Pemanfaatan Algoritma Pencocokan String dalam Pencarian Kata Tidak Baku pada KBBI

Daffa Pratama Putra, 13518033
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13518033@std.stei.itb.ac.id

***Abstraksi*—KBBI atau Kamus Besar Bahasa Indonesia adalah kamus umum yang menjadi acuan tertinggi dalam penulisan bahasa Indonesia yang baku. Kamus ini sering digunakan untuk mencari arti kata, namun tidak jarang juga digunakan untuk mencari kata baku yang tepat. Sering terjadi kesalahan penulisan kata yang menyebabkan kata menjadi tidak baku. Oleh karena itu makalah ini akan membahas pencarian kata baku pada KBBI dengan pendekatan algoritma pencocokan *string*.**

Kata Kunci—Algoritma, Kata, String, KBBI

I. PENDAHULUAN

Sejak zaman dulu, bahasa menjadi alat berkomunikasi yang paling efektif. Terbukti dari masa ke masa sejak bahasa pertama kali ditemukan, perkembangannya menjadi sangat pesat. Manusia dapat berbahasa karena adanya penyebaran bahasa secara dari mulut ke mulut maupun tulisan. Bahasa Indonesia sendiri nyatanya bukan bahasa yang asli, tetapi perpaduan berbagai bahasa, salah satu penyumbang terbesar terbentuknya bahasa Indonesia adalah bahasa Melayu.

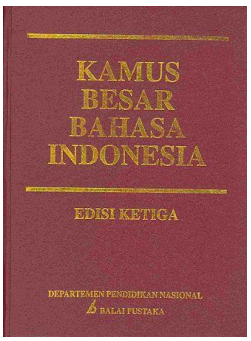
Meskipun terbukti bahasa Indonesia kini banyak digunakan, namun penutur asli bahasa Indonesia, yakni orang Indonesia sendiri belum berbahasa Indonesia dengan baik dan benar, terutama dalam penulisan. Tidak jarang ditemukan kesalahan dalam penulisan berbahasa Indonesia. Salah satu kesalahan yang sering ditemukan dalam tulisan berbahasa Indonesia salah satunya adalah penggunaan kata tidak baku. Penggunaan kata tidak baku pada teks formal muncul karena kebiasaan menggunakan kata yang tidak baku dalam percakapan sehari-hari atau informal, sehingga seringkali tercampur antara kata yang baku dan tidak baku.

Bahasa Indonesia sendiri memiliki kamus umum bernama Kamus Besar Bahasa Indonesia atau biasa disebut dengan KBBI. Layaknya kamus pada umumnya, KBBI berisi arti dari semua kata bahasa Indonesia. KBBI umumnya digunakan untuk membantu menemukan arti kata bahasa Indonesia, namun banyak juga digunakan untuk mencari kata baku yang tepat. Kata baku biasanya digunakan dalam teks dan percakapan formal. Penggunaan kata tidak baku akan berakibat fatal jika orang yang membaca ataupun mendengar kata tersebut menangkap makna yang berbeda. Kata tidak baku yang sering digunakan antara lain, “Stop”, “Ijin”, “Jumat”, dan “Kenapa”.

II. DASAR TEORI

A. *Kamus Besar Bahasa Indonesia (KBBI)*

Kamus Besar Bahasa Indonesia atau KBBI adalah kamus umum resmi yang menjadi acuan tertinggi berbahasa Indonesia yang baku. KBBI disusun oleh Badan Pengembangan dan Pembinaan Bahasa, diterbitkan oleh Balai Pustaka, dan dinaugi oleh Kementrian Pendidikan dan Kebudayaan Indonesia. KBBI ini menjadi kamus terlengkap dan terakurat yang pernah diterbitkan, sehingga kata bahasa Indoesia apapun dapat ditemukan di dalam KBBI.



Gambar 1 : Cetakan resmi KBBI edisi ketiga (Sumber : https://id.wikipedia.org/wiki/Kamus_Besar_Bahasa_Indonesia diakses pada 2 Mei 2020)

KBBI pertama kali diterbitkan pada tahun 1988 dan telah memiliki 62.100 lema. Lema adalah kata atau frasa masukan yang terdapat di dalam kamus. Edisi pertama ini merupakan pengembangan dari Kamus Bahasa Indonesia yang terbit 5 tahun sebelumnya. Sampai makalah ini ditulis, KBBI telah memiliki lima edisi. Setiap edisi yang diterbitkan, terjadi penambahan lema. Seperti pada edisi keempat, KBBI diperbanyak dengan kosakata dari kamus istilah, namun perlu diingat bahwa KBBI bukan kamus istilah. Edisi terbaru diterbitkan pada tahun 2016 oleh Menteri Pendidikan dan Kebudayaan saat itu, Muhadjir Effendy. Edisi kelima ini memiliki 108.000 lema dengan versi cetaknya setebal 2.040 halaman.

