ECSE 526 - Assignment #2

Emil Janulewicz

February 25, 2015

1 Elements of the State Space

In our report analysis, we use the variables N, M, E1, E2, E3, W,and P as described in the assignment. All new variables will be explained.

At the start of each hour, a particular state can be described by the following:

1. The number of open operating rooms $O_M \in [0, M]$
2. The number of people in the waiting room $O_N \in [0, N]$
3. The number of new patients that have arrived $O_P \in [x|f(x) >= 0.001]$, where O_P is a random variable characterized by the probability mass function $f(x)$ (In the case of the continuous Gaussian, the discretized version will in fact have a higher mean than the original since negative values contribute probability mass to the positive values).

In fact, we can combine O_N and O_P into one variable $O_T = O_M + O_P$, so that now $O_T \in [0, N + \max(O_P)]$. Therefore, we have a total of $|O_M| \times |O_T|$ states described with only two variables.

We can show why O_M and O_T is enough to describe a particular state by introducing the following notation:

- $O_A = \min(O_T, O_M)$: the number of patients accepted to the operating rooms.
- $O_N = \min(N, \max(O_T - O_A, 0))$: the number of patients in the waiting room.
- $O_R = \max(O_T - O_A - O_N, 0)$: the number of patients rejected.

For each state, the possible actions available are to either close a number of rooms, open a number of rooms, or do nothing at all. Let A_O and A_C be the maximum number of rooms we can open and close respectively. We then have, $A_O = M - O_M$, and $A_C = O_M$

It is important to note here that our decision to either open or close rooms is taken after we have accepted patients into the operating rooms. Consider we have 3 open operating rooms where 2 contain patients and the other is empty. If we decide to close 3 rooms, we can close one right away (thus saving E3 dollars), however, the other two will be closed at the start of the next hour once the patients are discharged.

For a particular state $s = (O_M, O_T)$ the number of possible actions $|A_s|$ is simply $|A_s| = A_O + A_C + 1 = M + 1$.

The entire set of actions for state s is $A_s = \{-O_M, -O_M + 1, ..., 0, ..., M - O_M - 1, M - O_M\}$, where a negative value corresponds to closing, 0 implies no action, and positive values imply opening.

2 Reward/Utility Function and Transition Model

2.1 Reward Function

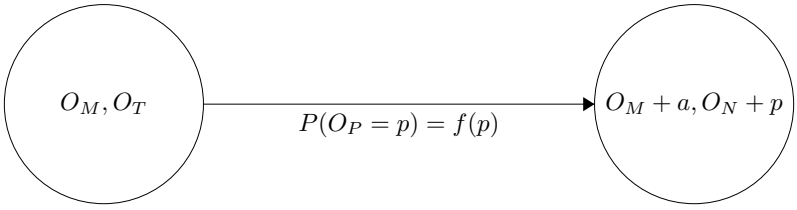
The reward function depends on the current state and the choice of action. For a particular state $s = \{O_M, O_T\}$ and action $a \in A_s$, we have the following reward function:

$$\begin{aligned} R(s,a) =& O_M E3 + O_N W + O_R P \\ &+\delta\left[a > 0\right]|a|(E1 + E3) \\ &+\delta\left[a < 0\right](|a|(E2) - \min(f, |a|)E3) \end{aligned} \tag{1}$$

where $\delta\left[True\right] = 1$, $\delta\left[False\right] = 0$, and $f = \max(0, O_M - O_T)$ describing the number of empty open rooms. f is required since if the room does not have any patients, we can close it right away and save the cost of E3 (given our action is to close).

2.2 Transition Model

Below we describe the transition model of being in state $s = \{O_M, O_T\}$ and performing an action $a \in A_s$ to end up in state s'.



It is clear that the number of open rooms $O_M + a$ in s' is deterministically evaluated given the initial value O_M and the action a. The randomness is introduced by the number of new patients arriving. Thus the total number of patients in s is equal to the number of patients waiting from s, O_N, plus the number of new patients arriving.

