1.题目：图的深度优先遍历和广度优先遍历

算法：见代码

代码行数：146

源代码：

//0表示无向图 1表示有向图

#include<iostream>

#include<queue>

#include<vector>

using namespace std;

//边类

class Edge{

public:

int adjVex;//终点的点编号

Edge \*nextEdge;//指向下一个节点的指针

};

//图类

class Vertex{

public:

char data;//点的信息

Edge \*firstEdge;//指向第一条关联边的指针

};

class Graph{

public:

vector<Vertex> vex;//邻接表

int vexNum,edgeNum;

char \*vexData;//存储点的信息的数组

int graphKind;//图的类型

};

void createGraph(Graph &G){

//输入点数和边数

cout<<"请以空格间隔 依次输入点数和边数"<<endl;

cin>>G.vexNum>>G.edgeNum;

//预设邻接表空间

G.vex.resize(G.vexNum);

//预设点信息数组的空间

G.vexData=new char[G.vexNum];

//输入点的信息

cout<<"请依次输入点的信息"<<endl;

for(int i=0;i<G.vexNum;i++){

cout<<"请输入编号为"<<i<<"的点的信息"<<endl;

char x;

cin>>x;

G.vex[i].data=x;

G.vex[i].firstEdge=NULL;//将指针置空

G.vexData[i]=x;

}

//输入图的类型

cout<<"请输入图的类型："<<endl;

cout<<"0.无向图 1.有向图"<<endl;

cin>>G.graphKind;

if(G.graphKind){

for(int i=0;i<G.edgeNum;i++){

cout<<"请依次输入有向边的起点和终点编号"<<endl;

int start,end;

cin>>start>>end;

//创建边

Edge \*newEdge=new Edge;

newEdge->adjVex=end;

//使用头插法进行创建

//先动尾 再动头

G.vex[start].firstEdge=newEdge->nextEdge;

G.vex[i].firstEdge=newEdge;

}

}

else{

for(int i=0;i<G.edgeNum;i++){

//对于无向图 注意要创建对称边（实际上是一条边）

for(int i=0;i<G.edgeNum;i++)

{

cout<<"请输入邻接两点的编号"<<endl;

int v1,v2;

cin>>v1>>v2;

//创建新边

Edge \*newEdge1=new Edge;

newEdge1->adjVex=v2;

newEdge1->nextEdge=G.vex[v1].firstEdge;

G.vex[v1].firstEdge=newEdge1;

//创建对称边

Edge \*newEdge2=new Edge;

newEdge2->adjVex=v1;

newEdge2->nextEdge=G.vex[v2].firstEdge;

G.vex[v2].firstEdge=newEdge2;

}

}

}

}

void DFS(Graph &G,int startIndex,bool \*ifVisited){

//首先将起点标为已访问

ifVisited[startIndex]=true;

//输出信息

cout<<G.vex[startIndex].data<<" ";

//然后对所有子节点进行递归

Edge \*p=G.vex[startIndex].firstEdge;//辅助指针 初始化为第一条关联边

while(p)

{

DFS(G, p->adjVex, ifVisited);

p=p->nextEdge;

}

}

void DFSTraverse(Graph &G){

//创建辅助数组 记录访问标志

bool ifVisited[G.vexNum];

for(int i=0;i<G.vexNum;i++){

ifVisited[i]=false;//初始化

}

//对所有未访问点进行一次DFS

for(int i=0;i<G.vexNum;i++){

if(ifVisited[i]==false){

DFS(G, i, ifVisited);

}

}

}

void BFS(Graph &G,int startIndex,bool \*ifVisited){

queue<int> helpQ;//辅助队列

//入队即访问

//起点入队

helpQ.push(startIndex);

ifVisited[startIndex]=true;

while(!helpQ.empty()){

//记录后出队

int indexOfOutVex=helpQ.front();

helpQ.pop();

cout<<G.vex[indexOfOutVex].data<<" ";