B. String

String pada bahasa pemrograman adalah sekumpulan barisan karakter. Karakter yang dimaksud dapat berupa angka, huruf, maupun simbol-simbol lainnya, seperti titik, koma, atau bahkan tanda pagar "#". Beberapa bahasa pemrograman tidak menyediakan tipe data String, namun dapat diasosiasikan dengan *array of char*. Misalkan pada bahasa C, inisialisasi *string* dituliskan dengan :

Char variabel[maksimum karakter];

String dapat memiliki makna atau hanya sekadar barisan karakter saja. Contoh *string* yang memiliki makna adalah kalimat atau kata. Kata "INFORMATIKA" adalah sebuah *string* dengan panjang 11. Seperti yang telah dijelaskan sebelumnya, bahwa *string* dapat diasosiasikan dengan *array of char* yang berarti kata "INFORMATIKA" merupakan barisan karakter sebagai berikut :

S : INFORMATIKA

i	0	1	2	3	4	5	6	7	8	9	10
S[i]	I	N	F	O	R	M	A	T	I	K	A

C. Algoritma Pencocokan String

1. Algoritma Brute Force

Algoritma Brute Force adalah algoritma yang paling umum digunakan untuk memecahkan suatu masalah. Algoritma Brute Force melakukan pendekatan yang lempeng atau *straight forward*. Dilakukan secara lempeng karena algoritma Brute Force ini memecahkan masalah dengan sangat sederhana, langsung, dan jelas caranya. Oleh karena itu algoritma Brute Force sering disebut dengan algoritma naif (*Naïve Algorithm*). Algoritma Brute Force sebenarnya bukanlah algoritma yang cerdas dan mangkus. Meskipun begitu, hampir semua persoalan dapat diselesaikan dengan algoritma Brute Force.

a) Pencocokan String

Persoalan pencocokan *string* merupakan salah satu persoalan yang dapat diselesaikan dengan pendekatan algoritma Brute Force. Pada algoritma Brute Force, *string* pada *pattern* akan dicocokkan satu persatu dengan teks dari kiri ke kanan. Sehingga setiap karakter pada teks akan dicocokkan satu persatu dengan karakter pada *pattern*. Berikut adalah contoh pencocokan *string* dengan algoritma Brute Force :

Misalkan diberikan teks T dan *pattern* P sebagai berikut :

T :

i	0	1	2	3	4	5	6	7	8
T[i]	D	E	C	R	Y	P	T	1	8

P :

j	0	1	2	3	4
P[j]	C	R	I	P	T

Pencocokan *string* dimulai dari indeks i ke-0 dan indeks j ke-0. Jika karakter *T[i]* dan *P[j]* ternyata cocok, maka indeks i dan j bertambah 1. Tetapi jika terjadi *mismatch* antara *T[i]* dan *P[j]*, maka indeks i bertambah 1 namun indeks j menjadi 0. Proses ini dilakukan satu persatu karakter hingga ditemukan *pattern* yang sesuai pada teks atau teks sudah mencapai akhir pencocokan.

Untuk persoalan pencocokan *string*, algoritma Brute Force memiliki kompleksitas waktu sebesar *O*(n) untuk kasus terbaik, yaitu bila karakter *pattern* pertama tidak pernah sama dengan karakter teks yang dicocokkan, sebesar *O*(mn) untuk kasus terburuk, yaitu bila karakter *pattern* yang dicocokkan terakhir terjadi *mismatch*, dan sebesar *O*(m+n) untuk kasus rata-rata, yaitu pada pencarian teks asli.

2. Algoritma Knuth-Morris-Pratt

Algoritma Knuth-Morris-Pratt atau algoritma KMP adalah salah satu algoritma pencocokan *string* yang dinilai lebih baik dari algoritma Brute-Force. Algoritma KMP dikembangkan oleh *computer scientits* bernama Donald E. Knuth, James H. Morris, dan Vaughan R. Pratt. Nama Knuth-Morris-Pratt merupakan gabungan dari ketiga pencipta algoritma ini.

