Penerapan Algoritma *Dynamic Programming* untuk Mencari Solusi Masalah *String Edit Distance*

Muhammad Ghifary

Sekolah Tinggi Elektro dan Informatika
Program Studi Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
e-mail: m_ghifary@students.itb.ac.id, if15023@students.if.itb.ac.id

ABSTRAK

Pada saat terjadi interaksi antara pengguna(*user*) dengan komputer yang berhubungan dengan manipulasi tekstual/literal, pengguna sering kali melakukan kesalahan pengetikan. Untuk meningkatkan kinerja para pengguna tersebut dalam melakukan perbaikan penulisan teks, komputer dapat diprogram agar mampu memberikan umpan balik (*feedback*) bahwa *user* telah melakukan kesalahan pengetikkan. Umpan balik tersebut dapat berupa pemberian alternatif teks yang benar atau langsung mengubah teks yang salah tersebut, seperti yang dilakukan oleh fitur *auto correction* dan fitur *Spelling and Check Grammar* pada aplikasi-aplikasi *word processor* ataupun pada fitur pemberian alternatif teks yang dicari pada *Web Search Engine*. Pada literatur-literatur, permasalahan ini dinamakan *String Edit Distance*. Salah satu cara untuk memecahkan masalah ini adalah dengan menerapkan algoritma *Dynamic Programming* untuk menghitung *score* dan menetapkan *string* yang benar yang paling mendekati (*closest match*) atau mirip dengan *string* yang dimasukkan oleh *user*. Pengembangan teknik ini banyak diterapkan pada bidang disiplin ilmu Bioinformatika untuk mengecek kemiripian rumus DNA atau protein (*sequence aligment*). Pemecahan masalah *string edit distance* menjadi titik temu antara bidang informatika dengan biologi. Pada makalah ini akan dibahas secara singkat bagaimana langkah-langkah teknis algoritma *Dynamic Programming* dalam memecahkan masalah ini, serta bagaimana kompleksitas ruang dan waktu dari langkah-langkah tersebut.

Kata kunci: *Dynamic Programming*, *String Edit Distance,* program dinamis, *sequence alignment, text processing,* optimalitas

1. PENDAHULUAN

Saat ini, fasilitas perbaikan *string* merupakan hal perlu dimiliki oleh sistem komputer untuk mempermudah dan memperbaiki kinerja para pengguna dalam memanipulasi teks-teks dengan komputer. Komputer dapat memberikan *feedback* kepada pengguna apabila terjadi kesalahan pengetikan teks. Berikut ini diberikan contoh-contoh kasus dimana fasilitas tersebut dipakai.

1. Para pengguna aplikasi *word processor* seperti *Ms-Word*, apabila mengaktifkan fasilitas *auto correction* pada saat melakukan pemrosesan teks, maka pada waktu mengetikkan *string* yang salah seperti 'juni', program akan langsung mengoreksinya menjadi, misalnya, 'june'. Begitu juga dengan fasilitas *Spelling and Check Grammar*. *String* yang secara tata tulis dianggap salah oleh program akan diberikan garis bawah.
2. Fasilitas *Search Engine* pada *Web* seperti *Google* memiliki fitur untuk memberikan *query* alternatif kepada pengguna yang memasukkan *query* yang salah. Misalnya, pada waktu pengguna mengetikkan "DinamicProgramming", maka *Google* akan memberikan *feedback* berupa pertanyaan "*Did you mean* ***Dynamic Programming*** *?*".

Pada contoh tersebut, *string* 'june' memiliki kekerabatan paling dekat (*closest match*) dengan 'juni', begitu juga *string* 'Dynamic Programming' dengan 'DinamicProgramming'. Salah satu cara untuk menghitung kekerabatan paling dekat dari kedua *string* tersebut adalah menggunakan algoritma *Dynamic Programming*. Dalam memecahkan kasus pemrosesan teks tersebut, algoritma *Dynamic Programming* memiliki kinerja yang lebih baik dibandingkan dengan metode *exhaustive search* dan *greedy*.

