2022春《数据结构》大作业报告

学号：191830079

姓名：李易

院系：计算机

一、小蓝鲸的奇妙冒险-第一季

1. **解题思路**

   考虑使用堆结构。我们先实现一个大根堆，利用城市人口数组直接建堆。然后从堆中连续弹出M个元素，并用一个变量保存堆顶弹出的值，此时得到的就是最大的M个元素中的最小值。

   我们的算法建堆耗费`O(N)`，M次弹出元素耗费`O(MlogN)`，故时间复杂度为`O(N + MlogN)`。(视M与N相对大小结果不同)

   空间复杂度为建堆所耗费的`O(N)`。

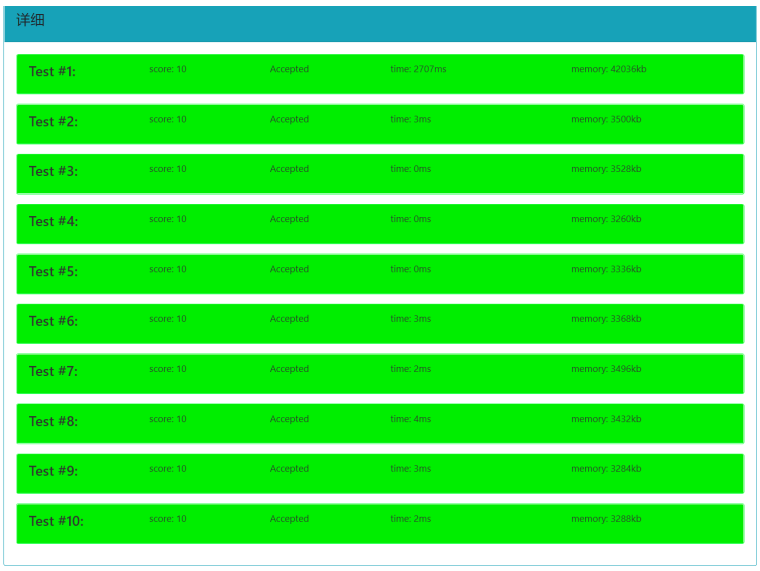
2. **核心代码+注释**

```
template<class T> void Heap<T>::Heapify() // 利用已有数组直接建堆
{
    for (int i = ((HeapSize - 1) - 1) >> 1; i >= 0; i--)
        percolateDown(i);
}
```

```
Heap<int> cities(population, N); // 城市人口数据建堆
    for (int i = 0; i < M; i++)
        result = cities.delMax(); // 不断取最大堆堆顶元素，取M次即可找到最大的M个数中最小的
    cout << result << endl;
```

3. **OJ 运行结果截图**

#49654	#76. 小蓝鲸的奇妙冒险-第一季	191830079	100	2724ms	42036kb	C++	2.2kb	2022-05-30 14:16:12



二、小蓝鲸的奇妙冒险-第二季

1.解题思路

此题与第一题不同处在于需要求出最大的`[x/M]`个元素的最小值，其中M是给定值，而x是动态变化的天数值。故我们这里可以考虑动态维护两个堆：小根堆用于存储最大的`[x/M]`个元素；大根堆用于存储较小的`x - [x/M]`个元素。每天新增一个元素，我们需考察其与小根堆堆顶的关系：若其大于小根堆堆顶，说明其是较大的那部分元素，应该插入小根堆，否则说明其是较小的那部分元素元素，插入大根堆。此外，在整个过程中，我们应该动态维护小根堆中元素个数始终为`[x/M]`，若其元素个数过多，则转移其堆顶到大根堆。

最后我们动态输出小根堆堆顶即可。可知此时堆顶一定是最大的`[x/M]`个元素的最小值。算法动态维护两个堆，在N次（一共N个城市）运行中每次都只进行常数次的插入或删除堆元素操作，两个堆的大小均不超过N，故总时间复杂度不超过`O(NlogN)`。空间复杂度为建堆所耗费的`O(N)`。

