MATH4406 - Assignment 4

Sean Watson - 42613512

1 Infinite Horizon Discounted MDP

1.1

A potential real-life application of this MDP could be an animal conservation problem where the states represent the population level of a certain species where we wish to retain numbers at manageable level, represented by state 0. A negative state would mean fewer/potentially endangered animals with -5 effectively being extinction. A positive state would mean greater numbers of the species than we want, effectively the species has become a pest, with state 5 being something like the carrying population of the region. One of our actions is to cull the population, more likely to reduce numbers but more expensive if the population is larger. There is potential for an increase in population when we try to cull though as we may miss pregnant animals or the hunt may be unsuccessful. On the other hand we can choose to add new animals into the population from outside the system. Obviously this is more likely to increase the population, but there is also the possibility that the new animals carry some kind of virus with them causing more of the population to die off. The costs for bringing new animals in may decrease as the population in the system grows because the species is less likely to be endangered if it is flourishing here. The boundary states are simply the extremes of these actions with animals costing exorbitant amounts when there are none in the system because of the issues with establishing a new population. Conversely a huge population is harder to get back under control so culling is more difficult and thus more expensive.

1.2

Depending on the level of λ the policy would vary since taking steps closer to the centre is costly it is likely that there would be a threshold policy such that after being a certain distance from 0 we take actions to move closer once more. If λ is quite low then we are really only looking at short term costs, which means the actions are more likely to be just to avoid the boundary but otherwise try to increase value. Conversely, if λ were quite high then we would have a long term goal to minimise costs since the cost on the boundary is so high and taking risky moves away from the centre is too dangerous over a long time span. So the optimal policy for $\lambda \approx 0$ would be to move away from 0 and increase short term gains. If $\lambda \approx 1$ we would want to always move towards 0 in the hope of never hitting the boundary states. Regardless of the value of λ the optimal policy is always symmetric about 0 due to the symmetry in the problem itself.

1.3 3-7

The Python code used for these questions can all be found in the appendix.

The following table displays the optimal policies for each of the 99 different values of λ. There are ranges over which different values of λ share the same optimal policy and so these have been given instead of a bulk listing for the sake of conciseness. Note that in the case of the 0 state either action is optimal due to the symmetry of the problem.

	State										
Range of λ	-5	-4	-3	-2	-1	0	1	2	3	4	5
0.96-0.99	3	1	1	1	1	-1/1	-1	-1	-1	-1	-3
0.93-0.95	3	1	1	1	-1	-1/1	1	-1	-1	-1	-3
0.86-0.92	3	1	1	-1	-1	-1/1	1	1	-1	-1	-3
0.64-0.85	3	1	-1	-1	-1	-1/1	1	1	1	-1	-3
0.01-0.63	3	-1	-1	-1	-1	-1/1	1	1	1	1	-3

As expected these policies are all threshold policies as after moving a certain distance from the centre we always take actions that are more likely to take us back to the centre. Further as λ increases the decision rules change from being based on maximising short term gains to minimising long term losses.

A Brute Force Evaluation Code

```
from __future__ import division
import itertools
from numpy import matrix
from numpy import linalg

P = {};
S = [-5,-4,-3,-2,-1,0,1,2,3,4,5]
lamb = 0.95
actions = {}
for s in S:
    if s == -5:
        P[s,3,-4] = .5
        P[s,3,-5] = .5
        actions[s] = [3]
    elif s == 5:
        P[s,-3,4] = .5
        P[s,-3,5] = .5
        actions[s] = [-3]
    else:
        P[s,1,s-1] = .25
        P[s,1,s+1] = .75
        P[s,-1,s-1] = .75
        P[s,-1,s+1] = .25

        actions[s] = [1,-1]
r = {}
for s in S:
    r[s] = [a*s for a in actions[s]]

x = list(itertools.product(S[1:-1], [-1,1]))
y = []
for i in range(len(x))[::2]:
    y.append([x[i], x[i+1]])
pol = list(itertools.product(*y))
pols = {}
for p in range(len(pol)):
    for q in range(len(pol[0])):
        pols[p,pol[p][q][0]] = pol[p][q][1]
```

```
def printSol(p):
    sol = [3]
    for s in S[1:-1]:
        sol.append(pols[p,s])
    sol += [-3]
    return sol

def brute(lamb):
    V = {}; VS = {}
    for p in range(len(pol)):
        # col. vector for rd
        rList = []
        for s in S:
            if s == 5:
                rList.append([-15])
            elif s == -5:
                rList.append([-15])
            else:
                rList.append([pols[p,s]*s])

        rd = matrix(rList)

        # make transition matrix
        pList = []
        for s in S:
            if s == 5:
                pList.append([0,0,0,0,0,0,0,0,0,0.5,0.5])
            elif s == -5:
                pList.append([0.5,0.5,0,0,0,0,0,0,0,0,0])
            else:
                pro = []
                for ss in S:
                if (s,pols[p,s],ss) in P:
                    pro.append(P[s,pols[p,s],ss])
                else:
                    pro.append(0)
            pList.append(pro)
        pd = matrix(pList)

        # identity matrix
        I = matrix([[0 if s != ss else 1 for ss in S] for s in S])
```

```
                #invert matrices
                inv = (I - lamb*pd).I

                # multiply by rewards
                V[p] = inv*rd

                VS[p] = sum(V[p].tolist()[ss][0] for ss in range(len(S)))

            pMax = max(VS, key=VS.get) # optimal policy
            # due to symmetry just set state 0's action to 1
            final = printSol(pMax)
            final[5] = 1
            tuple(final)

         # loop over all the values of lambda
        res =
        for lamb in [x/100 for x in range(1,100)]:
            ans = brute(lamb)
            if ans in res:
                res[ans] = res[ans] + [lamb]
            else:
                res[ans] = [lamb]
```

