# Perancangan Google Assistant Sederhana dengan Penerapan Algoritma Pencocokkan String

Kevin Austin Stefano 13518104[1]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*[1]13518104@std.stei.itb.ac.id*

***Abstract*—Di era digitalisasi ini, kehidupan manusia berubah dengan cepat. Aspek kehidupan manusia mulai berjalan berdampingan dengan teknologi yang semakin kreatif dan berkembang. Teknologi di masa ini menolong manusia untuk melakukan banyak hal dan memberikan suatu cara pandang baru. Salah satunya yang populer adalah Google Assistant. Penulis memodelkan perancangan Google Assistant sederhana dengan mengimplementasikan Algoritma Pencocokkan String.**

***Keywords*—algoritma knuth-morris-pratt, boyer-moore, regex, google assistant**

## I. Pendahuluan

Globalisasi dan digitalisasi berkembang pesat saat ini seiring berkembangnya ilmu pengetahuan. Oleh karena itu, teknologi terus dikembangkan secara inovatif dan kreatif demi menjawab tantangan era ini. Kehidupan yang semakin modern menuntut teknologi-teknologi terbaru untuk terus menghasilkan manfaat positif bagi orang banyak.

Salah satu teknologi yang populer saat ini adalah penggunaan Google Assistant. Google Assistant adalah fitur yang dikembangkan oleh Google untuk menghadirkan asisten virtual bagi kehidupan manusia sehari-hari. Jika dahulu kita harus mengetikkan apa yang kita inginkan di mesin pencarian Google, sekarang kita bisa langsung mengatakan hal yang ingin kita ketahui dan Google Assistant akan memberikan langsung jawabannya.


Gambar 1. Google Assistant
(https://cdn0.tnwcdn.com/wp-content/blogs.dir/1/files/2017/05/share-796x418.jpg)

Beberapa fitur dapat dilakukan oleh Google Assistant saat kita mengatakan hal yang ingin dilakukan oleh Google Assistant, seperti memutar musik, mengetahui cara untuk melakukan sesuatu, set alarm, dan masih banyak lagi. Kita hanya perlu mengatakan "Bagaimana cara memasang dasi yang tepat?" Maka secara otomatis Google Assistant akan menampilkan video tutorial memakai dasi yang benar. Atau saat kita ingin mengetahui terjemahan suatu kalimat, kita bisa langsung mengatakan "Apa Bahasa Mandarinnya aku cinta kamu?" dan Google Assistant akan memberikan jawabannya langsung pada kita.


Gambar 2. Ilustrasi penggunaan Google Assistant
(https://assistant.google.com/intl/id_id/platforms/speakers/)

Pada makalah ini, kita akan mencoba melihat bagaimana penerapan Google Assistant dalam menginterpretasikan kalimat-kalimat yang diucapkan oleh penggunanya, mengubahnya menjadi teks, mengekstraksi teks, dan mencari jawaban yang akan diberikan kepada pengguna. Kita akan merancang sebuah Google Assistant sederhana yang mampu mendengarkan apa yang diinginkan pengguna dan kemudian menggunakan algoritma pencocokkan string yang ada, seperti Algoritma Knuth Morris Pratt, Boyer-Mooyer, dan regex untuk mencari *keyword* yang diinginkan untuk menghasilkan jawaban-jawaban yang diinginkan.

## II. Landasan Teori

### 2.1. Karakter, String, dan Substring

Karakter atau *character* adalah entitas terkecil dari suatu kata yang terdiri dari huruf dalam alfabet (vokal maupun konsonan), digit(atau *numeric*), dan simbol khusus lainnya (seperti @,$,%) yang mana terdaftar dalam kumpulan karakter ASCII.

String adalah tipe data yang merupakan kumpulan atau gabungan dari beberapa karakter menjadi suatu kesatuan. String menyimpan barisan/beberapa karakter dan mengemasnya menjadi satu unit string.

Substring adalah himpunan bagian dari string yang berisikan urutan karakter yang letaknya berdekatan pada suatu string. Misalnya kita memiliki suatu string "Cuaca hari ini indah", maka salah satu substring yang mungkin adalah "Cuaca hari" substring tersebut adalah bagian dari string dan letaknya berdekatan.

Substring terdiri dari 2, yaitu *prefix* (awalan) dan *suffix* (akhiran). Suatu substring dapat kita katakan sebagai suatu *prefix* jika letaknya sebagai "awalan" suatu string dan berdekatan. Bila kita bentuk secara general, maka jika s adalah string, c adalah karakter, dan p adalah prefix, maka s = pc. Sebagai contoh string sebagai berikut.

| H | A | R | I | | S | A | B | T | U |
|---|---|---|---|---|---|---|---|---|---|

Maka *prefix* yang mungkin dalam string di atas adalah P={H, HA, HAR, HARI, HARI , HARI S, HARI SA, HARI SAB, HARI SABT, HARI SABTU}.

