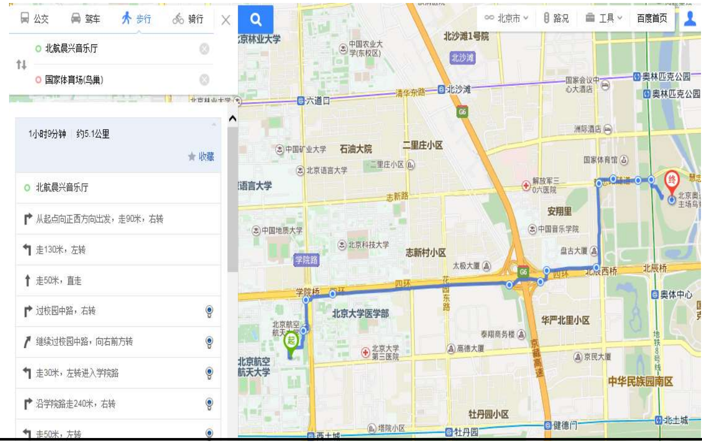
北京航空航天大学
BEIHANG UNIVERSITY

数据结构与程序设计（信息类）

Data Structure & Programming

北京航空航天大学 数据结构课程组
软件学院 林广艳
2023年春

地图导航

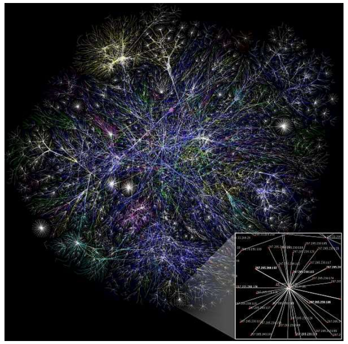


内容提要

1. 图的基本概念
2. 图的存储结构
3. 图的遍历
4. 最小生成树
5. 最短路径
6. AOV网与拓扑排序*
7. AOE网与关键路径*
8. 网络流量问题*

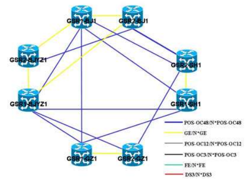
图：网络拓扑

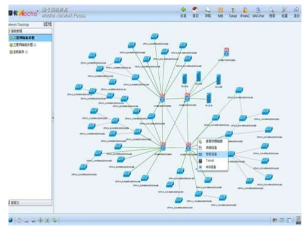
Internet Map



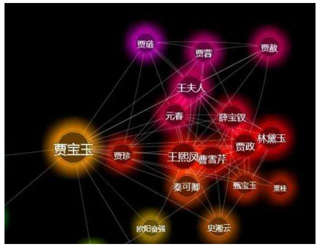
计算机网络

CNCNET骨干网超级核心节点之间互连结构图





图：社交网络（Social Network）



地铁线路图

内容提要

1. 图的基本概念
2. 图的存储结构
3. 图的遍历
4. 最小生成树
5. 最短路径
6. AOV网与拓扑排序*
7. AOE网与关键路径*
8. 网络流量问题*

问题1：北京地铁乘坐线路查询

- 编写一个程序实现北京地铁乘坐线路查询，输入为起始站名和终点站名，输出为从起始到终点的换乘线路。要求给出：
  - ✓ 乘坐站数最少的换乘方式（理论最快，通常用于计算票价）。如从西土城到北京西站：

    西土城-10-黄庄-4-国家图书馆-9-北京西站
  - ✓ 换乘次数最少的换乘方式（最方便）。如从西土城到北京西站：

    西土城-10-六里桥-9-北京西站

1. 图（Graph）的基本概念

图是由顶点的非空有穷集合与顶点之间关系(边或弧)的集合构成的结构, 通常表示为:

G = (V, E)

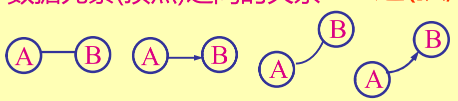
其中, V 为顶点集合, E 为关系(边或弧)的集合。

非空有穷集合

图中的顶点和关系(边)

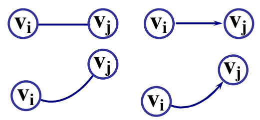
一个数据元素----顶点(Vertex) A

数据元素(顶点)之间的关系---- 边(弧,Edge)



边（弧）的表示

(1) 用图形



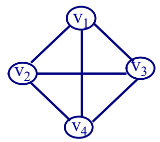
(2) 用符号 (v_i, v_j) 或 $<v_i, v_j>$

顶点偶对

(3) 用语言

- 顶点v_i与v_j是这条边的两个邻接点。
- 一条边依附于顶点v_i和顶点v_j。

示例

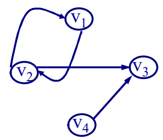


$G_1=(V_1, E_1)$

其中

$V_1 = \{ v_1, v_2, v_3, v_4 \}$

$E_1 = \{ (v_1,v_2), (v_1,v_3), (v_1,v_4), (v_2,v_3), (v_2,v_4), (v_3,v_4) \}$



$G_2=(V_2, E_2)$

其中

$V_2 = \{ v_1, v_2, v_3, v_4 \}$

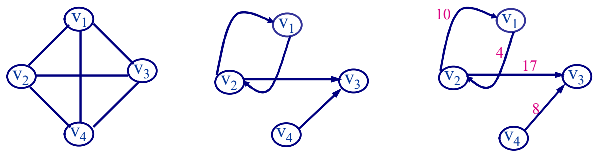
$E_2 = \{ <v_1,v_2>, <v_2,v_1>, <v_2,v_3>, <v_4,v_3> \}$

图的分类

◆ 无向图　对于 $(v_i,v_j)\in E$,必有 $(v_j,v_i)\in E$，并且偶对中顶点的前后顺序无关。

◆ 有向图　若 $<v_i,v_j>\in E$是顶点的有序偶对。

◆ 网(络)　与边有关的数据称为**权**,边上带权的图称为**网络**。



无向图　　有向图　　网(络)

顶点的度

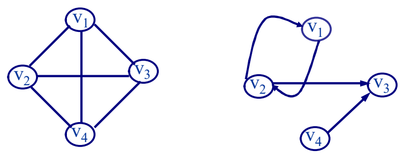
依附于顶点v_i的边的数目,记为$TD(v_i)$。

对于有向图而言, 有:

