Corrigé : gestion de versions de grands textes (X PC 2023)

Partie I. Différentiels par positions fixes

Question 1.

```
def textes_égaux(texte1, texte2):
    for i in range(len(texte1)):
        if texte1[i] != texte2[i]:
            return False
    return True
```

Cette fonction a une complexité en $O(n)$, où n désigne la longueur commune aux listes `texte1` et `texte2`.

Question 2.

```
def distance(texte1, texte2):
    s = 0
    for i in range(len(texte1)):
        if texte1[i] != texte2[i]:
            s += 1
    return s
```

Cette fonction a une complexité en $O(n)$, où n désigne la longueur commune aux listes `texte1` et `texte2`.

Question 3.

```
def aucun_caractère_commun(texte1, texte2):
    d = {}
    for c in texte1:
        if c not in d:
            d[c] = None
    for c in texte2:
        if c in d:
            return False
    return True
```

Le dictionnaire `d` est l'ensemble des caractères utilisés dans `texte1` ; la valeur associée à chaque clef n'a pas d'importance (on pourrait utiliser le type `set` s'il était au programme).

Question 4.

```
1  def différentiel(texte1, texte2):
2      diff = []
3      modif = False
4      for i in range(len(texte1)):
5          if texte1[i] != texte2[i] and not modif:
6              modif = True
7              début = i
8              avant = [texte1[i]]
9              après = [texte2[i]]
10         elif texte1[i] != texte2[i] and modif:
11             avant.append(texte1[i])
12             après.append(texte2[i])
13         elif texte1[i] == texte2[i] and modif:
14             modif = False
15             diff.append(tranche(début, avant, après))
16     if modif:
17         diff.append(tranche(début, avant, après))
18     return diff
```

La variable booléenne `modif` indique si une tranche est en cours d'enregistrement. La fonction `différentiel` parcourt les caractères des deux listes, et quatre cas peuvent se présenter :

- les caractères diffèrent et une tranche n'est pas en train d'être calculée (ligne 5) : on débute la création d'une tranche ;
- les caractères diffèrent et une tranche est en train d'être calculée (ligne 10) : la création de la tranche se poursuit ;
- les caractères coïncident et une tranche est en train d'être calculée (ligne 13) : la création de la tranche se termine ;
- les caractères coïncident et une tranche n'est pas en train d'être calculée (il n'y a rien à faire).

En fin de parcours, on clôt une éventuelle tranche en cours de formation (ligne 16).

Lors du parcours chacune des opérations effectuées est de complexité constante donc la complexité totale est un $O(n)$, où n désigne la longueur commune aux deux textes.

Question 5.

```
def applique(texte1, diff):
    texte2 = texte1[:]
    for tr in diff:
        for i in range(début(tr), fin(tr)):
            texte2[i] = après(tr)[i - début(tr)]
    return texte2
```

Cette fonction recopie le premier texte (complexité linéaire) puis, pour chaque tranche du différentiel, modifie les caractères concernés. Chauqe caractère du premier texte est modifié au plus un fois donc la complexité totale est un $O(n)$, où n est la longueur du premier texte.

Question 6.

```
def inverse(diff):
    inv = []
    for tr in diff:
        inv.append(tranche(début(tr), après(tr), avant(tr)))
    return inv
```

Cette fonction a une complexité en $O(p)$ où p est le nombre de tranches contenues dans `diff`.

Question 7.

```
def modifie(texte_versionné, texte):
    historique(texte_versionné).append(différentiel(courant(texte_versionné), texte))
    remplace_courant(texte_versionné, texte)
```

Les fonctions `historique` et `remplace_courant` sont de complexités constantes donc cette fonction a même complexité que la fonction `différentiel`, à savoir un $O(n)$, où n est la longueur de `texte`.

```
def annule(texte_versionné):
    diff = historique(texte_versionné).pop()
    remplace_courant(texte_versionné, applique(courant(texte_versionné), inverse(diff)))
    return courant(texte_versionné)
```

Les fonctions `inverse` et `applique` sont de complexités linéaires donc cette fonction a une complexité en $O(n)$ où n est la longueur du texte.

