项目说明文档

《离散数学》课程实验报告

——最小生成树

作 者 姓 名： 杨滕超

学 号： 2151298

指 导 教 师： 唐剑锋

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 题目简介](#_Toc495668153) 4

[1.1 题目原理背景](#_Toc495668154) 4

[1.1.1 最小生成树](#_Toc495668163) 4

[1.1.2 构造最小生成树的方法](#_Toc495668163) 4

[1.1.3 Prim算法](#_Toc495668163) 4

[1.1.4 Kruskal算法](#_Toc495668163) 4

[1.1.5 Solin算法](#_Toc495668163) 5

[1.1.6 Rosenstieh和管梅谷算法](#_Toc495668163) 5

[1.1.7 Dijkstra算法](#_Toc495668163) 5

[1.2 题目概述](#_Toc495668155) 6

[1.3 程序功能](#_Toc495668154) 6

[2 解题思路](#_Toc495668156) 7

[2.1 存图方式](#_Toc495668155) 7

[2.1 求解最小生成树](#_Toc495668155) 7

[3 数据结构设计](#_Toc495668161) 7

[3.1 边结点结构体](#_Toc495668162) 7

[3.1.1 说明](#_Toc495668167) 7

[3.1.2 UML图](#_Toc495668167) 8

[3.2 邻接表结点结构体](#_Toc495668162) 8

[3.2.1 说明 8](#_Toc495668167)

[3.2.2 UML图](#_Toc495668167) 8

[3.3 最小生成树](#_Toc495668162) 9

[3.3.1 说明](#_Toc495668167) 9

[3.3.2 UML图](#_Toc495668167) 9

[3.4 堆](#_Toc495668162) 10

[3.4.1 说明](#_Toc495668167) 10

[3.5 优先队列](#_Toc495668162) 10

[3.5.1 说明](#_Toc495668167) 10

[3.5.2 UML图](#_Toc495668167) 10

[3.6 向量](#_Toc495668162) 11

[3.6.1 说明](#_Toc495668167) 11

[3.6.2 UML图](#_Toc495668167) 11

[3.7 Vector游标类](#_Toc495668162) 12

[3.7.1 说明](#_Toc495668167) 12

[3.7.2 UML图](#_Toc495668167) 12

[4 核心算法](#_Toc495668161) 12

[4.1 堆排序](#_Toc495668174) 12

[4.1.1 算法说明](#_Toc495668176) 12

[4.1.2 主要代码](#_Toc495668176) 13

[4.2 顶点和便数量输入以及算法选择](#_Toc495668174) 14

[4.2.1 算法说明](#_Toc495668176) 14

[4.2.2 主要代码](#_Toc495668176) 14

[4.3 边输入](#_Toc495668174) 16

[4.3.1 算法说明](#_Toc495668176) 16

[4.3.2 主要代码](#_Toc495668176) 16

[4.4 kruskal算法](#_Toc495668174) 19

[4.4.1 流程图](#_Toc495668176) 19

[4.4.2 算法说明](#_Toc495668176) 19

[4.4.3 主要代码](#_Toc495668176) 20

[4.5 prim算法](#_Toc495668174) 21

[4.5.1 流程图](#_Toc495668176) 21

[4.5.2 算法说明](#_Toc495668176) 21

[4.5.3 主要代码](#_Toc495668176) 22

[4.6 对外接口函数](#_Toc495668174) 23

[4.6.1 算法说明](#_Toc495668176) 23

[4.6.2 主要代码](#_Toc495668176) 24

[4.7 并查集](#_Toc495668174) 24

[4.7.1 算法说明](#_Toc495668176) 24

[4.7.2 主要代码](#_Toc495668176) 24

[5 实验结果](#_Toc495668161) 26

[5.1 样例测试](#_Toc495668174) 26

[5.2 输入顶点数为0](#_Toc495668174) 26

[5.3 输入边数为0](#_Toc495668174) 27

[5.4 输入边数过少](#_Toc495668174) 27

[5.5 输入平行边](#_Toc495668174) 27

[5.6 输入非连通图](#_Toc495668174) 28

[5.7 输入顶点和边数量错误](#_Toc495668174) 28

[5.8 输入边错误](#_Toc495668174) 28

6 [心得体会](#_Toc495668161) 29

1 题目简介

* 1. 题目原理背景
     1. 最小生成树

一个连通图的生成树是原图的极小连通子图，它包含原图中的所有顶点，而且有尽可能少的边。这意味着对于生成树来说，若砍去它的一条边，就会使生成树变成非连通图；若给它增加一条边，就会形成图中的一个回路。

对于一个连通网络(即带权有向图)来说，构成最小生成树的准则有3条：

1. 有n个顶点的生成树有且仅有n - 1条属于该网络的边来连接所有的顶点；
2. 不能使用产生回路的边；
3. 树的总代价达到最小，即树中所有边的权重总和达到最小。

这里我们需要注意到的一点是：最小生成树的总代价一定最小，但这并不表明各边的代价一定最小。

若具有较小的边权值的边出现权值相同的情况，则构成的最小生成树可能并不唯一，但权值总和一定唯一。若所有边上的权值互不相同，则构造出来的最小生成树一定唯一。

* + 1. 构造最小生成树的方法

1. 避圈法：按照边的权重，从小到大依次将边添加到生成树中，如果构成圈则不选。这类构造最小生成树的方法主要有三种：Prim算法、Kruskal算法和Solin算法。
2. 破圈法：按照边的权重，删除圈中权重最大的边。这类构造最小生成树的方法主要有两种：管梅谷算法和Dijkstra算法。
   * 1. Prim算法

Prim算法的基本思路是：从连通网络N = {V,E}中的某一顶点u0出发，选择与它关联的具有最小权重的边(u0,v),将其顶点u0和v从V移入到生成树的顶点集合U中。以后每一步从一个顶点u在U中，而另一个顶点v不在U中的各条边当中选择权重最小的边(u, v),把顶点v从V移入到U当中，这意味着边(u, v)加入到生成树的边集合中。如此继续下去，直到V中所有顶i但都移入到U中，V变为空为止。

* + 1. Kruskal算法

Kruskal的基本思路是：设有一个连通网络N = {V,E}，顶点集合V中有n个顶点。最初先构造一个包括全部n个顶点和0条边的非连通图F，每个顶点自成一个连通分量Ti，并把E中的所有边按其权重从小到大排列放到H中。以后每一步从H中移出一条权重最小的边，如果的它的两个端点在F的不同连通分量上，则把该边和它的两个端点加入到生成树中，并把这两个连通分量合并；如果它的两个端点在同一个连通分量中，此边放弃，不加入生成树。如此重复，经过n - 1步，最终得到一棵有n - 1条边的各边权值总和最小的的最小生生成树。