Pada dasarnya algoritma ini mirip seperti algoritma Brute-Force dalam melakukan pencocokan *string*, yaitu pencocokan dari kiri ke kanan *string*. Namun yang membedakan adalah pergeseran yang dilakukan ketika mencocokkan. Pergeseran yang dilakukan algoritma KMP mempertimbangkan kesamaan *substring* pada prefiks dan sufiks dari *pattern*. Oleh karena itu diperlukan *pre-processing* untuk menentukan jumlah pergeseran yang dilakukan untuk setiap indeks pada *pattern*. *Pre-processing* ini disebut dengan fungsi pinggiran.

a) Fungsi Pinggiran (Border Function)

Fungsi pinggiran pada algoritma KMP disebut juga dengan *border function* atau *failure function*. Fungsi ini menghitung jumlah pergeseran yang dilakukan dengan menghitung panjang terbesar *substring* pada prefiks dari *pattern*, P[0..k], yang juga merupakan sufiks dari *pattern* tersebut, P[1..k], dengan $k = j-1$. Fungsi pinggiran ini dimaksudkan agar karakter yang sudah dicocokkan pada prefiks, tidak perlu dicocokkan kembali, sehingga menghemat pencocokan *string*. Berikut adalah contoh mencari fungsi pinggiran :

Misalkan diberikan *pattern* P sebagai berikut :

P :

a	b	a	c	a	b

Fungsi pinggiran :

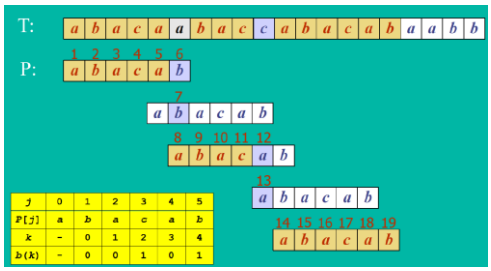
j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
k	-	0	1	2	3	4
B[k]	-	0	0	1	0	1

Misalkan untuk mencari fungsi pinggiran dari karakter "c", yaitu karakter dengan indeks 3 pada *pattern* P. Maka nilai k dari karakter "c" tersebut adalah 2 (k = 3 – 1). Sehingga akan dicari panjang terbesar dari *substring* prefiks P[0..2] yang juga merupakan *substring* sufiks P[1..2]. Maka *substring* prefiks P[0..2] adalah {a, ab, aba} dan *substring* sufiks P[1..2] adalah

{a, ba}. Terlihat untuk panjang *substring* adalah 1 untuk prefiks P[0..2] dan sufiks P[1..2] adalah sama, yaitu *substring* "a". Maka nilai fungsi pinggiran untuk karakter "c" tersebut adalah *B[2] = 1*.

b) Pencocokan String

Pencocokan *string* dimulai dari indeks ke-0 teks dan indeks ke-0 *pattern*. Lalu bergerak ke kanan hingga *pattern* ditemukan atau pergeseran tidak bisa dilakukan lagi karena teks sudah mencapai akhir. Ketika terjadi *mismatch* pada pencocokan *string* dari *pattern* terhadap teks pada indeks ke-i, maka fungsi pinggiran tersebut akan digunakan untuk menentukan berapa banyak pergeseran yang harus dilakukan selanjutnya. Berikut adalah contoh pencocokan *string* dengan algoritma KMP :



Gambar 2 : Pencocokan string dengan algoritma KMP
(Sumber : Bahan Kuliah IF2211 Strategi Algoritma : Pencocokan String (String/Pattern Matching), diakses pada 25 April 2020)

Algoritma KMP memiliki kompleksitas waktu sebesar *O*(m+n) dengan *O*(m) adalah kompleksitas waktu menghitung fungsi pinggiran dan *O*(n) adalah kompleksitas waktu pencocokan *string*.

3. Algoritma Boyer-Moore

Algoritma Boyer-Moore atau algoritma BM merupakan algoritma pencocokan *string* dengan teknik yang cukup unik namun dinilai paling efisien. Algoritma Boyer-Moore menggunakan dua teknik, yaitu teknik *looking-glass* dan *character-jump*. Kedua teknik ini yang membuat algoritma BM lebih efisien daripada algoritma pencocokan *string* yang dijelaskan sebelumnya.

a) Teknik Looking-glass

Teknik *looking-glass* pada algoritma BM akan mencocokkan *pattern* dengan teks dengan pencocokan karakter secara mundur. Teknik *looking-glass* digunakan karena lebih efisien sebab mempercepat pergeseran yang harus dilakukan selanjutnya secara optimal ketika ditemukan *mismatch*.

