Perbandingan Algoritma *Brute Force* dan Pemrograman Dinamis pada Persoalan *Autocorrect*

Brianaldo Phandiarta - 13520113
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520113@std.stei.itb.ac.id

***Abstract*—*Autocorrect* merupakan salah satu teknologi yang dapat meminimalisir kesalahan manusia. Fitur *autocorrect* harus dibuat dengan algoritma yang efektif dan efesien. Dalam makalah ini, akan dibahas perbandingan efesiensi dari penggunaan algoritma *brute force* dan pemrograman dinamis. Dari hasil implementasi, pendekatan menggunakan pemrograman dinamis lebih efisien dengan kompleksitas waktu O(*mn*) dibandingkan dengan algoritma *brute force* dengan kompleksitas waktu O(3^k) dengan *k* merupakan panjang maksimal dari *m* dan *n*.**

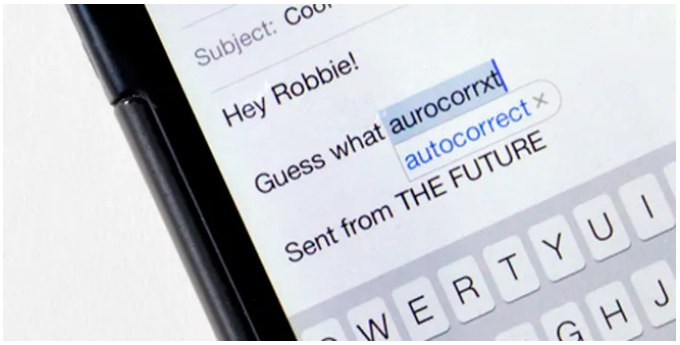
Keywords—pemrograman dinamis; brute force; autocorrect; Levenshtein Distance; Wagner-Fischer

I. PENDAHULUAN

Teknologi telah berkembang secara pesat seiiring dengan kemajuan ilmu pengetahuan. Perkembangan teknologi yang sangat pesat ini telah memberikan manfaat dalam berbagai bidang, salah satunya efektivitas dalam pekerjaan manusia. Dengan teknologi, manusia tidak perlu menghabiskan waktunya untuk mengerjakan hal-hal yang repetitif yang dapat dilakukan secara otomatis. Selain itu, manusia juga tidak perlu mengkhawatirkan kesalahan-kesalahan minor karena kesalahan minor tersebut seharusnya dapat diatasi menggunakan teknologi.

Salah satu teknologi yang dapat mengurangi kesalahan-kesalahan minor karena ketidaktelitian manusia adalah teknologi *autocorrect*. *Autocorrect* membantu manusia ketika terjadi kesalahan pengejaan kata atau *misspelling*. Seiring dengan perkembangan teknologi, teknologi *autocorrect* juga semakin berkembang. Perkembangan ini tidak terbatas pada kecepatan saja, namun ketepatan juga.

Teknologi *autocorrect* telah digunakan pada berbagai aplikasi, khususnya aplikasi yang berfokus pada pengetikan seperti aplikasi pengolah kata (Microsoft Word). Teknologi *autocorrect* ini juga digunakan pada berbagai *integrated development environment* (IDE) dengan memberikan rekomendasi variabel ataupun fungsi untuk mempercepat pengerjaan seorang *coder*. Selain itu, papan tombol (*keyboard*) pada telepon pintar juga terintegrasi dengan teknologi *autocorrect*.



Gambar 1.1 *Autocorrect* pada telepon pintar
Sumber: https://www.wired.com/2014/03/make-autocorrect-suck-less/

II. DASAR TEORI

A. *Autocorrect*

Autocorrection atau dikenal juga sebagai *autocorrect* adalah suatu teknologi atau fitur yang dapat memeriksa dan memperbaiki kesalahan penulisan kata secara otomatis. Secara umum, tujuan *autocorrect* adalah memperbaiki kesalahan umum pengetikan sehingga dapat menghemat waktu bagi pengguna. Namun, terdapat banyak kontroversi penggunaaan fitur *autocorrect* yang tidak akan dibahas pada makalah ini oleh penulis.

