网络层路由实验

无04 2019012137 张鸿琳

实验目的

- 理解和掌握链路状态法和距离向量法的实现原理和区别；
- 理解和掌握链路状态法和距离向量法处理链路故障和新增链路的方式；
- 分析和对比链路状态法和距离向量法的计算复杂度和实际收敛速度；
- 初步掌握编程实现路由选择算法的能力。

实验内容

- 理解实验原理，熟悉网络仿真器和图形化交互式界面；
- 阅读实验提供代码，掌握网络仿真器运行原理，理解网络仿真器中数据包、链路、客户端和路由器的具体实现；
- 阅读并补全链路状态法的关键代码，确定算法原理和代码实现的对应关系，掌握 Dijkstra 算法在其中的应用；
- 在网络仿真器中运行链路状态法，对比链路状态正常、链路故障和链路新增三种情况时路由路径和路由表变化；
- 在网络仿真器中运行距离向量法，对比链路状态正常、链路故障和链路新增三种情况时路由路径和路由表变化；
- 记录并对比链路状态法和距离向量法收敛速度，并分析实验现象。

实验原理

路由是网络层至关重要的功能，需要确定从发送方到接收方所采用的一个“好”的路径，具体体现在成本最低、速度最快、开销最小等标准，为方便表述，下文统一用开销最小作为“好”的路径标准。具体而言，网络通过路由协议事先约定路由信息的发送过程的规定和标准，进行路由信息交换。常用的路由协议包括路由信息协议（RIP）、开放式最短路径优先协议（OSPF）和边界网关协议（BGP） 等。路由选择算法则在路由协议中起着至关重要的作用，采用的路由选择算法决定了寻找最终路径的结果。常用的路由选择算法包括链路状态法（Link State，LS， 对应 OSPF 协议）和距离向量法（Distance Vector，DV，对应 RIP 协议），本次实验将针对上述两种路由选择算法进行实现，并根据实验现象对比分析。

链路状态法的基本原理是节点之间传递链路状态和开销等信息，基于收到的链路状态信息，每个节点能够找到通往任意节点的最小开销路径。特别地，网络通过可靠洪泛（Reliable Flooding）保证所有节点都能得到来自其他节点的链路状态信息：一个节点沿着所有与其直接相连的链路把其链路状态信息发送出去，接收到这个信息的每个节点再沿着所有与它相连的链路转发出去，这个过程一直持续直到这个信息到达网络中所有节点。

距离向量法的基本原理是在每个路由器节点构建一个距离向量，其中包含与所有其他路由器节点的开销，并将其距离向量传递给邻居节点。网络在运行距离向量法时，节点之间互相交换其所拥有的距离向量，并更新自身路由表，当所有节点获取一致的路由信息时，则算法收敛。每个节点仅知道自己路由表的内容，即由自身发送数据包到其他节点的距离和下一跳节点。

实验过程与结果

5.2 链路状态法理解与实现

首先对初始化函数中的变量作注释，如下：

```
def __init__(self, addr, heartbeatTime):
    """class fields and initialization code here"""
    Router.__init__(self, addr)  # initialize superclass - don't remove
    self.routersLSP = {}  ### 索引为路由器地址，存储对应路由器的链路状态（邻居节点及到其开销）
    self.routersAddr = {} ### 索引为端口号，存储该端口指向的紧邻路由器地址
    self.routersPort = {} ### 索引为紧邻路由器地址，存储通往该路由器的端口
    self.routersNext = {} ### 索引为所有可到达路由器的地址，存储为了到达该路由器而需途径的紧邻路由器地址（下一个路由器）
    self.routersCost = {} ### 索引为所有可到达路由器的地址，存储当前去往该路由器的总开销
    self.seqnum = 0 ### 发送出的LSP数据包序列号，避免旧数据包信息覆盖新数据包信息
    self.routersLSP[self.addr] = LSP(self.addr, 0, {})

    self.lasttime = None
    self.heartbeat = heartbeatTime
```

