计算机网络实验

实验四：路由算法实验

学院：信息学院

姓名：黄泸明

学号：22920172204130

专业：网络空间安全

一、实验目的

目的：学习和掌握距离向量算法

内容：编程实现并分析以下过程

- 模拟路由收敛
- 模拟拓扑变化
- 制造路由回路
- 抑制路由回路

二、实验内容与分析

实验环境

CentOS 7.7 + GCC 4.8.5
Win10+Python3.6

实验环境配置

本次实验使用五台虚拟机模拟五个路由器，网络连接均为NAT模式，端口统一用20000，五个路由器的名称及对应IP如下：

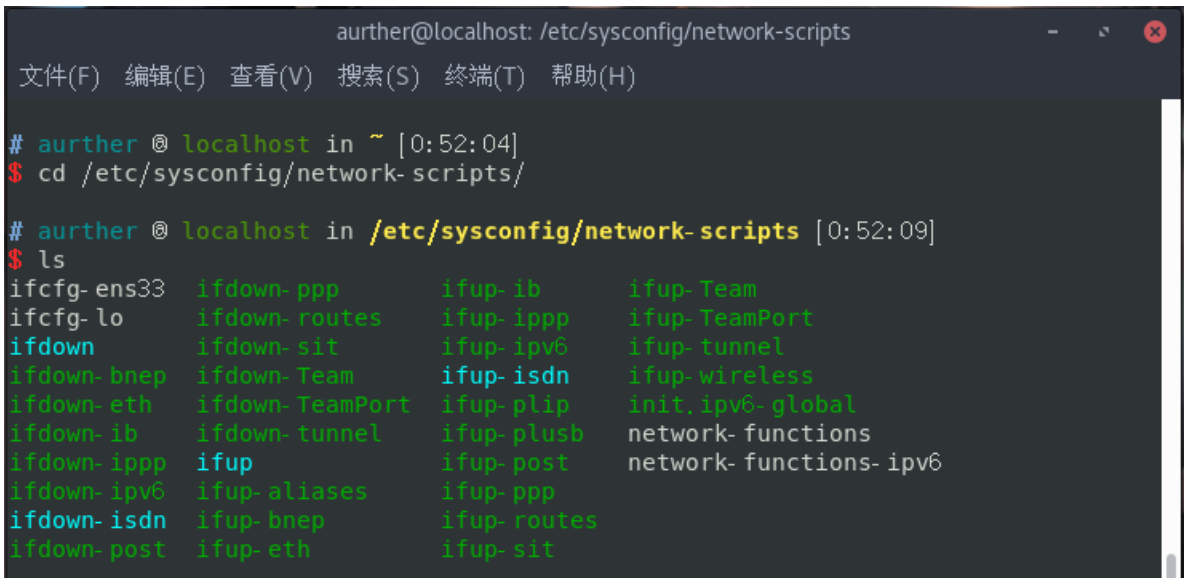
A ： 192.168.126.65
B ： 192.168.126.66
C ： 192.168.126.67
D ： 192.168.126.68
E ： 192.168.126.69

网络配置方法如下：
以路由器A为例

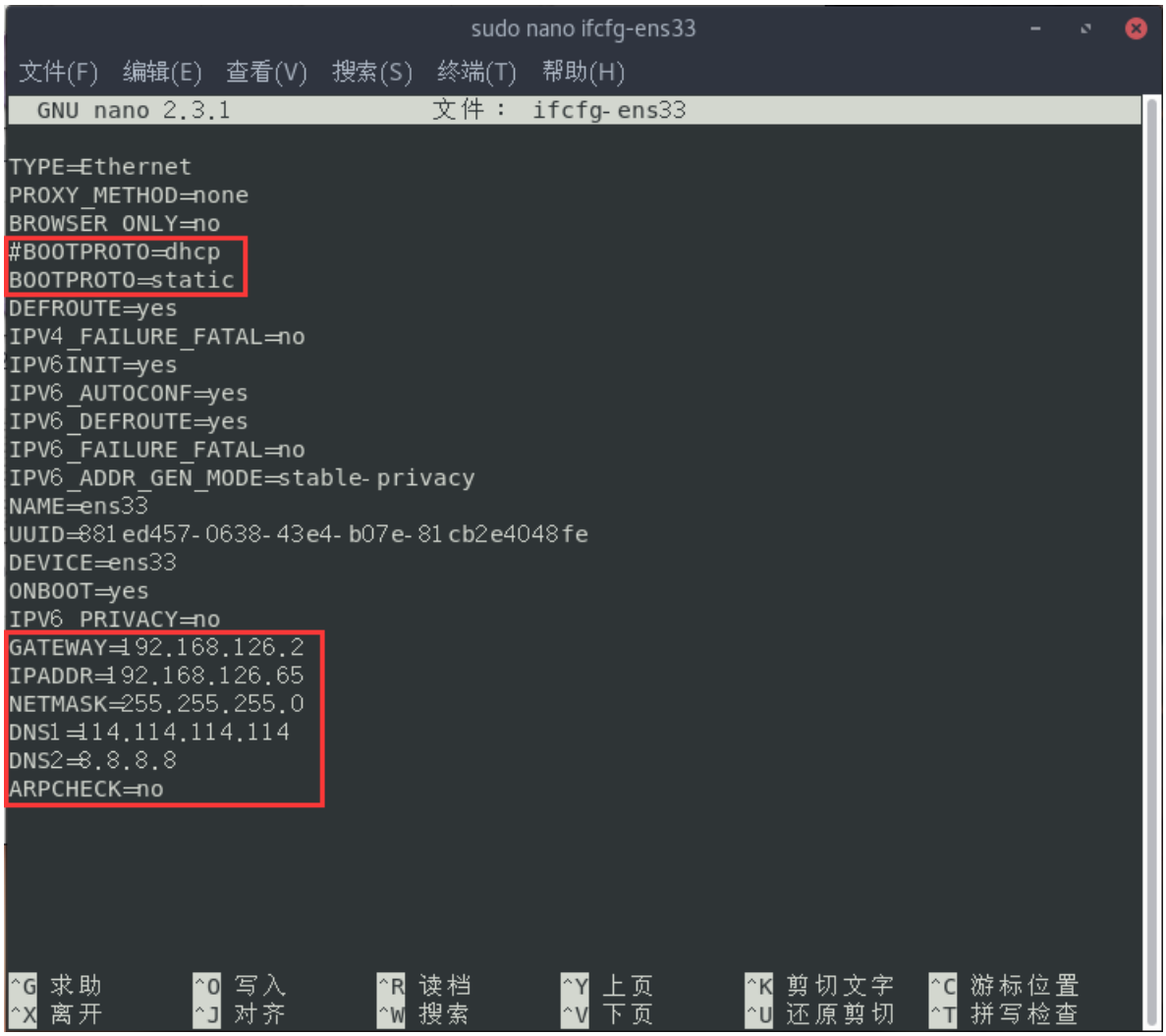
在终端输入如下命令

```
cd /etc/sysconfig/network-scripts/
ls
```

可以看到第一个网卡ifcfg-ens33



用nano或vim对其进行编辑



注释BOOTPROTO=dhcp，将其设置为静态IP

```
#BOOTPROTO=dhcp
BOOTPROTO=static
```

并在后面添加IP配置信息

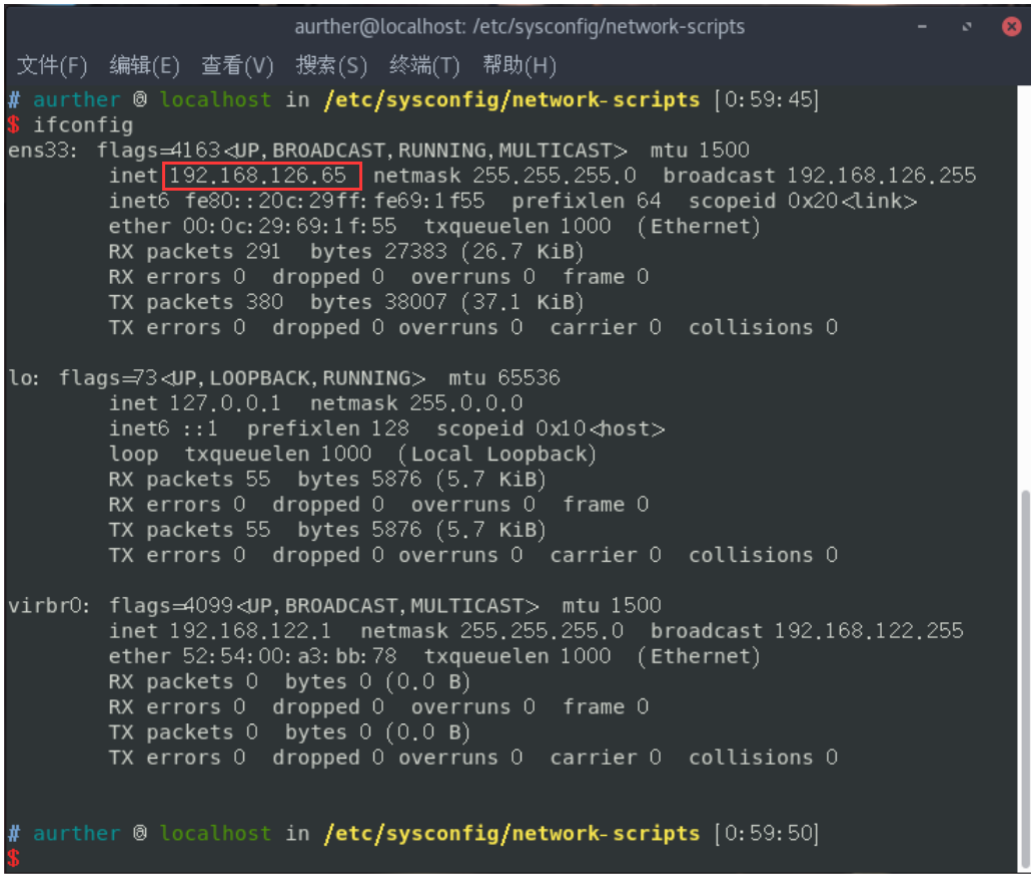
```
GATEWAY=192.168.126.2
IPADDR=192.168.126.65
NETMASK=255.255.255.0
DNS1=114.114.114.114
DNS2=8.8.8.8
ARPCHECK=no
```

