网络算法基础项目二

一、基本原理

- **Flow-Mod 信息**

Flow-Mod 消息是 OpenFlow 控制器对 OpenFlow 交换机设置流表项的消息。可对流表项进行添加、删除、变更设置等操作。整个消息可以分为三部分：openflow 主体部分、match 部分、instruction 部分。match 部分是匹配条件，instruction 部分是指令，当一个数据包满足匹配条件就会执行 instruction 中的指令。

- **ARP**

ARP 协议，即地址解析协议，可以通过解析 IP 地址得到 MAC 地址。主要通过报文工作，ARP 报文分为 ARP 请求和 ARP 应答报文两种。

ARP 请求报文：

当一个主机想要找出另一个主机的 MAC 地址时，首先会查看自己的 ARP 缓存表，若在 ARP 缓存表中找不到对应的 MAC 地址，则将缓存该数据报文,然后以广播方式发送一个 ARP 请求报文。ARP 请求报文中的发送端 IP 地址和发送端 MAC 地址为 h1 的 IP 地址和 MAC 地址，目标 IP 地址和目标 MAC 地址为 h2 的 IP 地址和全 0 的 MAC 地址。

ARP 应答报文：

受到请求报文的主机比较自己的IP地址和ARP请求报文中的目标IP地址，当两者相同时将ARP请求报文中的发送端的IP地址和MAC地址存入自己的ARP表中。之后以单播方式发送ARP应答报文给发送端，其中包含了自己的MAC地址（只有验证成功的主机才会发送ARP应答报文。

- **h1 ping h2 的过程**

1.h1 查看自己的 ARP 缓存表，若其中有 h2 对应的表项，将直接利用 ARP 表中的 MAC 地址，对 IP 数据包帧封装，并将数据包发送给 h2；

2.若 h1 的 ARP 缓存表中没有 h2 对应的表项，将缓存该数据报文，然后以广播方式发送一个 ARP 请求报文；

3.h2 比较自己的 IP 地址和 ARP 请求报文中的目标 IP 地址，当两者相同时将 ARP 请求报文中的发送端（即 h1）的 IP 地址和 MAC 地址存入自己的 ARP 表中。之后将 ARP 应答报文单独发送给 h1；

4. h1 收到 ARP 应答报文后，将 h2 的 MAC 地址加入到自己的 ARP 缓存表中，同时将 IP 数据包封装并发送出去。

二、Kruscal 算法思路

首先创建一个用于储存最小生成树的列表 mst_edge，判断图是否全连通（若图中节点数目小于 0 或边的数量小于节点数目-1，则说明该图不是全连通，将返回一个空列表，即没有最小生成树），若图全连通，则将图中所有边以[节点 a,节点 b,权重]的形式存储在列表 linked_edge 中，根据边的权重，可使用 sort 方法将其按从小到大的顺序排序，之后遍历 linked_edge 列表中的元素，判断将其加入后是否符合成环的条件，若不符合，则将其添加到 mst_edge 列表中，否则不加，将 linked_edge 列表中的元素全部遍历一遍后，输出 mst_edge 列表。

