Informatique tronc commun : concours blanc 2025

Corrigé

MPI, Lycée Leconte de Lisle

1 Compression du message d'Alice : codage arithmétique

Q1 On peut utiliser le code suivant :

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

Le codage de `s='abaabaca'` est alors :

`0 10 0 0 10 0 11 0` (11 bits)

Le texte peut être décodé sans ambiguïté : lorsqu'on rencontre un bit `0` on décode en `'a'`, lorsqu'on rencontre un `1`, on lit le bit suivant, si c'est un `0` on décode en `'b'` sinon en `'c'`.

1.1 Analyse du texte source

Q2 On parcourt les caractères de la chaîne `s` :

```python
def nbCaracteres(c:str, s:str) -> int :
    compteur = 0
    for l in s:
        if l == c:
            compteur += 1
    return compteur
```

Il était aussi correct de parcourir les indices :

```python
def nbCaracteres(c:str, s:str) -> int :
    compteur = 0
    for i in range(len(s)):
        if s[i] == c:
            compteur += 1
    return compteur
```

Q3 Pour l'entrée `s='abaabaca'` la fonction renvoie la liste `['a', 'b', 'c']` c'est-à-dire la liste sans doublon des caractères présents dans `s`. Son principe de fonctionnement est le suivant :

- on parcourt les indices de la chaîne s.
- on maintient à jour dans listeCar la liste des caractères de s[0..i] sans doublon (c'est l'invariant de boucle qui est initialement vrai).
- à l'itération d'indice i si le caractère s[i] n'est pas présent dans listeCar on l'ajoute.
- à la fin, l'invariant est vrai donc listeCar contient la liste des caractères de s[0..n] = s

Q4 La boucle for du programme s'exécute exactement n fois. Le calcul c in listeCar prend dans le pire cas un temps $O(k)$. La complexité est donc $\boxed{O(nk)}$.

Q5 La fonction retourne une liste de couples (c, k) où c est un caractère de la chaîne s et k le nombre d'occurrences du caractère c dans la chaîne s. La commande analyseTexte('babaaabca') retourne

```
[('b', 3), ('a', 6), ('c', 1)]
```

Q6 La fonction commence par appeller listeCaracteres qui coûte $O(nk)$ (Q4). Elle exécute ensuite une boucle for qui itère k fois. Le corps de la boucle a un coût $O(n)$ (nbCaracteres (Q2)). La complexité est donc :

$$O(nk) + k \times O(n) = O(nk) + O(nk) = \boxed{O(nk)}$$

Q7 On utilise un dictionnaire dans lequel les *clés* sont les caractères qui apparaissent dans s et les *valeurs* sont les nombres d'occurrences.

```python
def analyseTexteDictionnaire(s:str):
    R = {} # dictionnaire vide
    for c in s:
        if c in R: # nouvelle occurrence de c
            R[c] = R[c] + 1
        else: # premiere occurence de c
            R[c] = 1
    return R
```

La boucle for s'exécute n fois, le corps de la boucle fait appel à un test d'appartenance au dictionnaire en $O(1)$ et une lecture/modification d'une valeur en $O(1)$, la complexité de analyseTexteDictionnaire est donc :

$$n \times (O(1) + O(1)) = \boxed{O(n)}$$

ce qui améliore la complexité précédente.

1.2 Exploitation d'analyses existantes

Q8

```sql
SELECT DISTINCT auteur FROM corpus;
```

Q9 La requête suivante permet d'obtenir N le nombre total de caractères en français dans l'ensemble de toutes les œuvres :

```
SELECT SUM(nombreCaracteres) FROM corpus WHERE langue = 'Français';
```

Nous pouvons alors calculer le nombre d'occurrences de chaque symbole dans l'ensemble des œuvres en français puis diviser le résultat par N à l'aide d'une requête imbriquée :

```
SELECT symbole, SUM(nombreOccurrences)/(SELECT SUM(nombreCaracteres) FROM cor-
pus WHERE langue = 'Français')
FROM caractere JOIN corpus JOIN occurrences
ON caractere.idCar = occurrences.idCar
AND corpus.idLivre = occurrences.idLivre
WHERE langue = 'Français'
GROUP BY symbole;
```