For completeness we can write:

$$P\left(s' = \{O_M + a, O_N + p\} | s = \{O_M, O_T\}, a\right) = f(p) \ \forall \ a \in A_s \tag{2}$$

2.3 State Utility

The following describes our utility function for a particular state s'.

$$U(s) = \min_{a \in A_s} \left(R(s,a) + \gamma \sum_{s'} P(s'|s,a) U(s') \right) \tag{3}$$

Since we are only dealing with positive costs, we wish to minimize our utility instead of maximizing. We also notice that our reward $R(s,a)$ is now a function of an action as well, hence we need to include in our minimization objective.

3 Design Parameters

3.1 Choice of γ

γ is required since this problem is of the type 'infinite horizon', i.e. there are no terminal states and thus will run forever. Setting $\gamma < 1$ effectively introduces a terminal state of cost 0 and can be interpreted as the process reaching a terminal state on average after $\frac{1}{1-\gamma}$ steps. This is because at each state, the probability of entering this pseudo-terminal state is $1-\gamma$. Ideally, γ should be as close as possible to 1 in order to see the long-term effects of the agent's actions. In our assignment, we set $\gamma = 0.999$. The trade-off is between accuracy and run-time complexity. As an example, consider the following parameters: -N 10 -M 5 –E1=20000 –E2=30 –E3=200 -W 5 -P 300 –dist=1 -l 2. As you can see, the cost of opening a room (E1) is quite dominant. Even with $\gamma = 0.9$, the horizon is too short (agent misses long-term effects), and no rooms are opened for all states. With $\gamma = 0.999$, the agent is able to see the long term benefits of opening a room (since E3>P) and thus we have a more sensible policy.

3.2 Convergence

Using equations 17.8 and 17.9 from [1], if $||U_{i+1} - U_i|| < \beta$ then $||U^{\pi_i} - U|| < 2\beta$, where $\beta = \epsilon(1-\gamma)/\gamma$. In essence, if our utility for all states at value iteration $i+1$ does not change by more then β from iteration i, we are guaranteed to stray from the optimal policy by no more then 2β. Let λ be the average number of new patients arriving each hour. For the case of $E3 < P$, we don't want to reject any patients and thus we can argue that our average cost per hour is at least $\lambda E3$. For the case where $E3 > P$, we would ideally like to reject all patients and thus our average cost per hour is at least λP. Putting it

together, our average cost per hour is at least $\lambda \min(E3, P)$. We thus set $2\beta = \lambda \min(E3, P)$ since the utility with the optimal policy will be at least 2β from the next best policy.

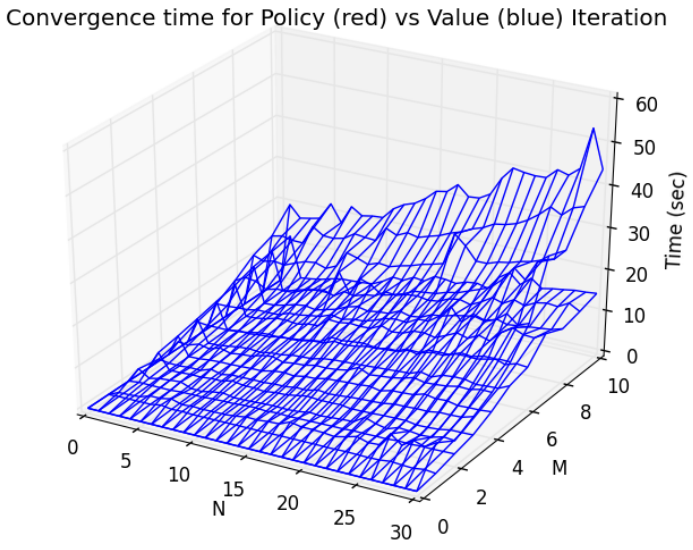


Figure 1: Time until convergence for fixed parameters:–E1=50 –E2=30 –E3=200 -W 5 -P 1000. Poisson distribution with $\lambda = 2$ was used

In Figure 1, the graph suggests that the complexity of policy iteration grows linearly in the number of states, $O(MN)$. For value iteration, the time complexity has a quadratic shape with respect to the state space, thus characterized by $O((MN)^2)$.