这里需要补充的是是关系如何判断一边的两个端点是否在同一个连通分量当中以及如何将两个处于不同连通分量当中的顶点进行合并。这里采用了一个叫做并查集的数据结构。所谓并查集，这是一种集合的表现形式，又称为不相交集。它支持以下三种基本操作：

1. 合并，将集合S中的两个子集合合并，要求两个子集合不互相叫，否则不执行合并操作。
2. 查找，查找集合S中单个元素x所在的子集合，并返回该子集合的名字。
3. 初始化，将集合S中的每一个元素都初始化为只有一个单个元素的子集合。
   * 1. Solin算法

本算法可以看作是Prim算法和Kruskal算法的结合，其基本思路是：将求联通带权图的最小生成树的过程分为若干阶段，每一个阶段选取若干条边。具体步骤如下：

1. 将每个顶点视为一棵树，图中所有顶点形成一个森林；
2. 为每棵树选取一条边，它是该树与其他树相连的所有边中权值最小的一条边，把该边加入生成树中。如果是某棵树选取的边已经被其他树选过，该边就不再选取。
3. 重复（2)操作，直到整个森林变成一棵树为止。
   * 1. Rosenstiehl和管梅谷算法

这是一种典型的破圈法，其基本思路为：设有一个连通网络N = {V,E}，预先把E中所有边按其权值从大到小排列放到H中，然后依次取出H中权值最大的边，如果该边处于一个回路中，就从N中删去这条边；如果该边没有处于某个回路中，则在N中保留它。如此重复，直到N中只剩下n - 1条边(n为V中顶点个数)称为一棵树为止。此时N就是原图中的一棵最小生成树。

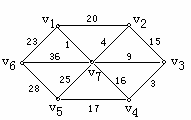
* + 1. Dijkstra算法

Dijkstra算法的基本思路为：对于一个连通网络N = {V, E}，把E中所有的边以方便的次序诸葛加入到初始为空的生成树的边集合T中。每次选择并加入一条边时，需要判断它是否会与先前加入T中的边构成回路。如果构成了回路，则从这个回路中将权值最大的边选退。如此重复，直到T中有n - 1条边为止，其中n是V中顶点个数。

* 1. 题目概述

如下图所示的赋权图表示某七个城市，预先计算出它们之间的一些直接通信道路造价（单位：万元），试给出一个设计方案，使得各城市之间既能够保持通信，又使得总造价最小，并计算其最小值。

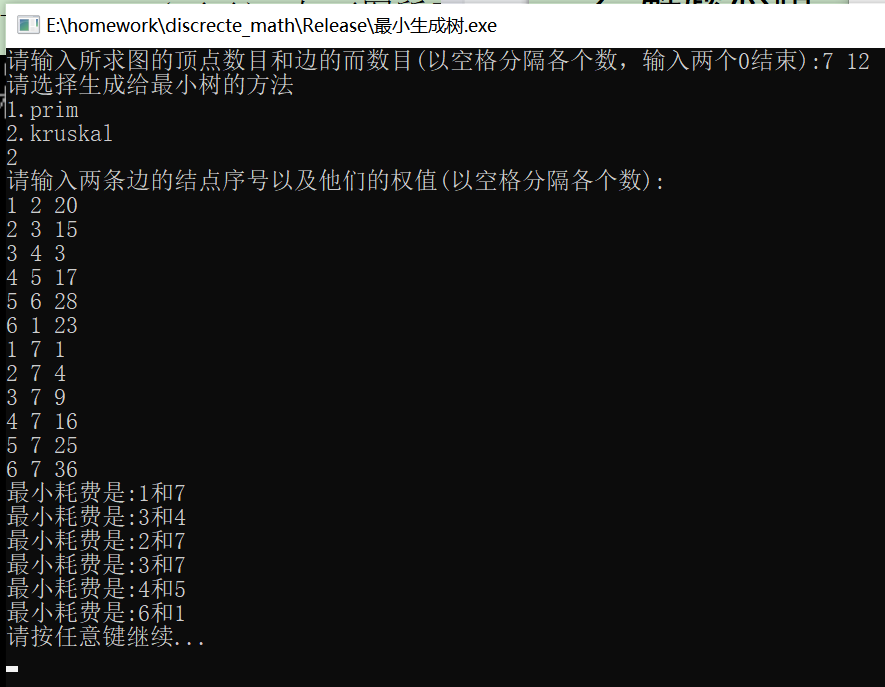
示例图如下：



* 1. 程序功能

根据用户输入的顶点和边的数量，请用户输入边以及边所对应的权值。输入边由两个顶点和权值组成，以空格分隔。最后输出生成最小生成树所需要连接的边(边的两个端点）。

程序示例如下：



2 解题思路

2.1 存图方式

在数据结构与算法设计中，对于图的存储我们一般有两种方式，一种是利用编程语言中的二维矩阵表示图的邻接矩阵；另一种是利用链表数组进行存储的邻接表。其中对于图的遍历，邻接矩阵的时间复杂度为O(n)；相对地，邻接表的时间复杂度为O(n + e)。而对于边的访问，邻接矩阵的时间复杂度为O(1),原因在于二维数组可以进行随机访问；而对于邻接表，访问的时间复杂度为O(e)。这里n表示为顶点的数量，e为边的数量。

因此，这里需要根据实际问题中的需求进行适合的数据结构的选择。若是稠密图，即边数接近n2级别，可以考虑选择邻接矩阵存储；若是稀疏图，即e << n2，可以考虑利用邻接表存储。同时，对于访问较多，较随机的操作，我们更考虑利用邻接矩阵，反之考虑利用邻接表。

对于本题目来说，若是仅仅考虑Kruskal算法的应用，我们可以利用边集数组进行图的存储，其中数组中的每一个元素是一个三元结构体，包含边的两个端点和权值三个信息。

我们进行更多思考，考虑Prim算法的实现，而对于Prim算法的存图方式，我们更倾向于邻接表的存储方式。原因在于：首先对于Prim算法执行的过程中需要进行图的遍历，虽然是顺序遍历，但对于日常生活中，我们所需要存储的图基本上都是稀疏图，因此使用邻接表存储更加节省空间；其次，对于稠密图来说，使用邻接表中的链表结构，在空间上的申请更加灵活，具有更好的优势。

2.2 求解最小生成树

对于本题目来说，我采用两种最小生成树的求解方式，包括Prim算法和Kruskal算法。对于算法的详细介绍上文已经提过，对于算法的说明将在后文进行详细阐述。

3 数据结构设计

3.1 边结点结构体

3.1.1 说明

这是为Kruskal算法服务的存储边的信息的结构体，其中每个结构体中元素的信息包括边的两个端点的编号以及边的权值。

利用构造函数对结点进行初始化。

代码如下：

struct edge {

int from, to, val;

