Penerapan Program Dinamis dengan Jarak Levenshtein untuk Mendeteksi Plagiarisme Karya Ilmiah dan Kode Program

Erick Wijaya 13515057[1]
Informatics Undergraduate Program
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganesha Street 10 Bandung 40132, Indonesia
[1]13515057@std.stei.itb.ac.id
wijaya.erick52@gmail.com

***Abstract*—Plagiarisme adalah tindakan penjiplakan karya orang lain tanpa mencantumkan referensi sumber. Plagiarisme adalah tindakan yang tidak etis dan dapat berbahaya untuk kasus tertentu seperti kode program. Makalah ini membahas tentang pendeteksian plagiarisme dengan menggunakan teknik Program Dinamis. Tahap pertama adalah menentukan Jarak Levenshtein dari dua sumber dengan menggunakan relasi rekurens dan Program Dinamis. Tahap berikutnya menentukan indeks kemiripan antara dua sumber. Tahap terakhir adalah menentukan apakah sebuah sumber berpotensi adalah plagiat dari sumber lain berdasarkan hasil indeks kemiripan yang dihitung.**

Keywords—program dinamis, relasi rekursens, plagiarisme, jarak levenshtein, indeks kemiripan

I. PENDAHULUAN

Komputer zaman sekarang memiliki kemampuan yang beraneka-ragam untuk menjalankan proses, melakukan perhitungan, dan mengeksekusi algoritma. Hampir semua orang menggunakan komputer untuk melakukan kegiatan produktif, seperti menulis makalah, membuat situs, mendesain berbagai hal, hingga menyusun program. Kemudahan yang diberikan komputer membuat orang-orang hampir tidak terlepas dari komputernya ketika bekerja. Akan tetapi, kemudahan yang disediakan komputer juga memiliki dampak negatif, yaitu kegiatan plagiarisme menjadi sangat mudah dilakukan. Hanya dengan mencari sumber, lalu pilih *copy/paste*, kita dapat mengambil seluruh konten pekerjaan orang lain.

Selama orang menulis referensi yang digunakan sebagai sumber, tindakan tersebut dapat diterima dengan baik. Akan tetapi, bila sumber tidak dicantumkan, orang tersebut melakukan plagiarisme. Plagiarisme jelaslah buruk karena plagiarisme merupakan bentuk pencurian kepemilikian intelektual, yaitu ide. Orang yang melakukan plagiarisme tidak menghargai usaha yang sudah dilakukan orang lain dan semena-mena menggunakan usaha orang lain tersebut untuk kepentingan dirinya sendiri. Plagiarisme sering ditemukan pada makalah dan karya-karya ilmiah. Namun, plagiarisme juga tak terbatas dari itu; plagiarisme dapat pula ditemukan pada desain dan kode program. Plagiarisme kode program berbahaya karena apabila kode yang ditiru masih memiliki *bug*, program plagiat juga akan memiliki *bug* serupa. Hal ini menyebabkan ketika kode program sumber sudah diperbaiki, kode program yang lama (memiliki *bug)* masih tersebar dan digunakan oleh orang lain.

Salah satu cara untuk mengatasi plagiarisme adalah dengan memeriksa dan membandingkan karya yang dibuat dengan karya-karya lainnya. Akan tetapi, pekerjaan ini membutuhkan usaha yang besar dan lama. Selain itu, plagiarisme umumnya dilakukan dengan mengubah kata-kata menjadi sinonimnya atau mengubah nama variable (pada program) sehingga ada kemungkinan pemeriksa tidak menyadari adanya plagiarisme. Solusi yang ditawarkan penulis pada makalah ini adalah membuat program/aplikasi pendeteksi plagiarisme. Program ini memanfaatkan teknik Program Dinamis untuk menyimpan tingkat kedekatan dari dua buah sumber. Makalah ini akan membahas tentang pendeteksian plagiarisme pada karya ilmiah dan kode program dengan teknik Program Dinamis.

II. DASAR TEORI

Sebelum penulis membahas mengenai cara mendeteksi plagiarisme dengan Program Dinamis, ada baiknya penulis memaparkan konsep Program Dinamis sehingga pembaca dapat memahami apa itu Program Dinamis dan bagaimana teknik tersebut dapat memecahkan permasalahan plagiarisme.

Setelah membahas konsep Program Dinamis (dari poin A sampai poin E), berikutnya akan dibahas lebih spesifik mengenai jarak levenshtein dan cara menghitungnya.

