Penerapan *Abstract Syntax Tree* dan Algoritme *Damerau-Levenshtein Distance* untuk Mendeteksi Plagiarisme pada Berkas *Source Code*

Stephanie Rusdianto[#1], Ria Chaniago[#2]

[#]*Departemen Informatika, Institut Teknologi Harapan Bangsa*
Bandung, Indonesia

[1]louisastephanie092@gmail.com
[2]ria@ithb.ac.id

Abstract*— Plagiarism source code is a program that is made up of other programs that have same syntax structure. In this research, the approach used to detect plagiarism is tree-based by building Abstract Syntax Tree base on grammar on two predefined plagiarism files source code. Damerau-Levenshtein Distance Algorithm will calculate the tree structure formed minimum distance value to get the percentage of similarity. Previously, the application calculated the value of threshold obtained from the average value of plagiarism plot paired pairs, and then were reduced to its standard deviation to be able to declare that both files are plagiarism or not. This research analyzes the best use of grammar between jexer rule or a combination of lexer and parser rule, the best use of preprocessing combination and the best use of distance value of Damerau-Levenshtein Distance Algorithm. Based on the tests performed, the use of grammar lexer and parser rule resulted the highest accuracy of 97.435 % by taking 118,115 seconds and threshold used is 88.2314 %.The combination of preprocessing resulted highest accuracy of 97.435% by using whole preprocessing existing or by using preprocessing comment only. For the best distance value is 4 with highest accuracy 97.435 %.

***Keywords*— plagiarism source code, grammar, lexer rule, parser rule, tree-based, abstract syntax tree, Damerau-Levenshtein Distance algorithm**

Abstrak*— Plagiarisme *source code* adalah jika sebagai sebuah program yang terbentuk dari program lainnya memiliki struktur *syntax* yang sama. Dalam penelitian ini, pendekatan yang digunakan untuk mendeteksi plagiarisme adalah *tree-based* dengan membangun *abstract syntax tree* atas dua berkas *source code* terduga plagiat berdasarkan *grammar* yang telah dirancang. Struktur *tree* yang terbentuk akan dihitung nilai jarak minimumnya dengan *Damerau-Levenshtein Distance Algorithm* untuk mendapatkan persentase kemiripan. Sebelumnya, aplikasi menghitung nilai *threshold* yang didapatkan dari nilai rata-rata kemiripan pasangan berkas plagiat yang dikurangi dengan simpangan bakunya untuk dapat menyatakan kedua berkas masukan plagiat atau tidak. Penelitian ini menganalisis penggunaan *grammar* terbaik antara *lexer rule* atau kombinasi *lexer* dan *parser rule*, penggunaan kombinasi *preprocessing* terbaik, serta penggunaan nilai jarak terbaik pada *Damerau-Levenshtein Distance Algorithm*. Berdasarkan pengujian yang dilakukan, penggunaan *grammar lexer* dan *parser rule* menghasilkan akurasi tertinggi, yaitu 97,435 % dengan memakan waktu 118,115 detik dengan nilai *threshold* 88,2314 %. Kombinasi *preprocessing* yang menghasilkan akurasi tertinggi 97,435 % menggunakan seluruh *preprocessing* yang ada atau dengan menggunakan *prerpocessing comment* saja. Untuk nilai jarak terbaik adalah nilai jarak sebesar 4 dengan akurasi tertinggi, yaitu 97,435 %.

***Kata Kunci*— plagiarism source code, grammar, lexer rule, parser rule, tree-based, abstract syntax tree, Damerau-Levenshtein Distance algorithm**

