TUGAS AKHIR – KI141502

MODUL DETEKSI PLAGIARISME KODE PROGRAM PADA SISTEM E-LEARNING PEMROGRAMAN

RUCHI INTAN TANTRA
NRP 5112100015

Dosen Pembimbing I
Rizky Januar Akbar, S.Kom., M.Eng.

Dosen Pembimbing II
Abdul Munif, S.Kom., M.Sc.

Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2016

3

(Halaman ini sengaja dikosongkan)

UNDERGRADUATE THESES – KI141502

CODE PLAGIARISM DETECTION MODULE ON E-LEARNING FOR PROGRAMMING

RUCHI INTAN TANTRA
NRP 5112100015

First Advisor
Rizky Januar Akbar, S.Kom., M.Eng.

Second Advisor
Abdul Munif, S.Kom., M.Sc.

Department of Informatics
Faculty of Information Technology
Sepuluh Nopember Institute of Technology
Surabaya 2016

3

(Halaman ini sengaja dikosongkan)

LEMBAR PENGESAHAN

MODUL DETEKSI PLAGIARISME KODE PROGRAM PADA SISTEM E-LEARNING PEMROGRAMAN

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Algoritma Pemrograman
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
RUCHI INTAN TANTRA
NRP: 5112100015

Disetujui oleh Pembimbing Tugas Akhir:

1. Rizky Januar Akbar, S.Kom., M.Kom. .....
   (NIP. 198701032014041001) (Pembimbing 1)

2. Abdul Munif, S.Kom., M.Sc. .....
   (NIP. 198608232015041004) (Pembimbing 2)

SURABAYA
JUNI, 2016

3

(Halaman ini sengaja dikosongkan)

MODUL DETEKSI PLAGIARISME KODE PROGRAM PADA SISTEM E-LEARNING PEMROGRAMAN

Nama Mahasiswa	**: RUCHI INTAN TANTRA**
NRP	**: 5112100015**
Jurusan	**: Teknik Informatika FTIF-ITS**
Dosen Pembimbing 1	**: Rizky Januar Akbar, S.Kom., M.Eng.**
Dosen Pembimbing 2	**: Abdul Munif, S.Kom., M.Sc.**

Abstrak

Plagiarisme kode program antara mahasiswa merupakan hal yang tidak asing lagi saat ini. Plagiarisme kode program dapat terjadi dengan cara menyalin kode program milik mahasiswa yang telah berhasil menyelesaikan tugas yang diberikan oleh dosen. Oleh karena itu dibuat sebuah sistem yang dapat mendeteksi plagiarisme kode program antar mahasiswa lalu mengklasifikasikannya berdasarkan tingkat kemiripannya.

Tugas akhir ini mengimplementasikan kakas bantu ANTLR sebagai parser kode program dan fungsi Listener sebagai tree-walker dari Abstract Syntax Tree (AST) yang dihasilkan ANTLR parser. Metode Levensthein distance dan modifikasi similarity adalah metode yang digunakan untuk menghitung nilai kemiripan antar program. Metode hierarchical clustering digunakan untuk mengelompokkan kode program.

Pada tugas akhir ini, hasil yang didapat untuk penghitungan nilai kemiripan antar kode program menggunakan modifikasi similarity lebih efektif dibandingkan metode Levensthein distance. Metode hierarchical clustering dianggap efektif dalam mengelompokkan kode program berdasarkan tingkat kemiripannya.

Kata kunci: plagiarisme, ANTLR, Listener, AST, Levensthein distance, Hierarchical clustering.

CODE PLAGIARISM DETECTION MODULE ON E-LEARNING FOR PROGRAMMING

Student's Name : RUCHI INTAN TANTRA
Student's ID : 5112100015
Department : Teknik Informatika FTIF-ITS
First Advisor : Rizky Januar Akbar, S.Kom., M.Eng.
Second Advisor : Abdul Munif, S.Kom., M.Sc.

Abstract

The practice of plagiarism is not a strange thing anymore, especially among the students that almost every day working on tasks assigned by the lecturer. The practice of plagiarism is done by the exchange of source code that have been successful. Therefore developing system to detect plagiarism of program code among students and classify them based on the degree of similarity became very important.

This study implements ANTLR tools as a code parser and Listener method as a tree-walker that visiting nodes of Abstract Syntax Tree. Levenshtein distance method and similarity modification is used to calculate the value of similarity between two programs. Hierarchical clustering method is used to classify the program code based on the degree of similarity.

In this undergraduate thesis, the results obtained using modified similarity was the effective way compared to Levenshtein distance method, to calculate the value of similarity. Hierarchical clustering method is considered effective in classifying the program code based on the degree of similarity.

Keywords : Plagiarism, ANTLR, Listener, AST, levensthein distance, hierarchical clustering.

KATA PENGANTAR

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

Alhamdulillahirabbil'alamin, segala puji bagi Allah Swt yang telah melimpahkan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul:

"Modul Deteksi Plagiarisme Kode Program pada E-Learning Pemrograman"

yang merupakan salah satu syarat dalam menempuh ujian sidang guna memperoleh gelar Sarjana Komputer. Selesainya Tugas Akhir ini tidak terlepas dari bantuan dan dukungan beberapa pihak, sehingga pada kesempatan ini penulis mengucapkan terima kasih kepada :

1. Allah SWT karena atas berkah dan rahmatNya penulis dapat menyelesaikan Tugas Akhir ini.
2. Bapak Singga Tantra dan Ibu Ninik Nadiarni selaku orang tua penulis yang selalu memberikan dukungan doa, moral, dan material yang tak terhingga kepada penulis sehingga penulis dapat menyelesaikan Tugas Akhir ini.
3. Bapak Rizky Januar Akbar, S.Kom., M.Eng. dan Bapak Abdul Munif, S.Kom., M.Sc. selaku pembimbing I dan II yang telah membimbing dan memberikan motivasi, nasehat dan bimbingan dalam menyelesaikan Tugas Akhir ini.
4. Ibu Dr.Ir. Siti Rochimah, M.T. selaku dosen wali penulis yang telah memberikan arahan, masukan dan motivasi kepada penulis.
5. Bapak Darlis Herumurti, S.Kom., M.Kom. selaku kepala jurusan Teknik Informatika ITS dan segenap dosen dan karyawan Teknik Informatika ITS yang telah memberikan ilmu dan pengalaman kepada penulis selama menjalani masa studi di ITS.

6. Sahabat dan keluarga penulis yang tidak dapat disebutkan satu per satu yang selalu membantu, menghibur, menjadi tempat bertukar ilmu dan berjuang bersama-sama penulis.

Penulis menyadari bahwa Tugas Akhir ini masih memiliki banyak kekurangan sehingga dengan kerendahan hati penulis mengharapkan kritik dan saran dari pembaca untuk perbaikan ke depan.

Surabaya, Juni 2016

DAFTAR ISI

DAFTAR GAMBAR

3

(Halaman ini sengaja dikosongkan)

DAFTAR TABEL

DAFTAR KODE SUMBER

BAB I
PENDAHULUAN

1.1 Latar Belakang

Perkembangan teknologi informasi tidak hanya memberikan kemudahan dalam pengaksesan informasi, tetapi juga menjadikan komputer sebagai alat bantu mutlak dalam pengolahan data sehingga memberi nilai tambah bagi pengguna. Salah satu ilmu komputer yang berkembang saat ini adalah pengimplementasian sebuah Sistem *E-Learning Pemrograman* yang digunakan sebagai pendukung proses pembelajaran mahasiswa Teknik Informatika. *E-learning* sendiri merupakan sebuah inovasi baru dalam proses belajar mengajar jarak jauh dengan menggunakan media elektronik khususnya internet sebagai sistem pembelajarannya. Sistem *E-Learning Pemrograman* ini digunakan mahasiswa sebagai sarana pembelajaran dan pengumpulan tugas yang diberikan oleh dosen. Sedangkan bagi dosen, sistem *E-Learning Pemrograman* digunakan untuk memantau dan mengevaluasi perkembangan belajar mahasiswa.

Perkembangan pesat pada dunia teknologi dan informasi memungkinkan mahasiswa untuk memperoleh informasi secara bebas dalam proses pengerjaan tugas *coding* yang diberikan oleh dosen. Selain itu mahasiswa sering melakukan *copy-paste* terhadap kode program mahasiswa lain yang telah berhasil mengerjakan tugas *coding* yang diberikan dosen. Sehingga tidak menutup kemungkinan hasil pengerjaan tugas mahasiswa satu dengan yang lainnya sama atau mungkin terjadi tindak plagiarisme kode program antar mahasiswa Salah satu fitur pada sistem *E-Learning Pemrograman* ini adalah mendeteksi plagiarisme kode program yang dilakukan oleh mahasiswa dalam satu kelas. Kode program tersebut didapat dari pengumpulan tugas dan ujian mahasiswa.

Dalam perkembangan dunia teknologi informasi saat ini telah banyak yang mengimplementasikan beberapa metode untuk

mendeteksi kemiripan antar kode program satu dengan kode program lainnya. Salah satu metode yang digunakan adalah *Levenshtein distance* dan *Hierarchical clustering*.

Sistem *E-Learning Pemrograman* pada fitur deteksi plagiarisme kode program ini diharapkan mampu mendeteksi plagiarisme kode program yang dilakukan oleh mahasiswa dalam satu kelas dan mengklasifikasikannya berdasarkan tingkat kemiripan antar kode program. Sehingga dosen dapat mengetahui mahasiswa mana saja yang memiliki tingkat kemiripan kode program yang sama dalam satu kelas.

1.2 Rumusan Masalah

Tugas akhir ini mengangkat beberapa rumusan masalah sebagai berikut:

1. Membangun *back-end* sistem *E-Learning pemrograman* untuk mendeteksi plagiarisme kode program antar mahasiswa dalam satu kelas.
2. Membangun *back-end* sistem *E-Learning pemrograman* untuk mengelompokkan atau mengklasifikasikan kode program berdasarkan tingkat kemiripan kode program antar mahasiswa dalam satu kelas.

1.3 Batasan Permasalahan

Permasalahan yang dibahas pada tugas akhir ini memiliki batasan sebagai berikut:

1. Aplikasi *back-end* pengecekan dan pengelompokan plagiarisme kode program dibuat menggunakan bahasa pemrograman Java.
2. Dataset yang digunakan adalah *source code* mahasiswa pada kelas Pemrograman Berorientasi Objek.
3. Dataset menggunakan bahasa C++ dan difokuskan pada paradigma Pemrograman Berorientasi Objek.
4. Bentuk *Sequence Tree* untuk setiap kode sumber dibuat dengan memodifikasi fungsi Listener pada ANTLR *parser*.

5. Metode yang digunakan untuk menghitung kemiripan kode sumber adalah *Levenshtein distance* dan modifikasi *similarity*.
6. Metode klasifikasi yang digunakan adalah metode *Hierarchical clustering*.
7. Aplikasi ditujukan untuk dosen dan mahasiswa pemrograman yang ada di Teknik Informatika.
8. Perangkat lunak yang digunakan adalah *Eclipse Luna*, ANTLR *plugin,* dan database PostgreSQL.

1.4 Tujuan

Tujuan dari tugas akhir ini adalah sebagai berikut:

1. Mendeteksi plagiarisme kode program antar mahasiswa dalam satu kelas yang difokuskan pada kemiripan struktur programnya.
2. Mengklasifikasikan kode program berdasarkan tingkat kemiripan kode program antar mahasiswa dalam satu kelas.

1.5 Manfaat

Dengan dibuatnya tugas akhir ini diharapkan dapat memberikan manfaat sebagai berikut:

1. Mempermudah dosen dalam mengetahui plagiarisme yang terjadi antar mahasiswa dalam satu kelas.
2. Membuat *repository* segmen kode program yang mirip yang dapat digunakan untuk mempercepat pengecekan kemiripan kode program.
3. Bagi penulis, tugas akhir ini bermanfaat sebagai sarana untuk mengimplementasikan ilmu dan algoritma pemrograman yang telah dipelajari selama kuliah agar berguna bagi masyarakat terutama mendukung sistem belajar mengajar dalam perkuliahan di bidang teknologi.

1.6 Metodologi

Pembuatan tugas akhir ini dilakukan dengan menggunakan metodologi sebagai berikut:

1.6.1 Penyusunan Proposal

Tahap awal untuk memulai pengerjaan tugas akhir adalah penyusunan proposal tugas akhir. Pada proposal ini, penulis mengajukan gagasan untuk menyelesaikan permasalahan mendeteksi plagiarisme kode program antar mahasiswa dalam satu kelas dan mengklasifikasikan berdasarkan tingkat kemiripannya.

1.6.2 Studi Literatur

Pada Tahap kedua adalah mencari informasi dan studi literatur yang relevan untuk dijadikan referensi dalam melakukan pengerjaan tugas akhir. Pada studi literatur ini, akan dipelajari sejumlah referensi yang diperlukan dalam pembuatan aplikasi yaitu mengenai *parsing* kode program menggunakan ANTLR menjadi AST, mendefinisikan bentuk AST hasil *parsing* tersebut ke dalam bahasa pemrograman, algoritma untuk membentuk AST ke dalam bentuk *sequence* tanpa mengubah struktur program, metode *Levenshtein distance* untuk menghitung tingkat kemiripan antar kode program, dan metode *Hierarchical clustering* untuk mengelompokkan kode program berdasarkan tingkat kemiripannya.

1.6.3 Implementasi Perangkat Lunak

Implementasi merupakan tahap untuk membangun metode-metode yang sudah diajukan pada proposal Tugas akhir. Untuk membangun aplikasi *back-end* pada modul *plagiarism detection system* dan *source code classification* ini menggunakan bahasa pemrograman java yang diimplementasikan menggunakan *Eclipse luna* sebagai IDE, *ANTLR plugin* sebagai kakas bantu untuk memparsing kode program menjadi bentuk AST, dan *Object*

Relational Database Management System PostgreSQL untuk menyimpan data *Sequence* hasil *parsing* tiap kode program dan derajat kemiripan antar kode program.

1.6.4 Pengujian dan Evaluasi

Pengujian akan dilakukan dengan menggunakan studi kasus mata kuliah PBO (Pemrograman Berorientasi Objek). Tahap pengujian pada aplikasi dilakukan dengan melakukan uji coba dataset jawaban mahasiswa berupa kode program C++ yang akan dicocokkan dengan kode program mahasiswa lain dalam satu kelas. Dataset jawaban yang digunakan merupakan data tugas kuliah mata kuliah PBO (Pemrograman Berorientasi Objek).

Evaluasi fitur pencocokan kesamaan jawaban antar mahasiswa dilakukan dengan menghitung akurasi yang dihasilkan dibandingkan dengan *ground truth* yang dibuat. *Ground truth* dibuat dengan melakukan penilaian manual kesamaan kode program satu sama lain. Kesimpulan akhir diambil dengan memperhatikan nilai akurasi.

1.6.5 Penyusunan Buku

Pada tahap ini disusun buku sebagai dokumentasi dari pelaksanaan tugas akhir yang mencangkup seluruh konsep, teori, implementasi, serta hasil yang telah dikerjakan.

1.7 Sistematika Penulisan Laporan

Sistematika penulisan laporan tugas akhir adalah sebagai berikut:

1. Bab I. Pendahuluan
   Bab ini berisikan penjelasan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi, dan sistematika penulisan dari pembuatan tugas akhir.
2. Bab II. Tinjauan Pustaka

Bab ini berisi kajian teori dari metode dan algoritma yang digunakan dalam penyusunan Tugas Akhir ini. Secara garis besar, bab ini berisi tentang ANTLR *parser*, AST (*Abstract Syntax Tree*), *Levensthein distance,* dan *Hierarchical clustering.*

3. Bab III. Perancangan Perangkat Lunak
   Bab ini berisi pembahasan mengenai perancangan bentuk *sequence* kode program (hasil *parsing* dari ANTLR), perancangan metode *Levenshtein distance*, modifikasi *similarity*, dan klasifikasi kode program menggunakan metode *Hierarchical clustering*.
4. Bab IV. Implementasi
   Bab ini menjelaskan implementasi yang berbentuk kode sumber dari algoritma pembentukan *sequence* kode program yang merupakan hasil *parsing* dari ANTLR, metode *Levensthein distance* dan modifikasi *similarity* untuk menghitung tingkat kemiripan kode program, dan metode *Hierarchical clustering* untuk klasifikasi.
5. Bab V. Hasil Uji Coba dan Evaluasi
   Bab ini berisikan hasil uji coba mendeteksi plagiarisme antar kode program dalam satu kelas berdasarkan kemiripan struktur programnya dan mengelompokannya menggunakan *Hierarchical clustering.* Uji coba dilakukan dengan menilai secara manual derajat kemiripan antar program satu dengan program lainnya dalam satu kelas menggunakan suatu penetapan *ground truth,* lalu menilai apakah hasil pengelompokan kode program yang mirip telah sesuai.
6. Bab VI. Kesimpulan dan Saran
   Bab ini merupakan bab yang menyampaikan kesimpulan dari hasil uji coba yang dilakukan, masalah-masalah yang dialami pada proses pengerjaan tugas akhir, dan saran untuk pengembangan solusi ke depannya.
7. Daftar Pustaka
   Bab ini berisi daftar pustaka yang dijadikan literatur dalam tugas akhir.

8. Lampiran
   Dalam lampiran terdapat tabel-tabel data hasil uji coba dan kode sumber program secara keseluruhan.

(Halaman ini sengaja dikosongkan)

BAB II
TINJAUAN PUSTAKA

Bab ini berisi *pembahasan* mengenai teori-teori dasar dan kakas bantu yang digunakan dalam tugas akhir. Teori-teori dan kakas bantu tersebut diantaranya adalah ANTLR *parser*, *Levenshtein distance, Hierarchical clustering,* dan beberapa teori serta kakas bantu lain yang mendukung pembuatan tugas akhir.

2.1 AST (Abstract Syntax Tree)

Abstract Syntax Tree merupakan sebuah pohon yang merepresentasikan struktur dari sebuah kode program dengan bahasa pemrograman tertentu. Analisis sintaksis dari kode sumber biasanya memerlukan transformasi urutan linier dari token menjadi pohon sintaks hirarkis. Contoh implementasi AST adalah pada kode program *if-else* dapat dibentuk menjadi 3 *node* [1].

Pada sistem *E-Learning Pemrograman*, AST diperoleh dari hasil *tree parser* oleh ANTLR. Setelah mahasiswa mengirim kode program melalui aplikasi, ANTLR mengubah secara otomatis kode program tersebut ke dalam bentuk pohon yang disebut dengan *Abstract Syntax Tree*. Bentuk AST inilah yang akan diproses lebih lanjut untuk menghitung tingkat kemiripan tiap kode program.

2.2 ANTLR

ANTLR mengambil sebuah input berupa *grammar* file dengan bahasa pemrograman tertentu dan menerjemahkannya menjadi kode program sesuai dengan Bahasa pemrograman tersebut. Versi 3 dari ANTLR dapat menghasilkan kode dalam bahasa pemrograman Ada95, ActionScript, C, C #, Java, JavaScript, Objective-C, Perl, Python, Ruby, dan Standard ML.

ANTLR dapat menghasilkan *lexer*, *parser*, *parser* pohon, dan kombinasi *lexer-parser*. Parser otomatis dapat menghasilkan *Abstract Syntax Tree* (AST) yang dapat diproses lebih lanjut

dengan *tree parser*. ANTLR menyediakan notasi tunggal yang konsisten untuk menentukan *lexer*, *parser*, dan *parser* pohon [2].

ANTLR memiliki fungsi listener dan visitor sebagai sebuah metode *tree-walking*. Fungsi listener lebih mudah dipakai tetapi kurang fleksibel karena tidak dapat mengubah alur eksekusi. Sedangkan fungsi visitor dapat mengatur atau mengubah suatu alur eksekusi sesuai dengan yang kita inginkan. Secara umum listener lebih tepat dipakai untuk membaca sebuah *file source* dengan alur yang pasti. Fungsi listener memiliki 2 buah metode yaitu `enter` dan `exit`. Sedangkan visitor hanya memiliki satu metode yaitu `visit` [3].

Seperti yang dijelaskan sebelumnya ANTLR adalah sebuah *lexer* dan *parser* generator dengan input berupa file *grammar*. Oleh karena itu input file *grammar* untuk kode program dengan bahasa C++ telah disediakan oleh ANTLR [4].

2.3 Levenshtein Distance

Levenshtein distance digunakan untuk mengukur nilai kesamaan atau kemiripan antara dua buah kata (string). Jarak *Levenshtein* diperoleh dengan mencari cara termudah untuk mengubah suatu string.

Secara umum, operasi mengubah yang diperbolehkan untuk keperluan ini adalah [5]:

- memasukkan karakter ke dalam string (*insertions*)
- menghapus sebuah karakter dari suatu string (*deletions*)
- mengganti karakter string dengan karakter lain. (*subtitutions*)

Persamaan cara menghitung kemiripan dua buah string dapat dilihat pada Persamaan 2.1. $lev_{a,b}(i,j)$ adalah nilai jarak (kemiripan) antara karakter ke-i pada kata **a** dan karakter ke-j pada kata **b**. Sedangkan $1_{(a_i \neq b_j)}$ merupakan indikator fungsi yang bernilai 0 apabila $a_i = b_j$ dan bernilai 1 apabila sebaliknya.

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & if \min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & otherwise. \end{cases} \quad (2.1)$$

Contoh penggunaan metode *Levenshtein distance* untuk mendeteksi kemiripan dua buah string [6]:

Nilai *Levenshtein distance* antara dua kata “kitten” dan “sitting” adalah 3.
Berikut penjelasannya:

1. **k**itten → **s**itten (substitusi "s" untuk "k")
2. sitt**e**n → sitt**i**n (substitusi "i" untuk "e")
3. sittin → sittin**g** (penambahan "g" pada akhir kata).

Contoh implementasi pseudocode [7] :

```
1   FUNCTION LevenshteinDistance(String
    S1,String S2)
2   initialize M as an empty Array[][]
3   M[0][0] = 0
4   FOR i = 1 to size of String S1
5       M[i][0] = i
6   FOR j = 1 to size of String S2
7       M[0][j] = j
8   FOR i = 1 to size of String S1
9       FOR j = 1 to size of String S2
10          IF S1[i] = S2[j]
11              cost = 0
12          ELSE
13              cost = 1
14          M[i][j] = min( M[i-1][j-1]+cost ,
            M[i-1][j]+1 , M[i][j-1]+1 )
15          return M[i][j]
```

Kode Sumber 2.1 Contoh Implementasi pseudocode metode Levensthein distance

2.4 Hierarchical clustering

Hierarchical Clustering adalah salah satu algoritma *clustering* yang dapat digunakan untuk meng-*cluster* dokumen *(document clustering*). Dari teknik *hierarchical clustering*, dapat dihasilkan suatu kumpulan partisi yang berurutan, dimana dalam kumpulan tersebut terdapat:

a. *Cluster–cluster* yang mempunyai poin–poin individu. *Cluster– cluster* ini berada di level yang paling bawah.
b. Sebuah *cluster* yang di dalamnya terdapat poin–poin yang dipunyai semua *cluster* di dalamnya. *Single cluster* ini berada di *level* yang paling atas.

Hasil keseluruhan dari algoritma *hierarchical clustering* secara grafik dapat digambarkan sebagai *tree*, yang disebut dengan *dendogram*. *Tree* ini secara grafik menggambarkan proses penggabungan dari *cluster–cluster* yang ada, sehingga menghasilkan *cluster* dengan level yang lebih tinggi [8].

2.5 PostgreSQL

PostgreSQL adalah sebuah sistem basis data yang disebarluaskan secara bebas menurut perjanjian lisensi BSD. Piranti lunak ini merupakan salah satu basis data yang paling banyak digunakan saat ini, selain MySQL dan Oracle. PostgreSQL menyediakan fitur yang berguna untuk replikasi basis data. Fitur-fitur yang disediakan PostgreSQL antara lain *complex queries, foreign keys, triggers, updatable views, transactional integrity, dan multiversion concurrency control* [9].

2.6 Regular Expression

Regular expression sering juga disebut *Rational expression*, merupakan sebuah teknik yang digunakan untuk

mencocokkan string teks seperti karakter tertentu, kata-kata, atau pola karakter. *Regular expression* digunakan oleh banyak teks editor, utilities, dan bahasa pemrograman untuk pencarian dan memanipulasi teks berdasarkan pola. Misalnya Perl, Ruby, dan Tcl memiliki *engine regular expression* yang kuat dibangun pada sintaks mereka [10].

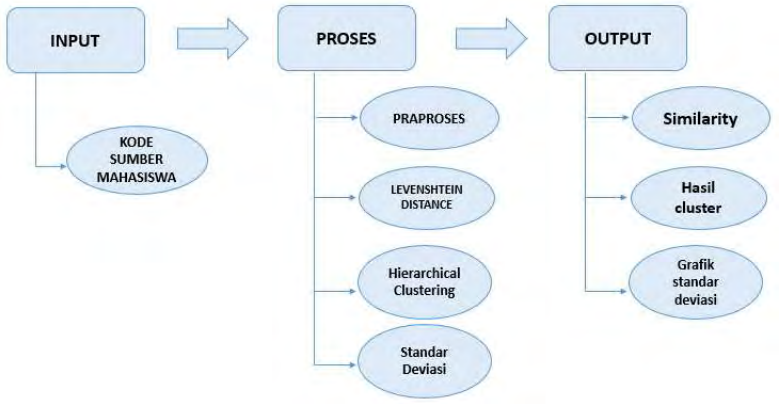
Beberapa notasi dasar dari *regular expression* antara lain sebagai berikut [11] :

1. **|** disebut juga alternasi atau pemilihan. Dapat dibaca sebagai "atau".
2. **( )** digunakan untuk mengelompokkan, fungsinya sama persis seperti tanda kurung pada matematika.
3. **[ ]** mengapit sebuah set karakter. Misal [0-9] cocok dengan angka 0 hingga 9
4. **?** berarti nol atau satu huruf /element di kiri bersifat opsional. Misal colou?r, cocok dengan "color" maupun "colour".
5. ^ dan **$** masing-masing dapat disebut sebagai "harus di awal" dan "harus di akhir"
6. **{n}** karakter atau kelompok huruf di kiri harus berjumlah sebanyak ***n***
7. **{min,max}** karakter atau kelompok huruf di kiri harus berjumlah minimal sebanyak ***min*** dan tidak boleh lebih dari nilai ***max.***

(Halaman ini sengaja dikosongkan)

BAB III
PERANCANGAN PERANGKAT LUNAK

Bab ini membahas mengenai perancangan dan pembuatan sistem perangkat lunak. Sistem perangkat lunak yang dibuat pada tugas akhir ini adalah mengolah data hasil *parsing* ANTLR dengan menggunakan fungsi *Listener* sebagai metode untuk mendapatkan hasil *parser* kode sumber berupa AST, dan membandingkan antar dua kode sumber menggunakan Metode *Levenshtein distance* dan modifikasi *similarity*, yang selanjutnya diklasifikasikan menggunakan *Hierarchical clustering.*



Gambar 3.1 Diagram input output sistem

Input dari sistem merupakan kode program mahasiswa dalam satu kelas. Selanjutnya masuk ke dalam tahap proses. Tahap proses terdiri dari 4 proses utama yaitu:

1. Praproses
   Pada proses ini hasil *parsing* kode program mahasiswa, yang berupa AST, diubah menjadi bentuk *sequence* menggunakan fungsi Listener.
2. Levenshtein distance

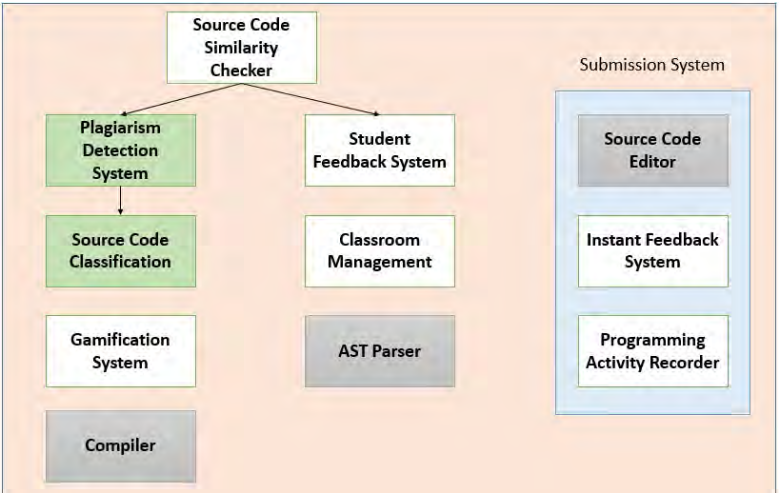
Pada proses ini dua buah hasil sequence dari tahap praproses sebelumnya dihitung nilai kemiripannya menggunakan metode *Levensthein distance* dan modifikasi *similarity*.