Sedangkan suatu substring dapat kita katakan sebagai *suffix* / akhiran jika letaknya pada "akhir" suatu string dan berdekatan. Bila kita generalisasi, jika s adalah string, c adalah karakter, dan p adalah prefix, maka s = cp. Jika bentuk string seperti contoh di atas, maka *suffix* yang mungkin adalah S={U, TU, BTU, ABTU, SABTU, SABTU, I SABTU, RI SABTU, ARI SABTU, HARI SABTU}.

### 2.2. Pendahuluan Algoritma Pencocokkan String

Algoritma pencocokkan string adalah algoritma yang digunakkan untuk mencari kemunculan *keyword* yang diinginkan pada rangkaian kalimat yang kita miliki. *Keyword* yang diinginkan biasa disebut *pattern* dan rangkaian kalimat biasa kita sebut dengan teks. Dengan algoritma ini maka kita bisa mendeteksi apakah *pattern* yang kita masukkan ada pada teks atau tidak, terletak pada indeks ke berapa, dan masih banyak lagi.

### 2.3. Jenis Algoritma Pencocokkan String

A. Algoritma Naïve (*Brute Force*)

Algoritma naïve atau *brute force* adalah algoritma pencocokkan atau pencarian string yang melakukan iterasi satu persatu pada text dan membandingkannya dengan *pattern.* Ide dari algoritma ini adalah kita membandingkan karakter pertama (i) dari text dengan karakter pertama pada *pattern*. Jika *mismatch* atau tidak sama, maka kita menggeser ke kanan satu karakter di text (i+1). Jika *match*, maka kita akan menggeser ke kanan satu karakter di text dan menggeser ke kanan satu karakter di *pattern*. Langkah-langkah di atas dilakukan hingga karakter di text semuanya sudah di iterasi dan di periksa.

Text:

| A | B | A | C | A | D | A | A | B |
|---|---|---|---|---|---|---|---|---|

Pattern

| A | D |
|---|---|

Maka prosesnya :

| Text | A | B | A | C | A | D | A | A | B |
|---|---|---|---|---|---|---|---|---|---|

Proses pencocokkan *pattern* dengan text

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Proses 1 | A | D | - | - | - | - | - | - | - |
| Proses 2 | - | A | D | - | - | - | - | - | - |
| Proses 3 | - | - | A | D | - | - | - | - | - |
| Proses 4 | - | - | - | A | D | - | - | - | - |
| Proses 5 | - | - | - | - | A | D | - | - | - |

Pada algoritma naïve, kita bisa melihat bahwa kasus terbaik terjadi saat karakter awal *pattern* tidak pernah sama dengan karakter yang diiterasi di teks. Kasus terburuknya adalah saat karakter tiap *pattern* sama dengan karakter teks, namun karakter akhir *pattern* berbeda dengan yang di teks. Sehingga kompleksitas kode rata-ratanya adalah O($m + n$).

B. Algoritma Knuth-Morris-Pratt

Awalnya, algoritma Knuth-Morris-Pratt dikembangkan oleh James H. Morris bersama Vaughan R. Pratt pada tahun 1966 dan Donald E. Knuth tahun 1967 secara terpisah. Algoritma ini dumumkan ke public pada tahun 1977. Dasar perancangan algoritma ini adalah karena melihat bahwa algoritma naïve melakukan banyak perbandingan antara karakter teks dan *pattern.* Sehingga, algoritma Knuth-Morris-Pratt memiliki ide algoritma yang mampu meningkatkan pergeseran dilakukan sehingga lebih efisien dibandingkan dengan algoritma naïve / *brute force*.

Ide umum dari algoritma KMP ini adalah dengan melakukan pergeseran karakter dari kiri ke kanan. Saat terjadinya *mismatch* antara text(T) pada T[i] denggan pattern (P) pada P[j] atau dengan kata lain T[i] != P[j], maka langkah yang kita lakukan adalah dengan menggeser *pattern* sedemikian rupa sehingga *largest prefix Pattern*[0..j-1] adalah sama dengan *suffix pattern* dari P[1..j-1] dan sejajar. Proses ini dapat dipersingkat dengan kita membuat tabel fungsi pinggiran atau tabel *border function* yang mana fungsi ini memberikan nilai

pergeseran terbesar yang bisa ditempuh oleh pencarian string.

Langkah Algoritma Knuth-Morris-Pratt adalah sebagai berikut.

