SC4003 CE/CZ4046
INTELLGENT AGENTS

Assignment 1

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Part 1, value iteration:

1. <u>Descriptions of implemented solutions:</u>

The board is 6x6, so there are 36 states. The action that we can take is up(u), down(d), left(l), right(r). Denote the outcome of taking an action as Outcome(action). Here, min_error is set to be 0.001.

The pseudocode is below:

```
function value_iteration
inputs:
A reward grid, with 0 as walls, -0.04 as empty spaces, +1 as positive rewards and -
1 as negative rewards, denote as r
gamma = 0.99
min_error = 0.001
Initiate a utility grid the same size as reward grid with each space as 0, denote
as util
Initiate a flag as True
While flag:
    Flag=False
for each state s in S do
    for each action a in {u,d,l,r} do
        calculate expected utility of action a, denote as U_a
    find maximum utility from { U_l ,U_d ,U_l ,U_r}, denote as U
util[s] = R[s] + gamma * U
if |util '[s] - util [s]| > min_error:
    flag = True
Return util
```

Here, `expected utility of action a` is as follows:

0.8*utility of state after taking a + 0.1*utility of a state 90 degrees to a + 0.1*utility of a state -90 degree to a

In my code, I use a **get_util** function to get the utility of state after action a. It takes in the input of two coordinates, the current coordinate, and the next coordinate. If the next coordinate is in walls, then return the utility of the current coordinate. If not, return the utility of the next coordinate. So, in above function, if the utility of a state 90 degrees to a is a wall, the get_util function will correctly return the state that the agent is currently in.

2. <u>Utilities of all states:</u>

I use (row,column) format to display utility.

(0, 0):90.28673703099518

(0, 1):0

(0, 2):85.67033517990042

(0, 3):84.59605956735648

(0, 4):83.4684624398819

(0, 5):84.23421366611227

(1, 0):88.79088236626853

```
(1, 1):86.38974779989127

(1, 2):85.16987614475947

(1, 3):85.13820861916281

(1, 4):0

(1, 5):81.91457675312205

(2, 0):87.4552305967836

(2, 1):86.1986363095398

(2, 2):84.01127132031264

(2, 3):84.01833088049253

(2, 4):84.04856162135566

(2, 5):82.84702697727458

(3, 0):86.16721937583227

(3, 1):85.15961138500153

(3, 2):84.03366170245809

(3, 3):82.05614033649435

(3, 4):82.86104177202844

(3, 5):83.02911157603411

(4, 0):85.01976588354405

(4, 1):0

(4, 2):0

(4, 3):0

(4, 4):80.69503296644798

(4, 5):81.796997630756

(5, 0):83.74913374957892

(5, 1):82.63232046800874

(5, 2):81.52965938339486

(5, 3):80.44097176598538

(5, 4):79.8064651229791

(5, 5):80.61640497945342
```

