网络算法基础项目一报告

一、 项目要求

1. 在 Mininet 上搭建一个 20 个节点网络（拓扑给定），每个网络节点下挂一个主机；

2. 使用 Ryu 连接 Mininet 中的交换机；

3. 并将拓扑读出来进行可视化展示；

4. 在 Ryu 上实现深度优先遍历算法，并找出任意两个主机间的最短路和最长路；

5. 使用最长路来配置任意两个主机间的通信连接

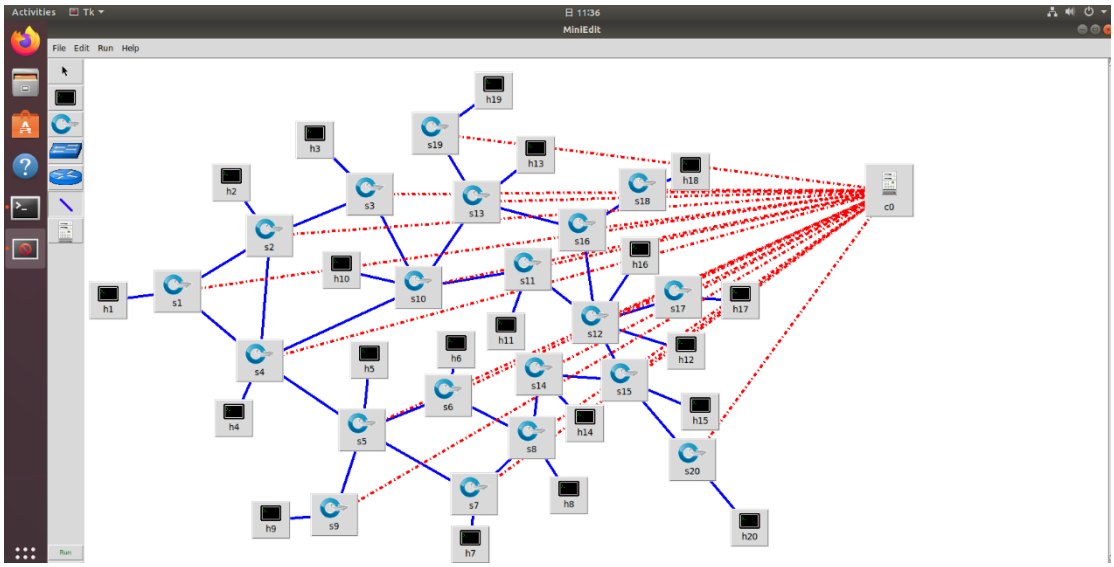
6. 将配置通的业务在可视化平台上进行展示

二、 算法思路

在 mininet 上通过 miniedit 可视化工具搭建特定的拓扑网络，并使用 ryu 控制器连接 mininet 上的交换机，在 ryu 上使用 DFS 算法探索两个节点之间的所有路径，通过比较得到最长路和最短路，用其配置两个主机间的通信连接，并使用 networkx 将其在可视化平台上展示。

三、 拓扑网络的搭建

1. 使用 mininet 自带的 miniedit 可视化工具搭建拓扑网络



2. 将搭建完成的拓扑网络保存为 py 文件，并将其打开，分别调用 net、nodes、links、dump 对于拓扑网络查看链路信息、可用节点信息、链路是否能正常工作和可用节点信息

```
s20 lo:  s20-eth1:s15-eth1 s20-eth2:h20-eth0
s16 lo:  s16-eth1:s13-eth2 s16-eth2:s12-eth4 s16-eth3:s18-eth1
s11 lo:  s11-eth1:s10-eth3 s11-eth2:s12-eth1 s11-eth3:h11-eth0
s1 lo:  s1-eth1:s2-eth1 s1-eth2:s4-eth1 s1-eth3:h1-eth0
s8 lo:  s8-eth1:s6-eth2 s8-eth2:s7-eth2 s8-eth3:s14-eth2 s8-eth4:h8-eth0
s12 lo:  s12-eth1:s11-eth2 s12-eth2:s15-eth3 s12-eth3:s17-eth1 s12-eth4:s16-eth2
 s12-eth5:h16-eth0 s12-eth6:h12-eth0
s17 lo:  s17-eth1:s12-eth3 s17-eth2:h17-eth0
s14 lo:  s14-eth1:s15-eth2 s14-eth2:s8-eth3 s14-eth3:h14-eth0
s5 lo:  s5-eth1:s4-eth4 s5-eth2:s9-eth1 s5-eth3:s6-eth1 s5-eth4:s7-eth1 s5-eth5:
h5-eth0
s9 lo:  s9-eth1:s5-eth2 s9-eth2:h9-eth0
s10 lo:  s10-eth1:s3-eth2 s10-eth2:s4-eth3 s10-eth3:s11-eth1 s10-eth4:s13-eth1 s
10-eth5:h10-eth0
s18 lo:  s18-eth1:s16-eth3 s18-eth2:h18-eth0
s13 lo:  s13-eth1:s10-eth4 s13-eth2:s16-eth1 s13-eth3:s19-eth1 s13-eth4:h13-eth0
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s4-eth2 s2-eth3:s3-eth1 s2-eth4:h2-eth0
s15 lo:  s15-eth1:s20-eth1 s15-eth2:s14-eth1 s15-eth3:s12-eth2 s15-eth4:h15-eth0
c0
mininet> nodes
available nodes are:
c0 h1 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h2 h20 h3 h4 h5 h6 h7 h8 h9 s1 s10
 s11 s12 s13 s14 s15 s16 s17 s18 s19 s2 s20 s3 s4 s5 s6 s7 s8 s9
mininet>
```

