Penerapan Penghitungan *Levenshtein Distance* Menggunakan *Dynamic Programming* pada *Spelling Checker* dan *Corrector*

Febi Agil Ifdillah - 13514010
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13514010@std.stei.itb.ac.id - febi_agil@students.itb.ac.id

***Abstrak*—Kemampuan mengeja kata merupakan hal yang sangat penting. Karena ejaan kata berkaitan erat dengan ketersampaian makna dari kata-kata tersebut. Penggunaan teknologi dalam mengoreksi ejaan saat ini ditujukan hanya sebagai alat bantu karena saat ini koreksi hanya dapat dilakukan pada bagian kecil kesalahan yang ada pada kata. Untuk itu, kemampuan mengeja yang dimiliki setiap individu harus tetap dilatih. Di sisi lain, penggunaan spell *checker* dan *corrector* dapat menghemat waktu. Untuk itu, para pengembang dan peneliti terus melakukan perbaikan-perbaikan kemampuan komputer dalam mengoreksi ejaan dan kata-kata. *Spelling Corrector* akan mengubah satu kata menjadi kata lain yang memiliki *Levenshtein Distance* terendah. Karena kata yang perlu diperiksa akan sangat banyak, kita harus melakukan optimasi. Pada paper ini optimasi akan dilakukan pada bagian penghitungan *Levenshtein Distance* dengan *dynamic programming*.**

Kata kunci—Dynamic programming, Levenshtein Distance, spelling checker, spelling corrector, string processing.

I. PENDAHULUAN

Kegiatan membaca memerlukan persepsi terhadap kata-kata secara utuh. Pembaca yang telah lancar dapat melakukan hal tersebut dengan cepat dengan menguasai dan memadukan koneksi antara kombinasi huruf-huruf dan suara yang dihasilkan oleh huruf-huruf tersebut.

Membaca dan mengeja, merupakan dua hal yang berkaitan erat. Belajar untuk mengeja membantu memperkuat koneksi antara huruf-huruf dengan suara dari huruf-huruf tersebut[4]. Joshi, Treiman, Carreker, dan Moats menjelaskan hubungan antara keduanya : "Korelasi antara mengeja membaca secara utuh sangatlah tinggi dikarenakan keduanya saling bergantung satu sama lain dan memiliki kesamaan secara umum: kecapakapan dalam berbahasa"[5].

Kegiatan mengeja, dalam beberapa tahun terakhir telah menjadi subjek bahasan yang begitu ramai. Menurut Joshi, et al. Tujuan utama dari bahasa, yang dalam hal ini adalah bahasa inggris, dan sistem penulisan dalam bahasa tersebut bukan hanya untuk memastikan bahwa pengucapan kata dan penulisannya menjadi akurat, melainkan juga untuk menjamin makna yang ada dalam kata-kata tersebut dapat tersampaikan dengan baik. Seringkali kita melihat beberapa potong paragraf beredar seperti berikut[4] :

"*Aoccdrnig to rscheearch by the Lngiusiitc Dptanmeret at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a total mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe*."

Yang jika ditranslasi akan menjadi :

"*According to research by the Linguistic Department at Cambridge University, it doesn't matter in what order the letters in a word are, the only important thing is that the first and last letter be at the right place. The rest can be a total mess and you can still read it without a problem. This is because the human mind does not read every letter by itself, but the word as a whole.*"

Paragraf tersebut sudah beredar luas sejak tahun 2003. Peredaran paragraf tersebut, entah karena sebagai fakta yang mengejutkan terkait kemampuan otak dalam kecakapan membaca, atau untuk melakukan pembelaan terhadap inkonsistensi dalam mengeja dengan argumen "hanya huruf pertama dan terakhir dalam kata" yang penting.

Namun, kembali lagi kepada tujuan awal dari dibentuknya aturan-aturan dalam bahasa seperti pengucapan, penulisan, dan sebagainya yang ada dalam bahasa-bahasa di dunia, semuanya diperuntukkan demi tersampaikannya makna dari kata-kata yang dipakai sehingga proses komunikasi dapat berjalan dengan baik.

Selain itu, dalam konteks teknologi misalnya, kemampuan tersebut sangat dibutuhkan. Perlu dilakukan pengecekan terhadap tulisan pada *web site* untuk memastikan bahwa tulisan yang dibuat adalah konten yang baik. Baik tidaknya suatu konten dapat diukur dari *accessibility* dan *readibility* dari konten tersebut. Pun sama halnya dengan *search engine* yang tidak mentolerir kesalahan eja. Bahkan *google*, dan *search*

engine lainnya selalu memberikan sugesti kata yang benar jika kita salah eja. Karena kesalahan dalam pengejaan kata dapat membuat hasil dari pencarian menjadi berbeda. Aspek profesionalisme dan kualitas tulisan juga dapat dilihat dari pengejaan dan pemilihan diksi yang tepat.

