Penerapan *String Matching* Pada *Auto-Correct* Berbasis Algoritma Levenshtein Distance

Adam Rotal Yuliandaru 13514091
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
adamrotal@gmail.com

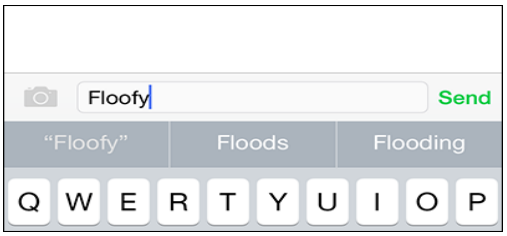
***Abstract*—*Auto-correct* merupakan sebuah fitur yang dapat membantu penggunanya dalam menuliskan sebuah kata atau kalimat. *Auto correct* telah banyak diaplikasikan dalam berbagai bidang seperti mesin pencari, daftar rekomendasi pada *e-commerce*, editor teks seperti Eclipse atau NetBeans. *Auto-correct* menerapkan prinsip pencocokan kemiripan string. Algoritma yang umum digunakan untuk mengimplementasikan *auto-correct* adalah Algoritma Levenshtein Distance yang menghitung *string metric* antara dua *string*. Dalam makalah ini, akan diimplementasikan algoritma algoritma Levenshtein Distance dan Brute Force untuk mengimplementasikan fitur *auto-correct*.**

***Keywords*—*Auto-correct*, Levenstein Distance, pencocokan string, kemiripan string, *string metric*, *Brute Force*.**

I. PENDAHULUAN

Seiring pada perkembangan zaman, manusia semakin sering menciptakan berbagai macam teknologi yang mampu mengubah perilaku seseorang Semakin berkembangnya teknologi memacu manusia untuk terus berinovasi untuk menciptakan sebuah teknologi yang mampu mempermudah kehidupannya. Selain mempermudah pekerjaan, teknologi juga dapat menggantikan manusia dalam mengerjakan sesuatu (otomatisasi).

Salah satu hal yang mendapatkan dampak otomatisasi adalah pengetikan teks pada gawai. Fitur fundamental yang hampir semua gawai dewasa ini adalah fitur *auto-correct*. *Auto-correct* merupakan sebuah fitur dalam gawai yang dapat membenarkan teks yang kita tulis pada gawai secara langsung. Dengan memeriksa huruf demi huruf yang telah diketikkan oleh pengguna maka system akan mencari dan mencocokkan huruf-huruf itu dengan huruf-huruf yang berada dalam database.



Gambar 1 : Contoh *auto-correct* pada Messages iPhone
Sumber : google.com

II. DASAR TEORI

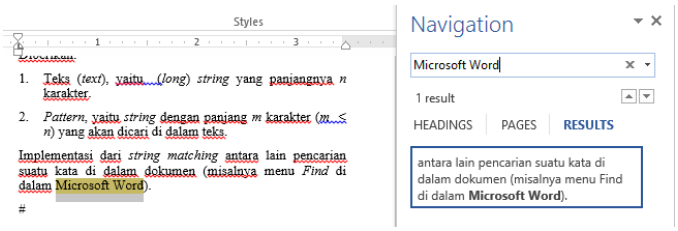
A. Pencocokan *String* (*String Matching*)

String matching adalah suatu algoritma yang digunakan untuk memecahkan masalah pencocokan suatu teks terhadapap teks lain. Persoalah *string matching* dirumuskan sebagai berikut:

Diberikan:

1. Teks (*text*), yaitu (*long*) *string* yang panjangnya *n* karakter.
2. *Pattern*, yaitu *string* dengan panjang *m* karakter (*m* < *n*) yang akan dicari di dalam teks.

Implementasi dari *string matching* antara lain pencarian suatu kata di dalam dokumen (misalnya menu *Find* di dalam Microsoft Word).



Gambar 2 : Contoh implementasi *string matching* pada Microsoft Word.
Sumber : dokumentasi pribadi

Contoh :

Pattern : hari

Teks : kami pulang ke rumah ketika **hari** mulai malam
^target

Pada makalah ini, penulis mengasumsikan jika *pattern* muncul lebih dari sekali di dalam teks, maka pencarian akan berhenti ketika *pattern* pertama pada teks ditemukan.