1. Kita merancang tabel *border function* yang melakukan membuat list tiap prefix *pattern* dan menghitung nilai *largest prefix patter*n [0..j-1] yang sama dengan *suffix pattern* dari P[i..j-1] dan sejajar dengan teks bersangkutan
2. Selanjutnya kita memulai pencocokan string dari indeks awal teks T[0] dan *pattern* P[0]. Selama belum ditemukan atau indeks teks(i) belom lebih dari panjang teks(len(i)), maka kita melakukan dua hal.
3. Jika teks T[i] sama dengan pattern P[j], maka i digeser satu ke kanan begitu juga j digeser satu ke kanan (i= i+1, j= j+1).
4. Jika terjadi *mismatch* di indeks j >0, maka kita memulai mencari *pattern* di table Border Function.
5. Jika terjadi *mismatch* di indeks j =0, maka indeks i digeser 1 dan indeks j dimulai dari 0.

```
while j is less than len Pattern
      if mismatch
            if j=0 then j_new= 0, i_new++
            else i_new =  border function
      else  { if match }
            i_new increase
            j_new increase
      endif
endwhile
```

Text:

| A | B | A | A | D | B | G | H | D |
|---|---|---|---|---|---|---|---|---|

Pattern:

| A | B | A | B | C |
|---|---|---|---|---|

Tabel *border function*

| j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| P[j] | A | B | A | B | C |
| k | - | 0 | 1 | 2 | 0 |
| B[k] | - | 0 | 0 | 1 | 0 |

Maka prosesnya :

| Text | A | B | A | A | D | B | G | H | D |
|---|---|---|---|---|---|---|---|---|---|

Proses pencocokkan *pattern* dengan text

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Proses 1 | A | B | A | B | C | - | - | - | - |
| Proses 2 | A | B | A | B | C | - | - | - | - |
| Proses 3 | A | B | A | B | C | - | - | - | - |
| Proses 4 | A | B | A | B | C | - | - | - | - |
| Proses 5 | - | - | A | B | A | B | C | - | - |
| Proses 6 | - | - | - | A | B | A | B | C | - |
| Proses 7 | - | - | - | A | B | A | B | C | - |
| Proses 8 | - | - | - | - | A | B | A | B | C |
| Proses 9 | - | - | - | - | - | A | B | A | B |
| Proses 10 | - | - | - | - | - | - | A | B | A |
| Proses 11 | - | - | - | - | - | - | - | A | B |
| Proses 12 | - | - | - | - | - | - | - | - | A |

Dalam mengkalkulasi tabel *border function*, kompleksitas algoritmanya adalah $O(m)$. Sedangkan dalam melakukan pencocokan terhadap string kompleksitasnya $O(n)$. Maka, kompleksitas totalnya adalah $O(m+n)$.

C. Algoritma Boyer-Moore

Algoritma pencocokkan string Boyer-Moore adalah algoritma yang dikembangkan oleh J. Strother Moore dan Robert S. Boyer pada tahun 1977. Algoritma Boyer-Moore tergolong cukup efisien dibandingkan algoritma-algoritma lainnya pada umumnya. Hal yang membuat algoritma ini spesial karena algoritma ini menggunakan 2 teknik yang mampu membuat pencocokkan antara pattern dan teks seminimal mungkin namun tetap memberikan hasil yang optimal.

Ide algoritma ini adalah kita melakukan pencocokan dari kanan ke kiri dan melakukan lompatan antara karakter sehingga lebih efisien. Dua teknik yang digunakan dalam algoritma ini adalah melakukannya dengan *looking-glass* teknik dimana memeriksa kecocokan pattern yang dimulai dari indeks terakhir P bergerak dari kiri ke kanan. Dan selanjutnya kita bisa melakukannya dengan teknik *character jump* dimana terdapat *mismatch* maka kita melakukan *jump* terhadap indeks text sehingga pemeriksaan berlangsung lebih efektif. Namun sebelum melakukannya kita membuat terlebih dahulu tabel *last occurrence*. Tabel ini berisikan indeks kemunculan terakhir karakter pada *pattern*. Untuk karakter-karakter yang tidak terdapat pada string *pattern*, maka kita mendefinisikan nilai karakter tersebut dengan -1, yang artinya tidak ada pada *pattern,*

Langkah yang kita bisa lakukan untuk membentuk algoritma Boyer-Moore adalah sebagai berikut.

1. Kita mendefinisikan terlebih dahulu table Last Occurrence mengenai informasi kemunculan terakhir suatu karakter pada *pattern* P. Untuk karakter yang tidak ada di *pattern*, kita definisikan dengan nilai -1.
2. Selanjutnya kita akan melakukan pencocokan string dari indeks *Pattern* dari kanan ke kiri P[len(pattern)..0]. Selama belum ditemukan atau indeks teks(i) belom lebih dari panjang teks(len(i)), maka kita akan melakukan 3 kasus.
   a. KASUS 1, Jika teks T[i] nilainya sama dengan Pattern P[j], maka kita melakukan 2

hal.

i. Jika indeks Pattern j sudah 0, pemeriksaan berhenti. Pattern sudah matching
ii. Jika belum, indeks i dan j sama sama digeser sekali ke kiri

b. KASUS 2, Jika indeks teks lebih kecil nilainya dengan yang di table *Last Occurrence*, maka lakukan $i_{new} = i + m - (i_o+1)$
c. KASUS 3, jika indeks teks lebih besar nilainya dengan yang di table Last Occurrence, maka lakukan $i_{new} = i + m - j$

```
while j is less than len Pattern
      if mismatch
            j_new= len (P)-1
            i_new = i+m-min(lo+1,j)

      else  { if match }
            i_new decrease
            j_new decrease
            if j =0 then stop
            endif
      endif
endwhile
```

