实验5-Dijkstra算法

问题分析

- **最短路径的求解**

  采用堆优化版的dijkstra，对朴素版dijkstra进行了优化，在朴素版dijkstra中时间复杂度最高的寻找距离最短的点O(n^2)可以使用最小堆优化。

  1. 一号点的距离初始化为零，其他点初始化成无穷大。
  2. 将一号点放入堆中。
  3. 不断循环，直到堆空。每一次循环中执行的操作为：弹出堆顶（与朴素版diijkstra找到S外距离最短的点相同，并标记该点的最短路径已经确定）。用该点更新临界点的距离，若更新成功就加入到堆中。

```
int dijkstra(int st,int ed)
{
    memset(dist, 0x3f, sizeof(dist));
    priority_queue<pii,vector<pii>,greater<pii>> heap;
    heap.push({0,st});
    dist[st]=0;
    while (heap.size()) {
        auto [distance,cur]=heap.top();
        heap.pop();
        for (auto [x,w]: h[cur]) {
            if (distance+w<dist[x]) {//如果有更短的路径则更新
                dist[x]=distance+w;
                heap.push({dist[x],x});
                continue;
            }
        }
    }
    if (dist[ed]==0x3f3f3f3f) {
        return -1;
    }
    return dist[ed];
}
```

- **如果从顶点s到顶点t有不止一条最短路径，那么输出路段数最少者**

  为实现该要求，我们定义一个cnt数组记录该最短路径上边的数量，在最短路径相等的情况下，如果边更少，则进行更新

```
if (cnt[x]>cnt[cur]+1) {//如果路段数更少,则更新
    heap.push({dist[x],x});
    pre[x]=cur;
    cnt[x]=cnt[cur]+1;
    continue;
}
```

- **如果具有最短路径并且路段数也是最少的路径至少有2条，那么输出按顶点编号的字典序最小者。**

  要实现该要求，我们在最短路径相等且路段数也相等的情况下，去比较两条路径经过节点的字典序，定义一个cmp函数实现此功能

```
bool cmp(int x,int y){//比较到达x和y的路径字典序
    vector<int> a,b;
    a.push_back(x);
    b.push_back(y);
    while (pre[a.back()]) {//把前面的节点都存入vector中比较
        a.push_back(pre[a.back()]);
        b.push_back(pre[b.back()]);
    }
    //路径是从后往前得到的,所以反转之后比较
    reverse(a.begin(), a.end());
    reverse(b.begin(), b.end());
    for (int i=0; i<(int)a.size(); i++) {
        if (a[i]==b[i]) {//节点相等则继续比较
            continue;
        }
        if (a[i]<b[i]) {//如果路径a当前节点编号更小则返回true
            return true;
        }
        return false;
    }
    return false;
```

```
}
```

- **打印出最短路径**

  定义pre[i]为最短路径上i节点前面的一个节点，打印时借助一个栈，先将当前节点压入栈中，如果当前节点仍存在前驱节点，把其前驱节点也压入栈中，循环进行即可。

  打印时，从栈顶到栈底的元素依次为最短路径上从起点到终点的节点编号，依次打印即可

```
void printsfp(int ed){
    stack<int> st;
    st.push(ed);
    while (pre[st.top()]!=0) {
        st.push(pre[st.top()]);
    }
    while (st.size()) {
        cout<<st.top();
        st.pop();
        if (!st.empty()) {
            cout<<"->";
        }
    }
}
```

运行结果截图

```
5
-1 10  -1 30 100
-1 -1 50 -1  -1
-1 -1 -1 -1 10
-1 -1 20 -1 60
-1 -1 -1 -1 -1
1 5

6
-1 1 12 -1 -1 -1
-1 -1 9 3 -1 -1
-1 -1 -1 -1 5 -1
-1 -1 4 -1 13 13
-1 -1 -1 -1 -1 4
-1 -1 -1 -1 -1 -1
1 6
Case 1
The least distance from 1->5 is 60
the path is 1->4->3->5
Case 2
The least distance from 1->6 is 17
the path is 1->2->4->6
```

