ANALISIS PERBANDINGAN ALGORITMA *DIJKSTRA* DAN *BELLMAN-FORD* UNTUK MENENTUKAN RUTE TERPENDEK DENGAN MENGGUNAKAN *SOFTWARE DEFINED NETWORK*

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Ulfa Kurniawati
NIM: 125150301111021

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2016

PENGESAHAN

ANALISIS PERBANDINGAN ALGORITMA *DIJKSTRA* DAN *BELLMAN-FORD* UNTUK MENENTUKAN RUTE TERPENDEK DENGAN MENGGUNAKAN SOFTWARE DEFINED NETWORK

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Ulfa Kurniawati
NIM: 125150301111021

Skripsi ini telah diuji dan dinyatakan lulus pada
29 Juli 2016
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Rakhmadhany Primananda, S.T, M.Kom	Dahnial Syauqy, S.T., M.T., M.Sc.
NIK: -	NIK: -

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 29 Juli 2016

Ulfa Kurniawati

NIM: 125150301111021

KATA PENGANTAR

Puji syukur kehadirat Allah SWT yang telah melimpahkan rahmat, taufik dan hidayah-Nya sehingga laporan skripsi yang berjudul "Analisis Perbandingan Algoritma *Dijkstra* Dan *Bellman-Ford* Untuk Menentukan Rute Terpendek Dengan Menggunakan Software Defined Network" ini dapat terselesaikan.

Penulis menyadari bahwa penyusunan skripsi ini tidak terlepas dari bantuan berbagai pihak. Oleh karena itu, penulis ingin menyampaikan rasa hormat dan terima kasih kepada:

1. Bapak Rakhmadhany Primananda, S.T, M.Kom selaku dosen pembimbing 1 yang telah membimbing penulis dengan penuh kesabaran hingga skripsi ini selesai.
2. Bapak Dahnial Syauqy, S.T., M.T., M.Sc selaku dosen pembimbing 2 yang telah membimbing dalam hal penulisan hingga skripsi ini selesai.
3. Ibu Sri Sudarmi, ibunda tercinta penulis yang tak henti-hentinya mendoakan penulis selama awal perkuliahan hingga detik ini.
4. Bapak Lutfi Argufi (Alm), ayahanda tercinta yang selalu mendoakan penulis.
5. Ardy Novian Erwanda yang selalu memembantu dan menemani penulis dalam proses pengerjaan penulis.
6. Irma Indrawati, adikku tercinta yang tak henti-hentinya memberi dukungan kepada penulis dan memberi semangat ketika penulis sedang putus asa.
7. Tomi Subiakto dan Ahmad Faisol, kakak-kakakku tercinta yang memberi dukungan dan semangat dalam menyelesaikan skripsi ini.
8. Bapak Rofiq, ibu Rosidah, ibu Watsiqoh, bapak suhartono selaku pakde, tante, dan om dari penulis. Terimakasih banyak atas support-nya kepada penulis.
9. Bapak Adharul Muttaqin,S.T., M.T. dan Sabriansyah Rizqika Akbar, S.T., M.Eng selaku Mantan Kaprodi dan Kaprodi Teknik Komputer serta segenap Bapak/Ibu Dosen, Staf Administrasi dan Perpustakaan Program Studi Teknik Komputer Universitas Brawijaya.
10. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D selaku Ketua Jurusan Teknik Informatika.
11. Lintang, Ike, Lilis, Firdha, Sefi, Maulita, Bila, Melly, Shanti, Winda, Ringga, Affan, Singgih, Rizky, Helmanda, dan teman-teman seperjuangan SISKOM 2012 tercinta, yang memberi motivasi untuk lulus cepat.
12. Ajeng Rindau, Ajeng Dikna dan teman-temanku SMA Ulul Alb@b yang memberikan motivasi dan dukungan untuk penulis sehingga penulis dapat masuk di universitas brawijaya ini.
13. Keluargaku yang ada di jombang dan di malang terimakasih karena telah menjadi semangat dan motivasiku untuk bisa mendapatkan gelar sarjana ini.
14. Ibu, Bapak, dan Teman-teman kos kertosento 41A yang memberi *energy* positif dan semangat kepada penulis.

15. Semua pihak yang telah membntu penulis, baik dari material maupun non-material. Terimakasih atas motivasi dan semangat yang telah ditularkan kepada penulis

Penulis mengucapkan banyak terima kasih dan penulis sadar akan adanya beberapa kekurangan pada laporan ini. Sehingga penulis mengharapkan adanya penyempurnaan dari pihak-pihak terkait. Semoga laporan ini dapat memberikan manfaat dan referensi untuk melakukan penelitan dalam penyempurnaan sistem.

Malang, 29 Juli 2016

Penulis
ulfakurniawan@gmail.com

ABSTRAK

Software defined network merupakan suatu pemodelan jaringan yang memisahkan antara *control plane* dan *data plane*. SDN mulai dikembangkan beberapa tahun terakhir dan sudah banyak diimplementasikan antara lain pada *routing* jaringan. Terdapat dua algoritma *routing* dalam penelitian ini yaitu algoritma *routing Dijkstra* dan *Bellman-Ford*. Kedua algoritma tersebut memiliki metode pencarian yng berbeda. Algoritma *Bellman-Ford* akan melakukan pencarian dengan menggunakan informasi dalam tabel *routing*-nya sendiri dan tetangganya, sedangkan Algoritma *Dijkstra* mengetahui keseluruhan informasi global dalam topologi jaringan sampai tujuan. Kedua algoritma ini akan diimplementasikan menggunakan *software defined network* untuk kemudian dilakukan analisis dan perbandingan. Implementasi ini menggunakan topologi Abilene dan menggunakan Pyretic sebagai *controller*-nya. Pengujian yang akan dilakukan antara lain pemilihan jalur, *bandwidth*, *throughput*, *delay*, waktu eksekusi, dan rute terpendek. Perbandingan hasil yang didapat dari pengujian pemilihan jalur bahwa program Algoritma *Dijkstra* dan *Bellman-Ford* akan melakukan pemilihan jalur berdasarkan 3 kondisi, urutan *input* tabel *routing*, *cost* minimum, dan jumlah *hop*. Sedangkan perbandingan dari pengujian *bandwidth* dan *throughput*, Algoritma *Dijkstra* memiliki kenaikan *throughput* yang kecil (di bawah Bellman-Ford) pada *bandwidth* yang kecil pula, namun memiliki kenaikan *throughput* yang cukup baik ketika diberikan *bandwidth* besar (di atas 750 Mbps). Sedangkan Algoritma *Bellman-Ford* memiliki peningkatan *throughput* yang cukup besar ketika *bandwidth* kecil (*bandwidth* di bawah 750 Mbps), namun pada *bandwidth* besar, *throughput* tidak mengalami peningkatan yang besar. Tidak ada perubahan nilai *delay* yang signifikan pada masing-masing algoritma. Untuk perbandingan pengujian waktu eksekusi, program *Dijkstra* memiliki waktu berkisar antara 0.00048 hingga 0.000633 detik. Sedangkan waktu eksekusi program *Bellman-Ford* memiliki kisaran waktu 0.341 hingga 0.405 detik. Untuk pengujian yang terakhir, pemilihan rute terpendek tidak terpengaruh oleh perubahan nilai *bandwidth* dan *throughput*.

Kata Kunci— *Software Defined Network, Dijkstra*, *Bellman-Ford*, Pyretic

ABSTRACT

Software Defined Network is a model of network that decouples the control plane and the data plane . SDN has been developed in few years ago, and already has had implementation for many network routings. There are two network routing algorithms implemented here. They are Dijkstra algorithm and Bellman-Ford algorithm. The two algorithms have a different routing method. The Bellman-Ford algorithm uses its routing table and its neighbors', the Dijkstra algorithm collects the whole information of the network topology from the source node to the destination. The both algorithms would be implemented using the software defined network model to analyze their performance. The implementation used the Abilene topology, and Pyretic as it's controller. They would be tested and analyzed for their performance of path selection, bandwidth, throughput, delay, execution time, and the shortest path They have gained. The analysis results obtained from path selection testing scenario showed that the two algorithms will perform path selection based on three conditions, input routing table, minimum cost, and a number of hops. Then the second testing scenario, analysis of the results of Dijkstra Algorithm's throughput increased sequentially as the bandwidth's increase. The Dijkstra algorithm had a small increase in throughput (compare to the Bellman- Ford) in a small bandwidth as well throughput (under Bellman- Ford) in a small bandwidth as well , but has a good increase in throughput when given large bandwidth (above 750 Mbps). While the Bellman-Ford algorithm has a throughput that is quite large when the bandwidth is small (under 750 Mbps), but on a large bandwidth, did not experience a spike. Analysis of third results showed no significant change in the value of delay for each algorithm. The execution time of both algorithms was 0.00048 - 0.000633 seconds for Dijkstra, and 0.341 - 0.405 seconds for Bellman-Ford. And the last test, the shortest path selecting performance did not affect by the bandwidth and throughput's change.

Keyword: Software Defined Network, Dijkstra, Bellman-Ford, Pyretic

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Perkembangan jaringan saat ini meningkat dengan pesat dalam berbagai bentuk dan model, seperti *delay* toleran network, software defined network, dan sebagainya. SDN atau *Software Defined Network* merupakan konsep untuk memisahkan *control plane* dan *data plane* dari perangkat jaringan. Protokol *Openflow* adalah protokol untuk merealisasikan konsep SDN yang bertujuan untuk penelitian dan eksperimen protokol jaringan (Mckeown et al, 2010). Protokol ini menjadikan SDN bersifat "*open*", dimana tidak harus terikat dengan berbagai vendor jaringan.

Vendor-vendor penyedia jaringan seperti cisco, juniper, NEC, dan lain sebagainya, memiliki mekanisme dan konfigurasi yang berbeda. Perbedaan ini mengakibatkan kesulitan jika ingin mengabungkan vendor yang berbeda dalam satu jaringan, hal ini menjadi salah satu alasan diciptakannya SDN. SDN juga menerapkan konsep jaringan yang memiliki *controller* terpusat, *controller* inilah yang bertanggung jawab untuk melakukan *forwarding* setiap paket yang akan berkomunikasi dalam *flow*. Pengguna SDN dapat menjalankan berbagai aplikasi jaringan dalam *controller* tersebut, termasuk diantaranya yang bersifat me-*manage* maupun *monitoring*.

Tujuan utama dari SDN sendiri adalah untuk menerapkan berbagai aplikasi jaringan yang mudah dan *universal* untuk diprogram pada *controller*-nya, misalnya teknologi *Load balancing*, Multimedia *multicast*, *Intrusion detection*, sampai berbagai macam *virtualisasi* jaringan bisa diterapkan menggunakan SDN ini. Untuk melakukan pemrograman, beberapa bahasa pemrograman berhasil diciptakan untuk memfasilitasi SDN, diantaranya Frenetic, Pyretic, dan Floodlight.

Pyretic merupakan gabungan dari python dan Frenetic. Pyretic adalah bahasa *friendly* programmer dan tertanam bahasa python sebagai sistem runtime yang mengimplementasikan program yang ditulis dalam Pyretic, pada *switch j*aringan. Pyretic menghasilkan abstraksi jaringan dan mengizinkan programmer untuk membuat perangkat lunak modular untuk SDN (Foster et al, 2011).

Agar suatu paket cepat diterima maka dibutuhkan protokol *routing*. Protokol *routing* adalah suatu protokol yang digunakan untuk mendapatkan rute dari satu jaringan ke jaringan yang lain. Ada 2 mekanisme *routing* yaitu *routing* statis dan *routing* dinamis. *Routing* statis pembuatan dan *pengupdatean routing* Tabel secara manual. *Routing* statis kurang efisien jika digunakan pada jaringan berskala besar. Sedangkan *routing* dinamis kebalikan dari statis, *routing* dinamis lebih memudahkan administrator karena *routing* akan menyesuaikan kebutuhan.

Protokol *routing* mempelajari semua *router* yang ada, menempatkan rute yang terbaik ke tabel *routing*, dan juga menghapus rute ketika rute tersebut sudah tidak valid lagi. Untuk membangun suatu *routing* protokol yang baik maka diperlukan algoritma *routing* protokol yang dapat menentukan rute terpendek pada berbagai macam topologi tanpa melakukan pengkonfigurasian ulang. Algoritma *routing* yang ada di jaringan salah satunya adalah Algoritma *Dijkstra* dan *Bellman-Ford*.

Kedua algoritma tersebut memiliki cara pencarian yang berbeda, Algoritma *Bellman-Ford* akan melakukan pencarian dengan menggunakan informasi dalam tabel *routing*-nya sendiri dan tetangganya, sedangkan Algoritma *Dijkstra* mengetahui keseluruhan informasi global dalam topologi jaringan sampai tujuan.

Algoritma *routing* akan mempermudah dalam administrasi jaringan, serta sudah banyak penelitian yang menganalisis perbandingan kedua algoritma tersebut dalam struktur jaringan biasa. Namun, analisis yang dilakukan tentu akan mengalami perbedaan dari sisi performa karena pada jaringan biasa tugas *control* dan *forwading* akan ditangani oleh masing-masing *router*, sedangkan pada SDN tugas *control* dan *forwading* dipisah.

Penelitian sebelumnya tentang analisis perbandingan Algoritma *Dijkstra* dan *Bellman-Ford* pernah dilakukan Vaibhavi Patel dari KalolInstitute of Technology & research Center, Gujarat, India; dalam paper yang berjudul "*A Survey Paper of Bellman-Ford Algorithm and Dijkstra Algorithm for Finding Shortest Path in GIS Application*" pada Februari 2014 lalu. Paper tersebut melakukan perbandingan kedua algoritma dalam penerapannya pada GIS (*Geographic Information Sistem*). Dimana parameter yang diuji meliputi kompleksitas waktu dan kompleksitas ruang, waktu respon kedua algoritma dihitung berdasarkan jumlah *switch* yang ditambah secara bertahap. Hasilnya, Algoritma *Dijkstra* lebih cepat dibandingkan *Bellman-Ford*. *Bellman-Ford* tidak cocok untuk diterapkan dalam jaringan skala besar (Patel, 2014).

Penelitian kedua, oleh Vina Rifiani dari PENS-ITS, melakukan "*Analisa Perbandingan Metode Routing Distance Vector Dan Link State Pada Jaringan Packet"*. Hasilnya Algoritma *Dijkstra* dan *Bellman-Ford* memiliki keunggulan dibeberapa pengujian QoS. Dalam paper ini digunakan simulator Network Simulator 2 (NS 2) (Vina, 2011).

Dari penelitian yang sudah ada, penulis berminat untuk melanjutkan analisis perbandingan Algoritma *Dijkstra* dan *Bellman-Ford* dengan menggunakan *Software Defined Network*. Untuk melakukan pengujian, penulis menggunakan beberapa parameter uji. Kedua algoritma akan dijalankan menggunakan *controller* Pyretic pada topologi *Abilene*. Topologi *Abilene* adalah suatu topologi yang sudah diterapkan di Amerika Serikat karena pernah digunakan untuk jalur kereta api yang melintasi perbatasan Amerika Serikat yang tercatat pada stasiun *Abilene*.

1.2 Rumusan masalah

Berdasarkan uraian latar belakang di atas, maka dapat dirumuskan permasalahan yaitu sebagai berikut :

1. Bagaimana cara mengimplementasikan Algoritma *Dijkstra* dan *Bellman-Ford* pada *Software Defined Network* (SDN) dengan topologi *Abilene*?
2. Bagaimana hasil pengujian dengan beberapa parameter uji (rute terpendek, *bandwidth, throughput, latency*, dan waktu eksekusi program) dari masing-masing algoritma?
3. Bagaimana analisis dan perbandingan dari kedua algoritma dilihat dari hasil pengujian yang telah dilakukan?

1.3 Tujuan

Tujuan yang akan diperoleh pada penelitihan ini akan dikelompokkan menjadi dua yaitu tujuan umum dan tujuan khusus. Tujuan umum berisi alasan mengapa suatu sistem dibuat. Sedangkan tujuan khusus akan menjawab tujuan dari rumusan masalah yang ada. Berikut adalah tujuan umum dan tujuan khusus:

- Tujuan umum
  Melakukan pengujian, menganalisis, dan membandingkan Algoritma *Dijkstra* dan *Bellman-Ford*.
- Tujuan khusus
  1. Mengimplemenasikan Algoritma *Dijkstra* dan *Bellman-Ford* pada *Software Defined Network* dengan topologi *Abilene*.
  2. Melakukan pengujian dengan beberapa parameter pada masing-masing algoritma.
  3. Menganalisis dan membandingkan kedua algoritma dari hasil pengujian yang telah didapat pada pengujian.

1.4 Manfaat

Manfaat berisi pihak-pihak yang akan diuntungkan jika sistem ini behasil dibuat. Berikut adalah manfaat yang akan diperoleh:

- Bagi administrator jaringan dapat menjadi *alternative* model jaringan dan referensi pemilihan algoritma *routing*.
- Bagi pemerintah agar lebih menfasilitasi pembangunan model jaringan *Software Defined Network* (SDN).
- Bagi penulis, penulis dapat mengasah keterampilan dalam bidang pemodelan jaringan komputer khususnya pemodelan jaringan *Software Defined Network* (SDN) dan penulis dapat mengimplementasikan suatu algroritma pada jaringan *Software Defined Network* (SDN).

1.5 Batasan masalah

Dalam melakukan analisis perbandingan Algoritma *Dijkstra* dan *Bellman-Ford* dengan menggunakan *software defined network*. Diperlukan suatu batasan masalah pada sistem akan dibuat, hal ini bertujuan agar pembahasan masalah tidak terlalu meluas. Maka batasan masalah yang akan dibahas adalah sebagai berikut :

- *Controller* yang digunakan adalah Pyretic.
- Masing-masing *switch* memiliki 1 *host*, dengan jumlah *switch* maksimal 11.
- 1 *host* yang menjadi server adalah 10.0.0.1
- Bahasa pemprograman yang digunakan adalah *python*.
- Pencarian rute terpendek ditentukan dengan nilai *cost* yang diperoleh minimum.
- Program tidak memungkinkan terjadinya perubahan *cost* ketika sedang dijalankan.
- Pegujian tidak menggunakan *cost* dengan nilai *negative*.

