Automatic Spelling Correction Menggunakan Levenshtein Distance pada Algoritma Brute Force Pattern Matching

Muhammad Bangkit Dwi Cahyono - 13521055
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): bangkitdc@gmail.com

***Abstract*—Peningkatan akurasi teks menjadi hal yang penting dalam berbagai aplikasi yang melibatkan pemrosesan teks, seperti pencarian informasi, pemrosesan bahasa alami, dan sistem kecerdasan buatan. Salah satu aspek yang penting dalam meningkatkan akurasi teks adalah dengan melakukan koreksi ejaan otomatis. Koreksi ejaan otomatis bertujuan untuk mengidentifikasi dan memperbaiki kata-kata yang salah eja dalam sebuah teks. Makalah ini mempelajari bagaimana cara menggunakan prinsip *pattern matching* dalam mendeteksi adanya kata salah eja pada suatu teks dari suatu masukan. Algoritma yang digunakan berdasar kepada algoritma *brute force pattern matching* dengan menggunakan *levenhstein distance* yang dimodifikasi agar pencarian dapat berjalan sinkron selagi pengguna mengetikkan teksnya.**

Keywords—brute force, pattern matching, autocorrect, spell check, levenhstein distance.

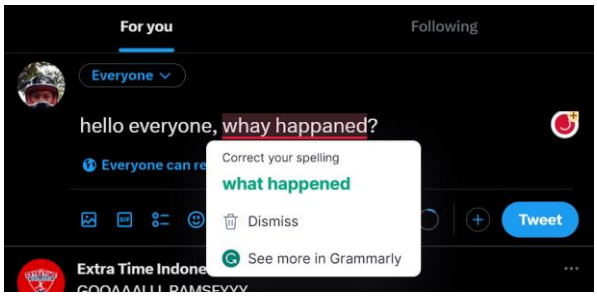
I. PENDAHULUAN

Di era digital sekarang ini, orang-orang dapat saling berkomunikasi dengan sangat mudah melalui internet. Salah satu bentuk komunikasi yang populer adalah melalui tulisan atau teks, baik melalui pesan singkat, email, media sosial, atau platform komunikasi lainnya. Namun, dalam proses penulisan teks, sering kali terjadi kesalahan ejaan yang tidak disengaja. Kesalahan ejaan ini dapat disebabkan oleh berbagai faktor, seperti kesalahan pengetikan, kurangnya perhatian, atau kurangnya pengetahuan tentang aturan ejaan.

Ketika teks yang salah eja digunakan dalam konteks yang penting, seperti dalam dokumen resmi, artikel ilmiah, atau komunikasi bisnis, kesalahan ejaan tersebut dapat mengurangi kualitas, kejelasan, dan profesionalisme teks tersebut. Oleh karena itu, diperlukan solusi yang efektif untuk melakukan koreksi ejaan secara otomatis guna meningkatkan akurasi dan kualitas teks.

Dalam konteks tersebut, teknik koreksi ejaan otomatis menjadi sangat relevan. Koreksi ejaan otomatis adalah proses mengidentifikasi kata-kata yang salah eja dalam sebuah teks dan mengusulkan kata-kata yang benar berdasarkan aturan ejaan yang berlaku. Metode-metode koreksi ejaan otomatis telah dikembangkan dan diterapkan dalam berbagai bahasa dan aplikasi, termasuk dalam bahasa Indonesia.

Dalam makalah ini, penulis ingin mengaplikasikan suatu prinsip yang telah dipelajari di mata kuliah IF2211 Strategi Algoritma, yaitu prinsip *string matching*. Dalam aplikasinya, algoritma yang digunakan adalah modifikasi dari algoritma *brute force pattern matching* dengan cara menambahkan perhitungan *Levenshtein Distance* pada setiap pengecekan string. Tujuan dari penggunaan *Levenshtein Distance* ini adalah agar pencarian ejaan kata yang salah tidak hanya kata yang eksak sama, tetapi bisa juga dengan kata dengan panjang *string* yang berbeda.