I. PENDAHULUAN

Plagiarisme atau plagiat dapat diartikan sebagai tindakan secara sengaja atau tidak sengaja dalam memperoleh nilai dari suatu karya ilmiah, dengan mengutip sebagian atau seluruh karya ilmiah pihak lain yang diakui sebagai karya milik sendiri tanpa izin pemiliknya [1]. Menurut Peraturan Menteri Pendidikan Nasional Republik Indonesia, definisi karya ilmiah adalah hasil karya akademik mahasiswa, dosen, peneliti, tenaga kependidikan di lingkungan perguruan tinggi yang dibuat dalam bentuk tertulis baik cetak maupun elektronik yang diterbitkan dan/atau dipresentasikan, termasuk didalamnya adalah perangkat lunak komputer atau *software* [2]. Salah satu komponen *software* adalah *source code* yang merupakan sekumpulan instruksi komputer dan definisi data yang dituangkan dalam bentuk yang dapat dimengerti dari sisi manusia dan komputer [3]. Plagiarisme *source code* [4] di definisikan sebagai sebuah program yang terbentuk dari program lainnya dengan beberapa perubahan. *Source code* plagiarisme dapat bervariasi mulai dari menjiplak sekumpulan kecil dari *source code* lain sampai keseluruhan dari *source code*. Khususnya dalam jurusan Informatika, sangat banyak tugas kuliah yang diberikan kepada mahasiswa dalam bentuk dokumen teks dan berkas *source code*. Beberapa mahasiswa sering kali melakukan plagiarisme dengan menduplikat berkas *source code* milik teman sekelasnya untuk memenuhi tugas yang diberikan oleh dosen. Mahasiswa yang terbukti melakukan plagiarisme bisa saja mendapatkan sanksi berupa pembatalan ijazah perguruan tinggi [5]. Untuk itu, dibutuhkan sebuah aplikasi untuk mendeteksi plagiarisme pada berkas *source code*.

Terdapat 4 kategori pendekatan [6] untuk mendeteksi plagiarisme pada berkas *source code*, yaitu *text-based* atau *string-based, token-based, tree-based* atau *parse tree* dan *program dependency graph*. Metode *token-based* [6]

mengubah setiap *code* yang ada ke dalam bentuk token-token dan mengubahnya ke dalam *lexical rules*. Dengan metode ini, perubahan nama dapat diatasi. Namun, metode ini tidak dapat menangani apabila terjadi perubahan susunan urutan *source code* yang diuji. Contoh *tools* yang menggunakan metode ini adalah CP-Miner [7] dan CCFinder [4]. Metode *text-based* [8] menganggap seluruh *syntax* pada berkas *source code* seperti baris-baris kalimat. Apabila semua *syntax* dalam berkas *source code* yang diuji sama persis, dapat disimpulkan bahwa kedua berkas tersebut sama. Contoh algoritme yang menggunakan metode *text-based* adalah *Longest Common Subsequence* (LCS) dan *Greedy String Tilling* [9]. Apabila terjadi perubahan *syntax* dalam berkas yang diuji, maka plagiarisme tidak dapat dideteksi. Metode *tree-based* [4,6,10] merupakan metode yang mengubah *source code* menjadi *abstract syntax tree* dan perbandingan dilakukan berdasarkan struktur *tree* yang terbentuk. Pendeteksian plagiarisme memberikan hasil yang lebih baik dengan metode ini karena menggunakan struktur *source code* dan dapat ditingkatkan dengan mengabstraksikan *syntax source code*. Metode *program dependency graph* [6] dapat mendeteksi kesamaan arti dalam *source code*. Namun, dibutuhkan komputasi yang besar dan kompleksitasnya tinggi. Metode untuk melakukan perhitungan tingkat kemiripan yang dibandingkan adalah *Damerau-Levenshtein Distance Algorithm* dengan *Levenshtein Distance Algorithm* [10,11]. Berdasarkan hasil penelitian pada kedua jurnal tersebut, *Damerau-Levenshtein Distance Algorithm* merupakan pengembangan dari *Levenshtein Distance Algorithm* di mana terdapat penambahan operasi transposisi yang sebelumnya tidak ada. Selain itu, metode tersebut dapat menangani perbedaan panjang kedua berkas yang akan dihitung kemiripannya.