2. *STRING EDIT DISTANCE*

Definisi dari *edit distance* dari 2 buah string s1 dan s2 adalah jumlah minimum dari mutasi yang perlu dilakukan dari s1 menjadi s2, dimana mutasi tersebut dapat berupa :

1. Penggantian karakter (*substitution*). Contoh : ‘dinamic’ -> ‘dynamic’)
2. Pemasukkan karakter (*insertion*). Contoh : ‘maam’ -> ‘malam’
3. Penghapusan karakter (*deletion*). Contoh : ‘rumahh’ -> ‘rumah’

Edit distance dari s1 dan s2 dapat kita nyatakan dengan symbol D(s1, s2). Semakin kecil nilai dari *edit distance* antara s1 dan s2 berarti semakin dekat kekerabatan antara s1 dan s2.

Terdapat 2 buah variabel yang kita definisikan untuk mensimbolkan perbedaan dari s1 dan s2 yaitu

- δ : *gap penalty* (bernilai *k* : jika ada karakter ‘-‘)
- $p(i, j)$ *: mismatch penalty*(bernilai *m* : jika terjadi ketidakcocokan antara karakter ke-i di s1 dan karakter ke-j di s2)

(lihat referensi [6])

Instansiasi dari penggunaan *edit distance* dapat dideskripsikan sebagai berkut.

Contoh : D(parisipadi, partisipasi)

String parisipadi akan di-*align* sehingga *fit* dengan partisipasi

```
p a r – i s i p a d i
| | | | | | | | | |
p a r t i s i p a s i
| | | | | | | | | | |
c c c i c c c c c s c   → Keterangan :
                          c : copy
                          i : insert
                          s : substitute
```

Karakter yang sama hanya di-*copy*, sedangkan pada karakater *gap* (‘-’) akan dilakukan *insertion* dan karakter yang berbeda dilakukan *substitution* (‘d’ dengan ‘s’).
Misalnya, kita tentukan nilai untuk δ *=1* jika ada karakter gap dan δ *=0* jika tidak ada, serta $p(i, j) = 1$ jika ada perbedaan karakter dan $p(i, j) = 0$ jika tidak ada, maka

: : D(parisipadi, partisipasi) $= \delta + p(i, j) = 1 + 1 = 2$

3. ALGORITMA *DYNAMIC PROGRAMMING*

Dynamic Programming merupakan sebuah algoritma pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan langkah (*step*) atau tahapan (*stage*) sedemikian sehingga solusi dai persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan. Pada penyelesaian metode in kita menggunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Algoritma *Dynamic Programming* memiliki karakteristik sebagai berikut.

1. Persoalan dapat dibagi mejadi beberapa tahap(*stage*), yang pada setiap tahap hanya diambil satu keputusan yang optimal.
2. Masing-masing tahap terdiri dari sejumlah status(*state*) yang berhubungan dengan tahap tersebut.
3. Hasil keputusan yang diambil pada tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
4. Ongkos (*cost*) pada suatu tahap bergantung pada ongkos tahap-tahap sebelumnya dan meningkat secara teratur dengan bertambahnya jumlah tahapan
5. Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan tahap sebelumnya.
6. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap *k* memberikan keputusan terbaik untuk tahap sebelumnya.
7. Prinsip optimalitas berlaku pada persoalan tersebut.

Ciri utama dari *dynamic programming* adalah prinsip optimalitas yang berbunyi : *jika solusi total optimal, maka bagian solusi sampai tahap ke-k juga optimal*.

Dari karakteristik poin ke-4 di atas, kita dapat menyimpulkan bahwa algoritma *Dynamic Programming* dapat diaplikasikan apabila peningkatkan ongkos secara linear dan diskrit sehingga optimasi parsial dapat dilakukan.

Dalam menyelesaikan persoalan dengan *Dynamic Programming*, kita dapat menggunakan 2 pendekatan berbeda yaitu

a. Maju (*forward* atau *up-down*) : bergerak mulai dari tahap 1, terus maju ke tahap 2,3,..,n. Urutan variabel keputusan adalah $x1,x2,...,x_n$
b. Mundur(*backward* atau *bottom-up*) : bergerak mulai dari tahap n, terus mundur ke tahap n-1, n-2,..,2,1. Urutan variabel keputusan adalah $x_n\ x_{n-1}\ ,x2,x1$.

Secara umum ada 4 langkah yang dilakukan dalam mengembangkan algoritma program dinamis:

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.

Untuk informasi selengkapnya tentang *Dynamic Programming* dapat dilihat pada referensi [1] dan [3].