三、代码详情

```python
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
from ryu.lib.packet import arp
from ryu.lib.packet import ipv6
from ryu.lib import mac



import networkx as nx



import matplotlib.pyplot as plt



import copy
import random
import numpy as np
```

```python
def draw(Matrix, linkList1):
    """画图,生成图片"""
    G = nx.Graph()
    weight = {}
    graph = []
    for i in range(len(Matrix)):
        for j in range(i):
            if Matrix[i][j] > 0 and Matrix[i][j] < 1000000:
                graph.append((i + 1, j + 1))
                graph.append((j + 1, i + 1))
                weight[(i + 1, j + 1)] = int(Matrix[i][j])
                weight[(j + 1, i + 1)] = int(Matrix[i][j])



    path = []
    for link1 in linkList1:
        path.append((link1[0] + 1, link1[1] + 1))
        path.append((link1[1] + 1, link1[0] + 1))



    for gra in graph:
        G.add_edge(gra[0], gra[1], color='black')



    for pat in path:
        G.add_edge(pat[0], pat[1], color='red')
    pos = nx.spring_layout(G)
    edges = G.edges()
    colors = [G[u][v]['color'] for u, v in edges]
    nx.draw_networkx_nodes(G, pos, node_size=300)
    nx.draw_networkx_edges(G, pos, width=2, edge_color=colors,
node_shape='p')
    nx.draw_networkx_labels(G, pos, font_size=13)
    nx.draw_networkx_edge_labels(G, pos, weight, font_size=10)

    plt.savefig('now_photo.png')



class Topo(object):
    def __init__(self, logger):
        self.switches = None
        self.host_mac_to = {}
        self.logger = logger
        self.edges = {}
        self.weights = 0;
        self.edge_weight = None;
        self.flag = 0


    def edges_weight(self):
        """给各边赋权重"""
        edge_weight = np.array(np.ones((20, 20)) * 1000000)


        for (s, t) in self.edges.keys():
            edge_weight[s - 1][t - 1] = random.randint(1, 10)
            # 保证两个节点到彼此的权重相同
            edge_weight[t - 1][s - 1] = edge_weight[s - 1][t - 1]

        return edge_weight

    def kruskal(self, edge_weight):
        """实现 Kruscal 算法，返回最小生成树各边"""
        """思路:先将所有连通的边加入 linked_edge 列表中，再使用 Kruscal 算法找到最小生成树，并存储各边至 mst_edge 中"""
        node_num = len(edge_weight)
        edge_num = 0

        # 得到边的个数
        for i in range(node_num):
            for j in range(i):
                if edge_weight[i][j] > 0 and edge_weight[i][j] < 10000000:
                    edge_num += 1

        mst_edge = []

        # 如果边的数量小于点的数量-1，即不是全连通，直接返回
        if edge_num < node_num - 1:
            return mst_edge
        linked_edge = []
        # 将连通的边加入 linked_edge
        for i in range(node_num):
            # 从 i 开始，遍历剩下的点
            for j in range(i + 1, node_num):
                # 如果两个节点之间存在边
                if edge_weight[i][j] < 10000000:
                    # 将该边加入集合，形式为[节点 i,节点 j,权重]
                    linked_edge.append([i, j, edge_weight[i][j]])
                    # i, j 均从 0 开始，为 0--12；所给图连通的边均加入
linked_edge


        # 将边按第二个元素即权重排序，边权重从小到大
        linked_edge.sort(key=lambda x: x[2])


        # 创建节点列表
        forest = [[i] for i in range(node_num)]
        # 每次取权重最小的边


        for edge in linked_edge:
            for i in range(len(forest)):
                if edge[0] in forest[i]:  # 边的左结点在该树内
                    m = i
                if edge[1] in forest[i]:  # 边的右结点在该树内
                    n = i


            # m==n 时，即两结点均在该树内
            # m!=n 时，合并树
            if m != n:
                mst_edge.append(edge)
                forest[m] = forest[m] + forest[n]
                forest[n] = []



        return mst_edge  # kruskal 算法计算出的最小生成树所含边



    def find_neighbors(self, src, list):
        """找到各结点的邻接结点，存储在二维列表 neighbors"""
        neighbors = [[] for i in range(len(list) + 1)]  # 最小生成树边为 n-1 条，要加 1
        for i in range(len(list) + 1):
            for edge in list:
                if i == edge[0]:
                    neighbors[i].append(edge[1])
                elif i == edge[1]:
                    neighbors[i].append(edge[0])
        return neighbors
        # 某个结点的邻接结点 e.g.neighbors[0]=[1,2,3]表示结点 0 邻接结点 1，2，3



    # src 当前操作的结点;pre_src 上一个结点
    def find_links(self, src, pre_src, links):
        result = []
        if len(links[src]) < 1:
            return result
        for node in links[src]:
            if node != pre_src:
                result.append((pre_src, src, node))
                newresult = self.find_links(node, src, links)
                result.extend(newresult)


        return result


    def cal_flowTables(self, src_dw, first_port):
        """收到包之后调用该函数，计算流表的转发，流表匹配源 ip，向生成树上其他端口转发"""
