PENGKAJIAN DAN ANALISIS TIGA ALGORITMA EFISIEN *RABIN-KARP, KNUTH-MORRIS-PRATT,* DAN *BOYER-MOORE* DALAM PENCARIAN POLA DALAM SUATU TEKS

Alsasian Atmopawiro – NIM : 13505074
Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if15074@students.if.itb.ac.id

Abstrak

Makalah ini membahas tentang pengkajian dan analisis tiga algoritma dalam pencarian pola dalam suatu teks atau *string-matching*. Ketiga algoritma itu adalah algoritma *Rabin-Karp*, algoritma *Knuth-Morris-Pratt*, dan algoritma *Boyer-Moore*. Pengkajian yang dilakukan mengenai kompleksitas waktu dan teknik yang digunakan dalam membandingkan suatu karakter dalam prosesnya. Kita bisa melihat pengaruh dari *preprocessing* yang dilakukan oleh beberapa algoritma.

String-matching atau pencocokan-kata adalah subjek yang penting dalam kaitannya dengan *text-processing*. Algoritma pencocokan-kata adalah komponen dasar yang dipakai dalam implementasi dari *software* yang berjalan dibawah sistem operasi. Walaupun data disimpan dalam cara yang berbeda, teks menjaga bentuk utama untuk menukar informasi. *String-matching* fokus pada pencarian satu, atau lebih umum, semua kehadiran kata (lebih umum disebut *pattern*) dalam sebuah teks. Metoda untuk melakukan perbandingan ada empat macam; dari kiri ke kanan yang merupakan cara standar; dari kanan ke kiri; perbandingan spesifik; perbandingan acak atau tidak-relevan.

Masalah dalam algoritma pencarian adalah mencari kata yang mempunyai panjang *m*, yang disebut *pattern*, dalam sebuah teks dengan panjang *n*. Penyelesaiannya mempunyai bermacam-macam cara yang mempunyai kelebihan dan kekurangan masing-masing misalnya dengan *hashing* pada algoritma *Rabin-Karp*, atau penggunaan tabel dari hasil *preprocessing* untuk algoritma *Knuth-Morris-Pratt* dan *Boyer-Moore*.

Kata kunci: *String-matching, pattern, text, hashing, failure function, Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore*, substring, kompleksitas.

1. Pendahuluan

String-matching atau pencocokan-kata adalah subjek yang penting dalam kaitannya dengan *text-processing*. Algoritma pencocokan-kata adalah komponen dasar yang dipakai dalam implementasi dari *software* yang berjalan dibawah sistem operasi. Bahkan, algoritma membuat metoda pemrograman menjadi penting sebagai paradigma bidang ilmu komputer lain. Akhirnya, algoritma ini juga memegang peranan penting dalam ilmu komputer teoritik dengan menyediakan masalah yang menantang.

Walaupun data disimpan dalam cara yang berbeda, teks menjaga bentuk utama untuk menukar informasi. Dapat kita lihat secara jelas dalam literatur atau pembicaraan dimana data dibentuk dari kamus. Data tersebut juga diterapkan sama pada ilmu komputer dimana data yang besar disimpan dalam bentuk file linear. Ada juga contoh lain, contohnya dalam biologi molekuler karena molekul biologi dapat kita anggap sebagai barisan nukleotida atau asam amino. Terlebih lagi, jumlah dari data yang tersedia di bidang itu meningkat duakali lipat tiap delapanbelas bulan. Inilah mengapa alasan algoritma harus efisien walaupun kecepatan komputer saat ini meningkat terus-menerus.

String-matching fokus pada pencarian satu, atau lebih umum, semua kehadiran sebuah kata (lebih umum disebut *pattern*) dalam sebuah teks. Semua algoritma yang akan dibahas mengeluarkan semua kehadiran pola dalam teks. Pola dinotasikan sebagai `x = x[0..m-1]`; *m* adalah panjangnya. Teks dinotasikan sebagai `y = y[0..n-1]`; *n* adalah panjangnya. Kedua string dibentuk dari set karakter yang disebut alphabet dinotasikan Σ dengan ukuran σ.

Aplikasi memberikan solusi tergantung untaian kata yang diberikan terlebih dahulu, teks atau pola. Algoritma yang berbasis pada penggunaan otomata atau properti kombinatorial dari untaian biasanya diimplementasikan untuk memroses dahulu pola dan menyelesaikan jenis problem pertama. Notasi indeks yang direalisasikan oleh pohon atau otomata digunakan untuk menyelesaikan jenis problem kedua.

Algoritma pencocokan kata yang akan dibahas akan mengikuti aturan berikut. Algoritma membaca teks dengan bantuan jendela yang ukurannya *m*. Mereka melakukan *align* kiri jendela dan teks, lalu membandingkan karakter pada jendela dengan karakter pada pola—disebut percobaan—dan setelah semua pencocokan pola atau setelah kesalahan, algoritma ini melakukan pergeseran jendela ke kanan. Eksekusi ini akan mengulang prosedur yang sama hingga mencapai jendela paling kanan. Mekanisme ini biasa disebut *sliding window mechanism*. Kita mengasosiasikan setiap percobaan dengan posisi *j* dalam teks ketika jendela diposisikan pada $y[j..j+m-1]$.

Algoritma *brute force* melokasikan semua kehadiran x dalam y dalam waktu O(*mn*). Peningkatan besar dari metoda ini dapat diklasifikasikan bedasarkan urutan dilakukannya perbandingan antara pola dan teks pada setiap percobaan. Ada empat klasifikasi: Cara paling natural untuk melakukan perbandingan adalah dari kiri ke kanan, yang sesuai dengan arah baca; Melakukan perbandingan dari kanan ke kiri yang biasanya mempunyai algoritma terbaik pada prakteknya; teori terbaik tercapai ketika perbandingan selesai dalam urutan yang spesifik; yang terakhir ada algoritma yang urutan perbandingannya tidak relevan (seperti *brute force*).

