TUGAS AKHIR - KI141502

KLASTERISASI GRAF UNTUK LOG AUTENTIKASI DENGAN ALGORITMA *MOLECULAR COMPLEX DETECTION*

I GUSTI NGURAH ARYA BAWANTA
NRP 5113100007

Dosen Pembimbing I
Royyana M. Ijtihadie, S.Kom., M.Kom., Ph.D.

Dosen Pembimbing II
Hudan Studiawan, S.Kom., M.Kom.

Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2017

TUGAS AKHIR – KI141502

KLASTERISASI GRAF UNTUK LOG AUTENTIKASI DENGAN ALGORITMA *MOLECULAR COMPLEX DETECTION*

I GUSTI NGURAH ARYA BAWANTA
NRP 511310000

Dosen Pembimbing I
Royyana M. Ijtihadie, S.Kom., M.Kom., Ph.D.

Dosen Pembimbing II
Hudan Studiawan, S.Kom., M.Kom.

Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2017

(Halaman ini sengaja dikosongkan)

UNDERGRADUATE THESES – KI141502

GRAPH CLUSTERING FOR AUTHENTICATION LOG USING *MOLECULAR COMPLEX DETECTION ALGORITHM*

I GUSTI NGURAH ARYA BAWANTA
NRP 5113100007

First Advisor
Royyana M. Ijtihadie, S.Kom., M.Kom., Ph.D.

Second Advisor
Hudan Studiawan, S.Kom., M.Kom.

Department of Informatics
Faculty of Information Technology
Sepuluh Nopember Institute of Technology
Surabaya 2017

(Halaman ini sengaja dikosongkan)

LEMBAR PENGESAHAN

KLASTERISASI GRAF UNTUK LOG AUTENTIKASI DENGAN ALGORITMA *MOLECULAR COMPLEX DETECTION*

TUGAS AKHIR

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

Oleh:
I GUSTI NGURAH ARYA BAWANTA
NRP: 5113100007

Disetujui oleh Pembimbing Tugas Akhir:

1. Royyana M. Ijtihadie, S.Kom., M.Kom., Ph.D. .....
   (NIP. 197708242006041001) (Pembimbing 1)

2. Hudan Studiawan, S.Kom., M.Kom. .....
   (NIP. 198705112012121003) (Pembimbing 2)

SURABAYA
JUNI, 2017

(Halaman ini sengaja dikosongkan)

KLASTERISASI GRAF UNTUK LOG AUTENTIKASI DENGAN ALGORITMA *MOLECULAR COMPLEX DETECTION*

Nama Mahasiswa	**: I GUSTI NGURAH ARYA BAWANTA**
NRP	**: 5113100007**
Jurusan	**: Teknik Informatika FTIF-ITS**
Dosen Pembimbing 1	**: Royyana M. Ijtihadie, S.Kom., M.Kom., Ph.D.**
Dosen Pembimbing 2	**: Hudan Studiawan, S.Kom., M.Kom.**

Abstrak

Pelanggaran access control merupakan serangan yang utama beberapa tahun silam. Tipe dari klasterisasi pada ancaman keamanan seperti ini tersimpan dalam sebuah log autentikasi. Investigator forensik akan melakukan analisa pada log agar mengetahui ancaman yang mungkin terjadi. Analisa tersebut dapat dipermudah dengan mengklaster log tersebut. Namun, karena belum adanya visualisasi pada log yang telah diklaster membuat analisa log menjadi sulit dan membutuhkan waktu yang lama.

Penulis menggunakan metode dengan membuat visualisasi dari data log yang telah terklasterisasi. Tahap pertama adalah melakukan preprocessing pada log. Lalu data preprocessing akan diklaster menggunakan algoritma Molecular Complex Detection (MCODE). Setelah terklaster, maka akan penulis akan memvisualisasikan hasil dari klaster tersebut.

Pada tugas akhir ini, hasil yang didapat yaitu algoritma Molecular Complex Detection dapat digunakan sebagai metode untuk klasterisasi log autentikasi yang ada dan dapat memberikan hasil akurasi yang bagus. Adapun untuk visualisasi graf, aplikasi Gephi sudah mampu memberikan visualisasi yang jelas dalam melihat perbedaan log yang serangan dan log yang

bukan serangan. Sehingga, dapat mempermudah analisis pada tahap selanjutnya.

Kata kunci: Log Autentikasi, Algoritma Molecular Complex Detection, Visualisasi Log, Gephi

GRAPH CLUSTERING FOR AUTHENTICATION LOG USING *MOLECULAR COMPLEX DETECTION* ALGORITHM

Student's Name	**: I GUSTI NGURAH ARYA BAWANTA**
Student's ID	**: 5113100007**
Department	**: Teknik Informatika FTIF-ITS**
First Advisor	**: Royyana M. Ijtihadie, S.Kom., M.Kom., Ph.D.**
Second Advisor	**: Hudan Studiawan, S.Kom., M.Kom.**

Abstract

Access control violation is major attack these past years. The clustering of these type of violation is stored in the authentication log. Forensics investigators will analyze the log to understand the possible attack to the system. Therefore, because there is no visualization of the clustered log, analyzing process takes a long time.

Author use a method that make a visualization from clustered log data. The first step is to preprocess the authentication log. Then the preprocessed data will clustered with Molecular Complex Detection (MCODE) algorithm. After the log is clustered, then author visualize the result of cluster.

In this undergraduate thesis, the result obtained is Molecular Complex Detection algorithm can be used for clustering the authentication log given and have a good accuracy. As for graph visualization, Gephi can visualize authentication log clearly. So, Ghepi can show the differences between attack log and log that not an attack. So that the visualization can simplify the authentication log analyzing process.

Keywords : Authentication Log, Molecular Complex Detection Algorithm, Log Visualization, Gephi

(Halaman ini sengaja dikosongkan)

KATA PENGANTAR

Segala puji bagi TuhanYang Maha Esa, yang telah melimpahkan rahmat-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul **KLASTERISASI GRAF UNTUK LOG AUTENTIKASI DENGAN ALGORITMA MOLECULAR COMPLEX DETECTION.** Dengan pengerjaan Tugas Akhir ini, penulis bias memperdalam dan meningkatkan apa yang telah didapatkan selama menempuh perkuliahan di Teknik Informatika ITS. Selain itu, penulis juga dapat menghasilkan suatu implementasi dari apa yang telah dipelajari selama proses perkuliahan. Selesainya Tugas Akhir ini tidak lepas dari bantuan dan dukungan beberapa pihak. Pada kesempatan ini penulis mengucapkan syukur dan terima kasih kepada :

1. Tuhan Yang Maha Esa atas anugerah dan pencerahan-Nya yang tidak terkira kepada penulis.
2. Bapak I Gusti Ngurah Sudiana dan Ibu Ni Ketut Surinih sebagai orang tua penulis, adik serta keluarga besar penulis di kampung halaman yang telah memberikan dukungan moral, material serta doa yang tak terhingga semenjak penulis merantau hingga mampu menyelesaikan Tugas Akhir ini, bahkan membangkitkan semangat penulis dan menyadarkan penulis ketika berada pada kondisi terburuk.
3. Bapak Royyana Muslim Ijtihadie, S.Kom, M.Kom, Ph.D. selaku pembimbing I dan Bapak Hudan Studiawan, S.Kom, M.Kom selaku pembimbing II yang telah membantu, membimbing dan memotivasi penulis mulai dari pengerjaan proposal hingga terselesaikannya Tugas Akhir ini.
4. Bapak Darlis Herumurti, S.Kom., M.Kom., selaku Kepala Jurusan Teknik Informatika ITS pada masa pengerjaan Tugas Akhir, Bapak Radityo Anggoro, S.Kom., M.Sc. selaku coordinator Tugas Akhir, dan segenap dosen Teknik Informatika yang telah memberikan ilmu dan

pengalamannya kepada penulis sebagai bekal menjalani kehidupan yang sebenarnya.

5. Teman-teman Asrama Tirtha Gangga yang bersedia menjadi teman berbagi cerita dan bermain selama penulis berada di Surabaya.
6. Teman-teman TC 2013 yang selalu Bersama penulis mulai dari awal tahun perkuliahan hingga proses penyelesaian Tugas Akhir.
7. Serta semua pihak yang turut membantu dalam penyelesaian Tugas Akhir ini.

Penulis menyadari bahwa tugas Akhir ini masih memiliki kekurangan. Untuk itu, dengan kerendahan hati, penulis mengharapkan kritik dan saran dari pembaca untuk perbaikan kedepannya.

Surabaya, Juni 2017

I Gusti Ngurah Arya Bawanta

DAFTAR ISI

(Halaman ini sengaja dikosongkan)

DAFTAR GAMBAR

DAFTAR TABEL

(Halaman ini sengaja dikosongkan)

DAFTAR KODE SUMBER

(Halaman ini sengaja dikosongkan)

BAB I
PENDAHULUAN

1.1 Latar Belakang

Aktivitas yang terdapat pada *secure shell* (SSH) server, baik aktivitas yang normal maupun aktivitas yang merupakan serangan terhadap sistem tersimpan dalam log file. Catatan dari aktivitas tersebut dapat mencapai ratusan ribu baris. Dalam aktivitas tersebut memungkinkan adanya pelanggaran *access control* yang merupakan serangan yang utama beberapa tahun silam. Catatan dari adanya pelanggaran pada kemungkinan ancaman keamanan *access control* seperti ini tersimpan dalam sebuah log autentikasi.

Investigator forensik akan melakukan analisa pada log agar mengetahui ancaman yang mungkin terjadi. Analisa tersebut dapat dipermudah dengan mengklaster log tersebut [1]. Log yang telah terklaster lebih mudah dianalisis karena log sudah terkelompok berdasarkan aktivitasnya. Namun, proses analisis dari log yang telah terklaster masih memerlukan proses analisis yang lama. Hal tersebut terjadi karena investigator forensik harus melihat data log per baris. Oleh karena itu, penulis memvisualisasikan hasil dari klasterisasi log, sehingga investigator forensik menganalisis log hanya dengan melihat visualisasi dari log.

Penulis menggunakan metode dengan membuat visualisasi dari data log yang telah terklaster. Pertama-tama penulis melakukan *preprocessing* pada log menghilangkan kata yang tidak diperlukan pada proses klasterisasi. Lalu data *preprocessing* akan diklaster menggunakan algoritma *Molecular Complex Detection* (MCODE) [2]. Setelah terklaster, maka penulis akan memvisualisasikan hasil dari klaster tersebut. Visualisasi yang akan ditampilkan penulis akan langsung dapat menunjukkan log yang merupakan serangan dan log yang bukan merupakan serangan.

Hasil yang diharapkan dari klasterisasi tersebut adalah bisa mendapat laporan analisis untuk investigator forensik. Adapun algoritma yang digunakan diharapkan dapat mengklaster data log dengan cepat dan akurat. Serta visualisasi graf dapat mempermudah pembacaan data log bagi investigator forensik.

1.2 Rumusan Masalah

Tugas akhir ini mengangkat beberapa rumusan masalah sebagai berikut:

1. Apakah algoritma *molecular complex detection* dapat diimplementasikan pada kasus tugas akhir ini?
2. Berapakah *range threshold* yang bagus untuk *molecular complex detection*?
3. Berapakah nilai *threshold* yang bagus untuk algoritma *molecular complex detection*?
4. Berapakah nilai akurasi terbaik yang didapatkan?
5. Apa pengaruh log yang digunakan terhadap *running time* dari sistem?

1.3 Batasan Permasalahan

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

1. Menggunakan dataset log autentikasi pada sistem operasi Linux.
2. Menggunakan bahasa pemrograman *Python* dan *Library NetworkX* untuk pembuatan graph pada sistem operasi *Windows 8.1*.
3. Menggunakan *Gephi* untuk visualisasi.

1.4 Tujuan

Tujuan dari tugas akhir ini adalah sebagai berikut:

1. Membuat perangkat lunak yang dapat mengklaster log autentikasi pada sistem operasi Linux.

2. Memvisualisasi log autentikasi yang telah diklaster dalam bentuk graf.

1.5 Manfaat

Dengan dibuatnya tugas akhir ini diharapkan dapat memberikan manfaat pada dunia forensik untuk mempermudah menganalisis log dengan cara memberikan visualisasi dari dari log yang telah dimasukkan ke klaster-klaster.

1.6 Metodologi

Pembuatan tugas akhir ini dilakukan dengan menggunakan metodologi sebagai berikut:

1.6.1 Penyusunan Proposal

Tahap awal tugas akhir ini adalah menyusun proposal tugas akhir. Pada proposal, diajukan gagasan untuk mengklaster log autentikasi dengan algoritma *Molecular Complex Detection*.

1.6.2 Studi Literatur

Pada tahap ini dilakukan untuk mencari informasi dan studi literatur apa saja yang dapat dijadikan referensi untuk membantu pengerjaan Tugas Akhir ini. Informasi didapatkan dari buku dan literatur yang berhubungan dengan metode yang digunakan. Informasi yang dicari adalah algoritma MCODE, log autentikasi, kontrol akses dan visualisasi pada *Gephi*. Tugas akhir ini juga mengacu pada literatur jurnal karya Hudan Studiawan, Baskoro Adi Pratomo dan Radityo Anggoro dengan judul "*Connected Component Detection for Authentication Log Clustering"* yang diterbitkan pada tahun 2016.

1.6.3 Implementasi Perangkat Lunak

Implementasi merupakan tahap untuk membangun metode-metode yang sudah diajukan pada proposal Tugas Akhir. Untuk membangun algoritma yang telah dirancang sebelumnya, maka dilakukan implementasi dengan menggunakan suatu perangkat lunak yaitu PyCharm sebagai IDE dan *Package* Anaconda.

1.6.4 Pengujian dan Evaluasi

Pada tahap ini algoritma yang telah disusun diuji coba dengan menggunakan data uji coba yang ada. Data uji coba tersebut diuji coba dengan menggunakan suatu perangkat lunak dengan tujuan mengetahui kemampuan metode yang digunakan untuk mengklaster log autentikasi. Data hasil klaster akan dibandingkan dengan data yang telah disediakan dan dibandingkan dengan algoritma *Adjusted Rand Index* (ARI). Hasil evaluasi juga mencakup *running time* dari setiap tahap klastering log.

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 forensik digital, kontrol akses, log autentikasi, *Molecular Complex Detection*, visualisasi dengan *gephi*, bahasa pemrograman *python* dan *library networkx.*

3. Bab III. Perancangan Perangkat Lunak
   Bab ini berisi pembahasan mengenai tahap-tahap dari aplikasi serta perancangan dari metode *Molecular Complex Detection* yang digunakan untuk mengklasterisasi log autentikasi.
4. Bab IV. Implementasi
   Bab ini menjelaskan implementasi yang berbentuk *Pseudocode* yang berupa *Pseudocode* dari metode *Molecular Complex Detection.*
5. Bab V. Hasil Uji Coba dan Evaluasi
   Bab ini berisikan hasil uji coba dari metode *Molecular Complex Detection* yang digunakan untuk mengklaster log autentikasi yang sudah diimplementasikan pada *Pseudocode*. Uji coba dilakukan dengan menggunakan dataset log autentikasi Linux. Hasil evaluasi mencakup nilai ARI yang menunjukkan kebenaran dari metode klastering tersebut serta *running time* dari metode yang ada.
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 *Pseudocode* program secara keseluruhan.

(Halaman ini sengaja dikosongkan)

BAB II
TINJAUAN PUSTAKA

Bab ini berisi pembahasan mengenai teori-teori dasar yang digunakan dalam tugas akhir. Teori-teori tersebut diantaranya adalah *Molecular Complex Detection*, log autentikasi dan beberapa teori lain yang mendukung pembuatan tugas akhir.

2.1 Forensik Digital

Forensik digital adalah aktivitas yang berhubungan dengan pemeliharaan, identifikasi, ekstraksi, dan dokumentasi bukti digital dalam kejahatan. Bukti-bukti dalam forensik digital sangatlah banyak, seperti semua bukti fisik (komputer, harddisk, dan lain lain), dokumen-dokumen yang tersimpan dalam komputer (gambar, pdf, video, dan lain lain), lalu lintas data dalam jaringan dan lain sebagainya. Karena luasnya cangkupan, forensik digital dibagi menjadi beberapa cabang seperti forensik komputer, forensik analisis data, forensik jaringan, forensik perangkat bergerak, forensik basis data, dan lain lain [3].

2.2 Kontrol Akses

Kontrol akses merupakan cara yang digunakan oleh administrator suatu sistem mengatur akses ke sistem tersebut. Akses yang diatur dapat berupa akses dokumen, akses file dan sebagainya. Izin untuk mengaksesnya disebut dengan otorisasi. Adapun cara untuk otorisasi yaitu dengan autentikasi [4]. Dalam tugas akhir ini, penulis menggunakan kontrol akses pada sistem operasi Linux.

2.3 Log Autentikasi

Log adalah daftar tindakan kejadian atau aktivitas yang telah terjadi dalam sebuah sistem komputer. Log ini tersimpan

dalam sebuah file log. File log dihasilkan oleh sistem untuk mencatat hal-hal yang terjadi pada sebuah sistem. Sehingga dalam log juga terdapat hal-hal yang berpotensi menimbulkan masalah. Masalah yang dapat ditimbulkan dapat berupa masalah keamanan sistem maupun yang lainnya.