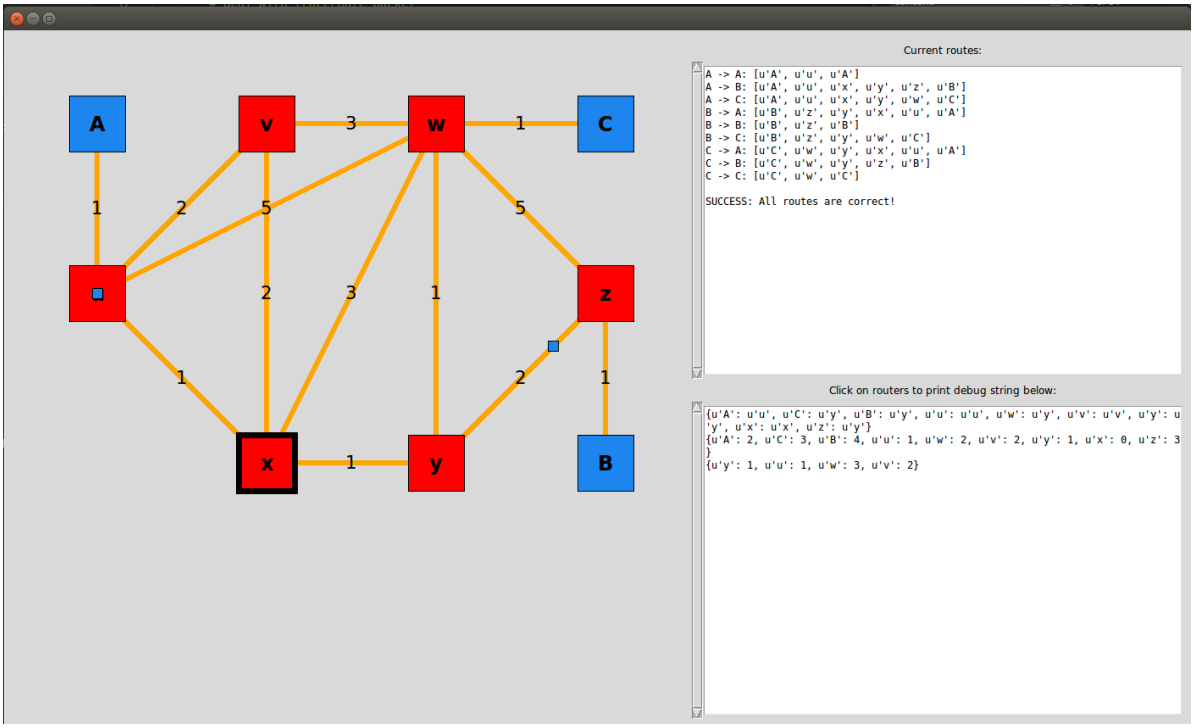
再根据链路状态法原理以及前向搜索算法流程，将路径规划函数补全代码，并对其进行注释，如下：

```
def calPath(self):
    # Dijkstra Algorithm for LS routing
    self.setCostMax()### 对routersCost进行初始化，将本来的开销信息抹除
    # put LSP info into a queue for operations
    Q = PriorityQueue()### 新建一个优先队列，作为试探表，便于找到试探表中开销最低的路径
    for addr, nbcost in self.routersLSP[self.addr].nbcost.items():### 对试探表进行初始化
        Q.put((nbcost, addr, addr))### 将紧邻的几个路由器的开销信息放入优先队列（试探表）
    while not Q.empty():### 当试探表非空时，持续进行规划
        Cost, Addr, Next = Q.get(False)### 取出当前优先队列（试探表）中开销最小的节点

        ### 若该取出的节点此前未有开销记录或者当前路径的开销比已记录开销更小，则据之更新当前记录，相当于加入证实表
        if Addr not in self.routersCost or Cost < self.routersCost[Addr]:
            ### TODO: Add two lines code to update Cost and Next for Addr
            self.routersCost[Addr] = Cost
            self.routersNext[Addr] = Next
            if Addr in self.routersLSP:### 若该新加入证实表的节点存在紧邻节点

                ### 将该新加入证实表的节点的紧邻节点的开销信息加入试探表
                for addr_, cost_ in list(self.routersLSP[Addr].nbcost.items()):
                    Q.put((cost_ + Cost, addr_, Next))
```

在补全代码后，运行正常网络，得到下图：



可以看到最终结果是正确的，记录信息，完成下面表格：

- 客户端之间的最小开销路径：

源客户端-目的客户端	最小开销路径
A→A	A→u→A
A→B	A→u→x→y→z→B
A→C	A→u→x→y→w→C
B→A	B→z→y→x→u→A
B→B	B→z→B
B→C	B→z→y→w→C
C→A	C→w→y→x→u→A
C→B	C→w→y→z→B
C→C	C→w→C