1.1. Dari kiri ke kanan

Metoda *hash* menyediakan metoda simpel yang menghindari nilai kuadratik dari perbandingan karakter, dan berjalan dalam waktu yang linier dibawah asumsi probabilitas rasional. Diperkenalkan oleh Harrison dan sepenuhnya dianalisis oleh *Karp dan Rabin*.

Asumsi bahwa panjang pola tidak lebih panjang daripada ukuran *memory-word* dalam suatu mesin, algoritma *Shift-Or* adalah algoritma efisien untuk menyelesaikan problem *string-matching* dan dapat beradaptasi dengan mudah untuk range yang cukup lebar untuk problem ini. Algoritma *Knuth-Morris-Pratt* adalah algoritma pencocokan pertama yang mempunyai waktu linier. Ditemukan oleh Morris dan Pratt, kemudian ditingkatkan performanya oleh Knuth dan keduanya. Perilaku pencariannya seperti proses rekognisi oleh otomasi dan sebuah karakter dalam teks dibandingkan pada karakter pada pola tidak lebih dari $\log_{\Phi}(m + 1)$. Hancart membuktikan bahwa delay ini dari algoritma terkait yang ditemukn oleh Simon melakukan perbandingan kurang dari $1 + \log_2 m$ per karakter. Ketiga algoritma tadi memperlihatkan paling banyak $2n – 1$ perbandingan karakter teks pada kasus terburuknya.

Pencarian dengan *Deterministic Finite Automaton* memperlihatkan inspeksi *n* karakter teks namun membutuhkan ekstra spasi dalam O(mσ). Algoritma pencocokan *Forward Dawg* memperlihatkan jumlah inspeksi yang sama menggunakan otomasi suffix dari pola.

Algoritma *Apostolico-Crochemore* adalah algoritma simpel yang melakukan perbandingan 2/3 *n* katakter teks pada waktu terburuk.

1.2. Dari kanan ke kiri

Algoritma Boyer-Moore dikenal sebagai algoritma yang paling efisien dalam berbagai aplikasi. Versi yang disederhanakan dari algoritma ini sering diimplementasikan dalam teks editor untuk perintah “search“ dan „substitute“. Cole membuktikan bahwa nilai maksimum dari perbandingan karakter adalah 3*n* setelah terjadinya proses awal untuk pola non-periodik. Namun algoritma ini mempunyai kasus terburuk kuadrat untuk pola periodik.

Beberapa varian dari algoritma Boyer-Moore menghindari kelakuan kuadrat dari waktu kompleksitasnya. Solusi yang paling efisien dari perbandingan karakter didesain oleh Apostolico dan Giancarlo, Crohemore el alii (*Turbo-BM*) dan Colussi (*Reverse Colussi*). Hasil empirik memperlihatkan bahwa variasi dari algoritma Boyer-Moore dan algoritma berbasis suffix otomasi atau suffix oracle oleh Chochemore el alii (*Backward Oracle Matching*) adalah yang paling efisien dalam prakteknya.

Algoritma Zhu-Takaoka dan Berry-Ravindan adalah varian dari algoritma Boyer-Moore yang membutuhkan ekstra tempat dalam $O(\sigma_2)$.

2. Algoritma pencarian kata

Masalah dalam algoritma pencarian adalah mencari kata yang mempunyai panjang *m*, yang disebut *pattern*, dalam sebuah teks dengan panjang *n*. Contoh sederhananya, mencari kata "par" dalam sebuah teks "saya lapar sekali." Algoritma simpel untuk keadaan ini adalah dengan melihat semua kemungkinan kata pada semua posisi yang mungkin.:

```
 1  function NaiveSearch(string  s[1..n],
string sub[1..m])
 2     for i from 1 to n
 3         for j from 1 to m
 4             if s[i+j-1] ≠ sub[j]
 5                 jump to next iteration
of outer loop
 6         return i
 7     return not found
```

Algoritma ini mungkin bekerja sangat baik untuk beberapa kasus, namun dalam kasus lain dapat menyebabkan waktu menjadi sangat lama seperti dalam contoh mencari kata dengan 10000 "a" diikuti oleh "b" dalam sebuah teks dengan 10 juta "a" diikuti oleh "b" yang mencapai kasus terburuk $\Theta(mn)$.

Algoritma *Knut-Morris-Pratt* mereduksi kasus terakhir mengunakan perhitungan awal untuk memeriksa setiap karakter hanya sekali. Algoritma *Boyer-Moore* tidak hanya melewati 1 karakter tapi sebanyak mungkin karakter agar pencarian berhasil, secara efektif mengurangi waktu untuk mengiterasi loop luar, sehingga karkter yang diperiksa dapat sekecil *n/m* dalam kasus terbaiknya. Algoritma Rabin-Karp terfokus pada mempercepat 3-6 baris.

2.1. Algoritma *Rabin-Karp*

Algorima *Rabin-Karp* menggunakan metoda hash dalam mencari suatu kata. Algoritma ini dibuat oleh Michael O. Rabin dan Richard M. Karp. Teori ini jarang digunakan untuk mencari kata tunggal, namun teori ini cukup penting dan sangat efektif bila digunakan untuk mencari lebih dari satu kata. Kasus terbaik dari algoritma ini bisa mencapai O(n), dimana *n* adalah panjang teks. Sayangnya kasus terburuk dari algoritma ini adalah $O(nm)$, dimana *m* adalah panjang kata yang menjadi sebab mengapa jarang digunakan. Keuntungan tersendiri dari algoritma ini adalah dapat mencari *k* kata dalam waktu rata-rata $O(n)$, yang tidak bergantung pada ukuran *k*.