3. 将搭建的拓扑网络保存为 python 文件

四、 控制器 python 文件的编写

```
from collections import defaultdict
from ryu.base import app_manager
from ryu.controller import ofp_event
```

```
from ryu.topology import event
from ryu.controller.handler import MAIN_DISPATCHER,
CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.topology.api import get_switch, get_all_link,
get_link
import copy
import random
from ryu.lib.packet import arp
from ryu.lib.packet import ipv6
from ryu.lib import mac

import os
import networkx as nx
import matplotlib.pyplot as plt

def draw_graph(graph, path, index, head=0, rear=0):
    """画图"""
    plt.figure()
    #从图中提取节点
    nodes = set([n1 for n1, n2 in graph] + [n2 for n1, n2 in
graph])
    #定义图表
    gra = nx.Graph()

    for node in nodes:
        gra.add_node(node)

    gra.add_edges_from(graph, color='b')
    gra.add_edges_from(path, color='r')

    edges = gra.edges()
    colors = [gra[u][v]['color'] for u, v in edges]

    nx.draw(gra, with_labels=True, edge_color=colors)

    #判断是否有名为photos的文件夹用于存储图片，若没有则创建一个
    if not os.path.exists("./photos/"):
        os.mkdir("./photos/")
    plt.savefig('./photos/' + 'index' + str(index) + ":" +
```

```
str(head) + "-" + str(rear) + ".png")
    plt.savefig('now_photo.png')


class topo(object):
    """topo 类"""
    def __init__(self, logger):
        """对参数进行初始化"""
        self.adjacent = defaultdict(lambda s1s2:
None)#adjacent 存储两交换机间的接口信息以及边的权重;
        self.switches = None
        self.host_mac_to = {}
        self.logger = logger

    def reset(self):
        """将 topo 类的属性重置"""
        self.adjacent = defaultdict(lambda s1s2: None)
        self.switches = None
        self.host_mac_to = None

    def get_adjacent(self, s1, s2):
        """调用 adjacent"""
        return self.adjacent.get((s1, s2))

    def set_adjacent(self, s1, s2, port):
        """将 两 交 换 机 间 的 接 口 和 边权存入 adjacent 中"""
        self.adjacent[(s1, s2)] = port

    def findpath(self, beg, end, sign, onepath, allpaths):
        """通过 DFS 算法探索所有路径,在探索过程中，onepath 存储一条路
径，并在到达终点时将其作为一个元素存入 allpath 列表中"""
        if beg == end:#处理开始点与结束点相同的情况
            allpaths.append(onepath.copy())
        else:
            for u in self.switches:
                if (self.get_adjacent(beg, u) is not None) and
(sign[u] != 1):
                    sign[u] = 1
                    onepath.append(u)
                    self.findpath(u, end, sign, onepath,
allpaths)
                    onepath.remove(u)
                    sign[u] = 0
```

```
    def longest_path(self, beg, end, first_port, last_port):
        """得到最长路"""
        self.logger.info(
            "topo calculate the longest path from ---{}-{}----
---{}-{}".format(first_port, beg, end, last_port))
        self.logger.debug("there is {}
swithes".format(len(self.switches)))

        sign = {}
        for s in self.switches:
            sign[s] = 0
        sign[beg] = 1

        onepath = []
        onepath.append(beg)

        #使用之前的 findpath 方法找到两个节点之间所有路径并将其存在
allpaths 中
        allpaths = []
        self.findpath(beg, end, sign, onepath, allpaths)

        #打印出所有路径
        print("paths num is: {}".format(len(allpaths)))
        print("all paths:")
        sp = allpaths[0]
        lp = allpaths[0]
        for i in allpaths:
            if (len(i) > len(lp)):
                lp = i
            if (len(i) < len(sp)):
                sp = i
            print(i)

        #打印出最短路和最长路
        print("the shortest path is: ")
        print(sp)
        print("the longest path is: ")
        print(lp)

        if beg == end:
            path = [beg]
        else:
            path = lp
```

```
        #将两个交换机之间的输入和输出端口以 Ryu 能识别的方式记录在
record
        record = []
        inport = first_port

        # s1 s2; s2:s3, sn-1  sn
        for s1, s2 in zip(path[:-1], path[1:]):
            outport = self.get_adjacent(s1, s2)

            record.append((s1, inport, outport))
            inport = self.get_adjacent(s2, s1)

        record.append((end, inport, last_port))
        return record, lp


# TODO Port status monitor

class DFSController(app_manager.RyuApp):
    """控制器配置"""
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]#表明使用的
openflow 版本为 1.3

    def __init__(self, *args, **kwargs):
        """将控制器初始化处理"""
        super(DFSController, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        ## datapaths 为 OpenFlow 下的各个交换机
        self.datapaths = []

        self.topo = topo(self.logger)

        #于判断某交换机是否泛洪过特定的发送源 mac 到目标 mac
        self.flood_history = {}

        self.arp_table = {}
        self.rarp_table = {}
        self.arp_history = {}

        self.initshow = 0
        self.index = 1
        self.lp_path = []

    def _find_dp(self, dpid):
```

```
        for dp in self.datapaths:
            if dp.id == dpid:
                return dp
        return None

    #fp_event.EventOFPSwitchFeatures 事件到来且处于
CONFIG_DISPATCHER 阶段时触发方法 switch_features_handler
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        """交换机向控制器传输自身的 features 信息，并将优先级设为最低，
并添加到流表"""
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        #match 为流表的匹配域，在此初始化
        match = parser.OFPMatch()
        #actions 为流表动作
        actions =
[parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,

ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions,
buffer_id=None):
        """控制器对交换机添加流表"""
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                      actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath,
buffer_id=buffer_id,
                                    priority=priority,
match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath,
priority=priority,
                                    match=match,
```

```
instructions=inst)
        datapath.send_msg(mod)

    def configure_path(self, longest_path, event, src_mac,
dst_mac):
        #在两节点之间配置最长路
        msg = event.msg
        datapath = msg.datapath

        ofproto = datapath.ofproto

        parser = datapath.ofproto_parser

        for switch, inport, outport in longest_path:
            match = parser.OFPMatch(in_port=inport,
eth_src=src_mac, eth_dst=dst_mac)
            actions = [parser.OFPActionOutput(outport)]
            datapath = self._find_dp(int(switch))
            assert datapath is not None

            inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]

            mod = datapath.ofproto_parser.OFPFlowMod(
                datapath=datapath,
                match=match,
                idle_timeout=0,
                hard_timeout=0,
                priority=1,
                instructions=inst
            )
            datapath.send_msg(mod)

    #当交换机发送数据包给控制器时且在特性消息接受到后到断开连接前的阶段
触发
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, event):

        msg = event.msg

        #取出数据包内的信息
        datapath = msg.datapath
        ofproto = datapath.ofproto
```

```
        parser = datapath.ofproto_parser

        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)

        eth = pkt.get_protocols(ethernet.ethernet)[0]

        #对 LLDP 类型数据包不处理
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return

        #得到起点和终点的 mac 地址
