1. 图

5.1图的基本概念

有向图：弧 <>

无向图：边 ()

简单图：不存在重复边，不存在顶点到自身的边(数据结构中只讨论简单图)

多重图：和简单图相对

完全图：

无向图：任意两个顶点直接都存在边，n(n-1)/2

有向图：任意两个顶点之间都存在方向相反的两条弧，n(n-1)

子图：G=(V,E) ,G’=(V’,E’)

在无向图中：

若从顶点v到顶点w有路径存在，则称vw是连通的

若图G中任意两个顶点都是连通的，则G为连通图，否则非连通图

极大连通子图（包括所有边）称为连通分量

极小连通子图：既要保持图的连通，又要使得边数最少的子图

在有向图中：

强连通：v到w和w到v都有路径

强连通图：图中任何一对顶点都是强连通的

极大强连通子图称为强连通分量

生成树生成森林

顶点的度

无向图的全部顶点的度之和是边数的两倍

有向图全都顶点的入度和出度之和相等且等于边

边的权和网

稠密图和稀疏图

第一个顶点和最后一个顶点相同的路径称为回路或环

在路径序列中，顶点不重复出现的路径称为简单路径

除第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路称为简单回路

有向树

5.2图的存储及基本操作

顺序存储结构

1. 邻接矩阵法

//顶点表 顶点数和弧数

#define MaxVertexNum 100

typedef char VertexType;

typedef int EdgeType;

typedef struct{

VertexType vex[MaxVertexNum];//顶点表

EdgeType Edge[MaxVertexNum][MaxVertexNum];//邻接矩阵，边表

int vexnum,arcnum;//图当前的顶点数和弧数

}MGragh;

G[i][j] 无权图=1表示(vi,vj)是G的边;=0表示不是边

有权图=u表示是边;=0或∞不是边

#define MaxVertexNum 100

typedef char DataType;

typedef int WeightType;

//用邻接矩阵表示图

typedef struct GNode \*PtrToGNode;//指向这个结点的一个指针

struct GNode{

int Nv;//顶点数

int Ne;//边数

WeightType G[MaxVertexNum][MaxVertexNum];

DataType Data[MaxVertexNum];//存顶点数据

};

typedef PtrToGNode MGraph;//以邻接矩阵存储的图类型

//MGraph初始化

//A.先初始化一个有VertexNum个顶点但没有边的图

typedef int Vertex;//用顶点下标表示顶点，为整型

MGraph CreateGraph(int VertexNum){

Vertex V,W;//VM是顶点类型，与真正的整型区分开

MGraph Graph;

//1.图的结点

//\*这里默认的顶点编号从0开始，到(Graph->Nv-1)结束

Graph=(MGraph)malloc(sizeof(struct GNode));

Graph->Nv=VertexNum;

Graph->Ne=0;

//2.加入边

for(V=0;V<Graph->Nv;V++){

for(W=0;W<Graph->Nv;W++)

Graph->G[V][W]=0;//或者INFINITY当是有权的时候

}

return Graph;

}

//B.向Mgraph插入边

typedef struct ENode \*PtrToENode;

struct ENode{

Vertex V1,V2;//有向边<V1,V2>

WeightType Weight;//权重

};

typedef PtrToENode Edge;

void InsertEdge(MGraph Graph,Edge E){

//插入边<V1,V2>

Graph->G[E->V1][E->V2]=E->Weight;

//若是无向图，还要插入边<V2,V1>

Graph->G[E->V2][E->V1]=E->Weight;

}

//C.完整的建立MGraph

//输入格式Nv Ne，V1 V2 Weight

MGraph BuildGrapg(){

MGraph Graph;

Edge E;

Vertex V;

int Nv,i;

//1.输入顶点数

scanf("%d",&Nv);

