

SDN Project Using Mininet, POX, and Custom Topology with Dijkstra's Algorithm

Prepared by

ENG.KHALED ELSAYED MOTAWEA

For the month of September

2024

This report outlines the steps for setting up a Software Defined Networking (SDN) project using Mininet, POX controller, and a custom topology with Dijkstra's algorithm for optimal path computation. The components used in this project are:

1. Custom Topology: Defined using Mininet.
2. Dijkstra Algorithm: Implemented in the POX controller.
3. Delay file: Contains the delay values for links between switches.

Prerequisites:

Before starting the project, ensure you have the following installed:

- Mininet: A network emulator that creates virtual network topologies.
- POX: A Python-based OpenFlow controller framework.
- Open vSwitch: For enabling SDN capabilities with Mininet.
- Python: To run the POX controller and the custom topology script.

1. Custom Topology (Mininet Topology)

The custom topology is defined in the `newtopo.py` file using Mininet's `Topo` class. The topology consists of 5 switches and 2 hosts, with links between switches that have varying delays and bandwidth.

Custom Topology Code:

```
```python
from mininet.topo import Topo

class CustomTopo(Topo):
 def _init_(self, **opts):
 # Initialize topology and default options
 Topo._init_(self, **opts)

 # Add switches
```

```

s1 = self.addSwitch('s1')
s2 = self.addSwitch('s2')
s3 = self.addSwitch('s3')
s4 = self.addSwitch('s4')
s5 = self.addSwitch('s5')

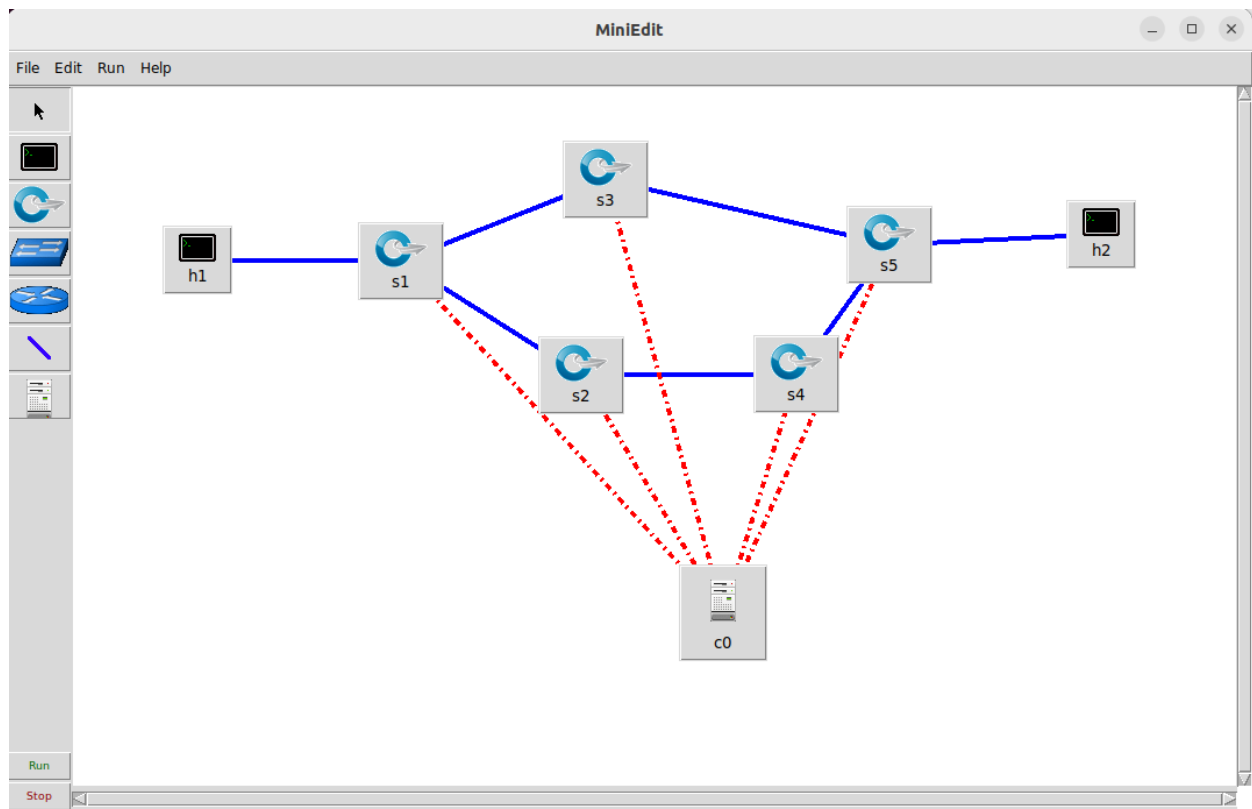
Add hosts
h1 = self.addHost('h1')
h2 = self.addHost('h2')