        dst_mac = eth.dst
        src_mac = eth.src

        # 检查这是否是一个 arp 包
        arp_pkt = pkt.get_protocol(arp.arp)
        if arp_pkt:
            self.arp_table[arp_pkt.src_ip] = src_mac


        dpid = datapath.id

        self.mac_to_port.setdefault(dpid, {})

        self.mac_to_port[dpid][src_mac] = in_port

        self.flood_history.setdefault(dpid, [])
        #ipv6 协定的广播数据包的目的 MAC 地址以 33：33 开头，如果之前
控制器没有 flood 该数据包就进行记
        if '33:33' in dst_mac[:5]:
            if (src_mac, dst_mac) not in
self.flood_history[dpid]:
                self.flood_history[dpid].append((src_mac,
dst_mac))
            else:
                return

        if src_mac not in self.topo.host_mac_to.keys():
            self.topo.host_mac_to[src_mac] = (dpid, in_port)

        if dst_mac in self.topo.host_mac_to.keys():
```

```
            final_port = self.topo.host_mac_to[dst_mac][1]
            src_switch = self.topo.host_mac_to[src_mac][0]
            dst_switch = self.topo.host_mac_to[dst_mac][0]

            # 用 longest_path 函数计算最长路径
            longest_path, lp = self.topo.longest_path(
                src_switch,
                dst_switch,
                in_port,
                final_port)

            if lp not in self.lp_path:
                graph = []
                for a in self.topo.switches:
                    for b in self.topo.switches:
                        if self.topo.get_adjacent(a, b) is not
None:
                            graph.append((a, b))
                graph_path = []
                for i in range(len(lp) - 1):
                    graph_path.append((lp[i], lp[i + 1]))
                    graph_path.append((lp[i + 1], lp[i]))
                draw_graph(graph, graph_path, self.index,
lp[0], lp[-1])
                self.index += 1
                self.lp_path.append(lp)

            self.logger.info(
                "The longest path from {} to {} contains {}
switches".format(src_mac, dst_mac, len(longest_path)))

            assert len(longest_path) > 0

            #打印出两个节点之间的最长路
            path_str = ''

            for s, ip, op in longest_path:
                path_str = path_str + "--{}-{}-{}--
".format(ip, s, op)

            self.logger.info("The longset path from {} to {}
is {}".format(src_mac, dst_mac, path_str))

            self.logger.info("Have calculated the longest path
```

```
from {} to {}".format(src_mac, dst_mac))

            self.logger.info("Now configuring switches of
interest")

            self.configure_path(longest_path, event, src_mac,
dst_mac)

            self.logger.info("Configure done")

            out_port = None
            for s, _, op in longest_path:
                if s == dpid:
                    out_port = op

        else:
            if self.arp_handler(msg):
                return
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        data = None

        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(
            datapath=datapath,
            buffer_id=msg.buffer_id,
            in_port=in_port,
            actions=actions,
            data=data
        )
        datapath.send_msg(out)


    #拓扑发现
    @set_ev_cls(event.EventSwitchEnter)
    def switch_enter_handler(self, event):
        """交换机进入"""
        self.logger.info("A switch entered.Topology
rediscovery...")
        self.switch_status_handler(event)
```

```
        self.logger.info('Topology rediscovery done')

    @set_ev_cls(event.EventSwitchLeave)
    def switch_leave_handler(self, event):
        """交换机离开"""
        self.logger.info("A switch leaved.Topology
rediscovery...")
        self.switch_status_handler(event)
        self.logger.info('Topology rediscovery done')

    def switch_status_handler(self, event):
        """使用副本避免对 network 产生影响"""

        all_switches = copy.copy(get_switch(self, None))

        # 获取交换机的 ID 值
        self.topo.switches = [s.dp.id for s in all_switches]

        self.logger.info("switches
{}".format(self.topo.switches))

        self.datapaths = [s.dp for s in all_switches]

        all_links = copy.copy(get_link(self, None))

        all_link_stats = [(l.src.dpid, l.dst.dpid,
l.src.port_no, l.dst.port_no) for l in all_links]
        self.logger.info("Number of links
{}".format(len(all_link_stats)))

        all_link_repr = ''

        for s1, s2, p1, p2 in all_link_stats:
            self.topo.set_adjacent(s1, s2, p1)
            self.topo.set_adjacent(s2, s1, p2)

            all_link_repr += 's{}p{}--s{}p{}\n'.format(s1, p1,
s2, p2)
        self.logger.info("All links:\n " + all_link_repr)


        #将拓扑结构可视化展示。
        self.initshow += 1
        if self.initshow == 13:
```

```
            graph = []
            for a in self.topo.switches:
                for b in self.topo.switches:
                    if self.topo.get_adjacent(a, b) is not
None:
                        graph.append((a, b))
            graph_path = []
            draw_graph(graph, graph_path, self.index)
            self.index += 1

    def arp_handler(self, msg):
        """处理 arp 请求"""
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)

        eth = pkt.get_protocols(ethernet.ethernet)[0]
        arp_pkt = pkt.get_protocol(arp.arp)

        if eth:
            eth_dst = eth.dst
            eth_src = eth.src

        if eth_dst == mac.BROADCAST_STR and arp_pkt:
            arp_dst_ip = arp_pkt.dst_ip

            if (datapath.id, eth_src, arp_dst_ip) in
self.arp_history:

                if self.arp_history[(datapath.id, eth_src,
arp_dst_ip)] != in_port:
                    return True
            else:
                self.arp_history[(datapath.id, eth_src,
arp_dst_ip)] = in_port


        #在得到 arp 头参数后构建 arp 数据包
        if arp_pkt:

            hwtype = arp_pkt.hwtype
```

```
            proto = arp_pkt.proto
            hlen = arp_pkt.hlen
            plen = arp_pkt.plen

            opcode = arp_pkt.opcode

            arp_src_ip = arp_pkt.src_ip
            arp_dst_ip = arp_pkt.dst_ip

        #从 arp 请求和 arp 回复得到 arp table。
            if opcode == arp.ARP_REQUEST:
                if arp_dst_ip in self.arp_table:
                    actions = [parser.OFPActionOutput(in_port)]
                    arp_reply = packet.Packet()

                    arp_reply.add_protocol(ethernet.ethernet(
                        ethertype=eth.ethertype,
                        dst=eth_src,
                        src=self.arp_table[arp_dst_ip]))

                    arp_reply.add_protocol(arp.arp(
                        opcode=arp.ARP_REPLY,
                        src_mac=self.arp_table[arp_dst_ip],
                        src_ip=arp_dst_ip,
                        dst_mac=eth_src,
                        dst_ip=arp_src_ip))

                    arp_reply.serialize()
                    out = parser.OFPPacketOut(
                        datapath=datapath,
                        buffer_id=ofproto.OFP_NO_BUFFER,
                        in_port=ofproto.OFPP_CONTROLLER,
                        actions=actions, data=arp_reply.data)
                    datapath.send_msg(out)

                    return True
        return False
```