2.1.1. Penggunaan *hashing* untuk pencarian pergeseran *substring*

Algoritma Rabin-Karp ini tidak melakukan pergeseran yang rumit untuk menyelesaikan masalah, algoritma ini mempercepat pengecekan kata pada suatu teks dengan menggunakan fungsi *hash.* Fungsi *hash* adalah fungsi yang mengkonversikan suatu kata menjadi nilai yang disebut nilai *hash* (*hashvalue*). Contohnya seperti hash("hello")=5. Rabin-Karp menggunakan fakta bahwa jika suatu kata adalah sama maka nilai *hash*-nya juga sama. Jadi kita dapat melihat bahwa yang perlu kita lakukan adalah mencari nilai *hash* dari kata yang kita punya kemudian mencari kata dalam teks yang mempunyai nilai *hash* yang sama.

Namun ada dua masalah dengan metoda tersebut. Pertama, karena terdapat banyak sekali kata dalam kalimat yang berbeda, untuk menjaga agar nilai *hash* kecil kita harus memberi beberapa kata dengan nilai *hash* yang sama. Yang berarti jika suatu nilai hash sama maka belum tentu sebuah kata itu sama; kita harus memastikan kembali bahwa keduanya benar-benar kata yang sama, yang membutuhkan waktu yang sama untuk panjang berapapun. Untungnya, fungsi *hash* yang baik menjanjikan bahwa selama inputnya masuk akal, kejadian tadi tidak akan terjadi terlalu sering, sehingga waktu kompleksitas tetap terjaga.

Algoritma Rabin-Karp sebagai berikut:

```
1  function RabinKarp
   (string s[1..n],   string sub[1..m])
2     hsub := hash(sub[1..m])
3     hs := hash(s[1..m])
4     for i from 1 to n
5         if hs = hsub
6             if s[i..i+m-1] = sub
7                 return i
8         hs := hash(s[i+1..i+m])
9     return not found
```

Baris 2, 3, 6, dan 8 masing-masing membutuhkan waktu Ω(m). Namun baris 2 dan 3 hanya dieksekusi sekali dan baris 6 hanya dieksekusi bila nilai *hash*-nya sama, yang kemungkinan akan terjadi lebih dari satu kali. Baris 5 dieksekusi *n* kali, dengan waktu yang konstan. Jadi permasalahan ada di baris 8.

Jika kita secara naif mengkomputasi ulang nilai *hash* untuk kata $s[i + 1 .. i + m]$, akan membutuhkan waktu dan karena dieksekusi setiap loop maka algoritma ini membutuhkan

waktu $\Omega(mn)$, sama dengan algoritma paling naif. Trik untuk mengatasi ini dengan cara melihat bahwa variabel *hs* telah mempunyai nilai *hash* dari $s[i + 1 .. i + m]$. Jika kita dapat menggunakan nilai tersebut untuk komputasi nilai *hash* setelahnya dengan waktu yang konstan, maka masalah kita terselesaikan.

Caranya dengan menggunakan *rolling hash*. *Rolling hash* adalah fungsi hash yang didesain khusus agar dapat melakukan operasi tadi.

Satu contoh sederhana adalah dengan menambah nilai untuk setiap karakter dalam kata. Lalu kita dapat menggunakan formula ini untuk menghitung nilai *hash* selanjutnya dengan waktu yang konstan:

```
s[i+1..i+m] = s[i..i+m-1] - s[i] + s[i+m]
```

Fungsi ini dapat digunakan, namun akan berakibat pada lebih seringnya eksekusi baris 6 dibandingkan fungsi *rolling hash* yang lebih rumit seperti yang akan kita bahas selanjutnya.

Perhatikan bahwa kita dapat memiliki fungsi *hash* yang buruk seperti fungsi konstan, baris 6 akan dilakukan *n* kali setiap loop. Karena membutuhkan waktu $\Omega(m)$, seluruh algoritma membutuhkan waktu terburuk $\Omega(mn)$.

2.1.2. Penggunaan fungsi *hash*

Kunci dari performa Rabin-Karp adalah komputasi yang efisien dari nilai *hash* dari *substring* selanjutnya dari teks. Fungsi *rolling hash* yang popular dan efektif memperlakukan tiap *substring* sebagai angka dalam beberapa basis, basis biasanya bilangan prima yang besar. Contohnya jika *substring* "hi" dan basisnya 101, nilai *hash*-nya menjadi $104 \times 101^1 + 105 \times 101^0 = 10609$ (nilai ASCII dari 'h' adalah 104 dan 'i' adalah 105).

Secara teknik, algoritma ini hanya mirip dengan angka benar (*true number*) dalam representasi sistem non-desimal, karena seperti dalam contoh kita mempunyai angka basis yang lebih rendah dari digit. Keuntungan esensial yang didapat dari representasi tersebut adalah kemungkinan mengkomputasi nilai *hash* dari *substring* selanjutnya dari sebelumnya dengan cara menghitung hanya operasi angka konstan, tidak tergantung dari panjang *substring*.

Misalkan:

Hash("abr") = n
Hash("bra") = $(n - (97 \times 101^2)) \times 101 + 97$

Contohnya jika kita mempunyai kata "abracadabra" dan kita mencari pola dengan panjang 3, kita dapat mengkomputasi *hash* dari "bra" dari *hash* "abr" (kata sebelumnya) dengan mengurangi angka yang ditambah untuk karakter 'a' dari "abr", contohnya 97×101^2 (97 adalah ASCII untuk 'a' dan 101 adalah basis yang kita gunakan), kalikan dengan basis dan tambahkan nilainya untuk karakter 'a' pada akhir "bra". Untuk lebih jelasnya lihat contoh diatas. Jika *substrings* dalam pertanyaan cukup panjang, algoritma ini mempunyai kelebihan dalam menghemat waktu dibanding skema *hash* yang lain.

Secara teori, ada algoritma lain yang dapat menyediakan komputasi ulang yang nyaman, contoh mengalikan nilai ASCII bersaman dari semua karakter sehingga menggeser *substring* hanya dilakukan dengan cara membagi karakter pertama dan mengalikan karakter kedua. Namun batasannya ada pada ukuran tipe data integer dan kebutuhan dalam penggunaan aritmaika modulo untuk mengecilkan skala ukuran nilai *hash*, sementara itu, fungsi *hash* yang naif yang tidak membentuk angka besar cepat, seperti menambah nilai ASCII, dapat menyebabkan *hash collision* dan jelasnya memperlambat algoritma. Maka fungsi *hash* yang diawal dijelaskan lebih dipilih dalam algoritma Rabin-Karp.