Mengenai hal tersebut, teknologi merupakan alat bantu yang efektif untuk mengurangi beberapa kesalahan eja yang umum. Untuk itu, penulis akan mencoba menguraikan penerapan teknologi untuk mengatasi kesalahan eja dengan menggunakan algoritma yang menggunakan strategi *Dynamic Programming*.

II. DASAR TEORI

A. Program Dinamis *(Dynamic Programming)*

Dynamic programming adalah pendekatan optimasi yang mentransformasikan program yang kompleks menjadi deretan permasalahan yang lebih sederhana. Program Dinamis (*dynamic programming*), memecahkan permasalahan dengan mengkombinasikan solusi dari pemecahan solusi pada *sub-problems*. Prinsip tersebut juga dapat kita lihat pada metode *divide-and-conquer*. *Divide-and-conquer* melakukan partisi terhadap suatu masalah kedalam beberapa *sub-problems* yang *disjoint*, lalu melakukan penyelesaian terhadap *sub-problems* tersebut secara rekursif. Setelah itu dilakukan proses penggabungan solusi-solusi tersebut untuk menyelesaikan masalah yang sesungguhnya di awal.

Berbeda dengan *divide-and-conquer*, *dynamic programming* menyelesaikan permasalahan yang memiliki *subsubproblems* yang sama atau dengan kata lain, upa-permasalahannya *overlap*. Jika permasalahan tersebut diselesaikan dengan *divide-and-conquer* maka akan terjadi proses pengulangan penyelesaian di bagian *subsubproblems* yang artinya, terdapat proses kerja yang berlebih.

Dynamic programming menyelesaikan *subsubproblems* tersebut hanya sekali lalu menyimpan solusinya dalam tabel. Sehingga dapat menghindari pengulangan pengerjaan *subsubproblems* yang sama. Selain itu, *dynamic programming* memiliki kemiripan dengan *greedy*, hanya saja pada teknik *greedy* hanya terdapat satu buah solusi.

Biasanya, *dynamic programming* digunakan untuk menyelesaikan permasalahan-permasalahan terkait optimasi (*optimization problems)* yang memiliki banyak kemungkinan penyelesaian. Setiap solusi dari permasalahan tersebut memiliki nilai dan kita akan memilih salah satu dari berbagai kemungkinan penyelesaian tersebut yang bernilai optimal. Optimal di sini dapat berarti nilai yang terkecil, maupun terbesar.

Program dinamis dapat digunakan untuk memecahkan berbagai permasalahan yang cukup kompleks seperti *Travelling Salesman Problem (TSP), Capital Budgeting,* tur terpendek pada sebuah graf, dan sebagainya. Secara umum, kita dapat melihat karakteristik penyelesaian persoalan dengan menggunakan program dinamis, yaitu sebagai berikut[Rinaldi] :

1. Terdapat sejumlah berhingga pilihan yang mungkin.
2. Solusi pada setiap tahap dibangun dari hasil solusi tahap sebelumnya.
3. Digunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

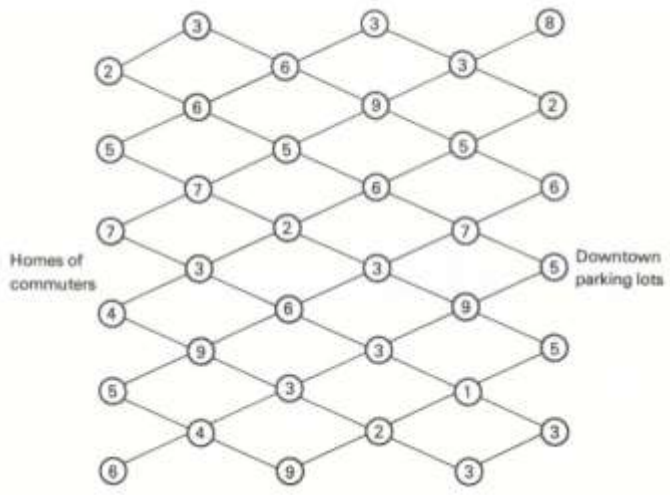
Pada program dinamis, rangkaian keputusan yang optimal dibuat berdasarkan prinsip optimalitas. Yakni prinsip yang mengatakan bahwa jika solusi total merupakan solusi optimal, maka bagian solusi sampai tahap ke-k juga optimal. Prinsip tersebut menjamin bahwa pengambilan keputusan pada suatu tahap adalah keputusan yang benar untuk keputusan pada tahap-tahap selanjutnya. Misalnya, persoalan *shortest path* memenuhi prinsip optimalitas. Karena jika a, x1, x2, … , xn, b adalah jalur terpendek dari simpul a ke simpul b di dalam sebuah graf, maka bagian xi ke xj pada jalur tersebut pun adalah jalur terpendek dari xi ke xj. Prinsip tersebut membuat program dinamis tidak perlu kembali ke tahap awal karena menggunakan hasil optimal dari tahap sebelumnya.

