一、图 G=(V,E), 用 Jonhnson 算法，计算所有结点对之间的最短路径。

图的初始化相关代码：

constexpr auto MaxInt = 30000; //表示极大值，即∞,不用32767是为了防止溢出constexpr auto MVNum = 100; //最大顶点数

typedef char VerTexType; //顶点的数据类型

typedef int ArcType; //边的权值类型

struct Base { }; //为未命名的类命名

typedef struct NamedType : Base

{

vector<VerTexType> vexs{ MVNum }; //顶点表

vector<vector<ArcType>> arcs{ MVNum ,vector<ArcType>(MVNum,MaxInt)}; //邻接矩阵

int vexnum = 0, arcnum = 0; //图的当前点数和边数

}AMGraph;//Adjacency Matrix Graph

void denseCreateUDN(AMGraph& G)//创建稠密有向图函数

void sparseCreateUDN(AMGraph& G)//创建稀疏有向图函数

Bellman\_Ford函数及其测试：

该函数输入的int形参S作为回路的起点

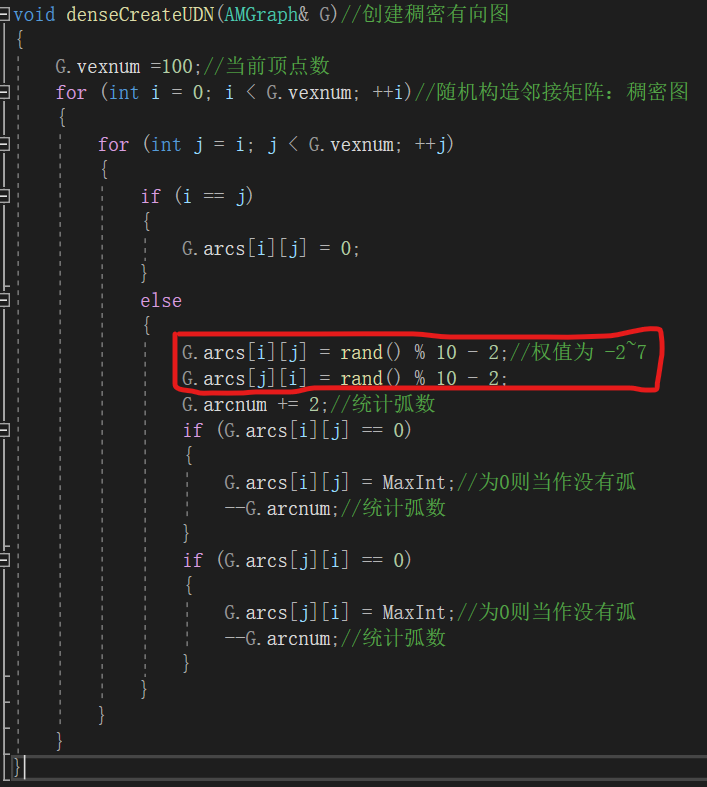
bool Bellman\_Ford(AMGraph& G, int S)

bool result = Bellman\_Ford(G, S);

if (result)

cout << "此有向图存在负权回路，计算两点之间距离可能没有意义" << endl;

函数测试：修改图的创建函数的如下语句，使得初始化的图带有负权值



函数输出：

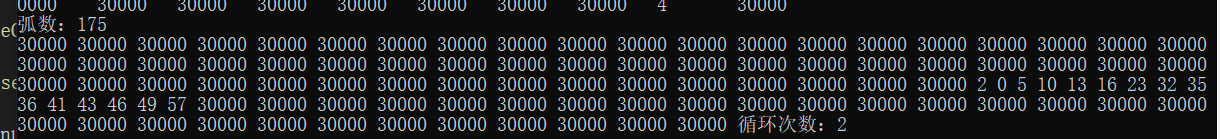


再使用没有负权值的稀疏图进行松弛操作的结果测试，函数中添加下列对每个顶点到起点的距离进行输出：

for (int i = 0; i < n; i++) {

cout << dis[i] << ' ';

