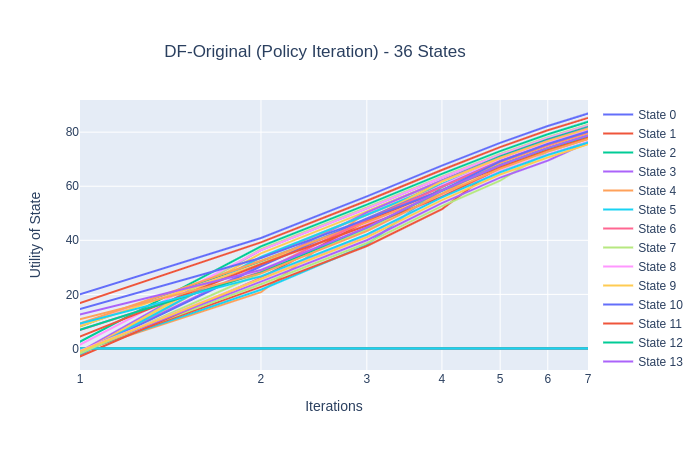
## 3.3 Utilities of all states

Refer to Appendix.

3.4 Plot of Utility Estimates as a function of the number of iterations



*Policy Iteration Graph from “Interactive Graph Results.html”*



*Policy Iteration Graph with Log(Number of Iterations) from “Interactive Graph Results.html”*

## 3.5 Policy Iteration Findings

In this implementation, a total of 7 iterations are required to reach convergence.

Experimentation with the K parameter yielded no significant change in results. The optimal policy was still found, albeit at a faster or slower pace depending on whether K was decreased or increased respectively.

We observe that Policy Iteration finished in significantly lesser iterations as compared to value iteration. However, each iteration took significantly longer as compared to value iteration.

# 4. Comparison between Value and Policy Iteration

Comparing both algorithms, we can see that value iteration converges faster than policy iteration to the same policy despite having a higher number of iterations. Upon further analysis, we observe that value iteration relies on finding the max utility value action every single iteration which would be an O(n) operation. This implies that value iteration would perform faster if the action space were to be small as is the case with the given gridworld problem with an action space of only 4. Policy iteration, which converges in a significantly lesser number of iterations is prefered when the action space is large.

We can also observe that policy iteration takes different numbers of iterations to converge after running it multiple times. This is so since the policy is instantiated with random actions before policy iteration is performed. We can hypothesize that convergence was achieved more quickly in these cases because the random policy initialised is already good to begin with.

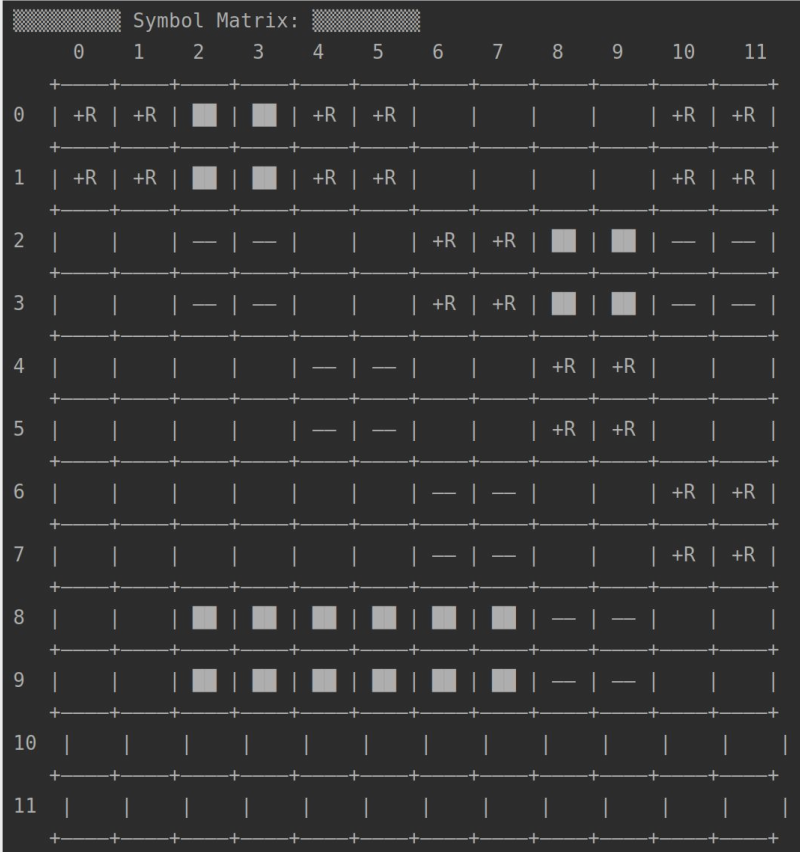
To conclude, with the current gridworld problem, policy iteration seems to be a better algorithm than value iteration.

## **5. Bonus Question**

To answer the question on how the number of states as well as the complexity can affect convergence, a more complex transition probability where a probability for the opposite intent was implemented. However, the results that the agent managed to learn the optimal policy as long as the probability of the intent probability was the highest was expected. Therefore, a scaled version of the original state space was implemented.

### 5.1 Scale=2

Symbol Matrix when scale = 2:



Optimal policy is to reach either (0:1, 0:1) or (0:1, 4:5) as they allow the agent to stay in the reward regions indefinitely.



*Scale =2 Diagram in Experiments.pdf*

After running both algorithms, both policies yielded similar policies and were unable to reach the optimal regions as mentioned earlier The agent instead prefered to take the shortest route to a reward state and risk being thrown off in the case of an unintended action.

However, when a higher discount factor of 0.9999 was used, it allowed the agent to prioritise long-term rewards and the optimal policy was achieved.

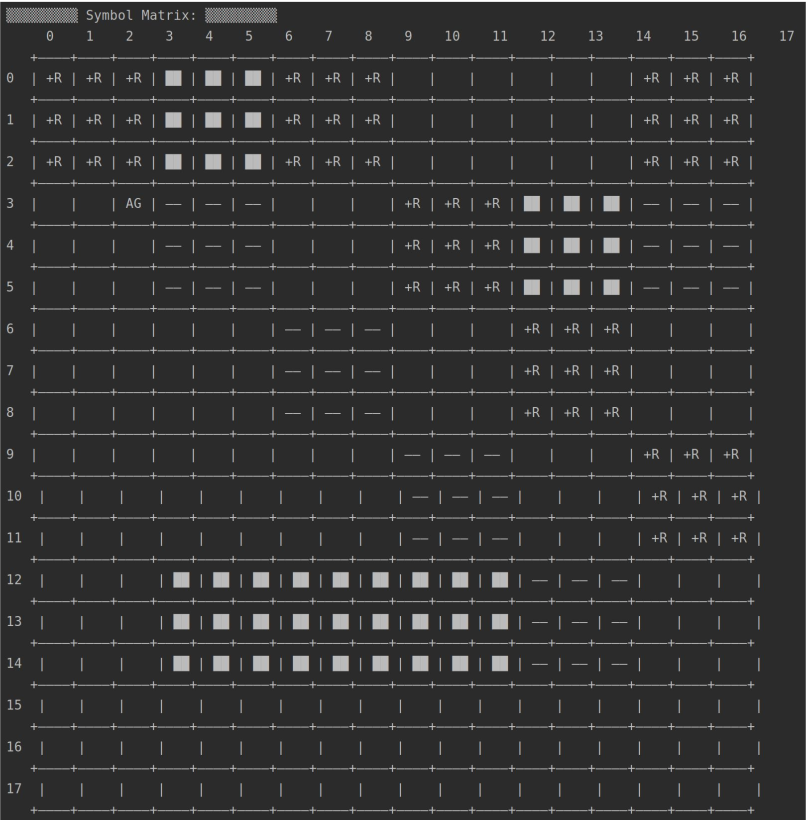
|  |  |
| --- | --- |
|  |  |
| Value Iteration (Scale=2, Discount=0.9999) | Policy Iteration (Scale=2, Discount=0.9999) |

We observe that the optimal policy was obtained for both after the discount factor was changed.

