数据结构第三次项目
路由问题

陈小羽 2016060106020 (单刷)

目录

1 题目要求

一般的网络都有一个网关（外接路由器）与 Iternel 网相连，负责获得外网发送至内部单位的信息，并转发至内网。现在假设内网有一些机器，每次能告诉网关自己和周围机器的连接情况，现在需要给网关设计一个算法，使得网关在收到连接信息之后，可以找到到达每个设备的最优的路径（路由表）

2 选择算法和实现

2.1 最短路径算法

首先我们把这个问题转化为图论中的问题。为了达到这个目的，我们可以令原来连接设备的网线作为图中的边，令原来两个设备之间连接的优劣程度作为边的边权，令设备作为图中的点，并设网关为点 1。我们把这个图记做 $G = (V, E)$，其中，V 表示点集, E 表示边集。

然后，原来的问题就转化为了求图中 1 点到其他的点的最短路的问题。因为题目中出现的这种场景导致边权不可能出现复数，所以，我们可以使用 Dijkstra 算法来解决这个问题。

因为 Dijkstra 算法的过程比较复杂，不太好用语言描述清楚，所以在这篇文章中，我们选择直接给出伪代码。具体参见 **算法 1**(在这个算法实现中，我使用了优先队列来优，化算法在渐近意义下的复杂度，但是并没有给出优先队列算法的细节，具体算法，我会在后面的章节中给出)。

下面让我们来分析一下这个算法的复杂度。首先，算法会把图中的所有的边都给遍历一遍。并且，Dijkstra 算法严格保证了每一个点出队之后就再也不会再次入队了。所以总的时间复杂度为 $\mathcal{O}\left((|V| + |E|) \times T(operation)\right)$ 这个 $T(operation)$ 表示的是优先队列单次操作的复杂度。所以又有 $T(operation) = \log(N)$。最后因为每一条边都可能会导致一个点入队，所以队列中的总的结点数最多为 $|V| + |E|$ 个。所以，综合所有以上这些信息的话，我们可以得出结论：总的时间复杂度为 $\mathcal{O}\left((|V| + |E|)\log(|V| + |E|)\right)$。这个结果相当于最原始的 $\mathcal{O}(|V||E|)$ 已经相当不错了。

2.2 优先队列（小根堆）

为了对原始版本的 Dijkstra 算法进行优化，我们在之前使用到了小根堆。在这里，我们来讨论一下小根堆具体的一些细节。

小根堆内部维护了一颗完全二叉树。这个完全二叉树上的元素要保证父亲结点储存的值要小于儿子结点储存的值。然后，这个小根堆要支持插入一个元素的操作和删除/查询最小元素的操作。因为它支持的操作的特殊性，小根堆有时又被称为优先队列，即队首元素是整个队列中的最小的元素。下面我们来看一下具体的操作要怎么实现，以及他们需要花费的代价。因为堆的操作相对简单，所以我们直接采用文字叙述。

插入 q.push() 将要加入的元素放到完全二叉树的最后一个位置，然后不断的把这个元素和它的父亲结点比较，如果这个元素小于父亲结点的话，就将这个元素和父亲结点交换，否则，插入，结束。因为堆是完全二叉树，深度为 $\log N$。所以插入操作最多进行 $\log N$ 次比较和交换。所以，插入操作的复杂度为 $\mathcal{O}(\log N)$, 其中 N 为堆中元素的个数。

