# 一重要词发现

## 1.1主要数据结构

实验四重要词发现主要包含几个类文件以及主文件，即acNode类、closeness类、ELL类、Hash类、LinkList类、LinkMatrix类、LinkThreeDimMatrix类、pageRankList类以及main主函数文件。

### acNode类

这个类的主要数据结构如下所示：

class acNode {  
public:  
 static int m; //树的阶数  
 static int stateNode; // 状态节点数  
 static int stateTransition;//状态转移次数  
 acNode \*\* ptr; // 下一层的指针ptr[0]表示的是上一层的最后一个叶子节点  
 std::string value; //存储叶子节点的字符串  
 int appearCount; //叶子节点的时候表示当前字符串出现的次数  
 int nodeKind; //0表示分支节点 1 表示叶子节点  
 acNode \* fail; // 失效指针  
 int dep;  
 acNode(int nodeKind);  
 void insert\_node(std::string value,bool flag = false);//将value插入到trie树中，只有root根节点可以调用该函数  
 int query(std::string &value,ofstream &ofs, long long int nowIndex,int &wordNum,vector<string> &v,bool flag = false);//查询树中节点在value中出现的次数  
 int queryWord(std::string &value); //查询一个单词出现的次数，在这之前需要调用过一次query  
 void init(); //root初始化函数 只有root可以调用  
 void buildFail();//使用BFS创建fail指针 只有root调用该函数  
 void close();//只有root可以调用，释放树节点空间  
 void dfsDestoryNode(acNode \*now);// 递归删除节点  
};

这个类在本实验不做过多的介绍，因此该类的具体实现在实验三中讲解过，相比较实验三中的acNode类，此次实验中只是在原先的query()函数中返回一个在本行中查找到的单词的集合，用于后续的建图。

### LinkList类

LinkList类的主要数据结构如下所示

class LinkList {  
public:  
 string value;  
 LinkList \*next;  
 int index;//表示当前节点value的索引  
 int du;//表示当前节点value中出去的边的个数  
 int weight;//权重  
public:  
 void setIndex(int index){this->index = index;}  
 void setDu(int du){this->du = du;}  
 int getWeight(){return weight;}  
 int getIndex(){return index;}  
 int getDu(){return du;}  
 LinkList(string &value,int weight = 1);//构造函数  
 bool operator<(LinkList link);  
 bool operator>(LinkList link);  
 void insertNode(string &str,bool p = false);//插入一个节点，按照升序插入  
 bool find(string &str);//查询某个单词是都存在  
 int getLen();//返回当前的len不包括头节点  
 string& getValue(){return value;}  
 LinkList\* getNext(){return next;}  
};

LinkList完成的功能是建立一个单链表，可以返回单链表的长度，需要注意的是因为头节点不存储建立的内容，因此返回的长度不包含头节点；可以插入节点，并且按照关键词的升序进行插入。

LinkList类的属性和方法的具体含义见表1-1和表1-2。

表1-1 LinkList属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| value | string | 表示结点的关键词 |
| next | LinkList\* | 下一个结点的链接 |
| index | int | 该结点的value对应的索引 |
| du | int | 表示当前结点value所对应的边的数目，也就是出度 |
| weight | int | 表示当前边的权重 |

表1-2 LinkList方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| void setIndex(int index) | 设置当前结点的index值 |
| void setDu(int du) | 设置当前结点的du值 |
| int getWeight() | 返回当前结点的weight值 |
| int getIndex() | 返回当前结点的index值 |
| int getDu() | 返回当前结点对应的value的du值 |
| LinkList(string &value,int weight=1) | 构造函数，传入当前结点的value值，以及默认的weight为1 |
| bool operator<(LinkList link) | 重载<运算符，按照value的<运算符结果进行比较 |
| bool operator>(LinkList link) | 重载>运算符，按照value的>运算符结果进行比较 |
| void insertNode(string &str,bool p = false) | 插入一个结点，插入的位置是当前结点后面所有的结点中按照value的升序进行插入（只有头节点调用才有意义） |
| bool find(string &str) | 查找某个value是否存在于链表中（实际应用中该函数未调用） |
| int getLen() | 返回当前结点后面结点的数目（只有头节点调用才有意义） |
| string &getValue() | 返回当前结点的value值 |
| LinkList\* getNext() | 返回当前结点的next值 |