### 

### 5.2 Scale=3

Symbol Matrix for scale = 3:



|  |  |
| --- | --- |
|  |  |
| Value Iteration (Scale=3, Discount=0.999999) | Policy Iteration (Scale=3, Discount=0.999999) |

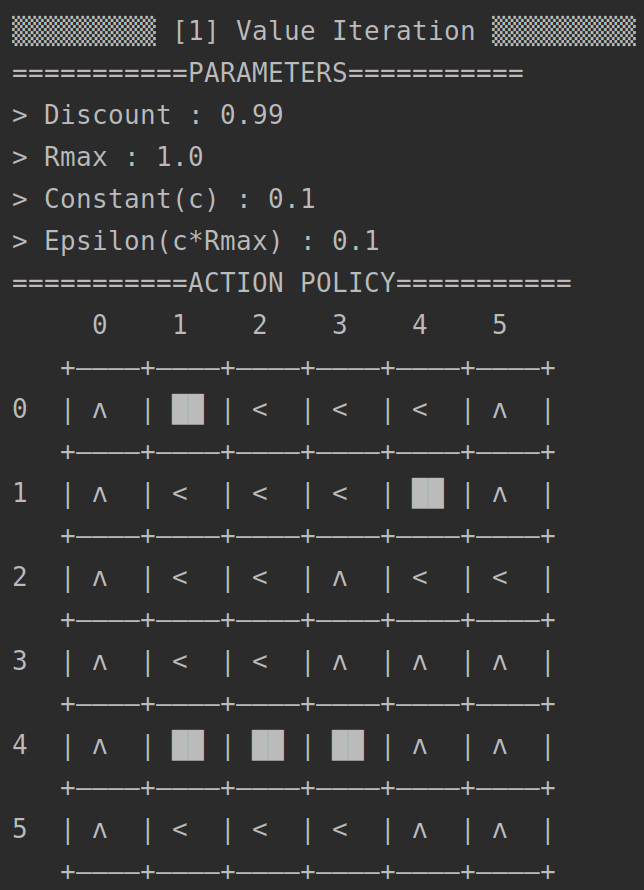
After learning that the discount rate was able to allow the agent to prioritise long-term rewards and hence achieve the optimal policy, the discount factor was set to 0.999999. However, the agent using value iteration was unable to achieve the optimal policy while the agent using policy iteration was only able to achieve the optimal policy on some occasions depending on the random policy initialised at first. This suggests that certain initial policies could cause the policy iteration agent to be stuck in a suboptimal policy.

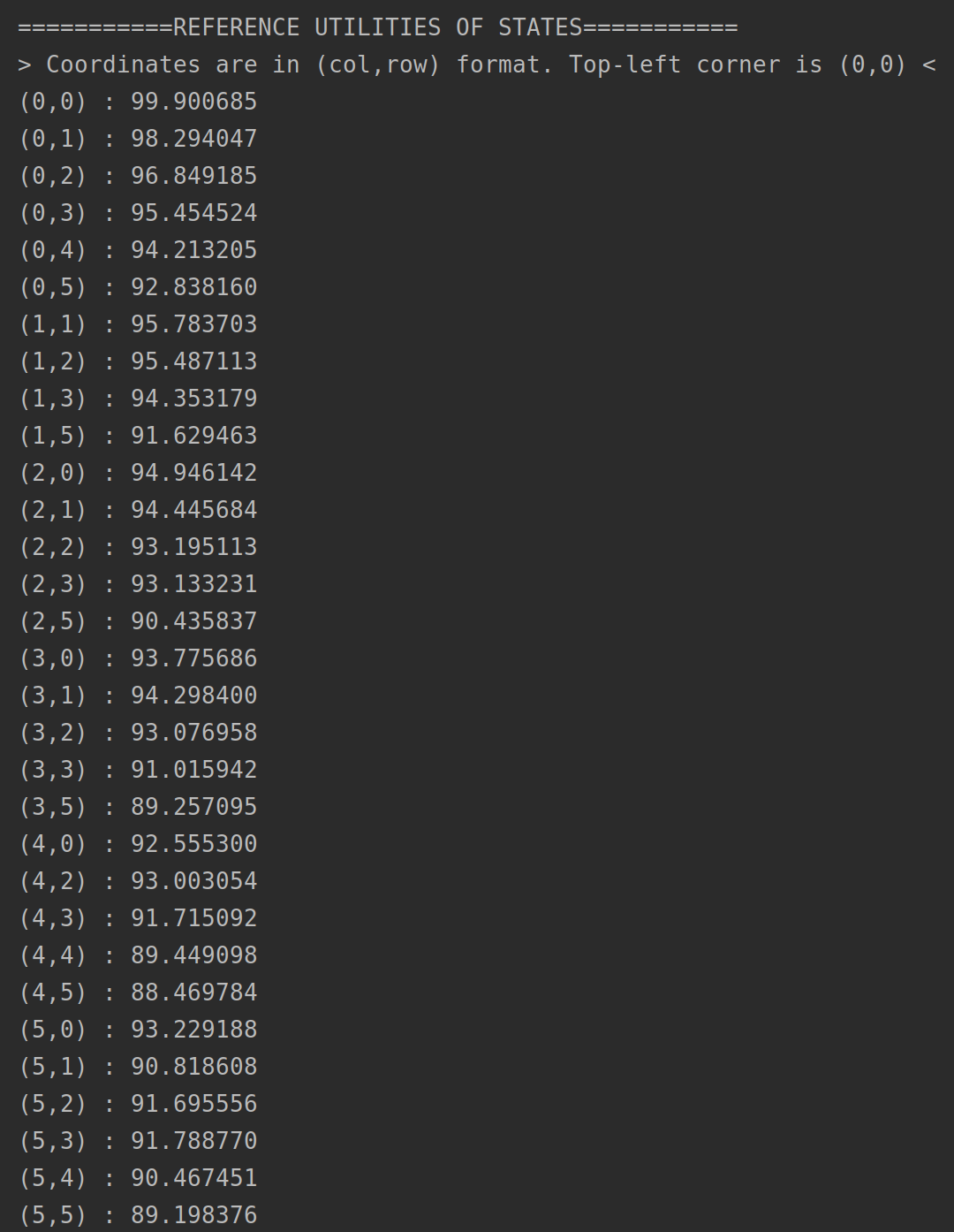
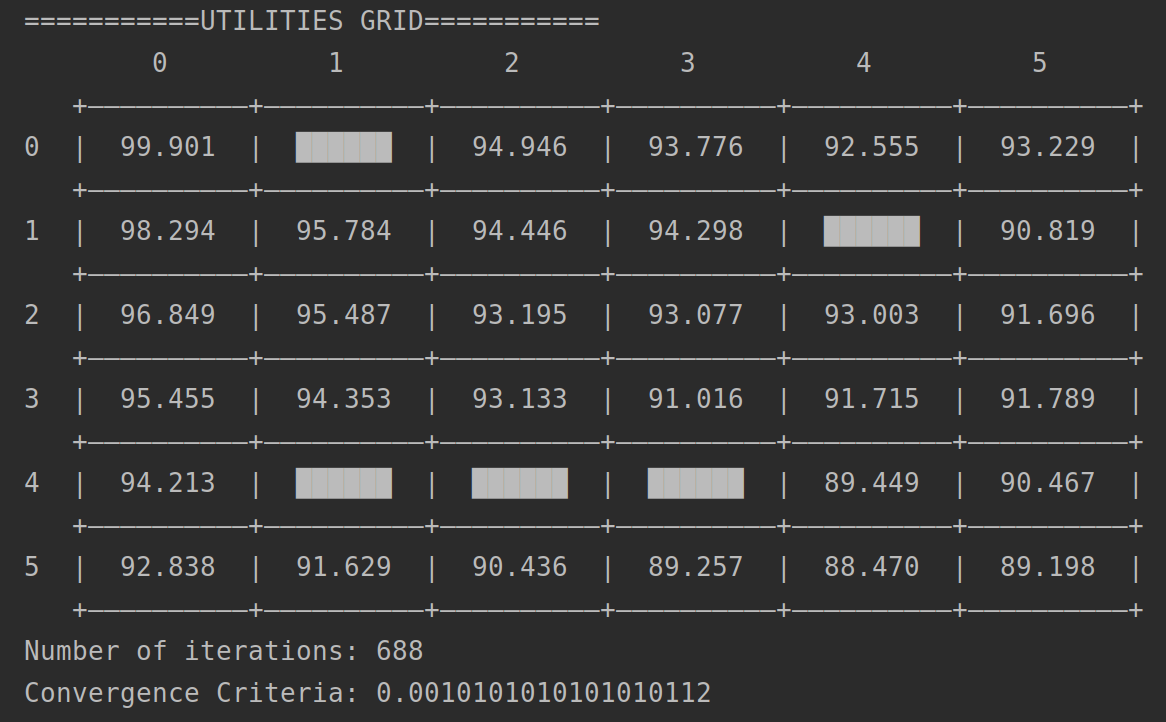
# **Appendix**

