Aplikasi Program Dinamis dalam Menoleransi Kata Kunci Pencarian dengan Algoritma Levenshtein Distance untuk Disposisi Tweets ke Dinas-Dinas dan Instansi di Bawah Pemerintah Kota Bandung

Ade Yusuf Rahardian - 13514079
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13514079@std.stei.itb.ac.id

***Abstrak*—Algoriima Levenshtein merupakan algoritma untuk memperoleh nilai *Levenshtein distance. Levenshtein distance* digunakan untuk mengukur nilai kesamaan atau kemiripan antara dua buah kata (*string*). Nilai *Levenshtein distance* diperoleh dengan mencari cara termudah untuk mengubah suatu *string* untuk menjadi *string* lain. Makalah ini menjelaskan bagaimana *levenshtein distance* dapat digunakan untuk menoleransi kata kunci pencarian.**

Kata Kunci—levenshtein; program dinamis; edit distance; twitter

I. PENDAHULUAN

Di era informasi yang sudah maju saat ini, manusia sebagai makhluk sosial mempunyai banyak sarana untuk saling berkomunikasi. Banyak cara yang dilakukan, antara lain melalui tatap muka, berkirim surat, berkirim *e-mail*, dan saling berkirim pesan melalui media sosial seperti *Facebook* dan *Twitter*.

Banyak alasan seorang individu atau kelompok memanfaatkan media komunikasi berupa media sosial. Salah satu contohnya adalah Pemerintah Kota Bandung menggunakan *Twitter* sebagai media komunikasi dengan masyarakat Bandung, terutama untuk menyerap keluhan dari masyarakat yang berkaitan dengan masalah yang menjadi wewenang dan tanggung jawab dinas-dinas dan instansi di dalam Pemkot. Namun, tidak selamanya cara ini efektif karena banyak *tweet* yang 'tercecer' karena tidak tersampaikan ke dinas-dinas yang bersangkutan. Hal ini dapat diatasi dengan menggunakan sebuah kata kunci untuk setiap dinas atau instansi agar *tweet* yang sebenarnya ditujukan untuk dinas atau instansi tertentu dapat didisposisi berdasarkan kata kunci tersebut (misal 'sampah' untuk Dinas Kebersihan).

Pencarian *tweet* dengan kata kunci ini dapat dilakukan dengan algoritma pencarian *string* seperti *Knuth-Morris-Prath* dan *Boyer Moore* (yang merupakan tugas besar ketiga mata kuliah Strategi Algoritma (IF2211) tahun ajaran 2015/2016, Institut Teknologi Bandung), lihat Apendiks A. Namun, algoritma-algoritma pencarian *string* ini merupakan *exact string matching*, jadi *string* yang dicari harus sama persis dengan kata kunci. Padahal kenyataannya, banyak makna yang sama dengan kata kunci, tetapi penulisannya tidak mirip. Hal ini disebabkan karena kesalahan pengetikan (misal 'Bandung' menjadi 'Bandug'), kata-kata yang sering disingkat (misal 'karena' menjadi 'krn'), dan kata-kata yang menjadi tidak baku karena masyarakat telah menganggapnya lazim (misal 'kalau' menjadi 'kalo'). Panjang karakter *tweet* dengan batas maksimal 160 karakter juga menjadi faktor yang membuat pengguna membuat pesan sesingkat mungkin tanpa menghilangkan makna (biasanya dengan menyingkat kata). Masalah seperti ini merupakan permasalahan *approximate string matching*, yaitu permasalahan pencocokan *string* di mana *pattern* yang dicari tidak harus ditemukan sama persis dengan *text* pada string.

Tujuan dari makalah ini adalah untuk memberi solusi pada *tweet* yang seharusnya terdisposisi ke sebuah dinas/instansi tertentu, tetapi tidak terdisposisi karena terdapat sedikit perbedaan antara kata di dalam *tweet* dengan kata kunci pencarian. Salah satu solusi dari permasalahan ini adalah dengan menggunakan program dinamis, yaitu dengan algoritma *Levenshtein distance*. Algoritma *Levenshtein distance* akan menghitung banyaknya pengubahan suatu *string* ke *string* lainnya, dengan harapan semakin sedikit pengubahan, maka semakin dekat pula makna kata tersebut dengan kata kunci yang dicari.

