Analisis Perbandingan Algoritma Dijkstra Dan Bellman-Ford Untuk Menentukan Rute Terpendek Dengan Menggunakan *Software Defined Network*

Ulfa Kurniawati, Rakhmadhany Primananda, S.T, M.Kom., Dahnial Syauqy, S.T., M.T., M.Sc.
Jurusan Sistem Komputer, Fakultas Ilmu Komputer, Universitas Brawijaya (UB)
Jl. Veteran No 8, Malang 65145, Indonesia
e-mail: ulfakurniawan @gmail.com, rakhmadhany@gmail.com, dahnial87@ub.ac.id

Abstrack - *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

I. PENDAHULUAN

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 jaringan. 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.

II. LANDASAN KEPUSTAKAAN

A. *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. 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.

Software defined networking (SDN) adalah sebuah konsep pendekatan jaringan komputer dimana sistem pengkontrol dari arus data dipisahkan dari perangkat kerasnya. Umumnya, sistem pembuat keputusan kemana arus data dikirimkan dibuat menyatu dengan perangkat kerasnya. Sebuah konfigurasi SDN dapat menciptakan jaringan dimana perangkat keras pengontrol lalu lintas data secara fisik dipisahkan dari perangkat keras data forwarding plane. SDN memiliki protokol yang bernama OpenFlow. OpenFlow merupakan open standar komunikasi protokol yang mampu melakukan pemisahan antara control plene dan data plane dari sebuah perangkat jaringan, serta mampu menciptakan komunikasi yang sangat baik antara control plane dan data plane.

B. 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.

C. 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.

D. Pyretic

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.

E. Mininet

Mininet adalah sebuah simulator jaringan open source yang mendukung protokol OpenFlow untuk arsitektur SDN. Salah satu software yang paling populer digunakan oleh penelitian SDN masyarakat. 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 merupakan emulator jaringan yang mensimulasikan koleksi dari host-end, switch,

router, dan link pada single kernel Linux. Masing-masing elemen ini disebut sebagai "host" menggunakan virtualisasi ringan untuk membuat sistem tampilan tunggal sehingga terlihat seperti jaringan yang lengkap, menjalankan kernel yang sama, sistem, dan user code. Mininet penting bagi komunitas open-source SDN. Mininet biasanya digunakan sebagai simulasi, verifikasi, testing tool, dan resource.

F. 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. Dan *routing* dinamis sekarang ini menjadi alternatif pilihanan karena sifatnya yang fleksibel terhadap perubahan.

G. 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 minim.

H. Algoritma Dijkstra

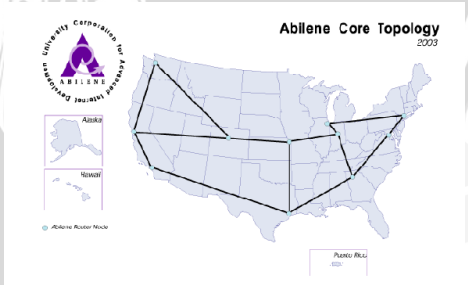
Algoritma Dijkstra merupakan salah satu varian dari algoritma Greedy, yaitu salah satu bentuk algoritma populer dalam pemecahan persoalan yang terkait dengan masalah optimasi atau pencarian solusi yang optimum karena algoritma ini sederhana (Michell Setyawati, 2011). Dari bahasanya, Greedy berarti rakus atau tamak. Prinsip algoritma Greedy yaitu ”take what you can get now!” yang artinya “ambil apa yang dapat anda peroleh sekarang!”. Prinsip inilah yang digunakan dalam pemecahan masalah optimasi. Algoritma Dijkstra disebut juga algoritma Link state.

I. Algoritma Bellman-Ford

Bellman-Ford memiliki struktur dasar menyerupai algoritma Dijkstra, akan tetapi dibandingkan dengan memilih simpul dengan bobot minimum yang belum digunakan secara Greedy, algoritma Bellman-Ford secara sederhana melakukan pengecekan terhadap semua (Michell Setyawati, 2011). Pengulangan ini memungkinkan jarak terpendek untuk dihitung secara akurat bertahap di dalam graf dengan kemungkinan suatu simpul dikunjungi dalam lintasan tersebut adalah sebanyak-banyaknya satu kali saja. Algoritma Bellman-Ford adalah algoritma untuk menghitung jarak terpendek (dari satu sumber) pada sebuah graf berbobot. Algoritma Bellman-Ford juga disebut algoritma distance vector.