4 Supplementary Analysis of Suggested Policy

In the cases where realistic parameters are entered, the policy is quite sensible. For example, consider the following parameters: -N 10 -M 5 –E1=40 –E2=20 –E3=120 -W 5 -P 500 –dist=1 -l 2

The last parameter signifies a poisson distribution with mean 2. Since the expected number of patients arriving is 2, the policy tries to keep around 2 rooms open for the most part. As an example, here are the actions for the case that we have 5 people in the hospital for various open room scenarios (positive action is open, negative action is close): $\{(O_M = 0, O_T = 5), a = 2\}$, $\{(O_M = 2, O_T = 5), a = 0\}$, $\{(O_M = 3, O_T = 5), a = -1\}$, $\{(O_M = 5, O_T = 5), a = -3\}$ However, once the waiting rooms become more full, we wish to avoid the scenario of rejecting patients and thus increase the number of open rooms.

If $E3 >> P$, it makes no sense for the hospital to treat any patients. In this scenario, no rooms are opened and the hospital prefers to reject all patients. Although the parameters are not sensible, the policy is the correct one given the conditions.

As a final example, consider the following parameters: -N 50 -M 5 –E1=40 –E2=30 –E3=150 -W 0 -P 300 –dist=1 -l 2. Notice that we pushed the waiting cost all the way to 0. However, since $E3 < P$, we definitely do not want to reject any patients; it's in our benefit to treat them all. However, when looking at the optimal policy, given no open rooms so far, the first time we see an action to open a room is when $O_T = 24$. Why would we fill up the waiting room to 24 before we start treating patients if we know we will eventually treat them? The reason is because $\gamma < 1$, and thus, the MDP knows that it has a chance of escaping to the pseudo-terminal state and thus not having to treat the poor patients.

Reference

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009, ISBN: 0136042597, 9780136042594.

