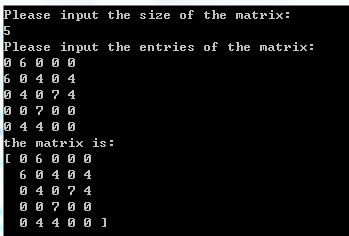
KruskalAlgorithm C++

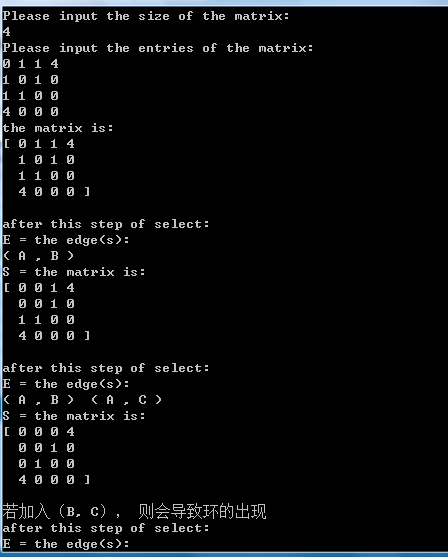
邱永臣

1. 测试用例

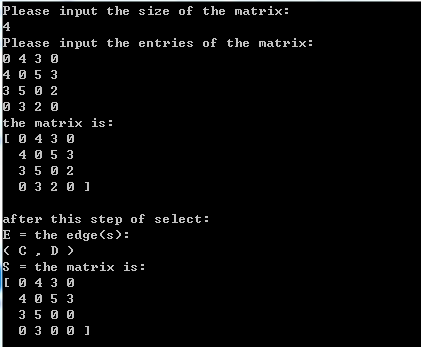
a.



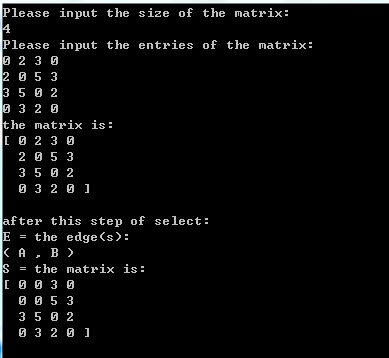
b.



c.



d.



1. 程序代码
2. main.cpp

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\* Kruskal Algorithm.

/\*

/\* main.cpp

/\*

/\* implemented by SomebodyWho

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "graphMatrix.h"

#include <iostream>

using namespace std;

int main() {

graphMatrix m;

cin >> m;

cout << m;

m.repeat();

m.showEdge();

return 0;

}

1. graphMatrix.h

/\*

graphMatrix.h

\*/

#include <iostream>

#include <array>

#include <set>

#include <vector>

#define MAXSIZE 100

#define MINSIZE 0

using namespace std;

class graphMatrix {

public:

friend ostream& operator<<(ostream& out, const graphMatrix& g);

friend istream& operator>>(istream& in, graphMatrix& g);

graphMatrix();

~graphMatrix();

int size() const;

bool empty() const;

int smallestEntryRow() const;

int smallestEntryColumn() const;

void selectEdgeOfLeastWeight(int verA, int verB);

void replaceSetOfEdges(int verA, int verB);

bool itMakesACycle(int verA, int verB);

void initEdgeMatrix();

void constructEdgeMatrix(int verA, int verB);

void findConnectComponents();

bool scanAllRow();

bool Cycle();

void showEdge();

void repeat();

void showstep();

private:

// 记录图对应的矩阵信息

int matrix[MAXSIZE][MAXSIZE];

int sizeOfMatrix;

// 记录已经被选出来的edge

set< pair<int, int> > edge;

// 在不知道某条边是否会形成环的情况下，我们把它和之前选出来的边放在

// 下面这个矩阵中，进而找出“加入该条边”之后形成的连通部分

int edgeMatrix[MAXSIZE][MAXSIZE];

// 采用逐行扫描的方式来构建出“不同的连通部分”

bool hasScanRow[MAXSIZE];

// 下面是一个以集合为元素的数组，一个元素代表着一个连通部分

array< set<int>, MAXSIZE> connectGraph;

};