B Value Iteration Algorithm

This code is written to be included in the same script as the previous Brute Force Evaluation, i.e. the data manipulations at the start are still required. To loop over the different λ values brute(lamb) is replaced with valueIter(lamb).

```
# initialise
V = {}
for s in S:
    V[s,0] = max(r[s])

def valueIter(lamb):
    # iterate
    n = 1; complete = False
    while complete == False:
        diff = []; dec = []
        for s in S:
            L = [[] for a in range(len(r[s]))]
```

```
                for a in range(len(r[s])):
                        L[a] = r[s][a] + lamb*sum(P[s,actions[s][a],j]*V[j,n-1]
                                                        for j in S if (s,actions[s][a],j) in P)
                V[s,n]= max(L)
                diff.append(abs(V[s,n]-V[s,n-1]))
                dec.append(actions[s][L.index(max(L))])
        if sum(diff) <= (1-lamb)/(2*lamb)*eps: #14,600 steps needed
                complete = True
                print dec
        else:
                n += 1
```

C Policy Iteration Algorithm

This code is written to be included in the same script as the previous Brute Force Evaluation, i.e. the data manipulations at the start are still required. To loop over the different λ values brute(lamb) is replaced with policyIter(lamb).

```
def evaluate(p, lamb):
        rList = []
        for s in S:
                if s == 5:
                        rList.append([-15])
                elif s == -5:
                        rList.append([-15])
                else:
                        rList.append([pols[p,s]*s])

        rd = matrix(rList)

        # make transition matrix
        pList = []
        for s in S:
                if s == 5:
                        pList.append([0,0,0,0,0,0,0,0,0,0.5,0.5])
                elif s == -5:
                        pList.append([0.5,0.5,0,0,0,0,0,0,0,0,0])
                else:
                        pro = []
                        for ss in S:
                                if (s,pols[p,s],ss) in P:
```

```
                    pro.append(P[s,pols[p,s],ss])
                else:
                    pro.append(0)
            pList.append(pro)
    pd = matrix(pList)
    I = matrix([[0 if s!= ss else 1 for ss in S] for s in S])

    return linalg.solve((I-lamb*pd), rd).tolist()

def policyIter(lamb):
    # 1. Initialise
    n = 1
    new = pol[0]
    old = 0
    complete = False
    # 2. Evaluate policy
    while complete == False:
        Vn = evaluate(pol.index(new), lamb)
        Vn = [Vn[i][0] for i in range(len(S))]
        V = {}
        for i in range(len(S)):
            V[S[i]] = Vn[i]
        # 3. Policy improvement
        old = new
        new = []
        acts = {}
        for s in S[1:-1]: # the boundary states don't matter since they're always the same
            act = []
            for a in range(len(actions[s])):
                act.append(r[s][a] + lamb*sum(P[s,actions[s][a],j]*V[j]
                                for j in S if (s,actions[s][a],j) in P))
            d1 = act.index(max(act))
            d1 = actions[s][d1]
            # ensure that if the previous d is amongst the current argmax then it is chosen
            if act[0] == act[1]:
                ol = printSol(pol.index(old))
                d1 = ol[S.index(s)+1]
            # if in state 0 choose 1 everytime, due to symmetry this doesn't change anything but avoids cycles
            if s == 0:
                new.append([s,1])
            else:
                new.append([s,d1])

new = tuple(tuple(new[i]) for i in range(len(new)))
# 4. if the new policy is equal to the old: stop
if new == old:
        complete = True
        return tuple(printSol(pol.index(new)))
```