Metode *tree-based* akan digunakan dalam membangun sistem aplikasi agar lebih akurat dalam mendeteksi plagiarisme. *Source code* akan mengalami tahap *preprocessing* sebelum diubah ke dalam bentuk *syntax tree*. *Syntax tree* akan dibentuk dari *source code* yang diperiksa. Diperlukan aturan tata bahasa pemrograman untuk mengabstraksi *source code* yang disebut dengan *grammar*. *Grammar* dirancang dari referensi ANTLR [11] untuk bahasa pemrograman C++. ANTLR merupakan *plugin generator parser* untuk membaca struktur berkas. *Grammar* akan dimodifikasi menyesuaikan dengan karakteristik data berkas *source code* yang akan diperiksa serta dirancang berbentuk *parser rule* dan *lexer rule*. Kemudian struktur *source code* yang telah didapatkan akan diproses untuk dihitung kemiripannya dengan algoritme *Damerau-Levenshtein Distance*. Nilai akhir jarak kedua berkas akan didapatkan dan persentase kemiripan akan dihitung dengan rumus *similarity* [10]. Dengan dirancangnya sistem aplikasi ini, diharapkan dapat membantu dosen pengajar dalam menemukan plagiarisme atas berkas *source code* dengan cepat dan otomatis.

II. KONTEN UTAMA (METODOLOGI/DASAR TEORI)

A. Bahasa Pemrograman C++

Bahasa C++ [12] diciptakan pada tahun 1980 oleh Bjarne Stroustrup berdasarkan C ANSI (*American National Standard Institute*). Kala itu, Bjarne Stroustrup melakukan pekerjaan untuk mendapatkan gelar Ph. D. Stroustrup bekerja dengan bahasa pemrograman Simula yang merupakan bahasa untuk simulasi. Simula merupakan varian bahasa yang dianggap sebagai bahasa pertama untuk mendukung paradigm pemrograman berorientasi objek. Stroustrup menemukan bahwa paradigma ini sangat berguna untuk pengembangan perangkat lunak, namun bahasa Simula terlalu lambat untuk penggunaan praktis.

Prototype C++ muncul pertama kali sebagai C yang dilengkapi dengan *class*. Bahasa tersebut disebut dengan C *with class* yang dimaksudkan untuk menjadi superset dari bahasa C. Tujuannya adalah untuk menambahkan pemrograman berorientasi objek ke dalam bahasa C. Di tahun 1983 sampai 1984, *C with class* disempurnakan dengan menambahkan fasilitas pembeban lebihan operator dan fungsi awalnya disebut dengan "*a better C*" yang kemudian disebut sebagai C++. Simbol "++" menandakan bahwa bahasa baru ini merupakan versi yang lebih canggih daripada C.

Pada tahun 1985, Stroustrup menerbitkan referensi untuk bahasa C++ dengan judul "*The C++ Programming*". Pada tahun yang sama, C++ diterbitkan sebagai produk komersial. Bahasa C++ kala itu berlum secara resmi distandarkan. Hal tersebut membuat buku referensi yang diterbitkan oleh Stroustrup sangat penting untuk dijadikan dasar. Pada tahun 1989, C++ diperbaharui lagi. Pada tahun 1990, The Annotated C++ dirilis sebagai referensi yang lebih lengkap. Kemudian, Borland International merilis *compiler* Borland C++ dan Turbo C++ yang digunakan untuk mengkompilasi kode C++. Selain Borland, ada beberapa perusahaan lain yang juga merilis *compiler* C++ seperti, Topspeed C++ dan Zortech C++.

Pada tahun 1998 standar internasional pertama untuk C++ diterbitkan sebagai ISO/IEC 14882 yang kemudian lebih dikenal dengan sebutah C++ 98. Pada tahun 2003 beberapa masalah yang dilaporkan pada C++ 98 ditanggapi dan dilakukan revisi kembali atas C++, sehingga standar diperbaharui kembali menjadi C++ 03. Pada tahun 2005 laporan teknis yang disebut sebagai TR1, dirilis. Di dalam TR1, berbagai fitur tambahan yang direncanakan akan ditambahkan pada C++ diperinci. Pada pertengahan tahun 2011, standar baru C++ yang disebut dengan C++ 11 telah selesai.

Struktur sintaks C++ [12] hampir sama dengan bahasa C, yang berbeda hanya beberapa fungsi dan *keyword*. Sintaks C++ terdiri atas sejumlah blok fungsi. Masing-masing fungsi terdiri dari satu atau beberapa baris pernyataan yang akan di-*compile* untuk melakukan tugas tertentu. Secara umum program C++ tersusun atas sebagai berikut (Gambar 1):

1) Bagian pengarah *compiler* dan *header file*,
2) Bagian deklarasi,
3) Bagian definisi,
4) Bagian komentar yang ditandai dengan simbol "//" atau "/* . . . */",
5) Bagian pernyataan.

