Packet Scheduling using Dynamic Programming

Project 1

Sadra Hemmati

Abstract:

Dynamic programming is a machine learning technique that utilizes knowledge of the environment, computational resources, and the Markov property in order to create a policy to optimally perform in an environment. With this powerful technique, a seemingly complex problem with many possibilities can be broken down and solved with a few lines of code. In this report, we cover the entire process of solving a multi-queue networking problem with a Matlab based dynamic program. Based on the obtained results, we conclude that dynamic programming with policy iteration is an effective and informative solution for this problem

Introduction and Background:

In communication network design, a network scheduler is an arbitrar that manages the sequence of network packets. The arbitrar behaves as a controller and decides which packet to pass when there are multiple ones incoming. There are different architectures for the communication systems such as FIFO (first-in-first-out). The queuing theory is the foundation for study of waiting lines. After creating the model queue length and waiting time can be predicted based on the distribution on which the packets arrive. However, there are many different ways to decide how to server a queue. One method for deciding is to use Artificial Intelligence techniques.

After transferring the queueing problem into AI domain, deployment of domain techniques such as Dynamic programming makes it feasible to optimize the performance of such systems.

Dynamic programming is the collection of algorithms that help find the optimal policy for an AI agent when it is defined as a Markov Decision Process(MDP). The assumption in Dynamic Programming is that the agent has the perfect model of the environment, (i.e.the transition probabilities and the state-action pairs). In our scenario, a model is given for two different queues with M slots and two λ probabilities of receiving a packet seen in Figure 1. At each point in time, only one queue can be served. How we determined to choose which queue to serve is described in the next section.

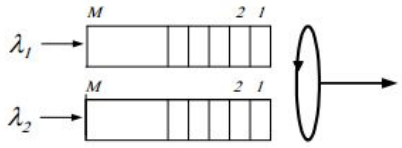


Figure 1

Design:

There are two possible models for queueing theory, early arrival and late arrival. In these two scenarios the agent decides to perform the action before or after the arrival of a new packet. For our design, we decided to assume that an action to remove a packet could only be done if the queue already had a packet. This means that an packet arriving cannot instantly be cleared from the queue before getting stored in the buffer.

a) We formulate this problem as a Markov Decision Process in which the states are the number of packets in each queue and there are two possible actions : serving queue one or serving queue two. The given conditions state that the rewards that the agent can get from the environment is 1 for each successful packet serving and -5 for overflow of packets in a queue.
Our state to state transition probabilities were formulated as the probability of receiving a packet in transitioning the current state while being offset by the action of removing a packet.
We could have chosen to have three actions instead of two actions (adding doing no action) however due to obvious benefit of serving packets at the all time, we decided to choose the more simple two-action scenario. In other words as long as unserved packets exist, it is sensible to continue serving packets by taking actions one or two. The only problematic case is when there are no packets in the queues in which case the state is not changed by the action of removing a packet but still has transition probabilities from incoming packets

Regarding the agent and boundary assessment it is noteworthy to mention that the agent is defined as the controller of the actions of serving queue 1 or serving queue 2. Since it has no control over the rewards that it receives we regard the environment the process that generates packets based on a distribution and gives rewards to the agent.

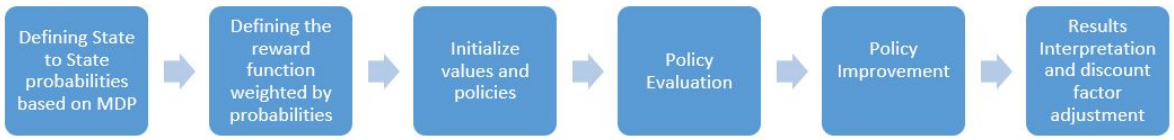
Design Objective:

Our main goal is to create an algorithm to reduce the drop rate of packets arrived. We must formulate the problem in a way that creates sensible value states and correctly represents the model. After running the algorithm, the policy evaluation should converge to a reasonably small threshold and the policy should converge into an optimal policy that achieves the highest reward possible.

Design Challenges:

The main challenge was modelling and defining all the edge cases for the state transition probabilities such as having an empty or full queue. Additionally, because we didn't design our model with an overflow state, we had to determine the weighted reward probabilities for the edge states. The solution to these challenges is described more in the technical approach.

