Mines-Ponts - Informatique commune - 2024

Corrigé

1. On peut par exemple utiliser le codage suivant :

Caractère	Code
`'a'`	0
`'b'`	10
`'c'`	11

   Pour décoder sans ambiguïté, on parcourt le message codé de gauche à droite. Si on voit un 0, on le décode en un `'a'`. Sinon, on regarde le bit suivant. Si c'est un 0, on décode les deux bits en un `'b'`, et sinon en un `'c'`. Puis on passe au bit suivant et on recommence.

   (Si on avait codé `'b'` par 00, le codage serait ambigu : on ne saurait pas s'il faut décoder 00 en `'b'` ou en `'aa'`.)

2. ```
   def nbCaracteres(c, s):
     n = 0
     for i in range(len(s)):
       if s[i] == c:
         n = n + 1
     return n
   ```

3. Lorsque `s='abaabaca'`, la fonction renvoie `['a', 'b', 'c']`. La fonction parcourt la chaîne de caractères de gauche à droite et pour chaque caractère, elle l'ajoute à la fin de la liste s'il n'est pas déjà présent.

4. La fonction est constitué d'une boucle parcourue `n` fois. À chaque passage dans la boucle, il y a un test d'appartenance dont la complexité est linéaire en la longueur de la liste `listeCar`, donc en $O(k)$. Tout le reste se fait en complexité constante. La complexité dans le pire des cas est donc $O(nk)$.

5. Cette fonction renvoie une liste de couples dont le premier élément est un caractère et le deuxième élément est le nombre d'apparitions de ce caractère dans `s`. Les couples sont rangés par ordre d'apparition des caractères. Par exemple, `analyseTexte('babaaaabca')` renvoie `[('b', 3), ('a', 6), ('c', 1)]`.

6. La fonction commence par un appel à `analyseTexte`, de complexité $O(nk)$. Il y a ensuite une boucle parcourue `k` fois. À chaque passage dans la boucle, il y a un appel à la fonction `nbCaracteres`, de complexité $O(n)$. Tout le reste se fait en complexité constante. La complexité dans le pire des cas est donc $O(nk)$.

7. ```
   def analyseTexteDic(s):
     R = {}
     for i in range(len(s)):
       c = s[i]
       if c in R:
         R[c] = R[c] + 1
       else:
   ```

```
        R[c] = 1
    return R
```

8. ```
   SELECT DISTINCT auteur
   FROM corpus;
   ```

9. ```
   SELECT c.symbole,
     SUM(o.nombreOccurrences) / (SELECT SUM(nombreCaracteres)
                                 FROM corpus
                                 WHERE langue = 'Français') AS frequence
   FROM occurrences o
   JOIN caractere c ON o.idCar = c.idCar
   JOIN corpus co ON o.idLivre = co.idLivre
   WHERE co.langue = 'Français'
   GROUP BY c.symbole;
   ```

10. - On obtient d'abord l'intervalle $[0.2; 0.3[$ correspondant au caractère `'b'`.
    - Le caractère `'a'` détermine alors le sous-intervalle $[0.2; 0.22[$ correspondant à la portion associée au caractère `'a'`.
    - Le caractère `'c'` détermine enfin l'intervalle $[0.206; 0.21[$.

11. ```
    def codage(s):
      (g,d) = (0,1)
      for i in range(len(s)):
        (g, d) = codeCar(s[i], g, d)
      return (g, d)
    ```