B. *Abstract Syntax Tree (AST)*

Abstract syntax tree (AST) [10] adalah sebuah representasi struktur sebuah *source code* dengan bahasa pemrograman tertentu. AST mewakili struktur data hirarkis yang berguna untuk proses analisis dan terjemahan. Untuk membangun sebuah *abstract syntax tree* dari suatu berkas *source code* bahasa tertentu diperlukan aturan tata bahasa khusus yang disebut dengan *grammar*. *Grammar* dapat dimodifikasi untuk memprediksi *code* secara sintaksis dan *semantic* sesuai dengan kebutuhan. Prediksi sintaksis merupakan penyesuaian *input* dengan *grammar*, sedangkan prediksi *semantic* digunakan untuk membangun representasi dalam bentuk struktur *tree*.

Bentuk *tree* dari hasil *parser* berkas *source code* dengan *grammar* menunjukkan struktur berkas tersebut. Gambar 2 merupakan contoh berkas *source code* berbahasa C++. Gambar 3 merupakan contoh representasi *source code* dalam bentuk AST yang dibentuk menggunakan *plugin* ANTLR (*Another Tool for Language Recognition*) berdasarkan *grammar* C++ pada ANTLR. *Node-node* yang dihasilkan *parser grammar* akan diambil untuk kemudian diproses lebih lanjut. Namun, hasil konversi struktur *source code* yang akan dirancang akan berbeda dengan hasil membangkitkan *tree* yang dilakukan oleh ANTLR *tools*. Hal tersebut terjadi karena *grammar* yang digunakan berbeda walaupun dalam proses perancangannya tetap berdasarkan pada *grammar* ANTLR. *Grammar* yang dirancang disesuaikan dengan data *sampling* yang telah dikumpulkan.

C. *Damerau-Levenshtein Distance Algorithm*

Damerau-Levenshtein Distance Algorithm [10,11] merupakan algoritme yang dikembangkan lebih lanjut berdasarkan *Levenshtein Distance Algorithm*. *Damerau-Levenshtein Distance* memiliki perhitungan biaya minimum untuk membuat dua buah *string* menjadi sama. Operasi yang digunakan dalam *Damerau-Levenshtein Distance Algorithm* sama dengan operasi yang digunakan pada *Levenshtein Distance*, yaitu operasi *insertion*, *deletion*, dan *substitution*. Operasi *transposition* ditambahkan di antara dua karakter atau *string* kemudian dibandingkan dengan sebelumnya yang belum ada. Misalnya, dua buah kata yang dinotasikan sebagai *source* (s) dan *target* (t). Variabel *m* dan *n* menyatakan panjang masing-masing kata. Variabel *i* dan *j* menyatakan posisi yang sedang diperiksa.

Aturan matriks pada *Damerau-Levenshtein Distance* adalah bahwa matriks *D*(*i*, *j*) adalah hasil dari nilai minimum antara nilai matriks *D*(*i*-1, *j*-1), *D*(*i*-1, *j*) dan *D*(*i*, *j*-1). Apabila elemen pada *sequence* antara kedua *source code* sama, maka matriks *D*(*i*-1,*j*-1) ditambah dengan nilai *cost* 0 dan 1 untuk kebalikannya. Matriks *D*(*i*-1, *j*) dan *D*(*i*, *j*-1) selalu ditambah dengan nilai 1. Jika posisi *i* dan *j* lebih besar dari 1 dan kata *source* ke-*i* sama dengan kata target ke-(*j*-1), serta pada kata *source* ke-(*j*-1) sama dengan kata target ke-*i*, maka *D*(*i*, *j*) akan mengambil nilai minimum antara dirinya dengan nilai pada matriks *D*(*i*-2, *j*-2). *Persamaan Damerau-Levenshtein Distance* tersebut dapat dilihat pada Gambar 4.