B. Algoritma *Brute Force* pada *String Matching*

Dengan asumsi bahwa teks berada di dalam *array* T[1..n] dan pattern berada di dalam array P[1..m], maka algoritma *brute force* pencocokan string adalah sebagai berikut :

1. Mula-mula *pattern* P dicocokkan pada awal teks T.
2. Deng bergerak dari kiri ke kanan, bandingkan setiap karakter di dalam *patter* P dengan karakter yang bersesuaian di dalam teks T samapi :
    a. Semua karakter yang dibandingkan cocok atau sama (pencarian berhasil), atau
    b. Dijumpai sebuah ketidakcocokan karakter (pencarian belum berhasil).
3. Bila *pattern* P belum ditemukan kecocokannya dan teks *T* belum habis, geser *pattern* P satu karakter ke kanan dan ulangi langkah 2.

Pseudo-code algoritmanya :

```
Procedure BruteForceSearch (input m, n : integer, input P :
                            array[1..m] of char, input T :
                            array[1..n] of char, output idx :
                            integer)
{ mencari kecocokan pattern P di dalam teks T. Jika
  ditemukan P di dalam T, lokasi awal kecocokan disimpan
  di dalam peubah idx.

  Masukkan : pattern P yang panjangnya m dan teks T
  yang panjangnya n. Teks T direpresentasikan sebagai
  string (array or character).

  Keluaran : posisi awal kecocokan (idx). Jika P tidak
  ditemukan, idx = -1.
}
Deklarasi
  s, j : integer
  ketemu : boolean

Algoritma
  S ← 0
  ketemu ← false
  while (s ≤ n-m) and (not ketemu) do
        j ← 1
        while (j ≤ m) and (P[j] = T[s+j]) do
              j ← j + 1
        endwhile
        { j > m or P[j] ≠ T[s+j] }

        if (j = m) then {kecocokan ditemukan}
              ketemu ← true
        else
              s ← s + 1
        endif
  endwhile
  { s > n – m or ketemu }

  if (ketemu) then
        idx ← s + 1
  else
        idx ← -1
  endif
```

Kompleksitas algoritma pencocokan string dihitung dari jumlah operasi perbandingan yang dilakukan. Kompleksitas kasus terbaik adalah O(*n*). Kasus terbaik terjadi jika karakter pertama pattern P tidak pernah sama dengan karakter teks T yang dicocokkan (kecuali pada pencocokan terakhir). Pada kasus ini, jumlah perbandingan yang dilakukan paling banuak n kali misalnya :

Teks : Ini adalah string panjang yang berakhir dengan zz
Pattern : zz

Kasus terburuk membutuhkan m(n – m + 1) perbandingan, yang mana kompleksitasnya adala O(mn), misalnya :

Teks : aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Pattern : aaaab

C. String Metric

Dalam matematika dan ilmu komputer, *string metric* (juga dikenal sebagai *string similarity metric* atau *string distance function*) adalah metrik yang mengukur jarak ("*inverse similarity*") antara dua *string* teks untuk pencocokan atau perbandingan *string* dan fuzzy *string searching*. Misalnya string "Sam" dan "Samuel" dapat dianggap dekat.

String metric digunakan secara luas pada sistem informasi yang memerlukan pencocokan untuk dapat mengambil suatu kesimpulan, misal *fraud detection* (deteksi penipuan), analisis sidik jari, deteksi plagiarism, analisis DNA dan RNA, analisi gambar, *evidence-based mechine learning*, *database deduplication*, *data mining*, *web interface*.

D. Levenshtein Distance

Vladimir Iosifovich Levenshtein adalah seorang ilmuwan dan matematikawan berkebangsaan Rusia yang melakukan penelitian terhadap *information theory*, *error-correcting code*, dan *combinatorial design*. Salah satu karyanya yang dikenal hingga saat ini adalah Levenstein distance yang dikembangkannya pada tahun 1965.