F. Topologi Abilene

Topologi Abilene adalah topologi backbone performa tinggi yang dibuat oleh komunitas Internet2 pada akhir 1990-an. Pada tahun 2007 topologi Abilene sudah pension, jaringan tersebut menjadi ditingkatkan dan dikenal sebagai "Internet2 Network" (Jehn-Ruey Jiang, 2014). Topologi Abilene banyak diimplementasikan di Amerika serikat karena memiliki kemiripan dengan rute yang ada di amerika serikat. Pada gambar 2.1 terdapat gambar topologi Abilene. Nama Abilene sendiri berasal dari salah satu stasiun yang ada di Kansas.



Gambar 2.1 Topologi Abilene
Sumber: Jehn-Ruey Jiang (2014)

III. METODOLOGI

A. 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:

- Kebutuhan fungsional

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

1. Sistem dapat menentukan jalur terpendek dari sejumlah *node* yang ditentukan.
2. Kedua algoritma dapat menghasilkan nilai yang dapat diukur sebagai parameter perbandingan.

- Kebutuhan non-fungsional

Perangkat lunak yang dibutuhkan untuk pengembangan sistem adalah sebagai berikut :

1. Sistem Operasi, sistem operasi yang digunakan untuk menjalankan perangkat lunak adalah Ubuntu 14.04 LTS untuk Laptop.
2. *Pyretic* digunakan sebagai *controller*. Untuk melakukan coding pada *Pyretic* menggunakan bahasaa pemprogaman *python*.
3. Simulator yang digunakan adalah mininet dengan menggunakan miniedit sebagai virtualisasinya.
4. Protokol yang digunakan adalah *openflow*.

B. Perancangan

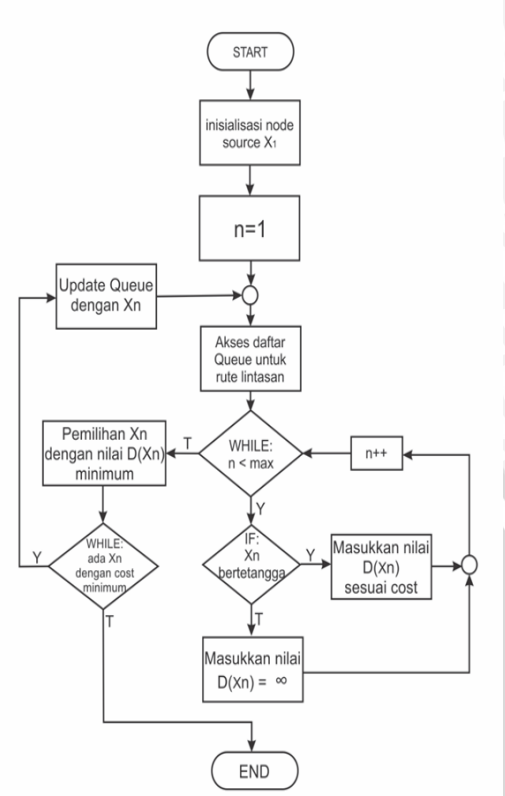
Sub bab perancangan ini akan menjelaskan mekanisme perancangan sistem. Pada perancangan ini akan dibagi menjadi 2 sub bab lagi yaitu perancangan algoritma dan perancangan arsitektur. Perancangan sangat diperlukan untuk mempermudah proses pendesinan sistem dan tahapan pengimplementasian.

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.

- Algoritma Dijkstra

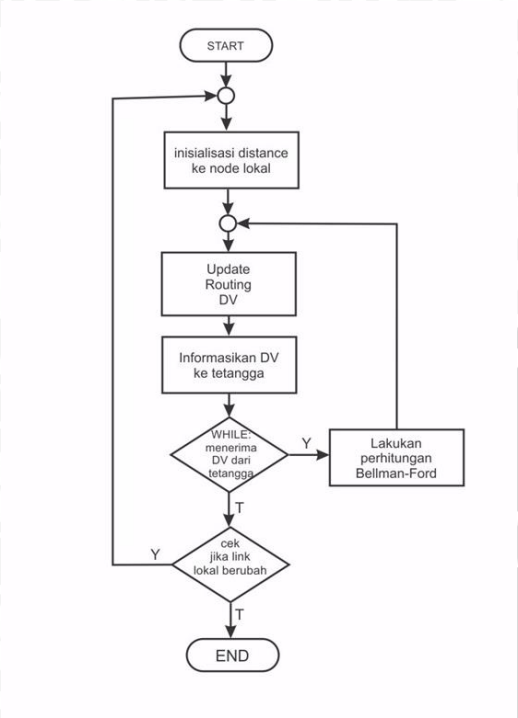
Perancangan algoritma Dijkstra akan menjelaskan mekanisme rute pencarian pada algoritma Dijkstra. Algoritma Dijkstra memiliki ciri khas pencarian dengan membandingkan nilai cost dengan tetangga lalu melakukan perhitungan cost pada setiap jalurnya sebelum menentukan rute terpendeknya. Berikut adalah alur mekanisme pencarian pada algoritma Dijkstra gambar 3.1.