}



Dijkstra算法：

void Dijkstra(AMGraph& G, int s)//斐波那契堆实现Dijkstra 最小二叉堆实现Dijkstra

最小优先队列，二选一：

binaryPriorityQueue<node> q; //二叉堆优先队列

//fibonacciPriorityQueue<node> q;//斐波那契堆优先队列

其中node定义如下：

struct node //顶点节点

{

int vlue;//表示访问的顶点

int key;//表示顶点到源点的距离

int parent;//表示顶点的父顶点

node()//构造函数

{

vlue = 0;

key = 0;

parent = NULL;

}

node& operator= (const node& t) {

this->key = t.key;

this->vlue = t.vlue;

this->parent = t.parent;

return \*this;

}

friend bool operator<(node a, node b) //因要实现最小堆,重定义优先级，以小为先

{

return a.key > b.key;

}

friend bool operator>(node a, node b)

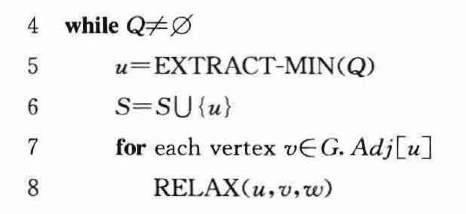
{

return a.key < b.key;

}

};

核心代码如下：



while (!q.empty()) //队列空说明完成了操作

{

node cd = q.top(); //取最小顶点

q.pop();

int u = cd.vlue;

if (visited[u]) //用于判断顶点是否已经在最短路径树中

continue;

visited[u] = true;

cout << "访问:" << u << " 距离:" << d[u].key << " 父结点:" << d[u].parent << "\t";

for (int i = 0; i < n-1; i++) //松弛操作

{

if (!visited[i] && d[i].key > d[u].key + G.arcs[d[u].vlue][i])

{

d[i].parent = u;

d[i].key = d[u].key + G.arcs[d[u].vlue][i];

q.push(d[i]);

}

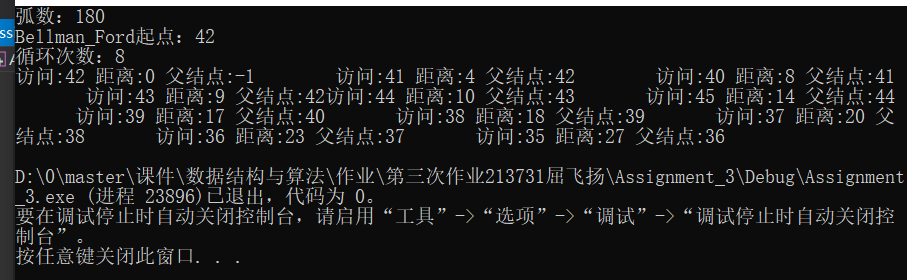
}

}

采用随机起点＋稀疏图进行测试：

int S = rand() % G.vexnum;//图的起点

Dijkstra(G, S);



距离：访问的节点距离源节点的最短距离，父节点为-1表示没有父节点，即该点是源节点

Jonhnson 算法：

1. 使用斐波那契堆来实现 Dijkstra 算法里面的最小优先队列

**使用斐波那契堆实现优先队列：结构如下**

//fibonacci堆实现优先队列

template<class T>

class FibHeap{

private:

struct node

{

T data;

bool mark = false;;

int degree = 0;

node\* left = this, \* right = this;

node\* child = nullptr, \* parent = nullptr;

};

int size = 0;//结点个数

node\* min = nullptr;//最小元素指针

node\* head = new node;//第一层元素的链表表头

T MIN\_VALUE;

void LeftInsert(node\* base, node\* tmp)//LeftInsert:在双向循环链表中把tmp插入base左边

void link(node\* y, node\* x)//把y插入到x下面

void Consolidate()//构建

public:

T top()//获得最小值

int empty()

void push(T data)//插入

FibHeap<T> Merge(FibHeap<T> H2)//合并两个斐波那契堆

void pop()//删除最小值

void Cut(node\* x, node\* y)//x为子结点，y为父结点

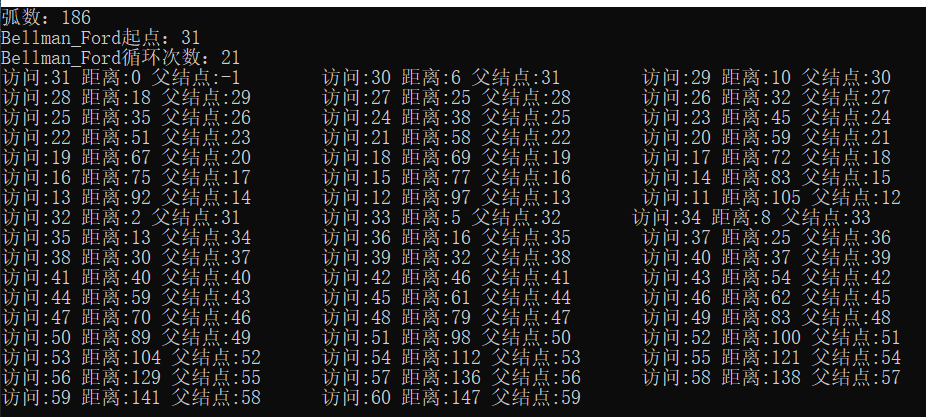
void CascadingCut(node\* y)//对父结点尝试进行级联切断

void DecreaseKey(node\* x, T key)//给定一个结点指针，并把它的值减小，重新调整堆的结构

void Delete(node\* x)//删除关键字操作（前提是有指向它的指针）。MIN\_VALUE需要在初始化类时给出，如int型设为INT\_MIN等

};

**测试如下：**



1. 使用最小二叉堆来实现 Dijkstra 算法里面的最小优先队列

**使用二叉堆实现优先队列：结构如下**

template<class T>

class binaryPriorityQueue{

private:

T\* pArray = new T[MVNum];// 根节点

int m\_length = 0;

public:

void swim(int k); //上浮

void sink(int k); //下沉

void push(T v); //插入

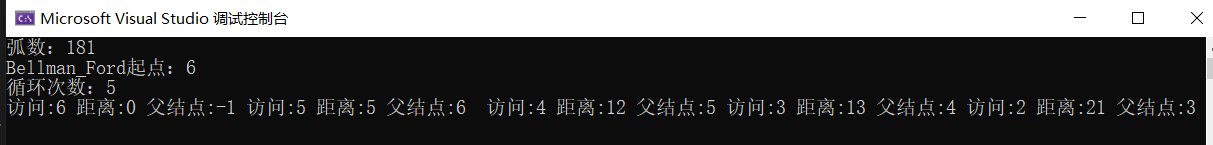
void pop();//删除堆顶

T top();//取顶点元素

bool empty();// 判断是否为空

}

**测试如下：**



**3、** 稀疏图、稠密图作为Jonhnson 算法的输入进行测试，两种实现方法的时间复杂度和空间复杂度分析：

**Johnson算法的实现：核心代码如下**

vector<vector<int>> Johnson(AMGraph& G,int s)

{

if (Bellman\_Ford(G,s))

cout << "the input graph contains a negative-weight cycle" << endl;

else

{

int h[MVNum];

int i, j, k;

vector<vector<int>> newWeight(MVNum,vector<int>(MVNum));

//将数组h[]的值设为运行BellmanFord后取得的值，h[i]为结点s到其他点的最短路径

for (i = 0; i < G.vexnum; i++)

h[i] = dis[i];

//遍历所有的边，将边的权值重新赋值,即将所有的边的权值改为负值

for (i = 0; i < G.vexnum; i++)

{

for (j = 0; j<G.vexnum; j++)

newWeight[i][j] = G.arcs[i][j] + h[i] - h[j];

}

vector<vector<int>> d(MVNum, vector<int>(MVNum));

//对每个结点运行dijkstra算法，求出每个点到其他点的最短路径，保存在key中

node\* key = new node[MVNum];

for (k = 1; k < MVNum; k++)

{

key = Dijkstra(G,k);

for (i = 1; i < MVNum; i++)

{

d[k][i] = key[i].key + h[i] - h[k];