Dalam dunia informasi dan ilmu computer, Levenshtein Distance adalah *string metric* yang digunakan untuk mengukur perbedaan antara dua *string*. Levenshtein Distance mengukur seberapa jauh perbedaan dua *string* jika dilakukan penyuntingan pada *string* tersebut. Penyuntingan yang dimaksud adalah *insertions*, *deletions*, dan *substitutions*.

Secara matematika, nilai Levenshtein distance antara dua *string* a, b adalah

$$\mathrm{lev}_{a,b}(|a|,|b|)$$

dimana

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

$1_{(ai \neq bj)}$ akan bernilai 1 jika $a_i \neq b_j$ dan bernilai 0 jika $a_i \neq b_j$. $Lev_{a, b}(i, j)$ adalah jarak antara anatara *i* karakter pertama dari a dengan *j* karakter pertama dari b.

Contoh, Levenshtein Distance antara "kitten" dan "sitting" adalah 3, dengan proses :

1. **k**itten ⟶ **s**itten (substitusi "s" dan "k")
2. sitt**e**n ⟶ sitt**i**n (substitusi "i" dan "e")
3. sittin ⟶ sittin**g** (penyisipan "g")

dalam contoh diatas terlihat proses perhitungan Levenshtein Distance. Proses pertama adalah dari kata "kitten" kita substitusi huruf 'k' dengan 's' sehingga mendapatkan kata "sitten" (1). Kemudian substitusi huruf 'i' pada "sitten" dengan 'e' sehingga menjadi "sittin" (2). Langkah terakhir, sisipkan huruf 'g' agar membentuk kata "sitting" (3). Sehingga dapat ditarik kesimpulan bahwa nilai Leveshtein distance antara "kitten" dan "sitting" adalah 3.

Pseudo-code algoritmanya :

```
function levDis (s1 : string, s2 : string) : integer
kamus
   i, cost : integer
   mBefore : array [0 .. s1.length] of integer
   mCurrent : array [0 .. s1.length] of integer
algoritma
   for i ← 0 to s1.length do
      {inisialisasi baris awal dengan nilai jarak
      perbandingan dengan kosong}
      mBefore[i] ← i

   for i ← 0 to s1.length do
      for j ← 1 to s2.length do
         if i = 0 then  {perbandingan dgn kosong}
            mCurrent[i] ← j
         else
            if s1[i] = s2[j] then
               cost ← 0
            else
               cost ← 1

            mCurrent[i] = minimum (
                  mBefore[i-1] + cost, {substitusi}
                  mCurrent[i-1] + 1, {penghapusan}
                  mBefore[i] + 1, {penambahan}
               )

      mBefore ← mCurrent

   return  mCurrent[s1.length]
```

Algoritma Levenshtein Distance memiliki kompleksitas waktu *O(mn)*, dengan m dan n adalah panjang masing-masing *string* yang dibandingkan. Dengan kata lain, jika kedua *string* memiliki panjang yang sama kompleksitas waktunya adalah $O(n^2)$. Kompleksitas ruang untuk algoritma ini adalah juga *O(n)* atau $O(n^2)$ karena suatu baris matriks hanya membutuhkan data dari baris sebelumnya saja, berarti hanya dibutuhkan $2 \times n$ ruang memori untuk menyimpannya.

String metric pada contoh "kitten" dan "sitting" :

		K	I	T	T	E	N
	0	1	2	3	4	5	6
S	1	1	2	3	4	5	6
I	2	2	1	2	3	4	5
T	3	3	2	1	2	3	4
T	4	4	3	2	1	2	3
I	5	5	4	3	2	2	3
N	6	6	5	4	3	3	2
G	7	7	6	5	4	4	3

Table 1 : String Metric "Kitten" dan "Sitting"

Pada tabel diatas, elemen baris 1 kolom 1 (T[1,1]) merupakan jumlah operasi yang dperlukan untuk mengubah substring dari kata SITTING yang diambil mulai dari karakter awal sebanyak 1 (S) ke substring dari kata KITTEN yang diambil mulai dari karakter awal sebanyak 1 (K). Sementara elemen T[3,5] adalah jumlah operasi antara SIT (substring yang diambil sebanyak 3 karakter dari awal) dengan KITTE (substring yang diambil sebanyak 5 karakter dari awal. Berarti elemen T[p,q] adalah jumlah operasi antara substring kata pertama yang diambil mualai dari awal sebanyak p dengan substring kata kedua yang diambil dari awal sebanyak q.