1.6 Sistematika Pembahasan

Sistematika pembahasan skripsi yang disusun ini akan dibahas pada bab-bab yang akan diuraikan di bawah ini:

BAB I : PENDAHULUAN

Menguraikan tentang latarbelakang permasalahan mencoba merumuskan inti permasalahan dan menentukan tujuan untuk kegunaan penelitian yang kemudian diikuti dengan pembatasan masalah, asumsi metodelogi penelitian serta sistematikan penulisan.

BAB II : LANDASAN KEPUSTAKAAN

Bab ini berisi kajian pustaka dan semua dasar–dasar teori untuk digunakan selanjutnya pada bagian pembahasan.

BAB III: METODOLOGI PENELITIAN

Dalam bab ini akan membahas studi literature, analisa kebutuhan, rancangan algoritma serta pengujian algoritma menggunakan metode *Software Defined Network.*

BAB IV: IMPLEMENTASI

Bab ini berisi tentang implementasi *Software Defined Network* pada Algoritma *Dijkstra* dan *Bellman-Ford* menggunakan topologi yang telah ditentukan.

BAB V: PENGUJIAN DAN ANALISIS

Bab ini berisi pengujian, analisis, dan perbandingan hasil dari subbab sebelumnya. Perbandingan antara kedua algoritma dengan melihat beberapa parameter pengujian yang telah ditentukan.

BAB VI: KESIMPULAN DAN SARAN

Bab ini berisi kesimpulan dari perancangan sistem dengan menggunakan *software define network.* Untuk lebih meningkatkan hasil akhir yang lebih baik maka diberikan juga saran-saran untuk perbaikan serta penyempurnaan proyek akhir ini.

BAB 2 LANDASAN KEPUSTAKAAN

Dalam bab ini akan dibahas mengenai kajian pustaka, teori-teori penting yang dapat menunjang dan menjadi acuan dalam pembuatan proyek akhir. Bagian tersebut meliputi penjelasan mengenai pengenalan *software defined network* (SDN), *Openflow*, *controller*, Pyretic, simulator, mininet, *routing*, Algoritma *Dijkstra*, Algoritma *Bellman-Ford*, dan topologi *Abilene*.

2.1 Kajian pustaka

Kajian pustaka akan berisi tentang perbandingan analisis yang pernah dilakukan dengan rencana penelitian yang akan dilakukan, berikut adalah perbandingannya.

Tabel 2.1 Kajian Pustaka

No	Nama Penulis, Tahun, dan Judul	Persamaan	Perbedaan	
			Penelitian terdahulu	Rencana penelitian
1.	Vaibhavi Patel [2014] *A Survey Paper of Bellman-Ford Algorithm and Dijkstra Algorithm for Finding Shortest Path in GIS Application*. KalolInstitute of Technology & research Center, Gujarat, India	Membanding kan Algoritma *Dijkstra* dan *Bellman-Ford* untuk mencari jalur terpendek	Melakukan Perbandingan algoritma dengan menggunakan aplikasi GIS (Geographic Information Sistem)	Melakukan perbandingn algoritma dengan menggunakan mininet sebagai simulator SDN
2.	Vina Rifiani [2011] *Analisa Perbandingan Metode Routing Distance Vector Dan Link State Pada Jaringan Packet*. PENS-ITS	Membanding kan algoritma *routing Dijkstra* dan *Bellman-Ford*	Menganalisis berdasarkan parameter QoS (*delay, packet loss, jitter* dan *throughput*)	Menganalisis berdasarkan beberapa parameter uji yang telah ditentukan.
3.	Jehn-Ruey Jiang[2014] *Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking*. National Central University, Taiwan.	Membanding kan *Dijkstra* dengan menggunakan SDN	Membanding kan extending *Dijkstra* dengan *Dijkstra* menggunakan SDN	Melakukan perbandingan algoritman *Bellman-Ford* dan *Dijkstra* menggunakan SDN

Tabel 2.1 (Lanjutan)

No	Nama Penulis, Tahun, dan Judul	Persamaan	Perbedaan	
			Penelitian terdahulu	Rencana penelitian
4.	ArnavShivendu [2015] *Emulation of Shortest Path Algorithm in Software Defined Networking Environment.* Sikkim Manipal Institute of Technology, East Sikkim, India	Pencarian Shortest Path dengan implementasi Algoritma *Bellman-Ford* dengan menggunakan SDN	Menggunakan *controller* terpusat POX	Menggunakan *controller* terpusat Pyretic

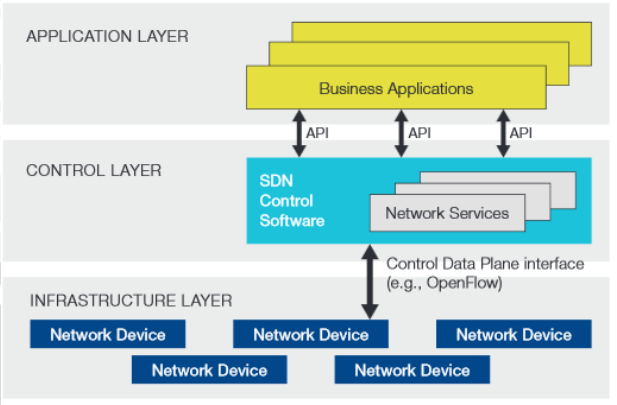
2.2 Dasar teori

Berdasarkan beberapa informasi yang didapat dari beberapa kajian pustaka, maka dalam "Analisis Perbandingan Algoritma *Dijkstra* Dan *Bellman-Ford* Untuk Menentukan Rute Terpendek Dengan Menggunakan *Software Defined Network*" terdapat beberapa dasar teori, antara lain:

2.2.1 Software Defined Network

Sejak berkembangnya pengguna internet di dunia vendor-vendor tersebut memiliki peran penting dalam mendukung terselenggaranya koneksi internet. Setiap perangkat yang berbeda vendor maka akan menggunakan *command* yang berbeda pula ketika hendak melakukan *Setting*, hal ini menjadi kendala tersendiri bagi seorang *network* administrator meski tidak terlalu menjadi masalah besar. Kendala lainnya yang juga cukup krusial dimana *network* admin tidak dapat dengan mudah melakukan kustomisasi pada jaringan sesuai degan kebutuhan. Pada tahun 2008 Nick Feamster dkk, pada jurnalnya *"The case for separating routing from routers"*, yang isinya memberikan solusi untuk melakukan pemisahan antara *control plane* dan *data plane*, dimana *control plane* akan diletakkan secara terpusat pada sebuah *controller*.

Dengan adanya pemisahaan tersebut maka diharapkan perangkat jaringan dapat di manage melalui *controller*-nya saja, maka untuk mewujudkan tersebut dibutuhkanlah sebuah API untuk mengkoneksikan seluruh perangkat jaringan kedalam sebuah *controller* yang dapat di program sesuai kebutuhan perusahaan. Pada jaringan konvisional saat ini, sistem pembuat keputusan arus data yang akan dikirim, dibuat menyatu dengan perangkat keras. Dengan konsep pendekatan jaringan, konfigurasi SDN dapat menciptakan jaringan dimana perangkat keras pengontrol lalu lintas data secara fisik dipisahkan dari perangkat keras yang bertugas sebagai *forwarding*.



Gambar 2.1 Arsitektur *software defined network*
Sumber: Jehn-Ruey Jiang (2014)

Berdasarkan Gambar 2.1 di atas terdapat tiga layer yang menyusun arsitektur SDN yaitu *application layer, control layer* dan *infrastructure layer*. Pada *application layer* merupakan layer yang berisikan aplikasi-aplikasi seperti *mail server*, *web server* maupun lainnya. *Control layer* memiliki peranan penting dalam mengendalikan sistem, pada layer ini merupakan bagian saat *controller* dipisahkan dari *data plane*. Sedangkan *data plane* merupakan sekumpulan dari perangkat *networking* yang akan dikendalikan oleh *control plane*. *Infrastructure layer* dapat beroperasi sesuai dengan perintah dari *controller*. Komunikasi antara perangkat dengan *controller* menggunakan protokol yang disebut *Openflow*. (Astuto et al, 2014).

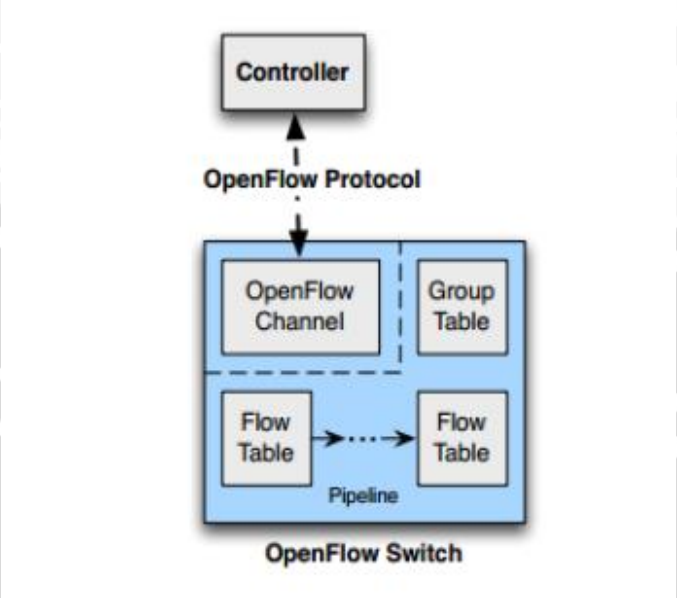
Software Defined Network dengan konsep pemisahan *control plane* dan *data plane* memberikan beberapa kelebihan di antaranya :

1. *Programmability* dan *Automation* adalah konsep terprogram yang mampu melakukan perubahan infrastruktur jaringan lokal tanpa bergantung pada vendor dengan dilakukan secara otomatis. Seperti kemampuan terhadap perubahan *policy* dan kemampuan *troubleshooting* maupun lainnya.
2. *Scalability* dan *Orchestration* merupakan kemampuan untuk mengatur dan mengelola ribuan perangkat melalui manajemen jaringan secara terpusat.

2.2.2 Openflow

Openflow adalah konsep yang sederhana, dimana melakukan sentralisasi terhadap kerumitan dari jaringan kedalam sebuah software kontroler, sehingga seorang administrator dapat mengaturnya dengan mudah dengan hanya mengatur kontroler tersebut. Konsep *Openflow* ini dengan ide awal adalah menjadikan sebuah network dapat di program atau dikontrol (Mckeown et al, 2010). *Openflow* merupakan protokol yang digunakan untuk komunikasi antara *network infrastructure devices* dengan SDN *control* software. Protokol *Openflow* sebagai media komunikasi antara *control plane* dengan *data plane* menggunakan

perangkat lunak pengendali (*controller*). Protokol *Openflow* melakukan sentralisasi terhadap kerumitan dari jaringan ke dalam sebuah software *controller* sehingga administrator dapat melakukan pengaturan jaringan melalui *Controller* tersebut dengan mudah. *Openflow* memiliki struktur cara kerja sebagai berikut.



Gambar 2.2 *Openflow*
Sumber: Jehn-Ruey Jiang (2014)

Berdasarkan Gambar 2.2 di atas menjelaskan jika *Openflow* terdiri dari satu atau lebih *flow table* dan/atau grup table. *Openflow controller* dapat memperbarui, menambah dan menghapus entri *flow* yang berada dalam Tabel *flow* kedua reaktif dan proaktif.

Protokol *Openflow* terdiri dari tiga komponen penting yaitu *header fields*, *action* dan *counter*. *Header fields* merupakan paket *header* yang mendefinisikan suatu *flow* dengan *fields* yang terdiri dari enkapsulasi seperti enkapsulasi segmen pada protokol VLAN *ethernet* standar. Jumlah paket dan *byte* setiap *flow* untuk membantu dalam membuang dan me-nonaktifkan *flow* terdapat pada *counter*. Selanjutnya untuk mekanisme kerja protokol *Openflow* setiap paket yang lewat ke *port* yang sesuai akan diteruskan menuju *destination*. Sehingga protokol *Openflow* tidak hanya dapat melakukan *forwarding* tetapi juga dapat dilakukan pengaturan pergerakan paket terhadap jaringan tersebut.

Mekanisme kerja protokol *Openflow* saat *switch Openflow* menerima paket yang datang dan tidak memiliki kecocokan terhadap *table flow* yang ada maka *switch Openflow* akan meneruskan paket menuju *controller Openflow. Controller* memberikan respon terhadap paket tersebut berdasarkan *table flow*. Sehingga mekanisme komunikasi antara *control plane* dengan *data plane* yang diatur melalui *controller* dapat memberikan respon yang sesuai terhadap setiap paket yang datang.

2.2.3 *Controller*

Software Defined Network memiliki beberapa *controller* yang dapat membantu konfigurasi komunikasi antara *application layer* dan *infrastructure layer. Controller* merupakan bagian arsitektur jaringan pada *Software Defined Network* yang berfungsi sebagai pusat pengontrolan atau logika jaringan. Seluruh kebijakan jaringan seperti *routing, switching* maupun pengaturan jaringan lainnya terdapat pada *controller*. Fungsional jaringan seperti *routing* dan lainnya dikembangkan melalui modul yang terdapat pada *controller* dengan bahasa pemrograman masing-masing. SDN memmiliki beberapa *controller* yaitu Pyretic, FloodLight, Open Daylight, Trema, POX dan Ryu.

Tugas *controller* antara lain menginterpretasikan pesan yang dikirim oleh *switch Openflow* dan menyediakan semua instruksi pada setiap spesifikasi (baik spesifikasi yang dibutuhkan, tambahan atau keduanya) untuk dapat diprogram kedalam *switch* (A Vishnoi, 2013).

2.2.3.1 Pyretic

Frenetic diperkenalkan sebagai bahasa tingkat tinggi untuk program *controller* untuk mengelolahan *switch* dalam jaringan SDN. *Frenetic* menyediakan *programmer* untuk mengklasifikasi dan menggabungkan lalu lintas jaringan serta sebagai fungsional untuk menggambarkan tingkat tinggi kebijakan paket-*forwarding* (Jehn-Ruey Jiang, 2014). Pyretic merupakan gabungan dari *python* dan Frenetic. Pyretic adalah bahasa *friendly programmer* dan tertanam bahasa *python* sebagai sistem *runtime* yang mengimplementasikan program yang ditulis dalam Pyretic pada *switch* jaringan.

Pyretic diperkenalkan sebagai SDN bahasa pemrograman atau *platform* yang meningkatkan tingkat abstraksi. Hal ini memungkinkan programmer untuk fokus pada bagaimana menentukan kebijakan jaringan di tingkat tinggi abstraksi (Foster et al, 2011). Fungsi abstrak akan melakukan pengambil paket sebagai masukan dan keluaran satu set paket baru. Pyretic juga memfasilitasi desain modular dengan menawarkan komposisi kebijakan dua operator, komposisi paralel dan komposisi berurutan, untuk memungkinkan *programmer* untuk menggabungkan beberapa kebijakan bersama-sama tanpa khawatir adanya bentrok. Pyretic programmer juga dapat membuat kebijakan yang dinamis yang perilakunya akan berubah *overtime*. Singkatnya, Pyretic memungkinkan *programmer* untuk SDN membuat aplikasi jaringan modular singkat pada tingkat tinggi abstraksi.

2.2.4 Simulator

Software Defined Network memiliki simulator jaringan yang dapat mempermudah melakukan simulasi di jaringan SDN. Simulator merupakan program yang berfungsi untuk menyimulasikan suatu peralatan. Ada beberapa simulator yang dapat mensimulasikan jaringan SDN antara lain NS3, EstiNet 9.0, dan Mininet.

2.2.4.1 Mininet

Mininet adalah sebuah simulator jaringan *open source* yang mendukung protokol *openflow* untuk arsitektur SDN. Salah satu *software* yang paling digunakan pada SDN adalah mininet. Mininet menggunakan pendekatan virtualisasi untuk membuat jaringan *virtual host, switch, controller*, dan *link*. Mininet adalah sebuah jaringan virtual realistis. Mininet menggunakan konsep dasar virtualisasi untuk membuat suatu sistem. Selain itu, Mininet mendukung topologi kompleks dan *user-defined* dan memberikan *python* API (Yilan Liu, 2015).

Mininet biasanya digunakan sebagai simulasi, verifikasi, *testing tool*, dan *resource*. Salah satu keunggulan mininet adalah mampu membuat topologi sesuai keinginan perancangnya. Mininet juga mampu membuat topologi yang cukup kompleks, lebih besar, dan topologi internet. Fitur lain yang sangat bagus dari mininet yaitu memungkinkan untuk kustomisasi packet *forwarding* sepenuhnya. Untuk mempermudah ketika menjalankan simulator mininet, dapat ditambahkan miniedit sebagai virtualisasinya. Dengan meggunakan miniedit, ketika merancang topologi tidak perlu melakukan *source code* antar *switch* di terminal.

2.2.5 Routing

Routing adalah suatu protokol yang digunakan untuk mendapatkan rute dari satu jaringan ke jaringan yang lain. Ada 2 mekanisme *routing* yaitu *routing* statis dan dinamis. *Routing* dinamis akan memberikan informasi dan mempelajari dari *router* sebelumnya. Sedangkan *routing* statis seorang *network* administrator mengkonfigurasi informasi tentang jaringan yang ingin ditujuan secara manual. *routing* statis tidak cocok digunakan untuk jaringan berskala besar. Sedangkan *routing* dinamis sekarang ini menjadi alternatif pilihanan karena sifatnya yang fleksibel terhadap perubahan.