        if self.weights == 0:
            self.edge_weight = self.edges_weight()
            self.weights = 1
        edgeList = self.kruskal(self.edge_weight)
        nodes_neighbor = self.find_neighbors(src_dw - 1, edgeList)  # 每个结点邻接的结点列表
        links = self.find_links(src_dw - 1, None, nodes_neighbor)  # （前个结点，当前结点，后个结点）
        print('起始结点为：', src_dw)  # 打印起始的结点
        if self.flag == 0:
            draw(self.edge_weight, edgeList)
            self.flag = 1
        edgeList1 = edgeList.copy()
        for i in range(len(edgeList1)):
            edgeList1[i][0] += 1
            edgeList1[i][1] += 1
        print('最小生成树各边包括：', edgeList1)  # 打印最小生成树各边


        temp1 = {}  # key 为两个邻接的结点，value 为两个邻接结点中后一个结点邻接
```

```
的结点列表
        for link in links:
            if (link[0], link[1]) not in temp1.keys():
                temp1[(link[0], link[1])] = [link[2]]
            else:
                temp1[(link[0], link[1])].append(link[2])



        temp2 = []
        index = [key[1] for key in temp1.keys()]
        # temp1中每个key中的第二个结点组成的列表，即为邻接结点大于等于2个的结
点
        for i in range(20):
            if i not in index:
                for key in temp1.keys():
                    if i in temp1[key]:
                        temp2.append((key[1], i, None))
                        # 中间结点无后继
            else:
                for key in temp1.keys():
                    if i == key[1]:
                        temp2.append((key[0], key[1], temp1[key]))
                        # 中间结点有后继，有/无前继



        ryu_FlowTables = []
        # 根据Ryu的格式配置流表路径
        for item in temp2:
            if item[0] is not None:
                if item[2] is None:
                    inport = self.edges[(item[1] + 1, item[0] + 1)]
                    outportList = [1]
                    ryu_FlowTables.append((item[1] + 1, inport,
outportList))
```

```
                else:
                    inport = self.edges[(item[1] + 1, item[0] + 1)]
                    outportList = [1]
                    for node in item[2]:
                        op = self.edges[(item[1] + 1, node + 1)]
                        outportList.append(op)
                    ryu_FlowTables.append((item[1] + 1, inport,
outportList))
            else:
                inport = first_port
                outportList = [1]
                for node in item[2]:
                    op = self.edges[(item[1] + 1, node + 1)]
                    outportList.append(op)
                ryu_FlowTables.append((item[1] + 1, inport,
outportList))


        return ryu_FlowTables, nodes_neighbor




# Ryu控制器
class KruscalController(app_manager.RyuApp):
    """控制器类"""
    # 指明OpenFlow版本为1.3
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(KruscalController, self).__init__(*args, **kwargs)



        self.mac_to_port = {}  # 全局的mac
表,{{datapath:mac->port},...,{datapath:mac->port}}
        self.datapaths = []
        self.flood_history = {}  # 泛洪历史表
        self.arp_history = {}  # arp 历史表
        self.flag = False
        self.mac_list = {1: '00:00:00:00:00:01',
                      2: '00:00:00:00:00:02',
                      3: '00:00:00:00:00:03',
                      4: '00:00:00:00:00:04',
                      5: '00:00:00:00:00:05',
                      6: '00:00:00:00:00:06',
                      7: '00:00:00:00:00:07',
                      8: '00:00:00:00:00:08',
                      9: '00:00:00:00:00:09',
                      10: '00:00:00:00:00:10',
                      11: '00:00:00:00:00:11',
                      12: '00:00:00:00:00:12',
                      13: '00:00:00:00:00:13',
                      14: '00:00:00:00:00:14',
                      15: '00:00:00:00:00:15',
                      16: '00:00:00:00:00:16',
                      17: '00:00:00:00:00:17',
                      18: '00:00:00:00:00:18',
                      19: '00:00:00:00:00:19',
                      20: '00:00:00:00:00:ff'}



        self.arp_table = {'192.168.0.1': '00:00:00:00:00:01',
                       '192.168.0.2': '00:00:00:00:00:02',
                       '192.168.0.3': '00:00:00:00:00:03',
                       '192.168.0.4': '00:00:00:00:00:04',
                       '192.168.0.5': '00:00:00:00:00:05',
                       '192.168.0.6': '00:00:00:00:00:06',
                       '192.168.0.7': '00:00:00:00:00:07',
                       '192.168.0.8': '00:00:00:00:00:08',
                       '192.168.0.9': '00:00:00:00:00:09',
                       '192.168.0.10': '00:00:00:00:00:10',
                       '192.168.0.11': '00:00:00:00:00:11',
