Warshall算法

3月14日17:12

假设十个点，1,2,3.....

现在假定寻找从4到7的一条可能路径，并输出路线。

V4到V7，k遍历：从1到10：

K=1，只能直接从4到7 or 4,1,7

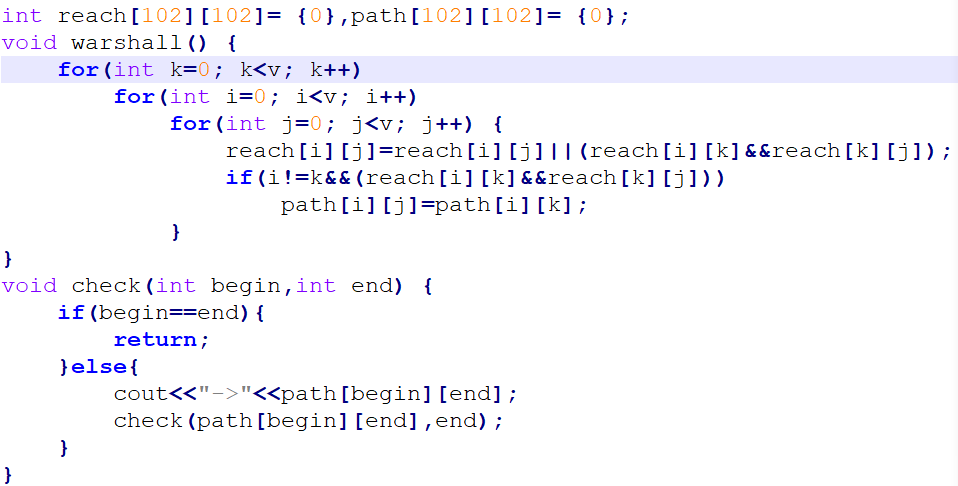
K=2, 4,7 4,1,2,7 4,2,1,7 4,1,7 4,2,7（利用的不是4到7的信息，是4到1和1到7的信息）

K=3， 4,3,7：从4到3的过程中，可以经过1与2，从3到7亦然。因此，你需要连接2,3或者1,3 etc。这也就是为什么判定中：如果第i行第j列！=1（k和前面的点接上了），则将i行与j行相加，并赋值给第i行。

实际上：if B[i][j]!=0；则从i可以到达的点既包括i本身就能到的点，也包括j本身能够到的点（先到j再出发）

故而i行（从i出发能够到的地方）=i行+j行。

设定目标并不能降低复杂度，因为你的最短路径必定要遍历全图，把传递闭包求完了才能判断路径。所以必定需要将整个传递闭包求解完全。



Path[i][j]记录的是从i到j，i之后的那个点

如果本身有边，那么path[i][j]=i，其他都存储-1，path

warshall的三重循环实际上是一个不断找中介点的过程，比如说你找从i到j的路，然后给中间找“桥”，这就是最外层循环是k的原因。所以你可以定义一个Path数组，记录从i到j的下一个桥是谁，然后递归，输出答案。那个warshall函数就是在存桥。

趁着你睡觉，我可以把一些值得你去思考的地方告诉你。就比如说你已经学过了复杂度为Theta(n^2)的Dijkstra算法(适用于没有负权回路的情况)，我现在告诉你复杂度为Theta(n^3)的Warshall算法同样可以用于求解图的连通性和最短路。那么n^3比n^2指数多了1，干了哪些额外工作呢？实际上，Warshall的三重循环跑完，整张图的性质都已经清晰了。这道题刚好是一个多次查询问题，它会问你很多pair之间的最短路径，因此明智的想法就是你一次性将它们全部算完，在内存中打一张表。至于如何存储路径，就是刚刚和你探讨的“找桥”的过程，把这个理解到位，才算对这个算法真正有所体悟，否则我问你这三层循环为什么是kij，你可能都不太清楚？如果有时间的话，可以好好思考下，加油！

