Zadanie 2 - trie and suffix trees - Paweł Kruczkiewicz

March 22, 2021

1 Pattern maching - wyszukiwanie wzorców

1.1 Paweł Kruczkiewicz

17.03.2021 r.

1.1.1 Treść

Celem zadania jest zapoznanie się z konstrukcjami trie oraz drzewem sufiksów.

1. Przyjmij następujący zbiór danych wejściowych:
    1. bbbd
    2. aabbabd
    3. ababcd
    4. abcbccd
    5. oraz załączony plik.
2. Upewnij się, że każdy łańcuch na końcu posiada unikalny znak (marker), a jeśli go nie ma, to dodaj ten znak.
3. Zaimplementuj algorytm konstruujący strukturę trie, która przechowuje wszystkie sufiksy łańcucha danego na wejściu.
4. Zaimplementuj algorytm konstruujący drzewo sufiksów.
5. Upewnij się, że powstałe struktury danych są poprawne. Możesz np. sprawdzić, czy struktura zawiera jakiś ciąg znaków i porównać wyniki z algorytmem wyszukiwania wzorców.
6. Porównaj szybkość działania algorytmów konstruujących struktury danych dla danych z p. 1 w następujących wariantach:
    1. Trie (1 pkt)
    2. Drzewo sufiksów bez wykorzystania procedury fast_find oraz elementów "link" (2 pkt)
7. Oczekiwany wynik ćwiczenia to kod źródłowy oraz raport w formie PDF.
8. *Dla zinteresowanych:* implementacja i testy z wykorzystaniem procedury fast_find oraz elementów "link" w drzewie sufiksowym (+2 pkt).

1.2 Ad. 1 i 2

Dane wejściowe dodajemy do listy wszystkich możliwyh słów (`texts`), a następnie dodajemy znak `$` na koniec każdego wyrazu

[3]:
```
texts = ["bbbd", "aabbabd", "ababcd", "abcbccd"]
text_from_the_file = '''
    Dz.U. z 1998 r. Nr 144, poz. 930