b) Teknik Character-jump

Algoritma Boyer-Moore juga didukung dengan teknik *character-jump*, yaitu teknik pergeseran dengan mempertimbangkan kemunculan karakter yang *mismatch* antara *pattern* dengan teks. Misalkan terjadi *mismatch* antara karakter "X" yaitu pada indeks ke-i pada teks, *T[i]*, dan karakter indeks ke-j pada *pattern*, *P[j]*. Ada tiga kasus kemunculan karakter yang *mismatch*, yaitu :

i. Kasus 1

Terjadi ketika karakter *mismatch T[i]* ditemukan di sebelah kiri karakter *mismatch P[j]*. Maka geser *pattern* ke kanan sampai karakter *T[i]* sejajar dengan karakter "X" pada *pattern*.

ii. Kasus 2

Terjadi ketika karakter *mismatch T[i]* ditemukan di sebelah kanan karakter *mismatch P[j]* yang berarti karakter *mismatch T[i]* sudah dicek sebelumnya. Maka geser *pattern* ke kanan sebanyak satu karakter, sehingga *P[j]* sejajar dengan *T[i+1]*.

iii. Kasus 3

Terjadi ketika karakter *mismatch T[i]* tidak ditemukan dimanapun pada *pattern*. Maka geser *pattern* ke kanan sebanyak panjang *pattern*, sehingga *T[i+1]* sejajar dengan *P[0]*.

c) Fungsi Last Occurance

Algoritma Boyer-Moore memerlukan *pre-processing* untuk mengetahui letak semua karakter pada teks. *Pre-processing* ini dinamakan dengan fungsi *last occurance* atau kemunculan terakhir karakter teks pada *pattern*. Jika karakter teks ada pada *pattern*, maka nilai *last occurance* nya adalah indeks terbesar kemunculan karakter tersebut pada *pattern*, namun apabila tidak ditemukan karakter tersebut pada *pattern*, maka nilai *last occurance* nya adalah -1. Berikut adalah contoh mencari *last occurance* dari teks :

Misalkan diberikan *pattern* P dan teks T sebagai berikut :

T :

i	0	1	2	3	4	5	6	7	8
T[i]	a	b	d	a	b	a	c	a	b

P :

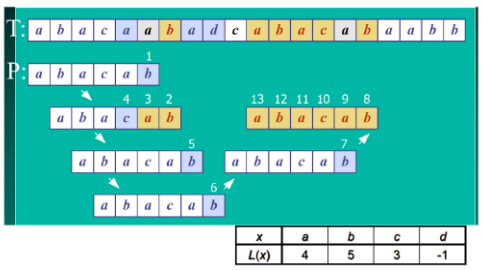
j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b

Fungsi *Last Occurance* :

x	a	b	c	d
L(x)	4	5	3	-1

c) Pencocokan String

Pencocokan *string* algoritma Boyer-Moore menggunakan kedua teknik di atas, yaitu teknik *looking-glass* dan teknik *character-jump* dengan memanfaatkan fungsi *last occurance*. Berikut adalah contoh proses pencocokan *string* menggunakan algoritma Boyer-Moore :



Gambar 3 : Pencocokan string menggunakan algoritma Boyer-Moore

(Sumber : Bahan Kuliah IF2211 Strategi Algoritma : Pencocokan String (String/Pattern Matching), diakses pada 26 April 2020)

Kompleksitas waktu yang dibutuhkan algoritma Boyer-Moore untuk melakukan pencocokan *string* adalah sebesar *O*(NM) untuk kasus terburuk, *O*(N/M) untuk kasus terbaik, dan *O*(N/M) untuk kasus rata-rata. Algoritma Boyer-Moore lebih cepat dibanding algoritma Brute Force untuk teks dengan karakter beragam, namun sedikit lambat untuk karakter yang sedikit.

4. *Regular Expression*

Regular Expression atau biasa disebut Regex adalah salah satu metode pencocokan *string* yang unggul. Metode Regex dinilai unggul karena pencocokannya menggunakan pola *regular expression* yang diberikan. Pada metode Regex, *pattern* yang dimasukkan dapat berupa *pattern* yang eksak atau pola yang ingin dicari. Untuk membuat *Regular Expression* digunakan beberapa simbol untuk menentukan *pattern* yang dicari. Berikut adalah beberapa simbol yang digunakan pada Regex :

Simbol	Deskripsi
\d	Semua angka, [0-9]
\w	Semua huruf, [a-zA-Z_0-9]
^	Awal *string*
$	Akhir *string*
[abc]	Karakter a atau b atau c
X?	X muncul satu atau tidak sama sekali
X*	X muncul nol atau banyak kali
X+	X muncul satu atau banyak kali
X{n}	X muncul tepat n kali
X{n,}	X muncul minimal n kali
X{m,n}	X muncul m sampai n kali

5. *Algoritma Levenshtein Distance*

Algoritma *Levenshtein Distance* adalah algoritma pencocokan *string* dengan metode menghitung jumlah operasi yang dilakukan pada dua *string* yang dibandingkan.Operasi yang dimaksud berupa penggantian (*replacement*), penyisipan (*insertion*), dan penghapusan (*deletion*) karakter. Algoritma *Levenshtein Distance* juga dapat digunakan untuk mengetahui seberapa cocok dua string. Algoritma ini dilakukan dengan pendekatan program dinamis.