GATEWAY和NETMASK可以在VMWare的虚拟网络编辑器查看

设置完成后重启网络服务

```
service network restart
```

利用ifconfig可以看到网络地址设置成功



同理依次设置其他4个虚拟机IP即可

实验原理

路由器的功能

- 路由选择（Routing）
  - 选择一条正确的路径——寻找下一跳（next hop）
    - 目的可达
    - 路径最优（距离最短、延迟最小、费用最低……）
  - 建立路由表
    - 静态
    - 动态（距离向量、链路状态、路径向量……）
- 转发（Forwarding）
  - 根据路由选择的结果，将数据包从输入接口转发至输出接口

距离向量路由算法

- 属于内部网关协议（IGP）
- 使用距离向量（Distance-Vector）算法建立路由表
- 使用定时更新维护路由表
- 常见协议：RIPv1、RIPv2……

DV算法基本思想

- 使用“距离”度量路由
  - 路由表保存到达各目标的最短距离及下一跳

目标	下一跳	距离
A	E	2
B	D	6
…	…	…

- 使用“距离向量”交换路由信息
  - 相邻路由器之间交换路由表，各自计算最佳路由——到达目标的最短距离及下一跳
- 所有路由器两两定时交换，将路由信息扩散至全网，最后达到收敛状态

DV算法原理

- 定义：
  - adj(i)为节点i的所有相邻节点的集合
  - c(i, n)为一对相邻节点i和n之间的距离
  - d(i, j)为从节点i到节点j之间的最短距离
- 公式：
  - d(i, i) = 0
  - d(i, j) = min[c(i, n) + d(n, j)]。其中n∈adj(i)
    - c(i, n)为初始条件，已知
    - d(n, j)为节点i从邻居节点n获知的路由信息，已知

实验代码

DVroute.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import sys, time, socket, copy, json
import prettytable as pt
import threading
from threading import Thread

__author__ = '黄泸明'
'''
addr2rName字典为建立地址和路由器名的一一对应
'''
addr2rName = {}
addr2rName[("192.168.126.65", 20000)] = 'A'
addr2rName[("192.168.126.66", 20000)] = 'B'
addr2rName[("192.168.126.67", 20000)] = 'C'
addr2rName[("192.168.126.68", 20000)] = 'D'
addr2rName[("192.168.126.69", 20000)] = 'E'


class Router(socket.socket):
    '''
    这是一个继承socket.socket的类，用于实现路由器功能：
    该路由器可用于有更新定时器（update）下DV路由算法，模拟路由表收敛的过程，
    对于无穷计数和路由回路，采用逆向毒化（poison reverse）加以解决。
    对于链路变化过程，可模拟linkChange（邻居链路建立和距离改变）
    和linkDown（邻居链路断开）功能

    '''
    def __init__(self, router_address, neighbor, addr2rName, MaxHop=15):
        #用父类socket.socket的初始化方法来初始化继承的属性
        #初始化包含五个参数：
        #router_address：路由器地址，形式为（ip,port）
        #neighbor：邻居路由器，类型为字典，(key,value) = (rName, {addr, cost})
        #addr2rName:字典为建立地址和路由器名的一一对应
        #MaxHop:最大跳数，缺省值为15，MaxHop+1(16)表示不可达
        super(Router, self).__init__(
            socket.AF_INET,
            socket.SOCK_DGRAM)  #该路由器采用UDP传输，socket.SOCK_DGRAM用于UDP协议
        self.__addr = router_address
        self.__neighbor = neighbor
        self.__addr2rName = addr2rName
        self.__MaxHop = MaxHop

        self.__name = self.__addr2rName[self.__addr]  #所创建的路由器名
        self.__rName2addr = {}  #字典建立addr2rName的反向查找
        for addr in self.__addr2rName:
            self.__rName2addr[self.__addr2rName[addr]] = addr
        #路由表字典，(key,value)=(dest,{nextHop,cost})，初始时，路由表仅有邻居节点
        self.__rtrTable = {}
        for dest in self.__neighbor:
            self.__rtrTable[dest] = {}
            self.__rtrTable[dest]['nextHop'] = dest
            self.__rtrTable[dest]['cost'] = self.__neighbor[dest]['cost']
        self.__neighCost = {}  #邻居链路的开销，(key, value) = (nextHop, cost)
        for nextHop in self.__neighbor:
            self.__neighCost[nextHop] = self.__neighbor[nextHop]['cost']

        #改变链路距离的一方发送距离改变信息在头部加上的标记
```

```
        self.__linkChangeFlag = '*'
        #链路断开的一方发送连接断开信息在头部加上的标记
        self.__linkDownFlag = '#'

        self.__rtrTable_history = None  #上次更新的路由表
        self.__convergedPrintTimes = 0  #路由表收敛后控制其只输出一次

        #逆向毒化(poison reverse)算法的开启标志，默认为开启状态
        self.__PoisonReverse = True

    def __updateTimer(self):
        '''为了方便观察,此处更新定时器的目标函数将打印路由表，
        向邻居发送路由表结合在一起。'''
        self.__showrt()
        self.__sendRtrTable()

    def __showrt(self):
        '''此处当相邻两次
        的路由表相同，则认为路由收敛（实际可能未收敛）'''
        '''打印样例
        Distance vector list is:
        +-------------+---------+------+
        | destination | nexthop | cost |
        +-------------+---------+------+
        |      B      |    B    |  2   |
        |      E      |    E    |  2   |
        |      C      |    B    |  10  |
        |      D      |    E    |  8   |
        +-------------+---------+------+
        '''

        if str(self.__rtrTable) != str(self.__rtrTable_history):
            #路由表如果有更新就输出新路由表信息
            print('Distance vector list is:')
            tb = pt.PrettyTable()
            tb.field_names = ['destination', 'nexthop', 'cost']
            for dest in self.__rtrTable:
                if self.__rtrTable[dest]['cost'] > self.__MaxHop:
                    self.__rtrTable[dest]['cost'] = self.__MaxHop + 1
                tb.add_row([
                    dest, self.__rtrTable[dest]['nextHop']
                    if self.__rtrTable[dest]['cost'] <= self.__MaxHop else ' ',
                    self.__rtrTable[dest]['cost'] if
                    self.__rtrTable[dest]['cost'] <= self.__MaxHop else 'inf'
                ])
            print(tb)
            #更新历史路由表，注意此处必须用深拷贝，否则会出错
            self.__rtrTable_history = copy.deepcopy(self.__rtrTable)
            self.__convergedPrintTimes = 0
        else:
            if self.__convergedPrintTimes == 0:
                #如果是第一次打印就输出路由表收敛信息，否则不打印路由表
                print('The network has converged:')
                tb = pt.PrettyTable()
                tb.field_names = ['destination', 'nexthop', 'cost']
                for dest in self.__rtrTable:
                    if self.__rtrTable[dest]['cost'] > self.__MaxHop:
                        self.__rtrTable[dest]['cost'] = self.__MaxHop + 1
                    tb.add_row([
                        dest, self.__rtrTable[dest]['nextHop']
                        if self.__rtrTable[dest]['cost'] <= self.__MaxHop else
                        ' ', self.__rtrTable[dest]['cost']
                        if self.__rtrTable[dest]['cost'] <= self.__MaxHop else
                        'inf'
                    ])
```