五、 结果展示

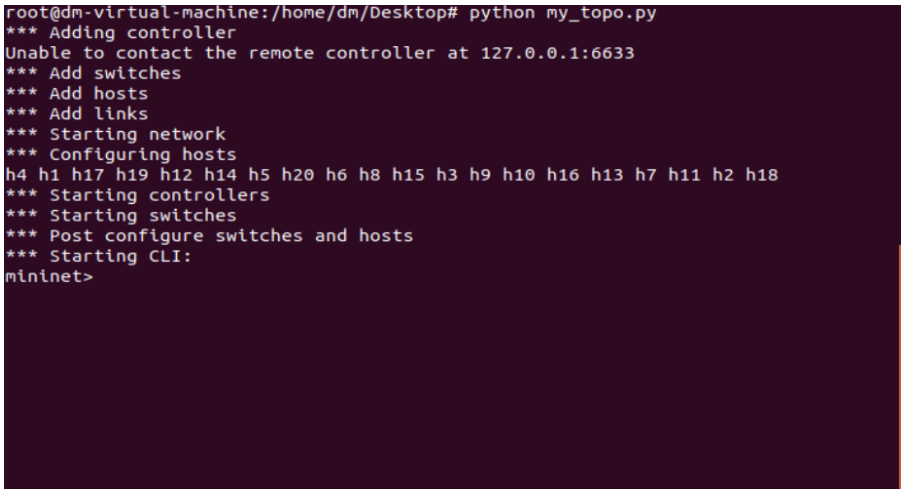
可视化生成的图片将保存在与 Dfscon.py 同级目录的 photos 文件夹里面，如果没有该文件夹，将自动创建一个，此外每次操作都会在 dfscon.py 同级目录下更新 new_photo