                       '192.168.0.12': '00:00:00:00:00:12',
                       '192.168.0.13': '00:00:00:00:00:13',
                       '192.168.0.14': '00:00:00:00:00:14',
                       '192.168.0.15': '00:00:00:00:00:15',
                       '192.168.0.16': '00:00:00:00:00:16',
                       '192.168.0.17': '00:00:00:00:00:17',
                       '192.168.0.18': '00:00:00:00:00:18',
                       '192.168.0.19': '00:00:00:00:00:19',
                       '192.168.0.255': '00:00:00:00:00:ff'
                       }

        self.topo = Topo(self.logger)

    # 由dpid找到相应的datapath
    def find_dp(self, dpid):
        for dp in self.datapaths:
            if dp.id == dpid:
                return dp
        return None

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        """向控制器传输交换机特征"""
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser



        match = parser.OFPMatch()
        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):
        """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, path, event, src_mac, dst_mac):
        """配置路径"""
        msg = event.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        for switch, inport, outportList in path:
            match = parser.OFPMatch(in_port=inport, eth_src=src_mac,
eth_dst=dst_mac)
            actions = []

            for outport in outportList:
                actions.append(parser.OFPActionOutput(outport))

            # 由 dpid 找到对应的 datapath
            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
            )
            # 下发流表
            datapath.send_msg(mod)



    # 监听 Packet_in 事件
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, event):
        msg = event.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser



        in_port = msg.match['in_port']



        # 获取数据
        pkt = packet.Packet(msg.data)



        # 假设为以太帧，获取帧头
        eth = pkt.get_protocols(ethernet.ethernet)[0]
```

```
        # 直接下发生成树流表
        if self.flag == False:
            for i in range(20):
                i = i + 1
                path1, nodes_neighbor = self.topo.cal_flowTables(
                    i,
                    1)
                print('switches_path', path1)
                for nodes_n in range(20):
                    nodes_n += 1
                    # 给路径中交换机下发流表
                    self.configure_path(path1, event, self.mac_list[i],
self.mac_list[nodes_n])
                    path2, nodes_neighbor = self.topo.cal_flowTables(
                        nodes_n,
                        1)
                    # 下发回路
                    self.configure_path(path2, event,
self.mac_list[nodes_n], self.mac_list[i])
                    #
[(nodes_n,1,[self.topo.edges[(nodes_n,i)]]),(i,self.topo.edges[(i,nod
es_n)],[1])]
            print('----------------done----------------')



            self.flag = True



        # 丢弃 LLDP 帧
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return
```

```
dst_mac = eth.dst
src_mac = eth.src



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, [])



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]
    mst_path, _ = self.topo.cal_flowTables(
            1,
            1)
        assert len(mst_path) > 0


        self.configure_path(mst_path, event, src_mac, dst_mac)
        self.logger.info("Configure done")


        out_port = []
        for s, _, op in mst_path:
            if s == dpid:
                out_port = op


    else:
        if self.arp_handler(msg):
            return


        out_port = []
        out_port.append(ofproto.OFPP_FLOOD)


# 交换机进入与离开时分别触发相应的函数
@set_ev_cls(event.EventSwitchEnter)
def switch_enter_handler(self, event):
    self.logger.info("一个交换机进入，重新发现拓扑")
    self.switch_status_handler(event)
    self.logger.info('拓扑发现完毕')
    @set_ev_cls(event.EventSwitchLeave)
    def switch_leave_handler(self, event):
        self.logger.info("一个交换机退出，重新发现拓扑")
        self.switch_status_handler(event)
        self.logger.info('拓扑发现完毕')


    def switch_status_handler(self, event):
        """配置交换机状态并打印出连通信息"""
        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.edges[(s1, s2)] = p1
            self.topo.edges[(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)


    def arp_handler(self, msg):
        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



        if arp_pkt:
            hwtype = arp_pkt.hwtype
            proto = arp_pkt.proto
            hlen = arp_pkt.hlen
            plen = arp_pkt.plen



            opcode = arp_pkt.opcode
```