Penulis menggunakan dataset log autentikasi pada kontrol akses di sistem operasi linux. Log ini terdapat pada *auth.log* pada folder */var/log* pada linux berbasis debian atau folder */var/log/secure* pada redhat. Log tersebut menyimpan *Pluggable Authentication Modules* (PAM) yang menyimpan akses user yang diperbolehkan dan akses *Secure Shell* (SSH), baik yang berhasil maupun gagal [5]. Log ini sangat berharga bagi investigator forensik untuk menganalisis suatu insiden maupun jejak dari penyerang. Contoh log autentikasi dapat dilihat pada Gambar 2.1.

```
Dec  1 00:02:02 161.166.232.16 sshd[23346]: Accepted publickey for XXXXX from 161.166.232.12 port 33045 ssh2
Dec  1 00:02:02 161.166.232.16 sshd[23346]: pam_unix(sshd:session): session closed for user XXXXX
Dec  1 00:02:02 161.166.232.16 sshd[23346]: pam_unix(sshd:session): session opened for user XXXXX by (uid=0)
Dec  1 00:02:02 161.166.232.16 sshd[23348]: Received disconnect from 161.166.232.12: 11: disconnected by user
Dec  1 00:02:02 161.166.232.16 sshd[23350]: Accepted publickey for XXXXX from 161.166.232.12 port 33046 ssh2
Dec  1 00:02:02 161.166.232.16 sshd[23350]: pam_unix(sshd:session): session closed for user XXXXX
Dec  1 00:02:02 161.166.232.16 sshd[23350]: pam_unix(sshd:session): session opened for user XXXXX by (uid=0)
Dec  1 00:02:02 161.166.232.16 sshd[23352]: Received disconnect from 161.166.232.12: 11: disconnected by user
Dec  1 00:02:03 161.166.232.14 sshd[10361]: Accepted publickey for XXXXX from 161.166.232.12 port 46925 ssh2
Dec  1 00:02:03 161.166.232.14 sshd[10361]: pam_unix(sshd:session): session opened for user XXXXX by (uid=0)
Dec  1 00:02:03 161.166.232.14 sshd[10368]: pam_unix(sshd:session): session closed for user XXXXX
Dec  1 00:02:03 161.166.232.14 sshd[10370]: Received disconnect from 161.166.232.12: 11: disconnected by user
Dec  1 00:02:03 161.166.232.14 sshd[10372]: Accepted publickey for XXXXX from 161.166.232.12 port 46929 ssh2
Dec  1 00:02:03 161.166.232.14 sshd[10372]: pam_unix(sshd:session): session closed for user XXXXX
Dec  1 00:02:03 161.166.232.14 sshd[10372]: pam_unix(sshd:session): session opened for user XXXXX by (uid=0)
Dec  1 00:02:03 161.166.232.14 sshd[10374]: Received disconnect from 161.166.232.12: 11: disconnected by user
Dec  1 00:02:03 161.166.232.14 sshd[10378]: Accepted publickey for XXXXX from 161.166.232.12 port 46931 ssh2
Dec  1 00:02:03 161.166.232.14 sshd[10378]: pam_unix(sshd:session): session closed for user XXXXX
```

Gambar 2.1 Contoh Log Autentikasi

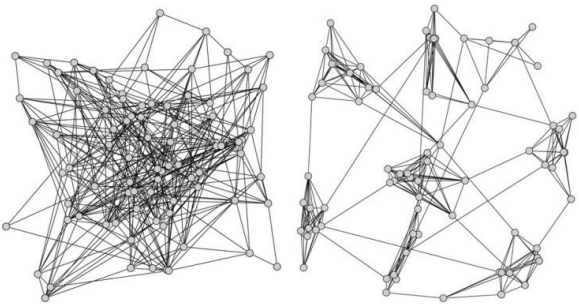
Penulis menggunakan dua buah dataset dalam tugas akhir ini, yaitu dataset *Hofstede2014* dan dataset *SecRepo*. Dataset *Hofstede2014* didapatkan dari penelitian oleh Rick Hofstede dkk. Log ini dikumpulkan dari bulan november tahun 2013 sampai dengan februari 2014 pada jaringan Universitas Twente [6]. Sedangkan dataset *Secrepo* didapatkan dari *Security Repository* yang dibuat oleh Mike Sconzo. Adapun dataset *SecRepo* ini mengandung banyak data log yang merupakan usaha login sistem yang gagal.

2.4 Klasterisasi

Klasterisasi adalah teknik umum yang digunakan untuk analisis data statistik, pengenalan pola, analisis *image* dan bioinformatika. Klasterisasi merupakan proses untuk mengelompokkan objek yang sama menjadi kelompok yang berbeda, atau membagi dataset menjadi subset, sehingga masing-masing data dalam subset berdasarkan ukuran yang telah ditentukan. Dapat dikatakan bahwa klaster adalah kelompok dari objek yang sama dari objek tersebut dan berbeda dari objek yang dimiliki oleh klaster yang lainnya. Adapun pembagian jenis algoritma klasterisasi data bisa dilakukan dengan klasterisasi hirarki maupun klasterisasi secara partisi. Contoh dari klasterisasi secara hirarki adalah dengan algoritma perhitungan jarak seperti jarak Manhattan dan jarak Euclidean. Sedangkan contoh dari klasterisasi secara partisi adalah algoritma K-means dan algoritma K-medoid [7].

2.5 Klasterisasi Graf

Klasterisasi graf adalah cara untuk mengelompokkan *node* dari graf menjadi klaster dengan mempertimbangkan struktur dari *edge* agar terdapat banyak edge antar didalam sebuah klaster dan sedikit diantara klaster. Tujuan dari klasterisasi graf ini adalah untuk membagi dataset menjadi klaster sehingga elemen-elemen yang telah ditentukan dalam sebuah klaster menjadi sama atau saling terhubung dengan menggunakan acuan yang telah ditentukan sebelumnya. Untuk mempermudah penjelasan tentang klasterisasi graf, dapat dilihat contoh pada Gambar 2.2. Pada Gambar 2.2, gambar graf kiri merupakan graf yang random dan belum terklaster, setelah dilakukan klasterisasi graf, terlihat pada graf kanan.



Gambar 2.2 Contoh Graf yang Belum Terklaster dan Graf yang Sudah Terklaster

2.6 Algoritma *Molecular Complex Detection* (MCODE)

MCODE merupakan algoritma yang akan mengecek kepadatan dari wilayah node sebuah jaringan node. Algoritma ini biasanya digunakan untuk mengelompokkan jenis protein kompleks. Sering digunkaan pada jaringan PPI [2]. Ada tiga tahap dari algoritma MCODE, yaitu *vertex weighting*, *complex prediction* dan *post-processing*(opsional) [8].

Pada tahap *vertex weighting*, vertex yang ada akan diberikan nilai *weight* berdasarkan kepadatan jaringan lokal vertex tersebut. Kemudian pada tahap *complex prediction* akan mencari jaringan lokal yang memungkinkan untuk melakukan klasterisasi dengan mengecek nilai *weight* masing-masing node. Pada tahap *post-processing* terdapat dua tahap yang dapat terjadi, yaitu *haircut* dan *fluff*. Pada tahap *haircut*, node yang hanya terhubung dengan sebuah node lain akan dipisahkan dari jaringan. Sedangkan pada tahap *fluff*, jangkauan dari jaringan klaster yang didapatkan sebelumnya akan dilebarkan dengan sebuah node diluar jaringan tersebut [8].

Tahap *post-processing* dapat menyebabkan hilangnya node yang telah dibuat sebelumnya dan merupakan tahap yang optional, maka penulis tidak mengikut sertakan tahap *post-processing* dalam tugas akhir ini.

2.7 Attack pada Autentikasi

Seperti yang kita ketahui, sistem autentikasi memungkinkan untuk diserang. Penyerangan terhadap autentikasi pun beragam, yaitu dapat dengan mencoba kombinasi *username* dan *password* dari pengguna yang terdaftar ataupun dengan menyerang server agar dapat memasuki sistem yang ada. Serangan ini sangat merugikan jika sampai berhasil, namun sistem log sudah mencatat semua kegiatan dari sistem sehingga dapat melacak jika terjadi serangan pada sistem. Berikut adalah daftar dari kemungkinan serangan yang dicatat oleh sistem.

- **Invalid user :** penyerang atau bot menggunakan brute-force attack untuk masuk ke dalam sistem
- **Recieved disconnect from :** penyerang menggunakan kode sendiri untuk brute-force ke sistem
- **Did not recieve identification string from :** penyerang mencoba untuk mencari username dan password pada sistem
- **Connection reset by peer :** terdapat masalah pada *authentication key* dan kemungkinan terjadi serangan brute-force
- **Could not write ident string :** terdapat orang yang menggunakan *port scanner* dan mengenai port 22.
- **Too many authentication failure :** limit MaxAuthTries sudah melebihi batas, kemungkinan terjadi serangan brute-force
- **Bad protocol version identification :** kemungkinan ada virus yang mencoba menyerang sistem dengan mencari kelemahan sistem
- **Connection closed :** Kemungkinan terjadinya serangan brute-force
- **Reverse mapping checking getaddrinfo :** client yang terhubung tidak mempunyai atau mempunyai bad DNS

- **This does not map back to the address :** reverse IP lookup tidak berhasil
- **Failed password :** penyerang kemungkinan mencoba password berulang kali
- **Ignoring max retries :** kemungkinan terjadi serangan brute-force
- **Recieved disconnect :** ada kemungkinan penyerang menggunakan kode sendiri untuk menyerang server, kemungkinan serangan brute-force
- **Open failed :** koneksi terputus sebelum mencapai SSH tunnel. Ada kemungkinan bukan serangan, tapi administrator lebih baik mengeceknya.
- **Root login refused :** akses root *restricted* , namun penyerang mencoba untuk masuk
- **Protocol major versions differ :** kemungkinan menyerang SSH server
- **Failed none :** penyerang mencoba masuk dengan password kosong

Pencatatan dengan kata diatas kemungkinan adalah serangan [9]. Penulis menggunakan daftar serangan diatas untuk mengklasifikasikan sebuah log merupakan sebuah serangan atau bukan.

2.8 Anaconda

Anaconda adalah *open data science platform* yang dibangun menggunakan bahasa pemrograman Python. Versi Anaconda *open-source* menyediakan distribusi dengan performa tingkat tinggi dari bahasa pemrograman Python dan R yang berisikan lebih dari 100 paket untuk *data science.* Anaconda menyediakan CLI (*command line interface*) berupa perintah "conda" sebagai manajemen lingkungan untuk bahasa pemrograman Python. Perintah "conda" bisa untuk *create, export, list, remove* dan *update* lingkungan yang bisa digunakan untuk

versi bahasa pemrograman Python atau paket yang dipasang didalamnya [10].

2.9 Bahasa Pemrograman *Python*

Python merupakan bahasa pemrograman yang lebih menekankan pada keterbacaan kode, sehingga lebih mudah dipahami dari kebanyakan bahasa pemrograman. Selain itu, *Python* juga mendukung hampir semua sistem operasi bahkan Sistem Operasi Linux. Kelebihan dari *python* yaitu merupakan bahasa pemrograman dinamis yang dilengkapi dengan manajemen memori otimatis [11].

2.10 NumPy

NumPy adalah paket dasar untuk komputasi ilmiah menggunakan Python yang berisi antara lain :

1. Objek *array* N dimensi
2. Fungsi yang mutakhir
3. Kakas bantu untuk mengintegrasi dengan *Pseudocode* C/C++ dan Fortran
4. Sangat berguna untuk aljabar linier, Fourier Transform, dan kemampuan angka *random*

Selain digunakan untuk hal ilmiah, NumPy juga bisa digunakan untuk *container* multidimensi yang efisien untuk data generik. Tipe data *arbitrary* dapat didefinisikan, ini memungkinkan NumPy secara lancar dan cepat mengintegrasikan dengan banyak tipe basis data. NumPy adalah singkatan dari *Numeric Python* atau *Numerical Python*. Adalah sebuah modul *Open Source* untuk bahasa pemrograman Python yang menyediakan fungsi-fungsi yang telah *precompiled.* NumPy memperkaya kemampuan bahasa pemrograman Python dengan data struktur yang efisien untuk komputasi yang membutuhkan *arrays* atau *matrices* multidimensi [12].

2.11 SciPy

SciPy adalah singkatan dari *Scientific Python*. SciPy adalah modul *Library Open Source* yang digunakan untuk bahasa pemrograman Python. SciPy sering digunakan bersama dengan modul NumPy. SciPy menambah kemampuan NumPy dalam fungsi seperti *minimization, regression,* dan *Fast Fourier-transformation* [13].

2.12 Library cProfile

Library cProfile digunakan untuk memunculkan *running time* dari sebuah fungsi pada *Python*. *cProfile* merupakan sebuah fitur yang sudah ada saat *Python* terinstall. Adapun statistik fungsi-fungsi yang ditampilkan termasuk subfungsi yang ada didalam fungsi yang dijalankan. Contoh hasil dari *cProfile* dapat dilihat pada Gambar 2.3.

```
         6692454 function calls (6685117 primitive calls) in 19.127 seconds

   Ordered by: cumulative time, file name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   19.127   19.127 pygraphc:12(main)
        1    0.000    0.000   11.595   11.595 pygraphc:90(graph_streaming)
        1    0.084    0.084   11.549   11.549 GraphStreaming.py:50(gephi_streaming)
     3624    0.034    0.000   11.227    0.003 client.py:54(flush)
```

Gambar 2.3 Contoh statistik *running time* dari *cProfile*

Pada Gambar 2.3, terdapat beberapa *header* dari hasil statistik *cProfile*, yaitu :

- *Ncalls* menunjukkan jumlah fungsi dipanggil
- *Totime* menunjukkan waktu yang dihabiskan oleh fungsi
- *Percall* menunjukkan rata-rata waktu yang dihabiskan fungsi dalam sekali pemanggilan (*totime* dibagi *ncalls*)
- *Cumtime* menunjukkan waktu kumulatif dari fungsi
- *Percall* menunjukkan waktu kumulatif tiap pemanggilan fungsi (*cumtime* dibagi *ncalls*)

- *Filename:lineno(function)* menunjukkan nama file, line keberapa dan nama fungsi yang dipanggil

Selain itu, terlihat bahwa total waktu yang dibutuhkan dalam pemanggilan pada Gambar 2.3 adalah 19,127 detik [14].

2.13 NetworkX

NetworkX adalah *library* dari *python* yang berfungsi untuk mempelajari graf dan jaringan. Beberapa fitur-fitur dari *networkx* adalah sebagai berikut :

1. Mengikuti struktur data bahasa *python* untuk graph, digraph, dan multigraph.
2. Node dapat berupa apa saja (misalkan teks, gambar, XML).
3. Edges dapat menyimpan data apa saja (misalkan *weight*, waktu).
4. Memiliki generator untuk graph klasik, graph acak dan jaringan sintetik.
5. Algoritma graph standar.
6. Kemampuan mengolah struktur jaringan dan metrik analisis jaringan.
7. Lisensi BSD open source.
8. Telah diuji: lebih dari 1800 unit test, > 90% cakupan kode.
9. Manfaat tambahan dari *python* : *prototyping* cepat, mudah untuk mengajar, multi-platform [15].

2.14 String Similarity dengan Levenshtein Distance

String similarity ini digunakan untuk membandingkan dua buah *string* dalam proses pembuatan graf. Algoritma yang digunakan untuk membandingan dua buah *string* tersebut adalah algoritma *Levenshtein Distance* atau *edit distance*. Dalam teori informasi, *Levenshtein distance* dua *string* adalah jumlah minimal

operasi yang dibutuhkan untuk mengubah suatu *string* ke *string* yang lain, di mana operasi-operasi tersebut adalah operasi penyisipan, penghapusan, atau penyubstitusian sebuah karakter. Algoritma ini dinamakan berdasarkan Vladimir Levenshtein yang ditemukannya pada tahun 1965. Pada tugas akhir ini, *Levenshtein distance* dirujuk dengan menggunakan kata jarak saja agar lebih singkat.

Untuk menghitung jarak dari dua *string* dapat menggunakan cara dengan matriks *(n + 1) x (m+1)* dimana n adalah panjang *string* pertama dan m adalah panjang *string* kedua. Berikut dua *string* yang akan digunakan sebagai contoh :

RONALDINHO
ROLANDO

Untuk mengubah *string* RONALDINHO menjadi ROLANDO diperlukan 6 operasi, yaitu :

1. Menstubstitusikan N dengan L :
   RO**N**ALDINHO ==> RO**L**ALDINHO
2. Menstubstitusikan L dengan N :
   ROLA**L**DINHO ==> ROLA**N**DINHO
3. Menstubstitusikan I dengan O :
   ROLAND**I**NHO ==> ROLAND**O**NHO
4. Menghapus O :
   ROLANDONH**O** ==> ROLANDONH
5. Menghapus H :
   ROLANDON**H** **==>** ROLANDON
6. Menghapus N :
   ROLANDO**N** ==> ROLANDO

Dengan menggunakan representasi matriks dapat ditunjukkan pada Gambar 2.4 :

		R	O	N	A	L	D	I	N	H	O
	0	1	2	3	4	5	6	7	8	9	10
R	1										
O	2										
L	3										
A	4										
N	5										
D	6										
O	7										

Gambar 2.4 Representasi matriks perbandingan *string*

Pada Gambar 2.4, elemen baris 1 kolom 1 (M[1,1]) adalah jumlah operasi yang diperlukan untuk mengubah *substring* dari kata ROLANDO yang diambil mulai dari karakter awal sebanyak 1 (R) ke *substring* dari kata RONALDINHO yang diambil mulai dari karakter awal sebanyak 1 (R). sementara elemen M[3,5] adalah jumlah operasi antara ROL (*substring* yang diambil mulai dari karakter awal sebanyak 3) dengan RONAL (*substring* yang diambil mulai dari karakter awal sebanyak 5). Berarti elemen M[p,q] adalah jumlah operasi antara *substring* kata pertama yang diambil mulai dari awal sebanyak p dengan *substring* kata kedua yang diambil dari awal sebanyak q. Sehingga dengan peraturan ini matriks dapat diisi, dan menghasilkan perthitungan seperti Gambar 2.5.

		R	O	N	A	L	D	I	N	H	O
	0	1	2	3	4	5	6	7	8	9	10
R	1	0	1	2	3	4	5	6	7	8	9
O	2	1	0	1	2	3	4	5	6	7	8
L	3	2	1	1	2	3	4	5	6	7	8
A	4	3	2	2	1	2	3	4	5	6	7
N	5	4	3	3	2	2	3	4	5	6	7
D	6	5	4	4	3	3	2	3	4	5	6
O	7	6	5	5	4	4	3	3	4	5	6

Gambar 2.5 Hasil representasi matriks dua string

Elemen terakhir (yang paling kanan bawah) adalah elemen yang nilainya menyatakan jarak kedua *string* yang dibandingkan [16].

2.15 Evaluasi dengan *Adjusted Rand Index* (ARI)

Untuk melakukan evaluasi dari tugas akhir ini, penulis menggunakan ARI. ARI digunakan untuk menghitung kemiripan antara dua buah klaster dengan mempertimbangkan semua *sample* dan menghitung pasangan klaster yang sama maupun berbeda pada klaster yang benar maupun *predicted*. Nilai dari ARI berada diantara 0.0 sampai 1.0, semakin mirip klasternya maka nilai ARI akan mendekati 1.0. Adapun syarat untuk menggunakan ARI yaitu memerlukan file *groundtruth* yang merupakan file referensi serta file *predicted* yang merupakan file yang akan dievaluasi [17].

