网络路由实验实验报告

Xuan

3.Init 函数内定义了路由器类及基本参数，包括 self.routersNext（表示该路由器到目标路径经过的下一跳路由器）、self.routersCost（表示目标路径的总链路花销）
self.routersPort 表示邻居路由器和 client、self.routersAddr 表示邻居的序号

handlePacket 函数是处理 client 发出的数据表 traceroute packet 和路由器发出的数据包 routing packet 的。对 traceroute packet，根据其目的地选择下一跳路由器和到下一跳的链路，从而把包发过去。routing packet 的 content 是邻居路由器新的局部 DV，用于更新该路由器的路由表（即执行 updateNode 函数），若 rtn != None 则表示路由表确实更新了，需要广播到邻居节点，即把新的路由表内容作为 content 装到本路由器的 routing packet 中，发送到邻居节点。本路由器的地址是 routing packet 的源节点地址 src，目标地址是 routing packet 的目标地址 dst,cost 是到目标地址的新花销。

updateNode 函数就是根据邻居节点 src 发过来的新的路由表 content 更新本路由器路由表信息。如果目标节点 dst 的链路还没有放进路由表中且目标节点不是本路由器的地址（self.addr），就先把目标节点及其链路花销（=到邻居节点 src 的花销+邻居节点到目标节点的花销）更新进路由表；如果原来的路由表已经有了目标节点 dst 的距离矢量，且 src in self.routersCost(即 packet 的源地址 src 是本路由器的邻居节点)，则与更新后的距离矢量进行比较，取花销最小的进行更新。比较方法是如果到目标节点 dst 的下一跳路由器不是发包过来的邻居节点 src，则取 self.routersCost[dst]和 self.routersCost[src] + cost 的最小值；否则则是发生了链路故障，需重新更新路由表

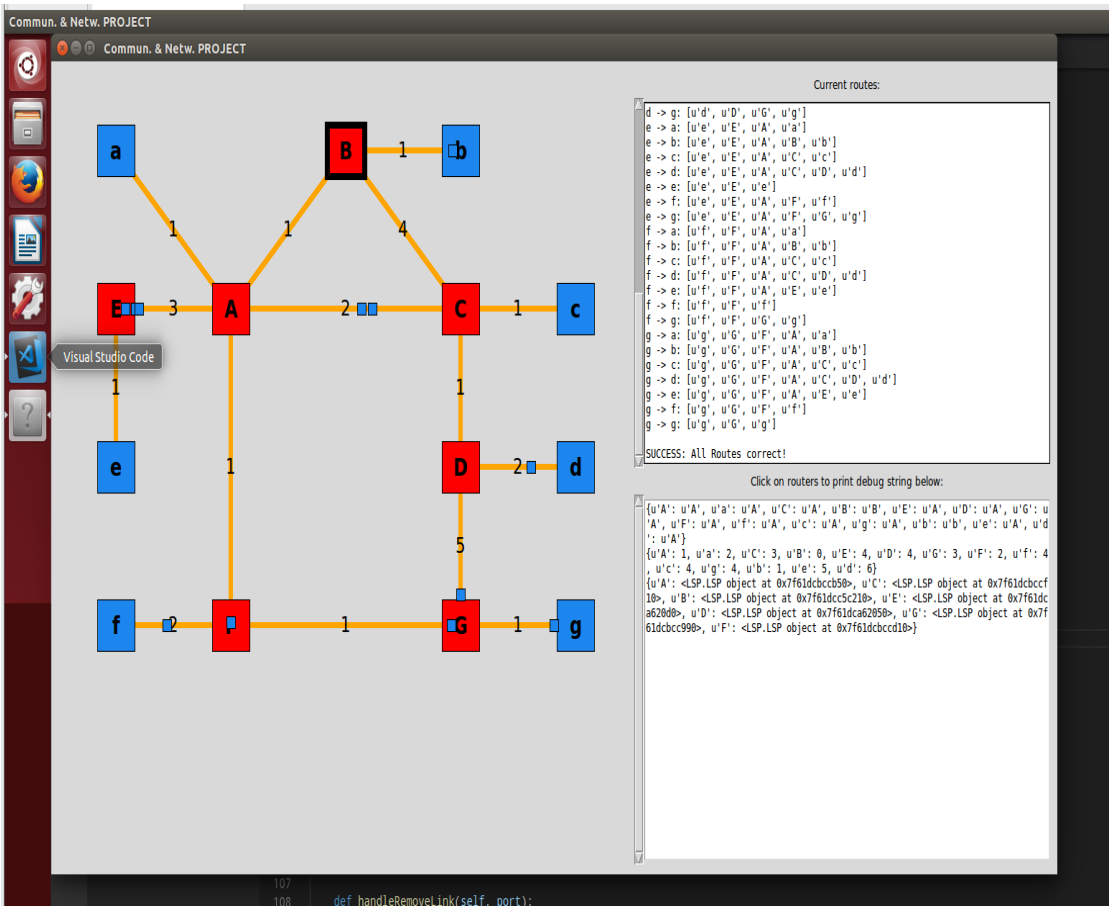
debugString 函数则是调试功能，可以把路由器的 routersAddr、routersNext、routersCost 等信息展示在 gui 右下角。

