## Devoir surveillé #1 (avec solutions)

*Samedi 2025-10-18 ; durée : deux heures*

Ce sujet est repris de l'épreuve d'informatique commune du *Concours Commun Mines Pont 2024*. Les seules différences avec le sujet original sont des modifications mineures du texte, la suppression de deux questions portant sur les bases de données, et la conversion d'une remarque en question.

**Rappels concernant le langage Python.** Il n'est pas possible d'utiliser des fonctions internes à Python sur les listes ou les chaînes de caractères telles que `min`, `max`, `count`, `remove`... Seules les instructions basiques telles que `len(liste)`, `liste.append(e)` sont autorisées. Le tranchage ou *slicing* ainsi que la concaténation sont également permis. Les programmes doivent être commentés lorsque c'est nécessaire pour justifier les choix. Il n'est pas utile de rappeler la signature des fonctions demandées dans les différentes questions.

# *Introduction à deux problèmes en communication numérique*

### Introduction

On s'intéresse au problème de communication entre deux personnes, nommées Alice et Bob qui cherchent à s'envoyer un message au travers d'un canal de communication (une bande de fréquences radio par exemple). Avant d'être lu par Bob, le message original d'Alice passe par plusieurs étapes que nous allons séparer de la manière suivante :

— une phase de *compression*, durant laquelle Alice cherche à trouver la représentation la plus compacte possible du message ;
— une phase d'*encodage* durant laquelle le message compressé est transformé en une succession de symboles transmissibles au travers du canal de communication utilisé ;
— une phase de *transmission* durant laquelle le message encodé circule sur le canal de communication et est susceptible de subir une altération ;
— une phase de *décodage* durant laquelle Bob décode le message qu'il a reçu, le message lui apparaît alors sous la forme compressée ;
— une phase de *décompression* durant laquelle Bob applique l'opération réciproque de la compression opérée par Alice.

Ce modèle est décrit par le schéma de la Figure 1.

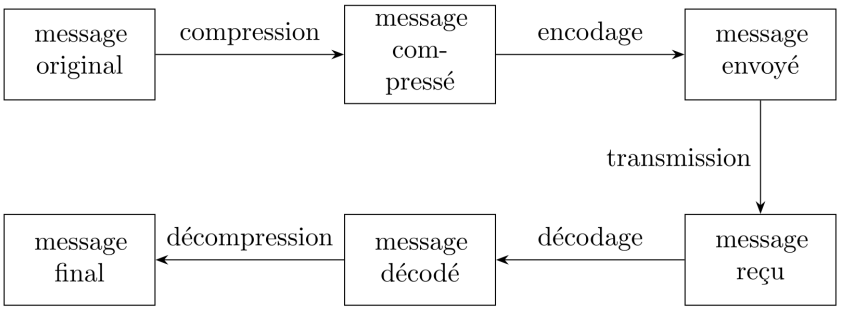



FIGURE 1 – Schématisation du modèle de communication considéré ici.

Dans cette épreuve, nous allons nous intéresser uniquement à deux phases : la compression du message d'origine par Alice en un message compact et le décodage par Bob d'un message transmis, potentiellement entaché d'erreurs.

# Compression du message d'Alice : codage arithmétique

Le codage ASCII (American Standard Code for Information Interchange) définit une norme où 128 caractères sont codés sur 7 bits. Ce codage est illustré par les quelques lignes suivantes :

| Caractère | Codage binaire | Équivalent décimal |
|---|---|---|
| `'a'` | 110 0001 | 97 |
| `'b'` | 110 0010 | 98 |
| `'z'` | 111 1010 | 122 |

Lorsqu'une chaîne de caractères n'utilise pas l'intégralité des 128 caractères proposés par le codage ASCII, il est possible de convenir d'une représentation différente, plus économique en nombre de symboles. Considérons la chaîne de caractères `s='abaabaca'`. On peut proposer de coder celle-ci à l'aide du tableau suivant :

| Caractère | Code |
|---|---|
| `'a'` | 00 |
| `'b'` | 01 |
| `'c'` | 10 |

Dans ce cas, la chaîne de caractères s est codée sur 16 bits par :

`00 01 00 00 01 00 10 00`

(où des espaces ont été introduites pour faciliter la lecture).
Dans un souci de compression de l'information, il est intéressant de représenter les caractères les plus fréquents par des expressions courtes et de ne plus nécessairement coder avec des codes de longueur constante chaque caractère. Dans l'exemple précédent, il est possible de coder le caractère `'a'` avec 1 bit, et les caractères `'b'` et `'c'` avec 2 bits afin de coder la chaîne `s` sur seulement 11 bits en tout.

1. Proposer une telle représentation en expliquant pourquoi celle-ci pourra être décodée sans ambiguïté. Vous ferez en sorte que la représentation binaire de `'a'` soit inférieure (quand interprétée comme un entier écrit en base deux) à celle de `'b'`, elle-même inférieure à celle de `'c'`.