Pada awal penciptaannya, fitur *autocorrect* diciptakan melalui pihak ketiga. Sekarang, fitur ini sudah diimplementasikan langsung pada sistem operasi seperti MacOS dan Windows. Fitur ini juga banyak digunakan pada aplikasi pesan teks maupun program computer seperti Microsoft Word.

Pada fitur *autocorrect* konvensional, hanya dilakukan perhitungan *edit distance* dari dua buah kata, yaitu kata yang ingin dicocokkan dengan kata yang terdapat dalam kamus. Seiiring dengan perkembangan zaman, terdapat berbagai alternatif yang lebih efektif dengan memanfaatkan data yang berisi kata-kata yang *common* atau yang sering digunakan.

B. *String*

String dalam bahasa pemrograman komputer adalah sebuah deret simbol atau barisan karakter. Sebuah string umumnya

dianggap sebagai tipe data dan sering diimplementasikan sebagai struktur data *array* dari karakter. Dalam bahasa formal, yang digunakan dalam logika matematika dan ilmu komputer teorits, sebuah string adalah sekuens dari simbol yang merupakan himpunan bagian dari alfabet.

Terdapat konsep string yang umum digunakan pada algoritma yang memproses string. Misalkan, terdapat *S* yang merupakan string dengan Panjang *m*.

$$S = x_0x_1..x_{m-1}$$

Prefiks atau subbagian awal dari string dapat direpresentasikan sebagai berikut.

$$S[0..k] = x_0x_1..x_k$$

Sufiks atau subbagian akhir dari string dapat direpresentasikan sebagai berikut.

$$S[k..m-1] = x_kx_{k+1}..x_{m-1}$$

Dengan *k* merupakan bilangan bulat diantara 0 dan *m-1*.

Berikut merupakan contoh dari string.

TABLE I. ILUSTRASI STRING

i	0	1	2	3	4	5
***S*[*i*]**	S	T	R	I	N	G

TABLE II. ILUSTRASI PREFFIX STRING *S*[0..3]

i	0	1	2	3
***S*[*i*]**	S	T	R	I

TABLE III. ILUSTRASI SUFFIX STRING *S*[3..5]

i	3	4	5
***S*[*i*]**	I	N	G

C. Edit distance

Dalam ilmu komputer, *edit distance* adalah jarak antara dua kata yang dihitung dari perbedaan sebuah string dengan string lainnya. Perbedaan tersebut dihitung dari banyaknya jumlah minimum operasi yang perlu digunakan untuk mengubah suatu string menjadi string lainnya. Terdapat beberapa algoritma dalam menghitung *edit distance* dari dua buah string, yaitu algoritma Levenshtein Distance, algoritma Longest Common Subsequence (LCS) Distance, algoritma Hamming Distance, algoritma Damerau-Levenshtein Distance, algoritma Jaro Distance, dan algoritma Jaro-Winkler Similarity.

Terdapat beberapa operasi yang umum digunakan pada algoritma-algorima tersebut, yaitu

1. *Deletion* atau delesi, yaitu penghapusan satu karakter pada string;
2. *Insertion* atau insersi, yaitu penambahan satu karakter ke dalam string;
3. *Substitution* atau substitusi, yaitu penggantian satu karakter dari string menjadi karakter lainnya.
4. *Transposition*, yaitu suatu operasi yang menukar posisi karakter dari suatu string dengan karakter lainnya pada string tersebut.

Perbedaan dari masing-masing algoritma tersebut adalah operasi-operasi yang boleh digunakan dalam pemrosesan string, yaitu

1. Algoritma Levenshtein Distance dapat menggunakan *deletion*, *insertion*, dan *substitution*;
2. Algoritma LCS Distance dapat menggunakan *insertion* dan *deletion*;
3. Algoritma Hamming Distance dapat menggunakan *substitution* sehingga hanya berlaku pada string dengan Panjang yang sama;
4. Algoritma Damerau-Levenshtein Distance dapat menggunakan *insertion*, *deletion*, *substitution*, dan *transposition* pada dua karakter yang bersebelahan;
5. Algoritma Jaro Distance dan Jaro-Winkler Similarity hanya dapat menggunakan *transposition*.

