项目说明文档

《离散数学》课程实验报告

——warshall算法

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 传递闭包](#_Toc495668163) 4

[1.1.2 Warshall算法](#_Toc495668163) 4

[1.2 题目概述](#_Toc495668155) 4

[1.3 程序功能](#_Toc495668154) 4

[2 解题思路](#_Toc495668156) 5

[3 数据结构设计](#_Toc495668161) 5

[3.1 向量](#_Toc495668162) 5

[3.1.1 说明](#_Toc495668167) 5

[3.1.2 UML图](#_Toc495668167) 5

[3.2 Vector游标类 6](#_Toc495668166)

[3.2.1 说明 6](#_Toc495668167)

[3.2.2 UML图](#_Toc495668167) 6

[3.3 bitset类](#_Toc495668170) 6

[3.3.1 说明](#_Toc495668167) 6

[3.3.2 UML图](#_Toc495668167) 7

[3.4 哈希表](#_Toc495668170) 7

[3.4.1 说明](#_Toc495668167) 7

[3.4.2 UML图](#_Toc495668167) 8

[3.5 String类](#_Toc495668170) 8

[3.5.1 说明](#_Toc495668167) 8

[3.5.2 UML图](#_Toc495668167) 9

[3.6 求解传递闭包类](#_Toc495668170) 9

[3.6.1 说明](#_Toc495668167) 9

[3.6.2 UML图](#_Toc495668167) 10

[4 核心算法](#_Toc495668161) 10

[4.1 warshall算法二维数组实现](#_Toc495668174) 10

[4.1.1 说明](#_Toc495668176) 10

[4.1.2 主要代码](#_Toc495668176) 11

[4.2 warshall算法位向量实现](#_Toc495668174) 12

[4.2.1 说明](#_Toc495668176) 12

[4.2.2 主要代码](#_Toc495668176) 12

[4.3 输出传递闭包](#_Toc495668174) 12

[4.3.1 说明](#_Toc495668176) 12

[4.3.2 主要代码](#_Toc495668176) 12

[4.4 算法选择](#_Toc495668174) 13

[4.4.1 说明](#_Toc495668176) 13

[4.4.2 主要代码](#_Toc495668176) 14

[4.5 初始化集合](#_Toc495668174) 14

[4.5.1 说明](#_Toc495668176) 14

[4.5.2 主要代码](#_Toc495668176) 15

[4.6 二元关系初始化](#_Toc495668174) 17

[4.6.1 流程图](#_Toc495668176) 17

[4.6.2 说明](#_Toc495668176) 18

[4.6.3 主要代码](#_Toc495668176) 18

[4.7 对外接口函数](#_Toc495668174) 19

[4.7.1 说明](#_Toc495668176) 19

[4.7.2 主要代码](#_Toc495668176) 20

[5 实验结果](#_Toc495668161) 20

[5.1 正确测试](#_Toc495668174) 20

[5.2 集合元素名称多字符](#_Toc495668174) 21

[5.3 结合元素个数输入错误](#_Toc495668174) 21

[5.4 有序对个数输入错误](#_Toc495668174) 22

[5.5 有序对输入重复](#_Toc495668174) 22

[5.6 有序对输入不在范围](#_Toc495668174) 23

[6 心得体会](#_Toc495668161) 23

1 题目简介

* 1. 题目原理背景
     1. 传递闭包

传递闭包是对现有关系添加尽量少的有序对，使其具有传递性。

传递闭包具有这样的性质：传递闭包：t(R)∘t(R)t(R)。

根据传递闭包的求解公式t(R) = RR2R3…Rn，我们需要对矩阵进行n次的矩阵乘法，需要注意的一点是：这里为了保证矩阵中只能出现0或1两种元素，乘法过程中使用的是逻辑加。具体过程表现为，假设两个矩阵A和B相乘得到矩阵C，C的第i行第j列的值为A的第i行的所有元素和对应的B的第j列的素有元素相乘后进行逻辑加的结果。该成算法的是时间复杂度为O(n4)，实在是过于庞大，因此有科学家提出了下面的优化。

* + 1. Warshall算法

Warshall在1962年提出了一种求传递闭包的的时间空间复杂度更低的Warshall算法。将传统算法的时间复杂度降低至O(n3)，相比与之前的O(n4)，本算法具有更高的时间效率，在实际问题应用中具有更好的效果，具有重大的实际意义。

* 1. 题目概述

求解用户给出的关系的传递闭包，并将其按照集合的形式输出。这里需要我们注意的是：其一，给出的程序代码中运行时候，会出现警告说明，从"R"中读取的数据无效：可读大小为"n\*2"个字节，但可能读取了"4"个字节。其二，R 的传递闭包作为一个集合的最后一个有序对的后面还有一个逗号，这不符合传递闭包的表现形式。其三，给出程序还不够健壮，不按规则输入有可能造成程序崩溃。

我们需要修复并完善给出代码，思考优化方法。

* 1. 程序功能

由用户输入集合A中元素的个数，然后根据用户输入的数字，输入集合中相应个数的元素。然后由用户输入二元关系R中的元素个数，根据输入的元素个数输入有序对，形式为两个元素名称之间用空格分隔，连个有序对之间用回车分隔。最后以集合的形式输出求解得出的传递闭包。

2 解题思路

利用warshall算法进行求解。其主要思想就是对于每一个顶点，将其作为“中转”的结点k，然后对所有情况i结点和j结点(其中0 <= i,j < n，n为集合A中的元素个数)进行枚举，检查i结点是否能通过k结点到达j。若可以则检查该关系中是否还不存在<i, j>有序对，若不存在则进行添加。重复上述过程，0 <= k < n，就可以求解得出传递闭包。

