Penelaahan Algoritma Rabin-Karp dan Perbandingan Prosesnya dengan Algoritma Knut-Morris-Pratt

Nicolas Andres[1], Christopher[2], Hadi Saloko[3]

Laboratorium Ilmu dan Rekayasa Komputasi
Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail : if14109@students.if.itb.ac.id[1], if14117@students.if.itb.ac.id[2], if14157@students.if.itb.ac.id[3]

Abstrak

Pencarian *string* sudah familiar dalam kehidupan sehari-hari terlebih dalam bidang ilmu komputer. Kita lihat saja dalam berbagai kasus yang muncul contohnya proses pembuatan makalah. Penulis melakukan pencarian literatur yang memanfaatkan pencarian *string* (dalam *search engine* ataupun fasilitas *search* pada piranti lunak yang dipakai).Bagaimanapun familiarnya, tetapi metode yang digunakan dalam membandingkan dan mencari kemunculan *string* sama sekali berbeda dengan perbandingan dan pencarian *string* yang biasa kita pikirkan. Dalam pencarian *string*, komputer sedapat mungkin menggunakan algoritma yang mangkus dan sangkil, baik dilihat dari segi waktu pencarian (*equivalen* kompleksitasnya) ataupun pemakaian memori. Dalam berbagai literatur yang penulis dapatkan, terdapat berbagai macam algoritma pencocokan *string* (*string matching)* yang dipakai dalam pencarian *string*. Setiap algoritma pencocokan *string* memiliki kelebihan atau kekurangan masing-masing dalam berbagai hal, misalnya kompleksitas, waktu *prepocessing*, memori yang terpakai, dan lain sebagainya. Pada makalah ini akan dibahas dan dianalisis kelebihan dan kekurangan dua algoritma yaitu algoritma *Rabin-Karp* dan *Knut-Morris-Pratt*.

***Kata kunci**: pencocokan string, string matching, algoritma Rabin Karp, algoritma Knut-Morris-Pratt, kompleksita, pencarian string*

1. Pendahuluan

Pencarian *string* sudah familiar dalam kehidupan sehari-hari terlebih dalam bidang ilmu komputer. Kita lihat saja dalam berbagai kasus yang muncul contohnya proses pembuatan makalah. Penulis melakukan pencarian literatur yang memanfaatkan pencarian *string* (dalam *search engine* ataupun fasilitas *search* pada piranti lunak yang dipakai). Dalam pencarian *string,* komputer menggunakan algoritma tertentu sehingga pencarian menjadi mangkus dan sangkil. Setiap algoritma tersebut memliki kelebihan dan kekurangan masing-masing karena suatu algoritma tidak selalu lebih baik daripada algoritma yang lain. Dari berbagai algoritma yang ada tidak ada suatu algoritma pun yang dapat kita katakan sebagai algoritma yang paling "bagus" ataupun yang paling "jelek" karena semuanya itu bergantung pada kasus yang akan diselesaikan.

2. Algoritma Rabin Karp

Rabin Karp adalah algoritma pencarian *string* dengan waktu *preprocessing* $\Theta(m)$ dan memiliki waktu *processing* Θ ((n-m+1)m) pada kasus terburuk, tetapi pada kebanyakan kasus, memiliki waktu *processing* yang jauh lebih baik (kasus terburuk hampir tidak pernah terjadi).

2.1. *Preprocessing* (sebelum proses)

Pada *preprocessing*, *Rabin Karp* menggunakan teori bilangan dalam penghitungan *hash key*, antara lain menggunakan aturan Horner dan aturan modulus. Waktu *preprocessing* $\Theta(m)$, sebenarnya terjadi pada saat proses meng-*hash* m karakter dalam *pattern* yang ingin dicari (menggunakan aturan Horner).

Contoh:
Variabel temp adalah *corresponding number value* yang dimaksud dan dihitung dalam fungsi *toNumber*.
Pattern p adalah *pattern* yang ingin dicari.
untuk contoh ini, p = "1350"

Basis b adalah basis yang digunakan dalam fungsi *hash*, untuk contoh ini, b = 10

Fungsi *toNumber* adalah fungsi sembarang untuk mencari *corresponding decimal value* (atau tipe lainnya, misal *corresponding float value*). Dalam contoh ini, fungsi *toNumber* hanyalah *type casting* karena basis yang digunakan adalah basis 10.