A. Definisi Program Dinamis

Program Dinamis (*Dynamic Programming*) adalah teknik pemecahan masalah dengan membagi solusi menjadi beberapa tahap komputasi, dengan ketentuan solusi pada setiap tahap dibangun dari solusi tahap sebelumnya. Ketika ingin memecahkan masalah, kita dapat melihat solusi yang sudah dihitung sebelumnya sehingga kita tidak perlu menghitung solusi yang sama apabila solusi tersebut sudah dihitung sebelumnya. Program Dinamis menggunakan solusi-solusi tahap sebelumnya untuk memecahkan masalah sehingga seringkali Program Dinamis diimplementasi secara rekursif.

B. Sejarah Penamaan Program Dinamis

Program Dinamis memiliki nama yang memiliki arti perencanaan (program) yang multitahap (dinamis). Akan tetapi, sebenarnya penamaan Program Dinamis memiliki sejarah yang unik. Pada tahun 1950an, Richard Bellman melakukan riset tentang Proses Keputusan Multitahap (*Multistage Decision Process*), akan tetapi tahun tersebut bukanlah tahun yang baik untuk melakukan riset matematika. Sekretaris pertahanan di negara tempat Bellman melakukan riset sedang sensitif dengan istilah riset dan matematika sehingga sangat memungkinkan akan timbul permasalahan apabila Bellman melakukan riset tersebut. Oleh karena itu, Bellman mempertimbangkan nama riset yang tidak melibatkan istilah matematika. Kemudian dia memiliki ide untuk menamai risetnya *Dynamic Programming* karena nama tersebut tidak akan dipermasalahkan pada konggres dan nama tersebut terdengar impresif.

C. Karakteristik Program Dinamis

Berikut adalah karakteristik Program Dinamis secara umum:

1. Program dinamis digunakan pada persoalan yang dapat **dibagi menjadi beberapa tahap** (*stage*), pada setiap tahap diambil solusi yang terbaik [2].
2. Masing-masing dari **setiap tahap terdiri dari sejumlah status** (*state*) yang terasosiasi dengan tahap tersebut [2].
3. **Prinsip Optimalitas**

   Mayoritas dari pengaplikasian Program Dinamis mengatasi permasalahan optimasi. Prinsip yang digunakan adalah apabila solusi akhir optimal, maka bagian solusi pada tahap ke-k juga harus optimal.
4. **Memoisasi**

   Solusi setiap tahap pada persoalan disimpan dalam memori agar ketika solusi pada suatu tahap dibutuhkan, kita tinggal mengambil hasil perhitungan solusi yang sudah disimpan pada memori. Penyimpanan pada memori umumnya dilakukan dengan *array* atau *matrix*. Memoisasi umumnya digunakan untuk pendekatan Program Dinamis *top-down*.
5. **Rekursi**

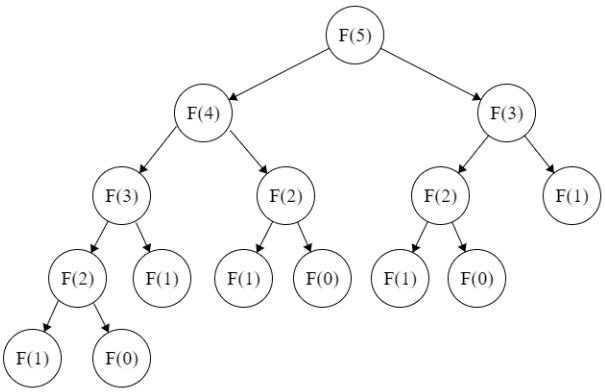
   Karena solusi dari persoalan membutuhkan solusi terbaik dari tahap-tahap sebelumnya, persoalan Program Dinamis secara umum didefinisikan dengan relasi rekurens.
6. Misalnya diberi sebuah tahap, maka solusi terbaik dari tahap tersebut **independen** terhadap solusi terbaik yang dihitung dari tahap sebelumnya [4].

D. Kelebihan dan Kekurangan Program Dinamis

Program dinamis merupakan teknik yang cukup impresif karena dapat **memecahkan suatu permasalahan dalam waktu yang relatif lebih cepat**. Hal ini demikian karena Program Dinamis mengatasi Permasalahan Tumpang Tindih (*Overlapping Problem*) dengan menggunakan teknik memoisasi. Sebagai contoh, mari kita lihat program fungsi Fibonacci sederhana berikut.

```
function Fibonacci(n : integer) → integer
{ Mengembalikan angka ke-n dari baris Fibonacci }
ALGORITMA
if (n = 0) then
  → 0
else of (n = 1) then
  → 1
else
  → Fibonacci(n-1) + Fibonacci(n-2)
```

Apabila kita membentuk pohon rekursi untuk n=5, dengan F(n) memiliki arti angka ke-n dari baris Fibonacci, maka kita memperoleh gambar berikut.



