Programming Assignment 3: Minimum Diameter Spanning Tree in Open vSwitches

Due: 11:59pm, April 9th (by timestamps in gitlab)
Early submission will receive 1% bonus points per 24hr ahead of the deadline.

A switching loop occurs in computer networks when there is more than link layer path between two endpoints (Figure 1), e.g., multiple connections between two switches or two ports on the same switch.

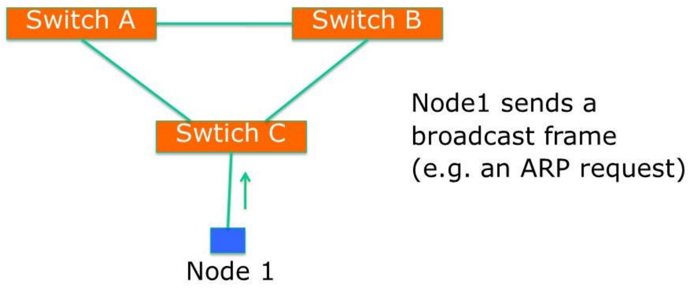


Figure 1 switching loop simple scenario

A switching loop creates broadcast storms as broadcast and multicast messages are (repeatedly) flooded by switches to every port other than the one the message arrives. In traditional networks, to create a loop-free logical topology, the spanning tree protocol (STP) runs distributed and blocks ports on each LAN segments other than the designated port.

Open vSwitch is a software implementation of a virtual multilayer network switch, designed to enable effective network automation through programmatic extensions. In software defined networks (SDN), by exploiting the topology awareness of SDN controllers, a centralized solution can be implemented. In particular, the controller calculates a spanning tree after receiving the adjacency information from all the vSwitches. It then sends an openflow command packet with OFPPC_NO_FLOOD to all the vSwitches in the resulting tree. This command sets the NO_FLOOD flag bit on the designated ports of the receiver vSwitches, to prevent broadcasting on those ports.

Problem Definition

The POX controller in Mininet implements STP to avoid switching loops. However, in STP, the root switch is elected based on identifiers of the switches. As a result, the resulting tree can have high network diameter (i.e., the maximum hop distance between two switches on the tree). A large network diameter leads to longer worst-case end-to-end latency in the LAN.

The goal of this programming assignment is to modify the current implementation of spanning tree algorithm in POX regarding the following objective function:

$$\mathbf{min}\ \{\ \mathbf{max}\ \{\mathbf{Shortest\ Path\ Length}\ (\boldsymbol{i}, \mathbf{j})\}\ \forall\ \mathbf{i}, \mathbf{j} \in \mathbf{Spanning\ Tree\ Vertices}\ \}$$

In other words, we want to construct a minimum diameter spanning tree. Graph diameter is the maximum shortest path length among all the node pairs of a given graph.

Environmental Setup

For this assignment, you will use the Mininet VM from Lab 3 and Lab 4. In the Gitlab directory, three python codes are provided.

- `PA3-topo.py`: a network topology file
- `skeleton.py`: implements a naïve DSTM algorithm. You need to complete two functions (`check_spanning_tree()` and `diameter()`) in the code. This code is self-contained and can be tested separately from Mininet.
- `spanning_tree.py`: implements a centralized spanning tree protocol. After verifying your implementation using `skeleton.py`, replace the functions `check_spanning_tree()` and `diameter()` in `spanning_tree.py`. Overwrite ~/pox/pox/openflow/spanning_tree.py and test it using `PA3-topo.py`. [We recommend that you back up the original spanning_tree.py]

To transfer the files from your host machine to the VM via *scp* from command line or a *sftp*/*Putty scp* client.