Gambar 3.1 Flowchart algoritma Dijkstra

- Algoritma Bellman-Ford

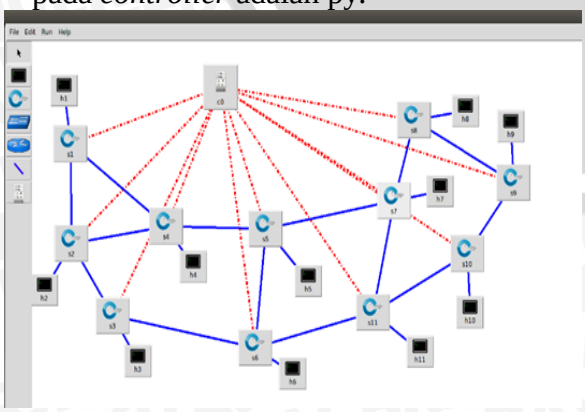
Perancangan algoritma Bellman-Ford akan menjelaskan mekanisme rute pencarian pada algoritma Bellman-Ford. Algoritma Bellman-Ford memiliki ciri khas pencarian dengan menanyakan cost tetangganya, kemudian saling berukar informasi cost. Setelah itu mulai menghitung cost pada setiap jalurnya sebelum menentukan rute terpendeknya. Dan apabila terdapat update maka antar node akan saling memberi informasi. Pada proses pemberian informasi ini sering terjadi perulangan antar node. Berikut adalah alur mekanisme pencarian pada algoritma Bellman-Ford gambar 3.2.