Rumus *similarity* yang digunakan adalah berikut ini:

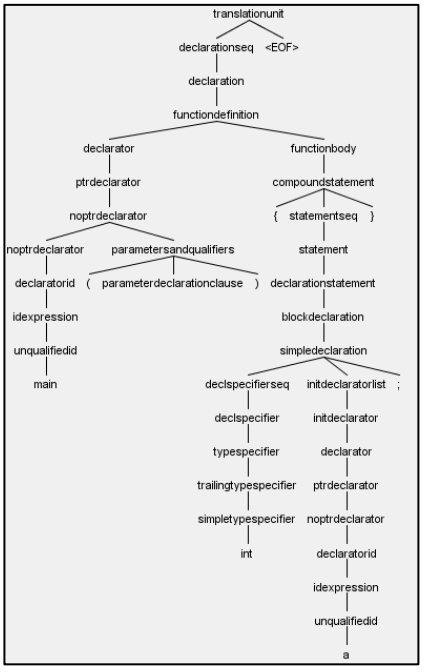
$$similarity = \left(1 - \frac{D[m,n]}{\max(m,n)}\right) \times 100\% \quad (1)$$

```
#include <stdio.h>                  //file header
#include <conio.h>                  //file header

Int main()                          //deklarasi main program
{                                   //blok pembuka
      int l;                        //pernyataan
      printf("Hello World!");       //definisi
      /*program selesai*/           //komentar
      return 0;                     //definisi pengembalian
}                                   //blok penutup
```

Gambar 1 Struktur Sintaks C++

```
#include <stdio.h>
#include <conio.h>
main()
{
      int a;
}
```

Gambar 2 Berkas *Source Code* C++



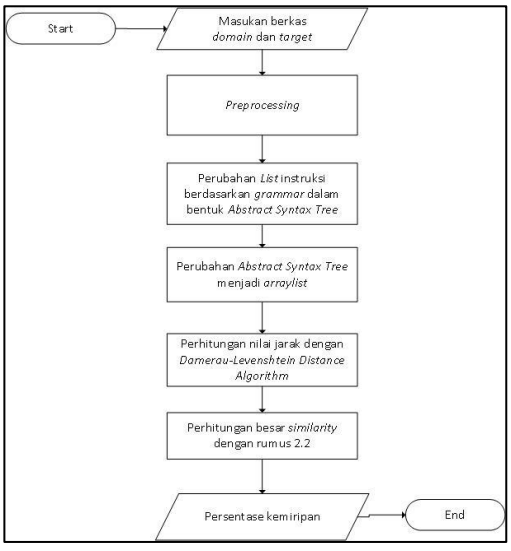
Gambar 3 Bentuk *abstract syntax tree* dari *Source Code* C++ pada Gambar 2 dengan ANTLR

III. HASIL DAN PEMBAHASAN

Proses dalam metode utama dalam sistem dimulai dari pemasukan berkas domain dan target seperti pada Gambar 5. Kedua berkas akan mengalami tahap *preprocessing*. Setiap instruksi yang telah melalui tahap *preprocessing* akan diubah berdasarkan *grammar lexer* dan *parser rule* yang telah dimodifikasi sesuai kebutuhan. Hasil perubahan tersebut akan disimpan dalam bentuk *object tree*. Kemudian, dari masing-masing *tree* akan dibentuk daftar yang akan digunakan untuk membangun matriks perhitungan. Matriks ini dibentuk untuk mencari jarak antar kedua berkas menggunakan *algoritme Damerau-Levenshtein Distance* Setelah jarak didapatkan, perhitungan persentase kemiripan akan dilakukan bedasarkan rumus *similarity*. Hasil besarnya nilai persentase kemiripan akan berpengaruh dalam penarikan kesimpulan apakah berkas target plagiat terhadap berkas domain atau tidak.

```
FUNCTION DamerauLevenshteinDistance(String S1,String S2)
        FOR i = 0 to size of String S1
              M[i][0] = i
        ENDFOR
        FOR j = 0 to size of String S2
              M[0][j] = j
        ENDFOR
        FOR i = 1 to size of String S1
              FOR j = 1 to size of String S2
                    IF (S1[i] == S2[j]) THEN cost = 0
                    ELSE cost = 1
                    ENDIF

                    Deletion = M[i-1][j]+1 ;
                    Insert = M[i][j-1]+1 ;
                    Substitution = M[i-1][j-1]+cost

                    M[i][j] = min(Deletion,Insert ,Substitution)

                    IF(i>1 AND j>1 AND S1[i] == S2[j-1] && S1[i-1] ==
                    S2[j])THEN
                    M[i][j] = min(M[i][j], M[i-2][j-2]+cost)

                    return M[i][j]

              ENDFOR

        ENDFOR
```

Gambar 4 Algoritme *Damerau-Levenshtein Distance*



Gambar 5 *Flowchart* global

Pada tahap ini akan dijelaskan hasil pengujian yang dilakukan. *Threshold* sebesar 88,2314478 % digunakan untuk pengujian *grammar lexer* dan *parser rule. Threshold* sebesar 83,1097034 % digunakan untuk pengujian *grammar lexer rule* saja.