Text

| A | B | A | C | A | A | B | A | D |
|---|---|---|---|---|---|---|---|---|

Pattern

| A | B | A | C | A | B |
|---|---|---|---|---|---|

Tabel *last occurrence*

| x | A | B | C | D |
|---|---|---|---|---|
| L(x) | 4 | 5 | 3 | -1 |

Maka prosesnya :

| Text | A | B | A | C | A | A | B | A | D |
|---|---|---|---|---|---|---|---|---|---|

Proses pencocokkan *pattern* dengan text

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Proses 1 | A | B | A | C | A | B | - | - | - |
| Proses 2 | - | A | B | A | C | A | B | - | - |
| Proses 3 | - | A | B | A | C | A | B | - | - |
| Proses 4 | - | A | B | A | C | A | B | - | - |
| Proses 5 | - | - | A | B | A | C | A | B | - |

D. Regex

*Regex* atau *regular expression* adalah sistematika berbasis teori otomata bahasa formal untuk mendeteksi keberadaan suatu pola pada bahasa. Perancangan regex terdiri dari kumpulan konstanta dan operator dan melakukan pencocokan terhadap pola yang dicari berdasarkan urutan karakter atau string. *Regex* akan melakukan pencarian string secara literal dengan pembatasan yang didefinisikan melalui quantifier maupun *boundary matchersnya*. *Quantifier* dalam *regex* digunakan untuk mendefinisikan jumlah pengulangan pola. Sedangkan *Boundary Matcher* digunakan untuk mencari pola yang yang ada pada suatu posisi tertentu.

Adapun konstruksi kelas dilakukan dengan beberapa pendekatan. Untuk konstruksi pembentukan kelas karakter, maka kita melakukan ketentuan ini.

Tabel 1. Konstruktor Kelas *Regex*

| Konstruktor | Deskripsi |
|---|---|
| [abc] | a, b, atau c |
| [^abc] | Selain a,b,c |
| [a-zA-Z] | a sampai z atau A sampai Z |
| [a-d[m-p]] | a sampai d atau m sampai p |
| [a-z&&[def]] | d, e atau f |
| [a-z&&[^bc]] | a sampai z, kecuali b dan c |
| [a-z&&[^m-p]] | a sampai z, dan bukan m sampai p |

Untuk mencari pola pada posisi tertentu di *regex* yang kita definisikan maka kita menggunakan *Boundary Matchers*. Adapun *Boundary Matchers* dari *regex* adalah sebagai berikut.

Tabel 2. *Boundary Matchers Regex*

| Boundary Matcher | Deskripsi |
|---|---|
| . | Semua karakter |
| \d | Digit [0-9] |
| \D | Selain digit [^0-9] |
| \s | Karakter spasi |
| \S | Selain karakter spasi [^s] |

Untuk mendefinisikan pengulangan maka kita menggunakan *Quantifie*r sebagai berikut

Tabel 3. *Quantifier*

| Quantifier | Arti |
|---|---|
| X? | X muncul satu atau tidak sama sekali |
| X* | X tidak muncul atau banyak |
| X+ | X muncul 1x atau banyak |
| x{n} | X muncul tepat n kali |
| x{n,} | X muncul paling sedikit n kali |
| x{n,m} | X muncul n sampai m kali |

## III. PEMBAHASAN

### 3.1. Karakteristik Google Assistant

Google Assistant adalah asisten virtual Google berbasis suara. Google Assistant memiliki banyak fitur yang dapat dilakukan, antara lain:

1. Melakukan *translate* dari Bahasa yang kita ucapkan
2. Set timer dan reminder
3. Mencari informasi secara *online*, seperti "tutorial menggunakan sesuatu", "tempat hotel", dan sebagainya
4. Memasang musik
5. Control terhadap *device* yang kita miliki
6. Melakukan akses terhadap informasi yang kita miliki, seperti akses dari calendar, notifikasi, mengirim pesan atau *message*.

Google Assistant akan mengenal kita dan memberikan informasi jawaban dari hal-hal yang kita katakan kepada Google Assistant untuk dilakukan. Secara garis besar kita mengidentifikasikan proses yang dilakukan oleh Google Assistant sebagai berikut.

1. Pengguna mengucapkan kalimat. Misalnya "Bagaimana cara memakai dasi?"
2. Google Assistant akan mendengarkan apa yang dikatakan oleh pengguna dan menyimpannya ke Google server untuk dilakukan analisis
3. Google akan mengubah suara kita yang sudah direkam dan mengubahnya menjadi teks.
4. Google akan mengidentifikasi *keyword* atau *pattern* yang ada di dalam teks tersebut. *Keyword* akan dicocokkan untuk mendapatkan jawaban yang diinginkan
5. Google memberikan jawaban hasilnya kepada pengguna.