Persoalan yang dapat diselesaikan dengan teknik program dinamis memiliki karakteristik sebagai berikut[1] :

1. Persoalan dapat dibagi menjadi beberapa tahap(*stage*), dan setiap tahap hanya dapat diambil satu keputusan.
2. Masing-masing tahap terdiri dari sejumlah status(*state)* yang berhubungan dengan tahap tersebut. Status yang dimaksud adalah berbagai macam kemungkinan masukan yang ada pada tahap tersebut. Masing-masing tahap dapat digambarkan dengan menggunakan graf multitahap(*multistage graph)*, dengan simpul-simpul pada graf tersebut merepresentasikan status. Sedangkan busur pada graf menyatakan tahap.
3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap selanjutnya.
4. Ongkos(*ongkos)* pada suatu tahap meningkat secara teratur dengan bertambahnya jumlah tahapan.
5. Ongkos pada suatu tahap bergantung pada ongkos pada tahap yang sudah berjalan sebelumnya 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 sehingga dapat diambil keputusan terbaik untuk setiap status pada tahap k+1.
8. Prinsip optimalitas berlaku pada persoalan tersebut.

Salah satu contoh persoalan yang dapat diselesaikan oleh program dinamis adalah persoalan yang digambarkan pada gambar 1. Gambar tersebut merepresentasikan peta jalan yang menghubungkan perumahan dengan tempat parkir untuk sekelompok pengemudi dalam suatu kota. Busur pada gambar

tersebut menggambarkan jalan dan simpul-simpul merepersentasikan persimpangan.



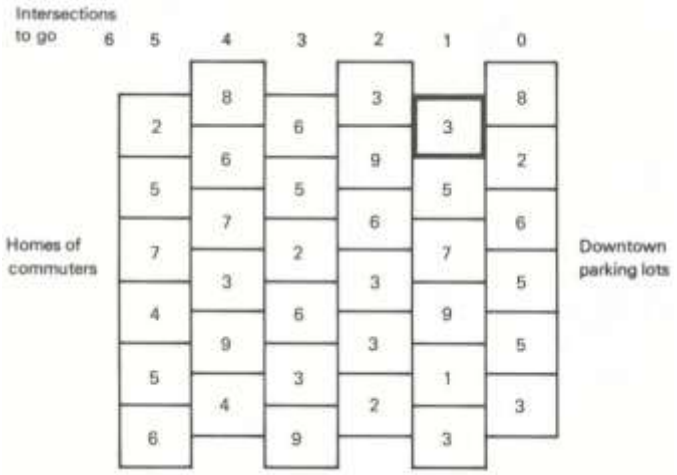
Gambar 1 - Contoh permasalahan dengan multitahap. Sumber : MIT (http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf)

Graf tersebut didesain memiliki *pattern* berlian agar setiap pengguna jalan harus melewati lima jalan untuk berkendara dari rumah ke *downtown*. Setiap pengguna akan mengalami *delay* pada setiap persimpangan jalan yang dilalui dan lama penundaan tersebut dapat diketahui dari angka yang tertera pada setiap persimpangan. Jalur yang dipilih tidak memengaruhi waktu penundaan. Artinya, dari arah manapun pengendara datang dan pergi, ia tetap akan mengalami *delay* dengan waktu yang sama. Persoalan ini meminta kita untuk meminimalkan waktu tunggu dalam satuan menit.

Cara paling naif yang dapat dilakukan untuk menyelesaikan persoalan ini adalah dengan menggunakan *brute force*, yakni mengenumerasi 150 jalur pada diagram. *Namun, apabila kita menggunakan* dynamic programming perhitungan yang dilakukan dapat dikurangi dengan cara berpindah secara sistematis dari satu sisi ke sisi lainnya sambil membangun solusi terbaik.

Dalam program dinamis, terdapat dua pendekatan yang dapat digunakan, yaitu program dinamis maju (*forward atau Top-down*) dan program dinamis mundur (*backward* atau *bottom-up*). Ketika mengembangkan sebuah algoritma dengan metode *dynamic-programming*, kita dapat mengikuti empat langkah berikut[MIT] :

1. Karakterisikkan struktur dari solusi yang optimal.
2. Definisikan nilai dari solusi yang optimal secara rekursif.
3. Lakukan perhitungan nilai dari sebuah solusi optimal, biasanya menggunakan *bottom-up fashion.*
4. Konstruksikan solusi optimal berdasarkan perhitungan yang telah dilakukan.