## *Main.java* Results (Original GridWorld, scale=1)



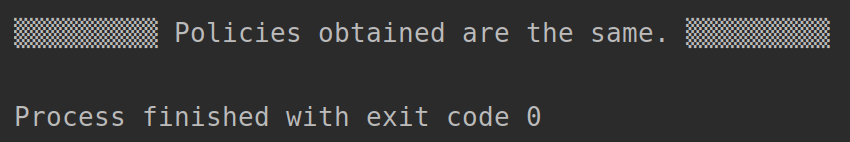
#### Value Iteration (Scale=1):



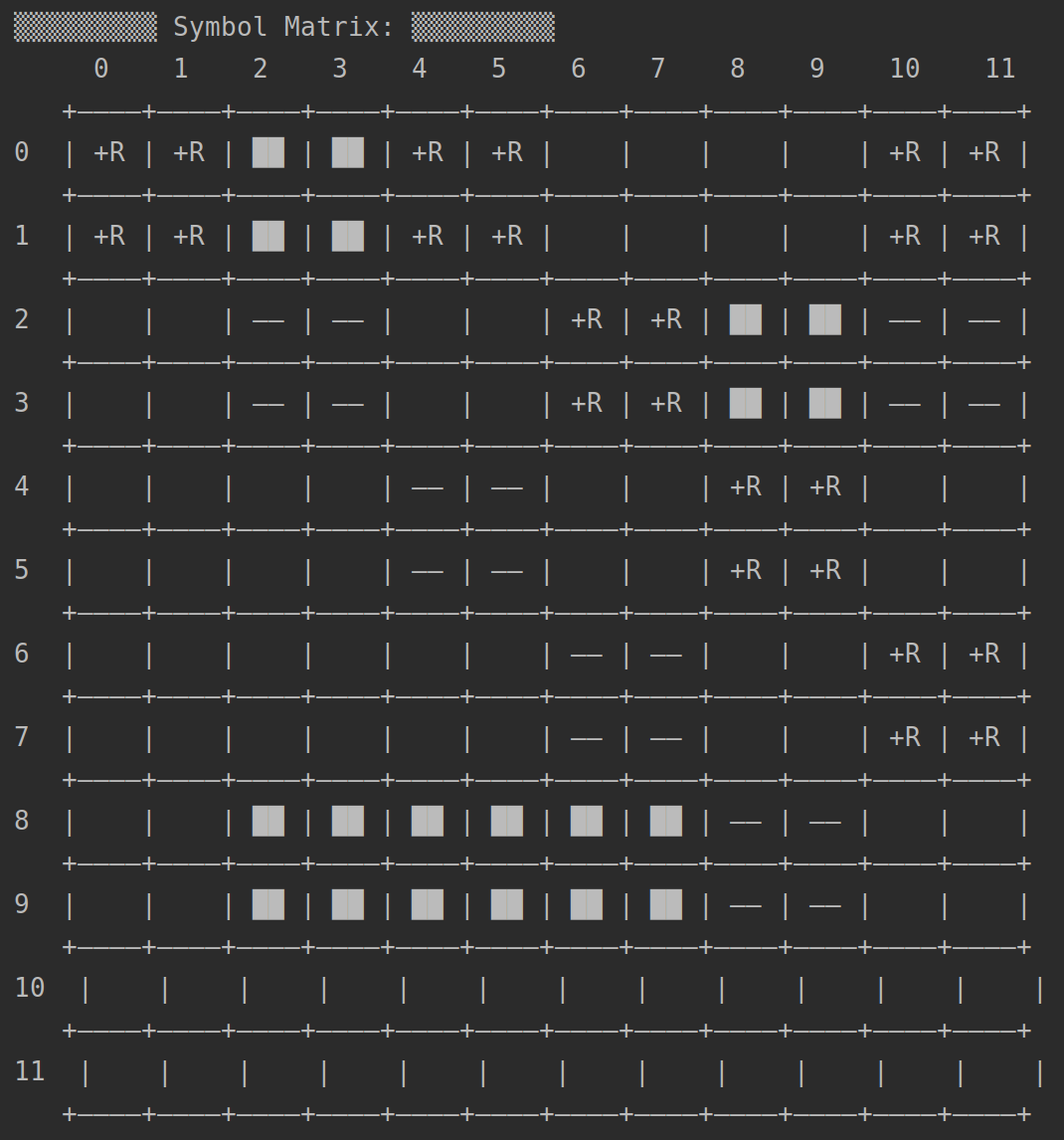


#### **Policy Iteration (Scale**=1**):**

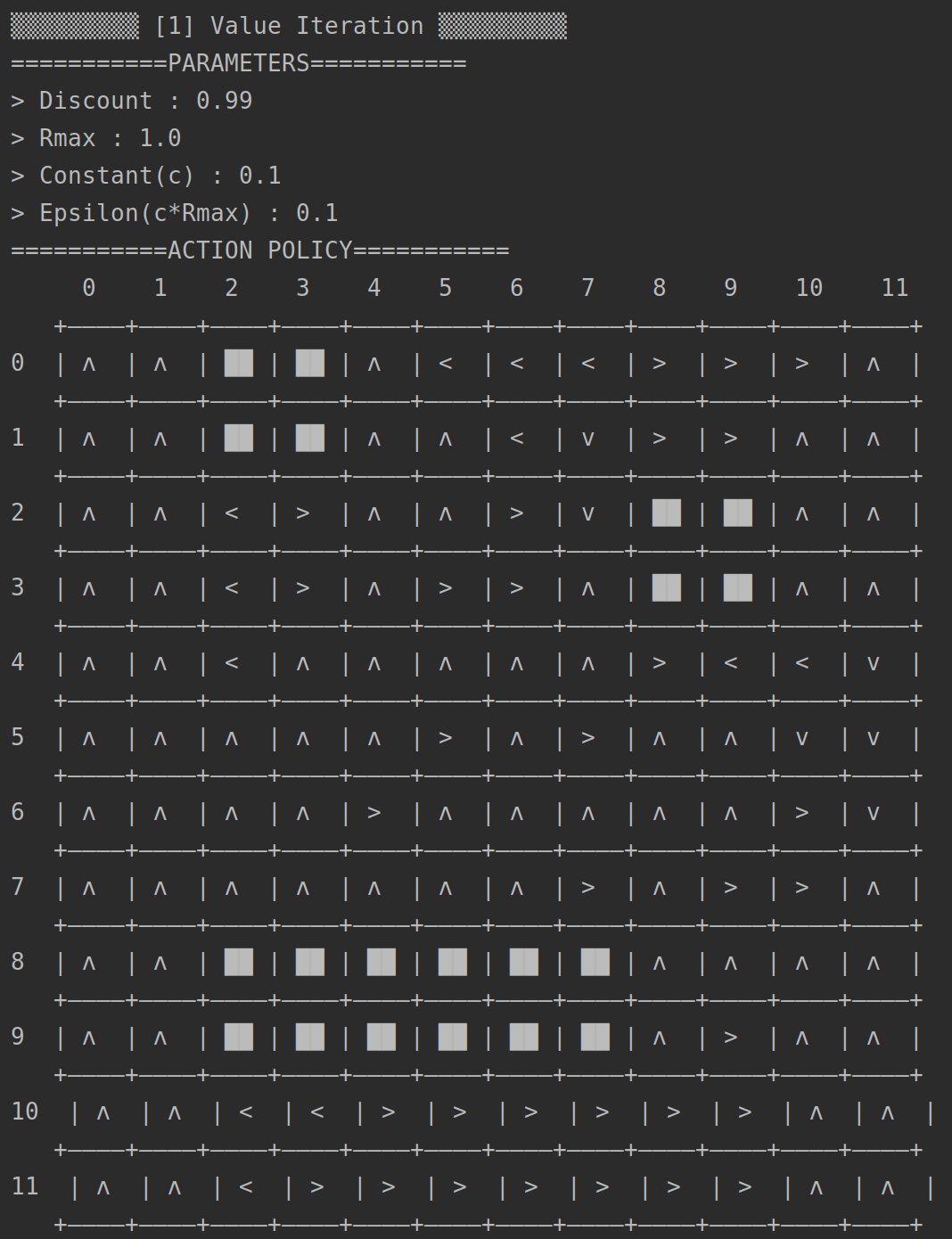
#### Final Results (Scale=1)