Elemen terakhir (kanan bawah) merupakan elemen yang nilainya menyatakan nilai Levenshtein Distance dari kedua string yang dibandingkan.

III. Cara Kerja Auto-Correct

Misalkan diberikan sebuah kata, auto-correct akan mencoba memilih koreksi ejaan yang paling memungkinkan untuk kata itu. Tidak ada cara untuk mengetahui dengan pasti (contoh : "lates" dikoreksi menjadi "late" atau "latest"?), makadari itu kita menggunakan teori probabilitas.

Teori probabilitas digunakan untuk menentukan peluang dari sebuah kata yang ingin dikoreksi, missal *w*, terhadap kata original yang benar, missal *w*.

$$\text{argmax}_c P\ (c|w)$$

dengan menerapkan teorema Bayes, didapat :

$$\text{argmax}_c P\ (w|c)\ P(c) / P(w)$$

karena P(w) selalu bernilai sama untuk setiap kemungkinan c, maka dapat diabaikan, sehingga

$$\text{argmax}_c P\ (w|c)\ P(c)$$

Keterangan :

P(c) adalah peluang c dikoreksi menjadi c itu sendiri. Hal ini disebut sebagai *language model*. Dengan kata lain, jika c telah terdapat pada database kata, maka c tidak perlu dikoreksi lagi. Makadari itu, P("aku") akan memiliki peluang yang relatif lebih tinggi, sedangkan P("zxzxzxzyyy") akan memiliki peluang mendekati 0.

P($w|c$) adalah peluang penulisan w pada teks ketika yang pengguna maksud adalah c. Dengan kata lain P($w|c$) adalah peluang kesalah pengetikan sebuah kata c.

argmax_c adalah sebuah fungsi yang menghitung semua nilai *feasible* dari c, dan kemudian memilih salah satu yang memberikan nilai probabilitas terbaik.

Teorema Bayes digunakan untuk mengubah ekspresi sederhana P($c|w$) menjadi lebih kompleks. Jawabannya adalah karena lebih mudah untuk memecah masalah dan menanganinya masing-masing secara eksplisit. Misalkan w = *"*thew*"* dan kandidat kata pengoreksinya adalah c = *"*the*"* dan c = *"*thaw*"*. Manakah yang memiliki P($c|w$) lebih besar ? "thaw" terlihat lebih memungkinkan karena hanya dibutuhkan substitusi "a" menjadi "e" yang mana adalah sebuah operasi kecil. Di sisi lain "the" terlihat lebih memungkinkan karena "the" merupakan sebuah kata yang umum digunakan dalam Bahasa Inggirs, dan mungkin pengetik tidak sengaja menekan "w". Intinya adalah bahwa untuk memperkirakan P($c|w$) kita harus mempertimbangkan baik probabilitas c dan probabilitas perubahan dari c ke w, sehingga memecahnya menjadi dua faktor, P($w|c$) dan P(c) akan lebih memudahkan pekerjaan.

IV. Implementasi

Pada implementasi auto-correct ini, penulis menggunakan bahasa Java dan sebuh *database* kata dasar Bahasa Indonesia yang dapat diunduh di internet.

Pencarian string dalam kamus dapat dilakukan dengan beberapa jenis algoritma. Diantaranya adalah algoritma *Brute Force*. Pencarian string dilakukan berdasarkan karakter-karakter yang bersesuaian pada kata yang akan dikoreksi. Pencarian akan dilakukan kata perkata, sehingga bila terdapat ketidakcocokan pada awal kata, maka pencarian akan langsung berlanjut pada kata berikutnya.