2.1.3. Pencarian pola jamak

Rabin-Karp lebih lemah ketimbang algoritma Knuth-Morris-Pratt, Boyer-Moore dan algoritma pencari kata tunggal karena lambatnya kasus terburuk algoritmanya. Namun, Rabin-Karp adalah algoritma yang dipilih untuk mencari kata/pola jamak.

Maka, jika kita ingin mencari *k* kata dengan jumlah yang besar, kita dapat menciptakan varian simpel dari Rabin-Karp yang menggunakan *bloom filter* atau struktur data *set* untuk mengecek apakah *hash* yang diberikan oleh kata merupakan bagian dari set nilai *hash* yang kita sedang cari.

Berikut adalah variasi algoritma *Rabin-Karp*:

```
function RabinKarpSet
```

```
  (string s[1..n], set of string subs, m)
{
  set hsubs := emptySet
  for each sub in subs
    insert hash(sub[1..m]) into hsubs
  hs := hash(s[1..m])
  for i from 1 to n
    if hs = hsubs
      if s[i..i+m-1]=a substr \w hash hs
        return i
    hs := hash(s[i+1..i+m])
  return not found
}
```

Disini kita mengasumsikan bahwa semua *substring* mempunyai panjang kata tetap *m*, namun asumsi ini dapat kita tiadakan. Kita hanya membandingkan nilai *hash* tertentu, terhadap nilai *hash* semua substring secara terus-menerus menggunakan *quick lookup* dalam set, dan verifikasi kesamaan yang kita temukan terhadap semua *substring* dengan nilai *hash* tersebut.

Algoritma lain dapat mencari sebuah kata dalam waktu O(*n*), dan dapat digunakan untuk mencari *k* kata dalam waktu O(*n k*). Sebaliknya, varian Rabin-Karp di atas dapat digunakan untuk mencari semua *k* kata dalam ekspektasi waktu O(*n*+*k*), karena tabel *hash* cek apakah sebuah *substring hash* sama dengan sebuah kata dalam waktu O(1).

2.2. Algoritma *Knuth-Morris-Pratt*

Algoritma *Knuth-Morris-Pratt* (KMP) mencari kehadiran sebuah kata *w* dalam teks *s* dengan melakukan observasi awal (*preprocessing*) yaitu ketika muncul ketidaksamaan kata ini mempunyai informasi mengenai kapan kesamaan selanjutnya bermula, dengan cara mengece ulang kata sebelumnya.

Algoritma ini dibuat oleh Knuth dan Pratt dan sendiri oleh J. H. Morris pada tahun 1977, namun ketiganya mem-*publish*-nya bersamaan.

2.2.1. Contoh algoritma KMP

Kita akan melihat alur algoritma KMP agar lebih mudah diresapi. Kita akan mempunyai dua integer *m* dan *i*. Variabel *m* menunjukan indeks karakter pada *s* dan variabel *i* menunjukan karakter pada *w*. Berikut ini adalah gambaran pada awal eksekusi:

```
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 0123456
```

Algoritma ini dimulai (setelah *preprocessing*, akan dibahas dibawah) dengan membandingkan karakter pada kata *w* secara paralel pada karakter pada *s*, bergeser ke indeks selanjutnya (*m*=*m*+1) jika sama. Namun ketika mencapai langkah ke-4 (*m*=3), kita mendapat *s*[*m*] adalah spasi dan yang menyatakan karakter tidak sama dengan *w*[*m*]='D'. Kita tidak akan kembali dari *s*[0], kita perhatikan bahwa tidak ada 'A' pada posisi 0 sampai 3 di *w* kecuali pada saat 0; karena pada saat *preprocessing* kita sudah mengecek semua posisi karakter, kita tau bahwa tidak akan ada peluang untuk menemukan awal kata *w* ('A') yang cocok dari awal. Oleh karena itu kita langsung menuju ke karakter selanjutnya (logis: mengapa kita harus capai-capai mundur kalau tidak ada?), dan mulai mencocokan dengan kata *w* kembali sehingga *m*=4 dan *i*=0.

```
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456
```

Pada langkah selanjutnya kita akan mendapatkan sebuah kata yang hampir lengkap yaitu "ABCDAB" saat *w*[6] (ketika *s*[10]). Karena belum menemukan kata yang tepat kita akan mencari lagi namun tidak dari awal. Ternyata pada sebelumnya kita menemukan bagian awal kata *w*. karena terdapat suatu awal yang sesuai yaitu "AB" oleh karena itu kita cukup me-*reset* *m*=8 dan *i*=2 dan melanjutkan pencocokan.

```
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:         ABCDABD
i:         0123456
```

Pengecekan selanjutnya akan langsung gagal karena spasi tidak sama dengan 'C', namun, karena tidak ada awal yang sama pada *substring* teks ('A'), maka seperti pada eksekusi pertama, kita mulai pecarian pada karakter selanjutnya dari *s*: *m*=11 dan reset *i*=0.

```
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:            ABCDABD
i:            0123456
```

Sekali lagi kita menemukan kata yang hampir cocok "ABCDAB" namun karakter selanjutnya 'C', tidak sama. Dengan alasan sebelumnya kita

buat m=15, dan i=2 karena ada kata "AB" pada akhir pencarian.

```
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:                ABCDABD
i:                0123456
```

Pada akhirnya kita dapat menemukan kata yang sepadan pada karakter awal $s[15]$.