Pada dinamis *routing*, protokol *routing* mengatur *router-router* sehingga dapat berkomunikasi satu dengan yang lain dan saling memberikan informasi *routing* yang dapat mengubah isi *routing* tabel, tergantung keadaan jaringannya. Dengan cara ini, *router-router* mengetahui keadaan jaringan yang terakhir dan mampu meneruskan datagram ke arah yang benar. Tujuan utama dari *routing* protokol adalah untuk membangun dan memperbaiki tabel *routing*. Dimana tabel ini berisi jaringan-jaringan dan *interface* yang berhubungan dengan jaringan tersebut.

Router menggunakan protokol *routing* ini untuk mengatur informasi yang diterima dari *router-router* lain dan *interfacenya* masing-masing, sebagaimana yang terjadi di konfigurasi *routing* secara manual. Protokol *routing* tidak hanya mempelajari semua *router* yang ada, namun juga menempatkan rute yang terbaik ke tabel *routing*, dan juga menghapus rute ketika rute tersebut sudah tidak valid lagi.

2.2.6 Algoritma *Routing*

Algoritma *routing* muncul dari perkembangan *routing* dinamis. Ketika topologi jaringan berubah karena perkembangan jaringan, admin jaringan melakukan konfigurasi ulang atau terdapat masalah pada jaringan, maka *router* akan mengetahui perubahan tersebut. Dasar pengetahuan ini dibutuhkan secara akurat untuk melihat topologi yang baru.

Algoritma *routing* adalah suatu mekanisme pencarian rute yang efisien untuk mengirimkan suatu paket. Sehingga paket yang diterima lebih cepat diterima oleh sisi klien. Algoritma *routing* memiliki tugas utama yaitu mencari rute terpendek pengiriman dengan waktu yang minimum. Suatu algoritma *routing* harus mampu mencari rute terbaik ketika terdapat jalur yang rusak.

Saat ini algoritma *routing* mulai berkembang dengan menawarkan mekanisme *routing* yang efisien. Salah satunya adalah Algoritma *Dijkstra* yang memiliki mekanisme pencarian dengan menghitung seluruh *cost* yang berhubungan dengan *link* pada setiap rute untuk mendapatkan metrik rute-rute yang terhubung. Sedangkan *Bellman-Ford* memberikan *router-router* kemampuan untuk mempublikasikan semua rute-rute yang diketahui (*router* bersangkutan).

2.2.6.1 Algoritma *Dijkstra*

Algoritma *Dijkstra* merupakan salah satu varian dari algoritma Greedy, yaitu salah satu bentuk algoritma pemecah persoalan yang terkait dengan masalah optimasi atau pencarian solusi yang optimum karena algoritma ini sederhana (Michell Setyawati, 2011). Algoritma *Dijkstra* disebut juga algoritma *Link state*.

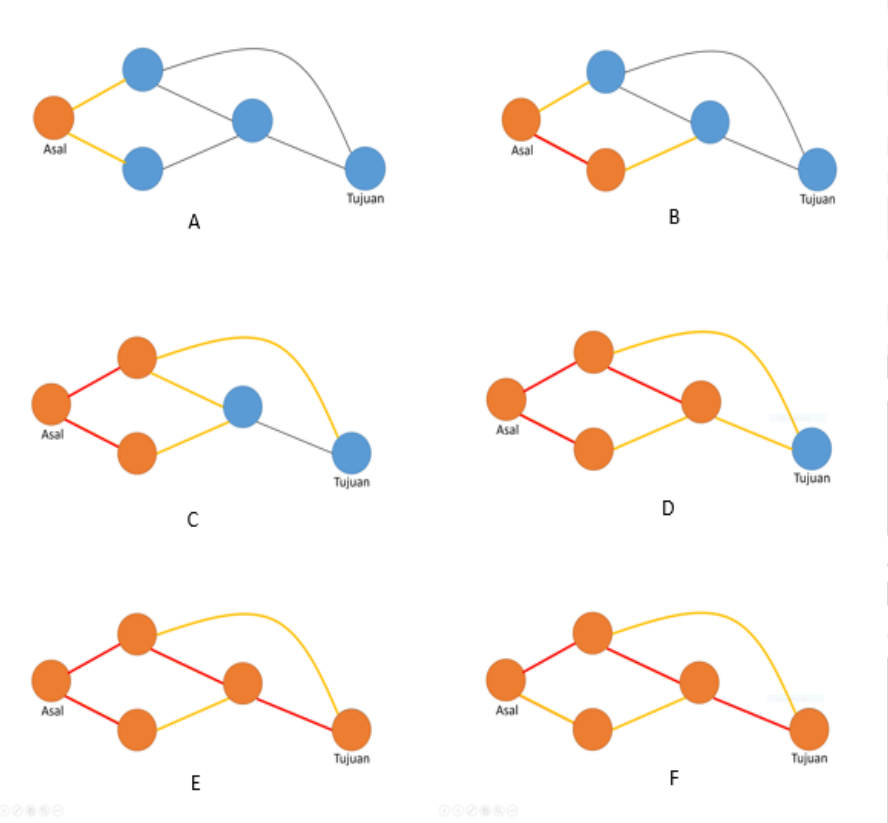
Algoritma ini mengelola suatu database kompleks dari informasi topologi. Algoritma *routing Dijkstra* mengurangi trafik *broadcast* karena Algoritma *Dijkstra* tidak secara periodik melakukan *broadcast* ataupun mengirimkan seluruh isi tabel *routing*-nya ketika melakukan *broadcast*. Berikut adalah rumus dari Algoritma *Dijkstra*.

$$D(v) = min(D(v), D(w) + c(w,v)) \quad (2.1)$$

Berdasarkan persamaan 2.1, $D(v)$ adalah jarak dari *switch* awal ke *switch* v. $c(w,v)$ adalah *cost* dari w ke v. Jadi, Algoritma *Dijkstra* akan melakukan pencarian rute terpendek dengan sebelumnya memilih *switch* awal, lalu membandingkan jarak dari dirinya sendiri ke *switch* tujuan dengan jarak milik tetangganya ditambah *cost* tetangganya ke *switch* tujuan.

Algoritma *routing Dijkstra* melakukan pertukaran data lengkap tabel rutenya ketika inisialisasi berlangsung. Selajutnya pertukaran *update* rutenya dilakukan secara multicast dan hanya pada saat terjadi perubahan. Dengan kondisi ini memungkinkan hanya perubahan saja yang dikirim ke *router-router* lain, bukan seluruh tabel *routing*-nya. Di dalam Algoritma *Dijkstra*, setiap *router* akan mempunyai informasi mengenai tetangga, seperti ID tetangga, tipe

hubungan, dan *bandwidth*. Dengan begitu, setiap *router* dapat mengetahui setiap jalur yang ada di dalam jaringan.



Gambar 2.3 Pencarian rute Algoritma Djikstra

Berdasarkan Gambar 2.3 di atas terdapat beberapa tahapan pencarian. Berikut adalah penjelasan dari beberapa tahapannya.

1. Pada Gambar 2.3 A, *switch* yang diberi jalur penghubung warna *orange* adalah jalur yang akan dibandingkan. Tahap pertama yang dilakukan adalah menunjuk satu *switch* yang berperan sebagai *switch* asal lalu *switch* asal akan melakukan pencarian dengan membandingkan *switch* yang menjadi tetangganya.
2. Pada Gambar 2.3 B, *switch* yang diberi jalur penghubung berwarna merah adalah *switch* yang dipilih karena memiliki *cost* minimum. Setelah itu akan dilakukan perbandingan kembali pada rute dengan jalur yang diberi warna *orange*.
3. Pada Gambar 2.3 C, setelah dilakukan perbandingan pada tahap sebelumnya. Didapatkan hasil kedua tetangga dari *switch* asal memiliki peluang yang sama untuk menjadi rute terpendek karena itu jalur penghubung akan diberi warna merah. Tetangga *switch* asal sekarang akan menjadi *switch* terpilih. *Switch* terpilih akan memiliki tugas melakukan perbandingan nilai *cost* tetangganya yang sebelumnya sudah ada.
4. Pada Gambar 2.3 D, sama seperti tahapan sebelumnya. *Switch* yang dipilih menjadi rute terpendek akan diberi jalur penghubung berwarna merah.

Sedangkan *switch* yang dibandingkan nilainya akan diberi warna jalur penghubung *orange*.

5. Pada Gambar 2.3 E, prosesnya masih sama dengan tahap sebelumnya. *Switch* akan melakukan perbandingan, setelah itu *switch* yang dipilih akan menjadi *switch* terpilih. *Switch* yang terpilih akan melakukan perbandingan lagi hingga sampai ke *switch* tujuan.
6. Pada Gambar 2.3 F, jalur penghubung berwarna merah yang telah mencapai *switch* tujuan menjelaskan jika rute terpendek dengan *cost* minimum telah ditentukan.

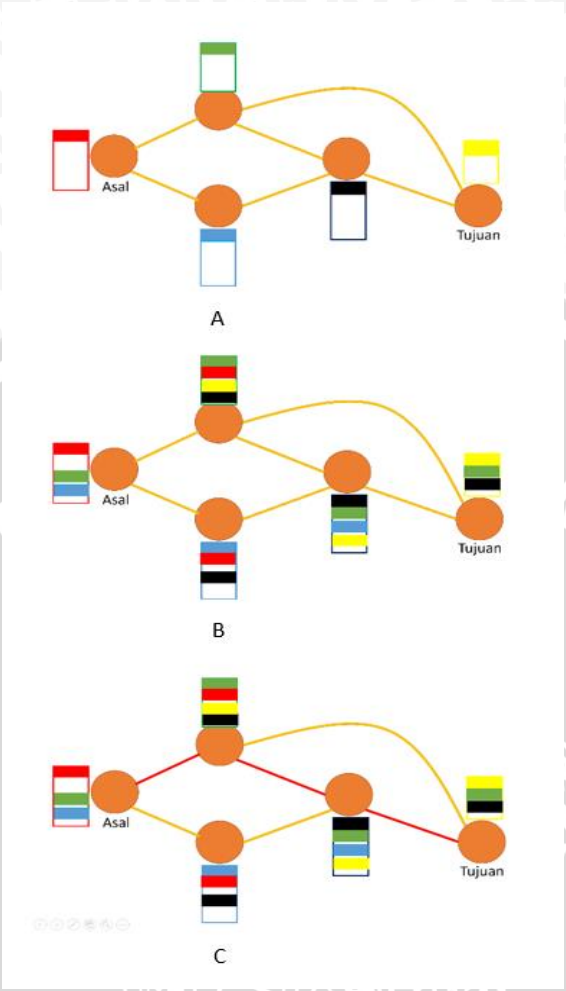
2.2.6.2 Algoritma *Bellman-Ford*

Algoritma *Bellman-Ford* memiliki struktur dasar menyerupai Algoritma *Dijkstra*, akan tetapi Algoritma *Bellman-Ford* secara sederhana akan melakukan pengecekan terhadap semua *router* (Michell Setyawati, 2011). Algoritma *Bellman-Ford* disebut juga dengan algoritma *distance vector*.

Algoritma *Bellman-Ford* bekerja dengan mempublikasikan tabel *routing* yang dimiliki ke tetangga. Apabila terdapat perubahan tabel *routing*, akan dilakukan *update* antar *router* yang saling berhubungan saat terjadi perubahan. Mekanisme *routing* Algoritma *Bellman-Ford* adalah dengan membiarkan setiap *router* mempunyai tabel *routing* dan sehingga dapat diketahui rute ke setiap *switch* dan lintasan yang dipakai menuju tujuan tersebut. Proses *update distance vector* pada masing-masing *router* dilakukan secara iteratif ketika terdapat perubahan nilai pada *link local* dan ketika terdapat pesan untuk *update distance vector* dari tetangga.

$$d_x(y) = min\{c(x,v) + d_v(y)\} \quad (2.2)$$

Berdasarkan persamaan 2.2, $d_x(y)$ adalah jarak dari *switch* x ke y. $c(x,v)$ adalah *cost* dari *switch* x ke *switch* v. Sedangkan $d_v(y)$ adalah jarak v ke y. Jadi, Algoritma *Bellman-Ford* akan melakukan pencarian rute terpendek dengan menghitung *cost* minimum dari *switch* awal ke tetangganya ditambah jarak tetangganya menuju *switch* tujuan. Dari persamaan tersebut, menjadikan Algoritma *Bellman-Ford* memiliki mekanisme pertukaran tabel *routing* dan melakukan perhitungan ke segala kemungkinan jarak.



Gambar 2.4 Pencarian rute Algoritma *Bellman-Ford*

Berdasarkan Gambar 2.4 di atas terdapat beberapa tahapan pencarian. Berikut adalah penjelasan dari beberapa tahapannya.

1. Pada Gambar 2.4 A, masing-masing *switch* akan memiliki tabel *routing*. Tabel *routing* ini berisi seluruh *switch* dengan *cost* yang dimiliki.
2. Pada Gambar 2.4 B, masing-masing *switch* akan melakukan pertukaran tabel *routing* antar tentangga. Setelah proses pertukaran tabel *routing*, akan dilakukan perbandingan nilai yang didapat dengan nilai yang sudah ada pada tabel *routing*. Apabila terdapat *cost* yang lebih kecil maka, nilai sebelumnya akan di-*update*.
3. Pada Gambar 2.4 C, apabila sudah tidak terjadi *update* nilai *cost* maka akan dilakukan pemilihan jalur terpendek. *Switch* yang diberi jalur penghubung warna merah adalah *switch* yang akan terpilih menjadi rute terpendek.

Tabel 2.2 Perbedaan Algoritma

Djikstra	Bellman-Ford
Tidak *support* nilai *cost* negatif	*Support* nilai *cost* negatif
Vertex asal mengetahui keseluruhan informasi global dalam topologi jaringan sampai *vertex* tujuan	Setiap *vertex* dalam topologi hanya memperdulikan informasi dalam tabel *routing*-nya sendiri dan tetangganya
Kompleksitas dari n node dan E *link* adalah O(nE)	Kompleksitas bervariasi
Tidak dapat terjadi *error loop*	Dapat terjadi *error loop* (*counting to infinity*)
Algoritma melakukan iterasi sesuai *path* sampai mendapatkan *node* tujuan dengan jarak minimum	Iterasi algoritma dilakukan setiap ada perubahan pada *link* tetangga atau penerimaan *message update* DV

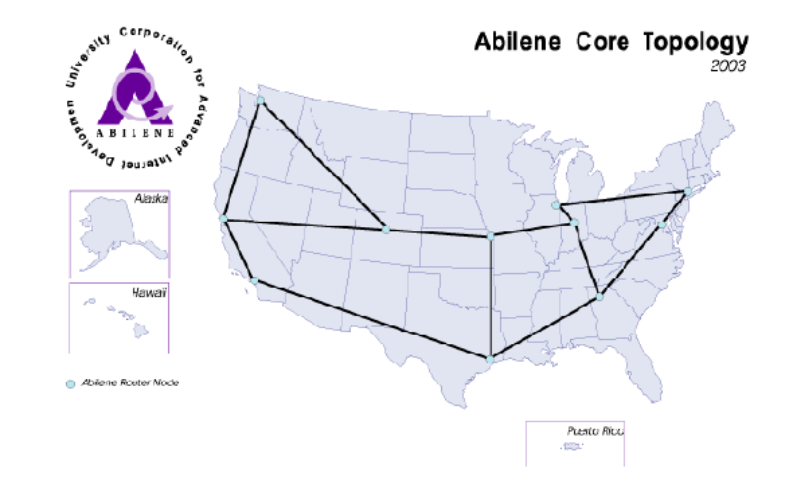
Berdasarkan Tabel 2.2 diatas, terdapat 5 perbedaan Algoritma *Bellman-Ford* dan Algoritma *Dijkstra* secara teori. Untuk mekanisme pencarian Algoritma *Bellman-Ford* akan melakukan pencarian dengan menggunakan informasi dalam tabel *routing*-nya sendiri dan tetangganya, sedangkan Algoritma *Dijkstra* mengetahui keseluruhan informasi global dalam topologi jaringan sampai tujuan.

2.2.7 Topologi *Abilene*

Topologi *Abilene* adalah topologi *backbone* performa tinggi yang dibuat pada akhir 1990-an. Pada tahun 2007 topologi *Abilene* sudah pensiun, jaringan tersebut ditingkatkan dan dikenal sebagai "*Internet2 Network*" (Jehn-Ruey Jiang, 2014). Topologi *Abilene* merupakan implementasi rute kereta yang sudah ada di Amerika Serikat. Pada Gambar 2.5 terdapat Gambar topologi *Abilene*. Nama *Abilene* sendiri berasal dari salah satu stasiun yang ada di Kansas.

Pembangunan topologi *Abilene* memiliki tujuan untuk menciptakan konektivitas antar simpul hingga mencapai 10 Gb/s pada akhir tahun 2006. Lebih dari 230 lembaga anggota berpartisipasi di *Abilene*, sebagian besar universitas dan beberapa lembaga perusahaan dan afiliasi di semua negara bagian AS.

Abilene merupakan *network backbone* berkecepatan tinggi dengan teknologi canggih yang belum tersedia secara umum pada skala *backbone* jaringan nasional. *Abilene* adalah jaringan pribadi yang digunakan untuk pendidikan dan penelitian, tetapi tidak sepenuhnya terisolasi pada hal tersebut, karena anggotanya biasanya memberikan akses alternatif ke banyak sumber daya mereka melalui Internet *public*.



Gambar 2.5 Topologi *Abilene*
Sumber: Jehn-Ruey Jiang (2014)

BAB 3 METODOLOGI

Pada bagian ini akan dijelaskan tentang studi literature, analisis kebutuhan, kebutuhan fungsional, kebutuhan non-fungsional, perancangan algoritma yang didalamnya terdapat perancangan Algoritma *Dijkstra* dan perancangan yang akan dibagi menjadi 2 yaitu perancangan algoritma dan perancangan topologi. Akan dijelaskan juga implementasi, pengujian, analisis, dan kesimpulan yang dilakukan dalam pengerjaan tugas akhir.