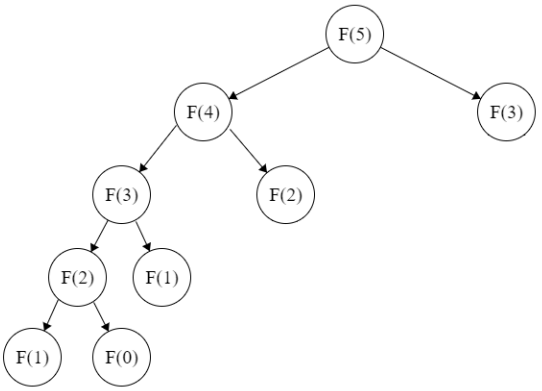
Gambar 1. Pohon pemanggilan fungsi Fibonacci(5)

Dari Gambar 2, kita dapat melihat bahwa pemanggilan F(3) dilakukan sebanyak dua kali, dan pemanggilan F(2) dilakukan sebanyak tiga kali. Hal ini sangat tidak efisien kareng sering dilakukan pemanggilan fungsi secara berulang sehingga menyebabkan kompleksitas program menjadi sangat tinggi. Kejadian ini yang disebut dengan **Permasalahan Tumpang Tindih (*Overlapping Problem*)**.

Permasalahan ini dapat diselesaikan dengan pendekatan Program Dinamis yaitu dengan membuat array global yang menyimpan setiap angka ke-i dari baris Fibonacci. Program Fibonacci yang pertama dapat dimodifikasi dengan pendekatan Program Dinamis menjadi program berikut.

```
dp : array[1..MAX] of integer { Global }
{ inisialiasi dp[0] = 0, dp[1] = 1, dp[2..MAX] = -1 }
function Fibonacci(n : integer) → integer
{ Mengembalikan angka ke-n dari baris Fibonacci }
ALGORITMA
if (dp[n] ≠ -1) then
  → dp[n] { ambil nilai dari memo }
else
  dp[n] = dp[n-1] + dp[n-2] { isi memo }
  → Fibonacci(n-1) + Fibonacci(n-2)
```

Sekarang, mari kita bentuk pohon pemanggilan fungsi untuk n=5, hasilnya adalah sebagai berikut.



Gambar 2. Pohon pemanggilan fungsi Fibonacci(5) dengan Program Dinamis

Apabila kita membandingkan program Fibonacci dengan dan tanpa teknik Program Dinamis, terlihat bahwa dengan teknik Program Dinamis, setiap tahap hanya dihitung sekali dan apabila nilai dari sebuah tahap dibutuhkan lagi, nilainya cukup diambil dari memori tanpa melakukan perhitungan ulang. Oleh karena modifikasi program Fibonacci dengan Program Dinamis, kompleksitas yang semula adalah $O(2^n)$ (setiap tahap dibagi menjadi 2 cabang) menjadi hanya O(n) (setiap tahap hanya dihitung sekali).

Walaupun pendekatan Program Dinamis dapat mempercepat kinerja program, ada alokasi memori yang harus dibayar. Pada contoh sebelumnya, kompleksitas ruang untuk fungsi Fibonacci dengan Program Dinamis adalah O(n) karena dibutuhkan *array* berukuran 0..n untuk menghitung Fibonacci yang ke-n. Hal ini berakibat teknik Program Dinamis menjadi **boros memori** untuk nilai n yang besar sehingga teknik ini menjadi kurang *feasible* untuk ukuran persoalan yang sangat besar.

Selain boros memori, Program Dinamis juga dibatasi dengan persoalan dengan *integer constraint*, artinya solusi yang mungkin dibangkitkan harus berhingga. Akibatnya, teknik Program Dinamis tidak dapat diterapkan untuk persoalan dengan *non-integer constraint*, misalnya persoalan *Fractional Knapsack*.

E. Pendekatan Program Dinamis

Pendekatan Program Dinamis terbagi menjadi dua pendekatan, yaitu **maju (*top-down*)** dan **mundur (*bottom-up*)**.

1. *Top-Down*

   Secara umum, implementasi Program Dinamis dengan *top-down* dilakukan secara **rekursif** sambal mencatat nilai yang sudah ditemukan (**memoisasi**). Pada *top-down*, penyelesaian masalah **dimulai dari kasus yang besar**, kemudian mencari dan menggunakan solusi untuk kasus permasalahan yang lebih kecil untuk menyelesaikan permasalahan yang besar. Fungsi Fibonacci dengan Program Dinamis pada poin D menggunakan pendekatan *top-down*.