Gambar 2 - Penyederhanaan representasi dari gambar 1, pada setiap persimpangan hanya dapat memilih dua jalur. Sumber : MIT (http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf)

B. Levenshtein *Distance*

Jarak Levenshtein (Levehnstein *Distance*) adalah angka terkecil dari penyisipan, penghapusan, dan substitusi yang dibutuhkan untuk mengubah suatu *string* menjadi *string* lainnya [7]. Dengan kata lain, jarak Levenshtein adalah jumlah operasi minimum untuk melakukan pengubahan satu *string* ke *string* lain. Dalam teori informasi dan ilmu komputer, Levenshtein Distance merupakan satuan pengukuran dari perbedaan antara dua *string*. Levenshtein *Distance* diciptakan oleh seorang bernama Vladimir Levenshtein pada tahun 1965.

String "POMME" dan "POIRE" ada Gambar 3 memiliki levenshtein distance sebesar 2, dan *string* "LEVENSHTEIN" dan "FRANKENSTEIN" memiliki levenshtein distance sebesar 6.

Gambar 3 - Contoh penghitungan levenshtein distance.

Sumber :
http://homepages.ulb.ac.be/~dgonze/TEACHING/levenshtein.pdf

Penghitungan *levenshtein distance* akan dibahas lebih lanjut pada bagian selanjutnya.

III. Menghitung Levenshtein Distance Dengan Dynamic Programming

Levenshtein *Distance* atau *edit distance* adalah jumlah operasi minimum yang diperlukan untuk membuat *string x* menjadi *string y*. Levenshtei*n Distance* diperlukan untuk menentukan kedekatan antara dua buah *string*. Mencari levenshtein distance adalah inti dari spelling corrector yang dibuat pada tulisan kali ini.

A. Metode Naif

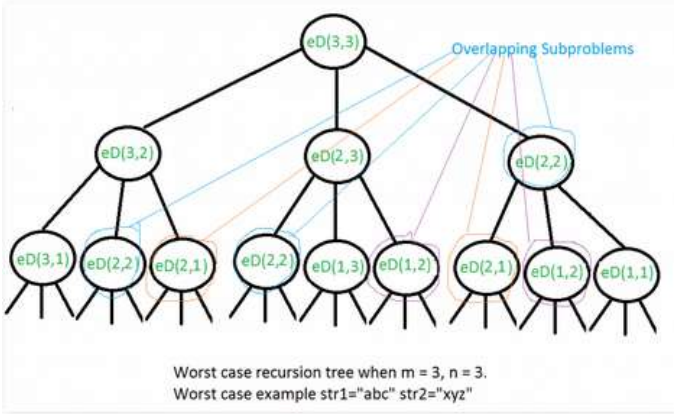
Misalkan string 1 : αx, dan string 2 = βy
Maka,

editdistance(αx, βy) = min(editdistance(α,β) + δ(x,y), editdistance(αx, β) +1, editdistance(α,βy) +1)

Berikut adalah translasi dari persamaan tersebut ke dalam bahasa python,

```
def editDistance(a,b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    delta = 1 if a[-1] != b[-1] else 0
    return min(editDistance(a[:-1], b[:-1]) + delta,
               editDistance(a[:-1], b) + 1,
               editDistance(a, b[:-1]) + 1);
```

Gambar 4 - Implementasi fungsi untuk mencari Levenshtein Distance dengan menggunakan bahasa python



Gambar 5 - Overlapping sub problem pada pemecahan masalah. Sumber : http://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/

Kompleksitas waktu dari algoritma di atas adalah eksponensial dan dalam kasus terburuk bisa mencapai O(3^m) operasi. Kasus terburuk terjadi ketika tidak ada satupun karakter dari kedua string yang cocok.

Algoritma ini pun mengalami redundansi dalam pemecahan *sub-problem*, yang digambarkan oleh gambar 5. Untuk kasus *string* "STIMA2014" dan "STIM214" terdapat 231 kali pengulangan pemanggilan fungsi dengan argumen "STIM" untuk kedua *string*. Hal tersebut terjadi karena adanya sub-problems yang mengalami *overlapping*.