3.1 Studi literatur

Studi literatur mempelajari mengenai penjelasan dasar teori yang digunakan untuk menunjang penulisan skripsi. Teori-teori pendukung tersebut diperoleh dari buku, artikel, jurnal, *e-book*, dan dokumentasi project. Teori-teori pendukung tersebut meliputi:

1. Pemodelan Jaringan

- Struktur pemodelan SDN
  Struktur ini digunakan sebagai patokan untuk membuat pemodelan jaringan yang terdapat di SDN, komponen apa saja yang dibutuhkan dalam pengerjaan.
- Cara kerja sistem terdistribusi
  Cara kerja yang digunakan adalah bagaimana cara mengatur atau men*Setting switch* pada jaringan agar dapat saling berbagi informasi dan berfungsi seperti yang diinginkan.

2. Bahasa *python*

- Pemrograman *pyton*
  Python digunakan sebagai bahasa pemprograman pada *controller* Pyretic sehingga dapat di simulasikan. Program yang akan dibut adalah Algoritma *Dijkstra* dan *Bellman-Ford*.

3.2 Analisis kebutuhan

Analisa kebutuan adalah digunakan secara umum tentang kebutuhan yang ada pada penelitian. Yang terdiri dari kebutuhan perangkat dan variabel yang diteliti. Analisa kebutuhan meliputi:

3.2.1 Kebutuhan fungsional

Analisis kebutuhan fungsional dilakukan untuk memberikan gambaran mengenai kemampuan dari suatu sistem. Berikut adalah kebutuhan fungsional suatu sistem:

- Sistem dapat menentukan jalur terpendek dari sejumlah *switch* yang ditentukan.
- Kedua algoritma dapat menghasilkan nilai yang dapat diukur sebagai parameter perbandingan.

3.2.2 Kebutuhan non-fungsional

Perangkat lunak yang dibutuhkan untuk pengembangan sistem adalah sebagai berikut :

- Sistem Operasi, sistem operasi yang digunakan untuk menjalankan perangkat lunak adalah Ubuntu 14.04 LTS untuk Laptop.
- Pyretic digunakan sebagai *controller*. Bahasa pemprogram pada Pyretic menggunakan bahasaa pemprogaman *python*.
- Simulator yang digunakan adalah mininet dengan menggunakan miniedit sebagai virtualisasinya.
- Protokol yang digunakan adalah *Openflow*.

3.3 Perancangan

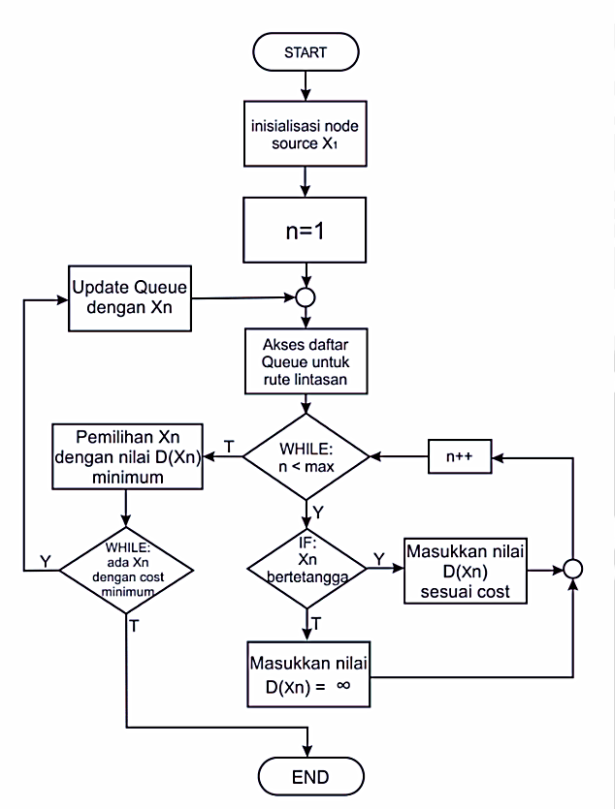
Subbab perancangan ini akan menjelaskan mekanisme perancangan sistem. Pada perancangan ini akan dibagi menjadi 2 subbab lagi yaitu perancangan algoritma dan perancangan topologi. Perancangan algoritma berisi penjelasan alur pencarian rute terpendek pada Algoritma *Dijkstra* dan Algoritma *Bellman-Ford*. Sedangkan, Perancangan topologi ini berisi topologi yang akan digunakan dalam melakukan implementasi. Perancangan sangat diperlukan untuk mempermudah proses pendesinan sistem dan tahapan pengimplementasian.

3.3.1 Perancangan algoritma

Perancangan algoritma berisi penjelasan alur pencarian rute terpendek pada masing-masing algoritma. Pada penelitian ini terdapat 2 algoritma yang akan diuji yaitu Algoritma *Dijkstra* dan *Bellman-Ford*. Masing-masing algoritma memiliki mekanisme pencarian rute yang berbeda. Hasil dari proses perancangan ini akan dibuat program dengan *extensi* file py, sehingga dapat dijalankan pada sebuah *controller*.

3.3.1.1 Perancangan Algoritma *Dijkstra*

Perancangan Algoritma *Dijkstra* akan menjelaskan mekanisme rute pencarian pada Algoritma *Dijkstra*. Algoritma *Dijkstra* memiliki ciri khas pencarian dengan membandingkan nilai *cost* dengan tetangga lalu melakukan perhitungan *cost* pada setiap jalurnya sebelum menentukan rute terpendeknya.

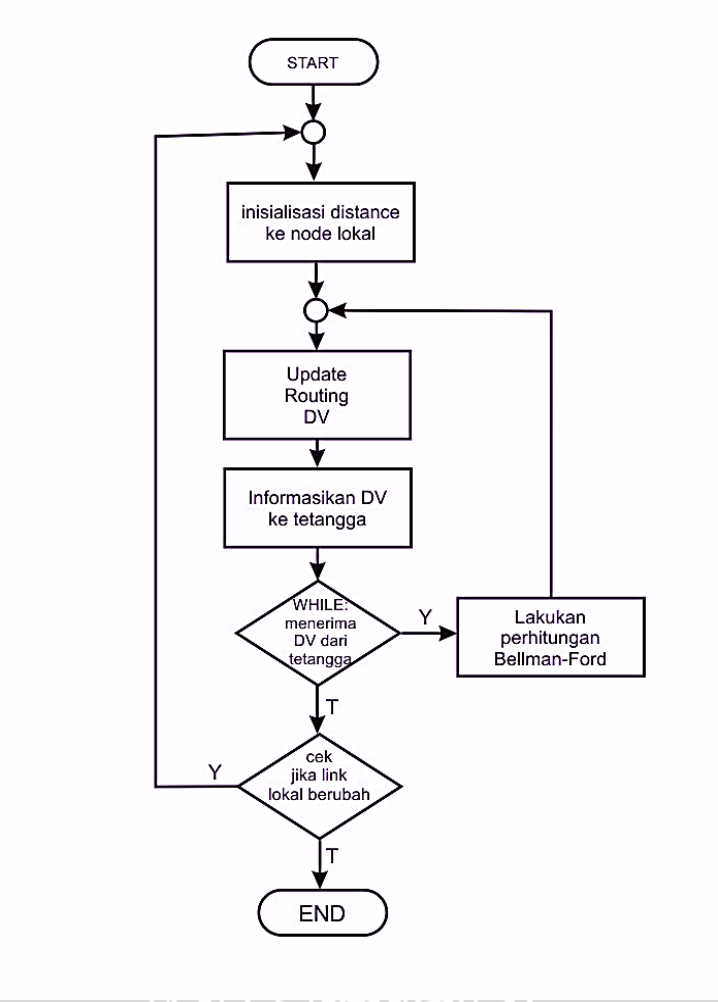


Gambar 3.1 Flowchart Algoritma *Dijkstra*

Berdasarkan Gambar 3.1 di atas mekanisme pencarian Algoritma *Dijkstra*, tahap pertama menunjuk satu *switch* yang berperan sebagai *switch* asal. Setelah itu melakukan pengecekan apabila *switch* tersebut bertetangga akan diberi nilai *cost*. Sedangkan apabila tidak bertetangga langsung akan diberi nilai *infinite*. Proses tersebut akan berulang hingga tidak ada *switch* yang akan dibandingkan. Bila proses pada *switch* asal selesai maka akan dilakukan pemilihan jalur berdasarkan *cost* minimum. *Switch* yang terpilih menjadi rute selanjutnya akan mengantikan tugas dari *switch* asal untuk mencari rute selanjutnya dengan mekanisme yang sama.

3.3.1.2 Perancangan Algoritma *Bellman-Ford*

Perancangan Algoritma *Bellman-Ford* akan menjelaskan mekanisme rute pencarian pada Algoritma *Bellman-Ford*. Algoritma *Bellman-Ford* memiliki ciri khas pencarian dengan menanyakan *cost* tetangganya, kemudian saling bertukar informasi *cost.* Setelah itu mulai menghitung *cost* pada setiap jalurnya sebelum menentukan rute terpendeknya. Apabila terdapat *update* maka antar *switch* akan saling memberi informasi. Pada proses pemberian informasi ini sering terjadi perulangan antar *switch*.



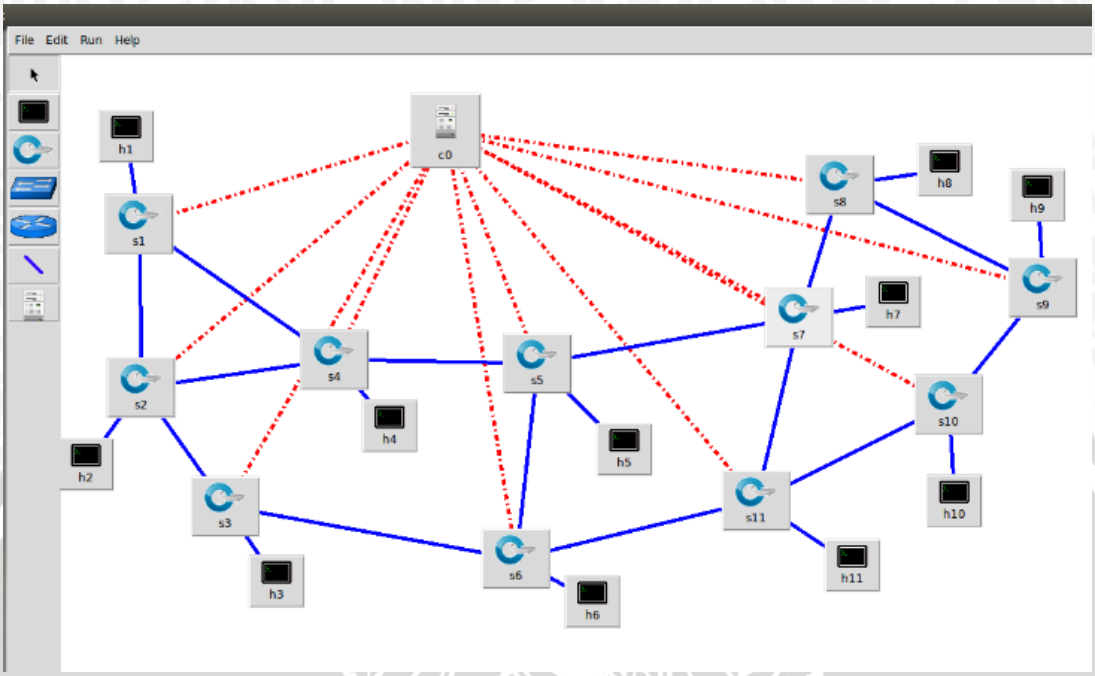
Gambar 3.2 Flowchart Algoritma *Bellman-Ford*

Berdasarkan Gambar 3.2 di atas mekanisme pencarian Algoritma *Bellman-Ford* adalah dengan membuat tabel *routing switch local* yang berisi *cost* dari *switch* x ke *switch* tetangganya. Melakukan *update*, jika terdapat perubahan *cost*. Setelah itu tabel *routing* yang ter-*update* akan dilakukan pertukaran tabel *routing* antar tentangga. Apabila setelah melakukan pertukaran tidak terjadi perubahan maka akan dilakukan pemilihan rute terpendek dengan *cost* minimum. Sedangkan apabila terjadi perubahan pada *link local* maka akan kembali ke proses yang awal yaitu membuat tabel *routing* pada *switch local*.

3.3.2 Perancangan topologi

Perancangan topologi ini berisi topologi yang akan digunakan dalam melakukan implementasi. Perancangan topologi ini akan dibuat dengan menggunakan mininet. Mininet adalah suatu *software* simulator jaringan *open source* yang mendukung protokol *Openflow* untuk arsitektur SDN. Agar mudah melakukan perancangan maka diperlukan miniedit sebagai virtualisasi gambar dari mininet. Pada perancangan ini menggunakan *controller* Pyretic. Pyretic

menggunakan bahasa pemprograman *python,* sehingga file *extensi* yang nantinya di-*upload* pada *controller* adalah py.



Gambar 3.3 *Design* topologi *Abilene* pada miniedit

Berdasarkan Gambar 3.3 di atas perancangan topologi akan menggunakan topologi *Abilene*, yang telah digunakan pada Negara Amerika Serikat. Topologi ini digunakan karena topologi ini memiliki kemiripan dengan wilayahnya. Pada perancangan ini setiap *switch* akan memiliki masing-masing satu *host*. Jumlah *switch* yang digunakan maksimal 11.

3.4 Implementasi

Implementasi sistem dilakukan dengan mengacu pada perancangan sistem yang telah dibuat. Adapun implementasi sistem sebagai berikut.

1. Melakukan instalasi *mininet* dan miniedit dengan sistem operasi *Linux*
2. Melakukan instalasi *controller* Pyretic
3. Membuat program Algoritma *Dijkstra* dan *Bellman-Ford* sebagai mekanisme *routing* jaringan.
4. Membuat topologi *Abilene* dengan menggunakan 11 *switch*, masing-masing *switch* memiliki 1 *host*.
5. Melakukan *routing* pada masing-masing algoritma.
6. Mengetahui hasil analisis sistem melalui pencarian rute.
7. Melakukan perbandingan dari analisis hasil masing-masing algoritma.

3.5 Pengujian

Pengujian sistem pada penelitian ini dilakukan agar dapat menunjukan bahwa sistem ini telah mampu bekerja sesuai dengan spesifikasi yang melandasarinya. Pengujian ini membandingkan algoritma *routing Dijkstra* dan *Bellman-Ford* dengan menggunakan *Software Defined Network*.

Pengujian akan dilakukan secara bergantian antar algoritma, dengan menggunakan topologi yang sama yaitu topologi *Abilene*. Pada topologi *Abilene* jumlah *switch* maksimal 11 dengan masing-masing 1 *host.* Pengujian akan mengamati pemilihan jalur, *bandwidth, throughput, latency* (*delay*), waktu eksekusi program, dan rute terpendek.

3.6 Analisis hasil

Dari konsep perancangan, implementasi dan pengujian terhadap sistem dengan menggunakan *software defined network* didapatkan hasil berupa pilihan jalur, *bandwidth, throughput, latency* (*delay*), waktu eksekusi program, dan rute terpendek. Hasil dari pengujian masing-masing algoritma akan dilakukan analisis dan perbandingan. Algoritma yang digunakan dalam pengujian adalah Algoritma *Dijkstra* dan Algoritma *Bellman-Ford*. Algoritma ini dipilih karena metode pencarian.

Analisis hasil yang didapatkan akan menjadi referensi dalam memilih *routing* algoritma. *Routing* algoritma diperlukan untuk mempercepat pengiriman paket. Masing-masing algoritma memiliki mekanisme pencarian rute yang khas sehingga perlu dilakukannya analisis perbandingan algoritma *routing* dengan menggunakan *Software Defined Network*.

3.7 Kesimpulan

Pada tahap ini merupakan tahap yang dilakukan setelah tahap-tahap selanjutnya terpenuhi yaitu tahap studi literatur, tahap analisis kebutuhan sistem, tahap perancangan sistem, tahap implementasi, tahap pengujian dan analisis hasil. Penarikan kesimpulan diambil dari hasil tahap pengujian dan analisis hasil, selain itu pada tahap ini juga ditambahkan saran sebagai hasil akhir yang diharapkan dapat digunakan sebagai penelitian selanjutnya.

BAB 4 IMPLEMENTASI

Pada bab ini akan dijelaskan tentang implementasi yang dilakukan dalam pengerjaan tugas akhir. Implementai akan dibagi menjadi tiga pembahasan yaitu implementasi algoritma, mplementasi file program, dan implemmentasi program pada *software defined network*.

4.1 Implementasi Algoritma

Implementasi algoritma akan menjelaskan program Algoritma *Dijkstra* dan *Bellman-Ford* pada *software defined network*. Masing-masing algoritma memiliki proses pencarian yang berbeda. Pada Algoritma *Dijkstra* memiliki mekanisme pencarian dengan menghitung seluruh *cost* yang berhubungan dengan *link* pada setiap rute untuk mendapatkan metrik rute-rute yang terhubung, sedangkan tidak *Bellman-Ford*.

4.1.1 Implementasi Algoritma Djikstra

Implementasi Algoritma *Dijkstra* akan membahas tentang Algoritma *Dijkstra* untuk diimplementasikan ke dalam sebuah program. Program algoritma ini dibuat berdasarkan persamaan 2.1 dan Gambar 3.1 *flowchart* Algoritma *Dijkstra*.

Tabel 4.1 *Source code* Algoritma *Dijkstra*

```
1.  ------------
2.   D = {}
3.   for v in Q:
4.      D[v] = Q[v]
5.      print "Q : ", Q
6.      if v == None or not dist.has_key(int(v,10)) :
7.  break
8.      for w in dist[int(v,10)]:
9.          vwlength = D[v]+dist[int(v,10)][w]
10.         if str(w) in D:
11.             if vwlength < D[str(w)]:
12.                 raise ValueError
13.         elif str(w) not in Q or vwlength < Q[str(w)]:
14.              Q[str(w)] = vwlength
15.              pred[w] = int(v,10)
16. print ""
17. print "Total semua cost <<switch:cost>> : "
18. print D
19. print "Jalur <<dest:pred>> : "
20. print pred
21. print "Waktu Eksekusi <<detik>>: "
22. print time.time() - theClock0
23.     return D,pred
24. --------------------
```

Berdasarkan Tabel 4.1 di atas maka dapat diketahui alur *source code* Algoritma *Dijkstra*. Berikut penjelasan gambar di atas.