顶点的**出度**：以顶点v_i 为出发点的边的数目,记为$OD(v_i)$。

顶点的**入度**：以顶点v_i 为终止点的边的数目,记为$ID(v_i)$。

$$TD(v_i) = OD(v_i) + ID(v_i)$$



有关度与边的性质

结论1　对于具有n个顶点,e条边的图,有

$$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$$

结论2　具有n个顶点的无向图最多有n(n-1)/2条边

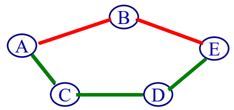
结论3　具有n个顶点的有向图最多有n(n-1)条边

- 边的数目达到最大的图称为**完全图**。
- 边的数目达到或接近最大的图称为**稠密图**，否则,称为**稀疏图**。

路径和路径长度

◆ 顶点v_x到v_y之间有路径$P(v_x,v_y)$的充分必要条件为:存在顶点序列 v_x, v_{i1}, v_{i2}, …, v_{im}, v_y，并且序列中相邻两个顶点构成的顶点偶对分别为图中的一条边。

$(v_x,v_{i1}), (v_{i1},v_{i2}), \ldots ,(v_{im},v_y)$ 或 $<v_x,v_{i1}>, <v_{i1},v_{i2}>, \ldots ,<v_{im},v_y>$ 都在E中

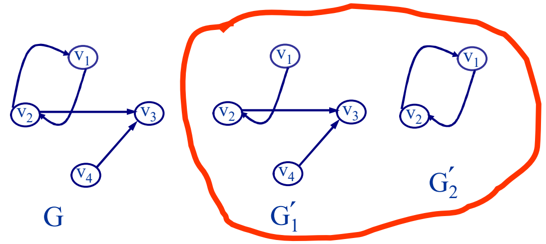


P(A, E):　A, B, E　(第1条路径)
　　　　　A, C, D, E　(第2条路径)

- 出发点与终止点相同的路径称为回路或环;
- 顶点序列中顶点不重复出现的路径称为简单路径。
- 不带权的图的路径长度是指路径上所经过的边的数目;
- 带权图的路径长度是指路径上经过的边上的权值之和。

子图

对于图G=(V,E) 与 G´=(V´,E´), 若有V´⊆V, E´⊆E,则称G´为G的一个子图。

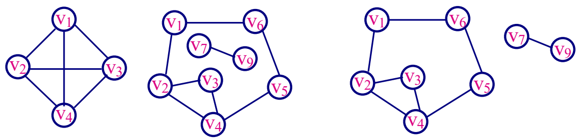


图的连通(Connected)

(1) 无向图(Digraph)的连通

无向图中顶点v_i 到v_j 有路径,则称顶点v_i 与v_j 是连通的。若无向图中任意两个顶点都连通, 则称该无向图是连通的。

连通分量 —— 无向图中的**极大**连通子图。



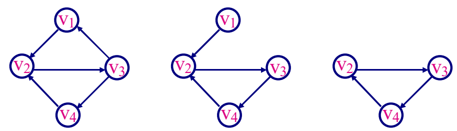
连通图　　非连通图　　2个连通分量

图的连通(Connected)

(2) 有向图(Directed Graph)的连通

若有向图中顶点v_i 到v_j 有路径,并且顶点v_j到v_i 也有路径，则称顶点v_i 与v_j 是连通的。若有向图中**任意**两个顶点都连通，则称该有向图是**强**连通的。

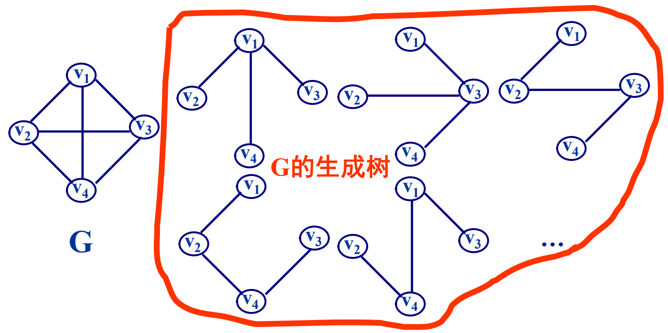
强连通分量 —— 有向图中的极大强连通子图。



(强) 连通图　　非连通图　　强连通分量

生成树（Spanning Tree)

包含具有n个顶点的连通图G的全部n个顶点，仅包含其n-1条边的极小连通子图称为G的一个生成树



生成树（续）

性质：

1. 包含n个顶点的图：连通且仅有n-1条边

   <=>无回路且仅有n-1条边

   <=>无回路且连通

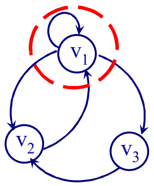
   <=>是一棵树
2. 如果n个顶点的图中只有少于n-1条边，图将不连通
3. 如果n个顶点的图中有多于n-1条边，图将有环（回路）
4. 一般情况下，生成树不唯一

内容提要

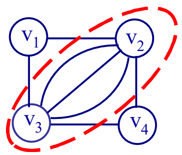
1. 图的基本概念
2. 图的存储结构
3. 图的遍历
4. 最小生成树
5. 最短路径
6. AOV网与拓扑排序*
7. AOE网与关键路径*
8. 网络流量问题*

本章不讨论的图

1. 带自身环的图



2. 多重图
（作业中有一个题）



只讨论简单图

2. 图的存储结构

对于一个图，需要存储的信息应该包括：

(1) 所有顶点的数据信息；

(2) 顶点之间关系(边或弧)的信息；

(3) 权的信息(对于网络)。

"第i个顶点"　"顶点i"

2.1 邻接矩阵存储方法

数组存储方法

- 顶点信息
- 边或弧的信息
- 权

◆ **核心思想　采用两个数组存储一个图。**

1. 定义一个一维数组VERTEX[0..n-1]存放图中所有顶点的数据信息

(若顶点信息为1,2,3,…,此数组可略)

2. 定义一个二维数组A[0..n-1,0..n-1]存放图中所有顶点之间关系的信息(该数组被称为**邻接矩阵**)，有

$$A[i][j]=\begin{cases}1 & \text{当顶点}v_i\text{到顶点}v_j\text{有边时}\\ 0 & \text{当顶点}v_i\text{到顶点}v_j\text{无边时}\end{cases}$$

对于带权的图，有 $A[i][j]=\begin{cases}w_{ij} & \text{当顶点}v_i\text{到}v_j\text{有边，且边的权为}w_{ij}\\ \infty & \text{当顶点}v_i\text{到顶点}v_j\text{无边时}\end{cases}$

25

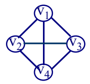
邻接矩阵特点

(1) 无向图的邻接矩阵一定是一个对称矩阵。

(2) 不带权的有向图的邻接矩阵一般是稀疏矩阵。

(3) 无向图的邻接矩阵的第i 行(或第i 列)非0 或非∞元素的个数为第i 个顶点的度数。

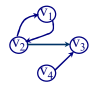
(4) 有向图的邻接矩阵的第i 行非0或非∞元素的个数为第i 个顶点的出度;第i 列非0或非∞元素的个数为第i 个顶点的入度。



$$\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{bmatrix}$$

V_1	V_2	V_3	V_4

(无向图)



$$\begin{bmatrix}0&1&0&0\\1&0&1&0\\0&0&0&0\\0&0&1&0\end{bmatrix}$$

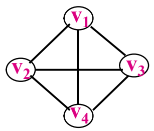
V_1	V_2	V_3	V_4

(有向图)

