Correction Mines-Ponts 2024

1. Compression du message

1. `'a'` peut être représenté `0`, `'b'` par `10` et `'c'` par `11`.

   Il n'y aura pas d'ambiguité : pour lire un caractère, on commence par lire un bit.
   Si c'est `0`, c'est que le caractère est un `'a'` stocké sur un seul bit.
   Si c'est `1`, le caractère est stocké sur 2 bits, et la lecture du bit suivant nous indique si c'est un `'b'` ou un `'c'`.

2. ```
   def nbCaracteres(c,s):
       nb = 0      # Compteur du nombre d'occurences
       n = len(s)  # Longueur de la chaine

       for i in range(n):  # On parcourt chaque caractère de la chaine
           if c == s[i]:       # Si c'est le bon
               nb += 1         # on incrémente le compteur

       return nb
   ```

3. Elle parcourt les caractères de la chaine `s`.

   Pour chaque caractère, s'il n'a pas déjà été rencontré (test de la ligne 6), on l'ajoute à la fin de la liste des caractères "rencontrés".

   Pour l'exemple `s='abaabaca'`, on obtiendra la liste `['a', 'b', 'c']`
   (`'a'` sera rencontré au premier passage, `'b'` au second et `'c'` au septième).

4. Les instructions des lignes 2, 3, 5, 7 et 8 sont en temps constants.

   La boucle de la ligne 4 va être effectuée n fois (une fois par caractère de `s`) et à chaque fois on va faire le test de la ligne 6, qui demande un parcours de la liste `listeCar` pour vérifier que `c` est nouveau ou pas.
   S'il y a k caractères distincts, dans le pire cas le test aura un coût de l'ordre de k (comparer avec chacune des valeurs).

   La complexité asymptotique dans le pire cas sera de l'ordre de nk.

5. Elle commence par créer la liste des caractères utilisés puis pour chaque caractère, crée un couple avec le caractère et son nombre d'occurrences et renvoit la liste des couples.

   Par exemple avec `'babaaaabca'` on obtiendra `[ ('b',3), ('a',6), ('c',1) ]`.

   Remarque: le premier caractère rencontré par `listeCaracteres` est un `'b'` !

6. La ligne 3 a une complixité en nk.

   On a ensuite un boucle à la ligne 4 qui va être effectuée k fois (une fois par caractère distinct).
   Cette boucle execute des instructions de temps constant, *sauf* l'appel à `nbCaracteres` qui a une complicité en n (la taille de la chaine).

   La boucle de la ligne 4 a donc une complexité en nk également, donc `analyseTexte` est en nk.

7. ```
   def analyseTexte(s):
       n = len(s)      # Longueur de la chaine
       occ = {}        # Dictionnaire pour stocker les occurences

       for i in range(n):  # On parcourt chaque caractère de la chaine
   ```

```
        c = s[i]
        if c in occ:    # On regarde si le caractère est déja vu
            occ[c] += 1     # OUI: on incrémente
        else:
            occ[c] = 1      # NON: on crée l'entrée

    return occ
```

La complexité est en n : rechercher si un caractère est déjà vu est maintenant en temps constant comme le reste de la boucle.

8. `SELECT DISTINCT auteur FROM corpus`

9. ```
SELECT symbole, SUM(nombreOccurrences)*1.0/total AS frequence
    FROM caractere
        JOIN occurences ON caractere.idCar = occurences.idCar
        JOIN corpus ON corpus.idLivre = occurences.idLivre
        JOIN (SELECT SUM(nombreCaracteres) AS total
                    FROM corpus WHERE langue="Francais")
    WHERE langue="Francais"
    GROUP BY symbole
```

   Il y a deux aggrégations à faire : sommer les nombres de caractères (sur tous les livres en français) et sommer les nombres d'occurences (par caractère).

   Les deux ne peuvent pas se faire simultanément : il est donc nécessaire d'en "cacher" une dans une requête imbriquée.

10. b correspond à l'intervalle $[0.2; 0.3[$. Cela se découpe en

    - ba sur $[0.20; 0.22[$. (20% du début de $[0.2; 0.3[$)
    - bb sur $[0.22; 0.23[$. (10% suivants)
    - bc sur $[0.23; 0.25[$. (20% suivants)
    - bd sur $[0.25; 0.29[$. (40% suivants)
    - be sur $[0.29; 0.30[$. (10% restants)

    ba correspond à $[0.20; 0.22[$, qui se découpe en

    - baa sur $[0.200; 0.204[$. (20% du début de $[0.20; 0.22[$)
    - bab sur $[0.204; 0.206[$. (10% suivants)
    - bac sur $[0.206; 0.210[$. (20% suivants)
    - bad sur $[0.210; 0.218[$. (40% suivants)
    - bae sur $[0.218; 0.220[$. (10% restants)

    Donc bac correspond à $[0.206; 0.210[$.

11. ```
def codage(s):
    n = len(s)
    g,d = 0,1   # Intervalle initial

    # On parcourt chaque caractère de la chaine
    for i in range(n):
```

```
        c = s[i]

        # On calcule le nouvel intervalle avec le caractère courant
        g,d = codeCar(c, g, d)

    return g,d
```

