IFT2125 Devoir 4

April 4, 2018

Question 1

On suppose que w est composé de N characters, i.e. $w = x_1, ..., x_n$. Par exemple, si $w =$ *helloworld* alors on a $w = x_1...x_9$ avec $x_1 = h$,...,$x_{10} = d$.

On illustre l'algorithme avec l'exemple *helloworld*. On voit qu'il y a plusieurs façon de séparer la liste de lettres en mots autres que la phrase 'hello world'. Par exemple,

he l **low** **or** ld

où l et ld auraient probablement des fréquences égale à 0. Supposons aussi que le score de chaque mot est comme suit:

w	he	hell	hello	low	or	world
$f(w)$	4	1	5	2	3	5

de sorte que la segmentation optimale est w_1 =hello, w_2 =world. On construit une un vecteur $T[i]$ où chaque élément correspond au score de la segmentation optimale avec i lettres. La Figure 1 montre la table T pour l'exemple *helloworld*.
Toutes les fois qu'on ajoute une lettre on doit verifier si un nouveaux mots est formé. Par exemple, quand on passe de *hel* à *hell* il faut comparer les scores des combinaisons (he, l, l), (he, ll), (h, ell) et (hell). À l'étape suivante on devra comparer les scores (he, l, l, o), (he, l, lo), (he, llo), (h, ello) et (hello). On remarque qu'à toutes les étapes on utilise le score des combinaisons des segmentations precedentes en concatenant le reste des lettres. Par exemple: ((he, l, l), o), ((he, l), l·o), ((he), l·l·o),...Le pseudocode pour cet algorithm est comme suit:

# lettres	0	1	2	3	4	5	6	7	8	9	10	(w_i)
	0											( )
h		0										(h)
he			4									(he)
hel				4								(he, l)
hell					4							(he, l, l)
hello						5						(hello)
hellow							6					(he, l, low)
hellowo								6				(he, l, low, o)
hellowor									6			(he, l, low, or)
helloworl										9		(he, l, low, or, l)
helloworld											10	(hello, world)

Figure 1: Segmentation avec programmation dynamique

Input: une suite de lettres $w = x_1...x_n$
Output: la segmentation optimale de w donnée
sous forme de liste seg_opt=$[w_1, ..., w_k]$

```
segmentations := [ ]
initializer un vecteur nul T de taille n
for i := 1 to n do
    scores := [ ]
    cat_list := [ ]
    for j := 1 to i do
        cat := x_j · x_{j+1} ··· x_i
        cat_list.append(cat)
        scores.append(T[j − 1] + f(cat))
    end
    bestscore := max(scores)
    T[i] := bestscore
    ix_bestscore := argmax(scores)
    new_seg := [segmentations[ix_bestscore], cat_list[ix_bestscore]]
    segmentations.append(new_seg)
end
seg_opt = segmentations[n]
return seg_opt
```

Algorithm 1: Segementation en programmation dynamique

Question 2

Le nombre maximum de chefs qui peuvent acceder a la salle au temps i sachant que la batterie a charger pendant j heures est égale a $\min(c(j), n_i)$. On note aussi que le si la capacité au temps j est égale à $c(j)$, alors la batterie était en charge durant les j heures precedentes et la pile était utilisé à l'heure $i - j - 1$. Le nombre de chefs ayant accedé à la salle au temps i avec un nombre d'heures de charge j est donc le nombre de chefs parmis les n_i qui peut acceder sachant que la capacité est $c(j)$ plus le nombre de nombre de chefs qui on pu acceder avant l'heure $i - j$.

Pour résoudre ce probleme, on construit une table $V_{i,j}$ de taille $k \times k$ ou les lignes representent l'heure et les colonnes les capacités possible de la batterie. On peut trouver le nombre maximum de chefs en utilisant la récurrence suivante

$$V_{i,j} = \begin{cases} \min(c(j), n_i) + \max V_{i-j-1,\cdot}, & \text{si } i - j > 1 \\ \min(c(j), n_i), & \text{sinon} \end{cases} \tag{1}$$

où $V_{i-j-1,\cdot}$ est la ligne $i - j - 1$ de la table V. La relation peut mener à un algorithm pour trouver le nombre maximum de chefs pouvant être admis. Le nombre maximum est donné par par le maximum de la ligne k. La figure 2 illustre comment l'algorithme fonctionne pour un exemple tres simple avec les données suivantes:

i	0	1	2	3	4	5
n_i	0	1	0	3	2	3

et en supposant que la fonction c est linéaire, i.e. $c(j) = j$.
Les calculs pour construire la table de la figure 2 sont montré à l'équation 2. Après avoir remplis la table, on doit retrouver l'horaire optimal. Soit $\mathrm{i}_1, ..., \mathrm{i}_l$ les heures d'utilisations de la pile. Si le maximum de la ligne k est en $V[k, 0]$, alors la derniere heure d'utilisation est $\mathrm{i}_l = k - 1$, et sinon la dernière heure d'utilisation est $\mathrm{i}_l = k$. Ensuite soit j_l l'indice maximum de la ligne i_l (la colonne). Alors $\mathrm{i}_{l-1} = \mathrm{i}_l - \mathrm{j}_l - 1$. Le maximum de la ligne i_{l-1} se trouve à la colonne j_{l-1} et de façon similaire on a que $\mathrm{i}_{l-2} = \mathrm{i}_{l-1} - \mathrm{j}_{l-1} - 1$. On peut utiliser cette relation de récurrence pour trouver tous les heures ou la pile est utilisé. On procède jusqu'à ce que $\mathrm{i}_{l-h} - \mathrm{j}_{l-h} - 1 = 0$.

	$c(0)$	$c(1)$	$c(2)$	$c(3)$	$c(4)$	$c(5)$	$\max V_{i,\cdot}$
0	0						
1	0	1					
2	1	0	0				
3	1	2	2	3			
4	3	2	3	2	2		
5	3	**4**	3	**4**	3	3	

Figure 2: Exemple d'application de l'algorithm `organize`$(n_1, ..., n_k)$

.

$$
\begin{aligned}
V_{1,1} &= \min(c(1), n_1) = 1 \\
\\
V_{2,0} &= \min(c(0), n_2) + \max V_{1,\cdot} = 1 \\
V_{2,1} &= \min(c(1), n_2) = 0 \\
V_{2,2} &= \min(c(2), n_2) = 0 \\
\\
V_{3,0} &= \min(c(0), n_3) + \max V_{2,\cdot} = 1 \\
V_{3,1} &= \min(c(1), n_3) + \max V_{1,\cdot} = 2 \\
V_{3,2} &= \min(c(2), n_3) = 2 \\
V_{3,3} &= \min(c(3), n_3) = 3 \\
\\
V_{4,0} &= \min(c(0), n_4) + \max V_{3,\cdot} = 3 \\
V_{4,1} &= \min(c(1), n_4) + \max V_{2,\cdot} = 2 \\
V_{4,2} &= \min(c(2), n_4) + \max V_{1,\cdot} = 3 \\
V_{4,3} &= \min(c(3), n_4) = 2 \\
V_{4,4} &= \min(c(4), n_4) = 2 \\
\\
V_{5,0} &= \min(c(0), n_5) + \max V_{4,\cdot} = 3 \\
V_{5,1} &= \min(c(1), n_5) + \max V_{3,\cdot} = 4 \\
V_{5,2} &= \min(c(2), n_5) + \max V_{2,\cdot} = 3 \\
V_{5,3} &= \min(c(3), n_5) + \max V_{1,\cdot} = 4 \\
V_{5,4} &= \min(c(4), n_5) = 3 \\
V_{5,5} &= \min(c(5), n_5) = 3
\end{aligned}
\tag{2}
$$

Soit `organize`$(n_1, ..., n_k)$ la fonction qui retourne les moments optimaux ou la pile devrait etre utilisée. Le pseudocode de cet algorithm peut etre ecrit comme à l'algorithm 2

```
Function organize(n_1, ...n_k):
    Input: n_1, ..., n_k
    Output: liste T contenant les moments ou la pile devrait être
    solicité pour maximiser N
    initializer une matrice nulle V[i, j] de taille k × k
    initializer T := [ ]
    for i := 1 to n do
        for j := 1 to i do
            if j − i > 1 then
                V[i, j] := min(c(j), n_i) + max V[i − j − 1, :]
            else
                V[i, j] := min(c(j), n_i)
            end
        end
    end
    if arg max V[k, :] = 0 then
        i := k − 1
    else
        i := k
    end
    while i > 0 do
        T.append(i)
        j := arg max V[i, :]
        i := i − j − 1
    end
    return sort(T, order=increasing)
end
```

Algorithm 2: Solicitation optimale de la pile (question 2)