Gambar 3. Skema proses Google Assistant

Google Assistant dibentuk dengan *natural language processing* (NLP). NLP adalah pengolahan bahasa natural dan membuat pengguna seolah-olah bisa berinteraksi langsung dengan komputer. Namun pada makalah ini, kita akan merancang Google Assistant sederhana dengan metode pencocokkan string yang sudah diajarkan.

### 3.2. Perancangan Google Assistant sederhana

Berdasarkan langkah proses yang dilakukan oleh Google Assistant, maka kita akan membuat Google Assistant sederhana yang melakukan proses pengidentifikasian kalimat menggunakan algoritma pencocokan string yaitu Algoritma Knuth-Morris-Pratt, dan Algoritma Boyer-Mooyer

Langkah pertama kita lakukan adalah membuat melakukan proses pengubahan suara menjadi teks. Dengan kakas *speech-recognizer* yang disediakan Python, maka kita bisa mengubah suara yang kita ucapkan menjadi teks.


Gambar 4. Ilustrasi *speech-to-text*

Langkah kedua yang kita lakukan adalah kita membuat kamus atau dictionary terhadap bagian-bagian yang kita identifikasikan. Contohnya misalnya jika ada teks masukkan "ingin pergi ke ITC Mangga Dua", maka karena ada ***identifier*** "pergi ke" yang sudah ada di kamus yang kita miliki maka kita akan menghubungkan *value*-nya ke proses selanjutnya. Dalam proses ini kita bisa melakukkan pencocokkan string dengan algoritma yang sudah kita miliki, Kita bisa menggunakan tiga buah algoritma yang sudah dijelaskan. Misalkan kita menggunakan Algoritma Knuth-Morris-Pratt untuk kasus ini.

Teks

| .. | I | N | | P | E | R | G | I | | K | E | | I | T | C | .. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Pattern

| P | E | R | G | I | | K | E | |
|---|---|---|---|---|---|---|---|---|

Tabel Border Function

| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| P[j] | P | E | R | G | I | | K | E | |
| k | - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| B[k] | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Proses

| .. | I | N | | P | E | R | G | I | | K | E | | I | T | C | .. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| - | P | E | R | G | I | | K | E | | - | - | - | - | - | - | - |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | P | E | R | G | I | | K | E | | - | - | - | - | - | - |
| - | - | - | P | E | R | G | I | | K | E | | - | - | - | - | - |
| - | - | - | - | P | E | R | G | I | | K | E | | - | - | - | - |

Karena kita menemukan ***identifier*** yaitu “pergi ke” yang sudah ada di kamus kita, maka kita mengambil value yang berada di indeks setelah kata “pergi ke “. Dalam hal ini kita mengambil value “ITC Mangga Dua” untuk diproses ke tahap selanjutnya.

Proses selanjutnya adalah kita membuat “proses yang dilakukan” untuk value yang telah didapat. Untuk membuat Google Assistant, kita membuat Google Assistant mampu melakukan 6 *task* sederhana sehingga proses dibagi menjadi 6 bagian. Misalnya identifier yang didapat adalah “pergi ke” dan valuenya adalah “ITC Mangga Dua” maka kita akan melakukan proses membuka google maps dan mencari nilai value “ITC Mangga Dua” di Google Maps. Dalam hal ini penulis mengusulkan bentuk kamus tipe, identifier, yang dilakukan, dan perancangan katalog. Berikut ini adalah tabel kamus yang bisa kita buat.

