Graphical Models - Assignment 2

James Gin, John Goodacre, Christopher Loy, Mark Neumann

December 2015

Division of work

We initially divided the questions up as follows:

- C Loy and M Neumann: 6.7, 6.9, 6.10, 6.12
- J Gin and J Goodacre: 5.8,7.4,7.13

This allowed us to start on questions in pairs, however almost every solution given below involved considerable collaboration between two or more members of the group, including "cross-pair"; the final submission has been agreed between all members to represent a collaborative effort.

Problem 5.8

The question gives 10000 characters of a noisy string with a known set of rules for the probabilities of transitioning between states. There are five first names and seven surnames. The question asks for a most likely decoding given the various rules.

We chose to implement the problem as a hidden markov model. In addition, the Viterbi algorithm seemed particularly applicable for finding the most likely hidden state. The hidden state has a set of rules whereby we first go from a random letter, to either the same state or a first name - once we are within a first name we continue until the end at which point we move to a second random state and the process repeats with the surname.

We chose to represent this using 83 variables. In short, the two random states representing whether we move to a first name next or a second. Also each letter of the first names and the second names is represented separately, referred to here as letter states. The reason for this is for example if we look at the first name david. The probability of transitioning from R1 - our first random state to the first letter state of a first name, is 0.2/5 (5 first names). However once we arrive at this state, which should generate the character 'd' the probability of transitioning to the second letter state, which generates 'a' is 1. This continues until the last letter state, where the probability of transitioning to the second random state R2 is again 1. R2 has transition probabilities to

the first letter states of the last names in a similar fashion. So our transition matrix is an 83x83 matrix. The emission matrix is more straight forward - both random states generate characters with a uniform probability of 1/26 chance. Letter states generate their corresponding character with a probability of 0.3, or can be another of the remaining 25 letters with a probability of 0.7. Thus, the emission matrix is a 26x83 matrix. Finally we know our starting state - a probability of 1 of being in R1.

This was implemented in the code below and the Viterbi algorithm from the BRML toolbox used. In addition we implemented a test to check that the decoded string maintained valid sequences ie. that events such as jumping from R1 to a second name were not possible, or indeed into the middle of any of the names.

Results We found 364 first names and 363 second names. Thus, 363 matches overall. The question asks for the counts of each first name, surname within a matrix. Where the numbers representing each name are given - i.e. for first names 1 for David ... and for surnames, 1 for Barber ...

The counts of each first name, surname pair are given below:- eg (1,1) equates to David Barber with a count of 6. The (i,j)th element corresponding to the name numbers in the question.

$$\begin{pmatrix} 6 & 2 & 9 & 13 & 15 & 9 & 12 \\ 8 & 5 & 3 & 8 & 20 & 10 & 13 \\ 6 & 5 & 9 & 10 & 11 & 7 & 19 \\ 8 & 9 & 7 & 18 & 19 & 13 & 19 \\ 12 & 6 & 10 & 9 & 8 & 10 & 15 \end{pmatrix}$$

```
function r = vit58
import brml.*
load noisystring.mat
%genstate
r1 = 1; r2 = 2;
%previous name
fname=1; lname=2;
%fnames
david = 1; anton = 2; fred = 3; jim = 4; barry = 5;
%lnames
barber = 6; ilsung = 7; fox = 8; chain = 9; fitzwilliam = 10;
quinceadams = 11; grafvonunterhosen = 12;
chars = 10000;
a=1;b=2;c=3;d=4;e=5;f=6;g=7;h=8;i=9;j=10;k=11;l=12;m=13;n=14;o=15;p=16;
q=17;r=18;s=19;t=20;u=21;v=22;w=23;x=24;y=25;z=26;
chararray = 'abcdefghijklmnopqrstuvwxyz';

names{1} = [d,a,v,i,d];
names{2} = [a,n,t,o,n];
names{3} = [f,r,e,d];
names{4} = [j,i,m];
names{5} = [b,a,r,r,y];
names{6} = [b,a,r,b,e,r];
```

```
names{7} = [i,l,s,u,n,g];
names{8} = [f,o,x];
names{9} = [c,h,a,i,n];
names{10} = [f,i,t,z,w,i,l,l,i,a,m];
names{11} = [q,u,i,n,c,e,a,d,a,m,s];
names{12} = [g,r,a,f,v,o,n,u,n,t,e,r,h,o,s,e,n];

curstate = 3;
for nn = 1:12
    nextState = curstate + length(names{nn}) - 1;
    name_states{nn} = curstate:nextState;
    curstate = nextState + 1;
end

tt = zeros(83,83); % generation state transition matrix
for nm = 1:5
    states = name_states{nm};
    tt(r1, states(1)) = 0.2 / 5; %progress from r1 to firstname
    for ch = 1:length(names{nm})-1
        tt(states(ch),states(ch+1)) = 1; %progress to next letter state
    end
    tt(states(end), r2) = 1; %progress from end to r2
end
for nm = 6:12
    states = name_states{nm};
    tt(r2, states(1)) = 0.2 / 7; %progress from r2 to lastname
    for ch = 1:length(names{nm})-1
        tt(states(ch),states(ch+1)) = 1; %progress to next letter state
    end
    tt(states(end), r1) = 1; %progress from end to r1
end
tt(r1, r1) = 0.8;
tt(r2, r2) = 0.8;

numchar = zeros(1,chars);
for ch = 1:chars
    numchar(1,ch) = find(chararray==noisystring(ch));
end

%generate noisy char transition matrix
ct = zeros(83,26);
%rand states
ct(r1,:)= 1/26;
ct(r2,:) = 1/26;
for nm = 1:12 %name states
    name = names{nm};
    states = name_states{nm};
    for ch=1:length(name)
        ss = states(ch);
        ct(ss,:) = 0.7 * 1/25;
```

```
        ct(ss,name(ch)) = 0.3;
    end
end

start_vec = zeros(1,83);
start_vec(1) = 1;

[viterbimaxstate
    logprob]=HMMviterbi(numchar(1:chars),tt',start_vec',ct');
checkSequence(viterbimaxstate, tt);

[fname sname] = count_names(viterbimaxstate);
% a little naughty but only 363 surname so add a dummy for sname - this
    does not
% generalise of course
sname = sname - 5; %to index names as per question
sname = [sname ;1000]; %the 1000 is a placeholder - there is a stranded
    firstname
names = [fname sname];

%the output of firstname surname counts - there was one more david but no
%matched surname
%1=david ... 1= Barber...for surnames (i,j) = (firstname,surname)
nmat_count = zeros(5,7);

for z=1:size(fname,1)
    i=fname(z);
    j=sname(z);
    if i<=5 && j<=7 %to take into account the stranded first name!
        nmat_count(i,j) = nmat_count(i,j)+1;
    end
end

%the final matrix count as per the question
nmat_count
end

%function to check we end up with a valid sequence and no crazy
    transitions
function r=checkSequence(sequence, transitionMatrix)
for s = 1:length(sequence)-1
    pnext = transitionMatrix(sequence(s), sequence(s+1));
    if pnext == 0
        disp('Invalid!')
        r=0;
        break
    end
end
r=1;
end
```

```
function r=noise(id, chars)
r = id + chars;
end

function [first_name second_name]=count_names(maxstate)
%assumes the maxstate list has been checked to have valid transitions
%if so it counts the number of occurences of each first name and surname
%given valid transitions we need only check the first letter
first_name = [];
second_name = [];
for i=1:10000
    if maxstate(i) == 3
        first_name = [first_name;1];
    elseif maxstate(i) == 8
        first_name = [first_name;2];
    elseif maxstate(i) == 13
        first_name = [first_name;3];
    elseif maxstate(i) == 17
        first_name = [first_name;4];
    elseif maxstate(i) == 20
        first_name = [first_name;5];
    elseif maxstate(i) == 25
        second_name = [second_name;6];
    elseif maxstate(i) == 31
        second_name = [second_name;7];
    elseif maxstate(i) == 37
        second_name = [second_name;8];
    elseif maxstate(i) == 40
        second_name = [second_name;9];
    elseif maxstate(i) == 45
        second_name = [second_name;10];
    elseif maxstate(i) == 56
        second_name = [second_name;11];
    elseif maxstate(i) == 67
        second_name = [second_name;12];
    end
end

end
```

