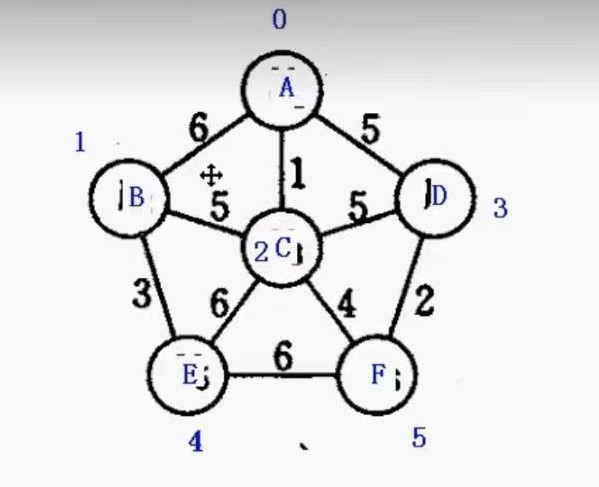
**最小生成树方法实现**



方法一：Prim算法

#include <iostream>

#include<malloc.h>

#include <assert.h>

using namespace std;

#define Default\_Vertex\_Size 10

#define T char

#define E int

#define MAX\_COST 0x7FFFFFFF

typedef struct GraphMtx {

int MaxVertices; //容量（可以不填满）

int NumVertices; //真实顶点数

int NumEdges; //真实边数

T\* verticeList; //定义指向顶点列表的指针

int\*\* Edge; //边不是0就是1，因此是int，又因为是一个二维列表，所以就用二级指针来指向整个二维列表

}GraphMtx;

void InitGraph(GraphMtx &g) {

g.MaxVertices = Default\_Vertex\_Size;

g.NumEdges = g.NumVertices = 0;

//对顶点列表的开辟

g.verticeList = (T\*)malloc(sizeof(T) \* (g.MaxVertices));

assert(g.verticeList != NULL);

//对边二维列表的开辟(之前没有遇到过，第一次学习)

g.Edge = (int\*\*)malloc(sizeof(int\*) \* g.MaxVertices);

assert(g.Edge != NULL);

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

g.Edge[i] = (int\*)malloc(sizeof(int) \* g.MaxVertices);

}

//将二维列表里的每一个数据都初始化为0

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

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

if (i == j) {

g.Edge[i][j] = 0; //即自己到自己的距离为0

}

g.Edge[i][j] = MAX\_COST;

}

}

}

void InsertVertex(GraphMtx&g, T v) {

if (g.NumVertices >= g.MaxVertices) { //先看看满不满

return;

}

g.verticeList[g.NumVertices++] = v;

}

void ShowGraph(GraphMtx& g) {

cout << " ";

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

cout << g.verticeList[i] << " ";

}

cout << endl;

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

cout << g.verticeList[i] << " ";

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

if (g.Edge[i][j] == MAX\_COST)

cout << "&" << " ";

else

cout << g.Edge[i][j] << " ";

}

cout << endl;

}

}

int GetVertexPos(GraphMtx& g, T v) {

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

if (g.verticeList[i] == v) {

return i;

}

}

return -1;

}

void InsertEdge(GraphMtx& g, T v1, T v2,E cost) {

int p1 = GetVertexPos(g, v1);

int p2 = GetVertexPos(g, v2);

if (p1 == -1 || p2 == -1) {

return;

}

g.Edge[p1][p2] = g.Edge[p2][p1] = cost;

g.NumEdges++;

}

void RemoveEdge(GraphMtx& g, T v1, T v2) {

int p1 = GetVertexPos(g, v1);

int p2 = GetVertexPos(g, v2);

if (p1 == -1 || p2 == -1) {

return;

}

if (g.Edge[p1][p2] == 0) {

return;

}

g.Edge[p1][p2] = g.Edge[p2][p1] = 0;

g.NumEdges--;

}

//比较复杂，既要删除顶点，也要删除相关行和列

void RemoveVertex(GraphMtx& g, T v) {

//下面是追求美观不追求效率的方法

//效率高的方法是用最后一行或最后一列进行覆盖

//先删除顶点

int p = GetVertexPos(g, v);

if (p == -1) {

return;

}

for (int i = p; i < g.NumVertices - 1; i++) {

g.verticeList[i] = g.verticeList[i + 1];

}

//统计被删除的边数

int num = 0;

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

if (g.Edge[p][i] != 0) {

num++;

}

}

//下面删除顶点所在行和所在列删除

//先删除行

for (int i = p; i < g.NumVertices - 1; i++) {

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

g.Edge[i][j] = g.Edge[i + 1][j];

}

}

//再删除列

for (int i = p; i < g.NumVertices - 1; i++) {

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

g.Edge[j][i] = g.Edge[j][i + 1];

}

}

g.NumVertices--;

g.NumEdges -= num;

}

void DestroyGraph(GraphMtx& g) {

free(g.verticeList);

g.verticeList = nullptr;

//释放二维列表要多一步，即释放两次

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

free(g.Edge[i]);

}

free(g.Edge);

g.Edge = nullptr;

g.MaxVertices = g.NumEdges = g.NumVertices = 0;

}

//找到第一个邻接顶点

//顺序是按照我们自己插入的顺序来看的

int GetFirstNeighbor(GraphMtx& g, T v) {

int p = GetVertexPos(g, v);

if (p == -1) {

return -1;

}

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

if (g.Edge[p][i] == 1) {

return i;

}

}

return -1;

