Algorytm Galila–Seiferasa

Bruno Pitrus

Algorytm G–S jest algorytmem wyszukiwania wzorca który wymaga jedynie stałej pamięci dodatkowej (eliminując tablicę „mismatch” używaną przez algorytmy typu Morrisa–Pratta) a jednocześnie wciąż działa w asymptotycznym czasie liniowym — osiągając tym samym teoretyczne optimum pod względem zarówno czasu jaki i pamięci.

W tym opisie, tak samo jak w załączonej implementacji algorytmu, stringi indeksowane są od 1. Wszelki kod należy interpretować zgodnie z semantyką języka Python.

Niech `m = len(word)`, `n = len(text)`. Rozważmy schemat algorytmu dopasowania wzorca

```
p,q= 0,0
while True:
  while text[p+q+1] == word[q+1]: q+= 1
  if q == m: yield p+1
  p,q= p',q'
```

W powyższym pseudokodzie `p` to jest obecny kandydat na pozycję wzorca, a `q` to prefiks o którym wiemy że się zgadza.

Naiwny algorytm *brute-force* używa `p'= p+1`, `q'= 0`. Ponieważ wiemy jednak że `text[p+1:p+q+1] == word[1:q+1]` to rozważanie `p'= p+x` ma sens tylko jeśli `word[1:q-x+1] == word[x+1:q+1]`; na tej obserwacji jest oparty algorytm Knutha–Morrisa–Pratta, obliczający `p'= p+shift[q], q'= q-shift[q] if q>0 else 0`, gdzie `shift[q]` jest długością najkrótszego okresu `word[1:q+1]`.

Algorytm Galila–Seiferasa jest oparty na tym schemacie oraz następującym twierdzeniu:

Twierdzenie (O dekompozycji). *Dla ustalonego (odpowiednio dużego) k, każde słowo* `x` *można przedstawić jako* `u+v`*, gdzie* `v` *ma co najwyżej jeden k-okres prefiksu a* `u` *jest długości* O(*długość najkrótszego okresu* `v`*).*

Słowo `z` nazywamy *k*-okresem prefiksu słowa `w` jeśli `z` jest słowem pierwotnym a z^k jest prefiksem `w`.

Do skonstruowania tej dekompozycji przyda nam się następujący lemat:

Lemat. *Dla słowa pierwotnego* `w` *istnieje dekompozycja* `w == w1 + w2` *taka że dla dowolnego słowa* `w'`*, słowo* `w2 +` w^{k-1} `+ w'` *nie ma k-okresu prefiksu krótszego niż* `len(w)`*.*

Pierwsza część algorytmu jest poświęcona znalezieniu dekompozycji wzorca takiej jak w twierdzeniu o dekompozycji.

```
s= 0
while word[s+1:] ma więcej niż jeden k-okres prefiksu:
  p2= długość drugiego najkrótszego k-okresu prefiksu
  Korzystając z lematu, znajdź
   s' < s + p2 takie że word[s'+1:] nie ma k-okresu prefiksu krótszego niż p2
  s= s'
```

Niech `l(s)` będzie długością najkrótszego okresu `word[s+1:]` a `p1(s)` długością najkrótszego *k*-okresu prefiksu. Indukcyjnie można dowieść, że w każdej iteracji pętli `s < c * min(l(s), p1(s))` dla `c = (k-1)/(k-2)` — więc w szczególności na końcu `len(u) < c * l(s)`.

W szczególności istnieje następujący prosty algorytm znajdowania `s'`:

```
s'= s
while word[s'+1:] ma k-okres prefiksu krótszy niż p2:
  usuń najkrótszy okres prefiksu
```

Przeanalizujmy złożoność ostatecznego kodu programu:

```
K= 4
s,p1,q1= 0,1,0
p2,q2= 0,0
mode=0
while True:
  if mode==1:
    #Znajdź najkrótszy k-okres prefiksu word[s+1:]
    while s+p1+q1 < m and word[s+p1+q1+1] == word[s+q1+1]: q1+= 1
    if p1+q1 >= p1*K: p2, q2= q1,0; mode=2; continue
    if s+p1+q1 == m: break
    p1+= (1 if q1==0 else (q1+K-1)//K); q1= 0
  elif mode==2:
    #Znajdź drugi najkrótszy k-okres prefiksu word[s+1:]. Jeśli nie istnieje to przejdź do dru-
giej fazy algorytmu.
    while s+p2+q2 < m and word[s+p2+q2+1] == word[s+q2+1] and p2+q2 < p2*K: q2+= 1
    if p2+q2 == p2*K: mode= 0; continue;
    if s+p2+q2 == m: break
    if q2 == p1+q1:
      p2+= p1; q2-= p1;
    else:
      p2+= (1 if q2==0 else (q2+K-1)//K); q2= 0
  else:
    #Zinkrementuj s
    while s+p1+q1 < m and word[s+p1+q1+1] == word[s+q1+1]: q1+= 1
    while p1+q1 >= p1*K: s+= p1; q1-= p1;
    p1+= (1 if q1==0 else (q1+K-1)//K); q1= 0
    if p1 >= p2: mode= 1
```

W celu przeanalizowania złożoności tej części algorytmu rozważmy wyrażenie

$$2s + (k+1)p_1 + q_1 + (k+1)p_2 + q_2$$

Jego wartość jest zawsze `O(len(word))` a każde przypisanie ją zwiększa.

```
p2,q2= 0,0
while True:
  while p2+s+q2 < n and s+q2 < m and text[p2+s+q2+1] == word[s+q2+1]: q2+= 1
  if q2 == m-s and text[p2+1 : p2+s+1] == word[1 : s+1]:
    yield p2+1
  if q2 == p1+q1:
    p2+= p1; q2-= p1
  else:
    p2+= (1 if q2==0 else (q2+K-1)//K); q2= 0
  if p2+s > n: return
```

Przed wykonaniem drugiej części algorytmu jeśli `word[s+1:]` ma k-okres prefiksu, to jego długość wynosi p_1. Ponadto twierdzimy, że $s < (k-1)p_1/(k-2)$ co nam zagwarantuje że bezpośrednie sprawdzanie równości podciągów zajmie łącznie `O(len(text))` czasu.

Literatura

Zvi Galil, Joel Seiferas, *Time-Space-Optimal String Matching*, Journal of Computer and System Sciences **26**, 280–294 (1983)