Auto-correct Menggunakan Program Dinamis

Garmastewira 13514068
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132,I Indonesia
13514068@std.stei.itb.ac.id

***Abstrak*—Pada aplikasi kategori *word-processing*, biasanya terdapat fitur yang bernama *auto-correct*, yang digunakan untuk membenarkan kata yang pengejaannya salah secara otomatis. Implementasi *auto-correct* yang dasar salah satunya menggunakan algoritma Levenshtein untuk menghitung *edit distance* antara dua kata. Algoritma Levenshtein sendiri dapat diimplementasikan secara *brute force*, atau menggunakan program dinamis *bottom-up*.**

Kata kunci—auto-correct, edit distance, program dinamis, brute force.

I. PENDAHULUAN

Aplikasi dengan kategori *word-processing* merupakan aplikasi yang sangat sering digunakan dan tentunya paling krusial. Pengguna menggunakan aplikasi *word-processing* untuk membuat berbagai macam dokumen, mulai dari laporan, soal ujian, serta dokumen-dokumen lain yang tak terhitung jenisnya. Namun, tentunya pengguna tidak mungkin mengetikkan semua kata dengan benar; selalu saja ada kata yang pengejaannya salah. Tidak hanya dalam aplikasi *word-processing*, aplikasi sosial media seperti Facebook, Twitter, dan LINE juga membutuhkan pengguna untuk menulis, dan tentunya rawan kesalahan.

Maka dari itu, sebuah fitur yang dinamai *auto-correct* dibuat untuk mengatasi persoalan. *Auto-correct* merupakan fitur yang selalu ada di *word-processor* dan platform *messaging* berbagai tipe. Produk-produk karya Apple, Google, dan Microsoft memiliki versi *auto-correct* yang berbeda-beda, tergantung bagaimana algoritma dan implementasi dari masing-masing. Makalah ini akan membahas apa itu pengertian *auto-correct, edit distance*, dan implementasi *edit distance* dan *auto-correct* baik dengan algoritma *brute force* ataupun program dinamis.

II. DASAR TEORI

A. *Auto-correct*

Auto-correct merupakan sebuah tipe program perangkat lunak yang mengidentifikasi kata-kata yang salah pengejaan, menggunakan algoritma untuk mengidentifikasi kata-kata yang mirip, dan kemudian menyunting text tersebut secara otomatis[1].

B. *Edit Distance*

Di ilmu komputer, *edit distance* merupakan sebuah cara untuk menghitung seberapa beda sebuah string dengan satu string lainnya dengan cara menghitung jumlah minimum operasi yang dibutuhkan untuk mengganti string pertama menjadi string lain tersebut. *Edit distance* digunakan pada berbagai bidang. Contohnya, di bio-informatika, *edit distance* digunakan untuk menghitung kesamaan antara dua buah sekuens DNA yang dapat direpresentasikan sebagai sebuah string yang terdiri dari huruf A, C, G, dan T[2].

Secara formal, diberikan dua buah string a dan b yang meruapakan anggota alfabet Σ (misalnya himpunan karakter ASCII, himpunan biner, dan lain-lain), maka *edit distance* d(a, b) merupakan nilai minimum dari sekian operasi untuk mentransformasikan a menjadi b dengan operas-operasi yang didefinisikan oleh Levenshtein sebagai berikut [2]:

- Penyisipan (*Insertion*) sebuah simbol. Apabila a = uv, maka menyisipkan simbol x pada a dapat memproduksi string xuv, uxv, dan uvx.
- Substitusi (*Substitution)* sebuah simbol. Apabila a = uxv, dan x → y, maka a = uyv.
- Penghapusan (*Deletion*) sebuah simbol. Apabila a = uv, maka penghapusan dapat memproduksi string u dan v.

Sebagai contoh, misalnya terdapat string a = "kitten", dan string b = "sitting". Maka, dibutuhkan *edit distance* sebanyak 3 untuk mengubah string a menjadi string b dengan rincian sebagai berikut:

- **k**itten → **s**itten (substitusi 's' menjadi 'k')
- sitt**e**n → sitt**i**n (substitusi 'e' menjadi 'i')
- sittin → sittin**g** (penyisipan 'g' pada akhir string)

Dengan menggunakan ketiga operasi yang didefinisikan oleh Levenshtein, maka *edit distance* antara string a = $a_1a_2...a_m$ dan b = $b_1b_2...b_n$ adalah d_{mn} yang didefiniskan dengan rekurens sebagai berikut[3]:

$$d_{i0} = i$$
$$d_{0j} = j$$

Basis

$$d_{ij} = \begin{cases} d_{i-1j-1}, a[i] = b[j] \\ \begin{cases} d_{i-1j}\ (Insert) \\ d_{ij-1}\ (Delete) \end{cases} \quad a[i] \neq b[j] \\ d_{i-1j-1}\ (Replace) \end{cases}$$

Di mana $i \leq m$ dan $j \leq n$, serta a_i adalah $a_1a_2...a_i$ dan b_j adalah $b_1b_2...b_j$.

