Algoritma Cepat Pencocokkan String

Daniar Heri Kurniawan / 13512064
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13512064@std.itb.ac.id

Abstrak*—Kecepatan dalam pemrosesan sangatlah penting dalam dunia modern yang dipenuhi teknologi canggih seperti saat ini, begitu pula dalam hal pencocokkan *string*. Algoritma pencocokkan *string* yang banyak digunakan saat ini adalah Algoritma *Knuth-Morris-Pratt* (KMP) dan Algoritma *Boyer-Moore* (BM). Dalam makalah ini penulis tidak akan membahas detil algoritma tersebut, tetapi penulis ingin memperkenalkan algoritma pencocokkan *string* yang lebih efisien dibanding kedua algoritma tersebut. Algoritma ini sekilas mirip algoritma *Boyer Moore* karena penulis melakukan pencocokkan *string* dimulai dari karakter terakhir *pattern*. Berbeda dengan BM biasa, penulis menambahkan beberapa fungsi yang dapat memangkas jumlah perbandingannya. Salah satu fungsinya adalah fungsi (*int getLongestPrefSuff (int Idx*)) untuk mencari *substring* terpanjang yang ada pada *matched-string* dan juga ada pada *unmatched-string*. Fungsi tersebut sekilas mirip *border-function* pada KMP, karena penulis memang mencoba mencari keunggulan setiap algoritma kemudian menggabungkannya dalam satu algoritma. Algoritma ini kompleksitasnya O(n+m) dengan n adalah jumlah teks dan m adalah jumlah *pattern*. Dari hasil uji kasus yang penulis lakukan dengan berbagai macam *input* teks dan *pattern*, penulis menyimpulkan bahwa algoritma ini lebih efektif daripada algoritma KMP dan BM. Misalnya pada *worst-case* ketika mencari *pattern* "baaaaa" dalam teks "aaaaaaaaa", algoritma BM membutuhkan 24 kali perbandingan, sedangkan pada algoritma modifikasi penulis, hanya dibutuhkan 6 kali perbandingan untuk memberikan kesimpulan bahwa *pattern* tidak ditemukan dalam teks.

***Kata Kunci—Knuth Morris Pratt Algorithm, Boyer Moore Algorithm, getLongestPrefSuff*.**

I. PENDAHULUAN

Algoritma pencocokkan *string* sangat banyak jenis terapannya dalam kehidupan sehari-hari, misalnya dalam *bio-informatics*. Dalam bidang *bio-informatics*, algoritma pencocokkan *string* dapat digunakan dalam proses pencarian zat-zat / bahan kimia / protein tertentu yang ada pada suatu rantai DNA individu / makhluk hidup. Kompleksitas algoritma yang digunakan sangat berpengaruh pada lama waktu proses dan memori proses yang dibutuhkan. Oleh sebab itu, penelitian / pengembangan algoritma pencocokkan *string* sangat dibutuhkan untuk mendukung kemajuan teknologi yang semakin kompleks ini.

Algoritma pencocokkan *string* ada berbagai macam, dan masing-masing memiliki kelemahan dan kelebihan tersendiri. Walaupun begitu bukan tidak mungkin untuk mengembangkan algoritma baru yang lebih unggul dibanding algoritma sebelumnya. Sejauh ini penulis belum mempelajari lebih jauh algotima pencocokkan *string* selain algoritma *Boyer Moore* dan algoritma *Knuth Morris Pratt* yang keduanya telah diajarkan dalam kuliah Strategi Algoritma IF2120. Dengan tujuan untuk memperoleh algoritma yang lebih efektif dan efisien (mangkus), penulis menggabungkan keunggulan dari kedua algoritma tersebut untuk disatukan dalam satu algoritma. Tentu saja penggabungan tersebut akan menambah kompleksitas algoritma, tetapi dengan hasil penggabungan ini diperoleh algoritma yang mampu membandingkan *string* lebih cepat karena jumlah perbandingan yang harus dilakukan semakin sedikit.

Penulis menyadari bahwa algoritma pencocokkan *string* ini sebenarnya bisa diaplikasikan pada berbagai bidang yang membutuhkan proses pencarian berdasarkan keterurutan indeks pada teks dan pattern. Oleh karena itu penulis menggeneralisir persoalan pencocokkan ini bisa terjadi pada sebuah tuple yang memiliki indeks (*primary key)* dan isi. Sehingga untuk mencari urutan isi yang sesuai dengan *pattern* tidak harus membandingkan isinya, namun bisa membandingkan dengan indeksnya saja. Terlebih lagi jika *cost* yang dibutuhkan untuk membandingkan isi jauh lebih besar daripada membandingkan indeksnya. Untuk mengakomodasi kemungkinan tersebut, penulis menerapkan kondisi pada algoritma baru ini sehingga tidak ada pembandingan indeks yang sama terjadi dua kali.