2.2.2. Deskripsi dan *pseudocode*

Contoh di atas menjelaskan seluruh kemungkinan dari algoritma KMP. Untuk saat ini, kita anggap terdapat suatu tabel T atau disebut *partial table* yang mempengaruhi algoritma sudah terbuat, tabel akan dijelaskan di bawah, yang menunjukan dimana algoritma akan melakukan suatu instruksi pencocokan setelah terjadi kegagalan. Isi dari T dibuat sedemikian rupa sehingga jika kita memulai pencocokan dari $s[m]$ yang gagal ketika membandingkan pada karakter ke $s[m + i]$ pada $w[i]$, maka pencocokan selanjutnya akan dimulai pada indeks $m + i - T[i]$ pada s ($T[i]$ adalah jumlah langkah mundur yang harus dilakukan), yang mempunyai dua implikasi. Pertama, $T[0] = -1$ yang berarti bahwa jika pada saat $w[0]$ tidak cocok, maka kita tidak bisa kembali ke karakter sebelumnya sehingga kita harus maju ke karakter selanjutnya; dan kedua walaupun kemungkinan karakter sama selanjutnya dimulai pada $m + i - T[i]$ seperti contoh di atas, kita tidak perlu mengecek karakter T[i] setelahnya, jadi kita melanjutkan pencarian mulai dari $w[T[i]]$. Dibawah ini contoh *pseudocode* untuk algoritma pencarian Knuth-Morris-Pratt:

```
algorithm kmp_search:
    input:
        an array of characters, S (the
text to be searched)
        an array of characters, W (the
word sought)
    output:
        an integer (the zero-based
position in S at which W is found)

    define variables:
        an integer, m ← 0 (the beginning
of the current match in S)
        an integer, i ← 0 (the position
of the current character in W)
        an array of integers, T (the
table, computed elsewhere)

    while m + i is less than the length
of S, do:
        if W[i] = S[m + i], let i ← i + 1
            if i equals the length of W,
return m
        otherwise, let m ← m + i - T[i],
and if i > 0, let i ← T[i]

    (if we reach here, we have searched
all of S unsuccessfully)
  return the length of S
```

2.2.3. Efisiensi algoritma pencarian KMP

Dengan memperhitungkan kehadiran tabel T, porsi pencarian dengan algoritma Knuth-Morris-Pratt mempunyai kompleksitas $O(k)$, di mana k adalah panjang dari S. Kita anggap semua ekseskusi yang dilakukan berada dalam kalang while, kita akan menghitung jumlah iterasi yang terjadi dalam loop ini. Untuk melakukannya kita harus memperhatikan nilai dalam tabel T. T dibentuk sedemikian rupa sehingga ketika pencocokan dimulai pada $s[m]$ gagal ketika mencocokan $s[m + i]$ pada $w[i]$, pencocokan selanjutnya harus dimulai pada karakter $s[m + (i - T[i])]$. Karena indeks tidak boleh lebih tinggi daripada m, sehingga $T[i] < i$.

Melihat kejadian tadi, kita akan melihat bahwa loop melakukan eksekusi paling banyak $2k$ kali. Untuk setiap iterasi, loop ini mengeksekusi salah satu dari dua cabang dalam loop:

Cabang pertama menambah i dengan satu dan tidak mengubah m, sehingga indeks $m + i$ yang merupakan karakter yang sedang diproses bertambah. Cabang kedua menambahkan $i - T[i]$ pada m, dan seperti yang kita lihat nilainya selalu positif. Lalu lokasi m pada awal kemungkinan pencocokan bertambah.

Jika diperhatikan sekarang loop akan berhenti jika $m + i = k$, maka setiap cabang dari loop dapat mencapai maksimum eksekusi k kali, karena bertambahnya m atau $m + i$, dan $m \leq m + i$; jika $m = k$, maka tentu saja $m + i \geq k$, sehingga karena bertambah satu-satu maka kita pasti mempunyai $m + i = k$ pada suatu titik sebelumnya, dan oleh karena itu kedua cara dapat dikerjakan.

Loop yang dieksekusi paling banyak $2k$ kali menunjukan bahwa kompleksitas waktu pencarian algoritma ini adalah O(k).

2.2.4. *Partial table* (*failure function*)

Kita akan membahas mengenai *partial table* yang kita gunakan dalam algoritma KMP yang kita jelaskan di atas. Tujuan dari tabel ini adalah

untuk mencegah algoritma melakukan pencocokan pada suatu karakter pada *s* lebih dari satu kali. Hal yang memungkinkan ini terjadi adalah kita tahu dimana pencocokan selanjutnya akan berlangsung, setelah kita mengecek beberapa bagian pada teks utama terhadap bagian dari kata (karakter per karakter). Dengan kata lain kita harus mengecek ulang kata itu sendiri dan membuat list kemungkinan posisi (pada *preprocessing*) yang melewati karakter-karakter yang tidak mungkin sama, namun tentu saja tidak melewati karakter yang sama.

Kita ingin dapat melihat kembali, untuk setiap posisi di *w,* panjang dari kemungkinan inisialisasi segmen terpanjang dari w yang menuju (namun tidak termasuk) posisi tersebut. Hal ini lebih baik dari pada kembali ke posisi *w*[0] yang sudah pasti gagal; sepanjang inilah kita harus mundur kebelakang (*backtrack*) untuk mencocokan *substring* berikutnya. Nilai tersebut disimpan dalam `T[i]` yang merupakan panjang kemungkinan inisialisasi segmen terpanjang dari *w* yang juga merupakan segmen dari *substring* yang berakhir pada *w*[*i* - 1]. Kita menggunakan konvensi dimana *string* kosong mempunyai panjang 0. Karena *mismatch* pada awal pola adalah kasus khusus (tidak ada kemungkinan mundur), kita set *T*[0] = -1, seperti yang didiskusikan diatas.