A. Pengujian Grammar Parser dan Lexer Rule.

Dalam pengujian ini akan dilakukan perbandingan antar berkas *source code* secara *general* dengan menggunakan *preprocessing* dan *grammar* yang terdiri dari *parser* dan *lexer rule.* Hasil pengujian dapat dilihat pada Tabel I.

Pengujian *grammar parser* dan *lexer rule* menghasilkan akurasi sebesar 97,3076923 %. Hal ini menyebabkan terdapat perbandingan yang bernilai *false positive* adalah kemiripan struktur pada data uji *domain* yang sebenarnya bukan plagiat. Sedangkan yang menyebabkan terdapat perbandingan yang bernilai *true negative* adalah perbendaan struktur *source code* walau dalam hal arti, keduanya mirip. Waktu yang diperlukan untuk memeriksa data uji dengan *grammar parser* dan *lexer rule* adalah 118,115 detik

B. Pengujian Grammar Lexer Rule.

Dalam pengujian ini, akan dilakukan perbandingan antar berkas *source code* secara *general* dengan menggunakan *preprocessing* dan *grammar* yang terdiri dari *lexer rule* saja. Hasil pengujian dapat dilihat pada Tabel II.

Pengujian *grammar lexer rule* menghasilkan akurasi sebesar 96,923076 %. Hal yang menyebabkan terdapat perbandingan yang bernilai *false positive* adalah kemiripan struktur pada data uji *domain* yang sebenarnya bukan plagiat. Penyebab terdapat perbandingan yang bernilai *true negative* adalah perbendaan struktur *source code* walau dalam hal arti, keduanya mirip. Dengan memakai hanya *grammar lexer rule* saja, waktu yang dibutuhkan untuk memeriksa data uji adalah 6,057 detik.

C. Pengujian Preprocessing

Pada pengujian ini, akan dilakukan perbandingan antar berkas *source code* dengan menguji kombinasi *preprocessing* yang ada pada aplikasi. Pengujian menggunakan grammar *parse*r dan *lexer rule*. Terdapat 4 macam *preprocessin*g, yaitu *preprocessing* deklarasi main *program*, *file header*, *comment* dan normalisasi. Hasil pengujian dapat dilihat pada Tabel III.

TABEL I
CONFUSION MATRIX PENGUJIAN *GRAMMAR PARSER* DAN *LEXER RULE*

	Positive	***Negative***
True	15	5
False	16	744

TABEL II
CONFUSION MATRIX PENGUJIAN *GRAMMAR LEXER RULE*

	Positive	***Negative***
True	16	4
False	20	740

TABEL III
HASIL PENGUJIAN PREPROCESSING

No.	*Preprocessing* yang digunakan				Akurasi
	Deklarasi *main()*	*File Header*	*Comment*	Normalisasi	
1	V	V	V	V	97,3076923 %
2	V	-	V	V	97,3076923 %
3	V	V	-	V	97,0512820 %
4	-	V	V	V	97,3076923 %
5	V	-	-	V	97,05128205%
6	-	V	-	V	97,05128205%
7	-	-	V	V	97,3076923 %
8	-	-	-	V	97,05128205%
9	V	V	V	-	97,05128205%
10	V	-	V	-	97,17948717%
11	V	V	-	-	96,92307692%
12	-	V	V	-	97,17948717%
13	V	-	-	-	96,92307692%
14	-	V	-	-	96,92307692%
15	-	-	V	-	97,3076923 %
16	-	-	-	-	97,05128205%

