2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：Djikstra 算法

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.13	实验学时	2		

一、实验问题描述

1、实验内容

Dijkstra 算法：对给定的一个（有向）图 G，及 G 中的两点 s、t，确定一条从 s 到 t 的最短路径。

附加要求：如果有多条最短路径，输出路段最少者，如果还有多条，输出字典序最小者。

2、实验目的

- 使用 Dijkstra 算法完成问题
- 理解分支界限法的基本思想，完成附加要求中的内容
- 探讨对 Dijkstra 算法的优化问题

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到在单源点最短路问题中，源点到自身的最短路一定为零。在所有与源点直接相连的点中，路径长度最短的边就一定属于该图的最短路树。因此，对于每一次加入的新的边，贪心地更新其对最短路的贡献，即可得到此连通块中单源点到其他所有目标点中的最短路径长度。

考虑在每一次松弛操作时记录更新来源，即可得到最短路径。**反向建图可贪心使字典序最小**。是否进行松弛操作逻辑需要根据题目要求进行调整。

2、算法设计与求解步骤

vis：当前该节点是否已经在最短路树上（或者说是否其最短路长度是否已经被确定）

G：原图的邻接矩阵，使用-1 表示无边

path：记录当前到此节点的最短路长度，路段个数和更新来源

- 重载路径节点类的小于运算符，用于更新时比较
- **反向建图**，寻找从终点向起点的最短路
- 初始化源点（终点）至自己的距离为 0，且访问过。其余节点均未访问。
- 在每一步中挑出最短路径长度最短的未被 vis 的节点，将其访问
- 考虑 vis 节点的所有松弛，实用小于运算符判断是否松弛
- 如果进行松弛，则记录更新来源，并更新最短路
- 重复以上操作直至所有节点都被 vis
- 从汇点（起点）回溯寻找符合要求的最短路

3、算法分析

考虑到 Dijkstra 算法的本质属于一种分支限界算法。对于一个 n 个节点的图，最多进行 n-1 次循环，每次循环确定一个活动节点。如果使用邻接矩阵，每次循环需要对所有 n 个点进行松弛检查。每次松弛操作时间复杂度为$O(1)$，故使用邻接矩阵的普通 Dijkstra 算法时间复杂度为$O(n^2)$，空间复杂度同样为$O(n^2)$。

回溯时最大深度为 n，每次回溯分支数为 1，时间成本为$O(1)$，故输出路径的时间复杂度为$O(n)$。

4、算法程序实现

篇幅原因，见附录或附件。

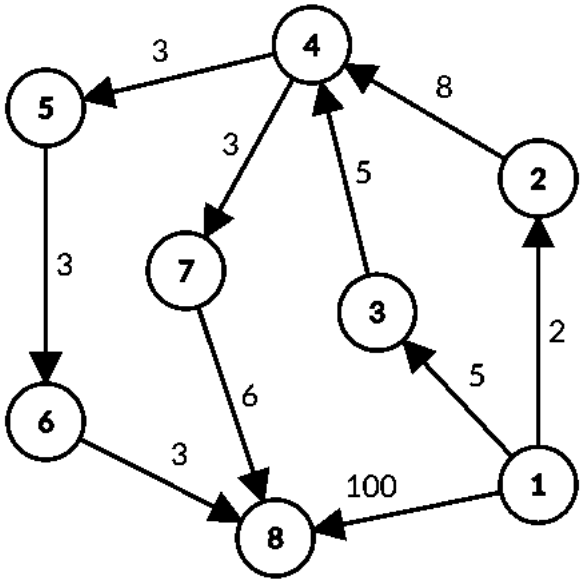
三、调试与运行

输入：

```
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

8
-1 2 5 -1 -1 -1 -1 100
-1 -1 -1 8 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1
-1 -1 -1 -1 3 -1 3 -1
-1 -1 -1 -1 -1 3 -1 -1
-1 -1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 6
-1 -1 -1 -1 -1 -1 -1 -1
1 8
```