3. Hierarchical clustering
   Pada proses ini hasil *similarity* antar dua buah kode sumber mahasiswa dikelompokkan berdasarkan tingkat kemiripannya.
4. Standar deviasi
   Pada proses ini dihitung standar deviasi pada masing-masing *cluster* yang telah dibentuk menggunakan metode Hierarchical clustering pada tahap sebelumnya. Proses ini menghasilkan grafik standar deviasi per jumlah *cluster*, yang digunakan untuk menentukan jumlah cluster terbaik yang akan dipilih.

Hasil keluaran (output) dari sistem adalah nilai *similarity* antar dua buah kode sumber mahasiswa yang disimpan ke dalam basis data, hasil pengelompokan kode sumber berupa *cluster*, anggota *cluster*, dan *range* persentase kemiripan per *cluster* yang ditampilkan pada web E-learning, dan grafik standar deviasi per jumlah *cluster*.

3.1 Desain Umum Sistem

Arsitektur sistem dari aplikasi *E-learning Pemrograman* secara umum ditunjukkan pada Gambar 3.2. Bagian yang berwarna hijau pada Gambar 3.2 merupakan modul-modul yang diimplementasikan oleh penulis, yaitu modul Plagiarism Detection System dan Source Code Classification. Modul Plagiarism Detection System digunakan untuk mendeteksi dan menghitung tingkat kemiripan antar kode program mahasiswa dalam satu kelas. Sedangkan modul Source Code Classification digunakan untuk mengklasifikasikan kode program mahasiswa yang memiliki tingkat kemiripan yang sama.



Gambar 3.2 Arsitektur sistem E-learning Pemrograman

Modul Plagiarism Detection System dan Source Code Classification yang diimplementasikan oleh penulis diintegrasikan ke dalam platform *e-learning*. Sistem *back-end* dari modul-modul yang diimplementasikan penulis menghasilkan file berekstensi .jar yang akan diakses oleh platform *e-learning* menggunakan PHP dan Framework Laravel.

3.2 Data masukan

Data masukan adalah data yang digunakan sebagai masukan awal dari sistem. Data yang digunakan dalam perangkat lunak pendeteksi plagiarisme antar mahasiswa dalam satu kelas ini adalah dua buah jenis kode program mahasiswa yang berbeda soal. Kode program jenis pertama berjumlah 38 kode program milik 38 mahasiswa dalam satu kelas. Sedangkan kode program jenis kedua berjumlah 21 kode program milik 21 mahasiswa dalam satu kelas. Kode program jenis kedua berisi 21 file berekstensi .cpp dan 21 file berekstensi .h (file header). Kedua jenis kode sumber tersebut merupakan hasil jawaban masing-masing mahasiswa dari dua buah

tugas berbeda yang diberikan pada mata kuliah Pemrograman Berorientasi Objek (PBO). Tugas yang diberikan tersebut mengenai bagaimana mengimplementasikan suatu *case* ke dalam paradigma pemrograman berorientasi objek. Kode sumber tersebut diimplementasikan menggunakan bahasa pemrograman C++.

Contoh data masukan kode program mahasiswa jenis pertama ditunjukkan pada Gambar 3.3. Sedangkan contoh data masukan kode program mahasiswa jenis kedua ditunjukkan pada Gambar 3.4.

```
using namespace std;

class Pet
{
public:
    char nama[10];
    int makan, mandi, main, tidur, hebat;


    void eat(int jumlah)
    {
        mandi -= (jumlah / 2);
        tidur -= 2;
        main -= 1;
        makan += jumlah;
        if (makan>10)
        {
            makan = 10;
            cout << nama << " kekenyangaaaan" << "\n";
        }
    }
```

Gambar 3.3 Contoh potongan kode sumber jenis pertama salah satu mahasiswa

Pada dataset jenis pertama, jawaban mahasiswa satu dengan yang lainnya sangat bervariasi, sehingga tingkat kemiripan antar kode program mahasiswa relatif kecil.

Sedangkan pada dataset jenis kedua, tingkat kemiripan antar kode program relatif besar.

```
void Ball::Render(wxDC *dc)
{
    dc->SetPen(*wxWHITE_PEN);
    dc->SetBrush(*this->brush);

    speedY = speedY + 0.5 * t;

    x = x + speedX;
    y = y + speedY;

    dc->DrawCircle(x, y, radius);

    t += 0.01f;
}
```

Gambar 3.4 Contoh potongan kode sumber jenis kedua salah satu mahasiswa

3.3 Proses

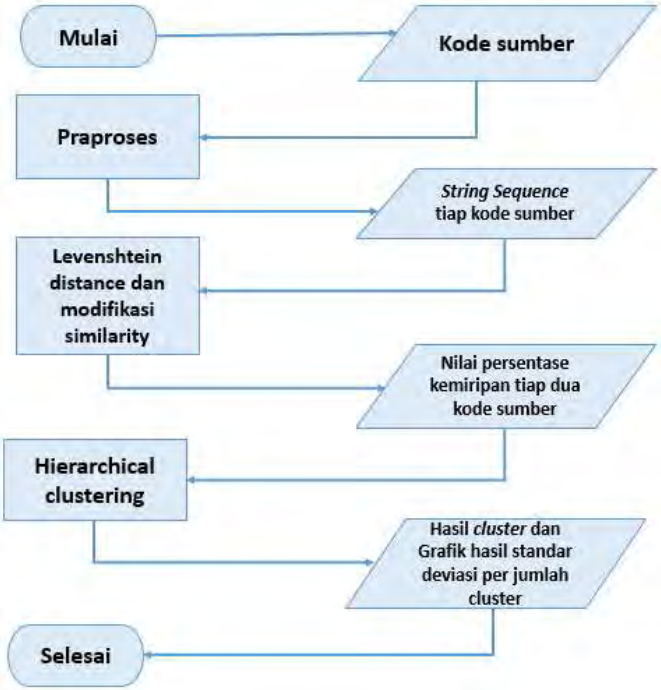
Rancangan *back-end* perangkat lunak untuk mendeteksi plagiarisme kode sumber mahasiswa ini menggunakan fungsi *Listener* pada ANTLR yang telah dimodifikasi, metode *Levensthein distance* dan penghitungan *similarity* yang telah dimodifikasi, dan metode klasifikasi *Hierarchical clustering.* Dimulai dengan tahap praproses yaitu mengolah dan mengubah data hasil *parsing* ANTLR menggunakan fungsi *Listener* yang telah dimodifikasi menjadi bentuk *sequence*. Bentuk *sequence* ini merepresentasikan bentuk AST hasil *parsing* ANTLR pada tiap kode sumber.

Tahap selanjutnya adalah membandingkan dan menghitung tingkat kemiripan antar dua buah *sequence* dari dua buah kode sumber menggunakan metode *Levenshtein distance* dan modifikasi *similarity*. Hasil dari proses membandingkan dua buah kode sumber ini adalah nilai *similarity* (nilai kemiripan antar dua buah kode sumber). Nilai-nilai tersebut digunakan dalam proses klasifikasi menggunakan metode *Hierarchichal clustering*.

Sehingga didapatkan kode sumber mana saja yang termasuk dalam satu *cluster* yang sama.

Untuk menentukan jumlah *cluster* yang akan diambil adalah dengan menghitung nilai standar deviasi dari masing-masing jumlah *cluster*. Nilai standar deviasi tersebut disajikan dalam bentuk grafik. Sehingga melalui grafik tersebut kita dapat mengetahui nilai jumlah *cluster* yang paling stabil untuk kemudian diambil sebagai jumlah *cluster* yang akan digunakan dalam pengelompokan plagiarisme kode sumber mahasiswa dalam satu kelas.

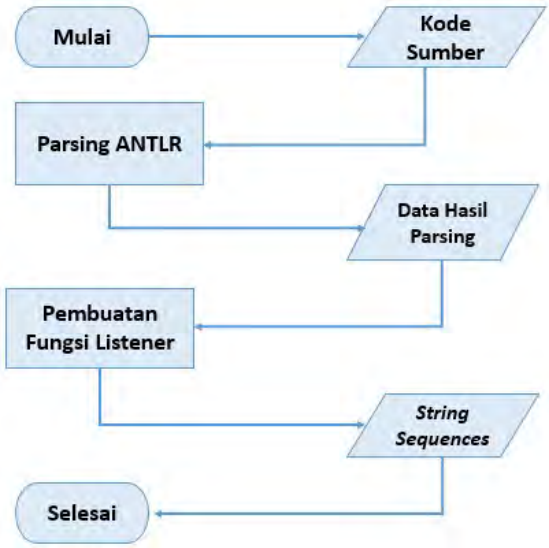
Diagram alir proses pendeteksian dan pengelompokan plagiarisme kode program mahasiswa ditunjukkan pada Gambar 3.5.



Gambar 3.5 Diagram alir proses deteksi plagiarisme kode program

3.3.1 Praproses

Praproses merupakan tahap untuk mengubah data masukan menjadi bentuk yang sesuai untuk proses pendeteksian plagiarisme dan proses klasifikasi. Pada proses pendeteksian plagiarisme kode program, praproses digunakan untuk mengubah data hasil *parsing* ANTLR menjadi bentuk *sequence* menggunakan fungsi *Listener* yang telah dimodifikasi. Langkah praproses digambarkan pada diagram di Gambar 3.6.



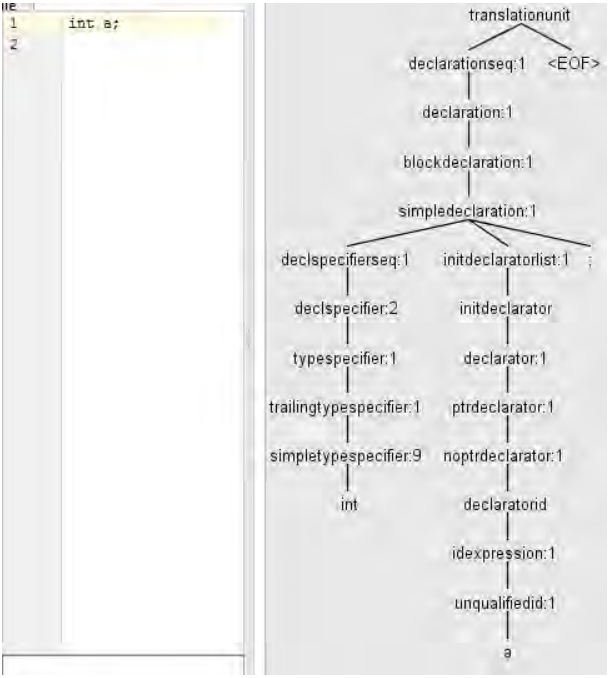
Gambar 3.6 Diagram alir tahap praproses

Data masukan dari tahap praproses ini adalah kode sumber mahasiswa. Kode sumber tersebut di-*parsing* menggunakan *plugin* ANTLR dan diolah menggunakan fungsi *Listener* pada ANTLR *parser*. Untuk memparsing kode program mahasiswa melalui *ANTLR parser* harus menggunakan suatu aturan yang disebut *grammar*. Grammar yang digunakan adalah *grammar C++* yang telah disediakan oleh ANTLR. Listener digambarkan sebagai

sebuah *tree-walking* yang berjalan-jalan mengunjungi setiap bagian *node-node* pada AST yang dihasilkan oleh ANTLR *parser*.

Listener memiliki dua fungsi utama yaitu `Enter` dan `Exit`. `Enter` digunakan ketika Listener tersebut mengunjungi suatu *node* pada AST dan `Exit` digunakan ketika Listener selesai mengunjungi suatu *node*. *Node-node* tersebut merupakan representasi dari isi *grammar* C++ yang akan digunakan ANTLR untuk mem-*parsing* kode program, sehingga menghasilkan data hasil *parsing* berupa AST *(Abstract Syntax Tree).*

Contoh hasil representasi kode sumber menjadi sebuah AST menggunakan ANTLR *parser* dan *grammar* C++ dapat dilihat pada Gambar 3.7.



Gambar 3.7 Contoh AST hasil parsing kode sumber

Tidak semua bagian/segmen dari kode sumber diambil untuk proses pengecekan plagiarisme. Karena ini merupakan pengecekan plagiarisme berdasarkan struktur kode sumbernya, maka hanya beberapa bagian/segmen dari kode sumber yang di *parsing* oleh *grammar C++* menggunakan Listener *parser* milik ANTLR. Bagian-bagian tersebut dijelaskan pada Tabel 3.1.

Tabel 3.1 Elemen/*Rule* program C++ yang di-*parsing*

No	*Rule* pada Grammar	Keterangan pada kode sumber
1	Declaration	Sebuah pendeklarasian suatu kode yang umumnya diakhir dengan **;** pada c++
2	Simpletypespecifier	Tipe data pada suatu variabel (*int, char, long, double*, dll)
3	Functiondefinition	Pendeklarasian suatu fungsi
4	Parametersandqualifiers	Parameter dalam sebuah fungsi
5	Functionbody	Isi suatu fungsi
6	Selectionstatement	Statement *if , else if, else, dan switch-case* pada c++
7	Iterationstatement	Statement perualangan berupa *for, while, do-while* pada c++
8	Condition	Sebuah kondisi didalam statement *if , else if, else, case, while, for, dan do-while* pada c++
9	Classpecifier	Pendeklarasian suatu kelas atau struct pada c++
10	Accesspecifier	Jenis *access specifier* pada anggota suatu kelas (*public, private***,** dan *protected*)
11	Basespecifier	Pendeklarasian sebuah kelas turunan pada c++

12	Assignmentexpression	Pendeklarasian sebuah *assignment* pada program c++
13	Assignmentoperator	Operator yang menghubungkan sebuah *assignment*. Operator terdiri dari **(=)** **(+=)** **(-=)** **(*=)** **(/=)** **(%=)**
14	Unqualifiedid	Penggunaan/deklarasi suatu variabel pada kode program C++
15	Literal	Penggunaan/deklarasi suatu angka maupun string literal pada kode program C++
16	Expressionlist	Isi parameter dari pemanggilan suatu fungsi
17	Postfixexpression	Mendeteksi *statement* yang diikuti dengan : access operator (**.**) dan (→) increment (++) decrement (--) pemanggilan fungsi (**( )**)
18	Shiftexpression	Mendeteksi penggunaan : **cout** (<<) **cin** (>>) pada kode program C++
19	Jumpstatement	Mendeteksi penggunaan ***break***
20	Classname	Mendeteksi suatu kelas apakah memiliki *constructor* atau tidak. Dengan melakukan

		pengecekan apakah nama kelas sama dengan fungsi
21	Braceorequalinitializer	Mendeteksi tipe data yang diikuti suatu *assignment operator*. Misal (int a = 10)
22	Ptroperator	Mendeteksi adanya deklarasi suatu pointer
23	Primaryexpression	Untuk mengidentifikasi adanya penggunaan this pointer (**this** →)
24	Classkey	Untuk membedakan deklarasi ***class*** dan ***struct***

Setelah kode sumber tersebut menghasilkan hasil *parsing* berupa AST, maka fungsi *Listener* tadi dimodifikasi untuk mengubah bentuk AST tersebut menjadi sebuah bentuk *sequence*. Modifikasi fungsi *Listener* dilakukan dengan merepresentasikan semua *rule* yang ada pada *grammar C++* hasil *parsing* tadi ke dalam suatu rangkaian huruf. Rangkaian huruf tersebut diikuti *parenthesis* (buka kurung dan tutup kurung) yang berguna untuk menandai bahwa huruf-huruf tersebut berada dalam satu level yang sama atau tidak. Kode-kode berupa rangkaian huruf yang merepresentasikan semua *rule* yang akan di-*parsing* pada *grammar C++* dijelaskan pada Tabel 3.2.

Tabel 3.2 Pengkodean *rule* grammar C++ menjadi rangkaian huruf

No	*Rule* pada Grammar	Kode huruf
1	Declaration	**A ( )**
2	Simpletypespecifier	**B : tipe** (tipe merupakan tipe variabel, misal B: int)
3	Functiondefinition	**C ( )**
4	Parametersandqualifiers	**D ( )**

5	Functionbody	**E ( )**
6	Selectionstatement	**H : statement** Misal (**H : if**)
7	Iterationstatement	**I : statement ( )** Misal (**I : for**)
8	Condition	**J (kondisi)** kondisi disini diikuti oleh operotor kondisi, yaitu : (||) (**&&**) (==) (>=) (<=) (>) (<) (**!**=) Selain itu kondisi juga dapat diikuti sutau variabel atau literal.
9	Classpecifier	**K ( )**
10	Accesspecifier	**L : *accessspecifier*** Misal (**L : public**)
11	Basespecifier	**M**
12	Assignmentexpression	Tidak dikodekan. Namun digunakan dalam tahap praproses
13	Assignmentoperator	Operator yang menghubungkan sebuah *assignment*. Operator terdiri dari : **(=)** **(+=)** **(-=)** **(*=)** **(/=)**

		(%=) Operator diikuti oleh suatu variabel (**V**) atau angka (literal)
14	Unqualifiedid	**V** simbol ini dapat diikuti oleh : operator dari sebuah *assignment* (misal **=V**) parameter pemanggilan fungsi (misal **PV**) kondisi (misal **JV**) statement dari *switch-case* (misal **GV**) simbol-simbol dari postfix expression (misal **VO**)
15	Literal	Angka yang tertera pada kode program misal : **1** , **2** Tidak hanya angka, namun literal juga meliputi string literal. Literal dapat diikuti oleh : operator dari sebuah *assignment* (misal **=2**) parameter pemanggilan fungsi (misal **P2**) kondisi (misal **J2**) statement dari *switch-case* (misal **G2**)
16	Expressionlist	**P** simbol ini dapat diikuti oleh : operator dari sebuah *assignment* (misal **=P**) variabel (misal **PV**) literal (misal **P2**) kondisi (misal **JP**)

		statement dari *switch-case* (misal **GP**) simbol-simbol dari postfix expression (misal **PVO**)
17	Postfixexpression	Mendeteksi *statement* yang diikuti dengan : Access operator (**.**) dan (→) **V.** (misal : Pet.makan) **V→** (misal : Pet → makan) Pemanggilan fungsi **VO** (misal : makan()) Increment **V--** (misal : input--) Decrement **V++** (misal : input++) Pemanggilan fungsi yang diikuti access operator **V.O** (misal : Pet.makan()) **V→O** (misal : Pet→makan())
18	Shiftexpression	Tidak dikodekan. Namun digunakan dalam tahap praproses
19	Jumpstatement	**break** (apabila berada didalam suatu switch-case, disimbolkan menjadi **Gbreak**)
20	Classname	Tidak dikodekan. Namun digunakan dalam tahap praproses

21	Braceorequalinitializer	Tidak dikodekan. Namun digunakan dalam tahap praproses
22	Ptroperator	Disimbolkan dengan (***** atau **&**)
23	Primaryexpression	**This**
24	Classkey	Diikuti oleh simbol **K**. Misal : **K:struct** atau **K:class**

Selain itu modifikasi untuk membentuk suatu *sequence* juga dilakukan pada beberapa bagian di dalam isi *grammar C++*.

Hasil keluaran dari fungsi *Listener* menghasilkan sebuah bentuk *sequence* yang berisi rangkaian huruf-huruf seperti pada Tabel 3.2 di atas, dan setiap rangkaian huruf tersebut memiliki level yang berbeda-beda sesuai dengan *parenthesis* yang mengikutinya. Umumnya setiap *sequence* yang berada di dalam sebuah *parenthesis* memiliki level yang lebih besar dibandingkan dengan *sequence* yang berada di luar sebuah *parenthesis*.

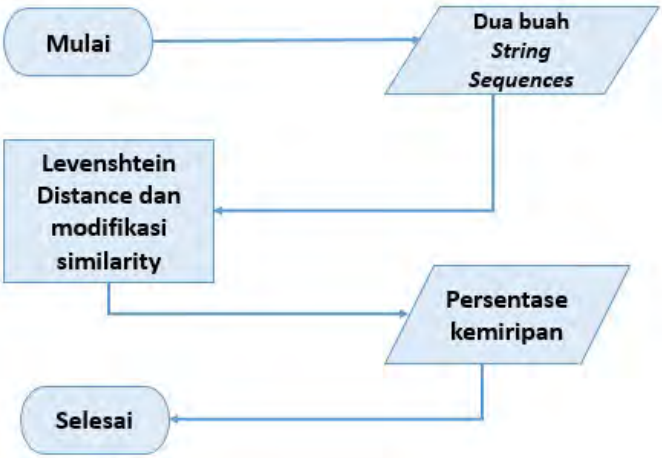
Hasil *sequence* dan level *sequence* pada tahap praproses disimpan ke dalam basis data.

3.3.2 Levenshtein Distance

Levensthein distance merupakan metode yang digunakan untuk menghitung nilai kemiripan antara dua buah kode sumber. Data masukan dalam proses penghitungan nilai kemiripan ini adalah data kode sumber yang berbentuk *string sequence*, hasil dari tahap praproses sebelumnya. Kelebihan metode *levensthein distance* adalah dapat mendeteksi kemiripan dua buah *string* yang diubah, dihapus, dan ditambah. Oleh karena itu metode ini juga sering disebut metode *Edit distance*.

Pada penghitungan *similarity* antar kode sumber dibutuhkan sedikit modifikasi untuk menyelesaikan beberapa *case* dalam pendeteksian plagiarisme kode sumber yang tidak dapat

sepenuhnya dideteksi oleh penghitungan *similarity* dari metode *Levensthein distance* pada umumnya. Langkah penghitungan kemiripan antara dua buah kode sumber dapat dilihat pada Gambar 3.7.



Gambar 3.8 Penghitungan kemiripan dua buah kode sumber

Konsep dasar metode *Levenshtein distance* adalah nilai matriks (*baris, kolom*) diambil dari nilai terkecil (*minimum*) antara nilai matriks(*baris-1, kolom-1*), matriks(*baris-1, kolom*), dan matriks*(baris, kolom-1*). Aturan lain dari algoritma ini adalah apabila elemen pada *sequence* dan level *sequence* kode program pertama dengan elemen pada *sequence* dan level *sequence* pada kode program ke dua bernilai sama, maka matriks pada indeks ke (*baris-1, kolom-1*) ditambah dengan nilai *cost* nol. Begitu pula sebalikanya, nilai *cost*-nya satu. Sedangkan nilai matriks pada indeks ke (*baris, kolom-1*) dan matriks pada indeks (*baris-1, kolom*) selalu ditambah dengan satu. Hasil akhir, berupa nilai jarak antara dua buah *string sequence*, dari metode *Levenshtein distance* ini ada pada baris terakhir dan kolom terakhir matriks.

Contoh implementasi penghitungan jarak antara dua buah *string sequence* menggunakan metode *Levenshtein distance* adalah sebagai berikut. Pada matriks dibawah ini, kolom berwarna kuning (*baris, kolom*) didapatkan dari nilai minimum dari tiga buah nilai :

1. kolom berwarna merah (*baris, kolom-1*) + 1
2. kolom berwarna merah muda (*baris-1, kolom*) + 1
3. kolom berwarna hijau (*baris-1, kolom-1*) + 0

Sequence kode program 1

Sequence kode program 2			K:class	C	D	E	B:long
		0	1	2	3	4	5
	K:class	1	0	1	2	3	4
	C	2	1	0	1	2	3
	D	3	2	1	0	1	2
	E	4	3	2	1	0	1
	B:int	5	4	3	2	1	1

Hasil akhir dari penghitungan matriks *Levenshtein distance* di atas ada pada kolom berwarna biru (baris terakhir, kolom terakhir). Persamaan 3.1 digunakan untuk menghitung nilai *similarity* dari hasil jarak *levenshtein* yang telah dihitung.

$$similarity = \left(1 - \frac{levenshtein}{\text{maksimum}(n_1, n_2)}\right) \times 100\% \quad (3.1)$$

$levenshtein$ merupakan nilai matriks pada baris terakhir dan kolom terakhir pada matriks penghitungan jarak levenshtein. n_1 merupakan nilai dari panjang *string sequence* kode program ke-1 dan n_2 merupakan nilai dari panjang *string sequence* kode program ke-2.

Kelemahan dari metode *Levenshtein distance* dalam mendeteksi plagiarisme antara dua buah kode program adalah tidak mampu mendeteksi isi kode program yang urutan posisi penulisannya ditukar (dibolak-balik). Sehingga untuk mengatasi

hal tersebut, diperlukan modifikasi terhadap pengecekan dan penghitungan *similarity* antar dua buah kode sumber.

Modifikasi *similarity* dilakukan untuk mendeteksi susunan/urutan isi kode program yang posisinya ditukar (dibolak-balik). Modifikasi dilakukan dengan cara melakukan pengecekan di setiap isi baris dan kolom matriks. Sehingga apabila terdapat *sequence* yang urutannya ditukar, dapat dideteksi, karena pengecekan *sequence* dilakukan secara menyeluruh pada tiap kolom dan baris. Apabila kedua *substring sequence* dan level *sequence* antara dua buah kode sumber bernilai sama, maka *cost* pada baris atau kolom tersebut bernilai satu. Selanjutnya *substring sequence* yang telah memiliki pasangan yang mirip dengan *substring sequence* lain akan ditandai. Proses penandaan ini digunakan untuk menandai bahwa tidak akan ada *substring sequence* yang memiliki kemiripan lebih dari satu dengan *substring sequnce* pembandingnya.

Penghitungan persentase kemiripan modifikasi *similarity* adalah dengan melakukan penjumlahan terhadap baris atau kolom yang memiliki nilai cost satu, selanjutnya hasil penjumlahan tersebut dibagi dengan panjang maksimum antara kedua buah *sequence* kode program yang dibandingkan. Penghitungan persentase kemiripan modifikasi penghitungan *similarity* dituliskan melalui Persamaan 3.2.

$$modifikasi\ similarity = \left(\frac{N}{\text{maksimum}(n_1, n_2)}\right) \times 100\% \qquad (3.2)$$

N merupakan jumlah *substring sequence* yang sama antara kedua buah *sequence* yang dibandingkan. n_1 merupakan nilai dari panjang *string sequence* kode program ke-1 dan n_2 merupakan nilai dari panjang *string sequence* kode program ke-2.