2. *Bottom-Up*

   Berbeda dengan pendekatan *top-down*, pada pendekatan ini penyelesaian masalah **dimulai dari kasus yang kecil**. Ketika merususkan relasi rekurens, kita sudah tahu solusi untuk kasus basis. Solusi basis digunakan untuk menentukan solusi dari tahap berikutnya. Pendekatan ini biasanya dianalogikan dengan **pengisian "tabel program dinamis"**. Salah satu hal yang harus diingat adalah walaupun permasalahan didefinisikan dengan relasi rekurens, pada pendekatan *bottom-up* penyelesaian masalah dilakukan secara **iteratif**.

F. Jarak Levenshtein

Jarak Levenshtein dari dua buah *sequence* adalah banyaknya operasi sisip, hapus, dan substitusi minimum yang dibutuhkan untuk mengubah sebuah *sequence* pertama menjadi sama seperti *sequence* kedua. Operator yang terlibat adalah operator sisip, hapus, dan substitusi. Operasi sisip dan hapus memiliki nilai 1 sedangkan operasi substitusi memiliki nilai 2. Secara umum *sequence* yang dimaksud adalah kumpulan karakter/kata, kadang-kadang dapat berupa kumpulan bilangan bulat.

Persamaan 1: Jarak Levenshtein

$$LD_{a,b} = n(I) + n(D) + n(S) \times 2$$

Dengan $LD_{a,b}$ adalah Jarak Levenshtein dari dua sumber, n(I) adalah banyaknya operasi sisip, n(D) adalah banyaknya operasi hapus, dan n(S) adalah banyaknya operasi substitusi.

Misalnya kita memiliki dua buah kata, yaitu "stima" dan "timoy", maka Jarak Levenshtein dari dua buah kata tersebut sebesar 4. Berikut adalah proses perhitungan jaraknya.

1. Hapus 's' : stima → tima 1
2. Substitusi 'a' dengan 'o' : tima → timo 2
3. Tambah 'y' : timo → timoy 1
4. Jarak Levenshtein = 1 + 2 + 1 = 4.

Berikut adalah ilustrasi operasi yang terjadi untuk mengubah kata "stima" menjadi "timoy".

```
String 1: S T I M A .
String 2: . T I M O Y
Operasi : I . . . S D
Operasi I adalah operasi sisip (Insert)
Operasi D adalah operasi hapus (Delete)
Operasi S adalah operasi substitusi (Substitution)
Jarak Levenshtein = n(I) + n(S) x 2 + n(D)
                  = 1 + 1 x 2 + 1
                  = 4
```

Jarak Levenshtein memiliki banyak kegunaan, diantaranya mendeteksi plagiarisme dari dua buah kode atau karya ilmiah,

mendeteksi kemiripan DNA, dan mencari kata terdekat untuk melakukan *auto-correct* [3]. Pada makalah ini penulis berfokus membahas mengenai penggunaan Jarak Levenshtein untuk mendeteksi plagiarisme untuk kode program (*source code*) dan file teks.

Jarak Levenshtein dapat digunakan untuk mendeteksi plagiarisme karena nilainya menunjukkan seberapa miripnya dua buah *sequence*. Jarak Levenshtein yang lebih rendah menunjukkan bahwa dua buah *sequence* memiliki kemiripan yang lebih tinggi. Sebaliknya, semakin jauh Jarak Levenshtein, semakin berbeda dua buah *sequence* tersebut. Artinya, apabila Jarak Levenshtein dari dua sumber bernilai rendah, ada potensi bahwa salah satu sumber adalah plagiat dari sumber yang lain.

Akan tetapi, Jarak Levenshtein belum cukup untuk mengetahui kedekatan dari dua sumber karena belum mempertimbangkan ukuran dari kedua sumber. Misalnya, apabila panjang kedua *sequence* sebesar 100, maka Jarak Levenshtein sebesar 80 menunjukkan kedua sumber memiliki kemiripan (*similarity*) sebesar 20%. Namun, bila panjangnya kita ubah menjadi 1000, kemiripan akan naik drastis menjadi 92%. Oleh karena itu, kita harus memiliki persamaan kedua, yaitu persamaan indeks kemiripan (*similarity*).

Persamaan 2: Indeks Kemiripan (*Similarity*)

$$S_{a,b} = 1 - (LD_{a,b} / \max(\text{length}(a), \text{length}(b)))$$

Dengan $S_{a,b}$ adalah indeks kemiripan dari sumber a dan b, $LD_{a,b}$ adalah Jarak Levenshtein, dan length(x) adalah operasi mendapat panjang dari x.

