Perbandingan dan Pengujian Beberapa Algoritma Pencocokan String

Hary Fernando

Program Studi Teknik Informatika Institut Teknologi Bandung
Jln. Ganesha No.10 Bandung,
e-mail: hary@hary.web.id

ABSTRAK

Pencocokan string atau *string matching* merupakan hal dasar yang sangat perlu dipelajari terutama dalam lingkup yang berkaitan dengan *text processing*. Algoritma pencocokan string merupkan komponen dasar yang digunakan dalam implementasi pembuatan perangkat lunak yang terdapat dalam sistem operasi. Pencocokan String juga memerankan peranan dalam bidang teori ilmu komputer dengan memberikan persoalan baik dari yang sederhana sampai yang kompleks untuk diselesaikan.

Terdapat banyak sekali algoritma pencocokan string yang terdapat sampai saat ini, antara lain adalah algoritma *Brute Force*, Boyer–Moore, Knuth-Morris-Pratt, algoritma Karp-Rabin, algoritma Shift Or, dan algoritma lainnya. Pada makalah ini akan dibahas algoritma-algoritma tersebut dan studi kasus pencocokan string dengan teks dan *pattern* yang diberikan.. Untuk selanjutya Pencocokan string akan mengandung makna yang sama dengan pencarian string .

Kata kunci: *Pattern Matching*, Pencocokan String, algoritma *Brute Force*, algoritma Karp-Rabin,, Algoritma shift-or, algoritma Knuth-Morris-Pratt, Algoritma Boyer-Moore.

1. PENDAHULUAN

Walaupun data di simpan dalam bentuk yang bermacam-macam, representasi teks tetap menjadi bentuk utama dalam perturakan informasi.

Menurut cara pembacaan teks, algoritma pencocokan string dapat dibedakan atas dua cara pembacaan :

1. dari kiri ke kanan

Alogoritma pencarian dengan teknik ini sangat banyak. Hampir sebagian besar algoritma pencarian menggunakan cara pembacaan teks dari kiri ke kanan.

2. dari kanan ke kiri

Dalam Algoritma ini, terdapat algoritma Boyer-Moore yang dianggap merupakan salah satu algoritma yang utama dan algoritma standar dalam pencocokan string[1].

Sampai saat ini terdapat banyak algoritma pencocokan string, menurut [2] ada sekitar 35 algoritma yang bisa digunakan, baik merupkan algoritma yang diciptakan dari awal maupun berupa pengembangan dari algoritma yang sudah ada. Untuk kompleksitas waktu beberapa algoritma yang akan dibahas, dapat dilihat di tabel 1.

2. ALGORITMA PENCOCOKAN STRING

Secara umum, istilah yang terdapat dalam pencocokan string antara lain teks dan pattern. teks (text) adalah (long) string yang panjangnya n. *pattern* yaitu string dengan panjang m karakter (m < n) yang akan dicari di dalam teks[3].

Tabel 1 Perbandingan Beberapa Algoritma Pencocokan String

Algorithm	Fase Preprocessing	Fase Pencarian
Brute Force	0 (tidak ada)	O(mn)
Rabin-Karp	O(m)	O(mn)
Shift Or	$O(m+\sigma)$	O(n)
Knuth-Morris-Pratt	O(m)	O(n+m)
Boyer–Moore	$O(m+\sigma)$	O(n/m), O(n)

2.1 Algoritma *Brute Force*

Algoritma *Brute Force* merupakan algoritma paling lempang untuk menyelesaikan persoalan pencocokan string.

Karakteristik algoritma *Brute Force* :

- Tidak perlu fase *prepocessing* (tahap sebelum melakukan search atau pencocokan string).
- Selalu berpindah tepat 1 langkah ke kanan.
- Perbandingan dapat dilakukan pada urutan apa saja.
- Fase pencarian memiliki kompleksitas **O**(mn)
- perbandingan karakter yang terjadi diharapkan 2n

Algoritma *Brute Force* melakukan pencarian pada setiap posisi di dalam teks antara 0 dan n-m, tidak peduli apakah terjadi pengulangan pola atau tidak. Kemudian, setelah setiap percobaan, *pattern* di geser tepat 1 posisi ke kanan.

2.2 Algoritma Karp-Rabin

Algoritma Karp-Rabin diciptkan oleh Michael O. Rabin dan Richard M. Karp pada tahun 1987 yang menggunakan fungsi *hashing*[4] untuk menemukan *pattern* di dalam string teks .