Problem 6.7

By collapsing the nxn lattice down into n variables, we require that the represented number of states is the same, meaning that we must take into consideration all possible combinations of n binary variables in each node, which given that we have binary variables in the lattice corresponds to 2^n states per

variable. As inference in a singly connected Markov Chain scales linearly with the number of nodes due to message passing and the commutativity of summations over the variables in the chain, the running time for calculating the normalisation constant Z is $O(n2^n)$.

Figure 1: The singly connected graph resulting from collapsing an nxn lattice down into n variables.

Compute log Z for n = 10

Consider the cluster potential X_i, which has 2^n possible states. One can calculate the pairwise potential of two cluster variables $\phi(X_i, X_j)$ where j is either $i+1$ or $i-1$, by enumerating over every possible state s_i and s_j, calculating the total unary (within the column) and pairwise potentials that the configuration would imply in the grid. Due to the design of the potential, consider representing the potential as the sum of the counts of neighbours that are in the same state denoted here $N_{i,j}$, such that $\phi(X_i = s_i, X_j = s_j) = e^{N_{i,j}}$. Using this potential and the fact that the resultant graph is a singly connected tree, we can calculate the total probability $1/Z$, therefore this approach complexity is bounded by $(2^n)^2 = 2^{2n}$ which is an improvement on the direct 2^{n^2} grid but still exponential.

Consider a two column system, which reduces to two cluster variables. For $n = 10$ it is computationally feasible to calculate the $2^{10} \times 2^{10}$ joint potential of this system. The sum of the rows represent the total potential of all of the possible 2-length paths that end in state s_i. We define the total potential ending in state s_i for a path length l as $T_{i,l}$. Clearly for $l = 2$ we have:

$$T_{i,2} = \sum_{j}^{2^n} e^{N_{i,j}}$$

Considering $T_{i,l+1}$, we can propagate the total potential to the next cluster variable in the chain by considering that the potential chains length l ending in s_i can be extended by multiplying them by the potential of all the possible transitions from s_i to another state:

$$T_{i,l+1} = \sum_{j}^{2^n} T_{i,l} e^{N_{i,j}}$$
$$= T_{i,l} \sum_{j}^{2^n} e^{N_{i,j}}$$

Clearly also:

$$Z = \frac{1}{\sum_{i}^{2^n} T_{i,n}}$$
$$\log(Z) = -\log\left(\sum_{i}^{2^n} T_{i,n}\right)$$

The following code performs this calculation to find that when $n = 10$, $\log(Z) = -268.7497$.

```
%Potential of cluster variables being in state1 and state2. States are
    numbered by finding the binary representation of the number and
    using the bits as the state of the i-th node in the column of the
    Ising grid each cluster variable represents.
function p = getPot(state1, state2, n)
s1 = dec2bin(state1);
s2 = dec2bin(state2);
count = 0;
for l = length(s1):n
    s1(l)='0';
end
for l = length(s2):n
    s2(l)='0';
end
for i = 1:(n-1)
    count = count + 1*(s2(i+1)==s2(i));
    count = count + 1*(s1(i+1)==s1(i));
end
for i = 1:n
    count = count + 1*(s1(i)==s2(i));
end
p = count;
end

function logZ = ising
%Calculate the total log Z using the approach explained above: create
    the joint potential of the first 2 states, then find the total
    potential ending in state s_i for chains length 3:10.
n = 10;
```

```
countmat = zeros(2^n);
for i = 1:2^n
    for j = 1:2^n
        countmat(i,j) = getPot(i,j,n);
    end
end

transmat = exp(countmat);

row = sum(transmat);
for i=1:8
    newpot = bsxfun(@mmult, transmat, row);
    row = sum(newpot, 2)';
end

logZ = -log(sum(row))

end
```

