הטכניון, מכון טכנולוגי לישראל

הפקולטה להנדסת חשמל

למידה במערכות דינאמיות

046005

סמסטר אביב תש"פ

תרגיל בית: מספר 1.

תאריך הגשה: 20/4/2020.

מגישים:

עידו יחזקאל	204397368
אביב רצון	203534573

Question 1 - longest increasing odd even subsequence

a) Like in the lecture lets define:
L_j : length of the longets increasing odd even subsequence ending at index j .
The recession formula is like the one in the lecture but with one more condition:

$$L_j = \begin{cases} \max\{L_i\} + 1 \ s.t. \ i < j \wedge a_i < a_j \wedge a_i \bmod 2 \neq a_j \bmod 2 \\ 1 \text{ otherwise} \end{cases}$$

$\mathrm{L}_1 = 1$

Algorithm:

```
1. Initiate array of size n called temp
2. temp[1]=1
3. For each j from 1 to n:
    i. For each i from 1 to j:
        1. If a[i] < a[j] and a[i]%2 != a[j]%2 and temp[i] < temp[j] +1:
            a. temp[i] = temp[j] +1
4. return the maximum of the array temp.
```

b) As one can see the DP algorithm includes for loop inside a for loop when each iteration includes $O(1)$ calculations so the time complexity is $O(n^2)$ and the space complexity is $O(n)$.
Comparing to the naïve approach which check all options which is $O(2^n)$ time complexity and $O(1)$ space complexity.

Question 2 – Counting

a)

$\Psi_1(2)=1$

$\Psi_2(4)=3$

$\Psi_3(2)=0$

$\Psi_N(N)=1$

$\Psi_1(X)=1$

b) $$\Psi_N(X)=\begin{cases}\sum_{i=1}^{x-1}\Psi_{N-1}(X-i) \text{ s.t. } \mathrm{N}\le \mathrm{X} \\ 0 \text{ otherwise}\end{cases}=\begin{cases}\Psi_N(X-1)+\Psi_{N-1}(X-1) \text{s.t. } \mathrm{N}\le \mathrm{X} \\ 0 \text{ otherwise}\end{cases}$$

$\Psi_1(X)=1$

c)

```
import numpy as np

N = 12
X = 800

interRes = np.zeros((N + 1, X + 1))

for x in range(1, X + 1):
    interRes[1, x] = 1

for n in range(2, N + 1):
    for x in range(1, X + 1):
        interRes[n, x] = interRes[n, x - 1] + interRes[n - 1, x - 1]

print(int(interRes[12, 800]))
```

Time complexity is $O(NX)$ and space complex is $O(NX)$.

$\Psi_{12}(800)=1980760285431223048732672$.

d)

1. The problem as a finite horizon decision problem:

   State Space is:

   S_k = all the optional sums we can get at step k when sum is lower than X

   .

   For example: $S_0=0; S_N=X$ and if we choose to accumulate the number 2 after one step so the system sate is $s_1=2$

   Action Space is:

   A_t = all the optional numbers to add the current sum without exceeding X

Cumulative cost function:

$C_N(h_N) = \sum_{k=0}^{N-1} c_k(s_k, a_k) + C_N(S_N)$ where the cost of each step is the cost of the action which is determined by the cost of digit we choose.

2. The bounded complexity is:

   There is N stages.

   Each stage contains at most X-1 states.

   Each stage contains at most X-1 actions.

   Therefore, as seen in lecture: $O(NX^2)$

Question 3 – Language model:

a)

$$P(Bob) = 0.25\cdot 0.2\cdot 0.325 = 0.01625$$
$$P(Koko) = 0$$
$$P(B) = 0.325$$
$$P(Bokk) = 0.25\cdot 0.2\cdot 0\cdot 0.2 = 0$$
$$P(Boooo) = 0.25\cdot 0.2\cdot 0.2\cdot 0.2\cdot 0.4 = 0.0008$$

b)

1. The problem as a finite horizon decision problem:
   State Space is:
   $S_t =$ all the optional words we can get at step t when length is smaller than K
   For example: $S_0 = \{B\}; S_1 = \{BB, BK, BO\}$.
   Action Space is:
   $$A_t = \{B, K, O\} \forall t < K-1$$
   $$A_{K-1} = \{-\}$$
   Cumulative cost function:
   $C_K(h_K) = C_N(S_N) \bullet \prod_{i=0}^{k-1} C_i(s_i, a_i)$ where the cost of each step is the probability of the action for example, the cost of the action:
   $$s_0 = B \rightarrow s_1 = BO$$
   $$c = 0.2$$
2. The bounded complexity is:
   There is K stages.
   Each stage contains at most 3 states.
   Each stage contains at most 3 actions.
   Therefore, as seen in lecture: $O(K)$
3. Instead of using multiplexing in order to calculate the cumulative probability we can use the log function propriety of:
   $\log(a\cdot b) = \log(a) + \log(b)$. And because the log function is a monotonically increasing function the maximum reach at the same point of $a\cdot b$.
   so the new additive function is
   $C_K(h_K) = \sum_{i=0}^{k-1} \log(C_i(s_i, a_i)) + \log(C_N(S_N))$.