   On peut choisir de représenter `'a'` par 0, `'b'` par 10 et `'c'` par 11 (qui satisfont les contraintes de l'énoncé). Une telle représentation pourra être décodée sans ambiguïté car (en supposant que l'on a déjà décodé un éventuel préfixe d'une chaîne) :

   — Si le prochain caractère est 0 on décode `'a'` et l'on passe au caractère suivant.
   — Si le prochain caractère est 1, on lit également le caractère suivant pour décider entre `'b'` et `'c'` et (ayant ainsi lu deux caractères) on passe au caractère suivant.

La représentation précédente emploie la même longueur pour coder les caractères `'b'` et `'c'` alors que le caractère `'b'` est deux fois plus présent que le caractère `'c'` dans la chaîne `s`.
Il est possible d'aller un cran plus loin et le codage arithmétique présenté dans cette étude permet un gain de compression comme s'il parvenait à représenter un caractère avec un nombre non entier de bits au prorata de sa fréquence d'apparition. Ce principe de compression est notamment utilisé par la norme *JPEG2000* de compression des images. Nous ne le présenterons cependant ici que dans le cadre de l'étude de chaînes de caractères.

### Analyse du texte source

L'objet de cette partie est d'analyser le contenu d'une chaîne de caractères `s` afin de déterminer :

— les caractères utilisés par la chaîne `s` ;
— le nombre d'occurrences de chacun.

2. Écrire une fonction nommée `nbCaracteres(c:str,s:str)->int` qui prend comme argument un caractère `c`, une chaîne `s` et qui renvoie le nombre d'occurrences (c'est-à-dire le nombre d'apparitions) de `c` dans `s`. La fonction doit avoir une complexité linéaire en $n$, la longueur de la chaîne `s`.

On propose :

```
def nbCaracteres(c, s):
    oc = 0
    for x in s:
        if (x == c):
            oc = oc + 1
    return oc
```

**3.** Pour déterminer la liste des caractères utilisés à l'intérieur d'une chaîne s on utilise la fonction définie ci-dessous :

```
1 def listeCaracteres(s:str):
2     listeCar = []
3     n = len(s)
4     for i in range(n):
5         c = s[i]
6         if not(c in listeCar):
7             listeCar.append(c)
8     return listeCar
```

Que renvoie cette fonction lorsque `s='abaabaca'` ? Expliquer succinctement le principe de fonctionnement de cette fonction.

Cette fonction renvoie `['a', 'b', 'c']`. Pour chaque caractère de `s` lus dans l'ordre des indices croissant, elle vérifie s'il n'est pas déjà présent dans `listeCar` (initialement vide), et l'y ajoute si ce n'est pas le cas.

**4.** En fonction de la longueur $n$ de la chaîne et du nombre $k$ de caractères distincts dans celle-ci, déterminer la complexité asymptotique dans le pire des cas de la fonction de la question Q3. Par exemple pour `s='abaabaca'`, on a $n = 8$ et $k = 3$. On négligera la complexité des `append` mais pas celle des tests d'appartenance de la forme `i in L`. Autrement dit, la ligne 7 est considérée comme étant de complexité constante et la ligne 6 de complexité linéaire en la longueur de la liste `listeCar`.

Par définition, on a nécessairement $n \geq k$. La complexité asymptotique dans le pire cas est un $O(n \times k)$, par exemple atteinte pour une chaîne de la forme `'abcabcabc...'` où les $k$ caractères se trouvent en début de chaîne. En effet, les $k$ premières itérations auront un coût dominé par la ligne 6, (en $O(i)$ à l'itération $i$), pour un coût total à constantes près de $\sum_{i=0}^{k-1} i$, soit un $O(k^2)$. Le coût des $n-k$ itérations suivantes est toujours dominé par la ligne 6, de coût $O(k)$. Au total, le coût est un $O(k^2) + O((n-k) \times k) = O(k^2) + O(n \times k - k^2) = O(n \times k)$.

**5.** On définit alors une fonction `analyseTexte(s:str)->list` :

```
1 def analyseTexte(s:str):
2     R = []
3     l = listeCaracteres(s)
4     for i in range(len(l)):
5         c = l[i]
6         R.append((c, nbCaracteres(c, s)))
7     return R
```

Expliquer ce que fait cette fonction et donner la valeur renvoyée par la commande :
`analyseTexte('babaaaabca')`

Cette fonction compte le nombre d'occurrence de chacun des caractères apparaissant dans son argument `s`, et renvoie le résultat sous forme d'une liste de tuples. La commande `analyseTexte('babaaaabca')` renvoie `[('b', 3), ('a', 6), ('c', 1)]`.

**6.** En fonction de la longueur $n$ de `s` et du nombre $k$ de caractères distincts présents dans `s`, (autrement dit $k$ est la longueur de `listeCaracteres(s)`), donner une estimation de la complexité asymptotique dans le pire des cas de la fonction `analyseTexte`.