我们的堆只实现了大根堆，小根堆我们可以通过仿照STL中的`int_greater`，重载`<`和`>`运算符即可。

2.核心代码+注释

```
class int_greater // 重载int实现小根堆
{
public:
    int elem;
    int_greater(int _elem = 0): elem(_elem) {}
    bool operator>(const int_greater &another) { return elem < another.elem; }
    friend ostream& operator<<(ostream& out, const int_greater &i) { out <<
i.elem; return out; }
    friend bool operator<(const int x, const int_greater& i) {return x > i.elem;
}
};
```

```
Heap<int_less> MaxHeap(N);  // 大根堆，用于存储较小的 x - [x/M] 个元素
Heap<int_greater> MinHeap(N); // 小根堆，用于存储最大的[x/M]个元素
for (int i = 1; i < N; i++) {
        scanf("%d", &temp);
        if (temp > MinHeap.getMax().elem) // 若其大于小根堆堆顶，说明其是较大的那部分
([x/M]个)元素，应该插入小根堆
            MinHeap.insert(int_greater(temp));
        else
            MaxHeap.insert(int_less(temp));
        int pivot = (i + 1 + M - 1) / M; // 这个值就是 [x/M]
        if (MinHeap.size() > pivot) { // 动态维护小根堆中元素个数始终为 [x/M]
            MaxHeap.insert(int_less(MinHeap.delMax().elem));
        }
        else if (MinHeap.size() < pivot) { // 动态维护小根堆中元素个数始终为 [x/M]
            MinHeap.insert(int_greater(MaxHeap.delMax().elem));
        }
        printf("%d ", MinHeap.getMax().elem); // 此时小根堆堆顶一定是最大的`[x/M]`个元
素的最小值
    }
```

3.OJ 运行结果截图

#49668	#77. 小蓝鲸的奇妙冒险-第二季	191830079	100	291ms	11180kb	C++	3.4kb	2022-05-30 15:02:38

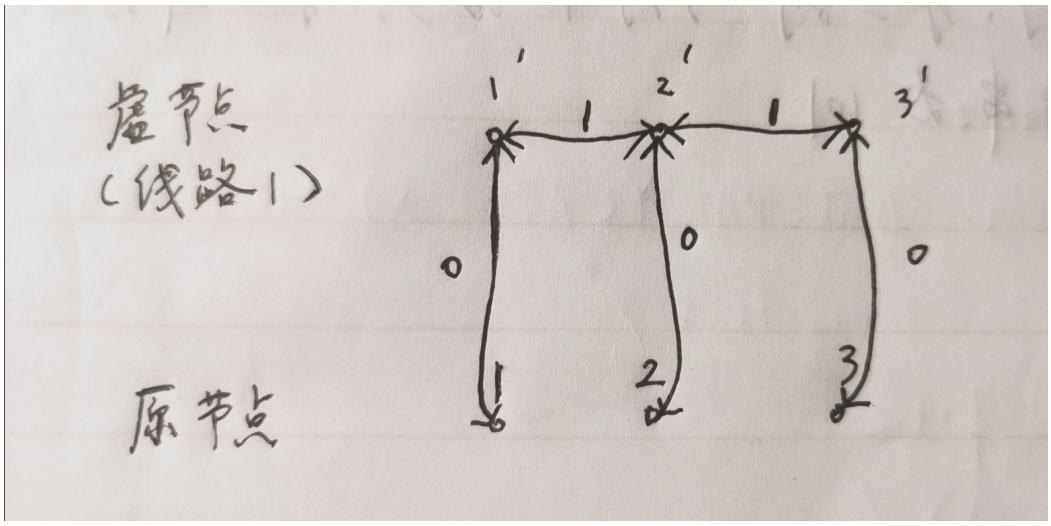
三、小蓝鲸的奇妙冒险-第三季

1.解题思路

显然考虑使用`Dijkstra`算法解决问题，与常规最短路问题不同的是，由于多条线路的存在，不同的国家（节点）之间存在多条边可以相联。显然不能在图中引入重边，所以我们应该利用分层图：设有n个国家（节点），m条线路，建图时，我们虚拟出`m X n`个国家虚节点，它们对应每条线路上对应的国家，若线路上这两个国家相连，则这两个国家所在线路的虚节点相连；此外，每条线路对应的虚节点与0 ~n范围内对应的国家节点（原节点）相连，权值为零。