Hasil persentase kemiripan menggunakan metode *Levenshtein distance* maupun modifikasi *similarity*, keduanya disimpan ke dalam basis data.

3.3.3 Hierarchical clustering

Hierarchichal clustering merupakan metode yang digunakan dalam proses pengelompokan/klasifikasi kode sumber berdasarkan tingkat kemiripannya. Konsep dasar *Hierarchical clustering* adalah mengambil nilai minimum dari setiap nilai kolom dan baris pada matriks dan menggabungkan kedua elemen baris dan kolom tersebut menjadi satu *cluster*. Matriks yang dimaksudkan tersebut berisi nilai jarak antar kode program. Jarak antar kode program dihitung melalui Persamaan 3.3.

$$dist = 100\% - persentase\ kemiripan \qquad (3.3)$$

Jika setiap pasangan kode sumber memiliki tingkat kemiripan yang besar maka jaraknya relatif kecil. Sehingga nilai minimum yang diambil dari hasil penghitungan jarak tersebut, merupakan nilai maksimum dari hasil penghitungan persentase tingkat kemiripan.

Metode *Minimum linkage* digunakan untuk mengambil nilai minimum dari isi matriks, yaitu jarak yang paling kecil antara dua buah kode program. *Single linkage* digunakan untuk meng-*update* nilai matriks pada saat penggabungan member dalam proses *clustering*. Proses *clustering* dimulai dari menggabungkan dua buah kode program yang mirip (jarak *similarity* yang paling kecil), hingga menggabungkan seluruh kode program menjadi satu *cluster*.

Algoritma *Hierarchical clustering* dapat dituliskan sebagai berikut:

1. Siapkan matriks berukuran n x n sejumlah mahasiswa dalam satu kelas. Matriks tersebut bersifat *mirror*. Masukkan nilai jarak antar kode program mahasiswa ke dalam matriks n x n tersebut. Misal pada matriks berukuran 6 x 6, seperti pada gambar di bawah ini.

Dist	A	B	C	D	E	F
A	0	0,71	5,66	3,61	4,24	3,2
B	0,71	0	4,95	2,92	3,54	2,5
C	5,66	4,95	0	2,24	1,41	2,5
D	3,61	2,92	2,24	0	1	0,5
E	4,24	3,54	1,41	1	0	1,12
F	3,2	2,5	2,5	**0,5**	1,12	0

Huruf **A, B, C, D, E, F** pada matriks di atas dianggap sebagai NRP masing-masing kode program.

2. Lalu ambil nilai matriks yang memiliki nilai terkecil. Pada gambar di atas nilai terkecil adalah nilai jarak antara kode program **D** dan **F** yaitu **0,5**.

3. Gabungkan **D** dan **F** menjadi satu *cluster*.

4. Kosongkan nilai matriks yang berkaitan dengan kode program **D** dan **F**.

Min distance(Single Linkage)					
Dist	A	B	C	D,F	E
A	0	0,71	5,66	?	4,24
B	0,71	0	4,95	?	3,54
C	5,66	4,95	0	?	1,41
D, F	?	?	?	0	?
E	4,24	3,54	1,41	?	0

5. Matriks yang kosong tadi, nilainya dihitung dengan menggunakan metode *Single linkage*.

Antara matriks D, F dengan A

$$d_{(D,F)\to A} = \min(d_{DA}, d_{FA}) = \min(3{,}61, 3{,}20) = 3{,}20$$

Antara matriks D, F dengan B

$$d_{(D,F)\to B} = \min(d_{DB}, d_{FB}) = \min(2{,}92, 2{,}50) = 2{,}50$$

Antara matriks D, F dengan C

$$d_{(D,F)\to C} = \min(d_{DC}, d_{FC}) = \min(2{,}24, 2{,}50) = 2{,}24$$

Antara matriks E dengan D, F

$$d_{E\to(D,F)} = \min(d_{ED}, d_{EF}) = \min(1{,}00, 1{,}12) = 1{,}00$$

Sehingga matriks tersebut menjadi

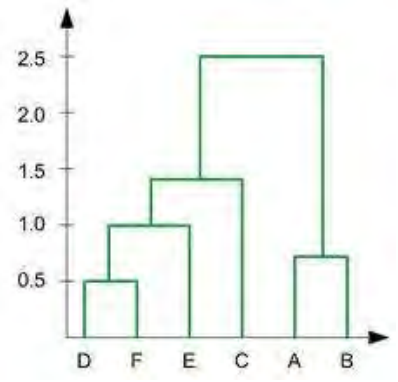
Dist	A	B	C	D,F	E
A	0	0,71	5,66	3,2	4,24
B	**0,71**	0	4,95	2,5	3,54
C	5,66	4,95	0	2,24	1,41
D, F	3,2	2,5	2,24	0	1
E	4,24	3,54	1,41	1	0

6. Selanjutnya cari lagi nilai minimum matriks dengan metode *minimum linkage*. Kemudian gabungkan menjadi satu *cluster* seperti pada langkah ke 2.

7. Ulangi langkah di atas, hingga semua matriks bergabung menjadi satu *cluster*.

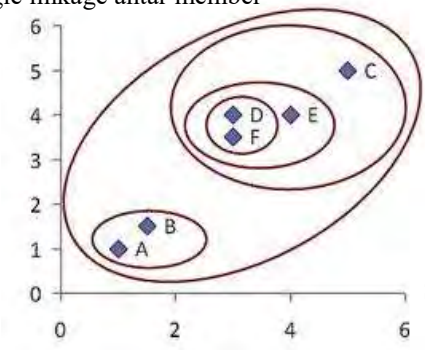
8. Hasilnya dalam bentuk dendogram adalah sebagai berikut. Hasil dendogram diperoleh dari nilai *minimum linkage* yang diperoleh pada tiap tahap penggabungan *cluster*.

Minimum linkage



9. Hasilnya dalam bentuk grafik XY adalah sebagai berikut. Grafik XY pada koordinat y menunjukkan *single linkage* member terhadap member lain dan koordinat x menunjukkan *single linkage* cluster terhadap cluster lain.

Single linkage antar member



Single linkage antar cluster

10. Dari hasil grafik dan diagram dendogram di atas dapat disimpulkan isi member tiap *cluster* adalah sebagai berikut:
    Cluster = 1 : **(A, B, C, D, E, F)**
    Cluster = 2 : **(A, B)** dan **(C, D, E, F)**

Cluster = 3 : **(A, B)** , **(D, E, F)** , dan **(C)**
Cluster = 4 : **(A, B)** , **(D, F)** , **(E)** dan **(C)**
Cluster = 5 : **(A) , (B) , (D, F) , (E),** dan **(C)**

Hasil *cluster* dan daftar anggota tiap jumlah cluster dari jumlah *cluster* = N-1 hingga menjadi satu *cluster* ditulis dan disimpan pada file yang berekstensi .txt.

3.3.4 Standar Deviasi

Tahap selanjutnya adalah menghitung nilai standar deviasi pada masing-masing *cluster* yang telah didapat pada tahap *clustering* sebelumnya, untuk menentukan jumlah cluster terbaik yang akan digunakan. Langkah–langkah menghitung standar deviasi tiap-tiap jumlah *cluster* adalah sebagai berikut :

1. Menghitung nilai *centroid* tiap *cluster* menggunakan Persamaan 3.4.

$$centroid(k) = \frac{1}{N} \times \sum_{i=0}^{i=N} similarity\_distance(i, k) \quad (3.4)$$

$centroid(k)$ merupakan nilai centroid pada member ke–k. $similarity\_distance(i, k)$ merupakan nilai persentase kemiripan antara member cluster ke-i terhadap member ke-k. N merupakan banyak member dalam satu cluster.

Misal *cluster* pertama berisi member **D** , **E**, dan **F**. Centroid pada *cluster* diperoleh dari hasil penghitungan rata-rata tiap member pada *cluster* dengan member-member pada matriks.

Nilai centroid **(D, E, F)→A** = (1/3) × (3,61 + 4,24 + 3,2)
= **3,68**

0	A	B	C	D	E	F
D	3,61	2,92	2,24	0	1	0,5
E	4,24	3,54	1,41	1	0	1,12
F	3,2	2,5	2,5	0,5	1,12	0
centroid	3,68	2,987	2,05	0,5	0,707	0,54

2. Menghitung jarak setiap member dalam satu *cluster* dengan centroidnya menggunakan *Eucledian distance*. Penghitungan pada member **D** adalah sebagai berikut:

0	A	B	C	D	E	F
D	3,61	2,92	2,24	0	1	0,5
centroid	3,68	2,987	2,05	0,5	0,707	0,54
beda	0,073	0,067	0,19	0,5	0,29	0,04
power	0,0054	0,004	0,036	0,25	0,086	0,001

Beda diperoleh dari selisih antara nilai matriks (D, A) , (D, B) , (D, C), (D, D) , (D, E) , dan (D, F) dengan centroidnya.
Power diperoleh dari hasil kuadrat masing-masing nilai **beda** per member *cluster*.
Nilai *Eucledian distance* diperoleh dari akar kuadrat dari penjumlahan semua nilai **power.**

Eucledian D

$$= \sqrt[2]{0{,}054 + 0{,}004 + 0{,}036 + 0{,}25 + 0{,}086 + 0{,}001}$$

$$= \mathbf{0,619}$$

Penghitungan nilai *Eucledian distance* yang sama juga dilakukan pada member **E** dan **F.**

Eucledian E = 1,453
Eucledian F = 1,066

3. Menghitung nilai standar deviasi tiap *cluster*
   Standar deviasi pada *cluster* dengan member **D , E ,** dan **F** dihitung melalui Persamaan 3.5.

$$Std = \sqrt{\frac{\sum_{i=0}^{i=N}(eucledian(i) - mean)^2}{N-1}} \quad (3.5)$$

   Pada langkah penghitungan nilai *eucledian distance* nomor 2 di atas, terdapat 3 buah nilai eucledian, yaitu eucledian **D,** eucledian **E** , dan eucledian **F**.
   Rata-rata (**mean**) dari ketiga nilai eucledian tersebut adalah **1,046.**

   Maka nilai Standar deviasi cluster tersebut adalah

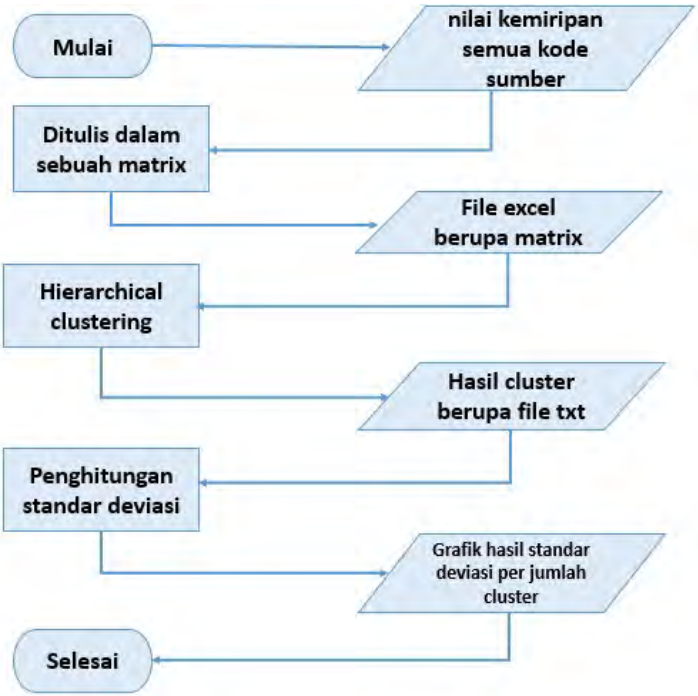
$$= \sqrt{\frac{(0{,}619 - 1{,}046)^2 + (1{,}453 - 1{,}046)^2 + (1{,}066 - 1{,}046)^2}{3-1}}$$

$$= \mathbf{0,417}$$

4. Menghitung rata-rata dari nilai standar deviasi tiap jumlah *cluster*
   Misal pada *cluster* di atas, jumlah cluster yang diambil adalah 3, maka :
   Cluster 1 = **A** dan **B**
   Nilai standar deviasi cluster 1 = **0**
   Cluster 2 = **C**
   Nilai standar deviasi cluster 2 = **0**
   Cluster 3 = D, E, dan **F**
   Nilai standar deviasi cluster 3 = **0,417**

   Sehingga nilai rata-rata pada *cluster* yang berjumlah 3 *cluster* tersebut adalah
   = (1/3) × (0 + 0 + 0,417) = **0,139**

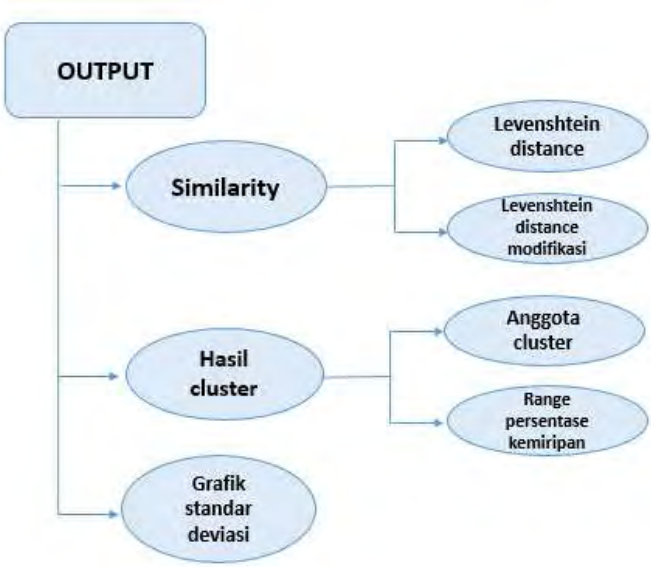
Diagram alir pengelompokan kode sumber berdasarkan tingkat kemiripannya dan penentuan jumlah *cluster* terbaik digambarkan pada Gambar 3.9.



Gambar 3.9 Proses klasifikasi kode sumber mahasiswa

3.4 Data Keluaran

Diagram data hasil keluaran yang dihasilkan sistem deteksi plagiarisme kode program ini dapat dilihat pada Gambar 3.10.



Gambar 3.10 Diagram data keluaran sistem

Data masukan akan diproses pada tahap praproses dengan menggunakan fungsi *Listener* yang telah dimodifikasi pada ANTLR untuk mem*parsing* kode sumber dan mengubah bentuk AST hasil *parsing* tersebut menjadi bentuk *sequence*. Bentuk *sequence* tiap kode sumber disimpan dalam basis data, yang selanjutnya akan digunakan untuk membandingkan tingkat kemiripan dua buah *sequence* dari dua buah kode sumber menggunakan metode *Levenshtein distance* yang telah dimodifikasi. Hasil penghitungan kemiripan antara dua buah kode program mahasiswa dengan menggunaka metode *Levenshtein distance* tanpa modifikasi dan metode *Levenshtein distance* yang telah dimodifikasi, keduanya disimpan ke dalam basis data.

Nilai tingkat kemiripan antar dua buah kode sumber dalam satu kelas diklasifikasi dan dikelompokkan berdasarkan tingkat kemiripannya menggunakan metode *Hierarchical clustering*.

Hasil dari proses klasifikasi adalah NRP kode sumber mana saja yang masuk ke dalam masing-masing *cluster*, *range* persentase kemiripan anggota-anggota dalam satu *cluster*, dan grafik hasil penghitungan *standar deviasi* masing-masing jumlah *cluster* yang digunakan untuk menentukan berapa jumlah *cluster* yang akan diambil.

Maka hasil keluaran yang akan ditampilkan dari sistem deteksi plagiarisme kode program ini adalah anggota-anggota tiap *cluster*, *range* persentase kemiripan antar anggota tiap *cluster*, dan grafik standar deviasi per jumlah *cluster.*

3.5 Basis Data

Rancangan basis data pada sistem untuk mendeteksi plagiarisme kode program ini terdiri dari 2 buah tabel (entitas), yaitu tabel *db_sequence* dan tabel *db_similarity*.

Hasil *sequence* dan level *sequence* tiap kode program pada tahap praproses disimpan ke dalam entitas *db_sequence* pada basis data. Atribut-atribut yang terdapat pada entitas *db_sequence* dijelaskan pada Tabel 3.3.

Tabel 3.3 Atribut-atribut pada entitas *db_sequence*

No	Atribut	Penjelasan
1	id_seq	ID tiap kode program mahasiswa yang digenerate melalui UUID
2	extension	Ekstensi dari file kode program mahasiswa (umumnya .cpp)
3	file_name	Nama file kode program mahasiswa
4	hashcode	Nilai *hashcode* tiap file kode program mahasiswa (bersifat unik)

5	kelas	Kelas dimana mata kuliah diambil
6	kode_soal	Kode soal dari tiap kode program mahasiswa
7	last_modified	*Timestamp,* waktu dimana file kode program mahasiswa terakhir dibuat
8	nrp	NRP mahasiswa
9	path	Letak file kode program mahasiswa disimpan
10	size	Ukuran file kode program mahasiswa (*byte*)
11	sequence	String panjang hasil konversi *sequence*
12	Level_sequence	String panjang hasil perhitungan level tiap elemen pada *sequence*

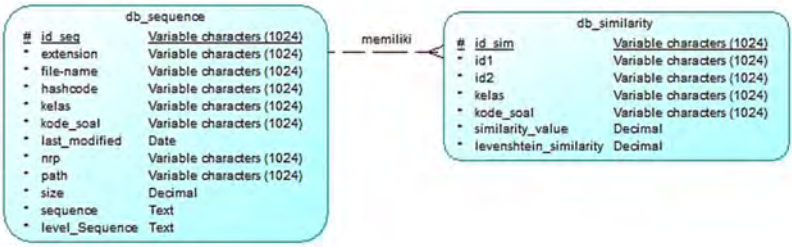
Nilai persentase kemiripan antar dua buah kode program mahasiswa disimpan pada entitas *db_similarity* di dalam basis data. Atribut-atribut pada entitas *db_similarity* dijelaskan pada Tabel 3.4.

Tabel 3.4 Atribut-atribut pada entitas *db_similarity*

No	Atribut	Penjelasan
1	id_sim	ID nilai *similarity* tiap kode program mahasiswa yang digenerate melalui UUID
2	id1	Nrp kode program mahasiswa pertama
3	id2	NRP kode program mahasiswa kedua
4	kelas	Kelas dimana mata kuliah diambil

5	kode_soal	Kode soal dari tiap kode program mahasiswa yang dibandingkan
6	similarity_value	Nilai persentase similarity, hasil dari metode *levensthein distance* yang telah dimodifikasi
7	levensthein_similarity	Nilai persentase similarity, hasil dari metode *Levensthein distance* yang tidak dimodifikasi

Cdm dari kedua tabel diatas dapat dilihat pada Gambar 3.11 dibawah ini.



Gambar 3.11 CDM dari tabel pada basis data sistem

BAB IV
IMPLEMENTASI

Bab ini berisi penjelasan mengenai implementasi dari perancangan yang sudah dilakukan pada bab sebelumnya. Implementasi berupa kode sumber untuk membangun program.

4.1 Lingkungan Implementasi

Implementasi deteksi plagiarisme kode program mahasiswa dalam satu kelas menggunakan metode *Levensthein distance*, modifikasi *similarity* dan *Hierarchical clustering* ini menggunakan spesifikasi perangkat keras dan perangkat lunak seperti yang ditunjukkan pada **Error! Reference source not found.**.

Tabel 4.1 Lingkungan Implementasi Perangkat Lunak

Perangkat	Jenis Perangkat	Spesifikasi
Perangkat Keras	Prosesor	Intel(R) Core(TM) i5-337U CPU @ 1.80 GHz
	Memori	8 GB 1600 MHz DDR3
Perangkat Lunak	Sistem Operasi	Windows 10 Home Single Language
	Perangkat Pengembang	Eclipse Luna , ANTLR V4, dan PostgreSQL V9.3

4.2 Implementasi

Pada sub bab implementasi ini menjelaskan mengenai pembangunan perangkat lunak secara detail dan menampilkan Kode Sumber yang digunakan mulai tahap praproses hingga klasifikasi. Pada tugas akhir ini data yang digunakan seperti yang telah dijelaskan di bab sebelumnya yaitu dua jenis kode program Mahasiswa dalam satu kelas yang mengambil mata kuliah Pemrograman Berorientasi Objek (PBO).

4.2.1 *Parsing* kode program menggunakan ANTLR

Hal pertama yang dilakukan pada tugas akhir ini adalah mem-*parsing* kode program mahasiswa ke dalam bentuk AST *(Abstrack Syntax Tree)* menggunakan fungsi *Listener* yang ada pada ANTLR. Kode program mahasiswa yang digunakan sebagai data masukan dari sistem ini, menggunakan bahasa pemrograman C++ dan berorientasikan objek. Untuk memparsing kode program mahasiswa melalui *ANTLR parser* harus menggunakan suatu aturan yang disebut *grammar*. Grammar yang digunakan adalah *grammar C++* yang telah disediakan oleh ANTLR. Kode program mahasiswa dibaca melalui Kode Sumber 4.1 dan di-*parsing* menggunakan fungsi *Listener* pada ANTLR melalui Kode Sumber 4.2.

```
1  File[] files = new File("E:/TA/Projects/antlr4-
   project/src/main/" + path_dataset +"/").listFiles()
2
3  for (File file : files)
4        if (file.isFile())
5              CPP14Lexer lexer = new
               CPP14Lexer(new ANTLRInputStream(
               Main.class.getResourceAsStream("/"
               + file.getName())))

6              CPP14Parser parser= new CPP14Parser(new
               CommonTokenStream(lexer)
```

Kode Sumber 4.1 Kode Program membaca file dan pemanggilan *lexer* dan *parser* ANTLR

```
1  ParseTree tree = parser.translationunit()
2  ParseTreeWalker walker = new ParseTreeWalker()
3  MyCPPListener listener = new MyCPPListener()
4  walker.walk(listener,tree)
```

Kode Sumber 4.2 Kode Program pemanggilan fungsi *Listener*

Kode Sumber 4.1 membaca seluruh file .cpp pada direktori dan menginisialisasi pemanggilan *lexer* dan *parser* pada ANTLR. Sedangkan Kode Sumber 4.2 melakukan pemanggilan fungsi *Listener* pada *ANTLR parser* dan menjalankan fungsi `walk` yang ada pada *Listener*. Fungsi `walk` tersebut akan berjalan mengunjungi setiap bagian *node* pada sebuah *Abstract Syntax Tree* (AST) yang telah dihasilkan oleh ANTLR *parser*. AST yang dihasilkan adalah bentuk *tree* yang merupakan representasi dari pemanggilan isi yang ada pada *grammar C++*.

4.2.2 Pembentukan *Sequence* kode program

Abstract Syntax Tree (AST) yang dihasilkan dari *grammar C++* sangat banyak dan menghasilkan cakupan yang terlalu luas untuk diidentifikasi. Oleh karena itu tidak semua isi pada *grammar C++* akan digunakan dalam proses deteksi plagiarisme. Sehingga hasil dari *Abstract Syntax Tree* (AST) tersebut di minimalkan dengan memodifikasi pemanggilan fungsi *Listener* yang ada pada kelas *MyCppListener* untuk diubah menjadi bentuk *sequence* yang siap dioalah dan disimpan ke dalam basis data. Fungsi *Listener* dimodifikasi agar menghasilkan bentuk *sequence* yang diinginkan, dengan terlebih dahulu melakukan inisialisasi variable-variabel yang akan digunakan, melalui Kode Sumber 4.3.

Kode Sumber 4.3 menjelaskan tentang inisialisasi variabel-variabel yang digunakan pada pembentukan *sequence* sebuah kode program pada fungsi *Listener*. Variabel-variabel tersebut digunakan untuk menandai bahwa *Listener* sedang mengunjungi suatu *node* atau telah selesai mengunjungi suatu *node* pada AST. Selain itu ada beberapa variabel yang digunakan untuk menyimpan beberapa isi *node* pada AST, yang selanjutnya digunakan untuk membentuk isi dari sebuah *sequence*.

```
1   initialize level as an empty StringBuffer
2   initialize ast as an empty StringBuffer
3   initialize CountLevel = 0
4   initialize flagClass = 0
5   initialize flagExpression = 0
6   initialize flagPostfix = 0
7   initialize flagCondition = 0
8   initialize flagSwitchcase = 0
9   initialize flagIf = 0
10  initialize flagIterDowhile = 0
11  initialize flagShift = 0
12  initialize flagShift2 = 0
13  initialize flagParametercall = 0
14  initialize flagParameterpostfix = 0
15  initialize flagString = 0
16  initialize flagBrace = 0
```

Kode Sumber 4.3 Kode program inisialisasi variabel pada Listener

Variabel ***level*** dan ***ast*** digunakan untuk menyimpan tiap bentuk *sequence* dan level *sequence* yang terbentuk pada setiap pemanggilan fungsi Listener. Variabel ***CountLevel*** digunakan untuk meng-*update* nilai level pada setiap pemanggilan fungsi *Listener*. Variabel ***flagClass*** digunakan untuk menandai bahwa *Listener* sedang mengunjungi suatu deklarasi kelas. Variabel ***flagExpression*** digunakan untuk mengidentifikasi adanya *assignment* pada *Listener* yang diikuti oleh sebuah *assignment operator*. Variabel ***flagPostfix*** digunakan untuk menandai adanya penggunaan postfix pada kode program.

Variabel ***flagCondition*** digunakan untuk menandai bahwa saat ini *Listener* sedang mengunjungi *fungsi condition* pada Listener (kondisi dari *if-else, while, for, switch-case,* dan *do-while*). Variabel ***flagSwitchcase*** digunakan untuk menandai bahwa saai ini *Listener* sedang mengunjungi fungsi *Selectionstatement* berupa *switch-case*. Variabel ***flagIf*** digunakan untuk menandai bahwa saai ini *Listener* sedang mengunjungi fungsi *Selectionstatement* berupa *if-else*. Variabel ***flagIterDowhile***

digunakan untuk menandai bahwa saat ini *Listener* sedang mengunjungi fungsi *Iterationstatement* berupa *do*-while. Variabel ***flagShift*** digunakan untuk menandai bahwa *Listener* telah selesai mengunjungi deklarasi suatu shiftexpression. Variabel ***flagShift2*** digunakan untuk menandai bahwa *Listener* telah selesai mengunjungi *shiftexpression* berupa *cout* dan *cin*.

Variabel ***flagParametercall*** digunakan untuk menandai adanya parameter pada sebuah pemanggilan fungsi. Variabel ***flagParameterpostfix*** digunakan untuk menandai adanya parameter yang diikuti oleh postfix pada sebuah pemanggilan fungsi. Variabel ***flagString*** digunakan untuk menandai adanya penggunaan string pada kode program. Variabel ***flagBrace*** digunakan untuk menandai adanya deklarasi tipe data yang diikuti oleh suatu *assignment operator*.

Sedangkan modifikasi terhadap pemanggilan *grammar C++* dilakukan pada fungsi *Listener*. Modifikasi yang terlihat pada Kode Sumber 4.4 adalah modifikasi terhadap pemanggilan fungsi `Declaration` pada *grammar C++*. Fungsi tersebut mengidentifikasi adanya suatu deklarasi (umumnya diakhiri tanda " **;** ") pada kode program. Metode Listener sendiri memiliki 2 fungsi utama yaitu fungsi `enter` dan `exit`.