Graph=CreateGraph(Nv);//初始化一个没有边的图

scanf("%d",&(Graph->Ne));//读入边数

if(Graph->Ne!=0){

E=(Edge)malloc(sizeof(struct ENode));//声明临时存边的结点

for(i=0;i<Graph->Ne;i++){//把边的消息读入

scanf("%d %d %d",&E->V1,&E->V2,&E->Weight);

InsertEdge(Graph,E);//插入边

}

}

//如果顶点有数据的话，读入数据

for(V=0;V<Graph->Nv;V++){

scanf("%c",&(Graph->Data[V]));

}

return Graph;

}

//简化版

typedef int MAXN;

int G[MAXN][MAXN],Nv,Ne;

void BuildGraph(){//全局变量所以不用返回了直接用void

int i,j,v1,v2,w;

scanf("%d",&Nv);

//CreateGraph

for(i=0;i<Nv;i++)

for(j=0;j<Nv;j++)

G[i][j]=0;//或INFINITY

scanf("d",&Ne);

for(i=0;i<Ne;i++){

scanf("%d %d %d",&v1,&v2,&w);

//InsertEdge

G[v1][v2]=w;

G[v2][v1]=w;

}

}

链接存储结构

1. 邻接表法

//顶点表 顶点域 边表头指针

//边表 邻接点域 指针域

#define MaxVertexNum 100

typedef struct ArcNode{

int adjvex;//邻接点域 该弧所指向的顶点的位置

struct ArcNode \*next;//指针域 指向下一条弧的指针

}ArcNode;//边表结点

typedef struct VNode{

VertexType data;//顶点域 顶点信息

ArcNode \*first;//指针域 指向第一条依附于该顶点的弧的指针

}VNode,AdjList[MaxVertexNum];//顶点表结点

typedef struct{

AdjList vertices;//邻接表

int vexnum,arcnum;//图的顶点数和弧数

}ALGraph;//ALGraph是以邻接表存储的图类型

邻接表：G[N]为指针数组，对应的矩阵每行一个链表，只存非0元素

#define MaxVertexNum 100

typedef int DataType;

typedef int Vertex;

typedef int WeightType;

typedef struct AdjVNode \*PtrToAdjVNode;

struct AdjVNode{

Vertex AdjV;//邻接点下标

WeightType Weight;//边权重

PtrToAdjVNode Next;

};

typedef struct VNode{

PtrToAdjVNode FirstEdge;

DataType Data;

}AdjList[MaxVertexNum];//AdjList是邻接表类型

typedef struct GNode \*PtrToGNode;

struct GNode{

int Nv;//顶点数

int Ne;//边数

AdjList G;//邻接表

};

typedef PtrToGNode LGraph;//以邻接表方式存储的图类型

//LGraph初始化

//A.初始化一个没有VertexNum个顶点但没有边的图

typedef int VertexNum;//用顶点下标表示顶点，为整型

LGraph CreateGraph(int VertexNum){

Vertex V,W;

LGraph Graph;

Graph=(LGraph)malloc(sizeof(struct GNode));

Graph->Nv=VertexNum;

Graph->Ne=0;

//\*这里默认编号是从0开始，到(Graph->Nv-1)

for(V=0;V<Graph->Nv;V++){

Graph->G[V].FirstEdge=NULL;

}

return Graph;

}

//B.向LGraph个插入边

//把一个顶点插入到一个链表里面

typedef struct ENode \*PtrToENode;

struct ENode{

Vertex V1,V2;//有向边<V1,V2>

WeightType Weight;//权重

};

typedef PtrToENode Edge;

void InsertEdge(LGraph Graph,Edge E){

struct AdjVNode \*NewNode;

//\*\*\*\*\*\*插入边<v1,v2>\*\*\*\*\*8

//为v2建立新的邻接点

NewNode=(PtrToAdjVNode)malloc(sizeof(struct AdjVNode));

NewNode->AdjV=E->V2;

NewNode->Weight=E->Weight;

//将v2插入v1的表头

NewNode->Next=Graph->[E->V1].FirstEdge;

Graph->G[E->V1].FirstEdge=NewNode;