D. Algoritma Levenshtein Distance

Dalam teori informasi, linguistik, dan ilmu komputer, Levenshtein Distance merupukan metrik, yaitu himpunan yang memiliki definisi jarak antara elemen himpunan, untuk mengukur perbedaan antara dua barisan. Secara informal, Levenshtein Distance merupakan jarak minimun perubahan karakter dengan menggunakan operasi yang telah dijelaskan pada bagian C yang diperlukan untuk mengubah satu kata menjadi kata lainnya.

Levenshtein Distance antara dua string, a dan b dengan panjang $|a|$ dan $|b|$ dapat dihitung dengan

$$\mathrm{lev}(a,b) = \begin{cases} |a| & \text{jika } |b| = 0 \\ |b| & \text{jika } |a| = 0 \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) & \text{jika } a_0 = b_0 \\ 1 + \min \begin{cases} \mathrm{lev}(\mathrm{tail}(a), b) \\ \mathrm{lev}(a, \mathrm{tail}(b)) \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) \end{cases} & \text{lainnya} \end{cases} \quad (1)$$

Dengan tail dari string *S* merupakan suffiks *S*[1..*m-1*] dengan *m* merupakan panjang dari *S*.

Dari persamaan, tiga fungsi pertama merupakan basis dari fungsi rekursif lev(a, b). Fungsi terakhir merupakan fungsi yang digunakan ketika dilakukan operasi pada string, yaitu secara berturut-turut adalah *deletion*, *insertion*, *substitution*.

Misalkan terdapat dua buah string A dengan isi "KOPI" dan B dengan isi "COFFEE". Untuk mengubah A menjadi B, terjadi operasi *substitution* 'K' menjadi 'C', operasi *insertion* 'F' sebanyak dua kali, operasi *substitution* 'P' menjadi 'E', dan operasi *substitution* 'I' menjadi 'E'. Sehingga Lavenshtein diperoleh 5.

Untuk mengubah B menjadi A, Terjadi operasi *substitution* 'C' menjadi 'K', operasi *deletion* 'F' sebanyak dua kali, operasi *substitution* 'E' menjadi 'P', operasi *substitution* 'E' menjadi 'I'. Sehingga Lavenshtein diperoleh 5.

E. Algoritma Brute Force

Algoritma *Brute Force* merupakan algoritma yang menyelesaikan suatu persoalan menggunakan pendekatan *straightforward*. Dalam pengimplementasiannya, algoritma *brute force* didasarkan pada pernyataan pada persoalan dan definisi atau konsep yang dilibatkan.

F. Pemrograman Dinamis

Pemrograman dinamis atau *dynamic programming* adalah metode pengoptimalan matematika dan metode pemrograman komputer. Secara sederhana, dalam menyelesaikan permasalahan, pemrograman dinamis akan memecah suatu permasalahan menjadi subpermasalahan atau beberapa tahap. Berbeda dengan algoritma *Greedy*, pemrograman dinamis akan menghasilkan rangkaian keputusan.

Dalam menyelesaikan permasalahan, pemrograman dinamis mengikuti prinsip optimalitas, yaitu jika solusi total optimal, subbagian solusi sampai tahap ke-k juga optimal dengan k merupakan bilangan bulat. Dengan kata lain, pencarian solusi untuk tahap ke-k+1 dapat menggunakan hasil optimal dari tahap ke-k tanpa harus kembali ke tahap awal.

Persoalan-persoalan umum yang dapat diselesaikan atau dioptimasikan menggunakan pemrograman dinamis memiliki karakteristik sebagai berikut.

