**计算机网络编程**

**实验报告**

**班级：07111707**

**组长：1120171189 崔程远**

**成员：1120172149 吴沁璇**

**1120172153 张澈**

**1120172163 王晓媛**

**1120172736 张鉴昊**

**1120172765 曾煜瑾**

**1120173326 曾紫飞**

**北京理工大学**

**计算机学院**

**2020年4月**

**第五章 实验2 链路状态路由（LS）算法**

**1. 实验目的**

编写一个基于链路状态的路由算法，生成路由表。

配置文件中存放一个二维矩阵，表示网络拓扑中各个路由结点与其它结点之间的邻接关系。若两个结点直接相邻，则对应矩阵元素的值为两个结点之间的距离（正整数）。若两个结点之间不直接相邻，则对应矩阵元素的值为空（或99，表示无穷远）。

**2. 实验内容**

运行程序，输出初始网络拓扑（配置文件中）对应的各个路由器的链路状态，之后根据最短路径优先SPF算法计算任意路由器之间的最短路径，输出每个路由器每步的最短通路结果，再输出最终态各个路由器的路由表。

**3. 实验原理**

一个主机通常与一台路由器相连接，该路由器即为主机的默认路由器。源主机的默认路由器称作源路由器，目的主机的默认路由器称作目的路由器。一个分组从源主机到目的主机的路由选择问题即从源路由器到目的路由器的路由选择问题。

路由选择算法可分为：全局式路由选择算法和分散式路由选择算法；

全局式路由选择算法：所有路由器掌握完整的网络拓扑和链路费用信息，例如链路状态（ls）路由算法

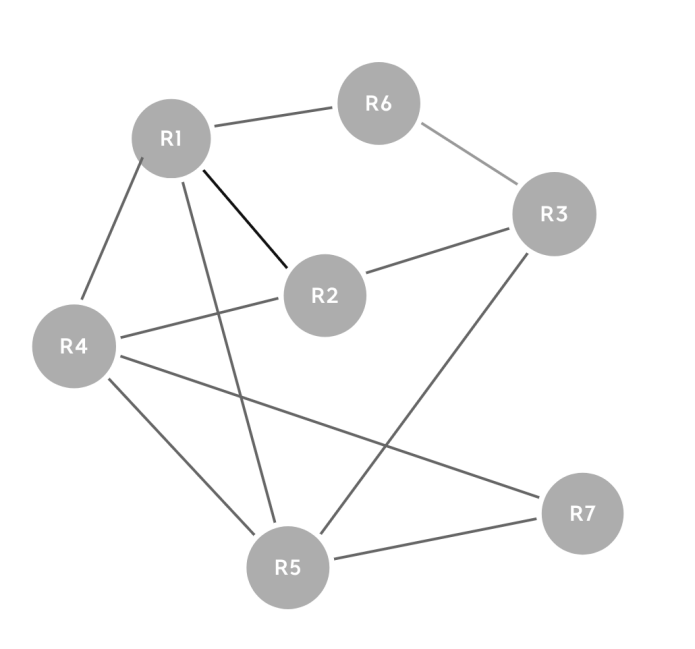
分散式路由选择算法：路由器只掌握物理相连的邻居以及链路费用，例如距离向量（dv）路由算法

下面主要介绍链路状态路由算法（Link-Stage）中的一种典型算法OSPF算法。

1. OSPF

Open Shortest Path First, 开放最短路径优先协议，是一种开源的使用最短路径优先（SPF）算法的内部网关协议（IGP）。常用于路由器的动态选路。

1. OSPF常见概念
   1. 邻居：宣告OSPF的路由器从所有启动OSPF协议的接口上发出hello数据包。如果两台路由器位于同一条数据链路上，并且它们根据互相的hello消息中指定的某些信息（比如id等）协商成功，那么它们就成为了邻居（Neighbor）。
   2. 邻接关系（Adjacency）：两台邻居路由器之间构成的一条点到点的虚链路，邻接关系的建立是由交换Hello信息的路由器类型和网络类型决定的。
   3. 链路状态通告（Link State Advertisement，LSA）：每一台路由器都会在所有形成邻接关系的邻居之间发送链路状态通告LSA。LSA描述了路由器所有的链路、接口、邻居等信息。OSPF定义了许多不同的LSA类型。
   4. 链路状态数据库（LSDB）：每一台收到来自邻居路由器发出的LSA的路由器都会把这些LSA信息记录在它的LSDB中，并且发送一份LSA的拷贝给该路由器的其他所有邻居。这样当LSA传播到整个区域后，区域内所有的路由器都会形成同样的LSDB。
2. OSPF基本原理
   1. OSPF算法是让每个路由器中的数据库储存整个网络的拓扑图，即每个路由器掌握了全局的信息，此时这个网络趋于稳定，便可以使用单源最短路径（Dijkstra）来选择路由。
   2. 每个路由器与其邻居的通信行为有以下几种：
      1. 保持联系：整个自治系统中，每个路由器都有唯一标识routerid(32-bit). 与其每个邻居间隔30s，发送一次hello 报文，确认对方是否活跃；二者相互通信发送hello，并收到对方回应hello。双方会周期性将自己的路由数据摘要发送给对方，一般30min
      2. 告知现今情况



如图，R1会周期性地将自己的路由摘要发送给所有邻居。比如对R6路由器，R1会发送称为DD报文的包，里面会说自己认识R6/R2/R4/R5, R6对比自己的信息库发现没有除自己之外的其他路由信息，于是就发送请求报文，请求告知这些陌生路由详情。这个请求报文称为LSR(链路状态请求报文)报文。

R1收到之后，发送LSU报文（链路状态更新）告知R6详情。R6收到之后，给R1个确认-LSACK报文。此时这两个路由器的信息库一致，即全毗邻关系。

**4. 实验环境**

1. C：gcc version 8.3.0

2. Java: openjdk version "10.0.2" 2018-07-17

OpenJDK Runtime Environment 18.3 (build 10.0.2+13)

OpenJDK 64-Bit Server VM 18.3 (build 10.0.2+13, mixed mode)

3. python: Python 2.7.16 (default, Feb 29 2020, 01:55:37)

1. **实验步骤**

以C语言代码为例说明具体思路：

1. 首先初始化配置信息，以实验要求中给出的示例为例：

//配置文件内容示例，共计5个结点，99代表不可达

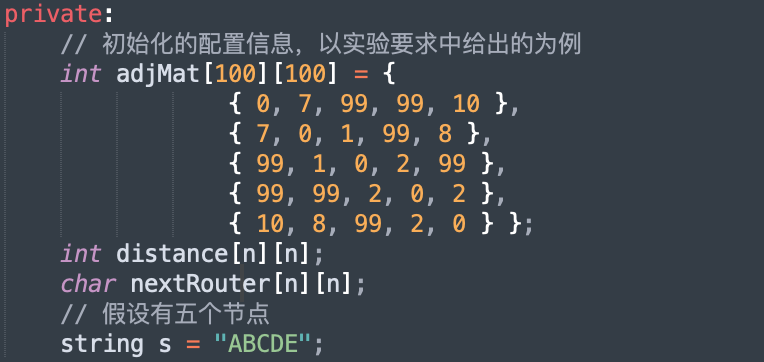
0 7 99 99 10

7 0 1 99 8

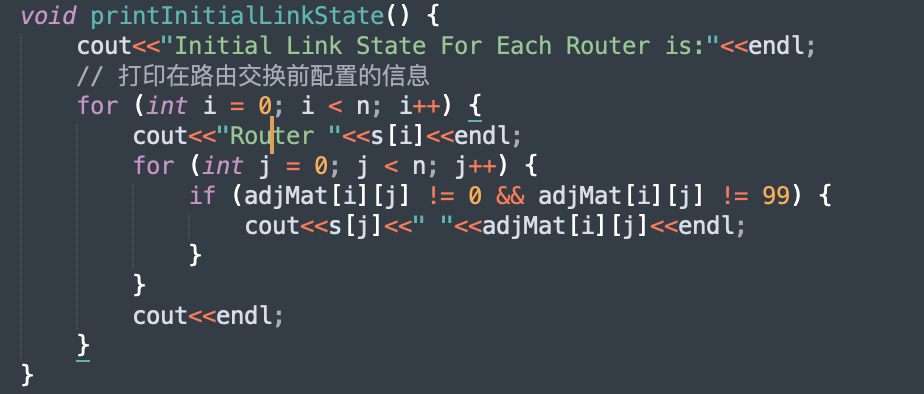
99 1 0 2 99

99 99 2 0 2

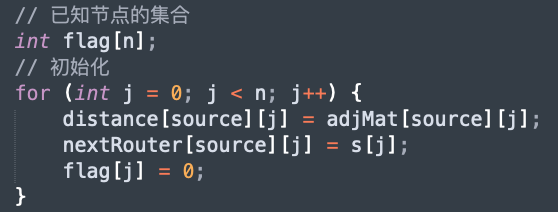
10 8 99 2 0



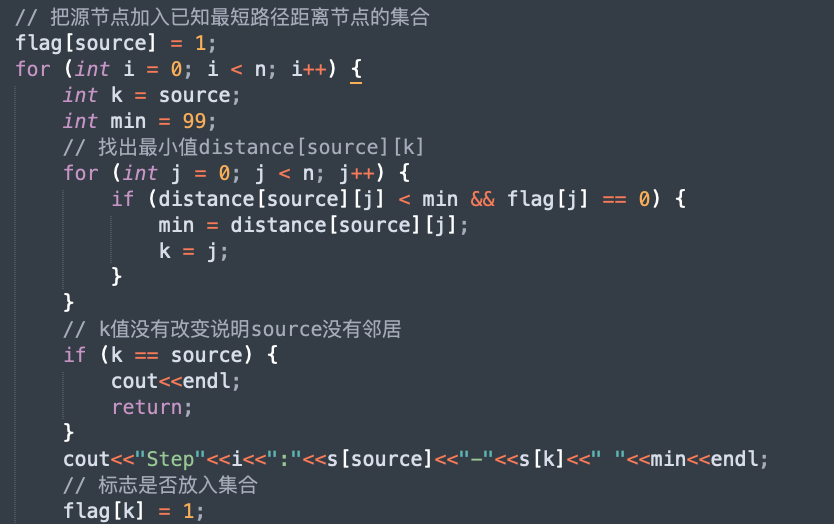
1. 屏幕上输出初始化的路由链接信息



1. 使用Dijkstra算法，计算以source为源点到个点的最小路径以及花费
   1. 设已知节点集合
   2. 初始化已知节点、距离矢量、下一个路由器的集合



* 1. 找出最小值

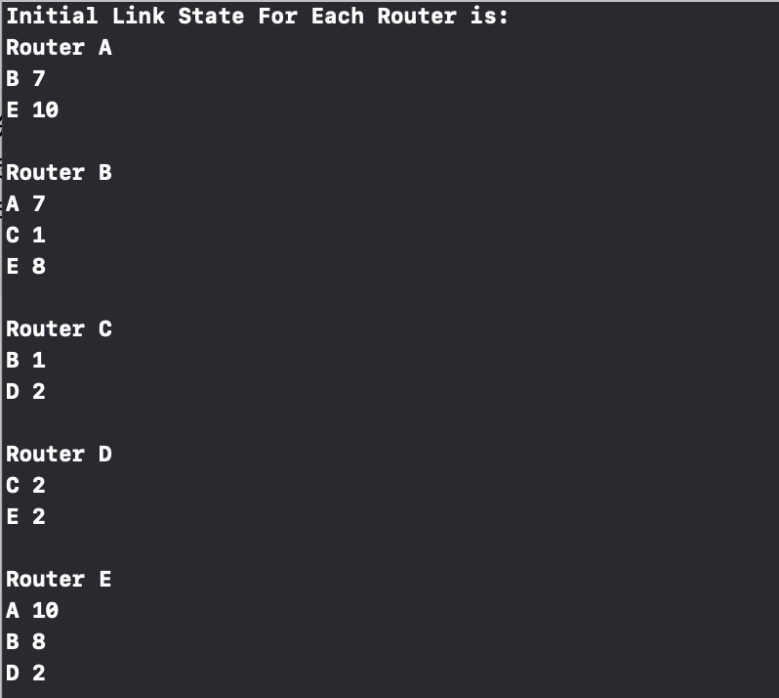


* 1. 松弛邻边

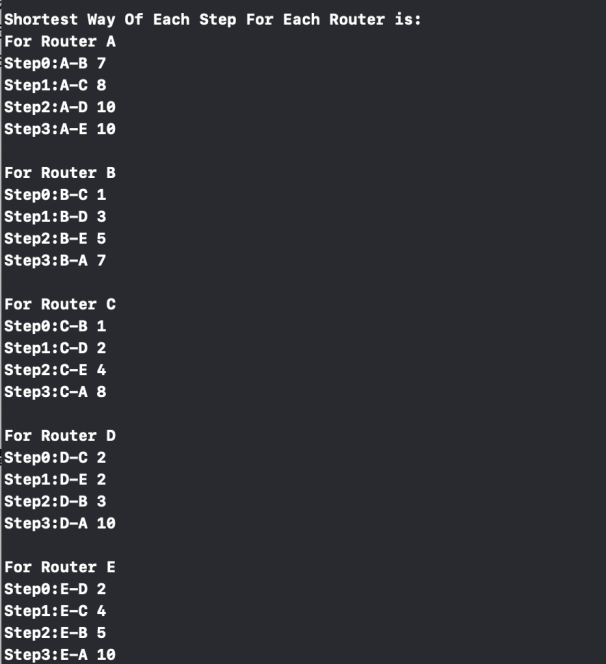


运行结果截图：

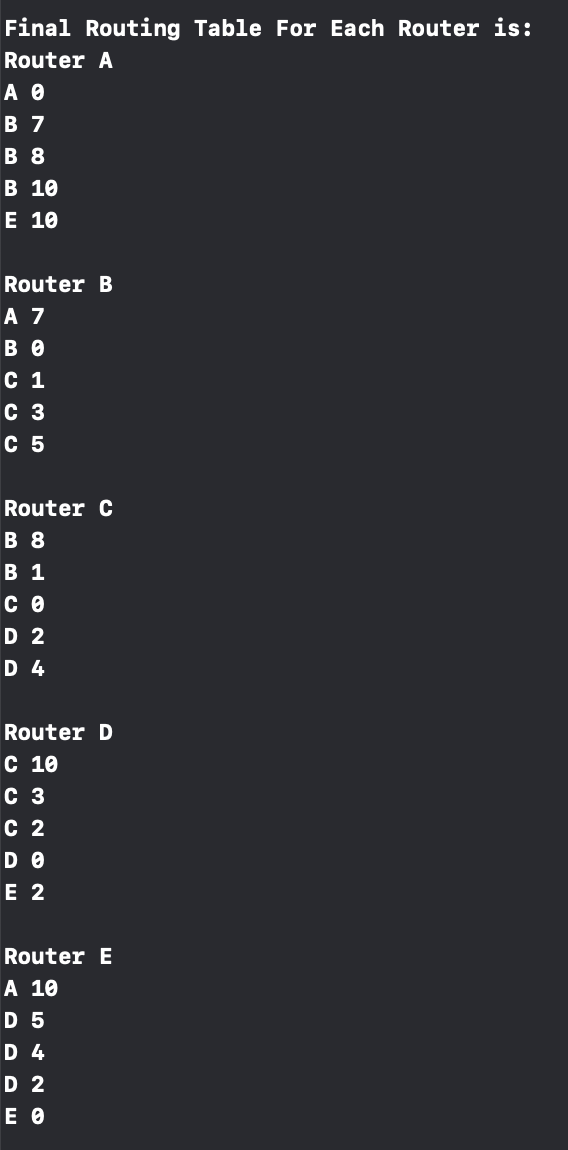
初始化后各路由器信息：

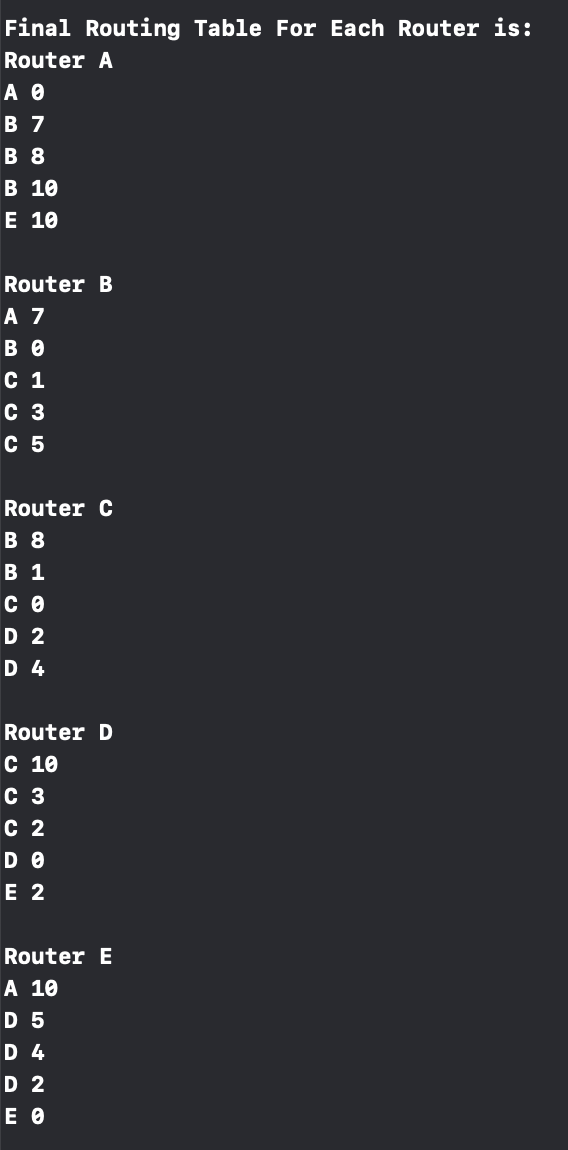


每个路由器之间的最短邻边



最终的路由表：





Java与Python编程思路与C语言基本相似，以下仅展示代码。代码均已上传[GitHub仓库](https://github.com/Skybluewater/ComputerNetwork/blob/master/4/LS_Algorimth)

**Python**：

class LSR:

n = 5

adjMat = [[0, 7, 99, 99, 10],

[7, 0, 1, 99, 8],

[99, 1, 0, 2, 99],

[99, 99, 2, 0, 2],

[10, 8, 99, 2, 0]]

distance = [[0] \* 5 for i in range(5)]

nextRouter = [[' '] \* 5 for i in range(5)]

s = "ABCDE"

def printInitialLinkState(self):

print("Initial Link State For Each Router is:")

for i in range(self.n):

print("Router " + self.s[i])

for j in range(self.n):

if (self.adjMat[i][j] != 0 and self.adjMat[i][j] != 99):

print("{} {}".format(self.s[j], self.adjMat[i][j]))

print()

def SPF(self, source):

print("For Router " + self.s[source])

flag = [0 for i in range(self.n)]

for j in range(self.n):

self.distance[source][j] = self.adjMat[source][j]

self.nextRouter[source][j] = self.s[j]

flag[source] = 1

for i in range(self.n):

k = source

min = 99

for j in range(self.n):

if (self.distance[source][j] < min and flag[j] == 0):

min = self.distance[source][j]

k = j

if k == source:

print()

return

print("Step {}: {}-{} {}".format(i, self.s[source], self.s[k], min))

flag[k] = 1

for j in range(self.n):

if (self.distance[source][k] + self.adjMat[k][j] < self.distance[source][j] and flag[j] == 0):

self.distance[source][j] = self.distance[source][k] + self.adjMat[k][j]

self.nextRouter[source][j] = self.nextRouter[source][k]

def printFinalRoutingTable(self):

print("Final Routing Table For Each Router is:")

for i in range(self.n):

print("Router " + self.s[i])

for j in range(self.n):

print("{} {}".format(self.nextRouter[i][j], self.distance[i][j]))

print()

if \_\_name\_\_ == '\_\_main\_\_':

operation = LSR()

operation.printInitialLinkState()

print("Shortest Way Of Each Step For Each Router is:")

for i in range(operation.n):

operation.SPF(i)

operation.printFinalRoutingTable()

**Java**

public class LSR {

public static int n = 5;

private int adjMat[][] = new int[][] {

{ 0, 7, 99, 99, 10 },

{ 7, 0, 1, 99, 8 },

{ 99, 1, 0, 2, 99 },

{ 99, 99, 2, 0, 2 },

{ 10, 8, 99, 2, 0 } };

private int distance[][] = new int[n][n];

private char nextRouter[][] = new char[n][n];

private String s = "ABCDE";

public void printInitialLinkState() {

System.out.println("Initial Link State For Each Router is:");

for (int i = 0; i < n; i++) {

System.out.println("Router " + s.charAt(i));

for (int j = 0; j < n; j++) {

if (adjMat[i][j] != 0 && adjMat[i][j] != 99) {

System.out.println(s.charAt(j) + " " + adjMat[i][j]);

}

}

System.out.println();

}

}

void SPF(int source) {

System.out.println("For Router " + s.charAt(source));

int flag[] = new int[n];

for (int j = 0; j < n; j++) {

distance[source][j] = adjMat[source][j];

nextRouter[source][j] = s.charAt(j);

flag[j] = 0;

}

flag[source] = 1;

for (int i = 0; i < n; i++) {

int k = source;

int min = 99;

for (int j = 0; j < n; j++) {

if (distance[source][j] < min && flag[j] == 0) {

min = distance[source][j];

k = j;

}

}

if (k == source) {

System.out.println();

return;

}

System.out.printf("Step %d: %c-%c %d\n", i, s.charAt(source), s.charAt(k), min);

flag[k] = 1;

for (int j = 0; j < n; j++) {

if (distance[source][k] + adjMat[k][j] < distance[source][j] && flag[j] == 0) {

distance[source][j] = distance[source][k] + adjMat[k][j];

nextRouter[source][j] = nextRouter[source][k];

}

}

}

}

void printFinalRoutingTable() {

System.out.println("Final Routing Table For Each Router is:");

for (int i = 0; i < n; i++) {

System.out.println("Router " + s.charAt(i));

for (int j = 0; j < n; j++) {

System.out.println(nextRouter[i][j] + " " + distance[i][j]);

}

System.out.println();

}

}

public static void main(String[] args) {

LSR operation = new LSR();

operation.printInitialLinkState();

System.out.println("Shortest Way Of Each Step For Each Router is:");

for (int i = 0; i < n; i++) {

operation.SPF(i);

}

operation.printFinalRoutingTable();

}

}

**6. 实验总结**

本次实验中，我们熟悉了几种路由选择算法：全局式路由选择算法(如链路状态（LS）路由算法)、分散式路由选择算法(例如距离向量（DV）路由算法)。了解了LS算法中的典型算法OSPF(OPEN SHORTEST PATH FIRST, 开放最短路径优先协议)的基本原理、相关概念，并使用典型LS路由算法Dijkastra算法完成了输出路由表的实验要求。