Contoh :

Kata yang akan diperiksa :

banxung

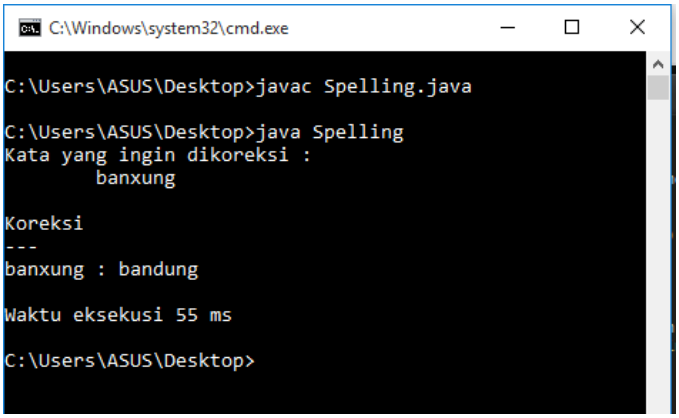
Kata dalam kamus :

a
ab
aba
.
.
.
bandu
bandul
bandung
bandusa
bandut
.
.
.
zurafah
zuriah
zus

Pencarian dilakukan secara *brute force*. Misalnya *pattern* yang dicari adalah "bandung", maka pada perbandingan terhadap kata "bandung" akan berhenti pada karakter ke-4.

Pada setiap perbandingan terhadap kata-kata di dalam kamus, juga dicari nilai Levenshtein distance-nya yang akan digunakan untuk menemukan kata uang kemiripannya terbesar terhadap kata yang sedang diperiksa.

Setelah proses selesai, maka akan ditampilkan kata yang paling mirip berdasarkan pencarian dan Levenshtein distance-nya.

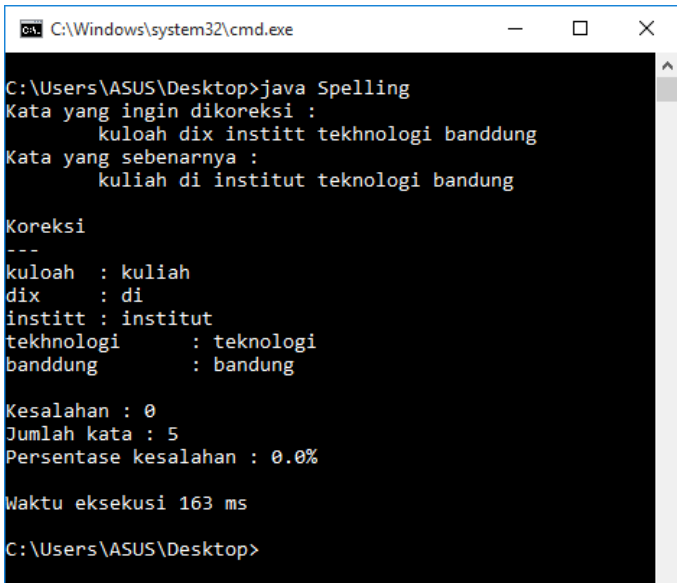


Gambar 3 : Implementasi Auto-Correct pada CMD
Sumber : dokumentasi pribadi

V. Analisis

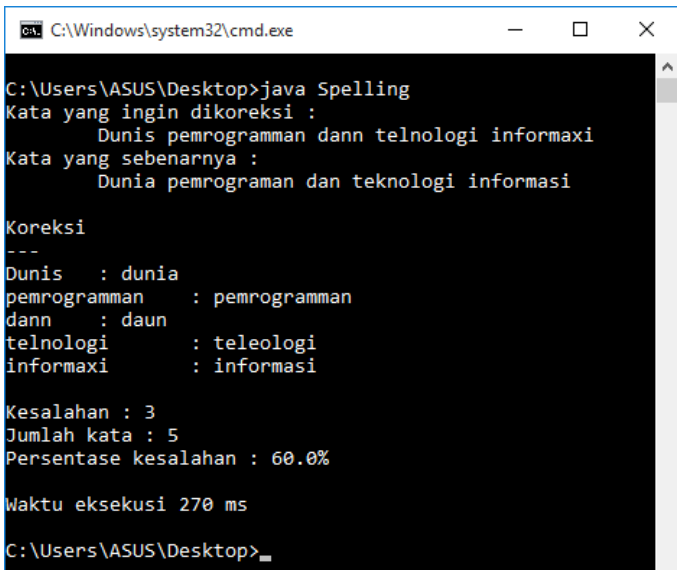
A. Percobaan

Untuk mengetahui seberapa besar akurasi auto-correct yang telah diimplementasikan, maka dilakuakan dimodifikasi pada program. Modifikasi program berupa penambahan input dari pengguna yang digunakan sebagai perbandingan kata yang diharapkan oleh pengguna.