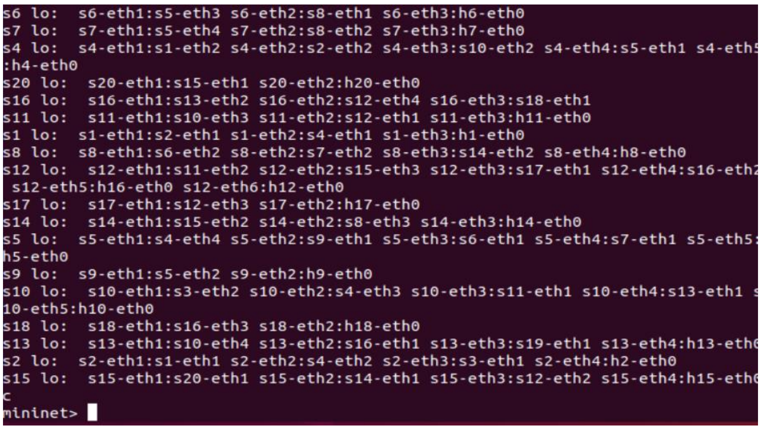
```
arp_src_ip = arp_pkt.src_ip
arp_dst_ip = arp_pkt.dst_ip



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
```

四、结果展示

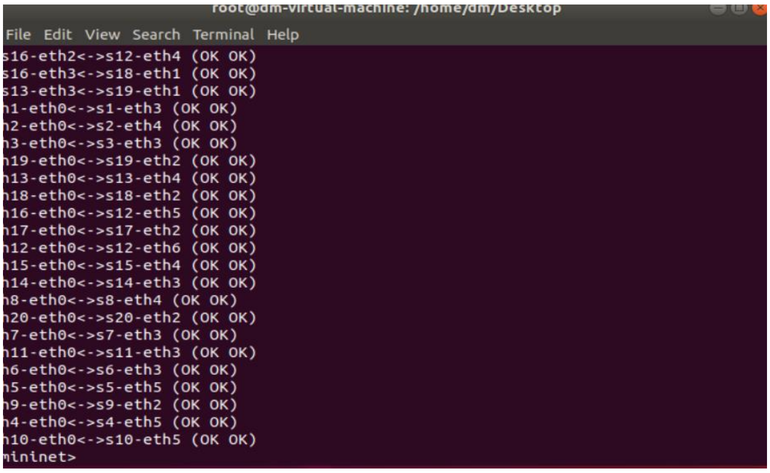
1. 测试拓扑信息

a. 查看链路信息



```
s6 lo:  s6-eth1:s5-eth3 s6-eth2:s8-eth1 s6-eth3:h6-eth0
s7 lo:  s7-eth1:s5-eth4 s7-eth2:s8-eth2 s7-eth3:h7-eth0
s4 lo:  s4-eth1:s1-eth2 s4-eth2:s2-eth2 s4-eth3:s10-eth2 s4-eth4:s5-eth1 s4-eth5
:h4-eth0
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
c
mininet> 
```