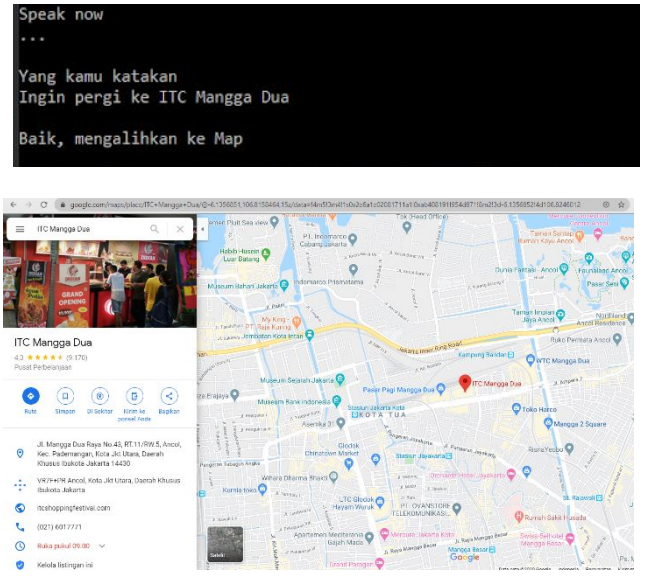
Tabel 4. Kamus

| Tipe | Identifier | Yang dilakukan |
|---|---|---|
| Tempat | “pergi ke “ + value<br>“dimana “ + value<br>“cara ke “ + value<br>“jalan ke “ + value | **Buka google map**<br>'https://www.google.com/maps/place/' + value |
| Alarm | “set waktu “+ jam<br>“alarm “+jam | **Set timmer**<br>untuk tipe jam |
| Tutorial | “cari cara“ + value<br>“bagaimana cara ” +value<br>“buka di youtube “ + value | **Buka youtube**<br>https://www.youtube.com/results?search_query=’ + value |
| Waktu | “jam berapa”<br>“pukul berapa” | **Check jam sekarang** |
| Translate | “terjemahkan ke bahasa” + bahasa + value<br>“apa bahasa” + bahasa + value | **Buka Google translate**<br>https://translate.google.co.id/?hl=id#view=home&op=translate&sl=id&tl=’ + bahasa + ‘&text=’ + value |
| Musik | “Putar musik"<br>“Putar lagu“ | **Buka spotify**<br>‘https://open.spotify.com/playlist/37i9dQZF1DXa2EiKmMLhFD:autoplay’ |
| KATALOG | | A, B, C, D, J, P, S, T |

Seperti yang bisa dilihat dari tabel di atas. Maka kita melakukan “proses yang dilakukan” dengan cara proses + value untuk tipe tempat, proses + bahasa untuk tipe *translate,* dan lainnya. Berikut adalah contoh *pseudocode*-nya.

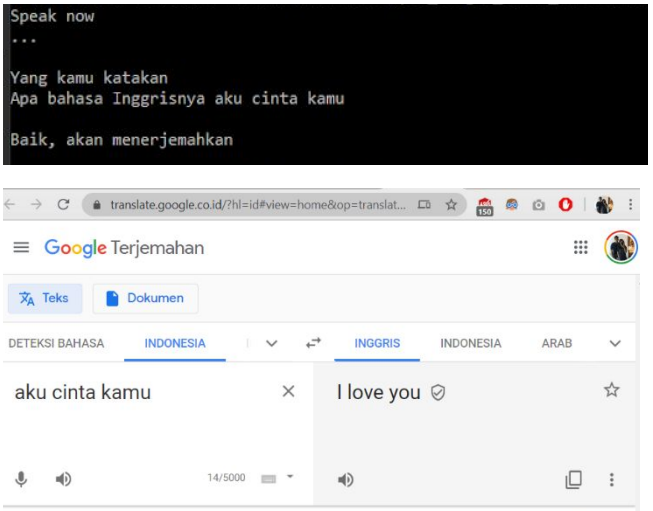
```
If identifier = ‘tempat’
      open in maps( url + value )
else if identifier = ‘alarm’
      set timmer (jam)
else if identifier = ‘tutorial’
      open in browser ( url + value )
else if identifier = ‘waktu’
      return TIME(jam)
else if identifier = ‘translate’
      open in google translate(url+id+value)
else if identifier = ‘music’
      open spotify
```

Proses akan dilakukan berdasarkan tipenya masing-masing. *Pattern* dari teks akan dicocokkan dengan *identifier* di kamus untuk melakukan “proses” sesuai tipenya masing-masing. Bentuk ini adalah bentuk tampilan untuk prosesnya adalah sebagai berikut.




Gambar 5. Ilustrasi proses (dimulai dari *text-to-speech*, pencarian *keyword* dengan algoritma pencocokkan string, melakukan “proses” sesuai *keyword*)

‘




Gambar 6. Ilustrasi proses dengan identifier = ‘translate’

## IV. Analisis

Penulis menganalisis bahwa untuk kamus berukuran kecil yang hanya memiliki *identifier* atau *pattern* dalam jumlah sedikit. Maka proses dapat dijalankan dengan cara sebelumnya dengan mengecheck setiap *identifier* dengan text yang ada. Jika ada 10 *identifier*, maka kita melakukan 10 kali proses pencocokkan string antara *pattern* dengan teks menggunakan algoritma Boyer-Moore dan Knuth-Morris-Pratt.

Namun, masalah timbul ketika kita memiliki *pattern* atau *identifier* berjumlah sangat banyak pada kamus yang kita memiliki. Namun, dengan menggunakan algoritma pencocokkan string, apakah memungkinkan untuk melakukan pengecekan terhadap *multipattern* berjumlah besar.

Oleh karena itu penulis mengusulkan 2 solusi sederhana yang mampu memberikan pencocokkan yang lebih optimal dan 1 solusi yang membuat pencarian lebih efektif dalam hal ini untuk pattern dengan jumlah besar dalam kamus.

Yang pertama, penulis melihat bahwa banyaknya kata-kata berimbuhan yang dapat digunakan dalam suatu teks, namun bisa membuat pencarian kita tidaklah optimal. Contohnya, kita memiliki *identifier* "pergi ke", namun pengguna bisa saja mengucapkan "bepergian ke". Imbuhan ber- dan -an bisa membuat *identifier* atau *pattern* "pergi ke" tidaklah ditemukan dalam teks yang kita miliki. Oleh sebab itu kita bisa melakukan yang pertama adalah *stemming*. *Stemming* adalah proses memetakan kata ke dalam akar dari kata tersebut dengan menghapus imbuhan dari kata yang kita miliki. Contohnya jika kita memiliki kata "menaiki" maka kata tersebut bisa kita *stemming* menjadi "naik" dengan menghapus imbuhan me- dan -i. Beberapa bahasa seperti Python, men-*support* proses *stemming* dengan kakas yang dimilikinya.