Partie II. Différentiels sur des positions variables

Question 8.

```
def poids(diff):
    p = 0
    for tr in diff:
        p += len(avant(tr))
        p += len(après(tr))
    return p
```

Les fonctions `avant` et `après` sont de complexités constantes donc ette fonction a une complexité en $O(p)$, où p est la longueur de `diff`.

Question 9. On a $\texttt{M[i+1][j+1]} = \begin{cases} \texttt{M[i][j]} & \text{si } \texttt{texte1[i]} = \texttt{texte2[j]} \\ \min(\texttt{M[i][j+1]}, \texttt{M[i+1][j]}) + 1 & \text{sinon} \end{cases}$.

Si `texte1[i]` = `texte2[j]`, aucune modification n'est nécessaire ; dans le cas contraire il faut soit supprimer le caractère `texte1[i]`, soit ajouter le caractère `texte2[j]`.

Question 10.

```
def levenshtein(texte1, texte2):
    n, m = len(texte1), len(texte2)
    M = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        M[i][0] = i
    for j in range(1, m + 1):
        M[0][j] = j
    for i in range(n):
        for j in range(m):
            if texte1[i] == texte2[j]:
                M[i + 1][j + 1] = M[i][j]
            else:
                M[i + 1][j + 1] = min(M[i + 1][j], M[i][j + 1]) + 1
    return M
```

La complexité de cette fonction est en $O(mn)$ où m et n désignent les longueurs des deux textes.

Question 11.

```
1  def différentiel(texte1, texte2, M):
2      diff = []
3      i, j = len(texte1), len(texte2)
4      while (i, j) != (0, 0):
5          if i == 0:
6              diff.append(tranche(0, [], 0, texte2[:j]))
7              j = 0
8          elif j == 0:
9              diff.append(tranche(0, texte1[:i], 0, []))
10             i = 0
11         elif texte1[i - 1] == texte2[j - 1]:
12             i, j = i - 1, j - 1
13         else:
14             après = []
15             while j > 0 and M[i][j] == M[i][j - 1] + 1:
16                 j -= 1
17                 après.append(texte2[j])
18             avant = []
19             while i > 0 and M[i][j] == M[i - 1][j] + 1:
20                 i -= 1
21                 avant.append(texte1[i])
22             diff.append(tranche(i, avant[::-1], j, après[::-1]))
23     return diff[::-1]
```

Dans le tableau `M`, un déplacement horizontal correspond à une insertion, un déplacement vertical à une suppression, et on ne modifie rien en cas de déplacement diagonal. L'algorithme consiste à partir de la case située en bas à droite (ligne 3) ; lorsque `texte1[i]` est différent de `texte2[j]` (ligne 13) on calcule le texte ajouté (lignes 14-17) et le texte supprimé (lignes 18-21) pour calculer la tranche (ligne 22). Les lignes 5-7 et 8-10 traitent des cas particuliers où l'un des mots est vide.

Remarque. Puisque la lecture se fait de droite à gauche et de bas en haut, il faut renverser les listes calculées ; pour ce faire j'utilise la syntaxe `lst[::-1]`.

Question 12.

```
def conflit(diff1, diff2):
    i, j = 0, 0
    while i < len(diff1) and j < len(diff2):
        tr1, tr2 = diff1[i], diff2[j]
        if fin_avant(tr1) < début_avant(tr2):
            i += 1
        elif fin_avant(tr2) < début_avant(tr1):
            j += 1
        else:
            return True
    return False
```

Lorsque `fin_avant(tr1)` < `début_avant(tr2)`, la tranche `tr1` ne rentre en conflit avec aucune tranche de `diff2` qui suit `tr2`.
La complexité est bien en $O(p+q)$ où p et q désignent les longueurs des deux listes `diff1` et `diff2` car la somme $i+j$ est incrémentée d'une unité à chaque étape.

