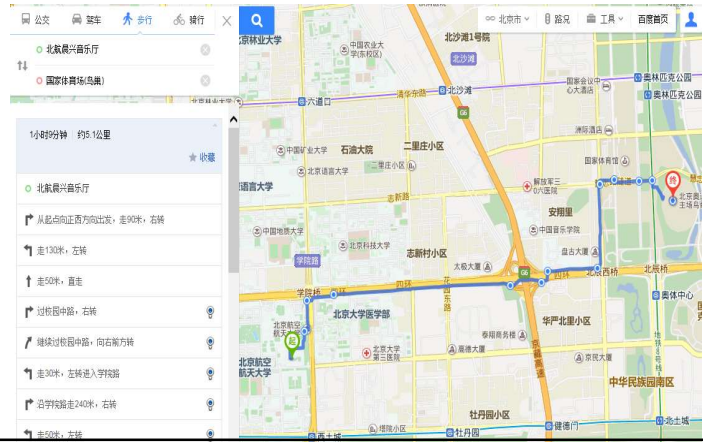
北京航空航天大学
BEIHANG UNIVERSITY


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

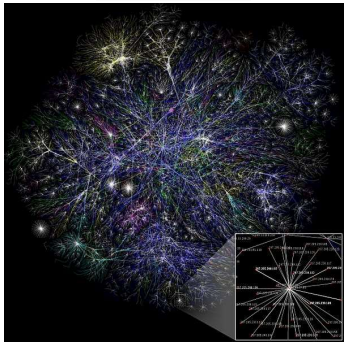


内容提要

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*

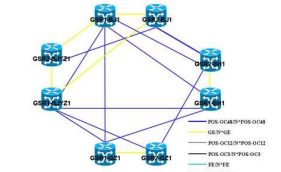
图：网络拓扑

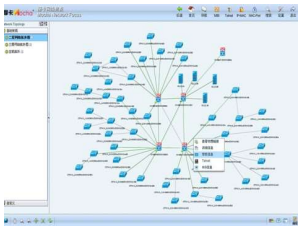
Internet Map



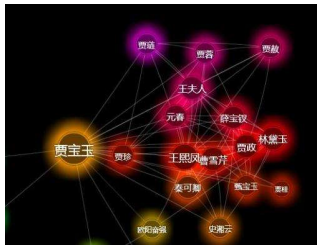
计算机网络

CNCNET骨干网超级核心节点之间互连结构图





图：社交网络（Social Network）



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*

地铁线路图

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)

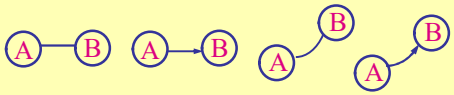
其中, V 为顶点集合, E 为关系(边或弧)的集合。

非空有穷集合

图中的顶点和关系(边)

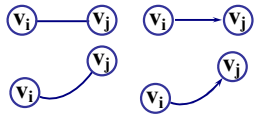
一个数据元素----顶点(Vertex) A

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

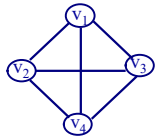
(v_i, v_j) 或 $<v_i, v_j>$

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

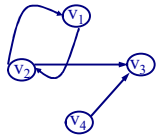


$G_1=(V_1, E_1)$

其中

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

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



$G_2=(V_2, E_2)$

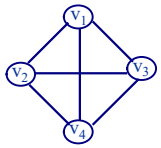
其中

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

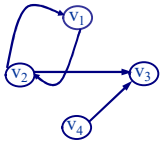
$E_2 = \{ <v_1,v_2>, <v_2,v_1>, <v_2,v_3>, <v_4,v_3> \}$

图的分类

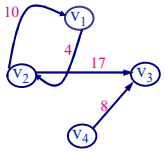
- 无向图　对于（v_i,v_j）∈E,必有（v_j,v_i）∈E，并且偶对中顶点的前后顺序无关。
- 有向图　若<v_i,v_j>∈E是顶点的有序偶对。
- 网(络)　与边有关的数据称为**权**，边上带权的图称为**网络**。



无向图



有向图



网(络)