- Baris 2, inisialisasi dictionary D untuk jarak.
- Baris 3-7, data Queue dimasukan pada *dictionary* D.
- Baris 8, mengecek apakah topologi sudah memiliki jarak.
- Baris 9, mencari jarak terpendek atau Algoritma *Dijkstra*.
- Baris 11, nilai w (tetangga) di ubah ke string lalu disimpan di D.
- Baris 12, pengecekan *error* dalam pemasukan *dictionary*.
- Baris 13, pengecekan nilai w
- Baris 14-15, *update* nilai Queue.
- Baris 17-18, melakukan pencetakan total *cost*.
- Baris 19-20, mencatat dan mencetak jalur yang akan dilalui.
- Baris 21, mencetak waktu eksekusi
- Baris 23, *return* nilai D, Pred.

4.1.2 Implementasi Algoritma Belman-Ford

Implementasi Algoritma *Bellman-Ford* akan membahas Algoritma *Bellman-Ford* jika diimplementasikan ke dalam sebuah program. Program algoritma ini dibuat berdasarkan persamaan 2.2 dan Gambar 3.2 *flowchart* Algoritma *Bellman-Ford*.

Tabel 4.2 Source code Algoritma *Bellman-Ford*

```
1.  -------------
2.  for k in G.edge:
3.     predIndic[k] = huge
4.  sameTable = False
5.  while not sameTable:
6.      sameIndic = 0
7.      for x in G.edge:
8.          for y in G.edge:
9.              for z in G.edge:
10.                 if G.has_edge(x,y) and
11.                    (routeTable[x][x][z] <
12.                     routeTable[y][x][z]):
13.
14.                     routeTable[y][x][z] =
15.                     copy.deepcopy
16.                     (routeTable[x][x][z])
17.
18.                     routeTable[y][y][z] =
19.                     min(routeTable[y][y][z],
20.                     routeTable[y][y][x]+
21.                     routeTable[y][x][z])
22.
23.                     sameIndic = sameIndic + 1
24.      print "<<<<<<UPDATE>>>>>>"
25.      for RT in G.edge:
26.          print "table node",RT,": "
```

Tabel 4.2 (Lanjutan)

```
27.         for RT2 in G.edge:
28.             print RT2," to ",routeTable[RT][RT2]
29.             for RT3 in G.edge:
30.                     if RT3 == self.srcswitch and
31.                        (routeTable[RT][RT2][RT3] +
32.                        routeTable2[RT][RT][RT2]
33.                         <predIndic[RT] and (RT != RT2):
34.                         pred[RT] = RT2
35.                         predIndic[RT]=copy.deepcopy
36.                         (routeTable[RT][RT2][RT3] +
37.                          routeTable2[RT][RT][RT2])
38.                     if RT == self.srcswitch and RT3 ==
39.                        self.dstswitch and RT == RT2:
40.                        totalcost =
41.                        routeTable[RT][RT2][RT3]
42
43.             print " "
44.         if sameIndic == 0: sameTable = True
45. print pred
46. print "Waktu Eksekusi: ",time.time() - theClock0
47. return pred,totalcost
48. ----------------------
```

Berdasarkan Tabel 4.2 di atas maka dapat diketahui alur *source code* algoritma. Berikut penjelasan gambar di atas.

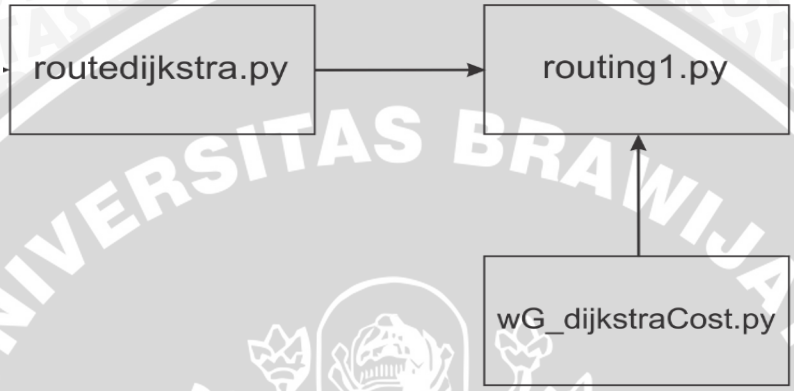
- Baris 2-3, nilai awal yang besar agar dapat dibandingakan dengan nilai yang lebih kecil
- Baris 4, indikator penyamaan *update* tabel *routing.*
- Baris 5, kondisi penyamaan *update* tabel *routing.*
- Baris 7-23, pengecekan seluruh tabel dan melakukan *update* (Algoritma *Bellman-Ford*).
- Baris 25-41, *monitoring update* tabel *routing.*
- Baris 30-37, *update* nilai pred (jalur).
- Baris 38-41, menghitung nilai *cost.*
- Baris 44, jika indikator same indic =0 sama maka tabel *routing* benar atau sudah sama.
- Baris 45-47, cetak jalur, waktu eksekusi, dan return nilai pred dan total *cost.*

4.2 Implementasi file program

Pada subbab ini akan menjelaskan alur file program. Pada masing-masing program terdapat beberapa file yang nantinya file ini akan diimpor ke file utama. Terdapat 2 program yaitu program *Dijkstra* dan *Bellman-Ford*. Pemecahan program menjadi beberapa file untuk memudahkan dalam melakukan perbaikan ketika *output* yang dihasilkan berbeda dengan hasil yang diharapkan.

4.2.1 File program *Dijkstra*

Untuk menjalankan program Algoritma *Dijkstra* dibutuhkan 3 file yang di dalamnya berisi fungsi-fungsi. Fungsi-fungsi inilah yang nantinya akan diimpor sehingga program *Dijkstra* dapat dijalankan. Pada pembuatan program Algoritma *Dijkstra* digunakan array 2 dimensi untuk melakukan pengenalan jalur berdasarkan *switch* awal dan *switch* tujuan. Jadi terdapat 2 slot pada array 2 dimensi. Slot yang pertama akan *switch* awal dan *switch* tujuan.



Gambar 4.1 Alur impor program *Dijkstra*

Berdasarkan Gambar 4.1 di atas terdapat 3 file program. File routeDijkstra.py dan wG_DijkstraCost.py yang berisi rumus dari algoritma djikstra dan pengaturan nilai *cost*. Sedangkan file program *routing*1.py berisi mekanisme *routing* dari IP address dan file ini adalah file utama yang nantinya file akan di-*upload* pada *controller*.

Tabel 4.3 *Source code* impor file route*Dijkstra*.py

```
1.  from multiprocessing import Lock
2.  from Pyretic.lib.query import *
3.  from Pyretic.core import *
4.  from Pyretic.lib.corelib import *
5.  from Pyretic.lib.std import *
6.  ----------
```

Berdasarkan Tabel 4.3 di atas maka dapat diketahui alur *source code* impor file route*Dijkstra*.py. Berikut penjelasan gambar di atas.

- Baris 1, *multiproses* untuk membuat *thread.*
- Baris 2-5, impor *library* untuk menjalankan Pyretic.

Tabel 4.4 *Source code* impor file wG_*DijkstraCost*.py

```
1.  from Pyretic.lib.query import *
2.  from Pyretic.core import *
3.  from Pyretic.lib.corelib import *
4.  from Pyretic.lib.std import *
5.  ----------
```

Berdasarkan Tabel 4.4, maka dapat diketahui alur *source code* impor file wG_*DijkstraCost*.py Baris 1-4, impor *Library* untuk menjalankan Pyretic.

Tabel 4.5 *Source code* impor file *routing*1.py

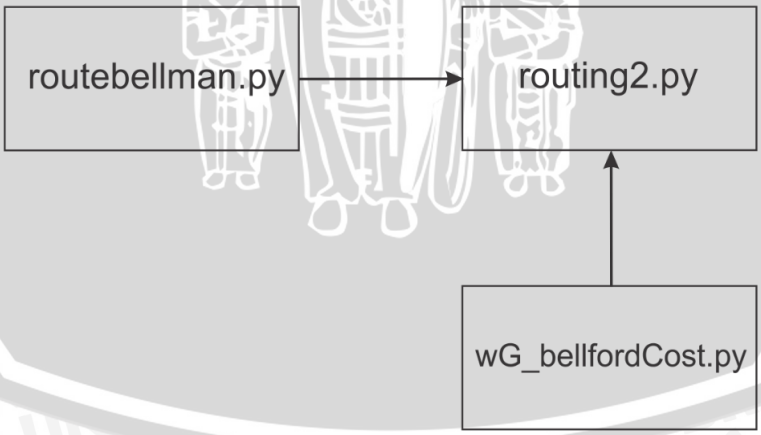
1.	`from Pyretic.modules.route`*`Dijkstra`* `import *`
2.	`from Pyretic.lib.corelib import *`
3.	`from Pyretic.lib.std import *`
4.	`from Pyretic.lib.query import *`
5.	`from Pyretic.modules.wG_`*`DijkstraCost`* `import *`
6.	`----------`

Berdasarkan Tabel 4.5 di atas maka dapat diketahui alur *source code* impor file *routing*1.py. Berikut penjelasan gambar di atas.

- Baris 1, file *routing*1.py akan melakukan impor file route*Dijkstra*.py.
- Baris 2-4, impor *library* untuk menjalankan Pyretic.
- Baris 5, file *routing*1.py akan melakukan impor file wG_*DijkstraCost*.py.

4.2.2 File program *Bellman-Ford*

Untuk menjalankan program Algoritma *Bellman-Ford* dibutuhkan 3 file yang didalamnya berisi beberapa fungsi-fungsi. Fungsi-fungsi inilah yang nantinya akan diimpor sehingga program *Bellman-Ford* dapat dijalankan. Pada pembuatan program Algoritma *Bellman-Ford* digunakan array 3 dimensi. Array 3 dimensi digunakan untuk melakukan pengenalan jalur. Pada array 3 dimensi terdapat 3 slot, di Algoritma *Bellman-Ford* ada mekanisme pengiriman tebel *routing* antar tetangga maka slot pertama akan diisi dengan identitas kepemilikan tabel *routing*, lalu slot kedua akan berisi *switch* asal dan slot terakhir akan berisi *switch* tujuan.



Gambar 4.2 Alur impor program *Bellman-Ford*

Berdasarkan Gambar 4.2 di atas terdapat 3 file program. File routebellman.py dan wG_bellford*Cost*.py yang berisi rumus dari Algoritma Djikstra dan pengaturan nilai *cost*. Sedangkan file program *routing*2.py berisi mekanisme

routing dari IP address dan file ini adalah file utama yang nantinya file akan di-*upload* pada *controller*.

Tabel 4.6 *Source code* impor file routebellman.py

```
1.  from multiprocessing import Lock
2.  from Pyretic.lib.query import *
3.  from Pyretic.core import *
4.  from Pyretic.lib.corelib import *
5.  from Pyretic.lib.std import *
6.  ----------
```

Berdasarkan Tabel 4.6 di atas maka dapat diketahui alur *source code* impor file routebellman.py. Berikut penjelasan gambar di atas.

- Baris 1, *multiproses* untuk membuat *thread.*
- Baris 2-5, impor *library* untuk menjalankan Pyretic.

Tabel 4.7 *Source code* impor file wG_bellford*Cost*.py

```
1.  from Pyretic.lib.query import *
2.  from Pyretic.core import *
3.  from Pyretic.lib.corelib import *
4.  from Pyretic.lib.std import *
5.  ----------
```

Berdasarkan Tabel 4.7 di atas maka dapat diketahui alur *source code* impor file wG_bellfordCost.py. Baris 1-4, impor *library* untuk menjalankan Pyretic.

Tabel 4.8 *Source code* impor file *routing*2.py

```
1.  from Pyretic.modules.routebellman import *
2.  from Pyretic.lib.corelib import *
3.  from Pyretic.lib.std import *
4.  from Pyretic.lib.query import *
5.  from Pyretic.modules.wG_bellfordCost import *
6.  ----------
```

Berdasarkan Tabel 4.8 di atas maka dapat diketahui alur *source code* impor file *routing*2.py. Berikut penjelasan gambar di atas.

- Baris 1, file *routing*2.py akan melakukan impor file routebellman.py
- Baris 2-4, impor *library* untuk menjalankan Pyretic
- Baris 5, file *routing*2.py akan melakukan impor file wG_bellfordCost.py

4.3 Implementasi sistem

Implementasi sistem akan menjelaskan tahapan memulai sistem hingga sistem dapat berjalan. Tahapan implementasi sistem antara lain proses *running* simulator mininet dan *running* virtualisasi miniedit, pembuatan topologi dan *running controller* Pyretic.

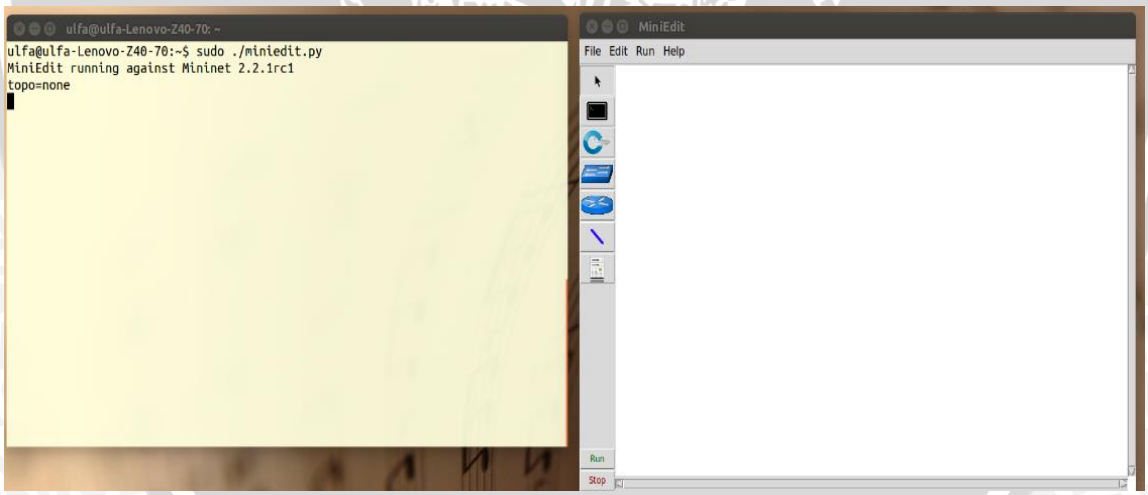
4.3.1 *Running* mininet dan miniedit

Untuk melakukan perancangan topologi, maka melakukan *running* simulator mininet. Untuk mempermudah membuat topologi setelah melakukan *running* simulator, maka dilakukan *running* miniedit. Dalam implementasi sistem, *running* mininet dan miniedit memiliki peranan awal yang penting.

```
ulfa@ulfa-Lenovo-Z40-70:~$ sudo mn -c
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes
killall controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-openflowd
 ovs-controller udpbwtest mnexec ivs 2> /dev/null
killall -9 controller ofprotocol ofdatapath ping nox_core lt-nox_core ovs-openfl
owd ovs-controller udpbwtest mnexec ivs 2> /dev/null
pkill -9 -f "sudo mnexec"
*** Removing junk from /tmp
rm -f /tmp/vconn* /tmp/vlogs* /tmp/*.out /tmp/*.log
*** Removing old X11 tunnels
*** Removing excess kernel datapaths
ps ax | egrep -o 'dp[0-9]+' | sed 's/dp/nl:/'
***  Removing OVS datapaths
ovs-vsctl --timeout=1 list-br
ovs-vsctl --timeout=1 list-br
*** Removing all links of the pattern foo-ethX
ip link show | egrep -o '([-_.[:alnum:]]+-eth[[:digit:]]+)'
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
```

Gambar 4.3 *Running* mininet pada terminal Ubuntu

Berdasarkan Gambar 4.3 di atas, merupakan gambar ketika mininet dijalankan pada terminal Ubuntu. Mininet adalah sebuah simulator jaringan *open source* yang mendukung protokol *Openflow* untuk arsitektur SDN. Mininet menggunakan konsep dasar virtualisasi untuk membuat suatu sistem.



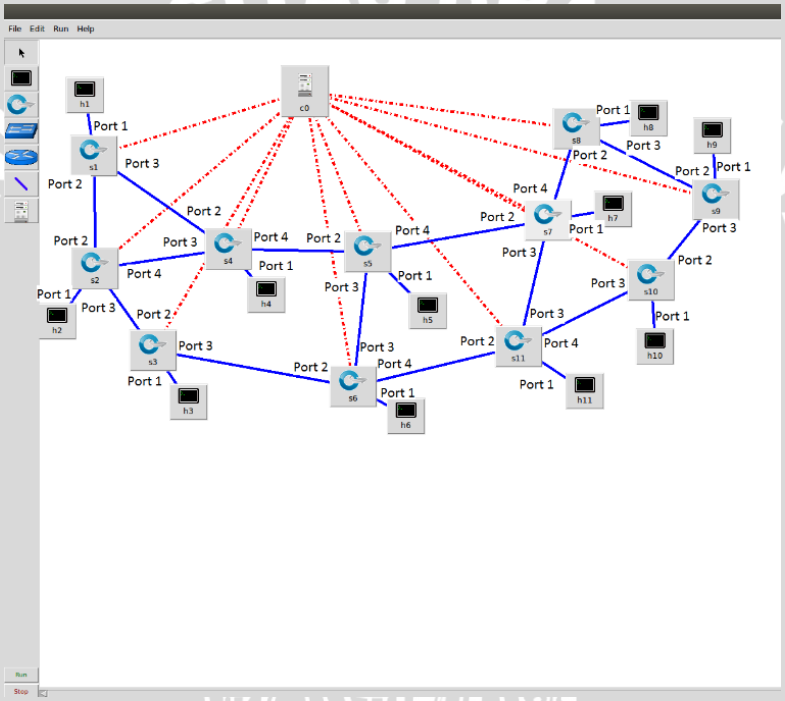
Gambar 4.4 Buka miniedit

Berdasarkan Gambar 4.4 di atas, merupakan gambar ketika miniedit dijalankan. Miniedit pada mininet merupakan visualisasi dari mininet. Pada miniedit terdapat beberapa tool antara lain kursor, *host*, *switch*, legacy *switch*, legency *router*, conector, dan *controller*.