G. Plagiarisme

Menurut KBBI, plagiarisme adalah penjiplakan yang melanggar hak cipta. Umumnya plagiarisme dilakukan dengan melakukan *copy-paste* dari sebuah sumber, kemudian beberapa kata diubah agar terlihat berbeda dari sumber, tetapi sumber yang digunakan tidak disebutkan di referensi. Pada kode program, plagiarisme umumnya dilakukan dengan mengubah nama variabel, sedangkan pada karya ilmiah plagiarisme dilakukan dengan mengubah beberapa kata dengan sinonimnya.

III. Implementasi Pendeteksian Plagiarisme dengan Jarak Levenshtein

Pada bagian ini, penulis akan membahas mengenai implementasi Jarak Levenshtein untuk mendeteksi plagiarism, dimulai dari mendefinisikan persoalan, menentukan basis dan rekurens persoalan, dan membentuk tabel program dinamis, membuat *pseudocode* program, dan menentukan solusi optimal dari tabel yang sudah dibuat.

A. Definisi Persoalan

Diberikan dua buah string A dan B, kita harus menentukan indeks kemiripan (*similarity*) dari kedua string tersebut. Indeks kemiripan dan Jarak Levenshtein dapat dihitung menggunakan persamaan 1 dan 2 dari bagian II.

Pada persoalan ini, kita akan mendefinisikan sebuah matriks *integer* yang berukuran (length(A)+1) x (length(B)+1). Matriks pada posisi (i,j) menyimpan Jarak Levenshtein dari string A saat panjang yang ke-i dan string B saat panjang yang ke-j. Artinya, solusi akhir terletak pada matriks di posisi (length(A), length(B)). Proses pengisian matriks ini akan dibahas lebih detil pada poin C.

B. Relasi Rekurens

Kita definisikan f(i,j) sebagai fungsi Jarak Levenshtein dari string A saat panjang yang ke-i dan string B saat panjang yang ke-j, dan dengan ketentuan 0 <= i <= length(A) dan 0 <= j <= length(B). Perlu diketahui bahwa string A dan B harus ditambahkan sebuah karakter "dummy" pada posisi terawal (indeks nol) agar indeks ke-i dan ke-j dapat mencerminkan panjang string A dan B yang sedang dievaluasi.

Dari definisi fungsi f(i,j), kita tahu bahwa untuk i=0, maka f(i,j) = j, dan untuk j=0, maka f(i,j) = i. Hal ini demikian karena apabila string A kosong, maka untuk mengubah string A menjadi B kita perlu melakukan operasi sisip sebanyak j kali. Apabila string B kosong, artinya kita melakukan operasi hapus sebanyak i kali. Kasus ini adalah **kasus basis**.

Basis

$$f(i,0) = i \text{ dan } f(0,j) = j$$

Dengan f(i,j) adalah Jarak Levenshtein dari string A saat panjang yang ke-i dan string B saat panjang yang ke-j.

Dalam **kasus rekurens**, kita memiliki empat kemungkinan, yaitu operasi sisip, hapus, substitusi, dan tidak terjadi operasi. Pada kasus operasi sisip, nilai f(i,j) akan bernilai sama dengan f(i,j-1)+1, sedangkan pada kasus menghapus, nilai f(i,j) akan bernilai f(i-1,j)+1. Pada kasus substitusi, nilai f(i,j) bernilai sama dengan f(i-1,j-1)+2. Kita hanya akan mengambil Jarak Levenshtein minimum dari ketiga operasi tersebut. Berikut adalah definisi kasus rekurens dari persoalan.

Rekurens

$$f(i,j) = \begin{cases} \min f(i-1,j) + 1 \\ \min f(i,j-1) + 1 \\ \min f(i-1,j-1) + 2 \; ; A_i \neq B_i \\ 0 \; ; A_i = B_i \end{cases}$$

Dengan f(i,j) adalah Jarak Levenshtein dari string A saat panjang yang ke-i dan string B saat panjang yang ke-j.

C. Proses Pengisian Matriks

Langkah pertama dalam pengisian matriks adalah mengisi nilai-nilai basis terlebih dulu. Karena kita mengisi matriks dari basis, pendekatan yang digunakan adalah ***bottom-up***.

Mari kita gunakan string A = “stima” dan B = “timoy” sebagai contoh.

