Zadanie 1 - pattern matching

March 8, 2021

1 Pattern maching - wyszukiwanie wzorców

1.1 Paweł Kruczkiewicz

03.03.2021 r.

1.1.1 Treść

Zaimplementuj algorytmy wyszukiwania wzorca:

```
a) naiwny
b) automat skończony
c) algorytm KMP
```

1. Zaimplementuj testy porównujące szybkość działania wyżej wymienionych algorytmów.
2. Znajdź wszystkie wystąpienia wzorca "art" w załączonej ustawie, za pomocą każdego algorytmu.
3. Porównaj szybkość działania algorytmów dla problemu z p. 2.
4. Zaproponuj tekst oraz wzorzec, dla którego zmierzony czas działania algorytmów (uwzględniający tylko dopasowanie, bez pre-processingu) automatu skończonego oraz KMP będzie co najmniej 5-krotnie krótszy niż dla algorytmu naiwnego.
5. Zaproponuj wzorzec, dla którego zmierzony czas obliczenia tablicy przejścia automatu skończonego będzie co najmniej 5-krotnie dłuższy, niż czas potrzebny na utworzenie funkcji przejścia w algorytmie KMP.

UWAGA - przed przejściem do części rozwijającej wyżej opisane punkty, należy uruchomić wszystkie komórki z części `Implementacje` w kolejności ich umieszczenia

1.2 Implementacje

1.2.1 Algorytm naiwny

Jest to najprostsze podejście do zadania - kod jest prosty, czytelny, jednak ma sporą złożoność (kwadratową)

[4]:
```
def pattern_matching_naive(word, pattern):
    result = []
    n = len(word)
    m = len(pattern)
```

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

[8]:
```
print(f'Wzorzec występuje z przesunięciami: {pattern_matching_naive("abbab",␣
↪"ab")}')
```

```
Wzorzec występuje z przesunięciami: [0, 3]
```

1.2.2 Automat skończony

Wyliczanie funkcji delta: W procesie przygotowawczym algorytmu musimy przygotować funkcję "delta", czyli funkcję przejścia w naszym automacie. Funkcja obliczająca funkcję delta przyjmuje wzorzec i alfabet; zwraca macierz (tablicę) o wymiarach {dł. wzorca} x {moc alfabetu}, będącą żądaną funkcją Obliczanie tejże funkcji polega na rozpatrzeniu dla każdego możliwego elementu iloczynu kartezjańskiego przejścia stanu i alfabetu dwóch przypadków:

```
a) gdy i-ty element przechodzi "poprawnie" - przechodzimy o stanu o jeden większego
b) wpp szukamy najlepszego stanu "mniejszego" dopasowania
```

[6]:
```
def compute_delta(pattern):
    def is_suffix(potential_suffix, main_word):
        m = len(potential_suffix)
        return main_word[-m:] == potential_suffix

    def compute_alphabet(pattern):
        result = set()
        for ch in pattern:
            result.add(ch)
        return result

    m = len(pattern)
    alphabet = compute_alphabet(pattern)
    delta = [dict() for _ in range(m+1)]
    for q in range(m + 1):
        for elem in alphabet:
            k = min(m, q + 1)
            while k > 0 and not is_suffix(pattern[:k], pattern[:q] + elem):
                k -= 1
            delta[q][elem] = k
    return delta
```

[7]:
```
print(compute_delta("abb"))
```

```
[{'b': 0, 'a': 1}, {'b': 2, 'a': 1}, {'b': 3, 'a': 1}, {'b': 0, 'a': 1}]
```

Automat

[9]:
```python
def pattern_matching_finite_automate(word, pattern, delta=None):
    result = []
    n, m, q = len(word), len(pattern), 0
    if delta is None:  delta = compute_delta(pattern)

    for s, ch in enumerate(word):
        q = delta[q][ch] if ch in delta[q] else 0
        if q == m:
            result.append(s + 1 - m)
    return result
```

[11]:
```python
print(f'Wzorzec występuje z przesunięciami:
 ↪{pattern_matching_finite_automate("abbab", "ab")}')
```

```
Wzorzec występuje z przesunięciami:[0, 3]
```

1.2.3 Algorytm Knutha-Morrisa-Pratta

Obliczanie funkcji delta

[12]:
```python
def compute_pi(pattern):
    m = len(pattern)
    pi = [0 for _ in range(m)]
    k = 0
    for q in range(1, m):
        while k > 0 and pattern[k] != pattern[q]:
            k = pi[k-1]
        if pattern[k] == pattern[q]:
            k += 1
        pi[q] = k
    return pi
```

Wyszukiwanie wzorców KMP

[17]:
```python
def pattern_matching_KMP(word, pattern, pi = None):
    result = []
    n, m, q = len(word), len(pattern), 0
    if pi is None: pi = compute_pi(pattern)

    for i in range(n):
        while q > 0 and pattern[q] != word[i]:
            q = pi[q-1]
        if pattern[q] == word[i]:
            q += 1
        if q == m:
            result.append(i - q + 1)
            q = pi[q-1]
    return result
```

[20]:
```
print(f'Wzorzec występuje z przesunięciami: {pattern_matching_KMP("abbab",␣
↪"ab")}')
```

```
Wzorzec występuje z przesunięciami: [0, 3]
```

1.3 Ad 1. - testy szybkości wyszukiwania wzorca

1.3.1 Generator danych

Generator tworzy:

1. Słowo o długości n
2. Wzorzec o długości m

Użyty alfabet to `['a', 'b']`, gdyż nie ma ona wpływu na złożoność wyszukiwania wzorca

[21]:
```
import random

def random_word_and_pattern(n, m):
    assert m <= n, "The pattern length is longer than the word itself"
    alphabet = list(map(chr, list(range(97, 123))))   #wszsytkie małe litery␣
↪alfabetu łacińskiego
    return ''.join(map(str, [random.choice(alphabet) for _ in range(n)])),\
           ''.join(map(str, [random.choice(alphabet) for _ in range(m)]))
```

1.3.2 Testowanie

W testach sprawdzamy jedynie szybkość znajdowania wzorców, NIE uwzględniamy czasu potrzebnego na preprocessing (czyli obliczenie delty dla automatu skończonego oraz listy pi dla KMP)

[23]:
```
import time

def check_matching_time(func, word, pattern):
    if func is pattern_matching_naive:
        a = time.time()
        func(word, pattern)
        b = time.time()
    elif func is pattern_matching_finite_automate:
        delta = compute_delta(pattern)
        a = time.time()
        func(word, pattern, delta)
        b = time.time()
    else:
        pi = compute_pi(pattern)
        a = time.time()
        func(word, pattern, pi)
        b = time.time()
    return round(b-a, 4)
```

```python
def matching_time_test(functions, n_and_m_list):
    for n, m in n_and_m_list:
        word, pattern = random_word_and_pattern(n, m)
        print(f'n = {n}, m = {m}')
        for func in functions:
            print(func.__name__, check_matching_time(func, word, pattern))
        print()
```

[24]:

```python
functions = [pattern_matching_naive, pattern_matching_finite_automate,␣
 ↪pattern_matching_KMP]

n_and_m_list = []
for i in range(4, 8):
    for j in range(1, 4):
        if i >= j:
            n_and_m_list.append( (10**i, 10**j))

matching_time_test(functions, n_and_m_list)
```

```
n = 10000, m = 10
pattern_matching_naive 0.0
pattern_matching_finite_automate 0.001
pattern_matching_KMP 0.0

n = 10000, m = 100
pattern_matching_naive 0.0117
pattern_matching_finite_automate 0.0
pattern_matching_KMP 0.0

n = 10000, m = 1000
pattern_matching_naive 0.0
pattern_matching_finite_automate 0.0
pattern_matching_KMP 0.0

n = 100000, m = 10
pattern_matching_naive 0.0103
pattern_matching_finite_automate 0.0082
pattern_matching_KMP 0.0122

n = 100000, m = 100
pattern_matching_naive 0.01
pattern_matching_finite_automate 0.01
pattern_matching_KMP 0.01

n = 100000, m = 1000
pattern_matching_naive 0.0202
pattern_matching_finite_automate 0.0102
pattern_matching_KMP 0.0102
```

```
n = 1000000, m = 10
pattern_matching_naive 0.1003
pattern_matching_finite_automate 0.1021
pattern_matching_KMP 0.1117

n = 1000000, m = 100
pattern_matching_naive 0.1086
pattern_matching_finite_automate 0.1302
pattern_matching_KMP 0.11

n = 1000000, m = 1000
pattern_matching_naive 0.2197
pattern_matching_finite_automate 0.1099
pattern_matching_KMP 0.1017

n = 10000000, m = 10
pattern_matching_naive 1.1477
pattern_matching_finite_automate 1.0404
pattern_matching_KMP 1.1008

n = 10000000, m = 100
pattern_matching_naive 1.1153
pattern_matching_finite_automate 1.1038
pattern_matching_KMP 1.0971

n = 10000000, m = 1000
pattern_matching_naive 2.0896
pattern_matching_finite_automate 1.0961
pattern_matching_KMP 1.1031
```

Jak widać w wielu testach najlepiej poradził sobie automat skończony. Nie wliczono jednak czasu potrzebnego na przygotowanie algorytmu, który w przedstawionej tutaj wersji ma złożoność O(m^3 * |alfabet|)

Co ciekawe w niektórych testach KMP radził sobie gorzej niż algorytm naiwny. Jest tak ze względu na asymptotyczną złożoność algorytmu naiwnego (O(m(n-m+1)), która dla m równego w przybliżeniu 0 lub n jest niemal liniowa. Dodatkowo algorytm naiwny nie wykonuje kilku dodatkowych kroków, które wykonuje automat oraz KMP, a algorytm naiwny - nie. W pozostałych przypadkach jednak, algorytm naiwny przeegrywa z dwoma pozostałymi

1.3.3 Ad. 2 - wyszukiwanie wzorca w ustawie

[25]:
```
import codecs
functions = [pattern_matching_naive, pattern_matching_finite_automate,␣
↪pattern_matching_KMP]
```

```python
def read_file(path):
    result = ''
    with codecs.open(path, encoding='utf-8') as file:
        result = file.readlines()
    return ''.join(result)

def find_pattern_in_file(path, pattern, funcs):
    text = read_file(path)
    for func in funcs:
        print(f'Function: {func.__name__}')
        result = func(text, pattern)
        print(f'Number of patterns matched: {len(result)}')
        print(f'First 10 matches: {result[:10]}')
        print(f'Last 10 matches: {result[-10:]}')
        print()


find_pattern_in_file("ustawa.txt", "art", functions)
```

```
Function: pattern_matching_naive
Number of patterns matched: 273
First 10 matches: [1183, 1538, 4774, 4816, 4963, 5169, 5236, 6052, 6143, 7390]
Last 10 matches: [203653, 205754, 209919, 212346, 215026, 215535, 220746,
221200, 226562, 226648]

Function: pattern_matching_finite_automate
Number of patterns matched: 273
First 10 matches: [1183, 1538, 4774, 4816, 4963, 5169, 5236, 6052, 6143, 7390]
Last 10 matches: [203653, 205754, 209919, 212346, 215026, 215535, 220746,
221200, 226562, 226648]

Function: pattern_matching_KMP
Number of patterns matched: 273
First 10 matches: [1183, 1538, 4774, 4816, 4963, 5169, 5236, 6052, 6143, 7390]
Last 10 matches: [203653, 205754, 209919, 212346, 215026, 215535, 220746,
221200, 226562, 226648]
```

1.4 Ad. 3 - sprawność wyszukiwania w powyższym przykładzie

[24]:
```python
functions = [pattern_matching_naive, pattern_matching_finite_automate,␣
 ↪pattern_matching_KMP]

text = read_file("ustawa.txt")
for func in functions:
```

```
    print(f'Function: {func.__name__};\ntime: {check_matching_time(func, text,␣
↪"art")} [s]\n')
```

```
Function: pattern_matching_naive;
time: 0.0401 [s]

Function: pattern_matching_finite_automate;
time: 0.0359 [s]

Function: pattern_matching_KMP;
time: 0.0409 [s]
```

W powyższym teście najlepiej poradził sobie automat skończony. Czemu tak jest?

Aby to wyjaśnić, należy sobie najpierw uświadomić, że w powyższym przykładzie m « n, więc nawet algorytm naiwny jest niemal asymptotycznie liniowy. Mamy zatem trzy algorytmy o takiej samej asymptotycznej złożoności. O szybkości algorytmu w tym wypadku decyduje m. in. stała jego wykonania, czyli liczba kroków, jakie wykonuje się w dla każdego elementu. Dla automatu jest ona najmniejsza - są to dwa (w przypadku znalezienia dopasowania - cztery) jednostkowe kroki, czyli o wiele mniej niż w algorytmie naiwnym czy KMP.

1.5 Ad. 4 - tekst oraz wzorzec, dla którego naiwny algorytm jest 5-krotnie wolniejszy od automatu skończonego i KMP

Algorytm naiwny ma złożoność liniową dla m w przybliżeniu równego 0 lub n (co zostało wyżej wytłumaczone), jednak w przeciwnym przypadku będzie on kwadratowy. Po krótkiej analizie widać, że najłatwiej będzie ten problem zaobserwować dla `m = (1/2)*n`. Sprawdźmy to zatem.

UWAGA - poniższy algorytm wykonywałby się wolno ze względu na złożoność tworzenia funkcji delta przy automacie skończonym, dlatego użyto jeynie algorytmu KMP

[26]:
```
functions = [pattern_matching_naive, pattern_matching_KMP]

matching_time_test(functions, [(10**i, (10**i)//2) for i in range(4, 7)])
```

```
n = 10000, m = 5000
pattern_matching_naive 0.0
pattern_matching_KMP 0.0

n = 100000, m = 50000
pattern_matching_naive 0.0686
pattern_matching_KMP 0.0101

n = 1000000, m = 500000
pattern_matching_naive 7.7966
pattern_matching_KMP 0.1121
```

Wyniki mówią same za siebie.

1.6 Ad. 5 - wzorzec, dla którego tworzenie delty w automacie skończonym jest ponad 5-krotnie wolniejszy niż dla KMP

Takim wzorcem jest tekst niepowtarzających się znaków, np. `abcdefghijklmnoprstuvwxyz`, ponieważ algorytm musi sprawdzić każde możliwe przejście. Algorytm obliczający tablicę pi dla KMP przejdzie po powyższym liniowo.

Tworzenie delty w przedstawionej wyżej implementacji ma złożoność O((m^3)|alfabet|). Jednak nawet algorytm o najlepszej złożoności (O(m|alfabet|)) będzie posiadał wymienioną wyżej wadę.

[23]:
```python
import time, random

prep_funcs = [compute_delta, compute_pi]
patterns = ["abcdefghijklmnoprstuvwxyz",  #alfabet łaciński
            "                     \
                    .            \
                    ", #sto najpopularniejszych znaków w kanji
            ''.join(map(str, [chr(random.randint(128, 1000)) for _ in␣
↪range(500)]))] #500 losowych znaków ASCII

def check_time(pattern, func):
    a = time.time()
    func(pattern)
    b = time.time()
    return b-a

for pattern in patterns:
    print(f'pattern: {pattern}')
    for func in prep_funcs:
        print(f'Function: {func.__name__}\ntime: {check_time(pattern, func)}␣
↪[s]')
    print()
```

```
pattern: abcdefghijklmnoprstuvwxyz
Function: compute_delta
time: 0.006978511810302734 [s]
Function: compute_pi
time: 0.0 [s]

pattern:
         .
Function: compute_delta
time: 0.45897698402404785 [s]
Function: compute_pi
time: 0.0 [s]

pattern: ˝¯¬  Ɏ ¦ ÉŬĈ  ə ů ȑ ä  ¦ð ǽ éĉĺ  á ē ˚ Ħ¾ Ñĺ
È Á Ü ĿĽĎ  Ź É¬ ̧Ü ˚ ŝ ® Ç Π  Ĥ  ȧØ ŴŁŌĬț ğĶÜ˝  ü ĵ ŪŰïĦ ŵú Ĩ
 Ï Ǵ Ŭ ōĺ ǿ  Æ źŢ ś  ūû óĀ˘ Σī  ď ãű  ¦ ū Ż  Ġ¨ØĝĘģå Ő      ù
```

```
 ĩŅŰPãāĻīŗ š  X ü Đ  Đf  ȕũ  Î Ÿ · ŕ ō  Ė ÿ P   ď¡ș Ī
Ğĉ Ā Ø Ψ ¡   Π  ĮMŷḡ ŋ øāP·Ų Â Ľ åĊ İ ¾     Ķą ū Eİ
ì KØ©Ċ Ĝħ Ŷ ĕ ¨Θ Åȑċ Ŋ ŋÌ ŰŤ°  Ä ŬĐ  ŵ  ÷    ĪţĐ õ × Λ
· Ă ȈŔ ëϒ  ĠϒĜĦMÄ ¸ Ωŭ
Function: compute_delta
time: 38.20241856575012 [s]
Function: compute_pi
time: 0.0 [s]
```

Na powyższym przykładzie niestety trochę trudno jest zauważyć, czemu funkcja `compute_pi` jest o wiele szybsza z prostego powodu - radzi sobie zbyt dobrze. Dla powyższych przykładów, gdzie nie powtarzają się niemal żadne znaki, funkcja pi działa liniowo, więc aby w ogóle zmierzyć jej czas potrzeba wzorca długości przynajmniej rzędu 10^4. Niestety, już dla wzorca długości 500 czas wyliczania delty wynosi kilkadziesiąt sekund.

Widać jednak, że w opisanym wzorcu wyliczanie tablicy pi jest szybsze od wyliczania delty