II. TEORI DASAR

A. Program Dinamis

Program dinamis (*dynamic programming*) adalah suatu algoritma pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan tahap (*stage*) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan

yang saling berkaitan. Algoritma *dynamic programming* memiliki beberapa karakteristik penyelesaian, yaitu :

(1) Terdapat sejumlah berhingga pilihan yang mungkin.

(2) Solusi pada setiap tahap dibangun dari hasil solusi tahap sebelumnya.

(3) Kita menggunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan prinsip optimalitas. Prinsip ini berbunyi "jika solusi total optimal, maka bagian solusi sampai tahap ke-k juga optimal".

Prinsip optimalitas berarti bahwa jika kita bekerja dari tahap k ke tahap k + 1, kita dapat menggunakan hasil optimal dari tahap k tanpa harus kembali ke tahap awal. Biaya (*cost*) pada tahap k + 1 = (biaya yang dihasilkan pada tahap k+1) + (biaya dari tahap k ke tahap 1). Secara matematis, biaya pada tahap k+1 dapat dituliskan sebagai berikut.

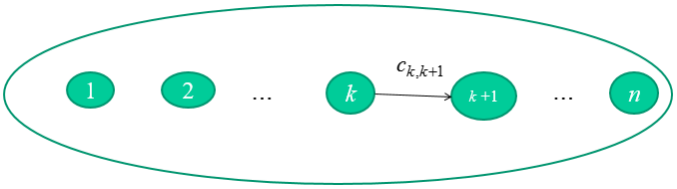
$$F_{k+1}(x) = c_{k+1} + F_k(x)$$

dengan :

$F_{k+1}(x)$ adalah total cost optimum hingga tahap ke k+1

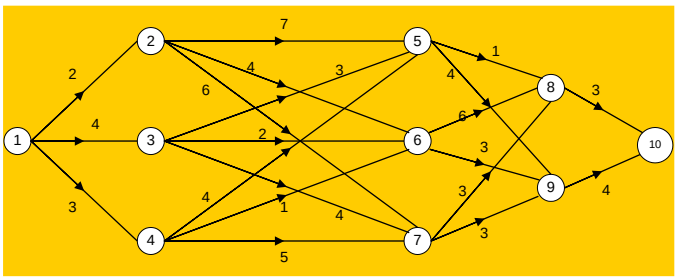
c_{k+1} adalah cost optimum tahap ke k+1

$F_k(x)$ adalah total cost optimum hingga tahap ke k



Gambar 1. Prinsip Optimalitas[2]

Algoritma *dynamic programming* didukung oleh teori graf khususnya graf multi tahap (*multistage graph*). Tiap simpul dari graf tersebut menyatakan status, sedangkan V_1,V_2, .. menyatakan tahap.



Gambar 2. Graf Multi Tahap[2]

Karakteristik persoalan yang dapat menggunakan program dinamis sebagai solusinya adalah :

1. Persoalan dapat dibagi menjadi beberapa tahap (*stage*), yang pada setiap tahap hanya diambil suatu keputusan.
2. Masing-masing tahap terdiri dari sejumlah status yang berhubungan 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 (*cost*) pada suatu tahap meningkat secara teratur (*steadily*) dengan bertambahnya jumlah tahapan.
5. Ongkos pada suatu tahap bergantung pada ongkos tahap-tahap yang sudah berjalan dan ongkos pada tahap tersebut.
6. Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan pada tahap sebelumnya.
7. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap *k* memberikan keputusan terbaik untuk setiap status pada tahap k+1.
8. Prinsip optimalitas berlaku pada persoalan tersebut.

Ada dua pendekatan yang dilakukan dalam program dinamis yaitu program dinamis maju dan program dinamis mundur. Misalkan x_1, x_2, …, x_n menyatakan peubah (variable) keputusan yang harus dibuat masing-masing untuk tahap 1, 2, …, n. Maka,

1. Program dinamis maju. Program dinamis bergerak mulai dari tahap 1, terus maju ke tahap 2, 3, dan seterusnya sampai tahap n. Runtunan peubah keputusan adalah x_1, x_2, …, x_n.
2. Program dinamis mundur. Program dinamis bergerak mulai dari tahap n, terus mundur ke tahap n – 1, n – 2, dan seterusnya sampai tahap 1. Runtunan peubah keputusan adalah x_n, x_{n-1}, …, x_1.

