Learning and Planning in Dynamic Systems (046194)

HW 1

Students : Misbah Ramadan 305115149

Oron Anschel 038060398

$\boxed{100}$

Question 1:

a.

$$\Psi_1(2) = 1 \;\; ; \;\; 2 = 2$$

$$\Psi_2(4) = 3 \;\; ; \;\; 4 = 1+3 = 3+1 = 2+2$$

$$\Psi_3(2) = 0 \;\; ;$$ ✓

$$\Psi_N(N) = 1; \;\; N = \underbrace{1+1+\ldots+1}_{N-times}$$

$$\Psi_1(X) = 1 \;\; ; \;\; X = X$$

b. $$\Psi_N(X) = \sum_{i=1}^{X-1} \Psi_{N-1}(i)$$

c.

```
function [t] = psiNX(N,X)

r=zeros(N,X);

for i=1:N
    for j=1:X
       if i==1
           r(i,j)=1;
       elseif i>j
           r(i,j)=0;
       elseif i==j
           r(i,j)=1;
       else
           r(i,j)=r(i-1,j-1)+r(i,j-1);
       end
    end
end

t=r(N,X);

end
```
✓

The matlab code will be found in Appendix A in the end.

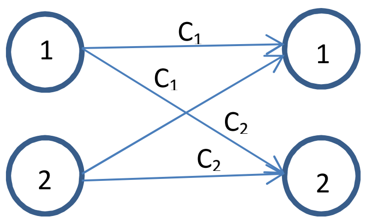
Explanation: In order to solve the recursive equation from previous part of the question, I used a matrix of dimensions N-on-X (N rows vs. X columns). Now we can notice that the recursive equation re-use a lot of the calculated values, so in order to not waste time calculating each one from the beginning we will use the calculated ones from previous run. Each place in the matrix represents the $\Psi_i(j)$. So it is pretty obvious from part a of the question that for all i==1 we will have 1 option and for all i>j we will have zero options and for i==j we will have only a single option which is i-times ones (the same as j-times ones). And in the general case each location in the matrix is the sum of left location and the upper left location. Let us see this in equations:

$$\Psi_N(X)=\sum_{i=1}^{X-1}\Psi_{N-1}(i)=\Psi_{N-1}(X-1)+\sum_{i=1}^{X-2}\Psi_{N-1}(i)=\Psi_{N-1}(X-1)+\Psi_N(X-1)$$

For that the memory complexity of the algorithm is the matrix size which is O(NX). And the time complexity is also O(NX) since we go over on all the columns and rows and for each we make a finite number of operations.

$$\Psi_{10}(1000)=2.6341*10^{21}$$ ✓

d. The idea is to have N-stages before we reach our goal, which is the sum of the numbers in the N-stages, which we demand to be X.
Each stage should include numbers, 1 to X-1 with arrows to all numbers in the following stage, while each arrow will have a cost which indicates the cost of the number we are going to.
For example :



Now, one important thing we should notice here, which is to make sure that we get combination of exactly N numbers. That means we shouldn't stop before going through exactly the N-stages, not more not less.
In order to do so we will define:
State space : $s_k=\{L_k,j\}$, where L_k represents what is left from X (depends on the decisions on previous k stages)

Action space: $A_k(s_k)=\left\{a_k\in\{1,\dots,X-1\}\,|\,(L_k-a_k)\geq\underbrace{(X-(k+1))}_{number\ of\ stages\ left}\right\}$

In words, after we reach for a certain number in stage k+1, we need to update L_k and make sure we still have enough stages to reach our goal and we didn't reach the goal earlier.

Transition function : $f_k(s_k,a_k)=\{L_{k+1}=L_k-a_k,\,j=a_k\}$

Cumulative cost function : $c(s_k,a_k)=c_{a_k},C_N=\sum_{k=1}^{N}c_{a_k}$.

In order to bound the complexity, at each stage (and we have N stages) we can choose numbers from 1 to X-1. To take the worst case scenario that we pick always the number 1, but in each stage we will have to choose between 1 to X-k number left (assuming N==X). In that case ✓

$\sum_{i=1}^{N}((X-i)\bullet\underbrace{(X-i-1)}_{the\ number\ of\ arrow\ going\ out\ from\ the\ s_k})\leq\sum_{i=1}^{N}(X-i)^2=\sum_{i=1}^{N}(i)^2=O\left(NX^2\right)$. That means we are

bounded with $O(NX^2)$ operations !
For the memory complexity we need to represent all the states in all stages which are bounded by $O(NX)$ (Note: that the arrows are not considered in the memory complexity since a representation by an array aside, which also represents the cost function, can be done. And that will add X to the complexity but $NX+X=O(NX)$).