2.2.5. Deskripsi dan *pseudocode* pembuatan *partial table*

Contoh diatas mengilustrasikan teknik umum untuk membuat tabel. Prinsipnya adalah dari pencarian keseluruhan: sebagian besar pekerjaan sudah dikerjakan saat menuju ke karakter saat ini, jadi hanya sedikit perlakuan yang perlu dilakukan. Dibutuhkan sedikit inisialisasi di awal agar berjalan sesuai rencana.

```
/* algorithm kmp_table */
Void kmp_table(int T[]) {
    /*kamus*/
    i=2; /*posisi saat ini*/
    j=0; /*indeks berbasis nol di w*/

    /*algoritma*/
    /*nilai awal diset sesuai algo*/
    t[0] = -1;
    t[1] = 0;
    while (i<strlen(w))
    {
        if (w[i-1]==w[j])
        /*kasus pertama: untaian kata
berlanjut*/
        {
            t[i] = j + 1;
            i++;
            j++;
        }
        else if (j > 0)
        /*kasus kedua: tidak berlanjut
namun dapat kembali lagi*/
        {
            j = t[j];
        }
        else /*kasus ketiga: benar-benar
tidak ada kandidat. I.S. j = 0*/
        {
            t[i] = 0;
            i++;
        }
    }
}
```

2.2.6. Contoh pembuatan *partial table*

Mari kita lihat contoh di atas kembali dari w = "ABCDABD". Kita set dahulu *T*[0] = -1. Untuk mencari *T*[1], kita harus menemukan posisi awal yang merupakan "A" yang juga merupakan awal kata *w*. Namun ternyata bukan "A", maka kita set *T*[1]=0. sama dengan selanjutnya *T*[2]=0.

Berikut langkah-langkah selanjutnya berdasarkan algoritma:

Inisialisasi awal:

i	0	1	2	3	4	5	6
W[i]	A	b	c	d	a	b	d
T[i]	**-1**	**0**	-	-	-	-	-

i=2, j=0, kasus tiga

i	0	1	2	3	4	5	6
W[i]	a	b	c	d	a	b	d
T[i]	-1	0	**0**	-	-	-	-

i=3, j=0, kasus tiga

i	0	1	2	3	4	5	6
W[i]	a	b	c	d	a	b	d
T[i]	-1	0	0	**0**	-	-	-

i=4, j=0, kasus tiga

i	0	1	2	3	4	5	6
W[i]	a	b	c	d	a	b	d
T[i]	-1	0	0	0	**0**	-	-

i=5, j=0, kasus satu

i	0	1	2	3	4	5	6
W[i]	a	b	c	d	a	b	d
T[i]	-1	0	0	0	0	**1**	-

i=6, j=1, kasus satu

i	0	1	2	3	4	5	6
W[i]	a	b	c	d	a	b	d
T[i]	-1	0	0	0	0	1	**2**

2.2.7. Efisiensi pembuatan *partial table*

Kompleksitas algoritma tabel adalah O(*n*), dimana *n* adalah panjang kata *w*. Kecuali beberapa inisialisasi, semua pekerjaan kita diselesaikan dalam kalang loop while, kita dapat melihat loop ini dieksekusi dalam waktu O(*n*), yang dapat diselesaikan meneksaminasi jumlah *i* dan *i* - *j*. dalam kasus pertama, *i* - *j* tetap, karena keduanya naik secara bersamaan. Di kasus kedua, *j* di-*assign* oleh *t*[*j*]—nilainya selalu kurang dari *j*—yang menyebabkan *i* – *j* naik. Pada kasus ketiga *i* dinaikkan dan j tidak, jadi baik *i* dan *i* - *j* naik. Karena $i \geq i - j$, ini berarti pada saat itu baik *i* atau batas bawah *i* naik; maka karena algoritma berhenti ketika $i = n$, algoritma ini harus berhenti setelah paling maksimum 2*n* iterasi loop, karena *i* - *j* mulai dari 1. Maka kompleksitas algoritma ini O(n).

2.2.8. Mengenai efisiensi algoritma

Melihat dari waktu kompleksitas kedua subalgoritma dari *partial table* dan proses pencocokan yaitu O(*k*) dan O(*n*), maka rata-rata waktu kompleksitasnya adalah O(*n* + *k*).

Dapat dilihat dari contoh diatas, algoritma ini memberikan keuntungan yang sangat besar pada saat melewati karakter-karakter dalam selang waktu tertentu. Semakin sedikit eksekusi untuk mundur, maka algoritma ini semakin cepat, yang dapat kita lihat dari kehadiran nol pada tabel T. Kata seperti "ABCDEFG" bekerja sangat baik dengan algoritma ini karena tidak ada pengulangan dalam katanya, sehingga tabel menjadi nol semua dengan awalnya -1. Namun bila kita melihat kasus algoritma dengan kata "AAAAAAA", kata ini sangat buruk, karena tabelnya akan seperti:

W[i]	a	a	a	a	a	a	a
T[i]	-1	0	1	2	3	4	5

Bentuk seperti ini adalah pola yang paling buruk untuk *partial table* T, dan dapat dilihat keburukannya jika menggunakan kata seperti S = "AAAAAABAAAAAABAAAAAAA". Algoritma ini akan mengecek seluruh 'A' terhadap karakter 'B' hingga berhenti, mencapai duakali dari jumlah karkter S. Walaupun pekerjaan untuk pembuatan tabel ini cepat untuk kata ini (karena tidak kembali ke belakang), pekerjaan ini dijalankan hanya sekali untuk *w* yang diberikan, namun akan melakukan eksekusi pencarian berkali-kali. Jika pada setiap waktu, *w* dicari dalam kata S, efisiensi rata-rata akan menjadi sangat buruk. Contoh tersebut cocok untuk algoritma pencarian Boyer-Moore.

Algoritma KMP ini menjanjikan karena mempunyai waktu pencarian yang linier, baik dalam kasus terbaik atau terburuk, sedangkan Boyer-Moore memiliki kasus terbaik yang baik, juga kasus terburuk yang buruk.