Prinsip optimalitas program dinamis maju, ongkos pada tahap k +1 merupakan hasil penjumlahan dari ongkos yang dihasilkan pada tahap k dengan ongkos dari tahap k ke tahap k + 1, untuk k = 1, 2, …, n – 1. Sedangkan prinsip optimalitas pada program dinamis mundur adalah ongkos pada tahap k merupakan hasil penjumlahan ongkos yang dihasilkan pada tahap (k + 1) dengan ongkos dari tahap (k + 1) ke tahap k , dengan k = n, n – 1, …, 1.

Langkah-langkah pengembangan algoritma program dinamis adalah sebagai berikut:

1. Karakteristikkan struktur solusi optimal.
2. Definisikan secara rekursif nilai solusi optimal.
3. Hitung nilai solusi optimal secara maju atau mundur.
4. Konstruksi solusi optimal

B. Approximate String Matching

Permasalahan *approximate string matching* adalah permasalahan pencocokan *string* di mana pola dan *string* yang dibandingkan tidak perlu sama persis. Terdapat sebuah nilai yang disebut *edit distance/ Levenshtein distance* yang

menggambarkan seberapa mirip *string* dengan pola yang dibandingkan. Misal terdapat dua buah *string*, yaitu *string* A dan *string* B. *String* B dapat ditransformasikan menjadi *string* A dengan tiga jenis operasi: *deletion* (penghapusan), *insertion* (penyisipan), dan *substitution* (penggantian). Penamaan *string* A sebagai tujuan dari transformasi dari *string* B akan digunakan pada tulisan ini

1. Operasi penghapusan
   Misal A = abcd dan B = aabbccdd. Maka *string* B dapat ditransformasikan menjadi *string* A dengan 4 buah operasi penghapusan, yaitu di posisi 2, 4, 6, dan 8.

	1	2	3	4	5	6	7	8
A =	a	-	b	-	c	-	d	-
B =	a	A	b	b	c	c	d	d

2. Operasi penyisipan
   Misal A = aabbccdd dan B = abcd. Maka *string* B dapat ditransformasikan menjadi *string* A dengan 4 buah operasi penyisipan yang dideskripsikan pada gambar berikut.

	1	2	3	4	5	6	7	8
A =	a	a	b	b	c	c	d	d
B =	a	-	b	-	c	-	d	-
		a		b		c		d

3. Operasi penggantian
   Misal A = abcd dan B = abce. Maka *string* B dapat ditransformasikan menjadi *string* A dengan sebuah operasi penggantian dengan mengganti karakter e menjadi karakter d.

	1	2	3	4
A =	a	b	c	d
B =	a	b	c	e

Levenshtein distance didefinisikan sebagai jumlah operasi minimum yang diperlukan untuk mentransformasi *string* B menjadi *string* A. *Levenshtein distance* tersebut dinotasikan dengan lev(A, B). Semakin kecil nilai *Levenshtein distance* antara A dan B maka semakin mirip *string* A dan B. *String* A dan *string* B dikatakan identik apabila lev(A, B) = 0. Penghitungan *Levenshtein distance* biasa dilakukan dengan menggunakan program dinamis.

Ada satu nilai lain yang dijadikan indikator kesamaan dua buah *string* yaitu *Hamming Distance*. *Hamming distance* sama dengan *Levenshtein distance*, tetapi operasi yang diperhitungkan hanyalah operasi penggantian.

Contoh kasus perhitungan *Levenshtein distance* dan *hamming distance* adalah sebagai berikut. Misal A = abcd dan B = abcef, maka terdapat dua operasi minimum yang harus dilakukan untuk mentransformasikan *string* B menjadi *string* A yaitu penghapusan ‘f’ serta substitusi ‘e’ dengan ‘d’. Nilai *Levenshtein distance*nya adalah dua dan nilai *Hamming distance*nya adalah satu.

III. PEMBAHASAN

A. Rancangan Program

Program disposisi *tweets* ini merupakan program berbasis web dan diimplementasi dengan bahasa pemrograman C# (lihat Apendiks B). Dalam pembuatannya, digunakan API TweetSharp untuk memperoleh 100 tweet terbaru berdasarkan kata kunci yang diminta oleh pengguna.

Selanjutnya, dilakukan iterasi pada setiap *string tweet* untuk mengetahui hubungan *tweet* tersebut dengan dinas tertentu. *Tweet* yang akan diperiksa tersebut dipotong-potong berdasarkan spasi agar dapat diperiksa setiap katanya.