```

```
                                  USTAWA
                              z dnia 20 listopada 1998 r.

          o zryczałtowanym podatku dochodowym od niektórych przychodów
                        osiąganych przez osoby fizyczne

                                 Rozdział 1
                              Przepisy ogólne

                                  Art. 1.
Ustawa reguluje opodatkowanie zryczałtowanym podatkiem dochodowym niektórych
przychodów (dochodów) osiąganych przez osoby fizyczne prowadzące pozarolniczą
działalność gospodarczą oraz przez osoby duchowne.

                                  Art. 2.
1. Osoby fizyczne osiągające przychody z pozarolniczej działalności
  gospodarczej opłacają zryczałtowany podatek dochodowy w formie:
   1) ryczałtu od przychodów ewidencjonowanych,
   2) karty podatkowej.
2. Osoby duchowne, prawnie uznanych wyznań, opłacają zryczałtowany podatek
  dochodowy od przychodów osób duchownych.
3. Wpływy z podatku dochodowego opłacanego w formie ryczałtu od przychodów
  ewidencjonowanych oraz zryczałtowanego podatku dochodowego od przychodów
  osób duchownych stanowią dochód budżetu państwa.
4. Wpływy z karty podatkowej stanowią dochody gmin.

                                  Art. 3.
Przychodów (dochodów) opodatkowanych w formach zryczałtowanych nie łączy się z
przychodami (dochodami) z innych źródeł podlegającymi opodatkowaniu na
podstawie ustawy z dnia 26 lipca 1991 r. o podatku dochodowym od osób
fizycznych (Dz. U. z 1993 r. Nr 90, poz. 416 i Nr 134, poz. 646, z 1994 r. Nr
43, poz. 163, Nr 90, poz. 419, Nr 113, poz. 547, Nr 123, poz. 602 i Nr 126,
poz. 626, z 1995 r. Nr 5, poz. 25 i Nr 133, poz. 654, z 1996 r. Nr 25, poz.
113, Nr 87, poz. 395, Nr 137, poz. 638, Nr 147, poz. 686 i Nr 156, poz. 776, z
1997 r. Nr 28, poz. 153, Nr 30, poz. 164, Nr 71, poz. 449, Nr 85, poz. 538, Nr
96, poz. 592, Nr 121, poz. 770, Nr 123, poz. 776, Nr 137, poz. 926, Nr 139,
poz. 932-934 i Nr 141, poz. 943 i 945 oraz z 1998 r. Nr 66, poz. 430, Nr 74,
poz. 471, Nr 108, poz. 685 i Nr 117, poz. 756), zwanej dalej "ustawą o podatku
dochodowym".
'''
texts.append(text_from_the_file)

texts = list(map(lambda x: x + '\uE000', texts))
print(texts)
```

```
['bbbd\ue000', 'aabbabd\ue000', 'ababcd\ue000', 'abcbccd\ue000', '\n    Dz.U. z
1998 r. Nr 144, poz. 930\n                                          \n
\n                                    \n
\n                                  USTAWA\n                              z dnia
20 listopada 1998 r.\n                                          \n         o
zryczałtowanym podatku dochodowym od niektórych przychodów\n
osiąganych przez osoby fizyczne\n                                         \n
Rozdział 1\n                                  Przepisy ogólne\n
\n                                  Art. 1.\nUstawa reguluje opodatkowanie
zryczałtowanym podatkiem dochodowym niektórych\nprzychodów (dochodów) osiąganych
przez osoby fizyczne prowadzące pozarolniczą\ndziałalność gospodarczą oraz przez
osoby duchowne.\n                                          \n
Art. 2.\n1. Osoby fizyczne osiągające przychody z pozarolniczej działalności\n
gospodarczej opłacają zryczałtowany podatek dochodowy w formie:\n   1) ryczałtu
od przychodów ewidencjonowanych,\n   2) karty podatkowej.\n2. Osoby duchowne,
prawnie uznanych wyznań, opłacają zryczałtowany podatek\n  dochodowy od
przychodów osób duchownych.\n3. Wpływy z podatku dochodowego opłacanego w formie
ryczałtu od przychodów\n  ewidencjonowanych oraz zryczałtowanego podatku
dochodowego od przychodów\n  osób duchownych stanowią dochód budżetu
państwa.\n4. Wpływy z karty podatkowej stanowią dochody gmin.\n
\n                                  Art. 3.\nPrzychodów (dochodów)
opodatkowanych w formach zryczałtowanych nie łączy się z\nprzychodami
(dochodami) z innych źródeł podlegającymi opodatkowaniu na\npodstawie ustawy z
dnia 26 lipca 1991 r. o podatku dochodowym od osób\nfizycznych (Dz. U. z 1993 r.
Nr 90, poz. 416 i Nr 134, poz. 646, z 1994 r. Nr\n43, poz. 163, Nr 90, poz. 419,
Nr 113, poz. 547, Nr 123, poz. 602 i Nr 126,\npoz. 626, z 1995 r. Nr 5, poz. 25
i Nr 133, poz. 654, z 1996 r. Nr 25, poz.\n113, Nr 87, poz. 395, Nr 137, poz.
638, Nr 147, poz. 686 i Nr 156, poz. 776, z\n1997 r. Nr 28, poz. 153, Nr 30,
poz. 164, Nr 71, poz. 449, Nr 85, poz. 538, Nr\n96, poz. 592, Nr 121, poz. 770,
Nr 123, poz. 776, Nr 137, poz. 926, Nr 139,\npoz. 932-934 i Nr 141, poz. 943 i
945 oraz z 1998 r. Nr 66, poz. 430, Nr 74,\npoz. 471, Nr 108, poz. 685 i Nr 117,
poz. 756), zwanej dalej "ustawą o podatku\ndochodowym".\n\ue000']
```