Gambar 1.1 Contoh fitur autocorrection pada aplikasi Grammarly
Sumber: dokumen penulis

II. TEORI DASAR

A. *Brute Force*

Algoritma *brute force* adalah suatu algoritma yang digunakan untuk pemecahan suatu masalah dengan pendekatan lempang (*straightforward*). Algoritma ini disusun berdasarkan pada pernyataan pada persoalan dan/atau definisi atau konsep yang dilibatkan. Algoritma ini dilakukan dengan sangat sederhana, langsung, dan jelas caranya.

Algoritma *brute force* pada umumnya tidak mangkus karena membutuhkan volume komputasi yang besar dan waktu penyelesaian yang lama. Oleh karena itu, algoritma ini lebih cocok digunakan untuk persoalan yang ukuran masukannya (n) kecil. Biasanya algoritma ini digunakan sebagai pembanding dengan algoritma lain yang lebih mangkus.

Namun, di samping kekurangan-kekurangannya, algoritma *brute force* memiliki kelebihan, yakni hampir semua persoalan dapat diselesaikan dengan algoritma ini. Selain itu, algoritma ini sangat mudah dimengerti dan sederhana. Dengan demikian, orang-orang yang tidak terlalu mendalami algoritma akan lebih mudah untuk mengerti persoalan-persoalan yang ada.

B. *Pattern Matching*

Pattern matching, atau pencocokan pola (dalam hal ini adalah *string*) merupakan proses mencari kecocokan *string* atau *substring* tertentu dalam sebuah teks atau rangkaian karakter yang lebih besar. Tujuan utama *pattern matching* adalah untuk mengidentifikasi kemunculan atau keberadaan suatu pola yang ditentukan dalam sebuah teks atau *string*. Sebagai contoh, apabila diberikan teks "ibukota Indonesia adalah Jakarta", apabila diberikan *pattern* "kota", maka harapannya akan mengembalikan indeks hasil yaitu 3 yang menandakan *pattern* ditemukan awal pada indeks ke-3 (asumsi indeks dimulai dari 0).

Pencocokan *string* sering digunakan pada kehidupan sehari-hari, seperti *search engine* (seperti Google) untuk mencari kata yang hendak dicari dan akan menampilkan kata eksak atau mendekati pada *website* yang tersedia di internet. Selain itu bisa juga dilakukan analisis citra untuk bidang forensik, misal untuk pencocokan sidik jari. Kemudian, bisa juga dimanfaatkan dalam bidang bioinformatika pada pencocokan rantai DNA.

Metode pada *pattern matching* beragam, misalnya algoritma *brute force pattern matching*, algoritma Knuth-Morris-Pratt (KMP), dan algoritma Boyer-Moore (BM). Pada makalah ini, penulis menggunakan algoritma *brute force pattern matching*.

C. *Brute Force Pattern Matching*

Algoritma *brute force pattern matching* adalah algoritma pencocokan *string* dengan pendekatan *brute force*. Pencarian dilakukan untuk mengecek huruf per huruf dalam suatu *string*. Langkah-langkah dari algoritma ini adalah sebagai berikut.

1. Pencarian dilakukan dengan sebuah iterasi dimulai dari indeks pertama pada teks T.
2. Pada setiap iterasi dilakukan pengecekan apakah pola P (*string* yang diuji) dimulai pada indeks tersebut di teks T.
3. Jika pola P ditemukan, kembalikan indeks dan iterasi dihentikan.
4. Jika pola P tidak ditemukan, maka indeks pencarian pada teks T akan dimajukan satu per satu. Iterasi dihentikan jika indeks pencarian sudah melebihi dari panjang teks T dikurang panjang pola P.

Berikut ini adalah *pseudocode*-nya (indeks dimulai dari 0).

```
function brute(input t, p: string) -> integer
{menghasilkan indeks jika ketemu, -1 jika tidak}
Deklarasi
   m, n, i, j : integer
Algoritma
   n <- t.length()
   m <- p.length()
   for i <- 0 to (n - m) do
        j <- 0
        while (j < m) and (t[i + j] = p[j]) do
                j ++
        if (j = m) then
                return i // match at i

   return -1 // no match
```