空间复杂度O(n^2)

27

示例

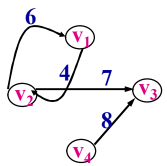


VERTEX1[0..3]

0	V_1
1	V_2
2	V_3
3	V_4

$$A_1=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{bmatrix}$$

邻接矩阵



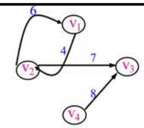
VERTEX2[0..3]

0	V_1
1	V_2
2	V_3
3	V_4

$$A_2=\begin{bmatrix}\infty&4&\infty&\infty\\6&\infty&7&\infty\\7&\infty&\infty&\infty\\\infty&\infty&8&\infty\end{bmatrix}$$

26

2.2 邻接表存储方法



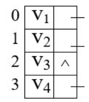
◆ **核心思想**　建立n个线性链表存储该图。

1. 每一个链表前面设置一个头结点,用来存放一个顶点的数据信息,称之为**顶点结点**。其构造为:

vertex	link

其中,vertex 域存放某个顶点的数据信息;
link 域存放某个链表中第一个边结点的地址。

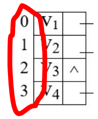
n个头结点怎么存储?



一维数组!

2. 第i个链表中的每一个链结点(称之为**边结点**)表示以第i个顶点为**出发点**的一条边;边结点的构造为:

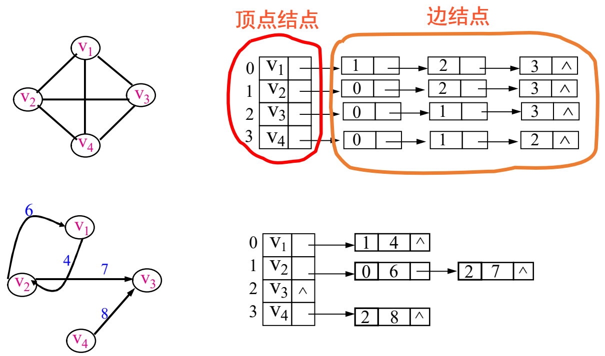
adjvex	weight	next



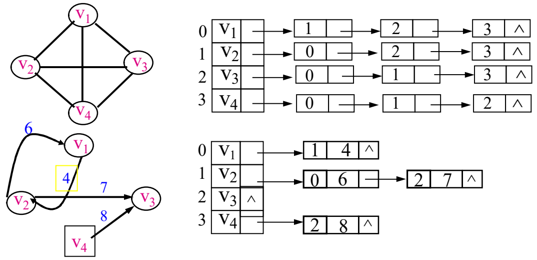
其中,next 域为指针域;
weight 域为权值域(若图不带权,则无此域);
adjvex 域存放以第i 个顶点为出发点的一条边的**另一端点在头结点数组中的位置**。

28

示例



邻接表特点

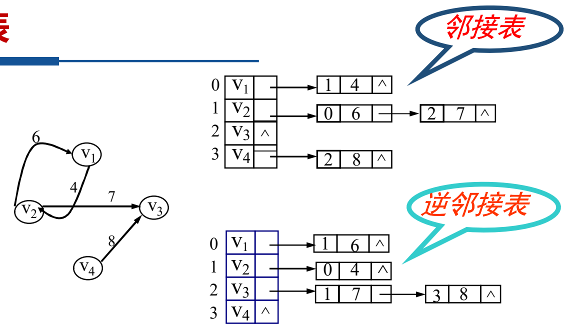


(1)无向图的第i个链表中边结点个数是第i个顶点**度数**。

(2)有向图的第i个链表中边结点个数是第i个顶点的**出度**。

(3)无向图的边结点个数一定为**偶数**；边结点个数为**奇数**的图一定是有向图。

逆邻接表



对于邻接表

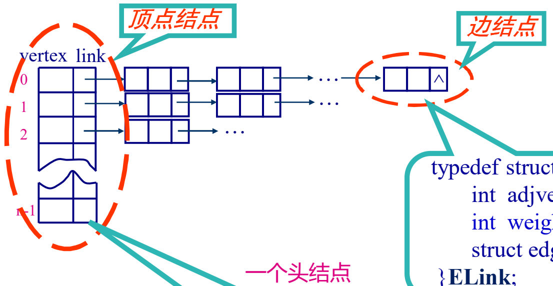
第i个链表中的每一个链结点(称之为边结点)表示以第i个顶点为**出发点**的一条边;

终止点

其他存储方法*

- 有向图的十字链表存储方法
- 无向图的多重邻接表存储方法

C语言描述：邻接表




```
typedef struct edge{
    int  adjvex;
    int  weight;
    struct edge  *next;
}ELink;
```

```
typedef struct ver{
    vertype  vertex;
    ELink  *link;
}VLink;
```

图的描述：
Vlink G[n];

C语言描述：邻接矩阵

```
#define MaxV   <最大顶点个数>
#define MaxE   <最大边数>
typedef struct  edge{
    int  weight;
    …
}Edge;
Vertype  Vertex[MaxV];
Edge  G[MaxV][MaxV];
```

边类型定义

顶点信息数组

边信息数组

C语言描述：邻接表（续）

```
#define MaxV   <最大顶点个数>
typedef struct  edge{
    int  adjvex;
    int  weight;
    struct edge  *next;
}ELink;
typedef struct ver{
    vertype  vertex;
    ELink  *link;
}VLink;
VLink  G[MaxV];
```

定义边结点类型

定义顶点结点类型

C语言描述：边集数组（稀疏图）

```
#define MaxV   <最大顶点个数>
#define MaxE   <最大边数>
typedef struct  edge{
    int v1, v2;
    int  weight;
}Edge;
Vertype  Vertex[MaxV];
Edge  G[MaxE];
```

定义边类型

顶点信息数组

边集数组

2.3 图的基本操作

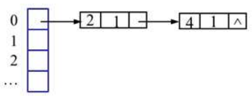
```
createGraph();     // 创建一个图
destoryGraph();    // 删除一个图
insertVex(v);      // 在图中插入一个顶点v
deleteVex(v);      // 在图中删除一个顶点v
insertEdge(v, w);  // 在图中插入一条边<v,w>
deleteEdge(v, w);  // 在图中删除一条边<v,w>
traverseGraph();   // 遍历一个图
```