### Hash类

该类的数据结构如下所示，其中包含另一个类linkList的实现，这个其实就是一个链表。因为Hash类的实现使用的是实验一实现的Hash类，其在实验一中进行过详解，因此这里不做过多的介绍。这个类完成的功能主要是对关键词和其索引进行Hash，优化查找速度。

class linkList{  
private:  
 linkList \*next;  
 int value; //-1表示该位置没有内容，因为表示的是索引，所以value一定大于等于0  
public:  
 linkList() {  
 next = nullptr;  
 value = -1;  
 }  
 void addLink(linkList \*nextLink){  
 linkList \*temp = next;  
 next = nextLink;  
 nextLink->setNext(temp);  
 }  
 void setNext(linkList \* temp){  
 next = temp;  
 }  
 linkList \* getNext(){  
 return next;  
 }  
 void setValue(int str) {  
 value = str;  
 }  
 int getValue(){  
 return value;  
 }  
};  
  
  
class Hash {  
private:  
 int M;//hash表的长度  
 linkList \*hashList;//链表的头节点  
 string filepath;//字典树的文件地址  
public:  
 Hash(string filepath,int M);  
 unsigned int getHashKey(string &value);  
 pair<int\*,int> query(string &value);//查询value的对应的key,<查询的数组，数组的长度>  
};

### LinkMatrix类

LinkMatrix类的功能是对关键词之间的边进行建图，使用链表的形式存储边，不过在最后的优化过程中该类被放弃使用，使用了速度更快更优的LinkThreeDimMatrix类。其数据结构如下所示

class LinkMatrix{  
private:  
 LinkList \*\*root;  
 int len;//行数 关键词的个数  
 Hash \*hash;  
 pair<int,int> \*wordEdgeNum;//边的个数 <个数，对应value的key>  
 int \*fa;//表示集合，fa[i]表示索引为i的单词的父亲  
 int \*faNum;//集合的大小  
public:  
 LinkList\*\* getRoot(){return root;}  
 LinkMatrix(string keyword,string content,int distance);//构造函数，传入一个文件表示关键词文件  
 void insertEdge(string &u,string &v);//插入一条边  
 void printMaxN(int n);//输出度数前n大  
 void dealAllEdge(string keyword,string content,int distance);  
 int getLen(){return len;}  
 int queryIndexOfWord(string value);//查询这个单词对应的root中的索引  
 void calFa();//计算父亲，进行并查集操作 closeness好像重复写了这一个功能，应该将其删除，使用这里的calFa  
 int find(int x);//并查集的find  
 int\* getFaNum(){return faNum;}  
 void calIndexAndDu();//计算所有节点的du和index  
};

LinkMatrix类的属性和方法的含义见表1-3和表1-4.

表1-3 LinkMatrix属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| root | LinkList\*\* | 用于存储边建图，root[i]表示的索引为i的关键词所有边的单链表； |
| len | int | 关键词的个数，所有关键词的索引的范围为[0,len-1]，关键词的索引是该单词是第几个出现 |
| hash | Hash\* | Hash类的引用，为了方便查找某个string的索引而设立 |
| wordEdgeNum | pair<int,int>\* | <边的个数，value对应的key> |
| fa | int\* | 并查集，表示哪些关键词在同一个集合（如果存在路径可达，则认为在同一个集合中） |
| faNum | int\* | 表示某个集合祖宗结点的元素个数，只有祖宗结点才有值，其余结点都为0 |

表1-4 LinkMatri方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| LinkList\*\* getRoot() | 返回root |
| LinkMatrix(string keyweord,string content,int distance) | 构造函数，根据keyword文件和content文件建图 |
| void insertEdge(string &u,string &v) | 插入一条边，也就是在root[indexU]后面创建一个结点v,在root[indexV]后面创建一个结点u |
| void printMaxN(int n) | 输出图中度数前n大 |
| int getLen() | 返回len，也就是图的关键词数 |
| int queryIndexOfWord(string value) | 查找value关键词对应的索引 |
| void calFa() | 计算并查集 |
| int find(int x) | 并查集的查找操作，也就是返回x的祖宗结点 |
| int \*getFaNum() | 返回faNum |
| void calIndexAndDu() | 计算root上所有链表的结点中的value对应的index和du |