Ilustrasi dari implementasi algoritma di atas adalah sebagai berikut.

```
Teks: NOBODY NOTICED HIM
Pattern: NOT

  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Gambar 2.1 Ilustrasi pattern matching menggunakan brute force
Sumber: slide perkuliahan brute force bag.1

D. *Levenshtein Distance*

Levenshtein Distance adalah sebuah metrik *string* yang digunakan untuk mengukur perbedaan dari dua sekuens/ barisan. Algoritma ini mengukur jumlah minimum perubahan karakter yang diperlukan untuk mengubah suatu kata menjadi kata lainnya.

Terdapat tiga jenis *edit operation* yang dilakukan untuk penerapan konsep *Levenshtein Distance*, yaitu sebagai berikut.

1. *Insertion* (Penyisipan)

   Operasi ini melibatkan penambahan satu karakter ke dalam sebuah *string* untuk mengubahnya menjadi *string* lain. Misalnya, jika ingin mengubah kata "cat" menjadi "cart", harus ada penyisipan karakter 'r' di antara 'a' dan 't'.

c	a	t

c	a	**r**	t

Gambar 2.2 Ilustrasi insertion pada levenshtein distance
Sumber: dokumen penulis

2. *Deletion* (Penghapusan)

   Operasi ini melibatkan penghapusan satu karakter dari sebuah *string* untuk mengubahnya menjadi *string* lain. Misalnya, jika kita ingin mengubah kata "book" menjadi "boo", kita harus menghapus karakter 'k'.

b	o	o	**k**

b	o	o

Gambar 2.3 Ilustrasi deletion pada levenshtein distance
Sumber: dokumen penulis

3. *Substitution* (Pergantian)

   Operasi ini melibatkan mengganti satu karakter dengan karakter lain di dalam sebuah *string* untuk mengubahnya menjadi *string* lain. Misalnya, jika kita ingin mengubah kata "cold" menjadi "cord", kita harus mengganti karakter 'l' dengan 'r'.

c	o	**l**	d

c	o	**r**	d

Gambar 2.4 Ilustrasi substitution pada levenshtein distance
Sumber: dokumen penulis

Dalam algoritma *Levenshtein Distance*, kumpulan operasi ini digunakan untuk menghitung jumlah minimum perubahan karakter yang diperlukan untuk mengubah satu *string* menjadi *string* lainnya. Dalam hal ini, jarak *Levenshtein* adalah jumlah minimum operasi penyisipan, penghapusan, dan pergantian yang diperlukan.

Misalkan terdapat dua buah *string*, *string a* dan *string b* dengan panjang *string* dinotasikan dalam |a| dan |b|, dapat dihitung *Levenshtein distance* menggunakan fungsi sebagai berikut.

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{jika } |b| = 0, \\ |b| & \text{jika } |a| = 0, \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) & \text{jika } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}(\operatorname{tail}(a), b) \\ \operatorname{lev}(a, \operatorname{tail}(b)) \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) \end{cases} & \text{lainnya.} \end{cases}$$

Gambar 2.5 Fungsi Levenshtein Distance
Sumber: https://id.wikipedia.org/wiki/Jarak_Levenshtein

Dengan *tail* dari *string* adalah string dari semua kecuali karakter pertama pada *string* tersebut. Berikut ini adalah implementasi dari fungsi *Levenshtein Distance* (indeks dimulai dari 0)

```
function levenshtein_dist(input word1, word2: string)
-> integer
{untuk setiap i dan j, dp[i][j] akan menyimpan jarak Levenshtein distance antara i karakter pertama dari word1 dan j karakter pertama dari word2}
Deklarasi
   m, n, i, j, cost : integer
   dp               : matrix of integer [0..m][0..n]
Algoritma
   m <- word1.length()
   n <- word2.length()
   if (m = 0) then return n
   if (n = 0) then return m
   dp <- int[m][n]

   for i <- 0 to m do
        dp[i][0] <- i
   for j <- 0 to n do
        dp[0][j] <- j

