lab_6_Aga_Patro

June 4, 2023

1 Aga Patro - lab 6

1.1 Zadanie 1. Przyjmij następujący zbiór danych wejściowych:

1.1.1 1.1 bbb$

[ ]:
```
data_0 = "bbb$"
```

1.1.2 1.2 aabbabd

[ ]:
```
data_1 = "aabbabd"
```

1.1.3 1.3 ababcd

[ ]:
```
data_2 = "ababcd"
```

1.1.4 1.4 abaababaabaabaabab$

[ ]:
```
data_3 = "abaababaabaabaabab$"
```

1.1.5 1.5 losowy tekst o długości 100 znaków

[ ]:
```
import string
import random

data_4 = "".join(random.choice(string.ascii_lowercase) for _ in range(96)) +␣
↪"qrpu"
```

1.1.6 1.6 załączony plik

[ ]:
```
with open("1997_714_head.txt") as file:
    data_5 = file.read()
```

1.2 Zadanie 2. Upewnij się, że każdy łańcuch na końcu posiada unikalny znak (marker), a jeśli go nie ma, to dodaj ten znak.

1.2.1 Za pomocą poniższego kodu upewniłam się, że każdy z łańcuchów zawiera na końcu unikalny marker "$". Jeśli marker znajdował się w środku łańcucha to przeniosłam go na koniec łańcucha, a jeśli go brakowało to dodałam go tam dodałam.

[ ]:
```
datas = [data_0, data_1, data_2, data_3, data_4, data_5]

def check_unique_markers(data):
    try:
        mark_index = data.index("$")
    except ValueError:
        data += "$"
        return data

    if mark_index == len(data) - 1:
        return data
    else:
        new_data = data[:mark_index] + data[mark_index + 1:] + "$"
        return new_data

counter = 0
for data in datas:
    data = check_unique_markers(data)
    print(f"-------------- DATA_{counter} --------------")
    print(f"Ostatnie 4 znaki łańcucha: ...{data[-4:]}\n\n")
    counter += 1
```

```
-------------- DATA_0 --------------
Ostatnie 4 znaki łańcucha: …bbb$


-------------- DATA_1 --------------
Ostatnie 4 znaki łańcucha: …abd$


-------------- DATA_2 --------------
Ostatnie 4 znaki łańcucha: …bcd$


-------------- DATA_3 --------------
Ostatnie 4 znaki łańcucha: …bab$


-------------- DATA_4 --------------
Ostatnie 4 znaki łańcucha: …rpu$
```

```
-------------- DATA_5 --------------
Ostatnie 4 znaki łańcucha: …m".$
```

1.3 Zadanie 3. Zaimplementuj algorytm konstruujący strukturę trie, która przechowuje wszystkie sufiksy łańcucha danego na wejściu.

[ ]:
```python
class TrieNode:
    def __init__(self, parent = None) -> None:
        self.parent = parent
        self.kids = dict()

    def create_path(self, text, index):
        if index == len(text):
            return

        elif text[index] not in self.kids.keys():
            self.kids[text[index]] = TrieNode(text[index])

        self.kids[text[index]].create_path(text, index + 1)

    def search(self, text, index):
        if len(text) == index:
            return True

        if text[index] in self.kids.keys():
            return self.kids[text[index]].search(text, index + 1)

        return False

def build_trie_tree(text):
    root = TrieNode()
    for index in range(len(text)):
        root.create_path(text, index)

    return root

def find_pattern_with_trie(text, pattern):
    tree = build_trie_tree(text)

    return tree.search(pattern, 0)
```