顶点的度

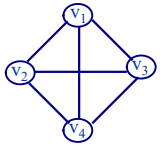
依附于顶点v_i的边的数目，记为$TD(v_i)$。

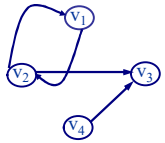
对于有向图而言，有:

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

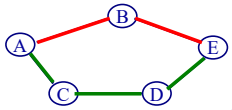
结论3　具有n个顶点的有向图最多有$n(n-1)$条边

- 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}), … ,(v_{im},v_y) 或 <v_x,v_{i1}>, <v_{i1},v_{i2}>, … ,<v_{im},v_y> 都在E中



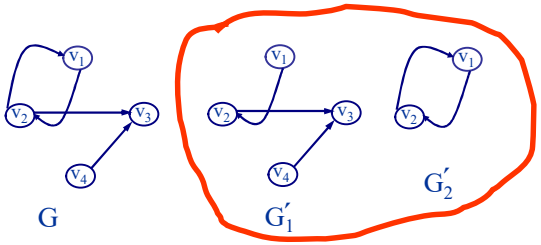
P(A, E): A, B, E　(第1条路径)

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。



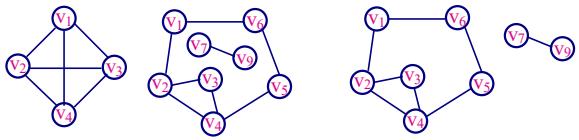
17

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。



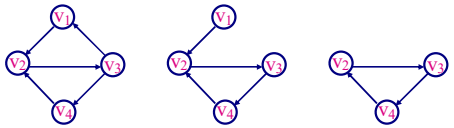
18

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。

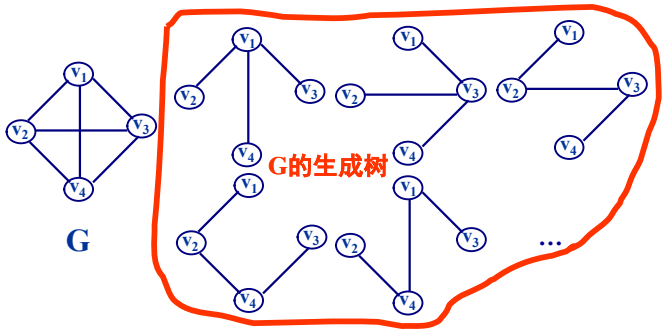
强连通分量 —— 有向图中的极大强连通子图。



19

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



20

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

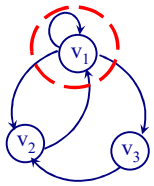
   <=>无回路且连通

   <=>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

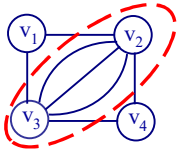
本章不讨论的图

1. 带自身环的图



2. 多重图

（作业中有一个题）



只讨论简单图

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*

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

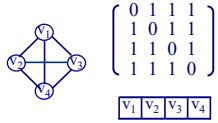
邻接矩阵特点

(1) 无向图的邻接矩阵一定是一个对称矩阵。

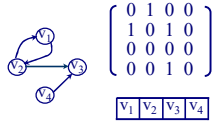
(2) 不带权的有向图的邻接矩阵一般是稀疏矩阵。

(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。



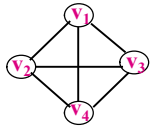
(无向图)



(有向图)

空间复杂度$O(n^2)$

示例

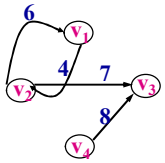


VERTEX1[0..3]

0 v1
1 v2
2 v3
3 v4

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

邻接矩阵

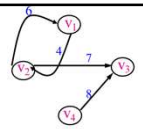


VERTEX2[0..3]

0 v1
1 v2
2 v3
3 v4

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

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。

"个头结点怎么存储?

0 v1 ^
1 v2 ^
2 v3 ^
3 v4 ^

一维数组!

