Pemanfaatan Algoritma *String Matching* untuk *Command Input* pada *Line Bot*

Jason Wiguna
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl Ganesha 10 Bandung 40132
Bandung, Indonesia
13516024@std.stei.itb.ac.id

Abstrak*—*Line Bot* menyediakan layanan *Line API* dimana pengguna dapat menyesuaikan jawaban dengan reply dari pengguna. Kemudian dari jawaban tersebut, respon akan diproses sehingga pengguna dapat melakukan banyak hal seperti membuat bot yang dapat menyimpan daftar kegiatan pada sebuah basis data, atau mengirimkan pesan secara periodik untuk pengingat. Untuk itu diperlukan sebuah mesin kata yang dapat mengambil *keyword* yang kemudian akan digunakan sebagai command untuk mengatur apa yang akan dilakukan oleh program ketika menerima input dari pengguna. Salah satu cara untuk memisahkan *keyword* tersebut adalah dengan menggunakan algoritma *string matching* seperti pada algoritma Knuth-Morris-Pratt, Boyer-Moore, atau *Regular Expression.

Kata kunci*—*Line; bot; string; matching

I. PENDAHULUAN

Line adalah sebuah media komunikasi yang memampukan pengguna untuk mengirim pesan kepada seorang pengguna yang lain. Line dapat digunakan di berbagai media seperti ponsel cerdas, komputer, komputer riba dan media pintar lainnya. Oleh karena kemudahan akses ini, Line berkembang dengan pesat hingga saat ini ada lebih dari 200 juta pengguna aktif Line setiap bulannya [1]. Penulis melihat potensi yang dapat dikembangkan dengan memanfaatkan media Line sehingga penulis membuat makalah mengenai pemanfaatan Line yang sedang berkembang dan mempunyai banyak pengguna ini supaya perkembangan teknologi dapat dimanfaatkan dengan baik dan semakin berkembang lagi.

Line pertama kali diluncurkan di Jepang pada tahun 2011 sebagai sebuah sarana untuk mengirimkan informasi melalui *internet*. Kemudian dalam beberapa bulan setelahnya, pengguna Line mencapai 100 juta pengguna. Karena kemajuan yang pesat ini dan semakin banyaknya orang yang mulai bergantung pada sosial media, semakin mudah untuk Line menagambil kesempatan ini untuk menyebarkan dampaknya ke orang yang lebih banyak lagi. Sekarang, Line menyediakan layanan *API* yang dapat digunakan oleh pengembang di mana pun di seluruh dunia sehingga Line menjadi pusat dari sosial media di dunia.

Sekarang ini, sudah banyak pengembang yang bermain dengan menggunakan Line sebagai sarana untuk mereka melakukan aktivitasnya. Salah satu aplikasinya adalah penjawab dan pemroses pesan otomatis melalui *Line API*. Dengan ini, sebuah aplikasi yang mencatat dan mengirimkan pesan pengingat dapat dibuat dengan mudah. Namun supaya lebih efektif, penulis ingin memaparkan metode supaya satu aplikasi dapat digunakan untuk melakukan beberapa fungsionalitas yang berbeda dengan cara memberikan kata kunci yang berbeda untuk setiap fungsionalitas. Pemilihan kata kunci ini dilakukan dengan menggunakan algoritma *String Matching*.

II. DASAR TEORI

A. *Pencocokan String (String Matching)*

Pencocokan *String* adalah algoritma yang digunakan untuk mencari kemunculan dari sebuah *pattern string* pada *string* lain yang lebih besar. Contohnya adalah pada saat menekan ctrl+f untuk mencari kata pada laman *web*. [2]

Beberapa contoh pencocokan *string*:

Pattern: hari

Text: ***hari*** kemerdekaan Indonesia

Pattern: apa

Text: si***apa***kah penemu lampu?