Add links between hosts and switches
self.addLink(s1, h1, **linkopts)
self.addLink(s5, h2, **linkopts)

Add links between switches with different delays and bandwidths
self.addLink(s1, s2, **linkopts_g)
self.addLink(s1, s3, **linkopts_h)
self.addLink(s2, s4, **linkopts_i)
self.addLink(s4, s5, **linkopts_j)
self.addLink(s3, s5, **linkopts_k)

Link options with different parameters
linkopts_g = dict(delay='10ms', loss=0)
linkopts_h = dict(delay='20ms', loss=0)
linkopts_i = dict(delay='30ms', loss=0)
linkopts_j = dict(delay='40ms', loss=0)
linkopts_k = dict(delay='50ms', loss=0)
linkopts = dict(delay='0ms', loss=0)
topos = {'custom': (lambda: CustomTopo())}
'''

```



## Description of Code:

Let's break down the code line by line. This code is for defining a custom network topology in Mininet, a network emulation tool.

## Imports

```
```python
from mininet.topo import Topo
```
```

- `from mininet.topo import Topo`: Imports the `Topo` class from the `mininet.topo` module. `Topo` is the base class for defining custom network topologies in Mininet.

## Custom Topology Class

```
```python
class CustomTopo(Topo):
```
```

- `class CustomTopo(Topo):` : Defines a new class `CustomTopo` that inherits from `Topo`. This class will be used to create a custom network topology.

```
```python
```

```
    def _init_(self, **opts):
```

```
```
```

- `def __init__(self, **opts):` : The `__init__` method is the initializer for the `CustomTopo` class. It should be `__init__`, not `_init_`.

```
```python
```

```
    # Initialize topology and default options
```

```
    Topo.__init__(self, **opts)
```

```
```
```

- `Topo.__init__(self, **opts)` : Calls the `__init__` method of the parent `Topo` class to initialize the topology with any options passed as keyword arguments (`**opts`).

## Adding Network Components

```
```python
```

```
    # Add switches
```

```
    s1 = self.addSwitch('s1')
```

```
    s2 = self.addSwitch('s2')
```

```
    s3 = self.addSwitch('s3')
```

```
    s4 = self.addSwitch('s4')
```

```
    s5 = self.addSwitch('s5')
```

```
```
```

- `self.addSwitch('s1')` : Adds a switch with the name `s1` to the topology. This method returns a reference to the switch object, which is stored in `s1`.

- `self.addSwitch('s2')` : Adds a second switch with the name `s2`, and similarly for `s3`, `s4`, and `s5`.

```
```python
```

```
    # Add hosts
```

```
    h1 = self.addHost('h1')
```

```
    h2 = self.addHost('h2')
```

```
```
```

- `self.addHost('h1')` : Adds a host with the name `h1` to the topology. This method returns a reference to the host object, which is stored in `h1`.

- `self.addHost('h2')` : Adds a second host with the name `h2`.

```
```python
    # Add links between hosts and switches
    self.addLink(s1, h1, **linkopts)
    self.addLink(s5, h2, **linkopts)
```
```

- `self.addLink(s1, h1, **linkopts)` : Adds a link between switch `s1` and host `h1` with options defined in `linkopts`.

- `self.addLink(s5, h2, **linkopts)` : Adds a link between switch `s5` and host `h2` with options defined in `linkopts`.

```
```python
    # Add links between switches with different delays and bandwidths
    self.addLink(s1, s2, **linkopts_g)
    self.addLink(s1, s3, **linkopts_h)
    self.addLink(s2, s4, **linkopts_i)
    self.addLink(s4, s5, **linkopts_j)
    self.addLink(s3, s5, **linkopts_k)
```
```

- `self.addLink(s1, s2, **linkopts_g)` : Adds a link between switch `s1` and switch `s2` with options defined in `linkopts_g`.

- `self.addLink(s1, s3, **linkopts_h)` : Adds a link between switch `s1` and switch `s3` with options defined in `linkopts_h`.

- `self.addLink(s2, s4, **linkopts_i)` : Adds a link between switch `s2` and switch `s4` with options defined in `linkopts_i`.

- `self.addLink(s4, s5, **linkopts_j)` : Adds a link between switch `s4` and switch `s5` with options defined in `linkopts_j`.

- `self.addLink(s3, s5, **linkopts_k)` : Adds a link between switch `s3` and switch `s5` with options defined in `linkopts_k`.

## Link Options

