Université Lille1 Sciences et Technologies

ASD

2011-2012
Licence STS - Semestre 4

TP3 : la distance d'édition

Le problème connu sous le nom de distance d'édition ou distance de Levenshtein consiste à calculer le nombre minimum d'opérations d'édition d'un mot pour le transformer en un autre mot. Etant donné cette distance, on peut alors représenter les opérations sous la forme de ce qu'on appelle un alignement de séquences. L'objectif de ce TP est de programmer par programmation dynamique le calcul de l'alignement optimal (c'est à dire de distance minimale) de deux séquences. La version programmation dynamique a été proposée en 1976 par Needleman et Wunsh dans le cadre de la recherche de similarité entre séquences d'ADN et reste aujourd'hui un outil majeur de comparaison de séquences génétiques dans le domaine de la bio-informatique.

Récupérez auparavant l'archive disponible sur le portail. Vous trouverez dans un répertoire `tp3` un programme `testtp3.ml` avec les entêtes des fonctions à écrire et le code pour tester et, comme toujours, un Makefile pour compiler votre programme. Vous trouverez aussi un exemple de fichier gnuplot `test.gp` pour apprendre à tracer des fonctions récursives.

1 Défnition du problème et calcul récursif

On définit sur les mots trois opérations élémentaires :

- la *substitution* : on remplace une lettre par une autre,
- l'*insertion* : on ajoute une nouvelle lettre,
- la *suppression* : on supprime une lettre.

Par exemple, sur le mot carie, si on substitue `c` en `d`, `a` en `u` et si on insère `t` après le `i`, on obtient durite. La *distance d'édition* entre deux mots `u` et `v` est le nombre minimal d'opérations pour passer de `u` à `v`. Ainsi, la distance de carie à durite est 3: deux subsitutions et une insertion. La distance de aluminium à albumine est 4: une insertion, `b`, une substitution, `i` en `e` et deux suppressions, `u` et `m`.

Si on note u' et v' les mots u et v privés de leur première lettre, une formule de récurrence pour calculer la distance d'édition est :

$$\begin{array}{rcl} d(\varepsilon, v) & = & |v| \text{(on fait } |v| \text{ insertions)} \\ d(u, \varepsilon) & = & |u| \text{ (on fait } |u| \text{ suppressions)} \\ d(au', av') & = & d(u', v') \\ d(au', bv') & = & 1 + min(d(u', v'), d(u, v'), d(u', v)) \\ & & \text{(la dernière opération est une substitution de la lettre } a \text{ en } b, \\ & & \text{ou une insertion de la lettre } b, \text{ ou une suppression de la lettre } a) \end{array}$$

où a et b sont deux lettres quelconques distinctes, ε est le mot vide et où $|u|$ représente la longueur du mot u.

Q 1 Écrire une fonction récursive

```
distance_recursive : string -> string -> int
```

qui calcule la distance d'édition de deux chaînes de caractères. Testez-la sur carie et durite, aluminium et albumine, puis sur un couple de mots un peu plus longs.

Q 2 Donner l'équation de récurrence à deux paramètres fournissant le nombre d'opérations d'additions dans le pire des cas.

Q 3 En supposant que les deux chaînes de caractères passées en paramètre sont de même longueur, dessinner avec Gnuplot cette fonction puis trouver expérimentalement une borne asymptotique (vous rendrez la courbe). Vous pourrez regarder comment faire avec les commandes Gnuplot disponibles dans le fichier `test.gp` fourni.

2 Calcul par programmation dynamique de la distance

Le problème de la fonction `distance_recursive` est qu'elle effectue de nombreux appels récursifs redondants. Cela conduit à une complexité exponentielle. La solution est de stocker les appels récursifs dans une table `table` de dimension $n \times m$, telle que `table.(i).(j)` soit la distance de `u.[0..i-1]` à `v.[0..j-1]`. On ajoute une colonne et une ligne pour traiter le mot vide. Cela donne finalement une table indexée par `0..n` et `0..m` si n et m sont respectivement les longueurs de u et v. Le résultat est `table.(n).(m)`. Par exemple la table pour carie et durite est

		c	a	r	i	e
	0	1	2	3	4	5
d	1	1	2	3	4	5
u	2	2	2	3	4	5
r	3	3	3	2	3	4
i	4	4	4	3	2	3
t	5	5	5	4	3	3
e	6	6	6	5	4	3

Q 4 Écrire une fonction

```
distance_dynamique : string -> string -> int
```

qui calcule la distance de deux chaînes de caractères sans appels récursifs, en construisant la table `table`. On pourra avantageusement créer une fonction

```
construire_table : string -> string -> int array array
```

qui calcule la table de programmation dynamique `table`.

Q 5 En supposant que $n = m$, donner le nombre d'opérations d'additions dans le pire des cas.

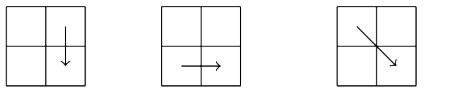
Q 6 Déduisez-en une borne asymptotique pour cet algorithme.

3 Obtention de l'alignement

On veut maintenant connaître la suite d'opérations qui mène de `u` à `v`. Pour cela, on peut visualiser les transformations par un petit schéma:

```
c a r i - e          a l - u m i n i u m
    | |   |          |   |   | | | |
d u r i t e          a l b u m i n e - -
```

Deux lettres identiques sont signalées par |. Un `-` dans le mot de départ correspond à une insertion, un `-` dans le mot d'arrivée correspond à une suppression, et une substitution est représentée par les deux lettres face à face, sans |. Il est possible de connaître la dernière opération appliquée en regardant comment la valeur de `table.(n).(m)` a été obtenue. Par exemple, si `table.(n).(m)` est égal à `table.(n).(m-1)`+1, alors la dernière opération est une insertion de `v.(m-1)`. Ci-dessous est représenté schématiquement d'où provient le résultat et l'opération correspondante :



insertion suppression substitution ou identité

Q 7 Retrouvez à la main à partir de la table de l'exemple la suite des transformations pour passer de carie à durite.

Q 8 Écrire une procédure :

```
alignement : string -> string -> unit
```

qui affiche la suite d'opérations sous la forme décrite ci-dessus. Pour cela, vous devez utiliser la table et construire l'historique des transformations en remontant dans la table à partir de `(n,m)`.