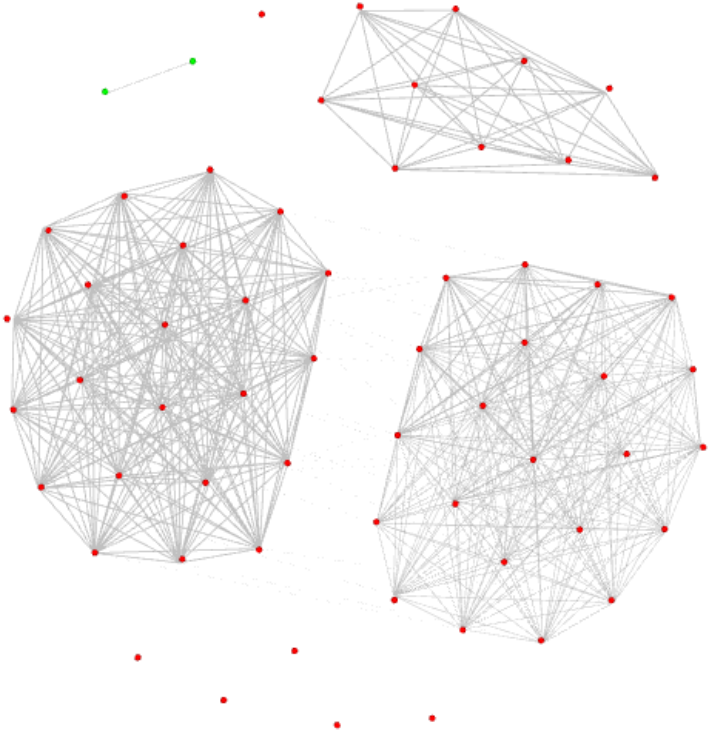
$$RI = \frac{a + b}{C_2^{n_{samples}}} \tag{2.1}$$

JIka C adalah *class* dari *groundtruth* dan K adalah klasternya. Maka nilai dari a adalah jumlah dari elemen yang sama pada set C dan set K. Sedangkan, nilai dari b adalah jumlah dari elemen yang berbeda pada set C dan set K. Dan nilai dari $C_2^{n_{samples}}$ adalah total dari pasangan yang mungkin dalam dataset tanpa mengurutkan dataset. Namun, rumus 2.1 tidak menjamin nilai dari RI mendekati nol jika ditentukan dataset yang acak, misalkan jika jumlah cluster dalam suatu urutan sama dengan nilai pada sampel. Oleh karena itu, digunakan nilai *expected* RI atau $E[RI]$ yang berguna untuk menangani dataset yang acak dengan dilakukan penyesuaian RI atau disebut dengan *adjusted rand index* seperti rumus 2.2.

$$ARI = \frac{RI - E[RI]}{\max(RI) - E[RI]} \tag{2.2}$$

2.16 Visualisasi dengan *Gephi*

Gephi adalah paket perangkat lunak untuk visualisasi dan analisis jaringan. *Gephi* merupakan perangkat lunak yang bersifat *open-source*. Dalam penggunaannya *gephi* dapat membuat visualisasi graf dengan *streaming* jalannya *code* atau dengan mengimport file graf. *Gephi* juga dapat melakukan *import* data dari jejaring sosial seperti *Facebook* dan *Twitter* lalu mengklaster dan membuat grafnya [18]. Contoh dari hasil visualialisasi pada Gephi dapat dilihat pada Gambar 2.6.



Gambar 2.6 Contoh Visualisasi pada Gephi

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 klasterisasi graf pada log autentikasi dengan algoritma *molecular complex detection.*

3.1 Data

Pada sub bab ini akan dijelaskan mengenai data yang digunakan sebagai masukan perangkat lunak untuk selanjutnya diolah dan dilakukan pengujian sehingga menghasilkan data keluaran yang diharapkan.

3.1.1 Data Masukan

Data masukan adalah data yang digunakan sebagai masukan awal dari sistem. Data yang digunakan dalam perangkat lunak klasterisasi graf pada log autentikasi terdiri dari dua buah dataset. Penulis membagi tiap dataset sebagai data masukan untuk *training* data dan *testing* data. Dataset pertama adalah dataset Hofstede2014 yang berjumlah 61 data, serta dibagi menjadi 41 data untuk *training* dan 20 data untuk *testing*. Dataset yang kedua adalah dataset SecRepo yang berjumlah 32 data serta dibagi menjadi 20 data untuk *training* dan 12 data untuk *testing*.

Contoh log sebagai data masukan ditunjukkan pada Gambar 3.1.

Gambar 3.1 Contoh dataset log sebagai data masukan

3.1.2 Data Keluaran

Data masukan akan diproses dengan menggunakan metode *preprocessing* dan *molecular complex detection.* Hasil dari proses pengklasteran tersebut akan divisualisasikan, disimpan dalam beberapa file yang nantinya akan digunakan untuk evaluasi dari algoritma *molecular complex detection*, serta disimpan dalam bentuk ringkasan ke dalam sebuah file.

3.2 Desain Umum Sistem

Rancangan perangkat lunak untuk klasterisasi graf pada log autentikasi menggunakan algoritma *molecular complex detection* dimulai dengan proses *preprocessing* pada log autentikasi. Setelah dilakukan proses *preprocessing* log autentikasi, maka akan dilakukan proses *graph modelling* atau tahap permbuatan graf. Setelah graf berhasil dibuat, maka graf tersebut akan diklaster dengan algoritma *molecular complex detection*. Kemudian akan dihitung nilai *adjusted rand index* (ARI) untuk proses evaluasi nantinya. Lalu penulis akan memvisualisasikan hasil dari klasterisasi tersebut ke Gephi. Diagram alir proses ini dapat dilihat dalam Gambar 3.2.

Proses awal yaitu *preprocessing*, sangat diperlukan dalam mempercepat proses klaster dan membuat hasil dari klaster menjadi lebih akurat. Perlu diketahui bahwa data log yang digunakan tidak diubah oleh penyerang. Proses *preprocessing* ini berguna untuk menghilangkan bagian-bagian dari log yang tidak diperlukan, misalkan pada klasterisasi ini penulis hanya fokus pada isi dari log, sehingga bagian *date time* tidak perlu diperhatikan. Perlu diketahui bahwa penulis tidak menghilangkan bagian dari log, melainkan hanya mengabaikan bagian log yang tidak diperlukan pada proses klaster. Diagram alir dari tahap *preprocessing* dapat dilihat pada Gambar 3.3.

Setelah tahap *preprocessing* selesai dilakukan, maka proses selanjutnya adalah proses pembuatan graf. Graf yang akan dibuat mempunyai beberapa bagian, yaitu node, edge dan weight. Pada tahap pembuatan node, jika ada data log yang sama, maka akan dijadikan satu node. Untuk pembuatan edge, dilihat dari *string similarity* (kesamaan string) dari dua buah node yang berbeda. Jika ada sedikit kesamaan, maka edge akan dibuat antar node tersebut. Untuk mencocokan kesamaan antar dua buah *string* tersebut, digunakan nilai *levenshtein distance* yang dimodifikasi.

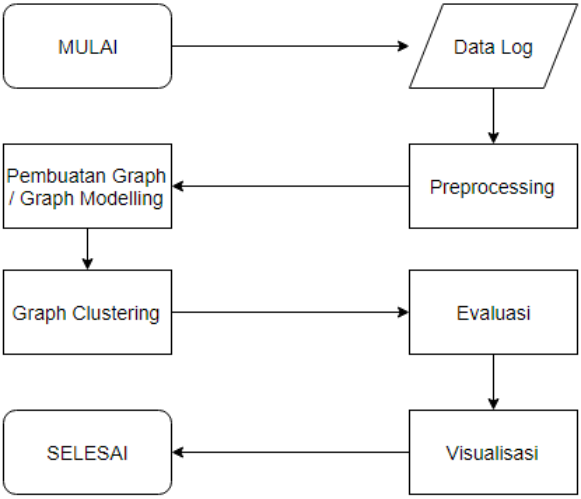
Setelah mendapatkan nilai dari *levenshtein distance*, maka nilai tersebut akan dibandingkan dengan *threshold* (batas) yang telah ditentukan, jika nilai *levenshtein distance* lebih besar dari *threshold*, maka *edge* akan dibuat antar dua node tersebut. Nilai *levenshtein distance* akan digunakan sebagai nilai *weight* dari edge tersebut. Diagram alir untuk proses pembuatan graf ini dapat dilihat pada Gambar 3.4.

Graf yang telah berhasil dibuat sebelumnya akan diklaster menggungkan algoritma *molecular complex detection*. Ada tiga tahap dari algoritma ini, yaitu *vertex weighting, complex prediction* dan *post-processing*. Namun, penulis hanya menggunakan tahap *vertex weighting* dan *complex prediction* pada tugas akhir ini karena tahap *post-processing* dapat menghilangkan node yang ada sedangkan penulis memerlukan semua node yang pernah dibuat untuk proses evaluasi nantinya.

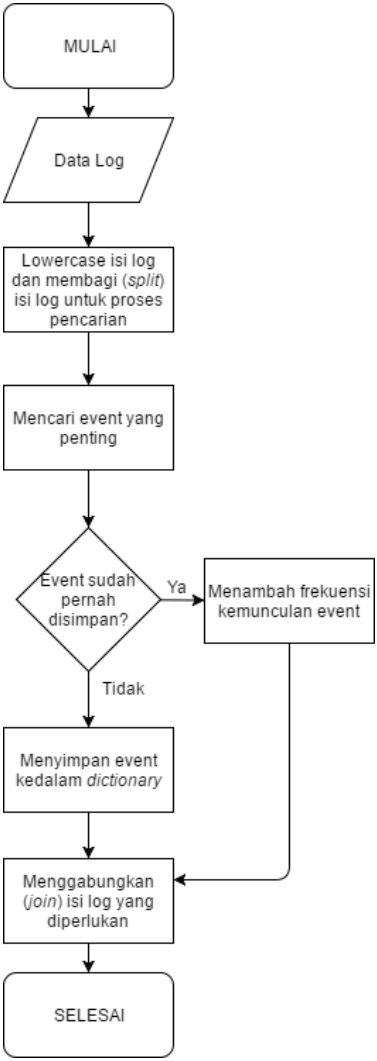
Diagram alir dari algoritma *molecular complex detection* dapat dilihat pada Gambar 3.5.

Pada tahap *vertex weighting*, tiap *vertex* yang ada akan diberikan *weight*. Nilai dari *weight* tergantung dari tetangga dari node tersebut. Pertama tahap ini akan memilih *node* yang akan diberikan weight, lalu mencari nilai *k-core* yang terbesar. Nilai *k-core* merupakan jumlah edge yang terhubung ke *node* yang dipilih sebelumnya. Kemudian menggunakan jumlah *edge* dan kemungkinan *edge*nya, dicari nilai kepadatan *core*nya. Nilai dari *weight*nya adalah nilai kepadatan *corenya* dikali dengan *k-core*.

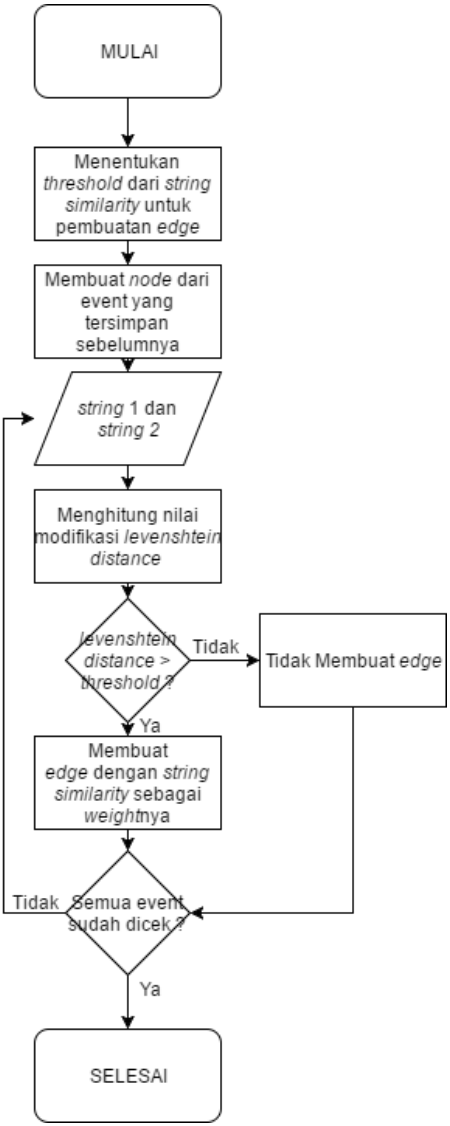
Pada tahap *complex prediction*, *range* klaster akan bertambah besar yang berawal dari *seed* keluar. Pertama penulis menentukan nilai *cut off* untuk menentukan *threshold* yang digunakan sebagai batas klaster. Nilai *threshold* inilah nantinya yang akan menentukan klaster-klaster dari suatu *node*. Jika nilai *weight* dari *node* tersebut lebih besar dari *threshold* maka *node tersebut* akan masuk ke dalam klaster yang ditentukan pada saat itu.



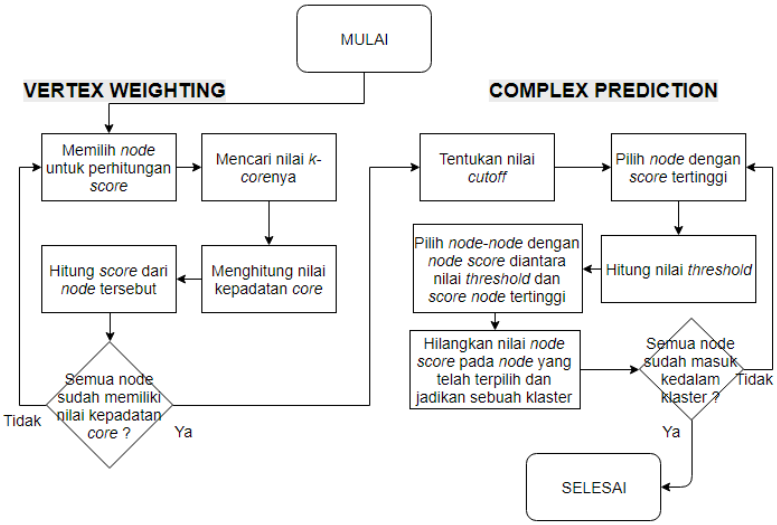
Gambar 3.2 Diagram alir pengerjaan tugas akhir



Gambar 3.3 Diagram alir proses *preprocessing*



Gambar 3.4 Diagram alir proses *graph modelling*



Gambar 3.5 Diagram alir algoritma *molecular complex detection*

3.3 Modifikasi *Levenshtein Distance*

Levenshtein distance merupakan salah satu metode yang digunakan untuk menentukan nilai dari jarak perubahan dari *string* pertama menjadi *string* kedua. Penulis memodifikasi nilai dari *levenshtein distance* menjadi bernilai dalam *range* 0 sampai 1. Semakin nilai dari modifikasi *levenshtein* distance mendekati 1, maka kedua string tersebut semakin mirip. Untuk mengubah nilainya penulis menggunakan rumus 3.1.

$$string_{similarity} = 1 - \frac{levenshtein_{distance}}{\max(len(str1), len(str2))} \qquad (3.1)$$

Untuk mencari *string similarity*, penulis membagi hasil dari *levenshtein distance* dengan panjang *string* yang terpanjang diantara dua *string* yang ada. Hasil pembagian tersebut

merupakan nilai antara 0 sampai 1, dimana nilai yang mendekati 1 maka kedua *string* tersebut akan semakin berbeda. Karena penulis menginginkan agar nilai yang mendekati 1 merupakan nilai yang menunjukan bahwa kedua *string* semakin mirip, maka penulis mengurangkan 1 dengan hasil tersebut untuk mendapatkan nilai *string similarity*.

Contohnya pada *string* "RONALDINHO" dan "ROLANDO". Awalnya nilai dari *levenshtein distance* merupakan nilai jumlah operasi untuk mengubah *string* pertama menjadi *string* kedua, yaitu 6 jumlah operasi pada *string* "RONALDINHO" dan "ROLANDO". Maka hasil bagi dari *levenshtein distance* dan panjang *string* terpanjang adalah 0.6, yang merupakan hasil dari 6 dibagi panjang *string* terpanjang, yaitu "RONALDINHO" (10 karakter). Sedangkan nilai dari *string similarity* dari kedua *string* tersebut adalah 0.4.

3.4 *Molecular Complex Detection*

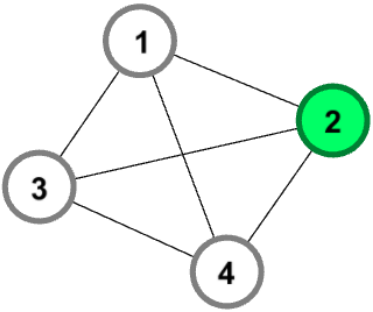
Molecular complex detection merupakan algoritma yang digunakan untuk klasterisasi graf pada data log autentikasi. Terdapat 3 tahap dalam algoritma *molecular complex detection*, yaitu *vertex weighting*, *complex prediction* dan *post-processing*. Namun, penulis hanya menggunakan tahap *vertex weighting* dan *complex prediction* dari algoritma *molecular complex detection*. Hal tersebut karena pada tahap *post-processing*, ada kemungkinan sebuah *node* akan dihilangkan, sehingga dapat menghambat proses evaluasi nantinya.

3.4.1 *Vertex Weighting*

Vertex weighting merupakan tahap pertama yang dilakukan untuk mengklasterisasi graf pada log autentikasi. Pada tahap ini, tiap *node* akan diberikan *score* yang nantinya akan digunakan untuk penentuan klaster. Adapun penentuan nilai *score* berdasarkan pada tetangga dalam jaringan lokal dari *node*

tersebut. Semakin banyak tetangganya, maka nilai dari *score*nya akan semakin besar.

Pertama-tama, *node* akan dipilih untuk diberikan *score*, misalkan ada *node* yang dipilih pada klaster seperti Gambar 3.6 adalah node 2. Lalu mencari nilai dari *k-core* terbesar, nilai *k-core* dapat dilihat dari jumlah edge pada *node* 2. Pada contoh klaster di Gambar 3.6, nilai *k-core*nya adalah 3.