Problem 6.9

Using the BRMLtoolbox, construct a junction tree for this distribution and use it to compute all the marginals of the symptoms, $p(s_i = 1)$**.**

The code for calculating and displaying the data below is as follows::

```
import brml.*

load('diseaseNet');

[jtpot, jtsep, infostruct] = jtree(pot);
[jtpot, ~, ~] = absorption(jtpot, jtsep, infostruct);

for i = 21:60
  jtpotnum = whichpot(jtpot,i,1);
  margpot=sumpot(jtpot(jtpotnum),i,0);
  disp(['p(s' num2str(i-20) '=1) = '
      num2str(margpot.table(1)./sum(margpot.table))]);
end
```

The resulting marginal probabilities calculated in this manner are shown in Table 1.

$p(s_1 = 1)$	0.44183	$p(s_{14} = 1)$	0.63296	$p(s_{28} = 1)$	0.46962
$p(s_2 = 1)$	0.45668	$p(s_{15} = 1)$	0.42954	$p(s_{29} = 1)$	0.52287
$p(s_3 = 1)$	0.44141	$p(s_{16} = 1)$	0.45879	$p(s_{30} = 1)$	0.71731
$p(s_4 = 1)$	0.49127	$p(s_{17} = 1)$	0.42756	$p(s_{31} = 1)$	0.5242
$p(s_5 = 1)$	0.49389	$p(s_{18} = 1)$	0.40426	$p(s_{32} = 1)$	0.3537
$p(s_6 = 1)$	0.65748	$p(s_{19} = 1)$	0.58209	$p(s_{33} = 1)$	0.51268
$p(s_7 = 1)$	0.50456	$p(s_{20} = 1)$	0.58959	$p(s_{34} = 1)$	0.5294
$p(s_8 = 1)$	0.26869	$p(s_{21} = 1)$	0.76127	$p(s_{35} = 1)$	0.38575
$p(s_9 = 1)$	0.64908	$p(s_{22} = 1)$	0.69559	$p(s_{36} = 1)$	0.48909
$p(s_{10} = 1)$	0.49074	$p(s_{23} = 1)$	0.5087	$p(s_{37} = 1)$	0.6336
$p(s_{11} = 1)$	0.42255	$p(s_{24} = 1)$	0.41996	$p(s_{38} = 1)$	0.5896
$p(s_{12} = 1)$	0.4291	$p(s_{25} = 1)$	0.35194	$p(s_{39} = 1)$	0.42316
$p(s_{13} = 1)$	0.54502	$p(s_{26} = 1)$	0.38961	$p(s_{40} = 1)$	0.52823
		$p(s_{27} = 1)$	0.32597		

Table 1: Table of marginal symptom probabilities.

Explain how to compute the marginals $p(s_i = 1)$ in a way more efficient than using the junction tree formalism. By implementing this method, compare it with the results from the junction tree algorithm.

The junction tree algorithm is rather inefficient in this case, as the resulting Markov network is highly connected between the disease nodes, meaning that clique sizes are large and hence run times dominated by the larger cliques.

Additionally, inspecting the junction tree that is generated, it can be seen that there are 49 cliques formed, of which 3 contain 12 variables, and 2 contain 10 variables. Calculating marginals from these cliques will be exponential and slow.

Individual marginals can be more efficiently calculated by using loopy belief propagation. This is an approximate method, but with the correct tuning still returns the same results as the exact junction tree calculation:

```
import brml.*

load('diseaseNet');

opt.tol=1; opt.maxit=2;
[marg, ~, ~] = LoopyBP(pot, opt);

for i = 21:60
  disp(['p(s' num2str(i-20) '=1) = ' num2str(marg{i}.table(1))]);
end
```

To within the four decimal places considered, this code produces identical results to junction tree results shown in Table 1.

Symptoms 1 to 5 are present (state 1), symptoms 6 to 10 not present (state 2), and the rest not known. Compute the marginal $p(d_i = 1|s_{1:10})$ for all diseases.

Table 2 shows the marginal disease probabilities as calculated using the following code:

```
import brml.*

load('diseaseNet');

newpot = squeezepots(setpot(pot, 21:30, [1 1 1 1 1 2 2 2 2 2]));
[jtpot, jtsep, infostruct] = jtree(newpot);
[jtpot, ~, ~] = absorption(jtpot, jtsep, infostruct);
for i = 1:20
  jtpotnum = whichpot(jtpot,i,1);
  marg=sumpot(jtpot(jtpotnum),i,0);
  disp(['p(d' num2str(i) '=1) = '
      num2str(marg.table(1)./sum(marg.table)) ' \\']);
end
```

$p(d_1 = 1)$	0.029776	$p(d_{11} = 1)$	0.28732
$p(d_2 = 1)$	0.38176	$p(d_{12} = 1)$	0.48983
$p(d_3 = 1)$	0.95423	$p(d_{13} = 1)$	0.8996
$p(d_4 = 1)$	0.39664	$p(d_{14} = 1)$	0.61956
$p(d_5 = 1)$	0.49647	$p(d_{15} = 1)$	0.92048
$p(d_6 = 1)$	0.43515	$p(d_{16} = 1)$	0.7061
$p(d_7 = 1)$	0.18749	$p(d_{17} = 1)$	0.20125
$p(d_8 = 1)$	0.70118	$p(d_{18} = 1)$	0.90849
$p(d_9 = 1)$	0.043127	$p(d_{19} = 1)$	0.86497
$p(d_{10} = 1)$	0.61031	$p(d_{20} = 1)$	0.88393

Table 2: Table of marginal disease probabilities given $s_{1...5} = 1$ and $s_{6...10} = 2$.