1.4 Zadanie 4. Zaimplementuj algorytm konstruujący drzewo sufiksów.

[ ]:
```
class SuffixEdge:
    def __init__(self, p, q) -> None:
        self.start = p
        self.end = q

class SuffixNode:
    def __init__(self) -> None:
        self.kids = dict()

    def create_path(self, text, p, q):
        letter = text[p]

        if letter not in self.kids.keys():
            self.kids[letter] = (SuffixEdge(p, q), SuffixNode())
        else:
            edge, node = self.kids[letter]
            edge_len = edge.end - edge.start + 1
            text_len = q - p + 1

            eq_len = 1
            for i in range(1, min(edge_len, text_len)):
                index_edge = edge.start + i
                index_text = p + i
                if text[index_edge] != text[index_text]:
                    break
                eq_len += 1

            if eq_len == edge_len:
                if eq_len != text_len:
                    node.create_path(text, p + edge_len, q)
                return

            new_node = SuffixNode()
            self.kids[letter] = (SuffixEdge(edge.start, edge.start + eq_len -␣
↪1), new_node)
            new_node.kids[text[edge.start + eq_len]] = (SuffixEdge(edge.start +␣
↪eq_len, edge.end), node)

            if eq_len != text_len:
                new_node.create_path(text, p + eq_len, q)

    def search(self, text, pattern):
        if pattern == '':
            return True
        if pattern[0] not in self.kids.keys():
```

```
            return False

        edge, next_node = self.kids[pattern[0]]
        m = edge.end - edge.start + 1
        n = len(pattern)
        if n <= m:
            return text[edge.start : edge.start + n] == pattern

        return text[edge.start : edge.end + 1] == pattern[:m] and next_node.
 ↪search(text, pattern[m:])

def build_suffix_tree(text):
    root = SuffixNode()
    lenght = len(text)
    counter = 0

    for index in range(lenght-1):
        counter += 1
        root.create_path(text, index, lenght-1)

    return root

def find_pattern_with_suffix(text, pattern):
    tree = build_suffix_tree(text)

    return tree.search(text, pattern)
```

1.5 Zadanie 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.

1.5.1 Algorytm Knutha-Morrisa-Pratta służący do wyszukiwania wzorców w tekście

[ ]:
```
def prefix_function(pattern):
    lps = [0] * len(pattern)
    l = 0
    i = 1

    while i < len(pattern):
        while l > 0 and pattern[i] != pattern[l]:
            l -= 1

        if pattern[i] == pattern[l]:
            l += 1

        lps[i] = l
        i += 1
```

```
    return lps

def kmp_string_matching(text, pattern, lps):
    result = []
    i = 0
    j = 0

    while i < len(text):
        if text[i] != pattern[j]:
            if j > 0:
                j = lps[j-1]
            else:
                i += 1
        else:
            i, j = i+1, j+1
            if j == len(pattern):
                result.append(i-j)
                j = lps[j-1]

    return len(result)

def is_there_pattern(text, pattern):
    if kmp_string_matching(text=text, pattern=pattern,␣
 ↪lps=prefix_function(pattern)) > 0:
        return True
    else:
        return False
```