   // hitung jarak
   for i <- 1 to m do
        for j <- 1 to n do
        cost <- 0 if (word1[i-1] = word2[j-1]) else 1
        dp[i][j] <- min (
                dp[i-1][j] + 1, // deletion
                dp[i][j-1] + 1, // insertion
                dp[i-1][j-1] + cost) // substitution

   return dp[m][n]
```

Sebagai ilustrasi, akan dihitung *Levenshtein Distance* dari dua *string*, yaitu “Saturday” dan “Sunday”. Langkah awalnya adalah dengan membuat matriks sebagai berikut.

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1								
u	2								
n	3								
d	4								
a	5								
y	6								

Pengisian sel pada matriks dilakukan dari indeks (1, 1). Dicari nilai minimum dengan fungsi *Levenshtein Distance*. Perhitungan pada (1,1) sebagai berikut.

$$lev(i-1,j)+1 = 1+1 = 2$$
$$lev(i,j-1)+1 = 1+1 = 2$$
$$lev(i-1,j-1)+\cos t = 0+0 = 0$$

Nilai minimum dari *insertion, deletion,* dan *substitution* tersebut adalah 0, sehingga pada matriks (1,1) akan diisi dengan angka 0. Kemudian lakukan hal ini hingga matriks terisi penuh seperti berikut.

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Nilai yang diambil sebagai hasil akhir perhitungan adalah pada sel paling kanan bawah (dp[m][n]). Dalam persoalan ini nilai tersebut adalah 3, maka dapat disimpulkan bahwa *Levenshtein Distance* dari kata "Saturday" dan "Sunday" adalah 3.

E. Ejaan Kata dan Konteks Kalimat

Kata adalah satuan gramatikal bebas yang paling kecil. Kata dapat berdiri sendiri dan dapat membentuk suatu makna bebas. Kata merupakan dua macam satuan, yakni satuan fonologik (bunyi) dan satuan gramatik. Sebagai satuan fonologik, kata terdiri dari satu atau beberapa suku (kata). Misalnya: belajar terdiri dari tiga suku kata yakni: be, la, jar. Sebagai satuan gramatik, kata dapat terdiri dari satu atau beberapa morfem seperti yang telah dijelaskan sebelumnya.

Kalimat adalah satuan gramatikal yang terdiri dari rangkaian kata yang dapat berdiri sendiri dan memberikan makna yang lengkap. Kalimat adalah satuan bahasa terkecil yang mengungkapkan suatu pokok pikiran. Sehingga, berbeda dengan frasa, kalimat akan memiliki keterhubungan subjek dan predikat. Berbeda dengan klausa pula, kalimat memiliki intonasi akhir. Boleh dikatakan bahwa kalimat adalah satuan gramatikal yang paling utuh jika dibandingkan dengan satuan lain di bawahnya.

Susunan kata yang tidak sesuai dengan konteks kalimat dapat mengubah makna kalimat secara keseluruhan. Ejaan kata dan konteks kalimat merupakan dua hal yang wajib diperhatikan dalam penyampaian informasi dalam proses komunikasi. Hal ini dilakukan untuk menghindari kesalahan pemaknaan dan pengertian terhadap informasi.

III. PEMBAHASAN

Untuk menyelesaikan permasalahan kesalahan ejaan kata Bahasa Indonesia, dilakukan modifikasi algoritma agar sesuai dengan tujuan yang hendak didapat. Selain itu, dibutuhkan sumber data mengenai kata-kata yang tersedia dalam kamus Bahasa Indonesia itu sendiri.

A. Pembuatan Daftar Kata Bahasa Indonesia

Sebelum diolah dalam beberapa rangkaian algoritma, perlu disiapkan data acuan yang berperan sebagai kamus data Bahasa Indonesia yang akan menjadi acuan pengejaan kata. Sumber data diambil dari https://github.com/kirralabs/indonesian-NLP-resources pada *leipzig indonesian sentence collection* dengan dataset Wikipedia (2021) sebanyak kurang lebih 600.000 kata.

Dataset ini berbentuk *txt file* yang kemudian diolah menjadi data struktur utama. Penulis menggunakan *tupple* untuk struktur ini dengan format (kata, *count*). Kata berupa string dan *count* berupa integer. *Count* adalah banyaknya kata tersebut muncul di Wikipedia, ini akan dipakai untuk pencocokan *string* di algoritma yang akan dijelaskan selanjutnya. Berikut adalah algoritma untuk pengolahan data dalam *pseudocode*.