Gambar 3.6 Pemilihan *node* dalam sebuah klaster

Setelah mendapatkan nilai *k-core*nya, maka tahap selanjutnya adalah menentukan nilai kepadatan *core*nya. Nilai kepadatan *core* dapat dicari dengan rumus 3.2.

$$CD = \frac{TE}{TPE} \tag{3.2}$$

Dimana CD adalah adalah *core density (*kepadatan *core),* TE adalah total *edge* dan TPE adalah total *possible edge* (edge yang mungkin) . Total *edge* adalah jumlah dari *edge* yang ada pada klaster, pada Gambar 3.4 terlihat bahwa total *edge*nya adalah 6. Sedangkan, nilai TPE ditentukan dari jumlah *edge* yang terhubung dengan *node* yang dipilih ditambahkan dengan 1 dan dikuadratkan. Maka nilai TPE dari contoh pada Gambar 3.6 adalah 4^2, yaitu 16. Dari nilai tersebut diketahui bahwa nilai dari kepadatan *core* dari *node* pada Gambar 3.6 adalah 0,375.

Tahap selanjutnya adalah mencari *score* dari node yang dipilih, yaitu node 2. Nilai *score* merupakan nilai perkalian antara *k-core* dan kepadatan *core*, rumusnya dapat dilihat pada rumus 3.3. Pada contoh klaster pada Gambar 3.6 diketahaui bahwa nilai *k-core*nya adalah 3 dan nilai kepadatan *core*nya adalah 0,375. Maka, *score* dari *node* yang terpilih pada Gambar 3.6adalah 1,125. Proses ini akan diulang sampai seluruh *node* mendapatkan *score*.

$$Score = K_{core} x\ CD \tag{3.3}$$

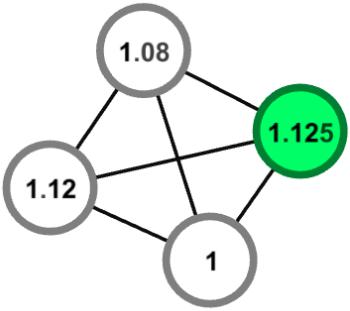
3.4.2 *Complex Prediction*

Complex prediction merupakan tahap lanjutan untuk mengklaster log autentikasi. Diperlukan inisialisasi nilai *cutoff* yang digunakan untuk menentukan *threshold* nantinya. Nilai *cutoff* ini merupakan nilai *range* dari sebuah klaster, misalkan nilai *score* dari sebuah *node* adalah 1 dan nilai *cutoff* 0.2, maka nilai *threshold*nya adalah 0,8. Hal tersebut berarti *node* yang mempunyai nilai 0,8 sampai 1 merupakan satu klaster. Semakin besar nilai dari *cutoff*, maka semakin besar juga jangkauan klaster. Perhitungan tersebut merujuk pada rumus 3.6.

$$threshold = (1 - cutoff) x\ score_{node} \tag{3.4}$$

Tahap pertama pada dari *complex prediction* adalah penentuan nilai *cutoff*, yang nantinya akan digunakan untuk menghitung nilai *threshold*, misalkan nilai *cutoff*nya adalah 0,1. Setelah menentukan nilai *cutoff*, maka akan dicari *node* yang memiliki nilai paling besar, untuk mempermudah penjelasan dapat dilihat contoh dari graf pada Gambar 3.7. Maka *node* yang terpilih pada pemilihan *node* dengan *score* terbesar adalah *node* dengan *score* 1,125. Setelah *node* terpilih, maka nilai *threshold*nya akan dihitung, karena nilai *cutoff*nya adalah 0,1 maka nilai *threshold*nya adalah 1.0125.

Selanjutnya *node-node* yang terhubung dengan *node* yang terpilih dan memiliki *score* diantara *threshold* dan *score node* yang terpilih akan menjadi satu klaster. Pada Gambar 3.7, *node-node* yang dijadikan sebuah klaster adalah *node* yang mempunyai nilai antara 1.0125 sampai 1,125. Node yang termasuk dalam satu klaster pada Gambar 3.7 adalah *node* dengan *score* 1.2 dan 1.125. Jika sebuah *node* sudah masuk ke sebuah klaster, maka nilai *score* dari *node* tersebut akan dihilangkan, hal tersebut berguna agar *node* tidak terpilih pada tahap pemilihan *node* dengan *score* terbesar. Proses ini diulang sampai seluruh *node* masuk ke dalam suatu klaster.



Gambar 3.7 Contoh graf dengan *score* pada *node*

(Halaman ini sengaja dikosongkan)

BAB IV
IMPLEMENTASI

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

4.1 Lingkungan Implementasi

Implementasi klasterisasi graf pada log autentikasi meggunakan algoritma *molecular complex detection* menggunakan spesifikasi perangkat keras dan perangkat lunak seperti yang ditunjukkan pada Tabel 4.1.

Tabel 4.1 Lingkungan Implementasi Perangkat Lunak

Perangkat	Jenis Perangkat	Spesifikasi
Perangkat Keras	Prosesor	Intel(R) Core(TM) i5-2.6 GHz
	Memori	8 GB DDR3
Perangkat Lunak	Sistem Operasi	Windows 8.1
	Perangkat Pengembang	JetBrains PyCharm 2.7.4 dan Gephi 0.9.1

4.2 Implementasi

Pada sub bab implementasi ini menjelaskan mengenai pembangunan perangkat lunak secara detail dan menampilkan *Pseudocode* yang digunakan mulai tahap *preprocessing* hingga klasterisasi graf menggunakan algoritma *molecular complex detection*. Pada tugas akhir ini data yang digunakan, seperti yang telah dijelaskan di bab sebelumnya, yaitu terdiri dari dua jenis dataset. Dataset jenis pertama berjumlah 61 buah data log yang

terbagi menjadi 41 data log untuk *training* data dan 20 data log untuk *testing* data. Dataset jenis kedua berjumlah 32 yang terbagi menjadi 20 data log untuk *training* dan 12 data log untuk *testing*.

4.2.1 *Preprocessing*

Tahap pertama dari *preprocessing* adalah menginisialisasi dari variabel-variabel yang akan digunakan nantinya. Inisialisasi dilakukan dengan menyiapkan variabel-variabel *array* untuk menyimpan *event* dan *integer* untuk menghitung panjang dari lognya. Proses inisialisasi ini dapat dilihat pada *Pseudocode* 4.1.

```
1   CLASS Preprocess(object):
2       FUNCTION __init__(self, logfile):
3           logfile <- logfile
4           logs <- []
5           loglength <- 0
6           event_list <- []
7           event_unique <- []
8           word_count <- {}
9       ENDFUNCTION
```

Pseudocode* 4.1 Kode inisialisasi variabel yang digunakan pada tahap *preprocessing

Selanjutnya, data log yang ingin di*preprocessing* akan dibaca terlebih dahulu. Pertama-tama, file log akan dibaca perbaris, lalu data log dan panjang log akan disimpan kedalam sebuah variabel. Proses pembacaan data log ini dapat dilihat pada *Pseudocode* 4.2.

```
1   FUNCTION __read_log(self):
2       f <- READ(logfile)
3       logs <- f.readlines()
4       loglength <- len(logs)
5   ENDFUNCTION
```

***Pseudocode* 4.2 Kode *read* log *file* dan menyimpan datanya ke variabel**

Kemudian, data log sebelumnya akan diubah menjadi *lowercase* dan log akan dibagi berdasar *message* tiap log. Setelah itu, kata-kata yang tidak diperlukan akan dihilangkan. Proses penghilangan kata-kata yang tidak diperlukan ini dapat dilihat pada *Pseudocode* 4.3.

```
1   FUNCTION get_event(self, doc, total_docs, docs):
2        logs_lower                                  <-
3   LOWERCASE(logs.SPLIT(message))
4        doc <- SUB('[^a-zA-Z]','',doc)
         additional_stopwords <- ['preauth', 'from',
6        'xxxxx', 'for', 'port', 'sshd', 'ssh']
7        FOR a in additional_stopwords:
8             doc <- doc.REPLACE(a,'')
9        doc <- doc.REPLACE('_','')
10       doc <- ' '.JOIN(doc.SPLIT())
         stopwords          <-          corpus.stopwords.
11       words('english')
         stopwords_result <- [w.LOWERCASE()] FOR w
         in doc.SPLIT() if w.LOWERCASE() not in
12       stopwords]
13       RETURN doc
    ENDFUNCTION
```

***Pseudocode* 4.3 Kode fungsi untuk menghilangkan kata yang tidak diperlukan**

Tahap selanjutnya yaitu menyimpan hasil akhir dari penghilangan kata-kata yang tidak diperlukan ke sebuah variabel. Variabel-variabel tersebut berbentuk *array* untuk penyimpanan log dan *integer* untuk penyimpanan panjang log. Kode dari proses ini dapat dilihat pada *Pseudocode* 4.4.

```
1   FUNCTION do_preprocess(self):
2        events_list <- []
3        events_unique <- []
4        index <- 0
5        index_log <- 0
6        FOR l in logs_lower:
7             auth_split <- l.SPLIT()
8             event_type,        event_desc        <-
              auth_split[0].split('[')[0],          '
              '.join(auth_split[1:])
9             event <- event_type + ' ' + event_desc
              events_list.append(event)
10            preprocessed_event <- get_event(event,
              logs_total, logs_lower)
              check_events_unique                   <-
              [e[1]['preprocessed_event']  FOR  e  in
              events_unique]
11            IF    preprocessed_event    not    in
              check_events_unique:
12                  events_unique.append([index,
                    {'event': event, 'status' : '' ,
                    'cluster' : index, 'frequency' :
                    1  ,  'member'  :  [index_log]  ,
                    'preprocessed_event'          :
                    preprocessed_event,   'attack'  :
                    attack}])
13                  INCREMENT(index)
14            ELSE :
15                  FOR e in events_unique:
16                       IF   preprocessed_event   ==
                         e[1]['preprocessed_event']:
17                            member <- e[1]['member']
18                            member.append(index_log)
19                            e[1]['member'] <- member
20                            e[1]['frequency']     <-
                              INCREMENT(
                              e[1]['frequency'] )
21            index_log = index_log + 1
22  ENDFUNCTION
```

***Pseudocode* 4.4 Kode untuk menyimpan data log ke variabel**

4.2.2 *Graph Modelling*

Tahap pertama dari proses *graph modelling* adalah inisialisasi dari variabel-variabel yang digunakan untuk pembuatan graf. Adapun variabel yang akan dilakukan inisialisasi berupa variabel yang berbentuk *list* yang digunakan untuk penyimpanan data *edge*, *weight*, dan *events* dari log, variabel graf yang digunakan untuk pembuatan graf serta variabel *integer* sebagai batas untuk nilai *string similarity* yang nantinya digunakan. *Pseudocode* dari proses ini dapat dilihat pada *Pseudocode* 4.5.

```
1   CLASS CreateGraph(object):
2       FUNCTION     __init__(self,     events_unique,
        string_similarity_threshold):
3           g <- nx.MultiGraph()
4           edges_dict <- {}
5           edges_weight <- []
6       ENDFUNCTION
```

Pseudocode* 4.5 Kode inisialisasi variabel pada proses *Graph Modelling

Variabel *g* merupakan sebuah variabel dari *library NetworkX* yang nantinya digunakan untuk pembuatan graf.

4.2.2.1 Modifikasi *Levenshtein Distance*

Pertama-tama, masing-masing *string* akan dibagi menurut kata-katanya. Tiap kata dari suatu *string* akan disimpan ke sebuah variabel *array*. Lalu dilakukan pengecekan untuk meminimalisir perhitungan jika ada kondisi yang terpenuhi sesuai yang telah ditentukan. Kemudian dilakukan inisialisasi variabel untuk penyimpanan nilai *Levenshtein Distance.* Kemudian nilai pengecekan pada setiap kata-kata yang ada pada *string* pertama dan *string* kedua. Perlu diketahui bahwa nilai dari *Levenshtein Distance* yang digunakan merupakan perbandingan kata-kata antar dua *string*, bukan perbandingan huruf pada dua *string*.

Adapun nilai *cost* antar dua buah kata-kata bernilai 1 jika kata tersebut berbeda dan 0 jika kata tersebut sama. Setelah seluruh kata-kata dari seluruh *string* dicek, maka akan dihitung nilai akhir dari *Levenshtein Distance*, yaitu dengan cara membulatkan nilai dari *Levenshtein Distance* yang telah dimodifikasi sejauh tiga desimal. Adapun perhitungan nilai *Levenshtein Distance* yang telah dimodifikasi yaitu dengan mengurangkan 1 dengan nilai *Levenshtein Distance* dan dibagi dengan panjang *string* terpanjang antara dua buah *string* yang dibandingkan. Proses ini dapat dilihat pada kode pada *Pseudocode* 4.6.

```
1   FUNCTION levenshtein(string1, string2):
2       word1 <- string1.SPLIT(" ")
3       word2 <- string2.SPLIT(" ")
4       IF string1 == string2 :
5           RETURN 0
6       ELIF LENGTH(string1) == 0:
7           RETURN LENGTH(string2)
8       ELIF LENGTH(string2) ==0:
9           RETURN LENGTH(string1)
10      v0 <- [NONE] * (LENGTH(word2)+1)
11      v1 <- [NONE] * (LENGTH(word2)+1)
12      FOR i IN RANGE(LENGTH(v0)):
13          v0[i] <- i
14      FOR i IN RANGE(LENGTH(word1)):
15          v1[0] <- i+1
16          FOR j IN RANGE(LENGTH(word2)):
17              cost <- 0 IF word1[i] == word2[j]
                ELSE 1
18              v1[j+1] <- MIN(v1[j]+1, v0[j+1]+1,
                v0[j]+cost)
19          FOR j IN RANGE(LENGTH(v0)):
20              v0[j] <- v1[j]
21      ret_val   <-   ROUND(1-v1[LENGTH(word2)]   /
        MAX(LENGTH(word1), LENGTH(word2)),3)
22      RETURN ret_val
23  ENDFUNCTION
```

Pseudocode* 4.6 Kode untuk modifikasi algoritma *levenshtein distance

4.2.2.2 Pembuatan Graf

Seperti yang kita ketahui, variabel *g* merupakan sebuah variabel dari *library NetworkX* yang digunakan untuk pembuatan graf. Maka dari itu, pembuatan node hanya dilakukan dengan memanggil fungsi yang telah disediakan oleh *Library NetworkX*. Proses tersebut dapat dilihat pada *Pseudocode* 4.7.

```
1  FUNCTION __create_nodes(self):
2      g.add_nodes_from(events_unique)
3  ENDFUNCTION
```

***Pseudocode* 4.7 Kode untuk membuat *node* pada graf**

Setelah pembuatan node berhasil, maka tahap selanjutnya adalah dengan membuat *edge* dari *node* yang telah dibuat sebelumnya. Pembuatan *edge* dilakukan dengan membandingkan nilai *threshold* dari *string similarity* antara dua buah *node*. Jika nilai dari *string similarity* antar dua buah *node* tersebut lebih besar dari *threshold* yang ditentukan, maka akan dibuatkan *edge* pada dua *node* tersebut. Adapun nilai *weight* dari *edge* yang dibuat meruapakan nilai dari *string similarity*-nya. Oleh karena penulis memerlukan nilai *string similarity*, maka penulis memanggil fungsi **levenshtein** yang telah dibuat sebelumnya pada *Pseudocode* 4.6. Kode dari pembuatan *edge* ini dapat dilihat pada *Pseudocode* 4.8.

```
1   FUNCTION __create_edges(self):
2       edges_combination <- [eu[0] FOR eu IN
        events_unique]
3       edge_index <- 0
4       FOR ec IN COMBINATION(edges_combination,2):
5           string1 <- g.node[ec[0]]['event']
6           string2 <- g.node[ec[1]]['event]
7           string_similarity                    <-
            levenshtein(string1, string2)
8           IF          string_similarity         >
            string_similarity_threshold:
9               g.add.edge(ec[0],ec[1],weight    <-
                string_similarity)
10              edges_weight.append((ec[0],ec[1],
                string_similarity))
11              edges_dict[(ec[0],ec[1])]        <-
12              edge_index
13              INCREMENT(edge_index)
14  ENDFUNCTION
```

***Pseudocode* 4.8 Kode untuk membuat *edge* pada graf**

4.2.3 *Molecular Complex Detection*

Setelah berhasil melakukan pembuatan *node* dari dataset yang ada, maka tahap selanjutnya adalah melakukan klaster dari dataset dengan algoritma *molecular complex detection*. Awalnya penulis membuat inisialisasi variabel-variabel yang digunakan dalam proses klaster ini. Adapun penulis membuat sebuah *Class Zerodict* yang digunakan untuk inisialisasi *dictionary*, *Pseudocode* dari *Class Zerodict* dapat dilihat pada *Pseudocode* 4.9.

```
1   CLASS zerodict(dict):
2       FUNCTION __missing__(self, k):
3           RETURN 0
4       ENDFUNCTION
```

Pseudocode* 4.9 Kode untuk *Class Zerodict

Ada beberapa variabel yang akan diinisialisasi oleh penulis, yaitu variabel *graph_real*, *graph*, *weight_threshold*,

edges dan *add_empty*. Variabel *graph_real* merupakan variabel yang meyimpan isi data dari graf yang asli, variabel *graph* berisi data graf yang berbentuk *dictionary*, *weight_threshold* berisi data *integer* yang merupkan nilai batas yang digunakan sebagai penentu klaster nantinya, *edges* merupakan variabel yang menyimpan data dari *edge* graf. *Pseudocode* dari inisialisasi variabel ini dapat dilihat pada *Pseudocode* 4.10.

```
1   CLASS MCODE:
2       FUNCTION __init(self, graph, edge_dict,
        edges_weight, weight_threshold):
3           graph_real <- graph
4           graph <- defaultdict(set)
5           weight_threshold <- 1-weight_threshold
6           edges <- defaultdict(zerodict)
7           FOR item IN edge_dict AND edges_weight:
8               graph[item[0]].add(item[1])
9               graph[item[1]].add(item[0])
10              edges[item[0]][item[1]] <- item[2]
11              edges[item[1]][item[0]] <- item[2]
12          addEmpty <- set([])
13          FOR item IN graph_real.node:
14              IF graph[item] == addEmpty:
15                  graph[item].add(None)
16      ENDFUNCTION
```