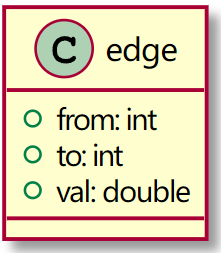
//构造函数

edge(int from = 0, int to = 0, double val = 0)

:from(from), to(to), val(val) {}

};

3.1.2 UML图



3.2 邻接表结点结构体

3.2.1 说明

这是邻接表需要利用的边的结点，里面存储的信息包括指向的顶点编号以及边权值的大小，最重要的是指向下一个顶点的指针。利用构造函数将结点进行初始化。

代码如下：

struct edgeNode {

int to;

double val;

edgeNode\* next;

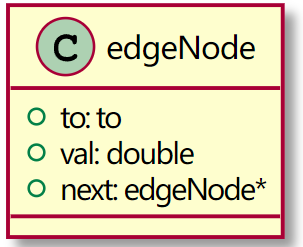
edgeNode(int to = 0, double val = 0, edgeNode\* next = NULL)

:to(to), val(val), next(next) {

}

};

3.2.2 UML图



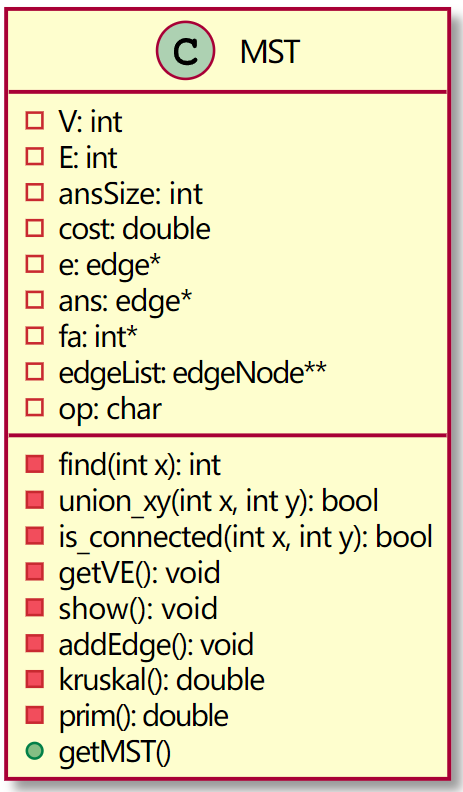
3.3 最小生成树类

3.3.1 说明

为了求解最小生成树，我们对本题目进行封装，其中的数据成员包括int类型的变量记录顶点数量、边数量以及答案数组的大小；double类型的变量记录花费的总代价，edge\*边结点的指针类型作为边集数组的首地址以及答案数组的首地址，int\*整型变量的指针fa作为并查集的数组首地址；edgeNode\*\*邻接表结点二重指针为邻接表做准备；char字符型变量作为记录用户输入选择的算法的变量。

成员函数主要包括并查集合并、查找、判断是否在同一个集合函数；输入顶点和边的数量函数；输入边的函数；构造最小生成树函数以及展示结果函数。

3.3.2 UML图



3.4 堆

3.4.1 说明

堆是一种树形结构，是完全树，因此常常利用数组储存。分为大根堆和小根堆，以小根堆为例，任意一个根的孩子结点的值总是不小于其父节点，同时两个孩子也是一个小根堆。

我们利用向量作为堆的基础，通过迭代方式，实现堆的操作，主要成员函数如下：

·上浮调整void siftFix(Iter beg, Iter end, Compare cmp);

·下沉调整void sinkFix(Iter beg, Iter end, Compare cmp);

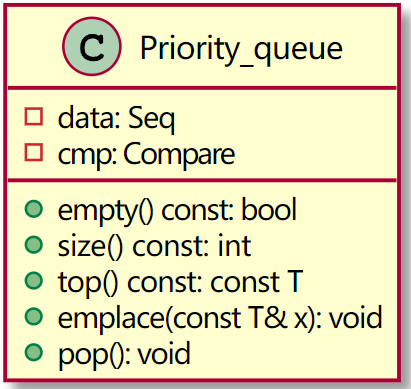
3.5 优先队列

3.5.1 说明

基于Vector类作为存储数据的数据结构，Heap类中的方法，形成优先队列，其中参数模板包括数据类型、数据底层结构（默认为Vector）以及元素之间的比较方法。比较方法作为仿函数传入类中作为类的数据成员。其中入队出队的时间复杂度均为O(logn)。

其中定义模板类型Seq与仿函数Compare，以适应不同的数据类型，进一步增强代码的可复用性。Seq类型代表优先队列底层的存储结构，Compare代表数据之间比较的仿函数。

3.5.2 UML图

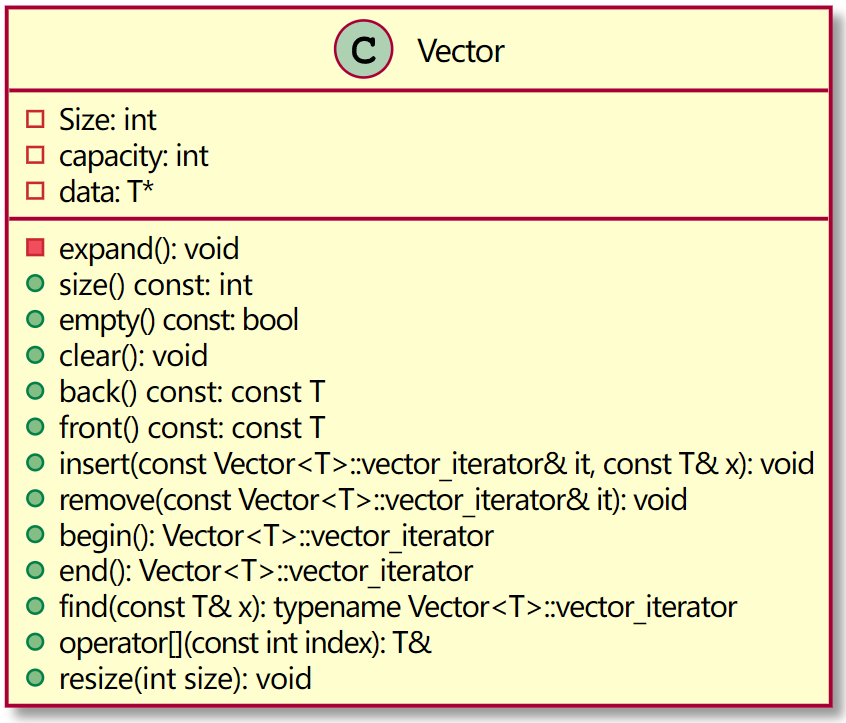


3.6 向量

3.6.1 说明

向量作为存储同一种类型数据的一维数组，相当于是数组的扩展，在其基础上增加方便程序员工作的操作。其中的优点在于：对于查找某个位置的元素的时间复杂度为O(1)，但是它的缺点也十分明显：向量的删除与插入的时间代价是巨大的，时间复杂度基本是O(n)级别；但特殊的是，在最后插入元素时，时间复杂度为O(1)

