Assignment 3

# Name : Youssef Attia

# ID : 8164

# Name : Ahmed Mahmoud Ali

# ID : 8062

# Name : Abdelrahman amr Salah

# ID : 8016

# Introduction

This Code implements **Markov Decision Processes (MDP)** using two primary algorithms: **Value Iteration** and **Policy Iteration**. These algorithms are commonly used in reinforcement learning to determine optimal policies in grid-based environments. The core concept revolves around determining the best set of actions to take in each state to maximize the total expected reward over time. This grid-based setup includes predefined rewards, actions (up, down, left, right), and state transition probabilities.

The grid is a 3x3 matrix where rewards are assigned, and the algorithms calculate utilities for each cell based on possible actions. The algorithms iterate until a policy (set of actions for each state) is optimized and convergence is achieved.

# Code Explanation:

## Imports and initialization:

import numpy as np  
import random  
  
Rows = 3  
Cols = 3  
Actions\_count = 4  
Actions = [(1, 0), (0, -1), (-1, 0), (0, 1)] # Down, Left, Up, Right  
Gamma = 0.99  
MAX\_ERROR = 1e-4 # Convergence threshold

## PrintGrid

def printGrid(arr, policy=False):  
 res = ""  
 for r in range(Rows):  
 res += "|"  
 for c in range(Cols):  
 if policy:  
 if r == 0 and (c == 0 or c == 2):  
 val = "T.State"  
 else:  
 val = ["Down", "Left", "Up", "Right"][arr[r][c]]  
 else:  
 val = str(arr[r][c])  
 res += " " + val[:7].center(7) + " |"  
 res += "\n"  
 print(res)

## Explanation:

* **Purpose**: This function is used to print the grid in a neatly formatted manner. Depending on the policy flag, it either prints the utilities (values) for each grid cell or the actions from the policy.
* **How it works:**

It loops through each row and column of the grid.

If policy is set to True, it prints the corresponding action (Down, Left, Up, Right) for each cell. Some cells, like the top-left and top-right corners, are special and labeled as "T.State" (Terminal State).

If policy is False, it prints the utility values for each cell.

## getValue:

def getValue(U, r, c, action):  
 rMove, cMove = Actions[action]  
 newR, newC = r + rMove, c + cMove  
 if newR < 0 or newC < 0 or newR >= Rows or newC >= Cols or (newR == newC == 1): # invalid move  
 return U[r][c]  
 else:  
 return U[newR][newC]

### Explaination

* **Purpose**: This function retrieves the utility value of a neighboring cell based on the given action (move direction).
* **How it works**:

It uses the Actions array to calculate the new row (newR) and column (newC) based on the current position (r, c) and the chosen action.

It checks if the move is valid (i.e., within the grid and not hitting an obstacle).

If the move is valid, it returns the utility of the new cell. If it's invalid (out of bounds or hitting an obstacle), it returns the current cell's utility.

## calcValue:

def calcValue(U, r, c, action):  
 v = -1  
 v += 0.1 \* Gamma \* getValue(U, r, c, (action - 1) % 4)  
 v += 0.8 \* Gamma \* getValue(U, r, c, action)  
 v += 0.1 \* Gamma \* getValue(U, r, c, (action + 1) % 4)  
 return v

### Explanation:

**Purpose**: This function calculates the expected value (utility) of taking a certain action from a given state. It considers the uncertainty in actions, i.e., there’s a chance the agent might move left or right instead of the intended direction.

**How it works**:

It calculates the utility of three possible moves: left of the intended direction (10% chance), the intended move (80% chance), and right of the intended direction (10% chance).

It sums these weighted utilities and discounts them by Gamma (which reduces future rewards' impact).

## valueIteration:

def valueIteration(arr, reward):  
 print("During the value iteration:\n")  
 while True:  
 temp = [[reward, 0, 10], [0, 0, 0], [0, 0, 0]]  
 error = 0  
 for r in range(Rows):  
 for c in range(Cols):  
 if r == 0 and (c == 2 or c == 0):  
 continue  
 temp[r][c] = max([calcValue(arr, r, c, action) for action in range(Actions\_count)])  
 error = max(error, abs(temp[r][c] - arr[r][c]))  
 arr = temp  
 printGrid(arr)  
 if error < MAX\_ERROR:  
 break  
 return arr

### Explanation:

**Purpose**: This function performs the **Value Iteration** algorithm to find the optimal utility values for each state in the grid.

