Abstract

In this experiment, we modeled real-time packet scheduling of 2 queues, each having different arrival rates and contending for the same output port, as a Markov Decision Problem. The optimal policy for selecting a queue for packet transmission was determined by policy iteration using two different reward functions, and then the results were compared by examining plots of the optimal policy and value function.
I. Background

Markov decision processes model decision-making in situations where the outcomes depend partially on random variables and on the actions of an agent. These models are useful for examining many optimization problems, such as those solved by reinforcement learning. Reinforcement learning is an area of machine learning concerned with teaching an agent to perform optimally in regards to a notion of cumulative reward. Dynamic programming is one family of algorithms within reinforcement learning which requires an accurate model of the environment. These algorithms are obtained by turning Bellman optimality equations into update rules. By formulating the environment as a Markov decision process, it is possible to use dynamic programming techniques to reach an optimal policy. The dynamic programming method used in this project is policy iteration, which has two major steps: policy evaluation and policy improvement. Policy evaluation updates the value function based on the current policy, while policy improvement updates the policy based on that new value function. These two steps eventually converge to an optimal policy and corresponding value function.

II. Introduction

In this experiment, we attempt to optimize the real-time packet scheduling of two independent first-in, first-out queues. All packets are assumed to have the same length and each time slot is identical in duration to the time it takes to transmit a packet. Each queue has a distinct packet arrival rate which follows a Bernoulli i.i.d. process; queue 1 has an arrival rate of \( \lambda_1 = 0.3 \) and queue 2 has an arrival rate of \( \lambda_2 = 0.6 \). The number of possible arrivals during the same time slot ranges from 0 to 2. Both queues have a maximum packet capacity of 8 packets. The agent will receive a reward of +1 for each packet that successfully departs the system and a reward of -5 for each packet that is lost due to queue overflow. In addition, we assume that packet arrivals and departures happen simultaneously, so that pulling from an empty queue while a packet arrives in that queue results in a reward of 0 and a queue size of 1. Our agent must determine the optimal policy for reducing packet loss by minimizing the size of the queues.

III. Design

Challenges

The primary challenge in modeling the MDP is that there are several special cases, since we must account for either queue being either empty or full or both queues being empty or full. As a result, the logical way to handle program execution is a series of “if” statements checking the sizes of the queues to handle these exceptions. Since this makes the code a bit clunky, we chose to model the transition probabilities and reward function in matrices separate from this code. The program then uses the queue sizes to select the appropriate probability and reward matrices which are then used to evaluate the value function and policy during the appropriate phases.
Objectives and Approach
The primary objectives in our design were simplicity and extendibility. We wanted to model the problem in a way that made sense and was easy to work with, so we represented our value function and policy in 2D matrices indexed on queue size.

Since there were only two possible actions (popping from queue 1 or popping from queue 2), we used only one matrix for the policy. This represented action 1. Action 2 was therefore simply 1 minus the policy for a particular state.

We chose to represent the rewards and state transition probabilities in 3D matrices indexed on the possible next state for each queue 1 and queue 2 and the action that led to that next state. Since the rewards and probabilities of transition change when a queue is either full or empty, we chose to represent these in separate matrices each representing the possible next-states for its particular current state. These matrices are listed in the appendix, but the following example should make this clear:

If \( i = |\text{Queue1}| \) and \( j = |\text{Queue2}| \), then the possible next states are

\[
\begin{bmatrix}
  i - 1, j - 1 & i - 1, j & i - 1, j + 1 \\
  i + 1, j - 1 & i + 1, j & i + 1, j + 1 \\
  i, j - 1 & i, j & i, j + 1
\end{bmatrix}
\]

The probability of transitioning from state \((i,j)\) to each of the above states depends on the arrival rate of new packets and the action taken leading to the next state. They are summed up for the general case \((i,j)\) in the following chart:

Thus, the probability of transitioning from \((i,j)\) to the above states is

\[
\begin{bmatrix}
  0 & .28 & .42 \\
  0 & .12 & .18 \\
  0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  0 & 0 & 0 \\
  .28 & .42 & 0 \\
  .12 & .18 & 0
\end{bmatrix}
\]

where the first matrix represents the probability of transition given action 1 and the second matrix represents the probability of transition given action 2.
Technical approach
The following flowchart outlines our technical approach. In essence, this is the basic design for dynamic programming; however our approach requires the initialization of the transition probability and reward matrices.