C. Program Dinamis

Program dinamis (*dynamic programming*) merupakan pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan tahapan (*stage*) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan[4]. Isitilah program dinamis muncul karena perhitungan solusi menggunakan tabel yang ukurannya dapat berubah.

Berikut adalah karateristik penyelesaian persoalan dengan program dinamis[4]:

- Persoalan dapat dibagi menjadi beberapa tahap (*stage*), yang pada setiap tahap hanya diambil satu keputusan.
- Masing-masing tahap terdiri dari sejumlah status (*state*) yang berhubungan dengan tahap tersebut. Secara umum, status merupakan bermacam kemungkinan masukan yang ada pada tahap tersebut.
- Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
- Ongkos (*cost*) pada suatu tahap meningkat secara teratur (*steadily*) dengan bertambahnya jumlah tahapan.
- Ongkos pada suatu tahap bergantung pada ongkos tahap-tahap yang sudah berjalan dan ongkos pada tahap tersebut.
- Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan pada tahap sebelumnya.
- Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap k memberikan keputusan terbaik untuk setiap status pada tahap k+1.
- Prinsip optimalitas berlaku pada persoalan tersebut.

Program dinamis memiliki dua pendekatan, yaitu pendekatan maju dan pendekatan mundur. Perbedaannya, pada pendekatan maju, program dinamis bergerak dari tahap 1, ke tahap 2, dan seterusnya hingga tahap ke-n, yang mana tahap ke-k+1 akan membutuhkan keputusan terbaik tahap ke-k. Sebaliknya, pada pendekatan mundur, program dinamis bergerak dari tahap n, ke tahap n-1, dan seterusnya hingga tahap ke-1, yang mana tahap ke-k membutuhkan keputusan terbaik tahap ke-k+1[4].

Penyelesaian program dinamis dapat juga dilakukan dengan pendekatan *top-down* dan *bottom-up*[5].

- Pendekatan *top-down* dikenal juga dengan tahap memoisasi. Dengan cara ini, maka pengerjaan dimulai dari tahap ke-n hingga tahap basis, dan setiap hasil suatu tahap disimpan hasilnya. Jadi, apabila ada tahap yang *overlapping*, maka dijamin hanya akan dihitung sekali saja
- Pendekatan *bottom-up* dikenal juga dengan tahap tabulasi karena biasanya dalam komputasinya menggunakan tabel. Dengan cara ini, maka pengerjaan dimulai dari basis hingga tahap ke-n. Berbeda dengan *top-down*, urutan komputasi dapat ditentukan.

III. Analisis Algoritma *Edit Distance*

A. Menggunakan Brute Force

Dengan menggunakan algoritma *Brute-Force*, maka rekurens yang sudah didefinisikan di bab II dapat dibuat kode semunya sebagai berikut:

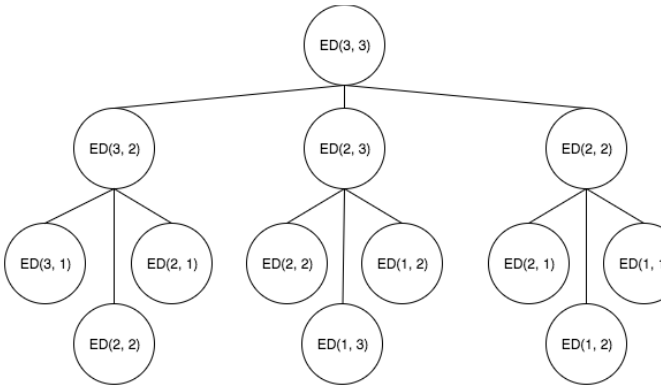
```
function editDist(str1: String, str2: String) → integer

KAMUS
        m: integer
        n: integer

ALGORITMA
        m ← length(str1)
        n ← length(str2)

        if (m = 0) then { Insert n times }
                → n
        else if (n = 0) then { Delete m times }
                → m
        else if (str1[m] = str2[n]) then
                → editDist(str1[1..m-1],
                   str2[1..n-1])
        else
                → 1 + min (
                { Insert }
                editDist(str1, str2[1..n-1]),
                { Remove }
                editDist(str1[1..m-1], str2[n]),
                { Subtitute }
                editDist(str1[1..m-1], str2[1..n-1])
                )
```