Karakterikstik Algoritma Karp-Rabin :

- menggunakan sebuah fungsi *hashing*
- fase *prepocessing* menggunakan kompleksitas waktu O(m)
- Untunk fase pencarian kompleksitasnya : O(mn)
- Waktu yang diperlukan O(n+m)

Fungsi *hashing* menyediakan metoda sederhana untuk menghindari perbandingan jumlah karakter yang quadratik di dalam banyak kasus atau situasi. Dari pada melakukan pemeriksaan terhadap setiap posisi dari teks ketika terjadi pencocokan pola, sepertinya lebih baik effisien untuk melakukan pemeriksaan hanya jika teks yang sedang kita proses memiliki kemiripan seperti pada *pattern*. Untuk melakukan pengecekan kemiripan antara dua kata ini digunakan fungsi *hash*.

Untuk membantu algoritma ini, maka fungsi *hash* harus memenuhi hal-hal berikut ini :

- dapat dihitung dengan efisien
- memilik perbedaan yang tinggi untuk berbagai jenis string
- *hash* ($y[j+1 .. j+m]$) dapat dihitung dari *hash*($y[j .. j+m-1]$) dan $y[j+m]$; yaitu : *hash*($y[j+1 .. j+m]$)= *rehash*($y[j]$, $y[j+m]$, *hash*($y[j .. j+m-1]$).

Untuk setiap *word*(8 bit) w yang memiliki panjang m, misalkan *hash*(w) didefinisikan sebagai berikut :

$$hash(w[0 .. m-1])=(w[0]*2^{m-1}+ w[1]*2^{m-2}+\cdots+ w[m-1]*2^{0}) \bmod q$$

dimana q merupakan bilangan yang besar.

Kemudian, lakukan *rehash* dengan rumus :

$$rehash(a,b,h)= ((h-a*2^{m-1})*2+b) \bmod q$$

Fase *prepocessing* dari algoritma Karp-Rabin mengandung perhitungan terhadap *hash*(x). Hal ini dapat dilakukan dalam waktu yang memilki kompleksitas O(m)

Selama fase pencarian, hal yang perlu dilakukan cukup membandingkan *hash*(x) dengan *hash*($y[j .. j+m-1]$) untuk $0 <= j < n-m$.

Jika kesamaannya ditemukan, masih perlu melakukan pemeriksaan kesamaan $x=y[j .. j+m-1]$ untuk karakter-karakter selanjutnya.

Kompleksitas waktu untuk fase pencarian dari algoritma Karp-Rabin ini adalah O(mn). Diharapkan jumlah karakter teks yang dibandingkan adalah O(m+n)

Algoritma Karp-Rabin ini banyak digunakan dalam pendeteksian pencontekan atau kecurangan. Contohnya pada makalah atau pada paper. Bila diberikan *source matrial* atau dokumennya, algoritma ini dapat dengan cepat mencari seluruh paper dari setiap kalimat, mengabaikan *lowercase* atau *uppercase*, tanda titik, tanda seru,tanda tanya serta tanda baca lainnya.

Fungsi *hash* yang digunakan biasanya berbasis bilangan prima besar.

Contoh: jika substringnya “hi” dan basisnya 101, maka nilai *hash*nya dapat di hitung : $104 \times 101^1 + 105 \times 101^0 = 10609$ (ASCI untuk 'h' adalah 104, dan untuk 'i' adalah 105)

2.3 Algoritma Shift Or

Algoritma Shift Or yang juga dikenal dengan nama shift-and, Bitap atau Baeza-Yates-Gonnet adalah algoritma pencarian *fuzzy string*[5].

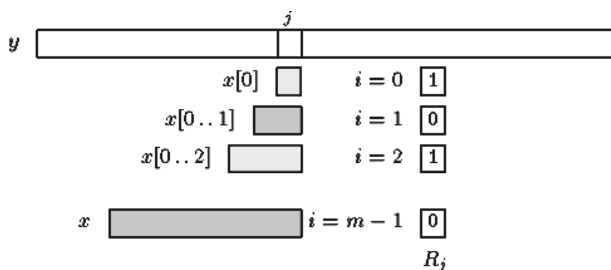
Algoritma Shift Or untuk pencarian string ditemukan oleh Balint Domolki pada tahun 1964[6] kemudian dikembangkan oleh R.K Shyamasundar pada tahun 1977 sebelum ditemukan kembali untuk pencarian string *fuzzy* oleh Menber dan Wu pada tahun 1991[8] berdasarkan kerja yang dilakukan oleh Ricardo Baeza-Yates dan Gaston Gonnet.