Par Q 4 la ligne 3 a un coût $O(n \times k)$. La boucle de la ligne 4 effectue $k$ appels à `nbCaracteres` qui sont chacun de coût $O(n)$ (puisque cette fonction parcourt la liste `s`), pour un coût total de la boucle de $O(n \times k)$. Le reste des opérations est de coût constant, et le coût total de la fonction est donc $O(n \times k) + O(n \times k) = O(n \times k)$.

7. Adapter la fonction de la question Q5 pour qu'elle utilise (et renvoie) un dictionnaire. Elle devra avoir une complexité :

   i. linéaire en la longueur $n$ de `s` ;
   ii. indépendante de $k$, le nombre de caractères distincts présents dans `s`.

   De plus, cette fonction devra impérativement ne parcourir qu'une seule fois la chaîne de caractères. On admettra qu'un test d'appartenance d'une clé à un dictionnaire se fait à coût constant. Par exemple, `analyseTexte('abracadabra')` renverra :
   `{'a':5, 'b':2, 'r':2, 'c':1, 'd':1}`

   Par les contraintes de la question, on ne peut plus faire appel à `listeCaracteres`. On propose :

```
def analyseTexte2(s):
    R = {}
    for x in s:
        if x not in R:
            R[x] = 1
        else:
            R[x] = R[x] + 1
    return R
```

### Compression

La compression par codage arithmétique consiste à représenter une chaîne de caractères s par un nombre réel déterminé à l'intérieur de l'intervalle $[0, 1[$.
Initialement, on attribue à chaque caractère utile une portion de l'intervalle $[0, 1[$ proportionnelle à sa fréquence d'occurrences. Par exemple, pour un alphabet à 5 lettres `'abcde'`, on pourrait avoir un tableau comme ci-dessous :

| Caractère | `'a'` | `'b'` | `'c'` | `'d'` | `'e'` |
|---|---|---|---|---|---|
| Fréquence | 0.2 | 0.1 | 0.2 | 0.4 | 0.1 |
| Intervalle | $[0, 0.2[$ | $[0.2, 0.3[$ | $[0.3, 0.5[$ | $[0.5, 0.9[$ | $[0.9, 1[$ |

La chaîne de caractères `s` est codée en partant de l'intervalle $[0, 1[$. À chaque caractère successif de celle-ci, on affine cet intervalle en ne considérant que la portion correspondant au caractère lu.
Si par exemple la chaîne à coder est `s = 'dac'` :

— on obtient d'abord l'intervalle $[0.5, 0.9[$ correspondant au caractère `'d'` ;
— le caractère `'a'` détermine alors le sous-intervalle $[0.50, 0.58[$ de $[0.5, 0.9[$ correspondant à la portion associée au caractère `'a'`.
— le caractère `'c'` détermine enfin l'intervalle $[0.524, 0.540[$.

La figure qui suit illustre ce processus.

8. En considérant la table des fréquences précédente, proposer l'intervalle correspondant à la chaîne `s='bac'`.

   Un calcul sans intérêt donne $[0.206, 0.21[$.

On suppose disposer d'une fonction `codeCar(car:str,g:float,d:float)->(float,float)` qui prend en argument un caractère `car` et les deux extrémités d'un intervalle $[g, d[$ et qui renvoie un tuple composé des extrémités du sous-intervalle de $[g, d[$ déterminé par le caractère `car`. En reprenant l'illustration précédente `codeCar('b',0,1)` produit `(0.2,0.3)` et `codeCar('a',0.5,0.9)` produit `(0.5,0.58)`.

9. Écrire une fonction `codage(s:str)->(float,float)` prenant en argument la chaîne `s` et fournissant en réponse le tuple `(g,d)` constitué des deux extrémités de l'intervalle $[g, d[$ produit par l'algorithme de codage précédent.

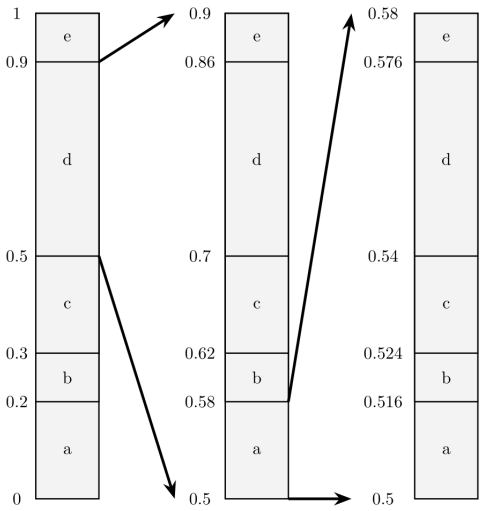



FIGURE 2 – Encodage de `s = 'dac'`.

On propose :

```
def codage(s):
    g, d = 0., 1.
    for x in s:
        g, d = codeCar(x, g, d)
    return g, d
```

Le codage arithmétique consiste alors à coder la chaîne `s` par un flottant `x` choisi arbitrairement à l'intérieur de l'intervalle $[g, d[$.