Pohon pencarian solusi dengan kode di atas (hanya hingga tingkat 2 pohon) adalah sebagai berikut:



Gambar 1. Hasil pencarian solusi edit distance dengan algoritma *brute force*

Terlihat bahwa sangat banyak sub-masalah yang muncul berkali-kali. Misalnya, pada Gambar 1, terlihat simpul ED(1, 2) muncul berkali-kali. Kenyataannya pada algoritma tersebut, ED(1, 2) harus dihitung dari awal untuk tiap simpul, padahal akan menghasilkan sub-masalah yang sama juga.

Seperti solusi dengan pembangunan pohon biasanya, kompleksitas waktu algoritma di atas memakan $O(b^m)$, di mana b adalah faktor percabangan dan m adalah kedalaman dari pohon. Untuk algoritma pencarian *edit distance* diatas, terlihat bahwa faktor percabangan nilainya 3 karena hanya didefinisikan 3 operasi untuk mentransformasikan suatu string ke string lain. Lalu, jumlah kedalaman adalah panjang dari string pertama. Jika misalkan panjang string pertama adalah m, maka kompleksitas algoritma ini adalah $O(3^m)$.

B. *Optimisasi Menggunakan Program Dinamis (Algoritma Wagner-Fischer)*

Dari algoritma yang dihasilkan oleh algoritma *edit distance* brute force sebelumnya, terlihat bahwa banyak sub-masalah yang saling bertumpukan (*overlapping subproblems*), yang mana merupakan karateristik dari persoalan yang dapat diselesaikan program dinamis.

Sebenarnya, dengan *approach top-down*, maka sangat mudah untuk optimisasi permasalahn berikut menjadi program dinamis. Program cukup menyimpan sebuah variabel global sebuah matriks, misalkan bernama **editDistanceResults**, yang menyimpan nilai hasil d_{ij}, dengan i indeks pada string 1 dan j indeks pada string 2, sehingga untuk i dan j yang sama, d_{ij} hanya akan dihitung sekali. *Pseudo-code* dengan algoritma *top-down* adalah sebagai berikut:

```
{ Variabel global }
{ Elemen matriks diinisialisasikan dengan -1 }
editDistResults:
matriks[length(str1)+1][length(str2)+1] of integer

function editDistDPTopUp(str1: String, str2:
String) → integer

KAMUS
        m: integer
        n: integer

ALGORITMA
        m ← length(str1)
        n ← length(str2)

        if (m = 0) then { Insert n times }
            → n
        else if (n = 0) then { Delete m times }
            → m
        else
            if (editDistResults[m][n] != -1) then
               → editDistResults[m][n]
            else
               if (str1[m] = str2[n]) then
                   → editDist(str1[1..m-1],
                       str2[1..n-1])
               else
                   → 1 + min (
                   { Insert }
                   editDist(str1, str2[1..n-1]),
                   { Remove }
                   editDist(str1[1..m-1], str2[n]),
                   { Subtitute }
                   editDist(str1[1..m-1],
                            str2[1..n-1])
                  )
```

Namun sayangnya, akan sulit untuk mengetahui kompleksitas waktu dari algoritma di atas, sehingga akan digunakan cara *bottom-up*, atau dikenal juga sebagai cara dengan tabulasi. Algoritma ini dikenal juga sebagai Algoritma Wagner-Fischer[5].

Program dinamis dengan pendekatan maju akan digunakan untuk menyelesaikan persoalan *edit* distance. Untuk struktur solusi optimal, maka akan ada m x n tahap, mulai dari x_1 hingga $x_{m \times n}$, di mana m adalah panjang string a dan n adalah panjang string b.

Untuk setiap tahap x_k, maka tahapan tersebut akan merepresentasikan $d_{(k-1 \text{ div } m) + 1, (k-1 \text{ mod } n) + 1}$. Sebagai contoh, jika string a = "bakso" dan string b = "sapu", maka tahap ke-7 dari program dinamis adalah $d_{(7-1 \text{ div } 4) + 1, (7-1 \text{ mod } 4) + 1} = d_{23}$.

Sebagai ilustrasi untuk penyelesaian, maka akan dibuat tabel seperti pada Gambar 2. Setiap baris merupakan huruf-huruf dari string a, sedangkan setiap kolom merupakan huruf-huruf dari string b. Terlihat bahwa terdapat sebuah baris dan kolom dengan huruf ‘ ‘ untuk menyatakan string kosong. Pada tabel ini, d_{ij} memperlihatkan nilai edit distance dari string a[1..i] dan string[b..j]. Sebagai contoh, $d_{3,3}$ artinya nilai edit distance dari “bak” menjadi “sap”.