	-	T	I	M	O	Y	
-	0	1	2	3	4	5	i=0
S	1						i=1
T	2						i=2
I	3						i=3
M	4						i=4
A	5						i=5
	j=0	j=1	j=2	j=3	j=4	j=5	

Gambar 3. Matriks yang sudah diinisialisasi dengan basis

Berikutnya, kita lakukan pengisian nilai pada setiap baris. Pengisian dilakukan dengan menggunakan kasus rekurens, yaitu mengambil jumlah operasi yang paling minimum. Mari kita isi tabel pada baris i=1.

$$f(1,1) = \min(f(0,1)+1, f(1,0)+1, f(0,0)+2)$$
$$= \min(2, 2, 2)$$
$$= 2$$
$$f(1,2) = \min(f(0,2)+1, f(1,1)+1, f(0,1)+2)$$
$$= \min(3, 3, 3)$$
$$= 3$$
$$f(1,3) = \min(f(0,3)+1, f(1,2)+1, f(0,2)+2)$$
$$= \min(4, 4, 4)$$
$$= 4$$
$$f(1,4) = \min(f(0,4)+1, f(1,3)+1, f(0,3)+2)$$
$$= \min(5, 5, 5)$$
$$= 5$$
$$f(1,5) = \min(f(0,5)+1, f(1,4)+1, f(0,4)+2)$$
$$= \min(6, 6, 6)$$
$$= 6$$

Pada tahap ini matriks pada baris ke-1 sudah terisi. Berikut adalah hasilnya.

	-	T	I	M	O	Y	
-	0	1	2	3	4	5	i=0
S	1	2	3	4	5	6	i=1
T	2						i=2
I	3						i=3
M	4						i=4
A	5						i=5
	j=0	j=1	j=2	j=3	j=4	j=5	

Gambar 4. Matriks yang sudah terisi sebagian kecil

Apabila kita melanjutkan tahap untuk sisa baris yang belum dievaluasi, matriks akan memiliki nilai berikut.

	-	T	I	M	O	Y	
-	0	1	2	3	4	5	i=0
S	1	2	3	4	5	6	i=1
T	2	1	2	3	4	5	i=2
I	3	2	1	2	3	4	i=3
M	4	3	2	1	2	3	i=4
A	5	4	3	2	3	4	i=5
	j=0	j=1	j=2	j=3	j=4	j=5	

Gambar 5. Matriks yang sudah terisi seluruhnya. Warna jingga menunjukkan solusi akhir.

Pada tahap ini matriks sudah terisi penuh. Selanjutnya kita dapat menentukan indeks kemiripan dengan solusi akhir yang kita miliki.

D. Menentukan Solusi Optimal

Dari matriks yang sudah terisi penuh, kita memperoleh Jarak Levenshtein dari string A dan B pada posisi (length(A), length(B)). Dari gambar 5, kita memperoleh Jarak Levenshtein antara string “stima” dan “timoy” sebesar 4.

Namun, kita juga harus menghitung indeks kemiripan string A dan B, karena indeks tersebut yang mencerminkan tingkat kemiripan dari dua string. Dengan menggunakan persamaan 2, kita memperoleh:

$$S_{A,B} = 1 - (4/5) = 0.2$$
$$S_{A,B}\ (\%) = 20\%$$

Indeks kemiripan ($S_{A,B}$) sebesar 20% menunjukkan bahwa string A = “stima” dan string B = “timoy” memiliki tingkat kemiripan sebesar 20%. Persentase inilah yang menjadi metriks penentu apakah sebuah sumber merupakan plagiat dari sumber lain.

E. Pseudocode

Berikut adalah *pseudocode* untuk menghitung Jarak Levenshtein. Karena pendekatan Program Dinamis yang digunakan adalah *bottom-up*, implementasi program disusun secara **iteratif**.

```
function Levenshtein(A, B : string) → integer
KAMUS
  stringA, stringB : integer
  lengthA, lengthB : integer
  dp : array[0..MAX][1..MAX] of integer
ALGORITMA
  { Ubah string agar indeks dimulai dari 1 }
  stringA ← "-" + A
```

```
  stringB ← "-" + B
  lengthA ← length(stringA)
  lengthB ← length(stringB)
  { Basis }
  for i in [0..lengthA]
    dp[i][0] = i
  for j in [0..lengthB]
    dp[0][j] = j
  { Rekurens }
  for i in [1..lengthA]
    for j in [1..lengthB]
      if (stringA[i] = stringB[j]) then
        dp[i][j] = dp[i-1][j-1]
      else
        dp[i][j] = min(
          dp[i-1][j] + 1, { hapus }
          dp[i][j-1] + 1, { sisip }
          dp[i-1][j-1] + 2, { substitusi }
        )
  → dp[lengthA][lengthB]
```

IV. Hasil Eksperimen dan Analisis

Berdasarkan *pseudocode* pada bagian III, penulis menyusun program untuk melakukan eksperimen plagiarisme beberapa kode program, teks, dan karya ilmiah. Hasil eksperimen tersebut ditabulasi pada tabel berikut.

