Raport - Algorytmy Tekstowe - Laboratorium 4

Łukasz Sochacki

Podpunkt 1 i 2.

W celu zaimplementowania algorytmu wyznaczającego odległość edycyjną między dwoma pewnymi łańcuchami została zaimplementowana funkcja, której kod znajduje się poniżej. Dodatkowo funkcja przedstawia kolejne kroki przekształcania pierwszego łańcucha w celu uzyskania drugiego łańcucha.

```
class Node:
    def __init__(self, steps):
        self.steps = steps
        self.parent = None
        self.todo = 'Stay'


def editDistance(first, second):
    n = len(first)
    m = len(second)
    operations = ['Stay', 'Change', 'Insert', 'Delete']
    table = []
    for i in range(n + 1):
        table.append([])
        for j in range(m + 1):
            table[i].append(Node(0))

    for i in range(n + 1):
        for j in range(m + 1):
            if i == 0 and j == 0:
                continue
            elif i == 0:
                table[0][j].parent = table[0][j - 1]
                table[0][j].steps = j
                table[0][j].todo = operations[2]
            elif j == 0:
                table[i][0].parent = table[i - 1][0]
                table[i][0].steps = i
                table[i][0].todo = operations[3]
            else:
                if first[i - 1] != second[j - 1]:
                    same = 1
                else:
                    same = 0
                tmp, val = min([(table[i - 1][j], table[i - 1][j].steps +
1),
                                (table[i][j - 1], table[i][j - 1].steps + 1),
                                (table[i - 1][j - 1], table[i - 1][j - 1].steps
+ same)], key=lambda a: a[1])
                if table[i - 1][j] == tmp:
                    table[i][j].todo = operations[3]
```

```
                elif table[i][j - 1] == tmp:
                    table[i][j].todo = operations[2]
                elif table[i - 1][j - 1] == tmp:
                    if first[i - 1] == second[j - 1]:
                        table[i][j].todo = operations[0]
                    else:
                        table[i][j].todo = operations[1]
                table[i][j].steps = val
                table[i][j].parent = tmp

    print(f'Odleglosc edycyjna wynosi : {table[n][m].steps}')
    ops = []
    current = table[n][m]
    while current.parent is not None:
        ops.append(current.todo)
        current = current.parent

    for i in range(len(ops) // 2):
        ops[i], ops[len(ops) - 1 - i] = ops[len(ops) - 1 - i], ops[i]

    i = 0
    j = 0
    third = first
    print(f'Pierwotne slowo - {first}')
    for op in ops:
        which = operations.index(op)
        if which != 3:
            if which == 1:
                changed = third[i]
                third = third[:i] + second[j] + third[i + 1:]
                print(f'{third} - zmieniono znak {changed} na znak
{second[j]}')
            elif which == 2:
                third = third[:i] + second[j] + third[i:]
                print(f'{third} - wstawiono znak {second[j]}')
            else:
                pass
            i += 1
            j += 1
        else:
            deleted = third[i]
            third = third[:i] + third[i + 1:]
            print(f'{third} - usunieto znak {deleted}')

    print(f'Otrzymane finalne slowo - {third}')
```

Podpunkt 3.

Poniżej zostały przedstawione otrzymane rezultaty po uruchomieniu funkcji przedstawionej powyżej dla poszczególnych par łańcuchów. Rezultaty przedstawiają otrzymaną długość edycyjną oraz kolejne kroki przekształceń aż do momentu uzyskania drugiego łańcucha.

- **Para ‘los’ i ‘kloc’**
  Odleglosc edycyjna wynosi : 2
  Pierwotne slowo - los
  klos - wstawiono znak k
  kloc - zmieniono znak s na znak c
  Otrzymane finalne slowo – kloc

- **Para Łódź i Lodz**
  Odleglosc edycyjna wynosi : 3
  Pierwotne slowo - Łódź
  Lódź - zmieniono znak Ł na znak L
  Lodź - zmieniono znak ó na znak o
  Lodz - zmieniono znak ź na znak z
  Otrzymane finalne slowo – Lodz

- **Para kwintesencja i quintessence**
  Odleglosc edycyjna wynosi : 5
  Pierwotne slowo - kwintesencja
  qwintesencja - zmieniono znak k na znak q
  quintesencja - zmieniono znak w na znak u
  quintessencja - wstawiono znak s
  quintessencea - zmieniono znak j na znak e
  quintessence - usunieto a
  Otrzymane finalne slowo - quintessence

- **Para ATGAATCTTACCGCCTCG i ATGAGGCTCTGGCCCCTG**
  Odleglosc edycyjna wynosi : 7
  Pierwotne slowo - ATGAATCTTACCGCCTCG
  ATGAGTCTTACCGCCTCG - zmieniono znak A na znak G
  ATGAGGCTTACCGCCTCG - zmieniono znak T na znak G
  ATGAGGCTCTACCGCCTCG - wstawiono znak C
  ATGAGGCTCTGCCGCCTCG - zmieniono znak A na znak G
  ATGAGGCTCTGGCCGCCTCG - wstawiono znak G
  ATGAGGCTCTGGCCCCTCG - usunieto G
  ATGAGGCTCTGGCCCCTG - usunieto C
  Otrzymane finalne slowo - ATGAGGCTCTGGCCCCTG

Podpunkt 4.