Dari informasi di atas, proses perhitungan *hash key*-nya:

temp = toNumber (p)
temp = toNumber (“1350”)
temp = b * (135) + 0
temp = b * (b * (13) + 5) + 0
temp = b * (b * (b * (1) + 3) + 5) + 0
temp = 10 * (10 * (10 * (1) + 3) + 5) + 0

atau dengan kata lain :

temp = $b^3 * 1 + b^2 * 3 + b^1 * 5 + b^0 * 0$
temp = 1350

Variabel h adalah *hash key* dari suatu *string* tertentu sebagai keluaran akhir yang dicari dalam keseluruhan proses ini.

Variabel q adalah *operand denumerator* modulus, untuk contoh ini, q = 13.

Akan tetapi, secara intuitif kita sadar jika *hash key* adalah temp, maka ada kemungkinan tidak ada tipe data yang mampu menampung temp tersebut. Hal ini dikarenakan nilai variabel temp membesar dengan sangat cepat (berbanding eksponensial dengan panjang *pattern*). Oleh karena itu, digunakan aturan modulus untuk memperkecil rentang *hash key* dari variabel temp yang didapat dari proses di atas :

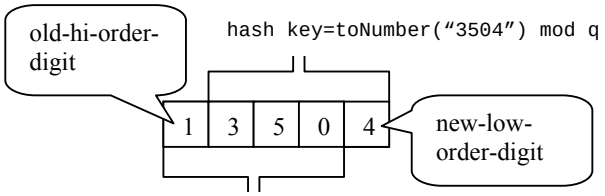
h = temp *mod* q
h = 1350 *mod* 13
h = 1

Dari contoh diatas terlihat :
Waktu *preprocessing* (penghitungan *hash key pattern* dengan panjang m) memiliki kompleksitas Θ(m) (sepanjang *pattern*).

Dengan menggunakan aturan horner, *corresponding number value* dari suatu *string* “3504” dapat dicari dengan mudah dari *corresponding number value* dari suatu *string* “1350”.

toNumber (“3504”)
=
(toNumber(“1350”) – 1 * b^{m-1}) * b + 4

b adalah basis yang digunakan
m adalah panjang *pattern* yang ingin dicari.
old-hi-order-digit bernilai 1
new-low-order-digit bernilai 4



hash key = toNumber(“1350”) mod q
hash key = fungsi (hash key lama ,old-hi-order-digit, new-low-order-digit)

Ini merupakan salah satu keunggulan algoritma Rabin-Karp. Baik dalam menghemat waktu pencarian *hash key* ataupun dalam penggunaan memori.

Sebenarnya dengan memakai aturan modulus, pembandingan antara satu *hash key* dengan *hash key* yang lainnya menjadi tidak sempurna. Hal ini dikarenakan :

hash_key1 ≡ hash_key2 (mod q)
tidak berart*i*
hash_key1 = hash_key2

Tetapi hal ini cukup karena yang dipakai adalah ketidaksamaan, bukan persamaan. Mengingat :

hash_key1 ≠ hash_key2 (mod q)
berart*i*
hash_key1 ≠ hash_key2

Catatan tambahan utuk pemilihan konstanta :
Biasanya basis b minimal sama dengan banyaknya jenis karakter yang ada dalam *field* pencarian. Misalnya untuk contoh diatas, *field* pencarian adalah sebuah file teks yang hanya berupa deretan angka bulat positif. Basis b dipilih 10 karena bayaknya jenis karakter yang mungkin muncul antara lain {‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’} berjumlah 10 jenis.

Biasanya variabel temp tersebut tidak perlu dicatat, melainkan langsung dipakai sebagai *operand numerator* modulus. Hal ini untuk menghemat variabel yang dipakai, sekaligus mengurangi kemungkinan *overflow* pada variabel temp.

Biasanya operand denumerator modulus-nya dipilih dengan aturan tertentu yaitu :

q Є bilangan prima yang cukup besar
dan
(b+1)*q tidak menyebabkan *overflow*

dimana b adalah basis yang digunakan dalam fungsi *hash* dan q adalah *operand* modulus yang ingin dipilih.
Alasan q harus dipilih bilangan prima yang cukup besar adalah untuk mengurangi kemungkinan dua buah *corresponding number value* yang sama. Misalnya :

(a mod c) = (b mod c)

Persamaan diatas akan lebih sulit terpenuhi jika c besar, dan itu yang kita inginkan.