Misalkan diberikan *string* a dan b. Untuk mempermudah mencocokkan *string* tersebut, dibuatlah matriks berukuran (m+2) x (n+2), m dan n berurutan adalah panjang *string* a dan b ditambah 1. Misalkan a adalah *string* “HONDA” dan b adalah *string* “HYUNDAI”. Maka bentuk matriksnya adalah sebagai berikut :

	“ ”	H	Y	U	N	D	A	I
“ ”								
H								
O								
N								
D								
A								

Elemen pada matriks tersebut diisi sesuai dengan fungsi *levenshtein distance* berikut :

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Gambar 4 : Fungsi Levenshtein Distance
(Sumber : https://dzone.com/articles/the-levenshtein-algorithm-1 diakses pada 2 Mei 2020*)*

Dengan keterangan :

- $\mathrm{lev}_{a,b}(i-1, j) + 1$: operasi penghapusan
- $\mathrm{lev}_{a,b}(i, j-1) + 1$: operasi penyisipan
- $\mathrm{lev}_{a,b}(i-1, j-1) + 1_{(\text{ai != bj})}$: operasi penghapusan
- $1_{(\text{ai != bj})}$: bernilai 1 jika elemen a_i != b_j

Sehingga matriks di atas menjadi :

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Gambar 5 : Matriks Levenshtein Distance
(Sumber : https://dzone.com/articles/the-levenshtein-algorithm-1 diakses pada 2 Mei 2020*)*

Nilai *levenshtein distance* adalah elemen matriks (m,n) atau yang terletak di pojok kanan bawah. Nilai tersebut menunjukkan banyaknya operasi yang dilakukan untuk mengubah *string* “HONDA” menjadi *string* “HYUNDAI”

III. PEMBAHASAN

A. *Kesalahan Penggunaan Kata Tidak Baku*

Penggunaan kata tidak baku sering ditemui dalam kehidupan sehari-hari, baik dalam lisan maupun tulisan. Kata tidak baku muncul karena penggunaan bahasa dalam percakapan sehari-hari atau keadaan informal, sehingga sering menimbulkan salah kaprah. Kata tidak baku yang sering digunakan antara lain sebagai berikut (sebelah kiri kata baku dan sebelah kanan kata tidak baku) :

- Atlet – Atlit
- Apotik – Apotek
- Bus – Bis
- Hafal – Hapal
- Izin – Ijin

- Miliar – Milyar
- Telur – Telor
- Risiko – Resiko
- Sopir – Supir
- Zaman – Jaman

Kata tidak baku tersebut memiliki makna yang sama, namun dengan penulisan yang salah.

B. *Pencarian Kata Tidak Baku dengan Algoritma Pencocokan String*

Algoritma pencocokan *string* banyak digunakan pada bidang yang berhubungan dengan teks atau tulisan. Persoalan yang dapat diselesaikan menggunakan algoritma pencocokan *string* antara lain pencarian kata pada teks, mendeteksi keberadaan kata pada kamus, dan mendeteksi bahasa pada teks. Pendekatan yang berbeda untuk masing-masing algoritma juga berdampak pada waktu penyelesaian persoalan.

Untuk melihat kemampuan algoritma pencocokan *string* pada persoalan mencari kata tidak baku dalam kamus, maka dilakukan uji coba. Uji coba dilakukan pada lima algoritma pencocokan *sring* yang terdiri dari :

- Algoritma Brute Force
- Algoritma Knuth-Morris-Pratt
- Algoritma Boyer-Moore
- Regular Expression
- Algoritma Levenshtein Distance

Uji coba dilakukan menggunakan bahasa pemrograman Python dengan basis data kata dasar KBBI yang didapat pada laman https://github.com/andrisetiawan/lexicon. Basis data kamus yang didapat pada laman tersebut berisikan 28.526 kata dari KBBI. Basis data ini memang tidak mengandung seluruh kata yang terdapat pada KBBI asli, namun dapat dijadikan representasi KBBI aslinya. Pengujian dilakukan dengan mencari kata tidak baku "Atlit", "Ijin", dan "Jaman" dengan harapan menghasilkan kata bakunya.

