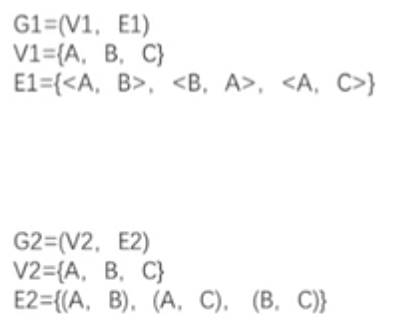
# 图的基本概念

图的表示 G=(V,E)

V表示图中点 且一定不为空（图中一定有顶点）顶点个数称为阶 |V|

E表示边的集合 |E| 一个图可以没有边 但是必须有顶点



有向图/无向图

有向图 弧头v 弧尾w <v,w>

有向图用尖括号 无向图用圆括号

简单图：不允许

1. 自身到自身的边
2. 同一条边重复出现

多重图：允许

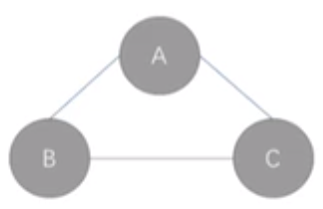
（1）自身到自身的边

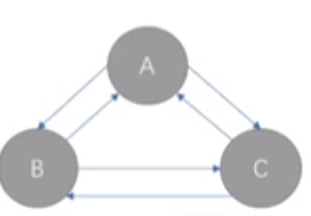
（2）同一条边重复出现



完全图

无向完全图：任意两个顶点都存在边

 由于每个结点都与其他N-1个结点链接，一共有N个结点，所以一共有N条边，但是由于多算了一次所以要除以2，故一共有N(N-1)/2条

有向完全图：任意两个节点都存在方向相反的弧

由于每个结点到其他的任意结点都有两条弧，所以不用除以2

所以有N（N-1）条边

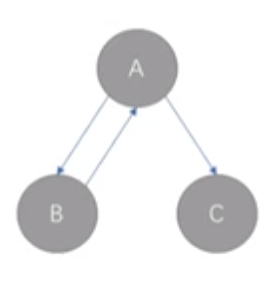
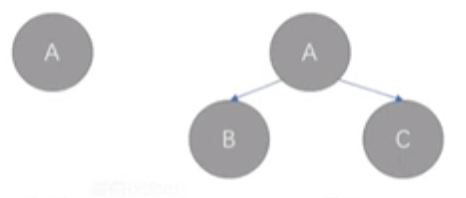
子图：

所有的V顶点都是另一图的子集（不能为空，因为图必须有顶点）

所有的E都是边集的子集，可以为空

那么改图就是另一个图的子图

生成子图 在子图的基础上，所有的V都包含在里面，就是生成子图

   
有可能无法形成图：

G=(V,E)

V={A,B,C}

E={(A,B,),(A,C),(B,A)}

那么子图

G=(V,E)

V={C}

E={(A,B),(B,A)}就不能生成一个图

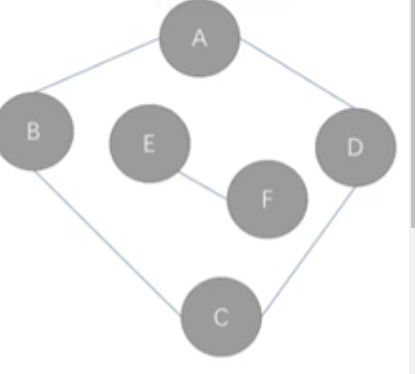
对于无向图：

连通：

顶点v到v·有路径

连通图：任意两个顶点都是连通的

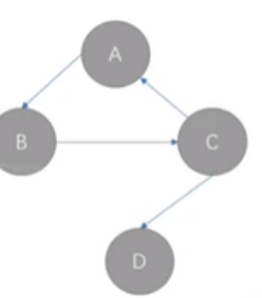
连通分量，无向图中的极大联通子图

1. 子图
2. 连通
3. 极大 指的是顶点足够多 并且包含依附的边

这里面 极大连通子图为 EF 和 ABCD

连通图的最小边情况是

极小联通子图 有n个顶点 n-1条边 也就是树的情况 再小就不连通了

对于有向图：

强连通：任意结点 从 v到w 和 w到v都要有路径

强连通分量

1 子图 2 强连通 3 极大 abc – d

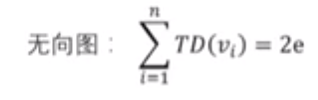
连通图的生成树 包含全部n个结点 n条边的极小联通子图

多一条边 形成回路 少一条边 形成森林

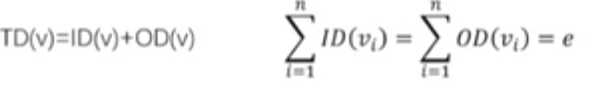
非连通图的生成森林，每个联通分量构成森林

度

1. 无向图 依附于该顶点的边数

度数之和为边数2倍

1. 有向图 出度 出去的度数 入度 进来的度数



顶点的度数等于出度+入度

并且所有顶点的入读或者出度等于e

有向树：一个顶点入度为0 其余顶点入度为1 为有向图

边可以加权 带权图称为网

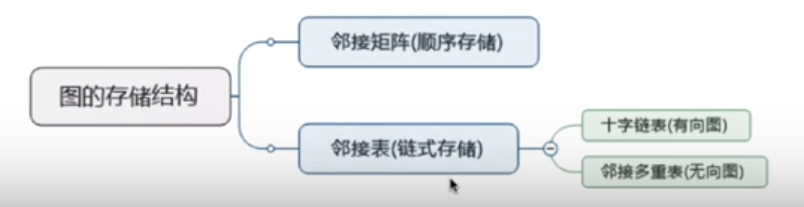
路径 顶点序列 边的数目是长度

回路 环 第一个和最后一个顶点相同的路径

简单路径/简单回路 不许有重复顶点 除了第一个和最后一个不行有重复结点

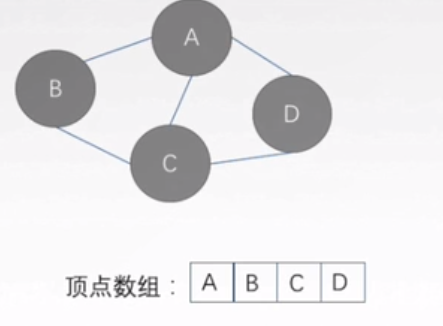
n-1 n

距离 两个结点路径的最小长度，不存在无穷



邻接矩阵：

顶点：一维数组

边：二维数组 邻接矩阵

邻接矩阵（i，j）=1 表示有边 0则无边

无向图对称

度数：比如求A 的度数 将一整行相加

有向图 不是对称的

入度是所在一列的数字和

出度是所在一行的数字和



带权图可以由邻接矩阵存储

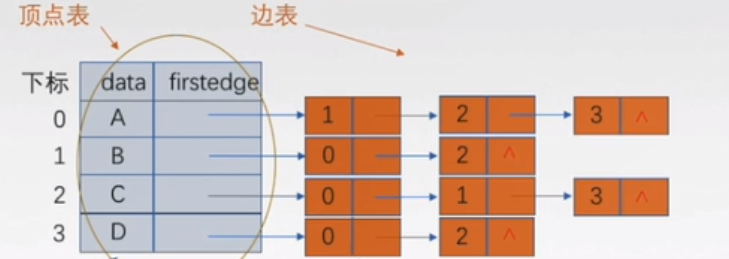
1. 带权边
2. 行列相同为0
3. 不存在的边 那么无限大

时间复杂度为

O（n^2）

稀疏图 即E远小于|V|^2

邻接表

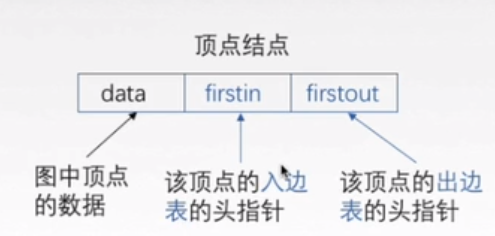


有向图称为出边表、只能反映出边的问题，不反应入度的问题，所以要将所有的边表都遍历，看看有几个指向（入度）

无向图没有入度出度 没有这个问题

# 十字链表&临界多重表

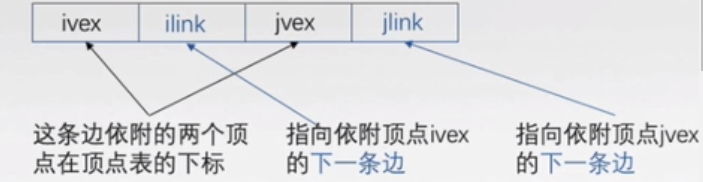
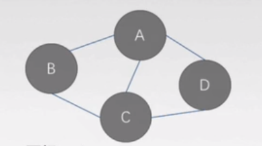
十字链表 对有向图的优化

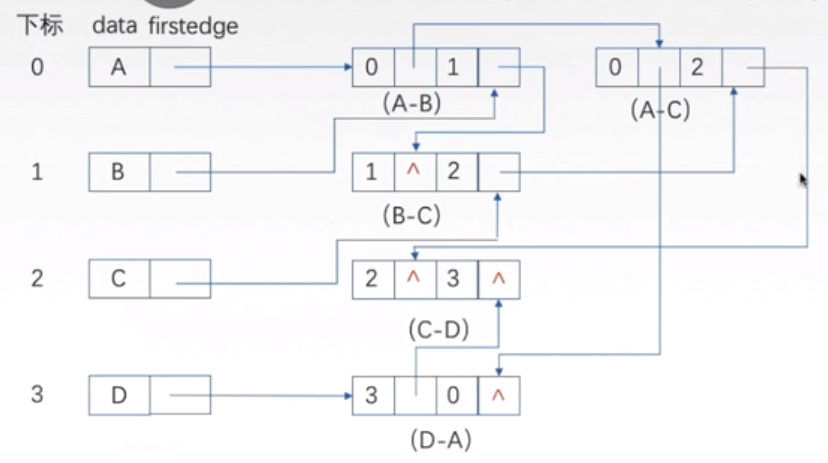




邻接多重表

有向图邻接表 修改比较麻烦





删除的时候，找到边表的指入指针，并且看它是ivex 还是 jvex 并且将相应的指针接过去即可

# BFS DFS

从任意一个结点出发 访问所有结点

设置访问数组Visited[] 表示顶点是否被访问

BFS 队列

bool BFSVisited[MAX\_VEX\_NUM] = {false};

int BFSCount = 0;

//设置一个counter来记录已经访问元素的个数

int BFSTravse(GraphPtr&graph,char v)

{

if (!graph)exit(1);

//如果图不存在，则退出程序

if (DG == graph->type) {

printf("图的类型不能是DG，请选用其他的遍历方式");

return -1;

//如果图的类型为有向图，那么返回-1

}

//初始化一个队列

QNodePtr queue;

LinkNode linkNode;

InitQueue(queue, linkNode);

//从c结点开始查找

printf("广度优先搜索开始，将从%c结点开始搜索\n", v);

BFSVisited[IndexOfVex(graph, v)] = true;

//Visited[]数组中该元素置true、表示已经访问过

EnQuene(linkNode, IndexOfVex(graph,v));

//将该节点入队（进入的是编号）

while (linkNode.num!=0) // 只要队列里有元素，就证明还有元素没有被访问到，则继续循环遍历

{

int cache;//设置一个缓存量来接收出队编号

DeQueue(linkNode, cache);

printf("BFS: [ %c ]", graph->vex[cache].Data);

//打印数据

BFSCount++;//counter自增

//现在要将raph->vex[cache].Data所指向的数据都入队，等待遍历

GraphNodePtr curPointer = graph->vex[cache].Connect;

//设置curPointer指针

while (curPointer != NULL) //检索到底

{

if (false == BFSVisited[IndexOfVex(graph,curPointer->Data)]) {

//如果该结点没有被检索过，进入循环

EnQuene(linkNode, IndexOfVex(graph,curPointer->Data));

//这个结点入队列

BFSVisited[IndexOfVex(graph, curPointer->Data)]=true;

//将这个结点设置为访问过

}

curPointer = curPointer->next;

//指针自增到下一个元素

}

}

return 1;

}

void BFSRecover()

{

for (int i = 0; i < MAX\_VEX\_NUM; i++)

BFSVisited[i] = false;

BFSCount = 0;

}

int BFS(GraphPtr&graph, int v)

{

int r = BFSTravse(graph, v);

if (BFSCount == graph->vexNum)

printf("广度优先搜索结束.\n");

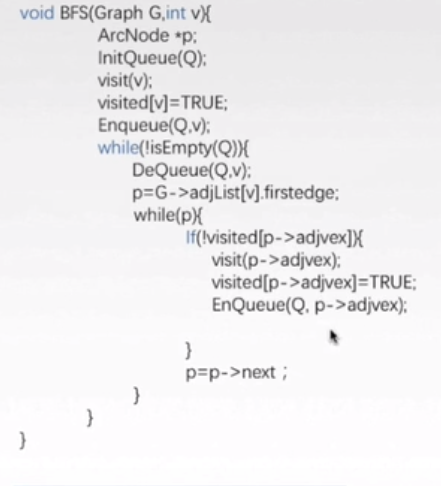
else if (BFSCount < graph->vexNum)