为了实现可视化，需要安装 networkx,numpy,matplotlib 三个包，并在终端输入以下命令更改默认 python 版本：

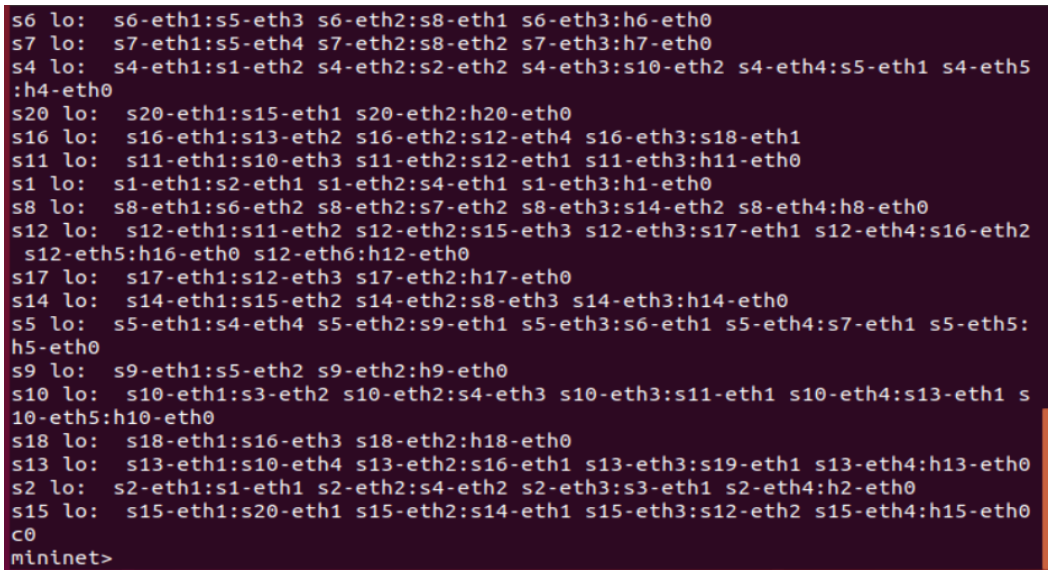
```
sudo update-alternatives --install /usr/bin/python python /usr/bin/python2 1
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 2
sudo update-alternatives --config python
```