II. TEORI DASAR

2.1 Algoritma Pencocokkan *String*

Algoritma pencocokkan *string* adalah algoritma yang digunakan untuk mencari / menemukan sebuah *pattern* pada teks. *String* merupakan kumpulan karakter, baik huruf, angka, tanda baca, dan sebagainya yang berada pada sebuah urutan indeks. Seperti yang telah penulis tekankan dibagian pendahuluan, algoritma pencocokkan *string* ini sebenarnya tidak terpaku pada elemen string saja, tetapi bisa pula dimanfaatkan untuk melakukan pencocokkan pada *array of tuple* yang memililki indeks sesuai isi *tuplenya*. Untuk selanjutnya penulis akan menggunakan istilah teks sebagai *string input* yang mungkin mengandung *pattern* dan istilah *pattern* sebagai *string* yang ingin dicari keberadaannya pada suatu teks. Misalnya:

Teks : “mencoba algoritma *string matching* yang baru”
Pattern : “baru”

Jika algoritma pencocokkan *string* diterapkan dengan *input* seperti diatas, maka hasil dari proses akan menyimpulkan bahwa *pattern* ditemukan pada teks. Informasi lebih detailnya, *pattern* ditemukan pada teks di indeks ke-40 (relatif terhadap 1).

2.2 Algoritma *Boyer Moore*

Algoritma *Boyer Moore* (BM) merupakan algoritma pencocokkan *string* yang memanfaatkan 2 teknik utama, yaitu : 1) *looking-glas*s teknik dan 2) *Character-jump. Looking-glass* teknik adalah cara menemukan *pattern* dalam teks dengan melakukan perbandingan mulai dari karakter paling belakang pada *pattern.* Sedangkan *character-jump* adalah pengambilan keputusan untuk menggeser indeks sesuai kondisi yang memenuhi. Ada 3 kondisi yang bisa menyebabkan *character-jump*, ketiga kondisi tersebut akan terjadi ketika hasil perbandingan antara karakter pada indeks *pattern* berbeda dengan karakter pada indeks teks. Untuk memproses setiap kondisi tersebut, algoritma boyer moore harus memanggil fungsi *lastOccurence()* terlebih dulu untuk mengetahui letak karakter terakhir suatu karakter pada *pattern.*

Teks : “ ???**x**ab????”
Pattern: ”??**c**ab??”

✓ Kondisi 1
Ketika terjadi ketidakcocokkan, yaitu antara “x” dan “c”, fungsi *lastOccurence* mengembalikan indeks dimana “x” ditemukan pada indeks sebelah kiri huruf “c” pada *pattern.* Maka *pattern* akan digeser sedemikian sehingga posisi “x” pada teks bersesuaian dengan “x” pada *pattern.*
Teks : “ ???**x**ab????”
Pattern: ” **x**?**c**ab??”

✓ Kondisi 2
Ketika terjadi ketidakcocokkan, yaitu antara “x” dan “c”, fungsi *lastOccurence* menemukan indeks dimana “x” bisa ditemukan pada indeks sebelah kanan huruf “c” pada *pattern*. Maka *pattern* digeser ke kanan satu karakter.
Teks : “ ???**x**ab????”
Pattern: ” ??**c**ab**x**?”

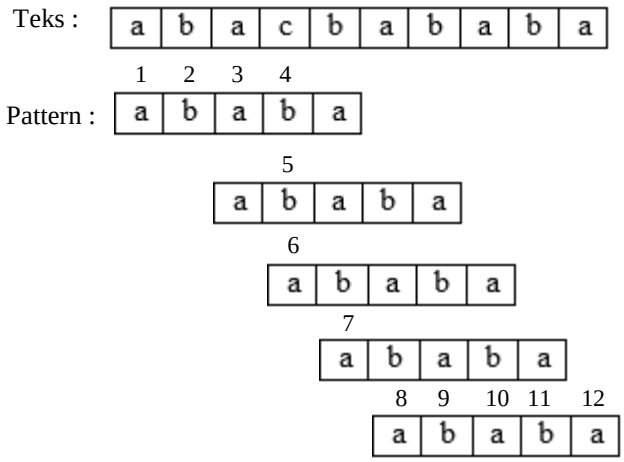
✓ Kondisi 3
Ketika 2 kondisi di atas tidak memenuhi, maka kondisi 3 diberlakukan, yaitu melakukan pergeseran sepanjang sisa karakter pada *pattern* yang belum dicocokkan.
Teks : “ ???**x**ab????”
Pattern: ”??cab??”

