Lista pierwsza z algorytmów textowych

Łukasz Klasinski

15 listopada 2020

Zadanie 1

Prove a stronger version of the periodicity lemma: if a string $s[1 \ldots n]$ has periods p and q, with $p + q \leq n + gcd(p, q)$, then $gcd(p, q)$ is also a period of s.

Bez straty ogólności załóżmy, że $p \geq q$. Mamy wtedy nastepujące przypadki:

- $p = q$

Wtedy trywialnie $gcd(p, q) = p = q$

- $p > q$ (bez straty ogólności)

Podzielmy słowo s na:

$$s = iw, |i| = p \text{ oraz } w \text{ to border } s \tag{1}$$
$$s = jv, |j| = q \text{ oraz } v \text{ to border } s \tag{2}$$

Fakt, że w oraz v są borderami s wynika z twierdzenia na wykładzie.

Aby udowodnić twiedzenie indukcyjnie pokażemy, że $p - q$ oraz q są okresami v. Dzięki temu indukcyjnie $q - (p - q)$ będzie również okresem v (korzystając za algorytmu do obliczania gcd).

- $p - q$ jest okresem v

Lemat 1 Jeśli x i y są borderami jakiegoś s i $|x| < |y|$, to x jest borderem y.

Lemat wynika z definicji borderów - jeśli x jest krótszym borderem od y, to znaczy, że musi mieć wspólny z nim prefix i postfix, zatem jest jego borderem.

Skoro $p > q$, to j jest prefiksem i. W takim razie $i = jj'$ dla jakiegoś j'. Ponieważ $s = iw = jv = jj'w$, to $v = j'w$. Z tego, że w jest krótszym borderem od v oraz lematu 1 wynika, że w jest borderem v. W takim razie $|j'| = p - q$, jest okresem v.

- q jest okresem v Aby to udowodnić, wystarczy pokazać, że $q \leq |v|$, bo inaczej v nie byłoby borderem s. Ustalmy $d = gcd(p, q)$. Jako, że $d \leq p - q$, mamy $q \leq p - d = p + q - d - q \leq |s| - q = |v|$.

Widać zatem, że $gcd(p, q)$ jest okresem v. Zauważmy teraz, że skoro q jest wielokrotnością $gcd(q, p)$, to nasz fragment j musi być wielokrotnością jakiegoś słowa j^*, ale w takim, razie ponieważ q jest okresem v, to v także jest pewną potęgą j^*. Zatem całe słowo $s = jv$ jest potęgą j^* z czego wynika, że $gcd(p, q)$ jest okresem s. ■

Zadanie 3

Write down the details of the streaming pattern matching algorithm that we have seen in the class (in pseudocode). Explain the invariants and provide a proof of correctness. Provide a precise explanation of the necessary modifications to make the algorithm only return false positives (and no false negatives).

Pseudokod dla algorytmu Porat & Porat

Algorytm składa się z 2 podprogramów - jeden który zarządza procesami i uruchamia je w odpowiednich miejscach tekstu, oraz samych procesów, które odpowiadają za znajdowanie patternu zaczynając od danego miejsca.

Wpierw wykonywany jest preprocessing na samym patternie w celu wyliczenia fingerprintów kolejnych podciągów patternu oraz jego borderów, używanych przy samym matchowaniu:

```
def preprocess(P=pattern):
  ϕ = []
  m = len(P)
  for i in 0..log(m):
    ϕ[i] = fingerprint(P[0..2^i])
    period[i] = min(P[0..2^i])
    periodϕ[i] = fingerprint(P[0..period[i]])
```

Następnie możemy wyodrębnić kod głównej pętli która przyjmuje dynamicznie dane ze strumienia i zarząda uruchamianiem/kończeniem procesów ktore osiągną dany status:

```
def porat_porat():
  zacznij process
  while input = read():
    wyślij input do wszystkich processów
    jeśli otrzymasz `abort` od jakieś procesu x:
      Uruchom nowy proces zaczynający od `input`
    jeśli jakiś process x zasygnalizuje `checkpoint`:
      zatrzymaj wszystkie podprocesy(x)
      zacznij nowy podproces dla x-a zaczynajac od `input`
  zatrzymaj wszystkie procesy
```

Na sam koniec pseudokod samego procesu machującego pattern do danych wejściowych:

```
def process():
  hash = None
  po otrzymaniu `input`:
    hash += fingerprint(input)  ## szybkie dodanie znaku do fingerprinta
    if |hash| == 2^i:
      if hash == ϕ[i]:
        signal `checkpoint` to main loop
      else:
        if hash[2^(i-1)] == periodϕ[i-1]:
          przesuń hash o period[i-1] (szybko)
        else:
          zakończ się
    if |hash| == |P|:
      zakończ się i zwróć `match`
```

Process z każdym otrzymanym symbolem rozszerza swój aktualny fingerprint i co 2^i otrzymanych literek sprawdza, czy odpowiada on odpowiadającemu fingerprintowi, który jest w tablicy $\phi[i]$. Jeśli jakiś fingerprint się nie będzie zgadzać, to sprawdza o ile możemy go przesunąć i kontynuuje pracę. Jeśli zmaczowany fingerprint był równy długości patternu, to zalazł match.