5 ValueIteration.py

```
import copy
import numpy as np
import sys
import getopt
import math as mt
from random import randint
import time as tm

def usage():
  print "\nUsage function\n"

  print 'python ValueIteration.py [-h] [-d num] [-l num] [-m num] [-s num]
     [-o num] [-c num]\
   [-b num] [-N num] [-M num] [-W num] [-P num]'

  print '-h, --help \n\t invoke the usage information'
  print '-d num, --dist=num\n\t 1 for poisson, 0 for gaussian'
  print '-l num, --lambda=num\n\t Mean for the poisson distribution'
  print '-m num, --mu=num\n\t Mean for the Gaussian distribution'
  print '-s num, --sigma=num\n\t Standard Deviation for the Gaussian
     distribution'
  print '-o num, --E1=num\n\t Cost of opening a room'
  print '-c num, --E2=num\n\t Cost of closing a room'
  print '-b num, --E3=num\n\t Cost of operating a room'
  print '-N num\n\t Number of seats in the waiting room'
  print '-M num\n\t Maximum number of rooms that could be operating'
  print '-W num\n\t Cost of waiting per hour per patient'
  print '-P num\n\t Cost of rejecting a patient'

def state_reward(open_rooms, people):
        cost=open_rooms*E3
        waiting=(people>open_rooms)*min(people-open_rooms,N)
        rejected=people-open_rooms-waiting
        rejected=rejected*(rejected>0)
        cost=cost+(waiting*W)+rejected*P
        #print 'W is ',(waiting/float(N)*W)
        return cost

def action_reward(open_rooms,free_rooms):
        return abs(open_rooms)*((open_rooms>0)*(E1+E3)+(open_rooms<0)*(E2))
            \
        -min(free_rooms,abs(open_rooms))*(open_rooms<0)*E3

def get_pmf(poisson,lamb,mu,sigma,epi):
        value=1
        pmf=[]
        k=0
        sum_n=0
```

```
        before_mean=1;
        while (value>=0.001 or before_mean):
                if (poisson):
                        value=lamb**k*mt.exp(-lamb)/mt.factorial(k)
                else:
                        value=1/(mt.sqrt(2*mt.pi*sigma**2))*mt.exp((-(k-mu)
                            **2)/(2*sigma**2))
                #print value
                #if (value>epi):
                if (value>=0.001):
                        pmf.append([k,value])
                if (value>0.001): before_mean=0;
                k=k+1
                sum_n=0.0
                for x in pmf:
                        sum_n=sum_n+x[1]
        sum_n=0.0
        for x in pmf:
                sum_n=sum_n+x[1]
        for i,item in enumerate(pmf):
                #print i,item
                pmf[i][1]=item[1]/sum_n
        return pmf

def get_poisson_pmf(lamb,epi):
        value=1
        poisson_pmf=[]
        k=0
        sum_n=0
        before_mean=1;
        while (value>=0.001 or before_mean):
                value=lamb**k*mt.exp(-lamb)/mt.factorial(k)
                print value
                #if (value>epi):
                if (value>=0.001):
                        poisson_pmf.append([k,value])
                if (value>0.001): before_mean=0;
                k=k+1
                sum_n=0.0
                for x in poisson_pmf:
                        sum_n=sum_n+x[1]
        sum_n=0.0
        for x in poisson_pmf:
                sum_n=sum_n+x[1]
        for i,item in enumerate(poisson_pmf):
                print i,item
                poisson_pmf[i][1]=item[1]/sum_n
       #poisson_pmf[:]=[x/sum(poisson_pmf) for x in poisson_pmf]

        return poisson_pmf

def get_normal_pmf(mu,sigma,epi):
        normal_pmf=[]
        value=1
        k=0
        before_mean=1;
```

```
        while(value>=0.001 or before_mean):
                value=1/(mt.sqrt(2*mt.pi*sigma**2))*mt.exp((-(k-mu)**2)/(2*
                    sigma**2))
                print 'bottom', 1/(mt.sqrt(2*mt.pi*sigma**2))
                print 'top',(-((k-mu)**2)),'bottom',(2*sigma**2)
                print value
                if (value>0.001): before_mean=0
                if (value>=0.001):
                        normal_pmf.append([k,value])
                k=k+1
        sum_n=0.0
        for x in normal_pmf:
                sum_n=sum_n+x[1]
        for i,item in enumerate(normal_pmf):
                print i,item
                normal_pmf[i][1]=item[1]/sum_n
        #normal_pmf[:,1]=[x[1]/1 for x in normal_pmf]

        return normal_pmf

def main(argv):

        try:
                opts,args=getopt.getopt(argv,"hd:l:m:s:o:c:b:N:M:W:P:",["
                    help","dist=","lambda=","mu=","sigma=","E1=","E2=","E3
                    ="])
        except getopt.GetoptError:
                print 'bad input'
                sys.exit(2)

        for opt,arg in opts:
                if opt in ("-h","--help"):
                        usage()
                        sys.exit()
                elif opt in("-d","--dist"):
                        global use_poisson
                        use_poisson=int(arg)
                elif opt in("-l","--lambda"):
                        global lambda_
                        lambda_=float(arg)
                elif opt in("-m","--mu"):
                        global mu
                        mu=float(arg)
                elif opt in("-s","--sigma"):
                        global sigma
                        sigma=float(arg)
                elif opt in ("-N"):
                        global N
                        N=int(arg)
                elif opt in ("-M"):
                        global M
                        M=int(arg)
                elif opt in ("-o","--E1"):