Kata-kata yang terdapat pada *tweet* dibandingkan dengan setiap kata kunci dinas terkait dengan menghitung *Levenshtein Distance*nya. Sebelumnya, ditentukan terlebih dahulu batas toleransi Levenshtein Distance untuk kata kunci yang dicari. Misal batas toleransi diatur nilainya menjadi ‘2’, maka setiap penghitungan antara kata kunci dinas terkait dengan kata yang terdapat dalam *tweet* yang bernilai kurang dari sama dengan dua akan didisposisi ke dinas tersebut.

B. Penghitungan Levenshtein Distance

Terdapat dua pendekatan dalam menghitung *Levenshtein distance*. Pertama, *Levenshtein distance* dihitung dengan cara rekursif murni. Cara ini tentu memakan waktu karena submasalah yang sama dihitung berulang-ulang. Kedua, *Levenshtein distance* dihitung dengan program dinamis *bottom-up* yang iteratif.

Pada masalah ini, solusi dibuat dengan menggunakan pendekatan kedua. Apabila terdapat dua buah *string* yang akan dibandingkan, misal A dan B, maka *program* dinamis akan menggunakan sebuah matriks dua dimensi dengan ukuran (a+1)x(b+1), di mana a dan b merupakan panjang *string* A dan *string* B berturut-turut. Matriks ini digunakan untuk menyimpan *Levenshtein distance* antara semua prefiks *string A* dan semua prefiks *string* B. Fungsi perhitungan *levenshtein distance* adalah sebagai berikut.

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

Ketika salah satu panjang string adalah 0, maka *Levenshtein distance* kedua string tersebut adalah panjang string yang lainnya. Apabila tidak, *Levenshtein distance* adalah nilai minimum dari tiga nilai yaitu $lev(i-1,j)+1$ yang merupakan operasi penghapusan, $lev(i,j-1)+1$ yang merupakan operasi penyisipan, dan $lev(i-1,j-1)+1$ yang merupakan operasi penggantian (Apabila a_i *dan* b_j sama, maka nilai dari levenshtein distance adalah $lev(i-1,j-1)$). Hasil dari perhitungan setiap fungsi lev akan disimpan ke dalam matriks agar dapat digunakan pada perhitungan selanjutnya.

Berikut merupakan kode untuk fungsi lev dalam bahasa C#.

```
static class LevenshteinDistance
{
    /// Compute the distance between two strings.
    public static int Compute(string s, string t)
    {
        int n = s.Length;
        int m = t.Length;
        int[,] d = new int[n + 1, m + 1];

        // Step 1
        if (n == 0)
        {
            return m;
        }

        if (m == 0)
        {
            return n;
        }

        // Step 2
        for (int i = 0; i <= n; d[i, 0] = i++)
        {
        }

        for (int j = 0; j <= m; d[0, j] = j++)
        {
        }

        // Step 3
        for (int i = 1; i <= n; i++)
        {
            //Step 4
            for (int j = 1; j <= m; j++)
            {
                // Step 5
                int cost;
                if (t[j - 1] == s[i - 1])
                {
                    cost = 0;
                }
                else
                {
                    cost = 1;
                }

                // Step 6
                d[i, j] = Math.Min(
                    Math.Min(d[i - 1, j] + 1,
                             d[i, j - 1] + 1),
                    d[i - 1, j - 1] + cost);
            }
        }
        // Step 7
        return d[n, m];
    }
}
```

Salah satu penggunaannya dalam aplikasi ini misalnya kata kunci dinas terkait yang ingin dicari adalah “angklung” dan kata yang sedang dicek dalam sebuah *tweet* adalah “bandung”. Maka string A = angklung dengan panjang string 8 dan string B = bandung dengan panjang string 7. Lalu, fungsi di atas akan membentuk matriks kosong berukuran 9 x 8. Awalnya, semua elemen matriks diinisialisasi dengan 0. Seperti tampak pada gambar di bawah ini.

		b	a	n	d	u	n	g
	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
k	0	0	0	0	0	0	0	0
l	0	0	0	0	0	0	0	0
u	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0

Gambar 3. Matriks Awal

Langkah berikutnya adalah mengisi baris dan kolom ke-0 dengan nomor baris dan nomor kolom. Hal ini merepresentasikan bahwa untuk mengubah sebuah *string* kosong menjadi sebuah *string* lain, dibutuhkan pengubahan (*Levenshtein distance)* sepanjang *string* lain tersebut. Berikut hasil dari pengisian baris dan kolom ke-0.