输出：

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

8
-1 2 5 -1 -1 -1 -1 100
-1 -1 -1 8 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1
-1 -1 -1 -1 3 -1 3 -1
-1 -1 -1 -1 -1 3 -1 -1
-1 -1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 6
-1 -1 -1 -1 -1 -1 -1 -1
1 8
Case 3
The least distance from 1->8 is 19
the path is 1->2->4->7->8
```

四、结果分析

首先，原题干中**描述有误**（无向图却给了非对称矩阵作为邻接矩阵）。报告中已做修正。

反向建图的正确性保证：考虑到有向图中任何一个由 s 到 t 的路径，在其反向图中，都有一条同样的 t 到 s 的反向路径。由最短路的定义可知，这两个问题时等价的。

反向建图的原因：在路径长度和路段数相同的条件下，为了保证输出满足字典序最小，可以采用**类似基数排序**的贪心算法——从最低位开始每次都尽可能取来源节点编号最小的那一条边。不同于基数排序的是，该贪心算法只需要保留最小的那个编号即可，而不需要维护所有的编号。为了实现“从最低位开始贪心”的操作，采取了反向建图的操作。

事实上，本题采用的是没有堆优化的 Dijkstra 算法。如果进一步使用堆对本题进行优化，可以将最短路部分时间复杂度缩减到 $O(n\log n)$ 。但考虑到邻接矩阵输入方式的 IO 就已经到了 $O(n^2)$ 的规模，总时间复杂度保持不变。

五、本次实验的收获、心得体会

本次实验考察对分支限界算法的理解，同时单源最短路问题也是一个必须要掌握的图论基本算法。事实上，Dijkstra 算法早在数据结构和离散数学课堂上就已经有提到。但堆优化 Dijkstra 问题是第一次出现的。

对于本实验中的附加部分，如果能够清晰地理解 Dijkstra 算法的本质，是很好实现的。在实现过程前或过程中，需要想清楚字典序最小与原图中边来源的对应关系，才不至于在记录路径的时候提升时间复杂度和空间复杂度。

附录：代码实现（也可见附件）

```
#include <iostream>
using namespace std;

const int maxn=55; // maximum of n
const int inf=0x3f3f3f3f;   // infinity

struct node{
    int dis,len,pre; // distance, length, previous node number
    node(){}
    node(int d,int l,int p):dis(d),len(l),pre(p){}
    void init(){
        dis=len=pre=inf;
```

```
    }
    bool operator<(const node &qmh) const{
        if (dis!=qmh.dis) return dis<qmh.dis;    // distance is of the first priority
        if (len!=qmh.len) return len<qmh.len;    // then the length of the route
        return pre<qmh.pre;                      // then the previous node number
    }
}path[maxn];

bool vis[maxn];  // whether it is visited
int G[maxn][maxn]; // the graph

void dijkstra(int n,int s){ // node count, start point
    for (int i=1;i<=n;++i){
        vis[i]=false;
        path[i].init();
    }
    vis[s]=true; // start point has been visited
    path[s]={0,0,-1}; // update the path of the start point
    for (int i=1;i<=n;++i){ // the conjuncted nodes of the start point
        if (G[s][i]!=-1){
            path[i]={G[s][i],1,s};
        }
    }
    while (true){
        int next=-1;
        for (int i=1;i<=n;++i){
            if (!vis[i]&&(next==-1||path[i].dis<path[next].dis)){
                next=i; // find the node of the first priority
            }
        }
        if (next==-1) break; // break if all the connected nodes are visited
        vis[next]=true;
        for (int i=1;i<=n;++i){
            if (vis[i]||G[next][i]==-1) continue;
            node update(path[next].dis+G[next][i],path[next].len+1,next);
            if (update<path[i]){
                path[i]=update;
            }
        }
    }
}

void get_path(int p){
    if (path[p].pre==-1){ // already found the end of the route
        cout<<p<<endl;
        return;
    }
    cout<<p<<"->";
    get_path(path[p].pre);
}

int main(){
    int n,t1,t2,kase=0;
    while (cin>>n){
        for (int i=1;i<=n;++i){
            for (int j=1;j<=n;++j){
                cin>>G[j][i];
            }
        }
```

```
        cin>>t1>>t2;
        dijkstra(n,t2);
        cout<<"Case "<<++kase<<endl;
        cout<<"The least distance from "<<t1<<"->"<<t2<<" is "<<path[t1].dis<<endl;
        cout<<"the path is ";
        get_path(t1);
    }
    return 0;
}

/*
8
-1 2 5 -1 -1 -1 -1 100
-1 -1 -1 8 -1 -1 -1 -1
-1 -1 -1 5 -1 -1 -1 -1
-1 -1 -1 -1 3 -1 3 -1
-1 -1 -1 -1 -1 3 -1 -1
-1 -1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 6
-1 -1 -1 -1 -1 -1 -1 -1
1 8
*/
```