Question 2:

a. P('Bob')= $P(B|-)\bullet P(o|b)\bullet P(b|o)\bullet P(-|b)=1\bullet 0.25\bullet 0.2\bullet 0.125=6.25\bullet 10^{-3}$

P('Koko')= $P(K|-)=0$

P('B')= $P(B|-)\bullet P(-|B)=1\bullet 0.125=0.125$

P('Bokk')= $P(B|-)\bullet P(o|B)\bullet P(k|o)\bullet \underbrace{P(k|k)}_{0}\bullet P(-|k)=0$ ✓

P('Boooo')= $P(B|-)\bullet P(o|B)\bullet P(o|o)^3\bullet P(-|o)=1\bullet 0.25\bullet 0.2^3\bullet 0.4=8\bullet 10^{-4}$

b. In order to formulate the problem as a finite horizon decision problem, we will choose the state space to be {'B','K','O'} **except** for the first and last stage. First stage will be the state {'B'}, since it is only possible to start words with the letter B, now the following k-1 stages will include the states {'B','K','O'} with arrows from each state to all states in the following stage. That means after starting in the letter B, we can choose one of 3 letters for the next letter and so on. Notice that the character '-', which means the end of a word is not included in the state space in the first K stages, since we are looking for words of length K.
Summary: First state will be B, next K-1 stages will include states {'B','K','O'} and last stage will include only '-'.
State space : $S_K=\{'B','K','O'\}$ ✓
Action space : $A_K=\{'B','K','O'\}$
Transition function : $f_k(s_k,a_k)=a_k$

Now the cost function will be defined $c_k(s_k,a_k)=\frac{1}{P\left(l_{a_k}|l_{s_k}\right)}$ and the cumulative cost function will also need to take into account the last transition to '-'.
$C=\prod_{i=1}^{K-1}c_i(s_i,a_i)\bullet c(s_K,-)$.

Complexity: Similar to previous question we will have to go over K+1 stages and in K-2 stages (not including the first and the last) we will have to check all the arrows that means 3*3 and on the first stage (if we are going from the end to the beginning) we need to check the 3 arrows. That will give us a total of ✓
$3+\underbrace{3\bullet 3+\cdots+3\bullet 3}_{K-2}+3=6+9(K-1)=O(K)$ in the time complexity and another O(K) in memory since we have to hold 3(K-1)+2 states plus the 16 which is the size of the matrix but in total we will get also a memory complexity of O(K).

Reduction: in order to do a reduction to an additive cost function we will use the 2nd base logarithm. If we apply the $\log_2$ on the $\frac{1}{P}$ (the probability matrix) then the cost function will turn to be $C = \sum_{i=1}^{K-1} \log_2(c(s_i, a_i)) + \log_2(c(s_K, -))$. That way we can use an additive cost function instead of a multiplicative cost function.

Multiplicative cost function vs. Additive cost function: ✓

Let us discuss two cases:

1) The most probable word of length 10 : in that case a multiplicative function is preferable since doing the log operation my make some approximations, hence we might not get the right answer or it will not be worth to do the manipulation!

2) The most probable word of length 100 : in that case multiplying 100 probabilities might get us a very small number that it might be counted as zero. For example if using matlab it might approximate the number to zero. But if we use the log operation it will solve this problem and summing up a 100 numbers will get us much closer to where we want. ✓

To sum it up, depends on the length we can decide what cost function is preferable. Or just pay the overhead in advance and use the log !

MATLAB:

The most probable word of length 10 is : "BBBBBBBBBO"

The MATLAB code will be found in Appendix B at the end.

Question 3:

a. In order to formulate the problem as a finite horizon decision problem we will define the state space to represent all the possible locations onboard. That means $S_K = \{(i,j) \mid i \le M, j \le N\}$, in words each state will be represented by the coordinate of the mouse's location. The actin space will include two operations up or down, formally we can define the action space as

$$A_k(s_k) = \left\{ a_k \in \{UP, RIGHT\} \mid \underbrace{i < M, j < N}_{\text{depends on the decision}} \right\}$$

$$f_k(s_k, a_k) = \{s_{k+1} = (i+1, j) \mid a_k = UP \text{ or } s_{k+1} = (i, j+1) \mid a_k = RIGHT\}$$

and the cumulative cost function will be defined as 1 if there is a cheese in the location we get to by taking the action a_k and 0 otherwise.

✓

b. The horizon of the problem is M+N-2 and that is because eventually we will have to make M-1 actions UP and N-1 actions Right in order to get to the right top corner. Now the Algorithm will have to check all possible locations and for that in the case of M=2 and N the complexity will be O(2N)=O(N) and for M=N it will behave as O(N^2). However the horizon will be N in the first case and 2(N-1) in the second case.

✓

c.

1) If both mice ignored each other one of the mice might end up not getting any cheese. And that is because if both mice went according to the optimal path one will get the cheese before the other, and in that scenario another optimal path might be available !