### LinkThreeDimMatrix类

这个类是对LinkMatrix类的优化，其真正完成了建图插边的操作，相比较LinkMatrix速度快了差不多百倍左右。其数据结构如下所示。

class LinkThreeDimMatrix {  
private:  
 // root[i][j]表示的当前value，不存储别的单词，  
 LinkList \*\*\*root;//root[i]表示的是索引为i的单词的所有边 root[i][j]表示的是索引为i的单词的边的hash表示  
 int N;//表示单词的个数  
 int M;//表示每个单词的边的列表的映射数组长度  
 Hash \*hash; // 对单词进行hash表示  
 int \*fa;  
 int \*faNum;  
 pair<int,string> \*du;  
public:  
 LinkThreeDimMatrix(string keyword,string content,string duFile,int M);//duFile表示存放du的文件  
 void dealAllEdge(string keyword,string content);  
 void insertEdge(string &u,string &v);  
 int queryIndexOfWord(string value);//查询这个单词对应的root中的索引  
 void calIndexAndDu();  
 int getN(){return N;}  
 int getM(){return M;}  
 LinkList\*\*\* getRoot(){return root;}  
 int \*getFaNum(){return faNum;}  
 int \*getFa(){return fa;}  
 void calFa();  
 int find(int x);  
 pair<int,string>\* printMaxDu(int K);  
 void SaveDuAndSort(string duFile);//保存du文件  
};

LinkThreeDimMatrix类的属性和方法的含义见表1-5和表1-6.

表1-5 LinkThreeDimMatrix类属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| root | LinkList\*\*\* | 相比较LinkMatrix多了一个\*，表明这其实是一个三维矩阵形式，也是类名的由来 |
| N | int | 表示关键词的个数 |
| M | int | 表示每个单词的边的列表的映射数组的长度 |
| hash | Hash\* | Hash类指针，不仅用于root[i]索引的查找，也用于root[i][j]索引的查找 |
| fa | int\* | 并查集的fa,和LinkMatrix中的fa含义相同 |
| faNum | int\* | 并查集的faNum，和LinkMatrix中的faNum含义相同 |
| du | pair<int,string>\* | 表示关键词string对应的度int |

表1-6 LinkThreeDimMatrix方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| LinkThreeDimMatrix(string keyword,string content,string duFile,int M) | 构造方法，表示关键词文件、内容文件，保存的度文件，以及每个单词的hash数组长度 |
| void dealAllEdge(string keyword,string content) | 处理两个文件所产生的边，创建图 |
| void insertEdge(string &u,string &v) | 插入一条边，边的两个结点分别是u和v |
| void calIndexAnDu() | 计算每个结点的index和du |
| int getN() | 返回N |
| int getM() | 返回M |
| LinkList\*\*\* getRoot() | 返回root |
| int \*getFaNum() | 返回faNum |
| int \*getFa() | 返回fa |
| void calFa() | 计算并查集 |
| int find(int x) | 并查集的查找函数，查找x的祖宗结点 |
| pair<int,string>\*printMaxDu(int K) | 输出度前K大的单词及其度 |
| void SaveDuAndSort(string duFile) | 将度按照降序白存在duFile文件中 |

### ELL类

在计算接近中心性的时候直接使用LinkThreeMatrix创建的图时消耗的时间是比较长的，因为LinkthreeMatrix是使用链表存图的，每次需要解引用消耗时间过长，因此ELL就是根据LinkThreeDimMatrix类中的图来创建自己的数组图。其数据结构如下所示：

class ELL {  
private:  
 LinkThreeDimMatrix \*ltm;  
 int N;//单词数目  
 int \*\*graph;//表示ELL的图 graph[i]表示节点i所连的边 graph[i][0]表示和这个节点相连  
 int \*\*weight;//和graph对应，表示相应的权重  
 int \*du;//表示度  
public:  
 ELL(LinkThreeDimMatrix \*ltm);  
 void buildGraph();  
 int \*\* getGraph(){return graph;}  
 int \*\* getWeight(){return weight;}  
 int \*getDu(){return du;}  
};