Problem 6.10

1. Draw a junction tree for this distribution and explain the computational complexity of computing $p(x_T)$, as suggested by the junction tree algorithm.

Initially, we draw the belief network for this distribution for clarity:

When we take this graph and moralise parents of nodes to form a Markov network, as the first step of forming a Junction tree, we see that because every X_i is a parent of Y, we are required to add direct paths between every X_i and X_j. This results in a the complete graph $K_{t+1} = \{X_1, X_2....X_t, Y\}$ as Y is

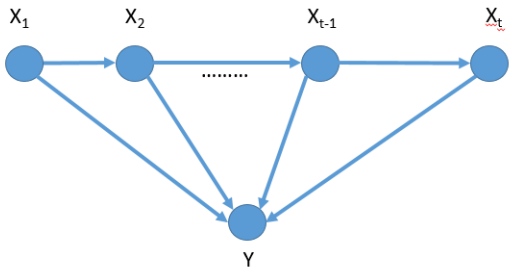

Figure 2: Belief Network for distribution

already connected to every X vertex. As the complete graph K_{t+1} is simply a size $t+1$ maximal clique, the graph is already triangulated by definition - there are no loops of more than size 3 without a chord, as if there were, there would be two vertices which were not directly connected, raising a contradiction.

The clique graph of this markov network corresponding to a maximal clique is simply a single node with one potential $\phi(K_{t+1})$ defined over every variable in the clique K_{t+1}:

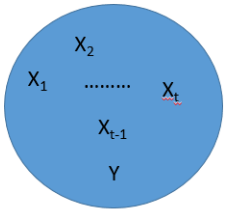

Figure 3: A single potential defined over a maximal clique is the minimal representation of the distribution as a Clique Graph

As the Junction Tree of the above belief network is a single clique and the variables are binary, using the JT methodology to compute the marginal $p(x_t)$ would result in a time exponential in the number of variables in the largest clique(in this instance the only clique). Since the variables are all binary, the computational complexity therefore grows with 2^{T+1}.

2. By using an approach different from the plain JTA above, explain how $p(x_T)$ can be computed in time that scales linearly with T.

We are free to express y in terms of some other, unknown variables in our graph, such that:

$$p(y|x_{1:T}) = p(y_1|x_1)p(y_2|y_1, x_2)\dots p(y_T|p_{T-1}, x_t)$$

In this case, our belief network appears as shown in Figure 4.

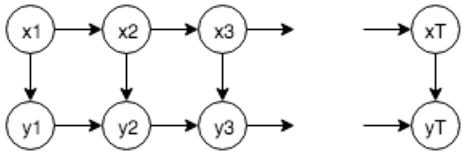

Figure 4: A hypertree representation of Figure 2, in which y has been split into multiple nodes.

This representation is now a tractable hypertree, for which we can solve by grouping pairs of $\{x_i, y_i\}$ and eliminating variables through message passing. To solve for x_T, we pass messages in the direction of $1 \rightarrow T$. Once this is complete, we will have a solution for $\{x_T, y_T\}$, in terms of $x_{1:T-1}$, which is known, and $y_{1:T-1}$. At this point, we can marginalise over these variables, using the observation that our decomposition of y only holds in the case that all values $y_i = y_{i+1} \forall i \in [1, t-1]$ - i.e. the pairwise potential $\phi(y_i, y_{i+1})$ is zero for all cases except uniformly matching y_is. Our marginal is then expressed as a sum of the two possible group states of y_i and x_T can be found.

Note that this formulation scales linearly in the size of T, although there are multiple calculations needed at each step of the hyperchain.

Problem 6.12

Show that eliminating clique 1 by summing over all variables in clique 1 not in separator 1 gives a new JT.

Given the initial distribution

$$p(\chi) = \frac{\phi_1(\chi_1)\prod_{c\geq 2}\phi_c(\chi_c)}{\psi_1(S_1)\prod_{s\geq 2}\psi_c(S_c)} \tag{1}$$

we select the clique potential $\phi_d(\chi_d)$ to be the potential of the single clique adjacent to χ_1. In order to represent the probability distribution after marginalisation of χ_1, we note that

$$p(\chi \setminus \chi_1) = \sum_{\chi_1 \setminus S_1} p(\chi) = \sum_{\chi_1 \setminus S_1} \frac{\phi_1(\chi_1)\phi_d(\chi_d)\prod_{c\geq 2, \neq d}\phi_c(\chi_c)}{\psi_1(S_1)\prod_{s\geq 2}\psi_c(S_c)} \tag{2}$$

so therefore by redefining $\phi'(\chi_d)$ and using the fact that the clique χ_1 only connects to χ_d(it is a leaf of the JT), meaning that the variables in the set $\{\chi_1 \setminus S_1\}$ only occur in χ_1, meaning that the sum over these variables only applies to the potential $\phi_1(\chi_1)$, we see that we can write:

$$\sum_{\chi_1 \setminus S_1} \frac{\phi_1(\chi_1)\phi_d(\chi_d)\prod_{c\geq 2,\neq d}\phi_c(\chi_c)}{\psi_1(S_1)\prod_{s\geq 2}\psi_c(S_c)} = \frac{\phi_d(\chi_d)\sum_{\chi_1\setminus S_1}\phi_1(\chi_1)\prod_{c\geq 2,\neq d}\phi_c(\chi_c)}{\psi_1(S_1)\prod_{s\geq 2}\psi_c(S_c)} \tag{3}$$

Using the substitution

$$\phi'(\chi_d) = \frac{\phi_d(\chi_d)\sum_{\chi_1\setminus S_1}\phi_1(\chi_1)}{\psi_1(\chi_1)} \tag{4}$$

we achieve the resulting JT:

$$\phi'_d(\chi_d)\frac{\prod_{c\geq 2,\neq d}\phi_c(\chi_c)}{\prod_{s\geq 2}\psi_c(S_c)} \tag{5}$$

Show that by continuing in this manner, eliminating cliques one by one, the last clique must contain the marginal $p(X_n)$.