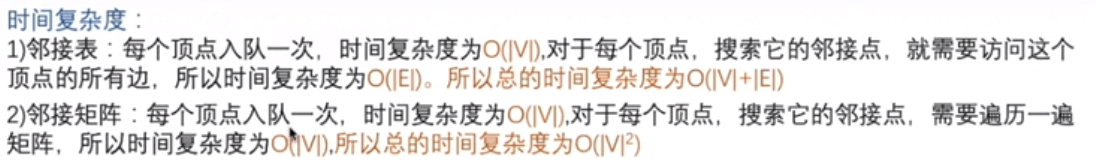
printf("广度有线索搜失败，图不是联通的\n");

BFSRecover();

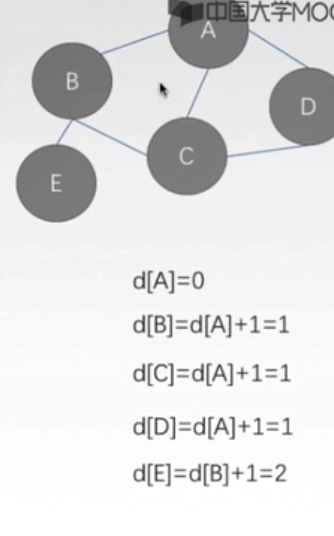
return r;

}

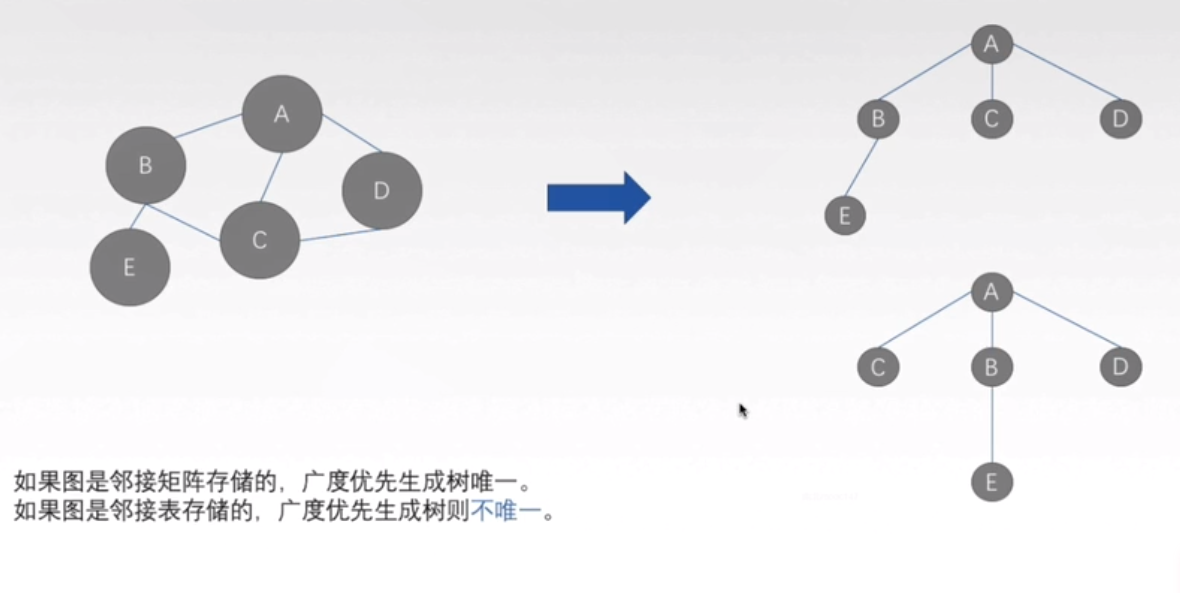




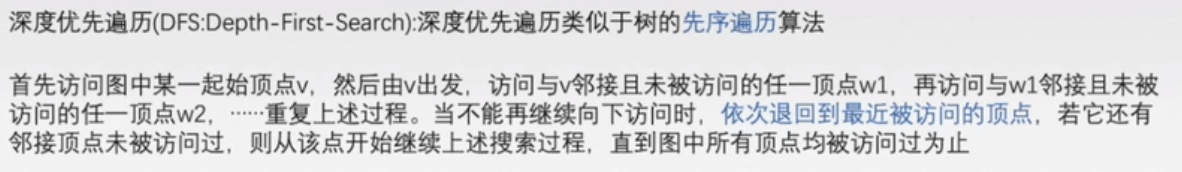
BFS应用 单源非带权图最短路径



# 广度优先生成树



# DFS





//深度优先搜索的递归实现

bool Visited[MAX\_VEX\_NUM] = {false};

int dfsCount = 0;//记录由多少结点已经被访问

//设置一个全局的Visited[]数组，当递归实现DFS的时候，搜索完一个结点就将其赋值true，这样所有的函数都可以公用该数组

int DFSTravse(GraphPtr&graph,char v)//从v这个点开始搜索

{

if (!graph)exit(1); //图不存在，结束程序

if (DG == graph->type) {

printf("有向图不能DFSTrave()遍历\n");

return -1;

}

Visited[IndexOfVex(graph,v)] = true;

//先访问输入的v结点，即先从v结点开始访问

if (0 == dfsCount) {

printf("深度优先搜索将从%c结点开始遍历\n", v);

}

printf("DFS: [ %c ]\n",v);

//打印v的内容

dfsCount++;

GraphNodePtr curPointer = graph->vex[IndexOfVex(graph, v)].Connect;

//设置搜索指针，便利搜索所有与v结点连接的结点

//即从GraphBox中v在的元素的指针域开始搜索，直到结尾NULL为止

while(curPointer != NULL)

{

if (false == Visited[IndexOfVex(graph, curPointer->Data)])

{

DFSTravse(graph, curPointer->Data);

//如果与v连接的某节点没有被访问过，那么使用递归的方式调用

//DFSTravse函数进行搜索，直到返回再继续

}

curPointer = curPointer->next;

//curPointer指针向后移动一个元素

}

return 1;

}

//用来恢复数组和count

void DFSRecover()

{

for (int i = 0; i < MAX\_VEX\_NUM; i++)

{

Visited[i] = false;

}

dfsCount = 0;

}

int DFS(GraphPtr&graph,char v)

{

int r=DFSTravse(graph, v);

if (dfsCount == graph->vexNum)

printf("深度优先搜索结束\n");

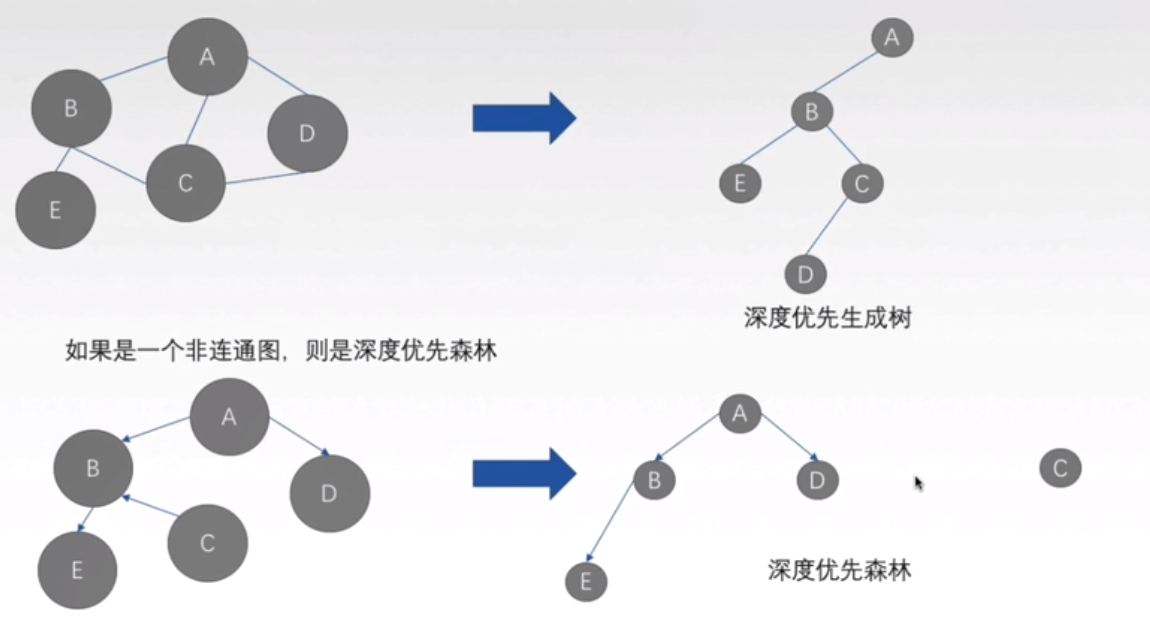
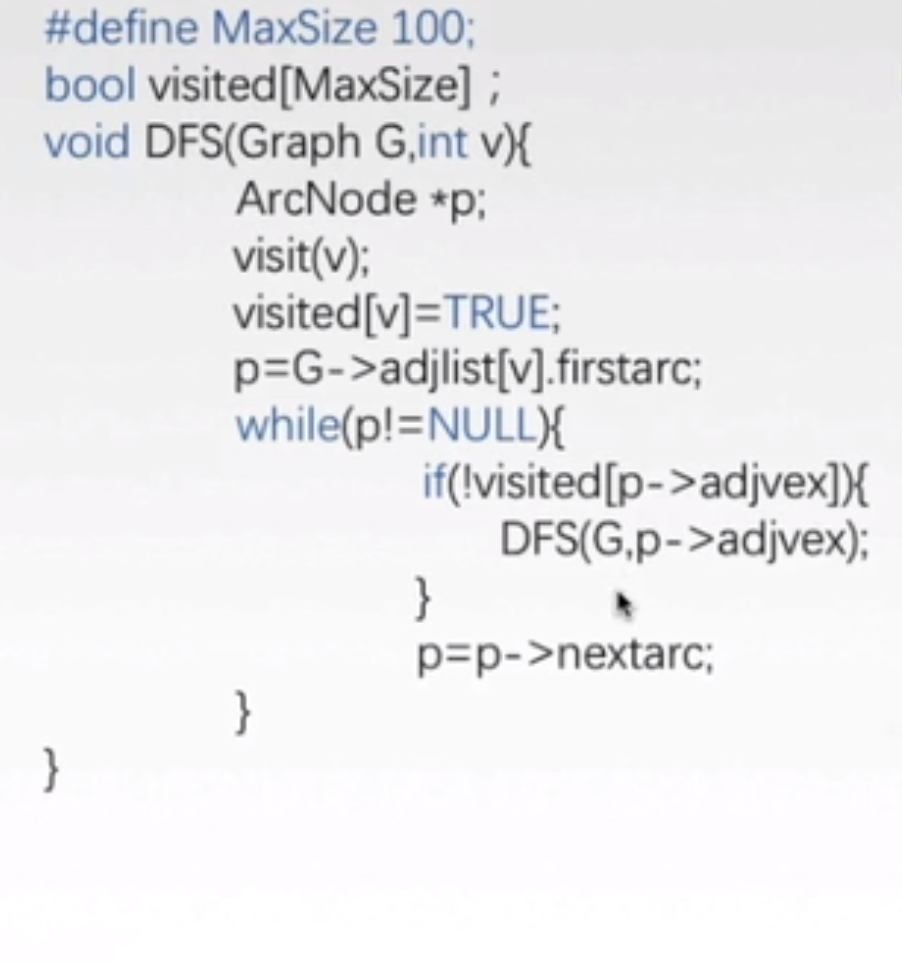
else if (dfsCount < graph->vexNum)

printf("图不连通，深度优先搜索失败\n");

DFSRecover();

return r;

}



# 图的应用

最小生成树 连通图的生成树是连通图的最小联通子图

包含全部节点，n-1条边

生成树是不唯一的

对于带权值的图，众多生成树中最小的为最小生成树

普利姆算法

需要两个数组

lowcost[ n ] 保存当前的生成树 （某个阶段），记录到各个顶点的权值

adjvex[ n ] 邻接顶点 找到边的两个顶点

1. 选择一个起始顶点v0 作为第一个顶点 ，从这个顶点到其他所有结点中选一个权值最小的边 ，把这条边加入生成树中，另一个结点记录为 v
2. 对剩下的其他所有顶点，分别检查这些顶点与顶点v的权值是否比这些顶点在lowcost数组对应的权值小 如果更小用更小的值更新lowcast数组
3. 从更新后的lowcast数组中继续挑选权值最小而且不在树种的边，然后加入生成树
4. 反复执行 2 3 4

两个数组的作用

Lowcast[n]用来记录当前比较结点与其他结点之间路径的权值

Adjvex[n]用来记录当前lowcast[n]对应的结点的连线的另一端结点

开始的时候 选定第一个结点 对于图 A—B 20 A—C 30 B—C 40来说

第一次选A

也就是 lowcast[3]={infinate，20，30}

并且 adjvex[3]={0,B,C}

这时候找到最小的为20 （查找lowcast[n]）打印输出 adjvex(B)—A

将lowcast设置为{infinate，0，30}

查找B，发现B也为0了 找C 发现A与C链接 设为40 并且更新adjvex为{A,B,C}

这时候打印C—A

也就是说 只有在有更小的结点的时候才更新 如果没有则不更新，所以adjvex里面总记录预期链接的最小的结点



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | INF | INF | INF | INF | INF | INF | INF |
| Adjvex | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

k=0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 5 | 11 | INF | INF | INF | INF |
| Adjvex | 0 | 0号 | 0号 | 0 | 0 | 0 | 0 |

输出 0—1

K=1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 0 | 11 | 12 | 2 | 8 | INF |
| Adjvex | 0 | 0号 | 0号 | 1 | 1 | 1 | 0 |