3.6.2 UML图

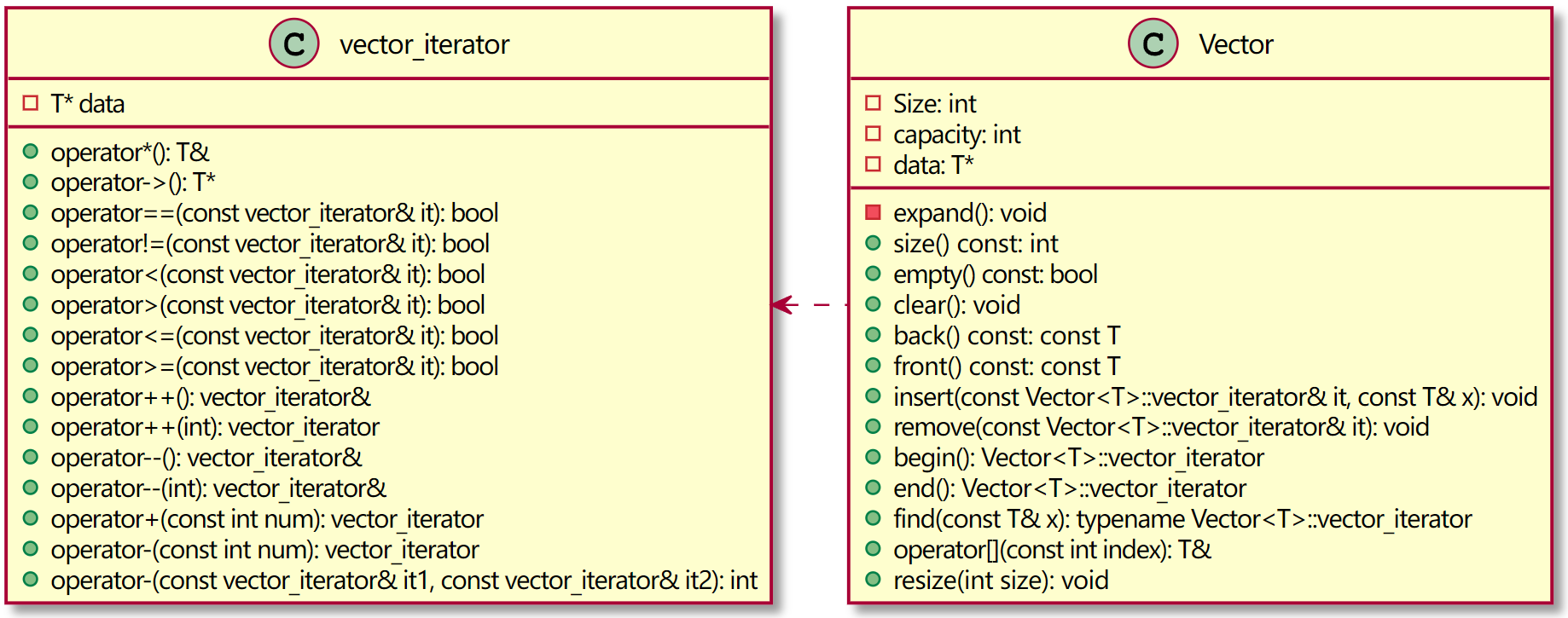


3.7 Vector游标类

3.7.1 说明

其中数据成员包含了指向Vector中数组元素的指针，并实现的重载\*、->、++、--、<、>等操作符。

3.7.2 UML图



4 核心算法

4.1 堆排序

4.1.1 算法说明

堆排序的基本思想是：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余n-1个元素重新构造成一个堆，这样会得到n个元素的次小值。如此反复执行，便能得到一个有序序列了。

首先需要将待排序序列调整为大顶堆，从最后一个非叶子结点开始进行向下调整，不断向上进行迭代，直到到堆顶为止。在本题目中，向下调整的过程为

（1）判断index位置有没有孩子；

（2）因为堆是完全二叉树，没有左孩子就一定没有右孩子，所以判断是否有左孩子；

（3）因为对的存储结构是数组，所以判断是否有左孩子即判断左孩子下标是否越界，即left>=size越界。

确定left或right，谁是index的最大孩子max；

（1）如果右孩子不存在，则max=left；

（2）否则，比较array[left]和array[right]值的大小，选择小的为min。

比较array[index]的值和array[max]的值，如果array[index]<=array[max]，则满足堆的性质，调整结束；

否则，交换 array[index] 和 array[max] 的值；

然后因为max位置的堆的性质可能被破坏，所以把 max视作 index，向下重复以上过程。

向下调整的时间复杂度为O(nlog2n)，将整个数组调整为堆的时间复杂度为O(nlog2n)。

之后从后往前，将最后的元素与堆顶元素交换，并从堆顶进行一次向下调整。由次进行n - 1次交换，得到的即为有序序列。

4.1.2 主要代码

void heapSort(edge\* nums, int n)

{

//初始化堆，从中间往前

for (int i = (n - 1) / 2; i >= 0; --i)

heapify(nums, i, n - 1);//下标从[i, n - 1]

//堆排序，将已经排好的堆 top 放到最后（相当于出堆）

for (int i = n - 1; i > 0; --i)//从后往前放top

{

edge t = nums[0];

nums[0] = nums[i];

nums[i] = t;

heapify(nums, 0, i - 1);//放到最后了，重新堆化

}

}

void heapify(edge\* nums, int start, int end)//[beg, end]

{

int dad = start, son = 2 \* dad + 1;

while (son <= end)

{

//取大的那个儿子

if ((son + 1) <= end && nums[son] > nums[son + 1])

++son;

if (nums[dad] < nums[son])//dad比son大，直接结束，因为要的是top最大

return;

//小的nums[dad]下沉到较大的子结点的位置

edge t = nums[dad];

nums[dad] = nums[son];

nums[son] = t;

dad = son;//迭代到更大的子结点

son = dad \* 2 + 1;

}

}

4.2 顶点和边数量输入以及算法选择

4.2.1 算法说明

这里对于用户输入的顶点和边的数量，但我们在这里需要判断输入是否正确。设置一个永真循环，若输入非法或负数，则输出输入错误提示，并提示重新输入；反之跳出循环。

若检查到输入的顶点数量或者边的数量为0，表示这样的图不需要构造最小生成树，直接退出函数，结束程序。

若检查到变得数量E < V - 1，其中V为顶点数量，则说明边的数量不足以构成连通图，我们需要输出提示，边数过少，然后退出函数，结束程序。

进入算法选择，本程序提供两种算法构造最小生成树，一种是prim算法，另一种是kruskal算法。在进行算法选择的同时，我们也需要进行用户输入的正确判断，若输入错误需要输出输入错误的相关提示，反之跳出循环。这里利用类内的数据成员char类型的op记录所输入的选择。