Fungsi *lastOccurence* dari *pattern* “aacabda” adalah sesuai tabel berikut :

x	a	b	c	d
L(x)	7	5	3	6

2.3 Algoritma *Knuth Morris Pratt*

Algoritma ini adalah pengembangan dari algoritma *Brute-force* dengan melakukan efisiensi perpindahan yang memanfaatkan fungsi pinggiran (*border function*). Algoritma ini memulai pencocokkan dari indeks *pattern* terdepan kemudian ke belakang. Ketika terjadi ketidak-cocokkan, maka akan dilakukan penggeseran *pattern.* Fungsi pinggiran adalah fungsi yang mengembalikan jumlah *suffix* terpanjang yang juga sebagai *prefix* pada suatu *matched-pattern (substring* dari *pattern).* Hasil dari fungsi pinggiran tersebut akan digunakan untuk memperhitungkan jumlah penggeseran indeks *pattern* relatif terhadap teks. Jumlah pergeserannya adalah panjang *matched string* dikurangi hasil fungsi pinggirannya. Misalnya :



Fungsi pinggiran dari *pattern* “ababa”, seperti pada tabel berikut :

J	1	2	3	4	5
P(j)	a	b	a	b	a
B(j)	0	0	1	2	3

2.4 Terapan Algoritma Pencocokkan *String*

Dalam bidang Ilmu Pengetahuan dan Teknologi banyak ditemukan aplikasi yang selalu dilengkapi dengan kemampuan *find* / menemukan suatu kata, misalnya pada teks editor, teks *viewer*, hingga *search engine*. Selain itu, pada bidang *bio-informatics* juga mulai banyak digunakan untuk memaksimalkan automatisasi sebuah alat, misalnya pada analisis citra sidik jari, dan pencarian rangkaian protein pada rantai DNA.

III. ALGORITMA PENCOCOKKAN CEPAT

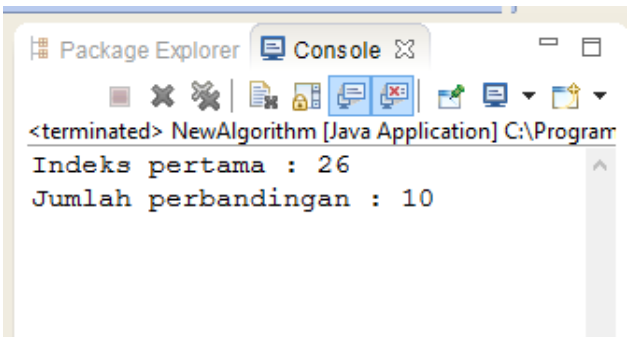
Dalam bab ini akan penulis uraikan secara detail algoritma baru yang telah dituliskan dalam bahasa *java*.

Penulis lebih memilih untuk menggunakan *true-code* daripada *pseudo-code* agar lebih mudah dipahami dan untuk mengurangi kesalahan penafsiran. Namun untuk mempersingkat penjelasan, penulis juga akan menerapkan *pseudo-code* pada beberapa bagian algoritma. Algoritma ini akan mengembalikan sebuah bilangan *integer* yang menandakan indeks awal *pattern* ditemukan pada teks. Jika hasil kembaliannya adalah -1, maka *pattern* tersebut tidak bisa ditemukan dalam teks.