4.3.2 Pembuatan topologi

Setelah miniedit dijalankan, maka proses perancangan topologi dapat dilakukan. Topologi yang digunakan pada penelitian ini adalah topologi *Abilene*. Topologi *Abilene* berasal dari implementasi wilayah yang ada di Amerika Serikat.

Penelitian ini akan digunakan *controller* Pyretic. Tahap menghubungkan *conector* dengan *controller, switch,* dan *host* perlu diperhatikan. Apabila pada proses menghubungkan ketiga komponen tersebut tidak urut maka akan mempengaruhi hasil *output.* Tahapan pemasangan ini hanya berlaku apabila *controller* yang digunakan adalah *controller* Pyretic. Proses menghubungankan *conector* akan dimulai dari *controller* ke *switch* lalu dari *switch* ke *host* dan setelah itu dari *switch* ke *switch* jika terdapat sambungan.

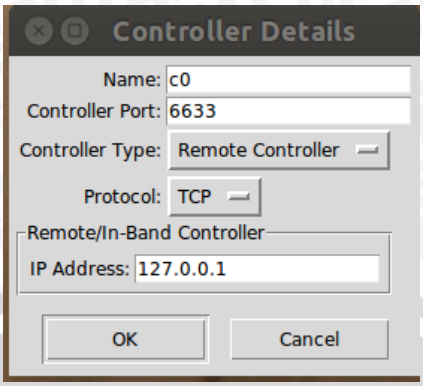


Gambar 4.5 Pembuatan topologi *Abilene*

Berdasarkan Gambar 4.5 di atas, merupakan gambar topologi *Abilene* pada miniedit. Topologi *Abilene* ini akan dibuat dengan menggunakan 11 *switch* yang masing-masing *switch* memiliki 1 *host*. Pada gambar di atas *controller* akan terhubung ke *switch* lalu ke *host*. *Host* akan memiliki *port output* 1 pada *switch*. Proses pemasangan antar komponen akan mempengaruhi pemberian *port output*.

4.3.3 Running *controller* Pyretic

Pyretic adalah *controller* yang mengimplementasikan bahasa python sebagai sistem *runtime* program. Sebelum melakukan running simulator, terlebih dahulu melakukan *Setting controller*.



Gambar 4.6 *Setting controller*

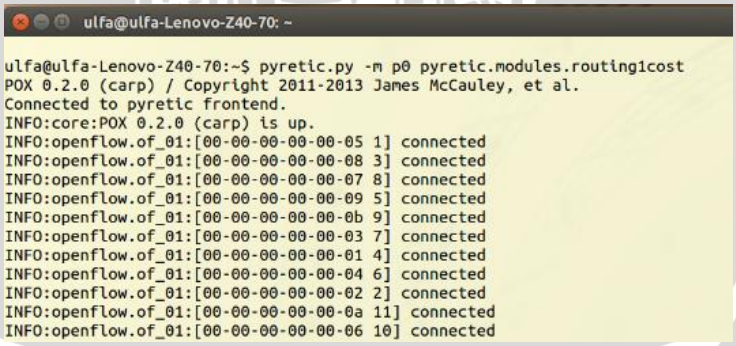
Berdasarkan Gambar 4.6 di atas, untuk melakukan *Setting controller type* dan *controller port* pada Pyretic, caranya mengklik kanan gambar *controller* dan memilih menu properties. *Setting controller* dengan *remote controller* dan *controller port* 6633.

Setelah melakukan *Setting controller type* dan *controller port*, maka file *routing* siap di-*upload* pada *controller*. Dengan sebelumnya klik *icon run* pada miniedit dan lakukan *upload.*

Tabel 4.9 *Source code upload* Pyretic

1.	`Pyretic.py -m p0 Pyretic.modules.nama_file`

Berdasarkan Tabel 4.9 di atas, baris 1 merupakan *source code* untuk melakukan *upload* file pada *controller. Source code* ditulis pada terminal yang ada Ubuntu. Untuk menjalankan Pyretic dibutuhkan buka terminal baru. Setelah menuliskan *source code* pada Tabel 4.9 di atas maka enter.



Gambar 4.7 Ter-*upload* program pada *controller* Pyretic

Berdasarkan Gambar 4.7 di atas, apabila tercetak di terminal *connected* maka program telah berhasil di *upload* pada *controller* Pyretic. Ketika program sudah ter-*upload* maka sistem telah siap dijalankan untuk mencari rute terpendek.

BAB 5 PENGUJIAN DAN ANALISIS

Pada bab ini akan dijelaskan tentang pengujian dan analisis yang dilakukan dalam pengerjaan tugas akhir. Ada 3 pembahasan pada bab ini yang pertama adalah pengujian. Pengujian akan berisi subbab pengujian Algoritma *Dijkstra* dan *Bellman-Ford.* Pembahasan yang kedua berisi analisis dari kedua algoritma yang didapat dari pengujian. Sedangkan untuk pembahasan ketiga akan dilakukan perbandingan dari hasil analisis.

5.1 Pengujian

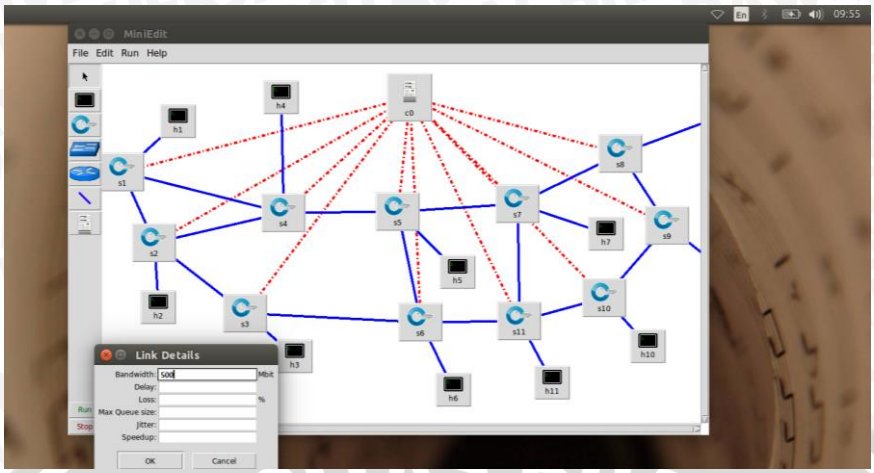
Pada subbab ini akan dilakukan pengujian untuk mempermudah melakukan analisis dan perbandingan Algoritma *Dijkstra* dan *Bellman-Ford* pada *Software Defined Network*. Pengujian akan menggunakan topologi *Abilene*. Pengujian yang akan dilakukan antara lain, pemilihan jalur, *bandwidth, throughput, latency* (*delay*), waktu eksekusi program, dan rute terpendek. Pada pengujian yang akan menjadi *switch* tujuan adalah *switch* dengan IP 10.0.0.1. Nilai *cost* antar *switch* akan ditentukan. Hasil dari setiap pengujian diambil dari pengujian ketika program pertama kali dijalankan, karena ketika program pertama kali dijalankan program akan melakukan dua mekanisme yaitu pencarian tabel *routing dan forwarding*.

Pengujian pemilihan jalur ini bersifat fungsional karena pada pengujian ini hanya akan mengamati mekanisme pemilihan jalur dari masing-masing algoritma. Pada pengujian pemilihan jalur akan terdapat 2 mekanisme pengujian, antara lain.

1. Berdasarkan total nilai *cost* dan jumlah *hop switch* yang dilewati sama
2. Berdasarkan total nilai *cost* sama namun jumlah *hop switch* yang dilewati berbeda.

Pada pengujian pemilihan jalur, masing-masing mekanisme pengujian akan melakukan 3 kali pengujian, dengan menggunakan *switch* awal yang berbeda disetiap pengujian dan memiliki total *cost* 20. Sedangkan pada pengujian *bandwidth, throughput, latency* (*delay*), waktu eksekusi program, dan rute terpendek akan dilakukan pengujian dengan nilai *cost* yang berbeda dengan pengujian pemilihan jalur, total *cost* yang akan digunakan adalah 19.

Untuk pengujian *throughput*dan *latency* (*delay*) digunakan tool iperf dan ping. Hasil pengujian dari *throughput* dan *latency* (*delay*) berupa nilai rata-rata. Pengujian *bandwidth, throughput, latency* (*delay*), waktu eksekusi program, dan rute terpendek akan menggunakan *switch* awal yang sama.



Gambar 5.1 *Setting bandwidth*

Berdasarkan Gambar 5.1 di atas, melakukan *Setting* ukuran *bandwidth*. pengujian *bandwidth* akan didahului dengan melakukan *Setting* ukuran *bandwidth*. *Bandwidth* akan di*Setting* dengan ukuran *bandwidth* kurang dari sama dengan 1000 Mbps. Ukuran ini diambil karena apabila lebih dari 1000 Mbps, *bandwidth* tidak dapat dibatasi oleh simulator. *Bandwidth*yang akan digunakan adalah *bandwidth* dengan ukuran 125, 250, 375, 500, 750, 875, dan 1000 Mbps. Parameter nilai dipilih secara random dengan perbedaan nilai 125 Mbps antar parameter. Karena *bandwitdth* sudah ditentukan maka nilai *bandwidth* akan menjadi parameter untuk pengujian *throughput, latency* (*delay*), waktu eksekusi program, dan rute terpendek.

Setelah melakukan *Setting bandwidth*, selanjutnya melakukan perhitungan nilai *throughput* dengan menggunakan tool iperf. Sebelum pengujian dilakukan maka terlebih dahulu menjalankan terminal yang ada dimasing-masing *host*.

Tabel 5.1 *Source code* iperf

Untuk server	`Iperf -s`
Untuk *client*	`Iperf -c alamat_IPserver -i interval_waktu -t waktu_kirim`

Berdasarkan Tabel 5.1 di atas, *source code* untuk melakukan uji parameter *throughput*. Pada pengujian ini telah ditentukan 1 *switch* yang menjadi server adalah *switch* tujuan dengan IP 10.0.0.1. *Switch* yang pertama dijalankan adalah *switch* server lalu *switch client*. *Switch client* akan dijalankan 3 terlebih dahulu. Ketika 3 *switch* awal sudah mulai berjalan maka 3 *switch client* selanjutnya mulai dijalankan. Hal ini dilakukan agar semua *switch* mendapatkan aliran data, sehingga tidak terjadi *connected failed: no route to host* pada terminal *client*. Nilai *throughput* diambil rata-rata yang tercetak pada terminal *switch client*. Pada pengujian ini digunakan interval waktu 2 dan waktu kirim 10.

Tabel 5.2 *Source code* ping

1	`Ping -c banyaknya_pengirim alamat_IP_switch_tujuan`

Berdasarkan Tabel 5.2, *source code* untuk melakukan uji parameter *latency*. Pada pengujian *latency* ini digunakan tool ping. *Switch* yang menjadi *switch* awal akan melakukan ping ke *switch* tujuan. Banyaknya pengiriman pada pengujian ini adalah 10. Hasil time yang tercetak pada terminal adalah nilai yang di analisis sebagai hasil.

Tabel 5.3 *Source code* waktu eksekusi program

```
1. --------
2. theClock0 = time.time()
3. --------
4. print "Waktu Eksekusi: ",time.time() - theClock0
5. ---------
```

Berdasarkan Tabel 5.3 di atas, merupakan *source code* program waktu eksekusi. Program akan melakukan perhitungan dari awal program dieksekusi. Satuan waktu yang dihasilkan adalah detik. *Source code* program di atas masuk ke dalam file routebellman.py dan route*Dijkstra*.py.

```
Total semua cost <<switch:cost>> :
{'11': 9, '10': 3, '1': 19, '3': 14, '2': 17, '5': 10, '4': 12, '7': 4, '6': 13,
 '9': 0, '8': 1}
Jalur <<dest:pred>> :
{1: 2, 2: 3, 3: 6, 4: 5, 5: 7, 6: 11, 7: 8, 8: 9, 10: 9, 11: 10}
Waktu Eksekusi <<detik>>:
0.000585079193115

==========Kesimpulan==========
Asal  : Node  9
Tujuan: Node  1
pathlist <<switch[port]>>:
9[3] -> 10[3] -> 11[2] -> 6[2] -> 3[2] -> 2[2] -> 1[1]
Total cost:  19
```

Gambar 5.2 Waktu eksekusi dan jalur dengan *cost* minimum

Berdasarkan Gambar 5.2 di atas, merupakan gambar dari program ketika sudah dijalankan pada terminal Ubuntu. Rute terpendek diambil dari perhitungan menggunakan rumus masing-masing algoritma.

5.1.1 Pengujian Algoritma *Dijkstra*

Pengujian Algoritma *Dijkstra* bertujuan untuk mengetahui nilai yang dihasilkan dari beberapa parameter yang telah ditentukan. Hasil dari pengujian akan dibandingkan.

Pengujian pemilihan jalur adalah pengujian yang akan menganalisis pemilihan jalur yang dilewati. Pengujian ini akan menggunakan total *cost* 20 dengan nilai *cost* antar *switch* di-*input* secara manual dan bersifat acak nilainya.

1. Jika *cost* sama dan jumlah *hop* yang dilewati sama. Pada pengujian ini akan diberi 2 rute yang sama-sama memiliki total *cost* yang sama dan *switch* tujuannya sama.

Tabel 5.1 Pengujian pemilihan jalur 1 Algoritma *Dijkstra*

No	*Switch*		Pemilihan Jalur	Jalur yang dilewati	*Cost* antar *switch*	Total *cost*
	Awal	Tujuan				
1.	10	1	10[3]-> 11[3] ->7[2]->5[2]-> 4 [2]-> 1 [1] 10[3]-> 11[2] -> 6[3]-> 5[2] ->4 [2]-> 1 [1]	10[3]-> 11 [3]-> 7[2] -> 5[2]->4 [2]-> 1 [1]	1->2 =3, 1->4=3, 2->3=6, 2->4=7, 3->6=2, 4->5=5, 5->7=4, 6->5=3, 6->11=7, 7->8=5, 7->11=6, 8->9=9, 9->10=8, 10->11=2	20
2.	9	1	9[3]->10[3]-> 11[2]->6[2]-> 3 [2]->2 [2]-> 1[1] 9[3]->10[3]-> 11[2]->6[3]-> 5 [2]->4[2]-> 1[1]	9[3]->10 [3]->11[2] ->6[2]->3 [2]->2 [2] -> 1[1]	1->2 =3, 1->4=3, 2->3=3, 2->4=8, 3->6=7, 4->5=5, 5->7=4, 6->5=5, 6->11=1, 7->8=7, 7->11=2, 8->9=2, 9->10=4, 10->11=2	20
3.	8	1	8[2]->7[4]-> 11[2]->6[2]-> 3[2]->2[2]-> 1 [1] 8[2]->7[2]->5 [3]->6[2]->3 [2]->2[2]->1[1]	8[2]->7[4] ->11[2]-> 6[2]->3[2] ->2[2]->1[1]	1->2 =5, 1->4=5, 2->3=1, 2->4=1, 3->6=2, 4->5=10, 5->7=4, 6->5=5, 6->11=7, 7->8=3, 7->11=2, 8->9=9, 9->10=9, 10->11=8	20

Berdasarkan Tabel 5.1 no 1, dari *switch* awal 10 ke *switch* tujuan 1 memiliki pilihan rute. Rute yang pertama dari *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [3], melalui *switch* 7 dengan *port output* [2], melalui *switch* 5 dengan *port output* [2], melalui *switch* 4 dengan *port output* [2], dan melalui *switch* 1 dengan *port output* [1]. Sedangkan pilihan rute yang kedua dari *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [2], melalui *switch* 6 dengan *port output* [3], melalui *switch* 5 dengan *port output* [2], melalui *switch* 4 dengan *port output* [2], dan melalui *switch* 1 dengan *port output* [1]. Untuk pemilihan dari *switch* 10 ke *switch* 1 adalah rute yang pertama dengan total *cost* 20.

Hasil pengujian dari Tabel 5.1, no 1 terdapat perbedaan nilai *cost* pada *switch* 11 ke 7 dan 11 ke 6. Perbedaan ini yang membuat pemilihan jalur yang dipilih adalah *switch* 11 ke 7 bukan 11 ke 6. Untuk no 2, jalur yang dipilih berdasarkan

input pertama pada tabel *routing*. Dapat dilihat jalur yang dipilih memiliki *switch* dengan nomor kecil. Sedangkan untuk no 3, terdapat perbedaan nilai *cost* pada *switch* 7 ke 11 dan *switch* 7 ke 5. Nilai *cost switch* 7 ke 11 lebih kecil dibanding *switch* 7 ke 5. Sehingga pemilihan jalur dipilih berdasarkan nilai *cost* minimum.

2. Jika *cost* sama dan jumlah *hop* yang dilewati beda. Pada pengujian ini akan diberi 2 rute yang sama-sama memiliki total *cost* yang sama dan *switch* tujuannya sama.

