ALGORITMA LEVENSHTEIN DALAM PENDEKATAN *APPROXIMATE STRING MATCHING*

Bernardino Madaharsa Dito Adiwidya

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jalan Ganesha 10, Bandung, Jawa Barat
e-mail: if17089@students.if.itb.ac.id

ABSTRAK

Algoritma *Levenshtein* merupakan algoritma yang digunakan untuk mencari jumlah operasi *string* yang paling sedikit untuk mentransformasikan suatu *string* menjadi *string* yang lain. Algoritma ini digunakan dalam pencarian *string* dengan pendekatan perkiraan (*Approximate String Matching*). Dengan pendekatan perkiraan ini, pencarian *string* target menjadi tidak harus sama persis dengan yang ada di dalam *string* sumber. Tentu saja pendekatan tersebut menjadi bahan analisis untuk pengaplikasiannya dalam aplikasi yang nyata, seperti untuk mesin pencari, pengecek ejaan, pendeteksian suatu rantai dalam DNA, pendeteksi pemalsuan, dan lain-lain.
Ada tiga buah operasi *string* yang digunakan yaitu operasi penghapusan, penyisipan, dan penggantian. Jumlah operasi minimal dari ketiga operasi tersebut terhadap dua buah *string* disebut dengan *Levenshtein distance* atau *edit distance*. Dengan algoritma ini, tidak perlu dilakukan seluruh percobaan kemungkinan penghapusan, penyisipan, atau penggantian terhadap *string*. Oleh karena itu, algoritma ini dapat digolongkan sebagai program dinamis. Nilai *edit distance* dapat diperoleh hanya dengan menyediakan matriks *cost* yang ukurannya tergantung panjang kedua *string*.

Kata kunci: Levenshtein, edit distance, approximate string matching, cost, operasi string.

1. PENDAHULUAN

Pada saat ini terdapat banyak mesin pencari atau *search engine* yang tersedia di internet. Mesin pencari tersebut sangat berguna jika kita ingin mencari suatu kata kunci atau *keyword* yang terletak dalam suatu halaman *web* yang berisi berita, artikel, atau informasi lainnya. Seringkali, dalam suatu sistem ketatabahasaan, kata kunci yang kita cari bukan merupakan ejaan yang baku, memiliki suatu imbuhan yang komponen katanya berbeda, ataupun salah ketik. Sebagai contoh, kita mengetikkan "hirarki", padahal ejaan bakunya adalah "hierarki". Tentu saja kita tidak ingin gagal memperoleh informasi yang kita maksudkan ataupun memperoleh informasi yang kurang lengkap. Selain itu, belum tentu suatu halaman *web* menggunakan ejaan yang benar pula. Selain itu, suatu kata juga dapat berubah komponen katanya akibat penambahan imbuhan, seperti dalam bahasa Inggris "rate" dan "rating". Tentu saja jika kita bermaksud untuk memperoleh informasi yang mengandung kedua kata meskipun kita hanya mengetikkan "rate". Bahkan, kita dapat saja salah mengetikkannya, seperti "rtae". Oleh karena itu, diperlukanlah suatu metode pendekatan pencarian string yang dapat memenuhi keinginan tersebut.

Untuk melakukan pencarian kemungkinan kata yang diinginkan tersebut, diperlukanlah suatu pendekatan pencarian *string* khusus. Dalam pencarian *string* biasa, yaitu pencarian yang eksak, terdapat berbagai algoritma yang sangat dikenal seperti *Knuth-Morris-Pratt*, *Boyer-Moore*, *Rabin-Karp*, dan lain-lain. Sedangkan pencarian *string* khusus itu adalah yaitu dengan pendekatan perkiraan (*Approximate String Matching*).

Dalam pendekatan tersebut, ada tiga macam operasi yang digunakan untuk mentransformasikan suatu *string* menjadi *string* yang lain. Operasi tersebut antara lain operasi penghapusan, penyisipan, dan penggantian. Operasi-operasi ini digunakan untuk menghitung jumlah perbedaan yang diperlukan untuk pertimbangan kecocokan suatu *string* dengan *string* sumber. Jumlah perbedaan tersebut diperoleh dari penjumlahan semua pengubahan yang terjadi dari masing-masing operasi. Penggunaan perbedaan tersebut diaplikasikan dalam berbagai macam algoritma *Hamming*, *Levenshtein*, *Damerau-Levenshtein*, *Jaro-Winkler*, *Wagner-Fischer*, dan lain-lain. Dalam makalah ini, dibahas algoritma *Levenshtein* sebagai cara pencarian perbedaan *string* tersebut.