Algorithm 1 Dijkstra 算法 (1)

```
1: int dis[maxn]                                          ▷ 用于储存起点到这个点的距离
2: int pre[maxn]                                          ▷ 用于储存这个结点在最短路中的前驱结点
3: PriorityQueue q                                        ▷ 优先队列 (用于优化 Dijkstra 算法), 优先弹出距离小的元素
4: function DIJKSTRA(起点 s, 点的个数 n)                   ▷ 这个函数会求出 dis 和 pre 两个数组
5:     for i ← 1 : n do                                   ▷ 初始化数组
6:         pre[i] = -1
7:         dis[i] = ∞
8:     end for
9:     dis[1] = 0
10:    q ← ∅
11:    q.push(s, 0)                                       ▷ 起点入队, 距离为 0
12:    while q 非空 do
13:        Front ← q.pop()                                ▷ 这里的 Front 是一个包含 id 和 dis 两个变量的结构
14:        if Front.dis < dis[Front.id] then              ▷ 一个结点有可能重复入队，我们取距离最小的一个
15:            continue
16:        end if
17:        for 每一个和 Front.id 相连的结点 to do
18:            if dis[to] > Front.dis + edge(Front.id, to).length then   ▷ edge(i, j) 表示的是一个边
19:                dis[to] ← Front.dis + edge(Front.id, to).length
20:                q.push(to, dis[to])
21:            end if
22:        end for
23:    end while
24: end function
```

弹出最小元素 q.pop() 显然，完全二叉树的根就是最小的元素。然后，我们只需要考虑如何删除这个元素就可以了。我们可以将完全二叉树的最后一个元素拿到根结点覆盖原来的根节点。然后将这个结点往下放，把这个结点儿子中最小的，比这个结点小的结点和这个结点换，然后不停的迭代，直到这个结点没有满足要求的儿子结点。因为树的深度的关系，这个操作的复杂度，显然也是 $\mathcal{O}(\log N)$ 的。

下面给出具体的程序:

```
namespace PQ{
        template <class Type, class Cmp>
        struct PriorityQueue{
                int size, N;
                Cmp cmp;
                Type *w;
                PriorityQueue(){
                        w = new Type[10];
                        N = 10; size = 0;
                }
                void resize(int nN){
                        Type* tmp = new Type[nN];
                        for(int i = 1; i <= size; i++)
                                tmp[i] = w[i];
                        swap(w, tmp), N = nN;
                        delete[] tmp;
                }
                void push(Type x){
                        if(size+1  == N) resize(2 * N);
                        int rt; w[rt = ++size] = x;
                        while(rt > 1) {
                                if(cmp(w[rt], w[rt>>1])) swap(w[rt], w[rt>>1]), rt >>= 1;
                                else break;
                        }
                }
                void pop(){
                        int rt; w[rt = 1] = w[size--];
                        while((rt<<1) <= size){
                                int nx = rt<<1;
                                if(nx+1 <= size && cmp(w[nx+1], w[nx])) ++nx;
                                if(cmp(w[nx], w[rt])) swap(w[nx], w[rt]), rt = nx;
                                else break;
                        }
                        if(size * 4 + 1 < N) resize(N >> 1);
                }
                Type top(){
                        return w[1];
                }
                bool empty(){
                        return size == 0;
                }
                ~PriorityQueue(){
                        delete[] w;
                }
        };
};
```

我实现了动态的堆。方法是：当堆满时，将这个堆重建，重新申请大小为 $2N$ 的内存空间。当堆中元素个数小于 $\frac{N}{4}$ 时，将堆重建，重新申请大小为 $\frac{N}{2}$ 的内存空间。这种操作可以保证堆在内存中是连续的。而且这种看似暴力的操作，在均摊意义下，并不会对堆的渐近复杂度又任何影响。