另外，要搞清楚check函数的递归实现。emmmm，有问题随时欢迎与我交流~，内容不要外传。

一般来说Dijkstra算法接收的参数是vs，ve，然后如果ve的查询结束，函数就结束，返回最小距离。但是如果你不传ve，程序就会到所有点都被访问完后结束。

此时就是固定起点。对于稠密图，warshall比Dijkstra快的多。ford算法编程实现比Dijkstra简单。

简单说下为什么稠密图的时候warshall比n次Dijkstra快。先从直观上看，warshall是迭代实现，Dijkstra则是递归实现。一般来说，如果两个算法本身都是同级别的，比如你分别用存储增强算法和动态规划算法计算斐波那契数列，复杂度都是O(n)，只是一个递归，一个迭代，那么循环要比递归优越一点。这在于递归需要不断地调用自己，其实类似于go-to语句。而且递归还有爆栈的风险(递归相当于不断地压栈和退栈)。

你可以理解为函数调用也需要时间，但是更进一步的话，它涉及到计算机的底层，它有一个缓存命中的问题。数组和向量(vector)在内存中是连续存储的，如果你按行读入它，缓存的命中率很高，因此也更快。简单来说，就是访问缓存的速度要远远快于访问内存。

但是这些都不本质，稠密图就是这个图连接的比较复杂，点与点之间边很多，Dijkstra算法相当于是每到一个点更新它周围的邻接点的距离，如果你边很多，你就要访问很多次。

这时候你还不如用warshall直接逐个点进行运算。

不高深，你会快就很懂的，都是必须掌握的知识，我只是把你后面需要用到的部分东西先提前告诉你。

可以先记个结论，以后慢慢来，先掌握好课上讲的。你到贵系那边，它们有些课程比较硬，还得一步一步来。

**Vector**

**4月18日7:46**

## 1. vector：

#### 1.1 vector 说明

* vector是向量类型，可以容纳许多类型的数据，因此也被称为容器
* 可以理解为动态数组，是封装好了的类；它能够像容器一样存放各种类型的对象，简单地说，vector是一个能够存放任意类型的动态数组，能够增加和压缩数据。
* 进行vector操作前应添加头文件#include <vector>
* vector在C++标准模板库中的部分内容，它是一个多功能的，能够操作多种数据结构和算法的[模板类](http://baike.baidu.com/view/1923683.htm" \t "https://blog.csdn.net/duan19920101/article/details/_blank)和函数库。

#### 1.2 vector初始化：

**方式1.**

vector<int>a(10);//定义具有10个整型元素的向量（尖括号为元素类型名，它可以是任何合法的数据类型），不具有初值，其值不确定。

**方式2.**

vector<int>a(10,1);//定义具有10个整型元素的向量，且给出的每个元素初值为1

**方式3.**

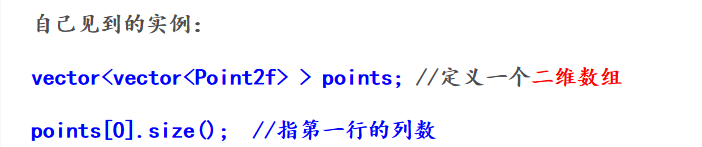
vector<int>a(b);//用向量b给向量a赋值，a的值完全等价于b的值

**方式4**.

vector<int>a(b.begin(),b.begin+3);//将向量b中从0-2（共三个）的元素赋值给a，a的类型为int型

**方式5.**

vector<int> a(b,b+7）; //从数组中获得初值int b[7]={1,2,3,4,5,6,7};



Point2f是它自己实现的一个struct或者class，我猜测是二维坐标。你可以理解为vector里面可以装你定义的任意类。比如你实现了一个类叫做Zcy，就可以构建一个元素为Zcy对象的vector，std::vector<Zcy> base。

为啥很多大佬都不喜欢usingnameapce std？这是，防止和命名空间内的定义重复 redef。std里面装了很多已经定义好的函数，比如可能你会自己定义swap，但其实人家都定义好了的。所以你可以只用一部分，什么叫做只用一部分呢？就是using std::vector ，而不是把整个命名空间都包含进来。