1. Persoalan yang dapat dibagi menjadi beberapa tahap dan hanya dapat diambil satu keputusan pada setiap tahapnya.
2. Setiap tahapannya, terdiri dari sejumlah status (kemungkinan masukan) yang memiliki hubungan dengan tahap tersebut.
3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
4. Ongkos pada setiap tahap kan meningkat secara teratur dengan bertambahnya jumlah tahapan.
5. Ongkos pada suatu tahap memiliki ketergantungan terhadap ongkos pada tahap-tahap yang sudah dilewati.
6. Adanya hubungan rekursif yang dapat mengidentifikasikan keputusan terbaik untuk setiap status pada tahap ke-k yang akan memberikan keputusan terbaik untuk setiap status pada tahap ke-k+1.
7. Prinsip optimalitas berlaku pada persoalan tersebut.

Persoalan pemrograman dinamis dapat dilakukan dengan menggunakan dua pendekatan, yaitu

1. Pemrograman dinamis maju (*forward* atau *up-down*)

   Pemecahan persoalan akan dilakukan dari tahap 1, 2, .., k-1, k sehingga akan mendapatkan rangkaian peubah keputusan $x_1, x_2, ..., x_{k-1}, x_k$. Dengan x merupakan status atau peubah keputusan.

2. Pemrograman dinamis mundur (*backward* atau *bottom-up*)

   Pemecahan persoalan akan dilakukan dari tahap k, k+1, …, 2, 1. sehingga akan mendapatkan rangkaian peubah keputusan $x_k, x_{k+1}, ..., x_2, x_1$. Dengan x merupakan status atau peubah keputusan.

Langkah-langkah dalam pengembangan algoritma pemrograman dinamis.

1. Menentukan karakteristik struktur solusi optimal, yaitu penentuan tahap, peubah keputusan, status, dan sebagainya.
2. Mendefinisikan secara rekursif solusi-solusi optimal, yaitu menentukan hubungan nilai optimal suatu tahap dengan solusi optimal tahap sebelumnya.
3. Melakukan perhitungan solusi optimal. Perhitungan solusi optimal dapat menggunakan tabel.
4. Melakukan rekonstruksi dari solusi optimal yang telah didapat.

G. Algoritma Wagner-Fischer

Algoritma Wagner-Fischer merupakan algoritma optimasi algoritma Levenshein Distance menggunakan pemrogramman dinamis. Algoritma ini ditemukan oleh Robert A. Wagner bersama dengan rekannya, Michael J. Fisher. Berikut merupakan *pseudocode* algoritma Wagner-Fischer.

```
function distance(input s1 : string,
                  input s2 : string,
                  input m : integer,
                  input n : integer)
{ Mengembalikan Levenshtein Distance menggunakan
  algoritma Wagner-Fischer }
KAMUS
  { Variabel }
  d : array[0..m] of array[0..n] of integer
  cost : integer
  { Fungsi Antara }
  function min(x1, x2, x3 : integer) → integer
  { Mengembalikan nilai terkecil dari x1, x2, dan
    x3}
ALGORITMA
  { Inisialisasi d }
  i traversal [1..m]
     d[i][0] ← i
  j traversal [1..n]
     d[0][j] ← j

  j traversal [1..n]
     i traversal [i..m]
       if s1[i] = s2[j] then
```

```
      cost ← 0
    else
      cost ← 1

    d[i][j] ← min(d[i - 1][j] + 1,
                  d[i][j - 1] + 1,
                  d[i - 1][j - 1] + cost)
```

Algoritma melakukan perhitungan komputasi Levenshtein Distance dengan menyimpan Levenshtein Distance sebelumnya, yaitu perbandingan prefiks dari dua buah string.

Untuk meperjelas, akan diberikan illustrasi dalam bentuk metrik. Misalkan terdapat dua buah string A dengan isi "KOPI" dan B dengan isi "COFFEE". Berikut merupakan representesi metrik dari algoritma Wagner-Fischer untuk mengubah A menjadi B.