Algoritma *Levenshtein*, atau sering disebut dengan *Levenshtein Distance* atau *Edit Distance* merupakan algoritma pencarian jumlah perbedaan *string* yang ditemukan oleh Vladimir Levenshtein, seorang ilmuwan Rusia, pada tahun 1965. Algoritma ini digunakan secara luas dalam berbagai bidang, misalnya mesin pencari,

pengecek ejaan (*spell checking*), pengenal pembicaraan (*speech recognition*), pengucapan dialek, analisis DNA, pendeteksi pemalsuan, dan lain-lain.

2. APPROXIMATE STRING MATCHING

Dalam metode ini, dilakukan suatu penghitungan perbedaan antara dua *string*. Penghitungan tersebut meliputi tiga operasi *string* seperti di bawah ini. Untuk contoh yang akan digunakan, diasumsikan *S* adalah *string* sumber pencarian dan *T* adalah *string* yang ingin dicari.

2.1 Operasi Penghapusan

Misalnya *S* = *memori* dan *T* = *meri*. Penghapusan dilakukan untuk karakter *m* pada lokasi ke-3 dan *o* pada lokasi ke-4. Dua operasi penghapusan tersebut menunjukkan transformasi *S* ke *T* yang diilustrasikan sebagai berikut.

	1	2	3	4	5	6
T =	m	e	–	–	r	i
S =	m	e	m	o	r	i

2.2 Operasi Penyisipan

Misalnya *S* = *brian* dan *T* = *barisan*. Operasi sisip dilakukan dengan menyisipkan *a* dan *s* pada posisi 2 dan 5 yang dapat ditunjukkan sebagai berikut.

	1	2	3	4	5	6	7
T =	b	a	r	i	s	a	n
S =	b	–	r	i	–	a	n
		a			s		

2.3 Operasi Penggantian

Misalnya *S* = *perasa* dan *T* = *pewara*. String *T* ditransformasikan menjadi *S* dengan melakukan penggantian (subtitusi) pada posisi ke-3 dan ke-5. Huruf *r* dan *s* pada *S* digantikan dengan *w* dan *r* pada *T*. Prosesnya dapat ditunjukkan sebagai berikut.

	1	2	3	4	5	6
T =	p	e	w	a	r	a
S =	p	e	r	a	s	a
			w		r	

Ketiga operasi di atas dapat digunakan dalam contoh berikut ini.

Misalnya *T* = *abrakadabra* dan *S* = *avrakhadabah*. *S* dapat ditransformasikan menjadi *T* sebagai berikut.

	1	2	3	4	5	6	7	8	9	10	11	12	13
T =	a	b	r	a	k	–	a	d	a	b	r	a	–
S =	a	v	r	a	k	h	a	d	a	b	–	a	h
		b									r		

Operasi yang dilakukan yaitu:

1. mengganti *v* pada posisi 2
2. menghapus *h* pada posisi 6
3. menyisipkan *r* pada posisi 11
4. menghapus *h* pada posisi 13

Secara keseluruhan terdapat 4 operasi, yaitu 1 penggantian, 2 penghapusan, dan 1 penyisipan.

3. ALGORITMA LEVENSHTEIN

Algoritma ini menghitung jumlah operasi *string* paling sedikit yang diperlukan untuk mentransformasikan suatu *string* menjadi *string* yang lain. Berdasarkan contoh di atas, nilai *edit distance* yang yang diperoleh adalah 4. Berikut ini akan dijelaskan langkah-langkah algoritma ini.

3.1 Langkah-langkah Algoritma

Pada dasarnya, algoritma ini menghitung jumlah minimum pentransformasian suatu *string* menjadi *string* lain yang meliputi penggantian, penghapusan, dan penyisipan. Algoritma ini digunakan untuk mengoptimalkan pencarian tersebut karena sangat tidak efisien jika dilakukan pencarian setiap kombinasi operasi-operasi *string* tersebut. Oleh karena itu, algoritma ini tergolong program dinamis dalam pencarian nilai minimal tersebut.