3. <u>Plot of utility estimates as a function of the number of iterations:</u>

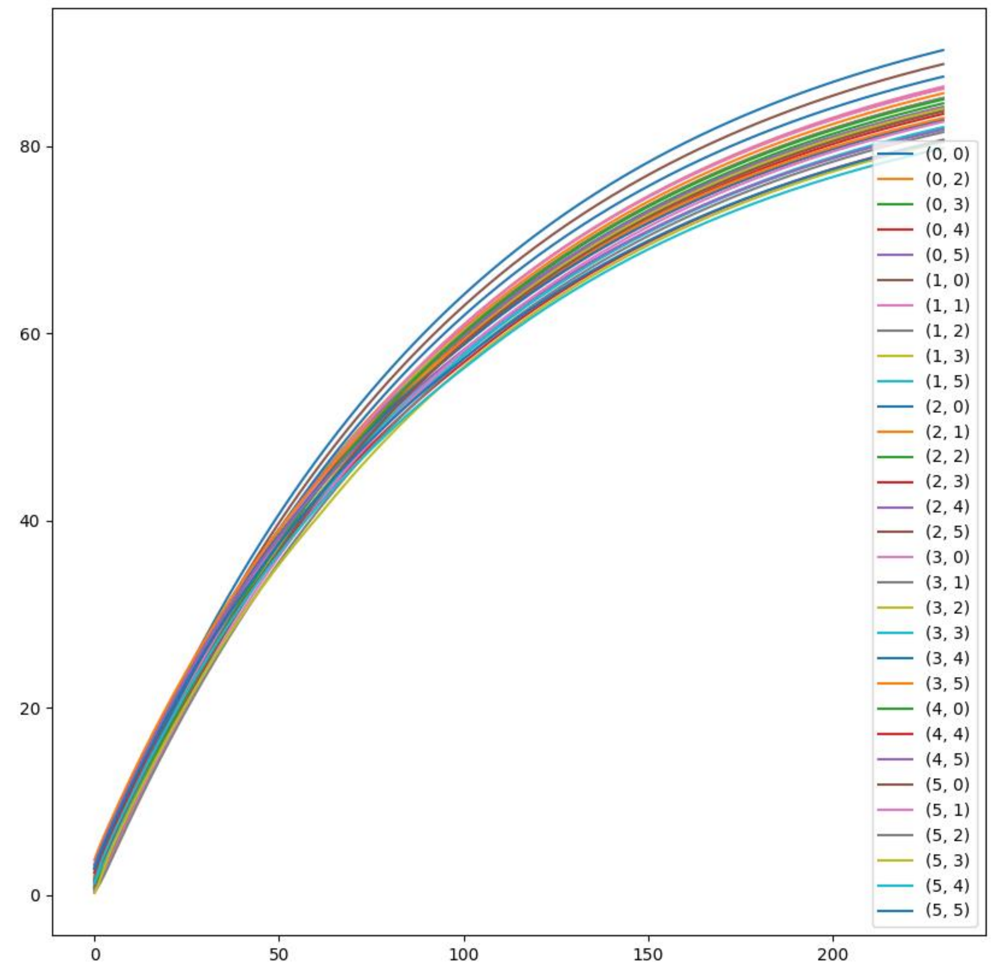


Figure 1-utility vs number of value iterations

4. <u>Optimal Policy</u>

Reference: up(u), down(d), left(l), right(r), 'na' means the place is walls

(0, 0):u

(0, 1):na

(0, 2):l

```
(0, 3):l

(0, 4):l

(0, 5):u

(1, 0):u

(1, 1):l

(1, 2):l

(1, 3):l

(1, 4):na

(1, 5):u

(2, 0):u

(2, 1):l

(2, 2):l

(2, 3):u

(2, 4):u

(2, 5):l

(3, 0):u

(3, 1):l

(3, 2):l

(3, 3):u

(3, 4):u

(3, 5):r

(4, 0):u

(4, 1):na

(4, 2):na

(4, 3):na

(4, 4):u

(4, 5):u

(5, 0):u

(5, 1):l

(5, 2):l

(5, 3):l
```

(5, 4):u

(5, 5):u

Part 1, Policy Iteration:

1. <u>Descriptions of implemented solutions:</u>

The pseudocode is below:

```
function policy_iteration
inputs:
A reward grid, with 0 as walls, -0.04 as empty spaces, +1 as positive rewards and -1 as negative rewards, denote as r
gamma = 0.99
error = 0.01
Initiate a utility grid the same size as reward grid with each space as 0, denote as util
Initiate a flag as True
Initiate a policy the same size as reward, default all action as left, denote as policy
While flag:
    Flag=False
    While utility not converged:
        for each state s in S do
            a = policy[s]
            calculate expected utility of action a, denote as $U_a$
            util[s]= R[s] + gamma * $U_a$
            if |util '[s] - util [s]| > min_error:
                utility is not converged
    for each state s in S do
        a = policy[s]
        calculate expected utility of action a, denote as U
        for each possible action from state s:
            calculate expected utility of action a1, denote as $U_{a1}$
        find maximum utility from { $U_l$ ,$U_d$ ,$U_l$ ,$U_r$}, record the action as a'
        if max({ $U_l$ ,$U_d$ ,$U_l$ ,$U_r$})>U:
            flag=True
            util[s]= max({ $U_l$ ,$U_d$ ,$U_l$ ,$U_r$})
            policy[s]=a'
Return policy
```