- 路由器 x 的转发表：

目的节点	下一跳转发节点	开销
A	u	2
B	y	4
C	y	3
u	u	1
v	v	2
w	y	2
x	x	0
y	y	1
z	y	3

- 路由器 x 的LSP信息记录：

紧邻节点	开销
y	1
u	1
w	3
v	2

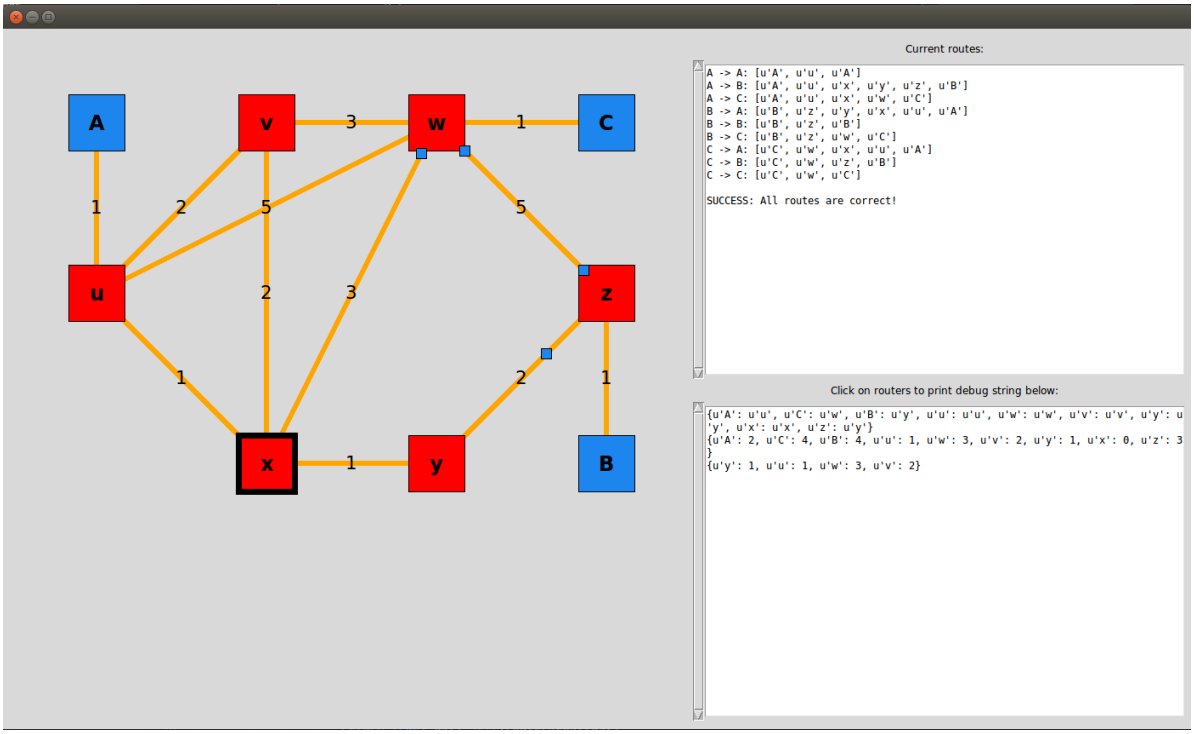
5.3 链路状态改变实验

给移除链路的函数加上注释，如下：

```
def handleRemoveLink(self, port):
    """handle removed link"""
    addr = self.routersAddr[port]### 存储当前路由器通过port端口相连的路由器的地址（即将要断开连接的路由器）
    self.routersLSP[self.addr].nbcost[addr] = COST_MAX### 把到将要断开的路由器的开销改为极大值，则等同于断开
    self.calPath()### 删去链路后重新进行路径规划

    content = {}### 存储即将发送的LSP数据包中的信息
    content["addr"] = self.addr### 当前路由器的地址
    content["seqnum"] = self.seqnum + 1 ### 当前路由器发出的LSP数据包序列号
    content["nbcost"] = self.routersLSP[self.addr].nbcost### 当前路由器到其紧邻节点的开销信息
    self.seqnum += 1### 更新LSP数据包的序列号
    for port1 in self.routersAddr:### 遍历紧邻的路由器
        if port1 != port:### 不把LSP数据包发送给断开连接的那个路由器
            packet = Packet(Packet.ROUTING, self.addr, self.routersAddr[port1], dumps(content))### 生成LSP数据包（routing类型）
            self.send(port1, packet)### 发送LSP数据包给紧邻的几个路由器，随后该LSP数据包会洪泛开来
    pass
```

运行链路故障（移除）网络，利用链路状态法进行路由选择，得到下图：



可以看到最终结果是正确的，记录信息，完成下面表格：

- 客户端之间的最小开销路径：

源客户端-目的客户端	最小开销路径
A→A	A→u→A
A→B	A→u→x→y→z→B
A→C	A→u→x→w→C
B→A	B→z→y→x→u→A
B→B	B→z→B
B→C	B→z→w→C
C→A	C→w→x→u→A
C→B	C→w→z→B
C→C	C→w→C