```
```python
# Link options with different parameters
linkopts_g = dict(delay='10ms', loss=0)
linkopts_h = dict(delay='20ms', loss=0)
linkopts_i = dict(delay='30ms', loss=0)
linkopts_j = dict(delay='40ms', loss=0)
linkopts_k = dict(delay='50ms', loss=0)
linkopts = dict(delay='0ms', loss=0)
```
```

- ``linkopts_g`, `linkopts_h`, `linkopts_i`, `linkopts_j`, `linkopts_k``: Define different parameters for links between switches. Each dictionary specifies ``delay`` and ``loss`` for the link.

- ``linkopts``: Defines the default link options with no delay and no packet loss.

## Topology Registry

```
```python
topos = {'custom': (lambda: CustomTopo())}
```
```

- ``topos = {'custom': (lambda: CustomTopo())}``: Registers the ``CustomTopo`` class with Mininet's topology registry under the name ``'custom'``. This lambda function creates an instance of ``CustomTopo`` when the ``'custom'`` topology is requested.

## Summary

- Imports: Required classes and functions from Mininet and Python libraries.

- ``CustomTopo`` Class: Defines a custom network topology with 5 switches and 2 hosts, with varying link parameters.

- Initialization: The class constructor should be corrected to ``__init__`` to properly initialize the topology.

- Link Options: Different parameters for each link are defined to simulate varying network conditions.

- Topology Registration: The custom topology is registered so it can be used in Mininet simulations.

## Running the Topology:

save the py script of the topo as newtopo.py.

To run the custom topology, execute the following command in your terminal:

```
```bash
$ sudo mn --custom newtopo.py --topo custom --controller remote --mac
```
```

This command starts the Mininet network with your custom topology and uses a remote POX controller.

## 2. Delay File (delay.csv)

The `delay.csv` file contains the delay values for the links between switches. This file is used by the POX controller to compute the shortest path using the Dijkstra algorithm.

delay.csv:

```
```
link,delay
g,10
h,20
i,30
j,40
k,50
```
```

This file lists the link names (`g`, `h`, `i`, `j`, `k`) corresponding to the connections between switches and the delay for each link in milliseconds.

**This file should be in this directory `/pox/pox/misc`**

### 3. Dijkstra Algorithm (POX Controller)

The Dijkstra algorithm is implemented in the POX controller to compute the shortest path between hosts based on link delays.

**The code resides in the `pox/pox/misc/dijkstra.py` file.**

#### Dijkstra Algorithm Code:

```
```python
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.util import dpidToStr
from pox.lib.addresses import EthAddr, IPAddr
from collections import defaultdict
from copy import deepcopy
import os
import csv

log = core.getLogger()
delayFile = "%s/pox/pox/misc/delay.csv" % os.environ['HOME']

hosts = {'h1': 's1', 'h2': 's5'}
linkNames = {
    'g': ('s1', 's2'),
    'h': ('s1', 's3'),
    'i': ('s2', 's4'),
    'j': ('s4', 's5'),
    'k': ('s3', 's5'),
}
portMappings = {
    's1': {'h1': 1, 's2': 2, 's3': 3},
    's2': {'s1': 1, 's4': 2},
    's3': {'s1': 1, 's5': 2},

```

```

's4': {'s2': 1, 's5': 2},
's5': {'h2': 1, 's4': 2, 's3': 3},
}

```

```

class Dijkstra(EventMixin):

```

```

    def __init__(self):
        self.listenTo(core.openflow)
        log.debug("Enabling Dijkstra Module")

        self.delays = {}
        self.switches = set()
        self.neighbors = defaultdict(set)

        # Read the delay file to initialize the topology
        with open(delayFile, 'r') as csvfile:
            links = csv.reader(csvfile)
            next(links) # skip the header
            for linkName, delay in links:
                s1, s2 = linkNames[linkName]
                self.delays[(s1, s2)] = int(delay)
                self.delays[(s2, s1)] = int(delay)
                self.switches.add(s1)
                self.switches.add(s2)
                self.neighbors[s1].add(s2)
                self.neighbors[s2].add(s1)

    def _dijkstra(self, source):
        distances = defaultdict(lambda: float('inf'))
        distances[source] = 0
        previous = {}
        unseen = deepcopy(self.switches)

```

```

while unseen:
    u = min(unseen, key=lambda x: distances[x])
    unseen.remove(u)
    for v in self.neighbors[u]:
        alt = distances[u] + self.delays[(u, v)]
        if alt < distances[v]:
            distances[v] = alt
            previous[v] = u
    return distances, previous

def _getPortMapping(self, source):
    distances, previous = self._dijkstra(source)
    ports = {}
    for destHost, destSwitch in hosts.items():
        if source == destSwitch:
            ports[destHost] = portMappings[source][destHost]
            continue
        while source != previous[destSwitch]:
            destSwitch = previous[destSwitch]
        ports[destHost] = portMappings[source][destSwitch]
    return ports

def _handle_ConnectionUp(self, event):
    switch = 's' + str(event.dpid)
    ports = self._getPortMapping(switch)
    for host, (ip, mac) in hosts.items():
        port = ports[host]

        # Install MAC-based flow
        msg_mac = of.ofp_flow_mod()
        msg_mac.match.dl_dst = EthAddr(mac)