Ada banyak algoritma *String Matching* yang telah ditemukan dan dikembangkan. Algoritma-algoritma tersebut menggunakan berbagai strategi yang mempunyai kemiripan yang dapat diklasifikasikan ke dalam beberapa jenis: [3]

- *From left to right*

  Strategi ini melakukan pencocokan *string* dari huruf paling kiri pattern ke yang paling kanan. Beberapa di antaranya:

1. Brute Force

2. *Karp and Rabin*
3. *Shift Or*
4. *Morris and Pratt*
5. *Knuth, Morris and Pratt*
6. *Deterministic Finite Automaton*
7. *Forward Dawg Matching*
8. *Apostolico-Crochemore*
9. *Not So Naive*

• *From right to left*

Strategi ini melakukan pencocokan string dari huruf paling kanan pattern ke yang paling kiri. Beberapa di antaranya:

1. *Boyer-Moore*
2. *Apostolico and Giancarlo*
3. *Crochemore et alii (Turbo BM)*
4. *Colussi (Reverse Colussi)*
5. *Quick Search*
6. *Reverse Factor*
7. *Turbo Reverse Factor*
8. *Zhu Takaoka*
9. *Berry-Ravindran*

• *In a specific order*

Strategi ini melakukan pencocokan string dengan terlebih dahulu melakukan partisi pada teks yang ingin dicari. Beberapa di antaranya:

1. *Two Way*
2. *Colussi*
3. *Galil-Giancario*
4. *Optimal Mismatch*
5. *Maximal Shift*
6. *Skip Search*
7. *KMP Skip Search*
8. *Alpha Skip Search*

• *In any order*

Strategi ini melakukan pencocokan string dengan urutan yang bebas. Beberapa di antaranya:

1. *Horspool*
2. *Tuned Boyer-Moore*
3. *Smith*
4. *Raita*

Makalah ini hanya akan membahas beberapa algoritma yang paling terkenal.

Algoritma yang pertama adalah algoritma *Brute Force*. Algoritma *Brute Force* melakukan pencarian dari kiri ke kanan dan iterasi dilakukan per huruf. Berikut adalah cara kerja algoritma *Brute Force*: [4]

Pattern: o wo

Text: hello world!

1. hello world!
   !=
   o wo
2. hello world!
   !=
   o wo
3. hello world!
   !=
   o wo
4. hello world!
   !=
   o wo
5. hello world!
   =
   o wo
6. hello world!
   =
   o wo
7. hello world!
   =
   o wo
8. hello world!
   =
   o wo

```
def BF(text, pattern)
    list = []
    n = text.length
    m = pattern.length
    for i in 1..(n-m+1) do
        j = 0
        while (j < m and text[i+j] == pattern[j])
            j+=1
        end
        if (j==m)
```

```
                list.push(i)
                list.push(i+pattern.length)
            end
        end
        return list
    end
```

Kompleksitas waktu dari algoritma *Brute Force* ini:

Kasus terburuk: O(mn)

Kasus terbaik: O(n)

Kasus rata-rata: O(m+n)

dimana n = panjang *string* teks dan m = panjang string *pattern*.

Satu algoritma *String Matching* yang sekarang mempunyai kompleksitas waktu yang linear adalah algoritma Boyer-Moore. Terdapat 2 jenis pergeseran pada algoritma Boyer-Moore: [5]

1. *Good-suffix shift (matching shift)*

	a	t	c	a	c	a	t	c	a	t	c	a
j	1	2	3	4	5	6	7	8	9	10	11	12
f	0	0	0	1	0	1	2	3	4	2	3	4

Good-suffix shift memposisikan bagian *string* dari teks yang telah cocok (misalnya m) dengan bagian *string* dari *pattern* yang paling kanan (setelah bagian teks yang cocok). Contohnya:

```
          v
....abcdabceabcfabc...
    ...xabcfabcfabc
        …xabcfabcfabc
```

Jika tidak ada bagian yang cocok seluruhnya, maka *pattern* akan digeser sampai bagian *prefix* dari *pattern* cocok dengan *suffix*-nya. Contohnya:

```
      v
...robocab....
   abacab
      abacab
```

2. *Bad-character shift (occurence shift)*

Bad-character shift menggeser pattern sampai karakter teks yang tidak cocok merupakan kemunculan terakhir pada *pattern*

```
          v
....abcdabceabcfabc...
    ...eabcfabcfabc
        …eabcfabcfabc
```

Bila tidak ada, maka *pattern* akan digeser hingga satu posisi di sebelah *bad-character* tersebut. Contohnya:

```
      v
...robocab....
   abacab
      abacab
```

Apabila terjadi ketidakcocokan, maka akan dipilih dari *good-character shift* dan *bad-character shift*, mana yang paling baik.