Pseudocode* 4.10 Kode inisialisasi algoritma *molecular complex detection

4.2.3.1 *Vertex Weighting*

Tahap kedua dari algoritma *molecular complex detection* adalah *vertex weighting*, pada tahap ini, penulis memberikan *score* pada *node* yang ada. Pertama-tama, nilai *core density* dari tiap node akan dihitung dengan cara membagi jumlah *edges* dari node yang terpilih dibagi dengan jumlah *edges* yang mungkin. Setelah itu, nilai dari *k-core* terbesar akan dicari, nilai ini digunakan untuk perhitungan nilai dari *score node* yang ada. Setelah mendapatkan nilai *k-core* terbesar, maka nilai dari *score* akan dihitung dengan mengalikan nilai *k-core* terbesar dengan

nilai *core density* pada tiap node. *Pseudocode* dari proses ini dapat dilihat pada *Pseudocode* 4.11.

```
FUNCTION _get_vertex_weights(self):
    empty_dict <- {}
    weights <- DICT((v,0) IF edges[v] == empty_dict ELSE (v,SUM(edges[v].itervalues()) / LENGTH(edges[v])^2) FOR v IN graph)
    FOR v IN enumerate(graph):
        neighborhood <- set((v,)) | graph[v]
        k <- 2
        WHILE True:
            invalid_nodes <- True
            WHILE invalid_nodes AND neighborhood:
                invalid_nodes <- set(n FOR n IN neighborhood IF LENGTH(graph[n]) AND neighborhood)<k)
                neighborhood <- neighborhood - invalid_nodes
            IF NOT neighborhood:
                BREAK
            weights[v] <- MAX(weights[v], k*SUM(edges[v][n] FOR n IN neighborhood) / LENGTH(neighborhood)^2)
            INCREMENT(k)
ENDFUNCTION
```

Pseudocode* 4.11 Kode program untuk mencari nilai *weight* pada *node

4.2.3.2 *Complex Prediction*

Tahap selanjutnya dari algoritma *molecular complex detection* adalah *complex prediction.* Pertama-tama, penulis membuat sebuah fungsi untuk memberikan id pada *node*. Fungsi ini berfungsi untuk mempermudah proses klaster nantinya. *Pseudocode* dari fungsi ini dapat dilihat pada *Pseudocode* 4.12.

```
1   FUNCTION set_cluster_id(graph,clusters):
2       FOR           cluster_id,cluster           IN
3       cluster.iteritem():
4           FOR node IN cluster:
5               graph.node[node]['cluster']        <-
6               cluster_id
    ENDFUNCTION
```

Pseudocode* 4.12 Kode untuk memberikan id pada tiap *node

Selanjutnya, penulis menginisialisasi beberapa variabel yang digunakan. Variabel-variabel yang diinisialisasi itu adalah variabel *unvisited, num_cluster, clusters* dan *cluster_id*. Variabel *unvisited* digunakan untuk menyimpan *node* yang belum diklaster, *num_clusters* untuk menyimpan jumlah dari klaster, variabel *clusters* digunakan untuk menyimpan data dari klaster yang ada dan *cluster_id* digunakan untuk menyimpan nilai dari id pada klaster.

Setelah itu, penulis mengecek seluruh *node* lalu mengklasternya sesuai dengan batas yang telah ditentukan sebelumnya. Adapun penulis menambahkan beberapa variabel, yaitu variabel *frontier, seed* dan *w*. Variabel *frontier* berfungsi untuk menyimpan data *node* yang sedang diproses pada saat itu, variabel *seed* berisi seluruh data *node* beserta dengan *core density* dari *node* tersebut dan variabel *w* digunakan untuk menyimpan nilai *threshold* dari klaster.

Selanjutnya penulis mengecek *node* yang ada pada variabel *frontier* untuk diproses. Lalu *node* yang ada pada variabel *unvisited* yang berisi *node* pada variabel *frontier* akan dihapus dan klaster akan ditambahkan dengan *node* yang ada pada variabel *frontier*. Setelah itu, nilai *score* dari *node* pada variabel *frontier* akan dibandingkan dan *node* akan dimasukkan ke sebuah klaster jika nilai *score* dari *node* memenuhi. Kemudian penulis mengganti data *node* yang ada pada variabel *frontier* dengan data *node* yang baru pada variabel *unvisited*. Setelah itu, nilai dari *cluster_id* akan ditambahkan 1. Kemudian penulis memanggil fungsi *set_label_id* untuk memberikan id pada klaster.

Pseudocode dari proses diatas dapat dilihat pada *Pseudocode* 4.13.

```
1   FUNCTION do_mcode(self):
2       unvisited <- set(graph)
3       num_clusters <- 0
4       clusters <- {}
5       cluster_id <- 0
6       FOR seed IN SORTED(weights,key=weights.get,
        reverse=True):
7           IF seed NOT IN unvisited:
8               CONTINUE
9           cluster <- set((seed,)), set((seed,))
10          frontier <- set((seed,)), set((seed,))
11          w <- weights[seed] * weight_threshold
12          WHILE frontier:
13              cluster.update(frontier)
14              unvisited <- unvisited - frontier
15              frontier <- set(n FOR n IN
                set.union(*(graph[n] FOR n IN
                frontier)) AND unvisited IF
                weights[n]>w)
16          IF LENGTH(cluster) > 0:
                clusters[cluster_id] <-
                list(cluster)
17              INCREMENT(cluster_id)
18      set_cluster_id(graph_real,clusters)
19      RETURN clusters
    ENDFUNCTION
```

Pseudocode* 4.13 Kode program untuk melakukan tahap *complex prediction

4.2.4 Visualisasi dengan Gephi

Untuk melakukan visualisasi dengan *Gephi*, penulis membuat *Class* yang bernama *GephiStreaming*. Perlu diketahui nama dari *Workspace* pada *Gephi* haruslah *workspace0* karena pada *Class GephiStreaming* hanya menerima *workspace0*. Tahap pertama pada *Class GraphStreaming* adalah inisialisasi variabel yang nantinya akan digunakan. Adapun variabel-variabel yang diinisialisasi adalah *g*, *edges*, *sleep_time*, dan *gstream*. Variabel *g*

adalah variabel yang menyimpan graf yang telah terklaster. Variabel *edges* berfungsi untuk menyimpan nilai *edge* dari graf yang telah terklaster. Variabel *sleep_time* berfungsi untuk menyimpan nilai *delay* yang digunakan pada saat *streaming* node. Variabel *gstream* merupakan sebuah variabel yang menyimpan *GephiClient*. *GephiClient* adalah protokol dan *plugin* yang diperlukan agar dapat mengkoneksikan *Python* dan *Gephi*. *Pseudocode* dari inisialisasi variabel tersebut dapat dilihat pada *Pseudocode* 4.14.

```
1   CLASS GraphStreaming(object):
2       FUNCTION __init__(self,graph_clusters,edges,
        sleep_time=10):
3           g <- graph_clusters
4           edges <- edges
5           sleep_time <- sleep_time
6           gstream <- GephiClient('
            http://localhost:8080/workspace0 ',
            autoflush=True)
7           gstream.clean()
8       ENDFUNCTION
```

***Pseudocode* 4.14 Kode program untuk inisialisasi variabel yang digunakan untuk *streaming* pada Gephi**

Selanjutnya, penulis memvisualisasikan tiap-tiap *node* dan *edge* pada *Gephi*. Pertama-tama, penulis menginisialisasi variabel *x,y,z* yang menunjukkan koordinat dari *node* yang ingin ditampilkan pada *Gephi*, serta variabel *r,g,b* yang berfungsi untuk mewarnai *node* ketika ditampilkan pada *Gephi*. Setelah itu, penulis melakukan iterasi pada graf yang ada dan mengecek tiap *node* serta menampilkan *node* tersebut. Adapun penentuan koordinat dari *node* yaitu dengan cara acak. Setelah koordinat dari *node* sudah dihitung, maka akan dilakukan pengecekan pada *node*, jika *node* tersebut merupakan serangan, maka variabel *r,g,* dan *b* akan berisi 1,0, dan 0, jika bukan serangan maka nilainya menjai 0,1 dan 0. Nilai tersebut merupakan nilai pada RGB dengan dibagi 255. Maka nilai RGB pada variabel *r,g,* dan *b* jika

terjadi serangan adalah 255, 0 dan 0 yang mengindikasikan warna merah, serta 0,255 dan 0 yang mengindikasikan warna hijau jika tidak terjadi serangan. Setelah itu, *node* akan ditampilkan pada *Gephi* serta nilai dari variabel *x*,*y* dan *z* yang merupakan koordinat dari *node* akan diubah menjadi 0 agar tidak mengganggu proses selanjutnya. Tahap tersebut akan terus diulang sampai seluruh *node* yang ada berhasil ditampilkan pada *Gephi*. Setelah itu, *edge* akan ditampilkan pada *Gephi* dengan *weight* yang telah ditentukan seperti sebelumnya. *Pseudocode*nya dapat dilihat pada *Pseudocode* 4.15.

```
1   FUNCTION gephi_streaming(self):
2       PRINT 'streaming node ...'
3       x <- 0
4       y <- 0
5       z <- 0
6       r <- {}
7       g <- {}
8       b <- {}
9       temp_node <- 0
10      FOR node IN g.nodes_iter(data=TRUE):
11          x <- x + RANDOM(-100,100)
12          y <- y + RANDOM(-100,100)
13          z <- z + RANDOM(-100,100)
14          temp_node <- node[0]
15          IF node[1]['attack'] == 1:
16              r[temp_node] <- 1
17              g[temp_node] <- 0
18              b[temp_node] <- 0
19          ELSE :
20              r[temp_node] <- 0
21              g[temp_node] <- 1
22              b[temp_node] <- 0
23          node_attributes <- { 'size': 10, 'r':
            r[temp_node], 'g': g[temp_node], 'b':
            b[temp_node], 'x' : x , 'y' : y , 'z' :
            z                                    ,
            'label' : node[1]['preprocessed_event'],
            'frequency'    :    node[1]['frequency'],
            'cluster':            node[1]['cluster'],
            'attack':node[1]['attack']}
24          gstream.add_node(node[0],
            **node_attributes)
25          x <- 0
26          y <- 0
27          z <- 0
28      PRINT 'streaming edge ...'
29      edges_only <- edges_keys()
30      FOR e IN edges_only:
31          TRY:
32              weight <- g[e[0]][e[1]]
33              edges_index <- edges[(e[0],e[1])]
34          EXCEPT KeyError:
35              weight <- g[e[1]][e[0]]
36              edges_index <- edges[(e[1],e[0])]
```

```
37          node_attributes      <-      {'r':192/225
            ,'g':192/225 ,'b':192/225}
38          gsteam.add_edge(edge_index,  e[0],  e[1],
            weight=weight[0]['weight'],
            directed=FALSE, **node_attributes)
39  ENDFUNCTION
```

***Pseudocode* 4.15 Kode program untuk melakukan *streaming* pada Gephi**

Setelah berhasil menampilkan *node* dan *edge* pada *Gephi*, penulis menghapus beberapa *edge* yang menghubungkan *node* dengan klaster yang berbeda. Pertama-tama, penulis mengirimkan nilai *edge* yang menghubungkan *node* yang berbeda klaster, kemudian menghapusnya dengan fungsi *delete_edge* yang telah disediakan. Adapun ada kemungkinan data *edge* yang dikirim penulis tidak berurutan, maka penulis harus mengecek kemungkinan adanya *error* dan mencoba menyelesaikan *error* tersebut dengan membalikkan urutan dari data *edge* yang dikirim penulis sebelumnya. *Pseudocode* dari proses tersebut dapat dilihat pada *Pseudocode* 4.16.

```
1   FUNCTION __init__(self,removed_edges):
2       FOR removed_edge IN removed_edges:
3           TRY:
4               gstream.delete_edge(edges[
                (removed_edge[0], removed_edge[1])])
5           EXCEPT KeyError:
6               gstream.delete_edge(edges[
                (removed_edge[1], removed_edge[0])])
7   ENDFUNCTION
8
```

Pseudocode* 4.16 Kode program untuk menghilangkan *edge* diantara *node* yang berbeda *cluster

4.2.5 Perhitungan nilai *adjusted rand index* (ARI)

Penulis menggunakan dua buah fungsi untuk menghitung nilai ARI. Fungsi pertama adalah *get_evaluated*, yaitu fungsi

yang bertujuan untuk membagi log menjadi beberapa bagian untuk mempermudah perhitungan dari ARI serta fungis *get_adjusted_rand* yang berfungsi untuk menghitung nilai ARI-nya. Pada fungsi *get_evaluated*, penulis membaca tiap *line* dari log kemudian membagi-bagi line tersebut dan menyimpannya pada variabel *evaluation_labels*. Variabel *evaluation_labels* inilah yang akan digunakan untuk perhitungan ARI. *Pseudocode*nya dapat dilihat pada *Pseudocode* 4.17.

```
1   FUNCTION get_evaluated(evaluated_file):
2       OPEN evaluated_file AS ef
3       evaluations <- READLINES(ef)
4       evaluation_labels                          <-
        [evaluation.SPLIT(';')[0]  FOR  evaluation  IN
        evaluations]
5       RETURN evaluation_labels
6   ENDFUNCTION
```

***Pseudocode* 4.17 Kode program untuk mengevaluasi file log**

Pada fungsi *get_adjusted_rand*, terdapat tiga buah variabel, yaitu *standard_labels*, *prediction_labels* dan *adjusted_rand_index*. Variabel *standard_labels* merupakan variabel yang berfungsi untuk menyimpan nilai variabel *evaluation_labels* pada log yang telah terklaster, sedangkan variabel *prediction_labels* berfungsi untuk menyimpan nilai variabel *evaluation_labels* pada log yang merupakan hasil yang diharapkan dari proses klaster. Kemudian variabel *adjusted_rand_index* merupakan nilai dari ARI antara *standard_labels* dan *prediction_labels*. Adapun perhitungan dari nilai ARI menggunakan fungsi yang telah disediakan oleh *sklearn*, yaitu fungsi *adjusted_rand_score* pada *Class metric*. *Pseudocode* dari proses tersebut dapat dilihat pada *Pseudocode* 4.18.

```
1   FUNCTION          get_adjusted_rand(standard_file,
    prediction_file):
2       standard_labels                              <-
        get_evaluated(standard_file)
3       prediction_labels                            <-
        get_evaluated(prediction_file)
4       adjusted_rand_index                          <-
        metric.adjusted_rand_score(standard_labels,
        prediction_labels)
5       RETURN adjusted_rand_index
6   ENDFUNCTION
```

Pseudocode* 4.18 Kode program untuk menghitung nilai *adjusted rand index

BAB V
HASIL UJI COBA DAN EVALUASI

Bab ini berisi penjelasan mengenai skenario uji coba dan evaluasi pada klasterisasi graf untuk data log autentikasi dalam dataset log. 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 klasterisasi graf untuk log autentikas dengan algoritma *molecular complex detection* 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) i5-2.6 GHz
	Memori	8 GB DDR3
Perangkat Lunak	Sistem Operasi	Windows 8.1
	Perangkat Pengembang	JetBrains PyCharm 2.7.4 dan Gephi 0.9.1

5.2 Data Pengujian

Subbab ini menjelaskan mengenai data yang digunakan pada uji coba. Seperti yang telah dijelaskan sebelumnya, terdapat dua buah dataset yang terbagi menjadi 61 data log *Hofstede2014* dan 32 data log *SecRepo*. Data tersebut kemudian diolah pada tahap *preprocessing* sehingga menghasilkan data log yang

memiliki kata-kata yang hanya diperlukan dari log sebelumnya. Kemudian, kata-kata yang penting hasil dari proses *preprocessing* akan digunakan untuk mencari nilai *string similarity* dari log tersebut dan nilai *string similarity* ini akan digunakan sebagai pembanding nilai batas dari pembuatan graf. Pada pembuatan graf, akan terdapat *node* dan *edge* yang dibuat, yang kemudian akan digunakan sebagai data masukan untuk klasterisasi dengan algoritma *molecular complex detection*. Selain digunakan sebagai data masukan untuk klasterisasi, *node* dan *edge* tersebut juga digunakan sebagai data masukan untuk visualisasi pada *Gephi*. Adapun hasil dari klasterisasi dengan algoritma *molecular complex detection* adalah sebuah klaster yang berisi sebuah log berada pada klaster apa. Proses-proses ini disajikan dalam diagram alir pada Gambar 5.10**Error! Reference source not found.**.

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 performa pada masing-masing skenario. Skenario pengujian dibagi menjadi dua bagian, yaitu uji fungsionalitas dan uji performa.

