Penerapan Algoritma Levenshtein Distance untuk Mengoreksi Kesalahan Pengejaan pada Editor Teks

Muhammad Bahari Ilmy[1], Nitia Rahmi[2], Roland L Bu'ulölö[3]

Laboratorium Ilmu dan Rekayasa Komputasi
Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail : if14062@students.if.itb.ac.id[1],
if14068@students.if.itb.ac.id[2], if14072@students.if.itb.ac.id[3]

Abstrak

Kesalahan pengejaan pada editor teks sering kali terjadi. Pemeriksaan kesalahan pengejaan biasa dilakukan ketika tulisan telah selesai dibuat. Pada teks yang pendek, hal tersebut tentu tidaklah sulit. Namun, ketika ukuran teks sudah mencapai lebih dari sepuluh ribu kata atau bahkan jutaan kata, pemeriksaan dengan cara tersebut sangat menyulitkan. Dengan penerapan sebuah program, masalah tersebut dapat diatasi. Dalam lingkup yang kecil, misalnya kegiatan mahasiswa, penerapan ini akan sangat berguna , misalnya pada pengoreksian karya tulis, skripsi dan sebagainya. Algoritma yang mangkus untuk mengoreksi kesalahan dengan pemrograman dinamis berdasarkan perbandingan string akan dibahas lebih lanjut dalam makalah ini.

Kata kunci: *program dinamis, levenshtein distance, edit distance.*

1. Pendahuluan

Kesalahan penulisan pada editor teks adalah hal yang paling sering terjadi. Sebuah program yang dapat mengoreksi kesalahan penulisan akan sangat berguna terutama ketika teks yang ditulis cukup panjang. Algoritma yang mangkus untuk menemukan kesalahan pengejaan berdasarkan kemiripan penulisan antara teks yang tertulis dengan database pada program akan sangat berguna untuk mengatasi permasalahan ini.
Tujuan pembuatan makalah ini adalah sebagai berikut.

1. Mempelajari metode pencocokan string dengan menggunakan pemrograman dinamis.
2. Mempelajari algoritma **Levenshtein Distance** dengan metode rekursif.
3. Menerapkan perbaikan pada algoritma Levenshtein Distance agar kompleksitas ruangnya lebih efisien.

2. Ruang Lingkup

Makalah ini membahas sebuah algoritma yang mangkus untuk mengoreksi kesalahan pengejaan. Kesalahan pengejaan tersebut ditemukan dengan menggunakan algoritma **Levenshtein Distance** yang didasarkan pada kemiripan pengejaan. Pengejaan yang salah akan dikoreksi/ditemukan kata yang mirip dengannya sebagai koreksi.

3. Teknik Pengoreksian

Editor teks, terutama *word processor* memiliki fasilitas untuk mengecek *spelling* kata-kata yang ada dalam dokumen. Pengecekan ini juga dapat membantu dalam pengoreksian kesalahan pengejaan tersebut. Koreksi hanya dapat dilakukan pada kata yang dikenal bahasanya. Oleh karena itu, diperlukan *database* bahasa yang sesuai.

Algoritma pemrograman dinamis diterapkan pada pengecekan setiap huruf di dalam kata terhadap kata-kata yang terdapat didalam *database*. Namun, pengecekan ejaan kata hanya dilakukan pada kata-kata di dalam kamus yang tidak terlalu jauh perbedaannya sehingga tidak perlu dilakukan perbandingan terhadap seluruh kata-kata di dalam *database* kamus.

Untuk menemukan kata yang dekat/tidak jauh perbedaannya dengan kata yang sedang dicek digunakan algoritma pencarian. Pencarian dilakukan hanya sebatas pada kata-kata di dalam *database* kamus yang mendekati kata yang sedang dicek. Hal ini dilakukan agar tidak perlu mencari di seluruh *database* kamus.