Berikut adalah contoh *string matching* menggunakan algoritma Boyer-Moore:

Pattern: EXAMPLE

Text: SSIMPLE EXAMPLE

1. SSIMPLE EXAMPLE
   EXAMPLE
2. SSIMPLE EXAMPLE
      EXAMPLE
3. SSIMPLE EXAMPLE
         EXAMPLE
4. SIMPLE EXAMPLE
         EXAMPLE

Implementasi program algoritma Boyer-Moore pada bahasa Ruby:

```
def BM(text, pattern)
    txt = text.downcase
    pat = pattern.downcase
    position = []
    if (pattern.length > text.length)
        return position
    end
    table = BM_Table(pat)
    i = pattern.length - 1
    pat_idx = pattern.length - 1

    while (i < text.length) do
        if (pat[pat_idx] == txt[i])
            if (pat_idx == 0)
                position.push(i)
position.push(i+pattern.length)
                pat_idx = pattern.length -
1
```

```
                    i += ((pattern.length * 2) - 1)
                else
                    pat_idx -= 1
                    i -= 1
                end
            else
                up = table[text[i]]
                if up == nil
                    i += pattern.length
                else
                    i += up
                end
                pat_idx = pattern.length - 1
            end
        end
        return position
    end

    def BM_Table(pattern)
        table = {}
        i = 0
        (pattern.downcase).each_char do |exc|
            if (i == pattern.length-1) and not (table.has_key?exc)
                table[exc] = pattern.length
            else
                table[exc] = (pattern.length - i - 1)
            end
            i += 1
        end
        return table
    end
```

Kompleksitas waktu dari algoritma Boyer-Moore ini:

Preprocessing: O(m+a)

Pencarian *string*:

- Kasus terbaik: O(n/m)
- Kasus terburuk: O(nm)

dengan n = panjang *string* teks dan m = panjang string *pattern* [6]

Algoritma lain yang sering digunakan adalah algoritma Morris dan Pratt, yang sekarang telah disempurnakan oleh Knuth, Morris dan Pratt, dan sekarang lebih dikenal dengan algoritma Knuth-Morris-Pratt (KMP). Cara kerja algoritma KMP adalah sebagai berikut: [7]

Pattern: atcacatcatca

Text: gatcgatcacatcatcacgaaaa

1. *Preprocess*: mencari fungsi pinggiran KMP dari *pattern*

   Arti dari f(4) = 1 yaitu

   - cari ukuran *prefix* terbesar dari P[1..4] (“atca”) yang merupakan *suffix*
   - cari ukuran dari “a” = 1

2. Melakukan pencarian *pattern* dalam teks. KMP melakukan pencarian *string* dari kiri ke kanan.

```
gatcgatcacatcatcacgaaaa
atcacatcatca
 atcacatcatca
  atcacatcatca
   atcacatcatca
    atcacatcatca
     atcacatcatca (matched)
```

Implementasi algoritma ini dalam bahasa Ruby:

```
def KMP(text, pattern)
    tulisan = text.downcase
    pat = pattern.downcase
    position = []
    j = 0
    k = 0
    table = KMP_Table(pattern)

    nP = 0

    while (j < tulisan.length) do
        if (pat[k] == tulisan[j])
            j = j+1
            k = k+1
            if k == pat.length
                position.push(j - k)
                position.push(j - k+pattern.length)
                nP = nP + 1
                k = table[k]
            end
        else
```