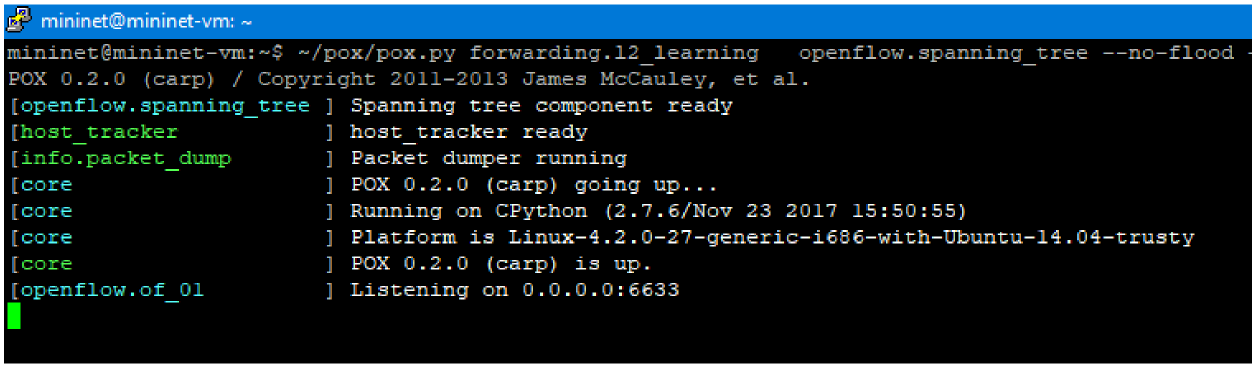
Start Mininet VM and two X-11 enabled ssh terminals to Mininet VM: One for running the controller; and the other for setting up an experimental network.

In the first terminal start the POX controller with the following command:

```
mininet@mininet-vm:~$ ~/pox/pox.py forwarding.l2_learning \
openflow.spanning_tree \
--no-flood --hold-down \
log.level --DEBUG samples.pretty_log openflow.discovery host_tracker info.packet_dump
```

More details about the above command and the arguments are provided in Appendix 1.

You should get the following output that confirms that the controller is running and waiting for incoming connections from vSwitches:



```
mininet@mininet-vm: ~
mininet@mininet-vm:~$ ~/pox/pox.py forwarding.l2_learning   openflow.spanning_tree --no-flood
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
[openflow.spanning_tree ] Spanning tree component ready
[host_tracker           ] host_tracker ready
[info.packet_dump       ] Packet dumper running
[core                   ] POX 0.2.0 (carp) going up...
[core                   ] Running on CPython (2.7.6/Nov 23 2017 15:50:55)
[core                   ] Platform is Linux-4.2.0-27-generic-i686-with-Ubuntu-14.04-trusty
[core                   ] POX 0.2.0 (carp) is up.
[openflow.of_01         ] Listening on 0.0.0.0:6633
```

In the second terminal, setup a remotely controlled network, consisting of 7 hosts and 7 vSwitches, forming the following topology:

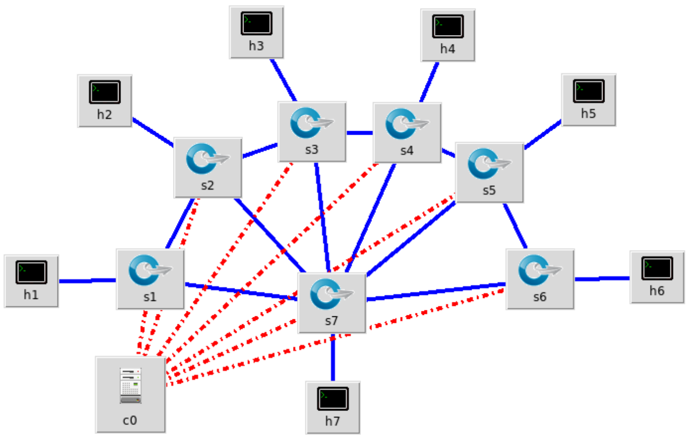
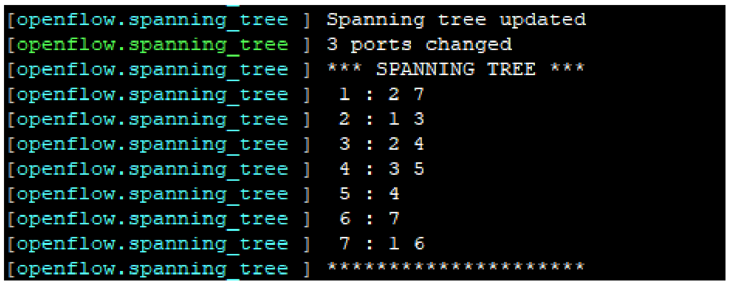


Figure 2 sample network topology

Use the provided python topology file PA3-topo.py and run:

```
mininet@mininet-vm:~$ sudo python PA3-topo.py
```