```
                print(tb)
                self.__convergedPrintTimes = 1  #控制其只打印一次

    def __recvRtrTable(self):
        '''用于接受邻居发来的距离向量，并更新距离向量表'''
        while True:
            try:
                data, addr = self.recvfrom(1024)  #接收的最大数据量bufsize = 1024
                data = data.decode(encoding='UTF-8', errors='ignore')
                '''首字节判断是否为linkChange和linkDown信息'''
                if data[0] == self.__linkChangeFlag:
                    self.__linkChange(addr, int(data[1:]), needSend=False)
                elif data[0] == self.__linkDownFlag:
                    self.__linkDown(addr, needSend=False)
                else:
                    self.__updatertrTable(addr, json.loads(data))
            except ConnectionError as e:
                print(e)
                pass

    def __sendRtrTable(self):
        '''向所有邻居发送距离向量信息'''
        for nextHop in self.__neighbor:
            rtrtable = copy.deepcopy(self.__rtrTable)
            if self.__PoisonReverse:  #使用逆向毒化算法
                ''' 若向目的邻居发送的距离向量中某个最佳路由下一跳为该邻居，则将跳数
                设置为最大跳数+1（不可达）'''
                for dest in self.__rtrTable:
                    if dest != nextHop and self.__rtrTable[dest][
                            'nextHop'] == nextHop:
                        rtrtable[dest]['cost'] = self.__MaxHop + 1
                    else:
                        pass
            else:  #不使用逆向毒化算法
                pass
            data = json.dumps(rtrtable)
            self.sendto(data.encode(encoding='UTF-8', errors='ignore'),
                        self.__rName2addr[nextHop])

    def __updatertrTable(self, addr, rtrtable):
        '''更新路由表，采用距离向量算法，对于相邻路由器X发来的路由表rtrtable，
        根据其的每一个项目（目的路由器为N）进行以下步骤：
        若 N是自己，则什么也不做，跳过
        否则 进行以下判断
            若 原来的路由表没有N，则将其添加到路由表中，距离为c[X]+rtrtable[N]
            否则 根据其自己的下一跳路由器做如下判断：
                若 N对于自己的下一跳是X,则用c[X]+rtrtable[N]替换路由表中项目(*)，
                否则 进行以下判断：
                    若 c[X]+rtrtable[N]<自己到N的距离，则更新路由器
                    否则 什么也不做
        (*)替换原因：这是最新的消息，以最新消息为准，无论替换后是变大还是变小
        '''
        From = self.__addr2rName[addr]
        for dest in rtrtable:
            if dest == self.__name:
                continue
            elif dest not in self.__rtrTable:
                self.__rtrTable[dest] = {}
                self.__rtrTable[dest]['nextHop'] = From
                self.__rtrTable[dest]['cost'] = min(
                    self.__neighCost[From] + rtrtable[dest]['cost'],
                    self.__MaxHop + 1)
            elif self.__rtrTable[dest]['nextHop'] == From:
                self.__rtrTable[dest]['cost'] = min(
                    self.__neighCost[From] + rtrtable[dest]['cost'],
```

```
                    self.__MaxHop + 1)
            elif self.__neighCost[From] + rtrtable[dest][
                    'cost'] < self.__rtrTable[dest]['cost']:
                self.__rtrTable[dest]['cost'] = min(
                    self.__neighCost[From] + rtrtable[dest]['cost'],
                    self.__MaxHop + 1)
                self.__rtrTable[dest]['nextHop'] = From
            else:
                pass

    def __parseUserInput(self):
        '''输入相应命令并选择相应功能'''
        while True:
            try:
                order = input().split()
                if order[0] == 'linkchange':
                    addr = (order[1], int(order[2]))
                    dist = int(order[3])
                    self.__linkChange(addr, dist, needSend=True)
                elif order[0] == 'linkdown':
                    addr = (order[1], int(order[2]))
                    self.__linkDown(addr, needSend=True)
                else:
                    print("InputError")
            except:
                print("InputError")

    def __linkChange(self, addr, dist, needSend):
        '''链路改变函数，输入要改变的目的邻居的addr以及改变后的跳数，其中布尔变量
        needSend表示是否向目的邻居发送改变信息，对于主动改变的一方，needSend=True，
        对于被动接受改变的一方，needSend=False。请注意，此函数也可以用于建立邻居关系。
        在距离改变后，立即重置self.__convergedPrintTimes和self.__rtrTable_history，
        使其在下个周期将更新后的路由表打印出来'''
        rName = self.__addr2rName[addr]
        '''如果目的addr不是其邻居，会将其加入本路由器的邻居中'''
        self.__neighbor[rName] = {}
        self.__neighbor[rName]['addr'] = addr
        self.__neighbor[rName]['cost'] = dist
        self.__neighCost[rName] = dist
        self.__rtrTable[rName] = {}
        self.__rtrTable[rName]['nextHop'] = rName
        self.__rtrTable[rName]['cost'] = dist
        self.__convergedPrintTimes = 0
        self.__rtrTable_history = None
        if needSend:
            data = self.__linkChangeFlag + str(dist)
            self.sendto(data.encode(encoding='UTF-8', errors='ignore'), addr)

    def __linkDown(self, addr, needSend):
        '''链路断开函数，输入要断开连接的目的邻居的addr，其中布尔变量needSend表示
        是否向目的邻居发送改变信息，对于主动改变的一方，needSend=True，对于被动接受
        改变的一方，needSend=False。在与邻居断开连接后，将链路距离设置为最大跳数+1
        （不可达），立即重置self.__convergedPrintTimes和self.__rtrTable_history，
        使其在下个周期更新后的路由表打印出来'''
        rName = self.__addr2rName[addr]
        self.__neighbor.pop(rName)
        self.__neighCost.pop(rName)
        self.__rtrTable[rName] = {}
        self.__rtrTable[rName]['nextHop'] = rName
        self.__rtrTable[rName]['cost'] = self.__MaxHop + 1
        self.__convergedPrintTimes = 0
        self.__rtrTable_history = None
        if needSend:
            data = self.__linkDownFlag
            self.sendto(data.encode(encoding='UTF-8', errors='ignore'), addr)
```

```
    def setPoisonReverse(self, openState):
        '''逆向毒化算法开启状态'''
        self.__PoisonReverse = openState

    def start(self):
        '''路由表开启，包含两个子线程，一个每隔时间T更新路由表，打印一次路由表，向邻居
        发送距离向量，此处为了方便观察，将其设置为10s，另一个接受用户的输入命令。主线
        程用于接收邻居发来的距离向量并对rtrTable做更新。'''
        self.bind(self.__addr)

        th1 = RepeatTimer(10, self.__updateTimer)
        th1.start()
        th2 = RepeatTimer(0, self.__parseUserInput)
        th2.start()

        self.__recvRtrTable()


class RepeatTimer(threading.Thread):
    '''定时器类，继承于threading.Thread类，interval为时间间隔'''
    def __init__(self, interval, target):
        Thread.__init__(self)
        self.interval = interval
        self.daemon = True
        self.stopped = False
        self.target = target

    def run(self):
        while not self.stopped:
            time.sleep(self.interval)
            self.target()


def parse_argv():
    '''解析运行时的参数（第一次运行时），其输入格式为
    "python3 DVroute.py listening_port ip1 port1 dist1 ip2 port2 dist2···",
    后面每个三元组代表每个邻居的距离信息'''
    s = sys.argv[1:]
    parsed = {}
    listening_port = s.pop(0)
    parsed['listening_port'] = int(listening_port)
    neighbor = {}
    for i in range(len(s) // 3):
        rName = addr2rName[(s[i * 3], int(s[i * 3 + 1]))]
        neighbor[rName] = {}
        neighbor[rName]['addr'] = (s[i * 3], int(s[i * 3 + 1]))
        neighbor[rName]['cost'] = int(s[i * 3 + 2])
    parsed['neighbor'] = neighbor
    return parsed


def get_host_ip():
    '''用于查询本机ip地址，返回值为ip'''
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(('8.8.8.8', 80))
        ip = s.getsockname()[0]
    finally:
        s.close()
    return ip


def main():
    '''主函数调用该路由器，生成一个最大跳数为15的路由器'''
```

```
    ip = get_host_ip()
    parsed = parse_argv()
    rt = Router(router_address=(ip, parsed['listening_port']),
                neighbor=parsed['neighbor'],
                addr2rName=addr2rName,
                MaxHop=15)
    #此处设置为逆向毒化算法为关闭状态，若要使用，将其注释即可
    rt.setPoisonReverse(openState=False)
    rt.start()


if __name__ == '__main__':
    main()
```

shell脚本文件

A1.sh

```
python3 DVroute.py 20000 192.168.126.66 20000 2 192.168.126.69 20000 2
```

B1.sh

```
python3 DVroute.py 20000 192.168.126.65 20000 2 192.168.126.67 20000 8 192.168.126.69
20000 6
```

C1.sh

```
python3 DVroute.py 20000 192.168.126.66 20000 8 192.168.126.68 20000 3
```

D1.sh

```
python3 DVroute.py 20000 192.168.126.67 20000 3 192.168.126.69 20000 6
```

E1.sh

```
python3 DVroute.py 20000 192.168.126.65 20000 2 192.168.126.66 20000 6 192.168.126.68
20000 6
```

A2. sh

```
python3 DVroute.py 20000 192.168.126.66 20000 2
```

B2. sh

```
python3 DVroute.py 20000 192.168.126.65 20000 2 192.168.126.67 20000 3
```

C2. sh