Tabel 5.2 Pengujian pemilihan jalur 2 Algoritma *Dijkstra*

No	*Switch*		Pilihan Jalur	Jalur yang terlewati	*Cost* antar *switch*	Total *Cost*
	Awal	Tujuan				
1.	10	1	10[3]->11[2]->6[2]->3[2]->2[2]->1[1] 10[2]->9[2]->8[2]->7[2]->5[4]->4[2]->1[1]	10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	1->2 =5, 1->4=5, 2->3=6, 2->4=9, 3->6=3, 4->5=3, 5->7=4, 6->5=9, 6->11=4, 7->8=2, 7->11=10, 8->9=4, 9->10=2, 10->11=2	20
2.	9	1	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1] 9[2]->8[2]->7[2]->5[2]->4[2]->1[1]	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	1->2=3, 1->4=3, 2->3=1, 2->4=9, 3->6=8, 4->5=5, 5->7=2, 6->5=9, 6->11=2, 7->8=7, 7->11=10,8->9=3, 9->10=3, 10->11=3	20
3.	8	1	8[2]->7[2]->5[2]->4[2]->1[1] 8[3]->9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	8[2]->7[2]->5[2]->4[2]->1[1]	1->2 =2, 1->4=9, 2->3=3, 2->4=8, 3->6=1, 4->5=2, 5->7=6, 6->5=9, 6->11=4, 7->8=3, 7->11=10, 8->9=1, 9->10=3, 10->11=6	20

Berdasarkan Tabel 5.2 no 1, dari *switch* awal 10 ke *switch* tujuan 1 memiliki pilihan rute. Rute yang pertama dari *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [2], melalui *switch* 6 dengan *port output* [2], melalui *switch* 3 dengan *port output* [2], melalui *switch* 2 dengan *port output* [2], dan melalui *switch* 1 dengan *port output* [1]. Sedangkan, pilihan rute yang kedua dari *switch* 10 dengan *port output* [2], melalui *switch* 9 dengan *port output* [2], melalui *switch* 8 dengan *port output* [2], melalui *switch* 7 dengan *port output* [2], melalui *switch* 5 dengan *port output* [4], melalui *switch* 4 dengan *port output* [2], dan melalui *switch* 1 dengan *port output* [1]. Untuk rute yang dilewati dari *switch* 10 ke *switch* 1 adalah rute yang pertama dengan total *cost* 20.

Hasil pengujian dari 5.2, no 1 dan no 2, jalur yang dipilih berdasarkan *input* pertama pada tabel *routing*. Dapat dilihat pada Tabel 5.2, jalur yang dipilih memiliki jumlah *switch* dengan nomor kecil. Sedangkan no 3, jalur yang dipilih berdasarkan *hop*. Memiliki perbedaan 2 *hop* yang dilewati, maka *hop* yang kecil akan dipilih.

Untuk pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi *program,* dan rute terpendek akan dibagi menjadi 2 kali pengujian. Pengujian pertama akan melihat nilai *bandwidth* dan *throughput*. Pengujian kedua akan melihat nilai *latency*, waktu eksekusi, dan jalur. Sama dengan pengujian pemilihan jalur, *cost* antar *switch* akan di-*input* secara manual dan nilainya bersifat acak. *Switch* awal dan tujuan akan dibuat sama disetiap pengujian.

Tabel 5.3 Pengujian *bandwidth*, *throughput*, latency (*delay*), waktu eksekusi program, dan rute terpendek Algoritma *Dijkstra*

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
1.	9	1	125	25,0	9008	0,00609874725342	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
2.	9	1	250	85,0	8999	0,00052094459 5337	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
3.	9	1	375	120	9008	0,00058507919 3115	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
4.	9	1	500	204	9005	0,00048899650 5737	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
5.	9	1	625	278	9008	0,0006339550 01831	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
6.	9	1	750	400	9009	0,00060915946 9604	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19

Tabel 5.3 (Lanjutan)

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
7.	9	1	875	535	8999	0,00046205520 6299	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
8.	9	1	1000	593	9008	0,00060486793 5181	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19

Dari Tabel 5.3 didapatkan hasil, apabila nilai *bandwidth* mengalami kenaikan maka nilai *throughput* juga akan naik nilainya. Untuk nilai *latency* tidak mengalami perubahan yang signifikan, apabila ada perubahan nilai *bandwidth*. Sedangkan waktu eksekusi program dalam mencari rute dikisaran 0.00048 hingga 0.000633 detik. Sedangkan untuk jalur yang dilewati, semua percobaan melewati rute yang sama. Dari *switch* 9 dengan *port output* [3], melalui *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [2], melalui *switch* 6 dengan *port output* [2], melalui *switch* 3 dengan *port output* [2], melalui 2 *switch* dengan *port output* [2], melalui *switch* 1 dengan *port output* [1]. Dari pemilihan jalur tersebut, memiliki total *cost* 19.

5.1.2 Pengujian Algoritma *Bellman-Ford*

Pengujian Algoritma *Bellman-Ford* bertujuan untuk mengetahui nilai yang dihasilkan dari beberapa parameter yang telah ditentukan. Hasil dari pengujian yang dilakukan dibandingkan untuk di dicari algoritma yang terbaik.

Pengujian pemilihan jalur adalah pengujian yang akan menganalisis pemilihan jalur yang dilewati. Pengujian ini akan menggunakan total *cost* 20 dengan nilai *cost* antar *switch* di-*input* secara manual dan bersifat acak nilainya.

1. Jika *cost* sama dan jumlah *hop* yang dilewati sama. Pada pengujian ini akan diberi 2 rute yang sama-sama memiliki total *cost* yang sama dan *switch* tujuannya sama.

Tabel 5.4 Pemilihan jalur 1 Algoritma *Bellman-Ford*

No	*Switch*		Pemilihan Jalur	Jalur yang terlewati	*Cost* antar *switch*	Total *cost*
	Awal	Tujuan				
1.	10	1	10[3]->11[3]->7 [2]->5[2]->4 [2] ->1 [1]	10[3]->11 [3]-> 7[2]-> 5[2]->4[2]-> 1 [1]	1->2 =3, 1->4=3, 2->3=6, 2->4=7, 3->6=2, 4->5=5, 5->7=4, 6->5=3, 6->11=7, 7->8=5, 7->11=6, 8->9=9, 9->10=8,10->11=2	20
			10[3]-> 11[2]-> 6[3]-> 5[2]->4 [2]-> 1 [1]			
2.	9	1	9[3]->10[3]-> 11[2]->6[2]->3 [2]->2 [2]->1[1]	9[3]->10[3] ->11[2]->6 [2]->3 [2]-> 2 [2]-> 1[1]	1->2 =3, 1->4=3, 2->3=3, 2->4=8, 3->6=7, 4->5=5, 5->7=4, 6->5=5, 6->11=1, 7->8=7, 7->11=2, 8->9=2, 9->10=4,10->11=2	20
			9[3]->10[3]->11 [2]->6[3]->5[2] ->4[2]-> 1[1]			
3.	8	1	8[2]->7[4]->11 [2]->6[2]->3[2] ->2[2]->1 [1]	8[2]->7[4] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	1->2 =5, 1->4=5, 2->3=1, 2->4=1, 3->6=2, 4->5=10, 5->7=4, 6->5=5, 6->11=7, 7->8=3, 7->11=2, 8->9=9, 9->10=9,10->11=8	20
			8[2]->7[2]->5 [3]->6[2]->3[2] ->2[2]->1 [1]			

Berdasarkan Tabel 5.3 no 1, dari *switch* awal 10 ke *switch t*ujuan 1 memiliki pilihan rute. Rute yang pertama dari *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [3], melalui *switch* 7 dengan *port output* [2], melalui *switch 5* dengan *port output* [2], melalui *switch* 4 dengan *port output* [2] melalui *switch* 1 dengan *port output* [1]. Sedangkan pilihan rute yang kedua dari *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [2], melalui *switch* 6 dengan *port output* [3], melalui *switch* 5 dengan *port output* [2], melalui *switch* 4 dengan *port output* [2], melalui *switch* 1 dengan *port output* [1]. Untuk jalur yang dilewati dari *switch* 10 ke *switch* 1 adalah rute yang pertama dengan total *cost* 20.

Hasil pengujian dari Tabel 5.4, no 1 terdapat perbedaan nilai *cost* pada *switch* 11 ke 7 dan 11 ke 6. Perbedaan ini yang membuat pemilihan jalur yang dipilih adalah *switch* 11 ke 7 bukan 11 ke 6. Untuk no 2, jalur yang dipilih berdasarkan *input* pertama pada tabel *routing.* Dapat dilihat jalur yang dipilih memiliki *switch* dengan nomor kecil. Sedangkan untuk no 3, terdapat perbedaan nilai *cost* pada *switch* 7 ke 11 dan *switch* 7 ke 5. Nilai *cost switch* 7 ke 11 lebih kecil dibanding *switch* 7 ke 5. Sehingga pemilihan jalur dipilih berdasarkan nilai *cost* minimum.

2. Jika *cost* sama dan jumlah *hop* yang dilewati beda. Pada pengujian ini akan diberi 2 rute yang sama-sama memiliki total *cost* yang sama dan *switch* tujuannya sama.

Tabel 5.5 Pemilihan jalur 2 algoritma Bellman-Fod

No	*Switch*		Pilihan Jalur	Jalur yang terlewati	*Cost* antar *switch*	Total *Cost*
	Awal	Tujuan				
1.	10	1	10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	1->2 =5, 1->4=5, 2->3=6, 2->4=9, 3->6=3, 4->5=3, 5->7=4, 6->5=9, 6->11=4, 7->8=2, 7->11=10, 8->9=4, 9->10=2, 10->11=2	20
			10[2]->9[2]->8[2]->7[2]->5[4]->4[2]->1[1]			
2.	9	1	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	1->2=3, 1->4=3, 2->3=1, 2->4=9, 3->6=8, 4->5=5, 5->7=2, 6->5=9, 6->11=2, 7->8=7, 7->11=10, 8->9=3, 9->10=3, 10->11=3	20
			9[2]->8[2]->7[2]->5[2]->4[2]->1[1]			
3.	8	1	8[2]->7[2]->5[2]->4[2]->1[1]	8[2]->7[2]->5[2]->4[2]->1[1]	1->2 =2, 1->4=9, 2->3=3, 2->4=8, 3->6=1, 4->5=2, 5->7=6, 6->5=9, 6->11=4, 7->8=3, 7->11=10, 8->9=1, 9->10=3, 10->11=6	20
			8[3]->9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]			

Berdasarkan Tabel 5.5 no 1, dari *switch* awal 10 ke *switch* tujuan 1 memiliki pilihan rute. Rute yang pertama dari *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [2], melalui *switch* 6 dengan *port output* [2], melalui *switch* 3 dengan *port output* [2], melalui *switch* 2 dengan *port output* [2], dan melalui *switch* 1 dengan *port output* [1]. Sedangkan, pilihan rute yang kedua dari *switch* 10 dengan *port output* [2], melalui *switch* 9 dengan *port output* [2], melalui *switch* 8 dengan *port output* [2], melalui *switch* 7 dengan *port output* [2], melalui *switch* 5 dengan *port output* [4], melalui *switch* 4 dengan *port output* [2], dan melalui *switch* 1 dengan *port output* [1]. Untuk jalur yang dilewati dari *switch* 10 ke *switch* 1 adalah rute yang pertama dengan total *cost* 20.

Hasil pengujian dari Tabel 5.5 no 1 dan no 2, jalur yang dipilih berdasarkan *input* pertama pada tabel *routing.* Dari Tabel 5.5, jalur yang dipilih memiliki jumlah *switch* dengan nomor kecil. Sedangkan no 3, jalur yang dipilih berdasarkan jumlah *hop*. Memiliki perbedaan 2 *hop* yang dilewati, maka *hop* yang kecil akan dipilih.

Untuk pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi *program,* dan rute terpendek akan dibagi menjadi 2 kali pengujian. Pengujian pertama akan melihat nilai *bandwidth* dan *throughput*. Pengujian kedua akan melihat nilai *latency*, waktu eksekusi, dan rute terpendek. Sama dengan pengujian pemilihan jalur, *cost* antar *switch* akan di-*input* secara manual dan nilainya bersifat acak. *Switch* awal dan tujuan akan dibuat sama disetiap pengujian.

Tabel 5.6 Pengujian *bandwidth*, *throughput*, latency (*delay*), waktu eksekusi program, dan rute terpendek Algoritma *Bellman-Ford*

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
1.	9	1	125	81,6	9010	0,341825 008392	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
2.	9	1	250	173	8999	0,373856 067659	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
3.	9	1	375	221	8999	0,396491 05072	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
4.	9	1	500	357	9009	0,394845 962524	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
5.	9	1	625	299	9010	0,392257 2879791	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
6.	9	1	750	223	9009	0,395108 938217	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
7.	9	1	875	385	9008	0,405818 939209	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
8.	9	1	1000	530	9008	0,368486 166	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19

Dari Tabel 5.6 didapatkan hasil, nilai *throughput* mengalami kenaikan yang cukup banyak ketika bandwith 125 Mbps menjadi 625 Mbps. Setelah itu mengalami penurunan *throughput* ketika *bandwidth* 625 Mbps menjadi 750 Mbps. Kemudian *throughput* kembali naik pada *bandwidth* 875 Mbps hingga 1000

Mbps. Untuk nilai *latency* tidak mengalami perubahan yang signifikan apabila ada perubahan nilai *bandwidth*. Sedangkan waktu eksekusi program dalam mencari rute dikisaran 0.341 hingga 0.405 detik. Untuk pemilihan rute yang dilalui, pada semua percobaan melewati rute yang sama. Rute yang dipilih dimulai dari *switch* 9 dengan *port output* [3], melalui *switch* 10 dengan *port output* [3], melalui *switch* 11 dengan *port output* [2], melalui *switch* 6 dengan *port output* [2], melalui *switch* 3 dengan *port output* [2], melalui 2 *switch* dengan *port output* [2], melalui *switch* 1 dengan *port output* [1]. Dari jalur yang telah dilewati dihasilkan total *cost* 19.

5.2 Analisis hasil pengujian

Pada subbab ini akan dibahas analisis hasil pengujian dari masing-masing algoritma. Dari hasil pengujian masing-masing algoritma, akan dilakukan analisis. Kemudian hasil dari analisis masing-masing algoritma, akan dibandingkan untuk memperoleh algoritma dengan rute terpendek yang menghasilkan nilai *cost* minimum.

5.2.1 Analisis hasil pengujian Algoritma *Dijkstra*

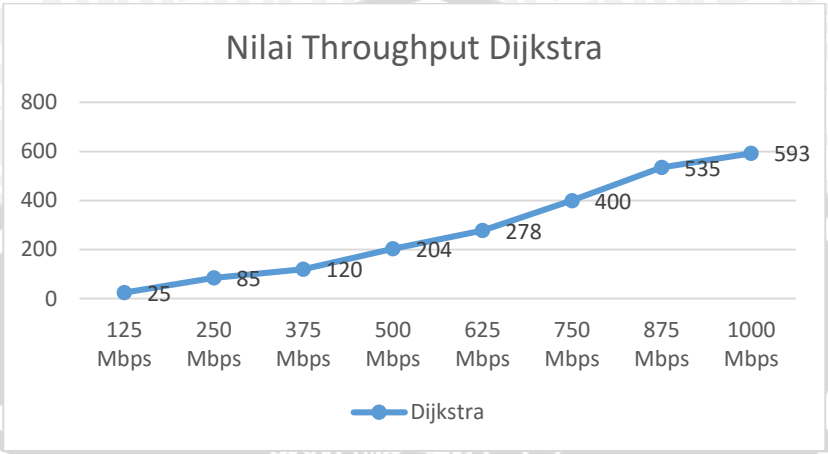
Analisis hasil pengujian Algoritma *Dijkstra* pengujian pemilihan jalur, didapatkan 3 kondisi ketika program melakukan pemilihan jalur. 3 kondisi ini antara lain.

1. Berdasarkan *input* awal yang masuk pada tabel *routing*, lihat Tabel 5.1. Tabel *routing* sendiri akan di-*input* secara urut dari urutan nomor *switch* kecil ke besar. Urutan peletakkan *switch* pada perancangan topologi ini yang nantinya akan mempengaruhi pemilihan jalur. Semakin banyak jumlah *switch* yang nomornya kecil maka semakin besar kemungkinan jalur tersebut terpilih.
2. Berdasarkan pemilihan *cost* dengan nilai kecil, lihat Tabel 5.1. Jika kondisi 1 memiliki *input switch* yang sama, maka kondisi 2 yang akan menentukan pemilihan jalurnya.
3. Berdasarkan *hop* yang pendek, lihat Tabel 5.2. Jika perbedaan *switch* yang dilewati lebih dari sama dengan 2, maka kondisi 3 yang akan menentukan pemilihan jalurnya.

Untuk analisis hasil pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi program, dan rute terpendek, didapat dari Tabel 5.3, antara lain.

1. Algoritma *Dijkstra* menunjukan perkembangan yang cukup signifikan ketika *bandwidth* ditambah dari 125 Mbps menjadi 625 Mbps. *Throughput* yang dihasilkan dari perubahan *bandwidth* tersebut terbukti mengalami penambahan, dapat dilihat pada Gambar 5.3.
2. Ketika dilakukan penambahan *bandwidth* 125 Mbps lagi, maka masih terdapat perubahan yang signifikan pada *throughput* tersebut. Penambahan *throughput* terus terjadi hingga *bandwidth* menjadi 1000 Mbps, dapat dilihat pada Gambar 5.3.
3. *Delay* tidak terlalu terjadi perubahan yang signifikan, ketika ada perubahan *bandwidth*.

4. Pada waktu eksekusi program *Dijkstra* ini membutuhkan waktu sekitar 0.00048 hingga 0.000633 detik.
5. Rute terpendek yang dilalui sama pada setiap pengujian, perubahan nilai *bandwidth*, *throughput*, *delay*, dan waktu eksekusi tidak mempengaruhi pencarian rute.



