写在前面：为什么把这两个页面合二为一了呢？因为只需要把最短路中的优先队列换成普通的队列，就变成了广度优先搜索。

Dijkstra的三步走

前提条件

Dijkstra算法（又称标号法），运行结果是从权重为 0 的起始点开始，为所有顶点标号——标权重。

要求权重全部为正。（为负的话请另寻他法）

头文件只有stdio.h（输入输出）（string.h（memset初始化）……

以及 queue 和C++里万能的语句 using namespace std;

需要两个数组。两个数组的大小都是顶点数。

数据类型小一点的数组vis（用于记录顶点是否已经编号了。

数据类型大一点的数组dis（用于记录顶点权重（即运行结果）。

您还需要一个特殊的优先队列（priority_queue<pair<int,int> > q;

介绍一下工具（priority_queue（又称**大顶堆**。每次默认弹出最大元素。

和通常的队列语法类似（pop是弹出（push是压入。

top是堆顶元素（empty是判断是否为空。

其中（pair 是特殊的结构体，两元素 first 和 second （

比较时默认先比较 first 大小（first 相同时比较 second 大小。这是在强调序关系的 priority_queue 中可以调用的原理基础。

make_pair 函数，将输入两个元素按顺序结合，成为输出的 pair 类型结构体。

第一步：初始化

利用 memset 函数，将 dis 数组全写成 0x3f （正最大值，代指距离无穷大），将 vis 数组全写为 0 （未访问过）。

将起始点（标号为 0 的点，可以不止一个）全部压入堆，同时将对应 dis 数组（之前置为最大值了）置为 0 。

语句为：

```
    q.push(make_pair(0,begin));
    dis[begin]=0;
```

根据堆特征（ push 和 make_pair 连用。因为要对权重（距离）排序，所以权重是第一元素。这里的 0 ，

代表距离为 0 。

第二步：找下一个标号点

```
    while(!q.empty())
    {
        int x=q.top().second;
        q.pop();
        if(vis[x])
        {
            continue;
        }
        vis[x]=1;
```

这段的意思：只要堆 q 非空——（空的话就表示图里全标完了，结束就完事了）

令 x 是要找的下一标号点（的编号，那么x是第二元素）。那么 x 一定在堆顶。

于是top □second □pop组合拳连用。

但是堆顶未必是想要的元素，没准 x 已经被访问过了。因此，检查 vis 数组。如果已经访问，就直接 continue 掉这一循环，从下一循环重新找，本循环仅仅 pop 了一个元素而已。

然后，置 vis 数组为 1 ，表示已经访问过。

第三步：标号

令 i 跑遍上文节点 x 的所有邻居□□ for 循环，跑遍即可□i未必是节点编号）

这里依赖于图的建构。如果用矩阵写，就跑遍矩阵的一行。如果用邻接表写，就跑遍邻接表的一行。

注意，对于无向图，建构时要将两个方向全部写入（序号、权重）。

对于每一个邻居节点，无论标过与否，都跑一遍。这里 y 是节点编号□ time 是对应权重，然后有：

```
            if(dis[y]>dis[x]+time)
            {
                dis[y]=dis[x]+time;
                q.push(make_pair(-dis[y],y));
            }
```

只有新权重（节点+权重）比老权重小的时候，才编号，其余时候并不编号。编号完了，就把这个刚编的节点压入堆。

注意！这里的堆，默认是大顶堆，而每次取的时候（见第二步），要取最小的元素。

因此**每次压入的时候，将每个权重（第一元素）取负**，变相将大顶堆**改造成小顶堆**。这个操作很巧妙。

总共只有这三个步骤。后面没有了。

完整代码

```
void Dijkstra(int begin)
{
    //步骤一
    memset(dis,0x3f,sizeof(dis));
    memset(vis,0,sizeof(vis));
    q.push(make_pair(0,begin));
    dis[begin]=0;
    //步骤二
    while(!q.empty())
    {
        int x=q.top().second;
        q.pop();
        if(vis[x])
        {
            continue;
        }
        vis[x]=1;
        //步骤三
        int i;
        for(i=0;i<top[x];i++)
        {
            int y=V[x][i].first;
            int time=V[x][i].second;
            if(dis[y]>dis[x]+time)
            {
                dis[y]=dis[x]+time;
                q.push(make_pair(-dis[y],y));
            }
        }
    }
}
```

利用队列的BFS非递归实现

```
void bfs(int v)//以v开始做广度优先搜索（非递归实现，借助队列）
{
    list<int>::iterator it;
    visited[v] = true;
    cout << v << " ";
    queue<int> myque;
    myque.push(v);
    while (!myque.empty())
    {
        v = myque.front();
```

```
        myque.pop();
        for (it = graph[v].begin(); it != graph[v].end(); it++)
        {
            if (!visited[*it])
            {
                cout << *it << " ";
                myque.push(*it);
                visited[*it] = true;//访问过
            }
        }
    }
    cout << endl;
}
```

From:
https://wiki.cvbbacm.com/ - **CVBB ACM Team**

Permanent link:
https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:namespace:%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2_bfs_%E4%B8%8E%E6%A0%87%E6%95%B0%E6%9C%80%E7%9F%AD%E8%B7%AF_dijkstra&rev=1589795809

Last update: **2020/05/18 17:56**