Technical Approach:



Flow Chart 1

Step 1)

$$P_{ss'}^{a} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$$

Formula 1

The first step of the design was to implement a function to find the transition probability seen in Formula 1. This was done by creating a 9x9x9x9 4 dimensional matrix that contains the combinations for the current state and the next state. This matrix creation script can be seen in the applyprob.m file at the end of the report. The applyprob.m file calculates the probability of transition between different states based on 4 basic probabilities of :

1) 2 packet arrived at both ques w/p of 0.3*0.6=0.18
2) no packet arrived at any ques w/p of 0.7*0.4=0.28
3) packet arrived at just the first queue w/p of 0.3*0.4=0.12
4) packet arrived at just the second queue w/p 0.6*0.7=0.42

The input to this matlab function is the number of the current packets in each queue and the output is the probability of transition into another state.

The probability of getting a packet at que1 is 0.3 and the probability of getting a packet at queue 2 is 0.6. So there are 4 different situations that we can anticipate:

	Packet arrived to 1	packet not arrived to 1
packet arrived to 2	0.3*0.6=0.18	0.6*0.7=0.42
packet not arrived to 2	0.3*0.4=0.12	0.7*0.4=0.28

An important thing to note here is to take into account the different possibilities that we have to take into account the marginal states which are the ones in which one of the packets is full. For example if queue 1 is full but que 2 is not full, (if(row~=9 & col==9)) the probability of queue 2 becoming full in the next transition is

matrix(row+1,9,row,col) = .42 + .18

and the probability of staying in this state is :

matrix(row,9,row,col) = .28 + .12

Each array of state2state matrix is a 9 by 9 matrix in which there are lots of zeros and some number of probability distributions. For example in for queue1 = 3 & queue2 = 6

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0.28	0.12	0
0	0	0	0	0	0	0.42	0.18	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

The next script, tranP.m, wraps around the results of applyprob.m in order to complete the transition probability. applyprob.m adds in transition probabilities from a state to a next state based on incoming packets and tranP.m offsets the input to these probability matrices based on an action. The final result is a function that can be called with a given state, action, and target state that returns the transition probability.

Step 2)

$$R^{a}_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}$$

Formula 2

The next step was to implement the reward function seen in Formula 2. This function and its different cases can be seen in applyReward.m. It outputs two 9x9x9x9 matrixes that tell the reward from state to state. One matrix for taking action one and one for action two. Based on the the current state, next state and the action taken, the expected value of getting a reward is calculated.

The rewards were generally 1 with the special case of 0 when both queues are empty. When full, the probability of getting a negative reward becomes 60% when taking action 1 and 30% when taking action two due to the nature of removing a packet taking away the possibility of the queue it was removed from overflowing.

For example, for the case of

```
if((q2==9) & (q1<9)& (q1~=1))
        matrix(q2,q1-1,q2,q1) = 1 + .60*(-5);
        matrix(q2,q1,q2,q1) = 1 + .60*(-5)
    end
```

For example if action taken is action one and the current state is a full queue two and the number of packets in the queue one is between 1 and less than 8(not full) the rewards that the agent will get is dependent on the next action for which it will transfer to which is dependent on

packet arrival. In our case the agent will receive a reward of 1 for serving one packet from the queue one but the chance of getting negative reward of (-5) depends on the chance of arriving a packet to queue 2 which is 0.6.

All other rewards are calculated likewise.

A similar wrapper function to tranP was made called tranR.m that takes in an input in order to choose which 9x9x9x9 matrix to use.

Step 3)

1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

Pseudocode 1

The code setup.m implements the initialization of the program. It creates all the transition probability matrixes and initialises the initial value function and policy seen in Pseudocode 1. The initial values are set to 0 and the policy is set to initially take action one. Action one was chosen because it is theoretically non-ideal so it is obvious when the algorithm optimizes the policy.

Step 4 & 5)

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

Pseudocode 2

3. Policy Improvement

policy-stable $\leftarrow$ *true*

For each $s \in \mathcal{S}$:

$b \leftarrow \pi(s)$

$\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s') \right]$

If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*

If *policy-stable*, then stop; else go to 2

Pseudocode 3

policyIteration.m implements the main function of the dynamic programming. It uses the previously implemented functions to create the value and optimal policy functions. The code follows very closely to Pseudocode 2 and 3.