（输入 2）

1. 启动 mininet,打开 my_topo.py 文件



2. 使用 net，nodes，links，dump 命令查看链路信息，节点，链路是否正常工作和节点信息

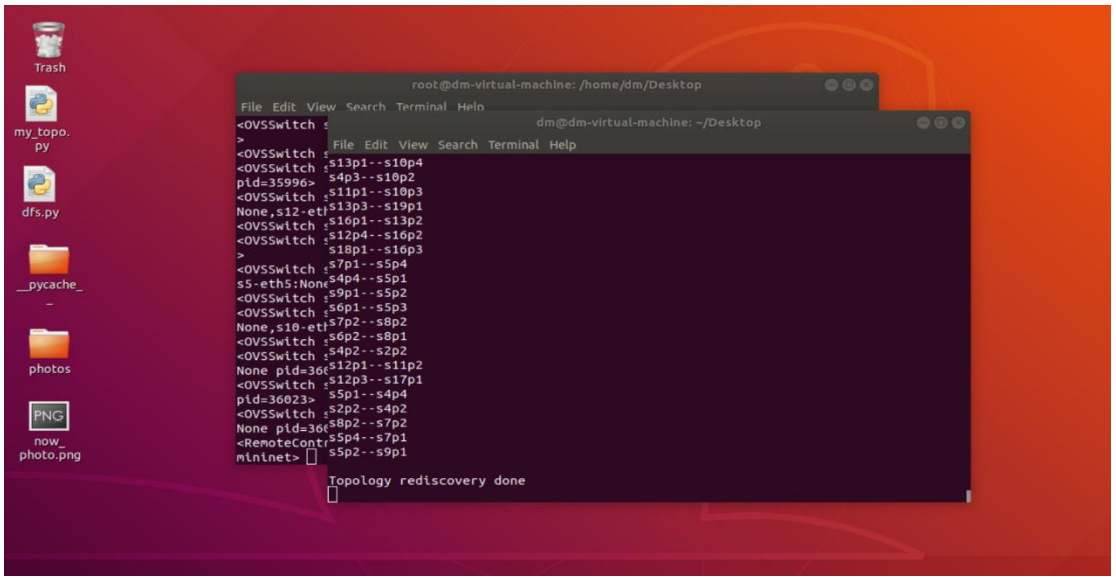


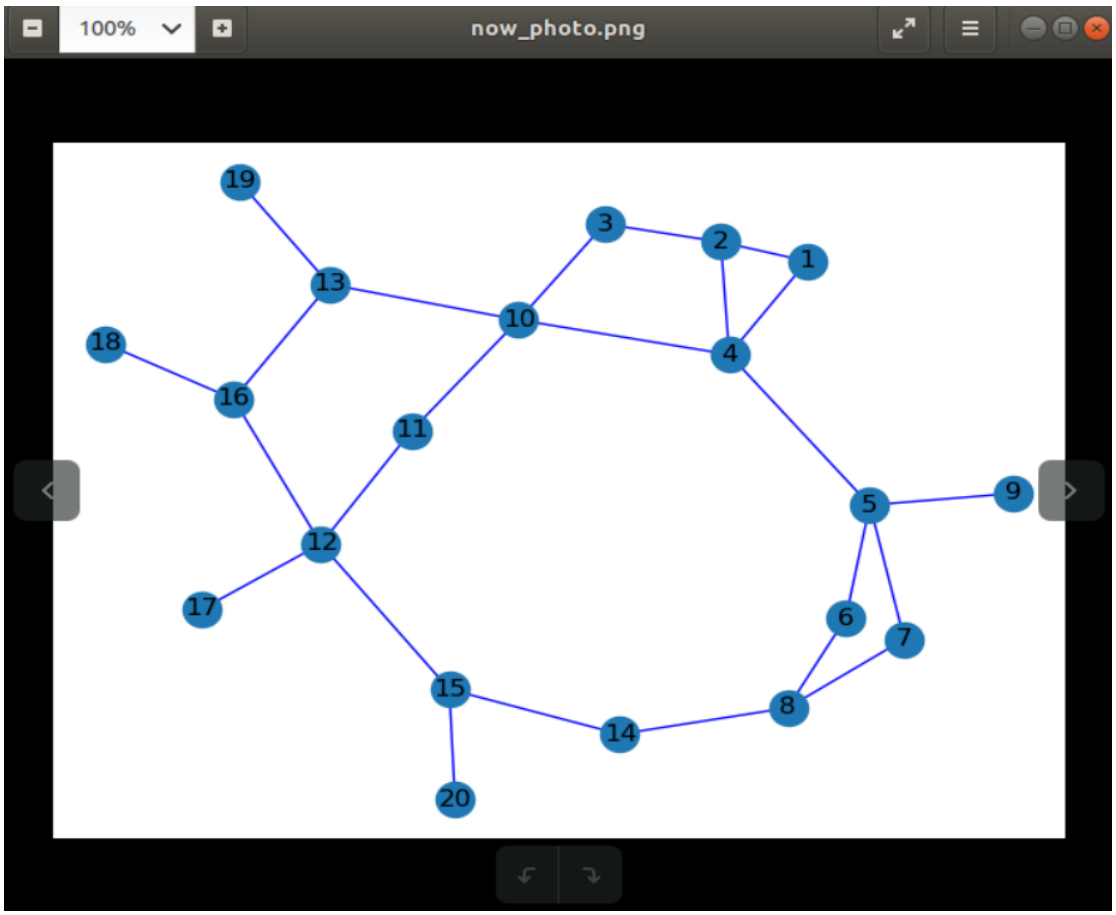
```
mininet> nodes
available nodes are:
c0 h1 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h2 h20 h3 h4 h5 h6 h7 h8 h9 s1 s10
 s11 s12 s13 s14 s15 s16 s17 s18 s19 s2 s20 s3 s4 s5 s6 s7 s8 s9
mininet>
```

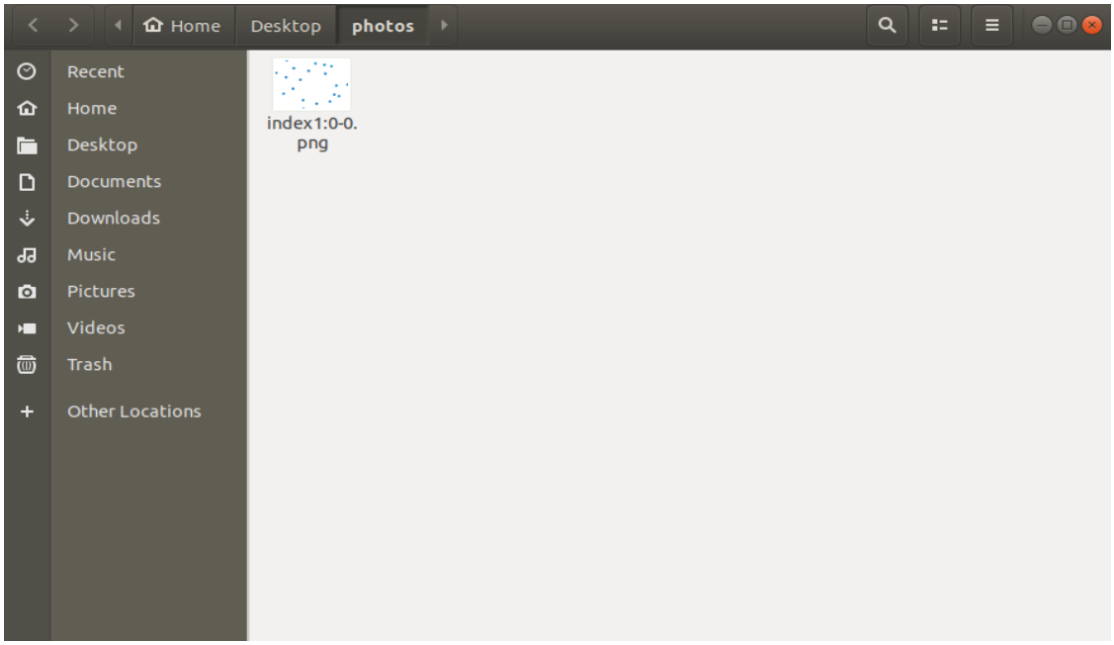
```
s16-eth2<->s12-eth4 (OK OK)
s16-eth3<->s18-eth1 (OK OK)
s13-eth3<->s19-eth1 (OK OK)
h1-eth0<->s1-eth3 (OK OK)
h2-eth0<->s2-eth4 (OK OK)
h3-eth0<->s3-eth3 (OK OK)
h19-eth0<->s19-eth2 (OK OK)
h13-eth0<->s13-eth4 (OK OK)
h18-eth0<->s18-eth2 (OK OK)
h16-eth0<->s12-eth5 (OK OK)
h17-eth0<->s17-eth2 (OK OK)
h12-eth0<->s12-eth6 (OK OK)
h15-eth0<->s15-eth4 (OK OK)
h14-eth0<->s14-eth3 (OK OK)
h8-eth0<->s8-eth4 (OK OK)
h20-eth0<->s20-eth2 (OK OK)
h7-eth0<->s7-eth3 (OK OK)
h11-eth0<->s11-eth3 (OK OK)
h6-eth0<->s6-eth3 (OK OK)
h5-eth0<->s5-eth5 (OK OK)
h9-eth0<->s9-eth2 (OK OK)
h4-eth0<->s4-eth5 (OK OK)
h10-eth0<->s10-eth5 (OK OK)
mininet>
```

```
<OVSSwitch s11: lo:127.0.0.1,s11-eth1:None,s11-eth2:None,s11-eth3:None pid=32708
>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=32711>
<OVSSwitch s8: lo:127.0.0.1,s8-eth1:None,s8-eth2:None,s8-eth3:None,s8-eth4:None
pid=32714>
<OVSSwitch s12: lo:127.0.0.1,s12-eth1:None,s12-eth2:None,s12-eth3:None,s12-eth4:
None,s12-eth5:None,s12-eth6:None pid=32717>
<OVSSwitch s17: lo:127.0.0.1,s17-eth1:None,s17-eth2:None pid=32720>
<OVSSwitch s14: lo:127.0.0.1,s14-eth1:None,s14-eth2:None,s14-eth3:None pid=32723
>
<OVSSwitch s5: lo:127.0.0.1,s5-eth1:None,s5-eth2:None,s5-eth3:None,s5-eth4:None,
s5-eth5:None pid=32726>
<OVSSwitch s9: lo:127.0.0.1,s9-eth1:None,s9-eth2:None pid=32729>
<OVSSwitch s10: lo:127.0.0.1,s10-eth1:None,s10-eth2:None,s10-eth3:None,s10-eth4:
None,s10-eth5:None pid=32732>
<OVSSwitch s18: lo:127.0.0.1,s18-eth1:None,s18-eth2:None pid=32735>
<OVSSwitch s13: lo:127.0.0.1,s13-eth1:None,s13-eth2:None,s13-eth3:None,s13-eth4:
None pid=32738>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None,s2-eth4:None
pid=32741>
<OVSSwitch s15: lo:127.0.0.1,s15-eth1:None,s15-eth2:None,s15-eth3:None,s15-eth4:
None pid=32744>
<RemoteController c0: 127.0.0.1:6633 pid=32680>
mininet>
```