该算法设计ｎ个中转结点的遍历，又对每个中转节点整个二维数组进行遍历，因此时间复杂度为O(n3)。

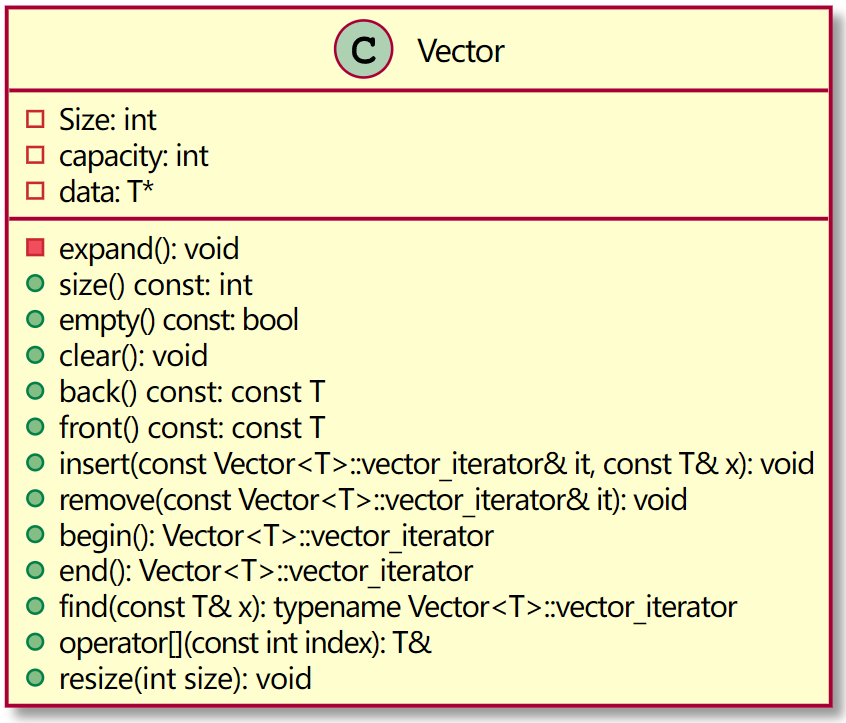
3 数据结构设计

3.1 向量

3.1.1 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)

3.1.2 UML图

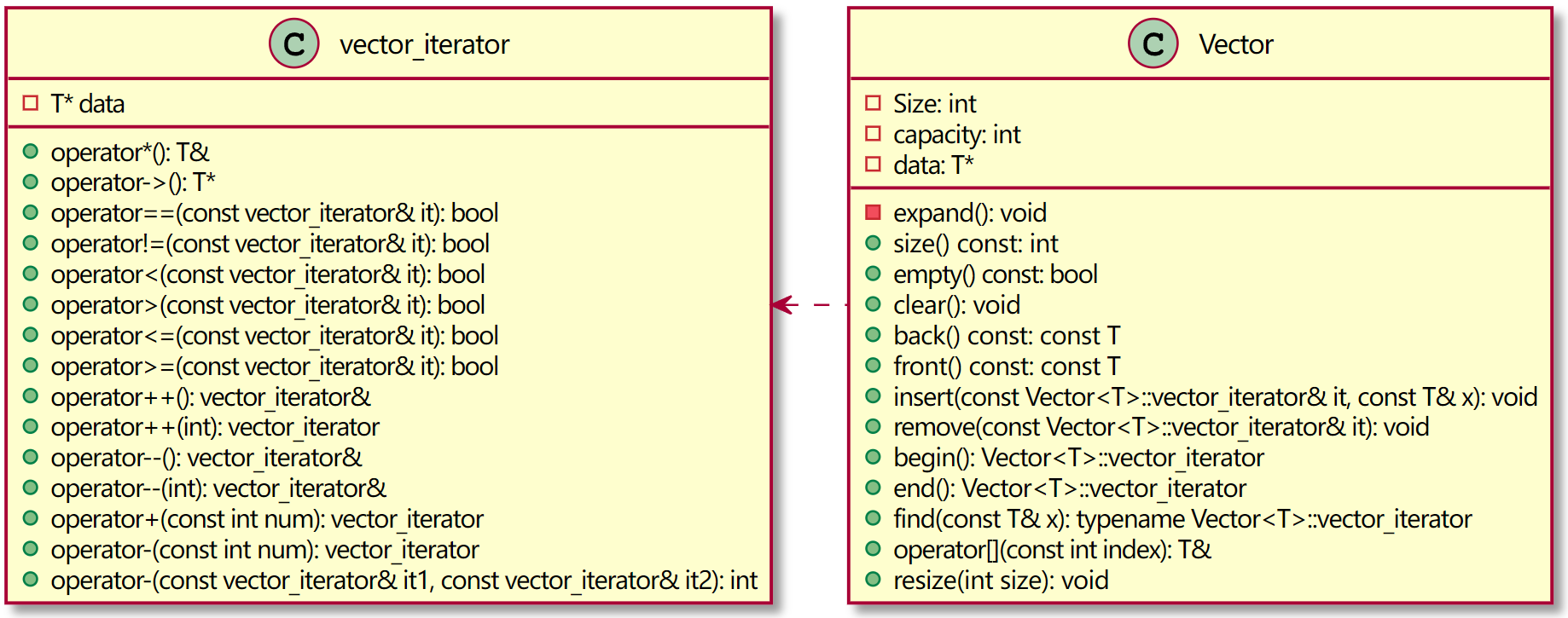


3.2 Vector游标类

3.2.1 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。

3.2.2 UML图



3.3 bitset类

3.3.1 说明

位向量是一种仅仅包含0和1的数组。长度为m的位向量所占空间要比包含m个空间的数组少得多。其原因在于这是一种二进制状态压缩，它是指将一个长度为m的bool数组用一个n位的二进制整数表示并存储的方法。其中C++的STL库中的bitset正是对此的最好解释。Bitset可以看作一个多为二进制数，每8位占用1个字节，相当于采用了状态压缩的二进制数组，并支持基本的位运算。在估计程序运行时间的同时，我们一般以32位整数的运算次数位基准，因此n位bitset执行一次位运算的复杂度可以视为n/32。

