Tutoriels

Initiation à la Programmation

ENSAE

Xavier Dupré

Table des matières

Chapitre 1

Commencer à programmer

1.1 Levenshtein, Viterbi

Ce tutoriel mélange des questions dont la réponse attendue est un extrait de programme et d'autres dont la réponse doit être rédigée. Le rendu doit contenir deux choses :

1. un (ou plusieurs) programme informatique ne faisant aucune référence à des fichiers ou ressources liées à l'ENSAE (pas de fichier W :..... par exemple),
2. un rapport imprimé (non manuscrit) regroupant les réponses à chaque question incluant les extraits de programmes.

L'élève devra envoyer par email à son chargé de TD dans un seul fichier ZIP l'ensemble des programmes et le rapport. Il devra également imprimer son rapport et le rendre à son chargé de TD.

Ce tutoriel s'intéresse à des algorithmes non abordés lors des travaux pratiques. On se propose d'aborder la programmation dynamique [1] au travers de deux exemples. Ce concept cache une méthode d'optimisation qui revient relativement souvent. Ce tutoriel en présentera deux formes appliquées dans des contextes différents.

Partie 1 : Levenstein .

La distance de Levenstein [2] est un algorithme qui permet de construire une distance entre deux séquences, ici des mots. On l'appelle aussi distance d'édition. Elle consiste à définir une transformation minimale permettant de passer d'un mot à un autre à partir de trois opérations simples : suppression, addition ou substitution de caractères. Par exemple, l'utilisateur d'un moteur de recherche souhaite des résultats concernant *restaurant* mais au lieu de cela, il a rentré `rstairnts`.

```
r  s t a i r  n t s
r e s t a u r a n t
```

En alignant les lettres entre elles, on construit une transformation qui permet de passer de l'un à l'autre :
– ajout d'un e
– substitution d'un i par un u
– ajout d'un a
– suppression d'un s

La distance d'édition est définie par le nombre d'opérations de la transformation minimale - à savoir celle qui contient le moins d'opérations -. Ici, on obtient : $4 = 2$ ajouts $+ 1$ suppression $+ 1$ substitution. La page http://fr.wikipedia.org/wiki/Distance_de_Levenshtein contient une description de l'algorithme qui permet de calculer cette distance :

1. http://fr.wikipedia.org/wiki/Programmation_dynamique
2. http://fr.wikipedia.org/wiki/Distance_de_Levenshtein

```
entier DistanceDeLevenshtein(caractere chaine1[1..longueurChaine1],
                             caractere chaine2[1..longueurChaine2])
   // d est un tableau de longueurChaine1+1 rangées et longueurChaine2+1 colonnes
   declarer entier d[0..longueurChaine1, 0..longueurChaine2]
   // i et j itèrent sur chaine1 et chaine2
   declarer entier i, j, coût

   pour i de 0 à longueurChaine1
       d[i, 0] := i
   pour j de 0 à longueurChaine2
       d[0, j] := j

   pour i de 1 à longueurChaine1
       pour j de 1 à longueurChaine2
           si chaine1[i] = chaine2[j] alors coût := 0
                                      sinon coût := 1
           d[i, j] := minimum(
                                      d[i-1, j  ] + 1,     // effacement
                                      d[i,   j-1] + 1,     // insertion
                                      d[i-1, j-1] + coût   // substitution
                                  )

   retourner d[longueurChaine1, longueurChaine2]
```

1) Transcriver ce pseudo-code en programme *Python* et vérifier que l'on obtient bien une distance de 4 pour l'exemple cité. (2 points)

2) On remplace la partie suivante :

```
           d[i, j] := minimum(
                                      d[i-1, j  ] + 1,     // effacement
                                      d[i,   j-1] + 1,     // insertion
                                      d[i-1, j-1] + coût   // substitution
                                  )
```

