2018

Semester 2

Assignment One

Connor McDowall
cmcd398
530913386

April 3, 2019

Listings

Contents

List of Figures

List of Tables

1 Joint Distributions

1.1 Event Definition

$$
\begin{aligned}
&x \epsilon X \\
&y \epsilon Y \\
&X = DiceOne \\
&Y = DiceTwo \\
&A : x + y = 7 \\
&B : x = 3 \\
&C : y = 4 \\
&X \epsilon 1, 2, 3, 4, 5, 6 \\
&Y \epsilon 1, 2, 3, 4, 5, 6
\end{aligned}
$$

1.2 Pairwise Independence

Event A, B and C are pairwise independent as any combination of two events is possible.
Both dice two an equal four and the sum of both dice can equal 7
Both dice one can equal three and the sum of dice 1 and 2 can still equal seven
Both dice two can equal four and dice three equal one
Therefore, the pairwise independence is shown through the following

$$
\begin{aligned}
P(A) &= \frac{1}{6} \\
P(B) &= \frac{1}{6} \\
P(C) &= \frac{1}{6}
\end{aligned}
$$

$$
\begin{aligned}
P(A \cap B) &= P(A|B) \times P(B) = P(A)P(B) = \frac{1}{6} \times \frac{1}{6} = \frac{1}{36} \\
P(A \cap C) &= P(A|C) \times P(C) = P(A)P(C) = \frac{1}{6} \times \frac{1}{6} = \frac{1}{36} \\
P(B \cap C) &= P(B|C) \times P(C) = P(B)P(C) = \frac{1}{6} \times \frac{1}{6} = \frac{1}{36}
\end{aligned}
$$

1.3 Mutual Independence

Events A, B and C are not mutually independent as satisfies pairwise independence (above) but not the following condition

$$
P(A \cap B \cap C) = P((A \cap B)|C) \times P(C) = \frac{1}{6} \times \frac{1}{6} \neq P(A) \times P(B) \times P(C) = \frac{1}{6} \times \frac{1}{6} \times \frac{1}{6}
$$

2 Question Two

2.1 Transition Probabilities Matrix

$$
\begin{aligned}
U &= Unfinished \\
P &= PoorCondition \\
G &= GoodCondition \\
S &= Scrapped \\
A &= AverageCondition
\end{aligned}
$$

	U	P	G	A	S
U	0	0.2	0.4	0.3	0.1
P	0	0.2	0.4	0.3	0.1
G	0	0	1	0	0
A	0	0	0	1	0
S	0	0	0	0	1

2.2 Limiting Distribution

You need the probabilities in reaching the absorbing states for the limiting distribution. We are not concerned with U or P.

$$
\begin{aligned}
Pr(G) &= P_{UG} + P_{UG} \times P_{UP} \times P_{PP} \\
&= 0.4 + 0.4 \times 0.2 \times \sum_{n=0}^{\infty} 0.2^n \\
&= 0.4 + 0.4 \times 0.2 \times \frac{1}{1 - 0.2} \\
&= 0.5 \\
Pr(s) &= P_{Us} + P_{US} \times P_{UP} \times P_{PP} \\
&= 0.1 + 0.1 \times 0.2 \times \sum_{n=0}^{\infty} 0.2^n \\
&= 0.1 + 0.1 \times 0.2 \times \frac{1}{1 - 0.2} \\
&= 0.125 \\
Pr(A) &= P_{UA} + P_{UA} \times P_{UP} \times P_{PP} \\
&= 0.3 + 0.3 \times 0.2 \times \sum_{n=0}^{\infty} 0.2^n \\
&= 0.3 + 0.3 \times 0.2 \times \frac{1}{1 - 0.2} \\
&= 0.375
\end{aligned}
$$

Therefore, the limiting distribution for unfinished goods is

U	P	G	S	A
0	0	0.5	0.125	0.375

2.3 Expected Profit

You need to calculate the mean hitting times from going from unfinished to a finished states (G,S or A). Form simultaneous equations to solve the problem.