这样构建的逻辑是：从国家原节点到某线路对应的虚节点，等于乘上对应线路出发；从虚节点回到对应原节点，等效于下车准备切换线路。这道题中切换线路不耗费时间，因此连接的权值为零。最后运行一遍`Dijkstra`算法，输出原节点中始末节点间最短距离即可。举例：

设存在三个国家，存在一条线路1->2->3，每两个国家之间流转需要花费的时间均为1，则我们建立的图如下所示：（图中边均为双向连接）



相关数据结构：堆与之前一样，图节点基于邻接表实现：（数组，并非基于链表）

```
struct node
{
    edge *neighbor;
    int val, degeree, father;
    node(int _val = 0): neighbor(new edge[10]), val(_val), degeree(0), father(0)
{}
    ~node() { delete[] neighbor; }
    void addEdge(edge _val) { neighbor[degeree++] = _val; }
};
```

时间复杂度分析：只需考察我们建立的图中的边数。需要注意，本题条件是`2<=每条路线途径国家数p<=10`，因此线路上所有国家不超过`10 X m`个。

建立边的过程有两次：虚节点与对应原节点，这部分边数为`O(20m) = O(m)`（考虑到双向）；线路上的虚节点之间连接，这部分同样为`O(m)`。

综上，算法的时间复杂度为`O(mlogm)`， 主要为`Dijkstra`算法运行开销；空间复杂度为`O(mn)`，主要为建图和建堆耗费的空间。

2.核心代码+注释

根据`m`条线路建图：

```
node *graph = new node[(m + 1) * n]; // m X n个虚节点和 n 个原节点
    // 其中 0 ~ n 对应原节点，之后每n个对应一条线路上的n个虚节点
for (int i = 1; i <= m; i++) {
        int p; // 读入该交通线途径的国家数量p
        cin >> p;
        int lineNodes[p];
        for (int j = 0; j < p; j++) {
            int lineNodenum;
            cin >> lineNodenum;
            lineNodes[j] = lineNodenum;
            graph[lineNodenum].addEdge(make_pair(0, i * n + lineNodenum)); // 每
条线路对应的虚节点与0 ~n范围内对应的国家节点（原节点）相连，权值为零
            graph[i * n + lineNodenum].addEdge(make_pair(0, lineNodenum));
        }
        for (int j = 0; j < p - 1; j++) { // 读入权值
            int weight;
            cin >> weight;
            graph[i * n + lineNodes[j]].addEdge(make_pair(weight, i * n +
lineNodes[j + 1])); // 若线路上这两个国家相连，则这两个国家所在线路的虚节点相连
            graph[i * n + lineNodes[j + 1]].addEdge(make_pair(weight, i * n +
lineNodes[j]));
        }
    }
```

`Dijkstra`算法：

```
MinHeap.insert(make_pair(0, src)); // 将源点加入堆中
    minDist[src] = 0;
    while (!MinHeap.isEmpty()) {
        edge e = MinHeap.getMax();
        MinHeap.delMax();
        int u = e.second;
        if (visited[u])
            continue;
        visited[u] = 1; // Finished
        for (int i = 0; i < graph[u].degeree; i++) {
            int v = graph[u].neighbor[i].second, weight =
graph[u].neighbor[i].first;
            if (!visited[v] && minDist[u] + weight < minDist[v]) { // 松弛
                minDist[v] = minDist[u] + weight;
                MinHeap.insert(make_pair(minDist[v], v)); // Add v to Fringe or
change v in Fringe
            }
        }
    }
```

3.OJ 运行结果截图

#50020	#78. 小蓝鲸的奇妙冒险-第三季	191830079	100	115ms	81992kb	C++	4.3kb	2022-06-03 17:26:23