打印1—5

K=5

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 0 | 11 | 12 | 2 | 0 | 3 |
| Adjvex | 0 | 0号 | 0号 | 1 | 1 | 1 | 5 |



打印 5—4

K=4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 0 | 11 | 12 | 0 | 0 | 0 |
| Adjvex | 0 | 0号 | 0号 | 1 | 1 | 1 | 5 |

打印 4—6

K=6

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 0 | 11 | 12 | 0 | 0 | 0 |
| Adjvex | 0 | 0号 | 0号 | 1 | 1 | 1 | 5 |

打印 6—2

K=2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 0 | 0 | 12 | 0 | 0 | 0 |
| Adjvex | 0 | 0号 | 0号 | 1 | 1 | 1 | 5 |

K=2

打印 2—3 12

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Lowcast | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Adjvex | 0 | 0号 | 0号 | 1 | 1 | 1 | 5 |

打印完成

**/\*有关prim算法\*/**

**#define PRIM\_SIZE 30**

**void Prim(MGraphPtr&graph)**

**{**

**int countend = 0;**

**int n = graph->vexNum;**

**int lowcast[PRIM\_SIZE];**

**//设置lowcast，起始节点到该节点的权值**

**int adjvex[PRIM\_SIZE];**

**//设置adjvex[]表示改结点链接的边的权值 0-INFINATE**

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

**{**

**lowcast[i] = INFINATE;**

**adjvex[i] = 0;**

**}**

**int min = 0;**

**//设置lowcast[]数组**

**int k = 0;**

**for (int c = 0; c < n; c++)**

**{**

**countend = 0;**

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

**{**

**if (lowcast[i] == 0)**

**countend++;**

**}**

**if (countend == n)**

**break;**

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

**{**

**if (lowcast[k]!=0&&lowcast[i]!=0&&graph->arcs[k][i] < lowcast[i])**

**{**

**lowcast[i] = graph->arcs[k][i];**

**adjvex[i] = graph->vex[k];**

**}**

**}**

**min = 0;**

**int Min = INFINATE;**

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

**{**

**if (lowcast[i]!=0&&Min >lowcast[i])**

**{**

**Min = lowcast[i];**

**min = i;**

**}**

**}**

**if (lowcast[min] != 0) {**

**printf("%c --- %c\n", graph->vex[min], adjvex[min]);**

**lowcast[k] = 0;**

**lowcast[min] =0;**

**}**

**k = min;**

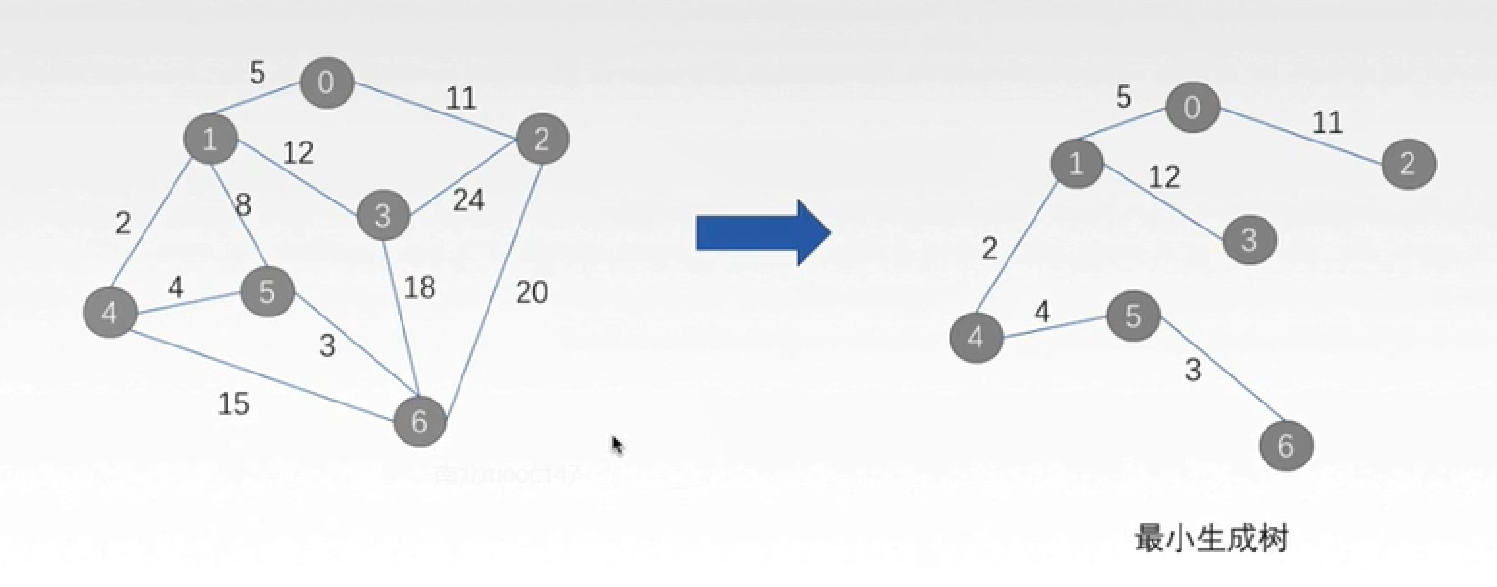
**}**

**}**

**复杂度都是O（n^2）且有且只与定点有关 所以适用于稠密图**

**克鲁斯卡尔算法**时间复杂度：elog2e  e为图中的边数

**回路 – 拓扑排序**



**如何判断有无回路？**

**类似双亲表示法：**

**使用并查集：**

|  |
| --- |
| **-4 0** |
| **0 1** |
| **1 2** |
| **1 3** |

|  |
| --- |
| **-4 4** |
| **4 5** |
| **4 6** |
| **4 7** |

**这样找 3（1）-> a(0)**

**7(4)->4(-4)**

**可以合并 也就是将一个集的开头设置为另一个集合的开头**

**Array[root2]=root1**

|  |
| --- |
| **-4 0** |
| **0 1** |
| **1 2** |
| **1 3** |

|  |
| --- |
| **0 4** |
| **4 5** |
| **4 6** |
| **4 7** |

**这样通过 1->0 和 4->0都能找到0，这样再加一根线就会成环**

**如果不相同，则不会成环 每个项目记录自己双亲**

**查找函数 int Find(int parent[],int X)**

**{**

**While(parent[X]>=0)**

**{**

**X=parent[X];**

**}**

**Return X;**

**}**

**合并 parent[root2]=root1;**





**Parents 0-6 =-1**

**先按照权值排序**

**对于 1-4 w=1 查找表 为1号 w=4 查找表 4**

**I!=4所以输出 ，并且链接**

**Parent[n]=m 也就是 1号写成 4**

**同样的如5-6**

**5带入得 -1 6带入得-6 输出**

**并且 parent[5]=6**

**对于 1-5**

**1->4->6**

**5->-1**

**5->6 不能加了**

**基本思想**：（1）构造一个只含n个顶点，边集为空的子图。若将图中各个顶点看成一棵树的根节点，则它是一个含有n棵树的森林。（2）从网的边集 E 中选取一条权值最小的边，若该条边的两个顶点分属不同的树，则将其加入子图。也就是说，将这两个顶点分别所在的两棵树合成一棵树；反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之（3）依次类推，直至森林中只有一棵树，也即子图中含有 n-1条边为止。

**大白话**：（1）将图中的所有边都去掉。（2）将边按权值从小到大的顺序添加到图中，保证添加的过程中不会形成环（3）重复上一步直到连接所有顶点，此时就生成了最小生成树。这是一种贪心策略。

代码：

Int Find(int \*parents,int x) //find函数是找一个孩子的最根位置的parent 找祖宗

{

While(parents[x]>0){ x=parents[x]}

Return x;

}

void merge(int \*parents ,int m,int n) 合并两个点成一个 边

{

Parents[n]=m;

}

Struct {

Int front;,

Int rear;

Int num;

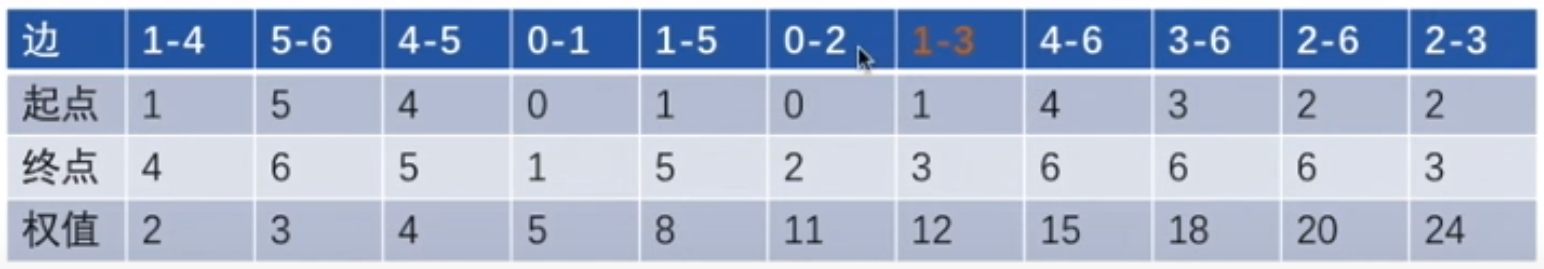
}

Void Min\_Tree\_Kruskal(int \*parents)

{

Edges edges[MAX\_SIZE];

Sort(edges);//将edges排序成以下表格



//初始化parents

For(int i=0;i<G.Vexnum;i++)

{

Parents[i]=-1;

}

For(int i=0;i<G.arcNum;i++)

{

int n=Find(parents,i);

int m=Find(parents ,i);

//如果一条弧的两个顶点不都是一个parents，就可以链接

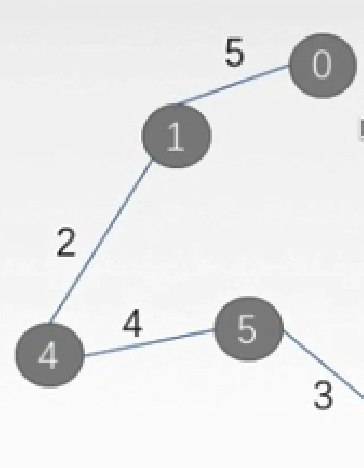
If(m!=n){

Merge(parents,m,n);

Printf(“%d🡪%d\n”,m,n);

}

//如果同一个parents 则什么也不做

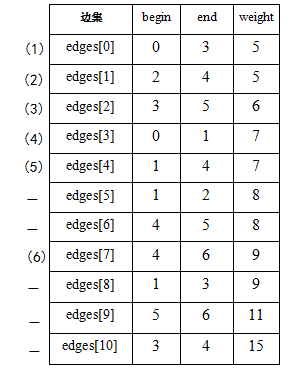
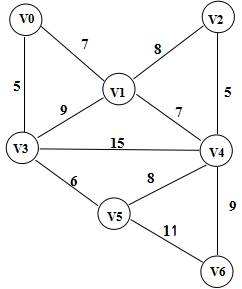
 比如 1-5 不能连接 因为1的parents是6 5的parents为 6

}

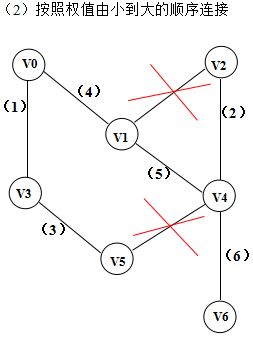
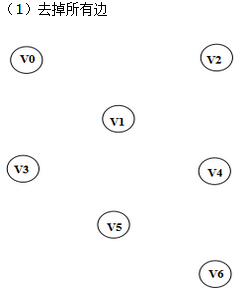
}

**难点**：判断某条边<u, v>的加入是否会在已经选定的边集集合中形成环。

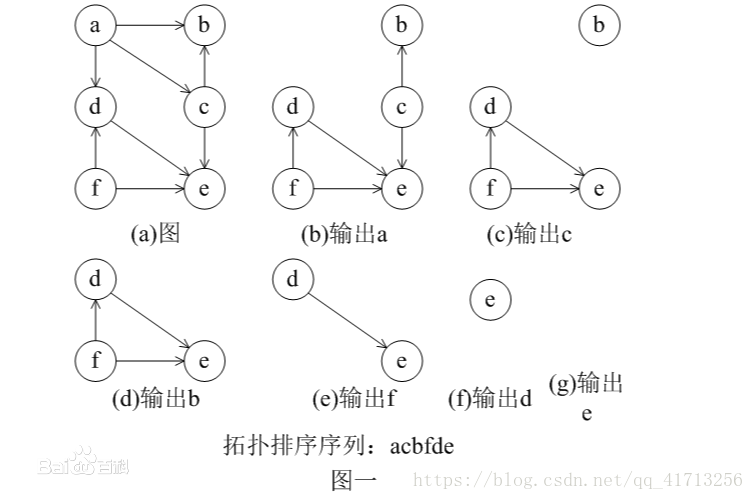
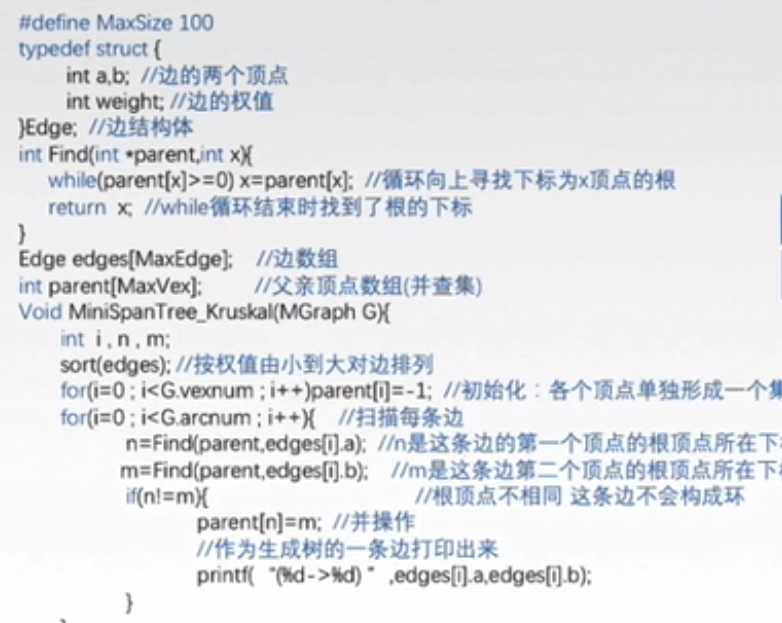
**解决办法**：使用[并查集](http://blog.csdn.net/dellaserss/article/details/7724401)，分别找出两个顶点u, v所在树的根节点。若根节点相同，说明u, v在同一棵树中，则u, v连接起来会形成环；若根节点不同，则u, v不在一棵树中，连接起来不会形成环，而是将两棵树合并。



