Problème 1. Correction orthographique

Dans ce problème, on considère des mots sur un alphabet fini. La longueur d'un mot u est notée $|u|$. Les lettres d'un mot u de longueur n sont notées $u_0, \ldots, u_{n-1}$. Pour $0 \leq i \leq j \leq n$, on note $u_{i,j}$ le sous-mot de u de longueur $(j-i)$ constitué des lettres $u_i, u_{i+1}, \ldots, u_{j-1}$. En particulier, $u = u_{0,n}$ et $u_{i,i}$ est le mot vide (aucune lettre) pour tout i.

1.1 Distances entre des mots

Question 1.1. Distance de Hamming La distance de Hamming entre deux mots de même longueur u et v, notée $\delta_H(u,v)$, est le nombre d'indices i tels que $u_i \neq v_i$.

(a) Quelle est la distance de Hamming entre les mots `plage` et `glace` ?

(b) Si l'on ne considère que des mots écrits avec les 26 lettres minuscules non accentuées de l'alphabet latin, combien y a-t-il de mots à distance au plus d d'un mot de longueur n donné ?

(c) Écrire un algorithme `hamming` qui prend en entrée deux chaînes de caractère u et v et qui renvoie leur distance de Hamming $\delta_H(u,v)$.

Question 1.2. Distance de Levenshtein La distance de Levenshtein entre deux mots u et v (de longueurs potentiellement distinctes), notée $\delta_L(u,v)$, est le nombre minimal de caractères qu'il faut supprimer, insérer ou modifier pour passer de u à v. Par exemple, la distance de Levenshtein entre les mots `pacte` et `plage` est 3 : partant de `pacte`, il suffit d'insérer un `l` en deuxième position (`placte`), de supprimer le `t` (`place`), et de remplacer le `c` par un `g` (`plage`). Il y a d'autres chemins possibles, mais on peut vérifier qu'il faut bien au moins 3 opérations pour passer d'un mot à l'autre.

(a) Quelle est la distance de Levenshtein entre les mots `chapeau` et `crapaud` ?

(b) Montrer que pour tout u et v de mêmes tailles, $\delta_L(u,v) \leq \delta_H(u,v)$.

(c) Montrer que pour tout u et v, $\delta_L(u,v) \leq \max(|u|,|v|)$.

(d) Montrer que la distance de Levenshtein vérifie l'inégalité triangulaire : pour tout u, v et w, $\delta_L(u,v) \leq \delta_L(u,w) + \delta_L(w,v)$.

Question 1.3. Calcul naïf de la distance de Levenshtein

(a) Montrer que si $|u| = m > 0$ et $|v| = n > 0$,

$$\delta_L(u,v) = \min \begin{cases} \delta_L(u_{1,m}, v) + 1 \\ \delta_L(u, v_{1,n}) + 1 \\ \delta_L(u_{1,m}, v_{1,n}) + 1_{u_0 \neq v_0} \end{cases}$$

où $1_{u_0 \neq v_0}$ vaut 1 si $u_0 \neq v_0$ et 0 sinon.

(b) En déduire un algorithme récursif (naïf) de calcul de la distance de Levenshtein entre deux mots.

(c) Analyser la complexité de votre algorithme.

Question 1.4. Programmation dynamique Afin d'améliorer la complexité de l'algorithme précédent, on recourt à la programmation dynamique. Soit u et v deux mots fixés. Pour $0 \leq i \leq m$ et $1 \leq j \leq n$, on note $\delta_{i,j} = \delta_L(u_{0,i}, v_{0,j})$.

(a) Exprimer $\delta_L(u,v)$ en fonction d'un $\delta_{i,j}$.

(b) Pour $0 < i,j \leq n$, exprimer $\delta_{i,j}$ en fonction de différentes valeurs de $\delta_{i',j'}$, en s'inspirant de l'équation (a).

(c) Donner, en fonction de i et j, la valeur des cas de base $\delta_{i,0}$ et $\delta_{0,j}$.

(d) Déduire des questions précédentes un algorithme `levenshtein` qui calcule la distance de Levenshtein entre deux mots, en remplissant itérativement un tableau qui contient la valeur $\delta_{i,j}$ en case (i, j).

(e) Quelle sont les complexités en temps et en mémoire de votre algorithme ?

(f) Comment peut-on améliorer la complexité mémoire de l'algorithme ?

1.2 Correction de mots

Pour effectuer de la correction orthographique, on dispose d'un *ensemble de mots valides*[1]. Cet ensemble est représenté sous la forme d'un arbre dont les arêtes sont étiquetées par des lettres de l'alphabet, et les nœuds peuvent être de deux natures : terminal ou non terminal. Un chemin de la racine vers un nœud définit naturellement un mot (la suite des lettres rencontrées), et l'ensemble des mots valides est donné par l'ensemble des chemins qui aboutissent sur un nœud terminal. Un nœud de l'arbre est identifié de manière unique par le chemin qui y conduit depuis la racine, ou de manière équivalente par le mot qu'il faut lire depuis la racine pour aboutir à ce nœud. Un exemple est donné par la figure 1.