Waiting for some seconds, a list of updates for spanning tree will be shown in the controller side. Since POX get the first switch ID in its list (s1 for this example) as root the final result should be like this:



```
[openflow.spanning_tree ] Spanning tree updated
[openflow.spanning_tree ] 3 ports changed
[openflow.spanning_tree ] *** SPANNING TREE ***
[openflow.spanning_tree ]  1 : 2 7
[openflow.spanning_tree ]  2 : 1 3
[openflow.spanning_tree ]  3 : 2 4
[openflow.spanning_tree ]  4 : 3 5
[openflow.spanning_tree ]  5 : 4
[openflow.spanning_tree ]  6 : 7
[openflow.spanning_tree ]  7 : 1 6
[openflow.spanning_tree ] *********************
```

The result corresponds to the calculated spanning tree in Figure 3.

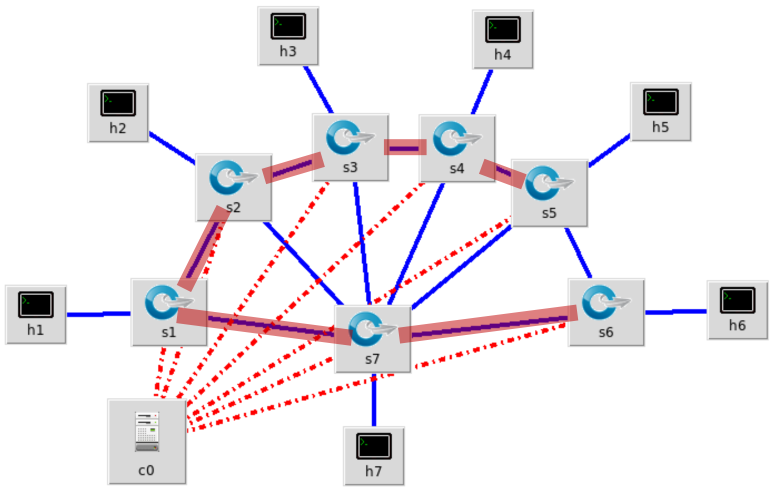


Figure 3 spanning tree build by current implementation of POX

Clearly, the diameter for the above tree is 6, namely, the hop distance between switches s5 and s6. This is inefficient for transferring data between h5 and h6.

In this assignment, you need to enhance the current spanning tree implementation by modifying the provided skeleton code "`spanning_tree.py`", which includes the current implementation of spanning tree calculation mechanism in POX.

If your implementation is correct, you should get the tree with diameter of 2 shown in Figure 4.

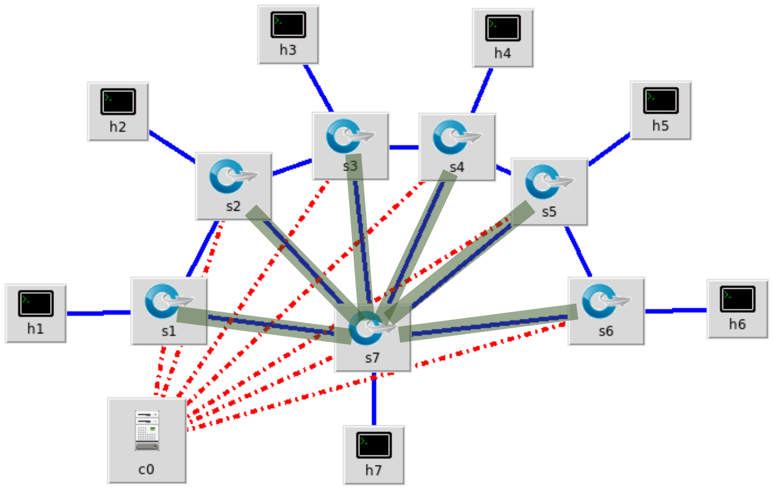


Figure 4 desired spanning tree

Understanding `spanning_tree.py`

Figure 5 shows the flowchart of `spanning_tree.py` since the moment vSwitches are connected to it until the time that the controller commands the vSwitches to configure their ports properly. It includes these general steps:

0. POX starts
    a. (function: `launch()`)
    b. (function: `start_spanning_tree ()`)

1. All vSwitches connect to the POX controller

2. POX keeps track of the connections from the vSwitches
    a. (function: `_handle_ConnectionUp (event)`)

3. POX fetches the topology of each vswitch
    a. (function: `_check_ports()`)

4. POX builds the adjacency graph and generate the spanning tree
    a. (function: `_update_tree (force_dpid = None)`)
    b. (function: `_calc_spanning_tree()`)
    c. --mdst → (function: `_calc_mdst ()`)
    d. --mdst → (function: `check_spanning_tree (sub_graph)`)

    e. --mdst → (function: `diameter (tree)`)

5. POX holds the whole network until the spanning tree is published
    a. (function: `_update_tree (force_dpid = None)`)

6. POX sends back the port policy to each individual vSwitch through their respective reserved connection
    a. (function: `_update_tree (force_dpid = None)`)

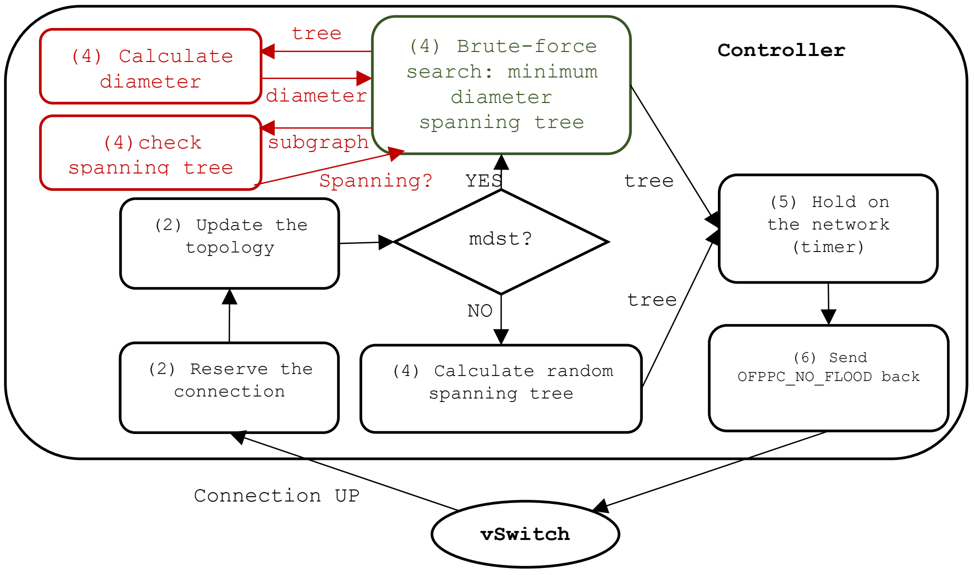
You need to modify the parts in Step 4 shown in red in Figure 5, including functions "`_calc_mdst()`" and "`_calc_diameter(tree)`".

function "`_calc_mdst()`": The adjacency matrix of network topology is constructed in the beginning. Find all possible subgraphs using brute-force search. Check if each subgraph is a spanning tree using function "`check_spanning_tree(sub_graph)`" and calculate the diameter of each tree using function "`diameter(tree)`". It returns the one with the minimum diameter as the result.

This function is provided in skeleton code.

function "`check_spanning_tree(graph G)`": It checks if the graph is loop-free (tree constraint) and if it was tree, does it cover all the nodes (spanning constraint)

function "`diameter(tree)`": Calculate the distances between all pair of nodes of the input tree using all-pairs shortest path algorithm and return the maximum one as the result.



(4) Calculate diameter
tree
diameter
(4) Brute-force search: minimum diameter spanning tree
Controller
(4) check spanning tree
subgraph
Spanning?
YES
tree
(2) Update the topology
mdst?
(5) Hold on the network (timer)
NO
tree
(2) Reserve the connection
(4) Calculate random spanning tree
(6) Send OFPPC_NO_FLOOD back
Connection UP
vSwitch

Figure 5 flowchart of switching loop handling in POX controller

Data Structure

One key data structure in `spanning_tree.py` and `skeleton.py` is the network graph to represent the connectivity among the switch.

The first data structure is a *list* variable denoted by *"nodes"*. It contains the ID of vertices of given graph.

The second data structure is a *dictionary*, denoted by *"G",* corresponds the adjacency of the input graph. Its *keys* are the vertices ID, listed in *"nodes"*. And its *values* are *list* of the neighbors ID of the corresponding key.