```
python3 DVroute.py 20000 192.168.126.66 20000 3
```

A3. sh

```
python3 DVroute.py 20000 192.168.126.66 20000 2
```

B3. sh

```
python3 DVroute.py 20000 192.168.126.65 20000 2 192.168.126.67 20000 3 192.168.126.68
20000 1
```

C3. sh

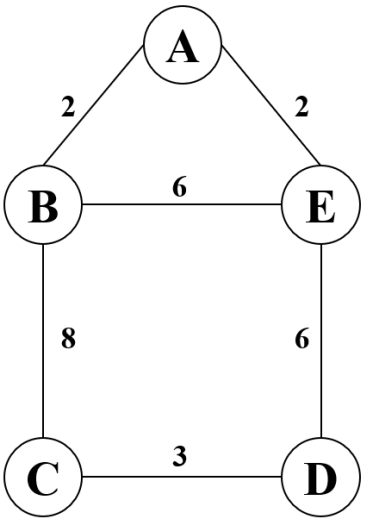
```
python3 DVroute.py 20000 192.168.126.66 20000 3 192.168.126.68 20000 1
```

D3. sh

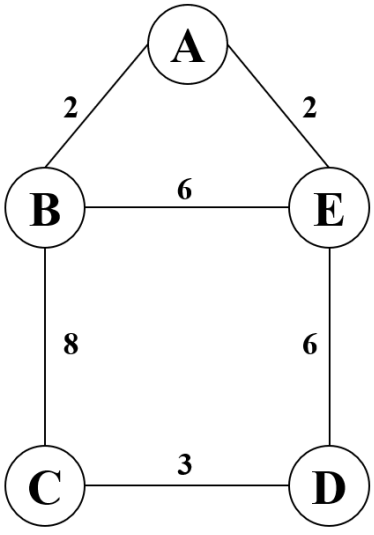
```
python3 DVroute.py 20000 192.168.126.66 20000 1 192.168.126.67 20000 1
```

任务1：模拟路由收敛

任务要求



- 已知网络拓扑如左图所示，请使用DV算法模拟该网络的迭代收敛过程。
- 程序要求：
  - 建议使用python3编程；
  - 使用socket编程实现分布式；
  - 每次迭代后（每隔Interval，如30s），各节点输出路由表，输出格式可参考本课件“算法示例：一次迭代后的路由表”；
  - 输出收敛后的路由表，即输出每对节点间的最短距离和下一跳。



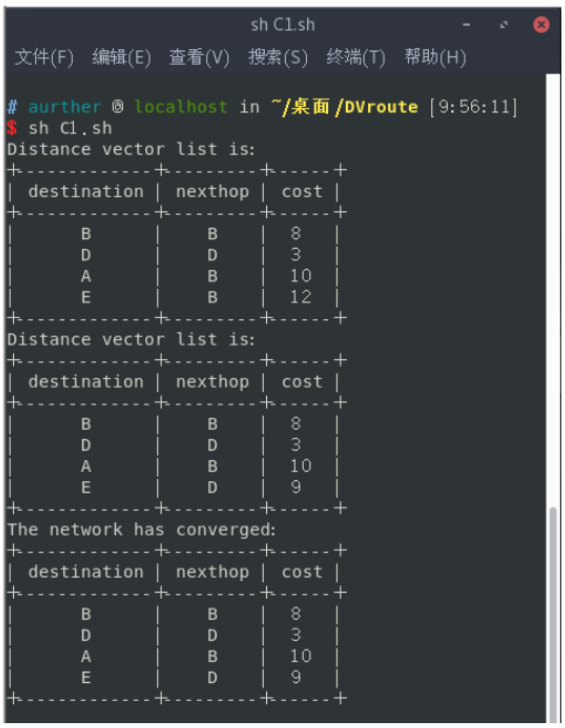
- 实验报告要求：
  - 给出程序源代码，并附上程序运行结果截图；
  - 编程风格良好，注释充分。

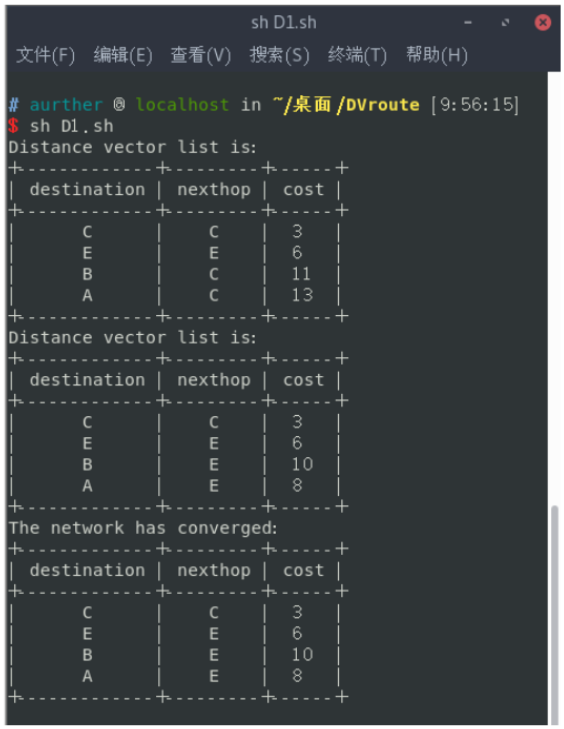
Solution

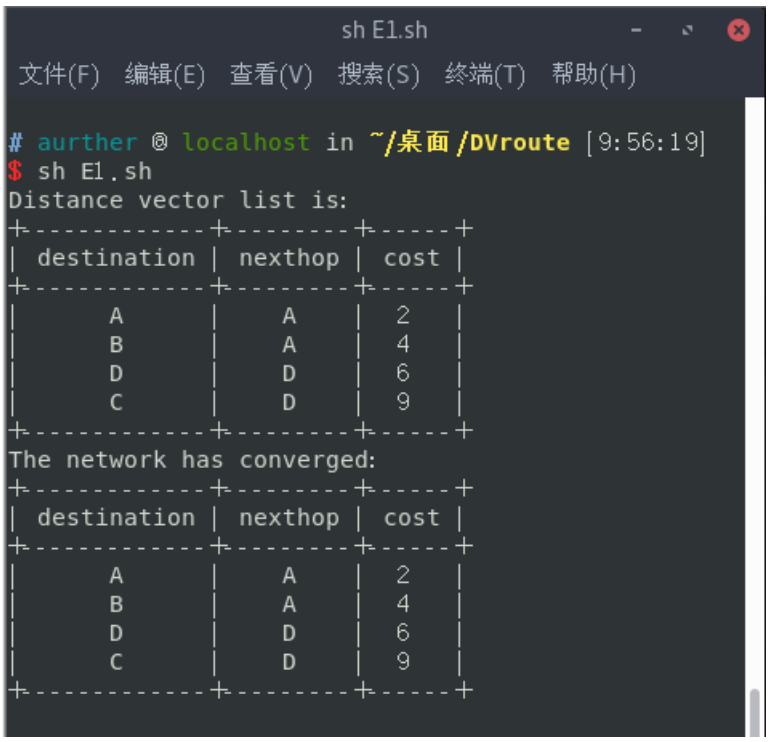
开启A、B、C、D、E五台虚拟机，分别运行shell脚本