}

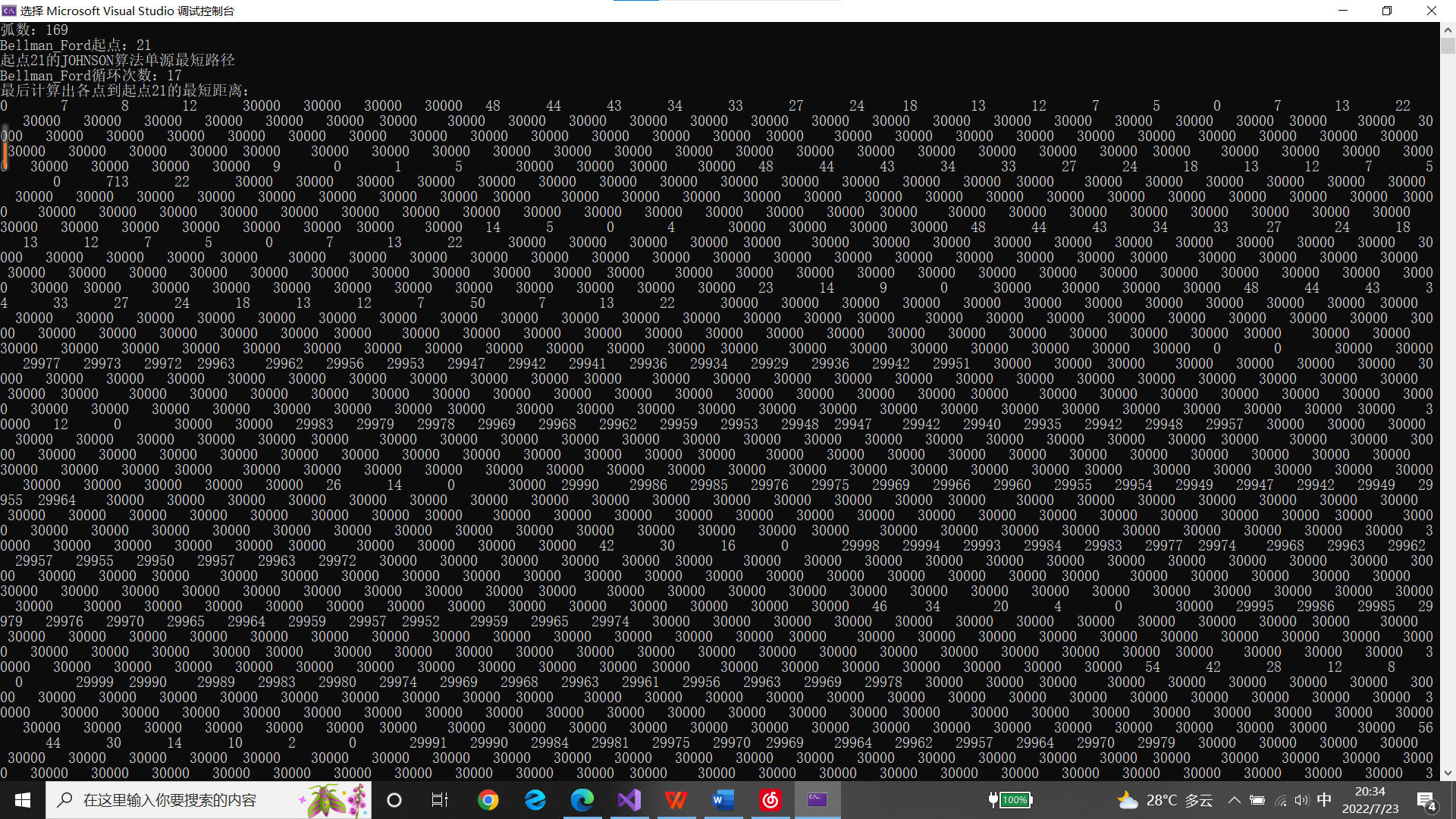
}

return d;