Tasks

We break down the tasks into two steps. In the first step, you complete the implementation of a naïve MDST algorithm. In the second step, you incorporate MDST into the spanning tree protocol of the POX controller.

STEP 1: Completing skeleton.py

A sample topology is included in skeleton.py for testing. A naive brute force solution is implemented that searches of all possible subgraphs in which each node connects to at least one edge. Your job is to verify whether the subgraph is:

- ✓ A spanning tree: by verifying our subgraph is loop free (tree) and covers all the nodes (spanning)
- ✓ And has minimum diameter

by completing two required functions:

- ➢ **`def check_spanning_tree (nodes, G)`**

Input:

- List *"nodes"* containing node IDs
- Dictionary *"G"* with (keys = node IDs) and (values = list of neighbor IDs for each node)

Returns:

- a Boolean value indicates if the graph *G* is a spanning tree (covering and loop-free) or not

- ➢ **`def diameter (nodes, G)`**

Input:

- List *"nodes"* containing node IDs
- Dictionary *"G"* with (keys = node IDs) and (values = list of neighbor IDs for each node)

Returns:

- The diameter of graph *G* as an integer number

To test your code, run it in **Python 2.7** with parameter **–mdst**. You should get this result:

```
c:\Python27\python.exe skeleton.py --mdst
*** SPANNING TREE ***
 1 : 7
 2 : 7
 3 : 7
 4 : 7
 5 : 7
 6 : 7
 7 : 1 2 3 4 5 6
*********************
```

Without --mdst parameter, it yields a default spanning tree.

STEP 2: Modifying the POX controller

Replace `check_spanning_tree()` and `diameter()` in `spanning_tree.py` and replace the `spanning_tree.py` in folder "`~/pox/pox/openflow/`. Rerun the POX controller with an additional argument

```
mininet@mininet-vm:~$ ~/pox/pox.py forwarding.l2_learning \
openflow.spanning_tree \
--no-flood --hold-down --mdist \
log.level --DEBUG samples.pretty_log openflow.discovery host_tracker info.packet_dump
```

Use the provided python topology file `PA3-topo.py` and run:

```
mininet@mininet-vm:~$ sudo python PA3-topo.py
```

Report

Upload the following items to the folder of programming assignment 3 in Gitlab:

1. **(60%)** Updated `skeleton.py` and `spanning_tree.py` files
2. Your report as a *.pdf* file including:
    a. **(20%)** Explain the algorithms you implemented to verify if a graph is a tree and to determine the diameter of a tree.
    b. **(10%)** The last resulted spanning tree shown in POX terminal (running with **`--mdst`** parameter) when you run the topology `PA3-test.py`
    c. **(10%)** From the outputs of Ping command, compare the histograms of round trip time between H5 and H6 when run POX with and without **`--mdst`** option.
3. **(Bonus 15 marks)** Implement a more efficient polynomial time complexity algorithm to determine a minimum diameter spanning tree in `spanning_tree.py.` Document the flow diagram of the algorithm and its complexity.

Appendix 1 - POX execution command line

Below is a list of the stock POX components we will used in this tutorial, with explanations of what each component does. To learn about other POX components, see the POX component documentation.

forwarding.l2_learning

The *L2 Learning* component makes OpenFlow switches act like Ethernet learning switches. It learns Ethernet MAC addresses, and matches all fields in the packet header so it may install multiple flows in the network for each pair of MAC addresses. For example, different TCP connections will result in different flows being installed.

This L2 Learning component creates flows with idle timeout values so switches will automatically delete flows that have not serviced packets during the timeout period (which is ten seconds). This helps keep flow tables manageable.

openflow.discovery

The *OpenFlow Discovery* component uses LLDP messages sent to and received from OpenFlow switches to discover the network topology. It also detects when network links go up or down. This information may be used by other components.

openflow.spanning_tree --no-flood --hold-down

The *Spanning Tree* component is required in cases where the topology of the network contains loops. It works with the OpenFlow Discovery component to build a view of the network topology and constructs a spanning tree by disabling flooding on switch ports that aren't on the tree. The options *no-flood* and *hold-down* are used to ensure no packets are flooded in the network before the component creates the spanning tree.