Here, the definition of "utility not converged" is whether the difference between old utility U(s) and new utility U'(s) is smaller than a user specified value. This will ideally give the best estimation of utility value of a given policy, but it will take a lot of iterations to accomplish this (about 1000) iterations for 0.001 min_error. To save time, I made an adjustment to the algorithm as the book suggested. Instead of setting the error threshold, the user can set a fixed number of iterations of policy evaluation. This will give a fairly accurate estimate while getting the optimal policy. The new algorithm will look something like below:

```
**function policy_iteration**
inputs:
A reward grid, with 0 as walls, -0.04 as empty spaces, +1 as positive rewards and -1 as negative rewards, denote as r
gamma = 0.99
threshold = 10
Initiate a utility grid the same size as reward grid with each space as 0, denote as util
Initiate a flag as True
Initiate a policy the same size as reward, default all action as left, denote as policy
While flag:
    Flag=False
    k = 0
    While k<threshold:
        for each state s in S do
            a = policy[s]
            calculate expected utility of action a, denote as $U_a$
            util[s]= R[s] + gamma * $U_a$
        k = k + 1
    for each state s in S do
        a = policy[s]
        calculate expected utility of action a, denote as U
        for each possible action from state s:
            calculate expected utility of action a1, denote as $U_{a1}$
        find maximum utility from { $U_l$ ,$U_d$ ,$U_l$ ,$U_r$}, record the action as a'
        if max({ $U_l$ ,$U_d$ ,$U_l$ ,$U_r$})>U:
            flag=True
            util[s]= max({ $U_l$ ,$U_d$ ,$U_l$ ,$U_r$})
            policy[s]=a'
Return policy
```

This will limit the number of inner iterations of policy evaluation as a fixed number, proportional to the number of states in the board. I will refer the old method as error-based method, and this new method as modified policy iteration.

2. Plot of optimal policy

Using the error-threshold based method:

(0, 0):u

(0, 1):na

(0, 2):l

(0, 3):l

(0, 4):l

(0, 5):u

(1, 0):u

(1, 1):l

(1, 2):l

(1, 3):l

(1, 4):na

(1, 5):u

(2, 0):u

(2, 1):l

(2, 2):l

(2, 3):u

(2, 4):l

(2, 5):l

(3, 0):u

(3, 1):l

(3, 2):l

(3, 3):u

(3, 4):u

(3, 5):u

(4, 0):u

(4, 1):na

(4, 2):na

(4, 3):na

(4, 4):u

(4, 5):u

(5, 0):u

(5, 1):l

(5, 2):l

(5, 3):l

(5, 4):u

(5, 5):u

Using modified policy iteration:

(0, 0):u

(0, 1):na

(0, 2):l

(0, 3):l

(0, 4):r

(0, 5):u

(1, 0):u

(1, 1):l

(1, 2):l

(1, 3):r

(1, 4):na

(1, 5):u

(2, 0):u

(2, 1):l

(2, 2):l

(2, 3):u

(2, 4):u

(2, 5):l

(3, 0):u

(3, 1):l

(3, 2):l

(3, 3):u

(3, 4):u

(3, 5):r

(4, 0):u

(4, 1):na

(4, 2):na

(4, 3):na

(4, 4):u

(4, 5):u

(5, 0):u

(5, 1):l

(5, 2):l

(5, 3):l

(5, 4):r

(5, 5):u

3. Utilities of all states

Using error-based method:

(0, 0):99.9931801219239

(0, 1):0

(0, 2):95.03887477149885

(0, 3):93.86848435014625

(0, 4):92.64816273591191

(0, 5):93.31864844761554

(1, 0):98.38661941615187

(1, 1):95.87635196868601

(1, 2):94.5384159062669

(1, 3):94.39120710275583

(1, 4):0

(1, 5):90.90796712695564

```
(2, 0):96.94183476557835
(2, 1):95.57983639361656
(2, 2):93.2879097230373
(2, 3):93.16981774199269
(2, 4):93.09593477791917
(2, 5):91.78808101610649
(3, 0):95.54724856623905
(3, 1):94.44596908180795
(3, 2):93.22608669994514
(3, 3):91.10883870718436
(3, 4):91.80796730254913
(3, 5):91.8813608541346
(4, 0):94.30599478204677
(4, 1):0
(4, 2):0
(4, 3):0
(4, 4):89.54189890841138
(4, 5):90.5600789483557
(5, 0):92.93102299753664
(5, 1):91.72239082348689
(5, 2):90.52882903531822
(5, 3):89.35014972138951
(5, 4):88.5625878237222
(5, 5):89.29101778748903
```

Using modified policy iteration:

```
(0, 0):58.341116225682256
(0, 1):0
(0, 2):54.86235031453036
(0, 3):54.074446998399154
(0, 4):53.38356999955859
```

```
(0, 5):54.57935785033926

(1, 0):57.2096147204974

(1, 1):55.167655766029426

(1, 2):54.33891114873495

(1, 3):54.425015141365066

(1, 4):0

(1, 5):52.559790796084535

(2, 0):56.23313856292177

(2, 1):55.32344775929071

(2, 2):53.48046994961688

(2, 3):53.67469906951431

(2, 4):54.073462108800776

(2, 5):53.2607115666858

(3, 0):55.29588444372478

(3, 1):54.59656590421955

(3, 2):53.779793841687834

(3, 3):52.065116047407535

(3, 4):53.211738766566775

(3, 5):53.62136116297584

(4, 0):54.457144300757655

(4, 1):0

(4, 2):0

(4, 3):0

(4, 4):51.37216507644101

(4, 5):52.69363079989277

(5, 0):53.52991240929574

(5, 1):52.71529134112841

(5, 2):51.911800547783336

(5, 3):51.11929151872999

(5, 4):50.77972914606042

(5, 5):51.81347372286636
```

4. Plot of utility estimates as a function of the number of iterations

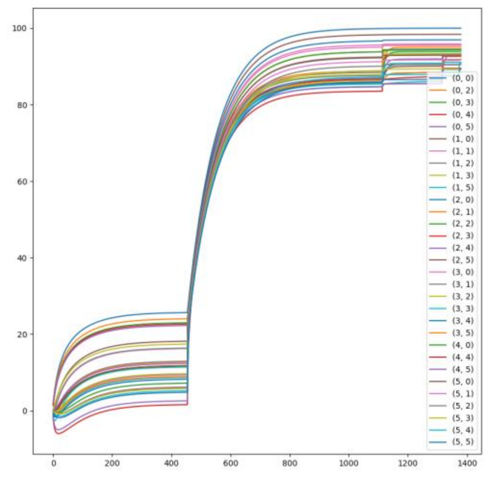


Figure 2-utility plot using error-based method

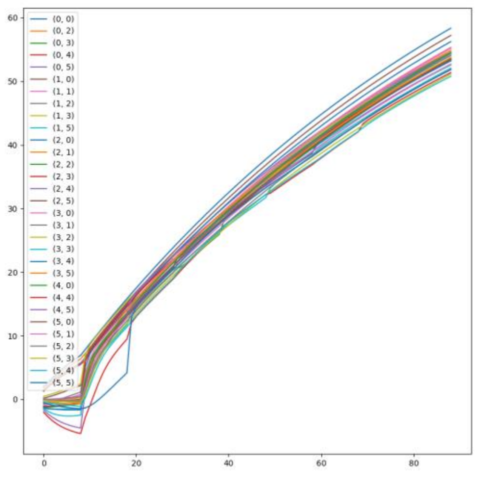


Figure 3-utility plot using modified policy iteration

Part 2: Design a more complicated maze environment of your own and re-run the algorithms designed for Part 1 on it. How does the number of states and the complexity of the environment affect convergence? How complex can you make the environment and still be able to learn the right policy?