2.3 生成路由表

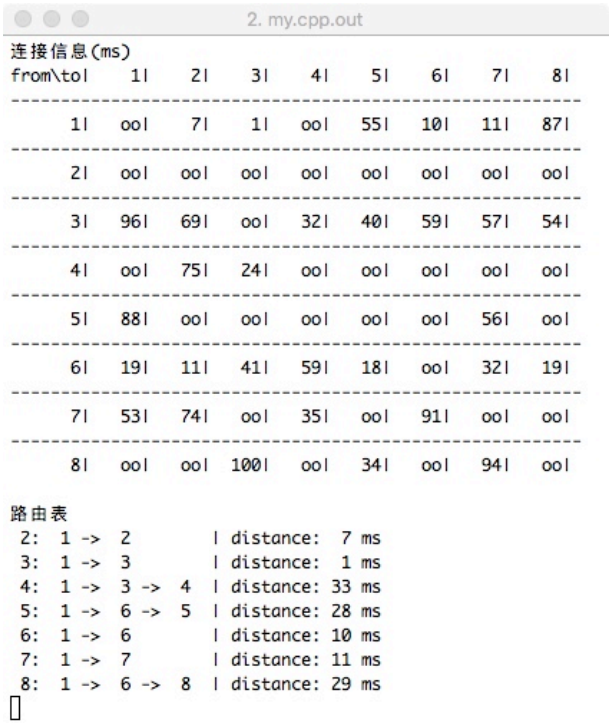
值得注意的是，上面的 Dijkstra 算法只能计算出 dis 和 pre 连个数组的值，但是这两个数组里面的值还算不上是真正的路由表。所以，我们还需要设计一个可以打印出路由表的算法。为了代码实现的简洁性，我选择了一个递归的算法来实现这个功能，具体可以看下面给出的**伪代码 2**。

Algorithm 2 打印路由表 (伪代码 2)

```
1: function printRoad(结点 x)                          ▷ 打印出以 x 结尾的路径
2:     if x ≠ 1 then                                   ▷ 到达了 1 点，递归结束
3:         printRoad(pre[x])
4:     end if
5:     输出 x
6: end function
7: function printTable                                 ▷ printRoad 的具体使用方法的示例
8:     for i ← 2 : n do
9:         printRoad(i), 换行
10:    end for
11: end function
```

2.4 小结

有了前面说的算法，就可以上机实现程序了。具体程序运行的效果如下图。



其中，连接信息是随机生成的，其原理显然不是这篇文章乃至这次实验的重点。每次敲击一下键盘，程序就会更新连接信息并且计算出相应的路由表。

下面，实现程序之后，就到了测试的环节。

3 测试

3.1 测试小根堆

因为这个程序中，小根堆是我自己手动实现的。所以在测试其他位置的问题之前，首先要测试小根堆写对没有。这种东西最好的测试方法就是将待测的代码和保证正确的代码运行在同一组输入上，然后比较输出即可。

好在 STL 中原生支持一个优先队列，这个模板类被包含在 <queue> 这个头文件中。所以，我们将原来的优先队列换成 STL 实现的优先队列就可以了 (其他一些地方也需要改，这里就不一一说明了)。

```
//PQ::PriorityQueue <pii, cmp> q;
priority_queue <pii, vector<pii>, cmp> q;
```

因为我的计算机是 Unix 环境，所以我使用 bash 和 python 实现了一个简单的脚本，可以用来测试程序的正确性。

```
#!/bin/bash
for(( i = 1; i <= 100; i = i + 1))
do
        ./make_data.py > main.in
        (time ./my.cpp.out < main.in > my.out) 2>> checkLog
        ./test.cpp.out < main.in > test.out
        diff my.out test.out > /dev/null
        if [ $? -ne 0 ]
                then
                        echo Wa >> checkLog
                        exit
                else
                        echo Ac $i >> checkLog
        fi
done
```

他的作用就是生成随机的数据喂给两个程序，然后检测这两个程序的输出是否相同。经过测试可以发现，对于这随机的 100 组数据，两个程序得到了完全相同的输出。所以，我们有理由相信，小根堆的算法和实现是正确的。

```
real    0m0.012s
user    0m0.009s
sys     0m0.002s
Ac 1

real    0m0.006s
user    0m0.003s
sys     0m0.001s
Ac 2

real    0m0.004s
user    0m0.001s
sys     0m0.001s
Ac 3
```