```

```

        msg_mac.actions.append(of.ofp_action_output(port=port))
        event.connection.send(msg_mac)

    # Install IP-based flow
    msg_ip = of.ofp_flow_mod()
    msg_ip.match.nw_dst = IPAddr(ip)
    msg_ip.match.dl_type = 0x800 # IPv4
    msg_ip.actions.append(of.ofp_action_output(port=port))
    event.connection.send(msg_ip)

log.debug("Dijkstra installed on %s", dpidToStr(event.dpid))

def launch():
    core.registerNew(Dijkstra)
'''

```

Code Explanation:

Let's break down the provided code line by line. This script is for a POX controller and implements a Dijkstra algorithm-based routing module.

Imports and Initializations

```

'''python
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.util import dpidToStr
from pox.lib.addresses import EthAddr, IPAddr
from collections import defaultdict
from copy import deepcopy
import os

```

```
import csv
'''
- `from pox.core import core` : Imports the core POX module for accessing global services
like logging.

- `import pox.openflow.libopenflow_01 as of` : Imports the OpenFlow library to interact
with OpenFlow messages and actions.

- `from pox.lib.revent import *` : Imports event handling utilities from POX, which allows
the use of event-driven programming.

- `from pox.lib.util import dpidToStr` : Imports a utility to convert data plane IDs (DPIDs) to
string representations.

- `from pox.lib.addresses import EthAddr, IPAddr` : Imports classes for handling Ethernet
and IP addresses.

- `from collections import defaultdict` : Imports `defaultdict` for easy default value
handling in dictionaries.

- `from copy import deepcopy` : Imports `deepcopy` for creating deep copies of objects.

- `import os` : Imports the operating system interface, used for file paths.

- `import csv` : Imports the CSV module to read CSV files.
'''python
log = core.getLogger()
delayFile = "%s/pox/pox/misc/delay.csv" % os.environ['HOME']
'''
- `log = core.getLogger()` : Initializes the logger for the controller to record messages and
debug information.

- `delayFile = "%s/pox/pox/misc/delay.csv" % os.environ['HOME']` : Constructs the file
path for the delay file, assuming it's located in a directory relative to the user's home
directory.
```

Data Structures

```
```python
```

```
hosts = {'h1': 's1', 'h2': 's5'}
```

```
```
```

- `hosts`: Dictionary mapping host names to their connected switch names.

```
```python
```

```
linkNames = {
```

```
 'g': ('s1', 's2'),
```

```
 'h': ('s1', 's3'),
```

```
 'i': ('s2', 's4'),
```

```
 'j': ('s4', 's5'),
```

```
 'k': ('s3', 's5'),
```

```
}
```

```
```
```

- `linkNames`: Maps link names to tuples of connected switch names.

```
```python
```

```
portMappings = {
```

```
 's1': {'h1': 1, 's2': 2, 's3': 3},
```

```
 's2': {'s1': 1, 's4': 2},
```

```
 's3': {'s1': 1, 's5': 2},
```

```
 's4': {'s2': 1, 's5': 2},
```

```
 's5': {'h2': 1, 's4': 2, 's3': 3},
```

```
}
```

```
```
```

- `portMappings`: Dictionary mapping switches to their ports connecting to other switches or hosts.

Dijkstra Class Definition

```
```python
class Dijkstra(EventMixin):
 ...

- `class Dijkstra(EventMixin)` : Defines the `Dijkstra` class which inherits from
`EventMixin` for event handling.

```python
    def __init__(self):
        self.listenTo(core.openflow)
        log.debug("Enabling Dijkstra Module")
    ...

- `def __init__(self):` : Initializes the class.

- `self.listenTo(core.openflow)` : Listens to OpenFlow events from the core POX module.

- `log.debug("Enabling Dijkstra Module")` : Logs a debug message indicating that the
Dijkstra module is being enabled.

```python
 self.delays = {}
 self.switches = set()
 self.neighbors = defaultdict(set)
 ...

- `self.delays` : Dictionary to store delays between switches.

- `self.switches` : Set to keep track of switch names.