//然后子代未访问的点入队（防止重复入队）

Edge \*p=G.vex[startIndex].firstEdge;//辅助指针 初始化为第一个子节点

while(p){

if(ifVisited[p->adjVex]==false){

helpQ.push(p->adjVex);

ifVisited[p->adjVex]=true;

}

p=p->nextEdge;

}

//继续进行出入队操作

}

}

void BFSTraverse(Graph &G){

bool ifVisited[G.vexNum];

for(int i=0;i<G.vexNum;i++){

ifVisited[i]=false;

}

for(int i=0;i<G.vexNum;i++){

if(ifVisited[i]==false){

BFS(G, i, ifVisited);

}

}

}

int main(){

return 0;

}

2.题目：dijkstra算法实现

算法：见代码

代码行数：137

源代码：

//基于邻接矩阵实现

//0表示无向网 1表示有向网

//注意在带权图中 0表示依然表示有效距离 如果要表示两者不邻接

//所有点从0开始编号

#include<iostream>

#include<vector>

#include<stack>

#include<algorithm>

#include<fstream>

#include<limits>

using namespace std;

const int inf=99999

//startIndex是起始点

class Vertex{

public:

int code;

char data;

};

class Graph{

public:

int vexNum,edgeNum;

int graphKind;

vector<vector<int>> matrix;

//注意还要有存储点信息的数组

vector<char> vdata;

};

//从文件读取数据

void createGraph(Graph &G){

fstream f;

f.open("data.txt",ios::in);

f>>G.vexNum>>G.edgeNum;

f>>G.graphKind;

//输入点的信息

for(int i=0;i<G.vexNum;i++){

char x;

f>>x;

G.vdata.push\_back(x);

}

//邻接矩阵初始化

G.matrix.resize(G.vexNum);

for(int i=0;i<G.vexNum;i++){

G.matrix[i].resize(G.vexNum);

}

//输入邻接矩阵

for(int i=0;i<G.vexNum;i++){

for(int j=0;j<G.vexNum;j++){

f>>G.matrix[i][j];

}

}

}

//选择最小距离函数 返回值为所取点的编号

int findCodeOfMinDistance(vector<int> &minDistance,vector<bool> &flag){

int codeOfVex;

int minDis;

//查找flag中第一个0 返回其下标作为minDis的初始化值

vector<bool>::iterator initIt=find(flag.begin(),flag.end(),false);

minDis=minDistance[initIt-flag.begin()];

codeOfVex=initIt-flag.begin();

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

if((\*it)<minDis){

minDis=(\*it);

codeOfVex=it-minDistance.begin();

}

}

return codeOfVex;

}

void Dijkstra(Graph &G,int startIndex){

//数据结构：

//1.当前最小路径上的最近邻接点

//2.经过1中邻接点从起点到达当前点的最小距离（注意是包含了以前的距离的）

//3.是否已纳入的标志

vector<int> nearestAdjVex;

vector<int> minDistance;

vector<bool> flag;

//数组初始化

nearestAdjVex.resize(G.vexNum);

minDistance.resize(G.vexNum);

flag.resize(G.vexNum);

//先将startIndex的flag初始化

flag[startIndex]=true;

//根据起始点进行初始化 用-1表示两点不邻接

for(int i=0;i<G.vexNum;i++){

nearestAdjVex[i]=startIndex;

if(G.matrix[startIndex][i]==-1){

//当不邻接时 需要将值设定为无穷大

minDistance[i]=INT\_MAX;

}

//邻接时

//考量距离时注意方向 对i应考虑其作为终点而不是起点

else{

minDistance[i]=G.matrix[startIndex][i];

}

if(i!=startIndex){

flag[i]=false;//避开起始点操作

}

}

//算法执行

//以下操作只需进行G.vexNum-1次

for(int i=0;i<G.vexNum-1;i++){

int codeOfMinDistanceVex=findCodeOfMinDistance(minDistance, flag);

//先将其标志置为1

flag[codeOfMinDistanceVex]=true;

//检查经过该点到达其他点距离是否更小

for(int i=0;i<G.vexNum;i++){

if(flag[i]==true){

continue;