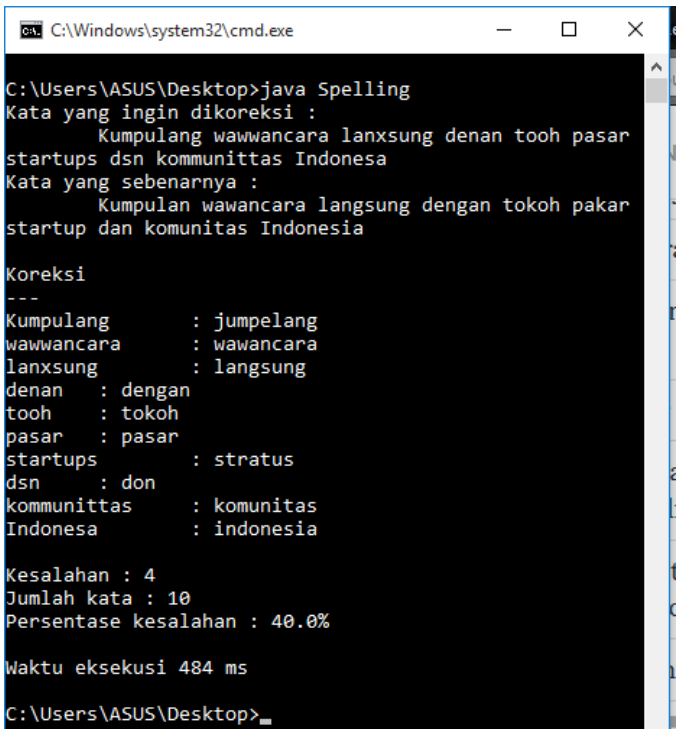


Gambar 4 : Contoh Modifikasi Program
Sumber : dokumentasi pribadi

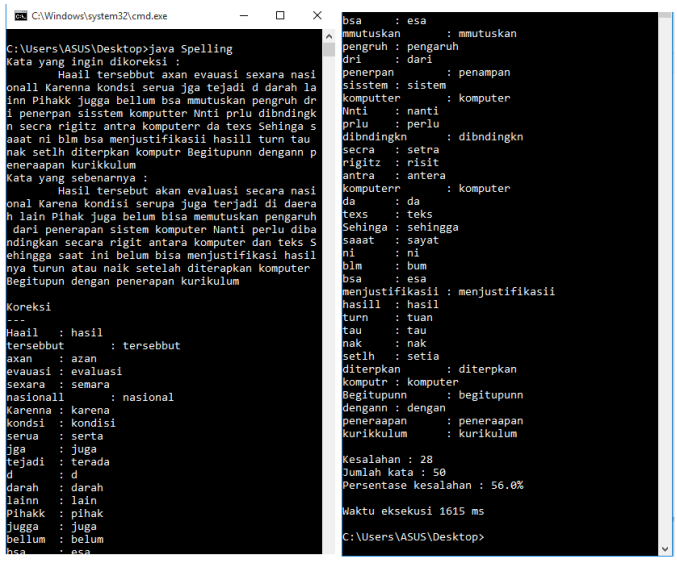
Pengujian tingkat keakuratan *auto-correct* yang telah diimplementasikan dilakukan dengan cara melakukan input sebanyak *n* kata kemudian dilakukan analisis pada tiap nilai *n*.



Gambar 5 : Contoh Implementasi dengan *n = 5*
Sumber : Dokumentasi pribadi



Gambar 6 : Contoh Implementasi dengan n = 10
Sumber : Dokumentasi pribadi



Gambar 7 : Contoh Implementasi dengan n = 50
Sumber : Dokumentasi pribadi

B. Analisis Percobaan

n (jumlah kata)	Kesalahan	Persentase kesalahan	Waktu eksekusi (ms)
5	3	60%	270
10	4	40%	484
50	28	56%	1615