$$M_{ij} = 1 + \sum_{k=1}^{n} P_{ik} \times M_{kj} \text{ where } i \neq j$$
$$M_{PG} = 1 + P_{PP}M_{PG} + P_{PG}M_{GG} + P_{PA}M_{AG} + P_{PS}M_{SG} + P_{PU}M_{UG}$$
$$M_{PS} = 1 + P_{PP}M_{PS} + P_{PG}M_{GS} + P_{PA}M_{AS} + P_{PS}M_{SS} + P_{PU}M_{US}$$
$$M_{PA} = 1 + P_{PP}M_{PA} + P_{PG}M_{GA} + P_{PA}M_{AA} + P_{PS}M_{SA} + P_{PU}M_{UA}$$

You can simplify the equations using the transition probability matrix

$$M_{PG} = 1 + P_{PP}M_{PG}$$
$$M_{PS} = 1 + P_{PP}M_{PS}$$
$$M_{PA} = 1 + P_{PP}M_{PA}$$

Rearranging these equations, you get

$$M_{PG} = 1.25$$
$$M_{PS} = 1.25$$
$$M_{PA} = 1.25$$

Use the same process for the other equations

$$M_{UG} = 1 + P_{UP}M_{PG} + P_{UG}M_{GG} + P_{UA}M_{AG} + P_{US}M_{SG} + P_{UU}M_{UG}$$
$$M_{US} = 1 + P_{UP}M_{PS} + P_{UG}M_{GS} + P_{UA}M_{AS} + P_{US}M_{SS} + P_{UU}M_{US}$$
$$M_{UA} = 1 + P_{UP}M_{PA} + P_{UG}M_{GA} + P_{UA}M_{AA} + P_{US}M_{SA} + P_{UU}M_{UA}$$

Eliminate terms and substitute equations from above

$$\begin{aligned} M_{UG} &= 1 + P_{UP}M_{PG} \\ &= 1 + 0.2 * 1.25 \\ &= 1.25 \\ M_{US} &= 1 + P_{UP}M_{PS} \\ &= 1 + 0.2 * 1.25 \\ &= 1.25 \\ M_{UA} &= 1 + P_{UP}M_{PA} \\ &= 1 + 0.2 * 1.25 \\ &= 1.25 \end{aligned}$$

Since the mean hitting time 1.25 for all possible routes through the system

$$\begin{aligned} \text{Expected Profit} &= 50 \times 0.5 + 40 \times 0.375 - 20 - 10 \times 1.25 \\ &= \$7.50 \end{aligned}$$

2.4 Updated Transition Probilities Matrix

	U	P_1	P_2	G	A	S
U	0	0.2	0	0.4	0.3	0.1
P_1	0	0	0.2	0.4	0.3	0.1
P_2	0	0	0	0.4	0.3	0.3
G	0	0	0	1	0	0
A	0	0	0	0	1	0
S	0	0	0	0	0	1

3 Question Three

3.1 Message

could you order a peri peri chicken sandwich for me and fries for alice
we want you to go to the closest restaurant
are you driving there in your car
where are you
how far are you from here
we are starving
what time will you show up here