2. METODE PERHITUNGAN *EDIT DISTANCE*

2.1. *Exhaustive Search* dan *Greedy*

Jika permasalahan *string edit distance* untuk 2 *string* yang masing-masing panjangnya sebesar m dan n ini dihitung dengan menggunakan *exhaustive search*, maka akan menghasilkan kompleksitas sebesar $O(3^{m+n})$ – lihat referensi [3] - . Sedangkan, jika permasalahan ini diselesaikan dengan metody *greedy*, maka solusi optimal global belum tentu dapat dicapai(tingkat keakuratannya rendah). Pada kesempatan ini kita tidak membahas penggunaan metode *exhaustive search* ataupun *greedy*, dan hanya membahas penggunaan algoritma *Dynamic Programming*.

2.2. *Dynamic Programming*

Pada permasalahan *string edit distance ini,* nilai *edit distance* D menjadi ongkos (*cost*) yang harus dicari solusi optimum parsial maupun globalnya. Misalnya, diketahui 2 buah yaitu *string W* dan *string V,* di mana $W = W1..Wi$ merupakan *string* yang akan diperbaiki, dan $V = V1..Vj$ merupakan *string* yang akan dihitung nilai *edit distance*-nya terhadap *string W,* maka persamaan umum rekursif dari *edit distance* dengan menggunakan algoritma *Dynamic Programming* prinsip pendekatan maju (*forward* atau *up-down*) dapat kita bentuk sebagai berikut.

$$D(i,j) = \begin{cases} basis: D(0,0) = 0 \\ rekurens: \min \begin{cases} D(i-1,j-1) + P(i,j) .....(1) \\ D(i-1,j) + \delta(i,j) + P(i,j) .....(2) \\ D(i,j-1) + \delta(i,j) + P(i,j) .....(3) \end{cases} \end{cases}$$

Keterangan :

- $P(i,j)$:*mismatch penalty*
- $\delta(i,j)$:*gap penalty*

Untuk persoalan ini, kita tentukan nilai *mismatch penalty* $P(i,j) = 0$,jika $Wi = Vj$ dan $P(i,j) = 1$, jika $Wi \neq Vj$, serta nilai *gap penalty* $\delta(i,j) = 1$,jika ada karakter '-', dan $\delta(i,j) = 0$,jika tidak ada.

Pada bagian rekurens, persamaan (1) menunjukkan bahwa perlu dilakukan penggantian (*substitution*) jika $Wi \neq Vj$ agar karakter $Wi = Vj$. Sedangkan, persamaan (2) menunjukkan bahwa perlu dilakukan pemasukkan karakter (*insertion*) serta persamaan (3) menunjukkan bahwa perlu dilakukan penghapusan karakter (*deletion*).

Untuk menyimpan nilai-nilai *edit distance* dari setiap langkahnya, maka kita bentuk suatu matriks M berdimensi m x n dimana m=[panjang(*V*) +1] dan n=[panjang(*W*) + 1] dimana M[0,0] merupakan tempat untuk menyimpan nilai basis yaitu 0.

Tabel 1 Matriks Nilai *Edit Distance*

		V 1	*V 2*	.		.	*Vm-1*	*Vm*
	0							
W1			insert					
W2		del	subs					
.								
.								
.								
Wn-1								
Wn								

Sebagai contoh persoalan di atas, kita tentukan *W* = 'PARL', dan *V* = 'PARK'. Selanjutnya kita bentuk matriks *M[i,j]* dimana *M[i,j] = D(i,j)*, dalam hal ini, *i = j*, dan $\delta(i,j) = 0$ karena $panjang(V) = panjang(W)$. Langkah-langkah penentuan nilai *edit distance*-nya adalah sebagai berikut.

Langkah ke-1 :mengisi baris ke-0

$D(0,0) = 0, basis$

$D(0,1) = \min(D(0,0) + 1) = 1, \{W0 \neq V1\}$

$D(0,2) = \min(D(0,1) + 1) = 2, \{W0 \neq V2\}$

$D(0,3) = \min(D(0,2) + 1) = 3, \{W0 \neq V3\}$

$D(0,4) = \min(D(0,3) + 1) = 4, \{W0 \neq V4\}$

Tabel 2 Langkah Penentuan Nilai Matriks M[0,j]