4. Algoritma *Levenshtein Distance*

Dalam teori informasi, *Levenshtein distance* dua *string* adalah jumlah minimal operasi yang dibutuhkan untuk mengubah suatu *string* ke *string* yang lain, di mana operasi-operasi tersebut adalah operasi penyisipan, penghapusan, atau penyubstitusian sebuah karakter. Algoritma ini

dinamakan berdasarkan **Vladimir Levenshtein** yang ditemukannya pada tahun 1965. Pada makalah ini, *Levenshtein distance* dirujuk dengan menggunakan kata jarak saja agar lebih singkat.

Algoritma dasar penentuan jarak dua *string* ini dapat dibentuk melalui hubungan rekursif.

Basis:

levDis(“”, “”) = 0
levDis(s , “”) = levDis(“” , s) = |s|

Di atas terdapat dua basis. Baris pertama menyatakan dengan jelas bahwa dua *string* kosong tidak memiliki jarak, berarti untuk mengubah *string* yang satu ke yang lain tidak diperlukan operasi apapun. Baris kedua menyatakan bahwa jarak antara suatu *string* tidak kosong dengan *string* kosong adalah sebesar panjang (jumlah karakter) di dalam *string* yang tidak kosong.

Rekurens:

```
levDis( s1+c1 , s2+c2 )
  =
    min (  ( levDis( s1 , s2 ) +
           ( if (c1 = c2) then
               0
             else
               1
           endif  ) ),
           (levDis( s1 + c1 , s2       ) + 1) ,
           (levDis( s1      , s2 + c2 ) + 1)
    )
```

Di atas tertulis bahwa kedua *string* yang dibandingkan tidak kosong. Keduanya memiliki karakter terakhir c1 dan c2. Di sini dijelaskan bahwa terdapat tiga alternatif untuk menentukan jarak/ kedua *string*. **Pertama**, Jika c1 dan c2 sama, maka c1 dan c2 tidak perlu dipertukarkan berarti jaraknya *levDis(s1,s2) + 0*, jika berbeda berarti hanya tinggal mengubah c1 menjadi c2 saja, berarti jaraknya *levDis(s1,s2) + 1*. Dalam hal ini operasi yang dilakukan adalah operasi substitusi. **Kedua**, dapat juga dilakukan operasi penghapusan c1 dari s1 dan mengubahnya menjadi s2 + c2 sehingga jaraknya menjadi *levDis(s1,s2+c2) + 1*. **Ketiga,** mirip seperti yang kedua dapat dilakukan juga operasi penyisipan c2 pada s1 + c1 yang telah diubah menjadi s2 sehingga jaraknya adalah *levDis(s1+c1,c2) + 1*. Ketiga alternatif di atas adalah semua kemungkinan perubahan yang ada, dan dari antara ketiganya dicari yang mana yang paling sedikit jaraknya dengan fungsi min yang mencari nilai paling minimum di antara tiga nilai.

Dalam implementasi algoritma rekursif ini, terdapat tiga kali pemanggilan rekursif untuk setiap rekurens. Hal ini membuat algoritma ini menjadi sangat lambat dan hanya baik digunakan pada *string* yang terdiri dari karakter-karakter yang sedikit saja. Oleh karena itu, pemeriksaan lebih lanjut menunjukkan bahwa jarak s1 dan s2 bergantung pada jarak s1’ dan s2’ saja di mana s1’ lebih pendek dari s1, dan s2’ lebih pendek dari s2. Sementara jarak s1’ dan s2’ bergantung pada jarak s1” dan s2” di mana keduanya lebih pendek dari yang sebelumnya. Hal ini menunjukkan bahwa teknik pemrograman dinamis dapat digunakan.

Untuk menghitung jaraknya tanpa menggunakan proses rekursif, digunakan matriks *(n + 1) × (m + 1)* di mana n adalah panjang *string* s1 dan m adalah panjang *string* s2. Berikut dua *string* yang akan digunakan sebagai contoh:

RONALDINHO
ROLANDO

Jika kita melihat sekilas, kedua *string* tersebut memiliki jarak 6. Berarti untuk mengubah *string* RONALDINHO menjadi ROLANDO diperlukan 6 operasi, yaitu:

1. Mensubstitusikan N dengan L
   RO**N**ALDINHO → RO**L**ALDINHO
2. Mensubstitusikan L dengan N
   ROLA**L**DINHO → ROLA**N**DINHO
3. Mensubstitusikan I dengan O
   ROLAND**I**NHO → ROLAND**O**NHO
4. Menghapus O
   ROLANDONH**O** → ROLANDONH
5. Menghapus H
   ROLANDON**H** → ROLANDON
6. Menghapus N
   ROLANDO**N** → ROLANDO

Dengan menggunakan representasi matriks dapat ditunjukkan tabel berikut:

		R	O	N	A	L	D	I	N	H	O
	0	1	2	3	4	5	6	7	8	9	10
R	1										
O	2										
L	3										
A	4										
N	5										
D	6										
O	7										

Pada tabel ini, elemen baris 1 kolom 1 (M[1,1]) adalah jumlah operasi yang diperlukan untuk mengubah *substring* dari kata ROLANDO yang diambil mulai dari *karakter awal sebanyak 1* (R) ke *substring* dari kata RONALDINHO yang diambil mulai dari *karakter awal sebanyak 1* (R). Sementara elemen M[3,5] adalah jumlah operasi antara ROL (*substring* yang diambil mulai dari karakter awal sebanyak 3) dengan RONAL (*substring* yang diambil mulai dari karakter awal sebanyak 5). Berarti elemen M[p,q] adalah jumlah operasi antara

substring kata pertama yang diambil mulai dari awal sebanyak p dengan ***substring kata kedua yang diambil dari awal sebanyak q***. Sehingga dengan peraturan ini matriks dapat diisi, menghasilkan:

		R	O	N	A	L	D	I	N	H	O
	0	1	2	3	4	5	6	7	8	9	10
R	1	0	1	2	3	4	5	6	7	8	9
O	2	1	0	1	2	3	4	5	6	7	8
L	3	2	1	1	2	3	4	5	6	7	8
A	4	3	2	2	1	2	3	4	5	6	7
N	5	4	3	3	2	2	3	4	5	6	7
D	6	5	4	4	3	3	2	3	4	5	6
O	7	6	5	5	4	4	3	3	4	5	6

Elemen terakhir (yang paling kanan bawah) adalah elemen yang nilainya menyatakan jarak kedua *string* yang dibandingkan. Sehingga algoritma untuk mengisi matriks sesuai dengan yang di atas adalah:

```
function levDis (s1 : string, s2 : string) : integer
kamus
  i, j, cost : integer
  m : array [0 .. s1.length, 0 .. s2.length] of integer
algoritma
  for i ← 0 to s1.length do
    for j ← 0 to s2.length do
      if i = 0 then
        m[i,j] ← j {perbandingan dengan kosong}
      else if j = 0 then
        m[i,j] ← i {perbandingan dengan kosong}
      else {implementeasi pemrograman dinamis}
        if s1[i] = s2[j] then
          cost ← 0
        else
          cost ← 1

        m[i,j] = minimum (
                    m[i-1, j-1] + cost, {substitusi}
                    m[i-1,j] + 1, {penghapusan}
                    m[i ,j-1] + 1, {penambahan}
                  )

  return  m[s1.length, s2.length]
```

Algoritma ini memiliki kompleksitas waktu *O(mn)*, dengan m dan n adalah panjang masing-masing *string* yang diperbandingkan. Dengan kata lain, jika kedua *string* memiliki panjang yang sama kompleksitas waktunya adalah $O(n^2)$. Kompleksitas ruang untuk algoritma ini adalah juga *O(mn)* atau $O(n^2)$ jika kedua *string* memiliki panjang sama. Namun, jika kita melihat bagian implementasi rekurens pada algoritma ini, sebenarnya suatu baris matriks hanya membutuhkan data dari baris sebelumnya saja, berarti hanya dibutuhkan $2 \times n$ ruang memori untuk menyimpannya. Oleh karena itu, dapat dibuat lagi perbaikan algoritma sehingga kompleksitas ruangnya menjadi *O(n)*. Algoritma perbaikannya adalah:

```
function levDis (s1 : string, s2 : string) : integer
kamus
  i, cost : integer
  mBefore : array [0 .. s1.length] of integer
  mCurrent : array [0 .. s1.length] of integer
algoritma
  for i ← 0 to s1.length do
    {inisialisasi baris awal dengan nilai jarak
    perbandingan dengan kosong}
    mBefore[i] ← i

  for i ← 0 to s1.length do
    for j ← 1 to s2.length do
      if i = 0 then  {perbandingan dgn kosong}
        mCurrent[i] ← j
      else
        if s1[i] = s2[j] then
          cost ← 0
        else
          cost ← 1

        mCurrent[i] = minimum (
              mBefore[i-1] + cost, {substitusi}
              mCurrent[i-1] + 1, {penghapusan}
              mBefore[i] + 1, {penambahan}
          )

    mBefore ← mCurrent

  return  mCurrent[s1.length]
```