```
with open(dataset, encoding='utf-8') as file:
    for line in file then
        data <- line.split('\t')
        word <- data[1].strip()
        number <- data[2].strip()
        dictionary.append((word, number))
```

B. Modifikasi Algoritma Brute Force Pattern Matching

Pada algoritma *brute force pattern matching* yang biasa, program hanya mengembalikan indeks pertama ditemukannya pola. Hal ini tidak cocok dengan persoalan yang ingin diselesaikan. Oleh karena itu, perlu adanya modifikasi pada kode program tentunya dengan memanfaatkan *Levenshtein Distance.*

Berikut adalah kode modifikasi dari algoritma utama *brute force pattern matching* dalam *pseudocode.*

```
function spell_checker(input s : string) -> (suggestions, corrected : array[])
Deklarasi
    index                                : integer
    suggestions, corrected, words        : array []
Algoritma
    words <- re.findall(r'\w+|[^\w\s]', s)
    index <- 0

    for word in words do
        if (ada di dictionary and valid) then
                closest <- find_closest(word)
                sim_ratio <- calc_sim(word, closest)
                if sim_ratio >= 0.75 then
                        corrected.append(closest)
                        suggestions.append((closest, index))
                else
```

```
                    corrected.append(word)
        else
                corrected.append(word)
        index <- word.length() + 1

    return suggestions, ' '.join(corrected)
```

Dalam implementasi tersebut dapat dilihat bahwa adanya iterasi kalimat masukan dari pengguna (*s*) lalu penulis memanfaatkan *regex* agar ter-*filter* menjadi kata-kata yang tidak mengandung tanda baca seperti '?', '!', dan lainnya. Selain itu juga menolak spasi agar tidak ikut diproses dalam algoritma.

Pada implementasinya, penulis menginginkan adanya kemiripan kata setidaknya 75%, misalnya jika ada kata "sa**i**a" dan "sa**y**a" maka ada perbedaan 1 dari 4 karakter (75%) akan tetap ikut dilakukan pencarian ejaan yang benar dalam database. Dengan 75% kecocokan diharapkan ejaan akan lebih akurat untuk *edge cases*.

Untuk mendapatkan *closest match* atau kata terdekat, selain memanfaatkan algoritma *Levenshtein Distance*, penulis juga menggunakan data *count* atau seberapa sering kata tersebut digunakan oleh manusia. Misalnya pada kata "kamu" dan "kami", jika ternyata pengguna memasukkan kata "kame", maka program akan bingung untuk memilih antara "kamu" atau "kami". Oleh karena itu, penulis menambahkan satu atribut, yaitu *count*. Dengan demikian, semisal "kamu" lebih sering dipakai daripada "kami", maka akan *autocorrect* ke "kamu". Hal ini tentu menjadi batasan dari program karena "kami" jadi tidak menjadi solusi dalam permasalahan ini karena program tidak dapat mendeteksi frasa/ makna kalimat secara keseluruhan.

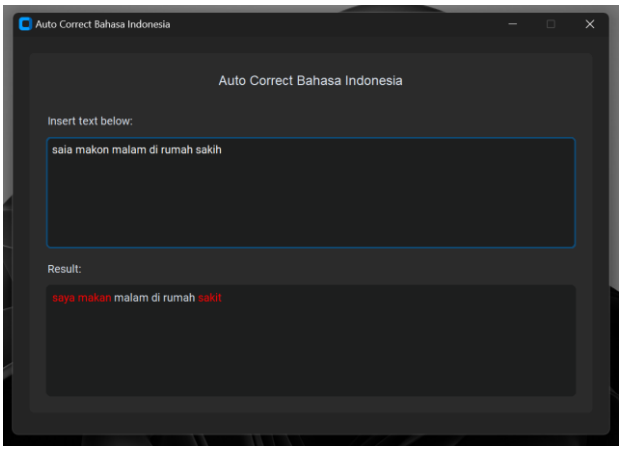
Untuk mempercepat perhitungan algoritma digunakan abs(w.length() – s.length()) <= 3, artinya maksimal hanya berbeda 3 huruf saja. Ini dilakukan untuk menambah performa aplikasi karena jika setiap kata harus menghitung ke 600.000 kata lainnya, akan memakan waktu lama.

Berikut adalah *pseudocode* untuk fungsi *find_closest*.

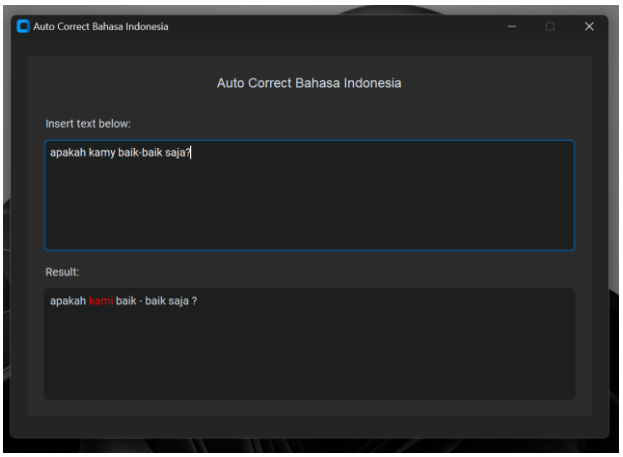
```
function find_closest(input s : string) -> string
Deklarasi
   closest                    : string
   min_dist, max_count, dist  : integer
Algoritma
   min_dist <- -infinity
   max_count <- 0
   for w, count in dictionary do
        if abs(w.length() - s.length()) <= 3 then
                dist <- lev_dist(s, w)
                if dist < min_dist then
                        max_count <- count
                        min_dist <- dist
                        closest <- w
                else if dist = min_dist and count >= max_count then
                        max_count <- count
                        min_dist <- dist
                        closest <- w