Możemy wyodrębnić 2 niezmienniki dla obu części programu (main loop oraz proces):

1. Nie ma więcej niż $3log(m)$ processów które działają jednocześnie
2. Każdy process, zakładając skończony input zwróci z dużym prawdopodobieństwem match, jeśli takowy istnieje w otrzymanych danych lub zakończy się.
3. Procesy przeanalizują wszystkie poprawne podciągi z pewnym wysokim prawdopodobieństwem.

Dowód niezmiennika 1:

Oznaczmy $l - process$ jako proces który zaczyna w checkpoincie l.

Pokażemy 2 rzeczy:

1. $l - process$ nie może wytworzyć $2l - process$ dopóki jego ojciec wiąż pracuje.
2. $l - process$ nie może mieć pra-wnuka.

D-d 1:

Zdefiniujmy A jako proces główny oraz B jako potomny od A równy $l - process$. Zauważmy, że A oraz B nie mogą istnieć dłużej niż $2l - 1$ otrzymanych zanków, ponieważ po $2l$ znakach A albo się zakończy, albo dotrze do checkpointa i zakończy swoje procesy potomne. Zauwazmy też, że aby proces B utworzył nowy $l2 - proces$, musi żyć przez co najmniej $2l$ znaków(bo musi znaleźć match od

długości 2l). W takim razie proces B nie może utworzyć $2l - process$ dopóki proces A pracuje.

D-d 2:

Mamy A i B jak wcześniej. Jako że B żyje najwyżej $2l - 1$ znaków, to może utworzyć syna $l - process\ C$. Ale aby C mogło utworzyć kolejny proces, to musi poczekać l znaków od czasu kiedy został stworzony. Sumarycznie $2l$ znaków od czasu kiedy powstało B. Oznacza to, że A juz nie możeistnieć.

Wniosek - nie może istnieć więcej niż 3 procesy na kazdą długość $logm$ patternu, zatem mamy maksymalnie $3logm$ procesów działających naraz.

Dowód niezmiennika 2: Wynika z algorytmu - jeśli znajdziemy odpowiadający fingerprint, to mamy match, jeśli nie to patrzymy o ile możemy przesunąć nasz hash(korzystając z okresu patternu), a jesli nie możemy to zabijamy dany proces. Zatem proces zawsze zawsze zakończy się zwracając match (niekoniecznie poprawny, bo mogą wystąpić kolizje) bądź zostanie przerwany. Oczywiście biorąc pod uwagę naturę hashowania - mamy niewielkie szanse na false positive.

Dowód niezmiennika 3 Jeśli znajdziemy częściowy(pełny) match, to wiemy, że w następnych $period[i]$ znakach nie będzie zadnego matcha. Zatem jeśli dany proces zacznie w dobrym checkpoincie (gdzie zaczyna sie szukany pattern), to kolejne fingerprinty będą równe aż do znalezienia szukanego matcha. Może wystąpić niewielka szansa na false negative po tym, jak po false positive-ie nie będą zgadzać się kolejne fingerprinty i przesuniemy sie $period[i]$ pozycji które mogły potencjalnie mieć w sobie match albo odpowiedni checkpoint który by go odnalazł.

No more False negatives

Aby naprawić False negatives, możemy nieco spowolnić program - zamiast przesuwać pattern o poprzedni okres w przypadku niezmachowania z danym fingerprintem, przesuwać go zawsze o 1. W ten sposób mamy gwarancję tego, że zawsze sprawdzimy wszystkie podciągi i nie przeoczymy żadnego matcha. Widać także że taka zmiana nie zmienia ogólnej złożoności programu, ponieważ w najgorszym przypadku oryginalny algorytm również miałby wszystkie okresy równe 1.

Nowy pseudokod:

```
def process():
  hash = None
  po otrzymaniu `input`:
    hash += fingerprint(input)  ## szybkie dodanie znaku do fingerprinta
    if |hash| == 2^i:
      if hash == ϕ[i]:
        signal `checkpoint` to main loop
      else:
```

```
        if hash[0..2^(i-1)] == periodϕ[i-1]:
            przesuń hash o 1(szybko)
        else:
            zakończ się
if |hash| == |P|:
    zakończ się i zwróć `match`
```