### Décodage

Pour effectuer le décodage d'un flottant `x`, il suffit de repérer dans quelle succession d'intervalles celui-ci se trouve. À titre d'exemple, reprenons le tableau précédent et considérons le nombre `x = 0.123`

| Caractère | `'a'` | `'b'` | `'c'` | `'d'` | `'e'` |
|---|---|---|---|---|---|
| Fréquence | 0.2 | 0.1 | 0.2 | 0.4 | 0.1 |
| Intervalle | $[0, 0.2[$ | $[0.2, 0.3[$ | $[0.3, 0.5[$ | $[0.5, 0.9[$ | $[0.9, 1[$ |

Puisque `x` appartient à l'intervalle $[0, 0.2[$, le premier caractère est un `'a'`. Puisque `x` appartient au sous-intervalle $[0.1, 0.18[$, le caractère suivant est un `'d'`, etc.

**10.** Déterminer le caractère qui suit `'ad'` dans la chaîne codée par `x = 0.123` en spécifiant le sous-intervalle qui a permis de décoder ce caractère.

L'intervalle correspondant à `'ad'` est $[0.1, 0.18[$, et 0.123 appartient au sous-intervalle $[0.116, 0.124[$ de ce dernier, soit au caractère `'b'`.

**11.** Dans le cadre de l'exemple de cette partie, indiquer deux chaînes qui peuvent correspondre au flottant `0.2`. Expliquer par une phrase ce qui est à l'origine de cette ambiguïté.

Le flottant `0.2` peut à la fois correspondre à la chaîne `'b'` et la chaîne `'ba'`. Cette ambiguïté est liée au fait que le codage n'inclut pas la longueur de la chaîne, et que des intervalles de chaînes de longueurs différentes ne sont pas toujours disjoints.

Une solution possible pour résoudre le problème précédent consiste à introduire un caractère nouveau signifiant la fin de la chaîne de caractères. Nous conviendrons de désigner ce caractère par `'#'`. Ce caractère se voit attribuer une plage non vide au voisinage de 0.

Dans la suite, on suppose que la table des fréquences est adaptée de sorte à prendre en compte la présence de ce nouveau caractère. On suppose disposer, en plus de la fonction `codeCar(car,g,d)` précédente, d'une fonction `decodeCar(x:float,g:float,d:float)->str` qui détermine le caractère correspondant à la valeur `x` quand celle-ci est comprise dans la plage de $[g,d[$. Par exemple `decodeCar(0.123,0,1)` donne `'a'` tandis que `decodeCar(0.123,0,0.2)` donne le caractère `'d'`.

**12.** Écrire une fonction `decodage(x:float)->str` produisant la chaîne de caractères `s` déterminée par la valeur de `x` (avec le caractère `'#'` compris).

Le plus simple est de construire la chaîne de caractère par concaténations successive, mais ceci peut être inefficace en Python. Cela étant dit, puisque le sujet n'impose pas un coût cible pour la fonction, on peut se contenter d'une version de coût quadratique en la longueur du résultat :

```
def decodage(x):
    g, d = = 0., 1.
    c = decodeCar(x, g, d)
    s = c
    while c != '#':
        g, d = codeCar(c, g, d)
        c = decodeCar(x, g, d)
        s = s + c
    return s
```

**13.** Quel obstacle pratique s'oppose à la mise en œuvre de la compression par codage arithmétique telle que décrite dans ce sujet ?

Les nombres flottants « `float` » ont une précision limitée, ce qui fait qu'à partir d'une certaine longueur de chaîne il deviendra impossible de distinguer des intervalles devant être disjoints. Ce codage ne peut donc s'employer que pour des chaînes de taille bornée, ou doit utiliser un autre type que `float` pour représenter celles-ci.

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

### Modélisation du canal de communication par un graphe

Dans cette partie, nous allons désormais considérer que le message compressé par Alice a été envoyé au travers d'un canal de communication. À cette fin, et indépendamment de la phase de compression étudiée dans la première partie, le message a subi une deuxième phase de transformation, dite d'encodage (*cf.* Figure 1).
Le message envoyé sur le canal est une suite de *symboles* à valeurs dans un *alphabet*, noté $\Sigma$, comportant $K$ symboles. Le choix d'un alphabet efficace n'est pas l'objet de notre étude et constitue un sujet à part entière.
Suite au passage dans le canal de communication, le message envoyé subit une *altération* de sorte que Bob reçoit une séquence de symboles de $\Sigma$ qui ne correspond pas nécessairement à celle qui a été émise.
Dans cette partie, nous allons voir une approche permettant à Bob de décoder le message reçu et de potentiellement corriger quelques erreurs liées à la transmission du message et à la connaissance *a priori* de la propension du canal de communication à altérer les symboles du message lors de la transmission.
La modélisation proposée est la suivante :