```
sh 路由器名1.sh
```

收敛后结果如下：



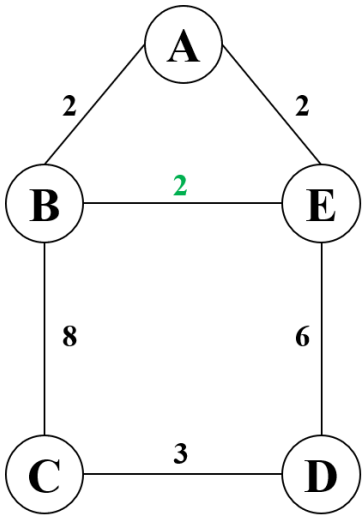




对照网络拓扑图后，验证收敛正确

任务2：模拟拓扑变化

任务要求



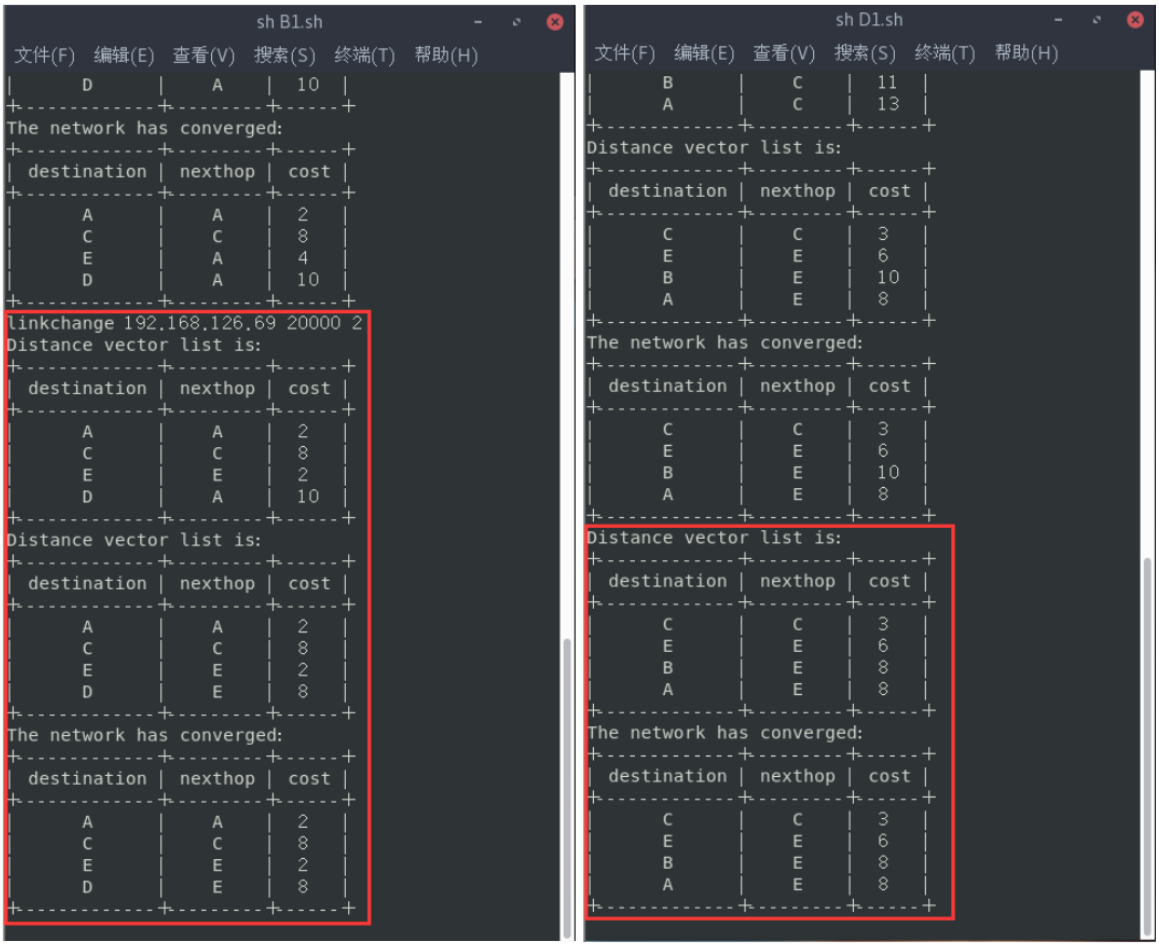
- 在任务1的网络收敛后，将B和E之间的距离更改6→2（好消息！），模拟该变化导致的重新收敛过程。
- 程序要求：
  - 同任务1
- 实验报告要求：
  - 同任务1

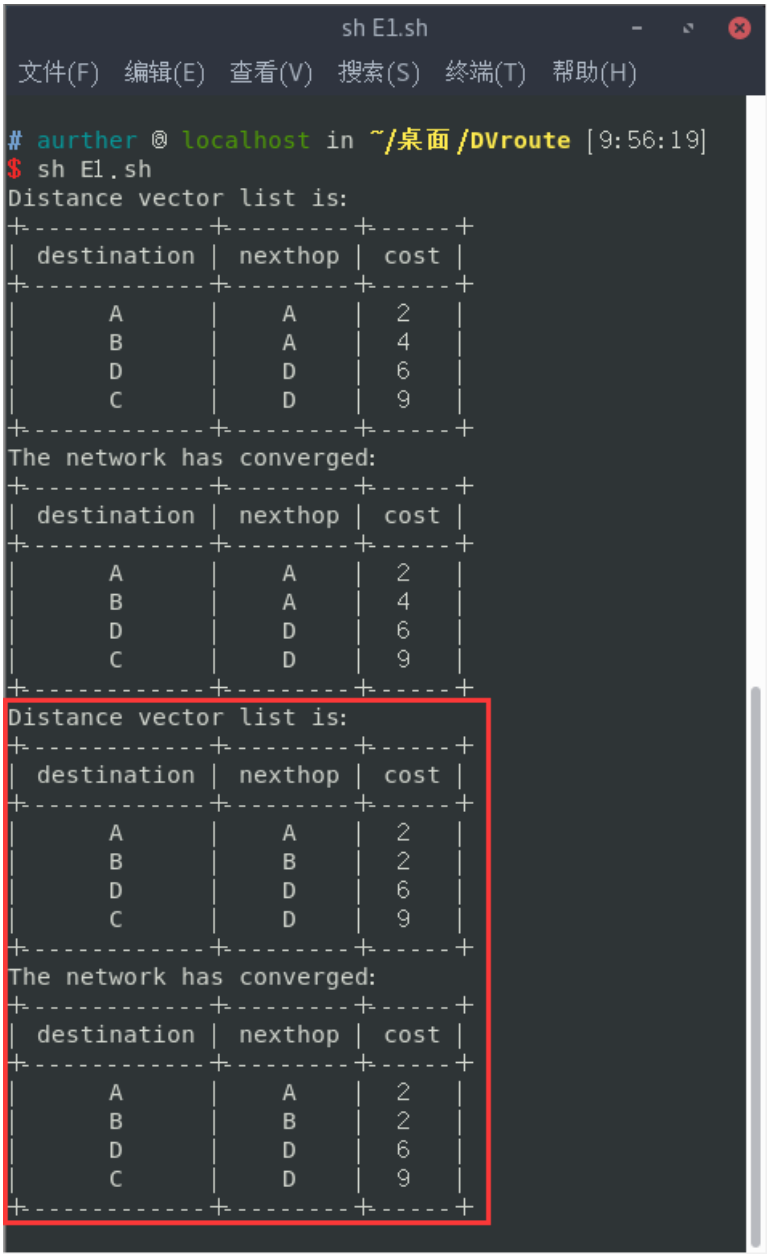
Solution

在任务一的基础上在B上继续输入距离改变命令

```
linkchange 192.168.126.69 20000 2
```

距离改变后，A、C的路由表未发生变化，得到B、D、E变化如下

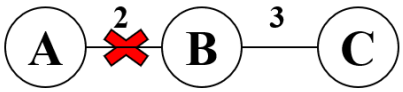




对照网络拓扑图后，验证收敛正确

任务3：制造路由回路

任务要求



- 将左图拓扑的A和B连接断开（坏消息！），模拟该变化导致的重新收敛过程。
- 程序要求：
  - 同任务1
- 实验报告要求：
  - 同任务1

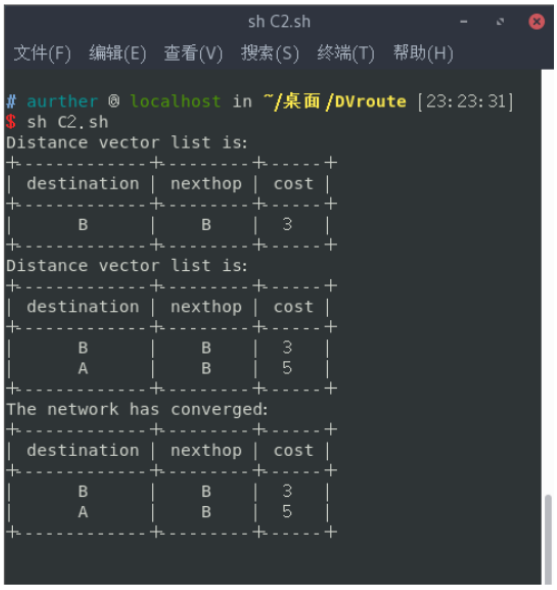
Solution

开启C、B、A三台虚拟机，分别运行shell脚本

注意：运行shell脚本顺序C必须在B前面，否则无法观察到路由回路效果（这是因为DV路由算法中若自己的路由表项的某一目的路由的下一跳为邻居X，则直接用邻居X发来的路由表项进行更新而不进行比较，因此要让B赶在C之前根据C发来的路由表替换关于目的路由A的信息，所以C要在B前启动）

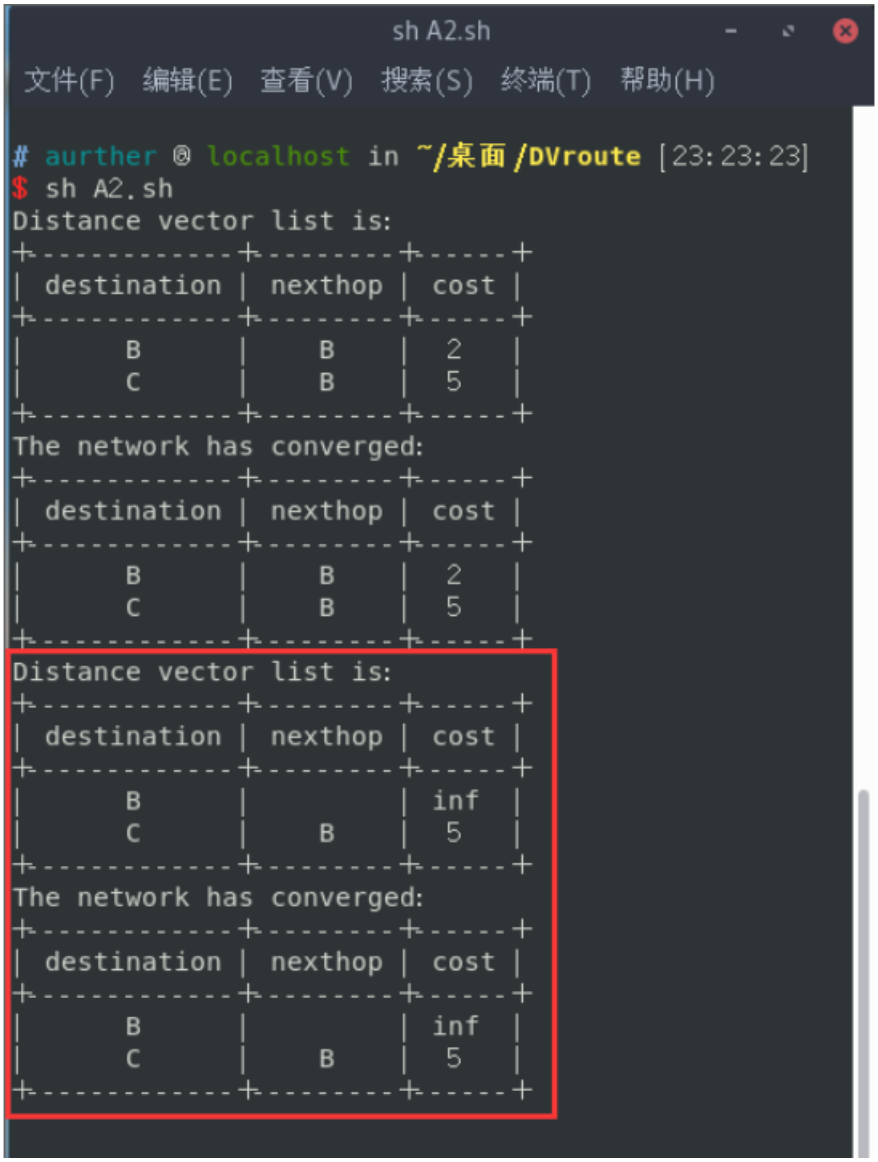
```
sh 路由器名2.sh
```

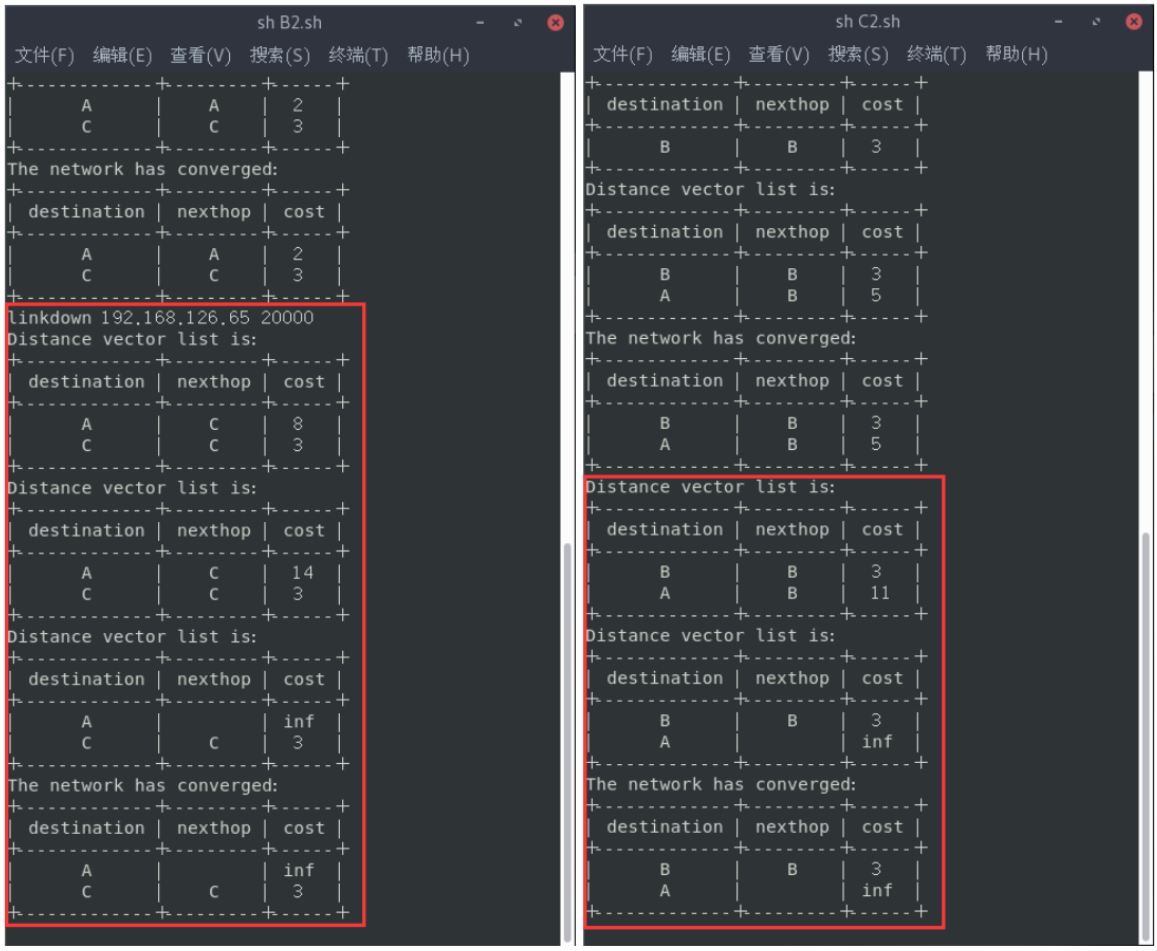
收敛后结果如下：



在B中输入如下命令