4.思考题
（1）self.routersNext 表示该路由器到目标路径经过的下一跳路由器
self.routersCost 表示目标路径的总链路花销
self.routersPort 表示到某个邻居节点的链路序号
self.routersAddr 的数组序号对应着该序号链路通向的邻居节点
要实现的算法也就是要正确表达每台路由器到其他节点的 self.routersCost 和 self.routersNext(其中 setCostMax 已正确表达到自身的)
(2)(3)
与 DV 算法不同，LS 算法中 routing packet 的 content 装的是 addr,seqnum,nbcost;
每台路由器创建一个链路状态数据包 (LSP)，其中包含与该路由器直连的每条链路的状态。这通过记录每个邻居的所有相关信息，包括 addr(本路由器地址)、seqnum(邻居的序号)、nbcost（到各个邻居的链路花销）。一旦建立了邻接关系，即可创建 LSP，并仅向建立邻接关系的路由器发送 LSP。而当本路由器接收到邻居的 LSP 时，会建立一个数据库，把所有节点（不仅是邻居）的 LSP 相应地保存为 self.routersLSP[addr]。（自身的 LSP 保存到 self.routersLSP[self.addr]），同时也会所有节点的 LSP 发到除去发给本路由器的邻居之外

的所有其他邻居。
例如，B 会把 B 的邻居 C 的 LSP[C]发给邻居节点 A，则 packet 的内容是，packet.srcAddr=B，packet.dstAddr=A,packet.content 里存储的信息是 Addr=C、seqnum=C 的所有邻居序号，nbcost=C 到相应的 seqnum 的花销。而 A 接收到 packet 后会把 content 的信息存储到 self.routersLSP[C]中，再发给除去 B 外的所有邻居。
LSP.py 中的更新函数 updateLSP 用于处理 routing packet，功能是更新（self.routersLSP[addr]）。实现是如果某个节点的 LSP 的 seqnum（邻居序号）>=新发过来的 seqnum 说明节点 Addr 没获取到新邻居，或者各花销与原花销一样，则不用更新，否则就更新 self.routersLSP[addr]的 seqnum 和 nbcost
4.编写的 calpath 函数如下：

```
def calPath(self):
    # Dijkstra Algorithm for LS routing
    self.setCostMax()
    # put LSP info into a queue for operatisons
    Q = PriorityQueue()
    ListN = [self.addr]
    for addr, nbcost in self.routersLSP[self.addr].nbcost.items():
        Q.put((nbcost, addr, addr))
    while not Q.empty():
        Cost, Addr, Next = Q.get(False)
        """TODO: write your codes here to build the routing table"""
        if Addr not in self.routersCost or self.routersCost[Addr] >= Cost:
            self.routersCost[Addr] = Cost
            self.routersNext[Addr] = Next
            ListN = ListN + [Addr]
            if Addr in self.routersLSP:
                for addr, nbcost in self.routersLSP[Addr].nbcost.items():
                    if addr not in ListN:
                        Q.put((nbcost + Cost, addr, Next))
```