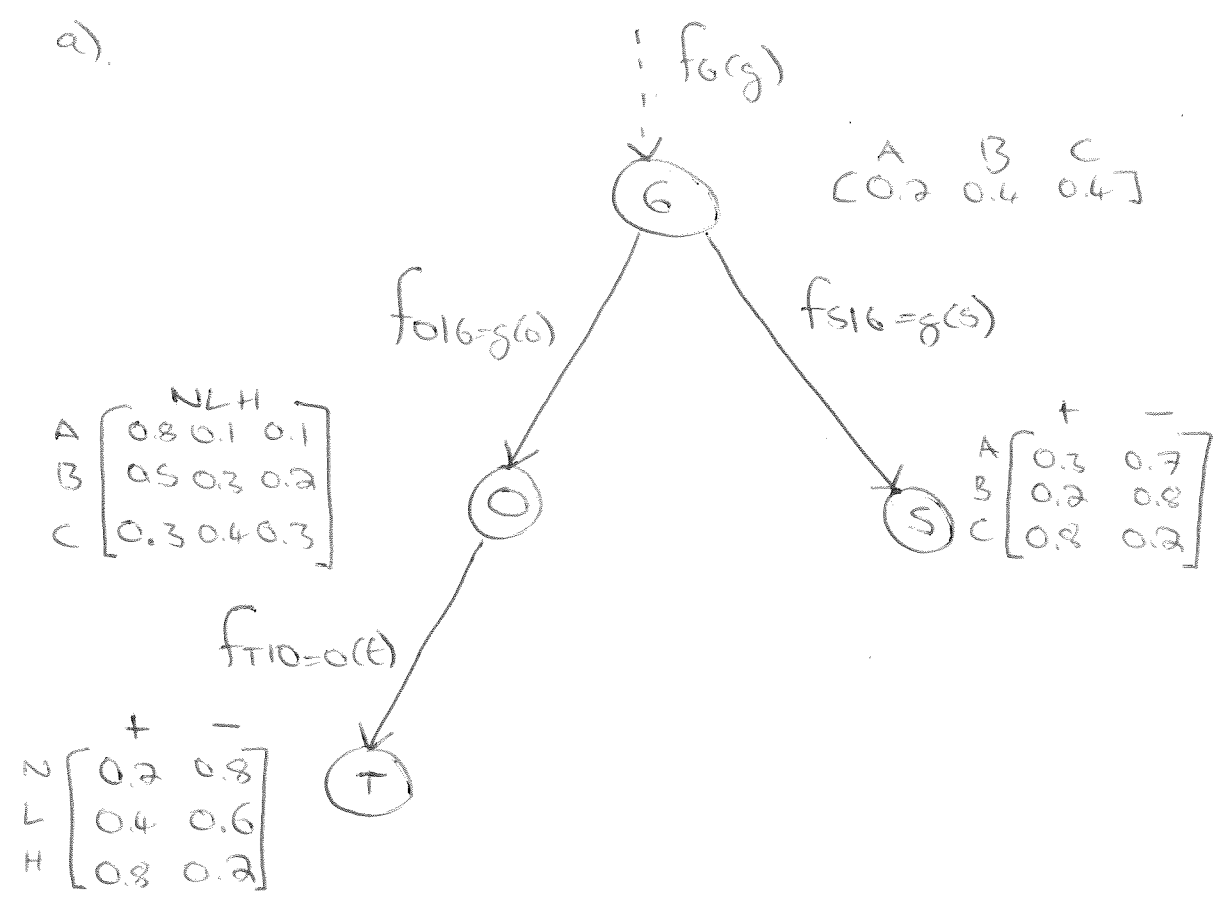
See code in the appendix 5.1

4 Oil Testing with Bayesian Networks and Decision Trees

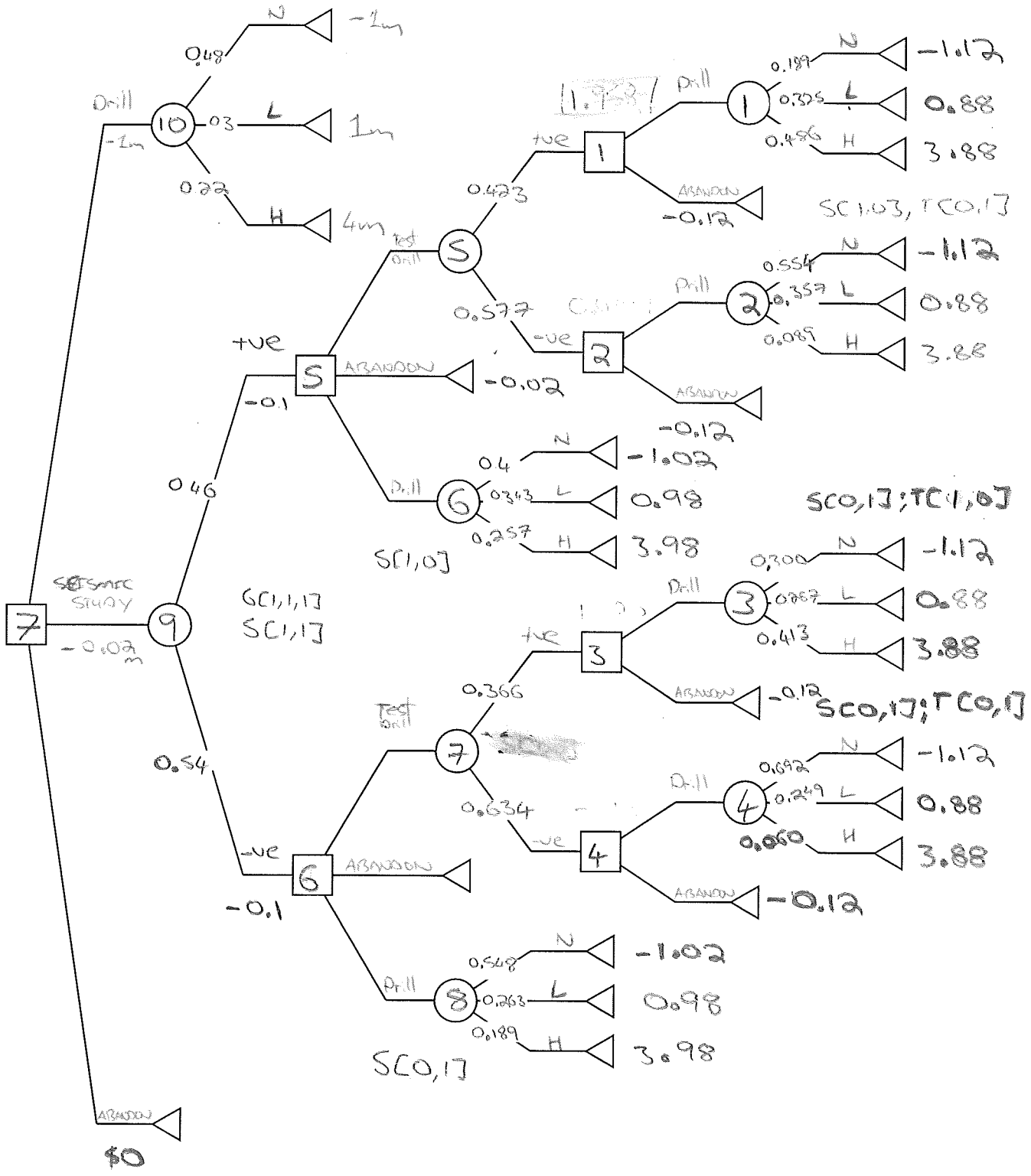
4.1 Bayesian Diagram and Decision Tree

Node G defines the outcome of the geological structure at the well
Node O defines the outcome of oil level given the geological structure of the well
Node T defines the outcome of the test drill given the oil level in the well
Node S defines the outcome of the seismic test given the geological structure of the well

a)


$f_G(g)$
G
A B C
[0.2 0.4 0.4]
$f_{O|G=g(O)}$
$f_{S|G=g(S)}$
O
S
T
$f_{T|O=o(t)}$
N L H
A [0.8 0.1 0.1]
B [0.5 0.3 0.2]
C [0.3 0.4 0.3]
+ −
A [0.3 0.7]
B [0.2 0.8]
C [0.8 0.2]
+ −
N [0.2 0.8]
L [0.4 0.6]
H [0.8 0.2]

b).



All Probabilities are rounded to 3DP for Diagram purposes. Unrounded values are used for Calculation.

See 5.2 for the setup of the belief propagation.

4.2 Intepretation

Using the probabilities and costs, the expected values at each chance and decision node (as per the labelled decision tree) are

Chance Node	Amount ($m)
1	1.95818931
2	0.03813252
3	1.51967613
4	-0.324439251
5	0.849565224
6	0.94956522
7	0.480000013
8	0.45037037
9	0.696000008
10	0.7

Figure 1: Chance Node Expected Values

Decision Node	Amount ($m)
1	1.95818931
2	0.03813252
3	1.51967613
4	-0.12
5	0.94956522
6	0.480000013
7	0.7

Figure 2: Decision Node Expected Values

If Anadarko is risk neutral, the optimal strategy is to drill for oil straight away with an expected profit of $700k. This is more than the expected value of seismic testing and test drilling pathway ($696k) and abandoning the well ($0).