- `self.neighbors` : `defaultdict` to store neighbors of each switch.

```python
    # Read the delay file to initialize the topology
    with open(delayFile, 'r') as csvfile:
        links = csv.reader(csvfile)
```

```

        next(links) # skip the header
    for linkName, delay in links:
        s1, s2 = linkNames[linkName]
        self.delays[(s1, s2)] = int(delay)
        self.delays[(s2, s1)] = int(delay)
        self.switches.add(s1)
        self.switches.add(s2)
        self.neighbors[s1].add(s2)
        self.neighbors[s2].add(s1)
'''

```

- Reads the delay file to initialize the topology:
- Opens the delay file.
- Reads the CSV file, skipping the header.
- For each link, it retrieves the switch pair and delay, storing them in `self.delays`.
- Updates the set of switches and neighbors accordingly.

```

'''python
def _dijkstra(self, source):
    distances = defaultdict(lambda: float('inf'))
    distances[source] = 0
    previous = {}
    unseen = deepcopy(self.switches)

    while unseen:
        u = min(unseen, key=lambda x: distances[x])
        unseen.remove(u)
        for v in self.neighbors[u]:
            alt = distances[u] + self.delays[(u, v)]
            if alt < distances[v]:
                distances[v] = alt
                previous[v] = u
    return distances, previous
'''

```

'''

- `def _dijkstra(self, source):` : Implements Dijkstra's algorithm for shortest path calculation:

- Initializes distances with infinity and sets the source distance to 0.
- Keeps track of previous nodes and unseen nodes.
- Updates distances and previous nodes based on the shortest paths found.
- Returns the shortest distances and the previous nodes for path reconstruction.

```python

```
def _getPortMapping(self, source):
 distances, previous = self._dijkstra(source)
 ports = {}
 for destHost, destSwitch in hosts.items():
 if source == destSwitch:
 ports[destHost] = portMappings[source][destHost]
 continue
 while source != previous[destSwitch]:
 destSwitch = previous[destSwitch]
 ports[destHost] = portMappings[source][destSwitch]
 return ports
```

'''

- `def \_getPortMapping(self, source):` : Determines the port mappings for forwarding packets based on the shortest path from the source switch.

- Calls `\_dijkstra` to get distances and previous nodes.
- Constructs the port mappings for each destination host based on the computed shortest path.

```python

```
def _handle_ConnectionUp(self, event):
    switch = 's' + str(event.dpid)
```

```

ports = self._getPortMapping(switch)
for host, (ip, mac) in hosts.items():
    port = ports[host]

    # Install MAC-based flow
    msg_mac = of.ofp_flow_mod()
    msg_mac.match.dl_dst = EthAddr(mac)
    msg_mac.actions.append(of.ofp_action_output(port=port))
    event.connection.send(msg_mac)

    # Install IP-based flow
    msg_ip = of.ofp_flow_mod()
    msg_ip.match.nw_dst = IPAddr(ip)
    msg_ip.match.dl_type = 0x800 # IPv4
    msg_ip.actions.append(of.ofp_action_output(port=port))
    event.connection.send(msg_ip)

    log.debug("Dijkstra installed on %s", dpidToStr(event.dpid))
'''

- ``def _handle_ConnectionUp(self, event):``: Handles the event when a switch connects to
the controller:

- Retrieves the switch name from the event.

- Gets the port mappings using ``_getPortMapping``.

- Installs flow entries for each host based on MAC and IP addresses.

- Sends flow modification messages to the switch to set up MAC and IP-based flows.

- Logs a debug message indicating the Dijkstra module setup on the switch.

'''python
def launch():
    core.registerNew(Dijkstra)
'''

```

- `def launch():` : Entry point for the POX controller. Registers a new instance of the `Dijkstra` class with the core POX system, enabling the Dijkstra-based routing functionality.