内容提要

示例：创建一个图（邻接表）

若有如下输入：

```
8
0 2 4 … -1
1 3 6 8 … -1
…
```



第一行为图的顶点个数，从第二行开始第一个数为顶点序号，第二个数字开始为该顶点的邻接顶点，每行以-1结束，则创建一个邻接表存储的图算法如下：

```
void createGraph(VLink graph[])
{
  int i,n,v1,v2;
  scanf("%d",&n);
  for(i=0; i<n; i++){
    scanf("%d %d",&v1,&v2);
    while(v2 != -1){
      graph[v1].link=insertEdge(graph[v1].link, v2);
      graph[v2].link=insertEdge(graph[v2].link, v1);
      scanf("%d",&v2);
    }
  }
}
```

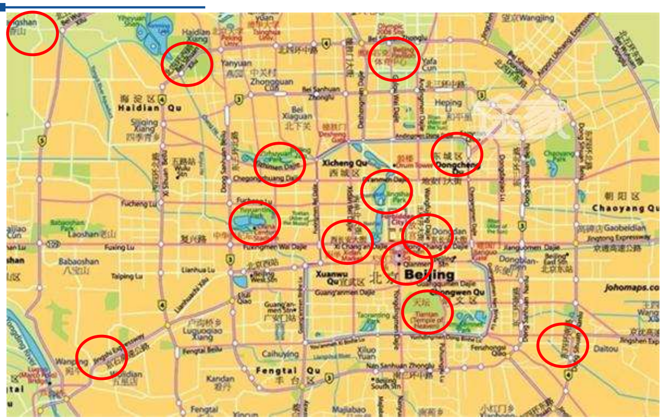
邻接矩阵：
graph[v1][v2]= graph[v2][v1]= 1;

```
#define MaxV    256
typedef struct  edge{
        int  adj;
        int  wei;
        struct edge *next;
}Elink;
typedef struct ver{
        ELink *link;
}Vlink;
VLink  G[MaxV];
```

```
//在链表尾插入一个节点
Elink  *insertEdge(ELink *head, int avex)
{
    ELink *e,*p;
    e =(ELink *)malloc(sizeof(ELink));
    e->adj= avex; e->wei=1; e->next=NULL;
    if(head == NULL){head=e; return head;}
    for(p=head;p->next != NULL; p=p->next)
        ;
    p->next = e;
    return head;
}
```

3. 图的遍历

某游客首次来京，想一次将旅游图上标明的景点玩到，他该如何设计线路即能将所有景点玩到，又不重复?



3. 图的遍历

◆ 从图中某个指定的顶点出发, 按照某一原则对图中所有顶点都访问一次, 得到一个由图中所有顶点组成的序列, 这一过程称为**图的遍历**。

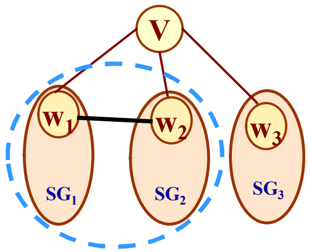
◆ 利用图的遍历

- 确定图中满足条件的顶点;
- 求解图的连通性问题，如求分量;
- 判断图中是否存在回路;
- ……

以无向图为例

41

W_1、W_2和W_3 均为 V 的邻接点，SG_1、SG_2 和 SG_3 分别为含顶点W_1、W_2和W_3 的子图。



访问顶点 V ：

for (W_1、W_2、W_3)

若该邻接点W_i未被访问,

则从它出发进行深度优先搜索遍历。

深度优先遍历(Depth First Search,DFS)

◆ **原则**

类似于二叉树的前序遍历

从图中某个指定的顶点v出发,先访问顶点v,然后从顶点v**未被访问过**的一个邻接点出发，继续进行深度优先遍历,直到图中与v相通的所有顶点都被访问;

若此时图中还有未被访问过的顶点, 则从另一个未被访问过的顶点出发重复上述过程,直到遍历全图。

完成一个连通分量的遍历

递归过程

42

如何判别V的邻接点是否被访问?

◆ 为了标记某一时刻图中哪些顶点是否被访问,定义一维数组visited[0..n-1], 有

$$visited[i]=\begin{cases}1 & \text{表示对应的顶点已经被访问}\\ 0 & \text{表示对应的顶点还未被访问}\end{cases}$$

44

算法实现

```
int Visited[N] = {0}; //标识顶点是否被访问过，N为顶点数
void travelDFS(VLink G[], int n){
    int i;
    for (i = 0; i < n; i++)
        Visited[i] = 0;
    for (i = 0; i < n; i++)
        if (!Visited[i])
            DFS(G, i);
}
```

```
#define MaxV    256
typedef struct  edge{
        int  adj;
        int  wei;
        struct edge *next;
}Elink;
typedef struct ver{
        ELink *link;
}Vlink;
VLink  G[MaxV];
```

```
void DFS(VLink G[], int v)
{
    ELink *p;
    Visited[v] = 1; //标识某顶点被访问过
    VISIT(G, v);    //访问某顶点
    for (p = G[v].link; p != NULL; p = p->next)
        if (!Visited[p->adj])
            DFS(G, p->adjvex);
}
```

```
//对比树的深度优先遍历算法
void DFStree(TNodeptr t){
    int i;
    if (t != NULL){
        VISIT(t); //访问t指向结点
        for (i = 0; i < MAXD; i++)
            if (t->next[i] != NULL)
                DFStree(t->next[i]);
    }
}
```

45

算法分析

如果图中具有n个顶点、e条边，则

✓ **若采用邻接表存储该图**

- 由于邻接表中有2e个或e个边结点，因而扫描边结点的时间为O(e)；
- 而所有顶点都递归访问一次，所以，算法的时间复杂度为O(n+e)。

✓ **若采用邻接矩阵存储该图**

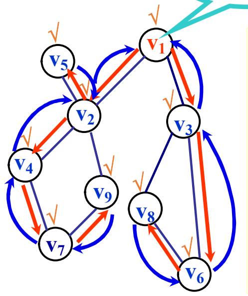
- 则查找每一个顶点所依附的所有边的时间复杂度为O(n);
- 而算法的时间复杂度为O(n²)。

47

示例：深度优先遍历

关于遍历序列的唯一性

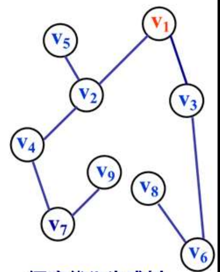
出发点



0	V_1	1	2		
1	V_2	0	3	4	8
2	V_3	0	5	7	
3	V_4	1	6		
4	V_5	1			
5	V_6	2	7		
6	V_7	3	8		
7	V_8	2	5		
8	V_9	1	6		

V_1 V_2 V_4 V_7 V_9 V_5 V_3 V_6 V_8

遍历序列



深度优先生成树

46

3.2 广度优先遍历(Breadth First Search,BFS)