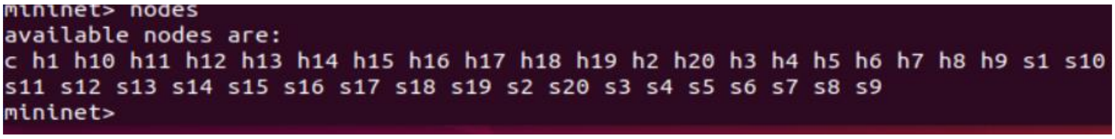
b. 查看链路是否可用



root@dm-virtual-machine: /home/dm/Desktop
File Edit View Search Terminal Help

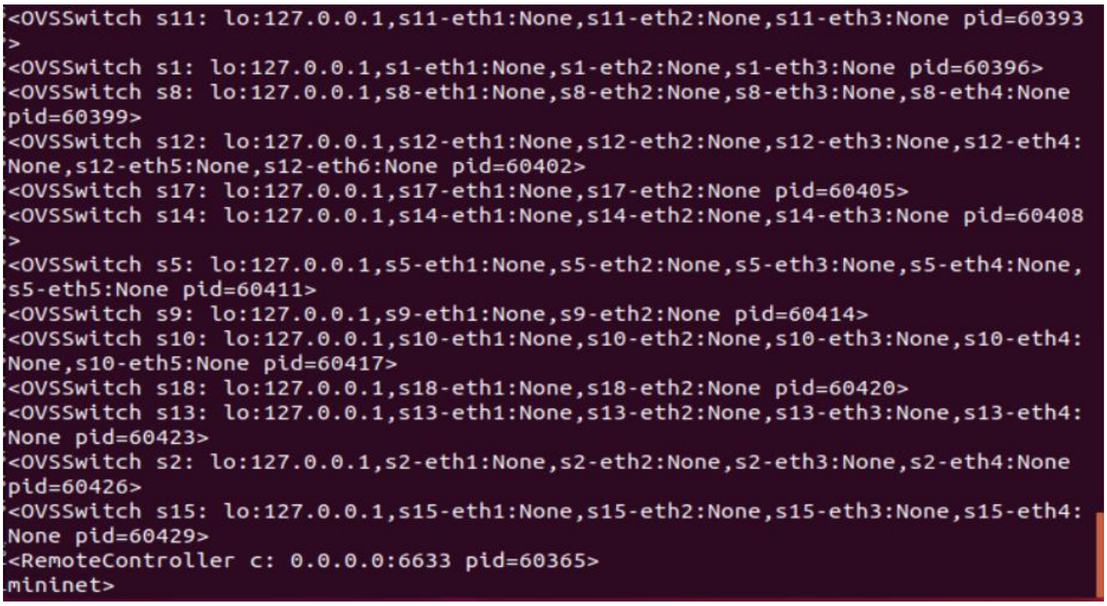
```
s16-eth2<->s12-eth4 (OK OK)
s16-eth3<->s18-eth1 (OK OK)
s13-eth3<->s19-eth1 (OK OK)
h1-eth0<->s1-eth3 (OK OK)
h2-eth0<->s2-eth4 (OK OK)
h3-eth0<->s3-eth3 (OK OK)
h19-eth0<->s19-eth2 (OK OK)
h13-eth0<->s13-eth4 (OK OK)
h18-eth0<->s18-eth2 (OK OK)
h16-eth0<->s12-eth5 (OK OK)
h17-eth0<->s17-eth2 (OK 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>
```

c. 查看可用节点



```
mininet> nodes
available nodes are:
c 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>
```