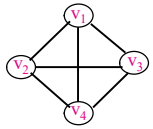
2. 第i个链表中的每一个链结点(称之为**边结点**)表示以第i个顶点为**出发点**的一条边;边结点的构造为：

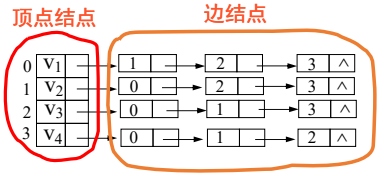
adjvex	weight	next

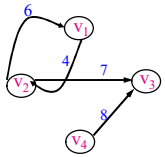
其中,next 域为指针域;
weight 域为权值域(若图不带权,则无此域);
adjvex 域存放以第i 个顶点为出发点的一条边的**另一端点在头结点数组中的位置**。

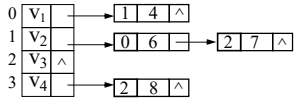
0 v1 ^
1 v2 ^
2 v3 ^
3 v4 ^

示例

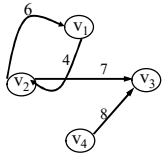


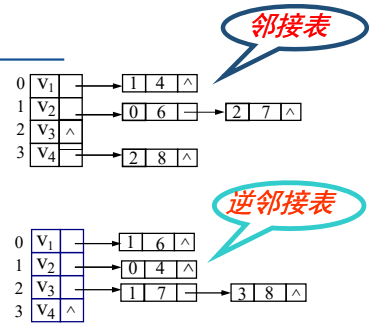






逆邻接表



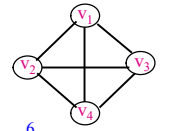


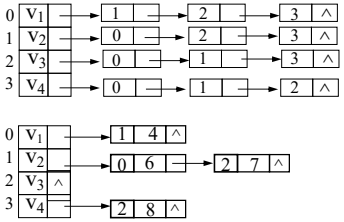
对于邻接表

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

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**。

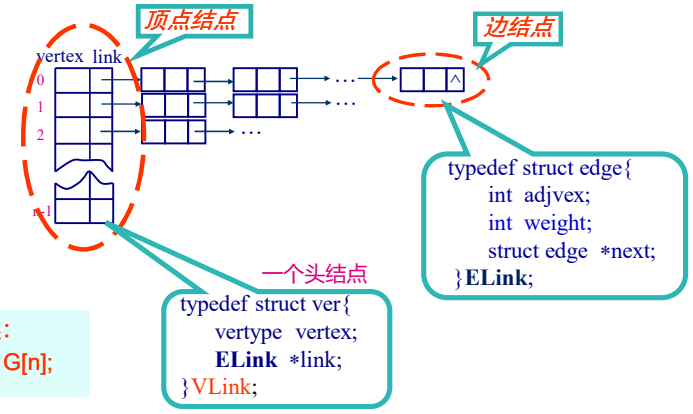
(2)有向图的第i个链表中边结点个数是第i个顶点的**出度**。

(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。

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



图的描述：
Vlink G[n];

33

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

35

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

34

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

36

2.3 图的基本操作

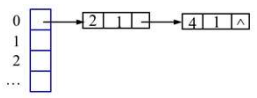
```
createGraph();     // 创建一个图
destoryGraph();    // 删除一个图
insertVex(v);      // 在图中插入一个顶点v
deleteVex(v);      // 在图中删除一个顶点v
insertEdge(v, w); // 在图中插入一条边<v,w>
deleteEdge(v, w); // 在图中删除一条边<v,w>
traverseGraph();  // 遍历一个图
```


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);
    }
  }
}
```

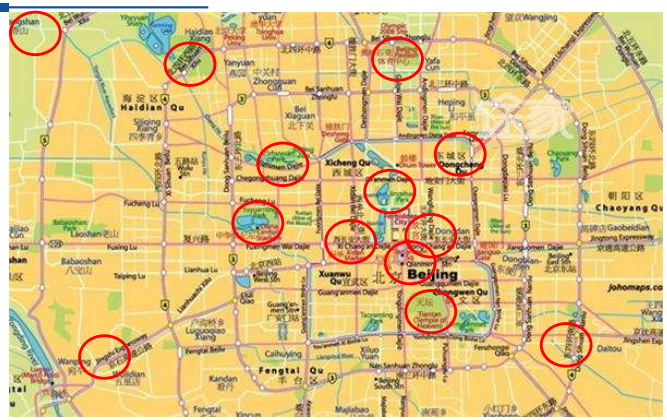
邻接矩阵：
graph[v1][v2]= graph[v2][v1]= 1;

```
#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?