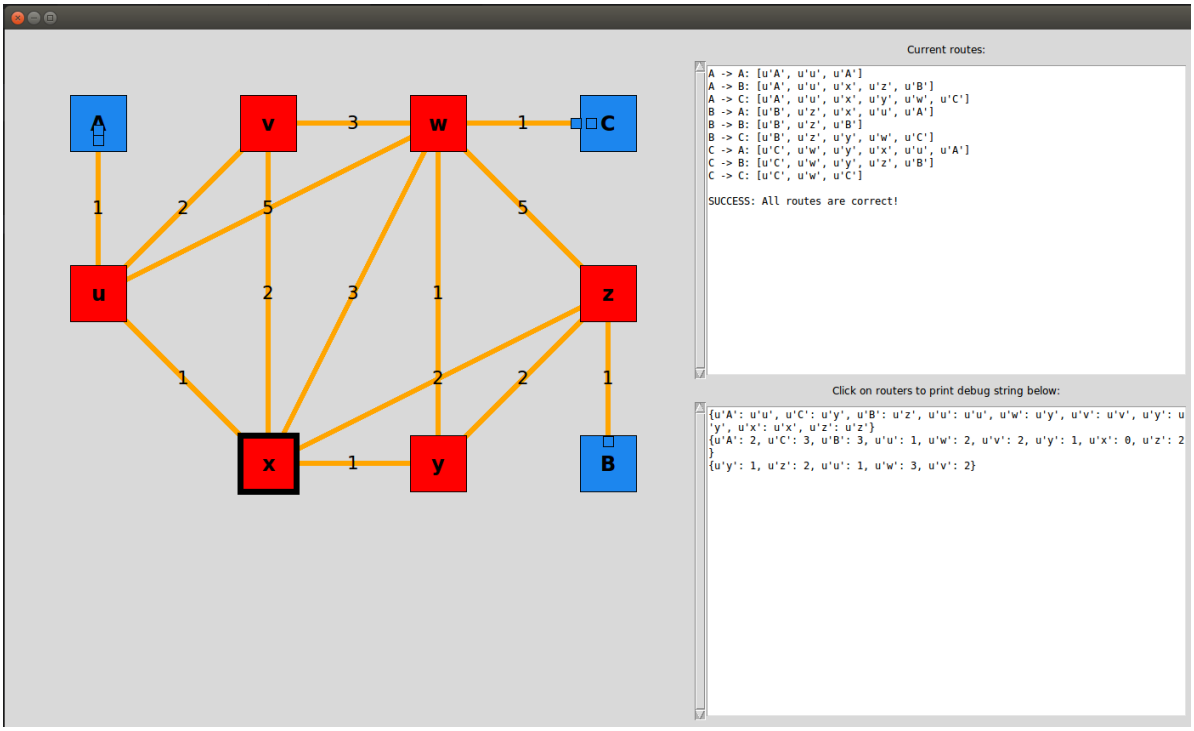
- 路由器 x 的转发表：

目的节点	下一跳转发节点	开销
A	u	2
B	y	4
C	w	4
u	u	1
v	v	2
w	w	3
x	x	0
y	y	1
z	y	3

- 路由器 x 的LSP信息记录如下：

紧邻节点	开销
y	1
u	1
w	3
v	2

此后再运行链路新增网络，得到下图：



可以看到最终结果是正确的，记录信息，完成下面表格：

- 客户端之间的最小开销路径：

源客户端-目的客户端	最小开销路径
A⟶A	A⟶u⟶A
A⟶B	A⟶u⟶x⟶z⟶B
A⟶C	A⟶u⟶x⟶y⟶w⟶C
B⟶A	B⟶z⟶x⟶u⟶B
B⟶B	B⟶z⟶B
B⟶C	B⟶z⟶y⟶w⟶C
C⟶A	C⟶w⟶y⟶x⟶u⟶A
C⟶B	C⟶w⟶y⟶z⟶B
C⟶C	C⟶w⟶C