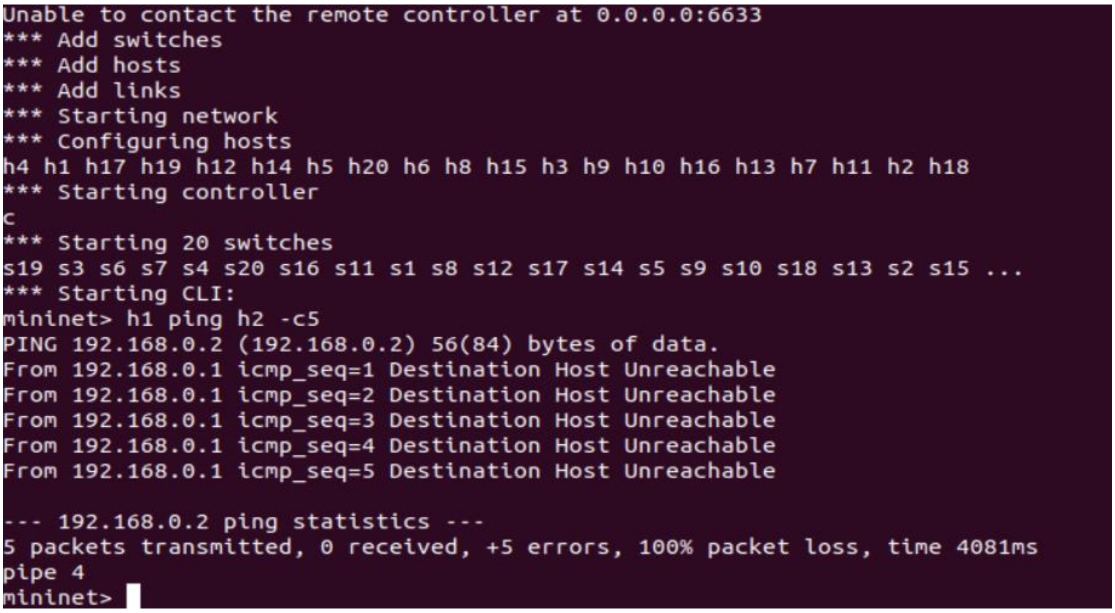
d. 查看节点信息



```
<OVSSwitch s11: lo:127.0.0.1,s11-eth1:None,s11-eth2:None,s11-eth3:None pid=60393
>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=60396>
<OVSSwitch s8: lo:127.0.0.1,s8-eth1:None,s8-eth2:None,s8-eth3:None,s8-eth4:None
pid=60399>
<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=60402>
<OVSSwitch s17: lo:127.0.0.1,s17-eth1:None,s17-eth2:None pid=60405>
<OVSSwitch s14: lo:127.0.0.1,s14-eth1:None,s14-eth2:None,s14-eth3:None pid=60408
>
<OVSSwitch s5: lo:127.0.0.1,s5-eth1:None,s5-eth2:None,s5-eth3:None,s5-eth4:None,
s5-eth5:None pid=60411>
<OVSSwitch s9: lo:127.0.0.1,s9-eth1:None,s9-eth2:None pid=60414>
<OVSSwitch s10: lo:127.0.0.1,s10-eth1:None,s10-eth2:None,s10-eth3:None,s10-eth4:
None,s10-eth5:None pid=60417>
<OVSSwitch s18: lo:127.0.0.1,s18-eth1:None,s18-eth2:None pid=60420>
<OVSSwitch s13: lo:127.0.0.1,s13-eth1:None,s13-eth2:None,s13-eth3:None,s13-eth4:
None pid=60423>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None,s2-eth4:None
pid=60426>
<OVSSwitch s15: lo:127.0.0.1,s15-eth1:None,s15-eth2:None,s15-eth3:None,s15-eth4:
None pid=60429>
<RemoteController c: 0.0.0.0:6633 pid=60365>
mininet>
```