◆ **原则**

类似于树的按层次遍历

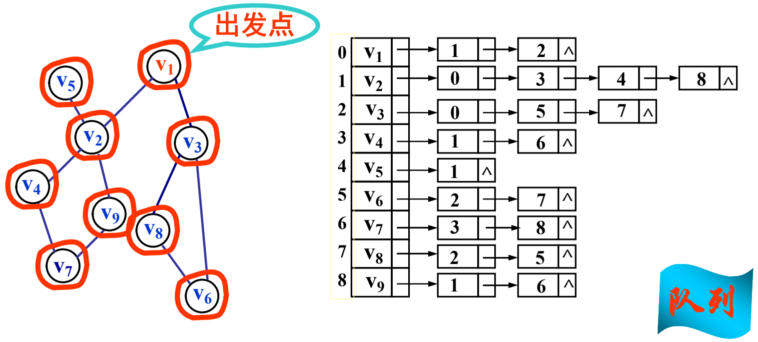
从图中某个指定的顶点v出发,先访问顶点v,然后依次访问顶点v的各个未被访问过的邻接点,然后又从这些邻接点出发, 按照同样的规则访问它们的那些未被访问过的邻接点，如此下去，直到图中与v 相通的所有顶点都被访问;

若此时图中还有未被访问过的顶点, 则从另一个未被访问过的顶点出发重复上述过程, 直到遍历全图。

完成一个连通分量的遍历

48

示例



遍历序列 $V_1\ V_2\ V_3\ V_4\ V_5\ V_9\ V_6\ V_8\ V_7$

DFS与BFS

- 对比这两个图的遍历算法
  - ✓它们在时间复杂度上是一样的
  - ✓不同之处仅仅在于对顶点的访问的顺序不同
- 具体用哪个取决于具体问题
  - ✓通常DFS更适合目标比较明确，以找目标为主要目的的情况
  - ✓BFS更适合在不断扩大遍历范围时找到相对最优解的情况

算法实现

```
#define MaxV    256
typedef struct  edge{
        int  adj;
        int  wei;
        struct edge *next;
}Elink;
typedef struct ver{
        ELink *link;
}Vlink;
VLink  G[MaxV];
```

```
int Visited[N] = {0}; //标识顶点是否被访问，N为顶点数
void travelBFS(VLink G[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        Visited[i] = 0;
    for (i = 0; i < n; i++)
        if (!Visited[i])
            BFS(G, i);
}
```

```
void BFS(VLink G[], int v){
    ELink *p;
    VISIT(G, v);    //访问当前顶点
    Visited[v] = 1; //标识某顶点被访问过
    enQueue(Q, v);
    while (!emptyQ(Q)) {
        v = deQueue(Q); //取出队头元素
        p = G[v].link;  //获取该顶点第一个邻接顶点
        //访问该顶点下的每个邻接顶点
        for (; p != NULL; p = p->next)
            if (!Visited[p->adjvex]) {
                VISIT(G, p->adjvex);    //访问当前顶点
                Visited[p->adjvex] = 1; //标识某顶点被访问过
                enQueue(Q, p->adjvex);
            }
    }
}
```

算法分析

采用邻接表存储： O(n+e)

采用邻接矩阵存储：O(n²)

问题：独立路径计算（非简单图）

- 老张和老王酷爱爬山，每周必爬一次香山。有次两人为从东门到香炉峰共有多少条路径发生争执，于是约定一段时间内谁走过对方没有走过的路线多谁胜。
- 给定一线路（无向连通图，两顶点之间可能有多条边），编程计算从起始点至终点共有多少条独立路径，并输出相关路径信息。

注：独立路径指的是从起点至终点的一条路径中至少有一条边是与别的路径中所不同的，同时路径中不存在环路。

问题：独立路径计算

【输入形式】：图的顶点按照自然数（0,1,2,...,n）进行编号，其中顶点0表示起点，顶点n-1表示终点。从标准输入中首先输入两个正整数n,e，分别表示线路图的顶点的数目和边的数目，然后在接下的e行中输入每条边的信息，具体形式如下：

<n> <e>
<e1> <vi1> <vj1>
<e2> <vi2> <vj2>
...
<en> <vin> <vjn>

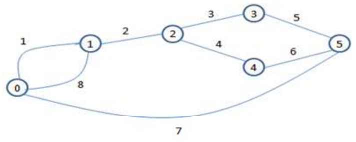
说明：第一行<n>为图的顶点数,<e>表示图的边数；第二行<e1> <vi1> <vj1>分别为边的序号（边序号的范围在[0,1000）之间，即包括0不包括1000）和这条边的两个顶点（两个顶点之间有多条边时，边的序号会不同），中间由一个空格分隔；其它类推。

【输出形式】

输出从起点0到终点n-1的所有路径（用边序号的序列表示路径且路径中不能有环），每行表示一条由起点到终点的路径（由边序号组成，中间有一个空格分隔，最后一个数字后跟回车），并且所有路径按照字典序输出。

【样例输入】

```
6 8
1 0 1
2 1 2
3 2 3
4 2 4
5 3 5
6 4 5
7 0 5
8 0 1
```



【样例输出】

```
1 2 3 5
1 2 4 6
7
8 2 3 5
8 2 4 6
```

【样例说明】

输出的第一个路径1 2 3 5，表示一条路径，先走1号边(顶点0到顶点1)，然后走2号边(顶点1到顶点2)，然后走3号边(顶点2到顶点3)，然后走5号边(顶点3到顶点5)到达终点。

问题：独立路径计算 – 数据结构设计

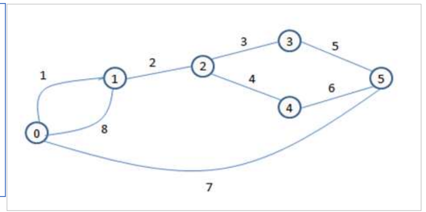
采用邻接表来存储图，邻接表设计如下：

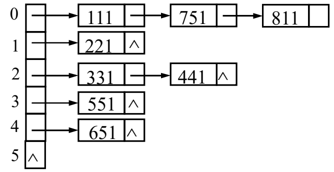
```
#define MAXSIZE 512
struct edge{//边结点结构
    int eno;     //边序号
    int adjvex; //邻接顶点
    //边的权重，可为距离或时间，本例为1
    int weight;
    struct edge *next;
};
```

```
struct ver
{ //顶点结构，邻接表下标即为顶点序号
    struct edge *link;
};
struct ver G[MAXSIZE]; //由邻接表构成的图
char Visted[MAXSIZE]={0};//标识相应顶点是否被访问
int paths[MAXSIZE];  //独立路径
```

