Assignment 2
CS 747

Saavi Yadav
170020003

TASK 1

1) Value Iteration

For value iteration, value for all states which are not terminal is evaluated. Values and policy are first initialised with zeros. For all possible actions and transitions from a given state, the action for which value is highest is assigned to newValue and the corresponding action is assigned to the policy for the given state. The iterations are continued till the difference between the previous values and current values are lower than tolerance.

```
def valueIteration (numStates, numActions, start, end, rewards, probability,
transitions, mdtype, discount):
   oldValue = np.zeros(numStates)
   actions = np.zeros(numStates)
   tol = 1e-10
   maxIter = 100000
   t = 0
   newValue = np.zeros(numStates)
   while(t<maxIter):
       oldValue = np.copy(newValue)
       for i in transitions:
           if i not in end:
               maxScore = -float('inf')
               bestAction = -1
               for j in transitions[i]:
                   transValue = 0
                   for trans in transitions[i][j]:
                       transValue += trans[2]*(trans[1] + discount* oldValue
[trans[0]])
                   if transValue > maxScore:
                       maxScore = transValue
                       bestAction = j
               actions[i] = bestAction
               newValue[i] = maxScore
       if np.sum(abs(oldValue-newValue))<tol: break
       oldValue = np.copy(newValue)
       t = t+1
   return oldValue, actions
```

2) Linear Programming

Pulp is used for solving the linear programming problem. No of equations are n*k where n = no of states and k = no of actions.

For each state and action, value functions are evaluated for each action and assigned as a constraint. For end states, value functions are assigned as 0. Policy value for a given state is assigned as action which gives the highest value function.

```
def lp(numStates,numActions,end,rewards,probability,transitions,discount):
    problem = LpProblem("MDP", LpMinimize)
    values = LpVariable.dict("valueFunction", range(numStates))
    LpSolverDefault.msg = 0
    problem += lpSum([values[i] for i in range(numStates)])
    V = np.zeros((numStates, 1), dtype = LpVariable)
    for i in range(numStates):
        V[i][0] = values[i]
    for i in transitions:
        if i not in end:
            for j in transitions[i]:
                value = 0
                for trans in transitions[i][j]:
                    value += trans[2]*(trans[1]+discount*values[trans[0]])
                problem += (values[i] >= value)    for i in end :
        problem += (values[i] == 0)    problem.solve()
    newValues = np.zeros(numStates)
    for i in range(numStates):
        newValues[i] = values[i].varValue    actions = np.zeros(numStates)
    for i in transitions:
        if i not in end:
            maxValue = -float('inf')
            bestAction = 0
            for j in transitions[i]:
                value = 0
                for trans in transitions[i][j]:
                    value += trans[2]*(trans[1]+discount*newValues[trans[0]])
                if value > maxValue:
                    maxValue = value
                    bestAction = j
            actions[i] = bestAction
    return newValues, actions
```

3) Howard Iteration

First the policy is randomly assigned and value function is assigned as zeros and then value function for all states is calculated. Now the old value functions