Test Case	Nama File A	Nama File B	Panjang Maksimum
1	textSumber.txt	textBaik.txt	629
2	textSumber.txt	textPlagiat.txt	629
3	matdis51.txt	matdis102.txt	14748
4	javaSumber.java	javaBaik.java	688
5	javaSumber.java	javaPlagiat.java	579
6	tubesErick.java	tubesOrglain.java	15549

Catatan: konten dari setiap file dapat diakses di akun GitHub penulis. Untuk selengkapnya lihat Apendiks (bagian VI).

Test Case	$LD_{A,B}$	$S_{A,B}$ (%)	Diagnosis Tingkat Kemiripan
1	454	27.82	Rendah
2	122	80.60	Tinggi
3	10454	29.12	Rendah
4	399	42.01	Rendah
5	134	76.86	Tinggi
6	11426	26.52	Rendah

Testcase nomor 1 dan 2 menunjukkan perbandingan indeks kemiripan antara sumber (File A) dengan tulisan yang dibuat (File B). Testcase nomor 4 dan 5 menunjukkan perbandingin indeks kemiripan antara program berbahasa Java yang dibuat tanpa melakukan plagiarisme dan dengan sengaja melakukan plagiarisme. Testcase nomor 3 adalah pengukuran indeks kemiripan dari kedua makalah IF2120 Matematika Diskrit tahun 2016/2017 yang memiliki topik serupa. Testcase nomor 6 adalah pengukuran indeks kemiripan antara tugas besar OOP (Pemrograman Berorientasi Objek) berbasis Java Swing yang dimiliki penulis dengan yang dimiliki orang lain.

Untuk testcase yang mengukur indeks kemiripan dari dua buah program, penulis menetapkan bahwa kedua program memiliki kemiripan yang **rendah apabila persentase kemiripan program dibawah 70%** karena umumnya setiap program memiliki banyak karakter yang sama seperti spasi, titik koma, dan kurung. Sedangkan, file teks atau karya ilmiah memiliki kemiripan yang **rendah apabila persentase kemiripan teks dibawah 50%**.

Dari tabel, kita dapat melihat bahwa testcase yang sengaja dibuat tidak plagiat (testcase 1 dan 4) memiliki persentase kemiripan yang kecil dan lebih rendah dari batas persentase yang sudah penulis tentukan. Sebaliknya, testcase yang dibuat dengan cara plagiarisme (testcase 2 dan 5) memiliki persentasi kemiripan yang cukup tinggi. Hal ini menunjukkan bahwa program yang dibuat berhasil mendeteksi plagiarisme dari kode program dan teks/karya ilmiah.

Pada testcase 3 dan 6, penulis mencoba melakukan eksperimen dari sumber yang sudah dipublikasikan. Untuk testcase 3, penulis mengambil dua makalah yang sama-sama bertopik implementasi struktur data pohon dalam permainan DOTA 2. Walaupun topiknya serupa, penulis sudah membaca dari dua makalah tersebut dan tahu bahwa keduanya memiliki gaya bahasa yang berbeda. Hasil eksperimen pun juga menunjukkan bahwa keduanya memiliki tingkat kemiripan yang rendah. Selain itu, penulis juga mencoba membandingkan kode program tugas besar OOP Java Swing milik penulis sendiri dengan orang lain. Program yang dibandingkan hanyalah bagian permainan utama (*Game*) dan tampilannya (*JFrame*). Hasil eksperimen mencerminkan bahwa kedua potongan kode program tugas besar tersebut memiliki tingkat kemiripan yang rendah.

Persentase kemiripan ditentukan dengan menggunakan persamaan 2. Pada persamaan tersebut, kita membutuhkan nilai Jarak Levenshtein antara string A dan B, dan panjang string maksimum antara string A dan B. Kedua komponen ini harus ada sebelum kita ingin menentukan persentasi kemiripan. Hal yang harus diingat bahwa Jarak Levenshtein belum dapat menentukan tingkat kemiripan, tetapi harus beserta dengan panjang maksimum sumber.