**去掉所有边**



边<1, 2>和<4, 5>在添加到图中的时候形成了环，所以不能将v1和v2，v4和v5连起来。



# **拓扑排序**

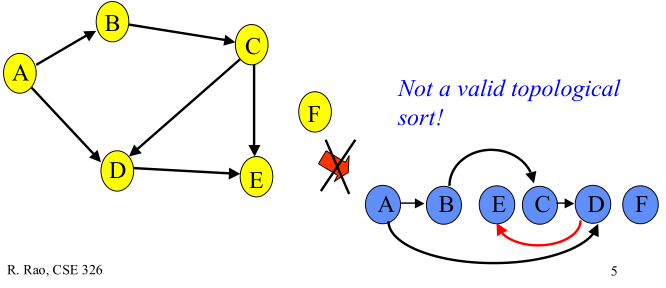
对一个[有向无环图](https://baike.baidu.com/item/%E6%9C%89%E5%90%91%E6%97%A0%E7%8E%AF%E5%9B%BE/10972513)(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边(u,v)∈E(G)，则u在线性序列中出现在v之前。通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。简单的说，由某个集合上的一个[偏序](https://baike.baidu.com/item/%E5%81%8F%E5%BA%8F/2439087)得到该集合上的一个[全序](https://baike.baidu.com/item/%E5%85%A8%E5%BA%8F/10577699)，这个操作称之为拓扑排序。

有向无环图（Directed Acyclic Graph, DAG）是有向图的一种，字面意思的理解就是图中没有环。常常被用来表示事件之间的驱动依赖关系，管理任务之间的调度。拓扑排序是对DAG的顶点进行排序，使得对每一条有向边(u, v)，均有u（在排序记录中）比v先出现。亦可理解为对某点v而言，只有当v的所有源点均出现了，v才能出现。

下图给出有向无环图的拓扑排序：



下图给出的顶点排序不是拓扑排序，因为顶点D的邻接点E比其先出现：



对于DAG的拓扑排序，显而易见的办法：

* 找出图中0入度的顶点；
* 依次在图中删除这些顶点，删除后再找出0入度的顶点；
* 然后再删除……再找出……
* 直至删除所有顶点，即完成拓扑排序

为了保存0入度的顶点，我们采用数据结构

# AOE网（Active On Edge）

顾名思义 活动（Active）用边标识 事件（EVENT）用结点表示

AOE网是一个带权的，有向，无环图

其中 权值代表活动所持续的时间，即只有在边上有权值

每一个结点 代表一个事件 表示在这个结点之前的活动已经完成，并且在这个结点之后的活动可以开始

特点：网中只有一个开始点和一个结束点 即有一个入读为0的点（源点），一个出度为0的点（汇点）

关注问题：完成整个工程至少需要多少时间？

哪些活动是影响工程进度的关键？

工程完成的最短时间是从源点到汇点的最长路径的长度，最长路径叫做关键路径

事件最早发生时间：从开始点v1到该点vi的最长路径（因为必须所有的活动都完成之后才能到达这个点，所以要找最长的，才是最早开始时间）

e(i)表示活动最早开始事件 l(i)表示活动最晚开始时间（不推迟整个工期的情况下）

ai的时间余量 l(i)-e(i) l(i) = e(i)称为关键活动，关键路径上的活动都是关键活动，提高非关键活动的速度无法改变工期的完成时间

对于事件的最早/最晚开始事件用 ve(i) vl(i)表示 对于活动 aj ，由弧<I,k>组成 那么活动aj的持续时间为dut(<I,k>)也就是弧上的权

有关系：

e(j)=ve(i)

l(j)=vl(i)-dut(<I,k>)

其中ve(j)=max{ve(i)+dut(<I,j>)}

从后往前推vl(i) : vl(i)=min{vl(j)-dut(<I,j>)}

例题：

3、 已知如图7-4所示的AOE-网，试求：

* 1. 每个事件的最早发生时间和最晚发生时间；
  2. 每个活动的最早开始时间和最晚开始时间；
  3. 给出关键路径。

|  |
| --- |
| 4 |

|  |
| --- |
| 4 |

|  |
| --- |
| 3 |

|  |
| --- |
| 5 |

|  |
| --- |
| 4 |

|  |
| --- |
| 3 |

|  |
| --- |
| 3 |

|  |
| --- |
| 6 |

|  |
| --- |
| 6 |

|  |
| --- |
| 1 |

|  |
| --- |
| 4 |

|  |
| --- |
| 5 |

|  |
| --- |
| 2 |

|  |
| --- |
| 2 |

|  |
| --- |
| 题三3 用图 |

先求出事件的发生时间 (event 各点 ve(i) vl(i))

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | V(0) | V(1) | V(2) | v(3) | V(4) | V(5) | V(6) | V(7) | V(8) | V(9) |
| Ve(i) | 0 | 5 | 6 | 12 | 15 | 16 | 16 | 19 | 21 | 23 |
| Vl(i) | 0 | 9 | 6 | 12 | 15 | 16 | 19 | 19 | 21 | 23 |
| 差 | 0 | 4 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |

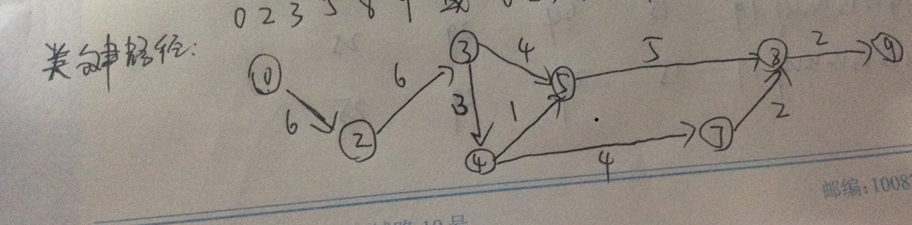
再求出活动的发生时间 （各条弧的e(i) ,l(i)）、

e(j)=ve(i)

l(j)=vl(i)-dut(<I,k>)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | E01 | E02 | E13 | E23 | E24 | E34 | E35 | E36 | E45 | E47 | E58 | E69 | E78 | E89 |
| 最早 | 0 | 0 | 5 | 6 | 6 | 12 | 12 | 12 | 15 | 15 | 16 | 16 | 19 | 21 |
| 最晚 | 4 | 0 | 9 | 6 | 12 | 12 | 12 | 15 | 15 | 15 | 16 | 19 | 19 | 21 |
| 差 | 4 | 0 | 4 | 0 | 6 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 |

这里面的事件差为0的即为关键事件，活动差为0的即为关键活动，这些活动的改变将影响工期

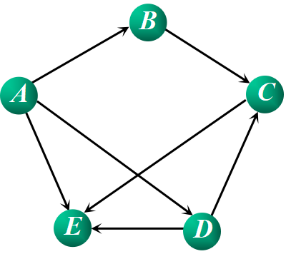


关键路径提升效率的程度是有限的，必须在不改变AOE-网的关键路径的情况下，提升活动的速度才有效。

# **dijkstra算法**

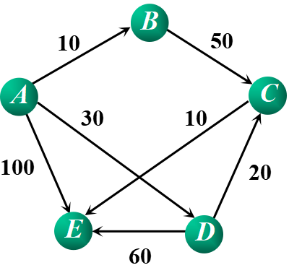
**一、最短路径**

　　①在非网图中，最短路径是指两顶点之间经历的边数最少的路径。



AE：1    ADE：2   ADCE：3   ABCE：3

　　②在网图中，最短路径是指两顶点之间经历的边上权值之和最短的路径。



AE：100   ADE：90   ADCE：60   ABCE：70

**③单源点最短路径问题**

　　问题描述：给定带权有向图G＝(V, E)和源点v∈V，求从v到G中其余各顶点的最短路径。

　　应用实例——计算机网络传输的问题：怎样找到一种最经济的方式，从一台计算机向网上所有其它计算机发送一条消息。

**④每一对顶点之间的最短路径**

　　问题描述：给定带权有向图G＝(V, E)，对任意顶点vi,vj∈V（i≠j），求顶点vi到顶点vj的最短路径。

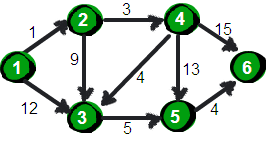
　　解决办法1：每次以一个顶点为源点，调用Dijkstra算法n次。显然，时间复杂度为O(n3)。 解决办法2：弗洛伊德提出的求每一对顶点之间的最短路径算法——Floyd算法，其时间复杂度也是O(n3)，但形式上要简单些。

**Dijkstra算法采用的是一种贪心的策略，声明一个数组dis来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合：T，初始时，原点 s 的路径权重被赋为 0 （dis[s] = 0）。若对于顶点 s 存在能直接到达的边（s,m），则把dis[m]设为w（s, m）,同时把所有其他（s不能直接到达的）顶点的路径长度设为无穷大。初始时，集合T只有顶点s**

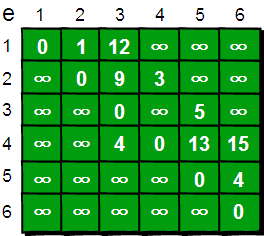
**然后，从dis数组选择最小值，则该值就是源点s到该值对应的顶点的最短路径，并且把该点加入到T中，此时完成一个顶点， 然后，我们需要看看新加入的顶点是否可以到达其他顶点并且看看通过该顶点到达其他点的路径长度是否比源点直接到达短，如果是，那么就替换这些顶点在dis中的值。 然后，又从dis中找出最小值，重复上述动作，直到T中包含了图的所有顶点。**

**Dijkstra算法**

**与Floyd-Warshall算法一样这里仍然使用二维数组e来存储顶点之间边的关系，初始值如下。**



**我们还需要用一个一维数组dis来存储1号顶点到其余各个顶点的初始路程，如下。**



**我们将此时dis数组中的值称为最短路的“估计值”。**

**Step 1:** 090657ofidcactthcig33i.png

**既然是求1号顶点到其余各个顶点的最短路程，那就先找一个离1号顶点最近的顶点。通过数组dis可知当前离1号顶点最近是2号顶点。当选择了2号顶点后，dis[2]的值就已经从“估计值”变为了“确定值”，即1号顶点到2号顶点的最短路程就是当前dis[2]值。为什么呢？你想啊，目前离1号顶点最近的是2号顶点，并且这个图所有的边都是正数，那么肯定不可能通过第三个顶点中转，使得1号顶点到2号顶点的路程进一步缩短了。因为1号顶点到其它顶点的路程肯定没有1号到2号顶点短**

**Step 2:**

**既然选了2号顶点，接下来再来看2号顶点有哪些出边呢。有2->3和2->4这两条边。先讨论通过2->3这条边能否让1号顶点到3号顶点的路程变短。也就是说现在来比较dis[3]和dis[2]+e[2][3]的大小。其中dis[3]表示1号顶点到3号顶点的路程。dis[2]+e[2][3]中dis[2]表示1号顶点到2号顶点的路程，e[2][3]表示2->3这条边。所以dis[2]+e[2][3]就表示从1号顶点先到2号顶点，再通过2->3这条边，到达3号顶点的路程。**

**Step 3:**

**我们发现dis[3]=12，dis[2]+e[2][3]=1+9=10，dis[3]>dis[2]+e[2][3]，因此dis[3]要更新为10。这个过程有个专业术语叫做“松弛”。即1号顶点到3号顶点的路程即dis[3]，通过2->3这条边松弛成功。这便是Dijkstra算法的主要思想：通过“边”来松弛1号顶点到其余各个顶点的路程。**

**同理通过2->4（e[2][4]），可以将dis[4]的值从∞松弛为4（dis[4]初始为∞，dis[2]+e[2][4]=1+3=4，dis[4]>dis[2]+e[2][4]，因此dis[4]要更新为4）。**

**Step 4:**

**刚才我们对2号顶点所有的出边进行了松弛。松弛完毕之后dis数组为：**

090706vmjy7l2ee2lyalia.png

**接下来，继续在剩下的3、4、5和6号顶点中，选出离1号顶点最近的顶点。通过上面更新过dis数组，当前离1号顶点最近是4号顶点。此时，dis[4]的值已经从“估计值”变为了“确定值”。下面继续对4号顶点的所有出边（4->3，4->5和4->6）用刚才的方法进行松弛。松弛完毕之后dis数组为：**