3. 在Dfscon.py所在目录下打开一个终端调用ryu-manager Dfscon.py --observe-links，此时在Dfscon.py所在目录下生成一个图片(new_photo.png,后续操作会将此图片更新),若此目录没有文件夹photos将自动创建一个，用来保存生成的图片


Trash
my_topo.py
dfs.py
__pycache__
photos
now_photo.png
root@dm-virtual-machine: /home/dm/Desktop
File Edit View Search Terminal Help
dm@dm-virtual-machine: ~/Desktop
File Edit View Search Terminal Help
s13p1--s10p4
s4p3--s10p2
s11p1--s10p3
s13p3--s19p1
s16p1--s13p2
s12p4--s16p2
s18p1--s16p3
s7p1--s5p4
s4p4--s5p1
s9p1--s5p2
s6p1--s5p3
s7p2--s8p2
s6p2--s8p1
s4p2--s2p2
s12p1--s11p2
s12p3--s17p1
s5p1--s4p4
s2p2--s4p2
s8p2--s7p2
s5p4--s7p1
s5p2--s9p1
Topology rediscovery done


100%
now_photo.png
19
3
2
1
13
10
18
4
16
11
9
5
12
17
6
7
15
8
14
20



4. h1 ping h2，将会看到

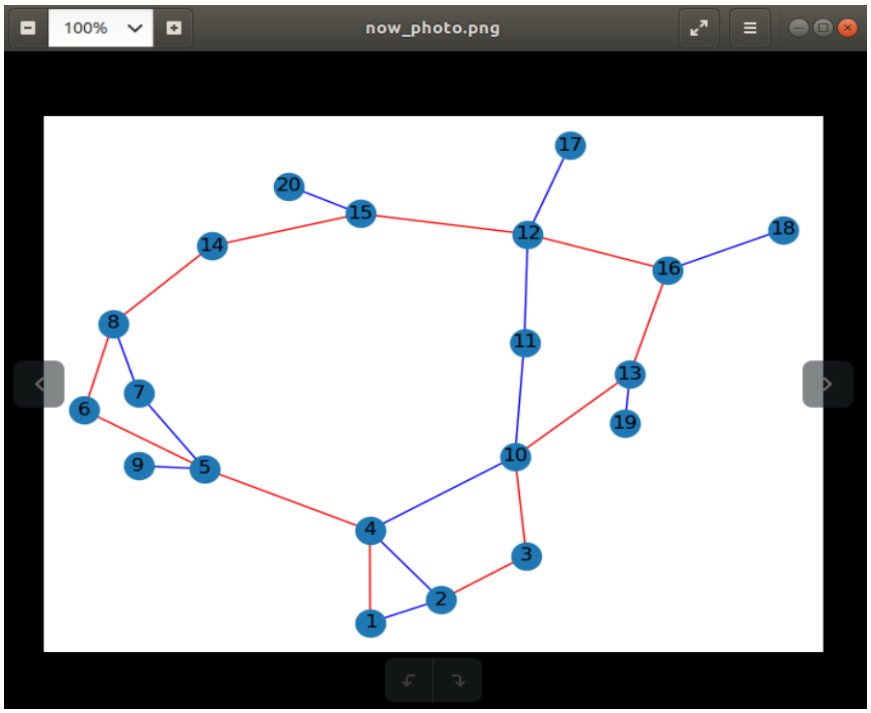
```
mininet> h1 ping h2 -c5
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=101 ms
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1145 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.419 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.180 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.224 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4059ms
rtt min/avg/max/mdev = 0.180/249.533/1145.484/449.682 ms, pipe 2
mininet>
```