A. Source Code

```
public class NewAlgorithm {

  private String Pattern = new String();
  private String Text = new String();
  private Vector<Integer> border =
        new Vector<Integer>();
  public int IdxMatch; /* relatif thd nol */
  public int numberOfCompare;

  /*  konstruktor */
  public NewAlgorithm() {
    IdxMatch = -1;
    numberOfCompare = 0;
  }

  /*  memasukkan pattern  */
  public void setPattern(String myString) {
    Pattern = myString;
    createBorderFunction();
  }

  /*  memasukkan teks  */
  public void setText(String myString) {
    Text = myString;
  }

  /* membuat border function  */
  public void createBorderFunction() {
     for (int i = Pattern.length() - 1; i > 0; i--) {
          String pref = Pattern.substring(i,
               Pattern.length());
          if (Pattern.matches("^" + pref + "\\w*")) {
               border.add(Pattern.length() - i);
               System.out.println(Pattern.length() - i);
          }
     }

  /*  mencari prefix matched string terpanjang yang
sama dengan suffix unmatched string */
  private int getLongestPrefSuff(int Idx) {
    int NumberOfMatch = Pattern.length() - Idx - 1;
    int i = 0;
    while (i < border.size()) {
      int length = border.elementAt(i);
      if (length <= NumberOfMatch) {
        return length;
      }
      i++;
    }
    return 0;
  }

  /*  mendapatkan index kemunculan terakhir sebuah
karakter */
  private int getIdxLastOccurence(String myString,
      char myChar) {
    for (int i = myString.length() - 1; i >= 0; i--) {
      if (myString.charAt(i) == myChar)
        return i;
    }
    return -1;
  }

  /*  memulai menjalankan algoritma */
  public void startNewAlgorithm () {
    if (Pattern.length() <= Text.length()) {
      /* Panjang pattern < panjang text*/
      int i = Pattern.length() - 1; /*  iterator untuk text */
      int j = Pattern.length()-1;/*iterator untuk pattern*/
      int length = 0;
      int markIdx = -99;
      boolean match = false;
      while (i < Text.length() && !match) {
        if (Text.charAt(i) != Pattern.charAt(j)) {
          length = getLongestPrefSuff(j);
          String subPattern =
               Pattern.substring(length, j + 1);

          int idxChar =
               getIdxLastOccurence(subPattern,
               Text.charAt(i));

          if (idxChar != -1) { /* char ada di pattern */
            idxChar += length; /* ditambah marker */
            markIdx = i;/*menandai idx yang pasti sama */
            i += Pattern.length() - (idxChar + 1);
            j = Pattern.length() - 1;
            /* mengupdate panjang LongestPrefSuff */
            length = getLongestPrefSuff(idxChar);
          } else { /* char tidak ada pada sisa pattern */
            /*menghilangkan tanda idx yang pasti sama */
            markIdx = -99;
            int NumberOfMatch = Pattern.length() - j - 1;

            /*merubah i */
            if (length != 0) {
              i = Pattern.length() +
                i + NumberOfMatch - length;
```

```
            } else
              i = Pattern.length() +
                i + NumberOfMatch;
            /*merubah j */
            j = Pattern.length() - 1;
          }
        } else {
          j--;
          i--;
        }
        if (i == markIdx){
         /*tidak akan membandingkan 2 kali */
          i--;
          j--;
        }
        if (j == (-1 + length)) {
          match = true;
          IdxMatch = i + 1 - length;
        }
        numberOfCompare++;
      }
    }
  }

  /* program utama */
  public static void main(String[] args) {
    NewAlgorithm myAlgo = new NewAlgorithm();
    myAlgo.setText("algoritma string matching baru");
    myAlgo.setPattern("baru");
    myAlgo.startNewAlgorithm();

    int idx = myAlgo.IdxMatch;
    int N = myAlgo.numberOfCompare;
    System.out.println("Indeks pertama : " +idx);
    System.out.println("Jumlah perbandingan : " + N);
  }
}
```



Gambar 1 : hasil eksekusi program

B. Penjelasan Program

Sebelum penulis menjelaskan lebih jauh tentang algoritma ini, penulis ingin menjelaskan istilah-istilah yang nanti akan dipakai dalam penjelasan program.

Misal :
Text : "abacaabaccabacabaabb"
Pattern : "abacab"

abacaabaccabacabaabb
abacab

- ✓ *Matched-string*
  Jumlah *string* pada *pattern* yang sudah dicocokkan dan sesuai dengan teks. Dalam algoritma ini perbandingan dimulai dari indeks terakhir pada *pattern* sehingga pada kasus di atas, "cab" adalah *matched-string* dengan panjangnya 3. *Matched-string* selalu mengandung *suffix pattern.*
- ✓ *Unmatched-string*
  Jumlah *string* pada *pattern* yang tidak sesuai dengan teks. Dalam algoritma ini perbandingan dimulai dari indeks terakhir pada *pattern* sehingga pada kasus di atas, "aba" adalah *unmatched-string* dengan panjangnya 3. *Unmatched-string* selalu mengandung *prefix pattern.*
- ✓ *Suffix*
  Berdasarkan *pattern* di atas, maka *suffix*nya :
  b
  ab
  cab
  acab
  bacab
- ✓ *Prefix*
  Berdasarkan *pattern* di atas, maka *prefix*nya :
  a
  ab
  aba
  abac
  abaca
- ✓ Fase
  Setiap satu kali pencocokkan *string* tanpa pergeseran *pattern*, penulis menganggapnya sebagai satu fase. Misalkan pada contoh berikut urutan perbandingan ditandai dengan sebuah *integer* dan ketika beralih fase ditandai dengan penulisan pada baris yang berbeda, maka :