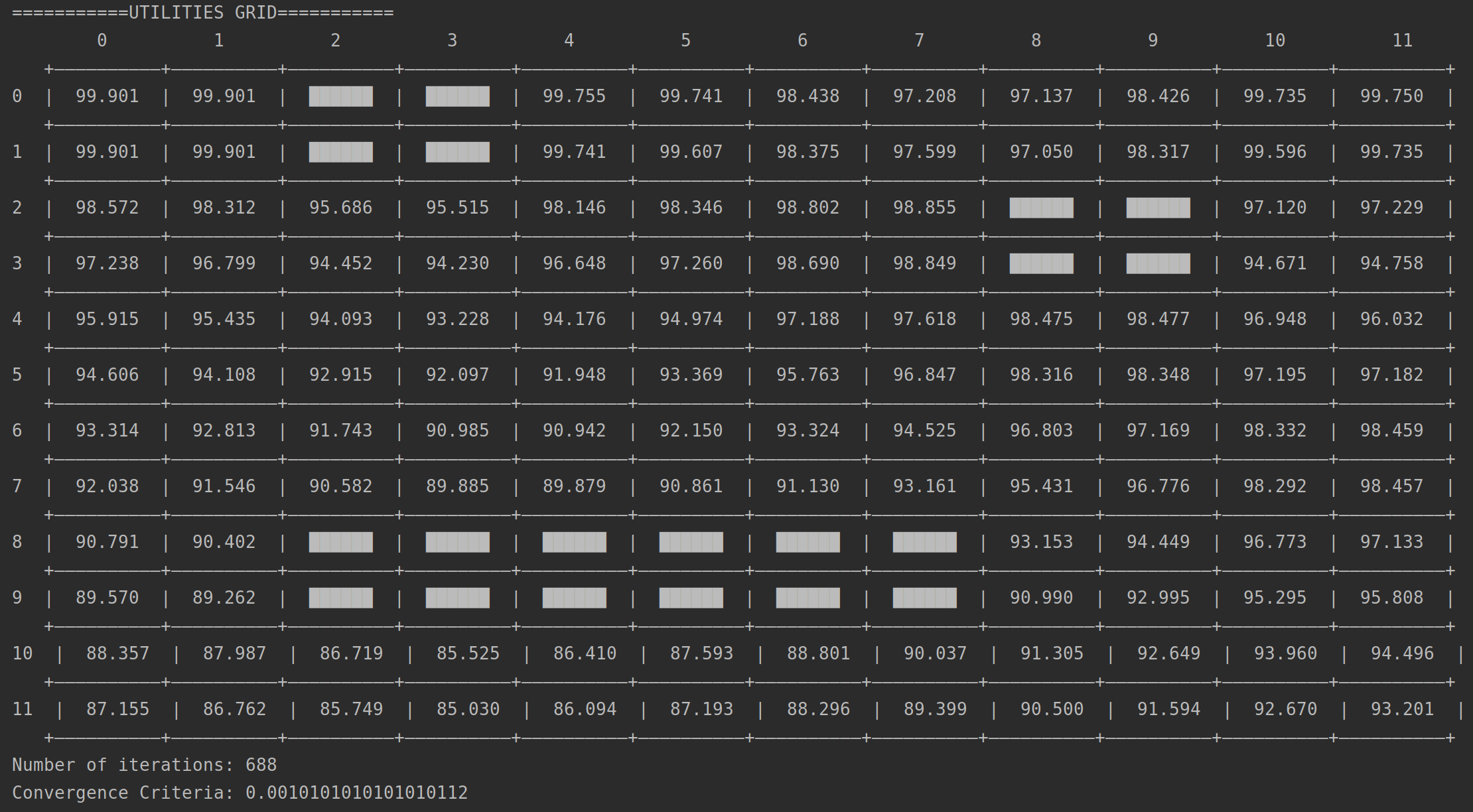


## *ComplexGridWorldExperiment1.java* Results (Scale=2)



#### Value Iteration (Scale=2):





===========REFERENCE UTILITIES OF STATES===========

> Coordinates are in (col,row) format. Top-left corner is (0,0) <

(0,0) : 99.900685

(0,1) : 99.900685

(0,2) : 98.571833

(0,3) : 97.237541

(0,4) : 95.914738

(0,5) : 94.606170

(0,6) : 93.313641

(0,7) : 92.038290

(0,8) : 90.791471

(0,9) : 89.570203

(0,10) : 88.356591

(0,11) : 87.155226

(1,0) : 99.900685

(1,1) : 99.900685

(1,2) : 98.311836

(1,3) : 96.799252

(1,4) : 95.434743

(1,5) : 94.107882

(1,6) : 92.813101

(1,7) : 91.546346

(1,8) : 90.401852

(1,9) : 89.261624

(1,10) : 87.986670

(1,11) : 86.761943

(2,2) : 95.685611

(2,3) : 94.452060

(2,4) : 94.092625

(2,5) : 92.914622

(2,6) : 91.743449

(2,7) : 90.581536

(2,10) : 86.718730

(2,11) : 85.748746

(3,2) : 95.515386

(3,3) : 94.229885

(3,4) : 93.227701

(3,5) : 92.096720

(3,6) : 90.985500

(3,7) : 89.885138

(3,10) : 85.525231

(3,11) : 85.030212

(4,0) : 99.755470

(4,1) : 99.740785

(4,2) : 98.145954

(4,3) : 96.648119

(4,4) : 94.176301

(4,5) : 91.947729

(4,6) : 90.942298