Pertama-tama, inisialisasikan nilai basis pada tabel yakni $d_{i0} = i$, dan $d_{0j} = j$ seperti pada Gambar 2. Pada Gambar 2, terlihat ada tiga panah dengan warna yang berbeda. Dari algoritma Levenshtein, maka jelas bahwa panah hijau merepresentasikan

bila insertion, panah jingga merepresentasikan deletion, dan panah biru merepresentasikan substitution.

		S	A	P	U
	0	1	2	3	4
B	1				
A	2				
K	3				
S	4				
O	5				

Gambar 2. Tabulasi pencarian solusi *minimum edit distance*

Contoh perhitungan tabel akan dicontohkan oleh Gambar 3, Gambar 4, dan Gambar 5. Pada Gambar 3, akan dihitung d_{11} (sel merah). Karena S ≠ B, jumlah edit distance dari S ke B adalah 1 + nilai minimal dari d_{10}, d_{00}, dan d_{01}. Masing-masing d_{10}, d_{00}, dan d_{01} direpresentasikan oleh sel kuning pada tabel. Terlihat nilai minimalny adalah d_{00} (dengan substitusi), maka nilai *edit distance* adalah 0 + 1 = 1. Pencarian untuk nilai d_{12} diilustrasikan oleh Gambar 4.

Kasus lainnya, yakni apabila huruf terakhir string sama, diilustrasikan oleh Gambar 5. Pada Gambar 5, yakni pencarian nilai d_{22}, karakter akhir substring a sama dengan substring b (huruf A). Maka, nilai d_{22} adalah d_{11}.

		S	A	P	U
	0	1	2	3	4
B	1	1			
A	2				
K	3				
S	4				
O	5				

Gambar 3. Tabel perhitungan d_{11}

		S	A	P		U
	0	1	2	3		4
B	1	1	2			
A	2					
K	3					
S	4					
O	5					

Gambar 4. Tabel perhitungan d_{12}

		S	A	P	U
	0	1	2	3	4
B	1	1	2	3	4
A	2	2	1		
K	3				
S	4				
O	5				

Gambar 5. Tabel perhitungan d_{22}

Dari Gambar 6, maka kita akan menemukan bahwa *edit distance* minimal untuk string "bakso" menjadi string "sapu" adalah 4. Untuk mengetahui detail operasi-operasi apa saja yang diperlukan, maka bangun path minimal dengan panjang d_{mn} + 1 dari sel ke-[m, n] hingga sel ke-[0, 0] dengan syarat setiap perpindahan sel, nilainya harus lebih kecil atau sama dengan.

		S	A	P	U
	0	1	2	3	4
B	1	1	2	3	4
A	2	2	1	2	3
K	3	3	2	2	3
S	4	3	3	3	3
O	5	4	4	4	4

Gambar 6. Hasil pencarian solusi edit distance dengan *dynamic programming*

Seperti yang telah dijelaskan pada Gambar 2, maka dari panah-panah pada Gambar 6, operasi-operasi yang telah dilakukan adalah substitution, deletion, dan substitution 2x berturut-turut (panah terakhir tidak dihitung). Sebagai bukti, berikut adalah cara mengubah string "bakso" menjadi "sapu" dengan langkah-langkah di atas:

1. bakso → baksu (substitusi)
2. baksu → baku (penghapusan)
3. baku → bapu (substitusi)
4. bapu → sapu (substitusi)

Catat bahwa solusi yang dihasilkan tidak hanya satu, mungkin saja terdapat solusi lain. Untuk contoh gambar 5, solusi lain adalah melakukan deletion terlebih dahulu, kemudian melakukan substitusi 3x berturut-turut.

Untuk menghitung *edit distance* menggunakan *dynamic programming*, kompleksitas waktu kini hanya menjadi O(mn) (baik kasus terbaik, kasus rata-rata, atau kasus terburuk seluruhnya akan membutuhkan sebanyak m x n operasi).