1. *Menggunakan Algoritma Brute Force*

Implementasi algoritma Brute Force pencocokan *string* pada bahasa Python adalah sebagai berikut :

```
def BF_Match(text, pattern):
  m = len(text)
  n = len(pattern)
  for i in range(0,m-n+1):
      j = 0
      while j < n and pattern[j] == text
[i+j]:
          j = j + 1
      if j == n:
          return i
  return -1
```

Dengan menggunakan algoritma Brute Force, hasil pencarian untuk ketiga kata uji adalah sebagai berikut :

```
C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : atlit
Hasil pencarian :
Tidak ditemukan kata yang mirip
Time elapsed : 0.3790769577026367

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : ijin
Hasil pencarian :
kijing
pijin
pijinasi
Time elapsed : 0.2920699119567871

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : jaman
Hasil pencarian :
jamang
Time elapsed : 0.28507208824157715
```

Gambar 6 : Pencarian kata tidak baku dengan algoritma Brute Force (Sumber : Koleksi pribadi penulis)

Seperti yang terlihat pada gambar 6 di atas, pencarian dengan algoritma Brute Force tidak satupun menghasilkan hasil yang diharapkan. Hal ini karena algoritma Brute Force melakukan pencarian secara *exact match*, sehingga hasil-hasil yang muncul hanya yang mengandung kata masukan.

2. *Menggunakan Algoritma Knuth-Morris-Pratt*

Implementasi algoritma Knuth-Morris-Pratt pencocokan *string* pada bahasa Python adalah sebagai berikut :

```
def failureFunction(pattern) :
  failure = [0] * len(pattern)
  m = len(pattern)
  i = 1
  j = 0
  while (i < m) :
      if (pattern[i] == pattern[j]) :
          failure[i] = j + 1
          i += 1
          j += 1
      elif (j > 0) :
          j = failure[j-1]
      else :
          failure[i] = 0
          i += 1
  return failure

def KMP_Match(text, pattern) :
  m = len(pattern)
  n = len(text)
  i = 0
  j = 0
  failure = failureFunction(pattern)
  while (i < n) :
      if (text[i] == pattern[j]) :
          if (j == (m - 1)) :
              return i - m + 1
          i += 1
          j += 1
```

```
        elif (j > 0) :
            j = failure[j-1]
        else :
            i += 1
    return -1
```

Dengan menggunakan algoritma Knuth-Morris-Pratt, hasil pencarian ketiga kata uji adalah sebagai berikut :

```
C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : atlit
Hasil pencarian :
Tidak ditemukan kata yang mirip
Time elapsed : 0.33008289337158203

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : ijin
Hasil pencarian :
kijing
pijin
pijinasi
Time elapsed : 0.348085880279541

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : jaman
Hasil pencarian :
jamang
Time elapsed : 0.32817673683166504
```

Gambar 7 : Pencarian kata tidak baku dengan algoritma Knuth-Morris-Pratt (Sumber : Koleksi pribadi penulis)

Algoritma Knuth-Morris-Pratt menghasilkan keluaran yang sama seperti pada algoritma Brute Force. Algoritma Knuth-Morris-Pratt gagal memberikan bentuk baku dari kata masukan. Hal ini karena algoritma Knuth-Morris-Pratt mencari *string* secara *exact match*.

3. Menggunakan Algoritma Boyer-Moore

Implementasi algoritma Boyer-Moore pencocokan *string* pada bahasa Python adalah sebagai berikut :

```
def lastOccuranceFunction(pattern):
    lastOccurance = [-1]*256
    for i in range (len(pattern)) :
        lastOccurance[ord(pattern[i])] = i
    return lastOccurance

def BM_Match(text, pattern) :
    m = len(pattern)
    n = len(text)
    i = m - 1
    lastOccurance =  lastOccuranceFunction
(pattern)
    if (i > (n - 1)) :
        return -1
    j = m - 1
    while (i <= (n - 1)):
        if (text[i] == pattern[j]) :
            if (j == 0) :
                return i
            else :
                i -= 1
                j -= 1
        else :
            lo = lastOccurance[ord(text[i]
)]
            i = i + m - min(j, lo + 1)
            j = m - 1
    return -1
```

Dengan menggunakan algoritma Boyer-Moore, hasil pencarian ketiga kata uji adalah sebagai berikut :

```
C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : atlit
Hasil pencarian :
Tidak ditemukan kata yang mirip
Time elapsed : 0.36509037017822266

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : ijin
Hasil pencarian :
kijing
pijin
pijinasi
Time elapsed : 0.3970987796783447

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : jaman
Hasil pencarian :
jamang
Time elapsed : 0.407102108001709
```

Gambar 8 : Pencarian kata tidak baku dengan algoritma Boyer-Moore (Sumber : Koleksi pribadi penulis)

Algoritma Boyer-Moore juga menghasilkan keluaran yang sama seperti dua algoritma sebelumnya. Dengan alasan yang sama, algoritma Boyer-Moore melakukan pencarian *string* secara *exact match*, sehingga yang muncul hanya kata yang mengandung kata masukan.