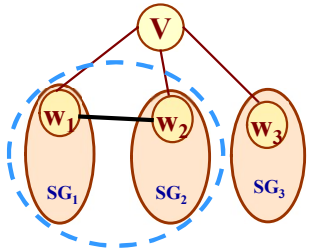
3. 图的遍历

- 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

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。

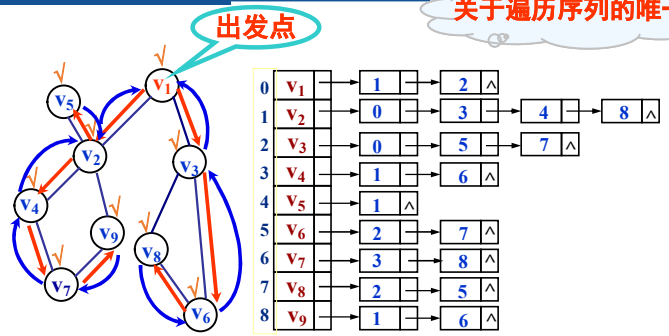
完成一个连通分量的遍历

递归过程

示例：深度优先遍历

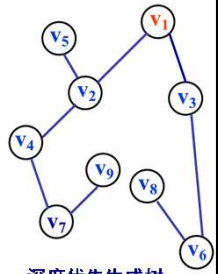
出发点

关于遍历序列的唯一性



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

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

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

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$O(n^2)$。

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]);
    }
}
```


3.2 广度优先遍历(Breadth First Search,BFS)

◆ **原则**

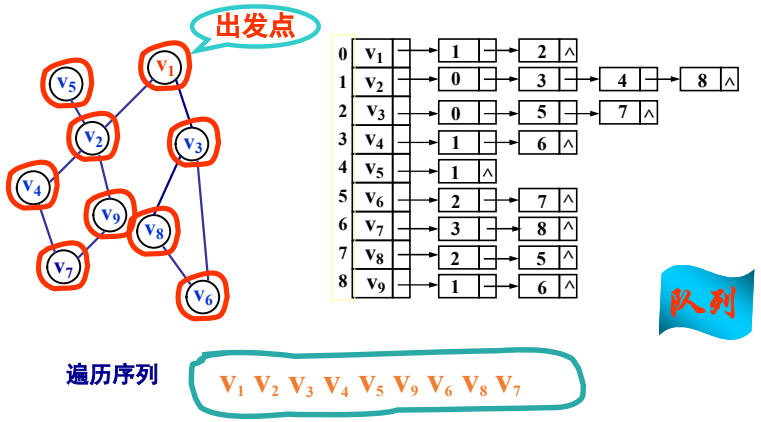
类似于树的按层次遍历

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

示例



49

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

51

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：$O(n^2)$

50

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。

52

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：

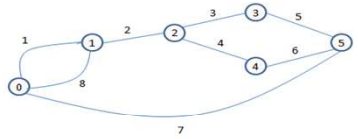
<n> <e>
<e1> <vi1> <vj1>
<e2> <vi2> <vj2>
...
<en> <vin> <vjn>

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。

53

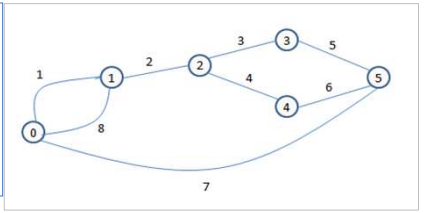
问题：独立路径计算 – 数据结构设计

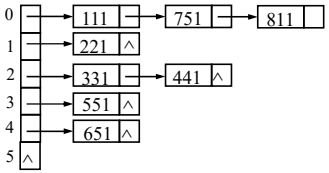
采用邻接表来存储图，邻接表设计如下：

```
#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

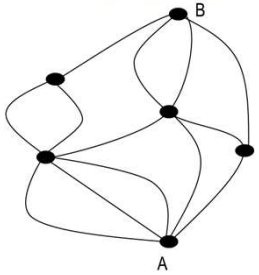




55

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）。



54

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);
```

56

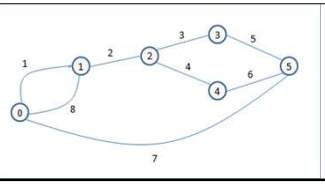
问题：独立路径计算 – 主要代码*