//\*\*\*\*\*\*若是无向图，还要插入<v2,v1>\*\*\*\*\*\*

//为v1建立新的邻接点

NewNode=(PtrToAdjVNode)malloc(sizeof(struct AdjVNode));

NewNode->AdjV=E->V1;

NewNode->Weight=E->Weight;

//将v1插入v2的表头

NewNode->Next=Graph->[E->V2].FirstEdge;

Graph->G[E->V2].FirstEdge=NewNode;

}

//C.完整的建立LGraph

复杂和简化版同上

1. 十字链表
2. 邻接多重表
3. 图的基本操作

Adjacent(G,x,y)判断图中是否存在边xy

Neighbors(G,x)列出G中与结点x邻接的边

InsertVertex(G,x)插入顶点x

DeleteVertex(G,x)删除顶点x

AddEdge(G,x,y)若边xy不存在则添加

RemoveEdge(G,x,y)若边xy存在则删除

FirstNeighbor(G,x)求图中顶点x的第一个邻接点，有则返回顶点号，无则返回-1

NextNeighbor(G,x,y)假设y是x的一个邻接点，返回除y之外x的下一个邻接点的顶点号，，若y是x的最后一个则返回-1

//邻接表转成矩阵

//假设图的顶点分别存储在v[n]数组中，

void Convert(ALGraph &G,int arcs[M][N]){

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

p=(G->v[i]).firstarc;

while(p!=NULL){

arci[i][p->data]=1;

p=p->nextarc;

}

}

}

5.3图的遍历

1. 广度优先搜索BFS (优先考虑最早发现的顶点)

是一种分层的查找过程，没有回退的情况，因此不是一个递归地算法，为了实现逐层的访问，借助辅助队列

bool visited[MAX\_VERTEX\_NUM];//访问标记数组

void BFSTraverse(Graph G){

//对图G进行广度优先遍历，设访问函数为visit()

for(i=0;i<G.vexnum;++i)visited[i]=FALSE;//访问标记数组初始化

InitQueue(Q);//初始化辅助队列Q

for(i=0;i<G.vexnum;++i){//从0号顶点开始遍历

if(!visited[i])//对每个连通分量调用一次BFS

BFS(G,i);//vi未访问过，从vi开始BFS

}

}

void BFS(Graph G,int v){

//从顶点v出发，广度优先遍历图G，算法借助一个辅助队列Q

visit(v);//访问初始顶点v

visited[v]=TRUE;//对v做已标记访问

Enqueue(Q,v);//v入队

while(!isEmpty(Q)){

Dequeue(Q,v);

for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w)){//检测v所有邻接点

if(!visited[w]){//w为v的尚未访问的邻接顶点

visit(w);//访问w

visited[w]=TRUE;//对w做以标记访问

EnQueue(Q,w);

}//if

}//for

}//while

}

空间复杂度:O(|v|)

时间复杂度:

邻接矩阵：O(|V2|)

邻接表:O(|V|+|E|)

Dijkstran单元最短路径算法

最短路径d（u，v）为任何路径中最少的边数，若没有通路=∞

void BFS\_MIN\_Distance(Grapg G,int u){//非带权

//d[i]表示u到i结点的最短路径

for(i=0;i<G.vexnumn;i++)

d[i]=∞;//初始化路径长度

visited[u]=TRUE;d[u]=0;

EnQueue(Q,u);

while(!isEmpty(Q)){//BFS算法主过程

DeQueue(Q,u);

for(w=FirstNeighbor(G,u);w>=0;w=NextNeighbor(G,u,w)){

if(!visited[w]){//w为u的未访问的邻接顶点

visited[w]=TRUE;//设已访问标记

d[w]=d[u]+1;//路径长度加1

EnQueue(Q,w);

}

}

}

}

Prim最小生成树算法(广度优先生成树)

