Vendredi 16 Janvier 2004

Démo 1 – Programmation Dynamique

Jean-Eudes Duchesne

1. Programmation dynamique

Fonction récursive : une fonction récursive est une fonction qui s'appelle elle-même dans le but de se résoudre. Une fonction récursive a besoin des conditions de base ou de conditions d'arrêts pour être définie.

Diviser pour régner / Programmation dynamique : Diviser pour régner et la programmation dynamique sont deux techniques de programmation qui utilise la puissance d'expression des fonctions récursives pour se résoudre. Quand un appel de fonction réduit le problème à un sous-domaine unique du domaine de solution de départ, on utilise l'approche diviser pour régner (ex : Tri d'un tableau ou la recherche dichotomique). Quand un appel récursif ne réduit pas l'ensemble solution à un domaine unique et est susceptible d'être recalculé par un autre appel récursif, alors on utilise plutôt la programmation dynamique.

Ex : Nombres de Fibonacci (Prog. Dyn.)

fib(0) = 1
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2) pour n > 1

fib(3) --> return fib(2) + **fib(1)**
| |___ return 1
|___ return **fib(1)** + fib(0)
| |___ return 1
|___ return 1

fib(5) -->
return fib(4) + fib(3)
| |___ return **fib(2)** + fib(1)
| | |____ return 1
| |___ return fib(1) + fib(0)
| | |___ return 1
| |___ return 1
|___ return fib(3) + **fib(2)**
| |___ return fib(1) + fib(0)
| | |___ return 1
| |___ return 1
|___ return **fib(2)** + fib(1)
| |___ return 1
|___ return fib(1) + fib(0)
| |___ return 1
|___ return 1

Notez comme dans l'arbre de solution de fib(3) et fib(5), certaines valeurs sont recalculées à plusieurs reprises. La programmation récursive n'arrive pas à solutionner fibonacci efficacement. Que faire ? Programmation dynamique ! Au lieu de recalculer les valeurs à chaque appel, une fois qu'une valeur est calculée, celle-ci est conservée dans une table pour pouvoir être réutilisée. Dans le cas de Fib(5) c'est tout comme si l'on parcourait seulement la branche extérieure de l'arbre.

Fibonacci, version récursive simple

```
PROCEDURE fib(n)
DEBUT
   SI n == 0 || n == 1 ALORS
      RETOURNER 1
   SINON
      RETOURNER fib(n-1) + fib(n-2)
FIN
```

Complexité (temps): $O(2^n)$

Fibonacci, version PD

```
A <- Tableau de taille n+1

PROCEDURE fib(n)
DEBUT
   SI A[n] = NULL alors
   DEBUT
      SI n == 0 || n == 1 ALORS
         A[1] = 1
      SINON
         A[n] = fib(n-1) + fib(n-2)
   FIN
   RETOURNER A[n]
FIN
```

Complexité (temps): $O(n)$

Ex : Fib version PD -> fib(5)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	3	5	8

2. Distance d'édition

Opérations : Substitution/Match (indentité), Insertion , Délétion

Pour comprendre les opérations entre S[1..m] et T[1..n], utilisons les caractères en position i et j.

Identité entre S[i] et T[j] : S'il y a identité ou substitution entre les caractères en position i et j, alors forcément, il faut que ces deux caractères soient alignés l'un à l'autre peut importe les autres caractères de S et T.

S	1 … i-1	i	I+1 … m
T	1 … j-1	j	j+1 … n

Insertion dans S : S'il y a insertion dans S, c'est tout comme si le caractère en position j existait dans T, mais non dans S, ce qui signifie que peu importe comment le caractère en position i est aligné, il doit l'être avec l'intervalle 1 … j-1.

S	1 … i	?	i+1 … m
T	1 … j-1	j	j+1 … n

Délétion dans S : S'il y a délétion dans S, c'est tout comme si le caractère en position i ne pouvait être aligné à aucun caractère de T. Ce qui signifie que pour modifier S à T il faudrait se débarrasser de ce caractère.

S	1 … i-1	i	i+1 … m
T	1 … j	?	j+1 … n

Règles :

Soit S[1…m] et T[1…n], la distance d'édition D(m,n) entre S et T est le nombre minimal d'opération qui transforme S à T (ou réciproquement).

Relation de récurrence :

```
D(i, j) = MIN [     D(i -1, j) + 1,          /* Délétion */
                    D(i, j-1) + 1,           /* Insertion */
                    D(i-1, j-1) + p(i, j) ]  /* Identité */
```

où p(i, j) = 0 si S[i] == T[j] et 1 sinon.

Conditions initiales :

```
D(i,0) = i,     i = 0 .. m
D(0,j) = j,     j = 0 .. n
```

Exemple :

D(i,j)	-	G	T	C	A	G	G	T
-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
C	**1**	1	2	2	3	4	5	6
A	**2**	2	2	3	2	3	4	5
T	**3**	3	2	3	3	3	4	4
A	**4**	4	3	3	3	4	4	5
G	**5**	4	4	4	4	3	4	5
T	**6**	5	4	5	5	4	4	4
G	**7**	6	5	5	6	5	5	5

Occurrence dans un texte à k erreurs :

Il est possible de modifier le problème de la distance d'édition à un problème de retrouver chaque occurrence d'un mot quelconque à k erreurs prêts dans une séquence. En effet, il suffit de changer les conditions initiales pour résoudre le nouveau problème. En imposant D(0, j) = 0 pour j de 0..n, l'on obtient le résultat escompté.