如果选择的算法输入为”1”，说明用户选择的是prim算法。由此构建邻接表，同时我们动态申请大小为V的数组空间作为链表数组。这里不要忘记，将数组初始化。

如果选择的算法输入为”2”，说明用户选择的是kruskal算法。我们需要动态申请并查集的数组空间以及边集数组空间，并且进行初始化。

最后，不论选择什么算法，都需要一定的空间存储记录答案的数组，其中记录最小生成树的边的信息。

4.2.2 主要代码

void getVE() {

cout << "请输入所求图的顶点数目和边的而数目(以空格分隔各个数，输入两个0结束):";

//错误判断

while (1) {

cin >> V >> E;

if (cin.fail() || V < 0 || E < 0)

{

cout << "请输入非负整数!" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else

break;

}

//没有顶点或没有边

if (V == 0 || E == 0)

return;

if (E < V - 1)

{

cout << "边数过少，不能生成最小生成树!" << endl;

return;

}

//选择算法

cout << "请选择生成给最小树的方法" << endl;

cout << "1.prim" << endl;

cout << "2.kruskal" << endl;

while (1) {

cin >> op;

if (op == '1' || op == '2')

break;

}

//选择prim算法构建邻接表

if (op == '1')

{

edgeList = new(nothrow) edgeNode \* [V];

if (edgeList == NULL)

{

cerr << "申请空间失败 " << endl;

exit(1);

}

//初始化

for (int i = 0; i < V; ++i)

edgeList[i] = NULL;

}

//选择kruskal算法构建并查集和边集数组

else

{

//边集数组申请空间

e = new(nothrow) edge[E];

if (e == NULL)

{

cerr << "申请空间失败 " << endl;

exit(1);

}

//并查集数组申请空间

fa = new(nothrow) int[V];

if (fa == NULL)

{

cerr << "申请空间失败 " << endl;

exit(1);

}

//初始化fa数组

for (int i = 0; i < V; ++i)

fa[i] = i;

}

//不管选择什么算法都要为答案数组申请空间

//答案数组同样申请空间

ans = new(nothrow) edge[E];

if (ans == NULL)

{

cerr << "申请空间失败 " << endl;

exit(1);

}

}

4.3 边输入

4.3.1 算法说明

根据用户输入边的数量E进行输入。这里同样要进行输入的错误判断，其中包括：非法字符、结点编号不在范围内以及权值为负数。判断无误后再进行图中边的添加，否则提示重新输入。

若选择是prim算法，我们需要对邻接表进行边结点的添加。首先通过边的起点查找到链表数组中对应的链表首元节点。这里需要注意的是需要先查找是否存在具有相同端点的平行边。若查找成功，需要进行权值的比较，取较小的那个权值，否则直接动态申请结点空间，添加入链表数组。

若选择的是kruskal算法，我们可以直接对边集数组添加edge元素。

4.3.2 主要代码

void addEdge() {

int from = 0, to = 0;

double val = 0;

cout << "请输入两条边的结点序号以及他们的权值(以空格分隔各个数):" << endl;

for (int i = 0; i < E;)

{

cin >> from >> to >> val;

if (cin.fail())

{

cout << "输入非法字符，请输入正整数!" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else if (from <= 0 || from > V || to <= 0 || to > V)

{

cout << "节点序号不在范围内，请重新输入!" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else if (val < 0)

{

cout << "权值不能为负数，请输入正整数!" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else

{

if (op == '1')//prim算法

{

--from, --to;

//存入邻接表，无向边，正反都要存

if (edgeList[from] == NULL)

edgeList[from] = new edgeNode(to, val);

else

{

edgeNode\* cur = edgeList[from];

while (cur && cur->to != to)

cur = cur->next;

//出循环 到最后了， 找到相同的

if (cur)//相同的边

{

if (cur->val > val)

cur->val = val;//取最小的存入

}

else//到最后没找到相同的边，插入头头

{

edgeNode\* newNode = new edgeNode(to, val, edgeList[from]);

edgeList[from] = newNode;

}

}

if (edgeList[to] == NULL)

edgeList[to] = new edgeNode(from, val);

else

{

edgeNode\* cur = edgeList[to];

while (cur && cur->to != from)

cur = cur->next;

//出循环 到最后了， 找到相同的

if (cur)//相同的边

{

if (cur->val > val)

cur->val = val;//取最小的存入

}

else//到最后没找到相同的边，插入头头

{

edgeNode\* newNode = new edgeNode(from, val, edgeList[to]);

edgeList[to] = newNode;

}

}

++i;//让循环继续

}

else//kruskal算法

{

//直接存入边集数组，无所谓有误重复，最后排序取前面权值小的

e[i++] = edge(from - 1, to - 1, val);

}

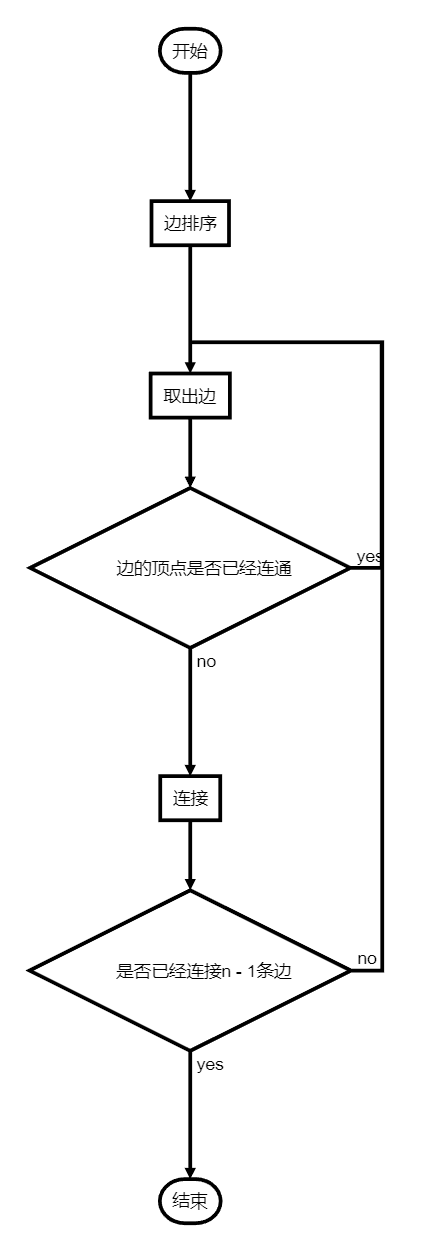
}

}

}

4.4 kruskal算法

4.4.1 流程图



4.4.2 算法说明

首先对边集数组进行堆排序。然后从小到大将边加入到最小生成树中，若构成回路则跳过该边，直到添加完至n - 1条边结束，跳出循环。在上述过程中，记录添加的边至答案数组。