  Teks: abacaabacaabcaabaabb
  ^{1}abcaab =>Fase 1
  ^{2}abcaab =>Fase 2
- ✓ *LongestPrefSuff*
  Indeks : 1 2 3 4 5 6 7 8
  Teks : a a d x a b a f
  Pattern : **a b** b c **a b**
  CurrentIdx : 4
  Pada *Pattern* dengan kondisi tersebut, *LongestPrefSuff* adalah "**ab**" yang merupakan

suffix dari *matched-string*, namun juga *prefix* dari *unmatched-string.*

a. Fungsi ***createBorderFunction*** **( )**
Berbeda dengan KMP, *border function* pada algoritma ini akan mencari panjang *suffix* dari teks yang juga *prefix* dari teks tersebut. Kemudian fungsi tersebut akan menyimpan hasilnya pada sebuah *array of integer* terurut membesar. Hal tersebut akan berguna untuk pemrosesan pada fungsi selanjutnya.

```
/*
   * function createBorderFunction() -> array of int
   *
   * KAMUS    :
   *       int [] Border ;
   *
   * ALGORITMA :
   * for (semua suffix pada pattern){
   *       if (suffix ke-i juga berupa preffix) then
   *              Border.add(panjang suffix tersebut);
   * }
   * return ArrInt;
*/
```

b. Fungsi *getLongestPrefSuff* **( )**
Fungsi ini memanfaatkan hasil dari pemrosesan fungsi *createBorderFunction*(). Dengan adanya fungsi ini, program akan tahu berapa panjang terbesar *suffix* dari *matched-string* yang juga sebagai *prefix* dari *unmatched-string*. Fungsi ini mendapat *input* parameter berupa indeks *pattern* yang terjadi ketidakcocokkan dengan teks. Sehingga tiap kali terjadi ketidakcocokkan, fungsi ini akan dipanggil dengan parameter yang bervariasi sesuai terjadinya ketidakcocokkan. Dengan mengetahui indeks ketidakcocokkan, fungsi ini akan mengetahui panjang *matched-string* dan *unmatched-string*. Hasil dari fungsi ini untuk selanjutnya memakai istilah LongestPrefSuff (akan dijelaskan kemudian).

```
/*
   * function getLongestPrefSuff(int Idx) -> integer
   * KAMUS :
   * Int NumberOfMatch
   *
   * ALGORITMA :
   * for ( semua Border ke-i yang terurut mengecil)
   *      if (panjang Border ke-i <= NumberOfMatch ) then
   *              return panjangnya ;
   *      }
   * }
   * return 0;
*/
```

c. Fungsi *getIdxLastOccurence* **( )**
Fungsi ini mirip seperti pada algoritma *Boyer Moore* yang akan mengembalikan sebuah *integer* berupa indeks tempat kemunculan terakhir suatu karakter pada *unmatched - string* (definisi bisa dilihat di penjelasan sebelumnya). Jika karakter tidak ditemukan pada *unmatched - string*, maka fungsi ini akan mengembalikan -1. Sebagai catatan untuk diperhatikan, indeks yang dikembalikan adalah indeks relatif terhadap *unmatched-string*. Sehingga ketika akan mencari indeks yang sesuai dengan *pattern*, hasil dari fungsi ini harus dijumlahkan dulu dengan panjang *LongestPrefSuff*.

```
/* function getIdxLastOccurence(String myString,
   *  char myChar) -> Integer
   *
   * KAMUS :
   *    integer i ;
   * ALGORITMA :
   *    for (setiap char ke - i yang ada di myString dimulai dari
   *     belakang)
   *         if ( char ke - i sama dengan myChar) then
   *             return i;
   *          endif
   *    endfor
   * return -1;
*/
```

d. Fungsi utama *startNewAlgorithm* **( )**
Fungsi ini yang akan menyatukan semua fungsi di atas menjadi satu kesatuan algoritma yang berbeda dari KMP maupun BM. Penulis tidak menjelaskan fungsi utama ini dalam notasi *pseudo code*, karena akan lebih panjang dan bisa menimbulkan banyak interpretasi yang berbeda-beda. Fungsi ini akan penulis paparkan dalam rangkaian pernyataan , sebagai berikut :
(misalkan i adalah *iterator* untuk teks dan j adalah *iterator* untuk *pattern*)