TABLE IV. METRIK WAGNER-FISCHER A KE B

	#	C	O	F	F	E	E
#	**0**	1	2	3	4	5	6
K	1	**1**	2	3	4	5	6
O	2	2	**1**	**2**	**3**	4	5
P	3	3	2	2	3	**4**	5
I	4	4	3	3	3	4	**5**

Terjadi operasi *substitution* 'K' menjadi 'C', operasi *insertion* 'F' sebanyak dua kali, operasi *substitution* 'P' menjadi 'E', dan operasi *substitution* 'I' menjadi 'E'. Sehingga Lavenshtein diperoleh 5.

Contoh lain, berikut merupakan representasi metrik dari algoritma Wagner-Fischer untuk mengubah B menjadi A.

TABLE V. METRIK WAGNER-FISCHER B KE A

	#	K	O	P	I
#	**0**	1	2	3	4
C	1	**1**	2	3	4
O	2	2	**1**	2	3
F	3	3	**2**	2	3
F	4	4	**3**	3	3
E	5	5	4	**4**	4
E	6	6	5	5	**5**

Algoritma Wagner-Fischer memiliki kompleksitas waktu sebesar O(*mn*) dengan *m* merupakan panjang string target dan *n* merupakan panjang string sumber.

III. IMPLEMENTASI

A. *Identifikasi Masalah*

Fitur *autocorrect* membutuhkan data sebagai pembanding pengetikkan yang dilakukan oleh pengguna. Untuk menyederhanakan masalah, akan digunakan subbagian dari kamus. Selain itu, kata-kata yang dipilih juga tidak berdasarkan pada kata yang sering digunakan.

Fitur *autocorrect* juga memerlukan algoritma dalam membandingkan kemiripan string. Selain itu, penulis juga akan membandingkan berbagai algoritma. Sehingga akan dilakukan implementasi algoritma Levenshtein Distance menggunakan algoritma *Brute Force*, pemrograman dinamis, dan algoritma Wagner-Fischer. Dalam hal ini, algoritma Wagner-Fischer akan menjadi pembanding utama dengan algoritma pemrograman dinamis yang akan dibuat penulis.

Terakhir, fitur *autocorrect* harus dapat memilih kata yang paling mirip. Sehingga, akan dilakukan implementasi pemilihan kata berdasarkan Levenshtein Distance yang telah didapat.

Implementasi akan dilakukan menggunakan bahasa pemrograman python.

B. *Implementasi Kamus*

Kamus akan disimpan dalam bentuk *text file* (.txt). Kamus akan berisi seribu kata yang diambil secara acak dari kamus Bahasa Inggris. Berikut merupakan cuplikan dari kamus.txt.

```
pickle
relieved
ants
gusty
rely
tall
sudden
.
.
.
business
macabre
even
top
```

Setelah itu, dibutuhkan algoritma untuk pembacaan kamus dari *text file*. Kemudian, kamus akan disimpan dalam *set*. Berikut merupakan implementasi pembacaan kamus.

```
def readKamus():
    d = {}
    f = open("Kamus.txt", "r")
    for word in f:
        d.add(word)
    return d
```

C. *Implementasi Levenshtein Distance Menggunakan Brute Force*

Algoritma *Brute Force* merupakan algoritma *straightforward*. Sehingga, dalam implementasinya, penulis hanya perlu menuliskam fungsi Levenshtein Distance dalam bentuk kode. Berikut merupakan implementasi algoritma Levenshtein Distance menggunakan *Brute Force*.

```
def editDistance1(str1, str2, m, n):
    if m == 0:
         return n

    if n == 0:
        return m

    if str1[m - 1]== str2[n - 1]:
        return editDistance1(str1, str2, m - 1,
                             n - 1)

    return 1 + min(editDistance1(str1, str2, m,
                                 n - 1),
                   editDistance1(str1, str2, m - 1,
                                 n),
                   editDistance1(str1, str2, m - 1,
                                 n - 1)
                   )
```

D. Implementasi Levenshtein Distance Menggunakan Pemrograman Dinamis

Berbeda dengan *Brute Force*, solusi penyelesaian masalah pada setiap tahap (status) akan disimpan ke metrik ataupun *hash table*.