- 路由器 x 的转发表：

目的节点	下一跳转发节点	开销
A	u	2
B	z	3
C	y	3
u	u	1
v	v	2
w	y	2
x	x	0
y	y	1
z	z	2

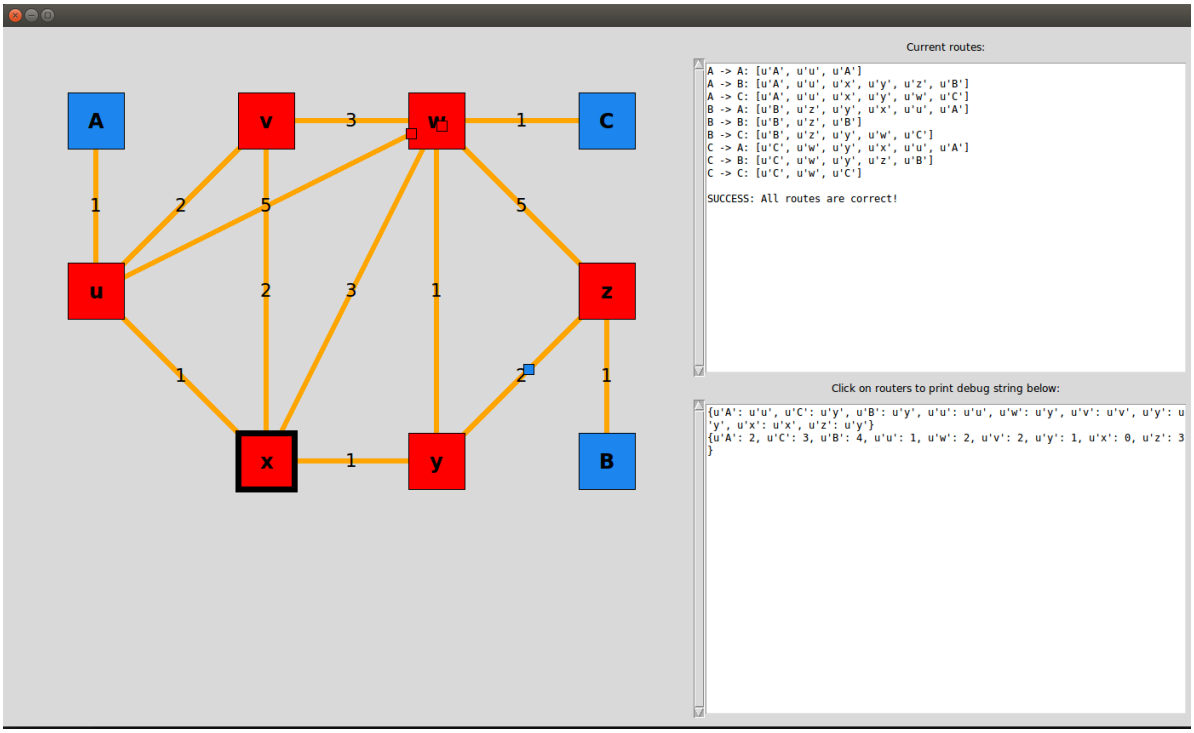
- 路由器 x 的LSP信息记录如下：

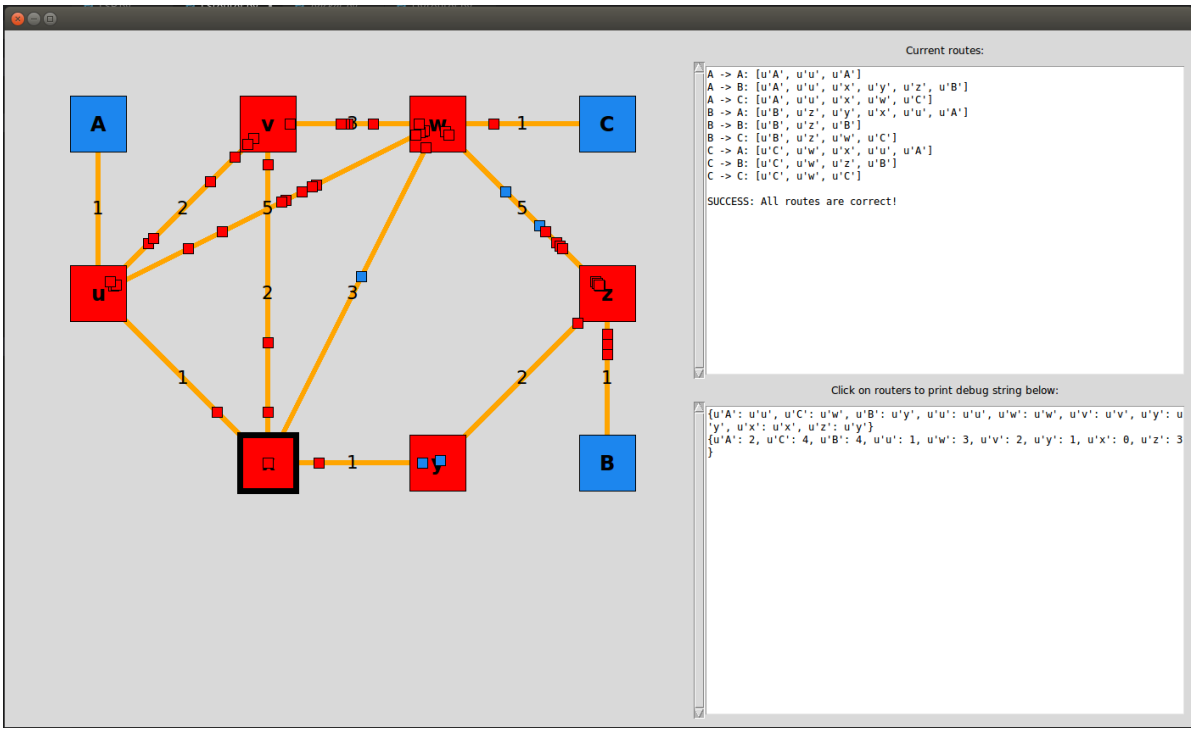
紧邻节点	开销
y	1
z	2
u	1
w	3
v	2

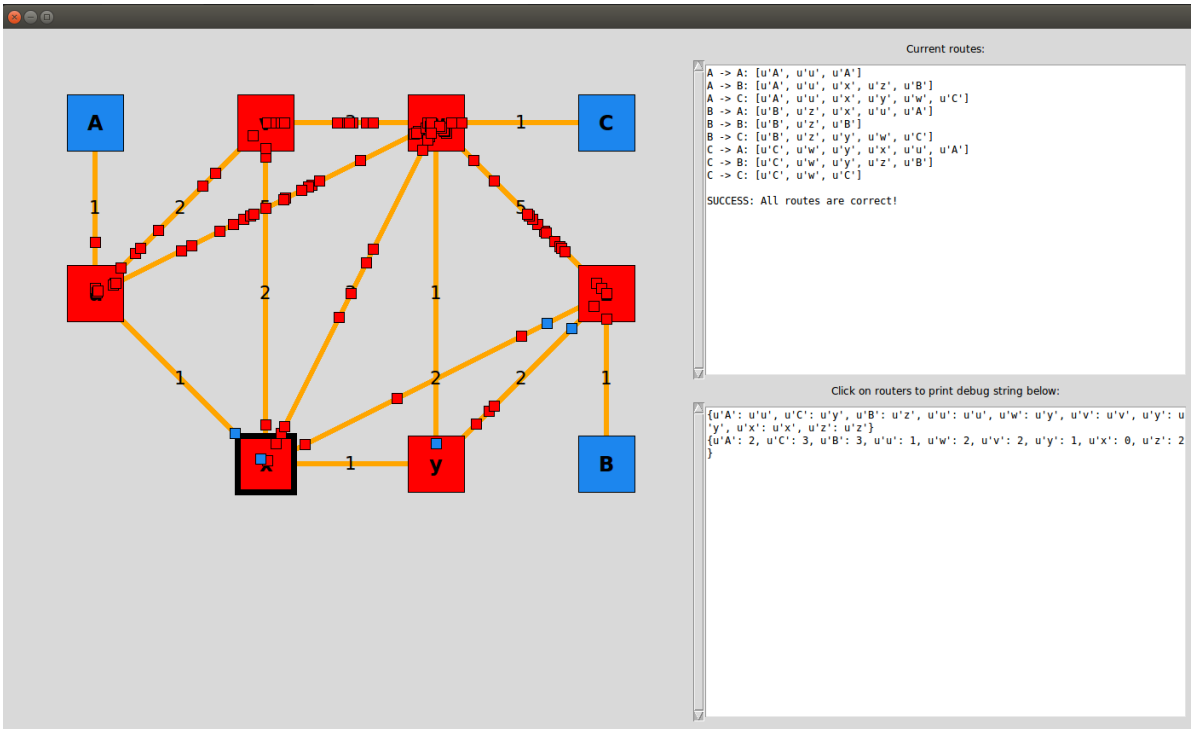
通过比较链路状态改变和正常状态时最小开销路径和路由器转发表、LSP 的信息，可以看到链路状态法处理链路状态改变的过程大致为：当发生某条链路的状态改变，则该链路对应的路由器将会向周围路由器发出更新的 LSP 信息数据包（洪泛开来），随后各个路由器会根据这收到的新的 LSP 信息重新进行路径规划（利用 Dijikstra 算法），进而得到新的转发表（最小开销），所有路由器的转发表都更新完毕后，当有数据包传入网络，该数据包会被路由器（根据其更新过后的转发表）不断转发，这些转发串联起来就是新的最小开销路径了。

5.4 距离向量法理解与实验

利用距离向量法对上面三种情况的链路网络进行路径规划（得到转发表），得到最终运行结果如下图（依次为正常、链路故障、链路新增）：


A
v
w
C
u
z
x
y
B
1
2
3
5
Current routes:
A -> A: [u'A', u'u', u'A']
A -> B: [u'A', u'u', u'x', u'y', u'z', u'B']
A -> C: [u'A', u'u', u'x', u'y', u'w', u'C']
B -> A: [u'B', u'z', u'y', u'x', u'u', u'A']
B -> B: [u'B', u'z', u'B']
B -> C: [u'B', u'z', u'y', u'w', u'C']
C -> A: [u'C', u'w', u'y', u'x', u'u', u'A']
C -> B: [u'C', u'w', u'y', u'z', u'B']
C -> C: [u'C', u'w', u'C']
SUCCESS: All routes are correct!
Click on routers to print debug string below:
{u'A': u'u', u'C': u'y', u'B': u'y', u'u': u'u', u'w': u'y', u'v': u'v', u'y': u
'y', u'x': u'x', u'z': u'y'}
{u'A': 2, u'C': 3, u'B': 4, u'u': 1, u'w': 2, u'v': 2, u'y': 1, u'x': 0, u'z': 3
}


A
v
w
C
u
z
x
y
B
1
2
3
5
Current routes:
A -> A: [u'A', u'u', u'A']
A -> B: [u'A', u'u', u'x', u'y', u'z', u'B']
A -> C: [u'A', u'u', u'x', u'w', u'C']
B -> A: [u'B', u'z', u'y', u'x', u'u', u'A']
B -> B: [u'B', u'z', u'B']
B -> C: [u'B', u'z', u'w', u'C']
C -> A: [u'C', u'w', u'x', u'u', u'A']
C -> B: [u'C', u'w', u'z', u'B']
C -> C: [u'C', u'w', u'C']
SUCCESS: All routes are correct!
Click on routers to print debug string below:
{u'A': u'u', u'C': u'w', u'B': u'y', u'u': u'u', u'w': u'w', u'v': u'v', u'y': u
'y', u'x': u'x', u'z': u'y'}
{u'A': 2, u'C': 4, u'B': 4, u'u': 1, u'w': 3, u'v': 2, u'y': 1, u'x': 0, u'z': 3
}