- ✓ Fungsi akan mengembalikan -1 jika tidak menemukan *pattern* pada teks, atau mengembalikan indeks tempat karakter pertama *pattern* ditemukan pada teks.
- ✓ Awal kondisi yang harus dipenuhi oleh fungsi ini sebelum melakukan pemrosesan adalah panjang *pattern* kurang dari atau sama dengan panjang *text.*
- ✓ Kondisi berikutnya adalah ketika kedua karakter pada indeks i dan j cocok, maka i dan j yang berfungsi sebagai *iterator* diturunkan nilainya dengan mengurangkannya dengan 1.

```
/*
 * i-- ;
 * j-- ;
 */
```

- ✓ Jika karakter ke - i pada teks tidak cocok dengan karakter ke - j pada *pattern*, maka ada 2 kondisi yang berlaku.

Kondisi 1 :
(karakter ada di *unmatched-string*)

```
Indeks  : 1 2 3 4 5 6 7 8
Teks    : a a c b a c a f
Pattern :   c a b
```

Ketika kondisi seperti ini maka *pattern* akan digeser agar karakter pada teks dan *pattern* bisa bersesuaian . Seperti pada contoh berikut :

Teks : a a c b a c a f
Pattern : c a b

Yang membedakan proses pergeseran pada algoritma ini dengan algoritma Boyer Moore yaitu adanya *update* untuk mendapatkan *LongestPrefSuff* sehingga pada fase berikutnya tidak membandingkan ulang bagian *string* yang sudah sama. Selain itu, *iterator* teks tempat terjadinya ketidakcocokkan juga ditandai ke dalam *markId* karena ketika pergeseran dilakukan pasti menghasilkan karakter yang cocok pada indeks teks tersebut. Perhatikan contoh berikut :

Indeks : 1 2 3 4 5 6 7 8
Teks : a a x a b a f
Pattern : x c a b
matched-string : ab
markIdx : 3
Fase berikutnya : a a x a b a f
x c a b
(indeks pada 'x' sudah pasti sama)

Kondisi 2 :

(karakter tidak ada di *unmatched-string*)

Indeks : 1 2 3 4 5 6 7 8
Teks : a a b b a c a f
Pattern : c a b

Pada kondisi inilah kemampuan sebuah algoritma sangat penting untuk dievaluasi. Ketika terjadi ketidakcocokkan seperti ini yang pertama harus dilakukan adalah mengisi nilai pada *markId* dengan -99 (Indeks tidak terdefinisi) karena pada keadaan ini terjadi perubahan fase (lihat definisi sebelumnya) yang mengakibatkan *markId* tidak mungkin digunakan.

Pada algoritma ini terdapat hal yang berbeda dari algoritma Boyer Moore ketika *pattern* memiliki *LongestPrefSuff* yang panjangnya > 0. Algoritma ini akan mereduksi *unmatched-string* menjadi *unmatched-string* dikurangi *LongestPref-Suff* tersebut. Sehingga ketika memanggil fungsi *getIdxLastOccurence*, pencarian tidak akan dilakukan pada karakter yang termasuk *LongestPrefSuff*. Contoh :

Indeks : 1 2 3 4 5 6 7 8
Teks : a a d a a b a f
Pattern : a b b c a b
CurrentIdx : 4
LongestPrefSuff : ab
unmatched-string : b

Pada contoh tersebut, karakter 'a' akan dicari pada *unmatched string* dan hasilnya -1 karena 'a' tidak ditemukan pada *unmatched-string*. Sehingga pergeseran yang dilakukan pada fase berikutnya adalah sbb:

Teks : a a d a a b b c a b
3 2 1
a b b c a b
7 6 5 4
a b b c a b

Mengacu pada pergeseran seperti contoh di atas, algoritma ini juga memanfaatkan *LongestPrefSuff*. Sehingga ketika berpindah fase, *pattern* akan bergeser dan *LongestPrefSuff* ("ab") bisa saling bersesuaian indeksnya. Kemudian jika tidak memiliki *LongestPrefSuff*, maka proses perbandingan akan dilakukan dengan menggeser *pattern* sejauh panjang *pattern* tersebut. Agar lebih jelas, perhatikan contoh berikut :