}

//执行到下面说明flag为0

int distanceToCheck=minDistance[codeOfMinDistanceVex]+G.matrix[codeOfMinDistanceVex][i];

if(distanceToCheck<minDistance[i]){

minDistance[i]=distanceToCheck;

nearestAdjVex[i]=codeOfMinDistanceVex;

}

}

}

//算法执行完毕 下进行输出

for(int i=0;i<G.vexNum;i++){

if(i==startIndex){

continue;

}

cout<<"从"<<startIndex<<"号点到"<<i<<"号点的最短距离为："<<minDistance[i]<<endl;

cout<<"最短路径为:"<<startIndex<<"->";

stack<int> ans;

int vexToCheck=i;

while(vexToCheck!=startIndex){

ans.push(i);

vexToCheck=nearestAdjVex[i];

}

while(!ans.empty()){

cout<<ans.top()<<"->";

ans.pop();

}

}

}

int main(){

return 0;

}

3.题目：

小明在他的果园里种了一些苹果树，这些苹果树排列成一个圆。为了保证苹果的品质，在种植过程中要进行疏果操作。为了更及时地完成疏果操作，小明会不时地检查每棵树的状态，根据需要进行疏果。检查时，如果发现可能有苹果从树上掉落，小明会重新统计树上的苹果个数（然后根据之前的记录就可以判断是否有苹果掉落了)。在全部操作结束后，请帮助小明统计相关的信息。

算法：运用前缀和思想 统计输入到当前时的苹果总数 若有不同说明掉果了

代码行数：54

源代码：

//算法：运用前缀和思想 在输入数据时就开始进行数据处理

#include<iostream>

#include<vector>

#include<unordered\_set>//运用哈希表进行查找相邻项

using namespace std;

int main(){

int T,D,E;

T=0;

D=0;

E=0;

int N;

cin>>N;

vector<int> currentNumOfApples;//记录当前树上有多少苹果

unordered\_set<int> treesThatLoseApples;//记录那些树有掉果

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

int m;//操作数

cin>>m;

for(int j=0;j<m;j++){

int a;

cin>>a;

//第一项特殊处理

if(j==0){

currentNumOfApples.push\_back(a);

continue;

}

//若输入的是正数且比当前苹果数少 说明掉果了

if(a>0&&a<currentNumOfApples[i]){

treesThatLoseApples.insert(i);

currentNumOfApples[i]=a;

}

else if(a<0){

//输入数小于等于0

currentNumOfApples[i]+=a;

}

}

}

//输入完成后 可以直接统计出T和D

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

T+=currentNumOfApples[i];

}

D=treesThatLoseApples.size();

//将每个树的编号依次视为最小项 查找(i+1)%N和(i+2)%N是否也在哈希表中

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

if(

treesThatLoseApples.find(i)!=treesThatLoseApples.end()&&

treesThatLoseApples.find((i+1)%N)!=treesThatLoseApples.end()&&

treesThatLoseApples.find((i+2)%N)!=treesThatLoseApples.end()

){

E++;

}

}

cout<<T<<" "<<D<<" "<<E<<endl;

return 0;

}

4.题目：暑假要到了。可惜由于种种原因，小P原本的出游计划取消。失望的小P只能留在西西艾弗岛上度过一个略显单调的假期……直到……

某天，小P获得了一张神秘的藏宝图。

西西艾弗岛上种有n棵树，这些树的具体位置记录在一张绿化图上。

简单地说，西西艾弗岛绿化图可以视作一个大小为(L+1)×(L+1)的01矩阵A，

地图左下角（坐标(0,0)）和右上角（坐标(L,L)）分别对应A[0][0]和A[L][L]。

其中A[i][j]=1表示坐标(i,j)处种有一棵树，A[i][j]=0则表示坐标(i,j)处没有树。

换言之，矩阵A中有且仅有的n个1展示了西西艾弗岛上n棵树的具体位置。

传说，大冒险家顿顿的宝藏就埋藏在某棵树下。

并且，顿顿还从西西艾弗岛的绿化图上剪下了一小块，制作成藏宝图指示其位置。

具体来说，藏宝图可以看作一个大小为(S+1)×(S+1)的01矩阵B（S远小于L），对应着A中的某一部分。

理论上，绿化图A中存在着一处坐标(x,y)（0≤x,y≤L-S）与藏宝图B左下角(0,0)相对应，即满足：对B上任意一处坐标(i,j)（0≤i,j≤S），都有A[x+i][y+j]=B[i][j]。