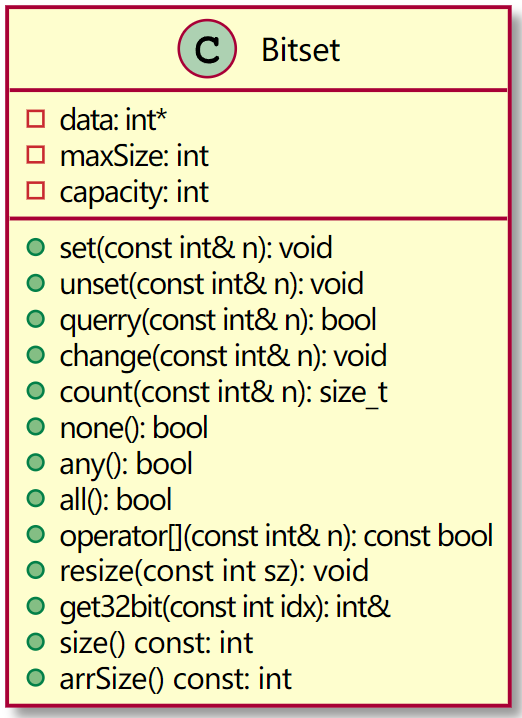
以上的复杂度指的不仅仅是时间复杂度还有空间复杂度，在充分利用小空间存储大量数据方面非常具有优势。

这里利用int数组以及相关位运算实现相关操作。一个int整型变量所占用的空间是32位，因此可以存储32个0/1的信息。

其中数据成员包括整型变量的指针data，用于作为动态申请空间的首地址；整型变量的maxSize表示当前最大容量以及整型变量capacity申请的int的空间大小。

成员函数包括将第n位置为0或1；查询第n位的状态；将第n位翻转；查询有多少元素的值位1等等。

3.3.2 UML图



3.4 哈希表

3.4.1 说明

哈希表又称散列表，是根据关键码而进行直接访问的数据结构，它通过把关键码利用哈希函数映射到表中的索引进行访问来加快查找速度。其中插入的元素通常是关键码和值形成的有序对，又称为键值对分别用key，value两个英文字母表示。

若关键字为k，则其值存放在f(k)的存储位置上。由此，不需比较便可直接取得所查记录。称这个对应关系f为散列函数，按这个思想建立的表为散列表。

对不同的关键字可能得到同一散列地址，即k1≠k2，而f(k1)==f(k2)，这种现象称为冲突。具有相同函数值的关键字对该散列函数来说称做同义词。综上所述，根据散列函数f(k)和处理冲突的方法将一组关键字映射到一个有限的连续的地址集（区间）上，并以关键字在地址集中的“像”作为记录在表中的存储位置，这种表便称为散列表，这一映射过程称为散列造表或散列，所得的存储位置称散列地址。

若对于关键字集合中的任一个关键字，经散列函数映象到地址集合中任何一个地址的概率是相等的，则称此类散列函数为均匀散列函数，这就是使关键字经过散列函数得到一个“随机的地址”，从而减少冲突。

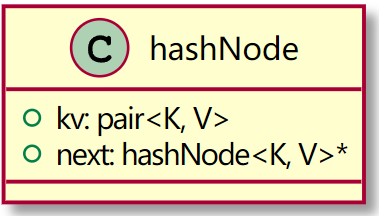
哈希函数能是对一个数据序列的访问过程更加迅速有效、通过散列函数，数据元素将被更快的定位。实际工作中我们需要是不同的情况采用不同的哈希函数，需要考虑的因素又：计算哈希函数所需要的时间、关键字的长度、哈希表的大小、关键字的分布情况以及记录的查找频率。

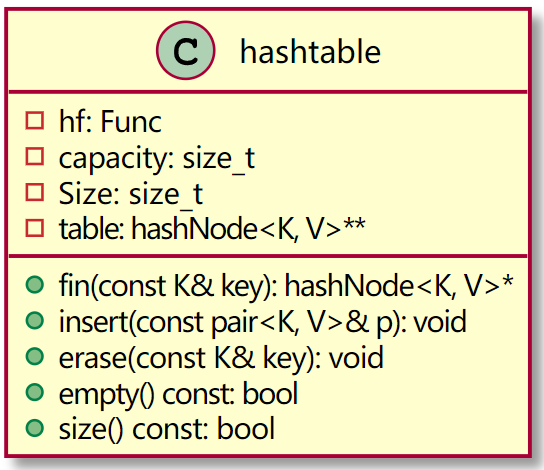
影响哈希冲突的因素主要有三个：哈希函数是否均匀、处理冲突的方法以及哈希表的装填因子。