W celu implementacji algorytmu obliczania najdłuższego wspólnego podciągu powstała funkcja, która przyjmuje dwa łańcuchy znaków i na ich podstawie wylicza najdłuższy wspólny podciąg. Poniżej znajduje się zaimplementowana funkcja.

```
def LCS(first, second):
    n = len(first) + 1
    m = len(second) + 1
    matrix = []
    for i in range(n):
        matrix.append([])
        for j in range(m):
            matrix[i].append(0)

    for i in range(n):
        for j in range(m):
            if i != 0 and j != 0:
                if first[i - 1] == second[j - 1]:
                    matrix[i][j] = matrix[i - 1][j - 1] + 1
                else:
                    matrix[i][j] = max(matrix[i - 1][j], matrix[i][j - 1])
            else:
                continue
    return matrix
```

Podpunkt 5.

W celu tokenizacji przetwarzanego tekstu została wykorzystana biblioteka spaCy, która zawiera gotowe algorytmy tokenizacji. W tym celu został zaimportowany gotowy tokenizer z biblioteki spaCy oraz dodatkowo wsparcie dla przetwarzania tekstów zapisanych w języku polskim.

```
from spacy.tokenizer import Tokenizer
from spacy.lang.pl import Polish
```

Następnie została zaimplementowa funkcja, która przyjmuje na wejściu tekst oraz zwraca go w postaci tokenów powstałych w procesie dzielenia tekstu.

```
def tokenizeText(text):
    nlp = Polish()
    tokenizer = Tokenizer(nlp.vocab)
    return tokenizer(text)
```

Podpunkt 6.

Otrzymany tekst w postaci pliku 'romeo-i-julia-700.txt' został początkowo zapisany do zmiennej oraz podzielony na tokeny za pomocą funkcji wymienionej w podpunkcie 5. . Następnie została wykorzystana funkcja wybierająca losowo tokeny w taki sposób, aby łącznie 3% losowo wybranych tokenów zostało zignorowanych i odrzuconych.

```
def randomTokens(text):
    ratio = 3 / 100
    result = []
    for el in text:
        if random() < ratio:
            continue
        else:
            result.append(el)

    return result
```

Następnie za pomocą dwóch zmiennych zostały zapisane dwie wersje otrzymanego tekstu po utracie przetworzonego dodatkowo przez powyższą funkcje w taki sposób, by ilość tokenów została zmniejszona o około 3% łącznej ilości.

```
with open('romeo-i-julia-700.txt', 'r', encoding='utf-8') as f:
    data = f.read()

tokenized = tokenizeText(data)
first, second = randomTokens(tokenized), randomTokens(tokenized)
```

Podpunkt 7.

Wykorzystując funkcję z podpunktu 4. została obliczona długość najdłuższego wspólnego podciągu między dwoma wersjami tekstu wymienionym powyżej. Dodatkowo została wyświetlona długość obu wersji tekstów w celu weryfikacji poprawności działania programu.

```
tokens = [len(tokenized), len(first), len(second)]
print(f'Oryginalna ilosc tokenow po podzieleniu tekstu na tokeny :
{tokens[0]}')
print(f'Ilosc tokenow w pierwszej wersji tekstu po usunieciu pewnych
tokenow : {tokens[1]}')
print(f'Ilosc tokenow w drugiej wersji tekstu po usunieciu pewnych tokenow
: {tokens[2]}')
print(f'Dlugosc najdłuższego podciągu wspólnych tokenów dla obu wersji
tekstow : {LCS(first, second)[tokens[1] - 1][tokens[2] - 1]}')
```

Przykładowy rezultat :

Oryginalna ilosc tokenow po podzieleniu tekstu na tokeny : **2272**

Ilosc tokenow w pierwszej wersji tekstu po usunieciu pewnych tokenow : **2184**

Ilosc tokenow w drugiej wersji tekstu po usunieciu pewnych tokenow : **2202**

Dlugosc najdłuższego podciągu wspólnych tokenów dla obu wersji tekstow : **2115**

Podpunkt 8.

W celu wykonania podpunktu została wykorzystana funkcja z podpunktu 4. . Wynikiem działania funkcji jest macierz, która zawiera informację o długości najdłuższego wspólnego podciągu dla danych fragmentów pierwszego łańcucha oraz drugiego. Następnie sprawdzając kolejne elementy otrzymanej macierzy zależnie od indeksów, które wyznaczają położenie danej wartości sprawdzamy, czy dany fragment nie znajduje się w obu plikach oraz z jakiego pliku pochodzi. Poniżej znajduje się kod funkcji.

```
def diff(first, second):
    n = len(first)
    m = len(second)
    matrix = LCS(first, second)
    result = []
    i = n - 1
    j = m - 1

    while i >= 0 and j >= 0:
        if first[i] != second[j]:
            if matrix[i][j - 1] >= matrix[i - 1][j]:
                result.append(f'> [linia {j}] Fragment : {second[j]}')
                j -= 1
            else:
                result.append(f'< [linia {i}] Fragment : {first[i]}')
                i -= 1
        else:
            i -= 1
            j -= 1

    for k in range(j, -1, -1):
        result.append(f'> [linia {k}] Fragment : {second[k]}')

    for k in range(i, -1, -1):
        result.append(f'< [linia {k}] Fragment : {first[k]}')

    for i in range(len(result)-1, -1, -1):
        print(result[i])
        i -= 1

    result.reverse()
```

```
    for el in result:
        print(el)
```

Podpunkt 9.

Poniżej znajduje się przykładowy fragment wyniku działania funkcji z podpunktu 8. Znak < oznacza, że dany łańcuch znajduje się w pierwszym pliku/tekście, gdzie > oznacza, że łańcuch znajduje się w drugim pliku/tekście. Numer w nawiasach kwadratowych oznacza numer linii w pliku.

> [line 16] ESKALUS

< [line 30] księcia

> [line 30] krewny

> [line 36] naczelnicy

< [line 58] księcia

> [line 87] *

> [line 124] *

< [line 155] liczący

> [line 169] Rzecz

> [line 200] gdzie

< [line 212] Plamiąc

> [line 218] dłonie