最后需要检查添加的边是否为n - 1条，若不是则返回-1，说明不能构造最小生成树，反之构造成功，返回构造所花费的代价。

判断是否构成回路，我们利用并查集，检查顶点所在的集合是否相同，不相同说明添加该边不会构成回路，添加该边进入答案集合，并将两个顶点所在的集合合并为同一个集合。

这里算法时间复杂度为分析为：堆排序O(Elog2E)，并查集的查找与合并O(Elog2V)。因此我们总体的时间复杂度为O(Elog2E + Elog2V)。

4.4.3 主要代码

double kruskal()

{

//边集数组从小到大排序

heapSort(e, E);

//答案数组记录下标，顺便记录连通的边的数量

int idx = 0;

for (int i = 0; i < E; ++i)

{

//已经在同一个集合中就不能添加边，否则成环

if (is\_connected(e[i].from, e[i].to))

continue;

//记录代价

cost += e[i].val;

//合并到同一集合中

union\_xy(e[i].from, e[i].to);

//加入答案数组

ans[idx++] = e[i];

//生成够了n - 1条边

if (idx == V - 1)

break;

}

//记录答案数组的大小

ansSize = idx;

//返回花费的代价

if (idx == V - 1)

return cost;

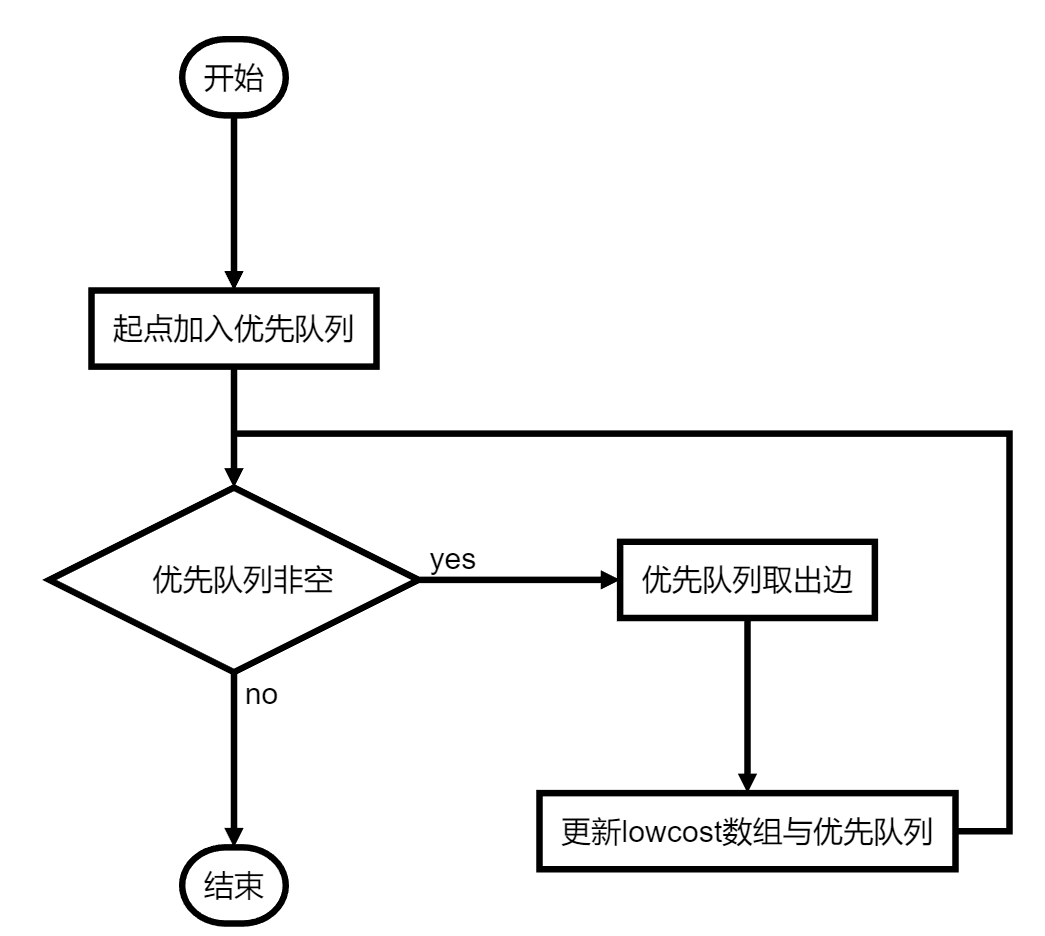
else

return -1;

}

4.5 prim算法

4.5.1 流程图



4.5.2 算法说明

Prim算法的具体过程为：任意选择图中的一个顶点作为生成树的根结点，并声明一个标记数组，记录已经加入生成树的集合中的结点；声明一个距离数组dis[]，其中dis[i]表示生成树集合中的顶点到顶点i的最短距离。这里我们利用优先队列优化，使得每次取到的都是队列中权值最小的边，每次循环中，我们需要判断取到的边的对应顶点是否已经在生成树的顶点集合中，若在则跳过本轮循环，若不在则将该点加入最小生成树的顶点集合——进行标记，并且加入答案边集。然后在该点周围寻找没有遍历过的且能更新dis[]的顶点，存在则加入优先队列，进入下一次循环。

可以发现Prim算法与Dijkstra算法类似，只是在更新dis数组的时候略有不同。前者是为了每条边的权值都尽可能短，后者是为了使得从开始的结点到其余每个结点的路径尽可能短。由此说明使用Dijkstra算法虽然也可以生成生成树，但并不一定是最小生成树。

上述过程由于利用堆优化，因此我们将时间复杂度从O(n2)降低到了O(logn)。

最后需要检查添加的边是否为n - 1条，若不是则返回-1，说明不能构造最小生成树，反之构造成功，返回构造所花费的代价。

4.5.3 主要代码

double prim(int start = 0)

{

//申请辅助数组空间

//vis记录顶点是否访问过

//dis记录从start顶点到其余顶点花费最小代价

bool\* vis = new(nothrow) bool[V];

if (vis == NULL)

{

cerr << "申请空间失败 " << endl;

exit(1);

}

double\* dis = new(nothrow) double[V];

if (dis == NULL)

{

cerr << "申请空间失败 " << endl;

exit(1);

}

//初始值

for (int i = 0; i < V; ++i)

{

vis[i] = false;

dis[i] = INT\_MAX;

