Algorytmy tekstowe – Dopasowanie wzorców

Tomasz Gargula

1 Implementacja algorytmów

1.1 Algorytm naiwny

```
def naive(data, pattern):
    n = len(data)
    m = len(pattern)
    result = []

    for s in range(n - m + 1):
        if pattern == data[s:s+m]:
            result.append(s)

    return result
```

1.2 Automat skończony

Do działania automatu skończonego potrzebne jest wygenerowanie tablicy przejścia.

```
def transition_table(pattern):
    Q = range(len(pattern) + 1)
    Sigma = set(pattern)
    delta = []

    for q in Q:
        delta.append({})
        for letter in Sigma:
            k = min(len(Q), q + 2) - 1
            text = pattern[:q] + letter
            while not text.endswith(pattern[:k]):
                k -= 1

            delta[q][letter] = k

    return delta
```

Zaimplementowano dwie wersje algorytmów (z wykonanym preprocessingiem i bez), aby ułatwić korzystanie z funkcji na późniejszym etapie.

```
def fa(data, pattern):
    delta = transition_table(pattern)
    return fa_preprocessed(data, delta)


def fa_preprocessed(data, delta):
    q = 0
    result = []

    for s, letter in enumerate(data):
```

```
        q = delta[q].get(letter, 0)
        if q + 1 == len(delta):
            result.append(s - q + 1)

    return result
```

1.3 Algorytm Knutha-Morrisa-Pratta

Poniżej zamieszczono kod z funkcją, która wyznacza funkcję przejścia niezbędną do prawidłowego działania algorytmu Knutha-Morrisa-Pratta.

```
def prefix_function(pattern):
    m = len(pattern)
    pi = [0] * m
    k = 0

    for q in range(1, m):
        while k > 0 and pattern[k] != pattern[q]: # check back
            k = pi[k - 1]
        if pattern[k] == pattern[q]: # letters match
            k += 1
        pi[q] = k

    return pi
```

Analogicznie jak w przypadku automatu skończonego, zaimplementowano dwie wersje algorytmu.

```
def kmp(data, pattern, pi=None):
    pi = prefix_function(pattern)
    return kmp_preprocessed(data, pattern, pi)


def kmp_preprocessed(data, pattern, pi):
    n = len(data)
    m = len(pattern)
    q = 0                # number of matching symbols
    result = []
    for i, letter in enumerate(data):
        while q > 0 and pattern[q] != letter:
            q = pi[q - 1]
        if pattern[q] == letter:
            q += 1
        if q == m:
            result.append(i - m + 1)
            q = pi[q - 1]

    return result
```

2 Testy porównujące szybkość działania

Czas działania algorytmu był mierzony poprzez różnicę czasów po i przed wykonaniem funkcji. W tym celu użyłem funkcji `time` z biblioteki `time`.

```
from time import time

def test(f, *args):
    start = time()
    f(*args)
    return time() - start
```

3 Wyszukiwanie wystąpień wzorca

Aby zademonstrować działanie algorytmów napisany został algorytm, znajdujący wszystkie wystąpienia wzorca "art" w ***załączonej ustawie***. Ten dokument został umieszczony w wersji tekstowej w folderze `data`.

```
from pathlib import Path

from algorithms.pattern_matching import fa, kmp, naive

if __name__ == '__main__':
    text = Path('data', 'act.txt').open('r').read()
    pattern = 'art'

    for f in (naive, fa, kmp):
        print(f'{f.__name__}:')
        print(f(text, pattern))
        print()
```

4 Porównanie szybkości działania

4.1 Wyszukiwanie wzorca "art" w ustawie

```
from pathlib import Path
from pprint import pprint

from algorithms.pattern_matching import fa, kmp, naive
from algorithms.time_test import test

if __name__ == '__main__':
    text = Path('data', 'act.txt').open('r').read()
    pattern = 'art'
    result = {}

    for f in (naive, fa, kmp):
        result[f.__name__] = test(f, text, pattern)

    pprint(result)
```

Wynik działania algorytmu:
`{'fa': 0.03156280517578125, 'kmp': 0.018553495407104492, 'naive': 0.023760318756103516}`

4.2 Propozycja danych, dla których czas działania algorytmu naiwnego jest co najmniej pięciokrotnie dłuższy od czasu działania automatu skończonego i algorytmu Knutha-Morrisa-Pratta

Zaproponowany tekst to: `('a' * 10000 + 'b' + 'a' * 100 + 'c') * 100`
Zaproponowany wzorzec to: `'a' * 7000 + 'b'`

```
from pprint import pprint

from algorithms.pattern_matching import fa_preprocessed as fa
from algorithms.pattern_matching import kmp_preprocessed as kmp
from algorithms.pattern_matching import naive, prefix_function, transition_table
from algorithms.time_test import test

if __name__ == '__main__':
    text = ('a' * 10000 + 'b' + 'a' * 100 + 'c') * 100
    pattern = 'a' * 7000 + 'b'
```

```
    result = {}

    delta = transition_table(pattern)
    pi = prefix_function(pattern)

    result['naive'] = test(naive, text, pattern)
    result['fa'] = test(fa, text, delta)
    result['kmp'] = test(kmp, text, pattern, pi)

    print(result['naive'] / result['fa'] > 5 and result['naive'] / result['kmp'] > 5)

    pprint(result)
```

Wynik działania algorytmu:

```
True {'fa': 0.1463453769683838, 'kmp': 0.13521957397460938, 'naive': 0.7533395290374756}
```

4.3 Propozycja wzorca, dla którego czas obliczenia tablicy przejścia automatu skończonego będzie co najmniej pięciokrotnie dłuższy niż czas potrzebny na utworzenie funkcji przejścia w algorytmie Knutha-Morrisa-Pratta

Zaproponowany wzorzec to: `'Wstawaj samuraju!'`

```
from pprint import pprint

from algorithms.pattern_matching import prefix_function, transition_table
from algorithms.time_test import test

if __name__ == '__main__':

    pattern = 'Wstawaj samuraju!'

    result = {}

    result['transition_table'] = test(transition_table, pattern)
    result['prefix_function'] = test(prefix_function, pattern)

    print(result['transition_table'] / result['prefix_function'] > 5)
    pprint(result)
```

Wynik działania algorytmu:

```
True {'prefix_function': 3.0994415283203125e-06, 'transition_table': 0.0003154277801513672}
```