		b	a	n	d	u	n	g
	0	1	2	3	4	5	6	7
a	1	0	0	0	0	0	0	0
n	2	0	0	0	0	0	0	0
g	3	0	0	0	0	0	0	0
k	4	0	0	0	0	0	0	0
l	5	0	0	0	0	0	0	0
u	6	0	0	0	0	0	0	0
n	7	0	0	0	0	0	0	0
g	8	0	0	0	0	0	0	0

Gambar 4. Matriks setelah pengisian baris dan kolom ke-0

Kemudian dilakukan pengisian dari baris pertama dan kolom pertama sampai ke baris terakhir dan kolom terakhir. Hasil akhir matriks akan terlihat seperti gambar di bawah ini.

		b	a	n	d	u	n	g
	0	1	2	3	4	5	6	7
a	1	1	1	2	3	4	5	6
n	2	2	2	1	2	3	4	5
g	3	3	3	2	2	3	4	4
k	4	4	4	3	3	3	4	5
l	5	5	5	4	4	4	4	5
u	6	6	6	5	5	4	5	5
n	7	7	7	6	6	5	4	5
g	8	8	8	7	7	6	5	4

Gambar 5. Matriks setelah selesai diisi

Dari Gambar 5., nilai *Levenshtein distance* dapat dilihat pada sel matriks paling kanan bawah, yaitu 4. Nilai ini menunjukkan bahwa untuk mengubah *string* 'angklung' menjadi 'bandung' perlu dilakukan empat pengubahan, yaitu penyisipan 'b' di depan *string*, penyisipan 'g' setelah 'n' pertama, penyisipan 'k' tepat setelah penyisipan'g' sebelumnya, dan penggantian 'l' menjadi 'd'.

IV. PENGUJIAN

Pengujian dilakukan untuk beberapa kasus saat melakukan pencarian. Dalam pengujian, dipilih batas atas *Levenshtein distance* sebesar dua. Jadi apabila diperoleh nilai *Levenshtein distance* kurang dari sama dengan dua, maka *tweet* tersebut dianggap sesuai dengan kata kunci pencarian yang dimaksud. Selain itu, pengujian bersifat *case-insensitive*. *Tweets* yang diperoleh dalam kasus pengujian ini merupakan *tweets* yang melakukan *mention* ke *@pemkotbandung* dan diperoleh tanggal 5 Mei 2016 jam 19.00 WIB.

A. Pencarian dengan Kata Sesuai Kamus Besar Bahasa Indonesia

Gambar 6. Pencarian dengan Kata Kunci 'bersih', 'pembersihan', dan 'Bojonegoro' yang didisposisi ke PDAM

Dari Gambar 6., nilai *Levenshtein distance* masing-masing kata dalam *tweet* yang sesuai dengan kata kunci adalah sebagai berikut.

No	Kata Kunci Pencarian	Kata dalam *tweet*	*Levenshtein Distance*
1	bersih	bebersih	2
2	pembersihan	Pmbrsihan	2
3	Bojonegoro	Bojonegoro	0

Tabel I. Tabel *Levenshtein Distance* untuk Pencarian dengan Kata sesuai Kamus Besar Bahasa Indonesia

B. Pencarian dengan Kata yang Lazim untuk Disingkat

Gambar 7. Pencarian dengan Kata Kunci 'tmnsari' yang didisposisi ke DBMP

Kata yang digunakan untuk menjadi kata kunci adalah 'tmnsari', kata ini lazim digunakan untuk menyingkat 'Tamansari' yang merupakan sebuah nama jalan di Kota Bandung. Dari Gambar 7., nilai *Levenshtein distance* masing-masing kata dalam *tweet* yang sesuai dengan kata kunci adalah sebagai berikut.