Fungsi `enterDeclaration` yang ada pada Kode Sumber 4.4 dijalankan otomatis ketika *Listener* mengunjungi/menemukan pendeklarasian tipe data, variabel, *assignment*, iterasi, fungsi, dan lain-lain. Modifikasi yang dilakukan adalah merepresentasikan hasil pemanggilan deklarasi-deklarasi tersebut dengan suatu kode (satu huruf atau rangkaian huruf) yang diikuti oleh *parenthesis* (tanda buka kurung). Pemanggilan fungsi `enterDeclaration` dilambangkan dengan huruf **A** dan diikuti dengan *parenthesis* buka kurung. Sedangkan fungsi `exitDeclaration` yang ada pada Kode Sumber 4.4 dijalankan otomatis setelah *Listener* selesai mengunjungi suatu pendeklarasian pada kode program mahasiswa. *Parenthesis* tutup kurung digunakan untuk menandai bahwa suatu

deklarasi pada kode program mahasiswa telah selesai dikunjungi oleh *Listener*.

Tanda *parenthesis* (buka kurung maupun tutup kurung) berfungsi sebagai penanda level tiap-tiap isi *sequence*. Penghitungan level didasarkan pada letak elemen tersebut di dalam *sequence*, elemen tersebut berada di dalam atau diluar *parenthesis.* Umumnya, apabila *Listener* mengunjungi/menemukan suatu deklarasi, maka akan diawali dengan *parenthesis* buka kurung yang diikuiti huruf **A** dan levelnya akan bertambah satu. Sedangkan apabila *Listener* telah selesai mengunjungi suatu deklarasi maka ditandai dengan *parenthesis* tutup kurung dan levelnya akan berkurang satu.

```
1   FUNCTION enterDeclaration(ctx)
2   CountLevel = 0
3   sequence = "A ( "
4   append sequence to ast
5   append CountLevel to level
6   CountLevel++
7
8   FUNCTION exitDeclaration(ctx)
9   sequence = ") "
10  IF countAccesspecifier > 0 THEN
11        CountLevel--
12        append sequence to ast
13        append CountLevel to level
14        countDeclaration = 0
15  END IF
16  CountLevel--
17  append sequence to ast
18  append CountLevel to level
```

Kode Sumber 4.4 Kode program fungsi `Declaration` pada Listener

```
1   FUNCTION enterClasspecifier(ctx)
2   flagClass = 1
3
4   FUNCTION enterClasskey(ctx)
5   sequence = "K:" + ctx.getText() + "( "
6   append sequence to ast
7   append CountLevel to level
8   CountLevel++
9
10  FUNCTION exitClasspecifier(ctx)
11  flagClass = 0
12  sequence = ") "
13  CountLevel--
14  append sequence to ast
15  append CountLevel to level
16
17  FUNCTION enterClassname(ctx)
18  nameofclass = ctx.getText()
19  IF nameofclass != "String" THEN
20        classname = nameofclass
21  END IF
```

Kode Sumber 4.5 Kode program fungsi `Classpecifier`, `Classkey`, dan `Classname` pada Listener

Kode Sumber 4.5 mengidentifikasi adanya pendeklarasian kelas/struct pada kode program, melalui fungsi `enterClasspecifier` untuk menandai adanya deklarasi suatu class/struct dan `enterClasskey` untuk membedakan antara deklarasi class dan struct. Fungsi `exitClasspecifier` digunakan untuk menandai bahwa *Listener* telah selesai mengunjungi suatu deklarasi class/struct. Modifikasi pada fungsi yang dilakukan sama dengan modifikasi pada fungsi `enterClasskey` dan `exitClasspecifier` sama dengan modifikasi `enterDeclaration` maupun `exitDeclaration` yang telah dijelaskan sebelumnya.

Sedangkan fungsi `enterClassname` pada Kode Sumber 4.5 digunakan untuk mengambil nama kelas dari suatu

deklarasi kelas pada kode program. Nama kelas ini nantinya akan digunakan untuk mengecek apakah terdapat fungsi yang namanya sama dengan nama kelasnya. Apabila teridentifikasi memiliki nama yang sama dengan nama kelasnya, maka fungsi tersebut masuk ke dalam jenis fungsi sebagai *constructor* atau *destructor*.

```
1   FUNCTION enterAccesspecifier
2   Accesspecifier = ctx.getText()
3   IF countAccesspecifier = 0 THEN
4         sequence = "L:" + Accesspecifier + "( "
5         append sequence to ast
6         append CountLevel to level
7         CountLevel++
8         countAccesspecifier = 1
9   ELSE IF countAccesspecifier > 0 THEN
10        CountLevel--
11        sequence = ") L:" + Accesspecifier + " ( "
12        append sequence to ast
13        append CountLevel to level
14        CountLevel++
15        countAccesspecifier = 0
16  END IF
```

Kode Sumber 4.6 Kode program fungsi `Accesspecifier` pada Listener

Kode Sumber 4.6 mengidentifikasi jenis *accesspecifier* (public, private, protected) yang dimiliki oleh kelas melalui fungsi `enterAccesspecifier`. Modifikasi yang dilakukan sama dengan modifikasi pada `enterDeclaration` maupun `exitDeclaration` yang sudah dijelaskan sebelumnya. Namun pada fungsi `enterAccesspecifier` telah mencakup modifikasi saat *Listener* selesai mengunjungi deklarasi *accesspecifier* suatu kelas.

```
1   FUNCTION enterBasespecifier(ctx)
2         sequence = "M "
3         append sequence to ast
4         append CountLevel to level
5         CountLevel++
```

Kode Sumber 4.7 Kode program fungsi `Basespecifier` pada listener

Kode Sumber 4.7 mengidentifikasi adanya deklarasi kelas turunan pada kode program melalui fungsi `enterBasepecifier`. Modifikasi yang dilakukan sama dengan modifikasi pada `enterDeclaration` yang sudah dijelaskan sebelumnya.

```
1   FUNCTION enterSimpletypespecifier(ctx)
2   typespecifier = ctx.getText()
3
4   FUNCTION enterDeclarator(ctx)
5   Declarator = ctx.getText()
6   IF (Declarator = classname AND flagClass != 0) THEN
7         sequence = "B:N "
8   ELSE IF Declarator = ("~" + classname) THEN
9         sequence = "B:NN "
10  ELSE
11        sequence = "B:" + typespecifier + " "
12  END IF
13  append sequence to ast
14  append CountLevel to level
15
16  FUNCTION exitDeclarator(ctx)
17  sequence = " "
18  append sequence to ast
```

Kode Sumber 4.8 Kode program fungsi `Simpletypespecifier` dan `Declarator` pada Listener

Kode Sumber 4.8 mengidentifikasi adanya deklarasi suatu tipe data dari suatu variabel pada kode program. Tipe data

diidentifikasi melalui fungsi `enterSimpletypespecifier`, sedangakan variabel dari tipe data tersebut diidentifikasi melalui fungsi `enterDeclarator` dan `exitDeclarator`. Fungsi `enterDeclarator` juga digunakan untuk mengidentifikasi adanya penggunaan *constructor* dan *destructor* pada kelas. Untuk pembentukan *parenthesis* pada *sequence* dan level *sequence* sama dengan fungsi-fungsi yang telah dijelaskan sebelumnya.

```
1  FUNCTION enterPtroperator(ctx)
2  sequence = ctx.getText()
3  append sequence to ast
```

Kode Sumber 4.9 Kode program fungsi `Ptroperator` pada Listener

Kode Sumber 4.9 mengidentifikasi adanya penggunaan pointer pada kode program. Pointer tersebut diidentifikasi melalui fungsi `enterPtroperator`.

```
1  FUNCTION enterFunctiondefinition(ctx)
2  sequence = "C ( "
3  append sequence to ast
4  append CountLevel to level
5  CountLevel++
6
7  FUNCTION exitFunctiondefinition(ctx)
8  sequence = ") "
9  CountLevel--
10 append sequence to ast
11 append CountLevel to level
```

Kode Sumber 4.10 Kode program fungsi `Functiondefinition` pada Listener

Kode Sumber 4.10 mengidentifikasi adanya pendeklarasian suatu fungsi pada kode program melalui fungsi `enterFunctiondefinition` dan `exitFunctiondefinition`. Modifikasi yang dilakukan sama

dengan modifikasi pada `enterDeclaration` maupun `exitDeclaration` yang sudah dijelaskan sebelumnya.

```
1   FUNCTION enterParametersandqualifiers(ctx)
2   sequence = " D ( "
3   append sequence to ast
4   append CountLevel to level
5   CountLevel = CountLevel + 2
6
7   FUNCTION exitParametersandqualifiers(ctx)
8   sequence = ")"
9   CountLevel = CountLevel - 2
10  append sequence to ast
11  append CountLevel to level
```

Kode Sumber 4.11 Kode Program fungsi `Parametersandqualifiers` pada Listener

Kode Sumber 4.11 mengidentifikasi parameter yang dimiliki suatu fungsi melalui fungsi `enterParametersandqualifiers` dan `exitParametersandqualifiers`. Modifikasi yang dilakukan hampir sama dengan modifikasi pada *`enterDeclaration`* maupun `exitDeclaration` yang sudah dijelasakan sebelumnya. Perbedaan hanya terletak pada penghitungan level *sequence*, jika sebelumnya levelnya di *increment* satu, pada fungsi ini level *sequence* ditambah 2.

Kode Sumber 4.12 mengidentifikasi isi dari suatu fungsi pada kode program melalui fungsi `enterFunctionbody` dan `exitFunctionbody`. Modifikasi yang dilakukan sama dengan modifikasi pada `enterDeclaration` maupun `exitDeclaration` yang sudah dijelaskan sebelumnya.

```
1   FUNCTION enterFunctionbody(ctx)
2   sequence = "E ( "
3   append sequence to ast
4   append CountLevel to level
5   CountLevel++
6
7   FUNCTION enterFunctionbody(ctx)
8   CountLevel--
9   sequence = ") "
10  append sequence to ast
11  append CountLevel to level
```

Kode Sumber 4.12 Kode program fungsi `Functionbody` pada Listener

Kode Sumber 4.13 mengidentifikasi adanya penggunaan *if-else* dan *swtich-case* pada kode program melalui fungsi `enterSelectionstatement` dan `exitSelectionstatement`. Modifikasi yang dilakukan hampir sama dengan modifikasi pada `enterDeclaration` maupun `exitDeclaration` yang sudah dijelasakan sebelumnya. Namun pada fungsi ini dilakukan pengecekan apakah suatu kode program menggunakan *if-else* atau *swtich-case.* Untuk membedakan antara *if-else* dan *switch-case*, level *sequence* pada penggunaan *if-else* ditambah 2, sedangkan level *sequence* pada penggunaan *Switch-case* ditambah 3.

Selain itu diberi tanda berupa variabel *flagIf* dan *flagSwitchcase* untuk membedakan kode huruf *sequence* antara penggunaan *if-else* dengan *swtich-case*. Pada penggunaan *swtich-case* diberi huruf "**G**" pada rangkaian huruf kode *sequence*-nya, untuk membedakan antara penggunaan *swtich-case* dan *if-else* agar lebih akurat.

```
1   FUNCTION enterSelectionstatement(ctx)
2   selectstat = ctx.getText()
3   split selectstat by "(" and store in array[]
4   sequence = "H: " + array[0] + "("
5   append sequence to ast
6   append CountLevel to level
7   IF array[0] contains "if" THEN
8         CountLevel = CountLevel + 2
9         flagIf = 1
10  ELSE IF array[0] contains "switch" THEN
11        CountLevel = CountLevel + 3
12        flagSwitchcase = 1
13  END IF
14
15  FUNCTION exitSelectionstatement(ctx)
16  selectstat = ctx.getText()
17  split selectstat by "(" and store in array[]
18  IF array[0] contains "switch" THEN
19        CountLevel = CountLevel - 3
20        flagSwitchcase = 0
21  ELSE
22        CountLevel = CountLevel - 2
23        flagIf = 0
24  END IF
25  sequence = ") "
26  append sequence to ast
27  append CountLevel to level
```

Kode Sumber 4.13 Kode program fungsi `Selectionstatement` pada Listener

Fungsi `enterSelectionstatement` pada Kode Sumber 4.13 dipanggil saat *Listener* mengunjungi suatu deklarasi *switch-case* atau *if-else*. Sedangkan fungsi `exitSelectionstatement` dipanggil saat *Listener* selesai mengunjungi suatu deklarasi *switch-case* atau *if-else* pada kode program yang di*parsing*.

```
1   FUNCTION enterJumpstatement(ctx)
2   jumpstat = ctx.getText()
3   IF (flagSwitchcase > 0 AND flagIf = 0) THEN
4         sequence = "G" + flagSwitchcase +
5   jumpstat
6         flagSwitchcase++
7   ELSE
8         sequence = jumpstat
9   END IF
10  append sequence to ast
11  append CountLevel to level
```

Kode Sumber 4.14 Kode program fungsi `Jumpstatement` pada Listener

Kode Sumber 4.14 menjelaskan tentang penggunaan ***break*** pada kode program melalui fungsi `enterJumpstatement`. Modifikasi yang dilakukan hampir sama dengan modifikasi pada `enterDeclaration` yang sudah dijelasakan sebelumnya. Fungsi `enterJumpstatement` juga digunakan untuk membedakan *sequence* maupun level *sequence* milik *case* satu dengan *case* lainnya pada penggunaan *switch-case*.

Kode Sumber 4.15 menjelaskan tentang penggunaan iterasi pada kode program melalui fungsi `enterIterationstatement` dan `exitIterationstatement`. Modifikasi yang dilakukan hampir sama dengan modifikasi pada `enterDeclaration` maupun `exitDeclaration` yang sudah dijelaskan sebelumnya. Namun pada fungsi ini dilakukan pengecekan apakah suatu kode program menggunakan iterasi berupa *for, while,* atau *do-while.* Namun dengan menggunakan *grammar C++*, *parser* tidak bisa mengidentifikasi kondisi dari iterasi berupa *do-while*, sehingga diperlukan sedikit modifikasi pada isi *grammar* yang dijelaskan pada Gambar 4.1, yaitu pada bagian yang diberi *highlight* berwarna biru.

```
1   FUNCTION enterIterationstatement(ctx)
2   iterstat = ctx.getText()
3   split iterstat by "(" and store in array[]
4   IF array[0] = "while" THEN
5         sequence = "I:while ( "
6   ELSE IF array[0] = "for" THEN
7         sequence = "I:for ( "
8   ELSE
9         flagIterDowhile = 1
10        sequence = "I:do ( "
11  END IF
12  append sequence to ast
13  append CountLevel to level
14  CountLevel++
15  countIter = 1
16
17  FUNCTION exitIterationstatement(ctx)
18  flagIterDowhile = 0
19  sequence = ") "
20  CountLevel--
21  append sequence to ast
22  append CountLevel to level
```

Kode Sumber 4.15 Kode program fungsi `Iterationstatement` pada Listener

```
iterationstatement
:
    While '(' condition ')' statement
    | Do statement While '(' condition ')' ';'
    | For '(' forinitstatement condition? ';' expression? ')' statement
    | For '(' forrangedeclaration ':' forrangeinitializer ')' statement
;
```

Gambar 4.1 Modifikasi *do-while* pada isi Grammar C++

```
1   FUNCTION enterCondition(ctx)
2   flagCondition = 1
3   sequence = "( "
4   append sequence to ast
5   append CountLevel to level
6   CountLevel++
7   Condition[] = ctx.getText()
8   FOR i = 0 to size of condition
9         IF condition[i] = '<' THEN
10              IF condition[i+1] = '=' THEN
11                    sequence = "J<= "
12              ELSE
13                    sequence = "J< "
14              END IF
15        ELSE IF (condition[i] = '>') AND
                  (condition[i-1]!= '-') THEN
16              IF condition[i+1] = '=' THEN
17                    sequence = "J>= "
18              ELSE
19                    sequence = "J> "
20              END IF
21        ELSE IF condition[i] = '=' THEN
22              IF condition [i+1] = '=' THEN
23                    sequence = "J== "
24                    i = i+2
25              ELSE
26                    sequence = "J= "
27              END IF
28        ELSE IF condition[i] = '!' THEN
29              sequence = "J!= "
30              i = i+2
31        ELSE IF condition[i] = '&' THEN
32              sequence = "J&& "
33              i = i+2
34        ELSE IF condition[i] = '|' THEN
35              sequence = "J|| "
36              i = i+2
37        ELSE IF condition[i] = '+' THEN
38              IF condition[i+1] = '+' THEN
39                    i = i+2
```

```
40              ELSE
41                    sequence = "J+ "
42              END IF
43        ELSE IF condition[i] = '-' THEN
44              IF condition[i+1] = '-' THEN
45                    i = i+2
46              ELSE
47                    sequence = "J- "
48              END IF
49        ELSE IF condition[i] = '*' THEN
50              sequence = "J* "
51        ELSE IF condition[i] = '/' THEN
52              sequence = "J/ "
53        ELSE IF condition[i] = '%' THEN
54              sequence = "J% "
55        END IF
56        append sequence to ast
57        append CountLevel to level
58  END LOOP
59
60  FUNCTION exitCondition(ctx)
61        flagCondition = 0
62        sequence = ") "
63        CountLevel--
64        append sequence to ast
65        append CountLevel to level
```

Kode Sumber 4.16 Kode program fungsi `Condition` pada Listener

Kode Sumber 4.16 mengidentifikasi adanya penggunaan kondisi didalam *selectionsatatement* maupun *iterationstatement* melalui fungsi `enterCondition` dan `exitCondition`. Modifikasi yang dilakukan pada fungsi `enterCondition` hampir sama dengan modifikasi pada `enterDeclaration` yang sudah dijelasakan sebelumnya. Namun pada fungsi ini dilakukan pengecekan tiap elemen pada kondisi untuk mengidentifikasi adanya operator **= , != , ==, >= , <=, > , <, AND,**

OR, dan adanya operator penjumlahan , pengurangan, perkalian, maupun pembagian.

Sedangkan fungsi `exitCondition` mengidentifikasi bahwa *Listener* telah selesai mengunjungi suatu kondisi didalam *selectionsatatement* maupun *iterationstatement.* Modifikasi yang dilakukan hampir sama dengan modifikasi pada `exitDeclaration` yang sudah dijelaskan sebelumnya.

```
1  FUNCTION enterExpressionlist(ctx)
2  flagParametercall++
3
4  FUNCTION exitExpressionlist(ctx)
5  flagParametercall = 0
6  flagParameterpostfix = 0
```

Kode Sumber 4.17 Kode program fungsi `Expressionlist` pada Listener

Kode Sumber 4.17 mengidentifikasi adanya penggunaan parameter dalam pemanggilan suatu fungsi pada kode program melalui fungsi `enterExpressionlist` dan `exitExpressionlist`. Modifikasi yang dilakukan adalah menandai sebanyak berapa jumlah parameter yang digunakan melalui fungsi `enterExpressionlist` dan menandai bahwa *Listener* telah selesai mengunjungi suatu parameter melalui fungsi `exitExpressionlist`.

```
1  FUNCTION enterShiftexpression(ctx)
2  flagshift++
3  shiftexpress = ctx.getText()
4  IF shifexpress contains "<<" OR ">>" THEN
5        flagshift2 = 1
6  END IF
```

Kode Sumber 4.18 Kode program fungsi `Shiftexpression` pada Listener

Kode sumber 4.18 mengidentifikasi adanya penggunaan fungsi `cout` dan `cin` pada kode program melalui fungsi `enterShiftexpression`.

```
1   FUNCTION enterPostfixexpression(ctx)
2   IF (flagShift > 0 AND flagShift2 = 1) THEN
3         flagShift--
4   ELSE
5         postfixexpres[] = ctx.getText()
6         flagString = 0
7         FOR i = 0 to size of postfixexpress
8               IF postfixexpress[i]  = '"' THEN
9                     flagString = 1
10                    break
11              ELSE IF postfixexpress[i] = '(' THEN
12                    break
13              END IF
14        END LOOP
15  END IF
```

Kode Sumber 4.19 Kode program fungsi `Postfixexpression` pada Listener (bagian 1)

Kode Sumber 4.19 mengidentifikasi adanya penggunaan postfix berupa *access operator* (**.**) dan (→) , pemanggilan fungsi (**( )**), *decrement* (**--**) , dan *increment* (++) pada kode program melalui fungsi `enterPostfixexpression`. Selain itu modifikasi juga dilakukan untuk mengidentifikasi adanya penggunaan *string* dan pengecekan terhadap penggunaan *shiftexpression* berupa fungsi `cout` dan `cin`. Identifikasi pada penggunaan string dilakukan dengan memberi tanda berupa variabel *flagString*. Sedangkan identifikasi terhadap penggunaan fungsi `cout` dan `cin` adalah dengan menghitung berapa kali fungsi `cout` dan `cin` dideklarasikan.

```
1   IF flagString = 0 THEN
2         IF (regexPattern2_1 = match) OR
3            (regexPattern2_2 = match) THEN
4               flagPostfix = 2
5               IF (flagSwitchcase > 0 AND
                (flagCondition=0) AND
                (flagExpression!=1) AND
                (flagParametercall = 0) THEN
6                     sequence = "G" +
7                     flagSwitchcase + "VO"
8                     append sequence to ast
9                     append CountLevel to level
10              ELSE IF (flagExpression!=1) AND
                (flagCondition = 0) AND
                (flagParametercall = 0)
11                    sequence = "VO"
12                    append sequence to ast
13                    append CountLevel to level
14              ELSE
15                    postfix = "O"
16              END IF
```

Kode Sumber 4.20 Kode program fungsi `Postfixexpression` pada listener (bagian 2)

Kode Sumber 4.20 mengidentifikasi adanya penggunaan postfix berupa pemanggilan fungsi yang ditandai dengan tanda buka dan tutup kurung seperti berikut **( )**. Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

RegexPattern2_1	`^(?:[a-zA-Z0-9]{0,61}[a-zA-Z0 9](?:\\((?:\\)))*$`
RegexPattern2_2	`^(?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9](?:\\((?:[a-zA-Z0-9(),.->+\"']{0,61}(?:[a-zA-Z0-9,>+/*().\"\']{0,61}[a-zA-Z0-9(),.->+\"\'])\\))))*$`

```
1   IF flagString = 0 THEN
2         IF (regexPattern3_1 = match) OR
3            (regexPattern3_2 = match) THEN
4               flagpostfix = 3
5               Postfix = ".O"
                IF (flagSwitchcase > 0 AND
                (flagCondition=0) AND
                (flagExpression!=1) AND
6               (flagParametercall = 0) THEN
7                     sequence = "G" +
8                     flagSwitchcase + "V.O"
9                     append sequence to ast
10                    append CountLevel to level
                ELSE IF (flagExpression!=1) AND
                (flagCondition = 0) AND
11              (flagParametercall = 0) THEN
12                    sequence = "V.O"
13                    append sequence to ast
14                    append CountLevel to level
15                    postfix = NULL
16              ELSE if flagParametercall = 1
17                    flagParameterpostfix = 1
18              END IF
```

Kode Sumber 4.21 Kode program fungsi `Postfixexpression` pada Listener (bagian 3)

Kode Sumber 4.21 mengidentifikasi adanya penggunaan postfix berupa *access operator* (**.**) yang diikuti dengan pemanggilan fungsi **( ).** Misal Bounce**.**makan() atau Bounce**.**makan(a,b) . Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

regexPattern3_1	^(?:[a-zA-Z0-9\\[\\]]{0,61}[a-zA-Z0-9](?:\\.[a-zA-Z0-9]{0,61}[a-zA-Z0-9])?(?:\\((?:\\))))*$
regexPattern3_2	^(?:[a-zA-Z0-9\\[\\]]{0,61}[a-zA-Z0-9](?:\\.[a-zA-Z0-9]{0,61}[a-zA-Z0-9])?(?:\\((?:[a-zA-Z0-

	9(),.>+\"']{0,61}(?:[a-zA-Z0-9,->+/*().]{0,61}[a-zA-Z0-9(),.->+\"'])\\\))))*$

```
1   IF flagString = 0 THEN
2         IF (regexPattern3_11 = match) OR
3            (regexPattern3_22 = match) THEN
4               flagpostfix = 3
5               Postfix = "->O"
                IF (flagSwitchcase > 0 AND
                (flagCondition=0) AND
                (flagExpression!=1) AND
6               (flagParametercall = 0) THEN
7                     sequence = "G" +
8                     flagSwitchcase + "V->O"
9                     append sequence to ast
10                    append CountLevel to level
                ELSE IF (flagExpression!=1) AND
                (flagCondition = 0) AND
11              (flagParametercall = 0) THEN
12                    sequence = "V->O"
13                    append sequence to ast
14                    append CountLevel to level
15                    postfix = NULL
16              ELSE if flagParametercall = 1
17                    flagParameterpostfix = 1
                END IF
```

Kode Sumber 4.22 Kode program fungsi `Postfixexpression` pada Listener (bagian 3)

Kode Sumber 4.22 mengidentifikasi adanya penggunaan postfix berupa *access operator* (→) yang diikuti dengan pemanggilan fungsi **( ).** Misal Bounce→makan() atau Bounce→makan(a,b) . Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

RegexPattern3_11	^(?:[a-zA-Z0-9\\[\\]]{0,61}[a-zA-Z0-9](?:\\-\\>[a-zA-Z09]{0,61}[a-zA-Z0-9])?(?:\\((?:\\))))*$
RegexPattern3_22	^(?:[a-zA-Z0-9\\[\\]]{0,61}[a-zA-Z0-9](?:\\-\\>[a-zA-Z09]{0,61}[a-zA-Z0-9])?(?:\\((?:[a-zA-Z0-9(),.->+_\"']{0,61}(?:[a-zA-Z0-9,->+/*_().\"']{0,61}[a-zA-Z0-9(),.->+_\"'])\\))))*$

Kode Sumber 4.23 mengidentifikasi adanya penggunaan postfix berupa *access operator* (**.**) yang diikuti dengan access operator lainnya yaitu (**.**) atau (→), lalu diikuti pemanggilan fungsi **( ).** Misal Bounce→minum**.**makan() atau Bounce→minum.makan(a,b) . Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

RegexPattern3_111	^(?:[a-zA-Z0-9->.\\[\\]]{0,61}[a-zA-Z0-9](?:\\.[a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9]))?(?:\\((?:\\))))*$
RegexPattern3_222	^(?:[a-zA-Z0-9->.\\[\\]]{0,61}[a-zA-Z09](?:\\.[a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9]))?(?:\\((?:[a-zA-Z0-9(),.->+\"']{0,61}(?:[a-zA-Z0-9,->+/*().]{0,61}[a-zA-Z0-9(),.->+\"'])\\))))*$

Banyak kemungkinan yang terjadi apabila tidak menggunakan batasan-batasan yang telah dideklarasikan menggunakan *regex*. Apabila tidak menggunakan aturan *regex*, semua postfix yang tidak sesuai dengan keinginan masuk ke dalam kategori *postfixexpression* yang diinginkan sebelumnya.

```
1   IF flagString = 0 THEN
2         IF (regexPattern3_111 = match) OR
3            (regexPattern3_222 = match) THEN
4               flagpostfix = 3
5               Postfix = ".O"
                IF (flagSwitchcase > 0 AND
                (flagCondition=0) AND
                (flagExpression!=1) AND
6               (flagParametercall = 0) THEN
7                     sequence = "G" +
8                     flagSwitchcase + "V->O"
9                     append sequence to ast
10                    append CountLevel to level
                ELSE IF (flagExpression!=1) AND
                (flagCondition = 0) AND
11              (flagParametercall = 0) THEN
12                    sequence = "V->O"
13                    append sequence to ast
14                    append CountLevel to level
15                    postfix = NULL
16              ELSE if flagParametercall = 1
17                    flagParameterpostfix = 1
18              END IF
```

Kode Sumber 4.23 Kode program fungsi `Postfixexpression` pada Listener (bagian 4)

Kode Sumber 4.23 mengidentifikasi adanya penggunaan postfix berupa *access operator* (→) yang diikuti dengan access operator lainnya yaitu (**.**) atau (→), lalu diikuti pemanggilan fungsi **( ).** Misal Bounce**.**minum→makan() atau Bounce**.**minum→makan(a,b) . Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut :

RegexPattern3_1111	`^(?:[a-zA-Z0-9-`
	`>.\\[\\]]{0,61}[a-zA-Z0-9](?:`
	`\\-\\> [a-zA-Z0-9](?:[a-zA-Z0-`
	`9]{0,61}[a-zA-Z0-`
	`9]))?(?:\\((?:\\))))*$`

RegexPattern3_2222	^(?:[a-zA-Z0-9->.\\[\\]]{0,61}[a-zA-Z0-9](?:\\-\\> [a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9]))?(?:\\((?:[a-zA-Z0-9(),.->+\"']{0,61}(?:[a-zA-Z0-9,->+/*().]{0,61}[a-zA-Z0-9(),.->+\"'])\\))))*$

```
1   IF flagString = 0 THEN
2         IF (regexPattern3_111 = match) OR
3            (regexPattern3_222 = match) THEN
4               flagpostfix = 3
5               Postfix = ".O"
                IF (flagSwitchcase > 0 AND
                (flagCondition=0) AND
                (flagExpression!=1) AND
6               (flagParametercall = 0) THEN
7                     sequence = "G" +
8                     flagSwitchcase + "V->O"
9                     append sequence to ast
10                    append CountLevel to level
                ELSE IF (flagExpression!=1) AND
                (flagCondition = 0) AND
11              (flagParametercall = 0) THEN
12                    sequence = "V->O"
13                    append sequence to ast
14                    append CountLevel to level
15                    postfix = NULL
16              ELSE if flagParametercall = 1
17                    flagParameterpostfix = 1
                END IF
```

Kode Sumber 4.24 Kode program fungsi `Postfixexpression` pada Listener (bagian 5)

```
1  ELSE IF regexPattern0 = match THEN
2        flag_postfix = 0
3        temp_postfix = null
4        postfix=null
5  END IF
```

Kode Sumber 4.25 Kode program fungsi `Postfixexpression` pada Listener (bagian 6)

Kode Sumber 4.25 mengidentifikasi adanya penggunaan postfix berupa koma pada bilangan. Misal **2.5** atau **0.02**. Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

RegexPattern0	^(?:[0-9]{0,61}[0-9](?:\\.[0-9]{0,61}[0-9]))*$

```
1  ELSE IF regexPattern11 = match THEN
2        IF (flagPostfix != 3 OR
         flagParametercall = 1) AND
         flagParameterpostfix = 0 THEN
3              flagPostfix = 1
4              postfix = "."
5        END IF
6  END IF
```

Kode Sumber 4.26 Kode program fungsi `Postfixexpression` pada Listener (bagian 7)

Kode Sumber 4.26 mengidentifikasi adanya penggunaan postfix berupa *access operator* (**.**) yaitu misal Bounce**.**makan. Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

RegexPattern11	^(?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9](?:\\.[a-zA-Z0-9]{0,61}[a-zA-Z0-9]))*$

```
1   ELSE IF regexPattern11 = match THEN
2         IF (flagPostfix != 3 OR
          flagParametercall = 1) AND
          flagParameterpostfix = 0 THEN
3               flagPostfix = 1
4               postfix = "."
5         END IF
6   END IF
```

Kode Sumber 4.27 Kode program fungsi `Postfixexpression` pada Listener (bagian 8)

Kode Sumber 4.27 mengidentifikasi adanya penggunaan postfix berupa *access operator* (→ **)** yaitu misal Bounce→makan. Identifikasi dilakukan menggunakan aturan *regex* yang dituliskan sebagai berikut.