这里处理冲突的方法是链地址法，即每个哈希函数对应一条链表，出现冲突则构建结点插入链表头。

同时哈希表的装填因子定义为：α= 填入表中的元素个数/哈希表的长度。这里判断装填因子α> 0.7时候进行哈希表的扩容。

3.4.2 UML图



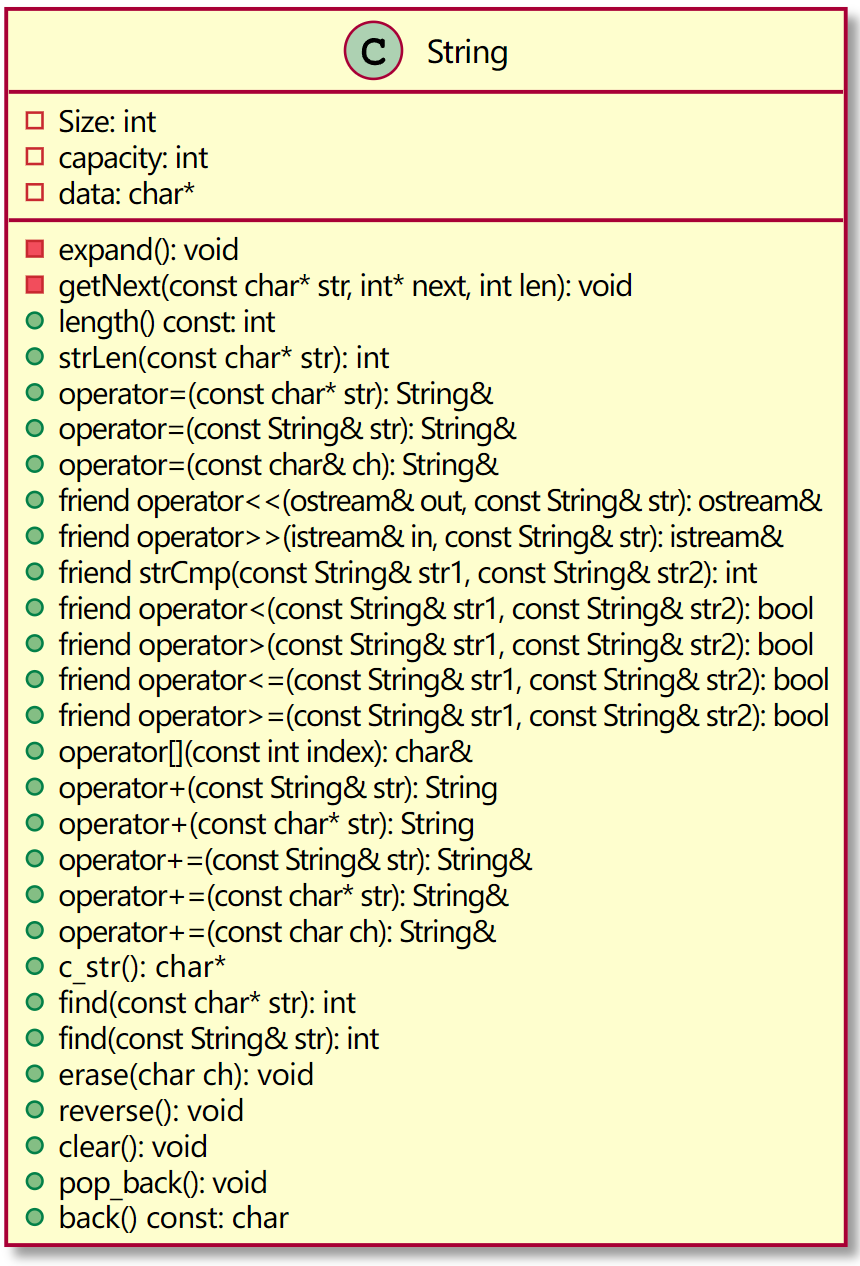


3.5 String类

3.5.1 说明

某种意义上来说，string其实不属于STL，它出现要更早。但这里我们主要讨论对string类部分功能的实现。字符串作为使用频率较高的一种数据结构，在程序发挥着巨大的作用。C++string类中配套的多种函数方便我们对字符串的处理，例如：比较、连接、替换、搜索等等。将其完全实现实现任务艰巨。因此我们实现其中的主要功能，以完成对主要的程序的要求。

3.5.2 UML图



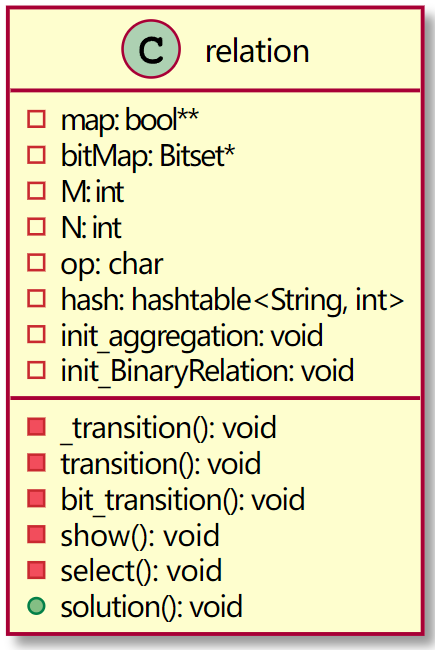
3.6 求解传递闭包

3.6.1 说明

我们将解决本题目的方法包装成一个类，其中对外接口实现了用户输入集合和关系，然后利用warshall算法求解出传递闭包并输出。其中数据成员包括bool\*\*类型的存储关系矩阵的二维数组首地址map、利用位向量存储关系矩阵的二维数组手地址Bitset\* bitMap、int整型变量的矩阵行列数M、int整型变量的二元关系中的有序对个数N、char字符类型记录用户输入的选择的算法op、hashtable<String, int> hash记录集合中元素以及对应的编号以及String\*类型的Set作为记录集合中元素名称的数组首地址。

成员函数包括仿照floyd求多源最短路径求取传递闭包的函数、warshall算法求取传递闭包函数、位向量求传递闭包函数、展示传递闭包集合的函数、算法选择函数、初始化集合函数、初始化二元关系函数以及对外接口函数。

3.6.2 UML图



4 核心算法

4.1 warshall算法二维数组实现

4.1.1 说明

这里我们发现，warshall算法的实现过程和图论当中求多源最短路径的floyd算法整体上类似，进一步学习之后，我们发现其全称确实叫做floyd-warshall算法。其中利用floyd算法主要思想，其主要思路为：穷举每个结点通过另一个节点是否能到达第三个不同的结点。最外层循环第k个结点，里面两层循环表示想要从第i个结点去到第j个结点，当可以从第i个结点去到第k个结点并且可以从第k个结点去到第j个结点的时候，便可以实现，在数组map[i][j]处置为1。即为存在<i, k>和<k, j>有序对，若<i, j>有序对不存在，则说明需要添加。

接着是沃舍尔算法，这个算法比前面的标准算法复杂度上少了一阶。沃舍尔算法使用了一条路径的“内点”的概念。即如a，b，c，.....，x样的一条路径，除去a和x之外的所有端点称为这条路径的内点。具体的操作方法是以R为开头构造一系列（n个）矩阵，他们是W0,W1,W2,W3,W4 .....，其中W0= MR。这看起来和标准算法差不多，但是沃舍尔算法的高阶矩阵的值并不是由前一阶的布尔幂运算得来。而是这样定义的：Wk=[wij[k]]，如果存在一条从vi到vj的路径且这条路径的所有内点都在集合{v1，v2，....，vk}（表中的前k个顶点）内，那么wij[k]=1，否则为0.（注意有两个合取的条件）。