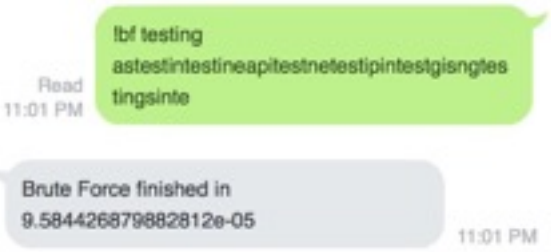
```
                k = table[k]
                if (k<0)
                        j += 1
                        k += 1
                end
            end
        end
        return position
    end

    Fungsi pinggirannya:
    def KMP_Table(pattern)
        pat = pattern.downcase
        count = []
        pos = 1
        cnd = 0

        count.push(-1)
        while (pos < pattern.length) do
            if (pat[pos] == pat[cnd])
                count.push(count[cnd])
                pos = pos + 1
                cnd = cnd + 1
            else
                count.push(cnd)
                cnd = count[cnd]
                while ((cnd >= 0) and (pat[pos] != pat[cnd])) do
                        cnd = count[cnd]
                end
                pos = pos + 1
                cnd = cnd + 1
            end
        end
        count.push(cnd)
        return count
    end
```

Kompleksitas waktu dari algoritma KMP ini adalah O(m+n), dengan n = panjang *string* teks dan m = panjang *string* pattern

III. Studi Kasus: Pemanfaatan Pencocokan Pola Dalam Membuat Perintah Pada Line Bot

A. Penggunaan Algoritma *Brute Force*

Dengan menggunakan algoritma *Brute Force* untuk



menyelesaikan contoh kasus diatas, waktu yang dibutuhkan adalah 0.09584 ms.

B. Penggunaan Algoritma Knuth-Morris-Pratt

Dengan menggunakan algoritma Knuth-Morris-Pratt untuk menyelesaikan contoh kasus diatas, waktu yang dibutuhkan adalah 0.04839 ms.

C. Penggunaan Algoritma Boyer-Moore

Dengan menggunakan algoritma Boyer-Moore untuk menyelesaikan contoh kasus diatas, waktu yang dibutuhkan adalah 0.03218 ms.

D. Penggunaan *Regular Expression*

Dengan menggunakan *Regular Expression* untuk menyelesaikan contoh kasus diatas, waktu yang dibutuhkan adalah 0.03480 ms.

IV. KESIMPULAN

Dari keempat contoh kasus di atas, dapat dilihat bahwa *Brute Force* menghasilkan waktu yang lebih lama dari ketiga strategi pencarian yang lain, bahkan 2 kali lebih lama. Dengan *string* yang lebih panjang, perbedaan kecepatan dari keempat kasus akan semakin signifikan. Pada *Line Bot* yang mempunyai banyak *chat* yang masuk, akan lebih mudah untuk mencari *keyword* perintah apabila menggunakan strategi pencarian yang lebih cepat.

UCAPAN TERIMA KASIH

Pertama-tama, penulis memanjatkan puji syukur kepada Tuhan yang Maha Esa atas berkat dan rahmat-Nya sehingga penulis dapat mendapatkan inspirasi dan menyelesaikan makalah ini tepat waktu. Penulis juga ingin mengucapkan terima kasih kepada orang tua dan keluarga yang telah mendukung dalam doa dan semangat kepada penulis sehingga dapat mengerjakan makalah ini dengan sepenuh hati. Selain itu, penulis juga ingin menyampaikan terimakasih kepada pengampu mata kuliah IF2211 Strategi Algoritma yaitu Dr. Ir. Rinaldi Munir, M.T., Dr. Nur Ulfa Maulidevi S.T., Dr. Masayu Leylia Khodra M.T., atas bimbingan, referensi, dan ilmu yang telah diajarkan di perkuliahan IF2211 Strategi Algoritma ini. Terakhir, terima kasih kepada rekan-rekan dan pihak lain yang membantu dalam proses pembuatan makalah ini.

REFERENSI

1. https://www.statista.com/statistics/327292/number-of-monthly-active-line-app-users/
2. https://catatananalgo.wordpress.com/2016/10/02/algoritma-string-matching-pencocokan-string/
3. http://www-igm.univ-mlv.fr/~lecroq/string/node2.html
4. http://www.stoimen.com/blog/2012/03/27/computer-algorithms-brute-force-string-matching/
5. https://stackoverflow.com/questions/13175739/what-are-the-shift-rules-for-boyer-moore-string-search-algorithm
6. http://www-igm.univ-mlv.fr/~lecroq/string/node14.html
7. https://www.slideshare.net/thinkphp/string-kmp

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 29 April 2012

Jason Wiguna/13516024