Berdasarkan Tabel III, dapat dilihat bahwa kombinasi *preprocessing* memberikan hasil akurasi bervariasi. Akurasi tertinggi dihasilkan oleh kombinasi *preprocessing* nomor 1, 2, 4, 7, dan 15 sebesar 97,3076923%. Akurasi tertinggi yang dihasilkan dengan penggunaan kombinasi *preprocessing* terbanyak adalah nomor 1, sedangkan akurasi tertinggi yang dihasilkan dengan penggunaan kombinasi *preprocessing* paling sedikit adalah nomor 15. Pada kombinasi *preprocessing* nomor 15, *preprocessing* yang digunakan hanya untuk bagian *comment* saja. Namun, jika berkas *source code* plagiat menerapkan pergantian struktur *source code* yang masih mempunyai arti yang serupa, tingkat kemiripan yang dihasilkan akan lebih rendah dibandingkan dengan kombinasi *preprocessing* yang menggunakan normalisasi. Proses normalisasi membuat instruksi dalam berkas *source code* tertentu menjadi seragam untuk meningkatkan persentase kemiripan kedua berkas.

D. Pengujian Nilai Jarak

Pada pengujian ini dilakukan variasi *value* pada nilai jarak *Damerau-Levenshtein Distance*. Nilai jarak yang diuji adalah 1, 4, dan 10. Nilai-nilai tersebut diambil secara acak untuk menguji nilai jarak yang menghasilkan akurasi tertinggi. Nilai *threshold* yang digunakan menyesuaikan dengan rata-rata yang dihasilkan menggunakan nilai jarak tertentu (Tabel IV).

Distance Algorithm yang menghasilkan akurasi sebesar 97,3076923 %. Namun, semakin besar nilai jarak tidak berarti semakin besar pula akurasi yang dihasilkan. Hal tersebut dilihat saat menggunakan nilai jarak 10 dan 15. Tingkat akurasi yang dihasilkan cenderung menurun.

Berdasarkan hasil pengujian yang dilakukan, *grammar* yang memberikan akurasi tertinggi adalah *grammar parser* dan *lexer rule.* Hal ini disebabkan bukan hanya mengubah instruksi *source code* secara *lexical* saja, namun meng-abstraksi artinya juga lewat *grammar parser rule.* Tidak semua kombinasi *preprocessing* yang digunakan menambah

TABEL IV
HASIL PENGUJIAN NILAI JARAK

No.	Nilai Jarak	Akurasi
1	1	97,3076923 %
2	4	97.43589743 %
3	10	97.3076923 %

akurasi deteksi plagiarisme *source code.* Namun, proses normalisasi dibutuhkan sebagai standar bentuk struktur instruksi *source code.* Jika nilai jarak pada *Damerau-Levenshtein Distance Algorithm* semakin besar, tidak berarti akurasi pun semakin besar. Pada suatu titik, akurasi akan menurun. Panjang berkas *source code* yang diperiksa berpengaruh dalam menentukan nilai jarak minimum dan persentase kemiripan. Hasil pengujian yang bernilai *true negative* disebabkan oleh struktur sintaks dari kedua berkas berbeda. Hasil pengujian yang bernilai *false positif* disebabkan struktur sintaks kedua berkas mirip, walaupun mungkin kenyataannya kedua berkas tidak plagiat.

IV. KESIMPULAN

Hasil akurasi metode *abstract syntax tree* dan *Damerau-Levenshtein Distance Algorithm* menggunakan *grammar parser* dan *lexer rule* pada deteksi plagiarisme adalah sebesar 97,3076923 %. Waktu komputasi yang dibutuhkan aplikasi adalah sebesar 118,115 detik. Hasil akurasi metode *abstract syntax tree* dan *Damerau-Levenshtein Distance Algorithm* menggunakan *grammar lexer rule* saja pada deteksi plagiarisme adalah sebesar 96,923076 %. Waktu komputasi yang dibutuhkan aplikasi adalah sebesar 6,057 detik.

Pada pengujian *preprocessing*, macam-macam kombinasi penggunaan *preprocessing* memberikan tingkat akurasi yang berbeda-beda. Tingkat akurasi tertinggi yang dicapai adalah 97,3076923 %. Kombinasi terbanyak dengan akurasi tertinggi adalah kombinasi seluruh *preprocessing* yang ada, yaitu deklarasi *main program*, *comment*, *file header*, dan normalisasi. Kombinasi paling sedikit dengan akurasi tertinggi adalah *preprocessing comment* saja. Apabila tidak menggunakan normalisasi, kemungkinan dua berkas *source code* dengan struktur yang berbeda, namun memiliki kesamaan arti yang dikategorikan sebagai bukan plagiat, semakin besar.