No	Kata Kunci Pencarian	Kata dalam *tweet*	*Levenshtein Distance*
1	tmnsari	Tamansari	2

Tabel II. Tabel *Levenshtein Distance* untuk Pencarian dengan Kata yang Lazim untuk Disingkat

C. Pencarian dengan Kata yang Tidak Sesuai dengan Aturan Bahasa Indonesia

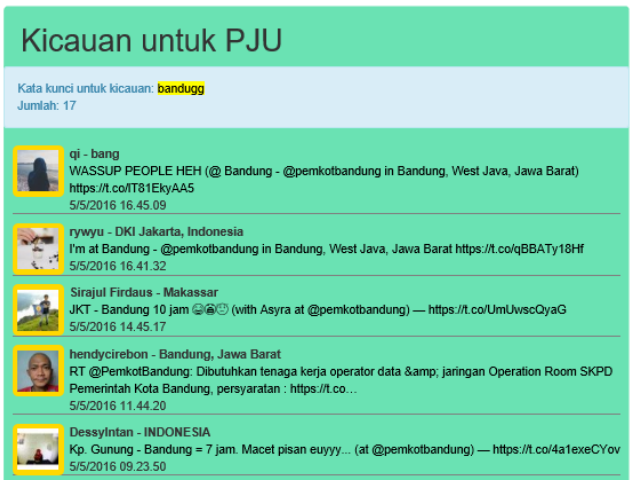
Gambar 8. Pencarian dengan Kata Kunci ‘KerjaBakti’ dan ‘gorong’ yang didisposisi ke PD Kebersihan

Kata yang digunakan untuk menjadi kata kunci adalah ‘KerjaBakti’ dan ‘gorong’. Dalam penulisan bahasa Indonesia yang benar seharusnya ditulis ‘kerja bakti’ dan ‘gorong-gorong’. Dari Gambar 8., nilai *Levenshtein distance* masing-masing kata dalam *tweet* yang sesuai dengan kata kunci adalah sebagai berikut.

No	Kata Kunci Pencarian	Kata dalam *tweet*	*Levenshtein Distance*
1	KerjaBakti	KrjaBkti	2
2	Gorong	grong2	2

Tabel III. Tabel *Levenshtein Distance* untuk Pencarian dengan Kata yang Tidak Sesuai dengan Aturan Bahasa Indonesia

D. Pencarian dengan Kata yang Salah dalam Pengetikan



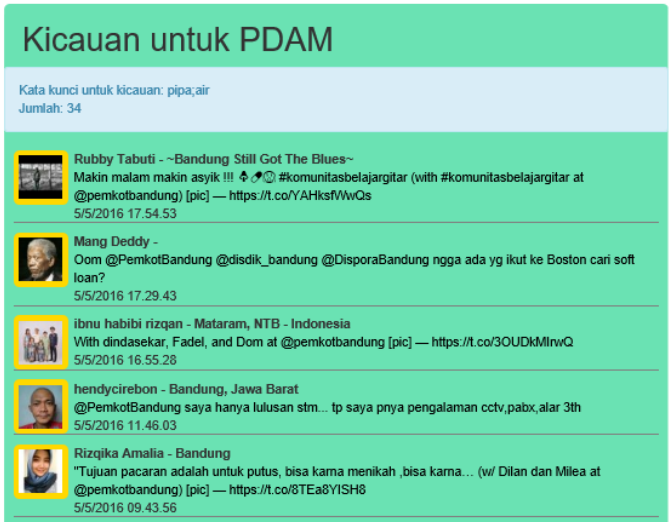
Gambar 9. Pencarian dengan Kata Kunci ‘bandugg’ yang didisposisi ke PJU

Kesalahan dalam mengetik merupakan hal yang sering terjadi bagi pengguna gawai. Kata yang diuji pada tahap ini adalah ‘bandugg’ yang seharusnya ditulis ‘bandung’. Dari Gambar 9., nilai *Levenshtein distance* antara kata dalam *tweet* yang sesuai dengan kata kunci adalah sebagai berikut.

No	Kata Kunci Pencarian	Kata dalam *tweet*	*Levenshtein Distance*
1	bandugg	Bandung	1

Tabel IV. Tabel *Levenshtein Distance* untuk Pencarian dengan Kata yang Salah dalam Pengetikan

E. Pencarian Kata Pendek (Jumlah Huruf Kurang dari 4)



Gambar 10. Pencian dengan Kata Kunci ‘pipa’ dan ‘air’ yang didisposisi ke PDAM

Kata yang digunakan untuk menjadi kata kunci adalah ‘air’ dan ‘pipa’. Dari Gambar 10., nilai *Levenshtein distance* masing-masing kata dalam *tweet* yang sesuai dengan kata kunci adalah sebagai berikut.