Karakterikstik utama algoritma Shift Or :

- menggunakan teknik perhitungan pada level bit
- algoritma ini efisien jika panjang *pattern* tidak lebih dari ukuran memori untuk menyimpan 1 *word*(8bit) di dalam komputer yang digunakan.
- fase *prepocessing* memerlukan waktu dengan kompleksitas $O(m + \sigma)$
- fase pencarian dengan kompleksitas O(n)
- dapat digunakan untuk pendekatan pencarian string

Algoritma Shift Or menggunakan operasi level bit. Misalkan R adalah *array of bit* dengan ukuran m. Vektor Rj adalah nilai dari vektor R seteleh karakter teks $y_{[j]}$ di proses (lihat gambar 1). Vektor Rj mengandung informasi tentang semua kecocokan dari awal dari x yang berakhir di posisi j di dalam teks untuk $0 < i <= m-1$ yaitu :

$$R_j[i] = \begin{cases} 0 & \text{if } x[0,i] = y[j-i,j], \\ 1 & \text{otherwise.} \end{cases}$$



Gambar 1: Vektor Rj dari teks y pada karakter ke j

Vektor R_{j+1} dapat dihitung setelah menghitung R_j, yaitu dengan cara sebagai berikut:

Untuk setiap Rj[i]=0 :

$$R_{j+1}[i+1] = \begin{cases} 0 & \text{if} \quad x[i+1] = y[j+1], \\ 1 & \text{otherwise}, \end{cases}$$

Dan

$$R_{j+1}[0] = \begin{cases} 0 & \text{if} \quad x[0] = y[j+1], \\ 1 & \text{otherwise}. \end{cases}$$

Jika Rj+1[m-1] = 0 maka kecocokan yang lengkap telah didapatkan.

Perubahan dari Rj ke Rj+1 dapat dihitung dengan sangat cepat sebagai berikut : untuk setiap c di Σ, misalkan S_c adalah *array of bit* dengan ukuran m sehingga untuk 0 <= I < m-1, maka $S_c[i] = 0$ jika dan hanya jika x[i] = c.

Array S_c menyatakan posisi dari karakter c di dalam *pattern* x. Setiap S_c dapat di hitung dahulu sebelum dilakukan fase pencarian. Dan perhitungan dari R_{j+1} mengurangi dua buah operasi, shift dan or : R_{j+1} = SHIFT(Rj) OR $S_y[j+1]$.

Misalkan bahwa panjang *pattern* tidak lebih panjang dari panjang tempat penyimpan 1 *word*(8 bit) di komputer, maka kompleksitas ruang dan waktu dari fase perhitungannya adalah O(m+ σ) dan kompleksitas waktu untuk fase pencarian adalah O(n) karena tidak bergantung pada ukuran alfabet dan panjang *pattern*.

Gambar 2 berikut adalah contoh fase pencari pada algoritma Shift Or :

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
0	G	0	1	1	1	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0
1	C	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	A	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	G	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	A	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	G	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
6	A	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
7	G	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

Gambar 2: Contoh pencarian string pada Shift Or

Dari gambar 2 dapat dilihat bahwa karena $R_{12}[7] = 0$, maka ini berarti bahwa terjadi pengulangan dari x yang telah ditemukan pada posisi 12-8+1 = 5.

Algoritma ini terkenal digunakan oleh program agrep pada sistem Unix khusunya dalam bidang yang berkaitan dengan *regular expression*.

2.4 Algoritma Knuth-Morris-Pratt

Algoritma Knuth-Morris-Pratt (KMP) bergerak dari kiri ke kanan seperti algortima *Brute Force* tetapi memilik kemampuan yang lebih baik dalam hal melakukan pergeseran *pattern*.

Rancangan algoritma Knuth-Morris-Pratt mengikuti analisis dari algoritma Morris dan Pratt yang sebelumnya telah ditemukan terlebih dahulu oleh J.H Morris (jr) dan V.R Pratt pada tahun 1970[7]. Bersama dengan D.E Knuth, algoritma ini menjadi Knuth-Morris-Pratt dengan berbagai perbaikan dari algoritma sebelumnya.