Teks : a a d a a b a c b c a b
3 2 1
c c b c a b
9 8 7 6 5 4
c c b c a b

Satu lagi yang penting mengenai *LongestPrefSuff*, ketika *LongestPrefSuff* panjangnya > 0 , *pattern* tidak perlu membandingkan keseluruhan karakter yang ada di *pattern*. *Pattern* cukup membandingkan karakter yang belum pernah dibandingkan. Pada contoh diatas, *pattern* dicocokkan dengan 7 perbandingan. Sehingga jika indeks kecocokkan pada *pattern* sudah mencapai panjang *LongestPrefSuff* + 1, maka *pattern* dan teks di nyatakan cocok atau *match*. Kondisi itu ada pada bagian algoritma berikut :

```
/*
 * if (j == (-1 + length)) {
 *    match = true;
 *    IdxMatch = i + 1 - length;
 * }
*/
```

- ✓ Kondisi berakhirnya algoritma ini ada dua macam, sebagian sudah disinggung pada penjelasan sebelumnya. Pertama ketika *iterator* i besarnya melebihi panjang teks dan yang kedua ketika terjadi kecocokkan antara *pattern* dengan teks.
- ✓ Algoritma ini juga menangani kondisi ketika *iterator* i *(iterator* untuk teks) sama dengan *markId* (lihat definisi pada penjelasan sebelumnya), *iterator* i dan j nilainya langsung di turunkan 1 indeks tanpa perlu membandingkan karakter lagi. Hal ini dilakukan karena kita tahu karakter yang ada pada *markId* pasti bernilai sama.

C. Proses Pencocokkan sesuai *Test Case*

Pada bagian ini penulis akan memberikan contoh lebih jelas tentang pencocokkan *string* menggunakan algoritma penulis. Pada contoh berikut akan diberikan jumlah pencocokkan atau perbandingan dengan penanda angka di

atas *pattern* dan ketika *pattern* berpindah fase akan ditandai dengan berpindahnya baris penulisan *pattern*. Ada 6 contoh teks beserta *pattern*nya yang akan penulis gambarkan urutan pembandingannya, sebagai berikut :

- Teks : "a *pattern matching algorithm*" (BM membutuhkan 11 kali perbandingan)

a		p	a	t	t	e	r	n		m	a	t	c	h	i	n	g		a	l	g	o	r	i	t	h	m
				1																							
r	i	t	h	m																							
								2																			
					r	i	t	h	m																		
														3													
										r	i	t	h	m													
															4												
											r	i	t	h	m												
																	5										
														r	i	t	h	m									
																							6				
																			r	i	t	h	m				
																								10	9	8	7
																								r	i	t	h	m

Jumlah perbandingan = 10
match = true

- Teks : "abacaabadcabacabbaabb" (BM membutuhkan 13 kali perbandingan)

a	b	a	c	a	a	b	a	d	c	a	b	a	c	a	b	b	a	a	b	b
					1															
a	b	a	c	a	b															
				3		2														
	a	b	a	c	a	b														
										4										
					a	b	a	c	a	b										
								7	6		5									
						a	b	a	c	a	b									
												11	10	9	8					
										a	b	a	c	a	b					

Jumlah perbandingan = 11
match = true

- Teks : "aaaaaaaaa" (BM membutuhkan 24 kali perbandingan)

a	a	a	a	a	a	a	a	a			
6	5	4	3	2	1						
b	a	a	a	a	a						
						b	a	a	a	a	a

Jumlah perbandingan = 6
match = false

- Teks : "afcabctabcabaxadcba"
  Perhatikan : pada fase terakhir terdapat *markId* dan juga *LongestPrefSuff*.

a	f	c	a	b	c	t	a	b	c	a	b	a	x	a	d	c	b	a
					1													
a	b	c	a	b	a													
								2										
			a	b	c	a	b	a										
									3									
				a	b	c	a	b	a									
								7		6	5	4						
							a	b	c	a	b	a						

Jumlah perbandingan = 7
match = true

- Teks : "aadaabbbcbb"

a	a	d	a	a	b	b	b	c	b	b
				2	1					
b	b	b	c	b	b					
						7	6	5	4	3
					b	b	b	c	b	b