are assigned as the values obtained from the previous iteration. This is continued till the difference between new and old values is less than tolerance. Now, using the value function obtained we calculate value function for all possible states and actions. Action for which the value function gives the highest value is the optimal policy. For terminating, we check if the optimal policy is the policy for which the Value function was obtained. If that is not the case, then the policy is updated and a new value function is calculated and again optimal policy is checked.

```
def evaluate(oldValues, numStates,numActions,start,end, transitions,discount,
policy):
   newValues = np.copy(oldValues)
   tol = 1e-15
   while True:
       oldValues = np.copy(newValues)
       for i in transitions:
           if i not in end:
               score = 0
               for trans in transitions[i][policy[i]]:
                   score += trans[2]*(trans[1]+discount*newValues[trans[0]])
               newValues[i] = score
       if np.sum(abs(oldValues-newValues))<tol:
           break
   return newValues

def howardIteration(numStates,numActions,start,end,rewards,probability,
transitions, mdtype, discount):
   policy = np.ones(numStates)
   oldPolicy = np.zeros(numStates)
   policy = np.zeros(numStates)
   for i in transitions:
       if i not in end:
           for j in transitions[i]:
               policy[i] = j
               break
   oldValues = np.zeros(numStates)
   newValues = np.zeros(numStates)
   while(True):
       oldValues = np.copy(newValues)
       oldPolicy = np.copy(policy)
       newValues = evaluate(oldValues,numStates,numActions,start,end,
transitions, discount,policy)
       oldValues = np.copy(newValues)
```

```
        breakLoop = True
        for i in transitions:
            if i not in end:
                maxScore = -float('inf')
                bestAction = -1
                for j in transitions[i]:
                    transValue = 0
                    for trans in transitions[i][j]:
                        transValue += trans[2]*(trans[1]+ discount*newValues
[trans[0]])
                    if transValue > maxScore:
                        maxScore = transValue
                        bestAction = j
                policy[i] = bestAction
                if (oldPolicy[i] != policy[i]):
                    breakLoop = False
                    break
        if breakLoop:
            break
    return oldValues, policy
```

TASK 2

Value Iteration gives the best solution with minimum time.

Encoder:-

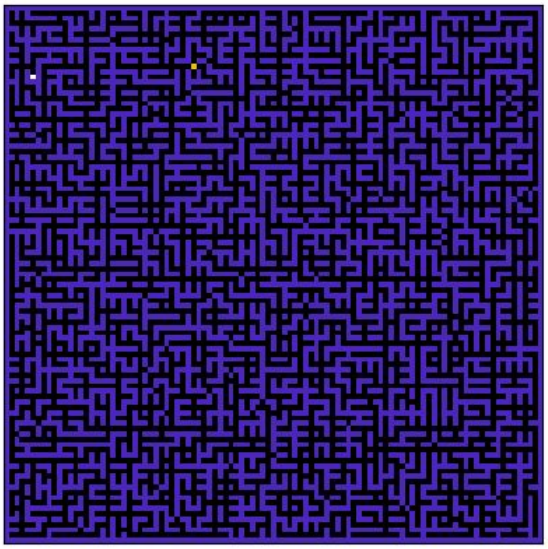
Start and end step is recognised by the cells with 2 and 3. States are all the cells in the maze grid. Actions are the direction in which we can move from a particular cell. Actions are = {0: 'N',1: 'S',2: 'E',3: 'W'}. Transitions from a state are recognised as the directions in which we can move from a particular cell/state. Encoder files the given grid with start state, end state, all possible transitions and discount.

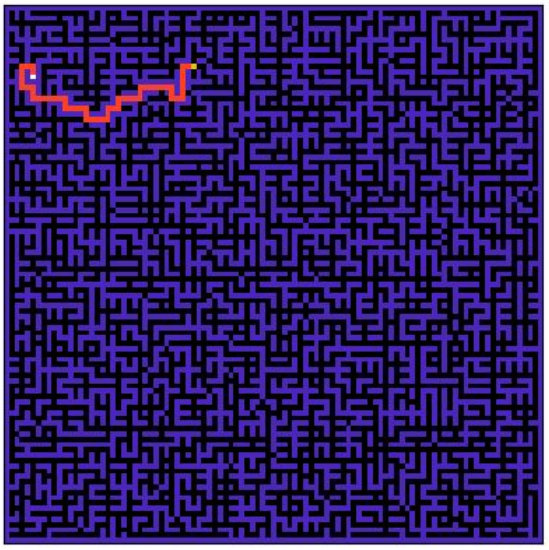
Decoder:-

Decoder starts from the start state given in the mdpfile and using the policy formulated from the planner function with algorithm as value iteration we move in a particular direction. This continues till we reach the end state.

Grid and Solution visualization:-

1) Grid 100 and solution





2) Grid 90 and solution