```
//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);
}
```

V_1

V_0



57

问题：独立路径计算 – 主要代码*

```
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;
}
```

```
#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
```

```
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*

```
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;
}
```

```
#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;
};
```

【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

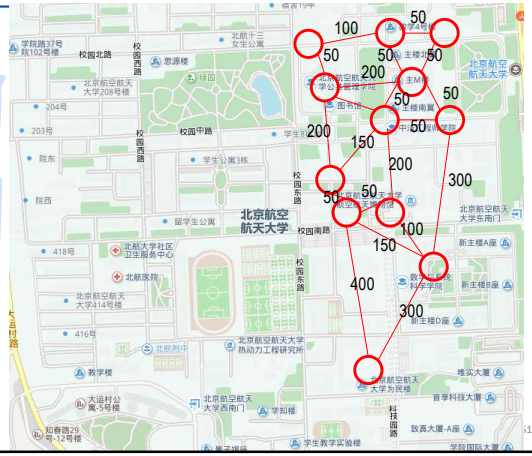
1. 图的基本概念
2. 图的存储结构
3. 图的遍历
4. 最小生成树
5. 最短路径
6. AOV网与拓扑排序*
7. AOE网与关键路径*
8. 网络流量问题*

60

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?



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

63

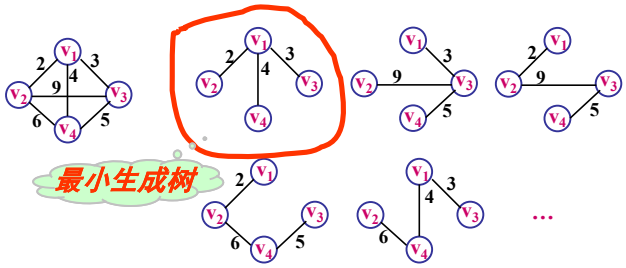
4.1 生成树

- 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**。

62

最小生成树

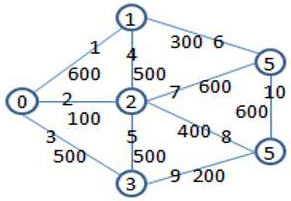
- 在带权连通图中，总的权值之**和最小**的带权生成树为**最小生成树**，也称为最小代价生成树,或最小花费生成树。



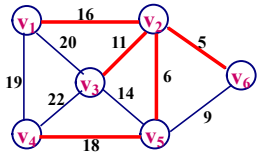
64

4.2 构造最小生成树

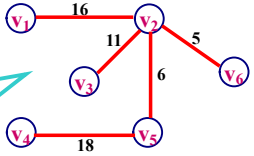
- **构造原则**
  - ✓ 只能利用图中的边来构造最小生成树；
  - ✓ 只能使用、且仅能使用图中的n-1条边来连接图中的n个顶点；
  - ✓ 不能使用图中产生回路的边。
- **构造算法**
  - ✓ 普里姆(Prim)算法
  - ✓ 克鲁斯卡尔(Kruskal)算法



例



最小生成树



最小生成树的权值 = 56

G:

V = { v1, v2, v3, v4, v5, v6 }

GE= { (v1, v2)16, (v1, v3)20, (v1, v4)19, (v2, v3)11, (v2, v5)6, (v2, v6)5, (v3, v4)22, (v3, v5)14, (v4, v5)18, (v5, v6)9 }

T:

U = { v1, v2, v6, v5, v3, v4 }

TE = { (v1, v2)16 (v2, v6)5 (v2, v5)6 (v2, v3)11 (v4, v5)18 }

最小生成树唯一吗?