Jumlah perbandingan = 7
match = true

- Teks : "abacaabacaabcaabaabb"

a	b	a	c	a	a	b	a	c	a	a	b	c	a	a	b	a	a	b	b		
				1																	
a	b	c	a	b																	
					2																
	a	b	c	a	b																
				4		3															
		a	b	c	a	b															
									5												
					a	b	c	a	b												
										6											
					a	b	c	a	b												
								8		7											
						a	b	c	a	b											
													9								
									a	b	c	a	b								
												11		10							
										a	b	c	a	b							
															14	13	12				
													a	b	c	a	b				
																a	b	c	a	b	

Jumlah perbandingan = 14
match = false

IV. HASIL DAN ANALISIS

Berdasarkan uji *test case* yang penulis lakukan, hasilnya menunjukkan bahwa algoritma ini tidak lebih buruk dari algoritma *Boyer Moore* maupun *Knuth Morris Pratt.* Penulis beranggapan demikian karena penulis belum menguji dengan pengujian yang terstruktur. Sejauh ini penulis masih menguji dengan sampel random dan hasil dari pengujian itu menunjukkan bahwa algoritma ini lebih baik dari segi minimalitas jumlah perbandingan. Penulis merancang algoritma ini dengan mempertimbangkan kompleksitas fungsi - fungsi tambahan yang harus di panggil. Jika diperhatikan dengan seksama, sebenarnya fungsi-fungsi tersebut hanya *meretrieve* atau mengambil data saja, kemudian memanipulasi data tersebut dengan operasi sederhana hingga mampu menghasilkan kondisi-kondisi yang sudah dijelaskan penulis di bagian penjelasan program.

Seperti algoritma KMP dan BM, beberapa fungsi harus memasukkan nilai terlebih dahulu pada sebuah variabel berdasarkan *pattern* yang diberikan sebelum pemrosesan dilakukan. Fungsi tersebut kompleksitasnya masih termasuk ke dalam fungsi *polinomial*, sehingga penulis menyimpulkan bahwa algoritma ini tidak memiliki permasalahan dalam hal kompleksitas jika dibandingkan dengan algoritma *Boyer Moore* maupun *Knuth Morris Pratt*. Algoritma ini menerapkan proses penanganan kasus yang sebenarnya merupakan modifikasi penulis dari algoritma BM dan KMP. Penulis memaksimalkan setiap keunggulan masing-masing algoritma tersebut dalam satu algoritma ini.

V. KESIMPULAN

Algoritma pencocokkan *string* yang sudah penulis pelajari pada mata kuliah Strategi Algoritma IF2211 (Institut Teknologi Bandung) memberi ide penulis untuk merancang algoritma ini. Algoritma ini memanfaatkan *LongestPrefSuff* untuk mengambil keuntungan dari adanya

kesamaan urutan karakter pada *pattern*. *LongestPrefSuff* adalah rangkaian karakter pada *suffix matched-string* yang juga *prefix unmatched-string* (istilah tersebut dijelaskan di bagian awal). Dengan mengombinasikan fungsi *getLongestPrefSuff*, *getIdxLastOccurence*, dan operasi *substring* , penulis telah menguji dengan random sampel bahwa jumlah perbandingan yang dibutuhkan algoritma ini lebih sedikit atau sama dengan jumlah perbandingan yang dibutuhkan oleh algoritma KMP dan BM untuk kasus uji (teks dan *pattern*) yang sama. Algoritma ini bisa dikembangkan lebih lanjut sehingga bisa diaplikasikan pada bidang-bidang lain, misalnya pada bidang *bio-informatics*. Dengan logika yang sudah penulis uraikan di bagian penjelasan, algoritma ini juga bisa diterapkan dalam bahasa pemrograman apapun, tidak terbatas pada program java seperti yang digunakan penulis.

VI. UCAPAN TERIMA KASIH

Alhamdulillah, terima kasih kepada Allah SWT yang telah memberikan rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan makalah ini tepat pada waktunya. Tidak lupa penulis mengucapkan terima kasih kepada Dr. Rinaldi Munir selaku dosen mata kuliah Strategi Algoritma yang telah mengajarkan banyak ilmunya kepada penulis. Selain itu penulis juga mengucapkan terima kasih kepada orang tua penulis dan semua pihak, yang tidak bisa penulis sebutkan satu persatu, yang telah membantu penulis dengan bantuan moril maupun materil.

REFERENCES

[1] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2013-2014-genap/Pencocokkan%20String%20(2014).ppt diakses tanggal 18 Mei 2014 , 10:21 AM

PERNYATAAN

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

Bandung, 18 Mei 2014

Daniar Heri Kurniawan / 13512064