1. 深度优先搜索DFS

bool visited[MAX\_VERTEX\_NUM]//访问标记数组

void DFSTraverse(Graph G){

//对图G进行深度优先遍历，访问函数为visit()

for(v=0;v<G.vexnum;++v)

visited[v]=FALSE;//初始化已访问标记数据

for(v=0;v<G.vexnum;++v){

if(!visited[v])

DFS(G,v);

}

}

void DFS(Graph G,int v){

//从顶点v出发，采用递归思想，深度优先遍历图G

visit(v);

visited[v]=TRUE;//设已访问标记

for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w)){

if(!visited[w]){//w为u的尚未访问的邻接顶点

DFS(G,w);

}//if

}//for

}//DFS

空间复杂度:O(|v|)

时间复杂度:

邻接矩阵：O(|V2|)

邻接表:O(|V|+|E|)

深度优先生产树(连通图)和生成森林(非连通图)

1. 图的遍历与图的连通性

图的遍历算法可以用来判断图的连通性

故在BDSTraverse()或DFSTraverse()中添加第二个for循环，在选取初始点，继续进行遍历，以防止一次无法遍历图的所有顶点。

// 2.判断一个无向图G是否为一颗树

//树的条件：无回路的连通图或者有n-1条边的连通图

//对连通的判定：用遍历全部顶点来实现 采用DFS

bool isTree(Graph& G){

for(i=1;i<=G.vexnum;++i)

visited[i]=FALSE;//初始化标记visited[]初始化

int Vnum=0,Enum=0;//记录顶点数和边数

DFS(G,1,vnum,visited);

if(Vnum==G.vexnum&&Enum==2\*(G.vexnum-1))return true;

else return false;

}

void DFS(Graph &G,int v,int &Vunm,int &Enum,int visited[]){

//深度优先遍历图G，统计访问过得顶点数和边数，通过Vnum和Enum返回

visited[v]=TRUE;Vnum++;//作访问标记，顶点计数

int w=FirstNeighbor(G,v);//取v的第一个邻接顶点

while(w!=-1){//当邻接顶点存在

Enum++;//边存在，边计数

if(!visited[w]){//当该邻接顶点未访问过

DFS(G,w,Vnum,Enum,visited);

}

w=NextNeighbot(G,v,w);

}

}

//3.图的DFS算法的非递归算法(采用邻接表形式)

//借用栈S，记录下一步可能访问的顶点，用visited[i]访问标记数组

//在visited[i]中记忆第i个顶点是否在栈内或曾经在栈内。若是，则不能在进栈

//由于使用栈，使得遍历的方式是从右端到左端的顺序进行的

void DFS\_Non\_Rc(AGraph &G,int v){

int w;//顶点序号

InitStack(S);

for(i=0;i<G.vexnum;i++)

visited[i]=FAlSE;//初始化visited

Push(S,v);visited[v]=TRUE;//v进栈并置visited[v]

while(!isEmpty(S)){

k=Pop(S);//栈中退出一个顶点

visit(k);//先访问，在讲其子结点入栈

for(w=FirstNeighbot(G,k);w>=0;w=NextNeighbor(G,k,w)){//k所有邻接点

if(!visited[w]){//未进过栈的顶点进栈

Push(S,w);

visited[w]=true;//做标记，以免再次入栈

}//if

}//for

}//while

}

//4.用DFS和BFS以邻接表方式存储的有向图是否有由vi到vj的路径(ij不等)

//5.用邻接表表示，输出从顶点vi到vj的所有简单路径

//

//

5.4图的应用

1. 最小生成树MST Minimum-Spanning-Tree

不带权：包含所有顶点，边数最少

带权：权值之和最小的那颗生成树

1. 最小生成树不是唯一的(即树形不唯一)

但当各边权值互不相等时，G的最小生成树是唯一的

1. 权值之和总是唯一的，且是最小的
2. 边数为顶点数减1

GENERIC\_MST(G){

T=NULL;

while T 未形成一颗生成树;

do找到一条最小代价边(u，v)并且加入T后不会产生回路;

T=T∪(u,v);