Adanya *overlapping* pada solusi pencarian levenshtein distance juga mengakibatkan waktu eksekusi menjadi sangat lama. Hal tersebut sangatlah merugikan dari segi sumberdaya dan waktu. Bahkan, untuk *string* "LEVENSHTEIN" dan "FRANKENSTEIN", diperlukan waktu lebih dari 1 menit 42 detik. Dengan pengulangan pemanggilan argumen "LEVEN" dan "FRANKEN" sebanyak 3653. Jumlah yang sangat besar.

```
Levenshtein Distance :  0
Jumlah pemanggilan prefix yang sama 3653
Waktu eksekusi :  0:01:42.745571
```

Gambar 7 - Waktu eksekusi dari pencarian levenshtein distance untuk

Algoritma tersebut dapat diperbaiki menjadi jauh lebih cepat untuk kasus di atas dan lainnya, dengan tanpa melakukan pemotongan string argumen. Maksudnya adalah, di setiap pemanggilan fungsi, string tidak akan dimanipulasi dan hanya akan melakukan pengecekan karakter terakhir untuk setiap string. Jika karakter tersebut sama, maka tidak perlu dilakukan operasi, artinya levenshtein distancenya tidak bertambah. Namun jika karakter-karakter terakhir dari kedua string berbeda, maka akan dicek lebih lanjut apakah ia melakukan salah satu operasi dari tiga (substitusi, penyisipan, penghapusan) sehingga levenshtein distance bertambah satu.

Untuk melakukan perbaikan pada algoritma tersebut, kita akan mengubah definisi dari fungsi yang telah dibuat sebelumnya. Fungsi kini akan memiliki empat parameter, yang pertama adalah str1 menandakan string pertama, yang kedua str2 untuk string kedua, selanjutnya adalah m dan n, yaitu panjang string pertama dan kedua, berurutan.

```
return 1 + min(editDistance(str1, str2, m, n-1),     # Insert
               editDistance(str1, str2, m-1, n),     # Remove
               editDistance(str1, str2, m-1, n-1)    # Replace
               )
```

Gambar 7 - Optimasi kode

Kita dapat melihat potongan program yang sudah diperbarui pada gambar 7. Operasi *insert* akan melakukan pemanggilan fungsi editDistance secara rekursif dengan m dan n-1. *Remove* akan memanggil editDistance dengan parameter m-1 dan n. *Replace* akan memanggil editDistance dengan parameter m-1, dan n-1. Potongan kode pada gambar 8 digunakan untuk memenuhi kondisi yang kita tentukan sebelumnya, bahwa jika karakter terakhir dari kedua string sama, tidak perlu melakukan operasi apapun.

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

Gambar 8 - Potongan kode untuk menangani kasus karakter terakhir pada kedua string sama

Dengan optimasi tersebut, kita dapat mempercepat waktu eksekusi. Bahkan, untuk kata "LEVENSHTEIN" dan "FRANKENSTEIN" waktu eksekusi dapat dipercepat sebanyak 4711 kali. Waktu eksekusi untuk kode yang telah dioptimasi dapat kita lihat pada gambar 9.

```
febiagil@febiagil:~/grive/SEMESTER 4/
hon ediDistance2.py
Waktu eksekusi DP :  0:00:00.021650
Levenshtein Distance :  6
```

Gambar 9 - Hasil eksekusi dari kode yang telah dioptimasi. Waktu eksekusi lebih cepat 4711 kali

B. Cara yang lebih baik – Dynamic Programming

Metode yang ditunjukkan pada bagian A ternyata memiliki kelemahan, karena sifatnya yang rekursif ada kemungkinan program tersebut akan memanggil kembali prefix atau string yang sebenarnya sudah ditemukan solusinya. Setelah dioptimasi, ada perbaikan yang signifikan. Namun, kita masih dapat melakukan optimasi dan membuat algoritma tersebut menjadi lebih cepat dengan menggunakan *dynamic programming*. Secara visual, kita dapat menggunakan tabel seperti di bawah ini. Jawaban, atau levenshtein distance, berada pada tabel dengan indeks (length(x)-1, length(y)-1). Karena kita menggunakan indeks mulai dari 0.

		x1	x2	...	xi	...
	0	1	2	3	4	5
y1	1					
y2	2		$d_{i-1, j-1}$	$d_{i, j-1}$		
...	3		$d_{i-1,j}$	$d_{i,j}$		
yj	4					
...	5					

Untuk mengisi sel pada indeks (length(x)-1, length(y)-1) dan mendapatkan jawaban, kita perlu mengisi sel-sel pada baris dan kolom lainnya terlebih dahulu. Pertama kita akan mengisi kolom pertama dan baris pertama pada baris tersebut dengan angka 0, 1, … n berurutan. Kembali kita ingat persamaan pada metode naïf sebelumnya, kolom dan baris lainnya diisi dengan nilai yang dihitung melalui persamaan sebagai berikut ,

Misalkan string 1 : αx, dan string 2 = βy
Maka,

$editdistance(\alpha x, \beta y) = min(editdistance(\alpha,\beta) + \delta(x,y), editdistance(\alpha x, \beta) +1, editdistance(\alpha,\beta y) +1)$