090714f2p1wppynngj2pep.png

**继续在剩下的3、5和6号顶点中，选出离1号顶点最近的顶点，这次选择3号顶点。此时，dis[3]的值已经从“估计值”变为了“确定值”。对3号顶点的所有出边（3->5）进行松弛。松弛完毕之后dis数组为：**

090722ywunackk35i8cni5.png

**继续在剩下的5和6号顶点中，选出离1号顶点最近的顶点，这次选择5号顶点。此时，dis[5]的值已经从“估计值”变为了“确定值”。对5号顶点的所有出边（5->4）进行松弛。松弛完毕之后dis数组为：**

090730eq6oqzyq7laqha9y.png

**最后对6号顶点所有点出边进行松弛。因为这个例子中6号顶点没有出边，因此不用处理。到此，dis数组中所有的值都已经从“估计值”变为了“确定值”。**

**最终dis数组如下，这便是1号顶点到其余各个顶点的最短路径。**

090738azt5clcozl899ekt.png

**现在来总结一下刚才的算法。算法的基本思想是：每次找到离源点（上面例子的源点就是1号顶点）最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。基本步骤如下：**

**将所有的顶点分为两部分：已知最短路程的顶点集合P和未知最短路径的顶点集合Q。最开始，已知最短路径的顶点集合P中只有源点一个顶点。我们这里用一个book[ i ]数组来记录哪些点在集合P中。例如对于某个顶点i，如果book[ i ]为1则表示这个顶点在集合P中，如果book[ i ]为0则表示这个顶点在集合Q中。**

**设置源点s到自己的最短路径为0即dis=0。若存在源点有能直接到达的顶点i，则把dis[ i ]设为e[s][ i ]。同时把所有其它（源点不能直接到达的）顶点的最短路径为设为∞。**

**在集合Q的所有顶点中选择一个离源点s最近的顶点u（即dis[u]最小）加入到集合P。并考察所有以点u为起点的边，对每一条边进行松弛操作。例如存在一条从u到v的边，那么可以通过将边u->v添加到尾部来拓展一条从s到v的路径，这条路径的长度是dis[u]+e[u][v]。如果这个值比目前已知的dis[v]的值要小，我们可以用新值来替代当前dis[v]中的值。**

**重复第3步，如果集合Q为空，算法结束。最终dis数组中的值就是源点到所有顶点的最短路径。**

**---------------------**

**作者：刺客五六柒**

**来源：CSDN**

**原文：https://blog.csdn.net/qq\_39521554/article/details/79333690**

**版权声明：本文为博主原创文章，转载请附上博文链接！**

**Prim与Dljkstra区别**

在图论中，Prim算法是计算最小生成树的算法，而Dijkstra算法是计算最短路径的算法。二者看起来比较类似，因为假设全部顶点的集合是V，已经被挑选出来的点的集合是U，那么二者都是从集合V-U中不断的挑选权值最低的点加入U，那么二者是否等价呢？也就是说是否Dijkstra也可以计算出最小生成树而Prim也可以计算出从第一个顶点v0到其他点的最短路径呢？答案是否定的，否则就不必有两个算法了。  
二者的不同之处在于“权值最低”的定义不同，Prim的“权值最低”是相对于U中的任意一点而言的，也就是把U中的点看成一个整体，每次寻找V-U中跟U的距离最小（也就是跟U中任意一点的距离最小）的一点加入U；而Dijkstra的“权值最低”是相对于v0而言的，也就是每次寻找V-U中跟v0的距离最小的一点加入U。  
一个可以说明二者不等价的例子是有四个顶点(v0, v1, v2, v3)和四条边且边值定义为(v0, v1)=20, (v0, v2)=10, (v1, v3)=2, (v3, v2)=15的图，用Prim算法得到的最小生成树中v0跟v1是不直接相连的，也就是在最小生成树中v0v1的距离是v0->v2->v3->v1的距离是27，而用Dijkstra算法得到的v0v1的距离是20，也就是二者直接连线的长度。

**练习题**

**一、单选题**

**（ C ）1. 在一个图中，所有顶点的度数之和等于图的边数的 倍。**

**A．1/2 B. 1 C. 2 D. 4**

**（ B ）2. 在一个有向图中，所有顶点的入度之和等于所有顶点的出度之和的 倍。**

**A．1/2 B. 1 C. 2 D. 4**

**（ B ）3. 有8个结点的无向图最多有 条边。n(n-1)/2**

**A．14 B. 28 C. 56 D. 112**

**（ C ）4. 有8个结点的无向连通图最少有 条边。N-1 最小生成子图**

**A．5 B. 6 C. 7 D. 8**

**（ C ）5. 有8个结点的有向完全图有 条边。n(n-1)**

**A．14 B. 28 C. 56 D. 112**

**（ B ）6. 用邻接表表示图进行广度优先遍历时，通常是采用 来实现算法的。**

**A．栈 B. 队列 C. 树 D. 图**

**（ A ）7. 用邻接表表示图进行深度优先遍历时，通常是采用 来实现算法的。**

**A．栈 递归 B. 队列 C. 树 D. 图**

**（ E ）8. 已知图的邻接矩阵，根据算法思想，则从顶点0出发按深度优先遍历的结点序列是**

A．0 2 4 3 1 5 6

B. 0 1 3 6 5 4 2

C. 0 4 2 3 1 6 5

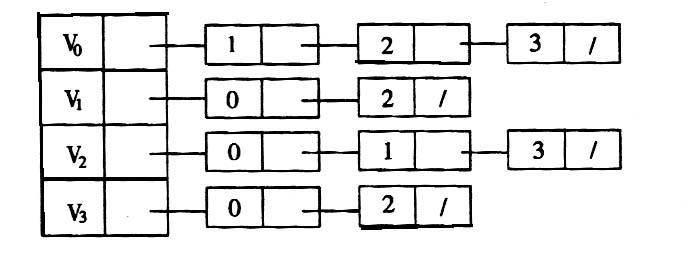
1. 0 3 6 1 5 4 2

E． 0 1 3 4 2 5 6

**（ C ）9. 已知图的邻接矩阵同上题8，根据算法，则从顶点0出发，按广度优先遍历的结点序列是**

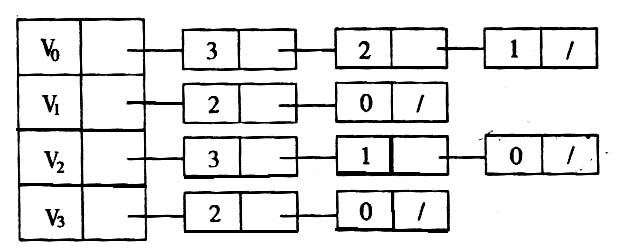
**A． 0 2 4 3 1 6 5 B. 0 1 3 5 6 4 2 C. 0 1 2 3 4 6 5 D. 0 1 2 3 4 5 6**

**（ D ）10. 已知图的邻接表如下所示，根据算法，则从顶点0出发按深度优先遍历的结点序列是**

****

**A．0 1 3 2 B. 0 2 3 1 C. 0 3 2 1 D. 0 1 2 3**

**（ A ）11. 已知图的邻接表如下所示，根据算法，则从顶点0出发按广度优先遍历的结点序列是**

****

**A．0 3 2 1 B. 0 1 2 3 C. 0 1 3 2 D. 0 3 1 2**

**（ A ）12. 深度优先遍历类似于二叉树的**

**A．先序遍历 B. 中序遍历 C. 后序遍历 D. 层次遍历**

**（ D ）13. 广度优先遍历类似于二叉树的**

**A．先序遍历 B. 中序遍历 C. 后序遍历 D. 层次遍历**

**二、填空题**

**1. 图有 邻接矩阵 、 邻接表 等存储结构，遍历图有 深度优先遍历 、 广度优先遍历 等方法。**

**2. 有向图G用邻接表矩阵存储，其第i行的所有元素之和等于顶点i的 出度 。**

**3. 如果n个顶点的图是一个环，则它有 n 棵生成树。 （以任意一顶点为起点，得到n-1条边）**

**4. 设有一稀疏图G，则G采用 邻接表 存储较省空间。**

**5. 设有一稠密图G，则G采用 邻接矩阵 存储较省空间。**

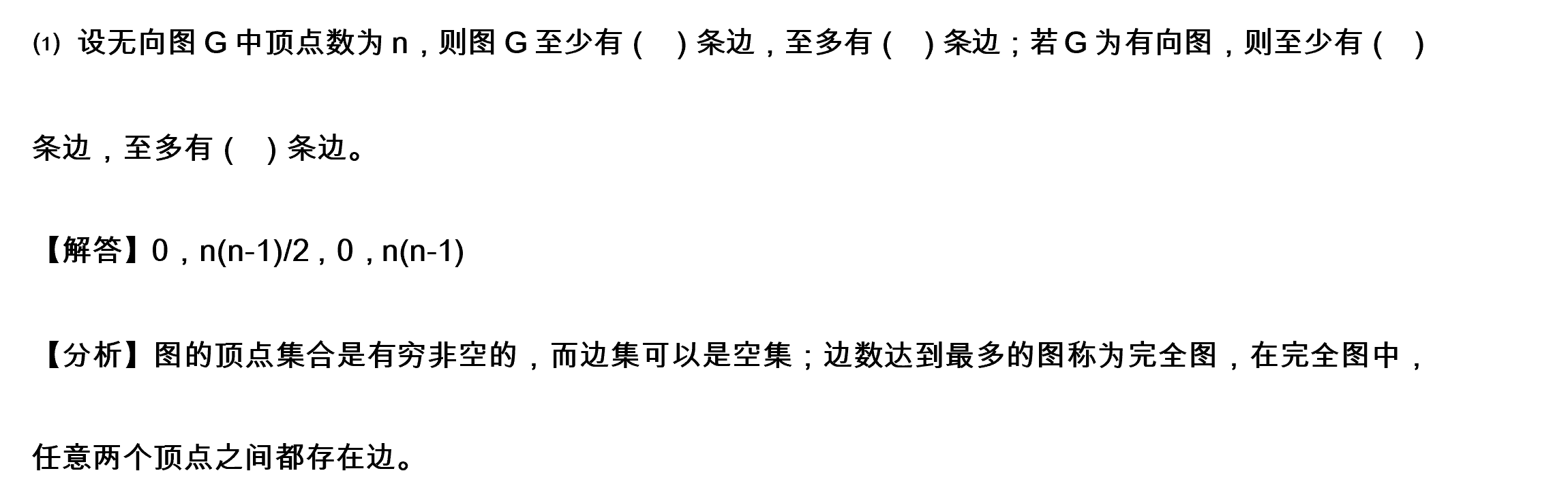
**6. 图的逆邻接表存储结构只适用于 有向 图。**

**7. 图的深度优先遍历序列 不是 惟一的。**

**8. n个顶点e条边的图采用邻接矩阵存储，深度优先遍历算法的时间复杂度为 O(n2) ；若采用邻接表存储时，该算法的时间复杂度为 O(n+e) 。**

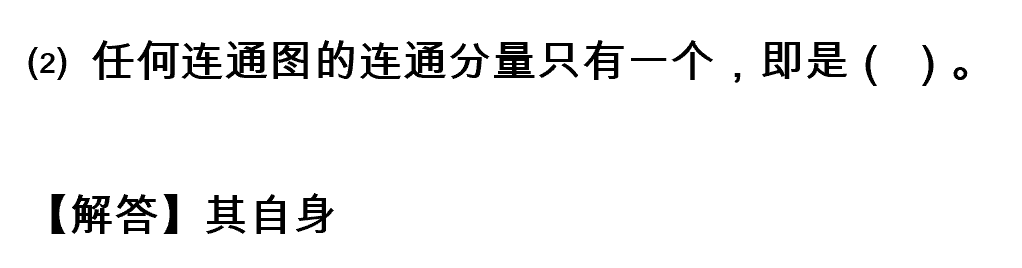
**9. n个顶点e条边的图采用邻接矩阵存储，广度优先遍历算法的时间复杂度为 O(n2) ；若采用邻接表存储，该算法的时间复杂度为 O(n+e) 。**