ELL类的属性及其方法的含义见表1-7和表1-8

表1-7 ELL属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| ltm | LinkThreeDimMatrix\* | 该属性用于创建ELL的图 |
| N | int | 关键词的个数 |
| graph | int\*\* | 表示ELL的图，graph[i]表示结点i所连的边的数组,graph[i][j]表示ELL图的结点i的所连的第j条边 |
| weight | int\*\* | 和graph用法相同，不过其值表示的该边的权重 |
| du | int\* | 表示度 |

表1-8 ELL方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| ELL(LinkThreeDimMatrix\*ltm) | 构造函数，使用ltm初始化图 |
| void buildGraph() | 真正创建图的函数 |
| int\*\* getWeight() | 返回weight |
| int \*\*getGraph() | 返回graph |
| int\* getDu() | 返回du |

### closeness类

closeness类主要完成的工作是计算接近中心性分数，其数据结构如下所示。

class closeness {  
private:  
 int wordNum;//单词的个数  
 double \*wordClose;//数组，每个元素表示一个单词的中心分数， wordClose[i] 表示的是第i个单词  
 LinkThreeDimMatrix \*ltm;//用于处理图  
 int \*fa;//这个是单词索引的祖宗  
 int \*faNum; //这个是单词索引祖宗集合的元素个数  
 int K;//表示找到前20个最大中心性分数  
 double \*score;//存储前20大中心性分数  
 int maxWeight;//当不连通的时候认为两个点的权重 默认等于wordNum  
 ELL \*ell;  
 long long int \*distance;  
public:  
 closeness(LinkThreeDimMatrix \*ltm,ELL \*ell,int K);  
 void dealAllPointWithELL();  
 double dealOnePointWithELL(int index,int \*dis);  
 void dealAllPoint();//处理所有节点的分数  
 double dealOnePoint(int index,int \*dis,int \*vis);//处理一个节点的分数,单词索引为index  
 pair<string,double> \*printCloseScores(int n,string file);//输出前n大，并且将各个分数保存在file文件中  
 int find(int x);  
};

closeness类的属性及其方法见表1-9和表1-10

表1-9 closeness类属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| wordNum | int | 关键词的个数 |
| wordClose | double\* | 单词的接近中心性分数 |
| ltm | LinkThreeDimMatrix\* | 三维矩阵表示形式的图 |
| fa | int\* | 并查集的fa |
| faNum | int\* | 并查集的faNum |
| K | int | 表示寻找前K大 |
| score | doubel\* | 接近中心性分数 |
| maxWeight | int | 对于不连通的结点默认的距离 |
| ell | ELL\* | ELL表示形式的图 |
| distance | long long int\* | 结点到其他所有结点的最短距离和 |

表1-10 closeness类的方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| closeness(LinkThressDimMatrix \*ltm,ELL \*ell,int K) | 构造函数 |
| void dealAllPointWithELL() | 使用ELL创建的图计算所有接近中心性分数 |
| double dealOnePointWithELL(int index,int \*dis) | 使用ELL创建的图计算一个结点到其他所有结点的最短距离和 |
| void dealAllPoint() | 使用LinkThreeDimMatrix创建的图计算所有结点的接近中心性分数 |
| double dealOnePoint(int index,int \*dis) | 使用LinkThressDimMatrix创建的图计算一个结点到其他所有结点的最短距离和 |
| pair<string,double>\*printCloseScore(int n,string file) | 输出接近中心性分数前n大，并且将其返回，将所有接近中心性分数存入到文件file中 |
| int find(int x) | 并查集的查找函数，返回x的祖宗结点 |