Karakteristik utama algorima KMP :

- kompleksitas ruang dan waktu untuk fase *prepocessing* adalah O(m)
- kompleksitas waktu untuk fase pencarian : O(n+m)

Pada algoritma KMP dikenal adanya fungsi pinggiran(*Border Function*)[5] yang didefinisikan sebagai ukuran terpanjang dari *pattern* yang juga akhiran dari *pattern* tersebut.

2.5 Algoritma Boyer-Moore

Algoritma Boyer-Moore dianggap sebagai algoritma pencocokan string yang paling efisien pada penggunaan biasa karena algoritma Boyer-Moore telah menjadi standar untuk pencarian string menurut[5].

Berbagai versi algoritma ini digunakan dalam *teks editor* untuk perintah pencarian dan pergantian (*find and replace*)

Algoritma pencocokan string Boyer-Moore didasarkan atas dua teknik :

1. Teknik *looking-glass,* menemukan *pattern* di dalam teks dengan menggerakan *pattern* mundur dimulai dari akhir teks.
2. Teknik *character-jump,* pergeseran karakter yang dilakukan saat terjadi ketidak cocokan

Karakteristik utama algoritma Boyer-Moore :

- melakukan perbandingan dari kanan ke kiri
- fase persiapan / *prepocessing* membutuhkan kompleksitas waktu O(m + σ)
- fase pencarian : kompleksitas waktunya O(mn)
- pada kasus terburuk, sebanyak 3n karakter teks yang dibandingkan untuk *pattern* yang tak berulang.
- Kasus terbaik O(n/m)

3. STUDI KASUS

Pada bagian ini akan dilakukan studi kasus pencarian *pattern* terhadap teks dengan menggunakan beberapa algoritma yang telah dibahas di atas.

Misalkan *pattern* yang dicari adalah `gcagagag` yang memiliki panjang 8 karakter dan teksnya : `gcatcgcagagagtatacagtacg` dengan panjang 24 karakter. Berikut dijelaskan langkah-langkah penyelesaian dengan beberapa algoritma yang telah dibahas.

3.1 Algoritma *Brute Force*

Langkah pencarian dengan algoritma *Brute Force* dapat dilihat sebagai berikut :

```
percobaan 1:
gcatcgcagagagtatacagtacg
GCAg....

percobaan 2:
gcatcgcagagagtatacagtacg
 g.......

percobaan 3:
gcatcgcagagagtatacagtacg
  g.......

percobaan 4:
gcatcgcagagagtatacagtacg
   g.......

percobaan 5:
gcatcgcagagagtatacagtacg
    g.......

percobaan 6:
gcatcgcagagagtatacagtacg
     GCAGAGAG

percobaan 7:
gcatcGCAGAGAGtatacagtacg
      g.......

percobaan 8:
gcatcGCAGAGAGtatacagtacg
       g.......

percobaan 9:
gcatcGCAGAGAGtatacagtacg
        Gc......

percobaan 10:
gcatcGCAGAGAGtatacagtacg
         g.......

percobaan 11:
gcatcGCAGAGAGtatacagtacg
          Gc......

percobaan 12:
gcatcGCAGAGAGtatacagtacg
           g.......

percobaan 13:
gcatcGCAGAGAGtatacagtacg
            Gc......

percobaan 14:
gcatcGCAGAGAGtatacagtacg
             g.......

percobaan 15:
gcatcGCAGAGAGtatacagtacg
              g.......

percobaan 16:
gcatcGCAGAGAGtatacagtacg
               g.......

percobaan 17:
gcatcGCAGAGAGtatacagtacg
                g.......

gcatcGCAGAGAGtatacagtacg
```