**How it works**:

It starts by initializing a grid with rewards for each state.

It then loops over each state in the grid and computes the maximum utility for that state by considering all possible actions.

This process repeats until the difference between the old and new utility values is less than MAX\_ERROR, meaning the utilities have converged.

It prints the grid at each iteration to show the progress of the utility updates.

## getOptimalPolicy:

def getOptimalPolicy(arr):  
 policy = np.zeros((Rows, Cols), dtype=int)  
 for r in range(Rows):  
 for c in range(Cols):  
 if r == 0 and (c == 2 or c == 0):  
 continue  
 # Choose the action that maximizes the utility  
 maxAction, maxv = None, -float("inf")  
 for action in range(Actions\_count):  
 v = calcValue(arr, r, c, action)  
 if v > maxv:  
 maxAction, maxv = action, v  
 policy[r][c] = maxAction  
 return policy

### Explanation:

**Purpose**: This function determines the optimal policy by choosing the action that maximizes the utility for each state.

**How it works:**

For each cell in the grid, it calculates the utility for each action (up, down, left, right) using the calcValue function.

It picks the action with the highest utility and stores it in the policy grid.

Returns the optimal policy, which is a 2D array showing the best action to take from each state.

## policyEvaluation:

def policyEvaluation(policy, arr, reward):  
 while True:  
 temp = [[reward, 0, 10], [0, 0, 0], [0, 0, 0]]  
 error = 0  
 for r in range(Rows):  
 for c in range(Cols):  
 if r == 0 and (c == 2 or c == 0):  
 continue  
 temp[r][c] = calcValue(arr, r, c, policy[r][c])  
 error = max(error, abs(temp[r][c] - arr[r][c]))  
 arr = temp  
 if error < MAX\_ERROR:  
 break  
 return arr

Explanation:

**Purpose**: This function evaluates the utility of a given policy. It computes the utility of each state based on the actions dictated by the current policy.

**How it works:**

It loops through each cell in the grid and uses the action prescribed by the policy to calculate the utility.

This process continues until the utilities converge (i.e., the difference between old and new utilities is smaller than MAX\_ERROR).

Returns the utility grid that corresponds to the current policy.

## policyIteration:

def policyIteration(policy, arr, reward):  
 print("During the policy iteration:\n")  
 while True:  
 arr = policyEvaluation(policy, arr, reward)  
 unchanged = True  
 for r in range(Rows):  
 for c in range(Cols):  
 if r == 0 and (c == 2 or c == 0):  
 continue  
 maxAction, maxv = None, -float("inf")  
 for action in range(Actions\_count):  
 value = calcValue(arr, r, c, action)  
 if value > maxv:  
 maxAction, maxv = action, value  
 if maxv > calcValue(arr, r, c, policy[r][c]):  
 policy[r][c] = maxAction  
 unchanged = False  
 if unchanged:  
 break  
 printGrid(policy, True)  
 return policy

### Explanation:

* **Purpose**: This function performs the **Policy Iteration** algorithm. It improves a given policy by alternating between policy evaluation and policy improvement.
* **How it works**:

First, it evaluates the current policy using policyEvaluation.

Then, it improves the policy by checking if a better action (with a higher utility) exists for each state. If a better action is found, it updates the policy.

This process repeats until no changes are made to the policy, indicating convergence.

The function prints the grid showing the policy at each iteration.