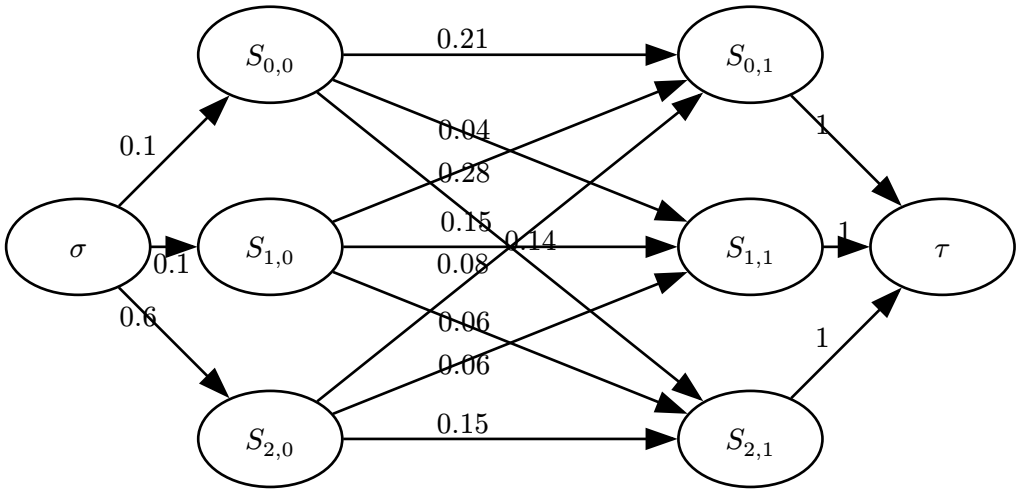
12. Les chaînes commençant par `'ad'` ont un code dans l'intervalle $[0.10; 0.18[$. `'ada'` correspond à $[0.10; 0.116[$ et `'adb'` correspond à $[0.116; 0.124[$. Le caractère qui suit `'ad'` dans la chaîne codée par `x=0.123` est donc `'b'`.

13. Les chaînes `'b'` et `'ba'` peuvent toutes les deux correspondre au flottant 0.2. Le problème est qu'on ne sait pas quand arrêter le décodage.

14. ```
    def decodage(x):
      s = ""
      (g, d) = (0, 1)
      c = decodeCar(x, g, d)
      while c != '#':
        s = s + c
        (g, d) = codeCar(c, g, d) # on calcule l'intervalle dans lequel est situé c
        c = decodeCar(x, g, d)
      s = s + '#'
      return s
    ```

15. - Le nombre de sommets est NK (il y en a un par couple $(i, j) \in [\![0, K-1]\!] \times [\![0, N-1]\!]$).
    - Le nombre d'arcs est $(N-1)K^2$ (il y en a K qui partent de chaque sommet, sauf pour la dernière couche verticale).

16.



17. Pour choisir un chemin de σ à τ, on a K possibilités à chaque couche verticale (en comptant la couche de σ, mais pas celle de obs_{N-1}). Il y a donc K^N tels chemins. De plus, le calcul de la probabilité associée à un chemin se fait en complexité $O(N)$. Un algorithme d'exploration exhaustive devient très vite inenvisageable, à moins de se contenter de petites valeurs de N et K. Par exemple, pour $K = 3$ et $N = 8$ comme dans l'énoncé, on obtient de l'ordre de $8 \times 3^8 \approx 5 \times 10^4$ opérations élémentaires, ce qui reste envisageable.

18.
```
def maximumListe(liste):
  maximum = liste[0]
  argMax = 0
  for i in range(1, len(liste)):
    if liste[i] > maximum: # inégalité stricte pour assurer qu'on prend le premier
argmax
      maximum = liste[i]
      argMax = i
  return (maximum, argMax)
```

19.
```
def glouton(Obs, P, E, K, N):
  (maximum, argMax) = maximumListe([E[Obs[0]][i] for i in range(K)])
  etats = [argMax]
  for j in range(N-1):
    (maximum, argMax) = maximumListe([E[Obs[j+1]][k] * P[argMax][k] for k in
range(K)]) # on emprunte l'arête de poids maximal
    etats.append(argMax)
  return etats
```

20. Chaque appel à `maximumListe` a une complexité en $O(\texttt{K})$ (il y a une boucle avec $\texttt{K} - 1$ passages et le reste se fait en temps constant). Dans la fonction `glouton`, il y a une boucle avec $\texttt{N} - 1$ passages. Dans chaque passage, il y a un appel à `maximumListe`, et le reste se fait en temps constant. La complexité est donc en $O(\texttt{NK})$.

21. L'algorithme glouton renvoie le chemin $\sigma \xrightarrow[0.6]{} S_{0,0} \xrightarrow[0.5]{} S_{0,1} \xrightarrow[1]{} \tau$ de probabilité 0.3. Mais le chemin $\sigma \xrightarrow[0.4]{} S_{1,0} \xrightarrow[0.9]{} S_{0,1} \xrightarrow[1]{} \tau$ a une probabilité $0.36 > 0.3$. L'algorithme glouton ne renvoie donc pas nécessairement le chemin de probabilité maximale.