(4,7) : 89.879212

(4,10) : 86.410467

(4,11) : 86.093691

(5,0) : 99.740785

(5,1) : 99.607133

(5,2) : 98.345666

(5,3) : 97.260164

(5,4) : 94.974134

(5,5) : 93.368833

(5,6) : 92.149521

(5,7) : 90.861351

(5,10) : 87.592846

(5,11) : 87.192876

(6,0) : 98.438174

(6,1) : 98.374596

(6,2) : 98.801603

(6,3) : 98.690018

(6,4) : 97.188095

(6,5) : 95.763360

(6,6) : 93.324320

(6,7) : 91.130075

(6,10) : 88.800553

(6,11) : 88.295541

(7,0) : 97.207958

(7,1) : 97.599270

(7,2) : 98.855046

(7,3) : 98.848521

(7,4) : 97.617697

(7,5) : 96.847106

(7,6) : 94.524503

(7,7) : 93.160881

(7,10) : 90.036640

(7,11) : 89.398998

(8,0) : 97.137319

(8,1) : 97.050297

(8,4) : 98.475117

(8,5) : 98.315669

(8,6) : 96.802713

(8,7) : 95.430566

(8,8) : 93.152542

(8,9) : 90.990397

(8,10) : 91.304912

(8,11) : 90.499809

(9,0) : 98.426435

(9,1) : 98.316558

(9,4) : 98.477031

(9,5) : 98.348410

(9,6) : 97.169271

(9,7) : 96.776466

(9,8) : 94.448602

(9,9) : 92.995095

(9,10) : 92.649445

(9,11) : 91.593586

(10,0) : 99.734685

(10,1) : 99.595952

(10,2) : 97.119551

(10,3) : 94.671207

(10,4) : 96.947607

(10,5) : 97.195247