**10. 用普里姆(Prim)算法求具有n个顶点e条边的图的最小生成树的时间复杂度为 O(n2) ；用克鲁斯卡尔(Kruskal)算法的时间复杂度是 O(elog2e) 。**



**对于无向图和有向图来说，必须有点集，但是不一定有边集，所以边最少为0 最多的情况下就是任意两个点都有链接，也就是完全图 即有向图 n(n-1) 无向图 n(n-1)/2**

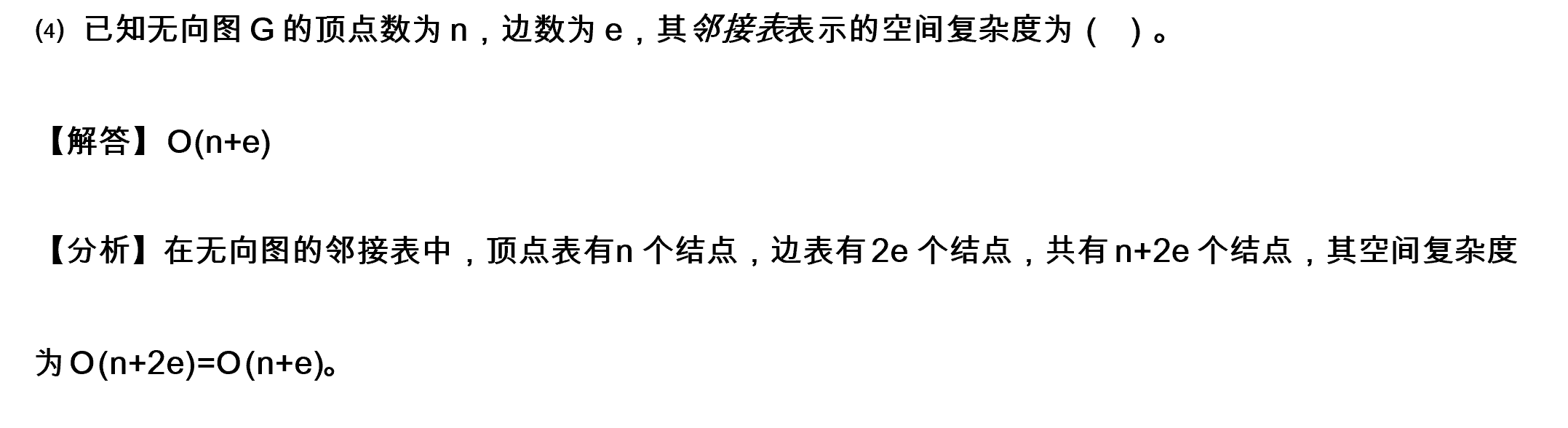
**但是对于连通图，就不一样了，最少要n-1条边才能保证联通**



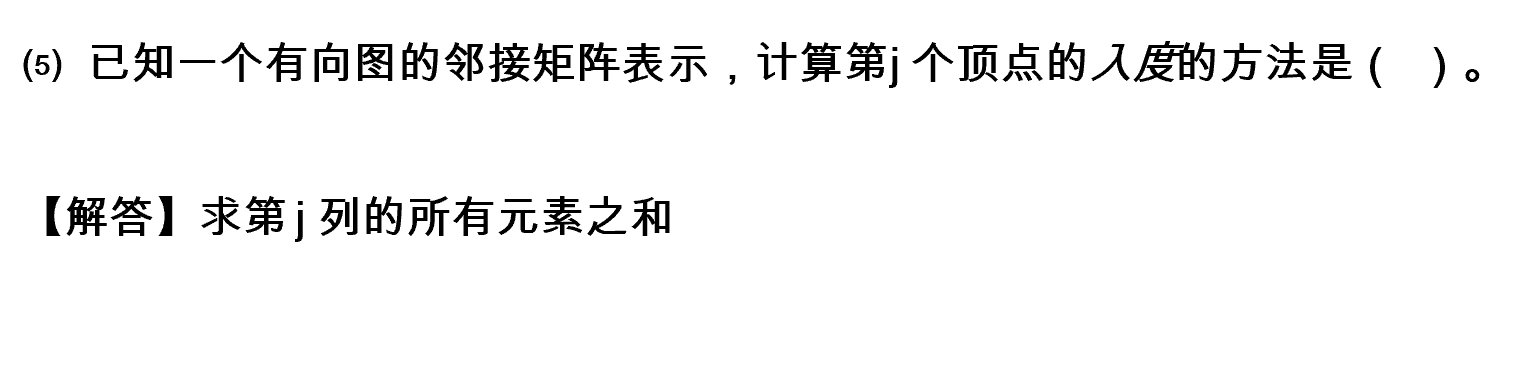
**连通图的要求**

1. **联通**
2. **子图**
3. **极大**

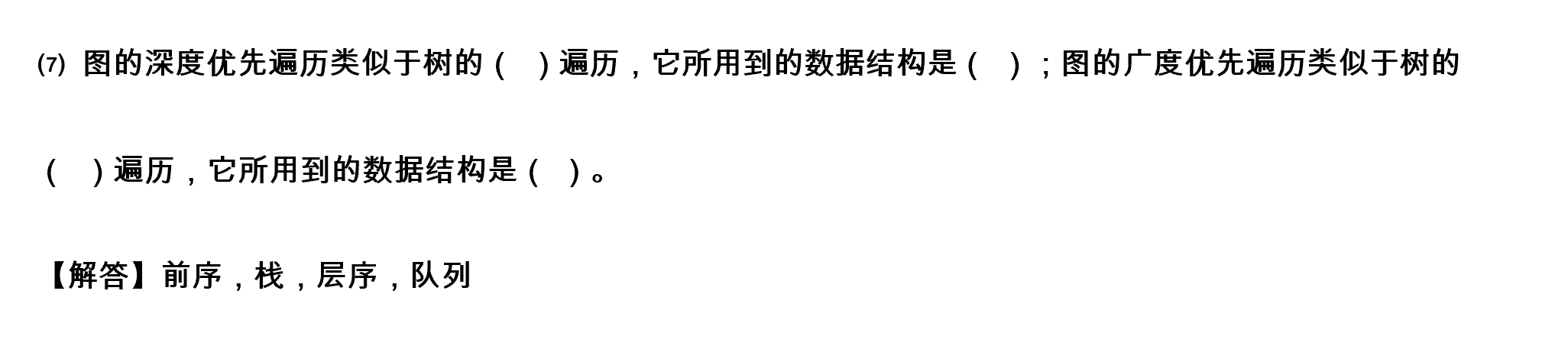
**由于求连通图的分量，那么连通图本来就是联通的，根据这个极大的原则，不管有向还是无向，那么都一定是整个图的自身就是联通分量，如果对于不连通的邮箱/无向图 可能有多个联通分量**