D(i,j)	-	G	T	C	A	G	G	T
-	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
C	**1**	1	1	0	1	1	1	1
A	**2**	2	2	3	0	1	2	2
T	**3**	3	2	3	1	1	2	2

Pour trouver les occurrences de S{CAT} dans T à k erreurs, il suffit de parcourir la dernière ligne de la table de programmation dynamique et de rapporter une occurrence à chaque fois qu'une valeur est plus petite ou égale à k. Par exemple, dans notre cas, pour k = 0 il n'y a aucune occurrence de CAT, mais pour un k=1, il y a 2 occurrences ; CA qui se termine en D(3,4) (avec délétion du T) et CAG qui se termine en position D(3,5) (avec substitution du G par un T).

3. Alignement Global

Opérations :

Dans le cas de la distance d'édition, le but était de transformer S à T avec le moins d'opérations possible (ou encore trouver le nombre d'erreurs minimal entre S et T). Maintenant, ce qui nous intéresse, c'est d'obtenir un alignement entre deux séquences. Ainsi, comme on ne transforme pas une séquence, la notion d'insertion et de délétion ne s'applique plus. À la place, nous allons appeler les opérations équivalentes : indels.

Règles:

Relation de récurrence :

$$D(i, j) = MAX \begin{cases} D(i-1, j) - 1, & /* \text{ Indel } */ \\ D(i, j-1) - 1, & /* \text{ Indel } */ \\ D(i-1, j-1) + p(i, j) & /* \text{ Match/Mismatch } */ \end{cases}$$

où $p(i, j) = 2$ si $S[i] == T[j]$ et -1 sinon.

Conditions initiales :

$D(i,0) = -i, \quad i = 0 .. m$
$D(0,j) = -j, \quad j = 0 .. n$

Exemple simple :

D(i,j)	-	G	T	C	A	G	G	T
-	**0**	**-1**	**-2**	**-3**	**-4**	**-5**	**-6**	**-7**
C	**-1**	-1	-2	0	-1	-2	-3	-4
A	**-2**	-2	-2	0	2	1	0	-1
T	**-3**	-3	0	-1	1	1	0	-1
A	**-4**	-4	-1	-1	1	0	0	-1
G	**-5**	-2	-2	-2	0	3	2	1
T	**-6**	-3	0	-1	-1	2	2	4
G	**-7**	-4	-1	-1	-2	1	4	3

De cette table, l'on dérive que le score global entre S et T est de 3.

Exemple suffixe :

Admettons que l'on veuille calculer les tables à partir des suffixes de S et T plutôt que de leur préfixes. Comment transformer les conditions initiales ainsi que les règles de récurrences ?

Réponse :

Relation de récurrence :

```
D(i, j) = MAX [   D(i+1, j) - 1,          /* Indel */
                  D(i, j+1) - 1,          /* Indel */
                  D(i+1, j+1) + p(i, j) ] /* Match/Mismatch */
```

où p(i, j) = 2 si S[i] == T[j] et -1 sinon.

Conditions initiales :

```
D(i,n) = -i,    i = m .. 0 /* ordre inverse !*/
D(m,j) = -j,    j = n .. 0
```

Pondérations

Jusqu'à présent, nous avons utilisé un modèle de pondération simple pour calculer les alignements. Par contre, plusieurs types de pondération existent :

1 – Pondération constante

Pondération que nous avons utilisée jusqu'à maintenant. On assigne simplement une valeur constante à chaque opération. Dans le cas de l'alignement global, l'on a fixé les indels ainsi que la substitution à –1 tandis que le match vaut 2.

2 – Matrice de poids

Au lieu d'avoir des valeurs fixes, l'on attribut à chaque situation une valeur différente via une matrice qui contient tous les coûts associés aux différentes situations rencontrées dans l'algorithme. Ainsi, l'alignement d'un caractère particulier peut avoir une valeur différente à un autre caractère. Ex :

P(i,j)	A	C	G	T	-
A	1	-2	-4	-1	-1
C	-2	3	-3	-2	-3
G	-4	-3	4	-4	-4
T	-1	-2	-4	2	-2
-	-1	-3	-4	-2	X

Note : La matrice n'est pas obligatoirement symétrique.

3 – Fonction quelconque

Le poids est calculé à partir d'une fonction et différents paramètres. Par exemple, si on veut pouvoir bonifier le fait de rencontrer plusieurs matchs consécutifs, il serait possible de créer une fonction qui tient compte de la longueur des matchs consécutifs pour l'attribution du score. Surtout utilisé pour la pondération des « gaps ». Voir alignement avec « gaps ».

4. Alignement local

Le but de l'alignement local est de comparer la similarité entre les séquences. Au lieu d'avoir un point de vu global, ce qui nous intéresse, c'est de trouver les régions communes entre les séquences.

Relation de récurrence :

$$D(i, j) = MAX \begin{bmatrix} 0, \\ D(i-1, j) + p(S[i],-), \\ D(i, j-1) + p(-, T[j]), \\ D(i-1, j-1) + p(i,j) \end{bmatrix}$$

où p(i, j) représente la valeur obtenue de la matrice de pondération pour les caractères S[i], T[j] et – (indel) .

Conditions initiales :

$$D(i,0) = 0, \quad i = 0 .. m$$
$$D(0,j) = 0, \quad j = 0 .. n$$

Notez que la fonction de récurrence l'on ajoute la valeur 0. Ceci a pour effet de ne jamais pénaliser le choix de commencer un nouvel alignement. C'est comme d'aligner tous les préfixes des séquences ainsi que tous leurs suffixes. Ainsi l'on s'assure que les régions similaires seront alignées. Par contre, il faut maintenant parcourir toute la table pour trouver les alignements locaux.