Atau, secara sederhana kita dapat katakan bahwa jika, xn dan yn bukan karakter yang sama, maka kita dapat mencari di,j dengan cara :

$$di,j = min(di,j-1 , di-1,j, di-1,j-1)$$

Namun jika xn dan yn adalah karakter yang sama, maka $d_{i,j} = d_{i-1, j-1}$. Mari kita mulai dari contoh yang sederhana. Untuk kata "STIMA" dan "STM" kita dapat menghitung *levenshtein distance* dengan menggunakan *dynamic programming* dan diakhir perhitungan, akan terbentuk tabel sebagai berikut :

		S	T	I	M	A
	0	1	2	3	4	5
S	1	0	1	2	3	4
T	2	1	0	1	2	3
M	3	2	1	1	1	2

Untuk melakukan pengisian nilai terhadap tabel, kita dapat mengisi kolom pertama dan baris pertama dengan angka 0, 1, … n, sesuai dengan banyaknya karakter yang ada pada baris atau kolom. Pengisian nilai pertama ditandai dengan warna merah. Hal yang perlu diketahui adalah, ketika kita hendak mengisi sel (0,0) tidak ada karakter yang terlibat untuk string x, begitupun string y. Untuk mengisi sel (1,1), karakter yang terlibat untuk *string* x adalah S, dan *string* y adalah S. Untuk mengisi sel (2,3) karakter yang terlibat adalah STI untuk y, dan ST untuk x. Contoh lainnya adalah, untuk mengisi sel (3,2) karakter-karakter yang terlibat untuk *string* x adalah STM, dan *string* y melibatkan ST, dan seterusnya.

Setelah baris pertama dan kolom pertama terisi, kita akan melanjutkan pengisian pada indeks 1,1. Pada sel tersebut kita membandingkan sel (0,0), (0,1), dan (1,0) kemudian mencari nilai minimum di antara ketiga sel tersebut. Dapat kita ketahui bahwa nilai minimumnya adalah 0 karena untuk pada *string* x melibatkan S, dan y pun S, karakter terakhir dari kedua string tersebut sekarang sama-sama S sehingga tidak perlu ditambahkan 1 (operasi). Untuk sel indeks (1, 2) kita lakukan perbandingan antara sel (0,1), (0,2) dan (1,1). Nilai minimum dari ketiga sel tersebut adalah 0. Karena karakter terakhir dari y saat ini adalah T, dan karakter terakhir pada x saat ini S, maka keduanya tidak lah sama sehingga nilai untuk indeks (1,2) ditambah satu menjadi 1, menandakan ada satu operasi terjadi, begitu seterusnya. Setelah selesai melakukan perhitungan tabel kita menghasilkan *levenshtein distance* sebesar dua. Nilai tersebut dapat kita lihat pada sel yang diberi tanda warna kuning.

Angka dua yang kita dapat pada perhitungan tersebut menandakan bahwa kita memerlukan dua buah operasi (penyisipan, penghapusan, atau substitusi) untuk membuat string pertama (STM) dan string kedua (STIMA) menjadi

sama. Jika kita perhatikan pada table di bawah ini, kita dapat menentukan operasi apa yang akan dilakukan dari bagaimana pemilihan angka minimum untuk mengisi tabel indeks (i,j). Caranya adalah dengan menelusuri sel dari indeks (length(x)-1, length(y)-1) sampai ke indeks (0,0). Kita mulai dari sel dengan indeks (3, 5), yaitu hasil dari penghitungan, lalu kita melihat dari mana nilai pada sel tersebut(dua) berasal.

Nilai pada sel (3, 5) didapatkan dari nilai minimum antara sel dengan indeks(3,4), (2,4) dan (2,5), yaitu (3,4) sehingga kita akan menuju indeks (0,0) melewati (3,4). Kemudian kita menuju sel dengan indeks (2,3) yaitu searah diagonal karena ST[M] dan STI[M] berakhiran M. Lalu ke sel dengan indeks (2,2) dan seterusnya sampai berakhir di indeks (0,0). Sel yang dilewati pada penelusuran tersebut ditandai dengan warna biru. Dapat dikatakan, bahwa kita perlu mengingat bagaimana cara kita mencapai indeks (3,5).