Step 6)

Result interpretation is discussed in the next section. After trial and error, a discount rate of .4 was chosen to produce optimal results.

Experiments and Results:

The pictures below show the results of the programming at different stages of the policy refinement. At all the images below,the left part is the value function and the right one is the policy associated with it. The relation between the policy and the value of each state has been explained in each part. For all of the tables, the rows represent queue 1 and the columns represent queue 2. For example, cell 0,0 represents the value of queue 1 and queue 2 being empty. Alternatively, the policy tables display a 1 for action one or a 2 for action two. Progression for the value function and policy improvement is plotted below with color gradients.

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

The value function(Left) and the correlated policy (Right) at the beginning

We initialized the dynamic programming with the policy of taking action 1 and with values of zero for all the states.After the first iteration, the last row which represents the value of packets in queue two are the least since with the policy of only taking action 1, the possibility of getting negative reward is very high.

0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
0	1	1.084	1.091056	1.091649	1.091698	1.091703	1.091703	1.091703
-3	-2.63	-2.5523	-2.53598	-2.53256	-2.53184	-2.53169	-2.53165	-2.53165

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

The value function(Left) and the correlated policy (Right) at the first iteration

0.127713	1.127898	1.358797	1.412079	1.424255	1.426906	1.427423	1.427513	1.427527
0.127713	1.127898	1.358797	1.412079	1.424255	1.426906	1.427423	1.427513	1.427527
0.127713	1.127898	1.358797	1.412079	1.424255	1.426906	1.427423	1.427513	1.427527
0.127713	1.127898	1.358797	1.412079	1.424255	1.426906	1.427423	1.427513	1.427527
0.124106	1.123988	1.354554	1.407451	1.419218	1.421588	1.42203	1.42211	1.422124
0.096858	1.096093	1.326051	1.378303	1.389615	1.39201	1.392516	1.392623	1.392646
-0.04547	0.953339	1.18284	1.234609	1.246247	1.248863	1.249451	1.249583	1.249613
-0.74355	0.255289	0.484052	0.536502	0.548523	0.551276	0.551905	0.552049	0.552082
-4.15304	-3.15576	-2.92589	-2.87293	-2.86073	-2.85793	-2.85728	-2.85713	-2.8571

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

The value function(Left) and the correlated policy (Right) at the fifth iteration

After the fifth iteration, the policy have not changed and the values of different states shows that being in the queue 2 is correlated to negative rewards(red blocks) and the best states to be are closest to top, right part which have the high possibility of getting reward of 1 after serving a packet from queue one.

0.316355	1.316355	1.417249	1.427429	1.428456	1.42856	1.42857	1.428571	1.428571
1.316355	1.417249	1.427429	1.428456	1.42856	1.42857	1.428571	1.428571	1.428571
1.417249	1.427429	1.428456	1.42856	1.42857	1.428571	1.428571	1.428571	1.428571
1.427429	1.428456	1.42856	1.42857	1.428571	1.428571	1.428571	1.428571	1.428569
1.428456	1.42856	1.42857	1.428571	1.428571	1.428571	1.428571	1.428569	1.428539
1.42856	1.42857	1.428571	1.428571	1.428571	1.428571	1.428569	1.428539	1.428067
1.42857	1.428571	1.428571	1.428571	1.428571	1.428569	1.428539	1.428067	1.420802
1.428571	1.428571	1.428571	1.428571	1.428569	1.428539	1.428067	1.420802	1.308787
1.428571	1.428571	1.428571	1.428569	1.428539	1.428067	1.420802	1.308787	-0.41823

2	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
2	2	2	1	1	1	1	1	1
2	2	2	2	1	1	1	1	1
2	2	2	2	1	1	1	1	1
2	2	2	2	1	1	1	1	1
2	2	2	2	2	1	1	1	1
2	2	2	2	2	2	2	2	2

The value function(Left) and the correlated policy (Right) at the21th iteration

After 21 iteration, the policy improvement results in adding more of taking the action 2 in the policy matrix. The value of states at the bottom row start to become more since now the agent can take action 2 hence reducing the high risk of negative rewards compared to previous states.