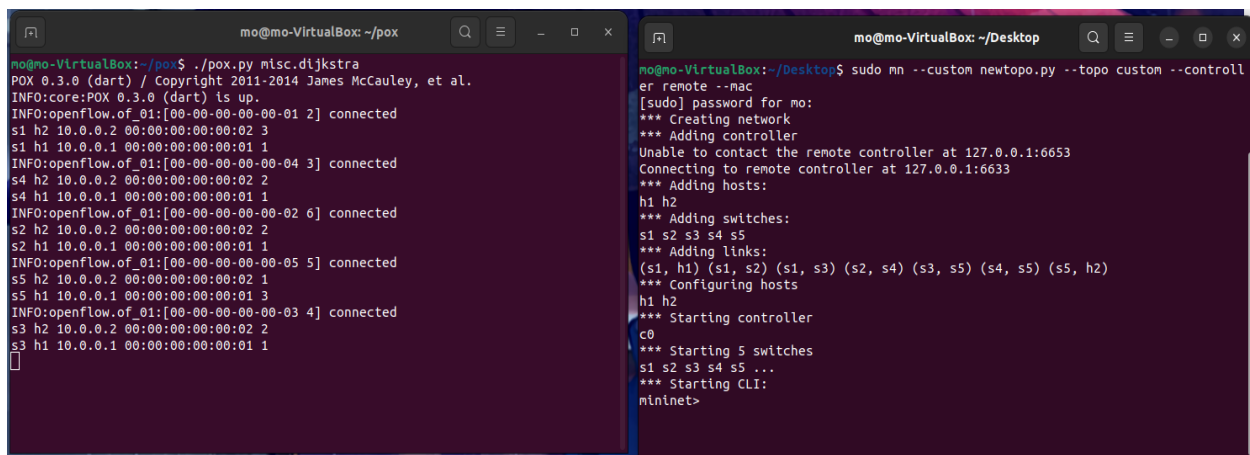
Summary

This script sets up a POX controller module that implements Dijkstra's shortest path algorithm to determine routing within a network. It reads delay information from a CSV file, computes the shortest paths between switches, and installs flow rules to handle packet forwarding based on those paths.

Now run the pox controller using this command from the pox directory

```
$ ./pox misc.dijkstra
```

This should be the output when running the topo and the controller :



```
mo@mo-VirtualBox: ~/pox
mo@mo-VirtualBox:~/pox$ ./pox.py misc.dijkstra
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
s1 h2 10.0.0.2 00:00:00:00:00:02 3
s1 h1 10.0.0.1 00:00:00:00:00:01 1
INFO:openflow.of_01:[00-00-00-00-00-04 3] connected
s4 h2 10.0.0.2 00:00:00:00:00:02 2
s4 h1 10.0.0.1 00:00:00:00:00:01 1
INFO:openflow.of_01:[00-00-00-00-00-02 6] connected
s2 h2 10.0.0.2 00:00:00:00:00:02 2
s2 h1 10.0.0.1 00:00:00:00:00:01 1
INFO:openflow.of_01:[00-00-00-00-00-05 5] connected
s5 h2 10.0.0.2 00:00:00:00:00:02 1
s5 h1 10.0.0.1 00:00:00:00:00:01 3
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
s3 h2 10.0.0.2 00:00:00:00:00:02 2
s3 h1 10.0.0.1 00:00:00:00:00:01 1
□

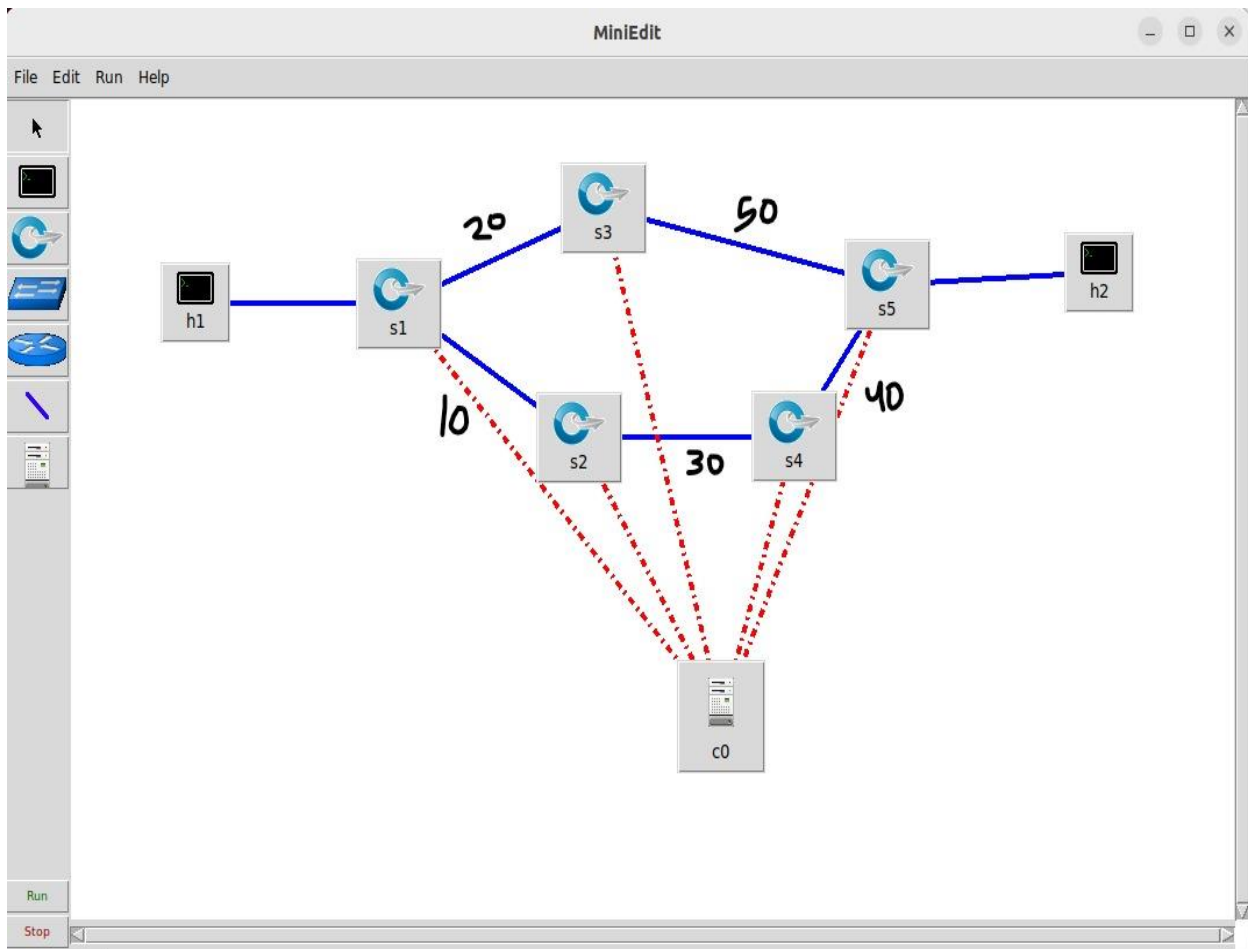
mo@mo-VirtualBox: ~/Desktop
mo@mo-VirtualBox:~/Desktop$ sudo mn --custom newtopo.py --topo custom --controller remote --mac
[sudo] password for mo:
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(s1, h1) (s1, s2) (s1, s3) (s2, s4) (s3, s5) (s4, s5) (s5, h2)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
mininet>
```