}

int GetNextNeighbor(GraphMtx& g, T v, T w) {

int pv = GetVertexPos(g, v);

int pw = GetVertexPos(g, w);

if (pv == -1 || pw == -1) {

return -1;

}

for (int i = pw + 1; i < g.NumVertices; i++) {

if (g.Edge[pv][i] == 1) {

return i;

}

}

return -1;

}

E GetWeight(GraphMtx& g,int v1,int v2) {

if (v1 == -1 || v2 == -1) {

return MAX\_COST;

}

return g.Edge[v1][v2];

}

void MinSpanTree\_Prim(GraphMtx& g,T vertex) {

int n = g.NumVertices;

E\*lowcost = (E\*)malloc(sizeof(E)\*n);

int \*mst = (int\*)malloc(sizeof(int)\*n); //相应起始顶点的数组

assert(lowcost != nullptr&&mst != nullptr);

//从某个顶点进行初始化

int k = GetVertexPos(g, vertex);

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

if (i != k) {

lowcost[i] = GetWeight(g, k, i);

mst[i] = k;

}

else {

lowcost[i] = 0;

}

}

//找当前的最小值

int min, min\_index;

int begin, end;

E cost;

for (int i = 0; i < n - 1; i++) { //需要找n-1条边

min = MAX\_COST;

min\_index = -1;

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

if (lowcost[j] !=0&&lowcost[j]<min) { //lowcost[j] !=0 等于0说明已经并入了最小生成树，因此不考虑

min = lowcost[j];

min\_index = j;

}

}

begin = mst[min\_index];

end = min\_index;

cout << g.verticeList[begin] << "-->" << g.verticeList[end] << ":" << min << endl;

lowcost[min\_index] = 0;

//比较A和C到各个顶点的权值大小

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

cost = GetWeight(g, min\_index, j);

if (cost < lowcost[j]) {

lowcost[j] = cost;

mst[j] = min\_index;

}

}

}

}

int main() {

GraphMtx gm;

InitGraph(gm);

InsertVertex(gm, 'A');

InsertVertex(gm, 'B');

InsertVertex(gm, 'C');

InsertVertex(gm, 'D');

InsertVertex(gm, 'E');

InsertVertex(gm, 'F');

InsertEdge(gm, 'A', 'B', 6);

InsertEdge(gm, 'A', 'C', 1);

InsertEdge(gm, 'A', 'D', 5);

InsertEdge(gm, 'B', 'C', 5);

InsertEdge(gm, 'B', 'E', 3);

InsertEdge(gm, 'C', 'D', 5);

InsertEdge(gm, 'C', 'E', 4);

InsertEdge(gm, 'C', 'F', 4);

InsertEdge(gm, 'D', 'F', 2);

InsertEdge(gm, 'E', 'F', 6);

ShowGraph(gm);

}

方法二：Kruskal算法

关键是判断会不会成环，即是不是连接的两个顶点属于相同的连通子图（相同的集合）。

typedef struct Edge {

int x; //弧尾

int y; //弧头

E cost;

}Edge;

//cmp函数C++方法

//int cmp(struct Edge a, struct Edge b) {

// return(a.cost - b.cost);

//}

int cmp(const void \*a, const void \*b) { //const void \*a 定义了一个无类型的指针a，可以指向任何类型的数据

return((\*(Edge\*)a).cost - (\*(Edge\*)b).cost);

}

bool Is\_Same(int \*father, int i,int j) {

while (father[i] != i) { //如果father[i] = i，说明该结点为孤立结点，否则就不是孤立结点，要寻找根结点

i = father[i]; //追溯到根结点

}

while (father[j]!= j) {

j = father[j];

}

return i == j; //如果i==j说明二者根结点相同，则为同一个集合

}

void Mark\_same(int \*father, int i, int j) {

while (father[i] != i) {

i = father[i];

} //一直找到i的根结点（即father[i]等于i）

while (father[j] != j) {

j = father[j]; //一直找到j的根结点（即father[j]等于j）

}

father[j] = i; //将i和j结点归并为同一个父结点(即i的父结点)，形成连通子图

}

void MinSpanTree\_Kruskal(GraphMtx& g) {

int n = g.NumVertices;

Edge\*edge = (Edge\*)malloc(sizeof(Edge)\*(n\*(n - 1) / 2));

assert(edge != nullptr);

int k = 0;

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

for (int j = i; j < n; j++) { //这里范围要注意，由于无向图的矩阵是对称的，只要找一半就行

if (g.Edge[i][j] != 0 && g.Edge[i][j] != MAX\_COST) {

edge[k].x = i;

edge[k].y = j;

edge[k].cost = g.Edge[i][j];

k++;

}

}

}

qsort(edge,k,sizeof(Edge),cmp); //快排函数 //edge:对边的内存排序 k:排序的个数 sizeof(Edge):每次排序的结构为边结构的大小

int \*father = (int \*)malloc(sizeof(int)\*n);

assert(father != nullptr);

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

father[i] = i; //初始化：自己的父结点就是自己

}

int v1, v2;

int num = 0;

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

if (!Is\_Same(father,edge[i].x,edge[i].y)) { //当两者不是同一个集合的时候才能相连

v1 = edge[i].x;

v2 = edge[i].y;

cout << g.verticeList[v1] << "-->" << g.verticeList[v2] << ":" << edge[i].cost << endl;

Mark\_same(father, edge[i].x, edge[i].y); //连接完之后要归到同一个父结点

num++;

}

if (num == n - 1) {

break;

}

}

}