As the above method demonstrates a general elimination procedure for a leaf clique of a junction tree, we re-use this process again in a recursive structure:

$$p(\chi^{(i+1)}) = \frac{\prod_{j\in R}\phi'_j(\chi_j)\prod_{c\notin Q}\phi_c(\chi_c)}{\prod_{s\geq i}\psi_c(S_c)} \tag{6}$$

where Q is the set of all already absorbed cliques, R is the set of cliques connected to cliques that have been absorbed and $\phi'_j(\chi_j)$ is the updated potential of the clique χ_j, the only clique directly attached to χ_i. This process is twice recursive, as once we have processed all of the original leaves of the graph, the next set of nodes which we process contain the marginals of the previous layer. As demonstrated above, each clique elimination corresponds to the marginal of the eliminated clique being represented within the potential of the clique to which it is connected, displayed below:

$$\phi'(\chi_j) = \frac{\phi_j(\chi_j)\sum_{\chi_i\setminus S_i}\phi_i(\chi_i)}{\psi_i(\chi_i)} \tag{7}$$

Therefore, when this marginalisation is run systematically over the entire graph, the result will be a nested structure of summations over individual clique variables minus their separators, correlating to the structure of the tree, where the potential of the root clique X_n has the following form:

$$\phi_n(\chi_n) = \frac{\phi_n(\chi_n)\sum_{\chi_{n-1}\setminus S_{n-1}}\phi'_{n-1}(\chi_{n-1})}{\psi_1(\chi_{n-1})} \tag{8}$$

where ϕ'_{n-1} also contains information about cliques which it absorbed. When we expand all of these messages out, we see that:

$$\phi'_n(\chi_n) = \frac{\phi_n(\chi_n) \sum_{\chi_{n-1} \setminus S_{n-1}} \phi_{n-1}(\chi_{n-1}) ..... \sum_{\chi_{n-2} \setminus S_{n-2}} \phi_{n-2}(\chi_{n-2})}{\psi_{n-1}(\chi_{n-1}) \psi_{n-2}(\chi_{n-2}) .... \psi_1(\chi_1)} \tag{9}$$

All of the variables being summed over at each point in the absorption process are unique(due to the specific order of elimination from leaves to the root), meaning that they are not involved in any of the other summation terms and specifically, variables which are excluded in the absorption routine for a particular clique are then included in the marginalisation when the parent subsequently gets absorbed, meaning that every variable is summed over exactly once throughout the entire process. We can re-write the above as a single summation over a larger product of clique potentials as follows:

$$\phi'_n(\chi_n) = \phi_n(\chi_n) \sum_{\chi_{1 \dots n-1} \setminus S_{1 \dots n-1}} \frac{\prod_{i=1}^{n-1} \phi_i(\chi_i)}{\prod_{i=1}^{n-2} \psi_i(\chi_i)} = \sum_{x_1 .... x_{n-1}} P(x_1 .... x_n) = p(x_n) \tag{10}$$

Explain how by now reversing the elimination schedule, and eliminating cliques one by one, updating their potentials in a similar manner to Equation(6.12.8), the updated cliques $\phi_j(X_j)$ will contain the marginals $p(X_j)$.

In order to demonstrate this fact, we first argue that cliques of distance one away from the root of the tree contain their respective marginals. By recursively defining the root of the tree at the current node, this argument will apply to the rest of the graph.

Given the root clique potential $\phi_n(\chi_n)$, we can define a message pass to the clique χ_{n-1} in the same way as previously, denoting the updated potential by:

$$\phi''_{n-1}(\chi_{n-1}) = \frac{\phi'_{n-1}(\chi_{n-1}) \sum_{\chi_n \setminus S_{n-1}} \phi_n(\chi_n)}{\psi'_n(\chi_n)} \tag{11}$$

As we have passed messages both ways across this separator, it contains the marginal of the variables in the separator. This is also the case for the updated clique potentials as we show below:

$$\frac{\phi'_{n-1}(\chi_{n-1}) \sum_{\chi_n \setminus S_{n-1}} \phi_n(\chi_n)}{\psi'_n(\chi_n)} = \frac{\frac{\phi_{n-1}(\chi_{n-1}) \sum_{\chi_d \setminus S_d} \phi_d(\chi_d)}{\psi_d(\chi_d)} \frac{\sum_{\chi_n \setminus S_{n-1}} \phi_n(\chi_n) \psi'_n(\chi_n)}{\psi_n(\chi_n)}}{\psi'_n(\chi_n)} \tag{12}$$

From the above, we can see that the potentials over the separators of χ_n are going to cancel, resulting in:

$$\frac{\boldsymbol{\phi_{n-1}(\chi_{n-1})} \sum_{\chi_d \setminus S_d} \phi_d(\chi_d) \sum_{\chi_n \setminus S_{n-1}} \phi_n(\chi_n)\psi'_n(\chi_n)\psi_n(\chi_n)}{\psi_d(\chi_d)} \tag{13}$$

However, we also note that again we have a separator message from the first wave of message passing over the variables. When we recursively expand this potential over the separators proceeding down the tree, we will again end up with a product of summations distributed across all variables, except those contained within $\phi_{n-1}(\chi_{n-1})$. However, note that in bold above, we see that we already have $\phi_{n-1}(\chi_{n-1})$ within our term, so the product of all of these sums forms the full distribution $p(\chi)$, meaning that we end up with:

$$\phi''_{n-1}(\chi_{n-1}) = \sum_{\chi_i \setminus \chi_{n-1}} \frac{\prod_{i=1}^{n} \phi_i(\chi_i)}{\prod_{i=1}^{n} \psi_i(\chi_i)} = \sum_{\chi_i \setminus \chi_{i-1}} p(x_1.....x_n) = p(\chi_{n-1}) \tag{14}$$

demonstrating that after a complete round of two-way message passing, the clique χ_{n-1} contains the marginal over all the variables contained in the clique.