Now testing the reachability between the two hosts and see the bandwidth in the topology also see the links, dump and the nodes as follows:

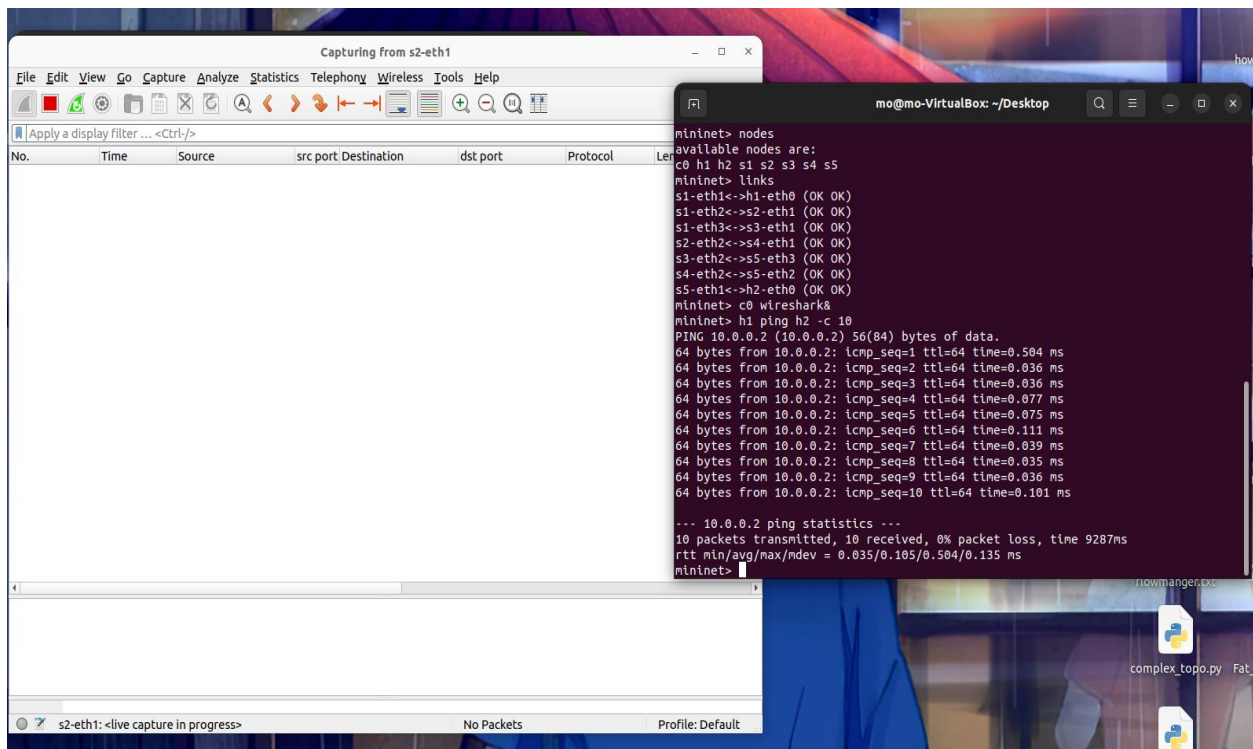
```
mo@mo-VirtualBox: ~/Desktop
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['37.4 Gbits/sec', '37.2 Gbits/sec']
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=25091>
<Host h2: h2-eth0:10.0.0.2 pid=25093>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=25098>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None pid=25101>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None pid=25104>
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None pid=25107>
<OVSSwitch s5: lo:127.0.0.1,s5-eth1:None,s5-eth2:None,s5-eth3:None pid=25110>
<RemoteController c0: 127.0.0.1:6633 pid=25083>
mininet> nodes
available nodes are:
c0 h1 h2 s1 s2 s3 s4 s5
mininet> links
s1-eth1<->h1-eth0 (OK OK)
s1-eth2<->s2-eth1 (OK OK)
s1-eth3<->s3-eth1 (OK OK)
s2-eth2<->s4-eth1 (OK OK)
s3-eth2<->s5-eth3 (OK OK)
s4-eth2<->s5-eth2 (OK OK)
s5-eth1<->h2-eth0 (OK OK)
mininet>
```