## Main Loop - Value Iteration:

r\_values = [100, 3, 0, -3]  
for r in r\_values:  
 Grid = [  
 [r, -1, 10],  
 [-1, -1, -1],  
 [-1, -1, -1]  
 ]  
 print("The initial U for ", r, " is:\n")  
 printGrid(Grid)  
 final = valueIteration(Grid, r)  
 policy = getOptimalPolicy(final)  
 print("The optimal policy is:\n")  
 printGrid(policy, True)

## **Main Loop - Policy Iteration:**

r\_values = [100, 3, 0, -3]  
for r in r\_values:  
 Grid = [  
 [r, -1, 10],  
 [-1, -1, -1],  
 [-1, -1, -1]  
 ]  
 policy = [[random.randint(0, 3) for j in range(Cols)] for i in range(Rows)]  
 print("The initial random policy for ", r, " is:\n")  
 printGrid(policy, True)  
 finall = policyIteration(policy, Grid, r)  
 print("The optimal policy is:\n")  
 printGrid(finall, True)

# Main Loops Explanation:

**Value Iteration Loop**:

* For each reward value in r\_values, the program initializes a grid and performs the value iteration algorithm.
* After convergence, it extracts and prints the optimal policy for each reward setting.

**Policy Iteration Loop**:

* For each reward value in r\_values, a random policy is generated, and policy iteration is used to improve it.
* After convergence, the program prints the optimal policy.

# Results:

The initial Values for 100 is:

| 100 | -1 | 10 |

| -1 | -1 | -1 |

| -1 | -1 | -1 |

During the value iteration:

| 100 | 78.002 | 10 |

| 78.002 | -1.99 | 6.72199 |

| -1.99 | -1.99 | -1.99 |

| 100 | 93.6443 | 10 |

| 93.6443 | 69.1652 | 8.25095 |

| 60.3835 | -2.9701 | 3.92980 |

| 100 | 96.7415 | 10 |

| 96.7415 | 83.2540 | 8.55368 |

| 78.8502 | 46.2357 | 5.62976 |

| 100 | 97.3548 | 10 |

| 97.3548 | 89.7740 | 8.61363 |

| 88.0028 | 70.6041 | 37.0228 |

| 100 | 97.4762 | 10 |

| 97.4762 | 92.7329 | 30.0275 |

| 91.8071 | 82.6778 | 59.4364 |

| 100 | 97.5002 | 10 |

| 97.5002 | 94.0364 | 52.0191 |

| 93.4752 | 88.0814 | 73.3378 |

| 100 | 97.5050 | 10 |

| 97.5050 | 94.5928 | 67.3833 |

| 94.1943 | 90.4724 | 81.1708 |

| 100 | 97.5060 | 10 |

| 97.5060 | 94.8337 | 76.6292 |

| 94.5060 | 91.5154 | 85.3610 |

| 100 | 97.5061 | 10 |

| 97.5061 | 94.9378 | 81.7785 |

| 94.6408 | 91.9688 | 87.5172 |

| 100 | 97.5062 | 10 |

| 97.5062 | 94.9829 | 84.5058 |

| 94.6992 | 92.1654 | 88.5996 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0024 | 85.9030 |

| 94.7245 | 92.2505 | 89.1324 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0108 | 86.6016 |

| 94.7354 | 92.2874 | 89.3909 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0145 | 86.9447 |

| 94.7402 | 92.3034 | 89.5149 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0160 | 87.1108 |

| 94.7422 | 92.3103 | 89.5738 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0167 | 87.1904 |

| 94.7431 | 92.3133 | 89.6015 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0170 | 87.2281 |

| 94.7435 | 92.3146 | 89.6145 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2458 |

| 94.7436 | 92.3151 | 89.6205 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2541 |

| 94.7437 | 92.3154 | 89.6233 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2580 |

| 94.7437 | 92.3155 | 89.6246 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2598 |

| 94.7438 | 92.3155 | 89.6252 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2606 |

| 94.7438 | 92.3155 | 89.6255 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2610 |

| 94.7438 | 92.3155 | 89.6256 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2612 |

| 94.7438 | 92.3155 | 89.6257 |

| 100 | 97.5062 | 10 |

| 97.5062 | 95.0172 | 87.2612 |

| 94.7438 | 92.3155 | 89.6257 |

The time taken: 0.0019519329071044922 Number of iterations: 24 and optimal policy is:

| T.State | Left | T.State |

| Up | Left | Down |

| Up | Left | Left |

The initial Values for 3 is:

| 3 | -1 | 10 |

| -1 | -1 | -1 |

| -1 | -1 | -1 |

During the value iteration:

| 3 | 6.72199 | 10 |

| 1.17800 | -1.99 | 6.72199 |

| -1.99 | -1.99 | -1.99 |

| 3 | 8.25095 | 10 |

| 1.60924 | 5.10592 | 8.25095 |

| -0.4610 | -2.9701 | 3.92980 |

| 3 | 8.55368 | 10 |

| 1.69463 | 6.51091 | 8.55368 |

| -0.0651 | 1.52432 | 5.62976 |

| 3 | 8.61363 | 10 |

| 1.71153 | 6.78910 | 8.61363 |

| 0.48660 | 3.76059 | 6.48277 |

| 3 | 8.62549 | 10 |

| 1.71488 | 7.04704 | 8.62549 |

| 2.19600 | 4.87895 | 6.83608 |

| 3 | 8.62784 | 10 |

| 1.71554 | 7.16833 | 8.62784 |

| 3.25131 | 5.38021 | 6.99118 |

| 3 | 8.62831 | 10 |

| 1.91471 | 7.22005 | 8.62831 |

| 3.75285 | 5.60230 | 7.05802 |

| 3 | 8.62840 | 10 |

| 2.35137 | 7.24245 | 8.62840 |

| 3.99811 | 5.69921 | 7.08699 |

| 3 | 8.62842 | 10 |

| 2.63207 | 7.25213 | 8.62842 |

| 4.14237 | 5.74134 | 7.09953 |

| 3 | 8.62842 | 10 |

| 2.80191 | 7.25631 | 8.62842 |

| 4.21781 | 5.75961 | 7.10495 |

| 3 | 8.62842 | 10 |

| 2.89528 | 7.25813 | 8.62842 |

| 4.25656 | 5.76753 | 7.10730 |

| 3 | 8.62842 | 10 |

| 2.94446 | 7.25891 | 8.62842 |

| 4.27591 | 5.77095 | 7.10832 |

| 3 | 8.62842 | 10 |

| 2.96953 | 7.25925 | 8.62842 |

| 4.28541 | 5.77244 | 7.10876 |

| 3 | 8.62842 | 10 |

| 2.98201 | 7.25940 | 8.62842 |

| 4.29001 | 5.77308 | 7.10895 |

| 3 | 8.62842 | 10 |

| 2.98813 | 7.25946 | 8.62842 |

| 4.29221 | 5.77336 | 7.10903 |

| 3 | 8.62842 | 10 |

| 2.99108 | 7.25949 | 8.62842 |

| 4.29325 | 5.77348 | 7.10907 |

| 3 | 8.62842 | 10 |

| 2.99249 | 7.25950 | 8.62842 |

| 4.29374 | 5.77353 | 7.10908 |

| 3 | 8.62842 | 10 |

| 2.99316 | 7.25951 | 8.62842 |

| 4.29397 | 5.77355 | 7.10909 |

| 3 | 8.62842 | 10 |

| 2.99347 | 7.25951 | 8.62842 |

| 4.29408 | 5.77356 | 7.10909 |

| 3 | 8.62842 | 10 |

| 2.99362 | 7.25951 | 8.62842 |

| 4.29413 | 5.77357 | 7.10909 |

| 3 | 8.62842 | 10 |

| 2.99369 | 7.25951 | 8.62842 |

| 4.29415 | 5.77357 | 7.10910 |

The time taken: 0.002955913543701172 Number of iterations: 21 and optimal policy is:

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The initial Values for 0 is:

| 0 | -1 | 10 |

| -1 | -1 | -1 |

| -1 | -1 | -1 |

During the value iteration:

| 0 | 6.72199 | 10 |

| -1.198 | -1.99 | 6.72199 |

| -1.99 | -1.99 | -1.99 |

| 0 | 8.25095 | 10 |

| -1.2372 | 4.8707 | 8.25095 |

| -2.3428 | -2.9701 | 3.92980 |

| 0 | 8.55368 | 10 |

| -1.2449 | 6.22911 | 8.55368 |

| -2.5058 | 1.52432 | 5.62976 |

| 0 | 8.61363 | 10 |

| -1.2465 | 6.77224 | 8.61363 |

| -0.1640 | 3.76059 | 6.48277 |

| 0 | 8.62549 | 10 |

| -1.2468 | 7.04704 | 8.62549 |

| 1.83874 | 4.87895 | 6.83608 |

| 0 | 8.62784 | 10 |

| 0.20941 | 7.16833 | 8.62784 |

| 2.92273 | 5.38021 | 6.99118 |

| 0 | 8.62831 | 10 |

| 1.35627 | 7.22005 | 8.62831 |

| 3.57121 | 5.60230 | 7.05802 |

| 0 | 8.62840 | 10 |

| 2.09694 | 7.24245 | 8.62840 |

| 3.92484 | 5.69921 | 7.08699 |

| 0 | 8.62842 | 10 |

| 2.52367 | 7.25213 | 8.62842 |

| 4.10993 | 5.74134 | 7.09953 |

| 0 | 8.62842 | 10 |

| 2.75475 | 7.25631 | 8.62842 |

| 4.20387 | 5.75961 | 7.10495 |

| 0 | 8.62842 | 10 |

| 2.87490 | 7.25813 | 8.62842 |

| 4.25051 | 5.76753 | 7.10730 |

| 0 | 8.62842 | 10 |

| 2.93564 | 7.25891 | 8.62842 |

| 4.27330 | 5.77095 | 7.10832 |

| 0 | 8.62842 | 10 |

| 2.96571 | 7.25925 | 8.62842 |

| 4.28428 | 5.77244 | 7.10876 |

| 0 | 8.62842 | 10 |

| 2.98036 | 7.25940 | 8.62842 |

| 4.28952 | 5.77308 | 7.10895 |

| 0 | 8.62842 | 10 |

| 2.98741 | 7.25946 | 8.62842 |

| 4.29200 | 5.77336 | 7.10903 |

| 0 | 8.62842 | 10 |

| 2.99077 | 7.25949 | 8.62842 |

| 4.29316 | 5.77348 | 7.10907 |

| 0 | 8.62842 | 10 |

| 2.99236 | 7.25950 | 8.62842 |

| 4.29370 | 5.77353 | 7.10908 |

| 0 | 8.62842 | 10 |

| 2.99310 | 7.25951 | 8.62842 |

| 4.29396 | 5.77355 | 7.10909 |

| 0 | 8.62842 | 10 |

| 2.99345 | 7.25951 | 8.62842 |

| 4.29407 | 5.77356 | 7.10909 |

| 0 | 8.62842 | 10 |

| 2.99361 | 7.25951 | 8.62842 |

| 4.29413 | 5.77357 | 7.10909 |

| 0 | 8.62842 | 10 |

| 2.99368 | 7.25951 | 8.62842 |

| 4.29415 | 5.77357 | 7.10910 |

The time taken: 0.0019524097442626953 Number of iterations: 21 and optimal policy is:

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The initial Values for -3 is:

| -3 | -1 | 10 |

| -1 | -1 | -1 |

| -1 | -1 | -1 |

During the value iteration:

| -3 | 6.72199 | 10 |

| -1.99 | -1.99 | 6.72199 |

| -1.99 | -1.99 | -1.99 |

| -3 | 8.25095 | 10 |

| -2.9701 | 4.79229 | 8.25095 |

| -2.9701 | -2.9701 | 3.92980 |

| -3 | 8.55368 | 10 |

| -3.9403 | 6.05756 | 8.55368 |

| -3.9403 | 1.52432 | 5.62976 |

| -3 | 8.61363 | 10 |

| -4.1561 | 6.77224 | 8.61363 |

| -0.5729 | 3.76059 | 6.48277 |

| -3 | 8.62549 | 10 |

| -2.2766 | 7.04704 | 8.62549 |

| 1.51020 | 4.87895 | 6.83608 |

| -3 | 8.62784 | 10 |

| -0.2547 | 7.16833 | 8.62784 |

| 2.78825 | 5.38021 | 6.99118 |

| -3 | 8.62831 | 10 |

| 1.15786 | 7.22005 | 8.62831 |

| 3.51195 | 5.60230 | 7.05802 |

| -3 | 8.62840 | 10 |

| 2.01072 | 7.24245 | 8.62840 |

| 3.89933 | 5.69921 | 7.08699 |

| -3 | 8.62842 | 10 |

| 2.48639 | 7.25213 | 8.62842 |

| 4.09887 | 5.74134 | 7.09953 |

| -3 | 8.62842 | 10 |

| 2.73861 | 7.25631 | 8.62842 |

| 4.19908 | 5.75961 | 7.10495 |

| -3 | 8.62842 | 10 |

| 2.86792 | 7.25813 | 8.62842 |

| 4.24844 | 5.76753 | 7.10730 |

| -3 | 8.62842 | 10 |

| 2.93261 | 7.25891 | 8.62842 |

| 4.27240 | 5.77095 | 7.10832 |

| -3 | 8.62842 | 10 |

| 2.96440 | 7.25925 | 8.62842 |

| 4.28389 | 5.77244 | 7.10876 |

| -3 | 8.62842 | 10 |

| 2.97979 | 7.25940 | 8.62842 |

| 4.28935 | 5.77308 | 7.10895 |

| -3 | 8.62842 | 10 |

| 2.98717 | 7.25946 | 8.62842 |

| 4.29193 | 5.77336 | 7.10903 |

| -3 | 8.62842 | 10 |

| 2.99066 | 7.25949 | 8.62842 |

| 4.29313 | 5.77348 | 7.10907 |

| -3 | 8.62842 | 10 |

| 2.99231 | 7.25950 | 8.62842 |

| 4.29369 | 5.77353 | 7.10908 |

| -3 | 8.62842 | 10 |

| 2.99308 | 7.25951 | 8.62842 |

| 4.29395 | 5.77355 | 7.10909 |

| -3 | 8.62842 | 10 |

| 2.99344 | 7.25951 | 8.62842 |

| 4.29407 | 5.77356 | 7.10909 |

| -3 | 8.62842 | 10 |

| 2.99360 | 7.25951 | 8.62842 |

| 4.29413 | 5.77357 | 7.10909 |

| -3 | 8.62842 | 10 |

| 2.99368 | 7.25951 | 8.62842 |

| 4.29415 | 5.77357 | 7.10910 |

The time taken: 0.0009796619415283203 Number of iterations: 21 and optimal policy is:

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

//////////////////////////////////////////////////////////////////////////////

The initial random policy for 100 is:

| T.State | Left | T.State |

| Left | Left | Left |

| Up | Up | Right |

During the policy iteration:

| T.State | Left | T.State |

| Up | Up | Up |

| Up | Left | Left |

| T.State | Left | T.State |

| Up | Left | Down |

| Up | Left | Left |

The time taken: 0.003874063491821289 Number of iterations: 3 and optimal policy is:

| T.State | Left | T.State |

| Up | Left | Down |

| Up | Left | Left |

The initial random policy for 3 is:

| T.State | Right | T.State |

| Left | Up | Down |

| Up | Left | Up |

During the policy iteration:

| T.State | Right | T.State |

| Up | Up | Up |

| Up | Left | Left |

| T.State | Right | T.State |

| Up | Up | Up |

| Up | Left | Up |

| T.State | Right | T.State |

| Up | Up | Up |

| Up | Right | Up |

| T.State | Right | T.State |

| Up | Right | Up |

| Right | Right | Up |

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The time taken: 0.00393366813659668 Number of iterations: 6 and optimal policy is:

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The initial random policy for 0 is:

| T.State | Down | T.State |

| Up | Right | Down |

| Down | Right | Right |

During the policy iteration:

| T.State | Right | T.State |

| Up | Up | Up |

| Up | Right | Right |

| T.State | Right | T.State |

| Up | Up | Up |

| Up | Right | Up |

| T.State | Right | T.State |

| Up | Right | Up |

| Right | Right | Up |

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The time taken: 0.011714935302734375 Number of iterations: 5 and optimal policy is:

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The initial random policy for -3 is:

| T.State | Down | T.State |

| Left | Right | Right |

| Right | Up | Left |

During the policy iteration:

| T.State | Right | T.State |

| Up | Up | Up |

| Up | Right | Up |

| T.State | Right | T.State |

| Up | Right | Up |

| Right | Right | Up |

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |

The time taken: 0.00487518310546875 Number of iterations: 4 and optimal policy is:

| T.State | Right | T.State |

| Down | Right | Up |

| Right | Right | Up |