结果如下



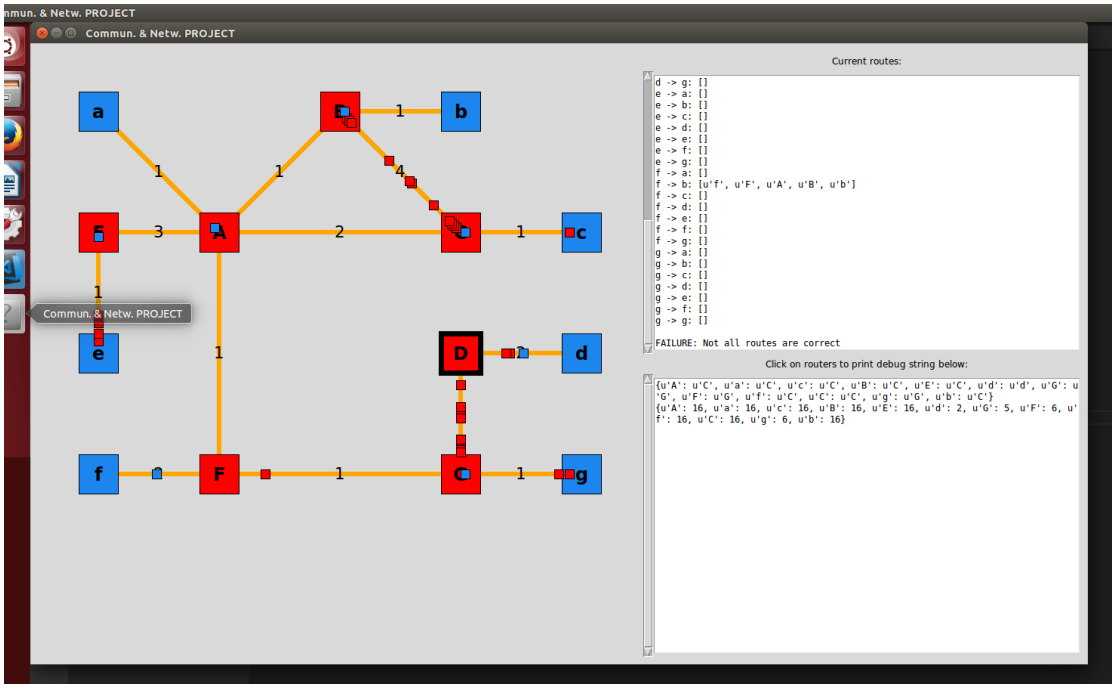
6.链路故障分析

由 0_net_event.json 和 gui 界面可知，链路故障为一段时间后 C 和 D 之间的链路被移除

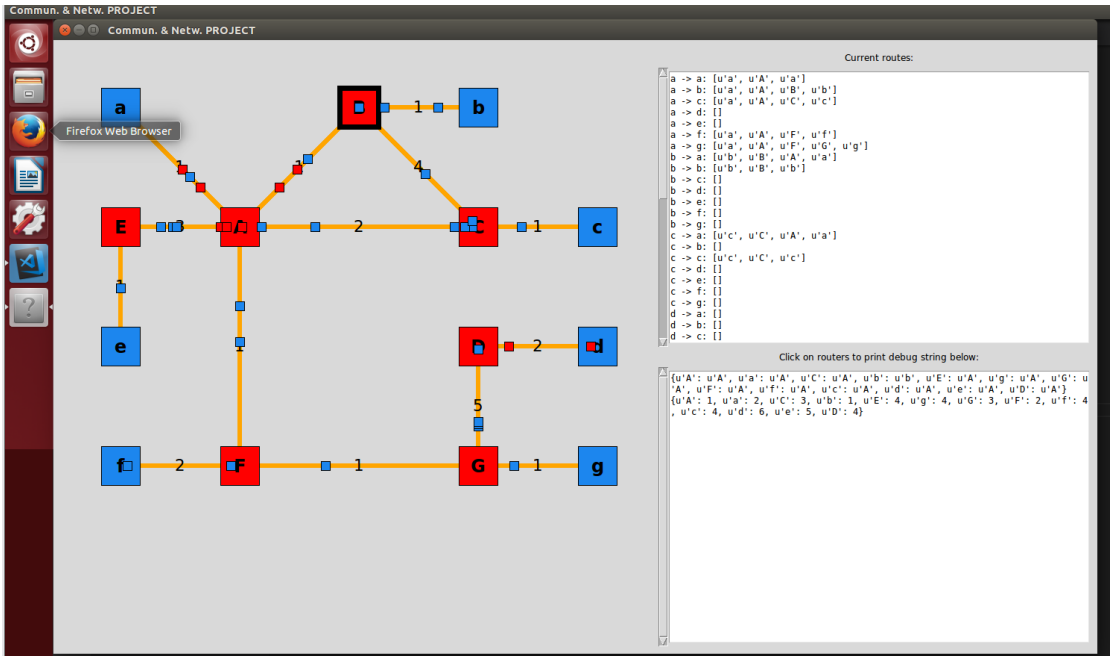
DV 算法:

观察发生故障节点之一 D 路由器:

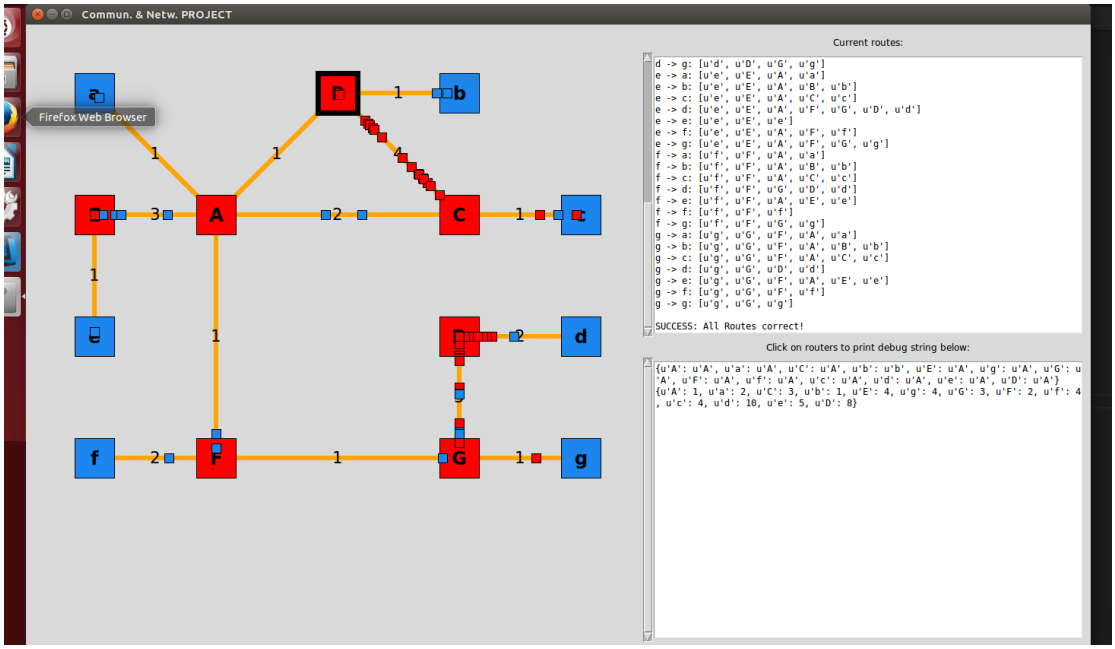
发生故障时的路由表为



可见 CD 链路故障后，通过 handleRemoveLink 函数把到 C 和 c 的距离设置成了 16（cost_max），再重新执行 DV 算法

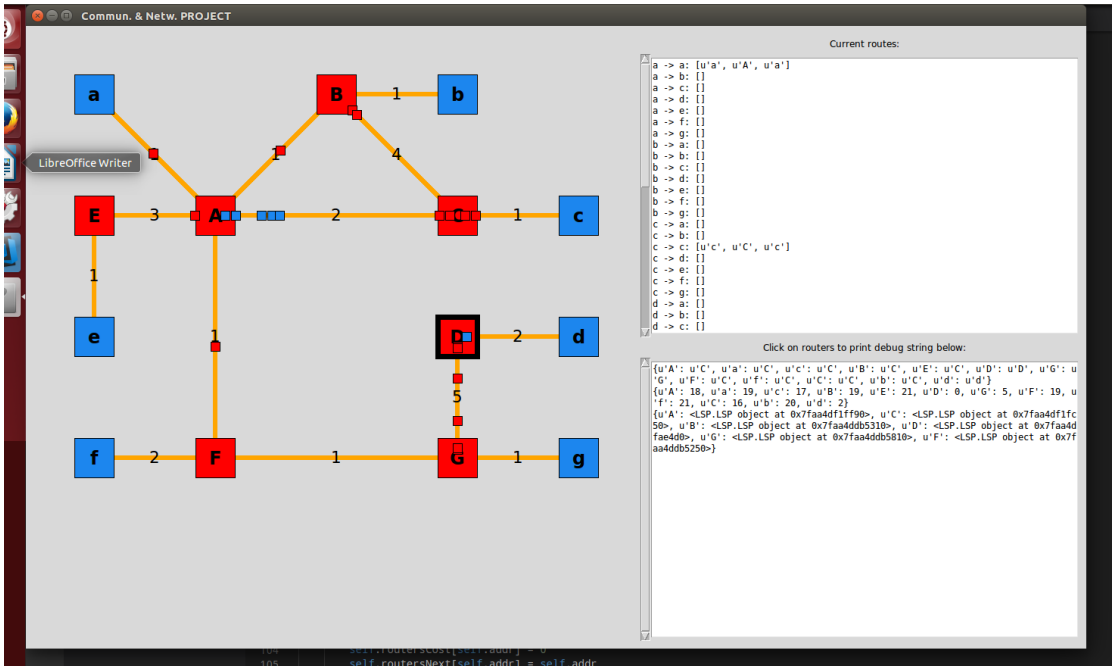


但是在发生故障后，其他节点需要一定时间（即新的 DV 算法执行后）才能知道 CD 间的链路被移除了，所以会仍按照原来的链路来走，导致错误的答案，如图中 B 路由器到 D 路由器，原来的距离是 4（B-A-C-D），链路故障后显然不对，需一定时间的迭代才能得出正确路径，如下图

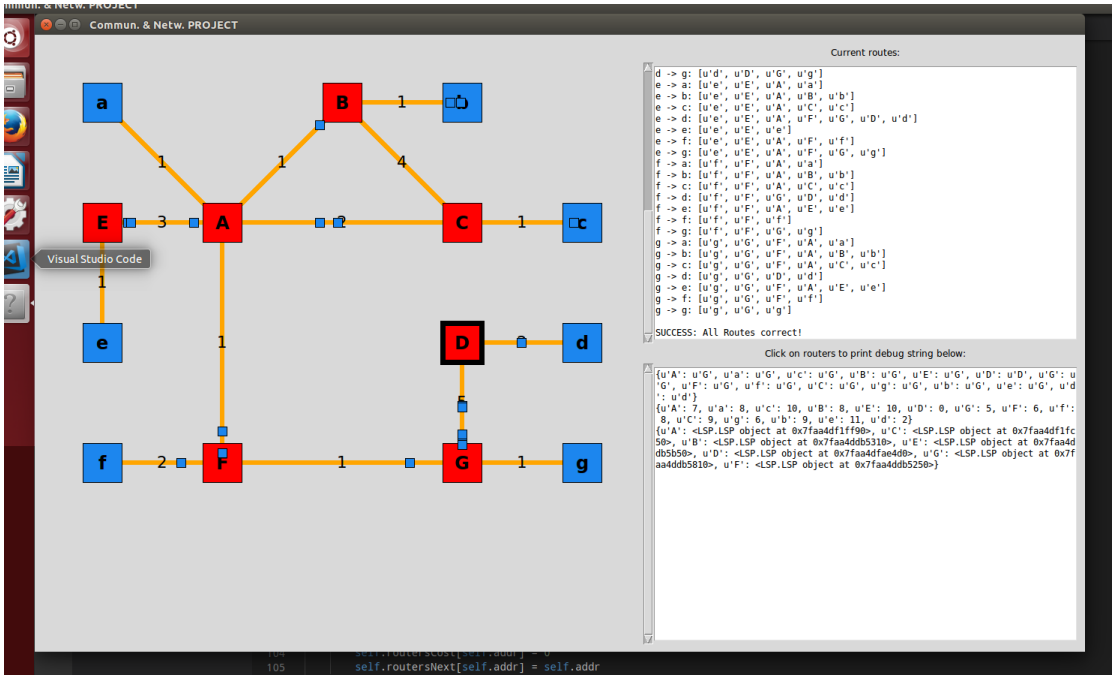


LS 算法：

仍观察 D，CD 间链路故障后，D 的路由表如下：



可知因更新后的 LSP 未转发到位导致路由结果是错误的。经过一段时间的转发后 D 节点终于得到新的正确的全局 LSP 信息，执行一次 Dijkstra 算法即可得到正确结果，所以收敛速度比 DV 算法快



附件：

代码：

```
def calPath(self):
        # Dijkstra Algorithm for LS routing
        self.setCostMax()
        # put LSP info into a queue for operatisons
        Q = PriorityQueue()
```

```
ListN = [self.addr]
for addr, nbcost in self.routersLSP[self.addr].nbcost.items():
    Q.put((nbcost, addr, addr))
while not Q.empty():
    Cost, Addr, Next = Q.get(False)
    """TODO: write your codes here to build the routing table"""
    if Addr not in self.routersCost or self.routersCost[Addr] >= Cost:
        self.routersCost[Addr] = Cost
        self.routersNext[Addr] = Next
        ListN = ListN + [Addr]
        if Addr in self.routersLSP:
            for addr, nbcost in self.routersLSP[Addr].nbcost.items():
                if addr not in ListN:
                    Q.put((nbcost + Cost, addr, Next))
```