Langkah-langkah pemrograman dinamis:

1. Menentukan karakteristik struktur solusi optimal.
   - Misalkan terdapat 2 string, yaitu *str1* dan *str2* dengan *m* merupakan panjang *str1* dan *n* merupakan panjang *str2*.
   - Misalkan $x_{1,1}$, $x_{1,2}$, …, $x_{1,n}$, …, $x_{m,n}$ adalah perhitungan Levenshtein Distance dari prefiks *str1*[0..*i*] dan *str2*[0..*j*] dengan $x_{i,j}$.
   - Misalkan digunakan pendekatan pemrograman dinamis maju.
   - Sehingga, rute yang dilalui adalah $x_{1,1} \rightarrow x_{1,2} \rightarrow \ldots \rightarrow x_{1,j} \rightarrow \ldots \rightarrow x_{i,j}$.
   - Pada proses ini akan terdapat *m* x *n* tahap dengan status yang berhubungan dengan masing-masing tahap adalah perhitungan Levenshtein Distance dari prefiks pada tahap sebelumnya.
2. Mendefinisikan secara rekursif solusi-solusi optimal.

   Relasi rekurens menggunakan (1).
3. Melakukan perhitungan solusi optimal.

   Perhitungan solusi optimal akan diimplementasikan menggunakan program.
4. Melakukan rekonstruksi dari solusi optimal yang telah didapat.

Berikut merupakan implementasi algoritma Levenshtein Distance menggunakan pemrograman dinamis.

```
d = {}

def editDistance2(str1, str2, m, n):
    key = m, n

    if m == 0:
        return n

    if n == 0:
        return m

    if key in d:
        return d[key]

    if str1[m - 1] == str2[n - 1]:
        return editDistance2(str1, str2, m - 1,
                             n - 1)

    d[key] = 1 + min(editDistance2(str1, str2, m,
                                   n - 1),
                     editDistance2(str1, str2,
                                   m - 1, n),
                     editDistance2(str1, str2,
                                   m - 1, n - 1)
                     )
    return d[key]
```

E. Implementasi Levenshtein Distance Menggunakan Algoritma Wagner-Fischer

Berikut merupakan implementasi algoritma Levenshtein Distance menggunakan algoritma Wagner-Fischer.

```
def editDistance3(str1, str2, l1, l2):
    if (l1 == 0):
        return l2
    if (l2 == 0):
        return l1
    arr = [[0 for _ in range (l2 + 1)] for _ in
           range (l1 + 1)]

    for i in range (l1 + 1):
        arr[i][0] = i

    for i in range (l2 + 1):
        arr[0][i] = i

    for i in range (1, l1+1):
        ch1 = str1[i - 1]

        for j in range (1, l2+1):
            ch2 = str2[j - 1]

            if (ch1 == ch2):
                m = 0
            else:
                m = 1

            arr[i][j] = min(
                min(arr[i - 1][j] + 1, arr[i][j - 1]
                                       + 1),
                arr[i - 1][j - 1] + m
            )

    return arr[l1][l2]
```

F. Implementasi Pemilihan Kata

Pertama, harus dibuat algoritma *similarity* antarstring memanfaatkan Levenshein Distance yang didapat. Algortima bekerja dengan cara membandingkan Levenshein Distance

dengan panjang maksimal dari dua buah string yang dibandingkan yang mengikuti rumus sebagai berikut.

$$\text{sim}(a,b) = 1 - \frac{\text{lev}(a,b)}{\max(|a|,|b|)} \quad (2)$$

Impementasi algoritma *similarity* adalah sebagai berikut.

```
def similarity(len, lev):
    return 1 - lev/len
```