4. Regular Expression

Pengujian dengan *regular expression* menggunakan *library* Python bernama re. Implementasi *regular expression* pada bahasa Pyton adalah sebagai berikut :

```
def Regex_Match(text, pattern) :
    x = re.search(pattern, text)
    if (x != None) :
        return x.start()
    return -1
```

Dengan menggunakan metode *regular expression*, hasil pencarian ketiga kata uji adalah sebagai berikut :

```
C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : atlit
Hasil pencarian :
Tidak ditemukan kata yang mirip
Time elapsed : 0.2750670909881592

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : ijin
Hasil pencarian :
kijing
pijin
pijinasi
Time elapsed : 0.32207489013671875

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : jaman
Hasil pencarian :
jamang
Time elapsed : 0.239058256149292
```

Gambar 9 : Pencarian kata tidak baku dengan regular expression (Sumber : Koleksi pribadi penulis)

Metode *regular expression* menghasilkan keluaran yang tidak sesuai dengan harapan. Hal ini karena pada persoalan ini, metode regex dipaksa untuk mencari secara *exact match*, sehingga yang muncul juga kata-kata yang mengandung kata masukan.

5. Menggunakan Algoritma Levenshtein Distance

Pengujian dengan algoritma *levenshtein distance* menggunakan *library* Python bernama python-Levenshtein untuk menghitung *levenshtein distance* dan fuzzywuzzy untuk mencari kata. Implementasi algoritma *Levenshtein Distance* pada bahasa Python adalah sebagai berikut :

```
def LD_Match(text, pattern) :
  return fuzz.token_sort_ratio(text, pat
tern)
```

Toleransi kecocokan *levenshtein distance* yang digunakan adalah sebesar 75 (satuan hitung fuzzywuzzy). Hasil yang didapatkan adalah sebagai berikut :

```
C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : atlit
Hasil pencarian :
alit
batolit
satelit
albit
atlet
calit
palit
ali
ati
Time elapsed : 0.8722243309020996

C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : ijin
Hasil pencarian :
pijin
jin
kijing
biji
ijon
izin
wiji
Time elapsed : 0.8242852687835693
```

Gambar 10 : Pencarian kata tidak baku dengan algortma Levenshtein Distance (Sumber : Koleksi pribadi penulis)

```
C:\Users\ASUS\Desktop\Makalah STIMA (master -> origin)
λ python main.py
Masukkan kata pencarian : jaman
Hasil pencarian :
jamang
jamban
aman
jambang
jambian
jasmani
amang
atman
daman
gaman
jahan
jajan
jalan
jamah
jamak
jamal
jamin
japan
jaran
laman
maman
paman
saman
taman
zaman
jamblang
jambulan
ama
jam
man
Time elapsed : 0.8443107604980469
```

Gambar 11 : Pencarian kata tidak baku dengan algortma Levenshtein Distance (Sumber : Koleksi pribadi penulis)

Algoritma *levenshtein distance* ternyata memiliki keluaran yang berbeda dengan keempat algoritma lainnya. Algoritma *levenshtein distance* sukses mencari kata baku dari kata masukan. Contohnya pada kata “atlit”, ditemukan kata bakunya yaitu “atlet”. Hal ini karena algoritma *levenshtein distance* akan mencari kata yang mirip dengan kata masukan, sehingga akan muncul kata bakunya.

C. Penjelasan Hasil Uji Coba

Berdasarkan hasil uji coba yang telah dilakukan, algoritma Brute Force, algoritma Knuth-Morris-Pratt, algoritma Boyer-Moore, dan *regular expression* gagal menghasilkan kata baku yang sesuai dengan kata tidak baku masukan. Namun hal yang berbeda didapatkan dari algoritma *levenshtein distance*. Dari kelima algoritma yang digunakan uji coba, algoritma *Levenshtein Distance* memberikan keluaran yang sesuai. Keluaran ini bisa didapatkan karena algoritma *Levenshtein Distance* dapat menentukan aksi perubahan yang dilakukan satu *string* untuk menjadi *string* pembanding yang lain, sehingga dapat menemukan *string* yang miri. Banyaknya aksi perubahan ini yang menentukan kemiripan antar *string* yang dibandingkan. Semakin kecil nilai *levenshtein distance*-nya, semakin mirip *string* tersebut. Pada persoalan ini, nilai *levenshtein distance* digunakan untuk mencari kata baku pada KBBI yang termirip dengan kata tidak baku yang diberikan, dengan harapan hasil yang dikeluarkan adalah bentuk baku dari kata tidak baku tersebut.