四、小蓝鲸的奇妙冒险-第四季

1.解题思路

这题与第三题高度类似。几个不同点针对更改代码即可：

①线路变为单向，对应建图时两个相邻城市所在线路的虚节点相连，此时为单向边；

②切换进入线路有耗费时间，而离开时没有，因此虚节点到原节点连接边权值为0，原节点到虚节点连接边权值为切换到对应线路所耗费的时间。

③线路中可能有重复城市，因此虚节点注意只和原节点连接一次，不然会出现重边。

此题中每条线路上节点数`a_i`满足：`2<= a_i <=n+1`，因此图中边数为`O(mn)`量级，算法的时间复杂度为`O(mnlogmn)`， 主要为`Dijkstra`算法运行开销；空间复杂度为`O(mn)`，主要为建图和建堆耗费的空间。

2.核心代码+注释

建图部分：

```
for (int i = 1; i <= m; i++) {
        int p = nodesCnt[i];
        int lineNodes[p];
        int present[n + 1] = {0}; // present用于标记虚节点到原节点是否已经相连
        for (int j = 0; j < p; j++) { // 读入线路上每个节点
            int lineNodenum;
            cin >> lineNodenum;
            lineNodes[j] = lineNodenum;
            if (!present[lineNodenum]) { // 确保虚节点到原节点还未相连，避免重复连接
                graph[lineNodenum].addEdge(make_pair(switchCost[i], i * n +
lineNodenum)); // 原节点到虚节点连接边权值为切换到对应线路所耗费的时间
                graph[i * n + lineNodenum].addEdge(make_pair(0, lineNodenum));
// 离开不耗费时间，因此虚节点到原节点连接边权值为0
                present[lineNodenum] = 1;
            }
        }
        for (int j = 0; j < p - 1; j++) {
            int weight;
            cin >> weight;
            graph[i * n + lineNodes[j]].addEdge(make_pair(weight, i * n +
lineNodes[j + 1])); // 两个相邻城市所在线路的虚节点相连，此时为单向边
        }
    }
```

3.OJ 运行结果截图

#50113	#79. 小蓝鲸的奇妙冒险-第四季（选做题）	191830079	90	139ms	54000kb	C++	4.6kb	2022-06-05 10:17:59

五、小蓝鲸的奇妙冒险-第五季

1.解题思路

这题只得了40分，因此思路应该存在问题。我想在第四题的基础上进行一些更改，只变化`Dijkstra`算法中松弛的部分。之前我们只考虑时间这一个权值作为边权，现在需要综合考虑时间和战力，因此建立这样一个结构：

```
struct edge
{
    int product; // 时间和战力的乘积
    int time, combat; // 时间和战力
    int nodeNum;
    edge (int _time = 0, int _combat = 0, int _nodeNum = 0) : time(_time),
combat(_combat), product(_time * _combat), nodeNum(_nodeNum) {}
    bool operator<(const edge& another) { return product < another.product; }
};
```

之后`Dijkstra`算法中根据乘积做松弛。建图过程与第四题相同。但这样的思路可能还是考虑不周，或许双权值时不能再这样按之前的思路更新每个节点的最小距离?

算法的复杂度同第四题。

2.核心代码+注释

```
while (!MinHeap.isEmpty()) {
        edge e = MinHeap.getMax();
        MinHeap.delMax();
        int u = e.nodeNum, prev_time = e.time, prev_combat = e.combat;
        if (visited[u])
            continue;
        visited[u] = 1; // Finished
        for (int i = 0; i < graph[u].degeree; i++) {
            int v = graph[u].neighbor[i].nodeNum;
            int cur_time = graph[u].neighbor[i].time + prev_time; // 加入边后新的
时间
            int cur_combat = graph[u].neighbor[i].combat + prev_combat; // 加入边
后新的战力
            int new_product = cur_time * cur_combat; // 动态更新乘积
            if (!visited[v] && new_product < minDist[v]) { // 根据乘积做松弛
                minDist[v] = new_product;
                MinHeap.insert(edge(cur_time, cur_combat, v)); // Add v to
Fringe or change v in Fringe
            }
        }
    }
```

3.OJ 运行结果截图

#51102	#80. 小蓝鲸的奇妙冒险-第五季（选做题）	191830079	40	623ms	64300kb	C++	5.1kb	2022-06-19 17:25:11