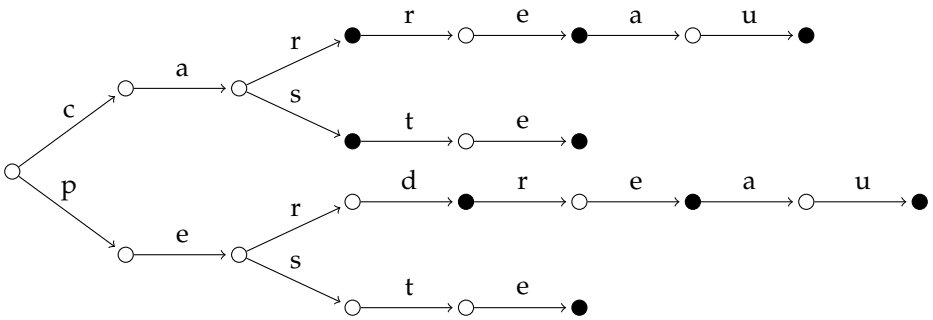

Figure 1: Exemple de représentation d'un ensemble de mots par un arbre. Les nœuds terminaux sont en noir, les non-terminaux en blanc. Par exemple, le mot `cas` appartient à l'ensemble car le nœud atteint en lisant `cas` est terminal, mais pas le mot `ca`.

Question 1.5. Arbres

(a) Donner la liste de tous les mots représentés par l'arbre de la figure 1.

(b) Montrer qu'il existe un unique arbre dont les feuilles sont terminales représentant un ensemble de mots donné.

(c) Soit A un arbre représentant un ensemble de mots S. Supposons que la racine de l'arbre ait k fils $A_1, \ldots, A_k$. Pour $1 \leq i \leq k$, on note x_i l'étiquette de l'arc allant de la racine de A à celle de A_i et S_i l'ensemble des mots représentés par A_i. Montrer que

$$S = \bigcup_{i=1}^{k} \left\{ u : u_0 = x_i, u_{1,|u|} \in S_i \right\}.$$

Les arbres décrits précédemment sont implantés à l'aide de la classe `Dico` fournie en figure 2.

La méthode `__getitem__` s'utilise de la manière suivante : si `d` est un objet de type `Dico` et `lettre` une des lettres étiquetant un arc sortant de la racine de `d`, alors `d[lettre]` renvoie le sous-arbre de `d` qui se trouve à l'extrémité de l'arc étiqueté par `lettre`. Par exemple, si `d` est l'arbre de la figure 1, `d[c]` renvoie le sous-arbre constitué des 10 sommets *du haut* et `d[p]` renvoie le sous-arbre constitué des 11 sommets *du bas*.

[1]Cet *ensemble de mots* est ce qu'on appelle en langage courant un *dictionnaire*, mais on évite dans la suite d'utiliser ce terme pour ne pas créer de confusion avec le type `dict` de Python.

```python
class Dico:
    def __init__(self, terminal = False):
        self.terminal = terminal
        self.fils = {}

    def est_terminal(self):
        "Renvoie True si la racine de self est terminale."
        return self.terminal

    def rend_terminal(self):
        "Modifie self pour rendre sa racine terminale."
        self.terminal = True

    def lettres(self):
        "Renvoie la liste des lettres qui étiquettent un arc sortant de la racine."
        return list(self.fils.keys())

    def __getitem__(self, lettre):
        "Renvoie le sous-arbre pointé par l'arc d'étiquette est lettre."
        return self.fils[lettre]

    def greffe_dico(self, dico, lettre):
        "Ajoute le sous-arbre dico à self, avec un arc étiqueté lettre."
        if lettre in self.lettres():
            raise ValueError("La lettre {} existe déjà".format(lettre))
        self.fils[lettre] = dico

    def ajout_lettre(self, lettre):
        "Ajoute un nouveau nœud non terminal comme fils de la racine, avec un arc\
                étiqueté lettre."
        self.greffe_dico(Dico(), lettre)
```

Figure 2: Implantation de la classe `Dico`.

Question 1.6. Dictionnaire Dans cette question, les algorithmes doivent être écrits comme méthodes de la classe `Dico`. On rappelle que le premier paramètre des méthodes, habituellement noté `self`, est l'objet de type `Dico`.

(a) Soit `d` l'objet de la classe `Dico` qui représente l'arbre de la figure 1. Que renvoient `d.est_terminal()` et `d.lettres()` ?