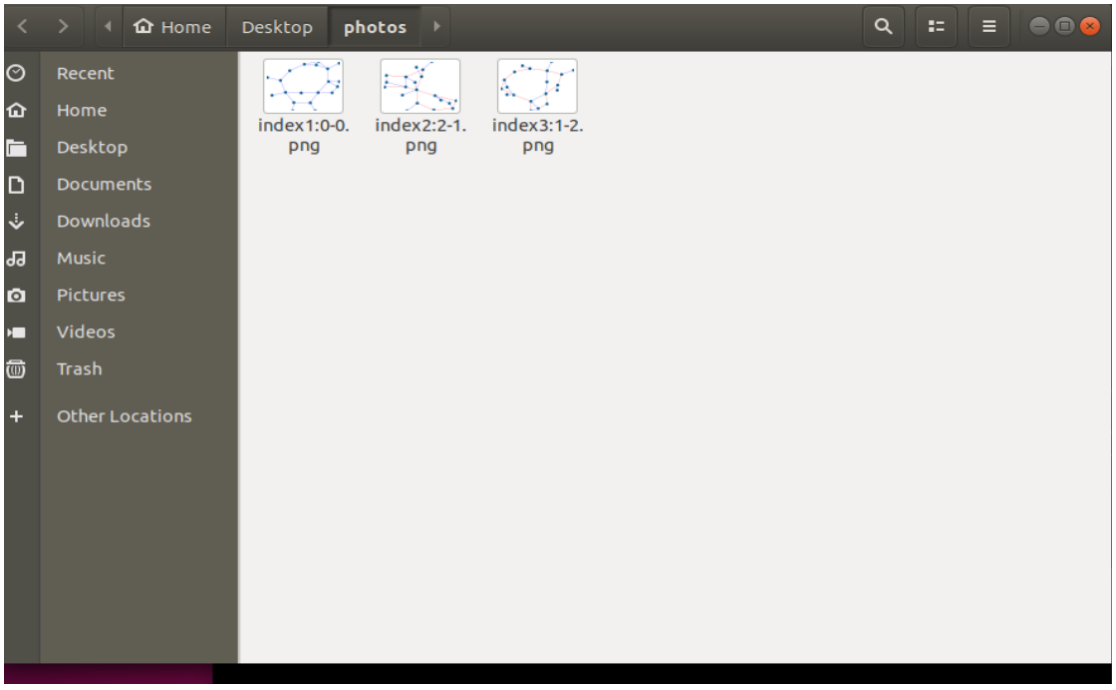
丢包率为 0

```
topo calculate the longest path from ---1-1-------2-4
paths num is: 7
all paths:
[1, 2]
[1, 4, 10, 3, 2]
[1, 4, 2]
[1, 4, 5, 6, 8, 14, 15, 12, 11, 10, 3, 2]
[1, 4, 5, 6, 8, 14, 15, 12, 16, 13, 10, 3, 2]
[1, 4, 5, 7, 8, 14, 15, 12, 11, 10, 3, 2]
[1, 4, 5, 7, 8, 14, 15, 12, 16, 13, 10, 3, 2]
the shortest path is:
[1, 2]
the longest path is:
[1, 4, 5, 6, 8, 14, 15, 12, 16, 13, 10, 3, 2]
The longest path from 00:00:00:00:00:01 to 00:00:00:00:00:02 contains 13 switche
s
The longset path from 00:00:00:00:00:01 to 00:00:00:00:00:02 is --1-1-2----1-4-4
----1-5-3----1-6-2----1-8-3----2-14-1----2-15-3----2-12-4----2-16-1----2-13-1---
-4-10-1----2-3-1----3-2-4--
Have calculated the longest path from 00:00:00:00:00:01 to 00:00:00:00:00:02
Now configuring switches of interest
Configure done
```

得到最长路



new_photo.png 更新



Photos 文件夹中存入新图片

5. pingall 检查所有主机间的连通情况

```
mininet> pingall
*** Ping: testing ping reachability
h4 -> h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h1 -> h4 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h17 -> h4 h1 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h19 -> h4 h1 h17 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h12 -> h4 h1 h17 h19 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h14 -> h4 h1 h17 h19 h12 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h5 -> h4 h1 h17 h19 h12 h14 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h20 -> h4 h1 h17 h19 h12 h14 h5 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h6 -> h4 h1 h17 h19 h12 h14 h5 h20 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h8 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
h15 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h3 h9 h10 h16 h13 h7 h11 h2 h18
h3 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h9 h10 h16 h13 h7 h11 h2 h18
h9 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h10 h16 h13 h7 h11 h2 h18
h10 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h16 h13 h7 h11 h2 h18
h16 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h13 h7 h11 h2 h18
h13 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h7 h11 h2 h18
h7 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h11 h2 h18
h11 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h2 h18
h2 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h18
h18 -> h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2
*** Results: 0% dropped (380/380 received)
mininet>
```