RegexPattern1	`^(?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9](?:\\-\\>[a-zA-Z0-9]{0,61}[a-zA-Z0-9]))*$`

Kode Sumber 4.28 mengidentifikasi adanya penggunaan suatu ***decrement*** dan ***increment*** pada kode program. Identifikasi dilakukan dengan melakukan pengecekan, apakah postfix tersebut mengandung *string* berupa "--" atau " ++ ". Jika postfix tersebut mengandung *string* berupa "--" , maka diidentifikasikan sebagai sebuah *decrement*. Sedangkan jika postfix tersebut mengadung *string* berupa "++" , maka diidentifikasikan sebagai sebuah *increment.*

Pengecekan penggunaan *increment* dan *decrement* juga dilakukan pada sebuah kondisi di dalam *while* dan *for* yang umumnya menggunakan postfix berupa "++" atau "--".

```
1   ELSE IF postfixexpress contains "--" THEN
2         IF (flagSwitchcase > 0 AND
          flagCondition = 0) THEN
3               sequence = "G" + flagSwitchcase
                + "V--"
4               append sequence to ast
5               append CountLevel to level
6               postfix = NULL
7         ELSE IF flagCondition = 0 THEN
8               sequence = "V--"
9               append sequence to ast
10              append CountLevel to level
11        ELSE IF flagCondition = 1 THEN
12              postfix = "--"
13              flagPostfix = 6
14        END IF
15
16  ELSE IF postfixexpress contains "++" THEN
17        IF (flagSwitchcase > 0 AND
          flagCondition = 0) THEN
18              sequence = "G" + flagSwitchcase
                + "V++"
19              append sequence to ast
20              append CountLevel to level
21              postfix = NULL
22        ELSE IF flagCondition = 0 THEN
23              sequence = "V++"
24              append sequence to ast
25              append CountLevel to level
26        ELSE IF flagCondition = 1 THEN
17              postfix = "++"
28              flagPostfix = 5
29        END IF
```

Kode Sumber 4.28 Kode program fungsi `Postfixexpression` pada Listener (bagian 9)

```
1   FUNCTION exitPostfixexpression(ctx)
2   IF flagPostfix = 1 THEN
        flagPostfix = -1
    END IF
3   IF flagShift = 0 THEN
4       flagShift2 = 0
5   END IF
```

Kode Sumber 4.29 Kode program fungsi `Postfixexpression` pada Listener (bagian 8)

Kode Sumber 4.29 mengidentifikasi bahwa *Listener* telah selesai mengunjungi sebuah *node* berupa *Postfixexpression* melalui fungsi `exitPostfixexpression`.

Kode Sumber 4.30 mengidentifikasi adanya deklarasi suatu *assignment* yang diikuti oleh *simpletypespecifier* melalui fungsi `enterBraceorequalinitializer` dan `exitBraceorequalinitializer`. *Braceorequalinitializer* melakukan pengecekan apabila sebuah *assignment operator* berupa (=) diikuti oleh suatu tipe data yang dimiliki oleh variabel disebelah kiri dari *assignment operator.* Misal int a = b + 1.

Pada fungsi `enterBraceorequalinitializer`, modifikasi dilakukan dengan melakukan pengecekan terhadap penggunaan operasi tambah, kurang, kali, bagi, dan mod yang dideklarasikan setelah *assignment operator* ditulis. Sedangkan pengecekan apakah *braceorequalinitializer* telah benar-benar selesai dikunjungi oleh *Listener*, dijalankan melalui fungsi `exitBraceorequalinitializer`. Modifikasi yang dilakukan pada `enterBraceorequalinitializer` hampir sama dengan modifikasi pada `enterAssignmentoperator`. Sedangkan modifikasi pada `exitBraceorequalinitializer` sama dengan modifikasi pada `exitDeclaration` yang sudah dijelaskan sebelumnya

```
1   FUNCTION enterBraceorequalinitializer(ctx)
2   bracexpress = ctx.getText()
3   split bracexpress by "=" and store in statement[]
4   operator = "="
5   flagBrace = 1
6   sequence = "( "
7   append sequence to ast
8   append CountLevel to level
9   CountLevel++
10  FOR i = 0 to size of statement
11        IF statement[i] = '+' THEN
12              sequence = operator + "+ "
13        ELSE   IF   statement[i]   =   '-'   AND
          statement[i+1] != '>' THEN
14              sequence = operator + "- "
15        ELSE IF statement[i] = '/' THEN
16              sequence = operator + "/ "
17        ELSE IF statement[i] = '%' THEN
18              sequence = operator + "% "
19        ELSE IF statement[i] = '*' THEN
20              sequence = operator + "* "
21        ELSE IF statement[i] = '"' THEN
22              Break
23        END IF
24        append sequence to ast
25        append CountLevel to level
26  END LOOP
27
28  FUNCTION exitBraceorequalinitializer(ctx)
29  flagBrace = 0
30  sequence = ") "
31  Count_level--
32  append sequence to ast
33  append CountLevel to level
```

Kode Sumber 4.30 Kode program fungsi `Braceorequalinitializer` pada Listener

```
1   FUNCTION enterAssignmentexpression(ctx)
2   assignmentexpress = ctx.getText()
3   split assignmentexpress by "=" and store in
    expression
4
5   FUNCTION exitAssignmentexpression(ctx)
6   IF flagExpression == 1 AND flagparamcall== 0
7         flagExpression = 0
8         sequence = ") "
9         countLevel--
10        append sequence to ast
11        append CountLevel to level
12  END IF
13  flagPostfix = 0
14  flagPrimary = 0
15  flagString = 0
```

Kode Sumber 4.31 Kode program fungsi `Assignmentexpression` pada Listener

Kode Sumber 4.31 mengidentifikasi adanya deklarasi suatu *assignment* melalui fungsi `enterAssignmentexpression` dan `exitAssignmentexpression`. Identifikasi dilakukan apabila suatu *assignment* memiliki sebuah *operator assignment* melalui fungsi `enterAssignmentexpression`. Sedangkan pengecekan apakah suatu *assignment* telah benar-benar selesai dikunjungi oleh *Listener*, dilakukan melalui fungsi `exitAssignmentexpression`.

Kode Sumber 4.32 mengidentifikasi adanya penggunaan suatu *assignment operator* pada sebuah *assignments expression* melalui fungsi `enterAssignmentoperator`. Kode Sumber 4.32 mengidentifikasi variabel yang dideklarasikan sebelum sebuah *assignment operator* ditulis dan mengidentifikasi adanya operasi tambah, kurang, kali, bagi, dan mod yang dideklarasikan setelah *assignment operator* ditulis.

```
1   FUNCTION enterAssignmentoperator(ctx)
2   IF flagCondition == 0
3         flagExpression = 1
4         IF flagLabel > 0 THEN
5               operator = "G" +flagLabel +
                ctx.getText()
6         ELSE
7               operator = ctx.getText()
8         END IF
9         IF flagPostfix == -1 THEN
10              sequence = "( " + operator + "V"
                + ctx.getText()
11        ELSE IF flagPostfix == -2 THEN
12              sequence = operator + "this" +
13              temp_postfix + " "
14              postfix = null
15        ELSE
16              sequence = "( " + operator + "V "
17        END IF
18        CountLevel++
19        append sequence to ast
20        append CountLevel to level
21        stateoperator = ctx.getText()
22        split stateoperator by "=" and store in
          statement[]
23        FOR i = 0 to size of statement
24              IF statement[i] = '+' THEN
25                    sequence = operator + "+ "
26              ELSE IF statement[i] = '-' AND
                statement[i+1] != '>' THEN
27                    sequence = operator + "- "
28              ELSE IF (statement[i] = '/' THEN
29                    sequence = operator + "/ "
30              ELSE IF statement[i] = '%' THEN
31                    sequence = operator + "% "
32              ELSE IF statement[i] = '*' THEN
33                    sequence = operator + "* "
34              ELSE IF statement[i] = '"' THEN
35                    Break
```

```
36              END IF
37              append sequence to ast
38              append CountLevel to level
39        END LOOP
40  END IF
```

Kode Sumber 4.32 Kode program fungsi `Assignmentoperator` pada Listener

```
1   FUNCTION enterUnqualifiedid(ctx)
2   IF flagExpression = 1 AND flagBrace = 1 AND
    flagCondition = 0 THEN
3         IF flagparametercall > 0 THEN
4               operator = operator + "P"
5         ELSE IF flagparametercall = 0 THEN
6               operator = operator.replace("P")
7         END IF
8         cek_postfix(operator)
9   ELSE   IF   flagparametercall   >   0   AND
10  flagCondition = 0 THEN
11        IF flagSwitchcase > 0 THEN
12              operator = "G" + flagSwitchcase + "P"
13        ELSE
14              operator = "P"
15        END IF
16        cek_postfix(operator)
17  ELSE IF flagCondition = 1 THEN
18        IF flagparametercall > 0 THEN
19              operator = "JPV"
20        ELSE
21              operator = "JV"
22        cek_postfix(operator)
23        END IF
24  END IF
```

Kode Sumber 4.33 Kode program fungsi `Unqualifiedid` pada Listener

Kode Sumber 4.33 mengidentifikasi penggunaan variabel pada kode program melalui fungsi `enterUnqualifiedid` pada *Listener*. Kode sequence yang diberikan kepada variabel adalah huruf **V** yang diikuti dengan suatu simbol atau rangkaian huruf, yang merepresentasikan variabel tersebut merupakan milik suatu *assignment*, kondisi, atau parameter. Variabel juga bisa diikuti oleh suatu postfix, yang diidentifikasi melalui Kode Sumber 4.34.

```
1   FUNCTION cek_postfix(operator)
2   IF postfix = null THEN
3         IF tempPostfix != null THEN
4               tempPostfix = null
5         ELSE
6               sequence = operator + "V"
7               append sequence to ast
8               append CountLevel to level
9         END IF
10  ELSE
11        IF flagPostfix = 1 THEN
12              tempPostfix = postfix
13        ELSE IF flagPostfix = 3 THEN
14              tempPostfix = postfix
15              flagPostfix = 4
16              FlagParameterpostfix = 0
17        END IF
18        sequence = operator + "V" + postfix
19        append sequence to ast
20        append CountLevel to level
21        postfix = null
22  END IF
```

Kode Sumber 4.34 Kode program pengecekan postfix pada suatu variabel

```
1   FUNCTION enterLiteral(ctx)
2   literal = ctx.getText()
3   IF (flagExpression = 1 AND flagCondition = 0)
4         IF flagParametercall > 0 THEN
5               operator = operator + "P"
6         ELSE IF flagparametercall = 0 THEN
7               operator = operator.replace("P")
8         END IF
9         cek_string(operator,literal)
10  ELSE   IF   (flagParametercall   >   0   AND
    flagCondition = 0) THEN
11        IF flagSwitchcase > 0 THEN
12              operator = "G" + flagSwitchcase + "P"
13        ELSE
14              operator = "P"
15        END IF
16        cek_string(operator,literal)
17  ELSE IF flagCondition = 1 THEN
18        IF flagParametercall > 0 THEN
19              operator = "JP"
20        ELSE
21              sequence = "J" + literal
22              append sequence to ast
23              append CountLevel to level
24        END IF
25  END IF
```

Kode Sumber 4.35 Kode program fungsi `literal` pada Listener

Kode sumber 4.35 mengidentifikasi adanya penggunaan sebuah angka atau *literal* pada kode program melalui fungsi `enterLiteral` pada *Listener*. Literal diikuti dengan suatu simbol atau rangkaian huruf, yang merepresentasikan penggunaan angka atau *literal* tersebut merupakan milik suatu *assignment*, kondisi, atau parameter. Literal juga bisa dimiliki oleh suatu *string literal*, yang diidentifikasi melalui Kode Sumber 4.36.

```
1   FUNCTION cek_string(operator,literal)
2   IF flagString = 1 THEN
3         sequence = operator + "String"
4   ELSE
5         sequence = operator + literal
6   END IF
7   append sequence to ast
8   append CountLevel to level
```

Kode Sumber 4.36 Kode program pengecekan *string literal*

```
1   FUNCTION enterPrimaryexpression(ctx)
2   primaryexpress = ctx.getText()
3   IF primaryexpress = "this"
4         IF flagCondition = 1 THEN
5               sequence = "Jthis" + postfix
6         ELSE IF flagParametercall = 1 THEN
7               sequence = "Pthis" + postfix
8         ELSE IF flagExpression = 1 THEN
9               sequence =operator+ "this" + postfix
10        END IF
11        IF flagPostfix = 1
12              temp_postfix = postfix
13              postfix = null
14              flagPostfix = -2
15        END IF
16  END IF
```

Kode Sumber 4.37 Kode program fungsi `Primaryexpression` pada Listener

Kode Sumber 4.37 mengidentifikasi penggunaan **this** pointer pada kode program melalui fungsi `enterPrimaryexpression` pada *Listener*.

Tidak semua metode-metode pada *grammar C++* akan digunakan dalam proses deteksi plagiarisme. Metode-metode mana saja yang akan dipanggil oleh *Listener* telah dijelaskan pada Tabel 3.1. Sedangkan tabel konversi dari AST hasil *parser* ke dalam bentuk *sequence* dijelaskan pada Tabel 3.2.

Setelah kode program mahasiswa dikonversi menjadi bentuk *sequence,* tahap selanjutnya adalah menyimpan bentuk *sequence* tersebut ke dalam entitas *db_sequence* yang atribut-atributnya telah dijelaskan pada Tabel 3.3.

Pembentukan atribut-atribut pada Tabel 3.3 diproses melalui Kode Sumber 4.38.

```
String s = file.getName().toString().split("--")[0]
kelas = s.split(" ")[0]
file_name = file.getName().toString().s("--")[1]
nrp = s.split(" ")[1]
int i = file.getName().lastIndexOf('.')
if (i > 0)
       extention = file.getName().substring(i+1)
size = (file.length());
Date date = new Date(file.lastModified())
Format format = new SimpleDateFormat("yyyy MM
dd HH:mm:ss")
last_modified = format.format(date).toString()
hashcode = Integer.toString(file.hashCode())
path = file.getAbsolutePath()
```

Kode Sumber 4.38 Kode program pembentukan atribut pada entitas *db_sequence*

Sedangkan artribut-atribut yang ada pada Tabel 3.3 dan yang dimiliki oleh tiap kode program mahasiswa disimpan ke dalam basis data menggunakan framework *Hibernate* melalui Kode Sumber 4.39.

Hasil pembentukan *sequence* dan level *sequence* tiap kode program yang di*parsing* dan dibentuk menggunakan modifikasi fungsi *Listener* sebelumnya, juga dimasukkan ke dalam basis data melalui Kode sumber 4.39.

```
1   SessionFactory sessionFactory =
    HibernateUtility.getSessionFactory()
2   Session session = sessionFactory.openSession()
3   session.beginTransaction()
4   DB_sequence se = new DB_sequence()
5   se.setKelas(kelas)
6   se.setFile_name(file_name)
7   se.setNrp(nrp)
8   se.setSequence(CodeSequence)
9   se.setlevel_sequence(level)
10  se.setExtension(extention)
11  se.setSize(size)
12  se.setLast_modified(last_modif)
13  se.setHashcode(hashcode)
14  se.setPath(path)
15  session.save(se)
16  session.getTransaction().commit()
17  session.close()
```

Kode Sumber 4.39 Kode program penyimpanan atribut-atribut entitas *db_sequence* ke dalam basis data

Kode Sumber 4.39 merupakan proses penyimpanan atribut-atribut tiap kode program mahasiswa ke dalam basis data menggunakan framework *Hibernate*. Fungsi `SessionFactory` yang ada pada kode sumber 4.39 digunakan untuk mengatur eksekusi yang akan dijalankan ke dalam database ketika akan menjalankan *query insert.*

4.2.3 Penghitungan kemiripan kode program mahasiswa menggunakan *Levenshtein Distance*

Data *Sequence* dan level *sequence* tiap kode program mahasiswa yang telah disimpan ke dalam basis data, dibaca dan diambil dari database melalui Kode Sumber 4.40.

```
1   SessionFactory sessionFactory =
    HibernateUtility.getSessionFactory();
2   Session session = sessionFactory.openSession();
3   session.beginTransaction();
4   String hql = "FROM DB_sequence";
5   Query query = session.createQuery(hql);
6   List results = (List) query.list();
7   Iterator it = results.iterator();
8   while(it.hasNext())
9         DB_sequence st = (DB_sequence)it.next()
10        Db_main.temp_sequence.add(st.getSequence())
11        Db_main.temp_idseq.add(st.getNrp())
12        Db_main.temp_level.add(st.getlevel_sequence()
13  session.getTransaction().commit();
14  session.close();
```

Kode Sumber 4.40 Kode program pengambilan data *sequence* dari database

Kode Sumber 4.40 merupakan proses pengambilan atribut-tiap kode program mahasiswa berupa *Sequence*, Level *sequence*, dan NRP dari dalam database menggunakan framework *Hibernate*. Fungsi `SessionFactory` yang ada pada Kode Sumber 4.36 digunakan untuk mengatur eksekusi yang akan dijalankan ke dalam database ketika akan menjalankan fungsi *query*. *Query* yang digunakan dalam proses pengambilan data tersebut adalah *query select*.

Tahap selanjutnya tiap *sequence* dan level *sequence* kode program mahasiswa yang telah diambil dari dalam database dibandingkan dan dihitung persentase kemiripannya antara satu dengan yang lainnya. Persentase kemiripan antar kode program mahasiswa tersebut dihitung menggunakan metode *Levenshtein distance* yang telah dimodifikasi. Langkah-langkah praproses dari algoritma *Levensthein distance* dilakukan melalui Kode Sumber 4.41.

```
1   FUNCTION init (FisrtCodeSequence,
    FirstLevel,
    SecondCodeSequence , SecondLevel)

2   initialize firstCodeSeq as an empty
    array[]
3   initialize firstLevelSeq as an empty
    array[]
4   initialize secondCodeSeq as an empty
    array[]
5   initialize secondLevelSeq as an empty
    array[]

6   n1 = size of FisrtCodeSequence
7   n2 = size of SecondCodeSequence
8   FOR i = 0 to n1
9         IF FisrtCodeSequence[i] = "(" OR
          FisrtCodeSequence[i] = "(" OR
          FisrtCodeSequence[i] = "A" THEN
10              do nothing
11        ELSE
12              add FisrtCodeSequence[i] to
                firstCodeSeq
13              add FirstLevel[i] to
                firstLevelSeq
14        END IF
15  END LOOP
16  FOR i = 0 to n1
17        IF SecondCodeSequence[i] = "(" OR
18        SecondCodeSequence[i] = "(" OR
          SecondCodeSequence[i] = "A" THEN
19              do nothing
20        ELSE
21              add SecondCodeSequence[i] to
                secondCodeSeq
22              add SecondLevel[i] to
                secondLevelSeq
23        END IF
24  END LOOP
```

```
25  IF n1 <= n2 THEN
          Levensthein(n1, firstCodeSeq,
          firstLevelSeq,
          n2, secondCodeSeq, secondLevelSeq)
26  ELSE IF n2 < n1 THEN
27        Levensthein(n2, secondCodeSeq,
          secondLevelSeq,
          n1, firstCodeSeq, firstLevelSeq)
28  END IF
```

Kode Sumber 4.41 Kode program praproses metode *Levenshtein distance*

Kode Sumber 4.41 merupakan proses pengecekan dan pembentukan bagian dari hasil *sequence* mana saja yang akan digunakan dalam proses pengecekan plagiarisme pada metode *Levenshtein distance*. Pengecekan dan pembentukan dilakukan dengan melakukan sedikit praproses pada hasil *sequence*, sebelum siap untuk diolah dan digunakan dalam proses penghitungan kemiripan.

Praproses dilakukan dengan membuang beberapa bagian pada *sequence*. Beberapa bagian yang dibuang dan tidak digunakan dalam proses penghitungan kemiripan adalah parenthesis buka kurung dan tutup kurung serta kode *sequence* berupa huruf A . Hal ini dilakukan untuk meningkatkan akurasi dan ketepatan metode *Levensthein distance* dalam menghitung nilai kemiripan antara dua buah kode program.

Sedangkan implementasi dari metode *Levenshtein distance* yang digunakan dalam penghitungan persentase kemiripan kode program dijelaskan melalui Kode Sumber 4.42. Apabila substring sequence dan level sequence pada kode program pertama memiliki nilai yang sama dengan substring sequence dan level sequence pada kode program kedua, maka nilai *cost*-nya adalah nol. Sebaliknya apabila tidak sama maka nilai *cost*-nya adalah satu.

```
1   FUNCTION
    Levenshtein(row,firstsequence,firstlevel,
    coloumn,secondsequence,secondlevel)
2   initialize matrix as an empty
    array[row][coloumn]
3   initialize similarity as an empty float
4   matrix[0]0] = 0
5   FOR i = 0 to row
6         matrix[i][0] = i
7   END LOOP
8   FOR j = 0 to coloumn
9         matrix[0][j] = j
10  END LOOP
11  FOR i = 1 to row
12        FOR j = 1 to coloumn
13              IF firstsequence[i] =
                secondsequence[j] AND
                firstlevel[i] = secondlevel[j]
14                    cost = 0
15              ELSE
16                    cost = 1
17              END IF
18              matrix[i][j] = min( matrix[i-
                1][j-1]+cost, matrix[i-1][j]+1,
                matrix[i][j-1]+1)
19        END LOOP
20  END LOOP
21  similarity = 1 - (matrix[row][coloumn] /
    max(row,coloumn)) * 100
```

Kode Sumber 4.42 Kode program implementasi metode *Levenshtein Distance*

Variable *cost* pada Kode Sumber 4.38 merupakan nilai *cost* untuk matriks pada indeks ke (*baris-1 , kolom-1*). Hasil Akhir yang merupakan jarak antara dua buah *sequence* dari dua buah kode program yang dibandingkan dihitung nilai kemiripannya melalui Kode Sumber 4.38 pada baris akhir.

Persamaan yang diimplementasikan pada kode sumber 4.38 digunakan untuk menghitung nilai similarity dari hasil jarak *levenshtein* yang telah dihitung. Rumus tersebut dapat diuraikan ke dalam Persamaan 3.1.

Kelemahan dari metode *Levenshtein Distance* dalam mendeteksi plagiarisme dua buah kode program adalah tidak mampu mendeteksi isi kode program yang urutan penulisannya ditukar (dibolak-balik). Sehingga untuk mengatasi hal tersebut, diperlukan sedikit modifikasi terhadap pengecekan dan penghitungan *similarity* melalui Kode Sumber 4.43.

```
1   FUNCTION modification_levensthein
    (row,firstsequence,
    firstlevel,coloumn,secondsequence,
    secondlevel)
2   initialize flag as an empty array[coloumn]
3   initialize similarity as an empty float
4   FOR i = 0 to coloumn
5         flag[i] = 0
6   END LOOP
7   total = 0
8   FOR i = 1 to row
9         FOR j = 1 to coloumn
10              IF firstsequence[i] =
                secondsequence[j] AND
                firstlevel[i] = secondlevel[j]
11                    IF flag[i] != 1 THEN
12                          flag[i] = 1
13                          total++
14                          END LOOP
15                    END IF
16              END IF
17        END LOOP
18  END LOOP
19  similarity = (total / max(row,coloumn)) * 100
```

Kode Sumber 4.43 Kode program modifikasi *similarity*

Modifikasi *similarity* diimplementasikan melalui Kode Sumber 4.43 dengan cara melakukan pengecekan di setiap isi baris dan kolom matriks. Sehingga apabila terdapat *sequence* yang urutannya ditukar, dapat dideteksi, karena pengecekan *sequence* dilakukan secara menyeluruh pada tiap kolom dan baris. Apabila ada yang sama maka *cost* pada baris atau kolom tersebut bernilai satu. Variabel *flag* digunakan untuk menandai bahwa tidak akan ada *substring sequence* yang memiliki kemiripan lebih dari satu dengan *substring sequnce* pembandingnya. Penghitungan persentase kemiripannya adalah jumlah baris atau kolom yang memiliki nilai *cost* satu dibagi dengan panjang maksimum antara kedua buah *sequence* kode program.