可以看到，因为Wk-1中的“1”也必然满足于Wk，所以Wk-1中的“1”可以直接继承到Wk之中，即wij[k-1]=1→wij[k]=1。而对于Wk-1中的0，从前段的定义中我们可以看到，对于wij[k-1]为0，而wij[k]为1的情况，当且仅当wik[k-1]=1且wkj[k-1]=1。（或者使用路径的说法——存在一条从vi到vk和一条从vk到vj的路径，使得当内点集合扩大到vk时才满足了前段定义的第二条条件）注意这里的下标k，不再泛指k≤n的正整数，而是实指当前正在运算的Wk的k。

总结来说，标准算法是通过n阶布尔幂来化简多重传递，并在过程中进行了矩阵合并的运算以保证所有路径都被保留；而沃舍尔算法在每阶矩阵的每个元素上，最多只进行2次布尔运算就可以取代标准算法的一个复杂度为O(n)的矩阵运算，因此效率得以提升。

4.1.2 主要代码

void \_transition() {

for (int k = 0; k < M; ++k)

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

for (int j = 0; j < M; ++j)

if (map[i][k] && map[k][j])

map[i][j] = 1;

}

void transition() {

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

for (int j = 0; j < M; ++j)

{

if (i == j)

continue;

if (map[j][i])//存在j -> i的路径

for (int k = 0; k < M; ++k)

map[j][k] |= map[i][k];//逻辑加，j -> i -> k

}

}

4.2 warshall算法位向量实现

4.2.1 说明

这里使用一维的位向量数组代替二维数组。我们都知道计算机中的数据都是以二进制的形式存储的，因此我们可以充分利用每一位的0和1的状态构建关系矩阵。根据之前介绍的位向量，我们可以成批地对每行进行逻辑加而非向上述算法中需要一个一个依次进行，因此总体的时间复杂度为O(n3/32)，这里说明当n较小的时候，该算法优化的效果较好，但当n较大的时候n > 1000时候，优化的效果可能越发不明显，比较这只是常数级别的优化。

4.2.2 主要代码

void bit\_transition() {

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

for (int j = 0; j < M ; ++j)

{

if (i == j)

continue;

if (bitMap[j].querry(i))//第j行逻辑加上第i行

{

int capacity = bitMap[j].arrSize();

for (int k = 0; k < capacity; ++k)

bitMap[j].get32bit(k) |= bitMap[i].get32bit(k);

}

}

}

4.3 输出传递闭包

4.3.1 说明

为了解决题目中出现的问题，我们在这里设置一个标志位，标志是否是第一个输出的有序对。当第一个有序对输出完成后，我们再以逗号和有序对的形式循环输出。

4.3.2 主要代码

void show() {

//标记是否是第一个输出的关系对

bool firstOutput = false;

cout << "R的传递闭包(集合形式)为:" << endl;

cout << "t(R) = {";

if (op == '1' || op == '3')//用二维矩阵求解

{

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

for (int j = 0; j < M; ++j)

if (map[i][j])

{

if (firstOutput)

cout << ",";

else

firstOutput = true;

cout << "<" << Set[i] << "," << Set[j] << ">";

}

}

else if (op == '2')//由位向量求解

{

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

{

for (int j = 0; j < M; ++j)

if (bitMap[i].querry(j))

{

if (firstOutput)

cout << ",";

else

firstOutput = true;

cout << "<" << Set[i] << "," << Set[j] << ">";

}

}

}

cout << "}" << endl;

}

4.4 算法选择

4.4.1 说明

这里给出提示，有用户选择对应的算法，然后执行相应的函数。同时我们需要对输入进行判断。若输入错误需要清除错误标记和跳过缓冲区中的错误，并输出输入错误提示，然后才能重新输入。这里对输入的char类型的变量op进行switch分支判断，若是’1’则执行warshall算法；若是’2’则执行利用位向量的warshall算法；若是’3’则退出程序。

4.4.2 主要代码

void select() {

op = '\0';

bool loop = true;

while (loop)

{

cout << endl;

cout << "请输入对应序号选择算法" << endl;

cout << "1:warshall算法" << endl;

cout << "2:利用bitset的warshall算法" << endl;

cout << "3:退出" << endl;

cin >> op;

switch (op) {

case '1':

this->transition();

this->show();

break;

case '2':

this->bit\_transition();

this->show();

break;

case '3':

loop = false;

break;

default:

cout << "输入错误操作，请重新输入!" << endl;

cin.clear();

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

break;

}

}

}

4.5 初始化集合

4.5.1 说明

首先由用户输入集合中元素的个数，然后根据个数进行元素的名称的输入，对集合进行初始化。这里需要注意的是对个数输入的判断，利用while永真循环，判断输入是否合法以及是否处于正确范围，若不是则输出错误提示，由用户重新输入，反之跳出循环。同时需要注意的还有我们利用String的实例化对象保存元素的名称，因此这里需要根据个数动态申请String类型的数组存储一系列元素的名称。

同时，根据个数输入元素名称的同时，我们需要判断名称是否出现重复，因此仍然利用哈希表进行快速搜索，时间复杂度为O(1)，大大提高了程序的循行速度，优化的输入的算法。

4.5.2 主要代码

void init\_aggregation()

{

cout << "请输入集合 A 中的元素个数(正整数)，按回车键输入下一项：" << endl;

while (1)

{

cin >> M;

if (cin.fail() || M <= 0)

{

cout << "输入有误，请重新输入!" << endl;

cin.clear();

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

}

else

break;

}

//为集合申请空间

Set = new(nothrow) String[M];

if (Set == NULL)

{

cerr << "申请空间失败" << endl;

exit(1);

}

cout << "请依次输入集合A中的" << M;

cout << "个元素(形如：abcd......这样的格式)，按回车键输入下一项：" << endl;

//输入集合中元素的名字

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

{

cin >> Set[i];

if (hash.find(Set[i]) == NULL)

hash.insert(make\_pair(Set[i], i));

else

{

cout << "集合中已有该元素，请重新输入!" << endl;

--i;

}

}

//为关系申请空间

map = new(nothrow) bool\* [M];

if (map == NULL)

{

cerr << "申请空间失败" << endl;

exit(1);