12. ad correspond à $[0.10; 0.18[$, qui se découpe en

    - ada sur $[0.100; 0.116[$. (20% du début de $[0.10; 0.18[$)
    - adb sur $[0.116; 0.124[$. (10% suivants)
    - adc sur $[0.124; 0.140[$. (20% suivants)
    - add sur $[0.140; 0.172[$. (40% suivants)
    - ade sur $[0.172; 0.180[$. (10% restants)

    0.123 est dans le second intervalle, donc le caractère suivant est un `'b'`.

13. Si on reprend le détail de la question 10, on voit que 0.2 peut correespondre à `'b'`, à `'ba'`, à `'baa'`, …

    Si on ne sait pas quelle est la longueur de la chaine, on peut imaginer des partitions de $[0, 1[$ de plus en plus petite : un $x \in [0; 1[$ peut être vu comme un élément de n'importe laquelle des partitions… Donc être identifié comme une chaine de longueur arbitraire.

14. ```
    def decodage(x):
        s = ''      # La chaine à renvoyer
        g,d = 0,1   # Intervalle initial

        fini = False
        while not fini:     # On continue jusqu'à trouver le caractère final
            c =  decodeCar(x, g, d)

            if c == '#':
                fini = True
            else:
                s = s + c   # On ajoute le nouveau caractère
                # On calcule le nouvel intervalle pour le caractère suivant
                g,d = codeCar(c, g, d)

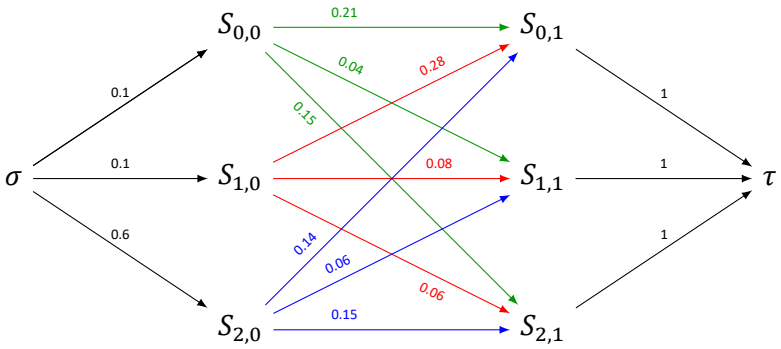
        return s
    ```

2. Décodage du message

15. Il y a NK sommets (K sommets pour le caractère 0, K sommets pour le caractère 1, … , K sommets pour le caractère $N-1$).

    Pour i allant de 0 à $N-2$, chacun des K sommets de la couche verticale i a un arc vers chacun des K sommets de la couche verticale $i+1$. Il y a donc $(N-1)K^2$ arcs.

16. On obtient le graphe suivant:
    ```



Quelques explications :

- pour les trajets $\sigma \to S_{i,0}$, l'énoncé indique de pondérer par $E_{obs_0,i}$, c'est à dire $E_{2,i}$ puisque le premier caractère obervé est un 2. On obtient donc les coefficients de la ligne "2" de la matrice E.
- pour les trajets $S_{i,0} \to S_{k,1}$, l'énoncé indique de pondérer par $E_{obs_1,k}P_{i,k}$, c'est à dire $E_{0,k}P_{i,k}$ puisque le second caractère obervé est un 0.