total percobaan: 17
perbandingan karakter yang terjadi : 30

3.2 Algoritma Karp-Rabin

Algoritma ini terlebih dahulu menghitung nilai *hash* dari *pattern* kemudian baru melakukan pencarian.

```
hash(gcagagag)=25757

hash(y[0..7])=25979
percobaan 1:
gcatcgcagagagtatacagtacg
........

hash(y[1..8])=25693
percobaan 2:
gcatcgcagagagtatacagtacg
 ........

hash(y[2..9])=26139
percobaan 3:
gcatcgcagagagtatacagtacg
  ........

hash(y[3..10])=27549
percobaan 4:
gcatcgcagagagtatacagtacg
   ........

hash(y[4..11])=25499
percobaan 5:
gcatcgcagagagtatacagtacg
    ........
```

```
hash(y[5..12])=25757
percobaan 6:
gcatcgcagagagtatacagtacg
     GCAGAGAG

hash(y[6..13])=25262
percobaan 7:
gcatcGCAGAGAGtatacagtacg
      ........

hash(y[7..14])=25277
percobaan 8:
gcatcGCAGAGAGtatacagtacg
       ........

hash(y[8..15])=25838
percobaan 9:
gcatcGCAGAGAGtatacagtacg
        ........

hash(y[9..16])=25405
percobaan 10:
gcatcGCAGAGAGtatacagtacg
         ........

hash(y[10..17])=26077
percobaan 11:
gcatcGCAGAGAGtatacagtacg
          ........

hash(y[11..18])=25883
percobaan 12:
gcatcGCAGAGAGtatacagtacg
           ........

hash(y[12..19])=27037
percobaan 13:
gcatcGCAGAGAGtatacagtacg
            ........

hash(y[13..20])=27822
percobaan 14:
gcatcGCAGAGAGtatacagtacg
             ........

hash(y[14..21])=26045
percobaan 15:
gcatcGCAGAGAGtatacagtacg
              ........

hash(y[15..22])=27357
percobaan 16:
gcatcGCAGAGAGtatacagtacg
               ........

hash(y[16..23])=25121
percobaan 17:
gcatcGCAGAGAGtatacagtacg
                ........


gcatcGCAGAGAGtatacagtacg
```

percobaan: 17
perbandingan karakter yang terjadi : 8

3.3 Algoritma Shift Or

Dengan menggunakan algoritma Shift Or, langkahn-langkahnya :

```
gcatcgcagagagtatacagtacg
  |
  transition: 11111111.g -> 01111111

  Gcatcgcagagagtatacagtacg
   |
  transition: 01111111.c -> 10111111

  GCatcgcagagagtatacagtacg
    |
  transition: 10111111.a -> 11011111

  GCAtcgcagagagtatacagtacg
     |
  transition: 11011111.t -> 11111111

  gcatcgcagagagtatacagtacg
      |
  transition: 11111111.c -> 11111111

  gcatcgcagagagtatacagtacg
       |
  transition: 11111111.g -> 01111111

  gcatcGcagagagtatacagtacg
        |
  transition: 01111111.c -> 10111111

  gcatcGCagagagtatacagtacg
         |
  transition: 10111111.a -> 11011111

  gcatcGCAgagagtatacagtacg
          |
  transition: 11011111.g -> 01101111

  gcatcGCAGagagtatacagtacg
           |
  transition: 01101111.a -> 11110111

  gcatcGCAGAgagtatacagtacg
            |
  transition: 11110111.g -> 01111011

  gcatcGCAGAGagtatacagtacg
             |
  transition: 01111011.a -> 11111101

  gcatcGCAGAGAgtatacagtacg
              |
  transition: 11111101.g -> 01111110

  gcatcGCAGAGAGtatacagtacg
```

```
             |
transition: 01111110.t -> 11111111

gcatcGCAGAGAGtatacagtacg
              |
transition: 11111111.a -> 11111111

gcatcGCAGAGAGtatacagtacg
               |
transition: 11111111.t -> 11111111

gcatcGCAGAGAGtatacagtacg
                |
transition: 11111111.a -> 11111111

gcatcGCAGAGAGtatacagtacg
                 |
transition: 11111111.c -> 11111111

gcatcGCAGAGAGtatacagtacg
                  |
transition: 11111111.a -> 11111111

gcatcGCAGAGAGtatacagtacg
                   |
transition: 11111111.g -> 01111111

gcatcGCAGAGAGtatacaGtacg
                    |
transition: 01111111.t -> 11111111

gcatcGCAGAGAGtatacagtacg
                     |
transition: 11111111.a -> 11111111

gcatcGCAGAGAGtatacagtacg
                      |
transition: 11111111.c -> 11111111

gcatcGCAGAGAGtatacagtacg
                       |
transition: 11111111.g -> 01111111

gcatcGCAGAGAGtatacagtacg
```