		S	T	I	M	A
	0	1	2	3	4	5
S	1	0	1	2	3	4
T	2	1	0	1	2	3
M	3	2	1	1	1	2

Lalu, apa arti dari penandaan tersebut? Secara semantik, pergeseran ke sel dengan indeks (i, j-1) menandakan terjadi operasi penghapusan karakter pada string y yang berkaitan dengan indeks (i,j). Misalnya pada indeks(3,5) bergeser ke (3,4) berarti karakter A pada string STIMA dihapus. Jika terjadi pergerakan ke indeks(i-1, j-1) atau secara diagonal dan karakter terkait sama, maka tidak terjadi operasi apapun. Namun, jika karakter terkait tidak sama dan kita sedang bergerak menelusuri secara diagonal, maka karakter terkait pada string y akan digantikan karakter yang berkaitan pada string x. Jika penelusuran bergerak dari indeks(i,j) ke (i-1, j) maka operasi yang dilakukan adalah penyisipan. Sekali lagi perlu diingat bahwa ketentuan tersebut berlaku karena penelusuran yang kita lakukan untuk menentukan operasi bermula dari ujung paling kiri bawah tabel dan akan bergerak ke sel dengan indeks (0,0).

```
febiagil@febiagil:~/grive/SEMESTER 4/Strate
Waktu eksekusi :  0:00:00.000030
Levenshtein Distance :  6
[[ 0  1  2  3  4  5  6  7  8  9 10 11 12]
 [ 1  1  2  3  4  5  6  7  8  9 10 11 12]
 [ 2  2  2  3  4  5  5  6  7  8  9 10 11]
 [ 3  3  3  3  4  5  6  6  7  8  9 10 11]
 [ 4  4  4  4  4  5  5  6  7  8  8  9 10]
 [ 5  5  5  5  4  5  6  5  6  7  8  9  9]
 [ 6  6  6  6  5  5  6  6  5  6  7  8  9]
 [ 7  7  7  7  6  6  6  7  6  6  7  8  9]
 [ 8  8  8  8  7  7  7  7  7  6  7  8  9]
 [ 9  9  9  9  8  8  7  8  8  7  6  7  8]
 [10 10 10 10  9  9  8  8  9  8  7  6  7]
 [11 11 11 11 10 10  9  8  9  9  8  7  6]]
```

Gambar 10 - Menggunakan Dynamic Programming, waktu eksekusi berkurang signifikan

Algoritma penghitungan levenshtein distance yang menggunakan *dynamic programming* memberikan performansi yang lebih baik. Waktu eksekusi menjadi lebih singkat secara signifikan. Untuk kata "LEVENSHTEIN" dan "FRANKENSTEIN", saat ini penyelesaiannya hanya membutuhkan 0,000030 detik seperti di gambar 10.

IV. Penerapan pada Spelling Corrector

A. Metode Naif

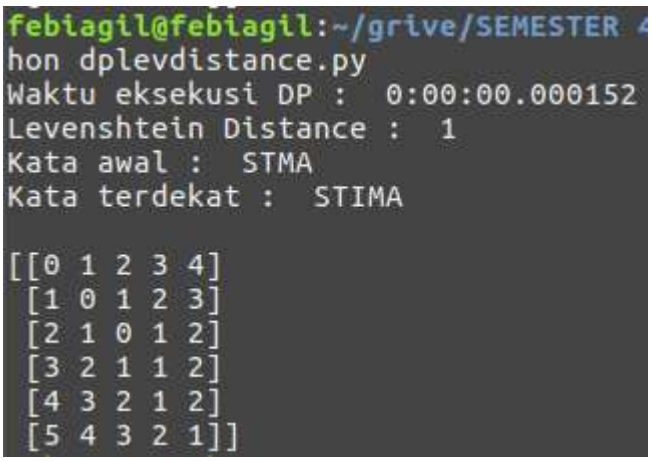
Andaikan kita punya n buah kata dalam kamus. Lalu kita memiliki sebuah kata yang akan kita koreksi menggunakan *spelling corrector*. Metode naif untuk melakukan koreksi adalah dengan menghitung levenshtein distance dari kata yang akan dikoreksi tersebut dengan semua kata yang ada dalam kamus. Lalu memilih kata mana yang memiliki levenshtein distance terkecil. Untuk mensimulasikan metode ini, kita dapat melihat potongan kode pada gambar 11.

```
kamus = ['AI', 'BASDAT','PROBSTAT', 'STIMA']

string2 = "STMA"

startTime = datetime.now()

nilaimin=9999
kataterdekat=""
for member in kamus :
    string1 = member
    len1 = len(string1)
    len2 = len(string2)
    result = editDistDP(string1, string2, len1, len2)

    if result[len1][len2]<nilaimin :
        nilaimin=result[len1][len2]
        kataterdekat = member

z = datetime.now() - startTime

print "Waktu eksekusi DP : ", z
print "Levenshtein Distance : ", result[len1][len2]
print "Kata terdekat : ", kataterdekat
print
print(np.matrix(result))
```

Gambar 11 - Potongan kode koreksi string dengan metode naif

Pada kode tersebut, kita melakukan *typo* dengan mengetikkan *string* "STMA", padahal yang kita maksud adalah "STIMA". Kamus yang kita miliki memiliki 4 buah *string*, yaitu AI, BASDAT, PROBSTAT, dan STIMA. Menggunakan metode naif, kita akan melakukan proses penghitungan *levenshtein distance* sebanyak empat kali dan memilih angka terkecil dan kata yang berkorespondensi dengannya. Potongan kode di atas menghasilkan koreksi yang benar.