}

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

map[i] = new(nothrow) bool[M];

if (map[i] == NULL) {

cerr << "申请空间失败" << endl;

exit(1);

}

}

//初始化map

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

for (int j = 0; j < M; ++j)

map[i][j] = false;

//bitset设置大小

bitMap = new(nothrow) Bitset[M];

if (bitMap == NULL)

{

cerr << "申请空间失败" << endl;

exit(1);

}

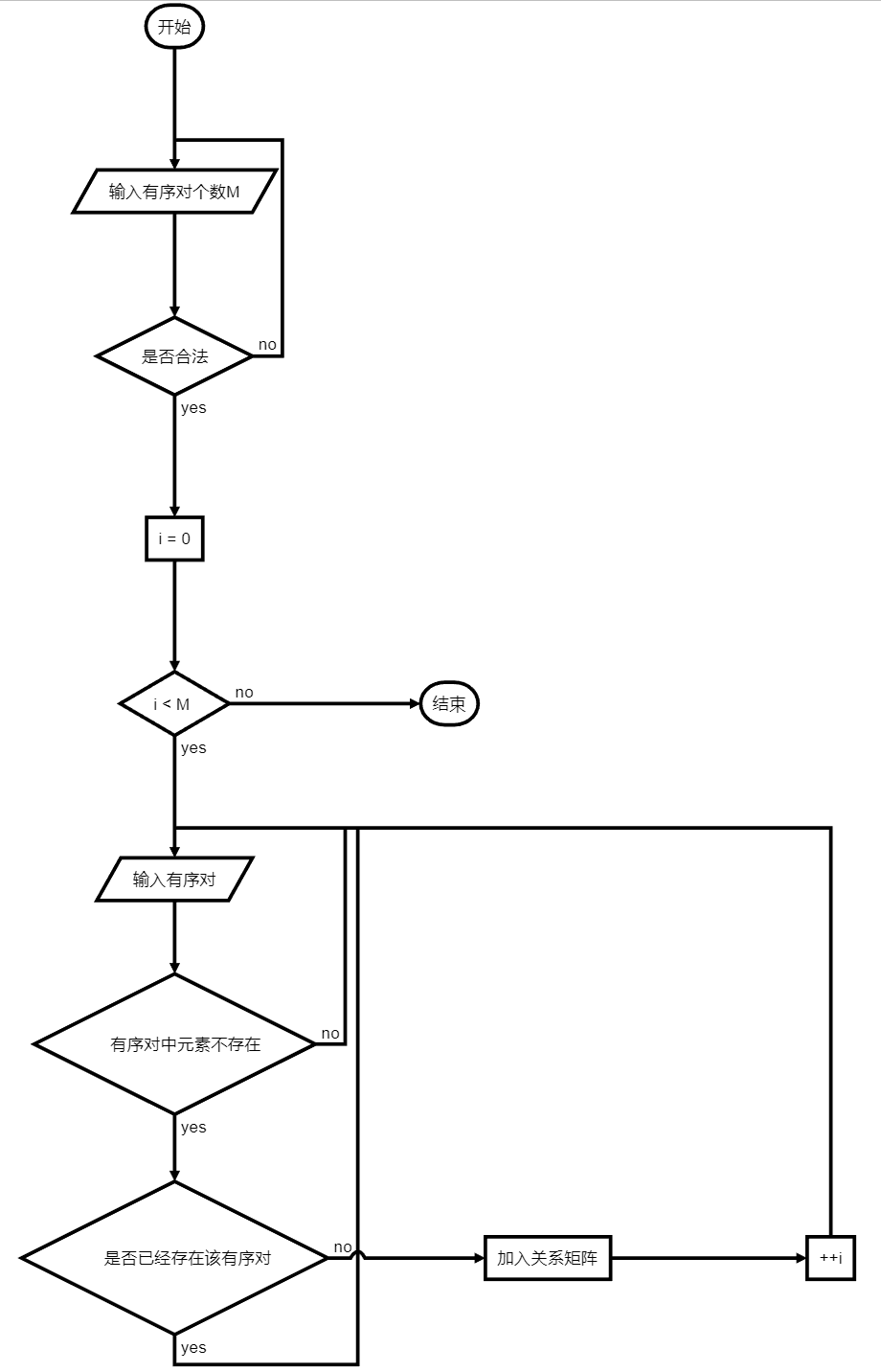
for (int i = 0; i < M; ++i)

bitMap[i].resize(M);

}

4.6 二元关系初始化

4.6.1 流程图



4.6.2 说明

首先由用户输入二元关系中的有序对个数，然后根据个数进行有序对的输入，对二元关系进行初始化。这里需要注意的是对个数输入的判断，利用while永真循环，判断输入是否合法以及是否处于正确范围，若不是则输出错误提示，由用户重新输入，反之跳出循环。

同理，有序对的输入也需要考虑输入的集合元素名称是否处于集合中，这里利用哈希表形成对String元素名称对在数组中存放的索引int的映射，由此实现快速查找，平均的时间复杂度为O(1)。其次，我们需要判断输入的有序对先前是否已经输入，因此在插入二维数组所代表的关系矩阵之后，确认该处值为0的时候才能对该有序对进行添加，否则都会输出输入错误的相关提示，请用户重新输入。

4.6.3 主要代码

void init\_BinaryRelation()

{

//计算二元关系个数上限

int Max = M \* M;

cout << "请输入二元关系 R 中的元素个数(正整数)，按回车键输入下一项：" << endl;

while (1)

{

cin >> N;

if (cin.fail() || N <= 0 || N > Max)

{

cout << "请输入范围在(0," << Max << ")" << "的正整数" << endl;

cin.clear();

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

}

else

break;

}

cout << "请依次输入 R 中的" << M << "个元素，一行是一个元素" << endl;

cout << "(形如：" << endl << "a b" << endl;

cout << "b c" << endl;

cout << "c d" << endl;

cout << "......" << endl;

cout << "这样的格式)，按回车键输入下一项：" << endl;

String temp1, temp2;