   return closest
```

IV. Hasil Eksperimen

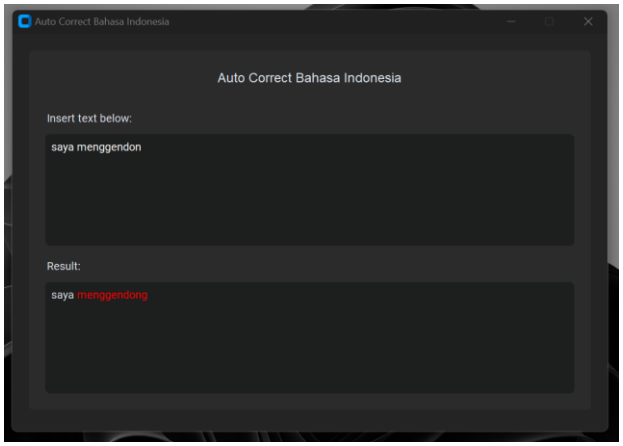
Eksperimen dilakukan dengan cara membuat aplikasi *desktop* menggunakan *framework* tkinter. Aplikasi bernama Auto Correct Bahasa Indonesia dan dapat diunduh di https://github.com/bangkitdc/spell-checker. Aplikasi yang dibuat cukup sederhana, pengguna cukup memberikan masukan kalimat, nanti otomatis program akan menampilkan ejaan yang benar di bawahnya.



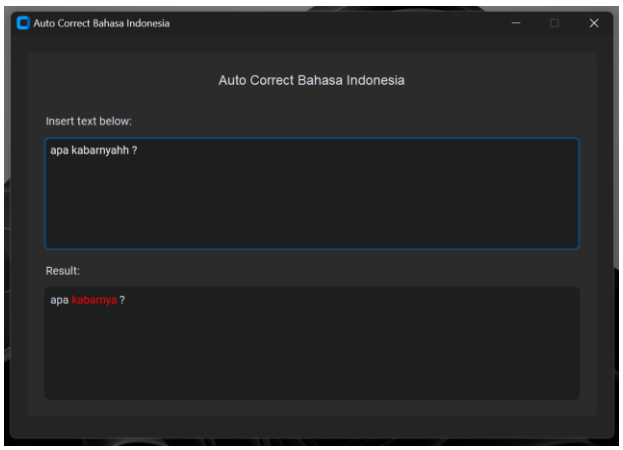
Gambar 4.1 Contoh pengujian kalimat (substitution)
Sumber: dokumen penulis



Gambar 4.2 Contoh pengujian kalimat (count tertinggi)
Sumber: dokumen penulis



Gambar 4.3 Contoh pengujian kalimat (insertion)
Sumber: dokumen penulis



Gambar 4.4 Contoh pengujian kalimat (deletion)
Sumber: dokumen penulis

Dari beberapa pengujian tersebut, dapat dilihat bahwa algoritma yang dipakai cukup efektif. Selain itu juga waktu untuk mendapatkan hasil lumayan cepat dengan konsiderasi data yang diiterasi adalah sebanyak 600.000 kata. Dengan adanya tambahan algoritma *greedy*, waktu eksekusi program dapat lebih cepat.

Pada *Gambar 4.1*, dapat dilihat bahwa *autocorrection* bekerja dengan baik dan dapat memberikan hasil kalimat dengan ejaan yang sudah diperbaiki. Aplikasi juga dapat memberi tahu bagian mana yang salah ejaannya sehingga pengguna tidak bingung. Ini berarti konsep *substituion* pada *Levenshtein Distance* bekerja dengan baik.

Pada *Gambar 4.2*, dapat dilihat bahwa ternyata kata "kami" lebih sering muncul dari kata "kamu", sehingga yang menjadi ejaan yang tepat adalah "kami". Ini memang menjadi batasan dari aplikasi yang penulis buat.

Pada *Gambar 4.3*, dapat dilihat bahwa konsep *insertion* pada *Levenshtein Distance* bekerja dengan baik. Lalu pada *Gambar 4.4*, dapat dilihat juga bahwa konsep *deletion* juga bekerja dengan sangat baik.

V. KESIMPULAN