If Anadarko is risk adverse, it is likely they will choose to undergo seismic testing and test drilling. The expected value is only $4000 less and they can choose to abandon the project if they are not comfortable with the test results along the way, minimising their potential losses.

5 Appendix

5.1 HMM code

```
def constructEmissions(pr_correct,adj):
    ## This function takes in a matrix detailing the adjacent letters on a
    # keyboard, and the probability of hitting the correct key and outputs
    # a matrix of emission probabilities
    #
    ## INPUT
    # pr_correct - the probability of correctly hitting the intended key;
    # adj - a 26 x 26 matrix with adj[i][j] = 1 if the i'th letter in the
    # alphabet is adjacent
    # to the j'th letter.
    #
```

```
    # OUTPUT
    # b - a 26 x 26 matrix with b[i][j] being the probability of hitting
    #  key j if you intended
    # to hit key i (the probabilities of hitting all adjacent keys are identical).

        # Import numpy in the function
    import numpy as np

    # Get the dimensions of the adj matrix to work out the number of letters
    letters = len(adj)

    # Create an array of ones to multiply the keyboard adjacency matrix to
    # find the number of adjacent letters by letter.
    oneArray = np.ones(letters)

    # Multiply the keyboard adjacency matrix with the ones array to get
    # the sum of each row, therefore the number of adjacent letters
    sums = np.matmul(adj,oneArray)

    # Use the sums array to calculate probability of hitting an adjacent key
    for i in range(0,letters):
        adj[i][:] = (adj[i][:]/sums[i])*(1-pr_correct)

    # Assign pr_correct down the diagonal
    for i in range(0,letters):
        adj[i][i] = pr_correct

    return adj
    # Assign the probability of hitting the correct key
def constructTransitions(filename):
    # This function constructs tranisition matricies for lowercase characters.
    # It is assumed that the file 'filename' only contains lowercase characters
    # and whitespace.
    ## INPUT
    #  filename is the file containing the text from which we wish to develop a
    #  Markov process.
    #
    ## OUTPUT
    #  p is a 26 x 26 matrix containing the probabilities of transition from a
    #  state to another state, based on the frequencies observed in the text.
    #  prior is a vector of prior probabilities based on how often each character
    #  appears in the text

    ## Read the file into a sting called text
    with open(filename,'r') as myfile:
        text = myfile.read()

    # Import numpy to use matrices
    import numpy as np

    # Create a list of the unique characters in a string
    uniqueChar2 = ['a','b','c','d','e','f','g','h','i','j',
    'k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

    # Create an array for the prior
    prior = np.zeros(len(uniqueChar2))

    # Count the total number of characters in a string excluding spaces and new lines
```

```
    count = 0
    for char in uniqueChar2:
        prior[count] = text.count(char)
        count = count + 1

    total = sum(prior)

    # Convert p to the prior probabilities
    for i in range(0,len(prior)):
        prior[i] = prior[i]/total

    # Initialise the transition probabilities matrix
    p = np.zeros((len(prior),len(prior)),dtype = float)
    # Use a for loop to assign values to the transiition probabilities matrix
    for i in range(0,len(text)-1):
        # Only assign values to the transition probability matrix if the
        # characters exist  in the unique character array
        if text[i] in uniqueChar2 and text[i+1] in uniqueChar2:
            # Use the character array to control indexing therefore array assignment
            p[uniqueChar2.index(text[i])][uniqueChar2.index(text[i+1])] =\
             p[uniqueChar2.index(text[i])][uniqueChar2.index(text[i+1])] + 1

    # Create a vector of ones to perform matrix multiplication
    oneArray = np.ones(len(uniqueChar2))

    # Multiply the transition probabilities matrix with the ones array to get
    # the sum of each row
    sums = np.matmul(p,oneArray)

    # Use the sums array to calculate transition probabilities
    for i in range(0,len(uniqueChar2)):
        p[i][:] = p[i][:]/sums[i]

    return (p,prior)

def HMM(p,pi,b,y):
    ## This function implements the Viterbi algorithm, to find the most likely
    # sequence of states given some set of observations.
    #
    ## INPUT
    #  p is a matrix of transition probabilies for states x;
    #  pi is a vector of prior distributions for states x;
    #  b is a matrix of emission probabilities;
    #  y is a vector of observations.
    #
    ## OUTPUT
    # x is the most likely sequence of states, given the inputs.

    # Import numpy
    import numpy as np

    # Set up the lengths for for loops