The *Spanning Tree* component will respond to changes in the network topology. If a link is broken, and if an alternate link exists, it can maintain connectivity in a network by creating a new tree that enables flooding on the ports connected to the alternate link.

When using the *Spanning Tree* component, also use a forwarding component that creates flows that have a timeout value set. In this example, we used the *L2 Learning* component.

host_tracker

The *Host Tracker* component attempts to keep track of hosts in the network. Host Tracker examines messages received by POX and learns MAC and IP of hosts in the network. Host Tracker will work in our example but it relies on packets coming to the controller. Packet forwarding in the network must be done reactively so we need to use a forwarding component like *forwarding.l2_learning*.

info.packet_dump

The *Packet Dump* component will display on the log console information about data packets received by POX from switches. This will help us see how the switches interact with the POX controller without running *tcpdump*.

log.level --DEBUG

The *Log Level* component allows the POX user to specify the amount of detail they will see in the log information produced by POX. The most detailed level is *DEGUG*, so we will use that in our example. By default, logs appear on the POX console so you will see them on the same terminal session used to start POX.

samples.pretty_log

The *Pretty Log* component formats log messages into a custom log format to provide attractive and readable log output on the POX console.

Appendix 2 - POX controller coding structure

When the application is started, the *launch()* function is automatically invoked. This is the function where the application registers all event listeners, and creates objects of any application class. For instance, the tutorial_l2_hub has the following *launch()* function:

```
def launch ():
    def start_switch (event):
        L2Hub(event.connection)

    core.openflow.addListenerByName("ConnectionUp", start_switch)
```

The above function instantiates a new L2Hub object for every new switch that connects to it.

Listening to events

There are two common ways for an application to register with the controller for events.

1. Register callback functions for specific events thrown by either the OpenFlow handler module or specific modules like Topology Discovery
   - From the launch or from the init of a class, perform *core.openflow.addListenerByName("EVENTNAME", CALLBACK_FUNC, PRIORITY)*
2. Register object with the OpenFlow handler module or specific modules like Topology Discovery
   - From typically the init of a class, perform *addListeners(self)*. Once this is added, the controller will look for a function with the name *_handle_EVENTNAME(self, event)*. This method is automatically registered as an event handler.

Parsing packets

POX provides several primitives to parse well-known packets. The ethernet packet received through a *packet_in* event can be extracted using this command and then the match for the flow rules can be created using the *from_packet* function:

```
packet = event.parsed
src_mac = packet.src
dst_mac = packet.dst
if packet.type == ethernet.IP_TYPE:
    ipv4_packet = event.parsed.find("ipv4")
    # Do more processing of the IPv4 packet
    src_ip = ipv4_packet.srcip
    src_ip = ipv4_packet.dstip
```

POX provides a simple way to construct the match headers from an incoming packet:

```
match = of.ofp_match.from_packet(packet)
```

Sending messages to switch

Sending packet_out, flow_mod and other OpenFlow messages to program the switch is simple with POX. The steps are to construct the appropriate message object and populate it with the decision fields and the send to the switch using the connection object *self.connection.send()*. Here is an example from the hub implementation:

```
        msg = of.ofp_packet_out()              # Create packet out
message
        msg.buffer_id = event.ofp.buffer_id    # Use the incoming packet
as the data for the packet out
        msg.in_port = packet_in.in_port        # Set the in_port so that
the switch knows
        msg.match = of.ofp_match.from_packet(packet)

        # Add an action to send to the specified port
        action = of.ofp_action_output(port = of.OFPP_FLOOD)
        msg.actions.append(action)

        # Send message to switch
        self.connection.send(msg)
```

Here is a list of events supported by POX:

Events generated from messages from switch	Packets parsed by pox/lib
FlowRemoved, FeaturesReceived	
ConnectionUp, FeaturesReceived	arp, dhcp, dns
RawStatsReply, PortStatus	eapol, eap
PacketIn, BarrierIn, SwitchDescReceived,	ethernet, icmp
FlowStatsReceived, QueueStatsReceived,	igmp, ipv4
AggregateFlowStatsReceived,	llc, lldp, mpls
TableStatsReceived, PortStatsReceived	rip, tcp, udp, vlan

Also see `*grep class pox/pox/openflow/libopenflow_01.py*` for a list of messages that the application can send to the switch.