2.3. Algoritma *Boyer-Moore*

Algoritma pencarian kata Boyer-Moore termasuk algoritma yang cukup efisien. Dikembangkan oleh Bob Boyer dan J Strother Moore pada tahun 1977. Algoritma ini akan bertambah cepat jika kata yang dicari panjang. Keefisienan yang algoritma ini punya berasal dari fakta bahwa, setiap pencocokan yang gagal antara teks dan kata yang dicari, algoritma ini menggunakan informasi yang didapat dari proses awal untuk melewati karakter-karakter yang tidak cocok.

2.3.1. Cara kerja algoritma *Boyer-Moore*

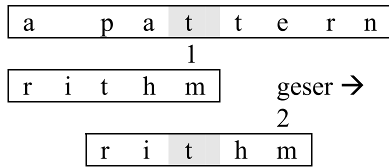
Algoritma *Boyer-Moore* didasarkan pada dua *heuristic*: looking-glass heuristic dan *character-jump heuristic*.

Looking-glass heuristic melakukan perbandingkan suatu karakter akhir pada kata *w* dengan suatu karakter pada teks *s*. Jika karakter tersebut sama maka jendela karakter akan berjalan mundur pada kedua string dan mengecek kembali kedua karakter. Ilustrasinya seperti tabel berikut, nomor merupakan langkah dari awal pencocokan.

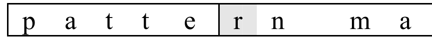
a	l	s	a	s	i	a	n
		4	3	2	1		
		s	a	s	i		

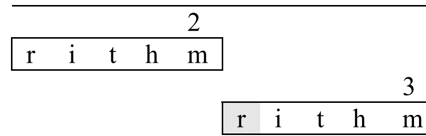
Character-jump heuristic melakukan suatu aksi ketika perbandingan antara dua karakter yang berbeda. Ada dua aksi yang tergantung pada teks *s* dan kata *w* yang dimiliki; jika *p* yaitu karakter pada *s* yang sedang diproses yang tidak cocok, maka ada dua kemungkinan aksi.

Pertama jika *p* terdapat dalam kata *w*, geser kata *w* agar karakter *p* pada *w* (pilih yang paling akhir jika ada lebih dari satu) sejajar dengan karakter *p* pada *s*. Di bawah ini ilustrasi dengan *s*="a pattern" dan *w*="rithm".



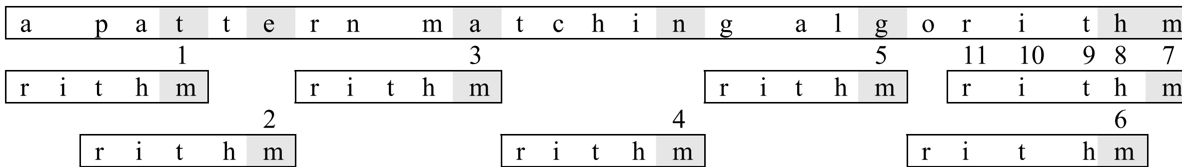
Kedua jika *p* tidak terdapat dalam kata *w*, geser kata *w* sehingga karakter awal kata *w* sejajar dengan karakter setelah *p* pada *s*. Di bawah ini ilustrasi dengan *s*=”pattern ma” dan *w*=”rithm”.





Dengan menggunakan dua *heuristic* tadi kita dapat melihat aplikasinya dalam contoh berikut: misalkan *s*=”a pattern matching algorithm” dan *w*=”rithm”. Dalam contoh ini kita akan memperlihatkan combinasi dari *looking-glass heuristic* dan *character-jump heuristic*.

Ilustrasi:



Karakter ‘X’ di posisi 8 menghilangkan semua kemungkinan di bawahnya. Dengan kata lain menghilangkan sebanyak 8 pengecekan dibandingkan dengan pencarian naif.

X
a n p a n m a n
a n p a n m a n
a n p a n m a n
a n p a n m a n
a n p a n m a n
a n p a n m a n
a n p a n m a n
a n p a n m a n

Apa yang membuat orang-orang terkejut dengan algoritma ini adalah ketika mereka melihat pemeriksaan yang dilakukannya—percobaannya mengecek apakah karakternya sama pada saat tertentu—berjalan mundur. Jika ini memulai pencarian pada awal teks untuk contohnya kata “ANPANMAN”, ini mengecek posisi kedelapan dari teks untuk melihat apakah mengandung “N”. jika menemukan karakter “N”, ini akan berpindah keposisi tujuh untuk melihat apakah mengandung “A” terakhir dari kata, dan terus-menerus sampai mengecek posisi pertama dari teks untuk “A”.

Akan semakin jelas mengapa kita harus menggunakan teknik ini ketika kita melihat kondisi jika pengecekan gagal, contohnya jika di posisi 8 adalah “X” bukan “N”. “X” tidak ada dalam “ANPANMAN”, itu berarti tidak ada kata yang cocok di awal teks—atau ditujuh posisi setelahnya, karena memang tidak ada karakter ‘X’. Setelah mengecek satu karakter, kita dapat melewati dan mencari lagi dimulai dari posisi ke 9.

Pembahasan di atas menjelaskan mengapa kasus terbaik dari algoritma ini, untuk teks sepanjang N dan pola kata dengan panjang M adalah O(N/M): pada kasus terbaiknya, hanya satu karakter pada M yang perlu untuk di periksa. Hal diatas juga terbalik dengan pernyataan jika kalimat semakin panjang maka waktunya akan bertambah lama.

2.3.2. Pemrosesan awal pada kata

Sebelumnya algoritma ini mengkomputasi dua tabel untuk memproses informasi yang didapat dari percobaan yang gagal: tabel pertama menghitung pergeseran yang harus dilakukan untuk memulai pencarian selanjutnya berdasarkan identitas karakter yang menyebabkan percobaan gagal (digunakan oleh *looking-glass heuristic*); yang lain membuat perhitungan yang mirip berdasarkan berapa banyak karakter yang telah dicocokan yang berhasil sebelum gagal. Kedua tabel tersebut memberikan nilai yang merupakan pergeseran yang harus dilakukan setelah percobaan gagal. Biasa juga disebut “*jump table*”.