This result generalises down to the leaves of the tree, as we can simply re-define the root of a sub-tree to be the clique χ_{n-1}. As all the messages passed up from any leaf which is the descendant of χ_{n-1} must by definition be contained within $\phi'\chi_{n-1}$, we can see that the reverse message pass will have the same effect as the above demonstrated example, yielding the marginals of the leaf cliques within the potentials of these cliques.

Hence explain why, after updating cliques according to the forward and reversed elimination schedules, the cliques must be globally consistent.

After a full round of message passing on the junction tree, we know that each pair of connected cliques is locally consistent. Therefore, summing over all variables excluding the variables included in a clique's separator will yield the potential of the separator after the first message has been passed over it, which corresponds to the marginal of those variables. This means that when we sum over a clique, we get:

$$\sum_{\chi_d \setminus S_d} \frac{\phi'_{\chi_d}}{\psi_d(\chi'_d)} = 1 \tag{15}$$

Therefore, when we sum over a clique, we can simply remove it and the connecting separator. This means that when summing over other cliques in the tree, we can still obtain their marginals via the same method as we propagate up the tree, meaning that local consistency is enforced everywhere - hence global consistency.

This is also the case almost by the definition of a Junction Tree as a clique graph satisfying the running intersection property. For a given pair of nodes,

the path between them must contain their intersections, so local consistency is enforced everywhere.

Problem 7.4

This question follows closely demoMDP.m, and represents a problem in which a pilot wishes to land an airplane. The matrix U(x,y) in the file airplane.mat contains the utilities of being in position x, y and is a very crude model of a runway and taxiing area. The airspace is represented by an 18 15 grid (Gx = 18, Gy = 15 in the notation employed in demoMDP.m). The matrix U (8, 4) = 2 represents that position (8, 4) is the desired parking bay of the airplane (the vertical height of the airplane is not taken in to account). The positive values in U represent runway and areas where the airplane is allowed. Zero utilities represent neutral positions. The negative values represent unfavourable positions for the airplane. By examining the matrix U you will see that the airplane should preferably not veer off the runway, and also should avoid two small villages close to the airport. At each timestep the plane can perform one of the following actions stay up down left right: For stay, the airplane stays in the same x, y position. For up, the airplane moves to the x, y + 1 position. For down, the airplane moves to the x, y 1 position. For left, the airplane moves to the x 1, y position. For right, the airplane moves to the x + 1, y position.

For part 1, we are told that the plane can fly in a deterministic manner and are asked to find the optimal sequence of moves from x=1, y=13.