Dalam algoritma ini, dilakukan penyeleksian panjang kedua *string* terlebih dahulu. Jika salah satu atau kedua *string* merupakan *string* kosong, jalannya algoritma ini berhenti dan memberikan hasil *edit distance* yang bernilai nol atau panjang *string* yang tidak kosong. Jika panjang *string* keduanya tidak nol, setiap *string* memiliki sebuah karakter terakhir, misalnya c1 dan c2. Misalnya bagian *string* pertama tanpa c1 adalah s1 dan bagian *string* kedua tanpa c2 adalah s2, dapat dikatakan penghitungan yang dilakukan adalah cara mentransformasikan s1+c1 menjadi s2+c2. Jika c1 sama dengan c2, dapat diberikan nilai *cost* 0 dan nilai *edit distance*-nya adalah nilai *edit distance* dari pentransformasian s1 menjadi s2. Jika c1 berbeda dengan c2, dibutuhkan pengubahan c1 menjadi c2 sehingga nilai *cost*-nya 1. Akibanya, nilai *edit distance*-nya adalah nilai *edit distance* dari pentransformasian s1 menjadi s2 ditambah 1. Kemungkinan lain adalah dengan menghapus c1 dan mengedit s1 menjadi s2+c2 sehingga nilai *edit distance*-nya dari pentransformasian s1 menjadi s2+c2 ditambah 1. Begitu pula dengan penghapusan c2 dan mengedit s1+c1 menjadi s2. Dari kemungkinan-kemungkinan tersebut, dicarilah nilai minimal sebagai nilai *edit distance*.

Untuk lebih jelasnya, dapat dilihat pada *pseudocode* berikut ini. Di bawah ini digunakan *s* sebagai *string* sumber dan *t* sebagai *string* target. *Pseudocode* ini berupa

suatu fungsi yang nilai kembaliannya adalah nilai *edit distance*.

```
n ← length(s)
m ← length(t)
if n = 0 then return m
else if m = 0 then return n
else
  for i = 0 to n do
    cost[0][i] ← i
  for i = 0 to m do
    cost[i][0] ← i
  for i = 1 to n do
    for j = 1 to m do
      if (s[i-1]=t[j-1]) then
        cost[j][i] ← 0
      else cost[j][i] ← 1
      a1 ← cost[j][i-1]+1
      a2 ← cost[j-1][i]+1
      a3 ← cost[j-1][i-1]+cost[j][i]
      cost[j][i] ← min(t1,t2,t3)
  return cost[m][n]
```

Berdasarkan *pseudocode* di atas, dapat ditunjukkan kompleksitas algoritmanya adalah O(|s|*|t|) atau dapat dianggap $O(n^2)$.

3.2 Contoh Kasus

Misalkan *s* = *malamram* dan *t* = *aram*. Di bawah ini merupakan sebagian dari operasi transformasi *s* ke *t* dengan berpatokan pada kesamaan *string* berurutan "am" pada keduanya.

1. `s: m a l a m r a m`
   `t: - a r a m - - -`
   String s ditransformasikan menjadi *t* dengan melakukan 4 penghapusan dan 1 penggantian. Jumlah perubahannya adalah 5.
2. `s: m a l a m r a m`
   `t: - - - - a r a m`
   String s ditransformasikan menjadi *t* dengan melakukan 4 penghapusan dan 1 penggantian. Jumlah perubahannya adalah 5.
3. `s: m a l a m r a m`
   `t: - a - - - r a m`
   String s ditransformasikan menjadi *t* dengan melakukan 4 penghapusan. Jumlah perubahannya adalah 4.
4. `s: m a l a m r a m`
   `t: - - - a - r a m`
   String s ditransformasikan menjadi *t* dengan melakukan 4 penghapusan. Jumlah perubahannya adalah 4.