另外还可以说明，这个算法运行起来是十分迅速的。

3.2 测试 Dijkstra 算法

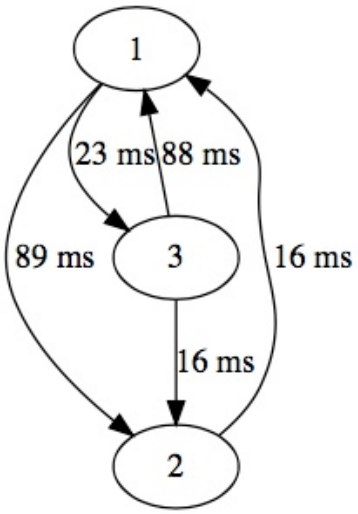
对于 Dijkstra 算法，我们就没有这么好的方式了。所以我们只有使用程序生成一些样例然后比较手动计算出的结果和程序是否是相同的。为了方便手动计算最短路，我使用了贝尔实验室开发的一个图形化软件，名字叫做:graphviz。这个软件有一个网页版的，网址是：http://www.webgraphviz.com 。我可以使用这个程序生成人类友好的图, 以此来方便我进行手动计算。下面给出了一些测试用例。

test1, n = 3

```
2. vim
digraph G{
"1" -> "2" [ label = "89 ms"]
"1" -> "3" [ label = "23 ms"]
"2" -> "1" [ label = "16 ms"]
"3" -> "1" [ label = "88 ms"]
"3" -> "2" [ label = "16 ms"]
}

路由表
 2:  1 ->  3 ->  2  | distance: 39 ms
 3:  1 ->  3        | distance: 23 ms

Press ENTER or type command to continue
```

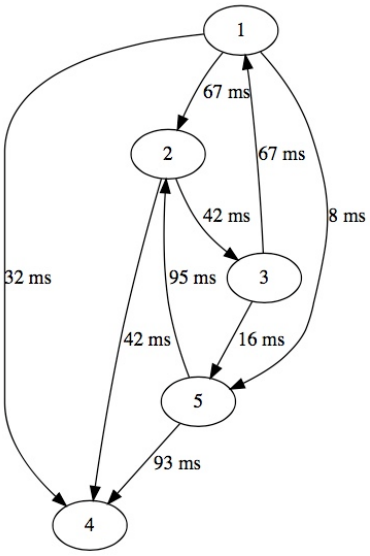


test2, n = 5

```
2. vim
digraph G{
"1" -> "2" [ label = "67 ms"]
"1" -> "4" [ label = "32 ms"]
"1" -> "5" [ label = "8 ms"]
"2" -> "3" [ label = "42 ms"]
"2" -> "4" [ label = "42 ms"]
"3" -> "1" [ label = "67 ms"]
"3" -> "5" [ label = "16 ms"]
"5" -> "2" [ label = "95 ms"]
"5" -> "4" [ label = "93 ms"]
}

路由表
 2:  1 ->  2        | distance: 67 ms
 3:  1 ->  2 ->  3  | distance: 109 ms
 4:  1 ->  4        | distance: 32 ms
 5:  1 ->  5        | distance:  8 ms

Press ENTER or type command to continue
```