Selanjutnya nilai persentase kemiripan antar dua buah kode program mahasiswa tersebut disimpan pada entitas *db_similarity* di dalam basis data. Atribut-atribut pada entitas *db_similarity* telah dijelaskan pada Tabel 3.4.

Atribut-atribut yang ada pada Tabel 3.4 tersebut disimpan ke dalam basis data melalui Kode Sumber 4.44.

```
1   SessionFactory sessionFactory =
    HibernateUtility.getSessionFactory()
2   Session session = sessionFactory.openSession()
3   session.beginTransaction();
4   DB_similarity se = new DB_similarity()
5   se.setId1(nrp1)
6   se.setId2(nrp2)
7   se.setSimilarity_value(score);
8   se.setLevenshtein_similarity(levenshtein_score)
9   session.save(se)
10  session.getTransaction().commit()
11  session.close()
```

Kode Sumber 4.44 Kode program penyimpanan atribut-atribut entitas *db_similarity* ke dalam basis data

Kode Sumber 4.44 merupakan proses penyimpanan atribut-atribut dari entitas *db_similarity* menggunakan framework

Hibernate. Fungsi `SessionFactory` yang ada pada Kode Sumber 4.44 digunakan untuk mengatur eksekusi yang akan dijalankan ke dalam database ketika akan menjalankan *query insert.*

4.2.4 Klasifikasi kode program mahasiswa menggunakan *Hierarchical clustering*

Hasil persentase kemiripan antara dua buah kode program yang telah dihitung sebelumnya, dihitung jaraknya dengan menggunakan Persamaan 3.3. Hasil dari penghitungan jarak *similarity* antara dua buah kode program disimpan terlebih dahulu ke dalam sebuah matriks dua dimensi.

Matriks dua dimensi tersebut berukuran N x N dan bersifat *mirror*. N merupakan jumlah kode program mahasiswa yang dibandingkan. Contoh matriks 3 x 3 yang merupakan hasil dari 3 buah kode program mahasiswa yang dibandingkan dapat dilihat pada Tabel 4.2.

Tabel 4.2 contoh matriks jarak kemiripan 3 buah kode program

0	Nrp ke-1	Nrp ke-2	Nrp ke-3
Nrp ke-1	0	2,45	50
Nrp ke-2	2,45	0	26,33
Nrp ke-3	50	26,33	0

Matriks tersebut ditulis ke dalam file Excel untuk mempermudah pembacaan dan proses klasifikasi. Proses penulisan matriks ke dalam bentuk file yang berekstensi .xls dilakukan melalui Kode Sumber 4.45.

```
1   Workbook wb = new HSSFWorkbook()
2   FileOutputStream fileOut = new
    FileOutputStream(fileName)
3   Sheet sheet = wb.createSheet(tabName)
4   Row[] row = new Row[data.length]
5   Cell[][] cell = new Cell[row.length][]
6
7   for(int i = 0; i < row.length; i ++)
8           row[i] = sheet.createRow(i)
9           cell[i] = new Cell[data[i].length]
10  for(int j = 0; j < cell[i].length; j ++)
11          cell[i][j] = row[i].createCell(j);
12          cell[i][j].setCellValue(data[i][j])
13
14  wb.write(fileOut)
15  fileOut.close()
```

Kode Sumber 4.45 Kode program menulis matriks ke dalam file Excel

```
1   double[][] data
2   FileInputStream fileIn = new
    FileInputStream(fileName)
3   Workbook wb = WorkbookFactory.create(fileIn)
4   Sheet sheet = wb.getSheetAt(tabNumber)
5   data = new double[sheet.getLastRowNum()+1][]
6   Row[] row = new Row[sheet.getLastRowNum()+1]
7   Cell[][] cell = new Cell[row.length][]
8   for(int i = 0; i < row.length; i++)
9           row[i] = sheet.getRow(i);
10          cell[i] = new
11          Cell[row[i].getLastCellNum()]
12          data[i] = new
13          double[row[i].getLastCellNum()]
14          for(int j = 0; j < cell[i].length; j++)
15                  cell[i][j] = row[i].getCell(j)
16                  data[i][j] = (double)
17                  cell[i][j].getNumericCellValue()
18  fileIn.close()
19  return data
```

Kode Sumber 4.46 Kode program membaca isi file Excel

Kode Sumber 4.46 melakukan pembacaan terhadap isi dari file excel untuk diolah lebih lanjut pada proses *clustering*. Tahap selanjutnya adalah mengimplementasikan metode *Hierarchical clustering* untuk mengelompokkan kode program berdasarkan tingkat kemiripannya. Diawali dengan tahap mencari nilai *minimum* (nilai terkecil) dari isi matriks melalui kode sumber 4.47.

```
1   initialize matrix as an array[][] of
    similarity
2   initialize tempmatrix as an array[][] of
    similarity
3   initialize member as an empty array[]
4   initialize cluster as an empty array[][]

5   FUNCTION minimum()
6   tempMinimum = 10000000
7   n = size of matrix
8   FOR i to n
9         FOR j to i
10              IF matrix[i][j] < tempMinimum
                  AND i != j AND matrix[i][j] > -1
11                      tempMinimum = matrix[i][j]
12                      row = i
13                      col = j
14              END IF
15        END LOOP
16  END LOOP
```

Kode Sumber 4.47 Kode program mencari nilai minimum matriks jarak *similarity*

Pada Kode Sumber 4.47, indeks baris dan kolom pada matriks yang memiliki nilai *minimum* disimpan ke dalam variabel *row* dan *col*. Nilai *row* dan *col* merepresentasikan NRP dari kedua buah kode program yang dibandingkan. Karena memiliki nilai minimum, maka *row* dan *col* digabungkan menjadi satu *cluster* melalui Kode Sumber 4.48.

```
1   FUNCTION joinCluster()
2   row,col = index of minimum value of matrix
3   p = min(row,col)
4   q = max(row,col)
5   tempmember = member[q]
6   FOR i=0 to tempmember
7         cluster[i][member[p]] = cluster[q][i]
8         cluster[q][i] = 0
9         tempmember--
10        member[p]++
11  END LOOP
12  member[q] = tempmember
13  n = length of matrix
14  FOR i = 0 to n
15        IF i!= p AND matrix[p][i]!= -3 AND
          matrix[i][p]!= -3 THEN
16              matrix[p][i] = -1
17              matrix[i][p] = -2
18        END IF
19  END LOOP
20  FOR i = 0 to n
21        matrix[q][i] = -3
22        matrix[i][q] = -3
23  END LOOP
```

Kode Sumber 4.48 Kode program penggabungan anggota *cluster*

Pada Kode Sumber 4.48 isi matriks pada indeks *row* dan *col* (*row* dan *col* merupakan member yang digabungkan menjadi satu *cluster*) akan dikosongkan terlebih dahulu untuk dihitung pada tahap selanjutnya. Proses pengosongan isi matriks tersebut ditandai dengan memberi nilai **-1** dan **-2**. Sedangkan isi matriks pada salah satu member (nilai maksimum antara *row* dan *col*) dihapus dengan memberi nilai **-3**. Selanjutnya isi matriks yang kosong tadi dihitung nilainya meggunakan metode *single linkage* melalui Kode Sumber 4.49.

```
1   FUNCTION minimumLinkage()
2   FOR i = 0 to n
3         FOR j = 0 to i
4               IF matrix[i][j] = -1 THEN
5                     temp = tempmatrix[i][j]
6                     FOR s = 1 to member[i]
7                           temp = min(temp ,
                      tempmatrix[j][cluster[i][s])
8                     END LOOP
9               ELSE IF matrix[i][j] = -2 THEN
10                    temp = tempmatrix[i][j]
11                    FOR t = 1 to member[j]
12                          temp = min(temp ,
                      tempmatrix[i][cluster[j][t])
13                    END LOOP
14              END IF
15              matrix[i][j] = temp
16        END LOOP
17  END LOOP
```

Kode Sumber 4.49 Kode program implementasi metode *Single linkage*

Kode Sumber 4.49 menjelaskan tentang metode *Single linkage* yang digunakan untuk mengisi matriks yang kosong pada tahap sebelumnya (ditandai dengan **-1** dan **-2**). Langkah selanjutnya mencari kembali nilai minimum dari matriks melalui kode sumber 4.47 dan menggabungkan isi *cluster* melalui kode sumber 4.48. Proses tersebut terus dilakukan hingga semua isi matriks dihapuskan atau bernilai **-3**. Apabila semua isi matriks bernilai -3, maka proses clustering telah mencapai *root* dari jumlah *cluster* keseluruhan yaitu jumlah cluster = 1.

Setelah proses *clustering* selesai, tahap selanjutnya adalah menghitung standar deviasi tiap *cluster* untuk menentukan jumlah *cluster* yang akan diambil. Proses ini dimulai dengan menghitung nilai *centroid* masing-masing *cluster*. Penghitungan *centroid* dimulai dari jumlah *cluster* = N-1 hingga jumlah *cluster* = 1 melalui Kode Sumber 4.50.

```
1   FUNCTION centroid()
2   initialize centroid as an empty array[][]
3   initialize matrix as an array[][] of similarity
4   initialize member as an array[] of number
    of cluster member
5   FOR i = 0 to size of cluster
6         FOR j = 1 to length of matrix
7               jumlah = 0
8               FOR k = 0 to member[i]
9                     jumlah = jumlah +
                      matrix[i][cluster[i][k]]
10              END LOOP
11              n = member[i]
12              centroid[i][j-1] = (1/n)*jumlah
13        END LOOP
14  END LOOP
```

Kode Sumber 4.50 Kode program menghitung nilai centroid masing-masing *cluster*

```
1   FUNCTION eucledian()
2   initialize beda as an empty array[]
3   initialize eucledian as an empty array[][]
4   initialize member as an array[] of number
    of cluster member
5   FOR i = 0 to number of cluster
6         FOR j = 0 to member[i]
7               jumlahBeda = 0
8               FOR k = 1 to length of matrix
9                     beda[k] =
                      pow(centroid[i][k] -
                      matrix[i][cluster[i][j]],2)
10                    jumlahBeda = jumlahBeda +
                      beda[k]
11              END LOOP
12              eucledian[i][j] = square root
                of jumlahBeda
13        END LOOP
14  END LOOP
```

Kode Sumber 4.51 Kode program menghitung jarak Eucledian antara tiap anggota *cluster* dengan centroidnya

```
1   FUNCTION std()
2   initialize standarDeviasi as an empty array[]
3   initialize member as an array[] of number
    of cluster member
4   FOR i = 0 to number of cluster
5         EucledianTotal = 0
6         mean = 0
7         totalSTD = 0
8         FOR j = 0 to member[i]
9               EucledianTotal = EucledianTotal
                + eucledian[i][j]
10        END LOOP
11        mean = EucledianTotal / member[i]
12        FOR j = 0 to member[i]
13              diff = pow(eucledian[i][j] -
                mean , 2)
14              totalSTD = totalSTD + square
                root of diff
15        END LOOP
16        IF member[i]-1 = 0 THEN
17              std = 0
18        ELSE
19              std = totalSTD /(member[i]-1)
20        END IF
20        standarDeviasi[i] = square root of std
21  END LOOP
```

Kode Sumber 4.52 Kode Program menghitung nilai standar deviasi masing-masing *cluster*

Cara untuk menghitung nilai centroid masing-masing *cluster* yang diimplementasikan pada Kode Sumber 4.50 ada pada Persamaan 3.4. Selanjutnya menghitung nilai jarak masing-masing member dalam satu *cluster* dengan nilai centroid *cluster* tersebut menggunakan metode *Eucledian distance*, yang diimplementasikan melalui Kode Sumber 4.51.

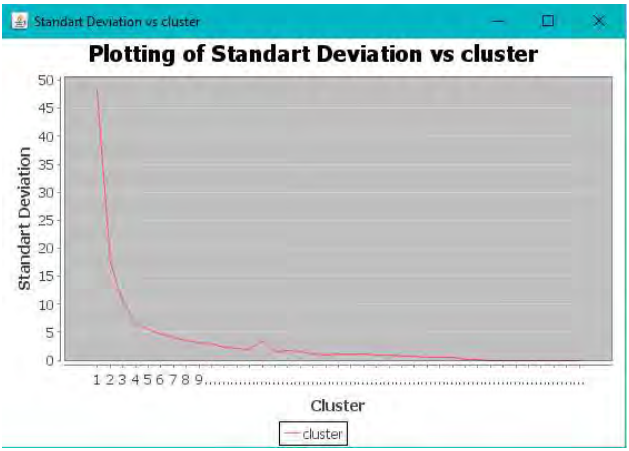
Setelah mendapatkan nilai jarak *Eucledian* antara masing-masing member dalam satu *cluster* dengan centroidnya,

selanjutnya dihitung nilai standar deviasi masing-masing *cluster* melalui Kode Sumber 4.52.

Cara untuk menghitung nilai standar deviasi dari masing-masing *cluster* yang diimplementasikan pada Kode Sumber 4.52 ada pada Persamaan 3.5. Setelah mendapatkan nilai standar deviasi dari masing-masing *cluster*. Maka dihitung nilai rata-rata dari semua standar deviasi yang ada pada jumlah *cluster* yang sama melalui Kode Sumber 4.53. Sehingga diperoleh nilai standar deviasi tiap jumlah *cluster*. Dari jumlah *cluster* = N-1 hingga jumlah *cluster* = 1. N merupakan banyaknya kode program yang dibandingkan.

```
1  FUNCTION average()
2  total = 0
3  FOR i = 0 to number of cluster
4        total = total + standarDeviasi[i]
5  END LOOP
6  averageOfSTD = total / number of cluster
```

Kode Sumber 4.53 Kode program menghitung nilai rata-rata standar deviasi tiap jumlah *cluster*



Gambar 4.2 Grafik hasil penghitungan standar deviasi

```
1   public Chart( String applicationTitle , String
    chartTitle , double[] array_cluster)
2         super(applicationTitle)
3         JFreeChart lineChart =
4         ChartFactory.createLineChart(
          chartTitle,
          "Cluster","Standart Deviation",
          createDataset(array_cluster ),
          PlotOrientation.VERTICAL,
          true,true,false)
5         ChartPanel chartPanel = new ChartPanel(
          lineChart )
6         chartPanel.setPreferredSize( new
          java.awt.Dimension( 560 , 367 ) )
          setContentPane( chartPanel )

7   private DefaultCategoryDataset createDataset(
    double[] array_cluster)
8         int pjg = array_cluster.length;
9         DefaultCategoryDataset dataset = new
10        DefaultCategoryDataset( )
11        for (int i = pjg -1; i>= 0; i--)
12              dataset.addValue( array_cluster[i]
                ,"cluster" , "" + (pjg-i))
13        return dataset
```

Kode Sumber 4.54 Kode Program untuk menampilkan grafik hasil penghitungan standar deviasi

Pada Kode Sumber 4.53 diperoleh nilai standar deviasi tiap masing-masing jumlah *cluster*, yang selanjutnya dipetakan ke dalam grafik XY melalui Kode Sumber 4.54. Gambar 4.2 adalah Hasil dari implementasi kode sumber 4.54 berupa tampilan grafik hasil penghitungan standar deviasi tiap jumlah *cluster*.

(Halaman ini sengaja dikosongkan)

BAB V
HASIL UJI COBA DAN EVALUASI

Bab ini berisi penjelasan mengenai skenario uji coba dan evaluasi pada deteksi dan klasifikasi plagiarisme kode program antar mahasiswa dalam satu kelas. Hasil uji coba didapatkan dari implementasi pada bab 4 dengan skenario yang berbeda. Bab ini berisikan pembahasan mengenai lingkungan pengujian, data pengujian, dan uji kinerja.

5.1 Lingkungan Pengujian

Lingkungan pengujian pada uji coba permasalahan deteksi dan klasifikasi plagiarisme kode program antar mahasiswa dalam satu kelas menggunakan spesifikasi keras dan perangkat lunak seperti yang ditunjukkan pada Tabel 5.1.

Tabel 5.1 Spesifikasi Lingkungan Pengujian

Perangkat	Jenis Perangkat	Spesifikasi
Perangkat Keras	Prosesor	Intel(R) Core(TM) i5-337U CPU @ 1.80 GHz
	Memori	8 GB 1600 MHz DDR3
Perangkat Lunak	Sistem Operasi	Windows 10 Home Single Language
	Perangkat Pengembang	Eclipse Luna , ANTLR V4, dan PostgreSQL V9.3

5.2 Data Pengujian

Subbab ini menjelaskan mengenai data yang digunakan pada uji coba. Seperti yang telah dijelaskan sebelumnya, data diambil dari dua jenis kode program Mahasiswa dalam satu kelas yang mengambil mata kuliah Pemrograman Berorientasi Objek (PBO). Dataset jenis pertama berjumlah 38 buah kode program dari 38

mahasiswa dalam satu kelas. Dataset jenis ke 2 berjumlah 21 buah kode program dari 21 mahasiswa dalam satu kelas. Pada dataset jenis kedua terdaat dua jenis file yaitu 21 file header (.h) dan 21 file yang berekstensi .cpp. Sehingga masing-masing file akan dihitung perfomanya pada tahap uji coba.

5.3 Skenario Uji Coba

Sebelum melakukan uji coba, perlu ditentukan skenario yang akan digunakan dalam uji coba. Melalui skenario ini, perangkat akan diuji apakah sudah berjalan dengan benar dan bagaimana akurasi pada masing-masing skenario. Dan membandingkan skenario manakah yang memiliki hasil lebih baik. Terdapat 2 macam skenario uji coba, yaitu :

1. Penghitungan akurasi *similarity* antar kode program, pengelompokan kode program, dan penentuan jumlah *cluster* terbaik pada dataset jenis pertama (tingkat kemiripan antar kode program relatif kecil)
2. Penghitungan akurasi *similarity* kode program, pengelompokan kode program, dan penentuan jumlah *cluster* terbaik pada dataset jenis kedua yang berekstensi file .cpp (tingkat kemiripan antar kode program relatif besar)
3. Penghitungan akurasi *similarity* kode program, pengelompokan kode program, dan penentuan jumlah *cluster* terbaik pada dataset jenis kedua yang berupa file header (tingkat kemiripan antar kode program sedikit relatif besar)

5.3.1 Skenario Uji Coba 1

Skenario uji coba 1 adalah penghitungan akurasi sistem pendeteksi plagiarisme kode program dengan menggunakan dataset jenis pertama yaitu dataset yang memiliki tingkat kemiripan antar kode program yang relatif kecil. Uji coba

dilakukan dengan menghitung akurasi pada penghitungan nilai *similarity* antar kode program, pengelompokan kode program berdasarkan tingkat kemiripannya, dan penentuan jumlah *cluster* terbaik yang akan dipilih.

Akurasi pada perhitungan kemiripan kode program dilakukan dengan 2 jenis *ground truth. Ground truth* yang pertama menilai secara manual kemiripan dua buah kode program dengan menggunakan **2** parameter, yaitu :

1. Persentase kemiripan **0% - 50%** (kategori tidak mirip)
2. Persentase kemiripan **51% - 100%** (kategori mirip)

Terdapat 38 kode program pada dataset jenis pertama yang memiliki total 703 nilai *similarity*. Dari total 703 nilai itu dihitung akurasinya dengan cara menilai secara manual, apakah kedua kode program termasuk dalam kategori tidak mirip atau mirip. Apabila nilai persentase kemiripan antar kedua kode program sesuai dengan kategori pada hasil penilaian manual, maka masuk dalam kolom *TP (true positive)* atau *TN (True Negative)*. Penghitungan akurasi, presisi, dan recall dilakukan melalui Persamaan 5.1.

$$akurasi = \frac{TP + TN}{total}$$

$$presisi = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN} \tag{5.1}$$

Penilaian akurasi kemiripan kode program pada uji coba 1 dengan **2** paramater dapat dilihat pada Tabel 5.2 dan Tabel 5.3. Label *Actual* merupakan label untuk hasil penilaian manual yang dilakukan penulis, sedangkan label *predicted* merupakan nilai dari hasil keluaran sistem.

Tabel 5.2 Hasil akurasi penghitungan kemiripan kode program pada uji coba 1 dengan 2 parameter (Levensthein distance)

Levenshtein distance		*Predicted*	
		Mirip	Tidak mirip
Actual	Mirip	0	14
	Tidak mirip	0	691
Akurasi		**98,29%**	
Presisi		0%	
Recall		0%	

Tabel 5.3 Hasil akurasi penghitungan kemiripan kode program pada uji coba 1 dengan 2 parameter (modifikasi similarity)

Modifikasi similarity		*Predicted*	
		Mirip	Tidak mirip
Actual	Mirip	6	6
	Tidak mirip	8	683
Akurasi		**98%**	
Presisi		42,85%	
Recall		50%	

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **2** parameter di atas, metode *modifikasi similarity* dan metode *Levenshtein distance* memiliki akurasi yang hampir sama yaitu **98%**. Namun nilai presisi dan recall yang dimiliki metode *Levenshtein distance* sangat rendah yaitu 0%. Hal ini dikarenakan metode *Levenshtein distance* tidak bisa mendeteksi kode program yang posisinya ditukar (dibolak-balik).

Ground truth yang kedua menilai secara manual kemiripan dua buah kode program dengan menggunakan **3** parameter, yaitu:

1. Persentase kemiripan **0% - 35%** (kategori tidak mirip)
2. Persentase kemiripan **36% - 65%** (kategori setengah mirip)

3. Persentase kemiripan **66% - 100%** (kategori mirip)

Hasil penilaian secara manual dan penilaian akurasi kemiripan kode program pada uji coba 1 dengan **3** paramater dapat dilihat pada Tabel 5.4 dan Tabel 5.5.

Tabel 5.4 Hasil akurasi penghitungan kemiripan kode program pada uji coba 1 dengan 3 parameter (Levensthein distance)

Levenshtein distance		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	0	0	0
	Setengah mirip	0	10	28
	Tidak mirip	0	3	662
Akurasi		**95,59%**		

Tabel 5.5 Hasil akurasi penghitungan kemiripan kode program pada uji coba 1 dengan 3 parameter (Levensthein modifikasi)

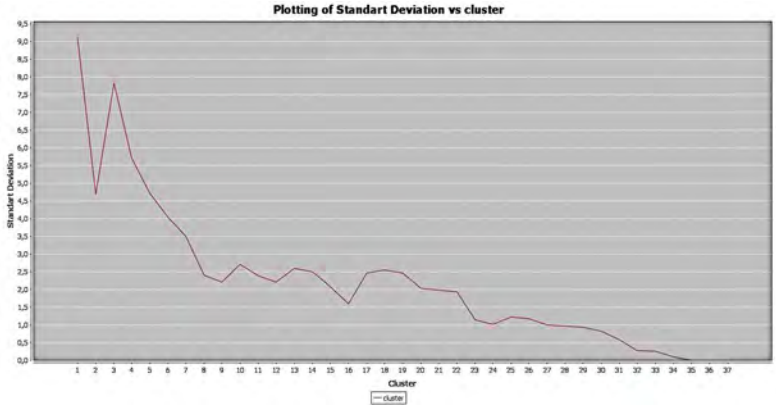
Modifikasi similarity		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	0	0	0
	Setengah mirip	0	38	0
	Tidak mirip	0	75	590
Akurasi		**89,33%**		

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **3** parameter diatas, metode *Levenshtein distance* memiliki akurasi lebih tinggi dibanding *Modifikasi similarity* yaitu mencapai **95,59%.** Sedangkan apabila menggunakan metode *modifikasi similarity* akurasinya mencapai 89,33%. Hal ini dikarenakan data masukan

pada uji coba pertama memiliki tingkat kemiripan yang relatif kecil.

Nilai yang digunakan dalam proses pengelompokan kode program adalah nilai kemiripan yang dihasilkan pada metode *Modifikasi similarity*. Pengelompokan kode program berdasarkan tingkat kemiripannya dilakukan menggunakan metode *Hierarchical clustering.* Sedangkan pemilihan jumlah cluster yang tepat dilakukan dengan menghitung standar deviasi masing-masing *cluster*. Proses penilaian akurasi pengelompokan kode program dan pemilihan jumlah cluster yang akan diambil dilakukan secara manual, apakah jumlah *cluster* dan pengelompokan kode programnya telah sesuai atau belum.

Hasil keluaran berupa grafik dari standar deviasi masing-masing jumlah *cluster* pada uji coba 1 digambarkan pada Gambar 5.1.



Gambar 5.1 Grafik hasil penghitungan standar deviasi *cluster* pada uji coba 1

Dari hasil keluaran berupa grafik di atas, dapat dilihat bahwa hasil grafiknya tidak stabil (*naik-turun*), namun pada jumlah cluster = 32, titik-titik setelahnya cenderung stabil menurun. Sehingga jumlah cluster yang dipilih adalah jumlah cluster = 32.

Penilaian akurasi pengelompokan kode program pada jumlah cluster = 32 dilakukan secara manual dengan menilai apakah kode program-kode program yang terdapat dalam satu *cluster* memang mirip satu sama lain dan apakah kode program yang menjadi satu-satunya member pada suatu *cluster* benar-benar tidak mirip dengan kode program manapun. Apabila sesuai maka nilainya *True*, apabila tidak sesuai maka nilainya *False*. Pemberian label *True-False* dan penentuan nilai *True-False* dilakukan secara manual oleh penulis.

Hasil pengelompokan kode program pada uji coba 1 dapat dilihat pada Tabel 5.6. Hasil pengelompokan terdiri dari NRP kode program mana saja yang termasuk dalam satu *cluster* dan *range* persentase kemiripan antar member dalam satu *cluster*. *Range* awal merupakan nilai minimum persentase kemiripan antar member dalam satu *cluster*. *Range* akhir adalah nilai maksimum persentase kemiripan antar member dalam satu *cluster*.

Untuk hasil *range* awal dan akhir yang memiliki nilai sama, umumnya dimiliki anggota *cluster* yang berjumlah satu atau dua. Jika anggota *cluster* berjumlah satu, rangenya bernilai 0% - 0%, karena persentase kemiripan ke dirinya sendiri bernilai 0%. Jika anggota *cluster* berjumlah dua, nilai *range* awal dan akhir sama dengan nilai kemiripan dua anggota tersebut.

Apabila ada anggota *cluster* yang berjumlah lebih dari dua dan memiliki *range* awal dan akhir yang sama, maka anggota-anggota *cluster* tersebut memiliki nilai persentase kemiripan yang sama antar tiap anggota di dalam *cluster* tersebut.