### pageRankList类

pageRankList类主要完成的是计算pageRank分数，其数据结构如下所示

class pageRankList {  
private:  
// LinkMatrix \*lm;//存储图  
 LinkThreeDimMatrix \*ltm;  
 double \*pr;//pr分数  
 int N;//单词数 这个值从lm中获取  
 double a;//表示随机走向别的节点的概率  
 double c;//表示小于该值则认为迭代完毕  
 int \*du;//表示该节点的出度  
public:  
 pageRankList(LinkThreeDimMatrix \*ltm,double a,double c);  
 void deal();//计算pageRank算法  
 double \* dealOneIteration();//迭代一次，返回迭代后的pr，不过这是一个new新的，需要delete[]  
 pair<string,double>\* printMaxPageRankScore(int K,string pageFile);  
};

pageRankList类的属性及其方法含义见表1-11和表1-12

表1-11pageRankList属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| ltm | LinkThreeDimMatrix\* | LinkThreeDimMatrix形式表示的图 |
| pr | double \* | 所有结点的pagerank值 |
| N | int | 关键词的个数 |
| a | double | 表示随机访问其他结点的概率 |
| c | double | 表示两次迭代的pr值的差值的绝对值小于c,则认为迭代完成 |
| du | int\* | 表示结点的出度 |

表1-12 pageRankList方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| pageRank(LinkThreeDimMatrix \*ltm,double a,double c) | 构造函数 |
| void deal() | 计算pageRank的算法函数 |
| double \*dealOneIteration() | 迭代一起的结果，返回新的pr值的数组 |
| pair<string,double>\* printMaxPageRankScore(int K,string pageFile) | 输出并返回前K大的pageRank分数，将所有pageRank分数及其value存放在文件pageFile中 |

## 1.2 主要流程

1 main函数调用LinkThreeDimMatrix ltm(keyword,content,duFile,300)来创建三维图

* 1. LinkThreeDimMatrix构造函数首先遍历一遍keyword文件以获得关键词的个数N
  2. 调用hash = new Hash(keyword,N)对关键词建立hash表
  3. 调用dealAllEdge(keyword,content)计算边的关系并且创建三维图

1.3.1 调用acNode类，返回content每一行中出现的keyword中关键词的数组v（可重复出现）

1.3.2 对数组使用两个for循环，对其任意两个关键词之间建边，建边调用的是insertEdge(v[i],v[j])(注意i!=j)

1.3.2.1 调用indexU=queruIndexOFWord(u)获取关键词u对应的root索引

1.3.2.2 调用indexV=hash->getHashKey(v)%M获得关键词v在M大小的数组上的索引

1.3.2.3 调用root[indexU][indexV]->insertNode(v)将v插入到u的边中（按照升序）

* 1. 调用calIndexAndDu()计算三维图中各个结点的索引和出度
  2. 根据建立的边的关系也就是图，调用calFa()计算各个关键词之间的并查集
  3. SaveDuAndSort(duFile)按照度降序将其保存在文件duFile中