4. We think the multiplicative is preferred because when applying the log function on numbers smaller than 1 the result is negative and because the representation of negative numbers requires more memory and more calculations so the time constants would be bigger than those without using it.

```python
import numpy as np


def return_index(l) -> int:
    translate = {"B": 0, "K": 1, "O": 2, "-": 3}
    return translate[l]


def return_letter(index) -> str:
    translate = {0: "B", 1: "K", 2: "O", 3: "-"}
    return translate[index]


def return_probs(l_t_1, l_t) -> float:
    translate = {"B": 0, "K": 1, "O": 2, "-": 3}

    probabilities = np.array([
        [0.1, 0.325, 0.25, 0.325],
        [0.4, 0, 0.4, 0.2],
        [0.2, 0.2, 0.2, 0.4],
        [1, 0, 0, 0]])

    return probabilities[return_index(l_t), return_index(l_t_1)]


L = 5  # for length

interRes = np.ndarray((L + 2, 4), dtype=object)
interRes[1, return_index("B")] = (1, "B")
interRes[1, return_index("K")] = (0, "K")
interRes[1, return_index("O")] = (0, "O")
interRes[1, return_index("-")] = (0, "-")

for l in range(2, L + 2):
    for curr in range(0, 4):
        trans2curr = [
            return_probs(return_letter(curr), "B") * interRes[l - 1, return_index("B")][0],
            return_probs(return_letter(curr), "K") * interRes[l - 1, return_index("K")][0],
            return_probs(return_letter(curr), "O") * interRes[l - 1, return_index("O")][0],
        ]

        max_index = np.argmax(trans2curr)

        interRes[l, curr] = (max(trans2curr), interRes[l - 1, max_index][1] + return_letter(curr))

print(interRes[6, 3])
```

The most probable word of size 5 is "BKBKO" with probability of 0.00676.

Question 4 – LCS of Three Strings:

i) The brute force solution is the calculate all subsequences of all the three strings and while runtime insert them into a database like a Trie tree, then choose the longest subsequence, we can also do it in run time by holding the current longest one and compare to its length. Therefore, this to evaluate all the options so it will be $O\left(2^m\right)$ where m is the length of the longest input string.

ii) let $c[i,j,k]$ denote the length of the LCS of X_i, Y_j Z_k.

$$c[i,j,k]=\begin{cases}0 \text{ if } i=0\vee j=0\vee k=0\\ c[i-1,j-1,k-1]+1 \text{ if } i,j,k>0\wedge x_i=y_j=z_k\\ \max\left\{c[i-1,j,k],c[i,j-1,k],c[i,j,k-1]\right\} \text{ if } i,j,k>0\wedge not\left(x_i=y_j=z_k\right)\end{cases}$$

$O(mnl)$ where m is the length of X and n is the length of Y and l is the length of Z.

iii)

```
m = X.length
n = Y.length
l = Z.length
let lcs[0…m,0…n,0…l] be new 3D matrix initialized with empty strings.
let temp[0…m,0…n,0…l] be new 3D matrix initialized with zeros.
for i = 1 to m:
        for j = 1 to n:
                for k = 1 to l:
                        if X[i] == Y[j] == Z[k]:
                                temp[i, j, k] = 1 + temp[i - 1, j - 1, k - 1];
                                lcs[i,j,k] = lcs[i - 1, j - 1, k - 1] + X[i]
                        elseif temp[i - 1, j , k] > temp[i, j-1, k] and temp[i - 1, j , k] > temp[i, j, k-1]:
                                temp[i, j, k] = temp[i - 1, j , k ];
                                lcs[i,j,k] = lcs[i - 1, j , k ];
                        elseif temp[i , j -1, k] > temp[i-1, j, k] and temp[i , j -1 , k] > temp[i, j, k-1]:
                                temp[i, j, k] = temp[i, j-1 , k ];
                                lcs[i,j,k] = lcs[i , j -1, k ];
                        else:
                                temp[i, j, k] = temp[i, j , k -1];
                                lcs[i,j,k] = lcs[i , j, k -1];
return lcs[m,n,l] and temp[m,n,l]
```

Question 5-MinMax dynamic programming:

a. The value function is
$V_k(s) = \max_{a \in A_k}\left(\min_{b \in B_k}\left(r_k(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right)\right)\right)$.
The motivation behind the value function is we would like to maximize the worst case therefore, the worst case is when our opponent choose to minimize the reward **given our chosen action** a_k at state s_k .So for a given action a_k we calculating the opponent worst action for us and after that we choose the maximum form all the possible action, in that way we can insure a lower bound to our achieved reward.

<u>Algorithm</u>:
```
1. let V[0…N] be an array.
2. let $\pi$ [0…N] be an array of actions.
3. $V_N(s) = r_N(s) \forall s \in S_N$
4. for k = N – 1 to 0:
        $V_k(s) = \max_{a \in A_k}\left(\min_{b \in B_k}\left(r_k(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right)\right)\right)$
        $\pi_k(s) = \arg\max_{a \in A_k}\left(\min_{b \in B_k}\left(r_k(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right)\right)\right)$
5. return $\pi$
```

b. The number of possible stages is N.
The number of possible states in each time is n.
The number of possible actions in player A in each time is A.
The number of possible actions in player B in each time is B.
There is a total of $n \cdot N$ states (excluding the final one), and for each one we need $A \cdot B$ computations, therefore, $O(A \cdot B \cdot n \cdot N)$.

c. Yes, because the number of states and action is still reasonable, moreover, there is at most 9 turns in the game. Let's formulate:
The states should be the state of the gameboard **after both players turns**.
For example: S_0 contains only the empty gameboard.
S_1 contains the gameboard after the two players turn, and so on.
The actions of players A and B is to choose location from all the empty locations.
The reward can be 0 during the game and only in the final states it should be -1 for player A loss, 0 for a tie and 1 for player A win.(Also we can use heuristic to evaluate players actions in order to promise player A winning faster).

d. No, we can't use this approach to solve the game of Chess because the actions space, state space and number of stages is much bigger than in Tic Tac Toe therefore the memory consumption is much bigger, so this is not feasible.