Setelah dilakukan perbandingan, setiap kata pada kamus akan disimpan ke dalam sebuah *priority queue* berdasarkan *similarity*. Berikut merupakan implementasi dari *priority queue*.

```
class PriorityQueue:
    def __init__(self):
        self.pq = []

    def isEmpty(self):
        return len(self.pq) == 0

    def enqueue(self, node):
        if (self.isEmpty()):
            self.pq.append(node)
        else:
            for i in range (0, len(self.pq)):
                if (node > self.pq[i]):
                    self.pq.insert(i, node)
                    return
            self.pq.append(node)

    def dequeue(self):
        return self.pq.pop()


class Node:
    def __init__(self, str, sim):
        self.parent = parent
        self.str = str
        self.sim = sim

    def __gt__(self, next):
        return self.sim > next.sim
```

Berikut merupakan implementasi dari program utama *autocorrect*.

```
kamus = readKamus()

word = input("\nMasukkan kata: ")

print("\n---------- Brute Force ----------")
start_time = time.time()
pq1 = PriorityQueue()
for kata in kamus:
    lev = editDistance1(word, kata, len(word),
                        len(kata))
    node = Node(kata, similarity(max(len(word),
                                     len(kata)),
                                 lev))
    pq1.enqueue(node)
print("Autocorrect: " + pq1.dequeue().str)
print("--- %s\tseconds ---" %
'{0:.16f}'.format(time.time() - start_time))

print("\n------- Pemrograman Dinamis -------")
start_time = time.time()
pq2 = PriorityQueue()
for kata in kamus:
    d = {}
    lev = editDistance2(word, kata, len(word),
                        len(kata))
    node = Node(kata, similarity(max(len(word),
                                     len(kata)),
                                 lev))
    pq2.enqueue(node)
print("Autocorrect: " + pq2.dequeue().str)
print("--- %s\tseconds ---" %
'{0:.16f}'.format(time.time() - start_time))

print("\n---------- Wagner-Fischer ---------")
start_time = time.time()
pq3 = PriorityQueue()
for kata in kamus:
    lev = editDistance3(word, kata, len(word),
                        len(kata))
    node = Node(kata, similarity(max(len(word),
                                     len(kata)),
                                 lev))
    pq3.enqueue(node)
print("Autocorrect: " + pq3.dequeue().str)
print("--- %s\tseconds ---" %
'{0:.16f}'.format(time.time() - start_time))
```

IV. Pengujian

Berikut merupakan beberapa pengujian dari implementasi program.

1. Pengujian I

```
Masukkan kata: muorn

---------- Brute Force ----------
Autocorrect: corn
--- 2.1815071105957031  seconds ---

------- Pemrograman Dinamis -------
Autocorrect: corn
--- 0.1059279441833496  seconds ---

---------- Wagner-Fischer ---------
Autocorrect: corn
--- 0.1084330081939697  seconds ---
```

Gambar 4.1 Pengujian I
Sumber: Dokumen Pribadi

2. Pengujian II

```
Masukkan kata: kalender

---------- Brute Force ----------
Autocorrect: calendar
--- 49.6916379928588867 seconds ---

------- Pemrograman Dinamis -------
Autocorrect: calendar
--- 0.1394970417022705  seconds ---

---------- Wagner-Fischer ---------
Autocorrect: calendar
--- 0.1267099380493164  seconds ---
```

Gambar 4.2 Pengujian II
Sumber: Dokumen Pribadi

3. Pengujian III

```
Masukkan kata: delihgt

---------- Brute Force ----------
Autocorrect: delight
--- 20.0648162364959717 seconds ---

------- Pemrograman Dinamis -------
Autocorrect: delight
--- 0.1449048519134521  seconds ---

---------- Wagner-Fischer ---------
Autocorrect: delight
--- 0.1267800331115723  seconds ---
```

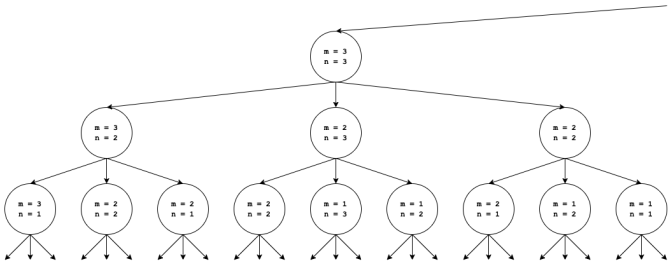
Gambar 4.2 Pengujian III
Sumber: Dokumen Pribadi