0.316355	1.316355	1.417249	1.427429	1.428456	1.42856	1.42857	1.428571	1.428571
1.316355	1.417249	1.427429	1.428456	1.42856	1.42857	1.428571	1.428571	1.428571
1.417249	1.427429	1.428456	1.42856	1.42857	1.428571	1.428571	1.428571	1.428571
1.427429	1.428456	1.42856	1.42857	1.428571	1.428571	1.428571	1.428571	1.428569
1.428456	1.42856	1.42857	1.428571	1.428571	1.428571	1.428571	1.428569	1.428539
1.42856	1.42857	1.428571	1.428571	1.428571	1.428571	1.428569	1.428539	1.428068
1.42857	1.428571	1.428571	1.428571	1.428571	1.428569	1.428539	1.428068	1.420802
1.428571	1.428571	1.428571	1.428571	1.428569	1.428539	1.428068	1.420802	1.308787
1.428571	1.428571	1.428571	1.428569	1.428539	1.428068	1.420802	1.308787	-0.41823

2	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	1
2	2	2	2	2	2	2	2	2

The value function(Left) and the correlated policy (Right) at the last iteration

At the final step, it is evident that the convergence of the policy iteration stage has led to policy of taking action 2 most of the time and taking action 1 just at the points. At the right margin where the occurrence of negative reward due to not serving the queue one is highly probable, the policy has been tuned to suggest taking action 1 to the agent(action 1 is more rewarding at these states). The block at the end row and end column has the lowest reward due to high possibility of negative reward.

The overall trend of the pictures suggests that beginning the “bad” policy of “taking action 1 at every state ”,the dynamic programming has led to a better policy which gives more rewards. It can be hypothesized that a different negative reward system (such as getting less reward for serving a disproportionate queue) could lead to more balanced results.

The final results show that the rewards stabilize once this *optimal* policy is reached. Because of how we defined the ordering of our actions versus incoming packets, the probability of getting a negative reward becomes zero at the perimeter if you always serve the full queue. Consequently, the only time it is possible to get a negative reward is in the state they are both full or a zero reward when they are both empty. This can clearly be seen in the value plot.

Summary

In this project, the first step was to formulate the problem of packet networking in the format of an MDP and defining the transition probabilities based on the “ late packet arrival” model. Then the rewards of different action-states was defined. In the dynamic programming section, the results of previous sections were used in the Bellman equation for policy iteration and improvement. Different discounting rates was tested and the optimal value of 0.4 was found. The effect of policy iteration on the value of different states have been discussed and explained and the power of dynamic programming techniques on this problem have been demonstrated. Ultimately, this project taught us how to formulate a problem as a MDP and gave us experience implementing our own dynamic program.

Appendix A:

Matlab Code:

applyProb.m -

```
function [ matrix ] = applyProb( matrix,row,col )
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

matrix(row,col,row,col) = .28
if(row<9),
matrix(row+1,col,row,col) = .42
else
end
if(col<9),
matrix(row,col+1,row,col) = .12
end
if(row<9),
    if(col<9),
        matrix(row+1,col+1,row,col) = .18
    end
end
if(row==9 & col==9)
    matrix(row,col,row,col) = 1
end
if(row~=9 & col==9)
    matrix(row+1,9,row,col) = .42 + .18
    matrix(row,9,row,col) = .28 + .12
end
if(row==9 & col~=9)
    matrix(9,col+1,row,col) = .12 + .18
    matrix(9,col,row,col) = .28 + .42
end

end
```

tranP.m -

```
function [ probability ] = tranP(que1,que2,action,que1Next,que2Next,state2state)
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here

que1 = que1 + 1;
que2 = que2 + 1;
que1Next = que1Next + 1;
que2Next = que2Next + 1;

if(action == 1)%
    if(que1-1 == 0)
    probability = state2state(que2Next,que1Next,que2,1);
    else
    probability = state2state(que2Next,que1Next,que2,que1-1);
    end
end

if(action == 2)
    if(que2-1 == 0)
    probability = state2state(que2Next,que1Next,1,que1);
    else
    probability = state2state(que2Next,que1Next,que2-1,que1);
    end
end

end
```