Pada pengujian nilai jarak, akurasi tertinggi dihasilkan dengan menggunakan nilai jarak 4, yaitu sebesar 97,43589743%. Akurasi tersebut lebih besar daripada menggunakan nilai jarak *default Damerau-Levenshtein Distance Algorithm* yang menghasilkan akurasi sebesar 97,3076923 %. Namun, semakin besar nilai jarak tidak berarti semakin besar pula akurasi yang dihasilkan. Hal tersebut dilihat saat menggunakan nilai jarak 10 dan 15. Tingkat akurasi yang dihasilkan cenderung menurun.

DAFTAR REFERENSI

[1] Republik Indonesia. Undang-Undang No. 17 Tahun 2010 Bab 1 Pasal 1 ayat 1 tentang Ketentuan Umum. Lembaran Negara RI Tahun 2010. Sekretariat Negara, Jakarta, 2010.

[2] Republik Indonesia. 2010. Undang-Undang No. 17 Tahun 2010 Bab III Pasal 12 tentang Lingkup dan Pelaku. Lembaran Negara RI Tahun 2010. Sekretariat Negara, Jakarta.
[3] The Institute of Electrical and Electronics Engineers, "IEEE Standard Glossary of Software Engineering Terminology," IEEE Standards Board. 1990.
[4] B. Muddu, A. Asadullah, and V. Bhat. "CPDP: A Robust Technique for Plagiarism Detection in Source Code," *Proceeding IWSC '13 Proceedings of the 7th International Workshop on Software Clones*, 2013, pp. 39-45.
[5] Republik Indonesia. Undang-Undang No. 17 Tahun 2010 Bab VI Pasal 12 ayat 1 tentang Peraturan Menteri Pendidikan Nasional Republik Indonesia. Lembaran Negara RI Tahun 2010. Sekretariat Negara, Jakarta, 2010.
[6] J. Feng, B. Cui, and K. Xia, "A Code Comparison Algorithm Based on AST for Plagiarism Detection", *Fourth International Conference on Emerging Intelligent Dara and Web Technologies*, IEEE, 2013.
[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code", *IEEE Transactions on Software Engineering*, Vol. 32, No. 3. March, 2006.
[8] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection", *IEEE Transactions on Software Engineering*, vol. 43, Issue No. 12 - Dec, 2017, pp. 1157-1177.
[9] Omer; Missen, Saad, Tazeen, Tenvir and Moosa, M., "EPlag: *A Two Layer Source Code Plagiarism Detection System*", IEEE, 2013.
[10] A. Munif, R. J. Akbar, R. I. Tantra, dan R. Ilavi. "Rancang Bangun Sistem *E-Learning* Pemrograman pada Modul Deteksi Plagiarisme Kode Program dan *Student Feedback System*" *Jurnal Ilmiah Teknologi Informasi*, Vol. 15, No. 1, January, 2017.
[11] GitHub. "Antlr/Grammars-v4." Internet: https://github.com/antlr/grammarsv4 [Oct. 02, 2017].
[12] "Belajar C++ Team." Internet: http://www.belajarcpp.com/2016/01/sejarah-bahasa-pemrograman-cpp.hml [Oct. 03, 2017].

Stephanie Rusdianto, kelahiran kota Bandung, Jawa Barat, Indonesia. Menamatkan pendidikan S1 Teknik Informatika di Institut Teknologi Harapan Bangsa tahun 2018. Penulis memiliki minat tinggi dalam bidang perangkat lunak komputer.

Ria Chaniago, menyelesaikan pendidikan S1 pada tahun 2013 di Teknik Informatika, Institut Teknologi Harapan Bangsa, dengan fokus bidang Kecerdasan Buatan. Melanjutkan pendidikan S2 di Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, pada tahun 2014 dan menyelesaikannya pada tahun 2016. Memiliki minat dan pengalaman bekerja sebagai peneliti di bidang Mesin Pembelajaran dan Pemrosesan Bahasa Alami.