		P	*A*	*R*	*K*
	0 →	*1* →	*2* →	*3* →	*4*
P					
A					
R					
L					

Langkah ke-2 :mengisi baris ke-1

$D(1,0) = \min(D(0,0) + 1) = 1, \{W1 \neq V0\}$
$D(1,1) = \min(D(0,0), D(1,0), D(0,1)) = \min(0,1,1) = 0, \{W1 = V1\}$
$D(1,2) = \min(D(0,1) + 1, D(0,2) + 1, D(1,1) + 1) = \min(2,3,1) = 1, \{W1 \neq V2$
$D(1,3) = \min(D(0,2) + 1, D(0,3) + 1, D(1,2) + 1) = \min(3,4,2) = 2, \{W1 \neq V$
$D(1,4) = \min(D(0,3) + 1, D(0,4) + 1, D(1,3) + 1) = \min(4,5,3) = 3, \{W1 \neq V$

Tabel 3 Langkah Penentuan Nilai Matriks M[1,j]

		P	A	R	K
	0	1	2	3	**4**
P	1 →	0 →	1 →	2 →	**3**
A					
R					
L					

Jika langkah-langkah tersebut dilanjutkan dengan cara yang sama hingga langkah terakhir yaitu langkah ke-5, maka

Langkah ke-5 :mengisi baris ke-4

$D(4,0) = \min(D(3,0) + 1) = \min(4) = 4, W4 \neq V0$
$D(4,1) = \min(D(3,0) + 1, D(3,1) + 1, D(4,0) + 1) = \min(4,3,5) = 3, W4 \neq V1$
$D(4,2) = \min(D(3,1) + 1, D(3,2) + 1, D(4,1) + 1) = \min(3,2,4) = 2, W4 \neq V2$
$D(4,3) = \min(D(3,2) + 1, D(3,3) + 1, D(4,2) + 1) = \min(2,1,3) = 1, W4 \neq V3$
$D(4,4) = \min(D(3,3) + 1, D(3,4) + 1, D(4,3) + 1) = \min(1,2,2) = 1, W4 \neq V4$

Tabel 4 Langkah Penentuan Nilai Matriks M[4,j]

		P	A	R	K
	0	1	2	3	**4**
P	1	0	1	2	**3**
A	2	1	0	1	**2**
R	3	2	1	0	**1**
L	**4** →	**3** →	**2** →	**1** →	**1**

Selanjutnya, kita akan melakukan konstruksi solusi optimal yang dibaca nilai *D(4,4)*. Maka, solusi-solusi optimal parsial yang dipilih untuk membentuk solusi optimal global adalah sebagai berikut.

Tabel 5 Konstruksi Solusi Optimal

		P	A	R	K
	0	1	2	3	**4**
P	1	0	1	2	**3**
A	2	1	0	1	**2**
R	3	2	1	0	**1**
L	**4**	**3**	**2**	**1**	**1(D(4,4))**

Dengan demikian, *edit distance* dari *V*='PARK' terhadap *W*='PARL' bernilai *D(i,j) = 1*.

Penggunaan algoritma *Dynamic Programming* untuk kasus ini, jika dilihat dari langkah-langkah yang sudah dibentuk di atas, langkah-langkah tersebut berupa traversal matriks M yang berdimensi m x n. Sedangkan, nilai *edit distance* yang ditentukan dilakukan dengan sendirinya akibat sifat rekursifitas persamaan *edit distance* yang dibentuk. Oleh karena itu, kompleksitas dari penggunaan algoritma ini sebesar *O(mn)*, lebih baik dari metode penyelesaian masalah dengan *exhaustive search*.

Untuk melihat perbedaan nilai *edit distance* jika *string* 'PARL' dibandingkan dengan string yang lain, sekarang kita definisikan *V*='SPEAK' dan *W*='PARL'. Dalam hal ini, $i \neq j$ dan ada $\delta(i,j) = 1$, karena $panjang(V) \neq panjang(W)$. Langkah-langkah penentuan nilai *edit distance* dan juga konstruksi solusi optimalnya tidak dijelaskan secara mendetail karena keterbatasan waktu dan tempat. Namun, kita dapat menentukan nilai *edit distance* dari *string V*='PEARL' terhadap W='PARL' menurut definisi yaitu sebagai berikut.

```
- P A R L
| | | | |
P E A R L
| | | | |
d s c c i
```

sehingga D(PARL,PEARL) = 3

Apabila teknik ini diimplementasikan pada aplikasi *word processor* ataupun *search engine*, maka program akan memberikan *feedback* berupa string *W* dengan nilai *D*('PARL', *W*) yang paling kecil, dimana *W* merupakan kumpulan *string* yang ada pada *database*.