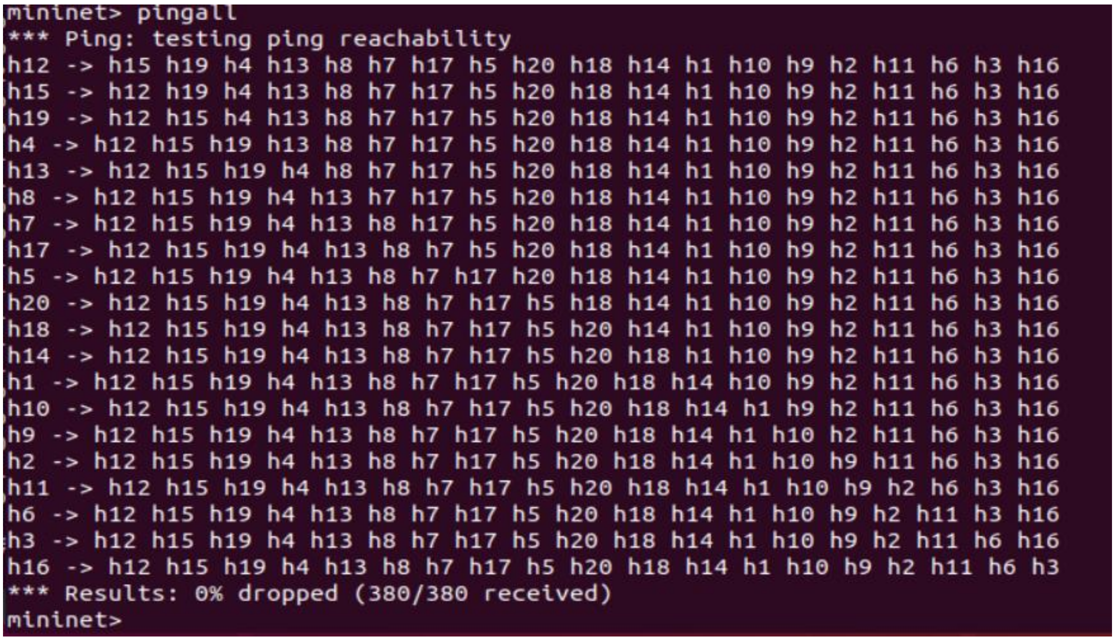
e. 查看连通性

连接 ryu 前



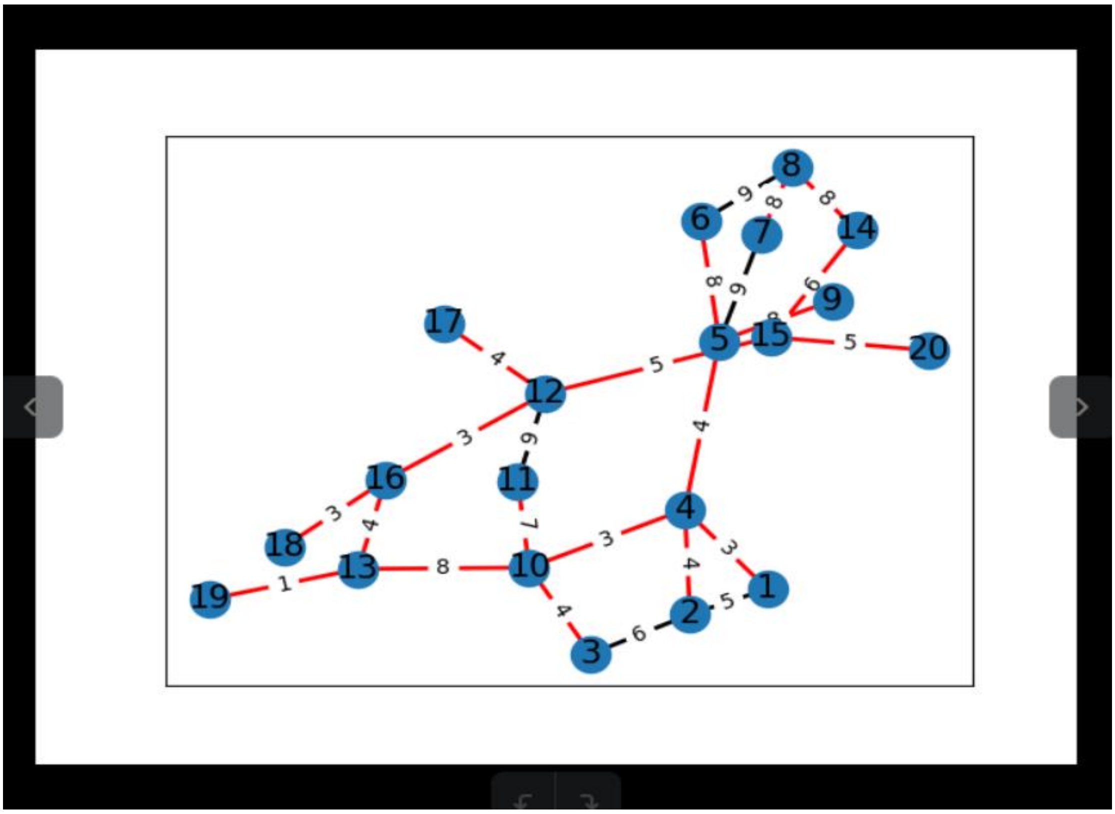
```
Unable to contact the remote controller at 0.0.0.0:6633
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h4 h1 h17 h19 h12 h14 h5 h20 h6 h8 h15 h3 h9 h10 h16 h13 h7 h11 h2 h18
*** Starting controller
c
*** Starting 20 switches
s19 s3 s6 s7 s4 s20 s16 s11 s1 s8 s12 s17 s14 s5 s9 s10 s18 s13 s2 s15 ...
*** Starting CLI:
mininet> h1 ping h2 -c5
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
From 192.168.0.1 icmp_seq=1 Destination Host Unreachable
From 192.168.0.1 icmp_seq=2 Destination Host Unreachable
From 192.168.0.1 icmp_seq=3 Destination Host Unreachable
From 192.168.0.1 icmp_seq=4 Destination Host Unreachable
From 192.168.0.1 icmp_seq=5 Destination Host Unreachable

--- 192.168.0.2 ping statistics ---
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 4081ms
pipe 4
mininet> 
```

连接 ryu 后



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

2. 查看最小生成树结果


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