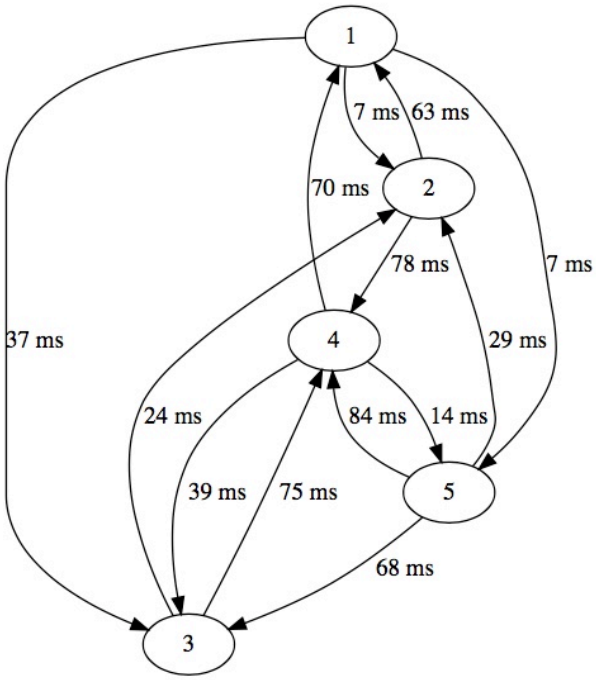


test3, n = 5

```
2. vim
digraph G{
"1" -> "2" [ label = "7 ms"]
"1" -> "3" [ label = "37 ms"]
"1" -> "5" [ label = "7 ms"]
"2" -> "1" [ label = "63 ms"]
"2" -> "4" [ label = "78 ms"]
"3" -> "2" [ label = "24 ms"]
"3" -> "4" [ label = "75 ms"]
"4" -> "1" [ label = "70 ms"]
"4" -> "3" [ label = "39 ms"]
"4" -> "5" [ label = "14 ms"]
"5" -> "2" [ label = "29 ms"]
"5" -> "3" [ label = "68 ms"]
"5" -> "4" [ label = "84 ms"]
}

路由表
 2:  1 ->  2        | distance:  7 ms
 3:  1 ->  3        | distance: 37 ms
 4:  1 ->  2 ->  4  | distance: 85 ms
 5:  1 ->  5        | distance:  7 ms

Press ENTER or type command to continue
```

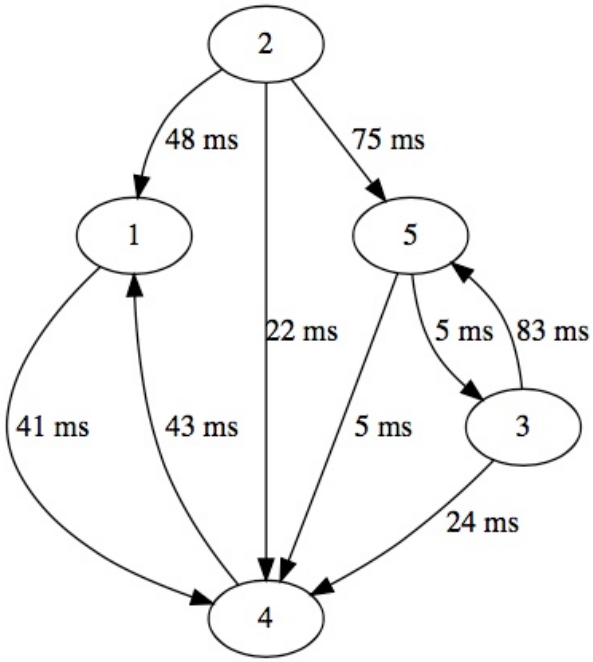


test4, n = 5

```
2. vim
digraph G{
"1" -> "4" [ label = "41 ms"]
"2" -> "1" [ label = "48 ms"]
"2" -> "4" [ label = "22 ms"]
"2" -> "5" [ label = "75 ms"]
"3" -> "4" [ label = "24 ms"]
"3" -> "5" [ label = "83 ms"]
"4" -> "1" [ label = "43 ms"]
"5" -> "3" [ label = "5 ms"]
"5" -> "4" [ label = "5 ms"]
}

路由表
 2: oo
 3: oo
 4:  1 ->  4  | distance: 41 ms
 5: oo

Press ENTER or type command to continue
```