— Bob observe une suite de N symboles $obs_0, \dots, obs_{N-1}$, que nous allons représenter par une liste Python `Obs =[obs0, ...]`.
— Pour simplifier, on supposera que l'alphabet $\Sigma$ est un ensemble de $K$ entiers consécutifs commençant à 0, de sorte que $\Sigma = [\![0, K-1]\!]$. Par exemple si $K = 3$ et $N = 8$, un message valide reçu par Bob pourrait être `[2,0,0,2,1,1,0,0]`.
— Chacun des symboles observés $obs_t$ correspond à l'altération d'un symbole $s_t$ envoyé par Alice. On note $[s_0, \dots, s_{N-1}]$ le message original ; pour reprendre l'exemple précédent, Alice pourrait avoir envoyé `[2,0,0,2,1,1,2,0]`.
— On connaît, pour chaque paire $(i,j) \in \Sigma^2$, la probabilité $E_{i,j}$ que le canal altère le symbole $j$ en un symbole $i$. On stocke ces probabilités dans une liste de listes `E` ; autrement dit, `E[i][j]` est la probabilité conditionnelle d'observer le symbole $i$ sachant que le symbole $j$ a été émis. Ici (par exemple) on pourrait

considérer :

$$E = \begin{pmatrix} 0.7 & 0.2 & 0.3 \\ 0.2 & 0.7 & 0.1 \\ 0.1 & 0.1 & 0.6 \end{pmatrix}$$

représentée par la liste de listes : `E = [[0.7,0.2,0.3],[0.2,0.7,0.1],[0.1,0.1,0.6]]`. Le fait que `E[2][0]` vaille `0.1` signifie donc que la probabilité que le symbole observé par Bob soit un 2 sachant qu'Alice a émis un 0 est de 0.1.

— On suppose également que le symbole courant $s_t$ envoyé par Alice a une incidence sur le symbole suivant $s_{t+1}$ qu'elle peut envoyer, au même titre que dans une langue comme le français, la probabilité d'observer un `'t'` dans un mot n'est pas la même suivant que le caractère précédent est un `'e'` ou un `'z'`.
Ainsi pour chaque couple de symboles $(i, j) \in \Sigma^2$, on suppose que l'on connaît la probabilité d'émettre le symbole $j$ à l'instant $t+1$ sachant que le symbole $i$ a été émis à l'instant $t$. On suppose également que cette probabilité ne dépend pas de $t$.
L'information concernant ces probabilités de transition d'un symbole à l'autre peut se stocker dans une matrice $P$ de taille $K \times K$, que l'on représente informatiquement par une liste de listes `P`. Chaque entrée `P[i][j]` donne la probabilité qu'Alice émette le symbole $j$ à l'instant $t+1$ sachant qu'elle a émis le symbole $i$ à l'instant $t$. En d'autres termes, `P[i][j]` $= P_{s_t=i}(s_{t+1} = j)$.
On prendra ici à titre d'exemple :

$$P = \begin{pmatrix} 0.3 & 0.2 & 0.5 \\ 0.4 & 0.4 & 0.2 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$$

représentée par la liste de listes : `P=[[0.3,0.2,0.5],[0.4,0.4,0.2],[0.2,0.3,0.5]]`.
Le fait que `P[2][0]` vaille `0.2` signifie donc que la probabilité que le symbole envoyé par Alice à l'instant $t+1$ soit un 0 sachant que celui envoyé à l'instant $t$ est un 2 vaut 0.2.

Nous allons désormais nous intéresser au problème du décodage : étant donné la liste `Obs =[obs0, ...]` des symboles observés par Bob, quelle séquence $\hat{s}_0, \dots, \hat{s}_{N-1}$ est la plus probable ? En d'autres termes, $\hat{s}_0, \dots, \hat{s}_{N-1}$ est l'estimation la plus probable faite par Bob du message d'origine $s_0, \dots, s_{N-1}$ étant données les observations $obs_0, \dots, obs_{N-1}$.
La modélisation précédente peut se représenter à l'aide d'un graphe défini comme suit (voir Figure 3 pour un exemple) :

— On crée un sommet $S_{i,j}$ pour chaque symbole possible $0 \leqslant i \leqslant K-1$ et chaque indice d'observation $0 \leqslant j \leqslant N-1$. Chaque couche verticale dans le graphe correspond à un caractère dans le message. Chaque strate horizontale correspond à un symbole.
— Au niveau de la $j$-ème couche verticale, les sommets $S_{i,j}$ pour $j < N-1$ ont pour successeurs les états $S_{k,j+1}$ pour tous les symboles $k$ possibles.
— Par commodité, on ajoute un état source $\sigma$ correspondant au début du message décodé et un état cible $\tau$ correspondant à la fin du message, ces états étant respectivement reliés à la première et la dernière couche.
— Le décodage du message envoyé par Alice correspond à un chemin entre $\sigma$ et $\tau$ dans ce graphe. A chaque sommet du chemin correspond une lettre décodée. Par exemple, le chemin passant par $S_{0,0}, S_{2,1}, S_{0,2}, S_{1,3}$ correspond au décodage de `[0, 2, 0, 1]`.