```
linkdown 192.168.126.65 20000
```

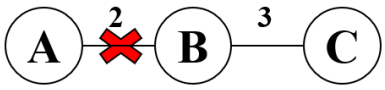

sh A2.sh
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
# aurther @ localhost in ~/桌面/DVroute [23:23:23]
$ sh A2.sh
Distance vector list is:
destination	nexthop	cost
B	B	2
C	B	5
The network has converged:
destination	nexthop	cost
B	B	2
C	B	5
Distance vector list is:
destination	nexthop	cost
B		inf
C	B	5
The network has converged:
destination	nexthop	cost
B		inf
C	B	5



观察可发现B、C之间产生了路由回路

任务4：解决路由回路

任务要求



- 如果使用逆向毒化技术，重新模拟A和B链接断开所导致的重新收敛过程。
- 程序要求：
  - 同任务1
- 实验报告要求：
  - 同任务1

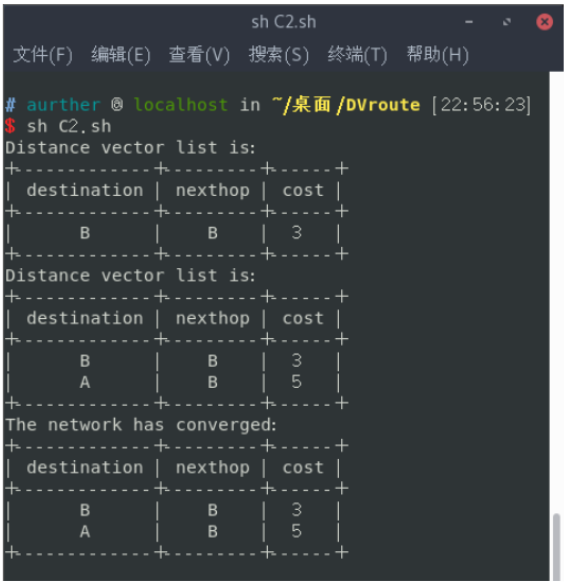
Solution

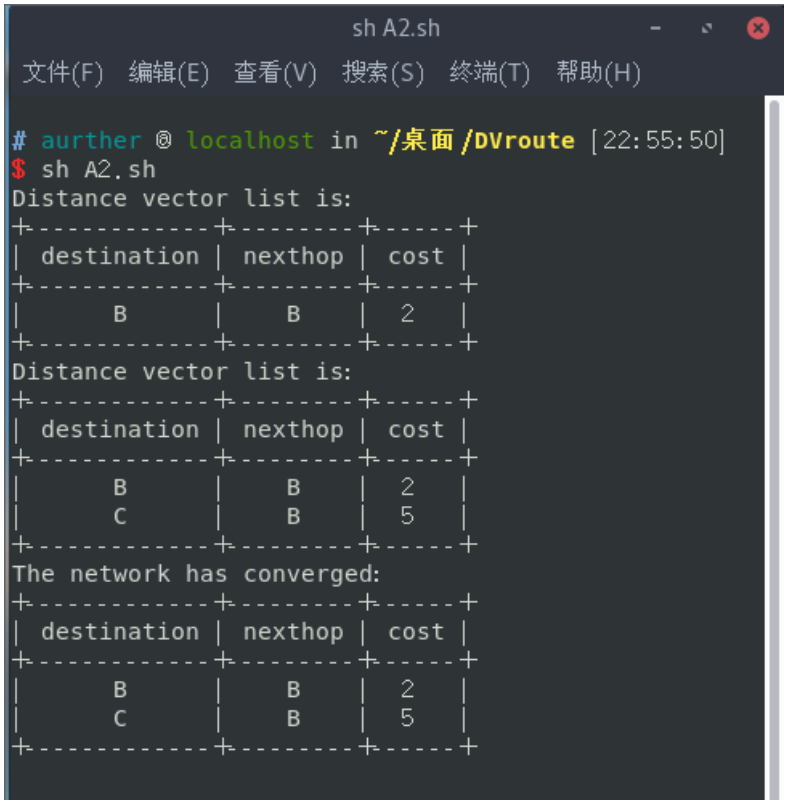
此任务需修改DVroute.py代码