}

//利用优先队列优化

Priority\_queue<edge> q;

q.emplace(edge(start, start, 0));

edge temp;

//记录最小生成树的顶点数

int cnt = 0;

while (!q.empty())

{

//取出最小距离

temp = q.top();

q.pop();

//遍历过了就跳过

if (vis[temp.to])

continue;

//没遍历过就加入答案

if(temp.from != temp.to)

ans[ansSize++] = temp;

//标记遍历过的

vis[temp.to] = true;

//记录最小生成树的顶点数

++cnt;

//更新代价

cost += temp.val;

//从temp.to周围找短的边

edgeNode\* cur = edgeList[temp.to];

while (cur)

{

//没有遍历过，并且从start到cur->to比temp.to到cur->to长

if (vis[cur->to] == false && cur->val < dis[cur->to])

{

dis[cur->to] = cur->val;

q.emplace(edge(temp.to, cur->to, cur->val));

}

//下一个

cur = cur->next;

}

}//end of while

//都连通

if (cnt == V)

return cost;

else

return -1;

}

4.6 对外接口函数

4.6.1 算法说明

首先调用顶点和边的数量输入函数，判断所输入的数量是否存在0，直接退出，结束函数。然后调用添加边的函数，进行算法的选择，得到prim算法函数或kruskal算法函数的返回值，若返回值为-1，说明不能构成最小生成树，由此输出错误提示，反之输出构成最小生成树的边的答案集合。

4.6.2 主要代码

void getMST() {

getVE();

if (V == 0 || E == 0)

return;

addEdge();

double res = -1;

if (op == '1')

res = prim();

else

res = kruskal();

//返回-1说明不连通

if (res != -1)

show();

else

cout << "不能生成最小生成树" << endl;

}

4.7 并查集

4.7.1 算法说明

并查集将fa数组初始化为每个顶点的双亲结点为自己，即fa[i] = i。

对于查找函数，利用while循环，当自己的双亲不是自己时，将自己的双亲改为双亲的双亲，进行路径压缩，以保证下一次查找的时候减少时间的花费。然后将双亲的值赋给自己，继续向上寻找。

对于合并函数，其中调用查找函数，找到x的根结点的编号和y的根结点编号，分别为root\_x和root\_y，将root\_x的双亲设置为root\_y，进行两个不同集合的合并。这里需要考虑两个集合的根是否相同，若相同则无需进行合并，返回false。由此可以将两个处于不同连通分量的顶点相连，合并两个连通分量。

对于判断两个顶点是否处于同一个集合，其中调用查找函数，找到x的根结点的编号和y的根结点编号，分别为root\_x和root\_y，判断两个集合的根是否相同，即可以判断两个顶点是否处于同一个集合。由此可以判断两个顶点是否处在同一个连通分量。

4.7.2 主要代码

inline int find(int x) {

while (fa[x] != x)

{

fa[x] = fa[fa[x]];

x = fa[x];

}

return x;

}

bool union\_xy(int x, int y)

{

//寻根

int root\_x = find(x);

int root\_y = find(y);

//判断根是否相同

if (root\_x == root\_y)

return false;

//不相同才合并

fa[root\_x] = root\_y;

return true;

}

bool is\_connected(int x, int y)

{

int root\_x = find(x);

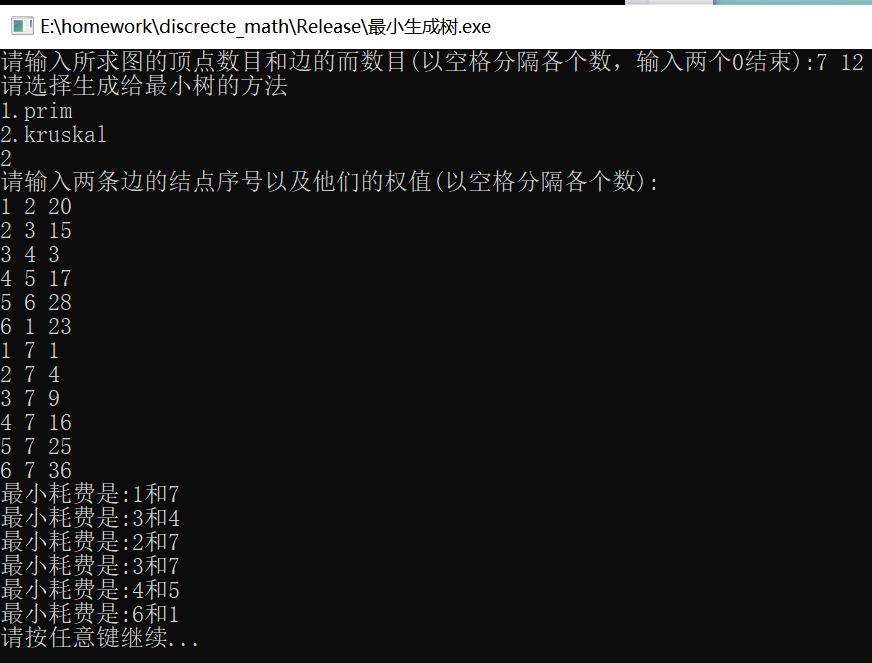
int root\_y = find(y);

return root\_x == root\_y;