**14.** En fonction de $N$ et de $K$, donner le nombre de sommets et d'arcs du graphe illustré par la Figure 3. On ne comptera pas les sommets source $\sigma$ et cible $\tau$, ni les arcs partant du sommet source $\sigma$ ni ceux arrivant à la cible $\tau$.

On a $N$ couches de $K$ sommets, et $K^2$ arcs entre chacune de ces couches, soit $N \times K$ sommets et $(N-1)K^2$ arcs.

On choisit désormais de pondérer chaque arc par la probabilité de transiter par cet arc. Autrement dit :

— Les arcs issus de la source $\sigma$ vers $S_{i,0}$ sont pondérés par $E_{obs_0}, i$ la probabilité d'observer le symbole $obs_0$ sachant que le symbole $i$ a été émis par Alice.
— Les arcs arrivant à la cible $\tau$ sont pondérés par 1 (en fin de message, on transite forcément vers l'état final).
— Les arcs internes entre $S_{i,j}$ et $S_{k,j+1}$ sont pondérés par la probabilité $E_{obs_{j+1},k} P_{i,k}$.

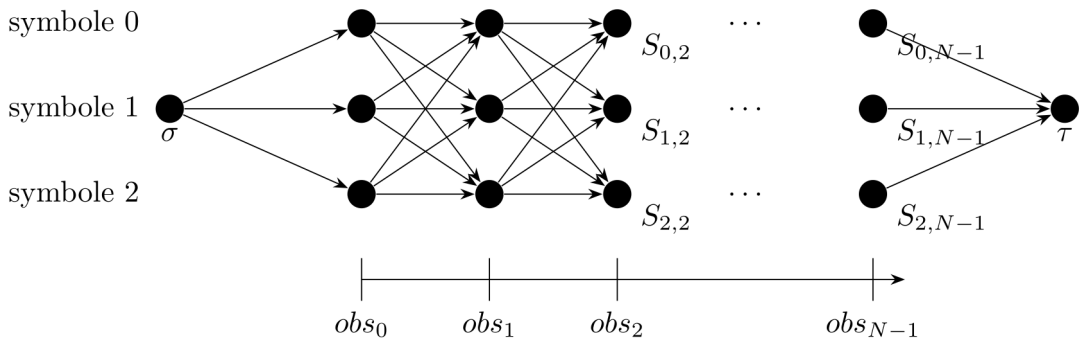
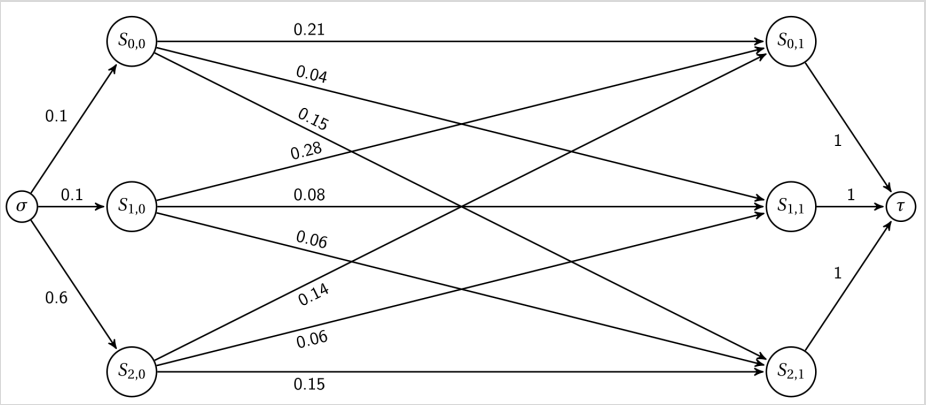



FIGURE 3 – Illustration du modèle de décodage considéré ici.

La probabilité d'un chemin $\sigma S_{i_0,0} S_{i_1,1} \cdots S_{i_{N-1},N-1} \tau$ entre $\sigma$ et $\tau$ est le produit des probabilités des arcs qui le composent. L'objectif va être de trouver le chemin de probabilité maximale dans ce graphe entre le sommet source $\sigma$ et le sommet cible $\tau$.

**15.** On suppose que Bob a observé la séquence `[2,0]`. En utilisant les matrices $E$ et $P$ données dans l'énoncé (avec $K = 3$), construire le graphe pondéré associé à ce message de longueur $N = 2$. Les arcs entre les sommets devront être pondérés par les probabilités correspondantes.

Emprunté à un corrigé de Matthieu Solnon.




**16.** On revient dans le cas général, $N$ et $K$ sont désormais quelconques. Indiquer combien il existe de chemins entre $\sigma$ et $\tau$ (un ordre de grandeur utilisant la notation $O$ ou $\Theta$ est accepté). Préciser si un algorithme d'exploration exhaustive est envisageable dans ce cas.