Gambar 3.2 Flowchart algoritma Bellman-Ford

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

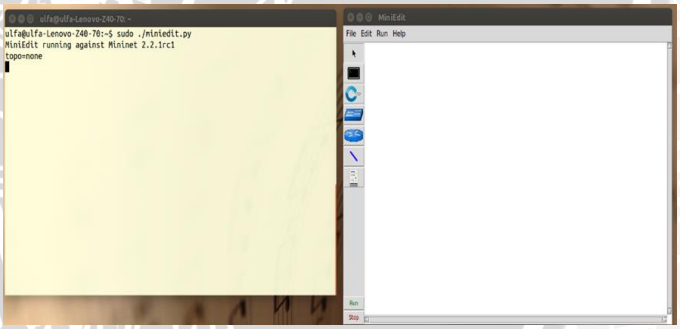
IV. IMPLEMENTASI

A. Running mininet dan miniedit

Untuk melakukan perancangan topologi maka diperlukan untuk melakukan running simulator mininet dan untuk mempermudah pengambaran topologi setelah melakukan running simulator maka dilakukan running miniedit yang berfungsi sebagai visualisasi dari mininet. Dalam implementasi sistem, running mininet dan miniedit memiliki peranan awal yang penting. Proses running akan dilakukan pada terminal Ubuntu. Berikut adalah gambar 4.1 *source code running* mininet pada terminal Ubuntu dan gambar 4.2 *source code* running miniedit.

```
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 4.1 *Running* mininet pada terminal Ubuntu



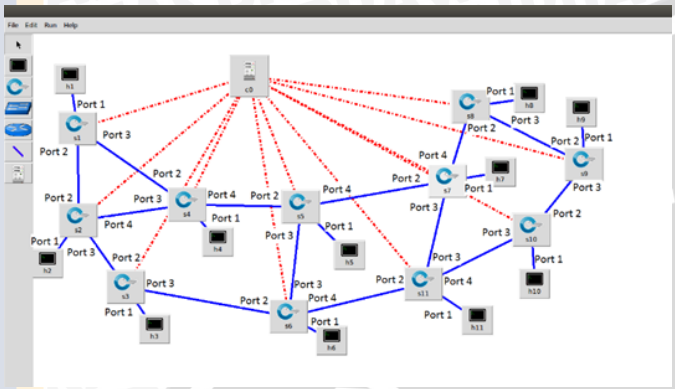
Gambar 4.2 *Running* miniedit

B. **Pembuatan Topologi**

Setelah miniedit terbuka, maka proses perancangan topologi dapat dilakukan. Pada perancangan topologi ini topologi yang digunakan adalah topologi Abilene. Topologi Abilene berasal dari implementasi wilayah yang ada di Amerika Serikat. Topologi ini akan dibuat dengan menggunakan 11 switch yang masing-masing switch memiliki 1 host.

Pada proses perancangan topologi tahapan pemasangan kabel antar controller, switch, dan host akan mempengaruhi output hasil yang didapat ketika program telah diupload di controller.

Tahapan pemasangan ini hanya berlaku apabila controller yang digunakan adalah controller Pyretic. Tahapan pemasangan ini dimulai dari pemasangan kabel dari controller ke switch lalu dari switch ke host dan setelah itu dari switch ke switch jika terdapat sambungan. Berikut adalah gambar 4.3 adalah gambar dari perancangan topologi Abilene pada miniedit.

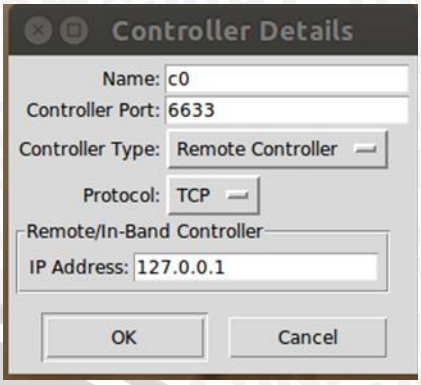


Gambar 4. 3 Perancangan topologi Abilene

C. Running Pyretic

Pyretic adalah controller yang mengimplementasikan bahasa python sebagai sistem runtime program yang ditulis dalam Pyretic pada switch jaringan. Seperti yang dijelaskan pada sub bab perancangan topologi controller yang akan digunakan adalah controller Pyretic.

Controller Pyretic memiliki ciri khas pada proses penyambungan kabel antar perangkat yang bersifat statis dan default. Proses penyambungan perangakat dimulai dari pemasangan dari controller ke *switch* kemudian switch ke host dan switch ke *switch.* Jika pada pemasangan tidak urut maka output yang dihasilkan berbeda. Untuk melakukan setting *controller type* dan *controller port* untuk Pyretic dengan cara mengklick kanan gambar *controller* dan memilih menu properties. Setting *controller* dengan *remote controller* dan *controller* port 6633. Gambar 4.4 adalah settingan *controller type* dan *controller port.*



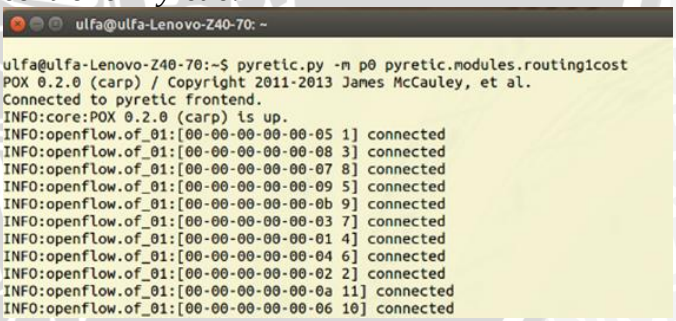
Gambar 4.4 Settingan *controller type* dan *controller port.*

Tabel 4.1 *source code upload* Pyretic

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

Setelah melakukan setting *controller type* dan *controller port*, maka file utama yang berisi fungsi main siap untuk di upload pada *controller*. Dengan sebelumnya klick *icon run* pada miniedit dan lakukan *upload.* Tabel 4.1, untuk melakukan *upload* file pada *controller.*

Setelah menuliskan *source coode* diatas maka enter, apabila tercetak di terminal seperti Gambar 4.5 maka program telah berhasil di *upload* pada *controller Pyretic.*



Gambar 4.5 Teruplоad program pada controller Pyretic

V. PENGUJIAN SISTEM

A. Pengujian 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 [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 5.1 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[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 5.2 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	0,0060987472 5342	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
2.	9	1	250	85,0	8999	0,0005209445 95337	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
3.	9	1	375	120	9008	0,0005850791 93115	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
4.	9	1	500	204	9005	0,0004889965 05737	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	0,0006091594 69604	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
7.	9	1	875	535	8999	0,0004620552 06299	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
8.	9	1	1000	593	9008	0,0006048679 35181	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.

B. Pengujian 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 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[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 5.4 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 5.5 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[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 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 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	0,341825008392	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
2.	9	1	250	173	8999	0,373856067659	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
3.	9	1	375	221	8999	0,39649105072	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
4.	9	1	500	357	9009	0,394845962524	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
5.	9	1	625	299	9010	0,392257287979 1	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
6.	9	1	750	223	9009	0,395108938217	9[3]->10[3]->11[2]->6[2]->3[2]->2[2]->1[1]	19
7.	9	1	875	385	9008	0,405818939209	9[3]->10[3]-> 11[2]->6[2]->3[2]->2[2]->1[1]	19
8.	9	1	1000	530	9008	0,368486166	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.

VI. Kesimpulan dan saran

A. 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.

B. 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] 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

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

[3] 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.

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

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

[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] 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.

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

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