【样例输入】

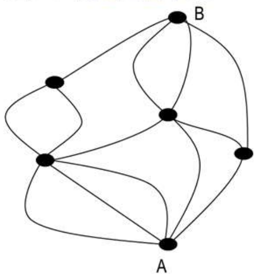
```
6 8
1 0 1
2 1 2
3 2 3
4 2 4
5 3 5
6 4 5
7 0 5
8 0 1
```





问题：独立路径计算----问题分析与算法设计

- **问题的实质**：给定起点（如图中A），对图进行遍历，并在遍历图的过程中找到到达终点（如图中B）的所有情况。
- 前面介绍的DFS和BFS算法都是从起始点出发对邻接顶点的遍历。而**问题是本文中两个点间可能有多个边**（如图所示）。
- **算法策略**：对DFS算法（或BFS）进行改进，在原来按邻接顶点进行遍历，**改为按邻接顶点的边进行遍历**（即从一个顶点出发遍历其邻接顶点时，按邻接顶点的边进行深度遍历，即只有当某顶点的所有邻接顶点的**所有边**都遍历完才结束该结点的遍历）。



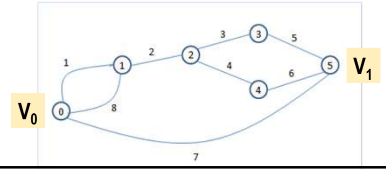
问题：独立路径计算 – 主要代码*

```
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE  512
struct edge {          //边结点结构
   int  eno;           //边序号
   int  adjvex;        //邻接顶点
   int  weight;        //边的权重（可为距离或时间），本文中为1
   struct edge  *next;
 };
struct ver { //顶点结构，邻接表下标即为顶点序号
   struct edge  *link;
} ;
struct ver Graph[MAXSIZE]; //由邻接表构成的图
char Visited[MAXSIZE] = {0}; //标识相应顶点是否被访问
int Paths[MAXSIZE]={0};          //独立路径
int Vnum;  //vertex number
int V0, V1; //start vertex, and end vertex
struct edge *insertEdge(struct edge *head, int avex, int eno);
void eDFS(int v,int level);
void printPath(int n);
```

问题：独立路径计算 – 主要代码*

```
//基于DFS的独立路径查找算法
void eDFS(int v, int level){
    struct edge  *p;
    if(v == V1)  {printPath(level); return;}
    for(p=Graph[v].link; p!= NULL; p=p->next)
    if( !Visited[p->adjvex]){
        Paths[level] = p->eno;
        Visited[p->adjvex] = 1;
        eDFS(p->adjvex,level+1);
        Visited[p->adjvex] = 0;
    }
}
```

```
//原来的DFS算法
void DFS(VLink  G[ ], int v) {
    ELink  *p;
    Visited[v] = 1; //标识某顶点被访问过
    VISIT(G, v); //访问某顶点
    for(p = G[v].link; p !=NULL; p=p->next )
        if( !Visited[p->adjvex] )
            DFS(G, p->adjvex);
}
```



问题：独立路径计算 – 主要代码*

```
#define MAXSIZE  512
struct edge{         //边结点结构
    int  eno;        //边序号
    int  adjvex;     //邻接顶点
    int  weight;     //边的权重(可为距离
                     //或时间)，本文中为1
    struct edge  *next;
};
```

```
struct ver { //顶点结构，邻接表下标即为顶点序号
    struct edge  *link;
} ;
struct  ver  G[MAXSIZE];     //由邻接表构成的图
int Visted[MAXSIZE] = {0}; //标识相应顶点是否被访问
int paths[MAXSIZE];          //独立路径
```

```
struct edge *insertEdge(struct edge *head, int avex, int eno)
{
    struct edge *e,*p;
    e =(struct edge *)malloc(sizeof(struct edge));
    e->eno = eno; e->adjvex = avex; e->weight = 1; e->next = NULL;
    if(head == NULL)
            {
                head=e;
                return head;
            }
    for(p=head; p->next != NULL; p=p->next)
        ;
    p->next = e;
    return head;
}
```

```
void printPath(int n)
{
        int i;
        for(i=0; i<n; i++)
            printf("%d ", Paths[i]);
        printf("\n");
        return;
}
```

问题：独立路径计算 – 主要代码*

```
#define MAXSIZE  512
struct edge{         //边结点结构
    int  eno;        //边序号
    int  adjvex;     //邻接顶点
    int  weight;     //边的权重(可为距离
                     //或时间)，本文中为1
    struct edge  *next;
};
```

```
int V0, V1;
int main(){
    int en,eno,i,v1,v2;
    scanf("%d %d", &Vnum, &en);
    for(i=0; i<en; i++){
        scanf("%d %d %d", &eno, &v1, &v2);
        Graph[v1].link = insertEdge(Graph[v1].link,v2, eno);
        Graph[v2].link = insertEdge(Graph[v2].link,v1, eno);
    }
    V0 = 0; V1 = Vnum-1; Visited[V0] = 1;
    eDFS(V0,0);
    return 0;
}
```

【样例输入】

```
6 8
1 0 1
2 1 2
3 2 3
4 2 4
5 3 5
6 4 5
7 0 5
8 0 1
```

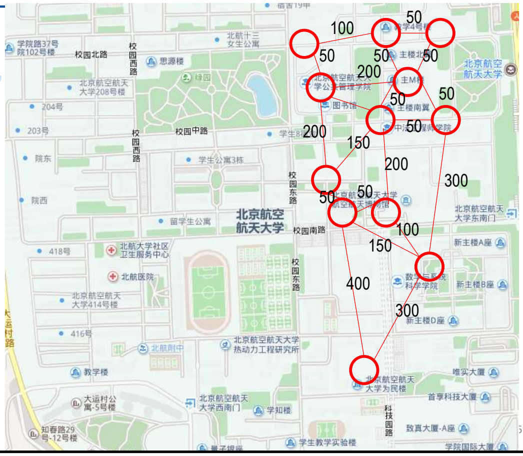
内容提要

1. 图的基本概念
2. 图的存储结构
3. 图的遍历
4. 最小生成树
5. 最短路径
6. AOV网与拓扑排序*
7. AOE网与关键路径*
8. 网络流量问题*

4. 最小生成树(Minimum Cost Spanning Tree)

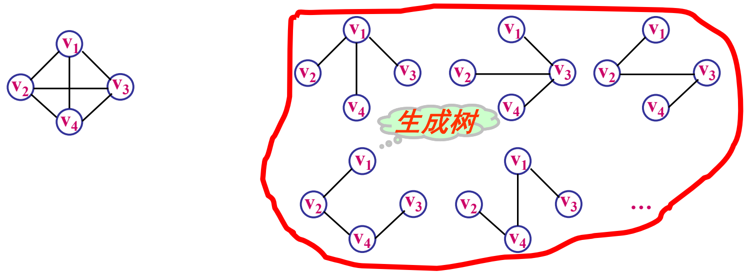
北航网络中心要给北航主要办公楼间铺设光缆以构建网络。

如何以最小的成本完成网络铺设?