1.3 Compression

Q10 On applique l'algorithme proposé :

mot	intervalle
`b`	$[0.2; 0.3[$
`ba`	$[0.20; 0.22[$
`bac`	$[0.206; 0.21[$

Q11

```
def codage(s:str) -> (float, float) :
    # Bornes de l'intervalle actuel
    g = 0.0
    d = 1.0
    for i in range(len(s)):
        (g, d) = codeCar(s[i], g, d)
    return (a, b)
```

1.4 Décodage

Q12 Le mot `'ad'` conduit à l'intervalle de codage $[0.1; 0.18[$:

mot	intervalle
`a`	$[0; 0.2[$
`ad`	$[0.1; 0.18[$

Il nous faut donc calculer à quelle proportion de l'intervalle $[0.1; 0.18[$ le nombre $x = 0.123$ se situe :

$$\frac{x - 0.1}{0.18 - 0.1} = \frac{0.023}{0.08} = \frac{23}{80} = 0.28\ldots$$

ce qui correspond à l'intervalle $[0.2; 0.3[$ donc le prochain caractère est `b`.

Q13 Les mots `ba` et `baa` correspondent au flottant `0.2`. Le fait que 0 est dans le premier intervalle implique qu'on peut ajouter des `a` au mot sans changer le codage, il faut savoir quand le mot codé termine.

Q14 On détermine le caractère c correspondant à x dans $[0, 1[$ frâce à la fonction `decodeCar`, ceci est la première lettre du mot à décoder. On répète ensuite le processus avec le sous-intervalle correspondant à c que l'on obtient avec la fonction `codeCar`, on répète le processus jusqu'à aboutir au caractère de terminaison # :

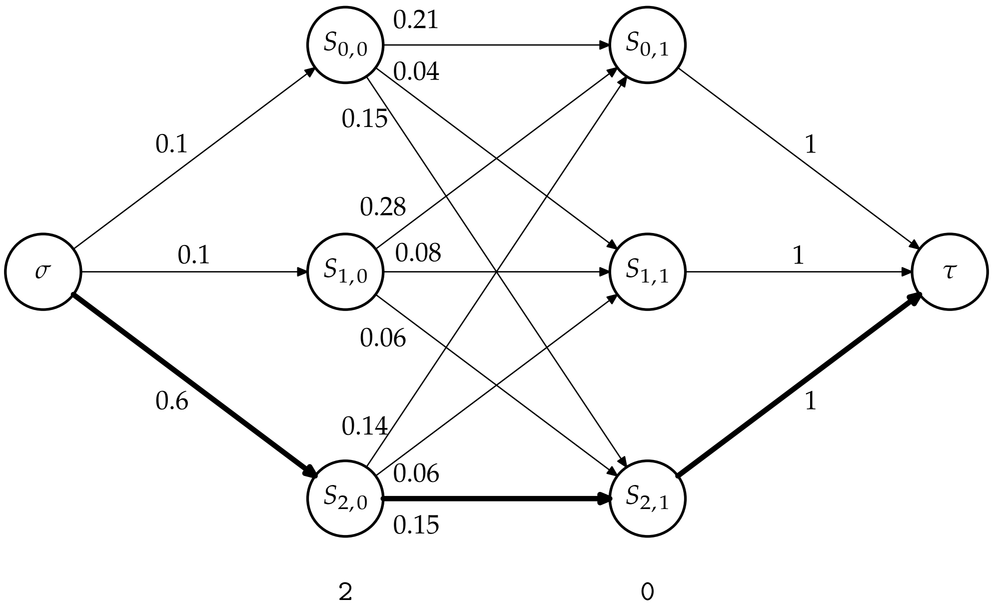
```python
def decodage(x:float) -> str :
    s = ''
    # Bornes de l'intervalle courant
    g = 0.0
    d = 1.0
    c = decodeCar(x, g, d)
    while c != '#':
        s = s + c
        g, d = codeCar(c, g, d)
        c = decodeCar(x, g, d)
    return s
```

2 Décodage du message reçu par Bob à l'aide de l'algorithme de Viterbi

2.1 Modélisation du canal de communication par un graphe

Q15 Il y a $\boxed{NK}$ sommets (en excluant σ et τ). Chaque sommet possède K arcs sortants, mais on ne doit pas compter les arcs sortants de la dernière colonne de sommets car ils mènent à τ ; il y a donc $\boxed{(N-1)K^2}$ arcs à comptabiliser.

Q16 On obtient :



À titre d'information, je représente également le chemin de probabilité maximale 0.09 (en trait épais), qui correspond au décodage `[2, 2]` (le dernier symbole a été corrigé).

Q17 Notons C_N le nombre de chemins dans le graphe pour un message de longueur $N \in \mathbb{N}$. On a initialement $C_0 = 1$. Si le message est de longueur $N \in \mathbb{N}^*$, on a initialement K choix pour le premier arc menant à $S_{i,0}$. On peut ensuite voir le reste du chemin comme celui dans un graphe possédant une colonne de moins et dans lequel l'état de départ est $S_{i,0}$. On obtient alors la relation de récurrence suivante :

$$C_0 = 1 \qquad \forall N \in \mathbb{N}^*,\ C_N = KC_{N-1}$$

C_N est donc une suite géométrique de terme général $\boxed{C_N = K^N}$. La nombre de chemins est exponentiel en la longueur du message donc une exploration exhaustive n'est pas envisageable.

2.2 Stratégie gloutonne

Q18

```
def argMax(liste:[float]) -> (float, int) :
    assert len(liste) > 0
    val_max = liste[0]
    imax = 0
    for i in range(1, len(liste)):
        if liste[i] > val_max:
            val_max = liste[i]
            imax = i
    return (val_max, imax)
```

La fonction `initialiserGlouton` proposé dans le sujet initial comporte certaines erreurs. Tout d'abord le prototype de la fonction est :

`initialiserGlouton(Obs:[int], E:[[float]], K) -> int`

car `Obs` est une liste et non une matrice. De plus, il faut bien parcourir la **ligne** d'indice `Obs[0]` et non la colonne (car les arcs issus de σ ont pour probabilité $E_{obs_0,i}$ pour $i \in [\![0, K-1]\!]$). Le programme proposé dans le sujet tente bien de faire cela mais comporte encore une erreur de syntaxe en ligne 2. En voici une version corrigée :

```
def initialiserGlouton(Obs, E, K):
    probasInitiales = [ E[Obs[0]][i] for i in range(K) ]
    s, symbole = maximumListe(probasInitiales)
    return symbole
```

Q19

```
def glouton(Obs:[int], P:[[float]], E:[[float]], K:int, N:int) -> [int] :
    chemin = []
    symbole_prec = initialiserGlouton(Obs, E, K)
    chemin.append(symbole_prec)
    for symbole in Obs[1:]:
        probas = [ P[symbole_prec][j] * E[symbole][j] for j in range(K)]
        s, symbole_prec = maximumListe(probas)
        chemin.append(symbole_prec)
    return chemin
```

Q20 Initialiser glouton a un coût $O(K) + O(K) = O(K)$ (construction de la liste par compréhension + recherche de l'argument maximum). La boucle `for` s'exécute $N-1$ fois. Le corps de la boucle a lui même une complexité $O(K) + O(K) = O(K)$. La complexité est donc :

$$O(k) + (N-1) \times O(K) = O(NK)$$

Cela améliore nettement la complexité exponentielle de l'approche exhaustive.

Q21 L'algorithme choisit d'abord l'arc de probabiltié 0.6 qui a la plus grande sortie parmi les arcs sortants de σ, puis il choisit l'arc de probabilité 0.5 qui est la plus grande probabilité parmi les arcs sortants de $S_{0,0}$. Ainsi le chemin obtenu, correspondant au décodage `[0, 0]`, a une probabilité $0.6 \times 0.5 = 0.3$.

Cependant ce n'est pas le chemin optimal, qui correspond au décodage `[1, 0]` ayant pour probabilité $0.4 \times 0.9 = 0.36$. Ainsi, l'approche gloutonne est abordable en terme de complexité mais ne conduit pas nécessairement au meilleur décodage possible.