(b) Écrire une méthode `contient(self, mot)` qui prend en entrée un mot et renvoie `True` si le mot appartient au `Dico`.

(c) Écrire une méthode `liste_des_mots(self)` qui renvoie la liste des mots du `Dico`.

(d) Écrire une méthode `ajout_mot(self, mot)` qui prend en entrée un mot et l'ajoute au `Dico`. *La méthode ajoutera des nœuds à l'arbre si nécessaire, et modifiera un nœud existant sinon. Attention : cette méthode doit modifier `self` et ne rien renvoyer (de la même manière que `ajout_lettre`, que l'on utilisera).*

Question 1.7. Corrections

(a) Écrire une méthode récursive `correction_hamming(self, mot, k)` qui prend en entrée un mot et un entier k, et renvoie la liste des mots du `Dico` à distance au plus k de `mot`.

(b) Écrire une méthode `correction_levenshtein(self, mot)` qui prend en entrée un mot et renvoie le `Dico` contenant exactement les mots à distance de Levenshtein au plus 1 de `mot`.

1.3 Compression du `Dico`

On souhaite compresser l'arbre en le représentant maintenant par un graphe orienté acyclique. Pour cela, on note que certains nœuds sont *équivalents* : ils sont de même nature (terminal ou non terminal), et ont les mêmes sous-arbres (avec les mêmes étiquettes). Par exemple, dans l'arbre de la figure 1, les nœuds atteints en lisant `car` et `perd` sont équivalents. En fusionnant les nœuds équivalents, on obtient le graphe orienté acyclique de la figure 3.

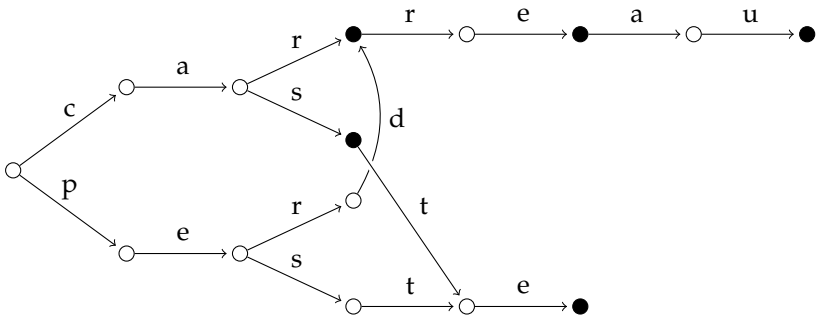

Figure 3: Exemple de représentation d'un ensemble de mots par un graphe orienté acyclique.

Question 1.8.

(a) Quelle est la liste des mots représentés par le graphe orienté acyclique de la figure 3 ?

(b) Pourquoi ne peut-on pas fusionner les nœuds atteints en lisant `ca` et `pe` ?

(c) Montrer que deux nœuds ne peuvent pas être équivalents si l'un est l'ancêtre de l'autre.

(d) Montrer que fusionner deux nœuds équivalents ne modifie pas l'ensemble des mots représentés.

Question 1.9. Pour compresser le `Dico`, on utilise l'algorithme suivant :

1. on partitionne les nœuds en classes d'équivalences ;
2. on fusionne tous les nœuds d'une même classe d'équivalence.

Pour implanter l'algorithme dans la classe `Dico`, on identifie chaque nœud de l'arbre avec le mot qui permet de l'atteindre.

(a) Écrire une méthode `tous_les_mots(self)` qui renvoie la liste de tous les mots présents dans l'arbre, qu'ils appartiennent ou non à l'ensemble de mots valides. *En appliquant la méthode à l'arbre de la figure 1, on doit obtenir une liste de 22 mots, commençant par* `["", "c", "p", "ca", "pe", ...]`.

(b) Écrire une méthode `parcours(self, mot)` qui renvoie le `Dico` atteint lorsqu'on lit le `mot`. Si le `mot` n'apparaît pas dans l'arbre, la méthode doit soulever une exception de type `ValueError`.

(c) Écrire une méthode `est_equivalent(self, dico)` qui teste si la racine de `self` est équivalente à la racine de `dico`.

(d) Écrire une méthode `compresse(self)` qui compresse le `Dico` grâce à l'algorithme décrit précédemment. *Indications. Numéroter les nœuds en utilisant l'indice du mot correspondant dans la liste renvoyée par* `tous_les_mots`*. Tester l'équivalence des nœuds deux à deux, et utiliser comme représentant unique d'une classe d'équivalence, le nœud d'indice le plus faible dans cette classe. Une fois les classes d'équivalence calculées, remplacer chaque nœud par le représentant de sa classe d'équivalence.*