Par :

```
           e = d[i-1, j  ] + 1
           i = d[i,   j-1] + 1
           s = d[i-1, j-1] + coût
           if s <= min (e,i) :                          # ligne A
               d [i, j] := s
               predecessor [i,j] = (i-1, j-1)
           elif e < min (s,i) :
               d [i, j] := e
               predecessor [i,j] = (i-1, j)
           else :
               d [i, j] := i
               predecessor [i,j] = (i, j-1)
```

En supposant que `predecessor` a été initialisé comme il le fallait, que contient-il ? (1 point)

Que contient `predecessor`[`longueurChaine1`, `longueurChaine2`] ? (valeur numérique attendue) (1 point)

3) A la ligne A, on remplace le symbol <= par =. Donner un couple de mots pour lequel le résultat devient différent ? Quelle version préférez-vous ? (1 point)

4) En déduire (et implémenter) un moyen de construire la séquence d'opérations qui permet de passer du premier mot au second ? (2 points)

Partie 2 : dés pipés, simulation d'un tirage

On dispose de trois dès pipés dont les probabilités de sortie de chacune des faces sont :

face	1	2	3	4	5	6
dé 1	0.1	0.2	0.3	0.1	0.1	0.2
dé 2	0.3	0.1	0.1	0.1	0.1	0.3
dé 3	0.1	0.2	0.2	0.2	0.1	0.2

On commence par le dé 1. Au lancer suivant, on choisit :
– le dé 1 si le nombre sorti est 1 ou 2,
– le dé 2 si le nombre sorti est 3 ou 4,
– le dé 3 si le nombre sorti est 5 ou 6,
On effectue trois tirages et on se demande quelle est la séquence de trois chiffres la plus probable. On procède d'abord empiriquement.

5) Ecrire une fonction qui tire un nombre aléatoire entre 1 et 6 selon un vecteur de six probabilités déterminées à l'avance. On utilisera la fonction `randint` du module `random` (on pourra utiliser la fonction de répartition d'une variable aléatoire discrète). (2 points)

6) Créer une fonction qui simule une séquence de trois tirages consécutifs en suivant les règles décrites en introduction. (1 point)

7) Générer 10000 tirages et compter le nombre de fois qu'apparaît chaque séquence. (1 point)

8) Quelle est la probabilité de la séquence 666 ? (1 point)

9) Ecrire une fonction qui détermine la séquence la plus probable. (1 point)

Partie 3 : matrice de transition

On définit une matrice 6×6 notée A_{ij} ou matrice de transition.

> a_{ij} est la probabilité d'obtenir le nombre j au second tirage
> sachant qu'on le premier tirage était i.

10) Démontrer que la probabilité ne change pas lorsqu'on passe du second tirage au troisième. (1 point)

Partie 4 : Viterbi

L'algorithme de Viterbi repose sur la constatation suivante : si on note la $S_t(i)$ la probabilité de la séquence la plus probable de t tirages se terminant par i alors au tirage $t+1$, on vérifie que :

$$S_{t+1}(j) = \max_i S_t(i) a_{ij} \tag{1.1}$$

11) Que vaut $S_0(i)$? (1 point)

12) Ecrire une fonction qui calcule $S_{t+1}(j)$ pour tout j si on suppose connu $S_t(i)$ pour tout i. (2 points)

13) Quelle est la probabilité de la séquence de trois tirages la plus probable et quelle est son dernier chiffre ? On pourra s'intéresser à ce que vaut $\max_i S_3(i)$. (1 point)

14) En vous inspirant de la première partie, écrire une fonction qui retourne la séquence la plus probable (et pas seulement sa probabilité ni son dernier chiffre ?) (2 points)

15) On suppose maintenant qu'on lance 10 fois les dés, toujours en suivant la règle qui détermine le dé utilisé au tirage suivant. Parmi les deux méthodes présentées (parties 2 et 4), quelle est celle que vous préconisez pour déterminer la séquence la plus probable et pourquoi ? (1 point)

Index

Table des matières