Perlu diketahui bahwa pada algoritma ini, tipe *string* dianggap memiliki properti *length* yang menyatakan jumlah karakter di dalam *string* tersebut dan dianggap bahwa *string* terdiri dari *array of characters* yang berindeks mulai dari 1 sampai jumlah karakter.

5. Pencarian String untuk mengoreksi kesalahan

Pencarian string dalam kamus dapat dilakukan dengan beberapa jenis algoritma. Diantaranya adalah algoritma *Brute Force*. Pencarian string dilakukan berdasarkan karakter-karakter yang bersesuaian pada kata yang akan dikoreksi.

Pencarian akan dilakukan kata perkata, sehingga bila terdapat ketidakcocokan pada awal kata, maka pencarian akan langsung berlanjut pada kata berikutnya.

Dalam proses pencarian, setiap kata yang ditemukan di dalam kamus akan dibandingkan dengan kata yang sedang diperiksa untuk mendapatkan jaraknya (Levenshtein distance). Setelah pencarian selesai, akan dicek lagi setiap jarak yang didapat sehingga dapat ditemukan kata-kata koreksinya yang mendekati kata yang sedang diperiksa untuk ditampilkan.

Contoh:

Kata yang sedang diperiksa:
Maragu

Kata dalam kamus:
a
abu
.
.
.
makan
malam
marah
marak
mata
.
.
.
zaman
zebra
.
.
.

Pencarian dilakukan secara *brute force*. Misalnya *pattern* yang dicari adalah "maragu" maka pada perbandingan terhadap kata "makan" akan berhenti pada karakter ke-3, sedangkan terhadap kata "marah" akan berhenti pada karakter ke-5.

Pada setiap perbandingan terhadap kata-kata di dalam kamus, juga dicari Levenshtein distance-nya yang akan digunakan untuk menemukan kata yang kemiripannya terbesar terhadap kata yang sedang diperiksa.

Setelah proses ini selesai, maka akan didaftar kata-kata yang paling mirip berdasarkan pencarian dan Levenshtein distance-nya untuk ditampilkan saat pengetikan.

6. Kesimpulan

Seiring dengan berkembangnya teknologi, pemeriksaan terhadap kesalahan pengejaan pada editor teks tidak lagi dilakukan secara manual melainkan dengan menggunakan sebuah algoritma, yaitu algoritma Levenshtein Distance yang menerapkan program dinamis untuk menghitung kemiripan sebuah string. Setelah tingkat kemiripan sebuah string diperoleh, secara brute force dicari string pada kamus yang tingkat kemiripannya tinggi dengan string tersebut. Komplesitas waktu algoritma pencarian kemiripan string adalah O(mn) dan komplesitas ruang algoritma tersebut adalah O(n).

7. Daftar Pustaka

1. Allison, L. 1999. *Dynamic Programming Algorithm for Edit Distance*, **http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/**, diakses tanggal 18 Mei 2006 pukul 18.00 WIB.
2. Wikipedia. 1999. *Levenshtein Distance*, **http://en.wikipedia.org/wiki/Edit_distance**, diakses tanggal 18 Mei 2006 pukul 17.30 WIB.
3. Trevisan, L. 2001. *Notes for Lecture 13 – Edit Distance*, **http://www.cs.berkeley.edu/~luca/cs170/notes/lecture13.pdf**, diakses tanggal 18 Mei 2006 pukul 17.25 WIB.