5.3.1 Skenario Uji Fungsionalitas

Pengujian ini didasarkan pada fungsionalitas yang disajikan sistem. Pengujian dilakukan dengan menjalankan fungsi-fungsi yang terdapat pada sistem. Adapun pengujian-pengujian gunsi yang akan diuji coba yaitu :

- Pengujian fungsi pada proses *preprocessing* dapat berjalan
- Pengujian fungsi pada proses pembuatan graf dapat berjalan

- Pengujian fungsi-fungsi pada algoritma *molecular complex detection* dapat berjalan
- Pengujian fungsi untuk menghitung *adjusted rand index*
- Pengujian fungsi untuk melihat *running time* dari fungsi-fungsi sistem
- Pengujian untuk visualisasi pada Gephi

5.3.1.1 Uji Coba Fungsi pada *Preprocessing*

Uji coba fungsi *preprocessing* akan dilakukan dengan menjalankan fungsi *preprocessing*. Cara pengujiannya yaitu dengan diberikan file log yang akan di*preprocessing*, contoh dari log tersebut terdapat pada Gambar 5.1. Log tersebut akan diinput kedalam fungsi *preprocessing* kemudian fungsi tersebut dijalankan dan output dari fungsi tersebut akan disimpan kedalam sebuah file. Pengujian dinyatakan berhasil jika hasil dari fungsi *preprocessing* sesuai dengan format dari fungsi *preprocessing*.

```
1 Dec  3 00:02:02 161.166.232.16 sshd[31292]: Accepted publickey for XXXXX from
  161.166.232.12 port 36382 ssh2
2 Dec  3 00:02:02 161.166.232.16 sshd[31292]: pam_unix(sshd:session): session closed
  for user XXXXX
```

Gambar 5.1 Log yang digunakan untuk pengujian fungsi *preprocessing*

Setelah fungsi *preprocessing* dijalankan, terdapat *output* seperti pada Gambar 5.2. *Output* dari fungsi *preprocessing* sudah sesuai dengan format yang diatur pada fungsi *preprocessing*, maka dapat dikatakan bahwa pengujian fungsional fungsi *preprocessing* sudah berhasil.

```
2 [1, {'status': '', 'end': 'Dec 3 02:53:48', 'start': 'Dec 3 00:05:30', 'attack': 1,
  'member': [41, 53, 96, 109, 116, 139, 161, 166, 178, 190, 195, 197, 198, 203, 206,
  226, 229, 242, 272, 277, 287, 291, 308, 311, 326, 347, 385, 404, 413, 455, 462, 465,
  480, 488, 489, 491, 506, 522, 526, 539, 575, 583, 584, 585, 593, 594, 601, 603, 605,
  661, 671, 672, 679, 700, 720, 739, 746, 755, 767, 768, 769, 770, 782, 787, 794, 803,
  813, 820, 824, 851, 852, 872, 912, 958, 965, 966, 975, 996, 1003, 1005, 1008, 1026,
  1045, 1047, 1075, 1088, 1137, 1141, 1162, 1175, 1181, 1186, 1208, 1221, 1231, 1244,
  1246, 1250, 1261, 1271, 1286, 1288, 1299, 1310, 1333, 1433, 1438, 1459, 1485],
  'cluster': 1, 'preprocessed_event': 'fatal read socket failed connection reset by
  peer', 'frequency': 109, 'event': 'fatal: read from socket failed: connection reset by
  peer [preauth]'}]
3 [2, {'status': '', 'end': 'Dec 3 23:17:01', 'start': 'Dec 3 00:17:01', 'attack': 0,
  'member': [126, 591, 1070, 1583, 1627, 1632, 1639, 1641, 1644, 1649, 1651, 1663, 1668,
  1692, 1717, 1722, 1877, 1902, 1962, 1981, 1992, 2004, 2013, 2015, 2028], 'cluster': 2,
  'preprocessed_event': 'pam unix cron session session opened user root by uid',
  'frequency': 25, 'event': 'pam_unix(cron:session): session opened for user root by (
  uid=0)'}]
```


Gambar 5.2 Hasil data log setelah fungsi *preprocessing* dijalankan

5.3.1.2 Uji Coba Fungsi pada Pembuatan Graf

Uji coba pembuatan graf, akan deilakukan dengan menjalankan fungsi pembuatan graf, yaitu fungsi *CreateGraph*. Cara pengujiannya yaitu dengan menjalankan Fungsi *CreateGraph* pada log yang telah melalui *preprocessing*. Uji coba akan dikatakan berhasil jika *node* dan *edge* beserta *weight* dari *edge* bisa dibuat. Adapun *node* dan *edges* yang berhasil dibuat ditampilkan di *Data Laboratory* yang merupakan sebuah fitur untuk melihat *node* dan *edge* dari graf pada *Gephi*.

Penulis menggunakan log hasil *preprocessing* pada Gambar 5.2. Kemudian penulis akan menjalankan Fungsi *CreateGraph* serta menampilkan graf tersebut pada *Gephi*. Pada Gambar 5.3, terlihat bahwa terdapat *node* yang berhasil dibuat. Terdapat beberapa atribut pada pembuatan *node* tersebut, yaitu *id*, *label*, *cluster*, *frequency* dan *attack*. Atribut *id* menunjukkan *id* dari *node* yang telah dibuat, *label* menunjukkan label dari *node*, *cluster* menunjukkan *node* tersebut berada pada klaster apa, *frequency* menunjukkan banyak *node* yang sama pada log dan

attack mengartikan *node* tersebut merupakan sebuah serangan atau bukan.

Id	Label	...	cluster	preprocessed_event	frequency	attack
0	received disconnect bye bye		6		1575	1
1	fatal read socket failed connection reset by peer		2		109	1
2	pam unix cron session session opened user root by uid		3		25	0
3	pam unix cron session session closed user root		3		25	0

Gambar 5.3 *Node* yang berhasil dibuat pada proses pembuatan graf

Pada Gambar 5.4 terlihat bahwa terdapat *edge* yang berhasil dibuat. Terdapat beberapa atribut yang dibuat pada oleh fungsi *CreateGraph*, yaitu *source, target, id* dan *weight*. Atribut *source* merupakan *node* asal dari *edge* tersebut, isi dari atribut *source* adalah id dari *node* dan label dari *node* awal. Atribut *target* merupakan *node* tujuan dari *edge* tersebut, isinya sama dengan atribut *source*, yaitu id dan label dari *node* tujuan. Atribut *id* berisi id dari *edge* yang menghubungkan *source* dan *target node*. Atribut *weight* berisi *weight* dari *edge* tersebut. Karena fungsi *CreateGraph* berhasil membuat *node* dan *edge* maka dapat dikatakan bahwa uji coba dari pembuatan graf sudah berhasil.

Source	Target	...	Id	...	...	Weight
22 - invalid user git	57 - invalid user oracle	...	352			0.6
29 - input userauth request invalid user ibm...	50 - input userauth request invalid user zab...	...	472			0.8
23 - input userauth request invalid user git	60 - input userauth request invalid user cron	...	372			0.8
6 - invalid user admin	28 - invalid user ibmuser	...	19			0.6

Gambar 5.4 *Edge* yang berhasil dibuat pada proses pembuatan graf

5.3.1.3 Uji Coba Algoritma MCODE

Uji coba algoritma *molecular complex detection* dilakukan dengan menjalankan algoritma *molecular complex detection* pada log yang telah melalui tahap *preprocessing* dan pembuatan graf. Log tersebut akan diklaster oleh algoritma

molecular complex detection kemudian hasil dari pengklasteran tersebut akan ditampilkan ke sebuah file. Acuan keberhasilan dari algortima *molecular complex detection* dapat dilihat pada file log yang telah dikelompokkan dalam klaster-klaster. Pada Gambar 5.5, terlihat bahwa log masih acak dan belum masuk pada sebuah klaster.

```
17; session opened; Dec  7 00:17:01 ip-172-31-27-153 CRON[29430]:
pam_unix(cron:session): session opened for user root by (uid=0)
17; session opened; Dec  7 00:17:01 ip-172-31-27-153 CRON[29430]:
pam_unix(cron:session): session closed for user root
9; invalid user; Dec  7 00:20:45 ip-172-31-27-153 sshd[29444]:
Invalid user admin from 115.29.39.238
9; invalid user; Dec  7 00:20:45 ip-172-31-27-153 sshd[29444]:
input_userauth_request: invalid user admin [preauth]
```

Gambar 5.5 Log yang belum terklaster

Hasil dari log pada Gambar 5.5 setelah algoritma *molecular complex detection* dijalankan dapat dilihat pada Gambar 5.6. Terlihat pada baris 1 sampai 4 pada Gambar 5.5 ada yang berbeda, hal tersebut menandakan bahwa log pada Gambar 5.5 belum terklaster. Sedangkan, pada Gambar 5.6 terlihat pada baris 2 sampai 4 log tersebut sama dan pada baris 1 mengartikan nomor klaster. Maka, dapat dikatakan uji coba pada algoritma *molecular complex detection* sudah berhasil.

```
Cluster #0
Dec  7 00:20:45 ip-172-31-27-153 sshd[29444]: Invalid user admin from
115.29.39.238
Dec  7 02:24:40 ip-172-31-27-153 sshd[29535]: Invalid user admin from
122.225.103.121
Dec  7 03:49:45 ip-172-31-27-153 sshd[29634]: Invalid user admin from
183.82.1.229
```

Gambar 5.6 Log yang sudah terklaster

5.3.1.4 Uji Coba Fungsi Perhitungan ARI

Uji coba fungsi perhitungan *adjusted rand index* dilakukan dengan membandingkan dua buah log, yaitu log *groundtruth* yang merupakan hasil yang diharapkan setelah proses klasterisasi dan log hasil dari klasterisasi. Kedua file tersebut akan dibandingkan, semakin sama kedua log tersebut, maka nilai *adjusted rand index* akan semakin mendekati 1, jika semakin berbeda, maka nilai *adjusted rand index* kedua file tersebut akan semakin mendekati 0. Acuan yang digunakan penulis untuk uji coba perhitungan nilai *adjusted rand index* yaitu berhasilnya mendapatkan nilai *adjusted rand index* dari dua buah log. Terlihat pada Gambar 5.7 bahwa nilai *adjusted rand index* berhasil didapatkan, maka dapat dikatakan bahwa uji coba fungsi perhitungan nilai *adjusted rand index* sudah berhasil.

```
ARI           : 0.999209104681
Runtime       : 1.30000019073 seconds

Process finished with exit code 0
```

Gambar 5.7 Nilai *adjusted rand index* pada log

5.3.1.5 Uji Coba Fungsi Perhitungan *Running Time*

Perhitungan *running time* dilakukan dengan menggunakan *Library Profiling* pada Python. Program yang dijalankan untuk melihat statistiknya yaitu program untuk *preprocess* log, pembuatan graf, klasterisasi, perhitungan ARI serta visualisasi pada graf. Adapun *Library Profiling* yang digunakan penulis adalah *cProfile*. Statistik dari penjalanan program tersebut dengan *cProfile* dapat dilihat pada Gambar 5.8.

```
      6692454 function calls (6685117 primitive calls) in 19.127 seconds

Ordered by: cumulative time, file name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   19.127   19.127 pygraphc:12(main)
     1    0.000    0.000   11.595   11.595 pygraphc:90(graph_streaming)
     1    0.084    0.084   11.549   11.549 GraphStreaming.py:50(gephi_streaming)
  3624    0.034    0.000   11.227    0.003 client.py:54(flush)
  3625    0.021    0.000   11.201    0.003 client.py:96(_send)
  3492    0.038    0.000   11.052    0.003 client.py:76(add_edge)
  3625    0.023    0.000   10.523    0.003 urllib2.py:131(urlopen)
  3625    0.065    0.000   10.499    0.003 urllib2.py:411(open)
  3625    0.035    0.000    9.966    0.003 urllib2.py:439(_open)
  7250    0.020    0.000    9.929    0.001 urllib2.py:399(_call_chain)
  3625    0.024    0.000    9.905    0.003 urllib2.py:1227(http_open)
  3625    0.166    0.000    9.881    0.003 urllib2.py:1152(do_open)
  3625    0.059    0.000    6.530    0.002 httplib.py:1099(getresponse)
  3625    0.084    0.000    6.354    0.002 httplib.py:446(begin)
 25375    0.188    0.000    5.650    0.000 socket.py:410(readline)
  3625    0.073    0.000    5.529    0.002 httplib.py:407(_read_status)
 10874    5.393    0.000    5.393    0.000 {method 'recv' of '_socket.socket' objects}
     1    0.674    0.674    3.329    3.329 Preprocess.py:73(do_preprocess)
  3625    0.012    0.000    3.031    0.001 httplib.py:1055(request)
```

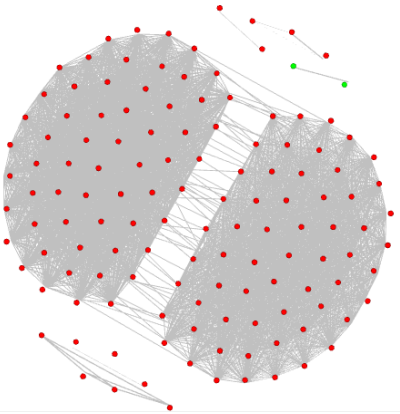
Gambar 5.8 Statistik *running* program pygraphc dengan *cProfile*

Dari data statistik diatas, dapat dilihat bahwa program *pygraphc* berjalan selama 19,127 detik, dimana 11,595 detik terdapat fungsi yang berjalan, yaitu fungsi *graph_streaming*. Data statistik ini sudah mencangkup subfungsi terdalam dari sebuah fungsi, sehingga dapat dikatakan bahwa statistik dari *cProfile* sudah lumayan lengkap.

5.3.1.6 Uji Coba Visualisasi pada Gephi

Graf yang berhasil dibuat selanjutnya akan divisualisasikan di Gephi. Adapun visualisasi *node* nantinya berupa dua buah warna, yaitu *node* dengan warna hijau dan *node* dengan warna merah. *Node* dengan warna hijau merupakan *node* yang merepresentasikan log yang bukan serangan, sedangkan *node* yang berwarna merah merepresentasikan log yang merupakan serangan. Visualisasi graf pada Gephi dapat dilihat pada

Gambar 5.9.



Gambar 5.9 Visualisasi graf yang berhasil dibuat pada Gephi

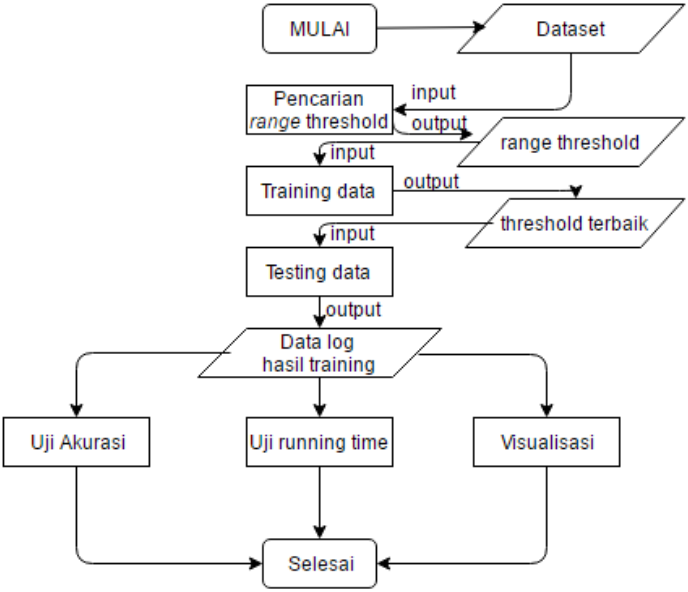
5.3.2 Skenario Uji Performa

Pengujian ini untuk menguji performa dari sistem. Pengujian yang akan dilakukan yaitu sebagai berikut :

- Mencari range nilai *threshold* yang akan digunakan untuk proses *training* dan *testing* data
- *Training* dan *testing* data
- Menghitung nilai akurasi dari algoritma *molecular complex detection*
- Mengevaluasi nilai dari *running time* tiap fungsi yang ada
- Menguji visualisasi dengan sebuah variabel

Adapun untuk mencari *range threshold*, penulis akan menguji seluruh *threshold* yang ada dan memilih beberapa *threshold* untuk digunakan pada proses *training* dan *testing* data, untuk menghitung nilai akurasi dari algoritma *molecular complex detection*, performa yang diukur adalah *running time* dan nilai akurasinya. Untuk pengujian *running time*, fungsi-fungsi inti yang

dilihat adalah fungsi untuk proses *preprocessing*, pembuatan graf dan proses klasterisasi. Kemudian untuk menguji visualisasi, akan dilihat visualisasi dari graf jika digunakan sebuah variabel. Adapun variabel yang digunakan yaitu jumlah baris dari log. Tahap-tahap uji coba performa sistem dapat dilihat pada Gambar 5.10.



Gambar 5.10 Diagram alir pengujian performa sistem

5.3.2.1 Uji Coba Pencarian *Range* dari *Threshold*

Uji coba pencarian *range* dari *threshold* yang ada dilakukan untuk mencari nilai *range threshold string similarity* dan *threshold weight* terbaik yang nantinya digunakan untuk *training* data algoritma *molecular complex detection*. Terdapat beberapa skenario untuk mencari nilai dari *threhold* tersebut. Tujuan dari skenario-skenario tersebut adalah untuk mencari

range dari *threshold* yang nantinya digunakan. Adapun untuk pencarian *range threshold* tersebut, digunakan dataset *Hofstede2014*, karena dataset tersebut mempunyai baris yang lebih sedikit daripada dataset *SecRepo* sehingga dapat menghemat waktu untuk pencarian *range*. Setelah mendapatkan *range* dari *threshold* tersebut, maka data *range* tersebut dapat digunakan untuk *training* data untuk mendapatkan nilai terbaik dari *threshold-threshold* yang ada. Kemudian, nilai terbaik dari *threshold* tersebut akan digunakan untuk *testing* data. Hasil dari uji coba pencarian *range* dari *threshold* yang ada dapat dilihat pada Tabel 5.2. Adapun *threshold string similarity* yang digunakan adalah 0.0, karena semua kata yang ada akan digunakan sebagai pertimbangan pembuatan graf.

Tabel 5.2 Hasil perhitungan akurasi algoritma dengan *threshold* MCODE 0.0 sampai 1.0

Threshold MCODE	Akurasi
0.0	88.74%
0.1	81.22%
0.2	73.98%
0.3	68.11%
0.4	61.03%
0.5	52.58%
0.6	40.65%
0.7	31.81%
0.8	16.94%
0.9	4.12%
1.0	0.4%

Berdasarkan hasil perhitungan akurasi algoritma pada *string similarity* 0.0, nilai *threshold* algoritma *molecular complex detection* yang mempunyai nilai diatas **50%** adalah **0.0** sampai dengan **0.5**. Sedangkan nilai *threshold* algoritma *molecular complex detection* yang mempunyai nilai dibawah **50%** adalah **0.6** sampai dengan **1.0.**

Hasil perhitungan akurasi dengan *threshold* **0.6** sampai **1.**0 pada algoritma *molecular complex detection* sangat jauh dari harapan, yaitu dibawah **50%**. Hal tersebut dikarenakan pada *threshold* tersebut, algoritma *molecular complex detection* tidak akan bekerja secara maksimal dalam melakukan klasterisasi log yang ada. Adapun proses klasterisasi tidak maksimal dikarenakan pada *threshold* tersebut, data-data hasil *preprocessing* dan pembuatan graf akan diolah dengan *threshold* yang sangat lebar, sehingga data-data berbeda klaster kemungkinan akan menjadi sebuah klaster dan data-data yang seharusnya pada klaster yang sama kemungkinan akan dijadikan klaster yang berbeda.

Hal tersebut juga berlaku pada *threshold string similarity*. Semakin besar *threshold* yang digunakan, maka data hasil *preprocessing* yang diolah menjadi *node* dan *edge* pada proses pembuatan graf akan semakin sedikit. Hal tersebut dikarenakan nilai perbandingan antara dua *string* yang sama akan semakin besar jika nilai *threshold* yang digunakan semakin besar. Oleh karena itu, penulis menggunakan *threshold* **0.0** sampai **0.5** untuk *string similarity* dan *threshold* algoritma *molecular complex detection* pada proses *training* data.