Note: Due to the inherent randomness of the code in this part, the graphs may vary if you run the code, but the trend should be there.

Since complexity is a fairly difficult concept to define, I simplify it as three variables, namely, board size, percentage of walls in the board, and the percentage of reward states in the board.

I decided to create a function named **generate_random_board** to generate a board. The parameter size is the board size (for simplicity, I make all boards squares), wall_probability is the probability that a tile is a wall, and reward_state_prob is the probability of a tile is a reward state. If a tile is a reward state, I will further let computer to flip a coin to decide if they are +1 or -1. Then I plot the graph of number of iterations needed for both methods while changing size, wall, and reward probability separately. Due to the constraint of my computer and time, I can only generate size from 5x5 to 30x30. Since there is some randomness involved, for each size, I generated 5 boards and take the average. The wall_probability and reward_state_prob is fixed as 0.2 for this experiment. I run both value iteration(VI) and policy iteration(PI) for this experiment. The policy iteration algorithm used is the modified version to save time.

The graph is shown below:

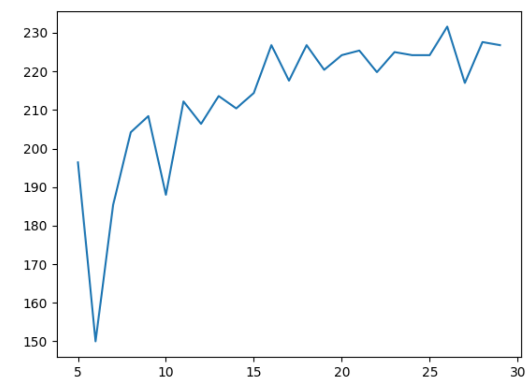


Figure 4-number of iteration vs size using VI

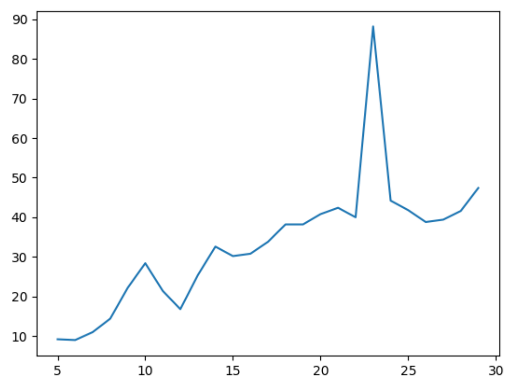


Figure 5- number of iteration vs size using PI

We can see both graph, number of iteration needed to reach convergeance increases with size. The policy iteration shows a flatter curve because I fixed the iteration threshold at 10 for all sizes, resulting in a not-so-accurate policy evaluation result. If the size increases to a certain number, the constraint of memory and processing speed of our current computer may not be able to handle it, but theoretically, it should still be solvable since there will still be n variables in n equations, and the matrix's column are all linearly independent.

