Antoni Szczepański - Raport lab1

Algorytmy wyszukiwania wzorca:

```python
def naive_string_matching(text, pattern):
    shifts=[]
    for s in range(0, len(text) - len(pattern) + 1):
        if(pattern == text[s:s+len(pattern)]):
            shifts.append(s)
            #print(f"Przesuniecie {s} jest poprawne")
    return shifts
```

```python
#AUTOMAT SKONCZONY
import re
def transition_table(pattern):
    alfabet=set()
    for a in pattern:
        alfabet.add(a)
    result = []
    for q in range(0, len(pattern) + 1):
        result.append({})
        for a in alfabet:
            k = min(len(pattern), q + 1)
            while True:
                if(re.search(f"{pattern[:k]}$", pattern[:q] + a)):
                    break
                k-=1
            result[q][a] = k
    return result

def fa_string_matching(text, pattern):
    shifts=[]
    q = 0
    delta=transition_table(pattern)
    for s in range(0, len(text)):
        if text[s] in delta[q].keys():
            q = delta[q][text[s]]
            if(q == len(delta) - 1):
                shifts.append(s+1-q)
                #print(f"Przesuniecie {s + 1 - q} jest poprawne")
                # s + 1 - ponieważ przeczytaliśmy znak o indeksie s, więc przesuniecie jest po tym znaku
        else:
            q=0
    return shifts
```

```python
#ALGORYTM Knutha-Morrisa-Pratta
def prefix_function(pattern):
    pi = [0]
    k = 0
    for q in range(1, len(pattern)):
        while(k > 0 and pattern[k] != pattern[q]):
            k = pi[k-1]
        if(pattern[k] == pattern[q]):
            k = k + 1
        pi.append(k)
    return pi

def kmp_string_matching(text, pattern):
    shifts=[]
    pi = prefix_function(pattern)
    q = 0
    for i in range(0, len(text)):
        while(q > 0 and pattern[q] != text[i]):
            q = pi[q-1]
        if(pattern[q] == text[i]):
            q = q + 1
        if(q == len(pattern)):
            shifts.append(i+1-q)
            #print(f"Przesuniecie {i + 1 - q} jest poprawne")
            q = pi[q-1]
    return shifts
```

Zad 1- testy

```
def tests(text,pattern):
    x=time.time()
    naive_string_matching(text,pattern)
    y=time.time()
    print(f"Naive string matching time {y-x}")
    x=time.time()
    fa_string_matching(text,pattern)
    y=time.time()
    print(f"FA string matching time {y-x}")
    x=time.time()
    kmp_string_matching(text,pattern)
    y=time.time()
    print(f"KMP string matching time {y - x}")
```

Przykładowe testy:

```
tests("agh"*100000, "agh")
```

```
Naive string matching time 0.12499547004699707
FA string matching time 0.17603778839111328
KMP string matching time 0.13902926445007324
```

```
tests("tekstowe"*50000, "tekst")
```

```
Naive string matching time 0.16196036338806152
FA string matching time 0.21889185905456543
KMP string matching time 0.15989351272583008
```

Zad 2

```
def art_w_ustawie():
    f=open("lab1_ustawa.txt","r",encoding='utf-8')
    ustawa=f.read()
    print("Naiwny przesuniecia:")
    print(naive_string_matching(ustawa, "art"))
    print("Naiwny liczba przesuniec",len(naive_string_matching(ustawa,"art")))
    print("FA przesuniecia:")
    print(fa_string_matching(ustawa, "art"))
    print("FA liczba przesuniec", len(fa_string_matching(ustawa, "art")))
    print("KMP przesuniecia:")
    print(kmp_string_matching(ustawa, "art"))
    print("KMP liczba przesuniec", len(kmp_string_matching(ustawa, "art")))
```

Zad 3

```
def art_w_ustawie_czasy():
    f = open("lab1_ustawa.txt", "r", encoding='utf-8')
    ustawa = f.read()
    tests(ustawa,"art")
```

Czasy wystąpienia wzorca „art” w ustawie:

```
Naive string matching time 0.09206962585449219
FA string matching time 0.08895349502563477
KMP string matching time 0.10099911689758301
```

Zad 4

```
def zad4():
    text="agh"*80000
    pattern="agh"*22000
    x=time.time()
    naive_string_matching(text,pattern)
    y=time.time()
    print(f"Czas algorytm naiwny: {y-x}")
    #print("Czas FA bez pre-processingu:",fa_string_matching_zad4(text,pattern))
    #obliczenie tablicy przejscia trwa bardzo dlugo gdy pattern jest dlugi
    print("Czas KMP bez pre-processingu:", kmp_string_matching_zad4(text, pattern))
```

```
Czas algorytm naiwny: 1.0357356071472168
Czas KMP bez pre-processingu: 0.14504170417785645
```

Samo dopasowanie w algorytmach fa i kmp ma czas liniowy, natomiast algorytm naiwny działa w czasie O((n-m+1)*m), dlatego tez czas algorytmu naiwnego w tym przypadku jest co najmniej 5 razy dłuższy.

Na potrzeby zadania 4 stworzyłem nowe algorytmy fa i kmp które zwracały czasy bez pre-processingu, mianowicie:

```
def fa_string_matching_zad4(text, pattern):
    shifts=[]
    q = 0
    delta=transition_table(pattern)
    x=time.time()
    for s in range(0, len(text)):
        if text[s] in delta[q].keys():
            q = delta[q][text[s]]
            if(q == len(delta) - 1):
                shifts.append(s+1-q)
                # s + 1 - ponieważ przeczytaliśmy znak o indeksie s, więc przesunięcie jest po tym znaku
        else:
            q=0
    y=time.time()
    return y-x
def kmp_string_matching_zad4(text, pattern):
    shifts=[]
    pi = prefix_function(pattern)
    q = 0
    x=time.time()
    for i in range(0, len(text)):
        while(q > 0 and pattern[q] != text[i]):
            q = pi[q-1]
        if(pattern[q] == text[i]):
            q = q + 1
        if(q == len(pattern)):
            shifts.append(i+1-q)
            q = pi[q-1]
    y=time.time()
    return y-x
```

Zad 5

```python
def zad5():
    pattern="agh"*150
    x = time.time()
    transition_table(pattern)
    y = time.time()
    print(f"Czas tablicy przejscia: {y - x}")
    x = time.time()
    prefix_function(pattern)
    y = time.time()
    print(f"Czas funkcji przejscia: {y-x}")
```

```
Czas tablicy przejscia: 0.9638714790344238
Czas funkcji przejscia: 0.0
```

Obliczenie tablicy przejścia trwa długo dla długich patternów, dlatego zaproponowałem taki wzorzec.