2.2. *Matching* (pencocokan)

Pada *processing*, setiap m deret (kontigu) karakter pada *field* pencarian dicari *hash key* dengan cara yang sama dengan pencarian *hash key* untuk *pattern*. Perlu diingat dengan menggunakan aturan horner, pencarian untuk m deret karakter dapat dengan mudah dihitung dari m deret karakter predesesornya.

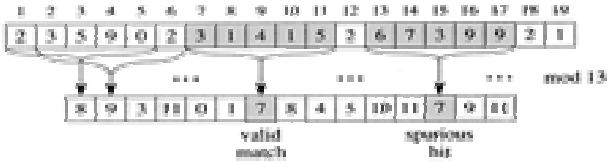
Sedangkan untuk proses pencocokannya, dipakai sebuah teorema yaitu :

> Sebuah *string* A akan identik dengan *string* B, jika (syarat perlu) *string* A memliki *hash key* yang sama dengan *hash key* yang dimiliki oleh *string* B.

Berdasarkan teorema di atas, dapat ditarik kesimpulan bahwa :

1. *String* yang sama *hash key*-nya, MUNGKIN sama *string*-nya. Oleh karena itu ,diperlukan pengecekan string lebih lanjut (biasanya cukup dipakai *naïve-search* karena kasus ini jarang terjadi jika dipilih *operand denumerator modulus* q yang cukup besar).
2. *String* yang tidak sama hash key-nya, TIDAK MUNGKIN sama *string*-nya. Oleh karena itu, tidak diperlukan pengecekan *string* lebih lanjut.

Jadi, pencocokan dilakukan dulu terhadap *hash key text field* terhadap *pattern.* Seandainya sama, barulah dicek kesamaan *string text field* terhadap *pattern.*



Jika algoritma Rabin-Karp ditulis secara keseluruhan dalam *pseudocode* :

```
procedure RABIN_KARP
(input T : array [1..max_const] of char , P :
array [1..max_const] of char , d : integer ,q
: integer)
{ T : text field, P : pattern, b : base, q :
operand denumerator modulus}
{procedure untuk mencetak kemunculan pattern
P poada text field T}
Deklarasi :
       n ,m : integer
{digunakan sebagai pemegang panjang string}
       p, h : integer
{digunakan sebagai pemegang hash key}
       i, s : integer
{digunakan sebagai counter}
       multiplier : integer
{digunakan sebagai pemegang multiplier
karakter terakhir dalam suatu deret}

Algoritma :
    n ← length [T]
    m ← length [P]
    multiplier ← b m-1 mod q
    p ← 0
    {preprocessing + pencarian hash key pertama
    dalam deret text field}
    for i ← 1 to m do
       p ←  (b * p + P[i]) mod q
       h ←  (b * h + T[i]) mod q
    endfor

    {matching}
    for s ←  0 to n-m do
       if ( p = h ) then
          if P[1..m] = T[s+1..s+m] then
            {pencocokan lebih lanjut}
            print "Match found with shift" + s
          endif

       if ( s < n-m ) then {penghitungan hash
       key deret sucessornya}
          h ← ( b * ( h – ( T[s+1] *
          multiplier) ) + T [s+m+1] ) mod q
       endif
    endif
  endfor
```

Sebagai tambahan, bisa ditambahkan suku tambahan b * q pada pengurangan (h – (T[s+1] * *multiplier*) untuk menjamin hasil pengurangan tetap positif [3].

Untuk jelasnya, baris terakhir *pseudocode* diganti menjadi :

h ← (b * (h + **(b* q)** – (T[s+1] * multiplier)) + T [s+m+1]) mod q

Dengan adanya penambahan, operasi modulus dijamin dapat bekerja semestinya (operasi modulus hanya menerima input berupa bilangan positif). Selain itu, baris diatas menjelaskan alasan pemilihan *operand denumerator* modulus q dimana :