Yang kedua adalah kita bisa melihat kata-kata yang "relatif sama" atau memiliki arti yang sama pada bahasa Indonesia. Contohnya adalah "pergi ke" memiliki arti yang sama dengan "jalan-jalan ke" atau "mainkan music" memiliki arti yang sama dengan "putar music". Hal-hal seperti ini juga membuat kamus kita tidaklah optimal. Oleh karena itu, untuk kata-kata yang memiliki makna sama, kita bisa melakukan proses "lematisasi" atau *lemmatization*. Lematisasi adalah proses untuk mencari akar kata yang berupa sinonim akar dari kata yang kita miliki.

Tabel 5. Ilustrasi *stemming* dan *lemmatization*

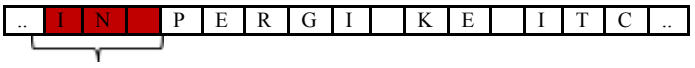
| Stemming | Lemmatization |
|---|---|
| Berpergian -> pergi<br>Menaiki -> naik<br>Mendengarkan -> dengar | hendak, mau -> ingin<br>setel -> putar |

Dengan adanya proses *stemming* (penghapusan imbuhan) dan *lemmatization* (pengubahan ke akar kata sinonim), maka diharapkan teks akan dicocokkan dengan kamus dengan lebih optimal. Hal ini disebabkan karena kata-kata dalam teks akan diekstrak terlebih dahulu menjadi akar kata (*root word)* kemudian barulah dicocokkan dengan pattern pada kamus yang kita miliki.

Selain proses pengoptimalan, penulis juga mengusulkan cara pencarian *pattern* yang lebih efektif. Penulis mengusulkan metode pencarian dengan kamus berbasis katalog. Penulis melihat berdasarkan fakta di lapangan bahwa pengguna mengucapkan perintah ke Google Assistant umumnya dengan panjang yang tidak terlalu besar. Misalnya seperti "Set Alarm pukul 12.00". **Karena teks yang kita miliki tidak terlalu besar, namun kita memiliki banyak *Pattern***, maka penulis mengusulkan metode pencarian berbasis katalog. Katalog adalah inisial dari setiap *identifier* yang ada di kamus. Algoritmanya adalah sebagai berikut.

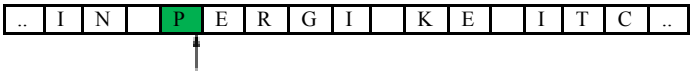
1. Proses iterasi karakter dari kiri ke kanan. Pencocokkan dilakukan terhadap katalog yang mungkin. Kita langsung mencari kode inisial dari kamus yang kita miliki, yaitu{A, B, C, D, J, P, S, T} dengan teks yang kita miliki.
2. Jika *mismatch* maka langsung ke karakter selanjutnya.
3. Jika karakter ada di katalog, maka kita lakukan sebagai berikut.
    a. Jika ada di katalog dan katalog menyimpan beberapa identifier, maka geser satu kali ke kanan hingga kandidat identifier hanya 1. Hitung berapa j langkah pergeserannya.
    b. Jika tinggal hanya satu identifier, check dari kiri ke len(pattern) ke kiri (Adaptasi Algoritma Boyer Moore) apakah string cocok atau tidak.

Berikut ini proses pencocokkan dengan identifier atau *pattern* yang kita miliki




Tidak ada di katalog

Karena "P" ada di katalog dan masih banyak kandidat *identifier* yang mungkin,, maka kita set j langkah =0 dan kita hitung langkah pergeseran ke kanan sampai kandidat *identifier* hanya 1 dengan menambahkan j dengan 1.

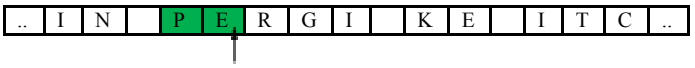



P ada di katalog, Set j = 0

Identifier yang mungkin : Pergi ke, Putar Musik, Pukul berapa, Putar Lagu
Karena jumlah_identifier >1, j++