```
318 def main():
319     '''主函数调用该路由器，生成一个最大跳数为15的路由器'''
320     ip = get_host_ip()
321     parsed = parse_argv()
322     rt = Router(router_address=(ip, parsed['listening_port']),
323                 neighbor=parsed['neighbor'],
324                 addr2rName=addr2rName,
325                 MaxHop=15)
326     #此处设置为逆向毒化算法为关闭状态，若要使用，将其注释即可
327     #rt.setPoisonReverse(openState=False)
328     rt.start()
329
```

保存后同任务三一样进行相应操作

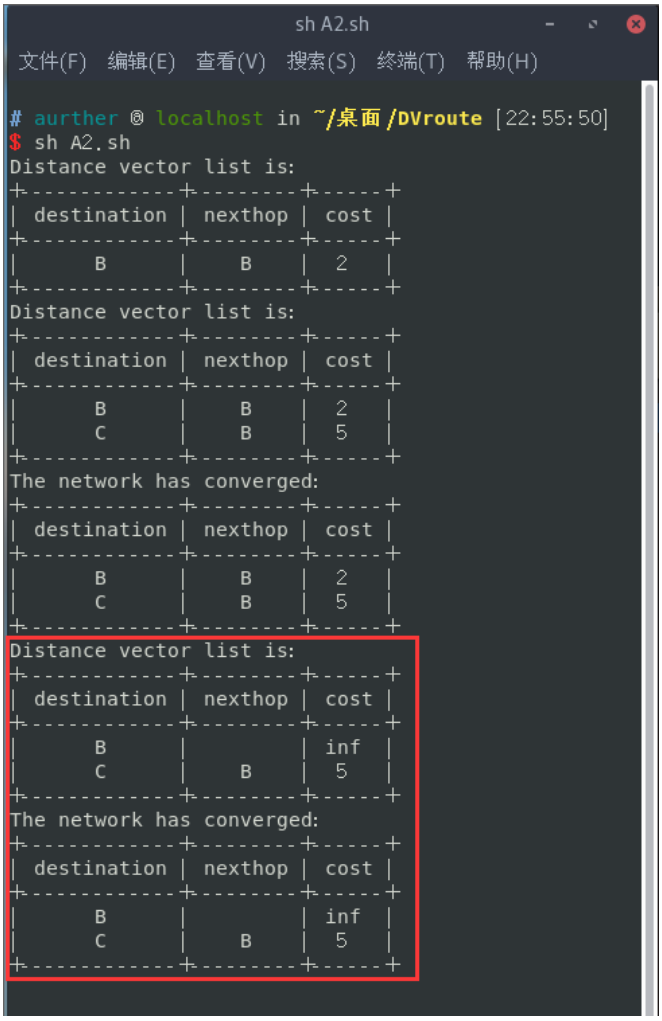
收敛后结果如下：

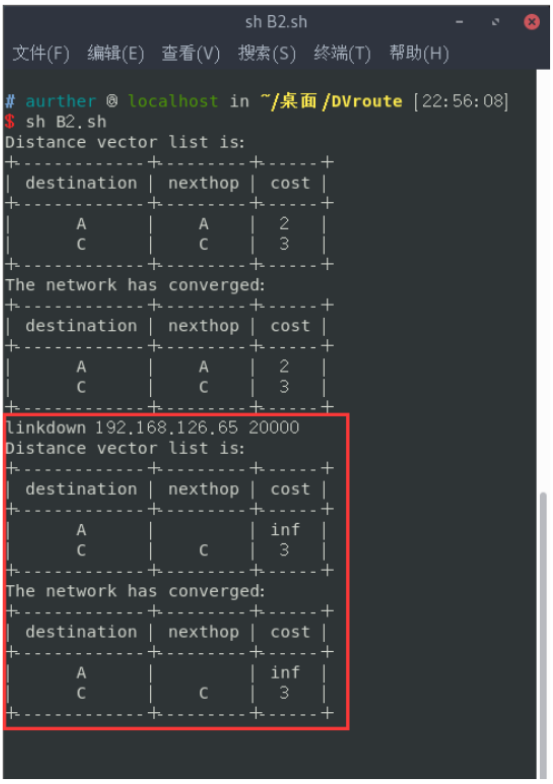


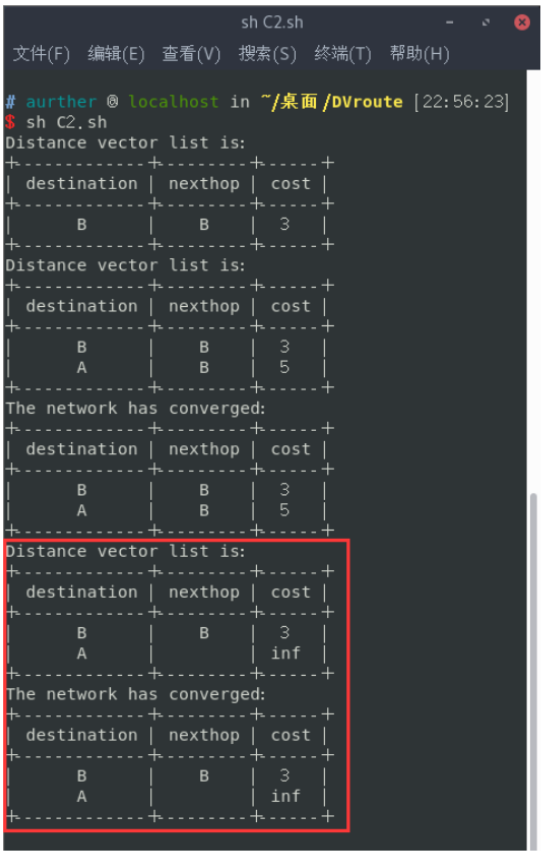


在B中输入如下命令