17. À chaque niveau vertical, on a K possibilités (aller vers $S_{0,j}$, $S_{1,j}$, …, $S_{K-1,j}$) , sauf au dernier où il n'y en a plus qu'une (aller vers τ).

    Il y a donc K^{N-1} chemins. Si N devient trop grand, l'exploration exhaustive n'est plus envisageable (croissance exponentielle en N)

18.
```python
def maximumListe(liste):
    n = len(liste)
    M = liste[0]    # le maximum (temporaire)
    arg = 0         # sa position

    for i in range(1,n):
        if liste[i] > M:    # On trouve un nombre strictement plus grand
            M = liste[i]    # On note ce nouveau candidat
            arg = i

    return M, arg
```

Remarque: en cas de maximum "multiple", on veut l'indice le plus petit possible, il faut donc bien mettre une inégalité stricte dans la boucle. Avec une inégalité large on aurait le dernier *indice où le max apparait.*

19.
```python
def glouton(Obs, P, E, L, N):
    courant = initialiserGlouton(Obs, E, K)
    chemin = [ courant ]        # Caractère 0

    for j in range(0,N-1):  # Recherche du caractère j+1
        # Proba d'aller de "i = courant" à "k" avec l'observation "obs[j+1]"
        i = courant
        obs = Obs[j+1]

        probas = [ E[obs][k] * P[i][k]  for k in range(K) ]

        # Le max determine le nouveau "courant"
        s, courant = maximumListe(probas)
```

```
        chemin.append(courant)

    return chemin
```

20. Cela revient à recherche N fois (une fois par observation) un maximum dans une liste de K valeur.

    Cette recherche de maximum étant linéaire, la complexité est en NK.

21. L'initialisation fait choisir l'arc de poids 0.6.
    On choisit ensuite l'arc de poids 0.5, ce qui donne un chemin de probabilité 0.03.

    Or choisir l'arc 0.4 puis l'arc 0.9 donnerait un chemin meilleur de probabilité 0.036.

22. En passant au logarithme, maximiser $P = \prod p_k$ (où p_k noterait la probabilité de l'arc k d'un chemin) revient à maximiser $\sum \ln p_k$, ou autrement dit, à minimiser $\sum(-\ln p_k)$.

    Si on remplace les poids p_k par les "distances" $(-\ln p_k)$ (qui sont des nombres positifs car les $p_k \in]0;1[$), on est sur un problème de recherche de plus court chemin.

    On peut utiliser l'algorithme de Dijkstra pour résoudre ce problème.

23. ```
    def construireTableauViterbi(Obs, P, E, K, N):
        T, argT = initialiserViterbi(E, Obs[0], K, N)

        # Construction des chemins optimaux jusqu'au niveau j
        for j in range(1,N):
            # Calcul du chemin vers S(i,j)
            for i in range(K):
                probas = [  T[k][j-1] * P[k][i] * E[Obs[j]][i]  for k in range(K)  ]

                # On note le max et son argument
                T[i][j], argT[i][j] = maximumListe(probas)

        return T, argT
    ```

24. Il faut regarder la dernière colonne pour trouver la meilleure proba (ici `1.8e-5`), le message termine par un `0`.

    Dans `argT`, la dernière colonne nous indique qu'on provenait de l'état `0`.

    On suit ensuite dans `argT` les prédecesseurs en remontant les colonnes (de droite à gauche): `1`, `1`, `2`, `0`, `0`, `2`, `-1` (on est alors remonté jusqu'à σ)

    Le message le plus probable serait donc `2 0 0 2 1 1 0 0`.

25. La fonction `initialiserViterbi` commence par créer une liste de taille K, puis deux listes de listes de tailles NK (à l'aide d'une double boucle).
    Il y a ensuite une boucle de taille K faisant des choses en temps constant.

    La complexité temporelle **et** spatiale est donc en NK.

    Ensuite la fonction `construireTableauViterbi` crée (ponctuellement) des listes de taille K et ne fait que modifier les listes doubles déja créées : **la complexité spatiale n'augmente pas et reste de l'ordre de NK.**

    Pour la complexité temporelle, on a une double boucle (nombre d'itérations $\simeq N \times K$) qui crée une liste de taille K, puis recherche un maximum par balayage : chaque passage a donc une complexité de l'ordre de K.

    La complexité temporelle est donc de l'ordre de NK^2.
    ```