Il y a $K$ chemins possibles entre $\sigma$ et l'un des sommets initiaux, puis $K$ façons de prolonger un chemin d'une couche $0 \leq i < N$ à la couche $i+1$ (correspondant aux $K$ arcs issus du dernier sommet du chemin), et une unique façon de rejoindre $\tau$ depuis la dernière couche. Au total : $K \times K^{N-1} = K^N$ chemins. Cette croissance exponentielle en $N$ fait qu'une exploration exhaustive est clairement inenvisageable, même pour de petites valeurs de $N$ (tant que $K > 1$).

### Stratégie gloutonne

Pour pouvoir implémenter correctement la recherche du chemin de probabilité maximale, il est utile de disposer d'une fonction auxiliaire qui sera utilisée dès que nécessaire.

**17.** Pour une liste `liste`, on appelle argument du maximum et on note `argMax` tout indice $i$ tel que `liste[i]` soit maximal. Proposer une fonction `maximumListe(liste:[float])->(float,int)` qui prend en entrée une liste de nombres et qui renvoie la valeur du maximum de la liste ainsi que le plus petit argument du maximum, *i.e.* le premier indice auquel cette valeur maximale apparaît.

On suppose l'argument de longueur non nulle, et propose :

```
def maximumListe(liste):
    mi = 0
    mv = liste[0]
    for i in range(1,len(liste)):
        if liste[i] > mv:
            mi = i
            mv = liste[i]
    return mv, mi
```

On souhaite appliquer un algorithme glouton pour trouver le chemin de probabilité maximale entre le sommet source $\sigma$ et le sommet cible $\tau$. On rappelle qu'un algorithme glouton cherche, à chaque étape, à faire le choix localement optimal. Ici, si à une étape on se retrouve au sommet $S_{i,j}$ , il s'agit de choisir l'arc de plus forte probabilité partant de ce sommet.
Dans un premier temps on écrit une fonction :
`initialiserGlouton(Obs:[[int]], E:[[float]], K:int)->int`
qui permet d'initialiser l'algorithme glouton en trouvant le sommet le plus probable parmi $S_{i,0}$ pour i variant entre 0 et $K-1$. Pour cela il faut regarder la colonne `Obs[0]` de `E` et relever l'indice de la plus grande valeur :

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

**18.** Proposer une fonction `glouton(Obs:[int],P:[[float]],E:[[float]],K:int,N:int)->[int]` qui renvoie la liste d'états obtenue par l'approche gloutonne. Même si cela n'est pas nécessaire, $K$, $N$ seront des arguments de cette fonction.

On propose :

```
def glouton(Obs, P, E, K, N):
    st = [initialiserGlouton(Obs, E, K)]
    for j in range(N-1):
        i = st[j]
        probas = [E[Obs[j+1]][k] * P[i][k] for k in range(K)]
        _, i = maximumListe(probas)
        st.append(i)
    return st
```

**19.** En fonction de $K$ et de $N$ , quelle est, en ordre de grandeur, la complexité temporelle asymptotique de l'approche gloutonne ?

Chacune des $N-1$ itérations de l'unique boucle ci-dessus construit une liste de longueur $K$ et effectue une recherche dans cette liste de coût linéaire en sa longueur ; le reste des opérations étant de coût linéaire en $K$ (l'appel à `initialiserGlouton`) ou constant, le coût total est n $O(N \times K)$.

**20.** Indiquer le chemin renvoyé par l'algorithme glouton appliqué à la Figure 4. Conclure quant à l'optimalité de l'approche.

Le chemin est `[0, 0]`, de probabilité 0.3, alors que le chemin `[1, 0]` est de probabilité 0.36. L'algorithme glouton ne renvoie donc pas un chemin de probabilité maximale.

### Stratégie de programmation dynamique

**21.** Expliquer en quoi rechercher un chemin de probabilité maximale pourrait se transformer en un problème de recherche de plus court chemin dans un graphe pondéré à poids positifs. Préciser alors quel algorithme pourrait être utilisé.

On peut réduire ce problème à celui d'une recherche de plus-court chemin entre $\sigma$ et $\tau$ dans le graphe où l'on a pondéré chaque arc de probabilité $x$ par $-\log(x)$ : le poids dans ce nouveau graphe d'un chemin de probabilité $p := \prod_{i=0}^{N-1} p_i$ dans le graphe initial est de $\ell_p := -\sum_{i=0}^{N-1} \log(p_i) = -\log(p)$. Puisque $x \mapsto -\log(x)$ est décroissante

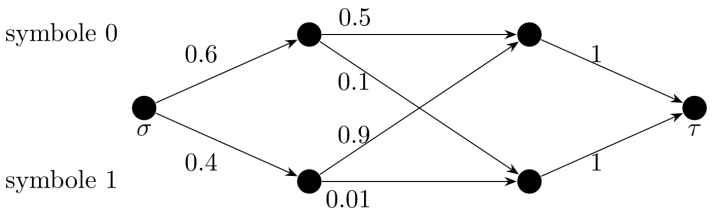



FIGURE 4 – Que donne l'algorithme glouton ici ?

et positive sur $[0,1]$, un chemin de probabilité maximale dans le graphe original est bien donné par un chemin de poids minimum (un plus-court-chemin) dans le nouveau graphe, et puisque ces poids sont positifs on peut par exemple utiliser l'algorithme « de Dijkstra » pour calculer un tel chemin.