IV. Penutup

A. Simpulan

Algoritma pencocokan *string* dapat dimanfaatkan untuk melakukan pencarian kata pada kamus. Beberapa algoritma pencocokan *string* seperti algoritma Brute Force, algoritma Knuth-Morris-Pratt, dan algoritma Boyer-Moore melakukan pencocokan *string* secara *exact match*. Metode *regular expression* lebih digunakan untuk pencocokan *string* dengan *pattern* yang memiliki format tertentu, seperti format jam, format tanggal, dan format penulisan. Namun metode *regular expression* juga dapat digunakan untuk mencari *string* yang *exact match*. Persoalan pencarian kata tdak baku pada KBBI ini lebih mudah menggunakan algoritma *Levenshtein Distance*. Algoritma *Levenshtein Distance* dapat menghitung kemiripan *string* masukan dengan *string* pembanding pada kamus, sehingga dapat menemukan kata baku yang sesungguhnya dari kata tidak baku pada masukan.

B. Saran

Saran yang dapat diberikan dari penulisan makalah ini adalah pemanfaatan algoritma pencocokan *string* ini perlu dikembangkan lagi menjadi sebuah aplikasi yang berguna dalam pengecekan penulisan teks, terutama dalam bahasa Indonesia. Algoritma pencocokan *string* ini dapat digabungkan dengan *machine learning* untuk menghasilkan aplikasi yang cerdas dan optimal, seperti *auto correct* dan *auto suggestion*.

Link Video Youtube

Berikut adalah *link* video Youtube yang membahas tentang topik makalah ini : https://youtu.be/pUgYcsiJRl4

Ucapan Terima Kasih

Penulis mengucapkan rasa syukur kepada Allah SWT. karena dengan rahmat dan berkahnya, penulis bisa menyelesaikan penulisan makalah ini dengan baik di tengah keadaan pandemi Corona ini. Penulis mengucapkan terima kasih kepada dosen mata kuliah IF2211 Strategi Algoritma, yakni Bapak Dr. Ir. Rinaldi Munir, M.T., Ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc., dan Ibu Dr. Masayu Leylia Khodra, S.T., M.T, karena telah memberikan pengetahuan mengenai strategi algoritma selama masa perkuliahan. Tak lupa juga, penulis mengucapkan terima kasih kepada teman- teman IF 2018 yang telah memberikan dukungan dan bantuan selama pengerjaan makalah ini.

Referensi

[1] Munir, Rinaldi. 2020. *Bahan Kuliah IF2211 Strategi Algoritma Pencokan String (String/Pattern Matching).* Bandung:Program Studi Teknik Informatika ITB

[2] Rosmala, Dewi dan Zulfikar Muhammad Risyad. 2017. *Algoritma Levenshtein Distance dalam Aplikasi Pencarian Kata Isu di Kota Bandung pada Twitter*. Bandung:Institut Teknologi Nasional Bandung

[3] Ilmy, Muhammad Bahari, Nitia Rahmi, dan Roland L. Bu'ulolo. 2006. *Penerapan Algoritma Levenshtein Distance untuk Mengoreksi Kesalahan Pengejaan pada Editor Teks*. Bandung:Institut Teknologi Bandung

[4] https://github.com/andrisetiawan/lexicon diakses 2 Mei 2020 pada 6.40 WIB

[5] https://salamadian.com/contoh-kata-baku-dan-tidak-baku/ diakses 2 Mei 2020 pada 7.20 WIB

[6] https://kbbi.web.id/ diakses 2 Mei 2020 pada 7.20 WIB

[7] https://dzone.com/articles/the-levenshtein-algorithm-1 diakses 2 Mei 2020 pada 7.30 WIB

[8] https://towardsdatascience.com/natural-language-processing-for-fuzzy-string-matching-with-python-6632b7824c49 diakses 2 Mei 2020 pada 7.40 WIB

[9] https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/ diakses 2 Mei 2020 pada 21.05 WIB

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.

Surabaya, 3 Mei 2020

Daffa Pratama Putra
13518033