jumlah karakter yang diperiksa: 24

3.4 Algoritma Knuth-Morris-Pratt

Dengan menggunakan algoritma Knuth-Morris-Pratt algorithm dilakukan langkah-langkah sebagai berikut :

```
kmpNext:
   0  1  2  3  4  5  6  7  8
  -1  0  0 -1  1 -1  1 -1  1

 percobaan 1:
 gcatcgcagagagtatacagtacg
 GCAg....
 Shift by 4 (3-kmpNext[3])

 percobaan 2:
 gcatcgcagagagtatacagtacg
     g.......
 Shift by 1 (0-kmpNext[0])

 percobaan 3:
 gcatcgcagagagtatacagtacg
      GCAGAGAG
 Shift by 7 (8-kmpNext[8])

 percobaan 4:
 gcatcGCAGAGAGtatacagtacg
             .c......
 Shift by 1 (1-kmpNext[1])

 percobaan 5:
 gcatcGCAGAGAGtatacagtacg
              g.......
 Shift by 1 (0-kmpNext[0])

 percobaan 6:
 gcatcGCAGAGAGtatacagtacg
               g.......
 Shift by 1 (0-kmpNext[0])

 percobaan 7:
 gcatcGCAGAGAGtatacagtacg
                g.......
 Shift by 1 (0-kmpNext[0])

 percobaan 8:
 gcatcGCAGAGAGtatacagtacg
                 g.......
 Shift by 1 (0-kmpNext[0])


 gcatcGCAGAGAGtatacagtacg
```

percobaan: 8
perbandingan karakter yang terjadi : 18

3.5 Algoritma Boyer-Moore

Pencariang dengan algoritma Boyer-Moore dapat dilihat sebagai berikut :

```
bmBc:
   a c g t
   1 6 2 8
  BmGs:
   0 1 2 3 4 5 6 7
   7 7 7 2 7 4 7 1

  percobaan 1:
  gcatcgcagagagtatacagtacg
  .......g
  Shift by 1 (bmGs[7]=bmBc[a]-8+8)

  percobaan 2:
  gcatcgcagagagtatacagtacg
```

```
 .....gAG
Shift by 4 (bmGs[5]=bmBc[c]-8+6)

percobaan 3:
gcatcgcagagagtatacagtacg
     GCAGAGAG
Shift by 7 (bmGs[0])

percobaan 4:
gcatcGCAGAGAGtatacagtacg
            .....gAG
Shift by 4 (bmGs[5]=bmBc[c]-8+6)

percobaan 5:
gcatcGCAGAGAGtatacagtacg
                ......aG
Shift by 7 (bmGs[6])


gcatcGCAGAGAGtatacagtacg
```

percobaan: 5
perbandingan karakter yang terjadi: 17

IV. KESIMPULAN

Terdapat banyak algoritma untuk pencocokan dan pencarian string, baik yang dilakukan dari kanan ke kiri maupun dari kiri ke kanan. Algoritma *Brute Force* merupakan algoritma paling lempang untuk menyelesaikan masalah pencocokan string ini.

Untuk single *pattern*, yaitu ketika *pattern* yang dicari hanya satu atau tunggal, algoritma Karp-Rabin kalah dibandingkan dengan Knuth-Morris-Pratt, Boyer-Moore dan algoritma pencocokan string cepat yang lain, karena kelambatannya dalam kasus terburuk. Tetapi algoritma Rabin-Karp adalah pilihan untuk pencarian dengan banyak *pattern*. Namun di sini tidak dibahas untuk pencarian *multi pattern*.

Algoritma Shift-Or menggunakan pengoperasian bit untuk mencari kesamaan *pattern* dan teks. Algoritma Knuth-Morris-Pratt menggunakan tabel fungsi pinggiran untuk melakukan pergesaran. Algorima Boyer-Moore telah menjadi standar untuk pencarian string karena keefisienannya dengan ciri khasnya melakukan pencarian dari kanan ke kiri.

REFERENSI

[1] Hume and Sunday (1991) "*Fast String Searching*" SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(11), 1221–1248, 1991

[2] http://www-igm.univ-mlv.fr/~lecroq/string Waktu akses 4 Januari 2010

[3] Munir. Rinaldi, "IF2251 STRATEGI ALGRORITMIK Diktat Kuliah Strategi Algoritmik", Departemen Teknik Informatika, 2005

[4]http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf Waktu Akses 4 Januari 2010

[5] Udi Manber, Sun Wu. "*Fast text searching with errors*." *Technical Report* TR-91-11. Department of Computer Science,University of Arizona, Tucson, June 1991

[6] Balint Dömölki, "*An algorithm for syntactical analysis, Computational Linguistics* 3", Hungarian Academy of Science, 1964

[7] MORRIS (Jr) J.H., PRATT V.R. "*A linear pattern-matching algorithm"*, Technical Report 40, University of California, Berkeley 1970