    n=len(y) # Number of oservations.
    m=len(pi) # Number of prior distributions.

    # Matrices, each row is a letter (for a given state)
    # for a given observation
    gamma = np.zeros((m,n))
```

```
    phi = np.zeros((m,n))

    # Create a character

    ## You must complete the code below
    for i in range(m):
        # Initialise the algorithm while converting the observation to
        # a number for indexing.
        gamma[i][0] = (b[i][y[0]])*pi[i]

    for t in range(1,n):
        for k in range(26):
            gamma[k,t]=0
            phi[k,t] = 0
            g=[]
            for j in range(26):
                # Calculate the transition probabilities and joint
                # probabilities for the previous state to work out the
                # gamma of this state, appending to a list
                g.append(p[j][k]*gamma[j][t-1])

            # Find the max argmax of the joint probability mulitplied
            # by the transmission probability
            gamma[k,t] = (b[k][y[t]])*np.max(g)
            phi[k,t] = np.argmax(g)

    best=0
    x=[]
    for t in range(n):
        x.append(0)

    # Find the final state in the most likely sequence x(n).
    for k in range(26):
        if best<=gamma[k,n-1]:
            best=gamma[k,n-1]
            x[n-1]=k

    for i in range(n-2,-1,-1):
        # Back track through everything until you get to the end
        x[i] = int(phi[int(x[i+1])][int(i+1)])

    return x

def main():
    # The text messages you have received.
    msgs=[]
    msgs.append('cljlx ypi ktxwf a pwfi psti vgicien aabdwucg vpd me and vtiex voe zoicw
    msgs.append('qe qzby yii tl gp tp yhr cpozwdt fwstqurzby')
    msgs.append('qee ypi xfjvkjv ygetw ib ulur vae')
    msgs.append('wgrrr zrw uiu')
    msgs.append('hpq fzr qee ypi vrpm grfw')
    msgs.append('qe zfr xtztvkmh')
    msgs.append('wgzf tjmr will uiu xjoq jp ywfw')

    print(msgs)

    #The probability of hitting the intended key.
    pr_correct= 0.5
```

```
# An adjacency matrix, adj(i,j) set to 1 if the i'th letter in the alphabet is next
# to the j'th letter in the alphabet on the keyboard.
adj=[[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,1],
     [0,0,0,0,0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0],
     [0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0],
     [0,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,0],
     [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0],
     [0,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0],
     [0,1,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,0],
     [0,1,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0],
     [0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0],
     [0,0,0,0,0,0,0,1,1,0,1,0,1,1,0,0,0,0,0,0,1,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],
     [0,1,0,0,0,0,0,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,1,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
     [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0],
     [0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0],
     [1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1],
     [0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0],
     [0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],
     [0,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
     [1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0],
     [0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1],
     [0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0],
     [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0]]

# Call a function to construct the emission probabilities of hitting a key
# given you tried to hit a (potentially) different key.
b=constructEmissions(pr_correct,adj)

# Call a function to construct transmission probabilities and
# a prior distribution
# from the King James Bible.
[p, prior]=constructTransitions('bible.txt')

# Run the Viterbi algorithm on each word of the messages
# to determine the
# most likely sequence of characters.
for msg in msgs:
    s_in = msg.split('␣') #divide each message into a list of words
    output=''

    for i in range(len(s_in)):
        y=[]

        for j in range(len(s_in[i])):
            y.append(ord(s_in[i][j])-97) #convert the letters
            # to numbers 0-25

        x=HMM(p,prior,b,y) #perform the Viterbi algorithm

        for j in range(len(x)):
            output=output+chr(x[j]+97) #convert the states x back to letters
```

```
            # Add each word to the output list
            if i!=len(s_in)-1:
                output= output + ' ' #recreate the message

        print(msg) #display received message
        print(output) #display decoded message
        print('')

if __name__ == "__main__":
    main()
```