applyReward.m -

```
function [ matrix ] = applyReward( matrix,q2,q1,action)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

%q2s are buffer 2 q1s are buffer 1

if(action==1)
    if((q2==1) & (q1==1))
        matrix(q2,q1,q2,q1) = 0;
        matrix(q2+1,q1,q2,q1) = 0;
    end

    if((q2==1) & (q1<9)& (q1~=1))
            matrix(q2,q1-1,q2,q1) = 1;
            matrix(q2,q1,q2,q1) = 1;
            matrix(q2+1,q1,q2,q1) = 1;
            matrix(q2+1,q1-1,q2,q1) = 1
    end
    if((q2<9) & (q1==1)& (q2~=1))
            matrix(q2,q1,q2,q1) = 0;
            matrix(q2+1,q1,q2,q1) = 0;
    end
     if((q2==1) & (q1==9))
            matrix(q2,q1-1,q2,q1) = 1;
            matrix(q2,q1,q2,q1) = 1;
            matrix(q2+1,q1,q2,q1) = 1;
            matrix(q2+1,q1-1,q2,q1) = 1
    end
    if((q2==9) & (q1==1))
            matrix(q2,q1,q2,q1) = 0 + .6*-5;
            matrix(q2,q1+1,q2,q1) = 0 + .6*-5;
    end
    %-----------------------------------
    %Middle
    if((q2>1) & (q1>1))
        if((q2<9) & (q1<9))
            matrix(q2,q1-1,q2,q1) = 1;
            matrix(q2,q1,q2,q1) = 1;
            matrix(q2+1,q1,q2,q1) = 1;
            matrix(q2+1,q1-1,q2,q1) = 1
        end
    %-----------------------------------
    %Fulls
        if((q2==9) & (q1<9)& (q1~=1))
            matrix(q2,q1-1,q2,q1) = 1 + .60*(-5);
            matrix(q2,q1,q2,q1) = 1 + .60*(-5)
        end

        if((q2<9) & (q1==9)& (q2~=1))
            matrix(q2,q1-1,q2,q1) = 1; %nothing
            matrix(q2+1,q1-1,q2,q1) = 1; %#2 arrival
            matrix(q2,q1,q2,q1) = 1; %#1 arrival
            matrix(q2+1,q1,q2,q1) = 1 %#both arrival
        end

        if((q2==9) & (q1==9))
            matrix(q2,q1-1,q2,q1) = 1 + .60*(-5);
            matrix(q2,q1,q2,q1) = 1 + .60*(-5)
        end

    end
end

if(action==2)
```

```
    if((q2==1) & (q1==1))
        matrix(q2,q1,q2,q1) = 0;
        matrix(q2+1,q1,q2,q1) = 0;
    end

    if((q2==1) & (q1<9)& (q1~=1))
            matrix(q2,q1,q2,q1) = 0;
            matrix(q2+1,q1,q2,q1) = 0;
    end
    if((q2<9) & (q1==1)& (q2~=1))
            matrix(q2-1,q1,q2,q1) = 1;
            matrix(q2,q1,q2,q1) = 1;
            matrix(q2,q1+1,q2,q1) = 1;
            matrix(q2-1,q1+1,q2,q1) = 1
    end
     if((q2==1) & (q1==9))
            matrix(q2,q1,q2,q1) = 0 + .3*-5;
            matrix(q2+1,q1,q2,q1) = 0 + .3*-5;
    end
    if((q2==9) & (q1==1))
            matrix(q2-1,q1,q2,q1) = 1;
            matrix(q2,q1,q2,q1) = 1;
            matrix(q2,q1+1,q2,q1) = 1;
            matrix(q2-1,q1+1,q2,q1) = 1
    end
    %-------------------------------------
    %Middle
    if((q2>1) & (q1>1))
        if((q2<9) & (q1<9))
            matrix(q2-1,q1,q2,q1) = 1;
            matrix(q2,q1,q2,q1) = 1;
            matrix(q2,q1+1,q2,q1) = 1;
            matrix(q2-1,q1+1,q2,q1) = 1
        end
    %-------------------------------------
    %Fulls
        if((q2==9) & (q1<9)& (q1~=1))
            matrix(q2-1,q1,q2,q1) = 1; %nothing
            matrix(q2-1,q1+1,q2,q1) = 1; %#2 arrival
            matrix(q2,q1,q2,q1) = 1; %#1 arrival
            matrix(q2,q1+1,q2,q1) = 1 %#both arrival
        end

        if((q2<9) & (q1==9)& (q2~=1))
            matrix(q2-1,q1,q2,q1) = 1 + .30*(-5);
            matrix(q2,q1,q2,q1) = 1 + .30*(-5)
        end

        if((q2==9) & (q1==9))
            matrix(q2-1,q1,q2,q1) = 1 + .30*(-5);
            matrix(q2,q1,q2,q1) = 1 + .30*(-5)
        end

    end
end
```