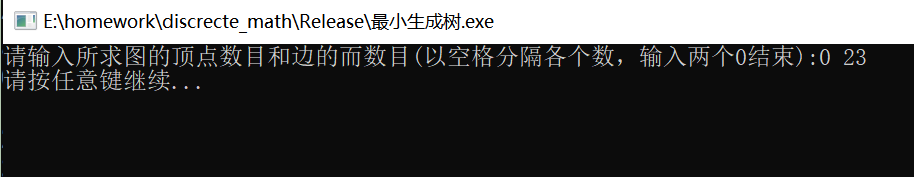
}

5 实验结果

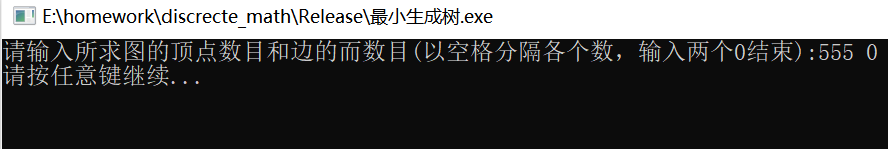
5.1 样例测试



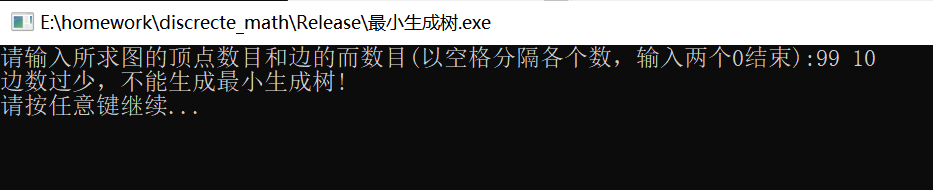
5.2 输入顶点数为0



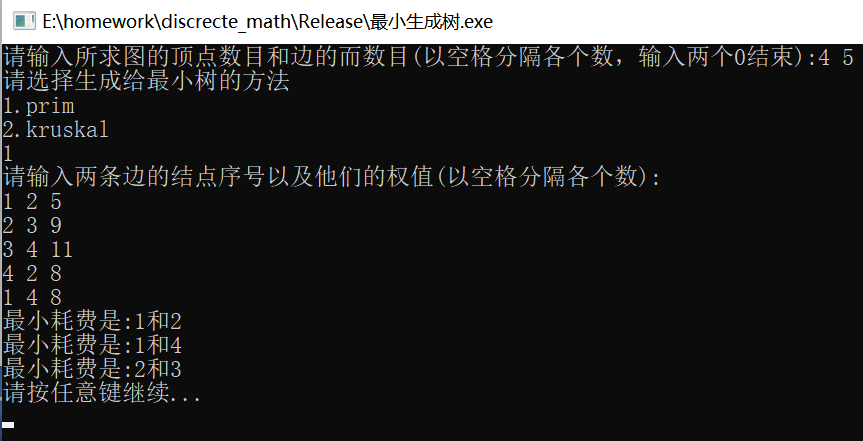
5.3 输入边数为0



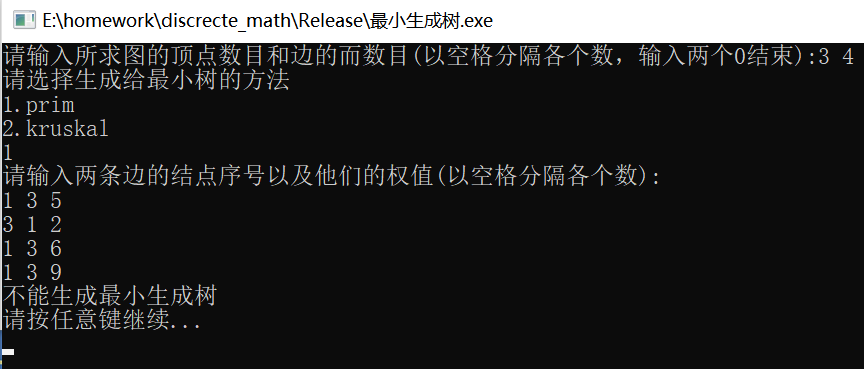
5.4 输入边数过少



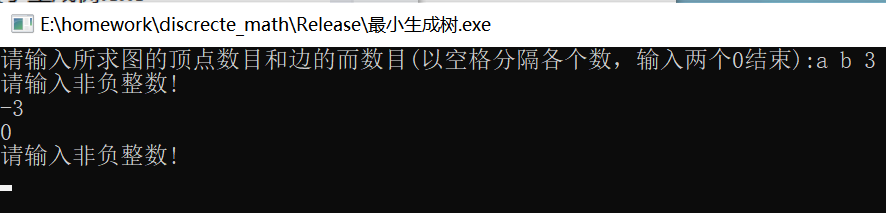
5.5 输入平行边

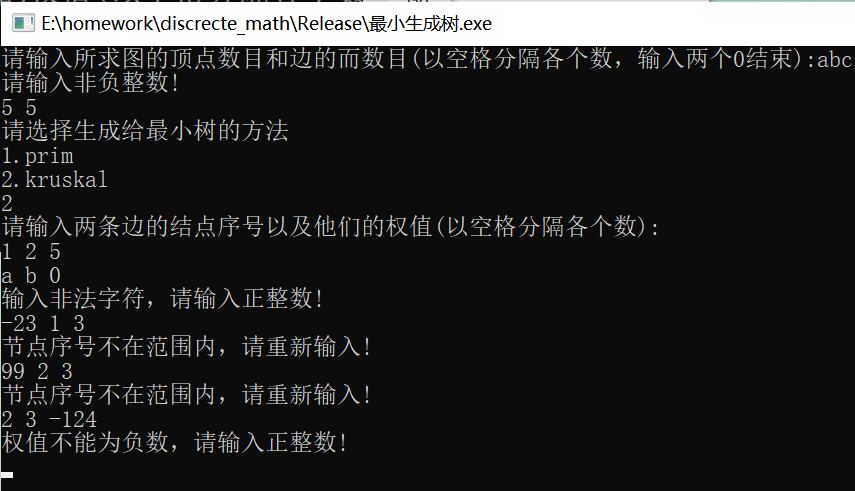


5.6 输入非连通图



5.7 输入顶点和边数量错误



5.8 输入边错误

6 心得体会

通过对本题目的分析与解答，我加深了对最小生成树的认识与理解，同时对于如何求解最小生成树的算法加深了一定的掌握，其中包括prim算法和kruskal算法。而对于两个算法，由从中引申出更多数据结构的设计，包括vector向量、heap堆、priority\_queue优先队列等等，更加加深了我对C++stl库中的容器的认识。

对于kruskal算法的分析：首先是要想每次都取出最小的边的同时保持较小的时间复杂度，一般的方法是在一个数组中进行n次的搜索，每次搜索都需要遍历整个数组，时间复杂度达到了O(e2)级别。我们应该对此进行优化的思考，利用已经学过的数据结构分析，于是很容易想到使用堆这种数据结构。最小堆的堆顶一定是堆中最小的元素，因此每次取堆中最小的元素，进行调整，再利用堆是完全二叉树的性质，这样每次搜索得到最小的元素的时间复杂度可以降低到O(log2e)级别，进行e次的出堆操作，时间复杂度相对来说，对于较大的数据量，确实可以由很大的提升。其次是对于并查集的应用，判断两个顶点是否处在于同一个连通分量，在数据结构课程中，这几乎和kruskal算法是一对“孪生兄弟”，因此不难想到这点。

对于prim算法，在编写的过程中，主要的难点也是对于算法的优化。一般的prim算法的时间复杂度为O(n2)级别的，原因在于进行n - 1趟的过程中，每一趟都要进行对于与某一个顶点相关联的所有结点中找到最小的一个，其时间复杂度为O(n)，因此总体的时间复杂度为O(n2)。与上述思路相同，我们可以在这里利用堆优化寻找最小元素的过程，由此降低时间复杂度。

同时对于类似本题目有着大量用户输入的操作，我们应该保持对输入非法的警惕，尽量地将用户输入的所有情况都考虑到，进行输入错误的提示输出，不至于让程序崩溃。我们也应该在此过程中养成这样良好的编程习惯，虽然这只是一个小程序，但对于日后工作接触大的项目来说有着重要的意义。