The code below is really an implementation of demoMDP with the slight variation that the grid size is now 18x15 and the utilities from airplane.mat. It covers both the deterministic case the probabilistic second part, the deterministic case being commented out.

```
function airplaneMDP

import brml.*
load airplane;
Gx = 18; Gy = 15; % two dimensional grid size
S = Gx*Gy; % number of states on grid
st = reshape(1:S,Gx,Gy); % assign each grid point a state

A = 5; % number of action (decision) states
[stay up down left right] = assign(1:A); % actions (decisions)
p = zeros(S,S,A); % initialise the transition p(xt|xtm,dtm) ie
    p(x(t)|x(t-1),d(t-1))

% make a deterministic transition matrix on a 2D grid:
for x = 1:Gx
```

```
   for y = 1:Gy
      p(st(x,y),st(x,y),stay)=1; % can stay in same state
      if validgridposition(x+1,y,Gx,Gy)
           %deterministic plane movement
         %p(st(x+1,y),st(x,y),right)=1;

         %include probabilistic action for going right
            p(st(x+1,y),st(x,y),right)=0.9;
             if validgridposition(x,y+1,Gx,Gy)
                 p(st(x,y+1),st(x,y),right)=0.1;
             else
                 p(st(x+1,y),st(x,y),right)=1;
             end
      end
      if validgridposition(x-1,y,Gx,Gy)
         p(st(x-1,y),st(x,y),left)=1;
      end
      if validgridposition(x,y+1,Gx,Gy)
         p(st(x,y+1),st(x,y),up)=1;
      end
      if validgridposition(x,y-1,Gx,Gy)
         p(st(x,y-1),st(x,y),down)=1;
      end
   end
end

gam = 0.95; % discount factor
U = reshape(U,S,1);
figure; imagesc(reshape(U,Gx,Gy)); colorbar; title('utilities'); pause

[xt xtm dtm]=assign(1:3); % assign the variables x(t), x(t-1), d(t-1) to
    some numbers

% define the transition potentials p(x(t)|x(t-1),d(t-1))
tranpot=array([xt xtm dtm],p);
% setup the value potential v(x(t))
valpot=array(xt,ones(S,1)); % initial values

maxiterations=30; tol=0.001; % termination criteria
% Value Iteration:
oldvalue=valpot.table;
for valueloop=1:maxiterations
   valueloop
   tmppot = maxpot(sumpot(multpots([tranpot valpot]),xt),dtm);
   valpot.table = U + gam*tmppot.table; % Bellman's recursion
   if mean(abs(valpot.table-oldvalue))<tol; break; end % stop if
       converged
   oldvalue = valpot.table;
   imagesc(reshape(valpot.table,Gx,Gy)); colorbar; drawnow
end
```

```
figure; bar3zcolor(reshape(valpot.table,Gx,Gy));

% Policy Iteration:
valpot.table=ones(S,1); % initial values
oldvalue=valpot.table;
figure;
for policyloop=1:maxiterations
   policyloop
   % Policy evaluation: get the optimal decisions as a function of the
       state:
   [tmppot dstar] = maxpot(sumpot(multpots([tranpot valpot]),xt),dtm);
   for x1=1:S
      for x2=1:S
         pdstar(x1,x2) = p(x2,x1,dstar(x1));
      end
   end
   valpot.table = (eye(S)-gam*pdstar)\U;
   if mean(abs(valpot.table-oldvalue))<tol; break; end % stop if
       converged
   oldvalue=valpot.table;
   imagesc(reshape(valpot.table,Gx,Gy)); colorbar; drawnow
end
figure; bar3zcolor(reshape(valpot.table,Gx,Gy));

bestpol = reshape(valpot.table,Gx,Gy);
pol_val = bestpol(1,13);

x_val = 1;
y_val = 13;
moves = [x_val, y_val];
% insert code to work out the best route
while pol_val < bestpol(8,4)
    left = -1000;
    right = -1000;
    up = -1000;
    down = -1000;
    if validgridposition(x_val-1,y_val,Gx,Gy)
      left = bestpol(x_val-1,y_val);
    end
    if validgridposition(x_val+1,y_val,Gx,Gy)
       right = bestpol(x_val+1,y_val);
    end
    if validgridposition(x_val,y_val-1,Gx,Gy)
      down = bestpol(x_val,y_val-1);
    end
    if validgridposition(x_val,y_val+1,Gx,Gy)
      up = bestpol(x_val,y_val+1);
    end
    next_utes = [down up left right];
    [M,I] = max(next_utes);
```

```
    if pol_val < M
        if I == 1
            y_val = y_val-1;
        elseif I == 2
            y_val = y_val+1;
        elseif I ==3
            x_val = x_val -1;
        elseif I ==4
            x_val = x_val +1;
        end
        moves = [moves; x_val, y_val]
        pol_val = bestpol(x_val,y_val);
    end
end

cut_off = 100; % finite time horizon
opts.maxiterations=3; opts.tol=0.00001; opts.plotprogress=1;
% rewards can also be a function of the action u(x,a), so make a reward
% that accounts for this:
[value action] = MDPemDeterministicPolicy(p,repmat(U', A, 1), cut_off,
    S, A, gam,opts);
figure; imagesc(reshape(value,Gx,Gy)); colorbar; drawnow; title('EM
    deterministic policy value')
figure; bar3zcolor(reshape(valpot.table,Gx,Gy)); title('EM deterministic
    policy value')
```

---

1. In the deterministic case the optimal route for the plane is (1,13), (1,12), (1,11), (2,11), (3,11), (4,11), (5,11), (6,11), (7,11), (8,11), (9,11), (10,11), (11,11), (12,11), (13,11), (14,11), (15,11), (15,10), (15,9), (15,8), (15,7), (14,7), (13,7), (12,7), (11,7), (10,7), (9,7), (8,7), (7,7), (6,7), (5,7), (4,7), (4,6), (4,5), (4,4), (5,4), (6,4), (7,4), (8,4).

2. In the second part of the question the plane has a fault and only goes right with a probability of 0.9. In this case the optimal route for the plane is (1,13), (1,14), (2,14), (3,14), (4,14), (5,14), (6,14), (7,14), (8,14), (9,14), (10,14), (11,14), (12,14), (13,14), (15,14), (16,14), (16,13), (16,12), (16,11), (16,10), (16,9), (15,9), (15,8), (15,7), (14,7), (13,7), (12,7), (11,7), (10,7), (9,7), (8,7), (7,7), (6,7), (5,7), (4,7), (4,6), (4,5), (4,4), (5,4), (6,4), (7,4), (8,4).

The altered route feels intuitive. The plane begins next to a village with negative utility. In the deterministic case we are happy with a policy which for example chooses a path from (1,11) to (2,11), in the non-deterministic case this path would involve the extra risk of instead of reaching the intended (2,11) hitting the village at (1,12). Thus, the optimal policy in the non-deterministic case is to choose the wider path going around the outside of the villages. The policy values can be seen in the figure below.

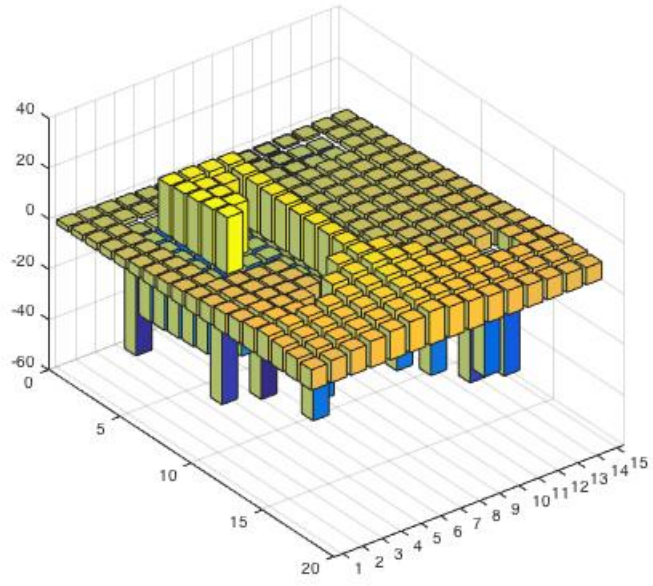

Figure 5: Policy Values - Plane starts at (1,13), highest utility (8,4) - non-deterministic

Problem 7.13

exerciseInvest.m contains the parameters for a simple investment problem in which the prices of two assets, a and b follow Markovian updating, as in section(7.8.3). The transition matrices of these are given, as is the end time T, initial wealth w_1, and initial price movements $\epsilon_1^a, \epsilon_1^b$ wealth and investment states.
We write a function `optdec(epsilonA1,epsilonB1,desired,T,w1,pars)` by representing the expected utility of a decision in terms of the Markov chain representing the joint states of all the variables. We are given transition distributions for the returns of product a and b $\epsilon_{1:T}^a$ and $\epsilon_{1:T}^b$. We then consider the distribution representing wealth at time t w_t which is a function of $\epsilon_t^a, \epsilon_t^b, w_{t-1}, d_{t-1}$ where d_{t-1} represents the decision taken at the previous timestep.