Dengan memanfaatkan konsep yang ada pada mata kuliah IF2211 Strategi Algoritma, kita dapat menyelesaikan berbagai macam persoalan yang ada di dunia nyata. Dalam kasus ini, pemanfaatan algoritma *brute force* dapat membantu kita untuk menyelesaikan persoalan ejaan kata yang salah pada teks kalimat. Dengan pemanfaatan algoritma *greedy* dan juga *Levenshtein Distance*, persoalan bisa dilakukan dengan performa yang lebih baik.

Dengan memodifikasi algoritma-algoritma yang sudah ada, kemudian menambahkan beberapa konsep tambahan dan beradaptasi dengan permasalahan yang ada, maka kita dapat menyelesaikan persoalan-persoalan yang ada. Algoritma yang ada dapat langsung diaplikasikan dalam bentuk aplikasi asli menggunakan bahasa pemrograman Python dengan bantuan *framework* tkinter.

LINK VIDEO YOUTUBE

https://youtu.be/SAYtKwg55M8

UCAPAN TERIMA KASIH

Penulis ingin menyampaikan rasa syukur kepada Tuhan Yang Maha Esa karena atas rahmat dan karunia-Nya, makalah ini dapat terselesaikan dengan baik dan tepat waktu. Penulis juga ingin mengucapkan terima kasih kepada seluruh pihak yang telah membantu pengerjaan makalah ini, terutama kepada dosen mata kuliah IF2211 Strategi Algoritma kelas 01, yakni Bapak Dr. Ir. Rinaldi Munir, M.T. Tidak lupa penulis ingin menyampaikan terima kasih kepada pihak dan sumber yang dijadikan referensi dalam pembuatan makalah ini.

REFERENSI

[1] Munir, Rinaldi. 2022. *Algoritma Brute Force (Bagian 1)*. Merupakan slide bahan ajar perkuliahan. Diakses pada tanggal 20 Mei 2023.

[2] Tim Dosen IF2211 Strategi Algoritma. 2021. *Pencocokan String (String/Pattern Matching)*. Merupakan slide bahan ajar perkuliahan. Diakses pada tanggal 20 Mei 2023.

[3] arvinpdmn. 2019. *Levenshtein Distance*. Diakses pada tanggal 22 Mei 2023.

[4] Nam, Ethan. 2019. *Understanding the Levenshtein Distance Equation for Beginners*. Diakses pada tanggal 22 Mei 2023.

[5] Dhanawaty, N.M., Satyawati, M.S., Widarsini, N.P.N. (2017). *Pengantar linguistik umum*. Denpasar: Pustaka Larasan. Diakses pada tanggal 22 Mei 2023.

[6] Kamus Besar Bahasa Indonesia. 2023. Diakses melalui website https://kbbi.web.id/ pada 22 Mei 2023.

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, 22 Mei 2023

Muhammad Bangkit Dwi Cahyono 13521055