Jadi, jika kita merujuk ke contoh di atas, program *word processor* akan mengoreksi *string* 'PARL' dengan 'PARK', bukan dengan 'PEARL'. Begitu juga pada *Search Engine* Google. Google, misalnya, akan memberikan *feedback* berupa pertanyaan "*Did you mean* ***PARK*** *?*". Ini dikarenakan nilai *D('PARL', 'PARK')<D('PARL','PEARL')*.

IV. KESIMPULAN

Kesimpulan yang bisa ditarik dari penggunaan algoritma *Dynamic Programming* untuk kasus *String Edit Distance* adalah bahwa algoritma ini bekerja cukup baik dengan kompleksitas yang lebih kecil dari metode *exhaustive search* dan tingkat akurasi pencarian solusi optimal yang lebih tinggi dari algoritma *greedy*. Namun, akibat penggunaan fungsi *cost* yang bersifat rekursif, maka akan menyebabkan penggunaan kapasitas yang cukup besar terhadap memori internal komputer saat

mengeksekusi fungsi ini. Selama tidak ada permasalahan yang berkenaan dengan kapasitas memori internal, algoritma *Dynamic Programming* merupakan pilihan yang paling sesuai.

Penerapan algoritma *Dynamic Programming* untuk mencari solusi masalah *String Edit Distance* banyak digunakan untuk memecakan berbagai kasus yang serupa, khususnya di bidang disiplin ilmu Bioinformatika. Beberapa perluasan dari metode tersebut diterapkan pada analisis sequens DNA atau protein. Sekuen DNA yang terdiri dari kombinasi basa ATCG, dapat dibandingkan dengan sekuen lain. Perbandingkan ini dikenal dengan istilah *sequence alignment*. Untuk tipe *local sequence alignment*, algoritma yang biasa digunakan adalah algoritma *Needleman-Wunsch*, dan untuk tipe *global sequence alignment*, digunakan algoritma *Smith-Waterman* dan BLAST (lihat referensi [2], [5] dan [7]).

REFERENSI

[1] Rinaldi Munir, "Diktat Kuliah IF2251 Strategi Algoritmik", Program Studi Informatika ITB, 2007.

[2] M.Reza Firdaus Zen, Sila Wiyanti P., M. Fajrin R., "Penerapan Algoritma Needleman-Wunsch sebagai Salah Satu Implementasi Program Dinamis pada Pensejajaran DNA dan Protein", Makalah tugas mata kuliah IF2251 Strategi Algoritmik, 2006, http://www.informatika.org/~rinaldi/Stmik/2005-2006/Makalah2006.htm/MakalahStmik2006-03.pdf, diakses tanggal 19 Mei 2007 pukul 19.30 .

[3] A. Zisserman, "Lecture 3", www.robots.ox.ac.uk/~az/lectures/opt/**lect3**.pdf, diakses tanggal 22 Mei 2007 pukul 05.30

[4] R. A. Nalistia, A. Widyanarko, A.Z. Sidiq, "*Dynamic Programming* untuk Optimasi Sponsorship Peserta Kompetisi Balap F1", Makalah tugas mata kuliah IF2251 Strategi Algoritmik, 2006, http://www.informatika.org/~rinaldi/Stmik/2005-2006/Makalah2006.htm/MakalahStmik2006-25.pdf, diakses tanggal 19 Mei 2007 pukul 19.33 .

[5] T.F. Smith, M.S. Waterman, "*Identification of Common Molecular Subsequences*", J. Mol. Biol., 147, 1981, 195-197, http://jaligner.sourceforge.net/references/smithandwaterman1981.pdf, diakses tanggal 22 Mei 2007 pukul 05.45

[6] Norbert Zeh, "CSci 3110:Dynamic Programming",http://users.cs.dal.ca/~nzeh/Teaching/Summer2007/3110/Notes/dp.pdf, diakses tanggal 22 Mei 2007 pukul 05.25.

[7] A. Wicaksono, "Aplikasi Pemrograman Dinamis dalam Bioinformatika",http://ilmukomputer.com/2006/08/19/aplikasi-pemrograman-dinamis-dalam-bioinformatika/, diakses tanggal 22 Mei 2007 pukul 05.33.