                        #print 'we got here'
                        global E1
                        E1=float(arg)
```

```
        elif opt in ("-c","--E2"):
                global E2
                E2=float(arg)
        elif opt in ("-b","--E3"):
                global E3
                E3=float(arg)
        elif opt in ("-W"):
                global W
                W=float(arg)
        elif opt in ("-P"):
                global P
                P=float(arg)


global gamma
gamma=0.999



if (use_poisson):
        prob_dist=get_pmf(1,lambda_,3,2,0.001)
        mean=lambda_
        print 'Using Poisson distribution with mean ',lambda_
        print prob_dist
        #sys.exit(2)
else:
        prob_dist=get_pmf(0,0,mu,sigma,0.001)
        t_m=0
        for s in prob_dist:
                t_m=t_m+s[0]*s[1]
        print 'Using Gaussian distribution with original mean',mu
            ,'. \nAfter discretizing it, the new mean is %1.2f' %
            t_m
        mean=t_m


Nm=M+1 #number of possible rooms that can be open (0,1,...,M)

Np=N+prob_dist[-1][0]+1 #number of people in the waiting room + new
     arrivals (0,1,...,N+max(prob_dist))


current_utility=np.zeros([3,Np,Nm]) #store the values of the
    utilities and actions
#current_utility[0,:,:] --> utilities at time step i
#current_utility[1,:,:] --> utilities at time step i+1
#current_utility[2,:,:] --> best possible action so far for each
    state


##VALUE ITERATION CODE

max_error=mean*min(E3,P)/2 #error value used for convergence of
    value iteration
#print max_error
delta=1e10
```

```
iteration=0
value_start=tm.time()
initial_delta=1e10
print 'Performing Value Iteration...'
while (delta>max_error):
        print 'iteration: ',iteration,' Algorithm Completion: ', (
            iteration>40)*(initial_delta-delta)/(initial_delta-
            max_error),'%'
        sys.stdout.write("\033[F")
        #print np.mean(current_utility[0,:,:]/(iteration+1))
        #print 'utility mean %r and STD %r'% (np.mean(
            current_utility[0,:,:]/(iteration+1)),np.std(
            current_utility[0,:,:]/(iteration+1)))
        iteration=iteration+1
        if (iteration==40): initial_delta=delta
        for i in xrange(Np):
                for j in xrange(Nm):
                        #print 'now on state: ',i,j
                        delta=0
                        min_val=1e100
                        wait_room=(i>j)*min(i-j,N)
                        #max_close=max(j-i,0)
                        max_close=j
                        free_rooms=max(0,j-i)
                        for a in xrange(-max_close,M-j+1,1):
                                cost=0.0
                                for state in prob_dist:
                                        #print state
                                        prob=state[1]
                                        cost=cost+prob*
                                            current_utility[0,
                                            wait_room+state[0],j+a]
                                cost=cost*gamma

                                cost=cost+state_reward(j,i)+
                                    action_reward(a,free_rooms)
                                #if (i==4 and j==4): print 'action
                                    ',a, 'value',cost
                                if (cost<min_val):

                                        min_val=cost
                                        min_action=a
                                        current_utility[2,i,j]=a

                        current_utility[1,i,j]=min_val ##here it is
                             bitches?

                        difference=(abs(current_utility[1,i,j]-
                            current_utility[0,i,j]))
                        #print 'difference is: ', difference
                        if (difference>delta):
                                delta=difference

        current_utility[0,:,:]=current_utility[1,:,:]
print 'iteration: ',iteration,' Value Iteration Algorithm Complete
    !: '
```

```
value_time=tm.time()-value_start
value_iteration_policy=current_utility[2,:,:]
#uncomment below to print out the final optimal policy (actions for
    each state)
# for i in xrange(Np):
#       for j in xrange(Nm):
#               print('For rooms %r and people %r, action is %1.0f
   '%(j,i,current_utility[2,i,j]))
# #print the average utility and STD
# print current_utility[0,:,:]
# print 'Average utility is %r with and STD of %r' % (np.mean(
   current_utility[0,:,:]*(1-gamma)),np.std(current_utility
   [0,:,:]*(1-gamma)))



#POLICY ITERATION CODE

##Initialize all utilities and actions
current_utility=np.zeros([3,Np,Nm])

##Uncomment below if you want to start with a random policy
# for i in xrange(Np):
#       for j in xrange(Nm):
#               current_utility[2,i,j]=M-j
iteration=0;
converged=False #converged will tell you if our policy has
   converged or not
k=50
policy_start=tm.time()
print 'Performing Policy Iteration....'
while (not converged):
        if (iteration>25): break
        iteration=iteration+1
        delta=1e10
        iter_p=0
        #First step is to perform a policy evaluation given our
            current policy
        for i in xrange(k):
        #while (delta>4*max_error/float(iteration)):
                iter_p=iter_p+1
                delta=0
                max_diff=0
                for j in xrange(Np):