```
linkdown 192.168.126.65 20000
```







可以发现不产生路由回路

思考题

任务要求

- 请举例说明为什么逆向毒化不能杜绝回路生成（提示：参考[RFC1058]）

Solution

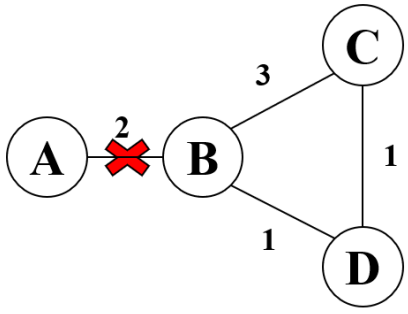
查阅RFC1058，发现如下一段话：

2.2.2. Triggered updates

Split horizon with poisoned reverse will prevent any routing loops that involve only two gateways. However, it is still possible to end up with patterns in which three gateways are engaged in mutual deception. For example, A may believe it has a route through B, B through C, and C through A. Split horizon cannot stop such a loop. This loop will only be resolved when the metric reaches infinity and the network involved is then declared unreachable. Triggered updates are an attempt to speed up this convergence. To get triggered updates, we simply add a rule that whenever a gateway changes the metric for a route, it is required to send update messages almost immediately, even if it is not yet time for one of the regular update message. (The timing details will differ from protocol to protocol. Some distance vector protocols, including RIP, specify a small time delay, in order to avoid having triggered updates generate excessive network traffic.) Note how this combines with the rules for computing new metrics. Suppose a gateway's route to destination N

意思是逆向毒化只能杜绝仅涉及两个网关的回路生成，而不能杜绝涉及三个网关之间的互相欺骗而导致的回路生成。

以下作者制造一个涉及三个路由器的路由回路：



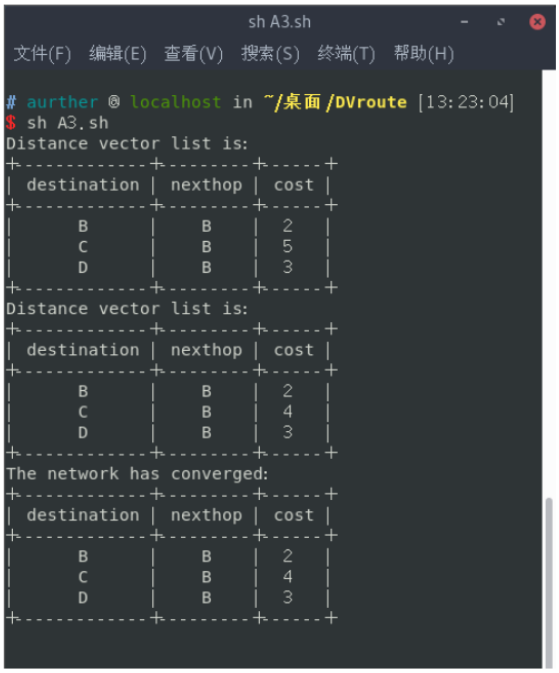
- 使用逆向毒化技术，重新模拟A和B链接断开所导致的重新收敛过程。

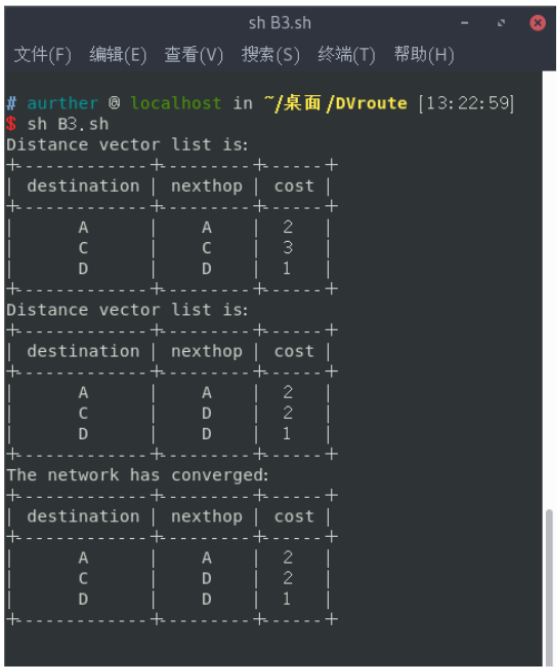
开启C、B、D、A四台虚拟机，逆向毒化设置为开启状态，分别运行shell脚本

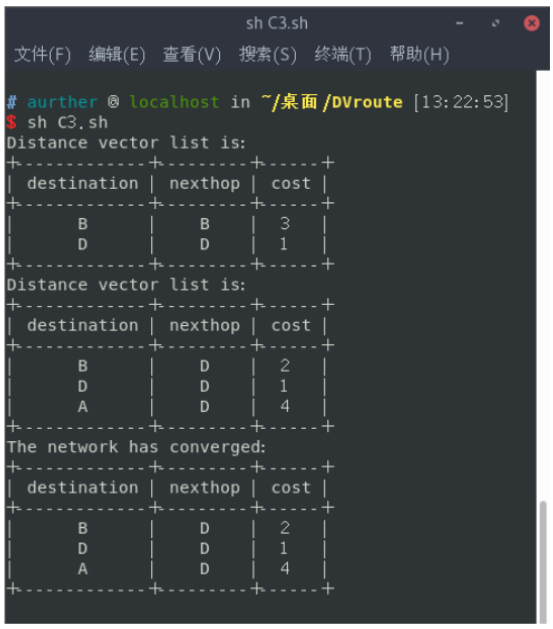
注意：运行shell脚本顺序应是C、B、D、A，否则无法观察到路由回路效果

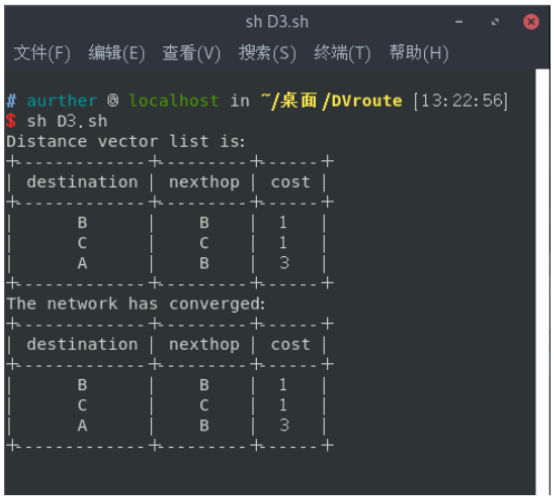
```
sh 路由器名3.sh
```

收敛后结果如下：



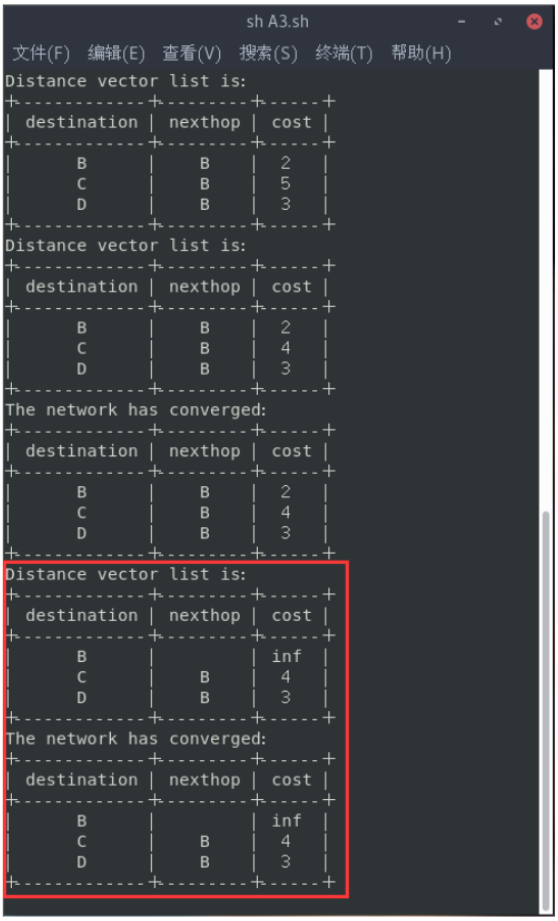


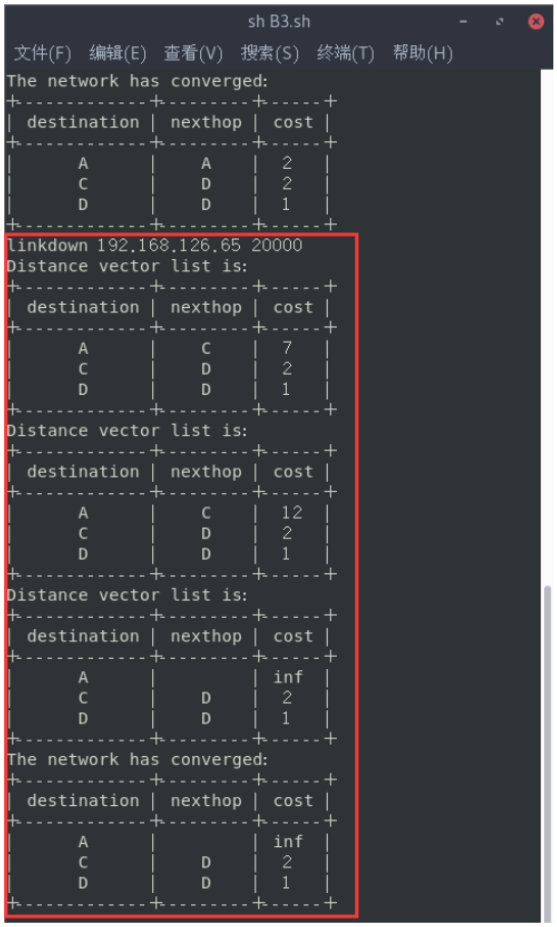


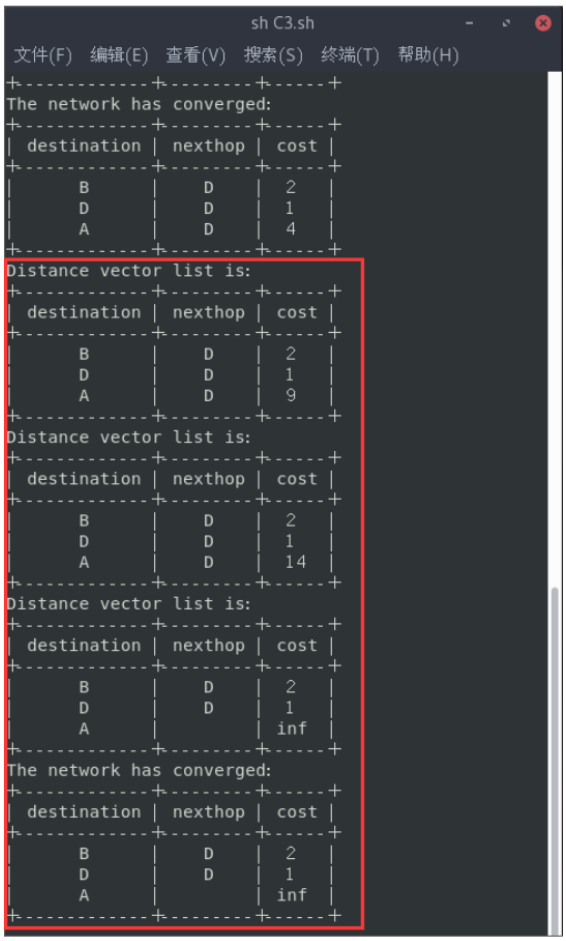


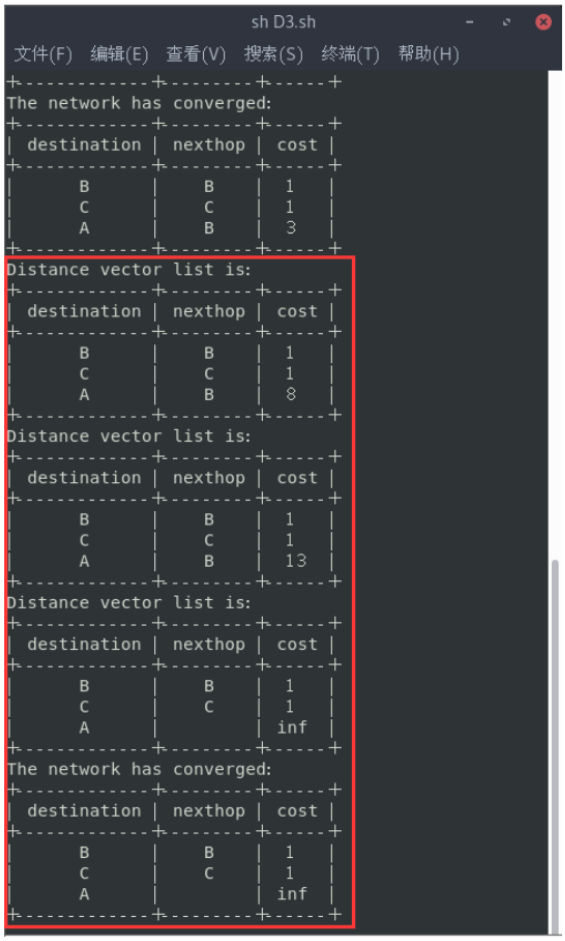
在B中输入如下命令

```
linkdown 192.168.126.65 20000
```









可以观察到B、C、D之间产生一个路由回路

之所以逆向毒化技术无法杜绝回路生成，是因为其定义——若向邻居X发送的路由表项中某一项的下一跳是邻居X,则将跳数设为不可达。因此其只能杜绝仅涉及两个网关的路由回路，而对于最佳路径上的下一跳是某个邻居的邻居，当有坏消息，则可能造成三个网关的互相欺骗而形成路由回路。

三、实验小结

本次实验是对DV路由算法的仿真，算法不算困难，在实验三socket编程实验中对聊天室程序的server.py做进一步修改即可完成实验。笔者通过继承socket.socket类生成一个Router类，方便对各项函数进行联系和管理。此次实验十分有趣，加深了我对路由算法的理解和认识，以及进一步熟练掌握python编程。另外，遗憾的是由于时间有限，笔者本次只实现了更新定时器一个功能，对于其他定时器功能，还需笔者对RFC和路由算法做进一步了解后方可实现。