设计调试中的问题

本次实验中，对于找到路段数最少的最短路径比较容易实现，在找路段数最小的字典序最小的路径时，有一些问题需要注意，在最开始时，我只是简单比较前一个节点的序号大小，发现这样并不是整个字典序最小的，要对两条路径的所有节点进行比较才能准确无误地找出字典序最小的路径，所以定义上面的cmp函数即可。

实验体会

这次实验是对Dijkstra算法的拓展应用，对算法主体进行修改，便能打印出最短路径，并且找到最短路径中路段数最少者，进一步还能求出具有最短路径并且路段数也是最少的路径中按顶点编号的字典序最小者。层层递进，也考验了对于细节的处理。在这个过程中，我学到了怎么逻辑清晰地去进行条件判断，在适当的地方应用continue可以减少一些重复的条件判断，进而简化代码的书写。通过本次实验我也学会了如何高效地去给程序加一些限制条件，以至于能够输出我们想要的结果。

程序源码

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;
const int N=2e5+10;
int n,m,dist[N],pre[N],cnt[N];//dist数组记录距离 pre数组记录路径前一个点 cnt数组记录路径上边的数量
vector<pii> h[N];

bool cmp(int x,int y){//比较到达x和y的路径字典序
    vector<int> a,b;
    a.push_back(x);
    b.push_back(y);
    while (pre[a.back()]) {//把前面的节点都存入vector中比较
        a.push_back(pre[a.back()]);
        b.push_back(pre[b.back()]);
    }
    //路径是从后往前得到的,所以反转之后比较
    reverse(a.begin(), a.end());
    reverse(b.begin(), b.end());
    for (int i=0; i<(int)a.size(); i++) {
        if (a[i]==b[i]) {//节点相等则继续比较
            continue;
        }
        if (a[i]<b[i]) {//如果路径a当前节点编号更小则返回true
            return true;
        }
        return false;
    }
    return false;
}

int dijkstra(int st,int ed)
{
    memset(dist, 0x3f, sizeof(dist));
    memset(cnt, 0x3f, sizeof cnt);
    priority_queue<pii,vector<pii>,greater<pii>> heap;
    heap.push({0,st});
    dist[st]=0;
```

```
    cnt[st]=0;
    while (heap.size()) {
        auto [distance,cur]=heap.top();
        heap.pop();
        for (auto [x,w]: h[cur]) {
            if (distance+w>dist[x]) {
                continue;//路径长度更大则不用更新
            }
            if (distance+w<dist[x]) {//如果有更短的路径则直接更新
                dist[x]=distance+w;
                heap.push({dist[x],x});
                pre[x]=cur;
                cnt[x]=cnt[cur]+1;
                continue;
            }
            //路径长度相等,比较路段数
            if (cnt[x]<cnt[cur]+1) {
                continue;//当前路段数更少则不用更新
            }
            if (cnt[x]>cnt[cur]+1) {//如果路段数更少,则更新
                heap.push({dist[x],x});
                pre[x]=cur;
                cnt[x]=cnt[cur]+1;
                continue;
            }
            //路段数相等,比较路径节点字典序
            if (cmp(cur,pre[x])) {
                heap.push({dist[x],x});
                pre[x]=cur;
            }
        }
    }
    if (dist[ed]==0x3f3f3f3f) {
        return -1;
    }
    return dist[ed];
}

void printsfp(int ed){
    stack<int> st;
    st.push(ed);
    while (pre[st.top()]!=0) {
```

```
            st.push(pre[st.top()]);
        }
        while (st.size()) {
            cout<<st.top();
            st.pop();
            if (!st.empty()) {
                cout<<"->";
            }
        }
    }

    int main(){
        int cas=1;
        while (cin>>n) {
            for (int i=1; i<=n; i++) {
                for (int j=1; j<=n; j++) {
                    int w;
                    cin>>w;
                    if (w!=-1) {
                        h[i].push_back({j,w});
                    }
                }
            }
            int st,ed;
            cin>>st>>ed;
            cout<<"Case "<<cas++<<'\n';
            cout<<"The least distance from "<<st<<"->"<<ed<<" is "
    <<dijkstra(st,ed)<<'\n';
            cout<<"the path is ";
            printsfp(ed);
            cout<<'\n';
        }
    }
```