1. graphMatrix.cpp

/\*

graphMatrix.cpp

\*/

#include "graphMatrix.h"

#define SOBIG 99999999

// overloading the '<<' operator to print the matrix.

ostream& operator<<(ostream& out, const graphMatrix& g) {

if (g.empty()) {

cout << "The matrix is empty.";

} else {

cout << "the matrix is: \n";

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

// output '[' at the beginning.

if (i != 0)

out << " ";

else

out << "[";

// output main

for (int j = 0; j < g.size(); j++) {

out << " " << g.matrix[i][j];

}

// output ']' at the end.

if (i != g.sizeOfMatrix - 1)

out << endl;

else

out << " ]";

}

}

out << endl << endl;

return out;

}

// overloading the '>>' operator to input the matrix.

istream& operator>>(istream& in, graphMatrix& g) {

int temp; // = 4;

cout << "Please input the size of the matrix: \n";

cin >> temp;

// check of valid size.

while (temp < MINSIZE || temp > MAXSIZE) {

cout << "Please input the size that is valid of the matrix: \n";

cin >> temp;

}

g.sizeOfMatrix = temp;

if (g.empty()) {

cout << "Warning! the matrix is empty.\n";

} else {

cout << "Please input the entries of the matrix:\n";

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

for (int j = 0; j < g.size(); j++) {

in >> temp;

//temp = i;

g.matrix[i][j] = temp;

}

}

}

return in;

}

// constructor.

graphMatrix::graphMatrix() {

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

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

matrix[i][j] = 0;

}

sizeOfMatrix = 0;

}

// ~deconstructor.

graphMatrix::~graphMatrix() {

}

// get the size of the matrix.

int graphMatrix::size() const {

return sizeOfMatrix;

}

// determinte if the matrix is empty.

bool graphMatrix::empty() const {

return size() == 0;

}

// find the row of the smallest weight in the matrix,

// we notice that 0 represent the biggest weight.

int graphMatrix::smallestEntryRow() const {

int smallest = SOBIG;

int rowIndex = -1, columnIndex = -1;

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

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

if (matrix[i][j] != 0)

if (matrix[i][j] < smallest) {

rowIndex = i;

columnIndex = j;

smallest = matrix[i][j];

}

}

}

return rowIndex;

}

// find the column of the smallest weight in the matrix,

// we notice that 0 represent the biggest weight.

int graphMatrix::smallestEntryColumn() const {

int smallest = SOBIG;

int rowIndex = -1, columnIndex = -1;

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

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

if (matrix[i][j] != 0)

if (matrix[i][j] < smallest) {

rowIndex = i;

columnIndex = j;

smallest = matrix[i][j];

}

}

}

return columnIndex;

}

// choose an edge (verA, verB) in S of least weight.

void graphMatrix::selectEdgeOfLeastWeight(int verA, int verB) {

edge.insert(pair<int, int>(verA, verB));

}

// replace S with S - {(verA, verB)}.

void graphMatrix::replaceSetOfEdges(int verA, int verB) {

matrix[verA][verB] = 0;

matrix[verB][verA] = 0;

}

// 判断选出该条边是否会导致环的出现

bool graphMatrix::itMakesACycle(int verA, int verB) {

initEdgeMatrix();

constructEdgeMatrix(verA, verB); // we construc a matrix that is from the

// original edges plusing (verA, verB).

findConnectComponents();

return Cycle();

}

void graphMatrix::initEdgeMatrix() {

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

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

edgeMatrix[i][j] = 0;

}

void graphMatrix::constructEdgeMatrix(int verA, int verB) {

for (set< pair<int, int> >::iterator it = edge.begin();

it != edge.end(); ++it)

edgeMatrix[it->first][it->second] = 1;

edgeMatrix[verA][verB] = 1;

edgeMatrix[verB][verA] = 1;

}

// find out all different components and store them

// in array connectGraph[MAXSZE].

// 构建不同“连通部分”

void graphMatrix::findConnectComponents() {

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

hasScanRow[i] = false; //

connectGraph[i].clear();

}

int rowI = 0;

// to get the first componnent.

while (connectGraph[0].empty()) {

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

if (edgeMatrix[rowI][j] == 1) {

connectGraph[0].insert(rowI);

connectGraph[0].insert(j);

}

}

hasScanRow[rowI] = true;

rowI++;

}

// to find out all the different components which are stored

// in connectGraph[i].