Question 13.

```
def fusionne(diff1, diff2):
    i, j = 0, 0
    diff = []
    for _ in range(len(diff1) + len(diff2)):
        if i == len(diff1):
            diff.append(diff2[j])
            j += 1
        elif j == len(diff2):
            diff.append(diff1[i])
            i += 1
        else:
            tr1, tr2 = diff1[i], diff2[j]
            if fin_avant(tr1) < début_avant(tr2):
                diff.append(diff1[i])
                i += 1
            else:
                diff.append(diff2[j])
                j += 1
    return diff
```

Cet algorithme suit le principe du tri fusion.

Partie III. Calcul de différentiels par calcul de plus courts chemins

Question 14.

```
def successeurs(texte1, texte2, sommet):
    n, m = len(texte1), len(texte2)
    (i, j) = sommet
    if i == n and j == m:
        return []
    elif j == m:
        return [((i + 1, j), 1)]
    elif i == n:
        return [((i, j + 1), 1)]
    else:
        succ = [((i + 1, j), 1), ((i, j + 1), 1)]
        if texte1[i] == texte2[j]:
            succ.append(((i + 1, j + 1), 0))
    return succ
```

On attribue aux arcs verticaux et horizontaux un poids égal à 1 car ils correspondent à une insertion ou à une suppression de caractère ; il n'y a un arc diagonal que lorsque les caractères `texte[i]` et `texte[j]` coïncident, et dans ce cas le poids est égal à 0 puisqu'il n'y a aucune modification à réaliser.

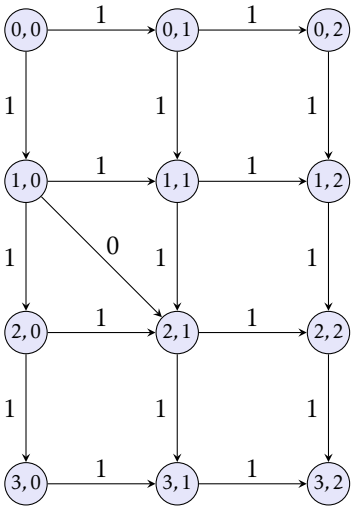
Question 15. L'algorithme présenté renvoie un dictionnaire donc les clés sont les sommets (i, j) que l'algorithme a rencontré et les valeurs la longueur du plus court chemin de $(0, 0)$ à (i, j), égal à la distance d'édition de `texte1[:i]` à `texte2[:j]`. Il s'agit très précisément des sommets dont la distance à l'origine est inférieure ou égale à la distance entre l'entrée et la sortie (puisque ce dernier est le plus grand par ordre lexicographique).

Question 16. Chaque sommet n'ayant qu'un nombre de voisins inférieur ou égal à trois, la mise à jour des distances est de complexité constante. Ainsi, seule la gestion de la file de propriété n'est pas de complexité constante mais de complexité logarithmique et si on note n et m les longueurs des deux textes, la complexité temporelle de l'algorithme est un $O(nm \log(np))$.
Cette complexité est moins bonne que celle de l'algorithme basé sur le principe de la programmation dynamique, mais l'avantage de celui-ci est qu'il se termine dès lors que le sommet `(len(texte1),len(texte2))` a été vu. Il calcule donc potentiellement moins de valeurs que l'algorithme basé sur la programmation dynamique.

Question 17. On choisit pour heuristique la fonction

```
h(texte1, texte2, (i, j)) = abs((len(texte1) - i) - (len(texte2) - j))
```

Il s'agit bien d'une heuristique admisible puisque c'est le nombre minimal d'insertion (ou de suppression suivant les cas) qu'il faudra réaliser.
Pour `texte1 = ['A', 'B', 'C']` et `texte2 = ['B', 'X']`, le graphe concerné et la matrice d'édition sont les suivants :



0	1	2
1	2	3
2	1	2
3	2	3

L'algorithme de Dijkstra calcule toutes les distances de la matrice d'édition, alors que l'algorithme A* ne calcule que les distances minimales suivantes :

0	1	
1	2	
2	1	2
	2	3