`h + (b* q) < (b+1)*q`

dijamin tidak *overflow*

3. Algoritma Knut-Morris-Prat

Jika algoritma Knut-Morris-Pratt ditulis secara keseluruhan dalam *pseudocode* :

```
procedure KMPsearch
(input m, n : integer, P : array[1..m] of char,
T : array[1..n] of char, output idx : integer)
{T: text field, P:pattern, m: length of text
field, n: length of pattern, idx: index found}
{ procedure untuk mencari kemunculan pattern P
pada text field T, parameter output idx diisi
index found, atau -1 jika tidak diketemukan }
Deklarasi :
  i, j : integer
  ketemu : boolean
  b : array[1..m] of integer

procedure HitungPinggiran
(input m : integer, P : array[1..m] of
character, output b : array[1..m] of integer)
{ Menghitung nilai b[1..m] untuk pattern
P[1..m] }

Algoritma :
  HitungPinggiran(m, P, b)
  j←0
  i←1
  ketemu←false
```

```
  while (i ≤ n and not ketemu) do
    while ((j > 0) and (P[j+1]≠T[i])) do
      j←b[j]
    endwhile

  if P[j+1]=T[i] then
      j←j+1
    endif
    if j = m then
      ketemu←true
    else
      i←i+1
    endif
 endwhile
if ketemu then
   idx←i-m+1
{ catatan: jika indeks array dimulai dari
0, maka idx←i-m }
 else
   idx← -1
 endif
```

Dimana *procedure* HitungPinggiran jika ditulis dalam *pseudocode* :

```
procedure HitungPinggiran
(input m : integer, P : array[1..m] of
char, output b : array[1..m] of integer)
{ Menghitung nilai b[1..m] untuk pattern
P[1..m] }
Deklarasi :
  k,q : integer

Algoritma:
  b[1]←0
  q←2
  k←0
  for q←2 to m do
    while ((k > 0) and (P[q] ≠ P[k+1])) do
      k←b[k]
    endwhile
    if P[q]=P[k+1] then
      k←k+1
    endif
    b[q]=k
  endfor
```

4. Analisis Kelebihan dan Kekurangan Dua Algoritma yang Dikaji

Dari penelaahan algoritma Rabin-Karp untuk *string matching*, terlihat perbedaan yang mendasar jika dibandingkan dengan algoritma Knut-Morris-Pratt adalah :

1. Rabin-Karp menelusuri karakter satu persatu pada deret karakter (kontigu), tetapi proses perbandingannya (penghitungan *hash key* nya) relatif mudah (dengan aturan Horner *hash key* selanjutnya dapat dihitung dari *hash key* sebelumnya), sedangkan Knut-Morris-Pratt "meloncati" beberapa karakter pada deret karakter setelah memproses pinggiran (awalan dan akhiran) yang relatif lebih sulit (karena hampir tidak berhubungan dengan pinggiran sebelumnya)
2. Rabin-Karp tidak terlalu mangkus dalam kompleksitas jika dibandingkan dengan Knut-Morris-Pratt, yang mengimplikasikan waktu *matching string* (pencocokan *string*) yang lebih lama.
3. Rabin-Karp hampir tidak membutuhkan memori tambahan seperti yang dibutuhkan Knut-Morris-Pratt untuk menyimpan pinggiran (awalan dan akhiran).

5. Kesimpulan

Berdasarkan penelaahan diatas dapat disimpulkan bahwa :
Knut-Morris-Pratt unggul dalam kasus pencarian *string* pada umumnya, tetapi Rabin-Karp lebih unggul dalam :

1. Kasus pencarian *string* dengan *pattern* yang panjang.
2. Kasus pencarian *string* dengan *multi pattern*.
3. Kasus pencarian *string* dengan *pattern* yang banyak diulang secara tidak sempurna pada *text field*. Terlebih jika jenis karakter yang dipakainya relatif sedikit.
   Misal :
   pattern = alalalali
   text field = alalalalalalalalalalalalalalal…..
4. Kasus pencarian *string* dengan *pattern* dan *text field* dengan panjang relatif sama, atau sama. Jadi bukan pencarian *string*, melainkan hanya pencocokan *string*.
   Misal :
   Pattern = abcdefghijklmnopqrstuvwxyza…z
   text field = abcdefghijklmnopqrstuvwxya…z
5. Kasus pencarian *string* dengan *text field* yang tidak sama karakter pertamanya dengan *pattern*.
   Misal :
   pattern = abbbb
   text field = bbbbbbbbbbbbbbbbbbbbbbbb…

Daftar Pustaka

[1] Munir, Rinaldi.2006. *"Strategi Algoritmik"*.Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.

[2] Lionov.Skom, "Bahan Kuliah Algoritma Lanjut", Universitas Katolik Parahyangan

[3] Sedgewick, Robert. 1983. *"Algorithms"*.Addison Wesley.

[4] Levitin. 2003."*Intoduction to the Design and Analysis of Algorithms*". Addison Wesley.