1.3 Ad. 3 - konstrukcja drzewa trie

Drzewo będziemy przechowywać w specjalnej klasie `Trie`. Każdy węzeł przechowujemy w klasie `Node`, w której oprócz dzieci będzie zapamiętywany rodzic. To się ponoć przydaje w fast-findzie, więc na wszelki wypadek to tutaj zostawię, ale nie mam pojęcia po co to jest

[4]:
```python
class Node:
    def __init__(self, parent):
        self.parent = parent
        self.children = dict()

class Trie:
    def __init__(self, text):
        self.root = Node(None)
        self.break_sign = '$'
```

```
        for i in range(len(text)):
            curr_suffix = text[i:]
            head, index = self.find(curr_suffix)
            self.graft(head, curr_suffix[index:])

    def find(self, text):
        curr_node = self.root
        i = 0
        while i < len(text) and text[i] in curr_node.children:
            curr_node = curr_node.children[text[i]]
            i += 1
        return curr_node, i

    def graft(self, node, text):
        for ch in text:
            new_node = Node(node)  # creating new node with node as a parent
            node.children[ch] = new_node  # adding new node to children of␣
↪current node
            node = new_node

    def search(self, substring):  #we don't count the $ sign at the end
        found_node, index = self.find(substring)
        return found_node.children != {} and index == len(substring)

    def print_trie(self, node=None):
        if node is None:  node = self.root
        print(node.children)
        for child in node.children.values():
            self.print_trie(child)
```

[5]:
```
trie = Trie(texts[0])
print(trie)
print(trie.search('bd'))
```

```
<__main__.Trie object at 0x0000026622FD0048>
True
```

1.4 Ad. 4

Implementacja drzewa sufiksów

[6]:
```
class SuffixTreeNode:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.children = dict()
        # elf.start_index = start_index
```

```
class SuffixTree:
    def __init__(self, text):
        self.root = SuffixTreeNode(0, len(text) - 1)
        self.full_text = text
        for i in range(len(text) - 1):
            curr_suff = text[i:]
            head, depth = self.slow_find(curr_suff)
            self.graft(head, depth, i)

    def graft(self, head, depth, i):
        new_node = SuffixTreeNode(depth + i, len(self.full_text) - 1)
        head.children[self.full_text[new_node.start]] = new_node

    def slow_find(self, text, depth=0, curr_node=None):  # return head; depth - length of the longest prefix of text
        # in suffix tree
        if curr_node is None: curr_node = self.root

        first_letter = text[0]
        next_node = curr_node.children.get(first_letter)

        if next_node is None:
            return curr_node, depth
        start, end = next_node.start, next_node.end
        childs_text_len = end - start + 1
        for i in range(1, childs_text_len):
            if self.full_text[start + i] != text[i]:
                return self.break_path(i, curr_node, next_node), depth + i
        return self.slow_find(text[childs_text_len:], depth + childs_text_len, next_node)

    def break_path(self, index, parent_node, next_node):
        old_start, old_end = next_node.start, next_node.end
        break_node = SuffixTreeNode(old_start, old_start + index - 1)
        next_node.start = old_start + index

        parent_node.children[self.full_text[break_node.start]] = break_node #wskazanie po literce na nowy node
        break_node.children[self.full_text[next_node.start]] = next_node  # wskazanie po literce na dotychczasowy node
        return break_node

    def search(self, substring, curr_node=None):
        if len(substring) == 0:
            return True
```

```
        if curr_node is None:
            curr_node = self.root

        first_letter = substring[0]
        next_node = curr_node.children.get(first_letter)
        if next_node is None:
            return False

        childs_text_len = next_node.end - next_node.start + 1
        for i in range(1, childs_text_len):
            if i == len(substring):
                return True
            if self.full_text[next_node.start + i] != substring[i]:
                return False

        return self.search(substring[childs_text_len:], next_node)
```

1.5 Ad. 5 - sprawdzenie poprawności obu struktur

Odpalając poniższą komórkę należy się liczyć z tym, że wyliczanie jej zajmie trochę czasu. Ze względu na dużą złożoność przeszukiwania i tworzenia struktury Trie - ostatniego tekstu nie zbadano