2 main函数调用ELL ell(&ltm)使用已有的三维图创建二维数组图

2.1 初始化各个变量N,du,graph,weight

2.2 调用buildGraph()函数创建数组二维图

2.2.1 遍历每一个关键词，对每个关键词将ltm中的二维图转换为一维数组

3 main函数调用closeness close(&ltm,&ell,20)，使用已有的两种建图方式初始化接近中心性算法

3.1 初始化各个变量fa,faNum,wordNum,maxWeight,wordClose,score,distance

4 main函数调用close.dealAllPointWithELL()，使用ELL建图的方式计算图的接近中心性

4.1 对每个关键词调用dealOnePointWithELL(i,dis)函数计算该关键词到其他所有关键词之间的最短距离和

4.1.1 使用迪杰斯特拉算法计算单源最短路，并返回该源点对其他所有关键词的最短路径和

5 main函数调用pageRankList prl(&ltm,0.15,0.00001)，使用ltm建图方式初始化pageRanK算法

5.1 初始化各个变量N,pr,du

6 main函数调用prl.deal()迭代结算图中各个结点的pageRank分数

6.1 迭代调用dealOneIteration()函数，直到两次的pr分数差值小于c

6.1.1 根据pagerank算法，使用乘积计算当前迭代的新pr值并返回

## 1.3 实验过程

实现过程显示见图1，总的运行时间100分钟左右。

最后的实验结果为见图2，图3，图4



图2 关键词出度

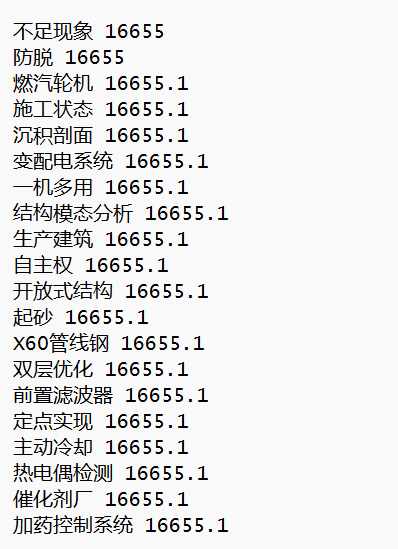


图3 关键词最短距离

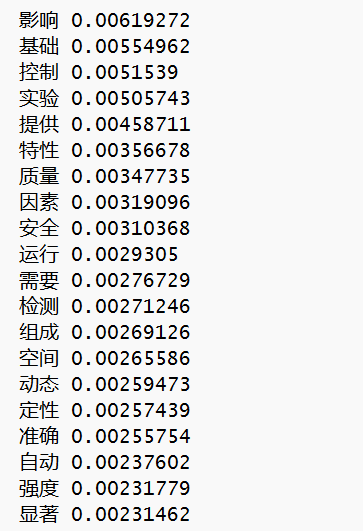


图4 关键词pageRank分数

## 1.4 技术指标

使用rank10w的数据集，内存占用为2834MB

## 1.5 遇到问题

这次的实验四相较前三个实验遇到的问题比较多，其中最明显的一个问题就是最开始给定的keyword数据集，由于过大，运行时间过长，导致计算不出来，后来给定了一个新的小数据集rank10w.txt

遇到的第二个问题就是使用一维单链表存储某个关键词的所有边的时候耗费时间比较长，因为如果是再添加相同的边需要增加权重，因此需要遍历单链表才能知道是否存在该边，即使是存储单链表的时候按照有序存储，耗费时间仍比较长。解决优化的办法是使用三维矩阵来进行存储。如图5所示，表示的是一个关键词的二维存储，假设需要添加一条边，首先对该边的结点v的value进行hash映射，然后将key对100求余得到索引index,此时这条边只需要插入到索引index所在的单链表就可以，这样极大的减小了插入查询的时间。

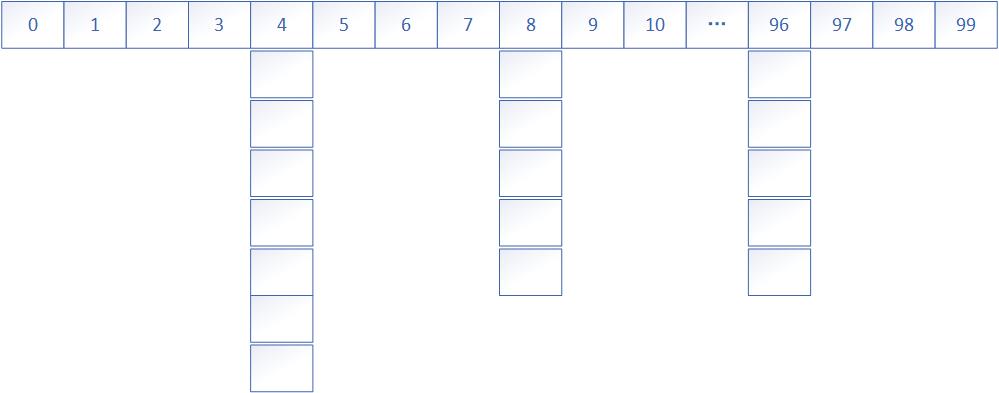


图5 一个关键词的二维存储

遇到的第三个主要问题是计算接近中心性的算法的时候耗费时间比较长，发现可能是一位内使用链表存储的图，因此每次访问解引用的时间可能比较长（相较于数组形式），因此创建ELL类使用已经创建好的三维图来初始化数组图，来达到优化的目的，最后速度提升了将近十倍左右，耗费时间为90分钟左右。

## 1.6 结论和总结

这次实验四相比较前三次实验花费的时间明显增多了，我想大数据集是一个比较重要的原因，但是也也正是这样的大数据集让我知道了代码优化数据结构的重要性，同样的内容，因为多次调用，数组形式比链表形式会快10倍左右。一个巧妙的数据结构，可以很好的优化一个专门的问题。

学会在大数据集问题中的优化或许也是这门课程想要教授给我们的内容。