1.5.2 Testy czy moje drzewa działają:

[ ]:
```
valid_patterns = ["aba", "aabaa", "qrpu"]
invalid_patterns = ["xyz", "2137", "olc"]
texts = [data_2, data_3, data_4]

for test_nb in range(len(texts)):
    print(f"-------------- TEST {test_nb} --------------\n")
    print(f"Checking text: {texts[test_nb]}")
    print(f"Valid pattern: {valid_patterns[test_nb]}")
    print(f"Is it there: {is_there_pattern(texts[test_nb],␣
 ↪valid_patterns[test_nb])} ")
    print(f"Trie tree: {find_pattern_with_trie(texts[test_nb],␣
 ↪valid_patterns[test_nb])}")
    print(f"Suffix tree: {find_pattern_with_suffix(texts[test_nb],␣
 ↪valid_patterns[test_nb])}\n")
    print(f"Inalid pattern: {invalid_patterns[test_nb]}")
```

```
    print(f"Is it there: {is_there_pattern(texts[test_nb],␣
↪invalid_patterns[test_nb])}")
    print(f"Trie tree: {find_pattern_with_trie(texts[test_nb],␣
↪invalid_patterns[test_nb])}")
    print(f"Suffix tree: {find_pattern_with_suffix(texts[test_nb],␣
↪invalid_patterns[test_nb])}\n\n")
```

```
-------------- TEST 0 --------------

Checking text: ababcd
Valid pattern: aba
Is it there: True
Trie tree: True
Suffix tree: True

Inalid pattern: xyz
Is it there: False
Trie tree: False
Suffix tree: False


-------------- TEST 1 --------------

Checking text: abaababaabaabaabab$
Valid pattern: aabaa
Is it there: True
Trie tree: True
Suffix tree: True

Inalid pattern: 2137
Is it there: False
Trie tree: False
Suffix tree: False


-------------- TEST 2 --------------

Checking text: ekrrhdhpaptmzbongponqtgvllzmoyyrdyczajjitkyoxmpltcqncsmfieenewwga
vuhumhubyfikyuxdpbozbebtneklfwfqrpu
Valid pattern: qrpu
Is it there: True
Trie tree: True
Suffix tree: True

Inalid pattern: olc
Is it there: False
Trie tree: False
```

```
Suffix tree: False
```

1.5.3 Z powyższych testów wynika, że moje drzewa działają

1.6 Zadanie 6. Porównaj szybkość działania algorytmów konstruujących struktury danych dla danych z p. 1.

[ ]:
```python
from timeit import default_timer as timer
valid_patterns = ["bb", "babd", "bab", "aaba", "qrpu", "fizyczne"]

def compare_time(test_number, data):
    start_trie = timer()
    trie_tree = build_trie_tree(data)
    end_trie = timer()
    trie_time = end_trie - start_trie

    start_suffix = timer()
    suffix_tree = build_suffix_tree(data)
    end_suffix = timer()
    suffix_time = end_suffix - start_suffix

    print(f"-------------- TIME FOR DATA_{test_number} --------------\n")
    if test_number == 5:
        print(f"Tekst: załączony plik ustawy")
    elif test_number == 4:
        print(f"Tekst: łańcuch 100 randomowych znaków")
    else:
        print(f"Tekst: {data}")
    print(f"Building trie tree: {trie_time}")
    print(f"Building suffix tree: {suffix_time}\n\n")
```

[ ]:
```python
datas = [data_0, data_1, data_2, data_3, data_4, data_5]

for index in range(len(datas)):
    compare_time(index, datas[index])
```

```
-------------- TIME FOR DATA_0 --------------

Tekst: bbb$
Building trie tree: 1.9626000721473247e-05
Building suffix tree: 2.3815999156795442e-05


-------------- TIME FOR DATA_1 --------------
```

```
Tekst: aabbabd
Building trie tree: 3.289500091341324e-05
Building suffix tree: 2.4304998078150675e-05


-------------- TIME FOR DATA_2 --------------

Tekst: ababcd
Building trie tree: 2.1161999029573053e-05
Building suffix tree: 1.410699769621715e-05


-------------- TIME FOR DATA_3 --------------

Tekst: abaababaabaabaabab$
Building trie tree: 0.0001406600022164639
Building suffix tree: 0.00021420300254249014


-------------- TIME FOR DATA_4 --------------

Tekst: łańcuch 100 randomowych znaków
Building trie tree: 0.007597126001201104
Building suffix tree: 0.0002929839974967763


-------------- TIME FOR DATA_5 --------------

Tekst: załączony plik ustawy
Building trie tree: 8.792844731000514
Building suffix tree: 0.0152418130019214
```

1.7 Zadanie 7. Dla załączonego tekstu czas wariantów 1 i 2 może być nieakceptowalnie długi - w tej sytuacji pomiń wyniki pomiarów dla tego tekstu.

1.7.1 Jak widać na powyższych wynikach, budowanie Trie Tree dla załączonego pliku zajęło “tylko” 8s, i myślę, że jest to akceptowalny czas :)