Tabel 5.6 Hasil pengelompokan kode program mahasiswa pada uji coba 1

Cluster	No	Member	Range persentase kemiripan
1	1	5114100015	0% - 0%
2	1 2	5114100016 5114100063	54.90% - 54.90%

3	1	5114100019	0% - 0%
4	1	5114100020	0% - 0%
5	1	5114100027	0% - 0%
6	1	5114100033	0% - 0%
7	1	5114100034	0% - 0%
8	1 2	5114100035 5114100064	61.36% - 61.36%
9	1	5114100041	0% - 0%
10	1	5114100043	0% - 0%
11	1	5114100044	0% - 0%
12	1	5114100051	0% - 0%
13	1 2 3 4	5114100052 5114100062 5114100179 5114100704	35.44% - 56.36%
14	1	5114100053	0% - 0%
15	1	5114100057	0% - 0%
16	1	5114100075	0% - 0%
17	1	5114100086	0% - 0%
18	1 2	5114100089 5114100100	53.97% - 53.97%
19	1	5114100105	0% - 0%
20	1	5114100110	0% - 0%
21	1	5114100115	0% - 0%
22	1	5114100122	0% - 0%
23	1	5114100128	0% - 0%
24	1	5114100143	0% - 0%
25	1	5114100147	0% - 0%
26	1	5114100151	0% - 0%
27	1	5114100154	0% - 0%
28	1	5114100162	0% - 0%

29	1	5114100165	0% - 0%
30	1	5114100177	0% - 0%
31	1	5114100178	0% - 0%
32	1	5114100188	0% - 0%

Hasil penilaian secara manual dan penilaian akurasi pengelompokan kode program pada uji coba 1 dapat dilihat pada Tabel 5.7.

Tabel 5.7 Penilaian manual hasil pengelompokan kode program pada uji coba 1

Cluster	Member	Hasil		Akurasi
		True	False	
1	5114100015	√		100%
2	5114100016 5114100063	√ √		100%
3	5114100019	√		100%
4	5114100020	√		100%
5	5114100027		√	0%
6	5114100033		√	0%
7	5114100034		√	0%
8	5114100035 5114100064	√ √		100%
9	5114100041		√	0%
10	5114100043	√		100%
11	5114100044	√		100%
12	5114100051	√		
13	5114100052 5114100062 5114100179 5114100704	√ √ √	 √	75%
14	5114100053	√		100%
15	5114100057	√		100%

16	5114100075		√	0%
17	5114100086	√		100%
18	5114100089 5114100100		√ √	0%
19	5114100105	√		100%
20	5114100110	√		100%
21	5114100115	√		100%
22	5114100122	√		100%
23	5114100128	√		100%
24	5114100143	√		100%
25	5114100147	√		100%
26	5114100151	√		100%
27	5114100154	√		100%
28	5114100162	√		100%
29	5114100165	√		100%
30	5114100177	√		100%
31	5114100178		√	0%
32	5114100188	√		100%
Rata-rata				**77,34%**

Berdasarkan hasil yang ditunjukkan oleh tabel 5.3 akurasi rata-rata dari pengelompokan kode program pada dataset jenis pertama mencapai **77,34%**.

Cluster ke 5 yang beranggotakan satu buah kode program bernilai *false* karena memiliki tingkat kemiripan yang hampir sama dengan anggota pada *cluster* ke 6 dan salah satu anggota pada *cluster* ke 13 (5114100704). Sehingga ketiga kode program tersebut dapat digabungkan menjadi satu *cluster*, tidak terpisah-pisah.

Cluster ke 7 yang beranggotakan satu buah kode program bernilai *false* karena memiliki tingkat kemiripan yang relatif sama dengan anggota pada *cluster* ke 16. Sehingga ketiga kode program

tersebut dapat digabungkan menjadi satu *cluster*, tidak terpisah-pisah.

Cluster ke 9 yang beranggotakan satu buah kode program bernilai *false* karena memiliki tingkat kemiripan yang relatif sama dengan anggota-anggota pada *cluster* ke 13. Sehingga kode program pada *cluster* ke 9 dapat digabungkan menjadi satu pada *cluster* ke 13.

Cluster ke 8 yang beranggotakan dua buah kode program bernilai *false* karena dianggap tidak memiliki tingkat kemiripan yang hampir sama satu sama lain. Salah satu anggota pada cluster ini (5114100100) lebih memiliki tingkat kemiripan yang relatif besar dengan anggota pada *cluster* ke 31. Sehingga kedua buah kode program ini dapat digabungkan menjadi satu *cluster.*

5.3.2 Skenario Uji Coba 2

Skenario uji coba 2 adalah penghitungan akurasi sistem pendeteksi plagiarisme kode program dengan menggunakan dataset jenis kedua berupa file .cpp , yaitu dataset yang memiliki tingkat kemiripan antar kode program yang relatif besar. Uji coba dilakukan dengan menghitung akurasi pada penghitungan nilai *similarity* antar kode program, pengelompokan kode program berdasarkan tingkat kemiripannya, dan penentuan jumlah *cluster* terbaik yang akan dipilih.

Akurasi pada perhitungan kemiripan kode program dilakukan dengan 2 jenis *ground truth. Ground truth* yang pertama menilai secara manual kemiripan dua buah kode program dengan menggunakan **2** parameter, yaitu :

1. Persentase kemiripan **0% - 50%** (kategori tidak mirip)
2. Persentase kemiripan **51% - 100%** (kategori mirip)

Terdapat 38 kode program pada dataset jenis kedua(berekstensi .cpp) yang memiliki total 210 nilai *similarity*. Dari total 210 nilai itu dihitung akurasinya dengan cara menilai secara manual, apakah kedua kode program termasuk dalam

kategori tidak mirip atau mirip. Apabila nilai persentase kemiripan antar kedua kode program sesuai dengan kategori pada hasil penilaian manual, maka masuk dalam kolom *TP (true positive)* atau *TN (True Negative)*. Penghitungan akurasi, presisi, dan recall dilakukan melalui Persamaan 5.1.

Hasil penilaian secara manual dan penilaian akurasi kemiripan kode program pada uji coba 2 dengan **2** paramater dapat dilihat pada Tabel 5.8 dan Tabel 5.9. Label *Actual* merupakan label untuk hasil penilaian manual yang dilakukan penulis, sedangkan label *predicted* merupakan nilai dari hasil keluaran sistem.

Tabel 5.8 Hasil akurasi penghitungan kemiripan kode program pada uji coba 2 dengan 2 parameter (Levensthein tanpa modifikasi)

Levenshtein distance		*Predicted*	
		Mirip	Tidak mirip
Actual	Mirip	90	32
	Tidak mirip	0	88
Akurasi		**84,76%**	
Presisi		100%	
Recall		73,77%	

Tabel 5.9 Hasil akurasi penghitungan kemiripan kode program pada uji coba 2 dengan 2 parameter (Levensthein modifikasi)

Modifikasi similarity		*Predicted*	
		Mirip	Tidak mirip
Actual	Mirip	117	5
	Tidak mirip	0	88
Akurasi		**97,62%**	
Presisi		100%	
Recall		95,90%	

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **2** parameter diatas,

metode *Modifikasi similarity* memiliki akurasi lebih tinggi dibanding metode *Levenshtein distance* yaitu mencapai **97,62%.** Apabila menggunakan metode *Levenshtein distance* akurasinya mencapai 84,76%. Hal ini dikarenakan metode *Modifikasi similarity* mampu mendeteksi kode program yang urutannya ditukar.

Ground truth yang kedua menilai secara manual kemiripan dua buah kode program dengan menggunakan **3** parameter, yaitu:

1. Persentase kemiripan **0% - 35%** (kategori tidak mirip)
2. Persentase kemiripan **36% - 65%** (kategori setengah mirip)
3. Persentase kemiripan **66% - 100%** (kategori mirip)

Hasil penilaian secara manual dan penilaian akurasi kemiripan kode program pada uji coba 2 dengan **3** paramater dapat dilihat pada Tabel 5.10 dan Tabel 5.11.

Tabel 5.10 Hasil akurasi penghitungan kemiripan kode program pada uji coba 2 dengan 3 parameter (Levensthein tanpa modifikasi)

Levenshtein distance		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	54	30	27
	Setengah mirip	0	19	7
	Tidak mirip	0	0	73
Akurasi		**69,52%**		

Tabel 5.11 Hasil akurasi penghitungan kemiripan kode program pada uji coba 2 dengan 3 parameter (Levensthein modifikasi)

Modifikasi similarity		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	106	5	0

	Setengah mirip	0	26	0
	Tidak mirip	0	4	69
Akurasi		**95,71%**		

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **3** parameter diatas, metode *Modifikasi similarity* memiliki akurasi lebih tinggi dibanding metode *Levenshtein distance* yaitu mencapai **95,71%.** Sedangkan apabila menggunakan metode *Levenshtein distance* akurasinya hanya mencapai 69,52%.

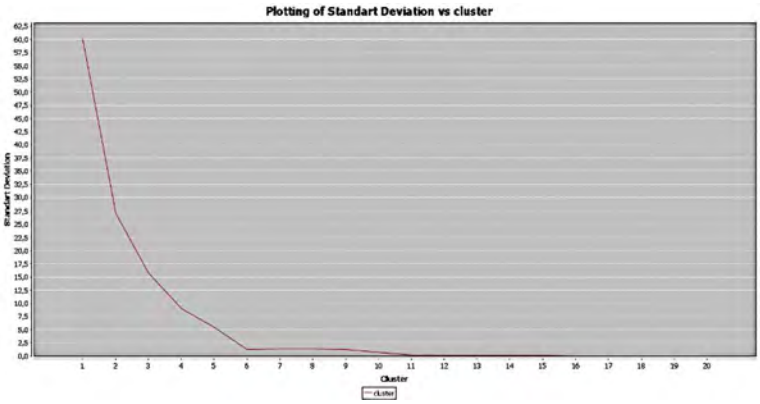
Oleh karena itu nilai yang digunakan dalam proses pengelompokan kode program adalah nilai kemiripan yang dihasilkan pada metode *Modifikasi similarity*. Pengelompokan kode program berdasarkan tingkat kemiripannya dilakukan menggunakan metode *Hierarchical clustering.* Sedangkan pemilihan jumlah cluster yang tepat dilakukan dengan menghitung standar deviasi masing-masing *cluster*. Proses penilaian akurasi pengelompokan kode program dan pemilihan jumlah cluster yang akan diambil dilakukan secara manual, apakah jumlah *cluster* dan pengelompokan kode programnya telah sesuai atau belum.

Hasil keluaran berupa grafik dari standar deviasi masing-masing jumlah *cluster* pada uji coba 2 digambarkan pada Gambar 5.2.

Dari hasil keluaran berupa grafik tersebut, dapat dilihat bahwa hasil grafiknya cenderung stabil dan mengalami titik balik dimana titik berada pada jumlah cluster = 6. Sehingga jumlah cluster yang dipilih adalah jumlah cluster = 6.

Penilaian akurasi pengelompokan kode program pada jumlah cluster = 6 dilakukan secara manual dengan menilai apakah kode program-kode program yang terdapat dalam satu *cluster* memang mirip satu sama lain dan apakah kode program yang menjadi satu-satunya member pada suatu *cluster* benar-benar tidak mirip dengan kode program mana pun. Apabila sesuai maka nilai nya *True*, apabila tidak sesuai maka nilainya *False*. Pemberian

label *True-False* dan penentuan nilai *True-False* dilakukan secara manual oleh penulis.



Gambar 5.2 Grafik hasil penghitungan standar deviasi *cluster* pada uji coba 2

Hasil pengelompokan kode program pada uji coba 2 dapat dilihat pada Tabel 5.12. Hasil pengelompokan terdiri dari NRP kode program mana saja yang termasuk dalam satu *cluster* dan *range* persentase kemiripan antar member dalam satu *cluster*. *Range* awal merupakan nilai minimum persentase kemiripan antar member dalam satu *cluster*. *Range* akhir adalah nilai maksimum persentase kemiripan antar member dalam satu *cluster*.

Untuk hasil *range* awal dan akhir yang memiliki nilai sama, umumnya dimiliki anggota *cluster* yang berjumlah satu atau dua. Jika anggota *cluster* berjumlah satu, rangenya bernilai 0% - 0%, karena persentase kemiripan ke dirinya sendiri bernilai 0%. Jika anggota *cluster* berjumlah dua, nilai *range* awal dan akhir sama dengan nilai kemiripan dua anggota tersebut.

Apabila ada anggota *cluster* yang berjumlah lebih dari dua dan memiliki *range* awal dan akhir yang sama, maka anggota-anggota *cluster* tersebut memiliki nilai persentase kemiripan yang sama antar tiap anggota di dalam *cluster* tersebut.

Tabel 5.12 Hasil pengelompokan kode program mahasiswa pada uji coba 2

Cluster	No	Member	Range persentase kemiripan
1	1	5113100045	0% - 0%
2	1 2	5113100091 5113100101	100% - 100%
3	1	5113100121	0% - 0%
4	1	5113100150	0% - 0%
5	1	5113100184	0% - 0%
6	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	5113100004 5113100009 5113100061 5113100180 5113100039 5113100050 5113100119 5113100152 5113100163 5113100134 5113100187 5113100005 5113100011 5113100020 5113100012	76.49% - 100%

Hasil penilaian secara manual dan penilaian akurasi pengelompokan kode program pada uji coba 2 dapat dilihat pada Tabel 5.13.

Tabel 5.13 Penilaian manual hasil pengelompokan kode program pada uji coba 2

Cluster	member	Hasil		Akurasi
		True	False	
1	5113100045		√	0%
2	5113100091 5113100101	√ √		100%
3	5113100121	√		100%
4	5113100150	√		100%
5	5113100184	√		100%
6	5113100004 5113100009 5113100061 5113100180 5113100039 5113100050 5113100119 5113100152 5113100163 5113100134 5113100187 5113100005 5113100011 5113100020 5113100012	√ √ √ √ √ √ √ √ √ √ √ √ √ √ √		100%
Rata-rata				**83,33%**

Berdasarkan hasil yang ditunjukkan oleh tabel 5.13 akurasi rata-rata dari pengelompokan kode program pada dataset jenis ke 2 yang berupa file .cpp, mencapai **83,33%.** Terdapat satu buah *cluster* yang bernilai *false*, yaitu *cluster* ke 1. Pada *cluster* tersebut hanya beranggotakan satu kode program, anggota tersebut bernilai *false* karena memiliki tingkat kemiripan yang hampir sama dengan anggota-anggota pada *cluster* ke 6. Yang menyebakan kode

program pada *cluster* ke 1 tersebut berbeda dengan anggota-anggota pada *cluster* ke 6 adalah adanya satu fungsi pada kode program yang tidak dimiliki oleh anggota-anggota lain pada *cluster* ke 6, namun isi selebihnya sama dengan anggota-anggota pada *cluster* ke 6. Sehingga lebih sesuai apabila kode program pada *cluster* ke 1 digabungkan dengan *cluster* ke 6.

5.3.3 Skenario Uji Coba 3

Skenario uji coba 3 adalah penghitungan akurasi sistem pendeteksi plagiarisme kode program dengan menggunakan dataset jenis kedua berupa file header , yaitu dataset yang memiliki tingkat kemiripan antar kode program yang sedikit relatif besar. Uji coba dilakukan dengan menghitung akurasi pada penghitungan nilai *similarity* antar kode program, pengelompokan kode program berdasarkan tingkat kemiripannya, dan penentuan jumlah *cluster* terbaik yang akan dipilih.

Akurasi pada perhitungan kemiripan kode program dilakukan dengan 2 jenis *ground truth. Ground truth* yang pertama menilai secara manual kemiripan dua buah kode program dengan menggunakan **2** parameter, yaitu :

1. Persentase kemiripan **0% - 50%** (kategori tidak mirip)
2. Persentase kemiripan **51% - 100%** (kategori mirip)

Terdapat 38 kode program pada dataset jenis kedua (file header) yang memiliki total 210 nilai *similarity*. Dari total 210 nilai itu dihitung akurasinya dengan cara menilai secara manual, apakah kedua kode program termasuk dalam kategori tidak mirip atau mirip. Apabila nilai persentase kemiripan antar kedua kode program sesuai dengan kategori pada hasil penilaian manual, maka masuk dalam kolom *TP (true positive)* atau *TN (True Negative)*. Penghitungan akurasi, presisi, dan recall dilakukan melalui Persamaan 5.1.

Hasil penilaian secara manual dan penilaian akurasi kemiripan kode program pada uji coba 3 dengan **2** paramater dapat

dilihat pada Tabel 5.14 dan Tabel 5.15. Label *Actual* merupakan label untuk hasil penilaian manual yang dilakukan penulis, sedangkan label *predicted* merupakan nilai dari hasil keluaran sistem.

Tabel 5.14 Hasil akurasi penghitungan kemiripan kode program pada uji coba 3 dengan 2 parameter (Levensthein tanpa modifikasi)

Levenshtein distance		*Predicted*	
		Mirip	Tidak mirip
Actual	Mirip	39	29
	Tidak mirip	0	142
Akurasi		**86,19%**	
Presisi		100%	
Recall		57,35%	

Tabel 5.15 Hasil akurasi penghitungan kemiripan kode program pada uji coba 3 dengan 2 parameter (Levensthein modifikasi)

Modifikasi similarity		*Predicted*	
		Mirip	Tidak mirip
Actual	Mirip	66	2
	Tidak mirip	9	133
Akurasi		**94,76%**	
Presisi		88%	
Recall		97,06%	

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **2** parameter di atas, metode *Modifikasi similarity* memiliki akurasi lebih tinggi dibanding metode *Levenshtein distance* yaitu mencapai **94,76%.** Apabila menggunakan metode *Levenshtein distance* akurasinya mencapai 86,19%.

Ground truth yang kedua menilai secara manual kemiripan dua buah kode program dengan menggunakan **3** parameter, yaitu:

1. Persentase kemiripan **0% - 35%** (kategori tidak mirip)
2. Persentase kemiripan **36% - 65%** (kategori setengah mirip)
3. Persentase kemiripan **66% - 100%** (kategori mirip)

Hasil penilaian secara manual dan penilaian akurasi kemiripan kode program pada uji coba 3 dengan **3** paramater dapat dilihat pada Tabel 5.16 dan Tabel 5.17.

Tabel 5.16 Hasil akurasi penghitungan kemiripan kode program pada uji coba 3 dengan 3 parameter (Levensthein tanpa modifikasi)

Levenshtein distance		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	33	0	12
	Setengah mirip	0	39	40
	Tidak mirip	0	3	83
Akurasi		**73,81%**		

Tabel 5.17 Hasil akurasi penghitungan kemiripan kode program pada uji coba 3 dengan 3 parameter (Levensthein modifikasi)

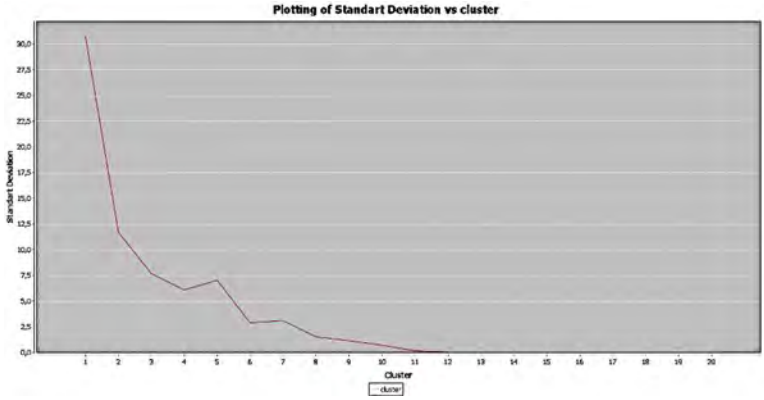
Modifikasi similarity		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	45	0	0
	Setengah mirip	4	72	3
	Tidak mirip	0	24	62
Akurasi		**85,24%**		

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **3** parameter di atas, metode *Modifikasi similarity* memiliki akurasi lebih tinggi dibanding metode *Levenshtein distance* yaitu mencapai **85,24%.**

Sedangkan apabila menggunakan metode *Levenshtein distance* akurasinya mencapai 73,81%.

Oleh karena itu nilai yang digunakan dalam proses pengelompokan kode program adalah nilai kemiripan yang dihasilkan pada metode *Modifikasi similarity*. Pengelompokan kode program berdasarkan tingkat kemiripannya dilakukan menggunakan metode *Hierarchical clustering.* Sedangkan pemilihan jumlah cluster yang tepat dilakukan dengan menghitung standar deviasi masing-masing *cluster*. Proses penilaian akurasi pengelompokan kode program dan pemilihan jumlah cluster yang akan diambil dilakukan secara manual, apakah jumlah *cluster* dan pengelompokan kode programnya telah sesuai atau belum.

Hasil keluaran berupa grafik dari standar deviasi masing-masing jumlah *cluster* pada uji coba 3 digambarkan pada Gambar 5.3.



Gambar 5.3 Grafik hasil penghitungan standar deviasi cluster pada uji coba 3

Dari hasil keluaran berupa grafik di atas, dapat dilihat bahwa hasil grafiknya cenderung tidal stabil (naik-turun) dan mengalami titik balik dimana titik berada pada jumlah cluster = 8. Sehingga jumlah cluster yang dipilih adalah jumlah cluster = 8.

Penilaian akurasi pengelompokan kode program pada jumlah cluster = 8 dilakukan secara manual dengan menilai apakah

kode program-kode program yang terdapat dalam satu *cluster* memang mirip satu sama lain dan apakah kode program yang menjadi satu-satunya member pada suatu *cluster* benar-benar tidak mirip dengan kode program manapun. Apabila sesuai maka nilainya *True*, apabila tidak sesuai maka nilainya *False*. Pemberian label *True-False* dan penentuan nilai *True-False* dilakukan secara manual oleh penulis.

Hasil pengelompokan kode program pada uji coba 3 dapat dilihat pada Tabel 5.18. Hasil pengelompokan terdiri dari NRP kode program mana saja yang termasuk dalam satu *cluster* dan *range* persentase kemiripan antar member dalam satu *cluster*. *Range* awal merupakan nilai minimum persentase kemiripan antar member dalam satu *cluster*. *Range* akhir adalah nilai maksimum persentase kemiripan antar member dalam satu *cluster*.

Untuk hasil *range* awal dan akhir yang memiliki nilai sama, umumnya dimiliki anggota *cluster* yang berjumlah satu atau dua. Jika anggota *cluster* berjumlah satu, rangenya bernilai 0% - 0%, karena persentase kemiripan ke dirinya sendiri bernilai 0%. Jika anggota *cluster* berjumlah dua, nilai *range* awal dan akhir sama dengan nilai kemiripan dua anggota tersebut.

Apabila ada anggota *cluster* yang berjumlah lebih dari dua dan memiliki *range* awal dan akhir yang sama, maka anggota-anggota *cluster* tersebut memiliki nilai persentase kemiripan yang sama antar tiap anggota di dalam *cluster* tersebut.

Tabel 5.18 Hasil pengelompokan kode program mahasiswa pada uji coba 3

Cluster	No	Member	Range persentase kemiripan
1	1 2 3	5113100011 5113100020 5113100012	95,46% - 100%
2	1	5113100009	0% - 0%

3	1 2 3 4 5 6 7 8 9	5113100004 5113100039 5113100050 5113100119 5113100152 5113100163 5113100134 5113100187 5113100005	76,94% - 100%
4	1 2 3	5113100045 5113100061 5113100180	100% - 100%
5	1 2	5113100091 5113100101	100% - 100%
6	1	5113100121	0% - 0%
7	1	5113100150	0% - 0%
8	1	5113100184	0% - 0%

Hasil penilaian secara manual dan penilaian akurasi pengelompokan kode program pada uji coba 3 dapat dilihat pada Tabel 5.19.

Tabel 5.19 Penilaian manual hasil pengelompokan kode program pada uji coba 3

Cluster	Member	Hasil		Akurasi
		True	False	
1	5113100011 5113100020 5113100012	√ √ √		100%
2	5113100009	√		100%

3	5113100004		√	77,77%
	5113100039	√		
	5113100050	√		
	5113100119	√		
	5113100152	√		
	5113100163	√		
	5113100134	√		
	5113100187	√		
	5113100005		√	
4	5113100045	√		100%
	5113100061	√		
	5113100180	√		
5	5113100091	√		100%
	5113100101	√		
6	5113100121	√		100%
7	5113100150	√		100%
8	5113100184	√		100%
Rata-rata				**97,22%**

Berdasarkan hasil yang ditunjukkan oleh Tabel 5.19 akurasi rata-rata dari pengelompokan kode program pada dataset jenis ke 2 yang berupa file header, mencapai **97,22%.**

Terdapat satu buah *cluster* yang anggotanya bernilai *false*, yaitu *cluster* ke 3. Pada *cluster* tersebut, terdapat dua buah anggota yang bernilai *false* karena kedua anggota tersebut labih sesuai apabila dijgabungkan menjadi satu *cluster* tersendiri. Yang menyebakan dua kode program tersebut berbeda dengan anggota-anggota lain pada *cluster* adalah adanya satu deklarasi pointer pada kode program yang tidak dimiliki oleh anggota-anggota lain dalam satu *cluster* dan cara pendeklarasian variabel pada kedua kode program tersebut sedikit berbeda. Sehingga lebih sesuai apabila dua buah kode program yang bernilai *false* pada *cluster* ke 1 dijadikan satu *cluster* tersendiri.

5.4 Evaluasi Umum Skenario Uji Coba

Berdasarkan 3 skenario uji coba yang telah dilakukan, dapat diketahui bahwa tingkat ketepatan sistem untuk mendeteksi plagiarisme kode program mahasiswa sudah sangat baik yaitu:

1. Akurasi pada uji coba pertama menunjukkan bahwa 38 kode program mahasiswa sebagai data masukan yang pertama memiliki tingkat kemiripan yang relatif kecil antara kode program satu dengan yang lainnya. Sehingga proses pengelompokannya cenderung berdiri sendiri-sendiri menjadi satu *cluster* sendiri.
2. Akurasi pada uji coba kedua cukup baik. Hasil akurasi menunjukkan bahwa 21 kode program berupa file yang berekstensi *.cpp* memiliki tingkat kemiripan yang relatif besar antara kode program satu dengan yang lainnya. Sehingga proses penentuan jumlah *cluster* dan pengelompokannya telah berhasil mengelompokkan kode program yang mirip ke dalam *cluster* yang sama.
3. Akurasi pada uji coba ketiga hampir sama dengan uji coba kedua. Data masukan pada uji coba kedua berjumlah 21 kode program berupa file header dari data masukan pada uji coba kedua. Proses penentuan jumlah *cluster* dan pengelompokannya telah berhasil mengelompokkan kode program yang mirip ke dalam *cluster* yang sama.
4. Evaluasi pada proses penghitungan nilai kemiripan antar dua kdoe program mengggunakan *Modifikasi similarity* dapat meningkatkan akurasi dalam mendeteksi plagiarisme kode program yang urutannya dibolak-balik.
5. Evaluasi pada proses pengelempokkan antara lain terdapat beberapa kode program yang seharusnya dikelompokkan ke cluster lain tetapi hasilnya menunjukkan bahwa kode program tersebut berdiri sendiri menjadi satu *cluster*. Hal ini diakibatkan karena tingkat kemiripannya rendah/dianggap berbeda dengan yang lain.

6. Penentuan jumlah *cluster* didasarkan pada nilai standar deviasi terhadap *centroid cluster*.

BAB VI
KESIMPULAN DAN SARAN

Bab ini berisikan kesimpulan yang dapat diambil dari hasil uji coba yang telah dilakukan. Selain kesimpulan, terdapat juga saran yang ditujukan untuk pengembangan perangkat lunak nantinya.

6.1 Kesimpulan

Kesimpulan yang didapatkan berdasarkan hasil uji coba deteksi dan klasifikasi plagiarisme kode program mahasiswa adalah sebagai berikut:

1. Sistem *back-end E-Learning* pada modul pendeteksian plagiarisme antara kode program dalam satu kelas sudah cukup baik mendeteksi kode program yang memang terbukti melakukan plagiarisme terhadap kode program lainnya.
2. Metode *Modifikasi similarity* terbukti lebih akurat dalam mendeteksi beberapa *case* dalam tindakan plagiarisme kode program, dibandingkan dengan metode *Levenshtein distance*.
3. Metode *Hierarchichal clustering* berhasil mengelompokkan kode program berdasarkan tingkat kemiripannya.