Les algorithmes évoqués à la question précédente ne sont cependant pas optimaux dans ce cas. L'algorithme optimal est dû à Andrew Viterbi et date de 1967. Il repose sur le paradigme de la programmation dynamique. On appelle $T_{i,j}$ la valeur de probabilité maximale entre la source et l'état $S_{i,j}. On peut alors établir l'équation de programmation dynamique suivante :

— $T_{i,j} = \max_{k\in[\![0,K-1]\!]} \left\{T_{k,j-1} \times P_{k,i} \times E_{obs_j,i}\right\}$ si $N-1 \geq j > 0$
— $T_{i,0} = E_{obs_0,i}$

Comme on cherche également à obtenir la valeur des états correspondant au chemin optimal, on maintient également le tableau des prédécesseurs suivant :

— $argT_{i,j} = \arg\max_{k\in[\![0,K-1]\!]} \left\{T_{k,j-1} \times P_{k,i} \times E_{obs_j,i}\right\}$ si $N-1 \geq j > 0$
— $argT_{i,0} = -1$

La valeur $-1$ du deuxième tableau est purement conventionnelle et ne sert qu'à représenter l'état source $\sigma$ qui ne correspond pas à une observation.
On suppose avoir codé une fonction :
`initialiserViterbi(E:[[float]], Obs0:int, K:int, N:int)->([[float]],[[int]])`
qui prend en entrée la matrice `E` d'émission, la valeur de la première observation `Obs0`, le nombre d'éléments `K` de $\Sigma$, le nombre d'observations `N` et qui renvoie deux tableaux (liste de listes) `T` et `argT` vérifiant les caractéristiques suivantes :

— `T` et `argT` sont de dimensions $K$ lignes par $N$ colonnes,
— `T[i][0]` contient la valeur de $E_{obs_0,i}$,
— `argT[i][0]` contient la valeur `-1`
— les autres valeurs de `T` et `argT` sont à `0`

En voici une implémentation possible :

```
def initialiserViterbi(E, Obs0, K, N):
    probasInitiales = [E[Obs0][i] for i in range(K)]
    T = [[0 for j in range(N)] for i in range(K)]
    argT = [[0 for j in range(N)] for i in range(K)]
    for i in range(K):
        T[i][0] = probasInitiales[i]
        argT[i][0] = -1
    return T, argT
```

**22.** Proposer une fonction (méthode de bas en haut de programmation dynamique)
`construireTableauViterbi(Obs:[int], P:[[float]], E:[[float]], K:int, N:int)->([[float]],[[int]])`
qui prend comme arguments la liste des observations `Obs`, la matrice des probabilités de transition `P` et la matrice des probabilités `E` et renvoie les deux listes de listes `T` et `argT` de taille $K \times N$.

Il suffit de recopier la formule… On propose :

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

**23.** L'algorithme de Viterbi codé en Python et appliqué à un message en entrée donne les tableaux T et argT suivants. Indiquer la séquence d'états la plus probable.

$$T = \begin{pmatrix} 0.1 & 0.084 & 0.018 & 0.00053 & 0.00021 & 8.9e-05 & 8.7e-05 & 1.8e-05 \\ 0.1 & 0.036 & 0.0054 & 0.00041 & 0.0011 & 0.00031 & 2.5e-05 & 3.5e-06 \\ 0.6 & 0.09 & 0.014 & 0.0053 & 0.00026 & 2.2e-05 & 1.9e-05 & 1.3e-05 \end{pmatrix}$$

$$argT = \begin{pmatrix} -1 & 2 & 0 & 0 & 2 & 1 & 1 & 0 \\ -1 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\ -1 & 2 & 0 & 0 & 2 & 1 & 1 & 0 \end{pmatrix}$$

On cherche l'état final le plus probable dans $T$, qui est ici 0, puis l'on remonte jusqu'à l'état initial en suivant les indices donnés par *argT*, ce qui donne (à l'envers) : $0, 0, 1, 1, 2, 0, 0, 2$, et à l'endroit : $2, 0, 0, 2, 1, 1, 0, 0$.

**24.** En fonction de $K$ et de $N$, donner l'ordre de grandeur de la complexité temporelle de l'approche de programmation dynamique, ainsi que la complexité spatiale.

La complexité spatiale est dominée par les tailles des tableaux `T` et `argT`, soit un $O(N \times K)$ (puisque le tableau `probas` créé à la ligne 5 n'est que de taille $O(K) = O(N \times K)$). La complexité temporelle est dominée par les $N \times K$ itérations des lignes 5 et 6, chacune de coût $O(K)$, soit un coût de $O(N \times K^2)$ au total (puisque la création des tableaux `T` et `argT` n'est que de coût $O(N \times K) = O(N \times K^2)$).

*Fin du sujet*