Algoritma di atas dapat menentukan jumlah perubahan yang diperlukan yang minimal, yaitu 4. Pemerolehan nilai *edit distance* ini dapat ditunjukkan sebagai berikut

Tabel 1 Analisis *cost* untuk *edit distance*

	0	1	2	3	4	5	6	7	8	9
0			m	A	l	a	m	r	a	m
1		0	1	2	3	4	5	6	7	8
2	a	1	1	1	2	3	4	5	6	7
3	r	2	2	2	2	3	4	4	5	6
4	a	3	3	2	3	2	3	4	4	5
5	m	4	3	3	3	3	2	3	4	4

Jika dirunut dari angka di pojok kanan bawah (yang menunjukkan nilai *edit distance*, berlatar belakang merah), dapat dibuktikan proses pembandingan yang digunakan adalah pada nomor (3) atau (4) dari contoh di atas. Sebenarnya, hanya ada satu kemungkinan cara pembandingan. Pada contoh di atas, misalnya dimulai pada posisi *edit distance* yakni pada posisi (5,9), ditemukan percabangan pada posisi (2,6), yaitu menuju (2,5) atau (1,5). Posisi (2,5) menunjukkan variabel `a1` dari *pseudocode* di atas, sedangkan (1,5) menunjukkan variabel `a3`. Untuk memperoleh 4, dapat dilakukan penghitungan 3+1 atau 4+0. Penentuan posisi yang minimal ini tergantung pemilihan yang ada dalam algoritma, yang pada *pseudocode* di atas berada pada fungsi `min()`.

3.3 Penggunaan Algoritma

Hasil *edit distance* yang diperoleh sebenarnya tidak dapat langsung dimanfaatkan dalam suatu aplikasi nyata, melainkan perlu ditambahkan atau dimodifikasi tergatung kebutuhan aplikasi tersebut. Tentu saja banyak aplikasi yang menggunakan algoritma *Levenshtein* ini. Algoritma ini digunakan dalam biologi untuk menemukan barisan yang sama pada asam nukleat dalam DNA atau asam amino dalam protein. Selain itu, algoritma ini banyak digunakan dalam bidang informatika seperti untuk pengecek ejaan, perkiraan dari pengucapan dialek, pemandu penterjemahan, mesin pencari, pemberi revisi *file* dengan membandingkan perbedaan dua buah *file*, pendeteksi pemalsuan, pengenal percakapan (*speech recognition*), dan sebagainya.

Seperti yang telah disebutkan sebelumnya, pada algoritma ini dapat dilakukan modifikasi atau pengembangan. Contohnya untuk mencari *linear-gap-costs* yang memodifikasi nilai *cost* pada *edit distance*. Selain itu, ada algoritma yang cukup terkenal yaitu *Longest Common Subsequence* (LCS) yang mencari upabarisan terpanjang dari dua buah barisan atau *string*. Algoritma lain juga dikembangkan untuk lebih mengoptimalkan algoritma ini seperti algoritma yang ditemukan oleh Ukkonen. Algoritma tersebut memiliki kompleksitas terburuk O(n*d) dan kompleksitas rata-rata

$O(n+d^2)$ dengan *n* adalah panjang *string* dan *d* adalah *edit distance*. Algoritma ini sangat mangkus jika nilai *d* sangat kecil.

4. KESIMPULAN

Pencarian jumlah operasi *string* dalam *approximate string matching* yang diperlukan untuk mentransformasikan suatu *string* menjadi *string* lain memerlukan algorima yang mangkus seperti algoritma *Levenshtein*. Algoritma ini merupakan program dinamis dengan kompleksitas $O(n^2)$.

REFERENSI

[1] http://algdoc.cs.nthu.edu.tw/webs@2/cyberhood/StringMatching/Chapter%201(String%20Matching).doc
Waktu akses: 27 Desember 2009 pukul 22.00 WIB

[2] http://levenshtein.net
Waktu akses: 28 Desember 2009 pukul 10.47 WIB

[3] http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/
Waktu akses: 28 Desember 2009 pukul 20.37 WIB

[4] http://www.let.rug.nl/~kleiweg/lev/levenshtein.html
Waktu akses: 28 Desember 2009 pukul 20.39WIB

[5] http://www.merriampark.com/ld.htm
Waktu akses: 28 Desember 2009 pukul 20.38 WIB