Then, for wall_probability in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9], I do the same while fixing the size as 30. The reward_state_prob is fixed as 0.2 as well. The graph is as shown:

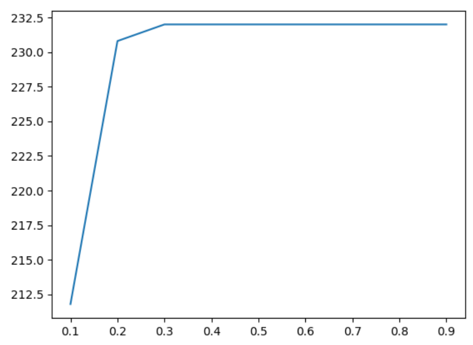


Figure 6- number of iterations vs wall probability using VI

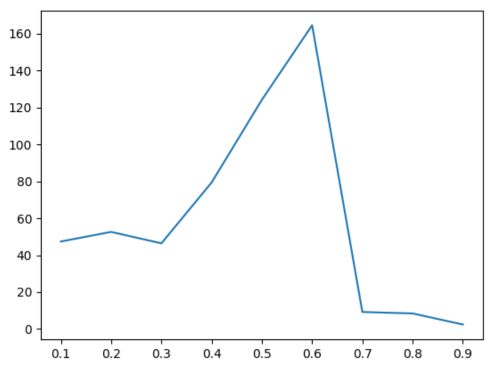
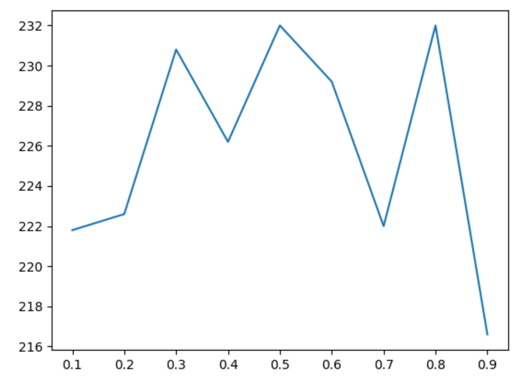


Figure 7- number of iterations vs wall probability using PI

Value iteration appears to have a relatively stable number of iterations while policy iteration peaks at 0.4-0.6. It makes sense because if the percentage of walls are a lot in a board game, the number of routes reaching the reward state is limited and may be easier to find. The reward state is also likely to be found near walls, and all the agent needs is to crash into walls. It is also relatively easy for very few walls because most routes will just be straight lines. In the middle probability (around 0.5), however, depending on the wall configuration, it will be more difficult to learn the right policy(seen as increased number of iteration needed)- or not getting it at all, if the error threshold set is too large, or the number of iteration threshold set is too small.

Lastly, Then, for reward_state_prob in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9], I do the same while fixing the size as 30. The wall_probability is fixed as 0.2 as well. The graph is as shown:



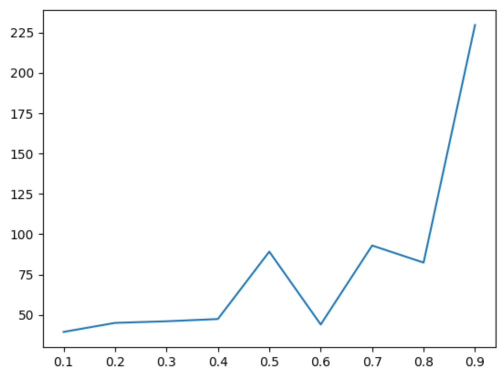


Figure 8- number of iterations vs reward probability using VI Figure 9-number of iterations vs reward probability using PI

Interestingly, value iteration shows a peak at 0.2-0.8 whereas policy iteration shows an exponential increase. The former makes sense as high number of rewards indicate that you can just try to move towards a reward point on the wall and keep crashing into the wall, and on your wall, you will most likely encounter reward points instead of empty spaces with penalties. Then value iteration will just reinforce the positive reward routes, leading to easy convergence. For policy iteration, the reason it takes lots of iteration may be because it is hard to choose whether to move when you are surrounded by so many rewards, and moving everywhere is the same, so at some squares the best action fluctuates a lot, leading to lots of iterations possibly the inability to converge.

If I have more time and a better computer, it is best to run more combinations of board sizes, percentage of walls in the board, and the percentage of reward states in the board and see if the above hypothesis holds.