}

都是贪心算法

1. 普利姆(Prim)算法O(|V|2)

从一点开始，每次都是与这点最小的边，在看下一个点连接的边是否比现在小

适合稠密的图

void Prim(G,T){

T=∅;//初始化空树

U={w};//任何一顶点

while((U-V)!=∅){

]

}

1. 克鲁斯卡尔(Kruskal)算法 O(|Elog|E||)

按权值的递增次序选边，不构成回路即可

适合边稀疏而顶点多的图

1. 最短路径

源点到到终点权值之和最小的一条

1. 单源最短路径问题

从某固定源点出发，求其到所有顶点的最短路径

A.无权图

按照递增(非递减)的顺序找出到各个顶点的最短路(BFS)

dist[S]= S到W的最短距离

dist[S]=0

path[W]= S到w的路上经过的某顶点

void UnWeighted(Vertex S){//源点S

dist[]=-1;path[]=-1;

EnQueue(S,Q);

while(!isEmpty(Q)){

V=DeQueue(Q);//最短算好了的

for(V的每个邻接点W){

if(disit[W]==-1){//未被访问过

dist[W]=dist[V]+1;

Path[W]=V;//V是W前一个顶点的编号

EnQueue(W,Q);

}

}

}

}

时间复杂度T=O(|E|+|V|)

B.有权图

按照递增(非递减)的顺序找出到各个顶点的最短路

Dijkstra算法

\*令S={源点S+已经确定了最短路径的顶点Vi}

\*对任一未收录的顶点v，定义dist[v]为s到v的最短路径长度，

但该路径仅经过s中的顶点。即路径{s->(Vi∈S)->v}的最小长度

s[]记录已求得的最短路径的顶点，初始化为0

\*若路径是按照递增(非递减)的顺序生成的，则

1. 真正的最短路径必须只经过S中的顶点
2. 每次从未收录的顶点中选一个dist最小的收录（贪心）
3. 增加一个v进入s，可能会影响另外一个w的dist值

disit[w]=min{dist[w],dist[v]+<v,w>的权重}

如果收录v使得s到w的路径变短，则：s到w的路径一定经过v，且v到w有一条边

1. **void** Dijkstra( Vertex s )
2. { **while** (1) {
3. V = 未收录顶点中dist最小者;
4. **if** ( 这样的V不存在 )
5. **break**;
6. collected[V] = **true**;
7. **for** ( V 的每个邻接点 W )
8. **if** ( collected[W] == **false** )
9. **if** ( dist[V]+E<V,W> < dist[W] ) {
10. dist[W] = dist[V] + E<V,W> ;
11. path[W] = V;
12. }
13. }
14. }//不能解决有负边的情况

T=O(?)不科学

方法一：直接扫描所有未收录顶点——O(|V|)

T=O(|v|2+|e|)(对于稠密图效果较好)

方法二：将dist存在最小堆中——O(log|V|)

更新dist[w]的值——O(log|V|)

T=O(|V|log|V|+|E|log|V|)=O(|E|log|V|)(对于稀疏图效果好)

dist[]记录从源点v0到其余各顶点当前的最短路径长度，初值为arcs[v0][i]

path[];path[i]表示从源点到顶点i之间的最短路径的前驱结点

1. 多源最短路径问题

求任意两顶点间的最短路径

方法一：直接将单源最短路算法调用|V|遍

T=O(|V|3+|E|\*|V|)对于稀疏图较好

方法二：FLoyd算法

T=O(|V|3)稠密图

Floyd算法

\*Dk次方[i][j]=路径{i->{l<=k}->j}的最小长度

\*D0，D1。。。。Dv-1次方[i][j]即给出了i到j的真正最短距离

\*最初的D-1次方是什么？

D矩阵应该初始化为什么？带权的邻接矩阵，对角元是0

如果i和j之间没有直接的边，D[i][j]应该定义为正无穷

void Floyd(){

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

}

}

1. 拓扑排序
2. 关键路径