Walaupun kita menggunakan penghitungan levenshtein distance menggunakan *dynamic programming*, metode naif akan menghabiskan sumber daya dan waktu yang cukup besar terutama jika kamus yang kita miliki terdiri dari ratusan ribu kata.



Gambar 12 - Hasil eksekusi spelling corrector

B. Metode yang lebih baik

Metode yang lebih baik dibandingkan *brute force* seperti pada bagian A adalah dengan menghasilkan semua kemungkinan dengan *Damerau-Levenshtein Distance* yang lebih kecil atau sama dengan dua. Damerau-Levenshtein Distance memperbolehkan kita melakukan empat operasi yaitu penghapusan, transpos, substitusi, dan penyisipan dari *query* yang akan kita koreksi lalu hasil tersebut kita cari dalam kamus[peternorvig].

Untuk sebuah kata dengan panjang n, dengan ukuran alfabet a, *edit distance* d=1, maka akan terjadi n buah penghapusan, n-1 transposisi, a*n pengubahan, dan a*(n+1) penyisipan dengan total kata yang digenerate dalam waktu pencarian sebanyak 2n+2an+a-1. Waktu ini jauh lebih baik daripada metode naif.

V. KESIMPULAN

Teknologi adalah alat bantu yang sangat baik yang dapat membantu kita dalam berbagai hal. Bahkan beberapa orang mungkin akan mengatakan bahwa belajar mengeja tidak lagi diperlukan karena teknologi dapat mengatasi hal tersebut dengan *spell checker*. Namun, perlu diingat bahwa *spell checker* hanyalah alat yang hanya dapat mengatasi kesalahan eja yang umum. Mengenai hal tersebut, *spell checker* sangatlah efektif dan menghemat waktu.

Penggunaan *levenshtein distance* untuk mencari string terdekat yang akan dipakai untuk mengoreksi suatu string lainnya dapat memberikan hasil yang cukup memuaskan. Dengan menggunakan *dynamic programming*, waktu pencarian solusi untuk *levenshtein distance* dapat dikurangi secara signifikan. Namun tidak hanya cukup sampai di situ, optimasi untuk *spelling checker* perlu dilakukan pada bagian-bagian lainnya. Karena pencarian string terdekat dengan *brute force* akan menghabiskan banyak sumberdaya.

UCAPAN TERIMA KASIH

Penulis ingin memanjatkan syukur kepada Allah SWT yang telah memberikan nikmat dan karunia-Nya sehingga penulis mampu belajar dan menyelesaikan makalah ini dengan tepat waktu. Saya juga ingin mengucapkan terimakasih kepada kedua orang tua saya yang begitu luar biasa. Makalah berjudul "Penerapan Penghitungan *Levenshtein Distance* Menggunakan *Dynamic Programming* pada *Spelling Checker* dan *Corrector*" ini tidak akan terwujud tanpa bimbingan dari Dr Ir. Rinaldi Munir, MT dan Dr. Nur Ulfa Maulidevi, ST., M.Sc. Terimakasih atas kesabarannya dalam membimbing dan teruslah berkarya.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. 2009. "Diktat Kuliah IF 2251 Strategi Algoritmik". Bandung: Program Studi Teknik Informatika STEI ITB.

[2] Halim, Steven., Halim, Felix, 2013. "Competitive Programming, 3rd Edition".

[3] Cormen, Thomas H., et al. "Introduction to Algorithms, 3rd Edition". MIT Press.

[4] Joshi, R., Treiman, R., Carreker, S., & Moats, L.. (2008-2009, Winter). The real magic of spelling: Improving reading and writing. American Educator , 9.http://www.aft.org/sites/default/files/periodicals/joshi.pdf p. 10.

[5] http://www.spellingcity.com/importance-of-spelling.html The Importance of Spelling by Susan Jones, M.Ed. 2/2009 Diakses pada Sabtu, 7 Mei 2016 jam 11.37

[6] http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf diakses pada Sabtu, 7 Mei 2016 13.26 WIB

[7] https://xlinux.nist.gov/dads//HTML/Levenshtein.html diakses pada Sabtu, 7 Mei 2016 20.12 WIB

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Mei 2016

Febi Agil Ifdillah (13514010)