当上述条件满足时，我们就认为藏宝图B对应着绿化图A中左下角为(x,y)、右上角为(x+S,y+S)的区域。

实际上，考虑到藏宝图仅描绘了很小的一个范围，满足上述条件的坐标(x,y)很可能存在多个。

请结合西西艾弗岛绿化图中n棵树的位置，以及小P手中的藏宝图，判断绿化图中有多少处坐标满足条件。

特别地，藏宝图左下角位置一定是一棵树，即A[x][y]=B[0][0]=1，表示了宝藏埋藏的位置。

算法：//存整张图会导致内存超限 所以只存储两张图1的坐标即可

//另外在判断点时 如果以该点作为左下角已经超过图的坐标范围 则可以直接舍弃 减少比对时间

//由于藏宝图的左下角一定是1 所以遍历原图中所有的1 将其视为左下角与藏宝图进行依次比对

//注意！unordered\_set容器不支持pair类型作为参数

//unordered\_map类型则不支持键重复 因此此题在C++中最好的做法还是采用vector

代码行数：67

源代码：

//存整张图会导致内存超限 所以只存储两张图1的坐标即可

//另外在判断点时 如果以该点作为左下角已经超过图的坐标范围 则可以直接舍弃 减少比对时间

//由于藏宝图的左下角一定是1 所以遍历原图中所有的1 将其视为左下角与藏宝图进行依次比对

//注意！unordered\_set容器不支持pair类型作为参数

//unordered\_map类型则不支持键重复 因此此题在C++中最好的做法还是采用vector

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

int main(){

int count=0;

int numOfTrees,l,s;

cin>>numOfTrees>>l>>s;

vector<pair<int,int>> greenGraph;

vector<bool> flag;//表示某点是否符合条件的标志数组

flag.resize(numOfTrees,true);//初始化 认为全符合要求

vector<vector<int>> treasureGraph;

treasureGraph.resize(s+1);

//输入绿化图

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

int x,y;

cin>>x>>y;

greenGraph.emplace\_back(x,y);

}

//输入藏宝图 注意藏宝图是以矩阵形式给出 并且输入的顺序是实际图中从上到下的顺序

for(int i=0;i<s+1;i++){

for(int j=0;j<s+1;j++){

int x;

cin>>x;

treasureGraph[s-i].push\_back(x);

}

}

//遍历绿化图中1的点 依次与藏宝图中的点进行比对

for(int k=0;k<greenGraph.size();k++){

//以该点为左下角超过范围的点要舍去

if(greenGraph[k].first+s>l||greenGraph[k].second+s>l){

flag[k]=false;

continue;

}

//若执行到下面说明未超范围

for(int i=0;i<s+1;i++){

for(int j=0;j<s+1;j++){

if(

(find(greenGraph.begin(),greenGraph.end(),pair<int,int>{greenGraph[k].first+i,greenGraph[k].second+j})!=greenGraph.end()&&treasureGraph[i][j]==0)

||

(find(greenGraph.begin(),greenGraph.end(),pair<int,int>{greenGraph[k].first+i,greenGraph[k].second+j})==greenGraph.end()&&treasureGraph[i][j]==1)

){

//不匹配 flag修改 退出两重循环 进入下一个点

flag[k]=false;

break;

}

}

//根据flag判断是否退出当前循环

if(flag[k]==false){

break;

}

}

}

//遍历flag 统计其中true的个数 即为最终答案

for(int i=0;i<flag.size();i++){

if(flag[i]==true){

count++;

}

}

cout<<count<<endl;

return 0;

}