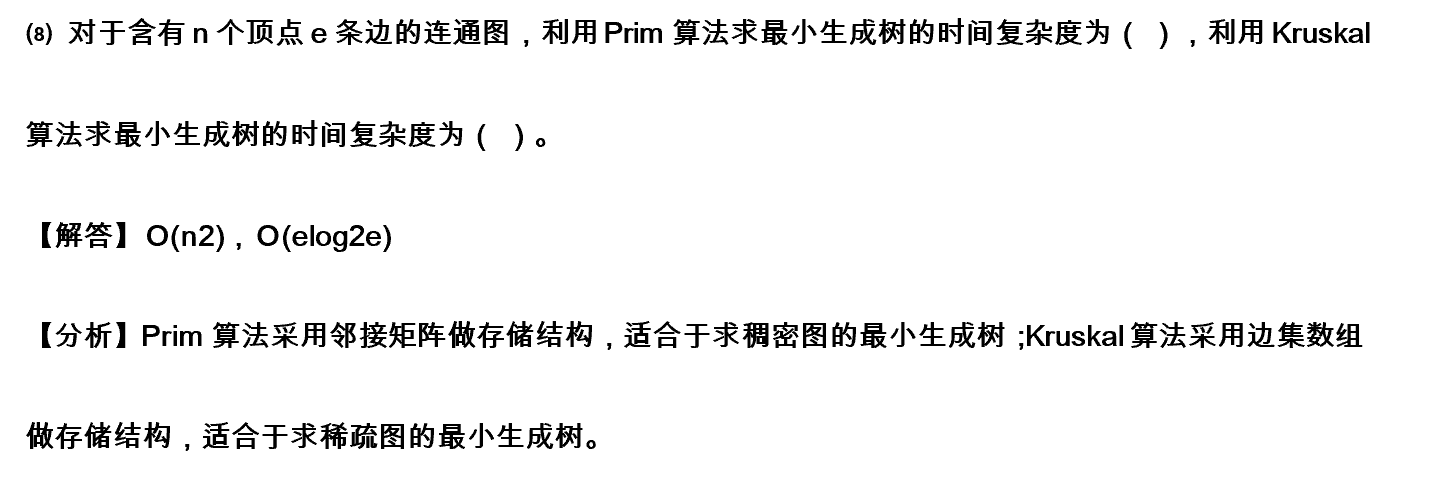
**这里面数字可以省去，有向图也是这个复杂度**



**对于邻接矩阵而言 行表示的是该元素的出度数量 列表示该元素的入度数量**

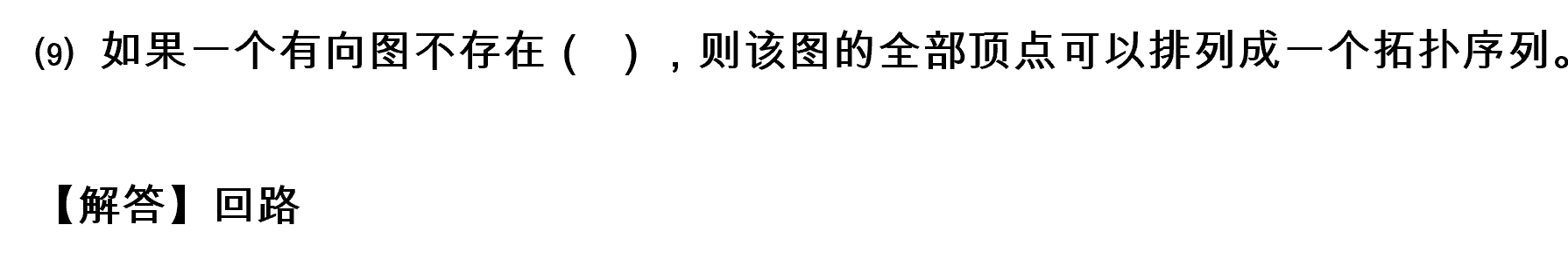


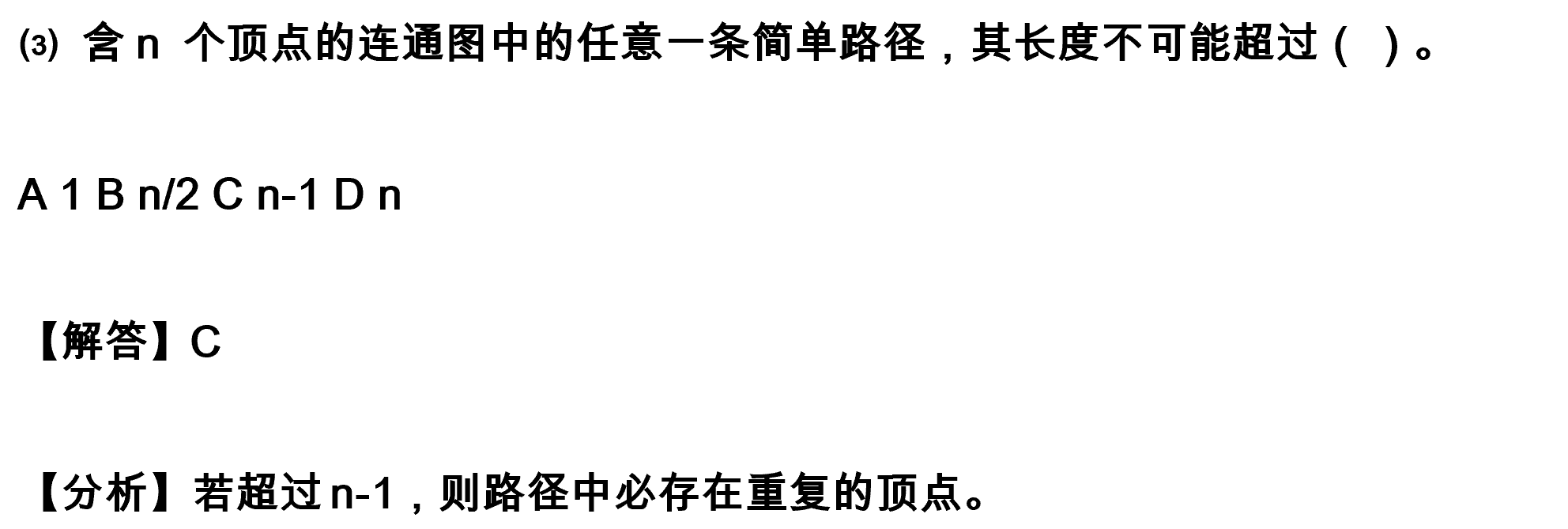
**DFS 要递归 用到栈 BFS要记录先后节点 用到Queue**



**这里面 elog2e e为边的数量**

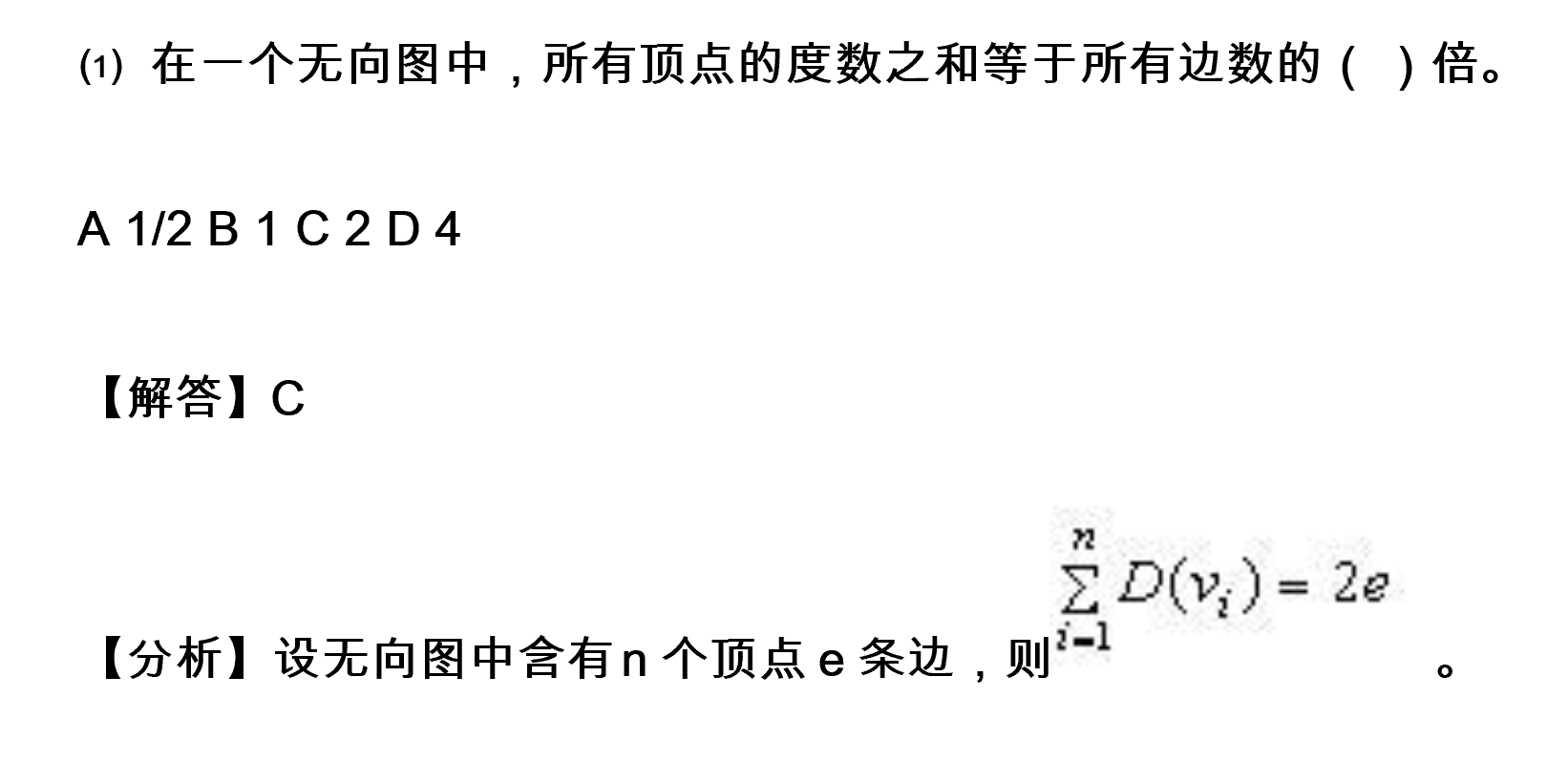
**图的生成树是它的一颗含有其所有顶点的无环连通子图,一幅加权图的最小生成树(MST)是它的一颗权值(树中的所有边的权值之和)最小的生成树.**

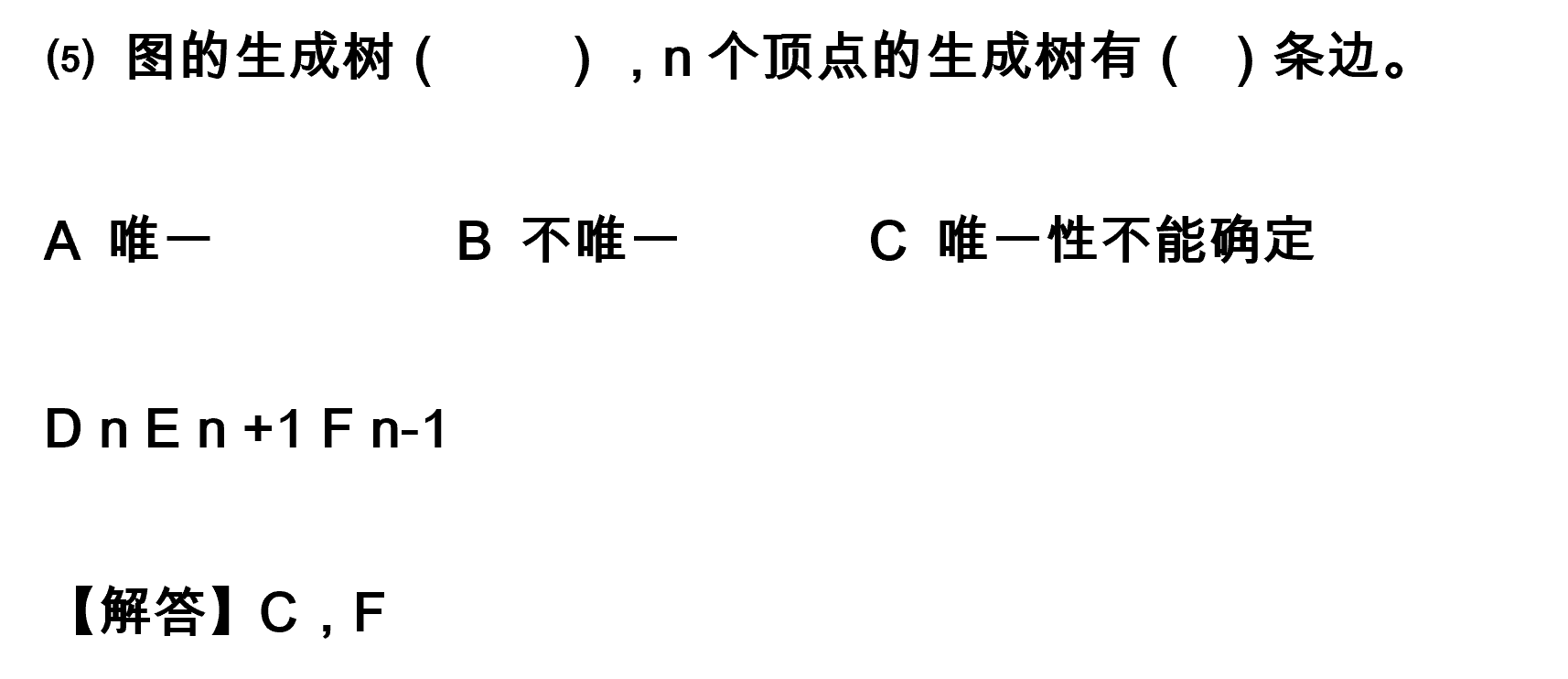




简单路径 没有环 n-1

简单回路 开头结尾一个顶点 有环（只有一个 ）n

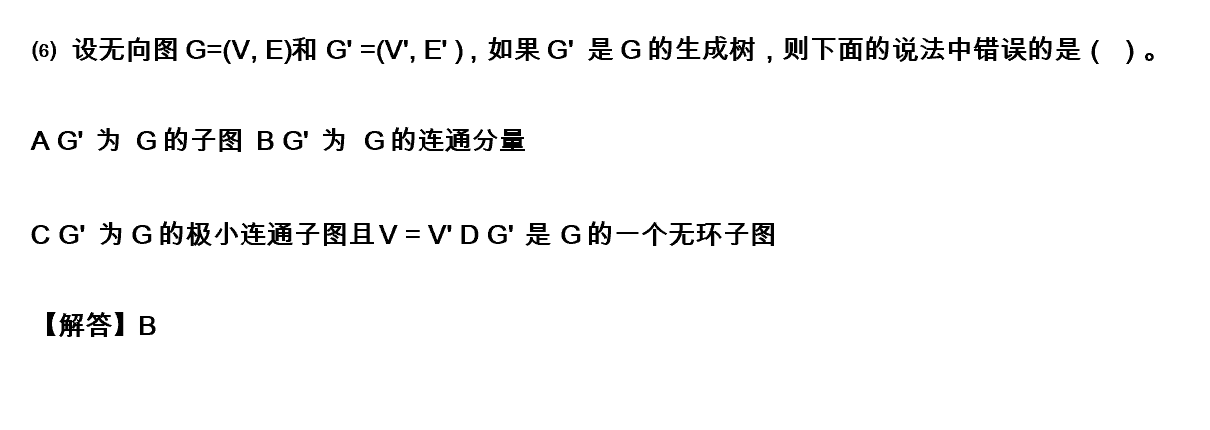




**图的生成树是它的一颗含有其所有顶点的无环连通子图**

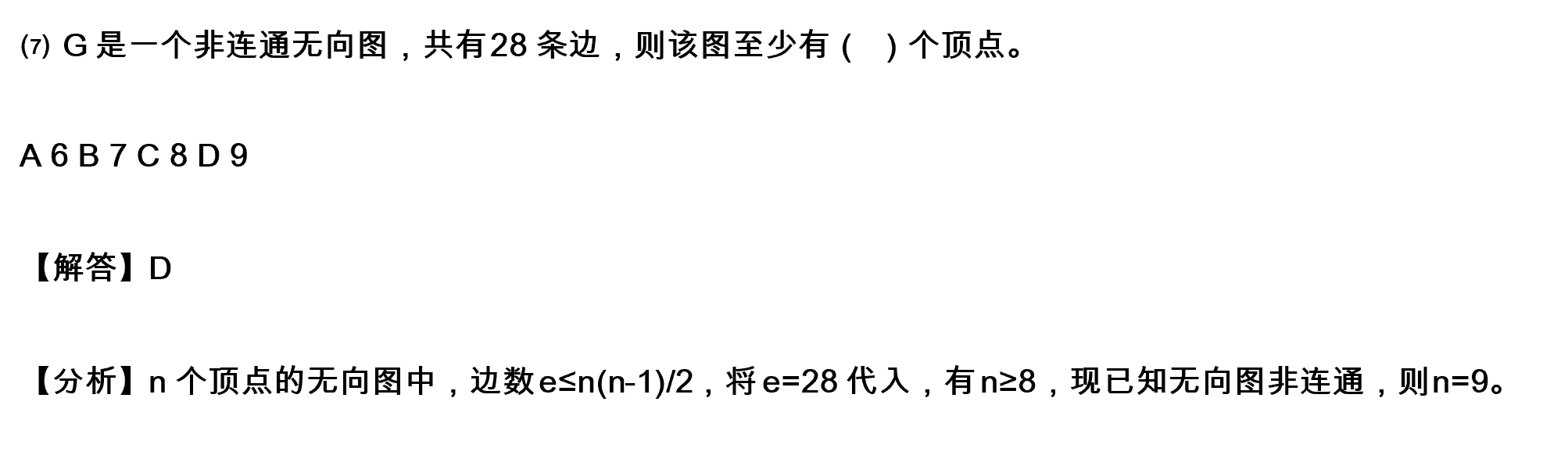
**所以任意的图，即使不是最小的 也是没有环的 即n-1条边**

[无向图](https://baike.baidu.com/item/%E6%97%A0%E5%90%91%E5%9B%BE/1680427)G的极大连通子图称为G的**连通分量**( Connected Component)。任何[连通图](https://baike.baidu.com/item/%E8%BF%9E%E9%80%9A%E5%9B%BE/6460995)的连通分量只有一个，即是其自身，非连通的[无向图](https://baike.baidu.com/item/%E6%97%A0%E5%90%91%E5%9B%BE/1680427)有多个连通分量。