22. On peut remplacer chaque poids p par $-\ln(p)$. On obtient alors un graphe à poids positifs, et le plus court chemin dans ce graphe correspond au chemin de probabilité maximale dans le graphe initial. On pourrait alors utiliser l'algorithme de Dijkstra.

23.

```
def construireTableauViterbi(Obs, P, E, K, N):
  (T, argT) = initialiserViterbi(E, Obs[0], K, N)
  for i in range(K):
    for j in range(1, N):
      probas = [T[k][j-1] * P[k][i] * E[Obs[j]][i] for k in range(K)]
      (T[i][j], argT[i][j]) = maximumListe(probas)
  return (T, argT)
```

24. La probabilité maximale d'un chemin se lit sur la dernière ligne : il s'agit de $1.8e-05$. Pour reconstituer la séquence d'états la plus probable, on parcourt les colonnes de `argT` à l'envers. On trouve $\sigma \longrightarrow S_{2,0} \longrightarrow S_{0,1} \longrightarrow S_{0,2} \longrightarrow S_{2,3} \longrightarrow S_{1,4} \longrightarrow S_{1,5} \longrightarrow S_{0,6} \longrightarrow S_{0,7} \longrightarrow \tau$.

25. La fonction `initialiserViterbi` a une complexité $O(\mathtt{K})$. Il y a ensuite deux boucles imbriquées, soit $O(\mathtt{NK})$ passages dans la boucle. Chaque passage dans la boucle intérieure a une complexité temporelle $O(\mathtt{K})$ à cause de l'appel à `maximumListe`. La complexité temporelle est donc $O(\mathtt{NK}^2)$. La complexité spatiale est en $O(\mathtt{NK} + \mathtt{K}^2)$ car il faut gérer des tableaux de taille $\mathtt{N} \times \mathtt{K}$ et de taille $\mathtt{N} \times \mathtt{K}$ (on ne sait pas si N est plus grand que K donc on ne peut pas simplifier l'expression).

Commentaires

- **Compression de données.** La première partie porte sur le codage arithmétique, qui est une technique de compression des données sans perte. Elle n'est cependant pas beaucoup utilisé. L'énoncé mentionne la norme JPEG 2000, mais celle-ci n'est pas très répandue.

  La compression sans perte s'oppose à la compression avec perte, qui est utilisée pour les images et les sons, car on peut y apporter des modifications importantes mais imperceptibles (ou très peu perceptibles) par l'humain.

  La technique de compression sans perte la plus connue est sans doute le codage de Huffman. Le code de la première question du sujet en est un exemple. Il consiste à coder les caractères les plus fréquents avec des petits mots de code, tout en évitant les ambiguïtés. Il n'est cependant pas optimal, ce qui vient essentiellement du fait qu'il faut coder chaque caractère sur un nombre entiers de bits (ce qui n'est pas le cas du codage arithmétique). Il existe d'autres codages sans perte plus efficaces, comme les codages par dictionnaire.

- **Problèmes de précision des flottants.** Comme indiqué à la fin de la première partie, le codage arithmétique tel qu'exposé dans le sujet est problématique à cause de la précision limitée des flottants. Pour pallier ce problème, on utilise une renormalisation : quand on s'approche de la limite de précision, on sauvegarde les premiers chiffres (ceux qui sont connus à coup sûr quels que soient les caractères pas encore codés), puis on translate les chiffres vers la gauche.

- **Algorithme de Viterbi.** La deuxième partie du sujet porte sur l'algorithme de Viterbi. Il fait partie du vaste domaine du codage correcteurs d'erreurs qui, comme son nom l'indique, s'intéresse aux codes que l'on peut décoder en détectant et en corrigeant un certain nombre d'erreurs. L'algorithme de Viterbi est utilisé massivement dans les télécommunications. De plus, il possède des applications en dehors des communications sur un canal bruité, puisqu'il permet de calculer dans un cadre assez général la cause la plus probable d'une observation. Il est notamment utilisé en reconnaissance vocale.