The transition matrix can be found by calculating the deterministic change in wealth given every state of $\epsilon_t^a, \epsilon_t^b, w_{t-1}, d_{t-1}$, where the state w_t is the nearest discrete state to the absolute wealth as follows:

$$w_t = w_{t-1}[d_{t-1}(1+\epsilon_t^a) + (1-d_{t-1})(1+\epsilon_t^b)]$$

Thus the distribution is such that $p(w_t|\epsilon_t^a, \epsilon_t^b, w_{t-1}, d_{t-1}) = 1$ when the above equation is true, and zero otherwise.

$u_{d_1=1.0}$	9,529.8
$u_{d_1=0.75}$	9,529.8
$u_{d_1=0.5}$	9,529.8
$u_{d_1=0.25}$	9,540.0
$u_{d_1=0}$	9,498.1

We then multiply the distribution of w_T by the utility function such that values of w higher than the `desired` variable return $10,000$ utility units.

Under utility maximising decision making, we can then perform message passing in the chain from $t = T : 2$, by repeatedly marginalising over the variables at t to form a function over the variables at $t-1$ and maximising the distribution over the decision variable d. This leaves us with the utility potential $U(d_1|\epsilon_1^a, \epsilon_1^b, w_1)$. As we are given the states of $\epsilon_1^a = 1, \epsilon_1^b = 1, w_1 = 1$, we find a utility potential in d_1:

Thus the decision is maximised when $d_1 = 0.25$. The following code shows the `optdec` function:

```
function [dec, val] = optdec(epsilonA, epsilonB, desired, T, w, pars)
    pot{epA(1, T)}.variables = 1;
    pot{epA(1, T)}.table = zeros(1:2);
    pot{epA(1, T)}.table(epsilonA) = 1;

    pot{epB(1, T)}.variables = T+1;
    pot{epB(1, T)}.table = zeros(1:3);
    pot{epB(1, T)}.table(epsilonB) = 1;

    pot{wealth(1, T)}.variables = 2*T+1;
    pot{wealth(1, T)}.table = zeros(26,1);
    pot{wealth(1, T)}.table(find(pars.WealthValue==w)) = 1;

    Utility = (pars.WealthValue>=desired)*10000;

    for t = 2:T %epsilon a
        tt = epA(t,T);
        pot{tt}.variables = [tt-1, tt];
        pot{tt}.table = pars.epsilonAtran;
    end
    for t = 2:T %epsilon b
        tt = epB(t,T);
        pot{tt}.variables = [tt-1, tt];
        pot{tt}.table = pars.epsilonBtran;
    end

    %generate the transition matrix based on the wealth formula
    transMat = zeros(2,3,26,26,5);
    for eA = 1:2
        for eB = 1:3
            for w0 = 1:26
```

```
            for d0 = 1:5
                epAState = pars.epsilonAval(eA);
                epBState = pars.epsilonBval(eB);
                dd = pars.DecisionValue(d0);
                wv0 = pars.WealthValue(w0);
                w2 = min(max(round(5*wv0*(dd * (1 + epAState) + (1-dd)
                    * (1 + epBState)))+1,1),26);
                transMat(eA, eB, w0, w2, d0) = 1;
            end
        end
    end
end

for t = 2:T %wealth
    tt = wealth(t, T);
    pot{tt}.variables = [epA(t,T), epB(t,T), tt-1, tt,
        decision(t-1,T)];%epsilonA t, epsilonB at t, wealth at t-1,
        decision at t-1

    pot{tt}.table = transMat;
end

pot = brml.setpotclass(pot, 'array');

% multiply utility to first wealth T
for i = 1:26
    pot{wealth(T, T)}.table(:,:,:,i,:) = pot{wealth(T,
        T)}.table(:,:,:,i,:) * Utility(i);
end

 %utility, wealth, epsilona t, epsilonb t
 maxpot = brml.const(1);
for t=T:-1:3 %stop to retrieve the utility potential u_1
    tomult = pot([epA(t,T), epB(t,T), wealth(t, T)]);
    tomult{4} = maxpot;
    joint = brml.multpots(tomult); %make joint
    sumPot = brml.sumpot(joint, [epA(t,T), epB(t,T), wealth(t,T)]);
        %sum over vars
    [maxpot, blah] = brml.maxpot(sumPot, decision(t-1, T));

end
t = t - 1;
tomult = pot([epA(t,T), epB(t,T), wealth(t, T)]);
tomult{4} = maxpot;
joint = brml.multpots(tomult); %make joint
sumPot = brml.sumpot(joint, [epA(t,T), epB(t,T), wealth(t,T)]); %sum
    over vars
vals =
    reshape(sumPot.table(epsilonA,epsilonB,find(pars.WealthValue==w),:),5,1);
[val, dec] = max(vals);
```

```
end

function r=epA(n, T) %generate pot index for epsilon A
r = n;
end

function r=epB(n, T) %generate pot index for epsilon B
r = n + T;
end

function r=wealth(n, T) %generate pot index for wealth
r = n + 2*T;
end

function r=decision(n, T) %generate pot index for decision
r = n + 3*T;
end
```