5.3.2.2 *Training* dan *Testing* data

Training data dilakukan dengan menguji dataset yang ada dengan *threshold* yang didapatkan sebelumnya untuk mencari nilai *threshold* terbaik yang nantinya akan digunakan untuk proses *testing* data. Hasil akurasi dari *testing* data ini nantinya akan digunakan sebagai hasil akhir untuk akurasi dari algoritma *molecular complex detection*. Adapun dataset yang digunakan untuk *training* data yaitu 41 buah data log dari 61 dataset *Hofstede2014* dan 20 buah data log dari 32 buah data dari dataset *SecRepo*. Adapun jumlah data log pada saat *training* data lebih banyak daripada saat *testing* data diharapkan dapat membuat *threshold* yang terpilih saat proses *training* data menjadi sesuai dengan yang diharapkan. Hasil training data pada dataset

Hofstede2014 dapat dilihat pada Tabel 5.3. Sedangkan hasil *training* data dengan dataset *SecRepo* dapat dilihat pada Tabel 5.4.

Tabel 5.3 Hasil *training* data pada dataset *Hofstede2014* (merah untuk akurasi terburuk dan biru untuk akurasi terbaik)

String Similarity	Threshold MCODE	Akurasi
0.0	0.0	88.74%
0.0	0.1	81.22%
0.0	0.2	73.98%
0.0	0.3	68.11%
0.0	0.4	61.03%
0.0	0.5	52.58%
0.1	0.0	87.90%
0.1	0.1	88.14%
0.1	0.2	78.46%
0.1	0.3	68.38%
0.1	0.4	62.99%
0.1	0.5	58.28%
0.2	0.0	88.5%
0.2	0.1	90.12%
0.2	0.2	85.04%
0.2	0.3	85.59%
0.2	0.4	83.36%
0.2	0.5	71.34%
0.3	0.0	88.5%
0.3	0.1	81.31%
0.3	0.2	78.63%
0.3	0.3	79.47%
0.3	0.4	78.27%
0.3	0.5	75.57%
0.4	0.0	88.54%
0.4	0.1	81.07%

0.4	**0.2**	**80.5%**
0.4	**0.3**	**79.47%**
0.4	**0.4**	**76.94%**
0.4	**0.5**	**75.54%**
0.5	**0.0**	**89.57%**
0.5	**0.1**	**80.27%**
0.5	**0.2**	**80.83%**
0.5	**0.3**	**79.11%**
0.5	**0.4**	**80.59%**
0.5	**0.5**	**78.39%**

Dari hasil yang didapatkan diatas, diketahui bahwa akurasi yang terbaik adalah **90.12%**. Nilai akurasi tersebut didapatkan dengan mengklaster log autentikasi menggunakan **0.2** sebagai *threshold* untuk *string similarity*nya dan **0.1** untuk *threshold* algoritma *molecular complex detection*. Maka, dari proses *training* data dari dataset *Hofstede2014* didapatkan **0.2** sebagai *threshold string similarity* dan **0.1** sebagai nilai *threshold* algoritma *molecular complex detection*. Kemudian nilai-nilai *threshold* ini akan digunakan untuk proses *testing* data pada dataset *Hofstede2014*.

Tabel 5.4 Hasil *training* data pada dataset *SecRepo (*merah untuk akurasi terburuk dan biru untuk akurasi terbaik*)*

String Similarity	Threshold MCODE	Akurasi
0.0	**0.0**	**87.77%**
0.0	**0.1**	**83.32%**
0.0	**0.2**	**89.97%**
0.0	**0.3**	**83.78%**
0.0	**0.4**	**76.11%**
0.0	**0.5**	**72.43%**
0.1	**0.0**	**88.47%**
0.1	**0.1**	**84.63%**
0.1	**0.2**	**94.51%**

0.1	**0.3**	**89.09%**
0.1	**0.4**	**81.42%**
0.1	**0.5**	**74.44%**
0.2	**0.0**	**87.75%**
0.2	**0.1**	**85.16%**
0.2	**0.2**	**89.36%**
0.2	**0.3**	**97.15%**
0.2	**0.4**	**97.09%**
0.2	**0.5**	**97.08%**
0.3	**0.0**	**87.75%**
0.3	**0.1**	**86.2%**
0.3	**0.2**	**90.44%**
0.3	**0.3**	**98.22%**
0.3	**0.4**	**98.22%**
0.3	**0.5**	**98.22%**
0.4	**0.0**	**88.44%**
0.4	**0.1**	**82.08%**
0.4	**0.2**	**82.12%**
0.4	**0.3**	**82.12%**
0.4	**0.4**	**82.12%**
0.4	**0.5**	**82.12%**
0.5	**0.0**	**87.75%**
0.5	**0.1**	**80.81%**
0.5	**0.2**	**80.81%**
0.5	**0.3**	**80.81%**
0.5	**0.4**	**80.81%**
0.5	**0.5**	**80.81%**

Dari hasil yang didapatkan diatas, diketahui bahwa akurasi yang terbaik adalah **98.22%**. Nilai akurasi tersebut didapatkan dengan mengklaster log autentikasi menggunakan **0.3** sebagai *threshold* untuk *string similarity*nya dan **0.3** untuk *threshold* algoritma *molecular complex detection*. Maka, dari proses *training* data dari dataset *SecRepo* didapatkan **0.3** sebagai

threshold string similarity dan **0.3** sebagai nilai *threshold* algoritma *molecular complex detection*. Kemudian nilai-nilai *threshold* ini akan digunakan untuk proses *testing* data pada dataset *SecRepo*.

Tahap selanjutnya dari uji coba ini adalah *testing* data. *Testing* data pada dataset *Hofstede2014* dilakukan dengan menggunakan nilai **0.2** sebagai nilai *threshold string similarity* dan **0.1** sebagai nilai *threshold* algoritma *molecular complex detection*. Sedangkan untuk dataset *SecRepo*, digunakan nilai **0.3** sebagai nilai *threshold string similarity* dan **0.3** sebagai nilai *threshold* algoritma *molecular complex detection*. Hasil dari *testing* data pada dataset *Hofstede2014* dapat dilihat pada Tabel 5.5. Sedangkan untuk hasil *testing* data pada dataset *SecRepo* dapat dilihat pada Tabel 5.6.

Tabel 5.5 Hasil *testing* data pada dataset *Hofstede2014 (*merah untuk akurasi terburuk dan biru untuk akurasi terbaik*)*

File	Jumlah baris	Jumlah Edge	Akurasi
Dec 1.log	1424	15	**92.17%**
Dec 10.log	3920	86	**89.54%**
Dec 11.log	9467	29	**99.53%**
Dec 2.log	2161	15	**75.83%**
Dec 22.log	1655	32	**89.26%**
Dec 27.log	1549	22	**82.94%**
Dec 30.log	2180	36	**85.29%**
Dec 4.log	7638	47	**84.69%**
Dec 8.log	8342	49	**99.74%**
Nov 13.log	672	1	**71.33%**
Nov 16.log	2414	18	**84.24%**
Nov.17.log	1315	19	**96.60%**
Nov 2.log	2799	33	**96.96%**
Nov 20.log	3261	33	**95.57%**
Nov 25.log	3804	40	**94.29%**
Nov 3.log	557	18	**97.84%**
Nov 5.log	670	5	**100%**

Nov 7.log	2279	9	**100%**
Nov 8.log	3162	10	**99.99%**
Nov 9.log	676	1	**71.33%**
Rata-rata			**90.36%**

Pada Tabel 5.5, dapat dilihat bahwa nilai rata-rata akurasi dari *testing* data pada dataset *Hofstede2014* dapat dikatakan sudah bagus, yaitu **90.36%**. Adapun perhitungan nilai akurasi terbaik diperoleh dengan menggunakan log "Nov 5.log" dan "Nov 7.log" sebagai log yang digunakan. Sedangkan perhitungan nilai akurasi terburuk mencapai **71.33%** jika menggunakan log "Nov 9.log" dan "Nov 13.log". Hal tersebut diakibatkan karena *edge* yang dimiliki masing-masing log tersebut sangatlah sedikit, yaitu hanya berjumlah 1.

Tabel 5.6 Hasil *testing* data pada dataset *SecRepo* (merah untuk akurasi terburuk dan biru untuk akurasi terbaik)

File	Jumlah baris	Jumlah Edge	Akurasi
dec-11.log	3310	1617	**99.97%**
dec-12.log	2500	7953	**99.96%**
dec-14.log	4256	1940	**99.98%**
dec-18.log	860	374	**99.60%**
dec-19.log	2591	408	**99.96%**
dec-2.log	17598	23720	**99.99%**
dec-24.log	2154	4243	**99.94%**
dec-25.log	2747	882	**99.96%**
dec-4.log	1337	8665	**99.82%**
dec-7.log	482	530	**98%**
dec-9.log	1385	602	**99.85%**
nov-30.log	688	4496	**99.62%**
Rata-rata			**99.72%**

Pada Tabel 5.6, dapat dilihat bahwa nilai rata-rata akurasi dari *testing* data pada dataset *SecRepo* dapat dikatakan sudah bagus, yaitu **99.72%**. Adapun perhitungan nilai akurasi terbaik

diperoleh dengan menggunakan log "dec-2.log" sebagai log yang digunakan. Namun, secara keseluruhan, nilai akurasi dari *testing* data pada dataset ini sudah sangatlah bagus, dikarenakan nilai akurasi dari semua log bernilai diatas **98%**, bahkan nilai rata-rata akurasinya mencapai **99%**. Faktor utama yang membuat dataset ini mempunyai nilai akurasi yang sangat tinggi karena sedikitnya variasi dalam lognya dan banyaknya data log yang sama.

5.3.2.3 Uji Coba Akurasi Algoritma MCODE

Uji coba akurasi algoritma *molecular complex detection* adalah uji coba performa yang dilakukan untuk menghitung nilai *adjusted rand index* dari log setelah terklaster dan log yang merupakan log yang sudah benar (*groundtruth file*). Skenario dalam uji coba perhitungan nilai akurasi algoritma *molecular complex detection* dilakukan dengan menggunakan enam buah *threshold* untuk algoritma *molecular complex detection*. Nilai *threshold* tersebut akan dipilih berdasarkan hasil dari uji coba pada subbab 5.3.2.1 untuk pengujian ini, yaitu **0.0** sampai dengan **0.5**. Selain itu, nilai *string similarity* yang digunakan untuk uji coba ini adalah **0.2**, karena merupakan *string similarity* terbaik dari hasil proses *training* data.

Threshold yang digunakan pada uji coba ini yaitu **0.0** sampai dengan **0.5**. Hasil dari perhitungan akurasi pada file "*Nov 7.log"* dapat dilihat pada Tabel 5.7.

Tabel 5.7 Hasil perhitungan akurasi pada file "Nov 7.log"

Threshold	Akurasi
0.0	**83.44%**
0.1	**100%**
0.2	**100%**
0.3	**85.77%**
0.4	**85.77%**
0.5	**65.86%**

Berdasarkan hasil perhitungan akurasi pada Tabel 5.7, dapat dilihat bahwa *threshold* **0.1** dan **0.2** mempunyai nilai akurasi yang paling bagus dari ketiga *threshold* yang diuji coba, yaitu **100%**. Sedangkan jika menggunakan *threshold* **0.5** maka nilai akurasi yang didapatkan menjadi sangat rendah, yaitu **65.86%**.

Hasil perhitungan akurasi dengan *threshold* **0.5** menjauhi nilai yang diharapkan. Hal tersebut dikarenakan pada *threshold* **0.5** *node* dan *edge* akan dibandingkan dengan tidak detail. Sehingga, jika ada persamaan sedikit saja antar sebuah *node* maka *node* tersebut akan dijadikan klaster yang sama, walaupun sebenarnya *node* tersebut adalah klaster yang berbeda. Kasus ini dapat dimisalkan dengan baris log "*Connection closed*" dan baris log "*Connection open*". Seharusnya, kedua log tersebut menjadi klaster yang berbeda, namun jika *threshold* yang digunakan **0.5** maka kedua log tersebut dapat dianggap sebagai klaster yang sama.

5.3.2.4 Evaluasi *Running Time*

Evaluasi *running time* merupakan perhitungan nilai *running time* dari fungsi-fungsi yang ada pada program. Adapun fungsi-fungsi yang akan dilihat *running time*nya adalah fungsi pada proses *preprocessing*, pembuatan graf dan proses klasterisasi. Terdapat tiga skenario pada skenario ini. Perbedaan dari skenario-skenario ini adalah pada data log yang digunakan. Log-log yang digunakan akan dipertimbangkan dari jumlah baris dari log tersebut, jumlah *edge* dari log tersebut serta nilai akurasi dari *testing* data log tersebut. Adapun dataset yang digunakan adalah dataset *SecRepo* karena dataset tersebut mempunyai baris log yang banyak serta nilai akurasi yang tinggi pada tahap *testing* data (diatas **90%**). Selain itu, terdapat empat log dari dataset *SecRepo* yang akan digunakan pada uji coba ini, data log yang dipilih nantinya yaitu dua buah data log yang jumlah baris lognya lebih banyak dari jumlah *edge*nya dan dua buah data log yang

mempunyai jumlah *edge* yang lebih banyak dari jumlah barisnya. Agar hasil yang didapatkan menjadi mudah untuk dianalisis, maka penulis menggunakan data log yang mempunyai jumlah baris dan jumlah *edge* diantara **1000** sampai dengan **10.000**. Data log yang dipilih dari log yang digunakan pada proses *training* data adalah berdasarkan selisih terbesar dari jumlah baris dan jumlah *edge* dari log yang bersangkutan. Untuk *string similarity* yang digunakan adalah **0.3** dan *threshold* dari algoritma *molecular complex detection* yang digunakan adalah **0.3**. Penggunaan nilai *string similarity* dan *threshold* tersebut berdasarkan nilai akurasi terbaik dari proses *training* data.

Tabel 5.8 Detail file yang digunakan dalam uji coba perhitungan *running time*

File	Jumlah Baris	Jumlah *Edge*
dec-4.log	1337	8665
dec-12.log	2500	7953
dec-11.log	3310	1617
dec-14.log	4256	1940

Detail dari file-file yang digunakan untuk uji coba ini dapat dililhat pada Tabel 5.8. Penulis menggunakan file pada Tabel 5.8 karena adanya perbedaan jumlah baris dari kecil ke besar dan jumlah *edge* yang tidak bergantung dari jumlah baris pada filenya. Hal tersebut dapat dilihat pada file *"dec-12.log"* dan file *"dec-11.log"*. Pada kedua file tersebut dapat dilihat bahwa jumlah baris dari file *"dec-11.log"* lebih besar dari file *"dec-12.log"*. Namun, jumlah dari *edge* yang dimiliki file *"dec-11.log"* lebih sedikit dibandingkan dengan jumlah *edge* yang dimiliki oleh file *"dec-12.log"*. Hal tersebut dapat diakibatkan karena isi dari log *"dec-12.log"* lebih bervariasi dari log lain yang digunakan pada proses uji coba ini.

Tabel 5.9 Hasil perhitungan *running time*

File	*Running Time (s)*			
	Preprocess	Pembuatan Graf	MCODE	Total
dec-4.log	1.528	0.899	8.89	11.317
dec-12.log	1.899	0.911	8.664	11.474
dec-11.log	1.994	0.484	1.363	3.841
dec-14.log	2.984	0.153	1.38	4.515

Hasil perhitungan *running time* dari file-file yang digunakan pada Tabel 5.8 dapat dilihat pada Tabel 5.9. Pada hasil tersebut, dapat dilihat bahwa waktu yang paling lama untuk *preprocessing* digunakan oleh file *"dec-14.log"*, sedangkan waktu yang paling lama untuk proses pembuatan graf dan proses pada algoritma *molecular complex detection* digunakan oleh file *"dec-12.log"*. Namun, pada proses pembuatan graf, perbedaan waktu yang dibutuhkan file-file tersebut tidak berbeda terlalu jauh yaitu masih kurang dari satu detik. Tidak seperti pada proses lainnya yang berbeda lebih dari satu detik.

Dari Tabel 5.9 dapat dilihat bahwa untuk proses *preprocessing*, log *"dec-14.log"* mempunyai waktu yang paling lama. Hal tersebut dikarenakan pada tahap *preprocessing*, baris log akan dibaca perline sehingga, yang mempunyai baris line paling banyak otomatis akan menghabiskan waktu yang paling banyak. Hal tersebut ditunjukkan dengan membandingkan jumlah baris tiap file yang diuji dengan *running* time pada saat *preprocessing*. Terlihat bahwa semakin banyak jumlah dari baris yang dimiliki log file log tersebut, maka semakin lama juga waktu *preprocessing* yang diperlukan. Sedangkan untuk tahap pembuatan graf, waktu yang paling lama dimiliki oleh file log "*dec-12.log*". Unutk klasterisasi dengan algoritma *molecular complex detection*, diketahui bahwa waktu yang paling lama dimiliki oleh file log *"dec-4.log"*. Diketahui dari kedua file log tersebut, selisih waktu yang dibutuhkan dalam tahap pembuatan graf dan klasterisasi tidak terlalu jauh. Hal tersebut diakibatkan

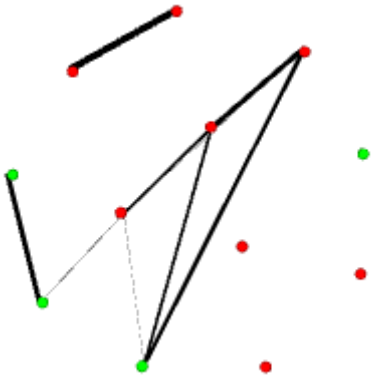
karena *edge* yang dimiliki kedua file tersebut tidak terlalu berbeda jauh. Tidak seperti file log "*dec-11.log*" dan "*dec-14.log*" yang mempunyai perbedaan *edge* yang sangat jauh dibandingkan *"dec-4.log"* maupun "*dec-12.log*". Adapun proses pembuatan graf dan klasterisasi algoritma *molecular complex detection* sangat dipengaruhi oleh jumlah *node* dan *edge* dari file yang digunakan karena pada tahap tersebut, akan terjadi proses pembuatan *edge* dan *node* pada proses pembuatan graf. Sedangkan pada proses klasterisasi, akan terjadi peninjauan ulang *node* dan *edge* pada graf yang telah berhasil dibuat untuk dimasukkan ke sebuah klaster, sehingga tahap-tahap ini akan sangat dipengaruhi oleh jumlah *node* dan *edge*nya.