5.2 Belief Propagation

```
import numpy as np

def main():
########################## USER INPUT STARTS HERE ##############################

    # Specify the names of the nodes in the Bayesian network
    nodes=['G','O','T','S']

    # Defining arcs which join pairs of nodes (nodes are indexed 1...N)
    B=[]

    B.append(['G','O'])
    B.append(['G','S'])
    B.append(['O','T'])

    # Set up information struction
    info={}

    # Set up conditional distribution structure
    M={}

    # Specify any given information for each event (a vector of 1s means
    # there is no information given for that event.
    # If information is given for an event, place a 0 corresponding
    # to any outcome that is impossible.
    info['G']=np.array([1,1,1])
    info['O']=np.array([1,1,1])
    info['T']=np.array([1,0])
    info['S']=np.array([0,1])

    # Specify conditional distributions
    M['G']=np.array([0.2,0.4,0.4])
    M['O']=np.array([[0.8,0.1,0.1],[0.5,0.3,0.2],[0.3,0.4,0.3]])
    M['T']=np.array([[0.2,0.8],[0.4,0.6],[0.8,0.2]])
    M['S']=np.array([[0.3,0.7],[0.2,0.8],[0.8,0.2]])

    #Specify the root node and a list of leaf nodes
    root_node='G'
    leaf_nodes=['T','S']

########################## USER INPUT ENDS HERE ################################

    # Set up structures to store parent and child information for each node
    parent={}
    children={}
```

```
count={}
# Define A to be the number of arcs in the Bayesian network
#A=len(B)

# Go through arcs, and define parents and children
for i in range(len(B)):
    if B[i][1] not in parent:
        parent[B[i][1]]=B[i][0]
    else:
        print("Multiple parent nodes dectected for node " + str(B[i][1]))

    if B[i][0] not in children:
        children[B[i][0]]=[]
        count[B[i][0]]=0

    count[B[i][0]]+=1
    children[B[i][0]].append(B[i][1])


# Set up structures for belief propagation algorithm
lambda_={}
lambda_sent={}
pi={}
BEL={}
pi_received={}

# First pass, from the leaf nodes to the root node
Q=leaf_nodes
while len(Q)!=0:
    i=Q.pop(0)

    lambda_[i]=info[i]
    if i in children:
        for j in children[i]:
            lambda_[i]=lambda_[i]*lambda_sent[j]

    if i in parent: # if the node is not the root node, send information to its pare
        lambda_sent[i]=M[i].dot(lambda_[i])
        count[parent[i]]-=1
        if count[parent[i]]==0:
            Q.append(parent[i])

# Second pass, from the root node to the leaf nodes
Q=[root_node]
while len(Q)!=0:
    i=Q.pop(0)
    if i not in parent: # if the node is the root node, pi is set to be the
        #prior distribution at the node
        pi[i]=M[i].T;
    else: # otherwise, pi is the matrix product of the message from the
        #parent and the conditional probability at the node
        pi[i]=M[i].T.dot(pi_received[i]);

    # compute a normalised belief vector
    BEL[i]=pi[i]*lambda_[i]
    BEL[i]=BEL[i]/sum(BEL[i])

    # send adjusted and normalised messages to each child
```

```
            if i in children:
                for j in children[i]:
                    pi_received[j]=BEL[i]/lambda_sent[j]
                    pi_received[j]=pi_received[j]/sum(pi_received[j]);
                    Q.append(j)

    # Display the updated distributions, given the information.
    for i in nodes:
        print(str(i) +": "+str(BEL[i]))



if __name__ == "__main__":
    main()
```