tranR.m -

```
function [ reward ] = tranR(que1,que2,action,que1Next,que2Next,rewardfor1,rewardfor2)
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here

que1 = que1 + 1;
que2 = que2 + 1;
que1Next = que1Next + 1;
que2Next = que2Next + 1;

if(action == 1)%
    reward = rewardfor1(que2Next,que1Next,que2,que1);
end

if(action == 2)
    reward = rewardfor2(que2Next,que1Next,que2,que1);
end
end
```

setup.m -

```
clear all,
clc
state2state = zeros(9 , 9 , 9 ,9)
rewardfor1 = zeros(9 , 9 , 9 ,9)
rewardfor2 = zeros(9 , 9 , 9 ,9)
policy = zeros(9 ,9)
values = zeros(9 ,9)

none=.7*.4
just1=.3*.4
just2=.6*.7
both=.6*.3

for i = 1:9
  for j = 1:9
    state2state = applyProb(state2state,i,j);
  end
end
stateDone = 1

for i = 1:9
  for j = 1:9
    rewardfor1 = applyReward(rewardfor1,i,j,1);
  end
end

reward1Done = 1
for i = 1:9
  for j = 1:9
    rewardfor2 = applyReward(rewardfor2,i,j,2);
  end
end

%randomize policy
for i = 1:9
  for j = 1:9
    policy(i,j) = 1;
    %policy(i,j) = binornd(1,.5) + 1;
  end
end

reward2Done = 1

setupDone = 1
```

policyIteration.m -

```
run setup.m

iterating = 1
total = 0;
while iterating == 1
    total=total+1
    %evaluation
    evaluating = 1
    while evaluating == 1

        theta = 0.001;
        delta = 0;
        discount = .4;

        for q2 = 1:9
          for q1 = 1:9
            v = values(q2,q1);
            sumOfValues = 0;
            for nextq2 = 1:9
              for nextq1 = 1:9
                  myProb = tranP(q1-1,q2-1,policy(q2,q1),nextq1-1,nextq2-1,state2state);
                  myReward = tranR(q1-1,q2-1,policy(q2,q1),nextq1-1,nextq2-1,rewardfor1,rewardfor2);
                  myNextVal = values(nextq2,nextq1);
                  sumOfValues = sumOfValues + myProb*(myReward + discount*myNextVal);

              end
            end
            values(q2,q1) = sumOfValues;
            delta = max(delta,abs(v - values(q2,q1)))
          end
        end

        if(delta < theta)
           evaluating = 0;
        end

    end

    %improvment
        policyStable = 1
        for q2 = 1:9
          for q1 = 1:9
              b = policy(q2,q1);
               action1sumOfValues = 0;
               action2sumOfValues = 0;
                for nextq2 = 1:9
                  for nextq1 = 1:9
                      myProb = tranP(q1-1,q2-1,1,nextq1-1,nextq2-1,state2state);
                      myReward = tranR(q1-1,q2-1,1,nextq1-1,nextq2-1,rewardfor1,rewardfor2);
                      myNextVal = values(nextq2,nextq1);
                      action1sumOfValues = action1sumOfValues + myProb*(myReward + discount*myNextVal);
                  end
                end

                for nextq2 = 1:9
                  for nextq1 = 1:9
                      myProb = tranP(q1-1,q2-1,2,nextq1-1,nextq2-1,state2state);
                      myReward = tranR(q1-1,q2-1,2,nextq1-1,nextq2-1,rewardfor1,rewardfor2);
                      myNextVal = values(nextq2,nextq1);
                      action2sumOfValues = action2sumOfValues + myProb*(myReward + discount*myNextVal);
                  end
                end
               if(action1sumOfValues>action2sumOfValues)
                  policy(q2,q1) = 1;
```

```
            elseif(action2sumOfValues>=action1sumOfValues)
              policy(q2,q1) = 2;
            end

            if(b~=policy(q2,q1))
                policyStable = 0;
            end
          end
        end

        if(policyStable==1)
            iterating = 0
        end

end
```