#### 1.3 vector对象的常用内置函数使用（举例说明）

概述如下：

两个向量元素互相赋值；对a的直接赋值；a的头和尾的查询；a的第i个元素；a去尾；a清空；a查看大小；判定a是否为空；a定点（定长度）删除；a定点（定长度）插入；a定点把b数组的一段插入；a的长度与容量；a的元素调整（多退少补）；a的扩容；向量的交换和赋值；

#include<vector>

vector<int> a,b;

a.assign(b.begin(),b.begin()+3);

//b为向量，将b的0-2个元素赋值给向量a

a.assign(4,2);

//a含有4个值为2的元素

a.back();

//返回a的最后一个元素

a.front();

//返回a的第一个元素

a[i];

//返回a的第i元素,当且仅当a存在

a.clear();

//清空a中的元素

a.empty();

//判断a是否为空，空则返回true，非空则返回false

a.pop\_back();

//删除a向量的最后一个元素

a.erase(a.begin()+1,a.begin()+3);

//删除a中第一个（从第0个算起）到第二个元素，也就是说删除的元素从a.begin()+1算起（包括它）一直到a.begin()+3（不包括它）结束

//删除之后自动左移吗？

会，但值得一提的是 vector的insert和erase慢的离谱。

a.push\_back(5);

//在a的最后一个向量后插入一个元素，其值为5

a.insert(a.begin()+1,5);

//在a的第一个元素（从第0个算起）位置插入数值5

a.insert(a.begin()+1,3,5);

//在a的第一个元素（从第0个算起）位置插入3个数，其值都为5

a.insert(a.begin()+1,b+3,b+6);

//b为数组，在a的第一个元素（从第0个元素算起）的位置插入b的第三个元素到第5个元素（不包括b+6）

a.size();

//返回a中元素的个数

a.capacity();

//返回a在内存中总共可以容纳的元素个数

//长度和容量的区别

1.vector是一个动态数组，除了描述元素个数的size之外，vector内部还有一个描述容量变量，姑且称为capacity，它表示当前vector能存储多少元素。vector占用的内存大小不是size而是capacity，当元素个数大于capacity时，vector会自动增大capacity，也就是重新申请一块更大的内存空间。除了这种自动增大以外，你也可以用reserve函数来向vector传达你希望它的capacity是多大(注意是“希望”，在某些情况下，比如你希望的容量小于vector中已有元素个数时，这个操作无效)，这样能够减少vector重新申请内存的次数，提升一些运行效率。

2.那似乎设置容量就没有意义啊？比如这段代码：#include<vector> vector<int> a,b;看上去没有设置容量，实际上：这个是已经封装好的一个现成的类，已经实现完了（已经预设了容量大小）。暴露给你（用户）的只是接口。但是如果你现在要自己手写个vector（就是这次的oop作业），就得设置容量。你include的是一个人家已经写好的vector类，内部已经把capacity的扩容机制完成了。所以做那个oop题需要自己写扩容机制，但是离散里面用的话直接用就可以了。完全可以用vector代替数组（不是代替链表）。

3.我都预设好了，系统还能给我自动扩充，我要他干啥？

比如你原来的capacity是8，然后你向里面push\_back元素，push到8个了，capacity会自动扩容，C++默认扩容到原来的1.5倍。但是，现在我直接把capacity干到1000，那么当元素个数＜1000时，我就不用扩容。扩容一次花费的时间为O(n)，你可以理解为扩容是要花时间的，我直接给它来个大的，就省事了。

1. 然后不建议把vector的初始容量设置得很大。因为会造成很多资源浪费，1000个位置你只用了100个，利用率太低。vector内部实现每次扩容1.5倍，(1.5)^n 增长很快。要达到1000，用不了多少时间，我控制下在300左右就好了。只用你装的数量合理，别一次往里面装10000000000000000个， 把自己的静态区搞崩了就行。
2. 注意，容量太大不仅利用率低，还有可能会内存泄漏。看你往里面装什么，如果你装的全是指针，记得自己析构它们指向的内存空间。
3. 默认size是0，可以随时resize。

a.resize(10);