// we had scan all rows, we would got all the components, or not.

for (int i = 0; !scanAllRow(); i++) {

int rowI = 0;

// to get the first componnent.

while (connectGraph[i].empty() && rowI < 100) {

if (!hasScanRow[rowI]) {

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

if (edgeMatrix[rowI][j] == 1) {

connectGraph[i].insert(rowI);

connectGraph[i].insert(j);

}

}

hasScanRow[rowI] = true;

}

rowI++;

}

// for every conponent(set connectGraph[i]), we scan the vertext

// represented by row stored in set's entry, and if we find some

// new vertext that is connected to the graph represented by

// set connectGraph[i], we have to scan from the beginning of

// set(of course, we label those had been scaned as 'hasScanRow'

// to avoid death recure.)

for (set<int>::iterator it = connectGraph[i].begin();

it != connectGraph[i].end();) {

bool shouldWeRebegin = false;

// avoid to scan those had been scanned before.

if (!hasScanRow[\*it]) {

// scan that row if it hadn't been scanned.

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

if (edgeMatrix[\*it][j] == 1)

connectGraph[i].insert(j);

// we had add somenew vertext, so we should rebegin.

shouldWeRebegin = true;

}

// label the row as hasing been scanned.

hasScanRow[\*it] = true;

}

// if we had add somenew vertext, we should rebegin.

if (shouldWeRebegin) {

it = connectGraph[i].begin();

// or just go ahead.

} else {

++it;

}

} // had got a small component.

} // had got all different components.

}

// determinite whether we had scan all rows.

bool graphMatrix::scanAllRow() {

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

if (!hasScanRow[i])

return false;

return true;

}

// 在已经构建出不同“连通部分”的情况下，判断环是否存在。

bool graphMatrix::Cycle() {

// 'deep into' every different componnent.

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

int numOfEdge = 0;

// in this componnent, we scan all the vertexts it has.

// just to calculate how many edges in it.

for (set<int>::iterator it = connectGraph[i].begin();

it != connectGraph[i].end(); ++it) {

// scan the row who represents the vertext in edgeMatrix.

for (int j = \*it; j < size(); j++) {

if (edgeMatrix[\*it][j] == 1)

numOfEdge++;

}

}

// if the number of edges N1 is large than (the numbers of edges N2 - 1)

// there must be a cycle.

if (numOfEdge > connectGraph[i].size() - 1)

return true;

}

// if there is no cycle.

return false;

}

// finally, we have to show all the edges of the minimal spanning

// tree that we construct.

void graphMatrix::showEdge() {

cout << "the edge(s):\n";

for (set< pair<int, int> >::iterator it = edge.begin();

it != edge.end(); ++it) {

char first = it->first + 'A';

char second = it->second + 'A';

cout << "( " << first << " , " << second << " ) ";

}

cout << endl;

}

// 关键的“重复”步骤。

void graphMatrix::repeat() {

selectEdgeOfLeastWeight(smallestEntryRow(), smallestEntryColumn());

replaceSetOfEdges(smallestEntryRow(), smallestEntryColumn());

showstep();

while (edge.size() != (size() - 1)) {

if (!itMakesACycle(smallestEntryRow(), smallestEntryColumn())) {

selectEdgeOfLeastWeight(smallestEntryRow(), smallestEntryColumn());

replaceSetOfEdges(smallestEntryRow(), smallestEntryColumn());

showstep();

}

// 如果构成了环，我们须放弃这条边。不然它仍是下一次循环的最小边，从而死循环

else {

cout << "若加入（";

char row = smallestEntryRow() + 'A';

char col = smallestEntryColumn() + 'A';

cout << row << ", " << col;

cout << "）， 则会导致环的出现" << endl;

replaceSetOfEdges(smallestEntryRow(), smallestEntryColumn());

}

}

}

// 这个函数只是用来辅助显示结果罢了。

void graphMatrix::showstep() {

cout << "after this step of select:\nE = ";

showEdge();

cout << "S = ";

cout << \*this;

}

1. 其他：无