✓

2) By defining the state as the coordinate of the mouse location, now we will have to hold two locations and each state will have to hold the location of the two mice. Not that will lead to a bigger number of states which will be $(MN)^2$ and the actions will be one of three possibilities, both goes up, both goes down, one goes up and one goes down. (Note that here we didn't differ between the mice).

✓

3) According to the same explanation as above, here we will have $(MN)^k$ states and the actions will be k+1 actions now. We can see that the problem is growing exponentially in the number of mice.

<u>Question 4:</u>

a. To formulate a dynamic programming algorithm for this problem we need to define the state space, action space, and the value function. As it was done in lecture 2, we will follow the Algorithm 1: Finite-horizon Dynamic Programming.

(i) We will initialize the value function : $V_N(s) = r_N(s_N)$ ✓

(ii) *Backward recursion* : For k=N-1,…,0, compute

$$V_k(s) = \max_{a_k \in A_k}\left\{\min_{b_k \in B_k}\left\{r_k(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right)\right\}\right\}, s \in S_k$$

(iii) *Optimal Policy*: Choose any control policy π_{a_k} that satisfies

$$\pi^*_{a_k}(s) \in \arg\max_{a_k \in A_k}\left\{\min_{b_k \in B_k}\left\{r_k(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right)\right\}\right\}, k = 0, \dots, N-1$$

The value function will represent what is the maximum *worst-case* reward at stage k. In other words, if we are at s_k then V_k will represent what is the maximum worst-case reward we will get if we continued to the end.

b. Assuming we have N stages $|S_k|=S$ states and $|A_k|=A$ actions with $|B_k|=B$ rival actions. With the definition above, and similar to the analysis we did in the class we will get that the computational complexity is $O(N \bullet S \bullet A \bullet B)$. ✓

c. Yes ! In order to use this approach to solve the tic-tac-toe game we will have to define the board status. That means at every step each rectangle on the board could be either empty, occupied with X or occupied with O. With this approach we will define the state space to include all the board statuses that can be with an indication if this is a status where the game ends (either in winning of one of the opponents, or a draw). If we ignore all cases with probabilities of the current move, we can define three rewards 0,1,-1. A move that doesn't end the game will have a reward 0, a move that will cause winning will have a reward of 1 and all other moves afterwards will have reward 1 and if the move causes a loss of the game then it will have a reward -1 and all moves afterwards will have reward -1. The action will be selecting an empty location on the board, and the transition function should update the status by taking us to the right state (which as explained before, will indicate the board's status). ✓

d. No, and this is because if the complexity of the problem. If we try to use the same approach like we used in tic-tac-toe to solve chess it will be nearly impossible. Assuming the game will end (and we can't just keep doing "stupid moves") in order to define it as a finite horizon decision problem, then the number of states can be something like $64^{\underbrace{2}_{black/white} \bullet (\underbrace{1}_{Rook} + \underbrace{1}_{Knight} + \underbrace{1}_{Bishop} + \underbrace{1}_{King} + \underbrace{1}_{Queen} + \underbrace{1}_{Pawn})}$ and that is already a huge number of states that we can't hold ! So an alternative approach should be considered.

✓

Appendix A:

```
%% Learning and Planning in Dynamic Systems
%% HomeWork 1, Question 1

function [t] = psiNX(N,X)

r=zeros(N,X);

for i=1:N
    for j=1:X
       if i==1
           r(i,j)=1;
       elseif i>j
           r(i,j)=0;
       elseif i==j
           r(i,j)=1;
       else
           r(i,j)=r(i-1,j-1)+r(i,j-1);
       end
    end
end

t=r(N,X);

end
```

Appendix B:

```
%% Question 2
% P=[0.5,0.125,0.25,0.125;0.4,0,0.4,0.2;0.2,0.2,0.2,0.4;1,0,0,0];
function [word] = mostProbableWord(K,P)
C=log(1./P); % the updated cost funcion
hist=[char('B'),char('K'),char('O')];
charWord=zeros(1,K);
Vk=inf(3,K); %1=b,2=k,3=o

for stage=K:-1:1
    if stage==K %that means we are at the stage before '-'
        for letter=1:3
            Vk(letter,K)=C(letter,4);
        end
    elseif stage~=1 && stage~=K
        for letter=1:3
            Vk(letter,stage)=min(C(letter,1:3)+Vk(1:3,stage+1)');
        end
    else % means stage=1
        Vk(1,stage)=min(C(1,1:3)+Vk(1:3,stage+1)');
        %the other two letters in the first stage are ignored !
    end
end

charWord(1)=hist(1);

for stage=2:K
    if stage==2
        [p,index]=min(C(1,1:3)+Vk(1:3,stage)');
        charWord(stage)=hist(index);
        prev_index=index;
    else
        [p,index]=min(C(prev_index,1:3)+Vk(1:3,stage)');
        charWord(stage)=hist(index);
        prev_index=index;
    end
end

word=string(charWord);

end
```

✓