[7]:
```
err_acc = 0
invalid_texts = ["xkcd", "abdc", "adc", "cda"]
for text in texts[:-1]:
    trie = Trie(text)
    for i in range(0, len(text)):
        for j in range(i, len(text)):
            if not trie.search(text[i:j]):
                err_acc += 1

for text in invalid_texts:
    if trie.search(text):
        err_acc += 1

print(f"Zakończono testy konstrukcji i wyszukiwania Trie. {err_acc} błędów")

err_acc = 0
for text in texts:
    sTree = SuffixTree(text)
    for i in range(0, len(text)):
        for j in range(i, len(text)):
            if not sTree.search(text[i:j]):
                err_acc += 1

for text in invalid_texts:
```

```
    if sTree.search(text):
        err_acc += 1

print(f"Zakończono testy konstrukcji i przeszukiwania SuffixTree. {err_acc}␣
↪błędów")
```

```
Zakończono testy konstrukcji i wyszukiwania Trie. 0 błędów
Zakończono testy konstrukcji i przeszukiwania SuffixTree. 0 błędów
```

1.6 Ad. 6

Testy czasowe

[8]:
```
import time

def check_time(structure):
    for i, text in enumerate(texts):
        a = time.time()
        textObject = structure(text)
        b = time.time()
        print(f'Czas konstrukcji {structure.__name__} dla tekstu nr {i+1}:␣
↪{round(b-a, 6)} [s]')

        a = time.time()
        textObject.search("abcd")
        b = time.time()
        print(f'Czas przeszukiwania tekstu "abcd" w {structure.__name__}␣
↪skonstruowanym dla tekstu nr {i+1}: {round(b-a, 4)} [s]')

check_time(Trie)
print()
check_time(SuffixTree)
```

```
Czas konstrukcji Trie dla tekstu nr 1: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w Trie skonstruowanym dla tekstu nr 1: 0.0 [s]
Czas konstrukcji Trie dla tekstu nr 2: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w Trie skonstruowanym dla tekstu nr 2: 0.0 [s]
Czas konstrukcji Trie dla tekstu nr 3: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w Trie skonstruowanym dla tekstu nr 3: 0.0 [s]
Czas konstrukcji Trie dla tekstu nr 4: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w Trie skonstruowanym dla tekstu nr 4: 0.0 [s]
Czas konstrukcji Trie dla tekstu nr 5: 6.200201 [s]
Czas przeszukiwania tekstu "abcd" w Trie skonstruowanym dla tekstu nr 5: 0.0 [s]

Czas konstrukcji SuffixTree dla tekstu nr 1: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w SuffixTree skonstruowanym dla tekstu nr 1:
0.0 [s]
Czas konstrukcji SuffixTree dla tekstu nr 2: 0.0 [s]
```

```
Czas przeszukiwania tekstu "abcd" w SuffixTree skonstruowanym dla tekstu nr 2:
0.0 [s]
Czas konstrukcji SuffixTree dla tekstu nr 3: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w SuffixTree skonstruowanym dla tekstu nr 3:
0.0 [s]
Czas konstrukcji SuffixTree dla tekstu nr 4: 0.0 [s]
Czas przeszukiwania tekstu "abcd" w SuffixTree skonstruowanym dla tekstu nr 4:
0.0 [s]
Czas konstrukcji SuffixTree dla tekstu nr 5: 0.028275 [s]
Czas przeszukiwania tekstu "abcd" w SuffixTree skonstruowanym dla tekstu nr 5:
0.0 [s]
```

Jak widać struktura drzewa sufiksów jest o wiele szybsza, zajmuje również zdecydowanie mniej pamięci. Jednak dla czasu wyszukiwania małych wzorców nie ma większej różnicy między jedną a drugą strukturą. Ciągle lepiej jest jednak używać drzew sufiksowych, ponieważ nie zabierają one tak dużo miejsca w pamięci komputera.

[ ]: