距离矢量(Distance Vecotr)算法

第五章实验 3 报告

赖泽强组

1 实验目的

- 熟悉距离矢量算法
- 理论实践相结合，提高实践能力

2 实验内容

使用三种语言，Java，Python，C++实现路由算法中的距离矢量算法，具体要求如下：

- 输入为子网的拓扑结构，采用临界矩阵表示，包括节点个数，相邻节点边个数，节点间的距离（延迟）。
- 输出为最终经过距离矢量算法计算后的路由表，可以输出每个路由器的表，也可以将全部路由器整合到一个表中，要求路由表的内容包括：到目的节点的距离，下一跳选择的路由器。

3 实验原理

距离矢量算法（DV）算法是动态路由算法中的一种。在数据层中，路由算法根据距离来决定数据包的最佳路径。

每个路由器（节点）维护一个如表 1 所示的路由表，在通过节点时即可根据目的地址来选择下一个节点，从而选择出最佳路径

Destination	Cost	Next hop

表 1.路由表

路由表通过相邻路由器传递距离矢量来更新维护，距离矢量代表到子网其他节点的距离，即每个节点需要传递的距离矢量等于路由表中 cost 当前列。

例如：节点 A 的距离矢量 $v_a =$（$d_1, d_2, \ldots, d_n$）代表V_a到子网中其他节点的距离分别为$d_1, d_2, \ldots, d_n$。每个节点根据收到的距离矢量来更新当前路由表，更新的公式如下：

$$d_x(y) = \min\{d_x(y)', c(x,v) + d_v(y), v \epsilon V\}$$

各个变量含义如下：

x:需要更新路由表的节点 y: x节点的目的节点 v:x节点的相邻节点 V:x的直接相邻路由器集	$d_x(y)'$:从x到y更新前的最短距离 $d_x(y)$:从x到y更新后的最短距离 c(x,v):x到相邻节点v的花费

若$d_x(y)'$与$d_x(y)$相等，则不需要更新路由表当前行，否则将路由表 1 当前行中 cost 项设置为$c(x,v) + d_v(y)$，next hop 设置为相邻节点v。

4 实验环境

4.1 开发环境

语言	环境
C++	macOS 10.14.4 Apple LLVM version 10.0.1 (clang-1001.0.46.3)
Java	Windows 10 1903 IntelliJ IDEA 2018.3.3
Python	Windows 10 1903 PyCharm 2018.3.3

4.2 部署环境

可在 Windows，Linux，macOS 三平台正确编译运行。

5 实验步骤

我们使用了三种语言模拟实现 distance vector 路由算法

5.1 C++/Java 部分

C++，Java 部分实验部分分为初始化，路由表更新，结果输出三个步骤。

5.1.1 初始化

- 数据定义

数据类型	说明
数组 A[ ][ ]	邻接矩阵表示路由器拓扑结构
数组 table_d[ ][ ][ ]	路由表的 cost 部分，table_d[x][y][z]代表第 z 次迭代后，节点 x 到 y 的距离
数组 table_line[ ][ ][ ]	路由表的 next hop 一部分，table_line[x][y][z]代表第 z 次迭代后，节点 x 终点为 y 时一跳的路径

- 代码实现

初始化部分从控制台中读取邻接矩阵，初始化 A 矩阵，通过初始化后的矩阵 A，初始化 table_d[0] table_line[0]，代码如下：

```
1  int t1, t2, w;
2      for(int i=0; i<m; ++i) {
3          cin >> t1 >> t2 >> w;
4          A[t1][t2] = w;
5          A[t2][t1] = w;
6      }
```

代码 1 Initialize adjacent matrix

```
for(int i=1; i<=n; ++i)
        for(int j=1; j<=n; ++j) {
            if(i == j) {
                table_d[0][i][j] = 0;
                table_line[0][i][j] = SELF;
            }
            else if(A[i][j] > 0) {
                table_d[0][i][j] = A[i][j];
                table_line[0][i][j] = j;
            }
            else {
                table_d[0][i][j] = INFINITY;
                table_line[0][i][j] = UNKNOWN;
            }
        }
```

代码 2 Initialize table_line, table_d

5.1.2 路由表更新

使用了自定义的 update 函数，使用遍历需要更新的路由器 x 的全部邻接路由器， 通过公式 2 来判断是否需要更新路由表 table_line 和 table_d。

$$d_x(y) = \min\{d_x(y)', c(x,v) + d_v(y), v \epsilon V\} \quad \text{公式 2}$$

在遍历完更新完全部路由器后，调用函数 converge 判断路由表 table_line[step] 和 table_line[step-1]; table_d[step] 和 table_d[step-1]中是否有数据项变化，如果没有这说明迭代已经稳定，否则继续调用 update 更新路由表。

代码如下：

```
do{
        step += 1;
        table_line[step] = table_line[step-1];
        table_d[step] = table_d[step-1];

        for(int i=1; i<=n; ++i) {
            // every adjacent point can update "my" routing table.
            for(int j=1; j<=n; ++j)
                if(A[i][j] > 0)
                    update(table_line, table_d, i, j, A[i][j], n, step);
        }
    }while((!converge(table_line, step) && step <= MAX_ITERATION));
```

代码 3 Calculate table_line, table_d

```
void update(Table& t_line, Table& t_d, int router, int line, int delay,
int n, int step)
{
    for(int i=1; i<=n; ++i) {
        if(t_d[step][router][i] > t_d[step-1][line][i] + delay) {
```

```
6                 t_d[step][router][i] = t_d[step-1][line][i] + delay;
7                 t_line[step][router][i] = line;
8             }
9         }
10    }
```

代码 4 Function of updating rounter's rount table

5.1.3 结果输出

代码输出包括了迭代的次数 step，迭代最终的距离矩阵 table_d[step]和线路矩阵 table_line[step]，并配以相关文字说明，需要注意的是 table_line 中值为零代表不存在下一跳，table_d 中距离为 9999 代表两个路由器间不存在线路可达。

5.2Python 部分

5.2.1 代码流程

在程序 myDVR.py 的 main 函数中，首先调用 add_table(tables)函数，读入配置文件 DVRinit.ini，初始化各路由表。之后调用 init_dist(tables)，使用元组、字典等数据结构记录下每个路由器的邻居信息以及到邻居路由的花费。以上为初始化过程。

考虑到各路由表的更新是独立且可能同时进行的，所以需要使用多线程来模拟每次更新的过程，但各路由表的独立更新又带来了数据不一致的问题即每个路由器获取的邻居路由表的信息可能是本次更新过程中刚刚更新过的也可能是上次更新过程后到现在还未更新的。

综合以上因素，在完成初始化过程后，调用 thread_update(tables,new_tables)，即多线程更新时每次更新过程各路由表都使用同一个路由表集合 tables，更新完成后生成一个新的路由表集合 new_tables，通过这样的规定确保了数据的一致性。thread_update()函数中为各路由表创建了一个线程运行 update 函数，该函数是本程序的主功能函数，在下文会详细介绍它。

5.2.2 重要数据结构

下面介绍几个程序中用到的重要数据结构。

- 路由表与路由表集合

  路由表：table=[name，[tar，cost，next_hop]*n]

  路由表集合：tables=[[name，[tar，cost，next_hop]*n]*m]

  路由表由名字 name，和 n 个表项组成，表项表示为[目标，花费，下一跳]，路由表集合由 m 个路由表构成。

  这里需要作出一点反思，使用列表嵌套的方式表示路由表及其集合的信息虽然在层次结构上比较鲜明，但由于多重列表的嵌套，给编程带来了一定的麻烦。

- 邻居信息

```
my_neibors={A：[B，C],#A 的邻居有 B 和 C
            B：[A，D]
            …}

neibor_cost={(A，B)：1,#A 到 B 花费为 1
           (B，C)：2
```

…}

即结合使用元组、列表、字典等数据结构，记录下每个路由器的邻居信息，包括路由器到邻居的花费，这些信息在后面的更新过程中是十分必要的。

5.2.3 Update 函数

update 函数通过传入一个路由表 table 和当前的表集合 tables，完成 table 的更新。函数的伪码表示如图 1 所示。

```
for neibor in neibors:#对每个邻居路由器
    for neitem in neibor:#对邻居路由表的每一项
      existflag=False#target 是否已存在于本路由表
      for myitem in mytable:#对本表的每一项
         if neitem.target==myitem.target:#若目的相同
            existflag=True
               #判断花费大小，是否要更新 myitem
            ...
            break
      if flag==False and neitem.target!=myname:
         #target 不存在,并且目的路由不是自身
          #将该 neitem 重设 next_hop、cost，加入 mytable
        ...
```

图 1. update 函数伪代码

遍历邻居路由器，对每个邻居路由表的每一项再遍历本路由表。如果邻居项与本表的某一项目标相同，则说明邻居项的目标已存在本表中，则要比较花费大小，判断是否要更新表项。如果邻居项目标不存在于本表，且该目标不是自身，则将邻居项修改 next_hop 和 cost 后加入本表。可以看出算法的时间复杂度为O(n)

通过以上步骤完成了一个路由表的一次更新，等到所有路由表都更新完成后，则生成一个新的路由表集合 new_tables，本次更新过程完成。程序可以设置更新过程的次数，当然也可以通过判断路由表集合是否不再变化来决定是否终止更新。

6 实验总结

这个实验的内容主要是实现 distance vector 的路由算法，这个实验与之前的无向图的最短路径非常相似，在编码的过程中没有遇到太大的问题。通过这个实验我们对动态路由算法的动态过程有了一个更加深刻的理解，同时对 dv 算法的原理有了更加深刻的体会。

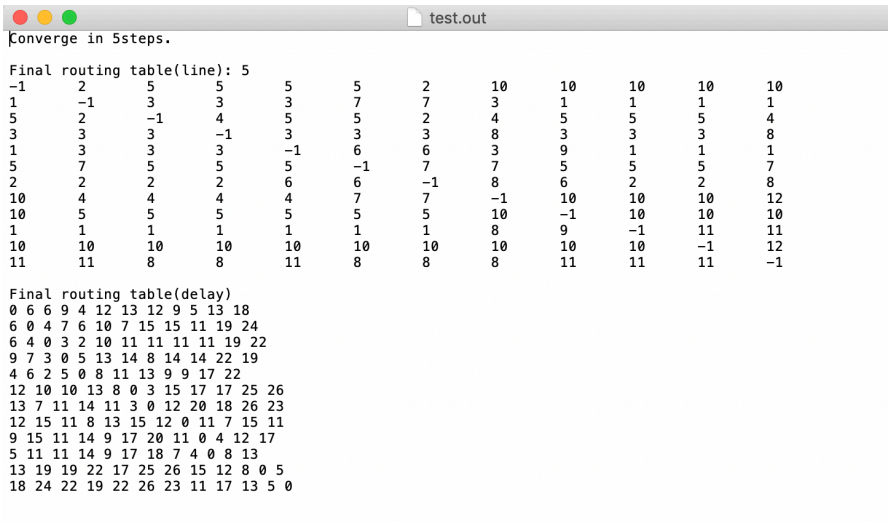
附录

A C++/Java 输入测试用例

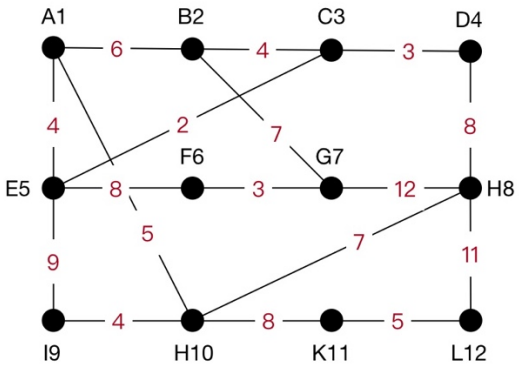
```
12
17
1 2 6
```

```
2 3 4
3 4 3
1 5 4
3 5 2
2 7 7
4 8 8
1 10 5
5 9 9
9 10 4
8 10 7
10 11 8
11 12 5
8 12 11
5 6 8
6 7 3
7 8 12
```

B C++运行结果截图



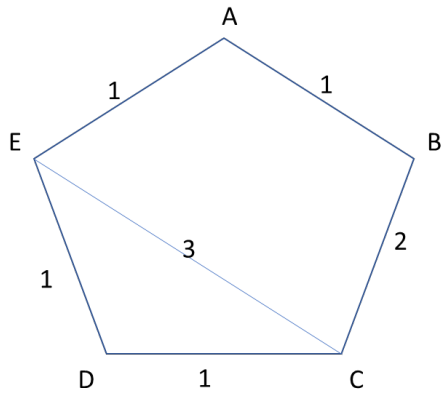
C C++/Java 输入测试用例的拓扑结构



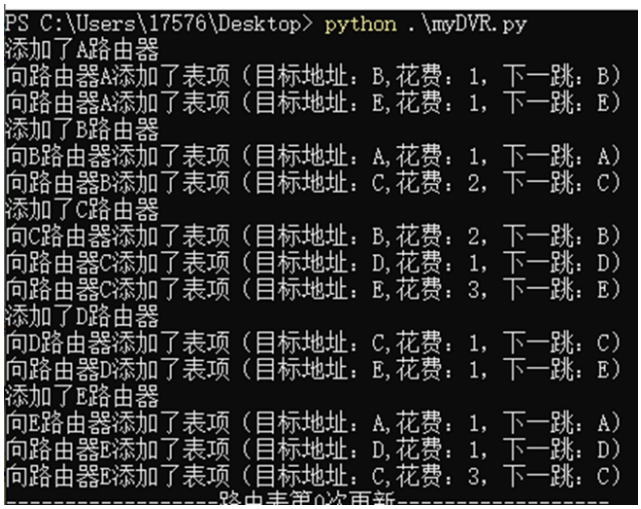
D Java 运行结果截图

```
Converge in 1steps.

Final routing table(line): 1
-1  2   5   0   5   5   2   10  10  10  10  0
1   -1  3   3   3   7   7   1   1   1   1   0
5   2   -1  4   5   5   2   4   5   2   2   0
3   3   3   -1  3   3   3   8   3   8   3   8
1   3   3   3   -1  6   6   3   9   1   1   0
5   7   5   5   5   -1  7   7   5   5   5   0
2   2   2   2   6   6   -1  8   6   2   2   8
10  4   4   4   4   7   7   -1  10  10  10  12
10  5   5   5   5   5   5   10  -1  10  10  0
1   1   1   8   1   1   1   8   9   -1  11  11
10  10  10  10  10  10  10  10  10  10  -1  12
11  11  8   8   11  8   8   8   11  11  11  -1

Final routing table(delay)
0 6 6 9999 4 12 13 12 9 5 13 9999
6 0 4 7 6 10 7 18 15 11 19 9999
6 4 0 3 2 10 11 11 11 15 23 9999
9 7 3 0 5 13 14 8 14 15 26 19
4 6 2 5 0 8 11 13 9 9 17 9999
12 10 10 13 8 0 3 15 17 17 25 9999
13 7 11 14 11 3 0 12 20 18 26 23
12 15 11 8 13 15 12 0 11 7 15 11
9 15 11 14 9 17 20 11 0 4 12 9999
5 11 11 15 9 17 18 7 4 0 8 13
13 19 19 23 17 25 26 15 12 8 0 5
18 24 22 19 22 26 23 11 17 13 5 0

Process finished with exit code 0
```

E Python 输入矩阵的拓扑结构



F Python 读入配置截图



G Python 运行结果截图