V. Analisis dan Pembahasan

Jika dilihat dari pengujian I, pengujian II, dan pengujian III; penggunaan algoritma *Brute Force*, pemrograman dinamis, dan algoritma Wagner-Fischer selalu menghasilkan hasil *autocorrect* yang sama, hal ini membuktikan bahwa penulis berhasil mengimplementasikan algoritma Levenshtein Distance menggunakan tiga algoritma yang berbeda.

Selain itu, dapat dilihat bahwa waktu eksekusi program menggunakan pemrograman dinamis dan algoritma Wagner-Fischer memiliki selisih waktu eksekusi yang sangat kecil. Berarti, efektifitas algoritma pemrograman dinamis dapat dikatakan mendekati efektifitas algoritma Wagner-Fischer. Hal ini bisa disebabkan karena algortima Wagner-Fischer juga menggunakan pendekatan pemrograman dinamis dalam mengoptimasikan algoritma Levenshtein Distance.

Algoritma *brute force* memiliki waktu eksekusi yang jauh lebih lama dari pemrograman dinamis dan algoritma Wagner-Fischer. Dari implementasi algoritma Levenshtein Distance menggunakan algoritma *brute force*, dapat dilihat bahwa setiap pemanggilan fungsi, akan terjadi secara rekursif pemanggilan tiga fungsi. Illustrasi pemanggilan subbagian program untuk pemanggilan algoritma dengan $m = 3$ dan $n = 3$ dapat dilihat pada gambar 5.1.



Gambar 5.1 Ilustrasi Pemanggilan Algoritma *Brute Force*
Sumber: Dokumen Pribadi

Sehingga, algoritma *brute force* memiliki kompleksitas waktu $O(3^k)$ dengan k merupakan panjang maksimal antara m dan n.

VI. Kesimpulan dan Saran

Pemrograman dinamis dan algoritma Wagner-Fischer lebih efektif dibandingkan algoritma *brute force*. Hal ini dapat dilihat dari waktu eksekusi program yang berbededa secara signifikan serta kompleksitas waktu dari masing-masing algoritma.

Untuk penelitian lebih lanjut, dapat dilakukan perbandingan ataupun optimisasi algoritma *edit distance* lainnya, seperti algoritma Damerau-Levenshtein Distance ataupun algoritma Jaro-Winkler Similarity.

VII. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa karena atas berkat dan rahmat-Nya, penulis dapat menyelsaikan makah ini. Penulis juga ingin mengucapkan terima kasih kepada Ibu Dr. Nur Ulfa Maulidevi, S.T, M.Sc. selaku dosen mata kuliah IF2211 Strategi Algoritma Kelas K2 serta Bapak Dr. Ir. Rinaldi Munir, M.T. yang menyediakan bahan ajar dan *tamplate* makalah sehingga memudahkan penulis dalam nenyusun makalah ini. Penulis juga mengucapkan terima kasih kepada kedua orang tua dan keluarga penulis yang senantiasa mendoakan dan mendukung pembuatan makalah ini.

Video Link di Youtube

https://youtu.be/No6vf8dOvfI

Referensi

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf. Diakses pada 20 Mei 2022.

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf. Diakses pada 20 Mei 2022.

[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf. Diakses pada 20 Mei 2022.

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf. Diakses pada 20 Mei 2022.

[5] https://www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm-2. Diakses pada 20 Mei 2022.

[6] Prasetyo, Agung, dkk. Algoritma Jaro-Winkler Distance: Fitur Autocorrect dan Spelling Suggestion pada Penulisan Naskah Bahasa Indonesia di BMS TV. JTIIK. 2018.

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, 20 Mei 2022

Brianaldo Phandiarta - 13520113