(10,6) : 98.331733

(10,7) : 98.292015

(10,8) : 96.772913

(10,9) : 95.294655

(10,10) : 93.959778

(10,11) : 92.669829

(11,0) : 99.749928

(11,1) : 99.734685

(11,2) : 97.229427

(11,3) : 94.758229

(11,4) : 96.032008

(11,5) : 97.181824

(11,6) : 98.458930

(11,7) : 98.456608

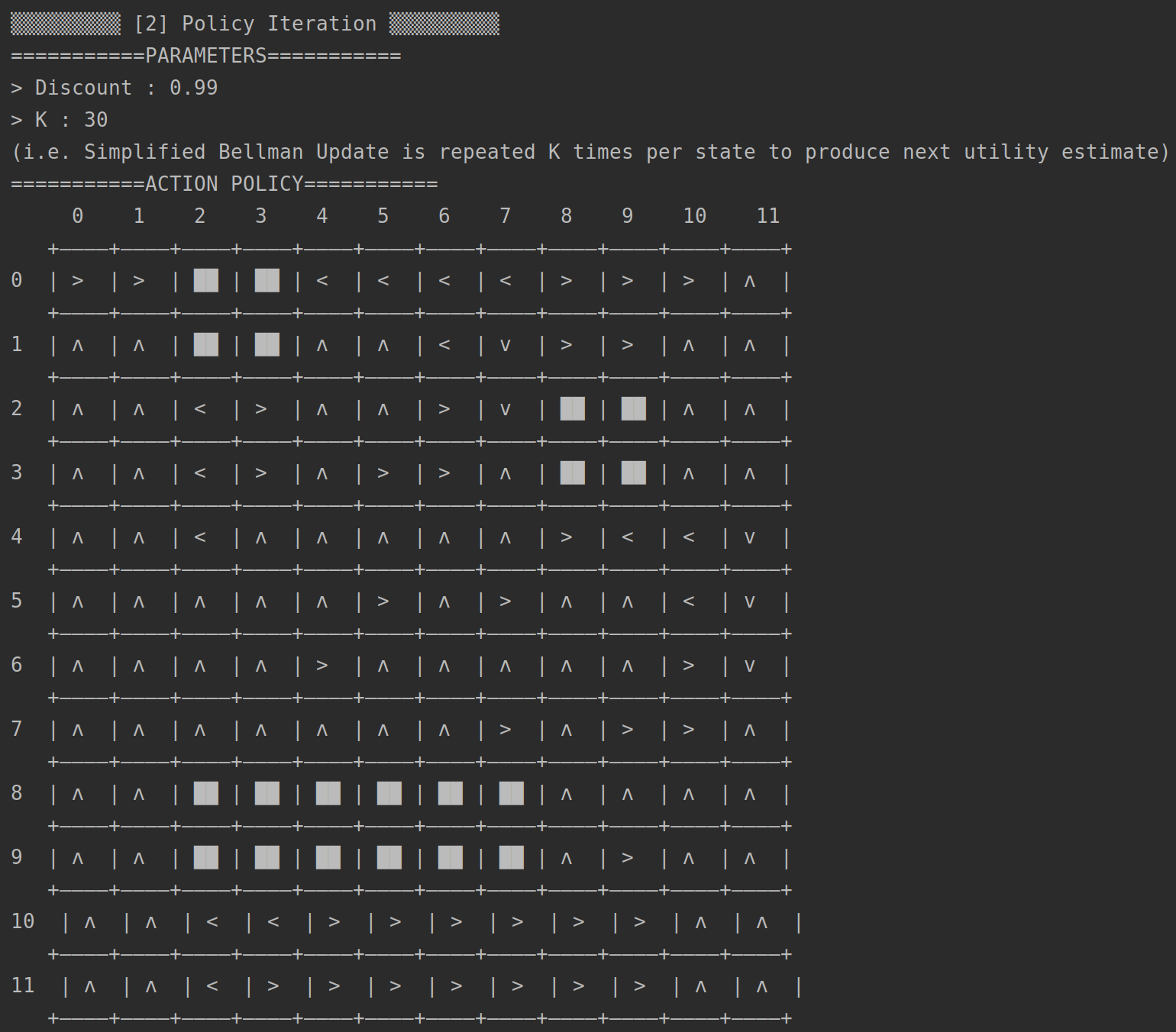
(11,8) : 97.133378

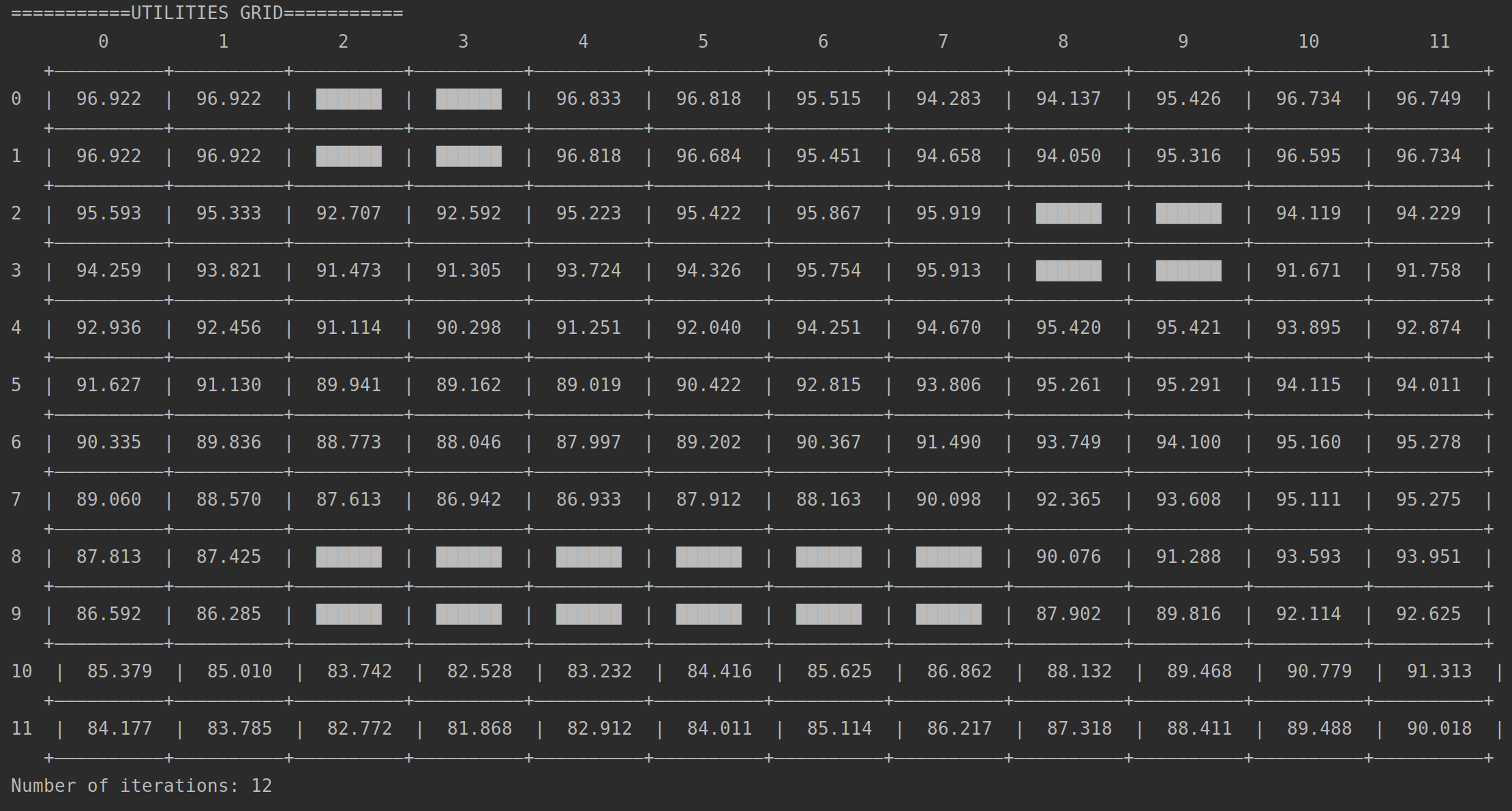
(11,9) : 95.807799

(11,10) : 94.495912

(11,11) : 93.200995

#### Policy Iteration (Scale=2):





===========REFERENCE UTILITIES OF STATES===========

> Coordinates are in (col,row) format. Top-left corner is (0,0) <

(0,0) : 96.921934

(0,1) : 96.921934

(0,2) : 95.593082

(0,3) : 94.258791

(0,4) : 92.935995

(0,5) : 91.627487

(0,6) : 90.335091

(0,7) : 89.059947

(0,8) : 87.813295

(0,9) : 86.592160

(0,10) : 85.378654

(0,11) : 84.177375

(1,0) : 96.921934

(1,1) : 96.921934

(1,2) : 95.333086

(1,3) : 93.820508

(1,4) : 92.456053

(1,5) : 91.129680

(1,6) : 89.835613

(1,7) : 88.569669

(1,8) : 87.425008

(1,9) : 86.284647

(1,10) : 85.009585

(1,11) : 83.784773

(2,2) : 92.706867

(2,3) : 91.473370

(2,4) : 91.114422

(2,5) : 89.940806

(2,6) : 88.772734