Untuk analisis performansi program, pertama kita menentukan fungsi T(n,m) yang menghitung banyaknya operasi dasar yang terjadi pada algoritma. Dari *pseudocode*

fungsi Levenshtein, kita lihat bahwa operasi yang paling sering dilakukan adalah melakukan pemeriksaan dan mengisi nilai matriks dp[i][j]. Karena proses ini terjadi pada loop berganda, waktu eksekusi menjadi kuadratik. Selain itu, pada kasus basis terjadi pengisian matriks untuk baris dan kolom pertama. Dengan ini kita dapat merumuskan formula T(n,m).

$$T(n,m) = (n+1) + (m+1) + (n \times m)$$

$$T(n,m) = nm + n + m + 2$$

Berikutnya, kita menentukan kompleksitas algoritma dari persamaan T(n,m) diatas. Kompleksitasnya menjadi:

$$\text{Kompleksitas} = O(nm + n + m + 2)$$

$$\text{Kompleksitas} = \mathbf{O(nm)}$$

V. KESIMPULAN

Berdasarkan hasil eksperimen dan analisis, dapat disimpulkan bahwa plagiarism pada kode program dan karya ilmiah dapat dideteksi dengan mengukur Jarak Levenshtein dan menghitung indeks kemiripan dari dua sumber. Indeks yang dihasilnya cukup akurat untuk mendeteksi potensi plagiarisme. Performansi program pendeteksi plagiarisme yang dibuat penulis memiliki kompleksitas kuadratik sehingga program dapat berjalan dengan baik pada ukuran testcase yang menengah kebawah, namun performa menurun apabila ukuran testcase sangat besar.

VI. APENDIKS

Implementasi program yang dibuat penulis beserta hasil eksperimen dapat diakses pada GitHub melalui tautan https://github.com/wijayaerick/makalah-stima-levenshtein.
Program ditulis dengan bahasa Java sehingga diperlukan Java JDK 7 untuk mengeksekusi program. Selain itu, setiap gambar yang terdapat di makalah ini adalah buatan penulis dan dapat diakses melalui tautan yang sudah disebutkan.

UCAPAN TERIMA KASIH

Ucapan syukur penulis panjatkan kepada Tuhan Yang Maha Esa, karena atas berkat, rahmat, dan penyertaan-Nya penulis dapat menyelesaikan makalah ini. Penulis juga mengucapkan terima kasih kepada Bapak Rinaldi Munir selaku dosen matakuliah Strategi Algoritma yang sudah menyempatkan waktunya untuk membagi ilmu dan mengajar dengan penuh pengertian kepada mahasiswanya. Terakhir, penulis juga berterima kasih kepada orang tua penulis yang secara terus-menerus mendukung penulis untuk menjalani studi di ITB.

REFERENSI

[1] Levitin, Anany. *Introduction to The Design and Analysis of Algorithms*. London: Pearson. 2011.

[2] Munir, Rinaldi. Diktat Kuliah IF2211 Strategi Algortima. Program Studi Teknik Informatika ITB.

[3] http://web.stanford.edu/class/cs124/lec/med.pdf (Diakses terakhir pada 18 Mei 2016 pukul 16.22 WIB)

[4] https://training.ia-toki.org/training/curriculums/1/courses/11/chapters/55/lessons/23/ (Diakses terakhir pada 17 Mei 2016 pukul 21.48 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, 18 Mei 2016

Erick Wijaya – 13515057

LAMPIRAN

Berikut adalah hasil eksperimen yang dilakukan penulis.

```
C:\Users\asus\Desktop\Stima\Makalah\src>java PlagiarismDetector tc/textSumber.txt tc/textBaik.txt
Panjang String Maksimum: 629
Jarak Levenshtein: 454
Similarity(%): 27.82

C:\Users\asus\Desktop\Stima\Makalah\src>java PlagiarismDetector tc/textSumber.txt tc/textPlagiat.txt
Panjang String Maksimum: 629
Jarak Levenshtein: 122
Similarity(%): 80.60
```

```
C:\Users\asus\Desktop\Stima\Makalah\src>java PlagiarismDetector tc/javaSumber.java tc/javaBaik.java
Panjang String Maksimum: 688
Jarak Levenshtein: 399
Similarity(%): 42.01

C:\Users\asus\Desktop\Stima\Makalah\src>java PlagiarismDetector tc/javaSumber.java tc/javaPlagiat.java
Panjang String Maksimum: 579
Jarak Levenshtein: 134
Similarity(%): 76.86
```

```
C:\Users\asus\Desktop\Stima\Makalah\src>java PlagiarismDetector tc/matdis51.txt tc/matdis102.txt
Panjang String Maksimum: 14748
Jarak Levenshtein: 10454
Similarity(%): 29.12

C:\Users\asus\Desktop\Stima\Makalah\src>java PlagiarismDetector tc/tubesErick.java tc/tubesOrglain.java
Panjang String Maksimum: 15549
Jarak Levenshtein: 11426
Similarity(%): 26.52
```