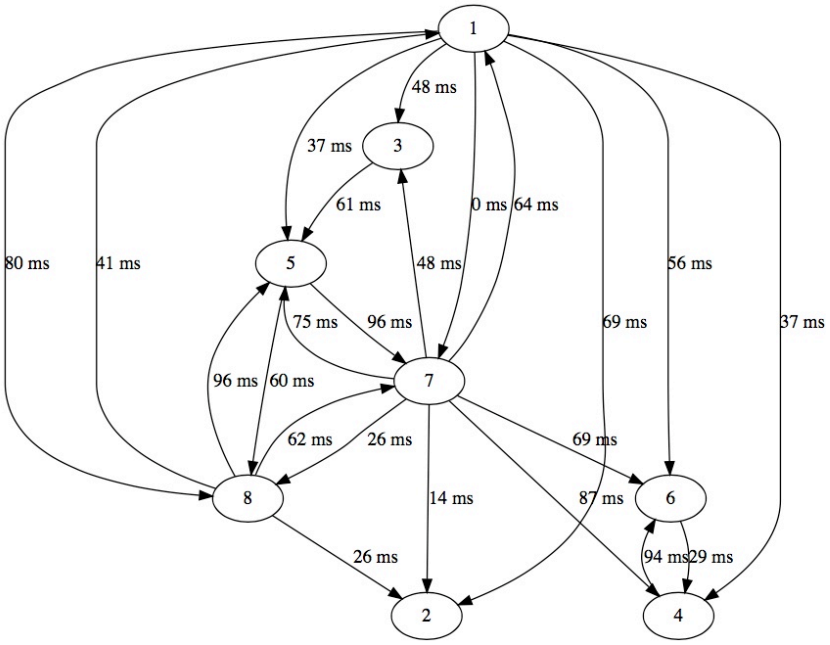


test5, n = 8

```
2. vim
digraph G{
"1" -> "2" [ label = "69 ms"]
"1" -> "3" [ label = "48 ms"]
"1" -> "4" [ label = "37 ms"]
"1" -> "5" [ label = "37 ms"]
"1" -> "6" [ label = "56 ms"]
"1" -> "7" [ label = "0 ms"]
"1" -> "8" [ label = "80 ms"]
"3" -> "5" [ label = "61 ms"]
"4" -> "6" [ label = "94 ms"]
"5" -> "7" [ label = "96 ms"]
"5" -> "8" [ label = "60 ms"]
"6" -> "4" [ label = "29 ms"]
"7" -> "1" [ label = "64 ms"]
"7" -> "2" [ label = "14 ms"]
"7" -> "3" [ label = "48 ms"]
"7" -> "4" [ label = "87 ms"]
"7" -> "5" [ label = "75 ms"]
"7" -> "6" [ label = "69 ms"]
"7" -> "8" [ label = "26 ms"]
"8" -> "1" [ label = "41 ms"]
"8" -> "2" [ label = "26 ms"]
"8" -> "5" [ label = "96 ms"]
"8" -> "7" [ label = "62 ms"]
}

路由表
 2:  1 ->  7 ->  2  | distance: 14 ms
 3:  1 ->  3        | distance: 48 ms
 4:  1 ->  4        | distance: 37 ms
 5:  1 ->  5        | distance: 37 ms
 6:  1 ->  6        | distance: 56 ms
 7:  1 ->  7        | distance:  0 ms
 8:  1 ->  7 ->  8  | distance: 26 ms

Press ENTER or type command to continue
```



3.3 小结

综合上面的各项测试数据，我们不难断定，这个程序已经不再怎么可能有什么大问题了。因为这个程序其他的位置，为了模拟网络环境的随机性，本来就是用了一些随机的算法，用于生成连接信息，所以，就算有一些小问题，也不会对最终的结果产生致命的影响。
至此，测试工作圆满结束:-)

4 感想与收获

通过这一次的数据结构作业，我又更加熟悉了小根堆的写法，还有 Dijkstra 算法的写法。通过测试，我尝试使用了 graphviz 来测试程序。通过写作文档练习了 LaTeX 的使用方法。感觉这一次的作业从多个方面训练了我自己。虽然涉及到的算法略显简单，但是我借此机会磨练了其他方面的技巧。感觉收获很多。