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

{

cin >> temp1 >> temp2;

auto res1 = hash.find(temp1);

auto res2 = hash.find(temp2);

if (res1 == NULL)//找不到这个元素

{

cout << "集合中没有" << temp1 << "这个元素，请重新输入!" << endl;

--i;

}

else if (res2 == NULL)

{

cout << "集合中没有" << temp2 << "这个元素，请重新输入!" << endl;

--i;

}

else//两个元素都属于集合中

{

if (map[res1->kv.second][res2->kv.second])//已经驶入了这个关系

{

cout << "已经输入这个关系，请重新输入!" << endl;

--i;

}

else//将关系记入关系矩阵

{

bitMap[res1->kv.second].set(res2->kv.second);

map[res1->kv.second][res2->kv.second] = 1;

}

}

}

}

4.7 对外接口函数

4.7.1 说明

经过类的封装，将本题目求解的方法包装在对外接口函数中，其中包括有用户输入进行集合的初始化以及二元关系的初始化，然后调用算法选择函数，根据用户的输入执行相应的函数，最后输出传递闭包的求解结果。

4.7.2 主要代码

void solution()

{

init\_aggregation();

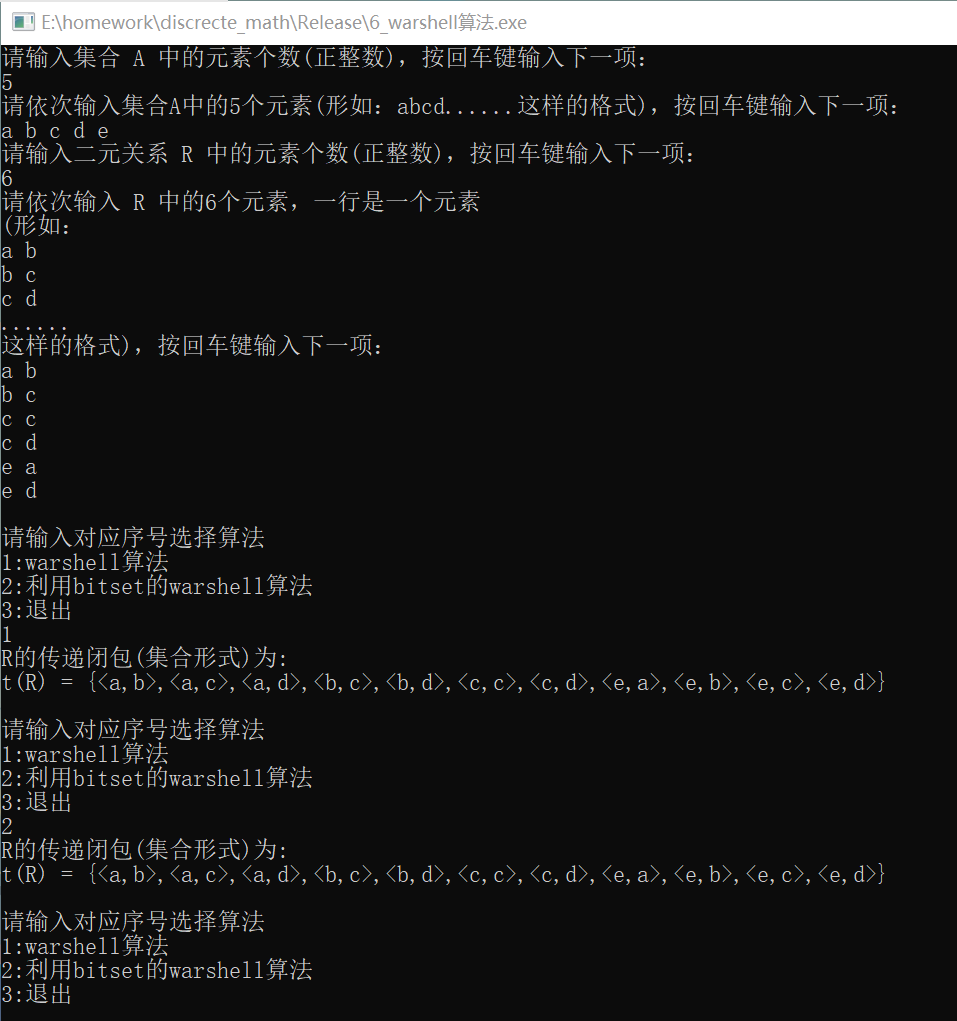
init\_BinaryRelation();

select();