【**选做**】在阅读 DVrouter.py 代码后，我给 DV 算法的更新函数做了注释，如下：

```
def updateNode(self, content):
    """update node with routing packet"""
    data = loads(content)### 加载DV数据包信息
    src = data["src"]### 发送该DV数据包的当前路由器的紧邻路由器地址src
    dst = data["dst"]### 上述紧邻路由器src所更新的到达路由器地址dst
    cost = data["cost"]### 由紧邻路由器src达到dst的最小开销

    if dst not in self.routersCost and dst != self.addr:### 若该dst此前并没有最小开销记录且dst不是当前路由器
        if src in self.routersCost:### 存在当前路由器到紧邻路由器src的最小开销记录，则进行DV更新
            self.routersCost[dst] = self.routersCost[src] + cost
            self.routersNext[dst] = src
            return True, dst, self.routersCost[dst]### 返回更新后的开销信息

    if dst in self.routersCost:### 若此前已经有当前路由器到dst的最小开销记录
        if src in self.routersCost:### 存在当前路由器到紧邻路由器src的最小开销记录

            ### 若经由紧邻路由器src到达dst路由器的开销更小，则进行DV更新，更新当前路由器到达dst的开销，同时将到达dst的途径紧邻路由器改为src
            ### 此外，如果本来到达dst的途径紧邻路由器就是src（且src不是当前路由器），则无条件更新开销，或许是为了更新链路断开信息
            if (self.routersCost[dst] > self.linksCost[src] + cost) or (self.routersNext[dst] == src and src != dst):
                self.routersCost[dst] = self.linksCost[src] + cost
                self.routersNext[dst] = src

                if self.routersCost[dst] >  COST_MAX:### 判断上面更新是否对应于链路断开情况
                    self.routersCost[dst] =  COST_MAX
                return True, dst, self.routersCost[dst]### 返回更新后的开销信息
    return None### 没有发生更新
```

根据上述代码，我理解的 DV 实现过程如下：

- 当某一 x 路由器收到来自周围紧邻的路由器发来的 DV 信息数据包时，则根据其进行自身 DV 的更新，更新时采用式子 $D_x(y)_{new} = \min[c(x,v) + D_v(y), D_x(y)_{old}]$，其中$D_v(y)$为紧邻 v 路由器距离 y 路由器最小的开销值（是包含在 DV 数据包中的信息），而 $c(x,v)$ 为当前 x 路由器到该紧邻的 v 的直接链路开销（记录在当前 x 路由器中），$D_x(y)_{old}$ 为此前记录的该 x 路由器到 y 节点的最小开销（记录在当前 x 路由器中），$D_x(y)_{new}$ 为更新后的该 x 路由器到 y 节点的最小开销（如果发生链路断开现象，则受到影响的 $D_x(y)_{new}$ 会无条件更新为某一对应于链路断开状态的极大值）；
- 每当某个路由器的 DV （距离向量）状态发生更新，则其将该更新转发给周围的紧邻路由器（路由器也会周期性地向紧邻路由器发送自身 DV 信息），该 DV 信息数据包中包含的是发送该数据包的路由器地址、其更新的到达节点的地址、更新后到达该节点的最小开销。

通过这一过程，在紧邻路由器之间不断的 DV 信息数据包交流中，最后各个路由器的转发表会收敛到某一结果，本质上实现了 $D_x(y) = \min_v[c(x,v) + D_v(y)]$ 这一式子（ v 为 x 的各个紧邻节点）。

5.5 路由选择算法效率比较

在不同的网络拓扑结构上执行不同路由选择算法，记录从开始运行到第一次出现路由正确的收敛时间，得到下表：

收敛时间(s)	链路正常	链路故障	链路新增
距离向量法	25.19	45.36	45.32
链路状态法	45.33	55.42	55.32

实验思考题

(1) 请给出你对 LSP.py 中 LSP 更新函数的执行过程的理解。

根据我的理解，我对 LSP 更新函数进行了注释，如下：

```
def updateLSP(self, packetIn):
    if self.seqnum >= packetIn["seqnum"]:### 若当下存储的LSP信息的序列号更大，则刚传入的LSP信息是旧的，无需更新
        return False
    self.seqnum = packetIn["seqnum"]### 若刚传入的LSP信息的序列号更大，则该传入信息是新的，更新存储的序列号
    if self.nbcost == packetIn["nbcost"]:### 若刚传入的LSP信息和当前存储的LSP信息一致，无需更新存储的LSP信息
        return False
    if self.nbcost != packetIn["nbcost"]:### 若刚传入的LSP信息和当前存储的LSP信息有所不同，则需要更新存储的LSP信息
        self.nbcost = packetIn["nbcost"]### 更新存储的LSP信息
        return True
```

从上面注释可以看出，LSP 更新函数的思路十分简洁，先根据序列号判断传入的数据是否是新的，当数据包是新的，则更新序列号，再根据传入数据包和当前存储数据包是否一致，若不一致则更新。