Part I
In Part I, we used the supplied reward function. The agent received a +1 reward for each packet that departed the system and a -5 reward whenever a packet was dropped (i.e. queue overflow). To do this, we set our reward matrices to the proper reward value. In most cases, this value was either 0 or 1, most often it was pulling from either an empty queue or non-full one. However, if one queue was full, there
was a certain probability that the action would result in either a reward of -5 or -4 or 0 or 1 (depending on whether or not it pulled a packet). In these cases, we simply used the expected value for the particular state, which weighted the probability of each reward.

Figure 1 shows the progression of policy improvement. Initially, we set the policy to .5 for all states, which means that it pulled from each queue with equal probability. During policy improvement, the action resulting in the greatest value was chosen, so after the initial setup, the policy has values of either 1 (pull from queue 1) or 0 (pull from queue 2).

Figure 1: The policy starts with every state at .5, and then picks the action resulting in the greatest value.

Figure 2 shows the progression of the value function. The value function was initialized to zero for all states, and the program successively estimates the value for the particular policy. Unsurprisingly, the value increases over time, as most states will result in a positive reward.
Figure 2: The value of the states increase with successive iterations.

Figure 3 shows the optimal policy for this model. As expected, this policy prefers queue 2 over queue 1, as queue 2 is more likely to overflow. However, as queue 1 fills, if queue 2 is relatively empty, the policy chooses queue 1 to prevent it from overflowing.

Figure 3: Red signifies pulling from queue 1, and blue is pulling from queue 2.
Figure 4. shows the value function for the optimal policy. As expected, the values for states appearing in the middle (i.e. when neither queue is approaching fullness) are worth more than those near the edges (i.e. when a queue is either empty or full). This is unsurprising, as a full queue is likely to result in a negative reward while an empty queue will result in no reward. This seems to imply that the policy is indeed optimal.

Figure 4: The value of states representing either or both queues near fullness are low, as they are more likely to result in negative rewards.

**Part II**

In Part II, we changed the reward function to prefer empty queues. As before, the agent received a +1 reward for each packet that departed the system and a -5 reward whenever a packet was dropped (i.e. queue overflow). However, in addition to these, the agent received a small negative reward equal to $\frac{1}{4} (|Queue1| + |Queue2|)$.

Figure 5. shows the resulting optimal policy for this case. It is not much different than the optimal policy above, implying that the above method effectively already accounts for queue size.

Figures 6. and 7. show the value function under the optimal policy. It shows that the value for empty queues is much greater than that for queues which are near full, which makes good intuitive sense.
Figure 5: The policy is not much different.

Figure 6: The value function from above.
Figure 7: The value function from a different angle.

IV. Summary
This project helped to solidify our understanding of the concept and implementation of policy iteration. Even with the relatively simple environment presented in this project, it was clear that defining a model to represent the rewards and transition probabilities for all states can easily become an overwhelming undertaking. Still, we managed to achieve a simplistic model of the environment and used it to find the optimal policy for serving these queues.

Appendix
The table below shows the transition probability and reward matrices for the next state given the current state provided in the title (88 is the state in which both queues are full, etc.). The first dimension is the matrix for the action of pulling a packet from queue 1, while the second dimension is for the action of popping from queue 2. The 3x3 matrices represent the next state such that if $i = |Queue1|$ and $j = |Queue2|$, then possible next states are:

$$
\begin{bmatrix}
  i - 1, j - 1 & i - 1, j & i - 1, j + 1 \\
  i, j - 1 & i, j & i, j + 1 \\
  i + 1, j - 1 & i + 1, j & i + 1, j + 1
\end{bmatrix}
$$

Note that given a certain initial state, a maximum of only four next states are possible, and in some cases, only two next states are possible. Thus, these impossible states have a value of 0.
**Matlab Code for Part 1**

```matlab
queue_size = 9;
theta = .01;
gamma = .9;

% init V(S) and pi(s,a)
V = zeros( queue_size, queue_size );
% pi( i, j ) is the likelihood of choosing q1 in state i,j
% 1 - pi( i, j ) is the likelihood of choosing q2
pi = repmat(.5, queue_size, queue_size);
delta = 1000;

policy_stable = 0;
while (policy_stable == 0)

    % Estimate the Value for the Policy
    while delta > theta
        delta = 0;

        % i is value of queue 1 and j is value of queue 2
        for i=1:queue_size
            for j=1:queue_size

                % Get the correct Probabilities and Rewards for transition
                if i > 1 && i < 9 && j > 1 && j < 9