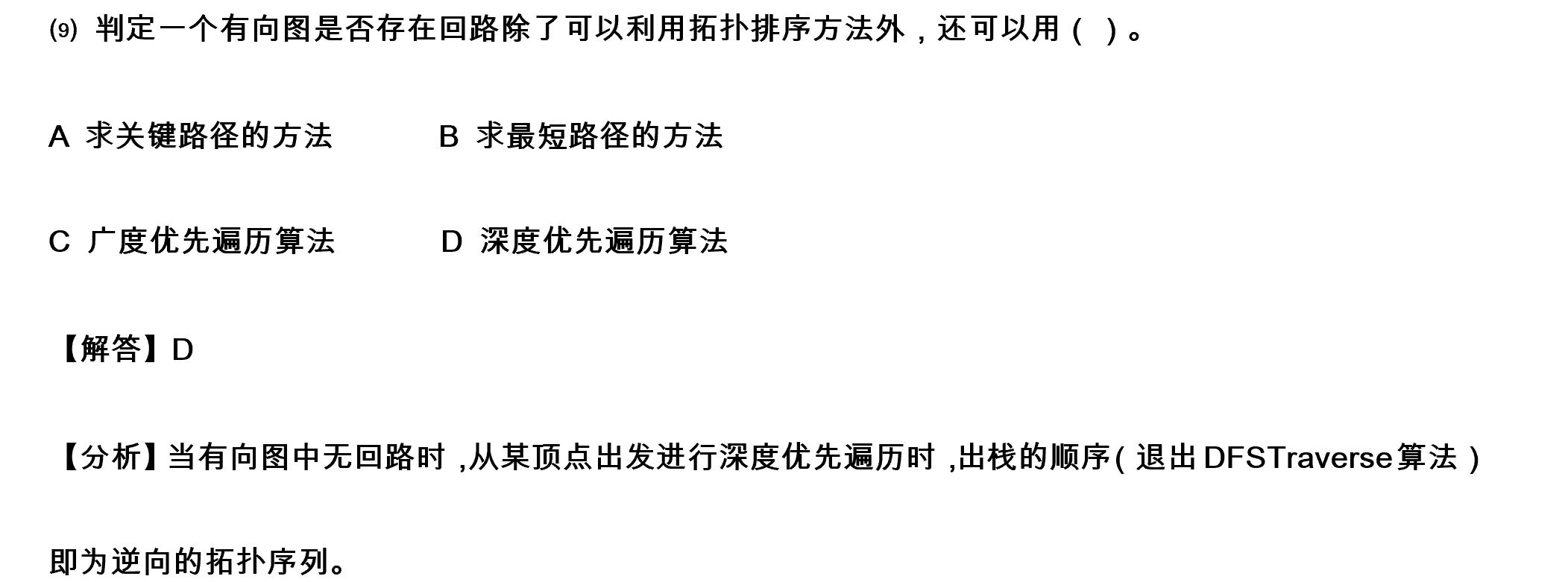


**联通分量是极大联通子树**

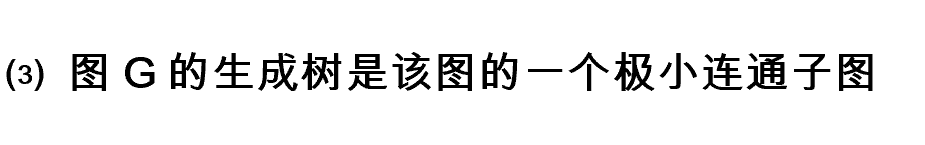
**生成树是极小联通子图**



**N个结点的连通图种 28边要9个点 ，那么非联通图相同结点下边更少 所以变数大于8 至少 9**

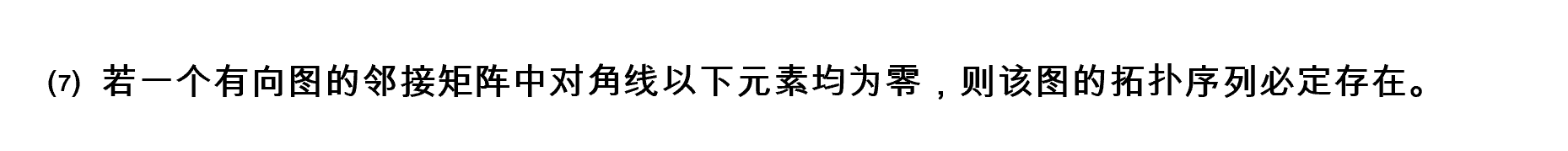


1. 拓扑排序: 还有顶点未输出,但已经不存在没有前驱的顶点了  
   2.深搜:从一个顶点出发存在搜回到自己的路径（所有节点都访问之后）

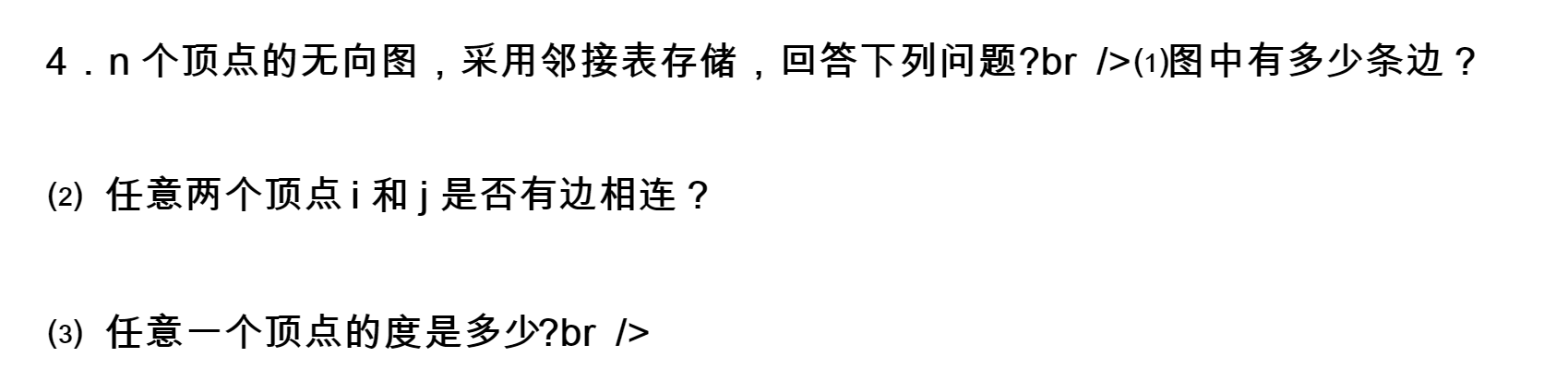


**生成子图-》联通-》生成树-》权值最小(最小生成树/最小生成子图)**

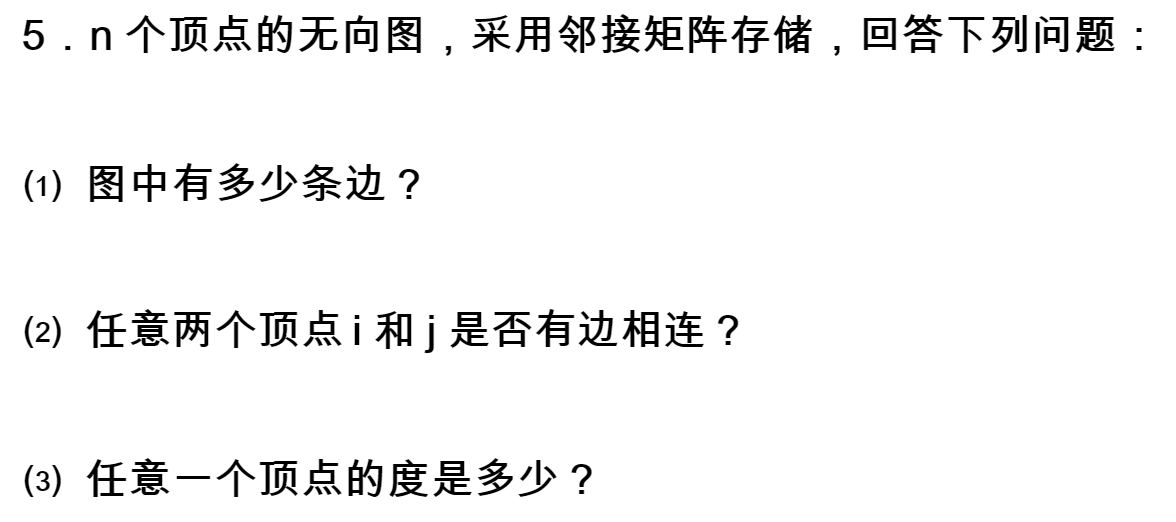




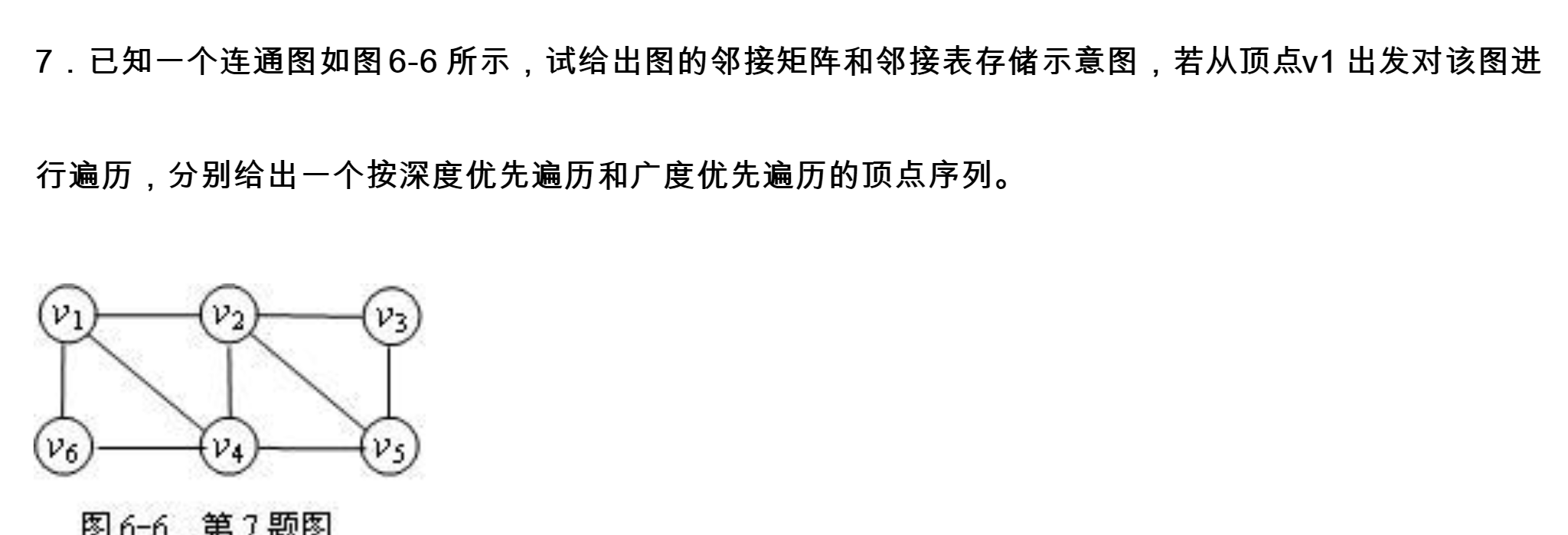
**T**



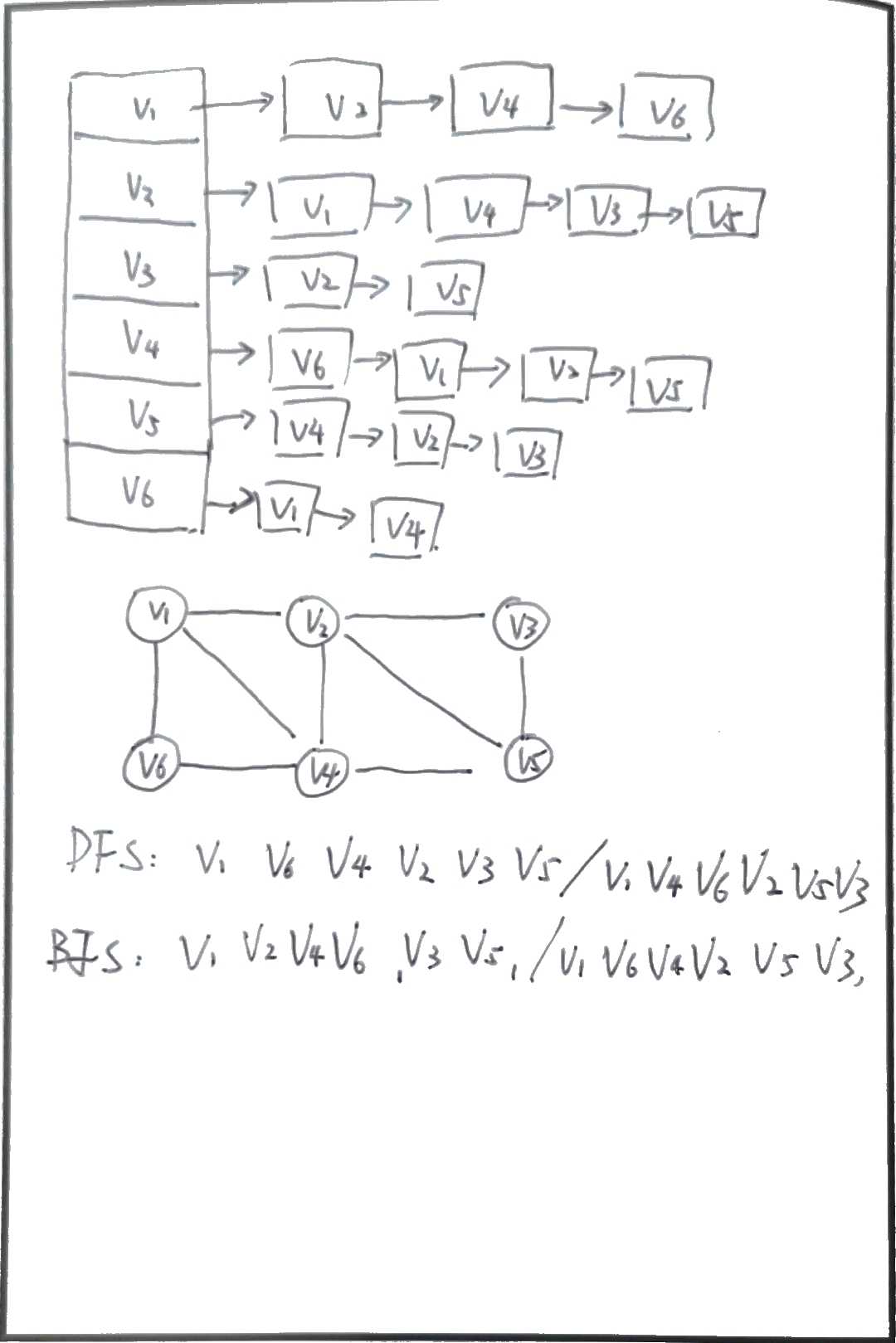
1. **边表元素个数/2**
2. **看i号的边表有没有j**
3. **看边表元素的个数**



1. **邻接矩阵为1的元素个数/2**
2. **看 I j号是不是为 1**
3. **看这一行有多个1**



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **V1** | **V2** | **V3** | **V4** | **V5** | **V6** |
| **V1** | **0** | **1** | **INF** | **1** | **1** | **1** |
| **V2** | **1** | **0** | **1** | **1** | **INF** | **INF** |
| **V3** | **INF** | **1** | **0** | **INF** | **1** | **INF** |
| **V4** | **1** | **1** | **INF** | **0** | **1** | **1** |
| **V5** | **INF** | **1** | **1** | **1** | **0** | **INF** |
| **V6** | **1** | **INF** | **INF** | **1** | **INF** | **0** |

****