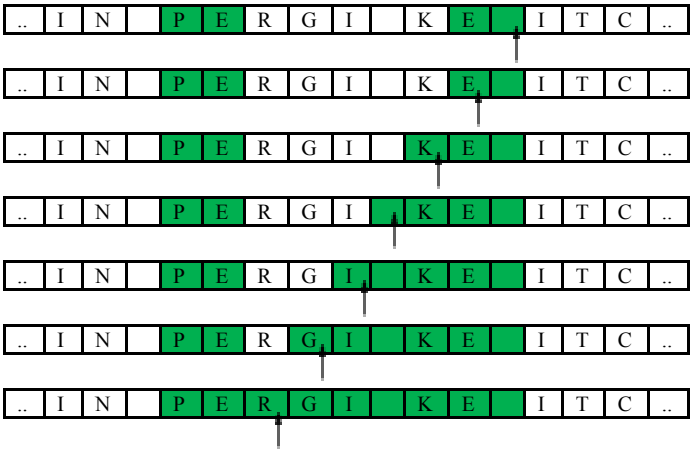
Kita geser sekali ke kanan (sehingga j=0+1). Kemudian kita check karakter sekarang. Karena "E" di *pattern* P[k] dalam hal ini j adalah 1 sehingga P[1]=="E" hanya ada di salah satu identifier yang mungkin yaitu "pergi ke ".Kemudia kita mulai pencocokkan *pattern* dari P[j..len(pattern)] dalam hal ini kita mengecheck "ergi ke ". Kita bisa memulai mencocokkan mulai

dari len(pattern) ke i atau P[(len(pattern)..j]




Check Identifier[j] ==’E’
Identifier yang mungkin : Pergi ke (tinggal 1) atau jumlah_identifier==1.
Lakukan proses algoritma dari *pattern* P[len(pattern)..j] yaitu “ergi ke” dari kanan ke kiri dan setelah itu stop.

Proses selanjutnya akan menjadi sebagai berikut ini.




Karena pattern “*match”* maka kita mengambil nilai value setelahnya. Rumus yang penulis ajukan adalah mengambil *value* yang terletak pada text = i + len(*pattern*) sehingga dalam hal ini kita mendapatkan value = “ITC Mangga Dua” untuk diproses yang sudah ada di kamus yang kita miliki.

Kita akan mencoba membandingkan jumlah proses pencariannya. Jika kita memiliki 1000 identifier dengan rata-rata panjang karakternya 10 di kamus dan memiliki panjang karakter teks 30 karakter. Maka jika proses sebelumnya, kita akan melakukan pencarian di teks (misal dengan algoritma Knuth-Morris-Pratt) dengan 1 *pattern* sebanyak 1000 kali. Maka banyak proses terburuk yang dilakukan adalah 30x1000 = 30000 proses. Sedangkan dengan algoritma pencarian katalog ini hanya berjumlah len(text) atau 30 proses perbandingan.

## V. Kesimpulan

Proses penerapan algoritma pencocokkan string sangatlah diperlukan dalam mencari pattern yang kita miliki di dalam teks dalam merancang Google Assistant. Namun, tetap perlunya optimalisasi agar proses ekstraksi teks berjalan lebih optimal dan maksimal. Penulis mengusulkan proses stemming dan lematisasi pada teks sehingga teks tidak “mentah” langsung dicari keywordnya, namun diekstrak terlebih dahulu sehingga teks menjadi teks yang “bersih” barulah kita lakukan proses pencarian keyword dengan algoritma pencocokkan string yang ada. Selain itu penulis juga mengusulkan pencarian string di teks berbasis katalog yang diharapkan mampu mencari *multipattern* di teks yang berukuran kecil dibandingkan dengan jumlah pattern yang terbilang banyak.

## VI. Ucapan Terima Kasih

Puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena atas berkat dan kasih-Nya, penulis dapat menyelesaikan makalah yang berjudul “Perancangan Google Assistant Sederhana dengan Penerapan Algoritma Pencocokkan String”.

Penulis juga berterima kasih kepada dosen penulis, Ibu Ulfa, Pak Rinaldi Munir, Bu Masayu yang telah membimbing penulis selama perkuliahan strategi algoritma ini sehingga penulis dapat menyelesaikan makalah ini.

Penulis juga berterima kasih kepada kedua orang tua penulis yang selalu memberikan dukungan dan motivasi kepada penulis sehingga perkuliahan ini dapat berjalan dengan lancar. Selain itu, penulis juga berterima kasih kepada teman-teman penulis yang mendukung penulis selama perkuliahan strategi algoritma.

## Referensi

[1] Munir, Rinaldi. 2006. Strategi Algoritma. Bandung: Institut Teknologi Bandung.

[2] Lothaire, M. (1997). Kombinatorik pada kata-kata . Cambridge: Cambridge University Press.

[3] https://www.pocket-lint.com/apps/news/google/137722-what-is-google-assistant-how-does-it-work-and-which-devices-offer-it
Diakses, 25 April 2020

[4] https://www.quora.com/How-does-Google-Assistant-actually-work
Diakses, 25 April 2020

[5] https://www.datacamp.com/community/tutorials/stemming-lemmatization-python
Diakses, 1 Mei 2020

## Lampiran

Untuk lebih memahami mengenai isi makalah, maka penulis menyediakan video tutorial mengenai pengerjaan proyek ini. Berikut ini adalah link untuk penjelasan di youtube.
https://youtu.be/YLf-nTizSK8

## 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, 1 Mei 2020

Kevin Austin Stefano / 13518104