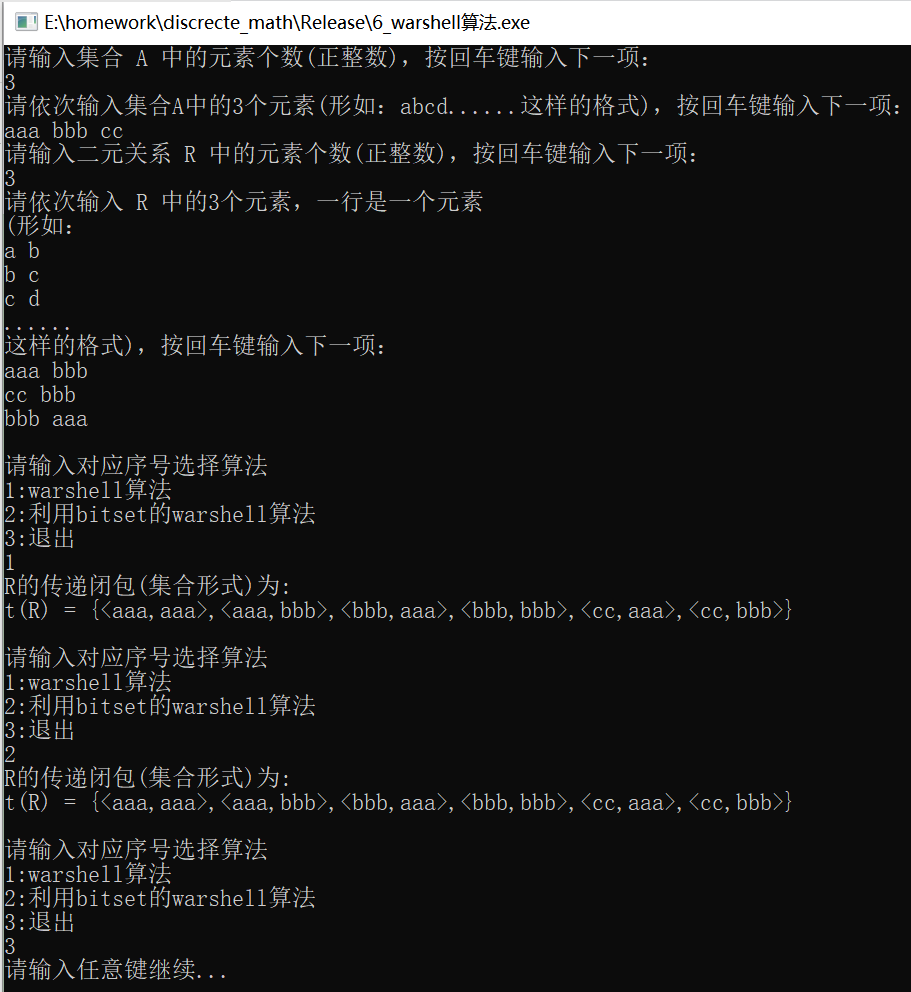
}

5 实验结果

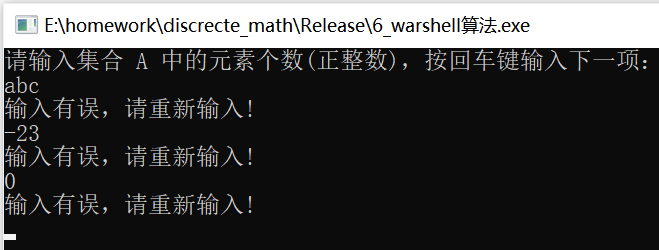
5.1 正确测试



5.2 集合元素名称多字符



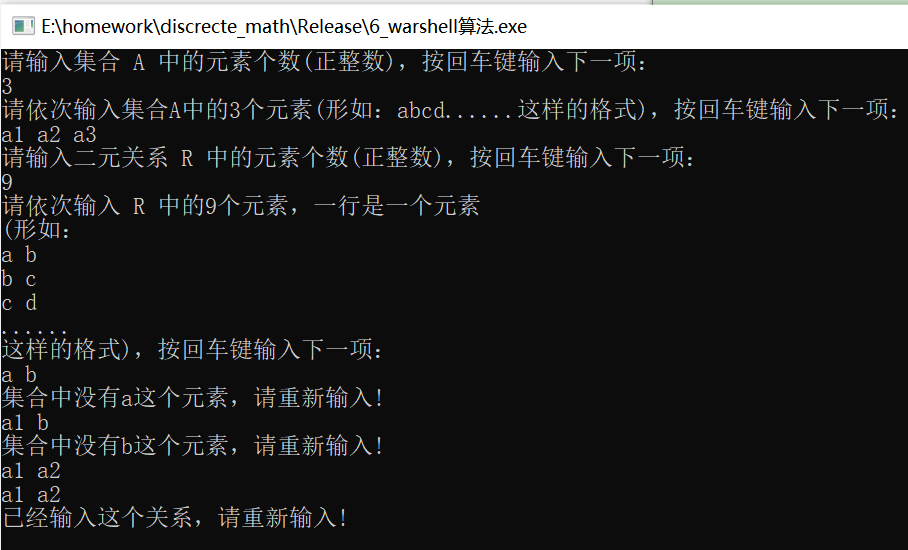
5.3 集合元素个数输入错误



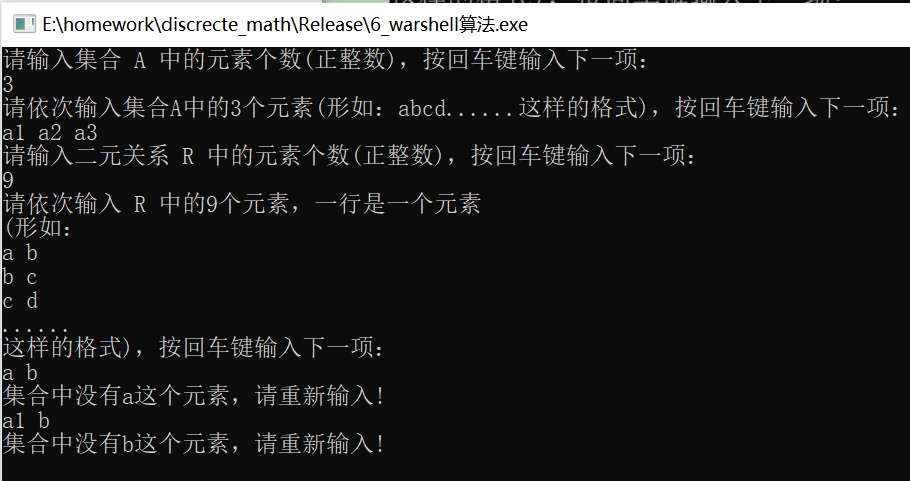
5.4 有序对个数输入错误



5.5 有序对输入重复



5.6 有序对输入不在范围



6 心得体会

通过对warshall算法求解传递闭包，将时间复杂度从传统算法的O(n4)减少到了O(n3)，再通过位向量优化，将算法优化至O(n3/32)，一步步地我体会到了算法优化的过程，利用已有的思想或者方法，应用于新的领域，是一种开创也是一种继承。

对于本题目来说，还有一个特点就是需要对已输入的数据进行多次查找，思考满足该功能并具有较好的时间复杂度的数据结构应该是哈希表，这是一种利用空间换时间的数据结构，由此得到快速的查找搜索。

通过对哈希表类和位向量类的实现，我又加深了对算法与数据结构的理解。将其应用于求解数学问题上是一个巨大的进步。同时，我也进一步理解了warshall算法与图论中的floyd算法的关联，两者利用了相同的思想，作用出不同的表现形式。因此在解决实际问题的时候，我们可以多思考已有的思想，应用于不同的方面。