(2,7) : 87.613019

(2,10) : 83.741636

(2,11) : 82.771584

(3,2) : 92.592399

(3,3) : 91.305261

(3,4) : 90.297606

(3,5) : 89.162387

(3,6) : 88.046378

(3,7) : 86.942391

(3,10) : 82.527531

(3,11) : 81.868202

(4,0) : 96.832871

(4,1) : 96.818171

(4,2) : 95.223172

(4,3) : 93.723981

(4,4) : 91.250585

(4,5) : 89.019021

(4,6) : 87.996531

(4,7) : 86.933331

(4,10) : 83.232399

(4,11) : 82.911744

(5,0) : 96.818171

(5,1) : 96.684388

(5,2) : 95.421732

(5,3) : 94.325707

(5,4) : 92.040267

(5,5) : 90.421925

(5,6) : 89.201683

(5,7) : 87.911773

(5,10) : 84.415905

(5,11) : 84.011108

(6,0) : 95.515415

(6,1) : 95.450664

(6,2) : 95.867048

(6,3) : 95.754232

(6,4) : 94.251242

(6,5) : 92.814990

(6,6) : 90.367380

(6,7) : 88.163286

(6,10) : 85.624850

(6,11) : 85.113831

(7,0) : 94.283166

(7,1) : 94.658253

(7,2) : 95.919378

(7,3) : 95.912781

(7,4) : 94.669835

(7,5) : 93.805559

(7,6) : 91.490241

(7,7) : 90.098419

(7,10) : 86.862313

(7,11) : 86.217197

(8,0) : 94.136853

(8,1) : 94.049831

(8,4) : 95.419706

(8,5) : 95.261448

(8,6) : 93.749075

(8,7) : 92.364586

(8,8) : 90.075993

(8,9) : 87.902353

(8,10) : 88.132137