No	Kata Kunci Pencarian	Kata dalam *tweet*	*Levenshtein Distance*
1	air	at	2
2	air	ada	2
3	air	and	2
4	pipa	pnya	2
5	pipa	bisa	2

Tabel V. Tabel *Levenshtein Distance* untuk Pencarian Kata Pendek (Jumlah Huruf Kurang dari 4)

V. ANALISIS

Dari hasil pengujian, kita dapat melihat bahwa nilai *Levenshtein distance* kurang dari sama dengan dua antara dua buah *string* dapat menyaring kata dalam *tweet* yang sebagian besar memiliki makna sama dengan kata kunci pencarian. Namun, pada pencarian kata dengan jumlah huruf sedikit, diperoleh *tweet* dengan kata yang memiliki makna jauh berbeda dari yang dimaksud.

VI. KESIMPULAN DAN SARAN

Algoritma *Levenshtein distance* cukup mumpuni dalam melakukan toleransi kata kunci pencarian untuk disposisi *tweets*. Namun, ada masalah ketika *string* yang dicari pendek. Masalah ini dapat diatasi dengan penyesuaian nilai toleransi berdasarkan panjang *string*. Dilihat dari sisi memori, algoritma ini tergolong boros memori karena memerlukan sebuah tabel baru ketika akan menghitung *Levenshtein distance* antara dua *string*.

Algoritma *Levenshtein distance* belum menangani kasus yang hanya memerlukan transposisi karakter yang mana merupakan kesalahan manusia yang cukup sering dalam pengetikan, seperti 'Bandung' menjadi 'Bnadung' Pada algoritma *Levenshtein distance*, transposisi karakter dihitung sebagai dua pengubahan. Masalah ini seharusnya dapat menjadi lebih baik lagi dengan menggunakan algoritma *Damerau-Levenshtein distance* yang merupakan versi lebih baik dari dari algoritma *Levenshtein distance*. Algoritma *Damerau-Levenshtein distance* menghitung transposisi karakter sebagai satu pengubahan.

VII. APENDIKS

Apendiks A – Aplikasi Disposisi *Tweet* dengan Menggunakan Algoritma KMP dan Boyer-Moore
Kode aplikasi dapat dilihat di
http://github.com/mickyyu96/TubesTwitterAPI/

Apendiks B – Aplikasi Disposisi *Tweet* dengan Menggunakan Algoritma Levenshtein
Kode aplikasi dapat dilihat di
https://github.com/adeyura/Disposisi-Tweets-With-Levenshtein

VIII. UCAPAN TERIMA KASIH

Penulis memanjatkan puji syukur kepada Tuhan Yang Maha Esa atas segala kenikmatan sehingga dapat menyelesaikan makalah ini. Penulis juga mengucapkan terima kasih kepada Bapak Dr.Ir. Rinaldi Munir, MT. dan Ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen pengajar mata kuliah Strategi Algoritma atas segala bimbingan serta ilmu yang telah diberikan kepada penulis. Penulis juga menyampaikan rasa terima kasih kepada Micky Yudi Utama dan Atika Azzahra Akbar yang telah menjadi rekan dalam pembuatan aplikasi disposisi *tweets* untuk versi *exact matching string* (dengan algoritma KMP dan BM), yang kemudian penulis modifikasi secara pribadi sebagai versi *approximate matching string* untuk pembuatan makalah ini. Rasa terima kasih juga penulis sampaikan kepada teman-teman penulis yang mendukung dan mendorong penulis dalam menyelesaikan makalah ini. Terakhir, penulis juga menyampaikan terima kasih kepada seluruh pihak yang ikut membantu pembuatan makalah ini baik yang secara langsung maupun tidak langsung.

REFERENSI

[1] Munir, Rinaldi. 2009. Diktat Kuliah Strategi Algoritma. Bandung : Program Studi Teknik Informatika Institut Teknologi Bandung.

[2] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2014-2015/Program%20Dinamis%20(2015).ppt, diakses pada 6 Mei 2016 pukul 18.00 WIB

[3] http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK2/NODE46.HTM, diakses pada 6 Mei 2016 pukul 18.20 WIB

[4] http://repositorio.uchile.cl/bitstream/handle/2250/126168/Navarro_Gonzalo_Guided_tour.pdf, diakses pada 6 Mei 2016 pukul 19.30 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, 7 Mei 2016

Ade Yusuf Rahardian (13514079)