//将a的现有元素个数调整至10个，多则删，少则补，其值随机

a.resize(10,2);

//将a的现有元素个数调整至10个，多则删，少则补，其值为2

a.reserve(100);

//将a的容量扩充至100

a.swap(b);

//b为向量，将a中的元素和b中的元素整体交换

a==b;

//b为向量，向量的比较操作还有 !=; >=; >; <=; <

## 2.基本操作vector的几种方式

2.1赋值与添加：

1.向向量a中添加元素

vector<int>a;

for(int i=0;i<10;++i){a.push\_back(i);}

2.从数组中选择元素向向量中添加

int a[6]={1,2,3,4,5,6};

vector<int> b;

for(int i=0;i<=4;++i){b.push\_back(a[i]);}

3.从现有向量中选择元素向向量中添加

int a[6]={1,2,3,4,5,6};

vector<int>b;

vector<int>c(a,a+4);

for(vector<int>::iterator it=c.begin();it<c.end();++it)

{

b.push\_back(\*it);

}

这一节是什么意思？

这段程序先用数组a下标[0,4)的部分构造了一个vector c；之后使用vector的迭代器（iterator）作为循环变量遍历了c，把c中的所有元素依次插入到了空vector b的尾部。

为什么是\*it呢？iterator可以当成容器里元素的指针。

4.从文件中读取元素向向量中添加

ifstream in("data.txt");

vector<int>a;

for(int i;in>>i){a.push\_back(i);}

5.常见错误赋值方式

vector<int>a;

for(int i=0;i<10;++i){a[i]=i;}//下标只能用来获取已经存在的元素

那我如何表达相同的意思？

定义的时候写成vector<int> a(10)；或者定义好a之后加一句，a.resize(10); 这样a[i]就实际存在了。

##### 2.2 从向量中读取元素

### 1.通过下标方式获取

int a[6]={1,2,3,4,5,6};

vector<int>b(a,a+4);

for(int i=0;i<=b.size()-1;++i){cout<<b[i]<<endl;}

### 2.通过迭代器方式读取

int a[6]={1,2,3,4,5,6};

vector<int>b(a,a+4);

for(vector<int>::iterator it=b.begin();it!=b.end();it++){cout<<\*it<<" ";}

**两种读取方式的区别？**

**对vector来说其实效果（和效率）都是一样的。**

**区别的话：第一种本质上是n次使用下标访问了一个随机访问容器（Random Access Container)中的一个位置，你可以把i的枚举顺序随便换成别的。**

**什么叫做RAC，这个Random是相对访问顺序来。有的容器只能顺序访问，比如set和list。RAC可以1234,4321甚至1324。**

**第二种本质上是使用了迭代器正序访问了一个容器的全部元素，你没有办法（轻易的）改变这个访问顺序。特别的，当容器不是vector这种随机访问容器的时候（比如set这种关联容器），第一种遍历方法就无法使用了。**

## 3.几个常用的算法

#include<algorithm>

sort(a.begin(),a.end());

//对a中的从a.begin()（包括它）到a.end()（不包括它）的元素进行从小到大排列

reverse(a.begin(),a.end());

//对a中的从a.begin()（包括它）到a.end()（不包括它）的元素倒置，但不排列，如a中元素为1,3,2,4,倒置后为4,2,3,1

copy(a.begin(),a.end(),b.begin()+1);

//把a中的从a.begin()（包括它）到a.end()（不包括它）的元素复制到b中。从b.begin()+1的位置（包括它）开始复制，覆盖掉原有元素

find(a.begin(),a.end(),10);

//在a中的从a.begin()（包括它）到a.end()（不包括它）的元素中查找10，若存在返回其在向量中的位置