(8,11) : 87.317733

(9,0) : 95.425969

(9,1) : 95.316093

(9,4) : 95.421404

(9,5) : 95.290513

(9,6) : 94.100452

(9,7) : 93.608337

(9,8) : 91.288070

(9,9) : 89.816178

(9,10) : 89.467847

(9,11) : 88.411007

(10,0) : 96.734219

(10,1) : 96.595486

(10,2) : 94.119085

(10,3) : 91.670741

(10,4) : 93.895080

(10,5) : 94.114909

(10,6) : 95.160289

(10,7) : 95.111094

(10,8) : 93.593415

(10,9) : 92.114413

(10,10) : 90.778622

(10,11) : 89.487788

(11,0) : 96.749463

(11,1) : 96.734219

(11,2) : 94.228962

(11,3) : 91.757763

(11,4) : 92.874280

(11,5) : 94.011440

(11,6) : 95.277851

(11,7) : 95.274969

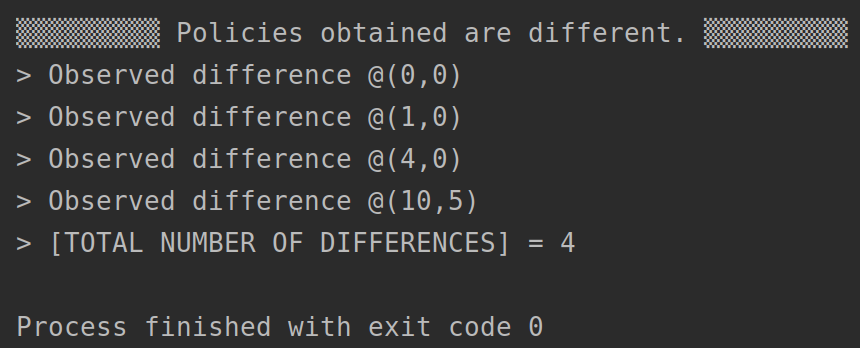
(11,8) : 93.951396

(11,9) : 92.625428

(11,10) : 91.313089

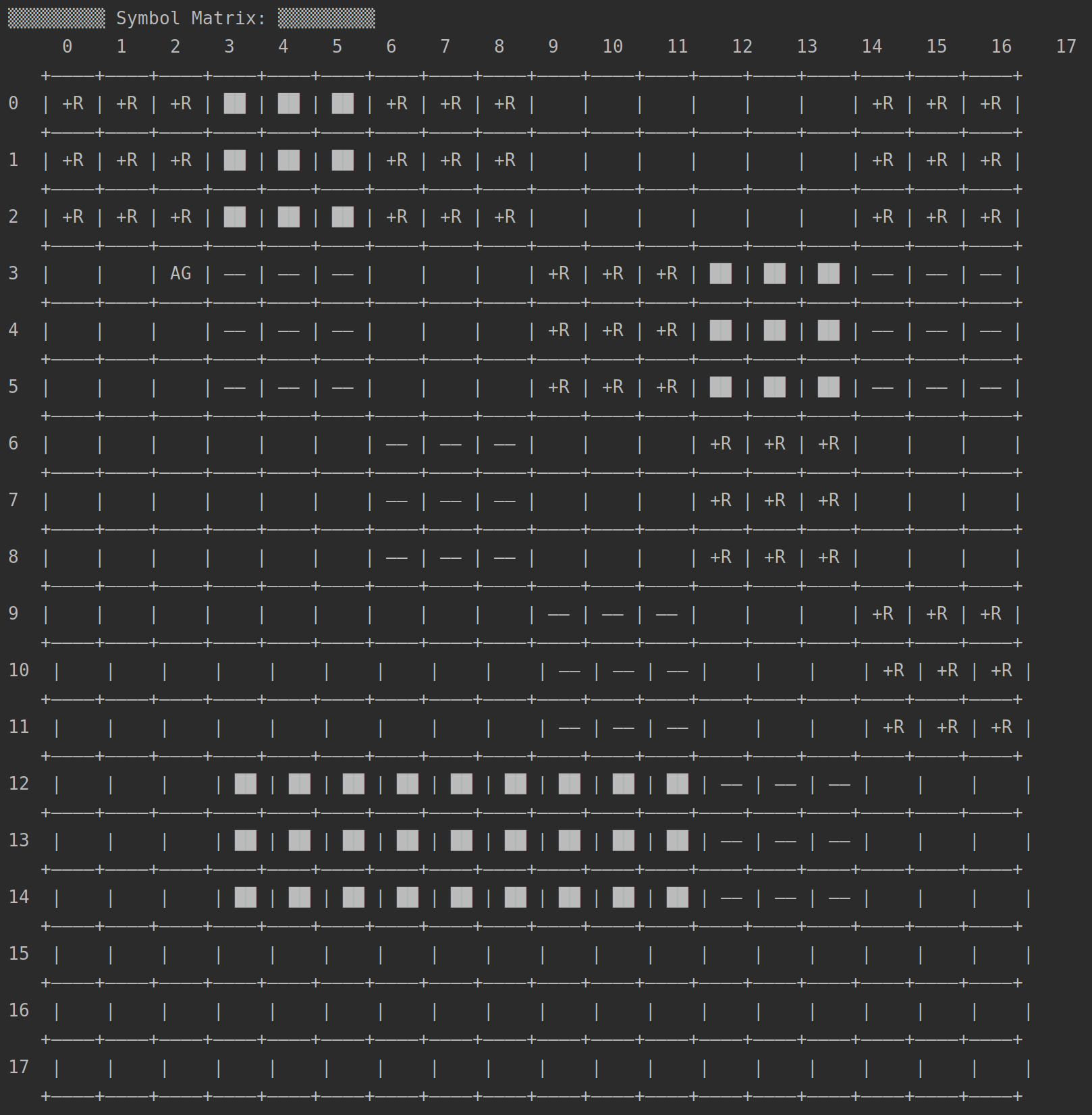
(11,11) : 90.017669

#### Final Results (Scale=2):



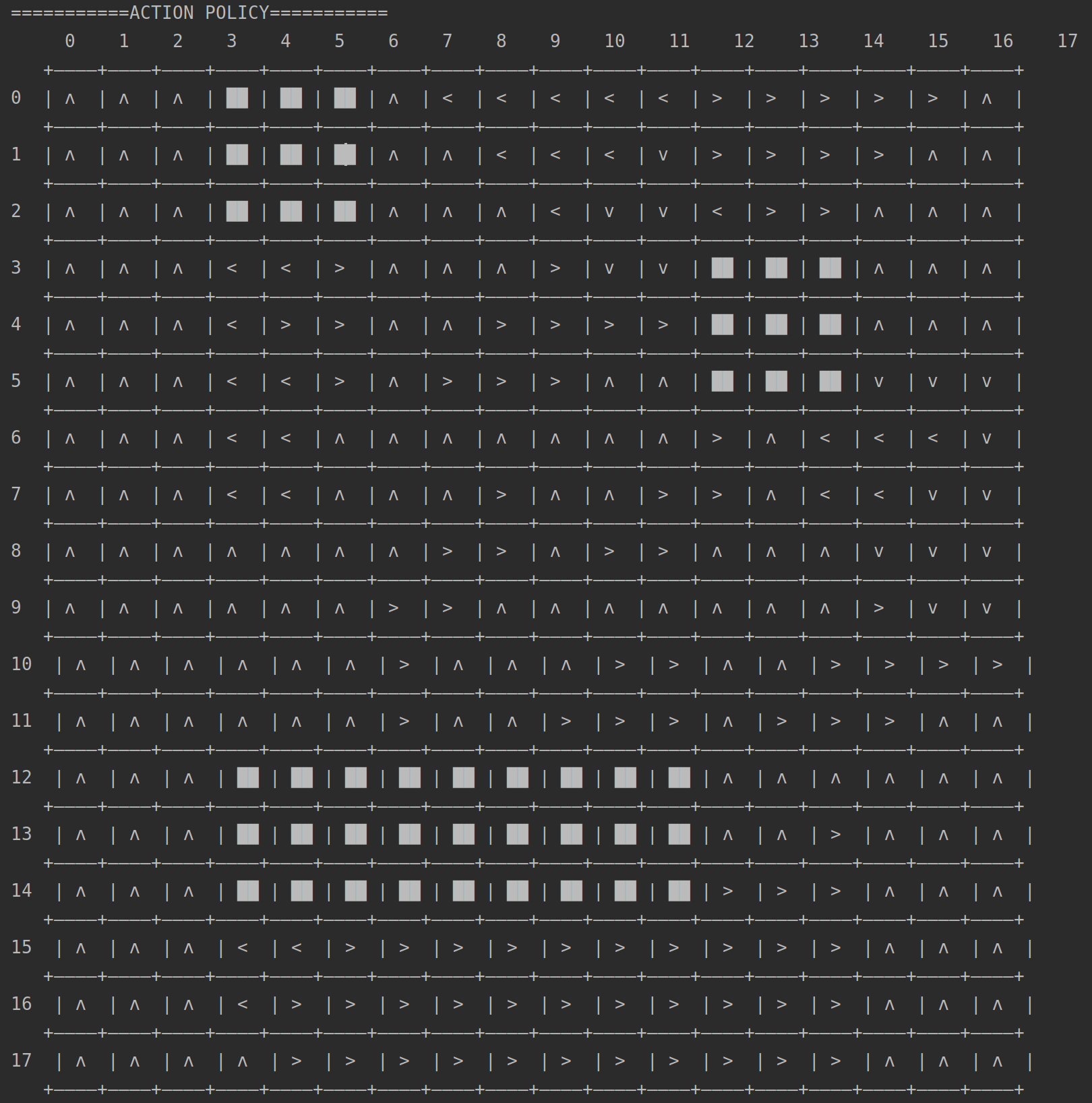
## 

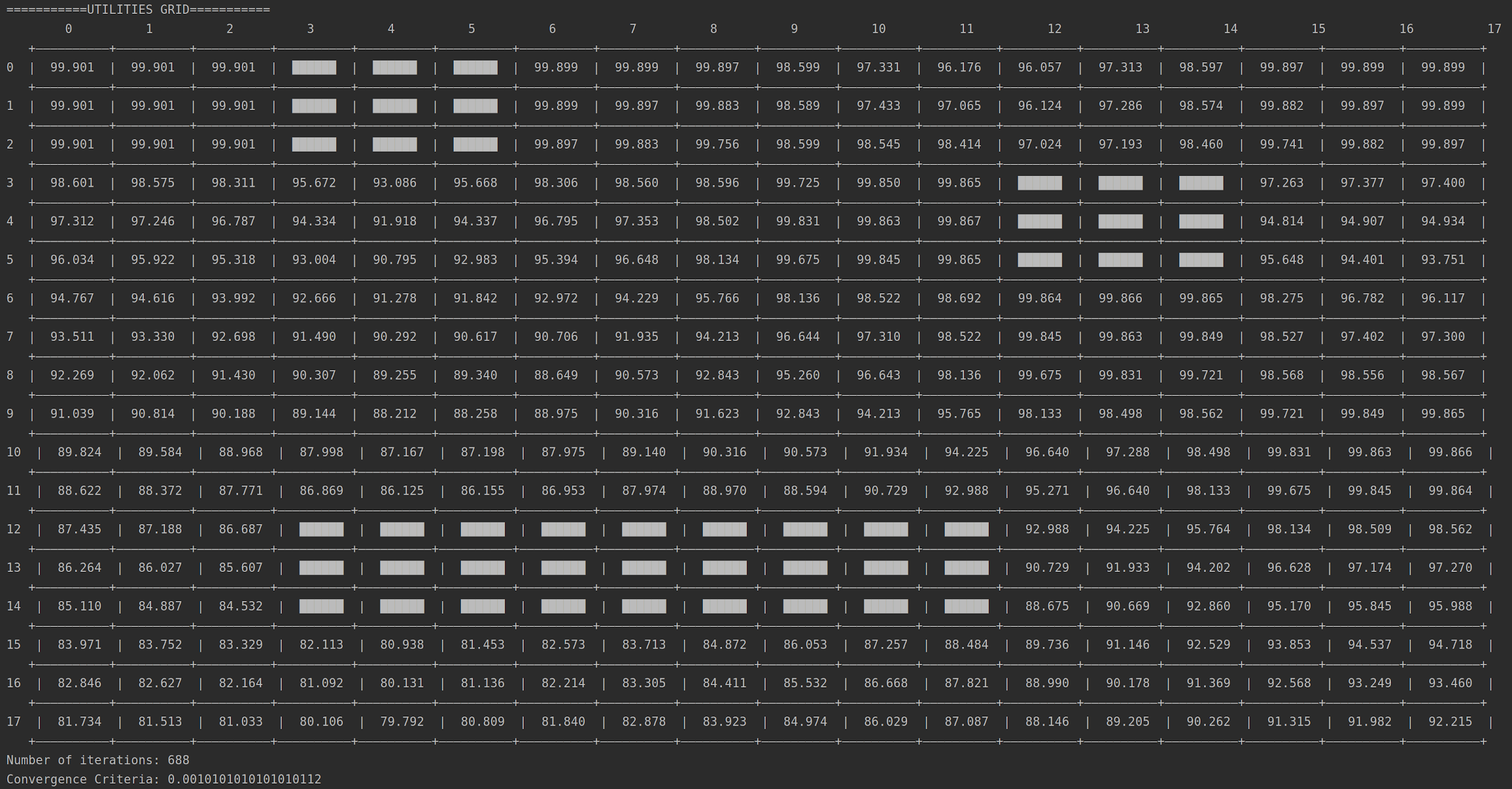
## *ComplexGridWorldExperiment2.java* Results (Scale=3)



#### Value Iteration (Scale=3):

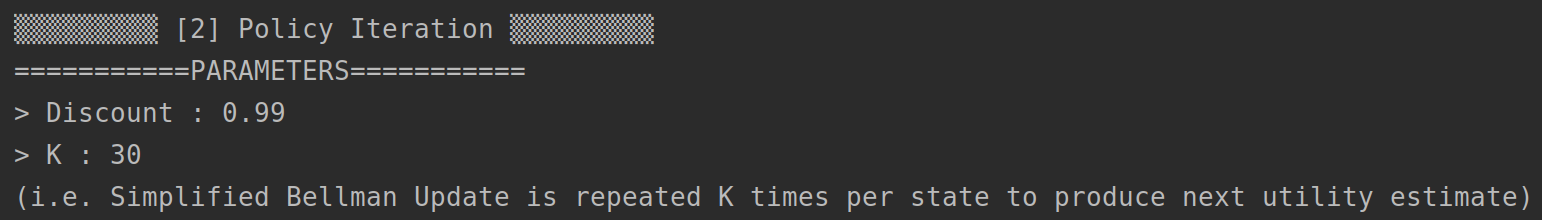


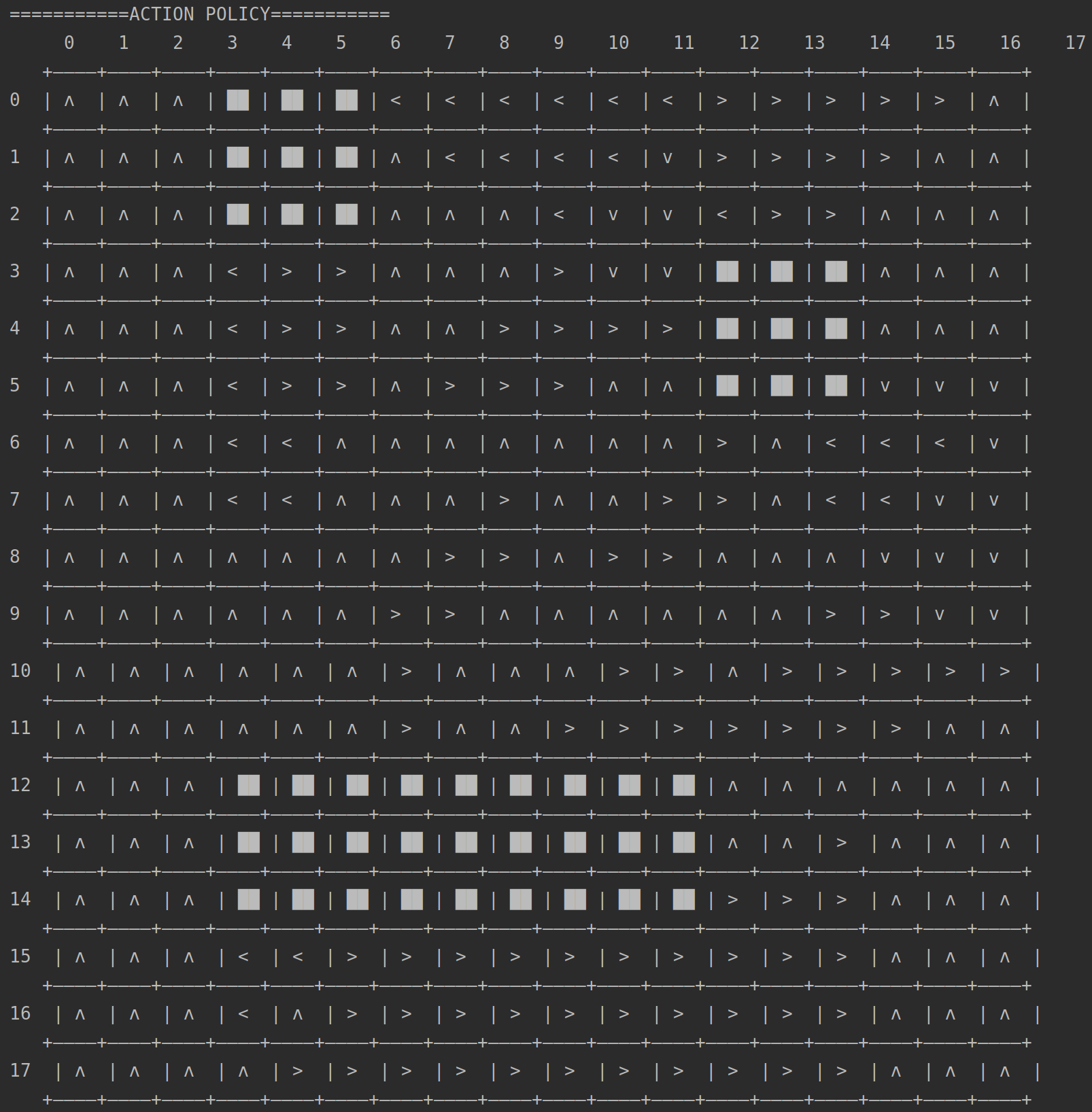




#### 

#### Policy Iteration (Scale=3):





#### Final Results (Scale=3):