Gambar 5.3 Grafik *throughput Dijkstra*

5.2.2 Analisis hasil pengujian Algoritma *Bellman-Ford*

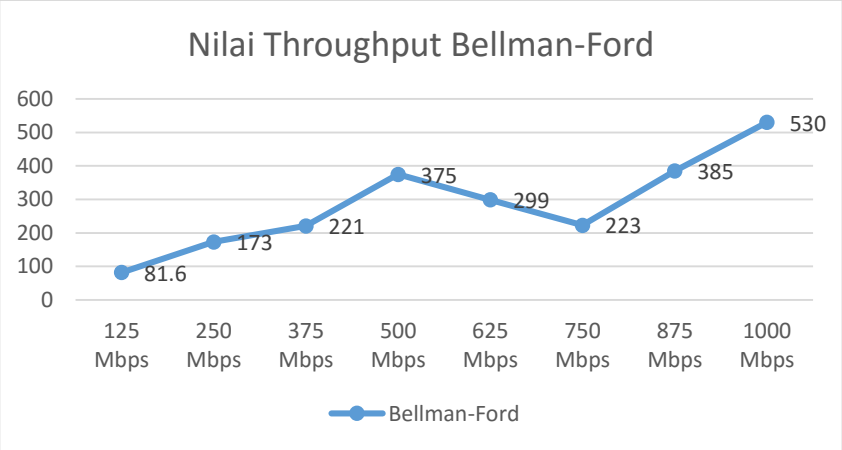
Hasil yang didapat dari pengujian pemilihan jalur, sama dengan hasil yang didapat pada pengujian pemilihan jalur Algoritma *Dijkstra*. Dari hasil tersebut didapatkan 3 kodisi pemilihan jalur, antara lain.

1. Berdasarkan *input* awal yang masuk pada tabel *routing*, dapat dilihat pada Tabel 5.4. Tabel *routing* sendiri akan di-*input* secara urut dari urutan nomor *switch* kecil ke besar. Urutan peletakkan *switch* pada perancangan topologi ini yang nantinya akan mempengaruhi pemilihan jalur. Semakin banyak jumlah *switch* yang nomornya kecil maka semakin besar kemungkinan jalur tersebut terpilih.
2. Berdasarkan pemilihan *cost* dengan nilai kecil, dapat dilihat pada Tabel 5.4. Jika kondisi 1 memiliki *input switch* yang sama, maka kondisi 2 yang akan menentukan pemilihan jalurnya.
3. Berdasarkan *hop* yang pendek, lihat Tabel 5.5. Jika perbedaan *switch* yang dilewati lebih dari atau sama dengan 2, maka kondisi 3 yang akan menentukan pemilihan jalurnya.

Untuk analisis hasil pengujian *bandwidth*, *throughput*, latency (*delay*), waktu eksekusi program, dan rute terpendek, didapat dari Tabel 5.6, antara lain.

1. Algoritma *Bellman-Ford* memberikan perkembangan yang signifikan pada nilai *throughput* ketika *bandwidth* ditambah dari 125 Mbps menjadi 625 Mbps, dapat dilihat pada Gambar 5.4.
2. Ketika dilakukan penambahan *bandwidth* menjadi 750 Mbps, nilai *throughput* mengalami penurunan. Sedangkan, apabila dilakukan penambahan 125 Mbps lagi, menjadi 875 hingga 1000 Mbps, nilai *throughput* mengalami kenaikan kembali, dapat dilihat pada Gambar 5.4.

3. *Delay* tidak terlalu terjadi perubahan nilai yang signifikan, ketika ada perubahan *bandwidth*.
4. Waktu eksekusi program *Bellman-Ford* dalam mencari rute dikisaran 0.341 hingga 0.405detik.
5. Rute terpendek yang dilalui sama pada setiap pengujian, perubahan nilai *bandwidth, throughput, delay,* dan waktu eksekusi tidak mempengaruhi pencarian rute.



Gambar 5.4 Grafik *throughput Bellman-Ford*

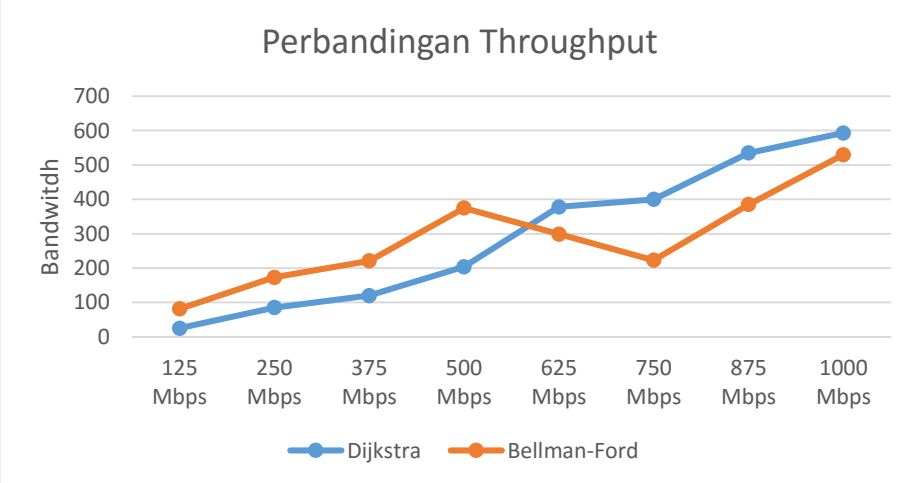
5.3 Perbandingan antar algoritma

Dari analisis pada masing-masing algoritma yang telah dilakukan, maka dapat diperoleh perbandingan dari keduanya. Perbandingan dari pengujian pemilihan jalur, didapatkan hasil yang sama dalam melakukan pemilihan jalur. Meski kedua algoritma sama-sama memiliki kondisi dalam melakukan pemilihan jalur, namun tetap akan menghasilkan total *cost* yang minimum. 3 kondisi pemilihan jalur ini, antara lain.

1. Berdasarkan *input* awal yang masuk pada tabel *routing,* dapat lihat pada Tabel 5.1 dan 5.4. Tabel *routing* sendiri akan di-*input* secara urut dari urutan nomor *switch* terkecil ke yang lebih besar. Urutan peletakkan *switch* pada perancangan topologi ini yang nantinya akan mempengaruhi pemilihan jalur. Semakin banyak jumlah *switch* yang dengan nomor yang kecil, maka semakin besar kemungkinan jalur tersebut terpilih.
2. Berdasarkan pemilihan *cost* dengan nilai kecil, dapat dilihat pada Tabel 5.1 dan 5.4. Jika kondisi 1 memiliki *input switch* yang sama, maka kondisi 2 yang akan menentukan pemilihan jalurnya.
3. Berdasarkan *hop* yang pendek, dapat dilihat pada Tabel 5.2 dan 5.5. Jika perbedaan *switch* yang dilewati lebih dari atau sama dengan 2, maka kondisi 3 yang akan menentukan pemilihan jalurnya.

Perbandingan dari analisis hasil *bandwidth, throughput, latency (delay),* waktu eksekusi program, dan rute terpendek, didapatkan 5 kesimpulan dari perbandingan antara Algoritma *Dijkstra* dan Belman-ford, yaitu.

1. Algoritma Djikstra mengalami kenaikan *throughput*yang terus bertahap sesuai dengan perubahan nilai *bandwidth*-nya. Sedangkan Algoritma *Bellman-Ford* memiliki penurunan pada *bandwidth*750 Mbps. Mengalami kenaikan yang cukup signifikan pada *bandwidth* 875 Mbps dan 1000 Mbps.
2. Algoritma *Dijkstra* memiliki kenaikan *throughput* yang kecil (di bawah *Bellman-Ford*) pada *bandwith* yang kecil pula, namun memiliki kenaikan *throughput* yang cukup baik ketika diberikan *bandwidth* besar (di atas 750 Mbps). Sedangkan, Algoritma *Bellman-Ford* memiliki *throughput* yang cukup besar ketika *bandwidth* kecil (di bawah 750 Mbps), namun pada *bandwidth* besar, tidak begitu mengalami pelonjakan, dapat dilihat pada Gambar 5.5. Nilai *bandwidth* dan *throughput* yang tinggi menandakan bahwa kualitas jaringan tersebut cepat, karena dapat mengirim data dalam jumlah yang banyak.
3. *Delay* masing-masing algoritma tidak mengalami perubahan yang signifikan dan *delay* tidak terpengaruh oleh waktu eksekusi tersebut. Nilai *delay* yang rendah akan menandakan bahwa kualitas jaringan tersebut cepat, karena pengiriman data dapat sampai tujuan dalam waktu yang lebih singkat.
4. Perbandingan waktu eksekusi program, kedua algoritma menunjukan bahwa Algoritma *Dijkstra* bisa dieksekusi lebih cepat daripada Algoritma *Bellman-Ford*. Waktu eksekusi program *Dijkstra* ini membutuhkan waktu sekitar 0.00048 hingga 0.000633 detik. Sedangkan, waktu eksekusi program *Bellman-Ford* dalam mencari rute dikisaran 0.341 hingga 0.405detik. Penyebab perbedaan waktu eksekusi ini dikarenakan Algoritma *Bellman-Ford* memiliki fitur *broadcast* tabel *routing* antar tetangga untuk kemudian dibandingkan nilainya dan ditentukan rute terpendeknya. Sedangkan, Algoritma *Dijkstra* tidak memiliki fitur tersebut sehingga Algoritma *Dijkstra* lebih cepet proses eksekusi programnya.
5. Rute terpendek yang dilalui sama, pada setiap pengujian. Nilai *bandwidth, throughput, delay,* dan waktu eksekusi tidak mempengaruhi pencarian rute.



Gambar 5.5 Grafik perbandingan *throughput*

BAB 6 KESIMPULAN DAN SARAN

Bab ini akan berisi subbab kesimpulan dan saran. Bab ini akan menjelaskan tetang kesimpulan yang diperoleh dari analisis hasil pengujian dan perbandingan antar algoritma. Serta saran yang perlu ditambahkan untuk penelitian selanjutnya.

6.1 Kesimpulan

Dari perancangan, implementasi, pengujian serta analisis terhadap Algoritma *Dijkstra* dan *Bellman-Ford* pada *Software Defined Network* maka dapat disimpulkan.

1. Perbandingan dari Algoritma *Dijkstra* dan *Bellman-Ford* pada pengujian pemilihan jalur, didapatkan 3 kondisi dalam melakukan pemilihan jalur. Kondisi yang pertama Berdasarkan *input* awal yang masuk pada tabel *routing*. Kondisi yang kedua berdasarkan pemilihan *cost* dengan nilai kecil. Untuk kondisi yang terakhir berdasarkan *hop* yang pendek.
2. Algoritma Djikstra mengalami kenaikan *throughput* yang terus bertahap sesuai dengan perubahan nilai *bandwidth*-nya. Sedangkan Algoritma *Bellman-Ford* memiliki penurunan pada *bandwidth*750 Mbps. Mengalami kenaikan yang cukup signifikan pada *bandwidth* 875 Mbps dan 1000 Mbps.
3. Algoritma *Dijkstra* memiliki kenaikan *throughput* yang kecil (di bawah *Bellman-Ford*) pada *bandwith* yang kecil pula, namun memiliki kenaikan *throughput* yang cukup baik ketika diberikan *bandwidth* besar (di atas 750 Mbps). Sedangkan, Algoritma *Bellman-Ford* memiliki *throughput* yang cukup besar ketika *bandwidth* kecil (di bawah 750 Mbps), namun pada *bandwidth* besar, tidak begitu mengalami pelonjakan. Nilai *bandwidth* dan *throughput* yang tinggi menandakan bahwa kualitas jaringan tersebut cepat, karena dapat mengirim data dalam jumlah yang banyak.
4. *Delay* masing-masing algoritma tidak mengalami perubahan yang signifikan dan *delay* tidak terpengaruh oleh waktu eksekusi tersebut. Nilai *delay* yang rendah akan menandakan bahwa kualitas jaringan tersebut cepat, karena pengiriman data dapat sampai tujuan dalam waktu yang lebih singkat.
5. Perbandingan waktu eksekusi program, kedua algoritma menunjukan bahwa Algoritma *Dijkstra* bisa dieksekusi lebih cepat daripada Algoritma *Bellman-Ford*. Waktu eksekusi program *Dijkstra* ini membutuhkan waktu sekitar 0.00048 hingga 0.000633 detik. Sedangkan, waktu eksekusi program *Bellman-Ford* dalam mencari rute dikisaran 0.341 hingga 0.405detik. Penyebab perbedaan waktu eksekusi ini dikarenakan Algoritma *Bellman-Ford* memiliki fitur *broadcast* tabel *routing* antar tetangga untuk kemudian dibandingkan nilainya dan ditentukan rute terpendeknya. Sedangkan, Algoritma *Dijkstra* tidak memiliki fitur tersebut sehingga Algoritma *Dijkstra* lebih cepet proses eksekusi programnya.
6. Rute terpendek yang dilalui sama, pada setiap pengujian. Nilai *bandwidth, throughput, delay,* dan waktu eksekusi tidak mempengaruhi pencarian rute.

6.2 Saran

Saran yang diberikan penulis untuk penyempurnaan penelitian ini. Penulis berharap dapat mengembangkan algoritma *routing* yang lain untuk bisa diimplementasikan pada *Software Defined Network*. Sedangkan pada pengujian perlu adanya beberapa parameter uji yang lain agar dapat mengembangkan menyempurnakan program lebih lanjut. Penelitian ini masih kurang dari sempurna dan perlu adanya beberapa perbaikan.

DAFTAR PUSTAKA

[1] Astuto, B.N.; Mendonça, M.; Nguyen, X.N.; Obraczka, K.; Turletti, T. A Survey of Software-Defined Networking: Past, Present, and Future of ProgrammableNetworks. *IEEE Commun. Surv. Tutor.* 2014, doi:10.1109/SURV.2014.012214.00180.

[2] Vinshoi, A.; Khumbhare, A., 21 Desember 2013, Open Flow 1.3.1 Sup*port*: *Controller* View, IBM, https://wiki.Opendaylight.org/images/d/dc/*Openflow*1.3_Sup*port*_for_Opendaylight.pd

[3]Feamster, Nick., dkk. 2008. *The Case for Separating Routing from Routers*. MIT Computer Science & AI Lab.

[4] Jiang, Jehn-Ruey., Huang, Hsin-Wen., Liao, Ji-Hau. and Chen, Szu-Yuan. 2014. *Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking*. National Central University, Taiwan.

[5] Mawarni, Sri. 2008. *Penerapan algoitma Dijkstra dalam mencari lintasan terpendek pada jaringan komputer*. Riau.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *Open flow: Enabling innovation in campus networks*," SIGCOMM CCR, vol. 38, no. 2, pp. 69{74, 2008}.

[7] Novandi, Raden Aprian Diaz. 2007. *Perbandingan Algoritma Dijkstra dan Algoritma Floyd-Warshall dalam Penentuan Lintasan Terpendek (Single Pair Shortest Path).* ITB, Bandung.

[8] Patel, Vaibhavi and Prof.ChitraBaggar. 2014. *A Survey Paper of Bellman-Ford Algorithm and Dijkstra Algorithm for Finding Shortest Path in GIS Application*. Gujarat, India.

[9] Rifiani, Vina. 2011. *Analisa Perbandingan Metode Routing Distance Vector Dan Link State Pada Jaringan Packet*. PENS-ITS. Surabaya.

[10] Setyawati, Michell. 2011. *Perbandingan Dijkstra, Bellman-Ford, dan juga Floyd-Warshall*. ITB, Bandung.

[11] Shivendu, Arnav., Dhakal, Dependra., Sharma, Diwas. 2015. *Emulation of Shortest Path Algorithm in Software Defined Networking Environment*. Sikkim Manipal Institute of Technology, East Sikkim, India

LAMPIRAN DATA HASIL PENGUJIAN SISTEM

1. Algoritma *Dijkstra*

- **Hasil pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi *program,* dan rute terpendek (percobaan 2)**

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
1.	9	1	125	25,2	9006	0,000632047652198	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
2.	9	1	250	87	9009	0,0004670629646	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
3.	9	1	375	125	9008	0,000461101531982	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
4.	9	1	500	210	9009	0,000615835189819	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
5.	9	1	625	280	8999	0,000459909439087	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
6.	9	1	750	330	9010	0,000460863113403	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
7.	9	1	875	540	9008	0.000458955764771	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
8.	9	1	1000	595	9010	0,000547885894775	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19

- **Hasil pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi *program,* dan rute terpendek (percobaan 3)**

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
1.	9	1	125	28	9000	0,000637054443359	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
2.	9	1	250	89,0	9000	0,0006051063537 6	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
3.	9	1	375	124	9001	0,0004618167877 2	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
4.	9	1	500	204	9008	0,0004758834838 87	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
5.	9	1	625	299	9008	0,0004501342773 44	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
6.	9	1	750	385	9010	0,0004618167877 2	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
7.	9	1	875	500	9010	0,0004680156707 76	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
8.	9	1	1000	593	9000	0,0004686311340 3	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19

2. Algoritma *Bellman-Ford*

- **Hasil pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi *program,* dan rute terpendek (percobaan 2)**

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
1.	9	1	125	85	9009	0,398635 149002	9[3]->10[3] ->11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
2.	9	1	250	175	9008	0,402714 967728	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
3.	9	1	375	231	9010	0.396852 970123	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
4.	9	1	500	245	9009	0,402057 886124	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
5.	9	1	625	305	9008	0,405014 99176	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
6.	9	1	750	204	9009	0,396769 046783	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
7.	9	1	875	345	9010	0,395752 906799	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19
8.	9	1	1000	535	9010	0,395299 911499	9[3]->10[3] -> 11[2]->6 [2]->3[2]-> 2[2]->1[1]	19

- **Hasil pengujian *bandwidth, throughput, latency (delay),* waktu eksekusi *program,* dan rute terpendek (percobaan 3)**

No	*Switch*		Parameter uji					Total *cost*
	Awal	Tujuan	*Bandwidth* (Mbps)	*Throughput* (Mbit/sec)	*Latency* (ms)	Waktu eksekusi (detik)	Jalur (*pathlist*)	
1.	9	1	125	89	9008	0,397276878357	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
2.	9	1	250	179	9010	0,362990140915	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
3.	9	1	375	220	9008	0,372627019882	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
4.	9	1	500	250	9010	0,376304149628	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
5.	9	1	625	302	9009	0,405323028564	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
6.	9	1	750	204	9002	0,36413192749	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
7.	9	1	875	365	8999	0,382005929947	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
8.	9	1	1000	532	9009	0,397735118866	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19