```
\begin{verbatim}
Prob = Probij;
Rewd = Rewdij;
else if i == 1 && j < 9 && j > 1
    Prob = Prob0j;
    Rewd = Rewd0j;
else if i == 1 && j == 9
    Prob = Prob08;
    Rewd = Rewd08;
else if i == 9 && j == 1
    Prob = Prob80;
    Rewd = Rewd80;
else if i < 9 && i > 1 && j == 1
    Prob = Probi0;
    Rewd = Rewdi0;
else if i < 9 && i > 1 && j == 9
    Prob = Probi8;
    Rewd = Rewdi8;
else if i == 9 && j == 9
    Prob = Prob88;
    Rewd = Rewd88;
end
end
end
end
end

% current value
v = V(i,j); sum = 0;
for a = 1:3
    for b = 1:3
        R = (pi(i,j) * Rewd(a,b,1)) + ((1-pi(i,j)) * Rewd(a,b,2));
        P = (pi(i,j) * Prob(a,b,1)) + ((1-pi(i,j)) * Prob(a,b,2));
        Vns = 0;
        if ( P ~= 0 )
            Vns = V(i + (a-2), j + (b-2));
        end
        sum = sum + P*(R + gamma*Vns);
    end
end
V(i,j) = sum;
delta = max( delta, abs( v - V(i,j) ) );
\end{verbatim}
surf(V);
hold on;
end

policy_update
policy_stable = 1;
for i=1:queue_size
    for j=1:queue_size
        % Get the correct Probabilities and Rewards for transition
        if i > 1 && i < 9 && j > 1 && j < 9
            Prob = Probiij;
            Rewd = Rewdiij;
        elseif i == 1 && j < 9 && j > 1
            Prob = Prob0j;
            Rewd = Rewd0j;
        elseif i == 1 && j == 9
            Prob = Prob08;
            Rewd = Rewd08;
        elseif i == 9 && j == 1
            Prob = Prob80;
            Rewd = Rewd80;
        elseif i < 9 && i > 1 && j == 1
            Prob = Probi0;
            Rewd = Rewdi0;
        elseif i < 9 && i > 1 && j == 9
            Prob = Prob18;
            Rewd = Rewd18;
        elseif i == 1 && j == 1
            Prob = Prob00;
            Rewd = Rewd00;
        elseif i == 9 && j > 1 && j < 9
            Prob = Prob8j;
            Rewd = Rewd8j;
        elseif i == 9 && j == 9
            Prob = Prob88;
            Rewd = Rewd88;
        end
    end
end
end
end
end
oldpi = pi(i,j);
sum1 = 0; sum2 = 0;
for a = 1:3
    for b = 1:3
        Vns = 0;
        if( Prob(a,b,1) ~= 0 )
            Vns = V(i + (a-2), j + (b-2) );
        end
    end
end
sum1 = sum1 + Prob(a,b,1) * (Rewd(a,b,1) + gamma*Vns);
Vns = 0;
if ( Prob(a,b,2) ~= 0 )
    Vns = V(i + (a-2), j + (b-2) );
end
sum2 = sum2 + Prob(a,b,2) * (Rewd(a,b,2) + gamma*Vns);
end
end

if( sum1 > sum2 )
    pi(i,j) = 1;
else
    pi(i,j) = 0;
end

if( oldpi ~= pi(i,j) )
    policy_stable = 0;
end
end
end
figure;
surf( pi );

Matlab Code for Part 2
In part 2, the code for calculating the reward in the value function and policy were changed thus, only the following lines are different:

<< R = (pi(i,j) * Rewd(a,b,1)) + ((1-pi(i,j)) * Rewd(a,b,2));
>> R = (pi(i,j) * Rewd(a,b,1)) + ((1-pi(i,j)) * Rewd(a,b,2)); R = R - disc_fac * (i+j);

<< sum1 = sum1 + Prob(a,b,1) * (Rewd(a,b,1) + gamma*Vns);
>> R = Rewd(a,b,1);
>> R = R - disc_fac * (i+j);
>> sum1 = sum1 + Prob(a,b,1) * (R + gamma*Vns);

<< sum2 = sum2 + Prob(a,b,2) * (Rewd(a,b,2) + gamma*Vns);
>> R = Rewd(a,b,2);
>> R = R - disc_fac * (i+j);
>> sum2 = sum2 + Prob(a,b,2) * (R + gamma*Vns);