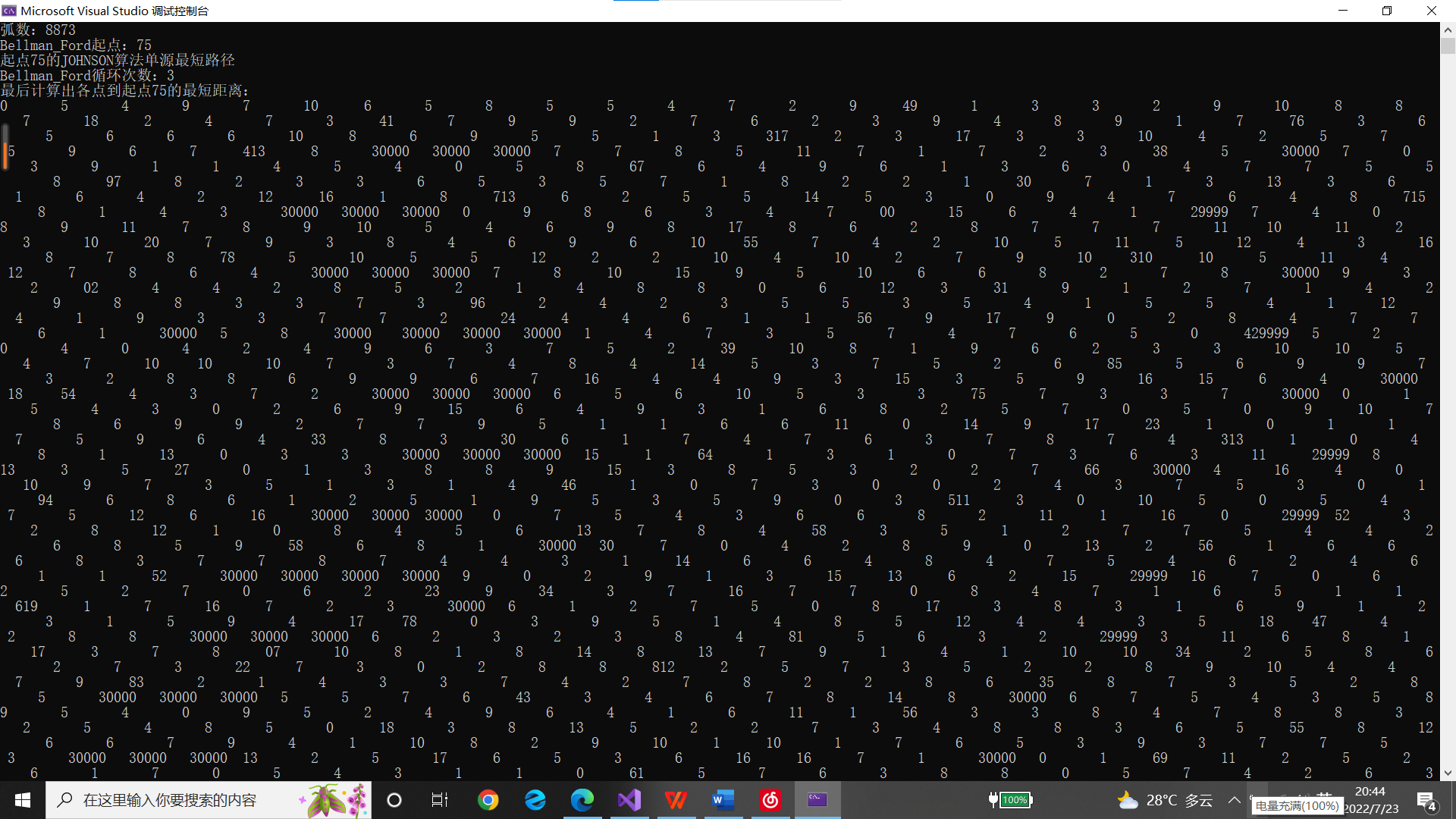
}

}

稀疏图测试：



稠密图测试：



时间复杂度和空间复杂度分析：

使用斐波那契堆实现Dijkstra的最小优先队列，Johnson算法的时间复杂度为O(*V*2*lgV+VE*)，使用最小二叉堆实现Dijkstra的最小优先队列，Johnson算法的时间复杂度为O(*VE lgV*)。

空间复杂度为O(*V+E*)。