Berikut adalah *pseudo-code* untuk algoritma *edit distance* dengan program dinamis *bottom-up*.

```
function editDistDPBotUp(str1: String, str2: String) → integer

KAMUS
        m: integer
```

```
        n: integer
        i, j: integer
        dp: matriks[0..m][0..n] of integer

ALGORITMA
        m ← length(str1)
        n ← length(str2)

        for (i = 0; i <= m; i++) do
            for (j = 0; j <= n; j++) do
                if (i = 0) then
                    dp[i][j] = j
                else if (j = 0) then
                    dp[i][j] = i
                else if (str1[i] = str2[j]) then
                    dp[i][j] = dp[i-1][j-1]
                else
                    dp[i][j] = 1 + min(dp[i][j-1],
                                       dp[i-1][j],
                                      dp[i-1][j-1])
        → dp[m][n]
```

IV. IMPLEMENTASI DAN ANALISIS *AUTO-CORRECT* MENGGUNAKAN *EDIT DISTANCE*

Dengan menggunakan *edit distance*, baik menggunakan algoritma brute force atau program dinamis, berikut adalah algoritma program untuk mengimplementasikan *auto-correct* yang telah diimplementasikan dalam bentuk program bahasa Java:

- Menyiapkan sebuah kamus yang terdiri dari kata-kata (di program, kamus memanfaatkan struktur data BST)
- Menyiapkan sebuah priority queue, yang elemennya adalah sebuah kata di kamus dan edit distancenya dengan kata masukan pengguna. Elemen yang paling depan adalah elemen yang nilai edit distancenya paling kecil.
- Meminta masukan kata pengguna. Jika kata pengguna tidak ada di kamus, program menghitung edit distance setiap kata di kamus dengan kata masukan pengguna, kemudian memasukkan hasil perhitungan ke priority queue.
- Menampilkan 10 kata yang menempati queue paling depan.

Pada program ini, kamus yang dimanfaatkan adalah kamus bahasa Inggris yang terdiri atas 99723 kata. Berikut adalah cuplikan hasil *run* program, yang mana Gambar 7 menggunakan algoritma *edit distance* dengan program dinamis, sedangkan Gambar 8 menggunakan *brute force*.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin/java ...
Ketikkan sebuah kata:
lavvy

Kata tidak terdefinisi di kamus.
Suggestions untuk anda:
savvy (ED: 1)
savoy (ED: 2)
livy (ED: 2)
divvy (ED: 2)
navy (ED: 2)
lacey (ED: 2)
wavy (ED: 2)
laity (ED: 2)
lay (ED: 2)
lanky (ED: 2)

Waktu yang dibutuhkan: 219 ms

Process finished with exit code 0
```

Gambar 7. Cuplikan hasil *run* program dengan *edit distance* yang menggunakan program dinamis

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/
Ketikkan sebuah kata:
lavvy

Kata tidak terdefinisi di kamus.
Suggestions untuk anda:
savvy (ED: 1)
savoy (ED: 2)
livy (ED: 2)
divvy (ED: 2)
navy (ED: 2)
lacey (ED: 2)
wavy (ED: 2)
laity (ED: 2)
lay (ED: 2)
lanky (ED: 2)

Waktu yang dibutuhkan: 49386 ms

Process finished with exit code 0
```

Gambar 8. Cuplikan hasil *run* program dengan *edit distance* yang menggunakan *brute force*.

Terlihat bahwa dengan perhitungan *edit distance* dengan menggunakan program dinamis memiliki 250x lebih cepat daripada menggunakan *brute force*.

V. SIMPULAN

Auto-correct merupakan fitur penting pada aplikasi *word-processing* dan sosial media. Salah satu cara untuk mengimplementasikan *auto-correct* dengan baik adalah dengan melakukan perhitungan *edit distance* yang menggunakan algoritma program dinamis.

UCAPAN TERIMA KASIH

Penulis ingin mengucapkan syukur sebesar-besarnya kepada Allah SWT yang atas rahmat dan kuasanya membantu penulis untuk menyelesaikan makalah ini. Penulis juga ingin menyampaikan rasa terima kasihnya kepada Bapak Rinaldi Munir dan Ibu Nur Ulfa Maulidevi atas bimbingannya dalam mata kuliah IF2121 – Strategi Algoritma.

REFERENSI

[1] WhatIs.com, "What is auto-correct?", <http://whatis.techtrget.com> [diakses pada 7 Mei 2016]

[2] Navarro, Gonzalo (1 Maret 2011). "A guided tour to approximate string matching". *ACM Computing Surveys*.

[3] Esko Ukkonen (1983). *On approximate string matching*. Foundations of Computation Theory. Springer. pp. 487–495.

[4] Munir, Rinaldi. "Strategi Algoritmik", Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2007.

[5] Gusfield, Dan (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge, UK: Cambridge University Press. ISBN 0-521-58519-8.

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

Garmastewira - 13514068