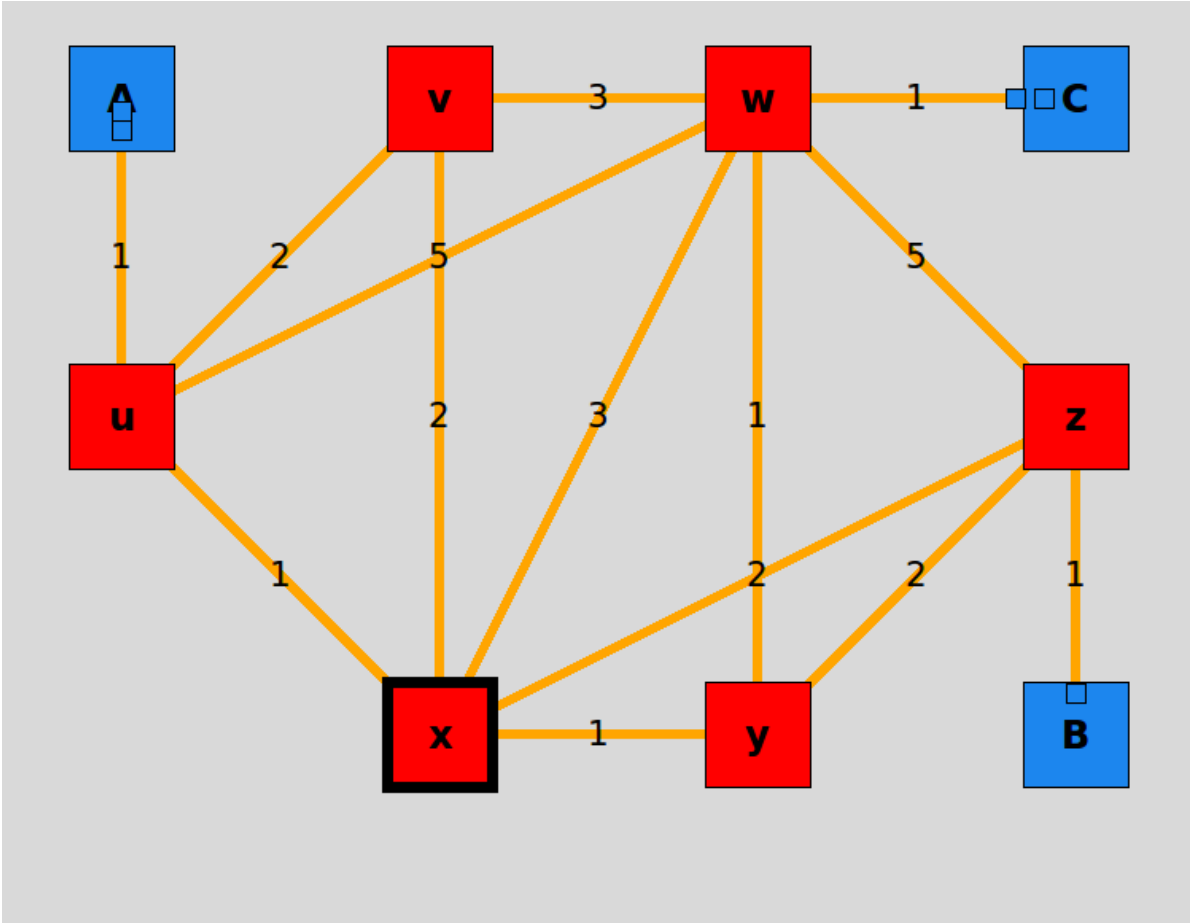
(2) 在距离向量法实现中，下面代码基于 Bellman-Ford 方程进行距离向量更新时，其中的 self.linksCost[src] 能否换成 self.routersCost[src]，请说明原因。

```
# # choose the less cost or new info
if dst in self.routersCost:
    if src in self.routersCost:
        if (self.routersCost[dst] > self.linksCost[src] + cost) or (self.routersNext[dst] == src and src != dst):
            self.routersCost[dst] = self.linksCost[src] + cost
            self.routersNext[dst] = src

            # set COST_MAX as infinity
            if self.routersCost[dst] >  COST_MAX:
                self.routersCost[dst] =  COST_MAX
            return True, dst, self.routersCost[dst]
```

self.linksCost[src] 不能换成 self.routersCost[src]，因为 self.linksCost[src] 中存储的是当前路由器直接到其紧邻路由器 src 的那条唯一链路的开销，而 self.routersCost[src] 中存储的是当前路由器到路由器 src 的一系列链路的总开销（更新后，就是到 src 的最小开销），而在 DV 算法的核心式子 $D_x(y) = \min_v[c(x,v) + D_v(y)]$ 中，$c(x,v)$ 指的是当前路由器 x 到其紧邻路由器 v 的那条直接链路的开销，这和 self.linksCost 的含义是对应的。而且常常出现 self.linksCost[src] 和 self.routersCost[src] 是不等的情况（也就是通往 src 的最小开销途径不是直接由当前路由器指向 src 的那条链路），此时将 self.linksCost[src] 替换成 self.routersCost[src] 后就会导致更新的转发表（即通往 dst 需要途径的紧邻路由器）有误。

举个例子，比如下图：



从路由器 x 的视角来看，（用 w 指代路由器 w 的地址）self.linksCost[w] 的值为 3，而 self.routersCost[w] 的值为 2，那么如果用 self.routersCost[src] 替换了 self.linksCost[src]，则在更新路径时，上述算法就有可能认为从 x 到 C 的最小开销链路为 x→w→C（也就是当 x 收到目的地为 C 的数据包时，x 会把该数据包转发到 w），但是实际上从 x 到 C 的最小开销路径为 x→y→w→C（当 x 收到目的地为 C 的数据包时，x 应该把该数据包转发到 y），这就是不适当的替换带来的错误。