6.2 Saran

Saran yang diberikan terkait pengembangan pada Tugas Akhir ini adalah:

1. Ada beberapa deklarasi dan penggunaan *syntax* pada C++ yang belum bisa dibaca oleh sistem, dikarenakan belum adanya pengembangan lebih lanjut terhadap isi *grammar C++* ANTLR. Antara lain penggunaan **array** dan isi kode program berupa ***else*** masih belum bisa berdiri sendiri menjadi level yang berbeda.

2. Grammar C++ yang telah disediakan ANTLR memiliki cakupan yang terlalu luas dalam representasi bentuk ASTnya. Sehingga diperlukan banyak modifikasi pada *grammar* untuk membaca beberapa deklarasi dan penggunaan *syntax* pada C++ yang belum bisa dibaca oleh sistem.

DAFTAR PUSTAKA

[1] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L., 1998, November. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on* (pp. 368-377).

[2] T. Parr, "ANTLR," 1989. [Online]. Available: http://www.antlr.org/. [Diakses 16 December 2015].

[3] T. Parr, The definitive ANTLR 4 Reference, United States: The Pragmatic Programmers, LLC, 2013.

[4] T. Parr, "ANTLR Grammars-v4," [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/cpp. [Diakses 17 December 2015].

[5] V. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, vol. 10, Soviet Physics Doklady, 1966, pp. 707-710.

[6] G. Navarro, "A guide tour to approximate string matching," ACM computing surveys (CSUR), 2001, pp. 31-88.

[7] Dani, T. G. Limandra dan L. R. E. Adiseputra, "Deteksi Kemiripan Kode Program dengan Metode Preprocessing dan Penghitungan Levenshtein Distance," Institut Teknologi Bandung, 2006.

[8] P. Kardi Teknomo, "Hierarchical Clustering Numerical Example," 2007. [Online]. Available: http://people.revoledu.com/kardi/tutorial/Clustering/Numerical%20Example.htm. [Diakses April 2016].

[9] "PostgreSQL," 1996. [Online]. Available: http://www.postgresql.org/docs/current/static/intro-whatis.html.

[10] J. E. Friedl, Mastering Regular Expression, O'Reilly, 2002.

[11] J. Goyvaerts, “Regular Expression Tutorial,” 2003. [Online]. Available: http://www.regular-expressions.info/tutorial.html. [Diakses 1 May 2016].

LAMPIRAN

Tabel A. 1 Daftar ID yang digunakan sebagai masukan sistem pada Uji Coba 1

ID	NRP
1	5114100015
2	5114100016
3	5114100019
4	5114100020
5	5114100027
6	5114100033
7	5114100034
8	5114100035
9	5114100041
10	5114100043
11	5114100044
12	5114100051
13	5114100052
14	5114100053
15	5114100057
16	5114100062
17	5114100063
18	5114100064
19	5114100075
20	5114100086
21	5114100089
22	5114100100
23	5114100105
24	5114100110
25	5114100115
26	5114100122
27	5114100128

28	5114100143
29	5114100147
30	5114100151
31	5114100154
32	5114100162
33	5114100165
34	5114100177
35	5114100178
36	5114100179
37	5114100188
38	5114100704

Tabel A. 2 Daftar ID yang digunakan sebagai masukan sistem pada Uji Coba 2 dan Uji Coba 3

ID	NRP
1	5113100004
2	5113100005
3	5113100009
4	5113100011
5	5113100012
6	5113100020
7	5113100039
8	5113100045
9	5113100050
10	5113100061
11	5113100091
12	5113100101
13	5113100119
14	5113100121
15	5113100134
16	5113100150
17	5113100152
18	5113100163

19	5113100180
20	5113100184
21	5113100187

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	0	21,3	28,58	33,71	28,2	34,8	24,56	38,11	25,65	33,35	35,54	35,18	21,25	34,44	15,02	21,99	17,95	31,52	19,06
2	21,25	0	22,13	19,15	14,77	26,04	42,11	20,68	30,36	26,15	24,58	14,52	42,32	20,79	22,34	36,29	54,91	18,58	43,15
3	28,58	22,1	0	45,11	17,13	31,17	25,13	43,04	24,13	28,15	39,71	29,36	19,61	39,62	18,11	24,13	19,11	45,08	19,11
4	33,71	19,2	45,11	0	24,84	31,08	19,15	53,6	20,86	28,52	33,62	30,97	14,91	34,9	20	17,88	16,61	45,47	14,48
5	28,2	14,8	17,13	24,84	0	28,86	18,47	25,18	24,84	25,84	21,83	21,63	17,13	19,13	19,13	14,43	15,11	18,47	15,78
6	34,8	26	31,17	31,08	28,86	0	33,73	31,65	34,93	47,73	45,62	20,65	31,38	31,42	26,04	31,38	25,45	24,25	29
7	24,56	42,1	25,13	19,15	18,47	33,73	0	17,74	36,57	31,83	26,92	13,56	45,62	19,33	22,81	35,1	39,48	17,81	52,65
8	38,11	20,7	43,04	53,6	25,18	31,65	17,74	0	21,11	26,59	35,87	33,23	16,88	39,25	16,04	20,27	16,47	61,37	13,93
9	25,65	30,4	24,13	20,86	24,84	34,93	36,57	21,11	0	33,54	29,24	15,82	45,53	24,65	21,38	37,25	26,9	18,2	33,11
10	33,35	26,2	28,15	28,52	25,84	47,73	31,83	26,59	33,54	0	39,22	20,65	30,13	28,99	22,74	30,7	23,31	22,36	26,72
11	35,54	24,6	39,71	33,62	21,83	45,62	26,92	35,87	29,24	39,22	0	30,33	22,81	51,22	23,99	25,15	21,06	31,83	20,47
12	35,18	14,5	29,36	30,97	21,63	20,65	13,56	33,23	15,82	20,65	30,33	0	12,92	33,23	12,92	13,56	12,59	33,55	10,98
13	21,25	42,3	19,61	14,91	17,13	31,38	45,62	16,88	45,53	30,13	22,81	12,92	0	21,27	27,9	50	41,36	15,92	44,25
14	34,44	20,8	39,62	34,9	19,13	31,42	19,33	39,25	24,65	28,99	51,22	33,23	21,27	0	18,36	23,2	19,33	31,45	16,43
15	15,02	22,3	18,11	20	19,13	26,04	22,81	16,04	21,38	22,74	23,99	12,92	27,9	18,36	0	29,13	24,29	12,89	23,31
16	21,99	36,3	24,13	17,88	14,43	31,38	35,1	20,27	37,25	30,7	25,15	13,56	50	23,2	29,13	0	36,47	18,95	47,62
17	17,95	54,9	19,11	16,61	15,11	25,45	39,48	16,47	26,9	23,31	21,06	12,59	41,36	19,33	24,29	36,47	0	15,54	41,68
18	31,52	18,6	45,08	45,47	18,47	24,25	17,81	61,37	18,2	22,36	31,83	33,55	15,92	31,45	12,89	18,95	15,54	0	13,65
19	19,06	43,2	19,11	14,48	15,78	29	52,65	13,93	33,11	26,72	20,47	10,98	44,25	16,43	23,31	47,62	41,68	13,65	0
20	27,75	12,3	25,18	20	16,79	18,4	11,94	23,24	16,79	17,43	23,88	29,68	12,59	36,79	10,98	14,52	11,31	22,59	11,94
21	26,38	29,4	39,21	37,04	15,44	28,24	26,49	43,9	28,24	25	33,93	22,92	25,9	30,93	15,31	24,72	25,31	40,91	23,54
22	29,31	26,2	47,25	44,69	17,13	34,66	23,31	53,6	25,58	31,83	45,47	27,11	21,04	40,11	21,61	26,15	21,04	51,15	19,33
23	24,56	29,7	24,13	23,42	24,18	39,07	29,67	24,9	35,18	33,54	29,83	19,04	28,97	24,17	28,97	30,36	31,04	17,43	29,67
24	32,68	16,3	38,69	35,83	19,79	23,79	19,49	41,85	20,65	23,22	25,52	31,24	16,06	24,93	11,18	16,92	13,76	53,6	14,34
25	29,68	31,1	36,69	35,75	20,81	37,29	28,04	37,98	30,36	31,83	43,29	23,24	28,79	37,69	30,31	32,58	28,04	31,08	24,25
26	20,52	40,2	26,65	20,43	12,76	21,9	33,35	21,95	27,59	20,47	28,67	17,75	35,58	24,65	22,34	37,25	45,85	19,33	38,3
27	29,31	15,8	34,62	37,32	16,79	16,54	20,4	38,47	17,7	18,47	25,77	30,97	15,4	27,7	10,4	10	14,25	40,54	13,09
28	29,58	17,5	32,12	37,23	20,47	17,52	18,99	35,78	20,81	17,52	29,58	30,65	15,34	31,4	10,23	12,06	16,06	36,15	14,6
29	26,74	17,2	31,56	27,88	12,42	13,94	18,04	31,56	18,45	13,12	29,11	28,72	13,53	47,15	9,02	11,08	15,59	30,7	17,63
30	29,68	23,2	26,15	23,83	27,52	27,79	27,29	24,06	29,31	30,31	32,83	26,13	23,74	30,93	13,15	17,68	21,72	20,47	23,74
31	27,22	12,9	26,88	22,79	20,15	17,02	18,38	24,83	17,02	18,04	33	34,21	13,95	35,39	9,53	10,22	11,92	26,2	15,66
32	26,38	15	20,11	20,86	17,45	27,22	12,5	23,22	20	27,29	32,18	23,56	11,89	26,58	21,88	18,75	13,76	14,41	12,5
33	27,84	24,9	31,17	30,22	17,13	24,31	26,52	32,5	24,31	26,52	38,68	25,81	23,77	34,79	15,48	14,92	20,45	29,56	18,79
34	23,08	19,3	33,18	34,05	14,43	30,18	19,33	31,24	26,22	25	36,26	19,36	14,5	27,06	25,52	23,45	15,18	27,29	15,18
35	43,97	28	46,73	42,57	24,84	41,6	28,52	50,65	28,99	51,41	46,73	32,26	26,18	42,07	18,7	27,58	22,43	50,76	21,97
36	28,22	40,9	31,17	25,97	21,49	43,79	43	28,29	41,39	40,36	37,43	18,72	54,55	34,3	34,55	56,37	40,91	25,38	41,83
37	25,63	11,2	25,08	24,27	17,18	20,72	12	21,81	12,54	19,08	21,27	25,63	11,18	19,63	11,18	12,54	11,73	27,81	9,27
38	27,49	33,1	31,17	27,67	26,86	46,16	48,83	26,18	35,87	44,9	38,61	20,33	37,8	27,06	32,29	35,44	33,86	23,11	36,23

Matriks Uji Coba 1 (bagian 1)

0	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
1	27,75	26,38	29,31	24,56	32,68	29,68	20,52	29,31	29,58	26,74	29,68	27,22	26,38	27,84	23,08	43,97	28,22	25,63	27,49
2	12,26	29,43	26,15	29,67	16,34	31,08	40,21	15,77	17,52	17,22	23,24	12,93	15	24,88	19,33	28,04	40,91	11,18	33,08
3	25,18	39,21	47,25	24,13	38,69	36,69	26,65	34,62	32,12	31,56	26,15	26,88	20,11	31,17	33,18	46,73	31,17	25,08	31,17
4	20	37,04	44,69	23,42	35,83	35,75	20,43	37,32	37,23	27,88	23,83	22,79	20,86	30,22	34,05	42,57	25,97	24,27	27,67
5	16,79	15,44	17,13	24,18	19,79	20,81	12,76	16,79	20,47	12,42	27,52	20,15	17,45	17,13	14,43	24,84	21,49	17,18	26,86
6	18,4	28,24	34,66	39,07	23,79	37,29	21,9	16,54	17,52	13,94	27,79	17,02	27,22	24,31	30,18	41,6	43,79	20,72	46,16
7	11,94	26,49	23,31	29,67	19,49	28,04	33,35	20,4	18,99	18,04	27,29	18,38	12,5	26,52	19,33	28,52	43	12	48,83
8	23,24	43,9	53,6	24,9	41,85	37,98	21,95	38,47	35,78	31,56	24,06	24,83	23,22	32,5	31,24	50,65	28,29	21,81	26,18
9	16,79	28,24	25,58	35,18	20,65	30,36	27,59	17,7	20,81	18,45	29,31	17,02	20	24,31	26,22	28,99	41,39	12,54	35,87
10	17,43	25	31,83	33,54	23,22	31,83	20,47	18,47	17,52	13,12	30,31	18,04	27,29	26,52	25	51,41	40,36	19,08	44,9
11	23,88	33,93	45,47	29,83	25,52	43,29	28,67	25,77	29,58	29,11	32,83	33	32,18	38,68	36,26	46,73	37,43	21,27	38,61
12	29,68	22,92	27,11	19,04	31,24	23,24	17,75	30,97	30,65	28,72	26,13	34,21	23,56	25,81	19,36	32,26	18,72	25,63	20,33
13	12,59	25,9	21,04	28,97	16,06	28,79	35,58	15,4	15,34	13,53	23,74	13,95	11,89	23,77	14,5	26,18	54,55	11,18	37,8
14	36,79	30,93	40,11	24,17	24,93	37,69	24,65	27,7	31,4	47,15	30,93	35,39	26,58	34,79	27,06	42,07	34,3	19,63	27,06
15	10,98	15,31	21,61	28,97	11,18	30,31	22,34	10,4	10,23	9,02	13,15	9,53	21,88	15,48	25,52	18,7	34,55	11,18	32,29
16	14,52	24,72	26,15	30,36	16,92	32,58	37,25	10	12,06	11,08	17,68	10,22	18,75	14,92	23,45	27,58	56,37	12,54	35,44
17	11,31	25,31	21,04	31,04	13,76	28,04	45,85	14,25	16,06	15,59	21,72	11,92	13,76	20,45	15,18	22,43	40,91	11,73	33,86
18	22,59	40,91	51,15	17,43	53,6	31,08	19,33	40,54	36,15	30,7	20,47	26,2	14,41	29,56	27,29	50,76	25,38	27,81	23,11
19	11,94	23,54	19,33	29,67	14,34	24,25	38,3	13,09	14,6	17,63	23,74	15,66	12,5	18,79	15,18	21,97	41,83	9,27	36,23
20	0	20,65	19,68	14,52	20,06	22,59	14,2	21,95	20,33	40,65	19,36	24,2	19,36	24,52	15,82	20,97	18,72	15,54	18,72
21	20,65	0	53,98	21,18	30,67	35,9	30	33,86	33,58	26,24	27,29	20,08	18,83	46,41	28,24	38,33	30,59	16,63	27,65
22	19,68	53,98	0	27,86	40,98	42,05	28,99	30,4	30,67	28,7	20,72	22,11	22,74	38,14	42,62	53,75	39,79	22,9	32,4
23	14,52	21,18	27,86	0	17,77	32,43	23,45	12,7	15,34	11,9	21,22	11,58	21,88	18,24	28,29	28,99	39,33	14,17	46,9
24	20,06	30,67	40,98	17,77	0	22,93	16,34	32,68	32,11	24,65	17,77	22,36	11,48	23,22	21,79	42,71	24,65	31,61	24,65
25	22,59	35,9	42,05	32,43	22,93	0	34,86	30	20,45	27,88	26,27	22,11	35,64	33,15	37,25	39,26	44,71	17,72	40,16
26	14,2	30	28,99	23,45	16,34	34,86	0	17,31	19,36	22,56	23,74	16,68	15,64	22,67	21,38	25,72	35,47	12,27	27,56
27	21,95	33,86	30,4	12,7	32,68	30	17,31	0	44,9	38,86	31,54	32,33	14,25	52,71	18,08	33,86	14,62	18,81	18,47
28	20,33	33,58	30,67	15,34	32,11	20,45	19,36	44,9	0	32,86	31,77	32,33	11,68	37,97	17,9	33,58	15,34	17,99	18,99
29	40,65	26,24	28,7	11,9	24,65	27,88	22,56	38,86	32,86	0	23,38	35,39	15,59	34,43	17,22	32,39	13,94	15,26	13,12
30	19,36	27,29	20,72	21,22	17,77	26,27	23,74	31,54	31,77	23,38	0	25,18	20,72	36,87	13,65	29,92	24,25	13,64	30,31
31	24,2	20,08	22,11	11,58	22,36	22,11	16,68	32,33	32,33	35,39	25,18	0	16	25,52	12,59	26,54	15,32	19,08	16
32	19,36	18,83	22,74	21,88	11,48	35,64	15,64	14,25	11,68	15,59	20,72	16	0	17,68	28,75	21,5	23,13	12	24,38
33	24,52	46,41	38,14	18,24	23,22	33,15	22,67	52,71	37,97	34,43	36,87	25,52	17,68	0	25,43	36,46	20,45	16,08	25,97
34	15,82	28,24	42,62	28,29	21,79	37,25	21,38	18,08	17,9	17,22	13,65	12,59	28,75	25,43	0	33,18	33,11	16,08	33,11
35	20,97	38,33	53,75	28,99	42,71	39,26	25,72	33,86	33,58	32,39	29,92	26,54	21,5	36,46	33,18	0	37,86	26,72	37,4
36	18,72	30,59	39,79	39,33	24,65	44,71	35,47	14,62	15,34	13,94	24,25	15,32	23,13	20,45	33,11	37,86	0	20,99	54,35
37	15,54	16,63	22,9	14,17	31,61	17,72	12,27	18,81	17,99	15,26	13,64	19,08	12	16,08	16,08	26,72	20,99	0	17,45
38	18,72	27,65	32,4	46,9	24,65	40,16	27,56	18,47	18,99	13,12	30,31	16	24,38	25,97	33,11	37,4	54,35	17,45	0

Matriks Uji Coba 1 (bagian 2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	88,41	94,66	80,36	80,36	80,36	88,41	56,08	88,41	89,93	29,11	29,11	88,41	8,04	87,5	36,46	88,41	88,41	89,93	49,37	82,16
2	88,41	0	87,86	83,82	83,82	83,82	81,82	50,51	81,82	80,69	26,92	26,92	81,82	7,85	81,66	27,13	81,82	81,82	80,69	48,11	88,24
3	94,66	87,86	0	83,19	83,19	83,19	90,91	53,54	90,91	87,41	28,38	28,38	90,91	8,42	90,83	36,46	90,91	90,91	87,41	46,85	86,93
4	80,36	83,82	83,19	0	96,21	98,1	84,55	49,5	85,46	76,49	27,65	27,65	85,46	9,53	86,24	27,97	85,46	85,46	76,49	48,75	85,72
5	80,36	83,82	83,19	96,21	0	98,1	84,55	49,5	84,55	76,49	28,02	28,02	84,55	10,5	85,33	27,97	84,55	84,55	76,49	50	85,72
6	80,36	83,82	83,19	98,1	98,1	0	84,55	49,5	85,46	76,49	28,02	28,02	85,46	10,5	86,24	27,97	85,46	85,46	76,49	50	85,72
7	88,41	81,82	90,91	84,55	84,55	84,55	0	52,54	99,1	82,36	29,11	29,11	99,1	8,19	98,19	32,22	99,1	99,1	82,36	46,22	92,74
8	56,08	50,51	53,54	49,5	49,5	49,5	52,54	0	52,54	60,11	33,11	33,11	52,54	5,56	52,04	28,79	52,54	52,54	60,11	51,53	49
9	88,41	81,82	90,91	85,46	84,55	85,46	99,1	52,54	0	82,36	29,11	29,11	100	8,19	99,1	32,22	100	100	82,36	46,22	92,74
10	89,93	80,69	87,41	76,49	76,49	76,49	82,36	60,11	82,36	0	28,74	28,74	82,36	9,25	81,52	40,35	82,36	82,36	100	50	76,49
11	29,11	26,92	28,38	27,65	28,02	28,02	29,11	33,11	29,11	28,74	0	100	29,11	3,65	29,11	14,19	29,11	29,11	28,74	26,92	26,56
12	29,11	26,92	28,38	27,65	28,02	28,02	29,11	33,11	29,11	28,74	100	0	29,11	3,65	29,11	14,19	29,11	29,11	28,74	26,92	26,56
13	88,41	81,82	90,91	85,46	84,55	85,46	99,1	52,54	100	82,36	29,11	29,11	0	8,19	99,1	32,22	100	100	82,36	46,22	92,74
14	8,04	7,85	8,42	9,53	10,49	10,49	8,19	5,56	8,19	9,25	3,65	3,65	8,19	0	8,26	11,87	8,19	8,19	9,25	6,97	7,85
15	87,5	81,66	90,83	86,24	85,33	86,24	98,19	52,04	99,1	81,52	29,11	29,11	99,1	8,26	0	31,36	99,1	99,1	81,52	46,22	92,68
16	36,46	27,13	36,46	27,97	27,97	27,97	32,22	28,79	32,22	40,35	14,19	14,19	32,22	11,9	31,36	0	32,22	32,22	40,35	18,36	26,29
17	88,41	81,82	90,91	85,46	84,55	85,46	99,1	52,54	100	82,36	29,11	29,11	100	8,19	99,1	32,22	0	100	82,36	46,22	92,74
18	88,41	81,82	90,91	85,46	84,55	85,46	99,1	52,54	100	82,36	29,11	29,11	100	8,19	99,1	32,22	100	0	82,36	46,22	92,74
19	89,93	80,69	87,41	76,49	76,49	76,49	82,36	60,11	82,36	100	28,74	28,74	82,36	9,25	81,52	40,35	82,36	82,36	0	50	76,49
20	49,37	48,11	46,85	48,75	50	50	46,22	51,53	46,22	50	26,92	26,92	46,22	6,97	46,22	18,36	46,22	46,22	50	0	44,32
21	82,16	88,24	86,93	85,72	85,72	85,72	92,74	49	92,74	76,49	26,56	26,56	92,74	7,85	92,68	26,29	92,74	92,74	76,49	44,32	0

Matriks Uji Coba 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	92,32	33,35	22,74	22,74	22,74	85,72	42,86	85,72	42,86	47,83	47,83	85,72	9,34	85,72	46,16	85,72	85,72	42,86	17,86	76,94
2	92,32	0	33,35	22,74	22,74	22,74	92,86	42,86	92,86	42,86	47,83	47,83	92,86	9,34	92,86	46,16	92,86	92,86	42,86	17,86	84,63
3	33,35	33,35	0	63,65	63,65	63,65	33,35	60,72	33,35	60,72	60,87	60,87	33,35	8,49	33,35	46,16	33,35	33,35	60,72	42,86	22,24
4	22,74	22,74	63,65	0	95,46	100	22,74	53,58	22,74	53,58	52,18	52,18	22,74	8,49	22,74	38,47	22,74	22,74	53,58	42,86	18,2
5	22,74	22,74	63,65	95,46	0	95,46	22,74	53,58	22,74	53,58	52,18	52,18	22,74	8,49	22,74	38,47	22,74	22,74	53,58	42,86	18,2
6	22,74	22,74	63,65	100	95,46	0	22,74	53,58	22,74	53,58	52,18	52,18	22,74	8,49	22,74	38,47	22,74	22,74	53,58	42,86	18,2
7	85,72	92,86	33,35	22,74	22,74	22,74	0	42,86	100	42,86	47,83	47,83	100	9,34	100	46,16	100	100	42,86	17,86	85,72
8	42,86	42,86	60,72	53,58	53,58	53,58	42,86	0	42,86	100	67,86	67,86	42,86	15,26	42,86	64,3	42,86	42,86	100	42,86	35,72
9	85,72	92,86	33,35	22,74	22,74	22,74	100	42,86	0	42,86	47,83	47,83	100	9,34	100	46,16	100	100	42,86	17,86	85,72
10	42,86	42,86	60,72	53,58	53,58	53,58	42,86	100	42,86	0	67,86	67,86	42,86	15,26	42,86	64,3	42,86	42,86	100	42,86	35,72
11	47,83	47,83	60,87	52,18	52,18	52,18	47,83	67,86	47,83	67,86	0	100	47,83	11,02	47,83	76,94	47,83	47,83	67,86	42,86	39,15
12	47,83	47,83	60,87	52,18	52,18	52,18	47,83	67,86	47,83	67,86	100	0	47,83	11,02	47,83	76,94	47,83	47,83	67,86	42,86	39,15
13	85,72	92,86	33,35	22,74	22,74	22,74	100	42,86	100	42,86	47,83	47,83	0	9,34	100	46,16	100	100	42,86	17,86	85,72
14	9,34	9,34	8,49	8,49	8,49	8,49	9,34	15,26	9,34	15,26	11,02	11,02	9,34	0	9,34	9,34	9,34	9,34	15,26	5,95	7,63
15	85,72	92,86	33,35	22,74	22,74	22,74	100	42,86	100	42,86	47,83	47,83	100	9,34	0	46,16	100	100	42,86	17,86	85,72
16	46,16	46,16	46,16	38,47	38,47	38,47	46,16	64,3	46,16	64,3	76,94	76,94	46,16	9,34	46,16	0	46,16	46,16	64,3	35,72	38,47
17	85,72	92,86	33,35	22,74	22,74	22,74	100	42,86	100	42,86	47,83	47,83	100	9,34	100	46,16	0	100	42,86	17,86	85,72
18	85,72	92,86	33,35	22,74	22,74	22,74	100	42,86	100	42,86	47,83	47,83	100	9,34	100	46,16	100	0	42,86	17,86	85,72
19	42,86	42,86	60,72	53,58	53,58	53,58	42,86	100	42,86	100	67,86	67,86	42,86	15,26	42,86	64,3	42,86	42,86	0	42,86	35,72
20	17,86	17,86	42,86	42,86	42,86	42,86	17,86	42,86	17,86	42,86	42,86	42,86	17,86	5,95	17,86	35,72	17,86	17,86	42,86	0	14,29
21	76,94	84,63	22,24	18,2	18,2	18,2	85,72	35,72	85,72	35,72	39,15	39,15	85,72	7,63	85,72	38,47	85,72	85,72	35,72	14,29	0

Matriks Uji Coba 3

(Halaman ini sengaja dikosongkan)

BIODATA PENULIS

Ruchi Intan Tantra merupakan anak dari pasangan Bapak Singga Tantra dan Ibu Ninik Nadiarni. Lahir di Surabaya pada tanggal 12 Agustus 1994. Penulis menempuh pendidikan formal dimulai dari TKK Karitas II Surabaya (1998-2000), SDK Karitas II Surabaya (2000-2006), SMP Negeri 26 Surabaya (2006-2009), SMA Negeri 5 Surabaya (2009-2012) dan S1 Teknik Informatika ITS (2012-2016). Bidang studi yang diambil oleh penulis pada saat berkuliah di Teknik Informatika ITS adalah Algoritma dan Pemrograman (Alpro). Penulis aktif dalam organisasi seperti Himpunan Mahasiswa Teknik Computer-Informatika (2013-2014). Penulis juga aktif dalam berbagai kegiatan kepanitiaan yaitu SCHEMATICS 2013 divisi KESTARI dan SCHEMATICS 2014 divisi REEVA. Penulis memiliki hobi mendengarkan musik, membaca buku, dan menyukai hal baru. Penulis dapat dihubungi melalui email: ruchintan@gmail.com.