Tabel pertama mudah untuk dihitung: Dimulai pada karakter akhir dari kata dengan indeks nol, dan maju ke karakter pertama; setiap bergeser ke kiri, indeks bertambah satu, dan jika karakter saat itu tidak berada dalam tabel, masukkan dalam tabel. Semua karakter lain mendapatkan nilai yang sama dengan panjang teks.

Contoh: untuk string ANPANMAN, tabel pertama seperti dibawah ini (untuk kejelasan, entri diberikan dengan urutan masuknya karakterdalam tabel):

Character	Shift
N	0
A	1
M	2
P	5
Karakter lain	8

Tabel kedua lebih sulit untuk dihitung: untuk setiap nilai *N* yang kurang dari panjang kata, pertama-tama kita menghitung pola yang terdiri dari *N* karakter akhir dari kata, dimulai dengan kesalahan pada karakter sebelumnya; lalu kita inisialisasi baris dengan kata dan menetapkan jumlah terkecil dari karakter yang bagian pola yang harus digeser kekiri sebelum dua pola cocok. Untuk contohnya, dibawah ini contoh tabel dua untuk kata ANPANMAN:

N	Pattern	Shift
0	~~N~~	1
1	~~A~~N	8
2	~~M~~AN	3
3	~~N~~MAN	6
4	~~A~~NMAN	6
5	~~P~~ANMAN	6
6	~~N~~PANMAN	6
7	~~A~~NPANMAN	6

Akan lebih mudah untuk memahami tabel di atas dengan melihat kolom pergeseran di tabel sebelah bawah, yang memperlihatkan pola yang harus digeser kekiri seperlunya agar cocok (Karakter yang tidak harus cocok ditulis dengan huruf kecil; pola "mAN" berarti cocokan string "AN" yang didahului selain 'M')

```
    ANPANMAN
    --------
          n  1
  aN         8
        mAN  3
       nMAN  6
      aNMAN  6
     pANMAN  6
    nPANMAN  6
    aNPANMAN 6
```

Kasus terburuk dari algoritma ini kira-kira O(NM). Kasus ini didapat ketika kata yang kita cari terdiri dari repetisi satu karakter, dan teks yang kita akan pakai terdiri dari M-1 repetisi dari karakter yang diawali contoh single dari karakter yang berbeda. Dalam skenario ini, algoritma ini harus memeriksa N-M+1 offsets yang berbeda dalam teks untuk pencocokan, dan setiap pengecekan terjadi M perbandingan.

Kasus terburuk untuk mencari semua kehadiran dalam teks memerlukan kira-kira 3*N perbandingan, maka kompleksitasnya O(n), tidak tergantung teksnya terdapat kata atau tidak. Ini dibuktikan oleh Richard Cole, bukti ini membutuhkan beberapa tahun untuk dibenarkan. Pada awalnya nilai maksimumnya mencapai 6*N, pada tahun 1980 dibuktikan tidak lebih dari 4*N, sampai Cole menemukan bukti baru.

2.3.3. Variasi algoritma BM : Turbo *Boyer-Moore*

Algoritma ini merupakan varian dari algoritma pencarian *Boyer-Moore* yang mengambil tambahan ruang untuk melengkapi pencarian kurang dari 2n perbandingan (*Boyer-Moore* memiliki 3n).

3. Kesimpulan

Dari ketiga algoritma yang kita bahas, kita dapat mengetahui fitur-fitur yang disediakan tiap algoritma:

Rabin-Karp: menggunakan fungsi *hash* untuk membandingkan suatu kata. Kasus terbaik dari algoritma ini bisa mencapai O(n), dimana n adalah panjang teks. Kasus terburuk dari algoritma ini adalah $O(nm)$, dimana m adalah panjang kata. Dapat mencari k kata dalam rata-rata waktu $O(n)$, berapapun nilai k. Paling baik digunakan saat ingin mencari kata jamak (lebih dari satu).

Knuth-Morris-Pratt: Perbandingan karakter dari kiri ke kanan. Rata-rata waktu kompleksitasnya adalah $O(n + k)$, dimana n adalah panjang kata dan k adalah panjang teks. Memiliki kompleksitas waktu yang linier.

Boyer-Moore: Perbandingan karakter dari kanan ke kiri. Kasus terbaik dari algoritma ini dapat mencapai O(n/m), sublinier. Namun memiliki kasus terburuk dengan 3*n* perbandingan karakter dengan pola non-periodik. Merupakan algoritma yang sangat efisien untuk penggunaan pola yang periodik.

DAFTAR PUSTAKA

[1] Algorithmdesign. (2006). *http://ww3.algorithmdesign.net/handouts/PatternMatching.pdf*. Tanggal akses: 28 Desember 2006 pukul 15:30.

[2] Lecroq, Thierry, Christian Charras. (2006). Handbook of Exact String-Matching Algoritms. *http://www-igm.univ-mlv.fr/~lecroq/string/string.pdf*. Tanggal akses: 28 Desember 2006 pukul 15:30.

[3] Li, Chen Guang. (2006), *www.scs.carleton.ca/~maheshwa/courses/5703COMP/SM_slides.pdf.* Tanggal akses: 28 Desember 2006 pukul 15:30.

[4] Wikipedia. (2006). Wikipedia, the free encyclopedia. *http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm*. Tanggal akses: 28 Desember 2006 pukul 15:30.

[5] Wikipedia. (2006). Wikipedia, the free encyclopedia. *http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm*. Tanggal akses: 28 Desember 2006 pukul 15:30.

[6] Wikipedia. (2006). Wikipedia, the free encyclopedia. *http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm*. Tanggal akses: 28 Desember 2006 pukul 15:30.

[7] Zachariasen, Martin. (2004). String Matching CLRS Chapter 32. http://www.diku.dk/undervisning/2004e/dat-algo/stringmatching.pdf*. Tanggal akses: 28 Desember 2006 pukul 15:30.*