Table 2 : Analisis Hasil Implementasi

Pada tiap nilai *n*, besarnya kesalahan berbanding lurus dengan nilai *n*. Namun hal yang menarik adalah persentase kesalahan untuk setiap *n* tidak berbanding lurus. Hal ini menunjukkan bahwa besarnya kesalahan tidak dipengaruhi oleh seberapa banyak kata yang ingin dikoreksi. Kesalahan terjadi karena program *auto-correct* mengoreksi kata yang diinput berdasarkan kamus dasar Bahasa Indonesia, sehingga kata yang diiharapkan belum tentu ada dalam database kamus dasar Bahasa Indonesia tersebut. Kesalahan juga terjadi karena kata yang ingin dikoreksi memiliki nilai Levenshtein Distance yang relatif besar ketika dibandingkan dengan kata yang diharapkan, namun terdapat sebuah kata dalam database kamus dasar Bahasa Indonesia yang bernilai Levenshtein Distance-nya lebih kecil sehingga terjadi kesalahan.

Pada tidap nilai *n*, waktu eksekusi berbanding lurus dengan nilai *n*. Semakin besar nilai *n*, semakin lama pula waktu eksekusi program. Hal ini disebabkan karena program yang dibuat menerapkan metode iteratif setiap kata dalam database kamus Bahasa Indonesia yang memiliki 28526 kata. Selain itu pencarian string yang digunakan adalah *brute* force yang memiliki kompleksitas O(mn) untuk kasus terburuknya. Prosesnya adalah setiap kata dicocokkan dengan kata yang ingin dikoreksi dan setiap proses pencocokkan dicari pula nilai Levenshtein Distance-nya. Jika kata yang ingin dikoreksi terdapat pada kamus dasar Bahasa Indonesia, maka tidak perlu dikoreksi. Jika tidak ditemukan, akan dipilih nilai Levenshtein Distance terbaik yang memungkinkan.

C. Pengembangan Dari Hasil Percobaan

Yang dapat dilakukan untuk mengembangkan program *auto-correct* antara lain :

1. Mengubah metode pencarian string dari *brute force* menjadi algoritma Knuth-Morris-Pratt yang memiliki kompleksitas waktu O(m+n) sehingga waktu pencarian akan jauh lebih cepat.
2. Mengupdate database kamus dasar Bahasa Indonesia sehingga kesalahan dapat diminimalisir.

VI. Simpulan

Algoritma *brute force*, dengan kompleksitas O(mn), kurang mangkus untuk diterapkan pada program *auto-correct* karena membutuhkan waktu yang lama ketika melakukan pencocokan string dalam database kata. Terlebih ketika database kata tersebut berukuran besar.

Algoritma Levenshtein Distance, dengan kompleksitas O(*n*), merupakan algoritma yang dapat mencari nilai perbedaan dari dua buah string, sehingga cocok untuk diterapkan pada program dengan fitur *auto-correct.*

Ucapan Terimakasih

Penulis pertama-tama ingin mengucapkan syukur kepada Tuhan Yang Maha Esa karena rahmat dan berkat-Nya yang selalu menyertai penulis hingga pembuatan makalah ini selesai. Penulis juga ingin berterima kasih kepada kedua orang tua penulis yang selalu memberi *support* dan semangat kepada penulis. Tak lupa penulis ucapkan terima kasih kepada Bapak Rinaldi Munir dan Ibu Nur Ulfa Maulidevi karena melalui pengjarannya, penulis dapat memahami konsep algoritma termasuk didalamnya algoritma pencarian string yang menjadi dasar makalah Strategi Algoritma IF2211.

Referensi

[1] Munir, Rinaldi, "Strategi Algoritma", Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2009.

[2] Navarro, Gonzalo, "A Guided Tour to Approximate String Matching", Dept. of Computer Science, University of Chile, 2001.

[3] Hjelmqvist Sten, "Fast, Memory Efficient Levenshtein Algorithm", http:/www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm, diakses tanggal 7 Mei 2016 pukul 15.00 WIB..

[4] Norvig, Peter, "How to Write a Spelling Corrector", http://norvig.com/spell-correct.html, diakses tanggal 7 Mei 2016 pukul 21.00 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.

Bandung, 8 Mei 2016

Adam Rotal Yuliandaru

13514091