                        for q in xrange(Nm):

                                wait_room=(j>q)*min(j-q,N)
                                cost=0.0
                                #for state in xrange(len(
                                    poisson_pmf)):
                                a=current_utility[2,j,q]
                                for state in prob_dist:
                                        prob=state[1]
```

```
                                        cost=cost+prob*
                                            current_utility[0,
                                            wait_room+state[0],q+a]
                                    free_rooms=max(0,q-j)
                                    cost=cost*gamma
                                    cost=cost+state_reward(q,j)+
                                        action_reward(a,free_rooms)
                                    current_utility[1,j,q]=cost
                                    diff=abs(current_utility[1,j,q]-
                                        current_utility[0,j,q])
                                    if (diff>delta):

                                        delta=diff

                    current_utility[0,:,:]=current_utility[1,:,:]
            ##now we check for the new updated best policy
            converged=True ##the moment we have an update, set it to
                false
            print 'Policy Update # ',iteration
            sys.stdout.write("\033[F")
            for i in xrange(Np):
                    for j in xrange(Nm):
                            min_val=1e100
                            wait_room=(i>j)*min(i-j,N)
                            max_close=j
                            free_rooms=max(0,j-i)

                            for a in xrange(-max_close,M-j+1,1):
                                    cost=0.0
                                    #for state in xrange(len(
                                        poisson_pmf)):
                                    for state in prob_dist:
                                            prob=state[1]

                                            cost=cost+prob*
                                                current_utility[0,
                                                wait_room+state[0],j+a]
                                    cost=cost+state_reward(j,i)+
                                        action_reward(a,free_rooms)

                                    if (cost<min_val):
                                            #print cost
                                            min_val=cost
                                            min_action=a
                                            #current_utility[2,i,j]=a
                            if (min_action!=current_utility[2,i,j]):
                                    current_utility[2,i,j]=min_action
                                    converged=False

#uncoment below if you wish to print the optimal policy
# for i in xrange(Np):
#         for j in xrange(Nm):
#                 print('For rooms %r and people %r, action is %1.0f
    '%(j,i,current_utility[2,i,j]))
policy_iteration_policy=current_utility[2,:,:]
policy_time=tm.time()-policy_start
```

```
        ##check for equality
        print '                                    '
        print 'Policy Iteration Complete!!!!'
        print 'Elapsed time for value iteration: %3.2f' % value_time,'
            seconds'
        print 'Elapsed time for policy iteration: %3.2f' % policy_time,'
            seconds'
        # print 'Value and Policy iteration converged to the same optimal
            policy: ',np.array_equal(value_iteration_policy,
            policy_iteration_policy)


        #Code to allow for querying a particular state and returning the
            action
        while (True):
                which_iter=int(raw_input("Would you like to choose from
                    Value(0) or Policy(1) Iteration: ((-1) to quit)"))
                if (which_iter==-1):break
                Nm_input=int(raw_input("Enter the number of open rooms (0,%
                    r): ('-1' to quit)" %M))
                Np_input=int(raw_input("Enter the number of total people
                    (0,%r): "%(Np-1)))
                #print('For rooms %r and people %r, action is %2.2f'%(
                    Nm_input,Np_input,current_utility[2,Np_input,Nm_input]))
                if (which_iter):
                        action_r=int(policy_iteration_policy[Np_input,
                            Nm_input])
                else:
                        action_r=int(value_iteration_policy[Np_input,
                            Nm_input])

                #print action_r
                if (action_r>0):
                        action_s="to open %r rooms" % action_r
                elif(action_r<0):
                        action_s="to close %r rooms" % abs(action_r)
                else:
                        action_s="to stay put"
                print 'Best action is ',action_s

        # for i in xrange(Np):
        #       for j in xrange(Nm):
        #               if (value_iteration_policy[i,j]!=
            policy_iteration_policy[i,j]):
        #                       print 'for %r and %r, vlaue gives %r and
            policy gives %r' % (i,j,value_iteration_policy[i,j],
            policy_iteration_policy[i,j])


if __name__=="__main__":
        main(sys.argv[1:])
```