it gives a great result as we see

Notic the costs on each link in this topo as follows:

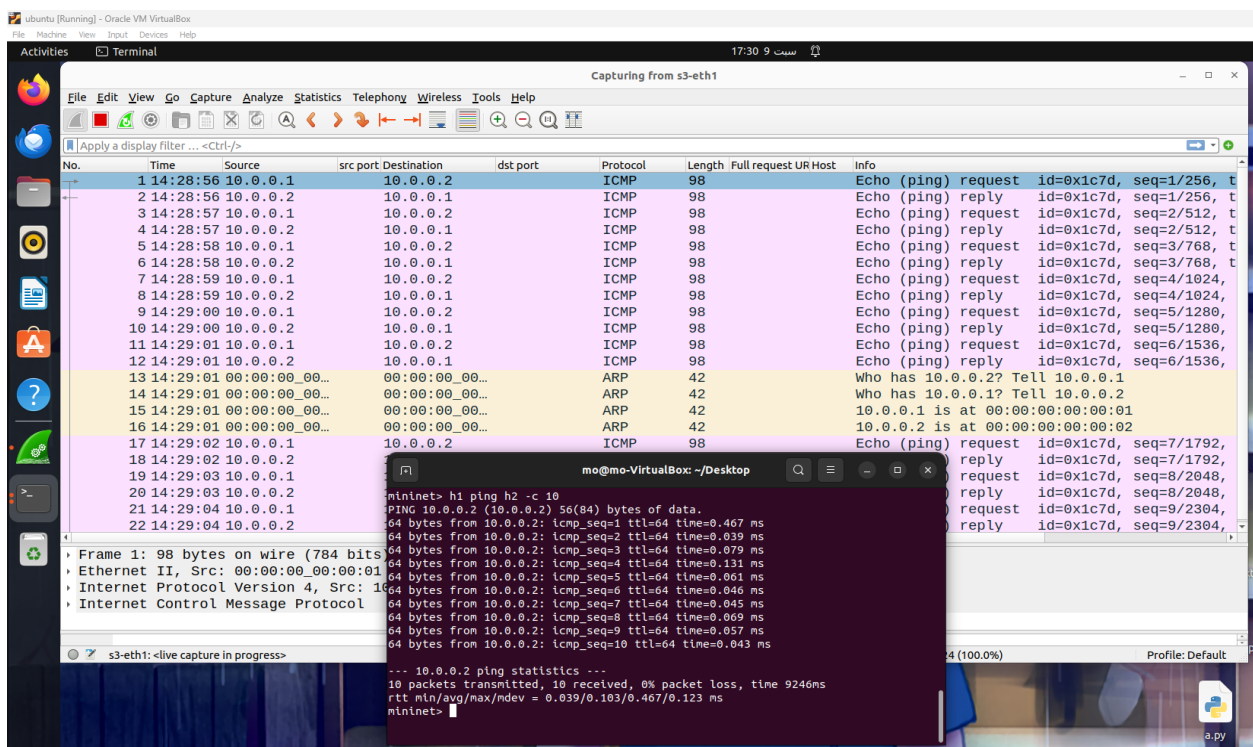


Now let's check on the Dijkstra algorithm if it works or not by capturing on one of the OpenvSwitches that the traffic should not bath through (like s2) because it has the higher cost or weight of the whole path ($10+30+40=80$) from h1 to h0(notice that the cost on the link that contains a host = 0).we will capture on s2 while pinging from h1 to h2 as follows :



As we see no icmp or arp packets sent from h1 to h2 or vice versa

Now let's do the same thing and capturing on one of the nodes that the traffic should both through (like s3) because it has the lowest total link summation ($20+50=70$) as follows:



As expected from the Dijkstra algorithm logic, the traffic chooses the shortest path that has a lower total link cost as we see when capturing on s3.

Best wishes ^_^