5.3.2.5 Uji Visualisasi

Uji visualisasi merupakan pengujian visualisasi dengan sebuah variabel dan bertujuan untuk melihat visualisasi dari variabel yang diberikan. Terdapat empat skenario dalam pengujian ini, yaitu dengan menggunakan variabel *edge* dari graf. Pengujian dilakukan dengan mengatur nilai *edge* dari *edge* yang rendah ke tinggi, kemudian akan dilihat visualisasinya. Kedua dataset akan digunakan dalam pengujian ini, namun nilai dari *string similarity* dan *threshold* algoritma *molecular complex detection* yang digunakan merupakan nilai terbaik pada masing-masing dataset. Diantara empat buah skenario yang ada, akan digunakan dua buah dataset *Hofstede2014* untuk dua buah skenario dan dua buah dataset *SecRepo* untuk dua buah skenario. Adapun dataset *Hofstede2014* akan digunakan untuk visualisasi dengan *edge* yang sedikit dan dataset *Secrepo* digunakan untuk visualisasi dengan *edge* yang berjumlah banyak.

5.3.2.5.1 Skenario Uji Coba 1

Skenario uji coba 1 adalah uji coba visualisasi dengan *file "Nov 7.log"* pada dataset *Hofstede2014*. *File* tersebut digunakan karena log tersebut mempunyai *edge* yang berjumlah 9 yang merupakan jumlah yang sedikit dan log tersebut merupakan file log yang mempunyai nilai akurasi tertinggi pada dataset *Hofstede2014*. Visualisasi dari log tersebut dapat dilihat pada Gambar 5.11.



Gambar 5.11 Visualisasi log "Nov 7.log"

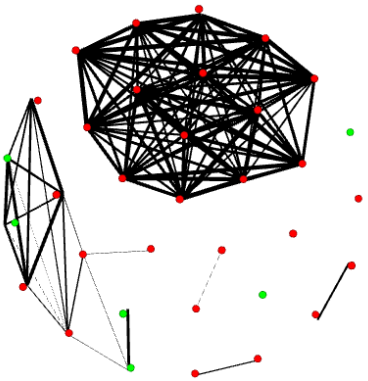
Dari Gambar 5.11 terlihat bahwa terdapat sedikit *node*, sehingga *edge* yang dibuatpun menjadi sedikit. Terlihat juga ada beberapa *node* yang bukan merupakan serangan. Dari visualisasi tersebut terdapat user yang gagal melakukan autentikasi, terdapat juga yang beberapa kali gagal lalu berhasil melakukan autentikasi serta ada user yang langsung berhasil melakukan autentikasi. Dari user yang pernah gagal melakukan autentikasi ada kemungkinan bahwa user tersebut melakukan *brute force* untuk berhasil terautentikasi.

Visualisasi pada Gambar 5.11 kurang bagus dikarenakan kurang terlihatnya hubungan antar *node* dan terdapat banyak *node* yang tidak mempunyai hubungan dengan *node* lainnya. Hal tersebut dikarenakan sedikitnya data log yang ada

sehingga *node* dan *edge* yang dibuatpun menjadi sedikit. Adapun jumlah *node* yang berhasil dibuat pada visualisasi pada Gambar 5.11 adalah 11 buah *node*.

5.3.2.5.2 Skenario Uji Coba 2

Skenario uji coba 2 adalah uji coba visualisasi dengan *file "Dec 17.log"* pada dataset *Hofstede2014*. *File* tersebut digunakan karena log tersebut mempunyai *edge* yang berjumlah 116 yang merupakan jumlah *edge* yang paling banyak pada dataset *Hofstede2014* serta file tersebut mempunyai nilai akurasi yang tinggi, yaitu **97.07%**. Visualisasi dari log tersebut dapat dilihat pada Gambar 5.12.



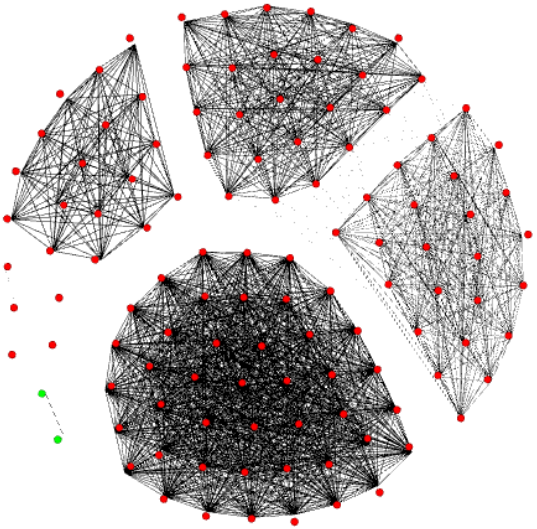
Gambar 5.12 Visualisasi log "Dec 17.log"

Jumlah *node* dari visualisasi tersebut adalah 33. Terlihat banyak *node* serangan yang saling berhubungan dan kemungkinan besar merupakan sebuah serangan terhadap sistem. Adapun perbandingan *node-node* singleton dengan seluruh *node* yang ada sudah lebih baik daripada skenario uji coba 1, yaitu 4 *node* singleton berbanding dengan 29 *node* yang mempunyai pasangan. Namun, perbandingan tersebut masih tergolong kurang

bagus. Sehingga informasi yang diterima dari visualisasi log tersebut masih kurang lengkap.

5.3.2.5.3 Skenario Uji Coba 3

Skenario uji coba 3 adalah uji coba visualisasi dengan *file "dec-11.log"* pada dataset *SecRepo*. *File* tersebut digunakan karena log tersebut mempunyai *edge* yang berjumlah 1617 yang merupakan jumlah yang sedikit pada dataset *Secrepo* dan memiliki nilai akurasi yang tinggi, yaitu **99.97%**. Visualisasi dari log tersebut dapat dilihat pada Gambar 5.13.



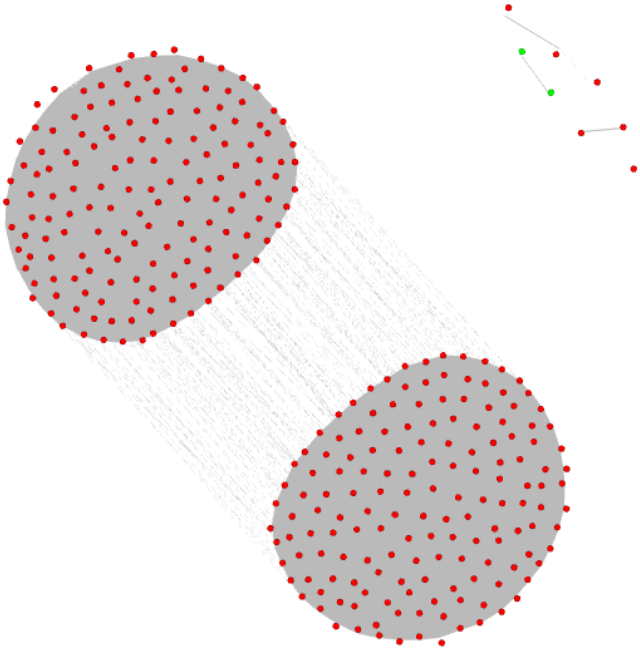
Gambar 5.13 Visualisasi log "dec-11.log"

Pada visualisasi log *"dec-11.log"* terdapat 114 *node* yang dibuat. Terdapat empat buah serangan utama yang terjadi pada log tersebut. Selain itu terdapat serangan yang saling berhubungan. Namun, tetap terdapat user yang berhasil terautentikasi ke sistem.

Dapat dikatakan visualisasi pada Gambar 5.13 sudah bagus, hal tersebut dikarenakan informasi yang diberikan sudah jelas, yaitu terdapat serangan pada sistem serta ada serangan yang saling berhubungan dan ada juga user yang berhasil masuk ke sistem. Dapat dilihat juga jumlah *node* singleton sangat sedikit dibandingkan dengan jumlah *node* yang mempunyai pasangan, yaitu 3 berbanding 111 *node*. Sehingga dapat diketahui bahwa *node* singleton tersebut merupakan suatu hal yang memang ada kemungkinan terjadi pada sistem namun tidak terlalu berpengaruh dikarenakan tidak ada *node* lain yang berhubungan dengan *node* tersebut.

5.3.2.5.4 Skenario Uji Coba 4

Skenario uji coba 4 adalah uji coba visualisasi dengan *file "dec-2.log"* pada dataset *SecRepo*. *File* tersebut digunakan karena log tersebut mempunyai *edge* yang berjumlah 23720 yang merupakan jumlah yang sangat banyak pada dataset *Secrepo* dan merupakan log yang mempunyai nilai akurasi tertinggi pada dataset *SecRepo*. Visualisasi dari log tersebut dapat dilihat pada



Gambar 5.14 Visualisasi log "dec-2.log"

Pada visualisasi log *"dec-2.log"* terdapat 315 *node* yang dibuat. Terdapat empat buah serangan utama yang terjadi pada log tersebut. Selain itu terdapat serangan yang saling berhubungan. Namun, tetap terdapat user yang berhasil terautentikasi ke sistem.

Dapat dikatakan visualisasi pada Gambar 5.14 sudah bagus, hal tersebut dikarenakan informasi yang diberikan sudah jelas, yaitu terdapat serangan pada sistem serta ada serangan yang saling berhubungan dan ada juga user yang berhasil masuk ke sistem. Dapat dilihat juga jumlah *node* singleton sangat sedikit dibandingkan dengan jumlah *node* yang mempunyai pasangan, yaitu 2 berbanding 315 *node*. Sehingga dapat diketahui bahwa *node* singleton tersebut merupakan suatu hal yang memang ada kemungkinan terjadi pada sistem namun tidak terlalu berpengaruh

dikarenakan tidak ada *node* lain yang berhubungan dengan *node* tersebut.

5.3.2.6 Evaluasi Umum Skenario Uji Coba

Berdasarkan skenario uji coba fungsional yang telah dilakukan, diketahui bahwa fungsionalitas dari sistem sudah dapat berjalan dengan baik. Fungsional sistem yang dimaksud adalah fungsi-fungsi seperti *preprocessing*, pembuatan graf, klasterisasi dengan algoritma *molecular complex detection*, perhitungan nilai *adjusted rand index*, perhitungan *running time* serta visualisasi di *Gephi*.

Berdasarkan skenario uji coba performa sistem yang telah dilakukan, dapat diketahui bahwa tingkat ketepatan sistem dalam mengklasterisasi graf sudah sangat bagus, yaitu mencapai **90.36%** untuk dataset *Hofstede2014* dan **99.72%** untuk dataset *SecRepo*. Adapun *threshold* yang digunakan untuk mendapatkan hasil tersebut yaitu **0.2** untuk *threshold string similarity* dan **0.1** untuk *threshold* MCODE pada dataset *Hofstede2014* dan **0.3** untuk *threshold string similarity* serta **0.3** untuk *threshold* MCODE pada dataset *SecRepo*. *Threshold-threshold* tersebut digunakan karena pada proses *training* data, didapatkan *threshold* tersebut mempunyai nilai akurasi tertinggi dibandingkan dengan *threshold* yang lain. Namun, tidak semua *threshold* yang ada digunakan pada proses *training* data, melainkan hanya pada *range* **0.0** sampai dengan **0.5**, hal tersebut dikarenakan *threshold* diatas **0.5** mempunyai akurasi yang buruk dan akan algoritma *molecular complex detection* tidak bekerja secara efisien jika *threshold* terlalu besar.

Untuk *running time* yang diperlukan fungsi-fungsi pada sistem sangat dipengaruhi dengan jumlah baris dari log serta jumlah *edge* dari log. Adapun jumlah baris dari log mempengaruhi waktu yang dibutuhkan sistem untuk tahap *preprocessing* sedangkan jumlah *edge* sangat mempengaruhi

waktu yang dibutuhkan sistem pada proses pembuatan graf dan klasterisasi graf dengan algoritma *molecular complex detection*.

Sedangkan pada uji coba visualisasi graf, nilai akurasi dari log yang digunakan kurang berpengaruh pada hasil visualisasi dari log tersebut. Namun, jumlah dari *edge* dari file log tersebut sangat berpengaruh pada hasil visualisasi. Selain itu, karena dataset *SecRepo* mempunyai lebih banyak *edge* dibandingkan dataset *Hofstede2014* maka hasil visualisasi pada dataset *SecRepo* menjadi lebih bagus dibandingkan dengan dataset *Hofstede2014*.

(Halaman ini sengaja dikosongkan)

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 didapat berdasarkan hasil uji coba klasterisasi log dengan algoritma *molecular complex detection* adalah sebagai berikut :

1. Implementasi algortima *molecular complex detection* dapat digunakan pada klasterisasi log autentikasi.
2. Pada kasus tugas akhir ini, *range threshold* yang bagus didapatkan dari **0.0** sampai dengan **0.5**.
3. *Threshold* terbaik yang didapatkan pada dataset *Hofstede2014* adalah **0.2** untuk *threshold string similarity* dan **0.1** untuk *threshold* algoritma *molecular complex detection*. Sedangkan t*hreshold* terbaik yang didapatkan pada dataset *SecRepo* adalah **0.3** untuk *threshold string similarity* dan **0.3** untuk *threshold* algoritma *molecular complex detection*.
4. Rata-rata akurasi terbaik yang didapatkan pada *testing* data adalah **99%**, yaitu pada dataset *SecRepo*.
5. *Running time* pada kasus tugas akhir ini dipengaruhi oleh jumlah baris dari log dan jumlah *edge* atau *node* dari log. *Running time* dari tahap *preprocessing* dipengaruhi oleh jumlah baris dari log. Sedangkan *running time* pada tahap pembuatan graf dan klasterisasi dipengaruhi oleh jumlah *node* dan *edge* dari log.

6.2 Saran

Saran yang diberikan terkait dengan pengembangan pada Tugas Akhir ini adalah :

1. Bisa memvisualisasi log dengan *real time* dengan cara menghitung seluruh proses secara dinamik, bukan statis seperti pada tugas akhir ini
2. Dapat dicoba menggunakan data log autentikasi dengan data yang didapatkan dari *server* asli (bukan dataset log)
3. Menggunakan data log lain selain log autentikasi

DAFTAR PUSTAKA

[1] H. Studiawan, B. A. Pratomo and R. Anggoro, "Connected Component Detection for Authentication Log Clustering," 2016.

[2] M. Wu, X. Li and C.-K. Kwoh, "Algorithms for Detecting Protein Complexes in PPI Networks: An Evaluation Study," *MCODE*, pp. 2-3.

[3] K. K. Sindhu and B. B. Meshram, "Digital Forensics and Cyber Crime Datamining," *Journal of Information Security*, 2012.

[4] S. F. Y. and T.-h. Kim, "IT Security Review: Privacy, Protection, Access Control, Assurance and System Security," *International Journal of Multimedia and Ubiquitous Engineering*, 2007.

[5] H. Studiawan, F. Sohel and C. Payne, "Graph Clustering and Anomaly Detection of Access Control Log for Forensic Purposes," *Forensics*, pp. 2-16, 2016.

[6] R. Hofstede, L. Hendriks, A. Sperotto and A. Pras, "SSH Compromise Detection using NetFlow/IPFIX," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 20-26, 2014.

[7] T. S. Madhulatha, "AN OVERVIEW ON CLUSTERING METHODS," *IOSR Journal of Engineering*, 2012.

[8] "AllegroViva," [Online]. Available: http://allegroviva.com/allegromcode/mcode-algorithm/. [Accessed 22 January 2017].

[9] H. Studiawan, "GitHub," [Online]. Available: https://github.com/studiawan/labeled-authlog/blob/master/possible_attacks.md. [Accessed 21 Mei 2017].

[10] "Anaconda Distribution | Continuum Analytics : Documentation," [Online]. Available:

https://docs.continuum.io/anaconda/index. [Accessed 22 January 2016].

[11] "Wikipedia," [Online]. Available: https://id.wikipedia.org/wiki/Python_(bahasa_pemrograman). [Accessed 22 January 2017].

[12] S. Community, "NumPy v1.10 Manual," 18 October 2015. [Online]. Available: http://docs.scipy.org/doc/numpy-1.10.1/index.html. [Accessed 22 January 2017].

[13] "SciPy," Scientific Computing Tools for Python, [Online]. Available: https://scipy.org/. [Accessed 22 January 2017].

[14] Python Software Foundation, "Python 2.7.13 documentation," 1990-2017. [Online]. Available: https://docs.python.org/2/library/profile.html. [Accessed 28 May 2017].

[15] "Overview -- NetworkX," [Online]. Available: https://networkx.github.io/. [Accessed 22 January 2017].

[16] M. B. Ilmy, N. Rahmi and R. L. Bu'ulölö, "Penerapan Algoritma Levenshtein Distance untuk Mengoreksi," *Paper*, pp. 1-3.

[17] scikit-learn, "scikit-learn 0.18.1 documentation," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html. [Accessed 22 January 2017].

[18] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Gephi. [Accessed 22 January 2017].

[19] S. E. Schaeffer, "Survey Graph Clustering," *Computer Science Review 1,* 2007.

BIODATA PENULIS

I Gusti Ngurah Arya Bawanta merupakan anak dari pasangan Bapak I Gusti Ngurah Sudiana dan Ibu Ni Ketut Surinih. Lahir di Karangasem pada tanggal 21 Maret 1995. Penulis menempuh Pendidikan formal dimulai dari TK Kumara (1999-2001), SDN 6 Besakih (2001-2007), SMP Cipta Dharma Denpasar (2007-2010), SMAN 4 Denpasar (2010-2013) dan S1 Teknik Informatika ITS (2013-2017). Bidang studi yang diambil oleh penulis saat berkuliah di Teknik Infomatika ITS adalah Komputasi Berbasis Jaringan (KBJ). Penulis aktif dalam organisasi seperti Tim Pembina Kerohanian Hindu ITS (2014-2016). Penulis aktif dalam berbagai kegiatan kepanitiaan, yaitu SCHEMATICS 2014 divisi kesekretariatan dan SCHEMATICS 2015 divisi pembuat soal untuk lomba NLC. Penulis juga menyukai kegiatan sosial dan pecinta alam. Penulis memiliki hobi membaca buku dan menyukai hal baru. Penulis dapat dihubungi melalui email: arya.bawanta@gmail.com.