4.1 生成树

◆ 一个连通图的**生成树**是包含着**连通图**的全部**n**个顶点，且仅包含其**n-1**条边的**极小连通子图**。



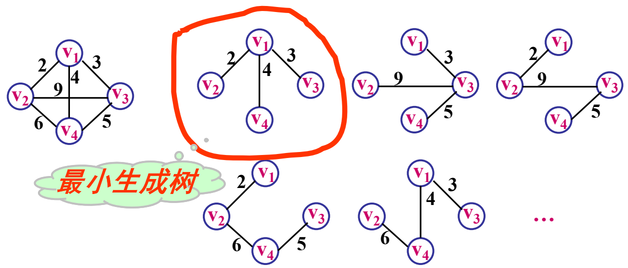
生成树的性质

◆ 性质

1. 包含n个顶点的图：连通且有n-1条边
   - 无回路且有n-1条边
   - 无回路且连通
   - 是一棵树
2. 如果n个顶点的图中只有少于n-1条边，图将不连通
3. 如果n个顶点的图中有多于n-1条边，图将有环（回路）
4. 一般情况下，生成树不唯一

最小生成树

◆ 在带权连通图中，总的权值之**和最小**的带权生成树为**最小生成树**，也称为最小代价生成树,或最小花费生成树。



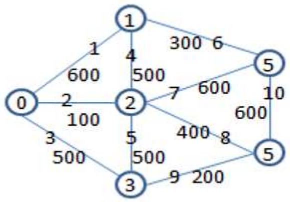
4.2 构造最小生成树

◆ **构造原则**

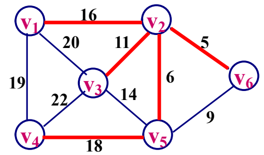
- ✓ 只能利用图中的边来构造最小生成树;
- ✓ 只能使用、且仅能使用图中的n-1条边来连接图中的n个顶点;
- ✓ 不能使用图中产生回路的边。

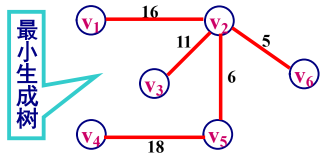
◆ **构造算法**

- ✓ 普里姆(Prim)算法
- ✓ 克鲁斯卡尔(Kruskal)算法



例





最小生成树的权值 = 56

G:

$V = \{ v_1, v_2, v_3, v_4, v_5, v_6 \}$

$$GE=\left\{\begin{matrix} (v_1, v_2)_{16}, & (v_1, v_3)_{20} \\ (v_1, v_4)_{19}, & (v_2, v_3)_{11} \\ (v_2, v_5)_{6}, & (v_2, v_6)_{5} \\ (v_3, v_4)_{22}, & (v_3, v_5)_{14} \\ (v_4, v_5)_{18}, & (v_5, v_6)_{9} \end{matrix}\right\}$$

最小生成树唯一吗?

T:

$U = \{ v_1, v_2, v_6, v_5, v_3, v_4 \}$

$TE = \{(v_1, v_2)_{16}\ (v_2, v_6)_5\ (v_2, v_5)_6\ (v_2, v_3)_{11}\ (v_4, v_5)_{18} \}$

4.3 普里姆算法

◆ **贪心法----逐步求解法**

是一种不追求最优解,只希望最快得到较为满意的解的方法。

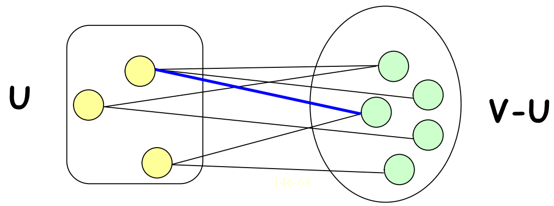
当追求的目标是一个问题的最优解时,

1. 分解问题成若干步骤完成
2. 每一步骤选择局部最优方案
3. 通过各步骤的局部最优选择达到整体的最优

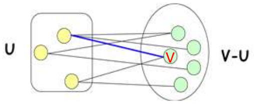
4.3 普里姆算法

◆ **一般情况下所添加的顶点应满足下列条件:**

在生成树的构造过程中，图中**n**个顶点分属两个集合：已落在生成树上的顶点集**U**和尚未落在生成树上的顶点集**V-U**，则应在所有连通**U中**顶点和**V-U**中顶点的边中选取权值最小的边。



4.3 普里姆算法



◆ **算法思想**

(1) 初始化U={v_0}。v_0到其他顶点的所有边为候选边;

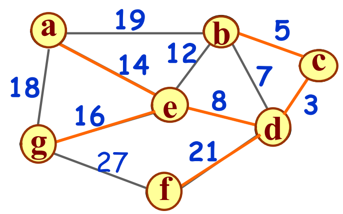
(2) 重复以下步骤n-1次,使得其他n-1个顶点被加入到U中:

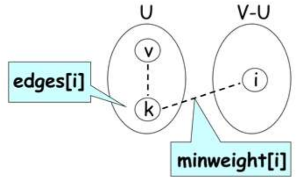
① 从候选边中挑选权值最小的边输出,设该边在V-U中的顶点是v,将v加入U中,删除U中其它顶点与v关联的边;

② 考察当前V-U中的所有顶点v_i,修改候选边:若(v,v_i)的权值小于原来和v_i关联的候选边,则用(v,v_i)取代后者作为候选边。

69

例如:



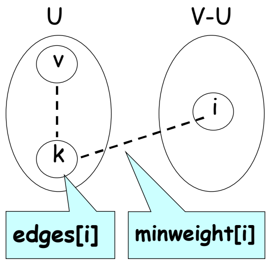


	0	1	2	3	4	5	6
	a	b	c	d	e	f	g
edges		c	d	e	a	d	e
minweight	0	0	0	0	0	0	0

4.3 普里姆算法

◆ **Prim算法数据结构说明**

设置辅助数组,对当前V - U集中的每个顶点,记录和顶点集U中顶点相连接的代价最小的边:



int weights[MAXVER][MAXVER];

当图G中存在边(i,j),则weights[i][j]为其权值,否则为一个INFINITY

int minweight[MAXVER];

存放未确定为生成树的顶点至已确定的生成树上顶点的边权重,

minweight[i] = 0 表示其已确定为最小生成树顶点

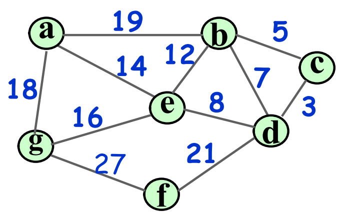
int edges[MAXVER];

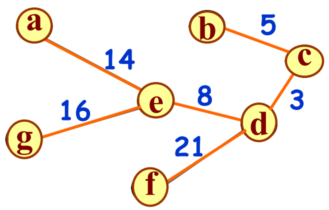
存入生成的最小生成树的边,如:

(i, edges[i]) 为最小生成树的一条边,应有n-1条边

70

用普里姆算法求最小生成树





算法实现

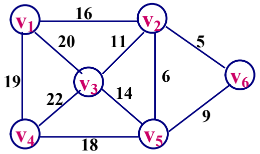
```c
#define MAXVER 512
#define INFINITY 32767
void Prim(int weights[][MAXVER], int n, int src, int edges[])
{ // weights为权重数组、n为顶点个数、src为最小树的第一个顶点、edge为最小生成树边
    int minweight[MAXVER], min;
    int i, j, k;
    for (i = 0; i < n; i++){   //初始化相关数组
        minweight[i] = weights[src][i]; //将src顶点与之有边的权值存入数组
        edges[i] = src;     //初始化第一个顶点为src
    }
    minweight[src] = 0; //将第一个顶点src顶点加入生成树
    for (i = 1; i < n; i++)  {  min = INFINITY;
        for (j = 0, k = 0; j < n; j++)
            if (minweight[j] != 0 && minweight[j] < min) { //在数组中找最小值，其下标为k
                min = minweigth[j];   k = j;
            }
        minweight[k] = 0; //找到最小树的一个顶点
        for (j = 0; j < n; j++)
            if (minweight[j] != 0 && weights[k][j] < minweight[j]) {
                minweight[j] = weights[k][j]; //将小于当前权值的边(k,j)权值加入数组中
                edges[j] = k;                  //将边(j,k)信息存入边数组中
            }
    }
}
```

4.4 克鲁斯卡尔(Kruskal)算法

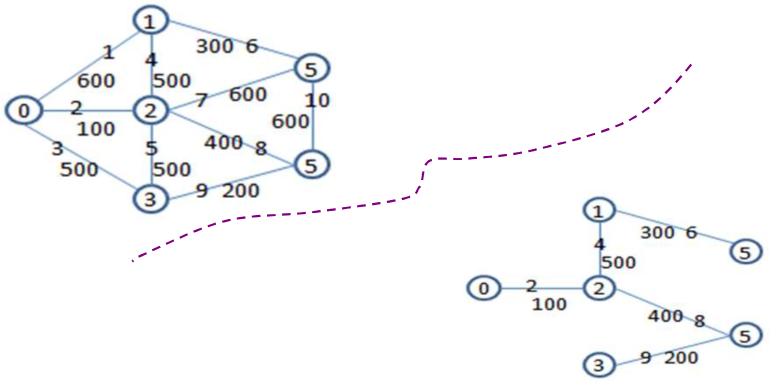
◆ **基本思想**

出发点：为使生成树上边的权值之和达到最小，则应使生成树中每一条边的权值尽可能地小。

具体做法: 先构造一个只含n个顶点的子图T，然后从权值最小的边开始，若它的添加不使T中产生回路，则在T上加上这条边，如此重复，直至加上n-1条边为止。



75



```
#define MaxV   <最大顶点个数>
#define MaxE   <最大边数>
typedef struct  edge{
      int v1, v2;
      int  weight;
}Edge;

Vertype  Vertex[MaxV];

Edge  G[MaxE];
```

定义边类型

顶点信息数组

边集数组

边集数组

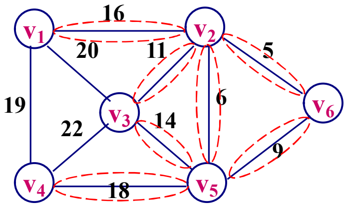
- 稀疏图

将边按权值从小到大的顺序排列的。

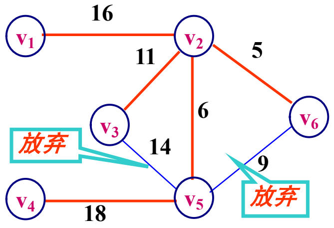
例

5	V_2	V_6
6	V_2	V_5
9	V_5	V_6
11	V_2	V_3
14	V_3	V_5
16	V_1	V_2
18	V_4	V_5
19	V_1	V_4
20	V_1	V_3
22	V_3	V_4

G(连通图)

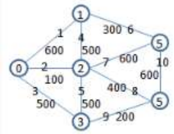


T(生成树)



问题：北航网络中心铺设光缆

【样例输入】
6 10
1 0 1 600
2 0 2 100
3 0 3 500
4 1 2 500
5 2 3 500
6 1 4 300
7 2 4 600
8 2 5 400
9 3 5 200
10 4 5 600



设计考虑:

1. 可用邻接矩阵存储网络图，数据结构：
   struct edge graph[MAXVER][MAXVER]; //邻接矩阵
   int edges[MAXVER]={0}; //生成树数组
   根据输入值对<id,v1,v2,wei>构造图：
   graph[v1][v2].id = id; graph[v1][v2].weight = wei;
   graph[v2][v1].id = id; graph[v2][v1].weight = wei;
2. 调用Prim算法得到最小生成树，存放在edges数组中
3. 根据生成树数组edges可得到生成树边序号为 graph[i][**edges[i]**].id的边，其权重为： graph[i][**edges[i]**].wei
4. 最小生成树按边序号进行排序输出.

```
struct edge {
    int id;
    int wei;
};
```

本算法的关键是如何判断选取的边是否产生与生成树中已保留的边形成回路?

- **若选定边的两个顶点不在同一个连通分量内，可加入该边。**
- **设辅助数组vset[MAXV],其值为顶点所属连通子图的编号。**

判断下列说法是否正确?

- 任意连通图中，假设没有相同权值的边存在，则权值最小的边一定是其最小生成树中的边。 ✓
- 任意连通图中，假设没有相同权值的边存在，则权值最大的边一定不是其最小生成树中的边。 ✗
- 任意连通图中，假设没有相同权值的边存在，则与同一顶点相连的权值最小的边一定是其最小生成树中的边。 ✓
- 采用克鲁斯卡尔算法求最小生成树的过程中，判断一条待加入的边是否形成回路，只需要判断该边的两个顶点是否都已经加入到集合U中。 ✗

对比两个算法，Kruskal算法主要是针对边展开，边数少时效率会非常高，所以对稀疏图有很大的优势；Prim算法对于稠密图，即边数非常多的情况会好一些

比较两种算法

算法名	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	O(eloge)
适应范围	稠密图	稀疏图

延伸阅读*：
上面我们介绍了*普里姆(Prim)算法*和*克鲁斯卡尔(Kruskal)算法*的基本原理，请同学自学这*Kruskal*算法的C实现。