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。

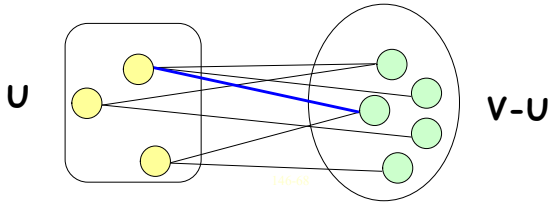
当追求的目标是一个问题的最优解时,

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

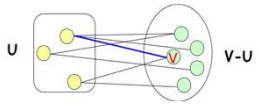
4.3 普里姆算法

- **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；

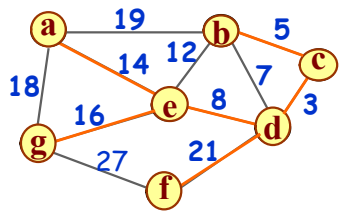
(2) 重复以下步骤n-1次,使得其他n-1个顶点被加入到U中：

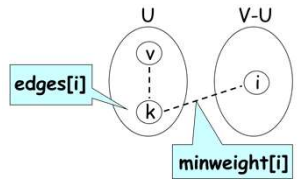
① 从候选边中挑选权值最小的边输出,设该边在V-U中的顶点是v,将v加入U中,删除U中其它顶点与v关联的边；

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。

例如：



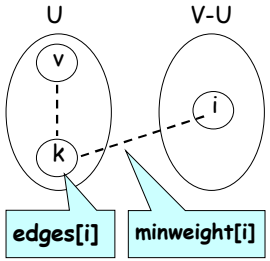


	0	1	2	3	4	5	6
	a	b	c	d	e	f	g
edges		c	d	e	a	d	e
minweight	0	0	0	0	0	0	0

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];

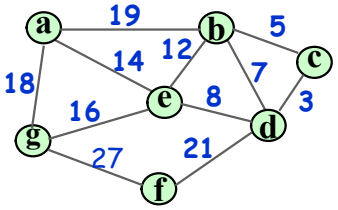
存放未确定为生成树的顶点至已确定的生成树上顶点的边权重,minweight[i] = 0 表示其已确定为最小生成树顶点

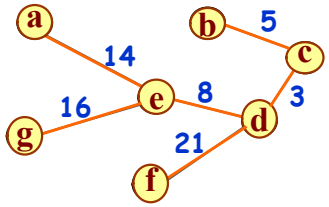
int edges[MAXVER];

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

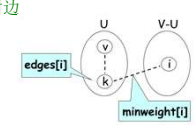
用普里姆算法求最小生成树





算法实现

```
#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
            }
    }
}
```

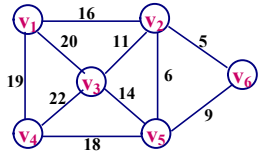


4.4 克鲁斯卡尔(Kruskal)算法

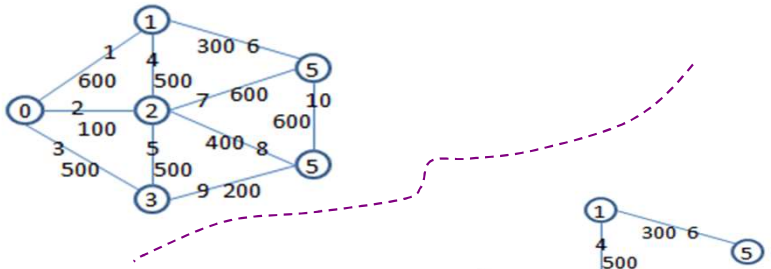
◆ **基本思想**

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

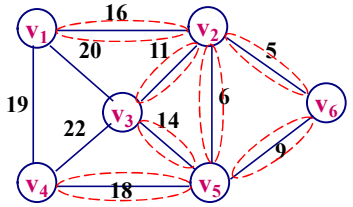
边集数组
- 稀疏图

将边按权值从小到大的顺序排列的。

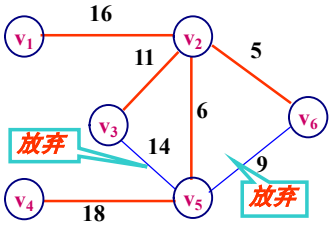
例

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

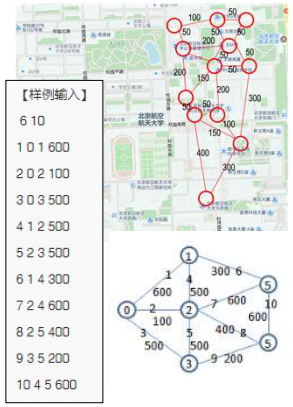
G(连通图)



T(生成树)



问题：北航网络中心铺设光缆



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。

问题：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

最短路径问题

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*

5. 最短路径问题

◆ 路径长度的定义

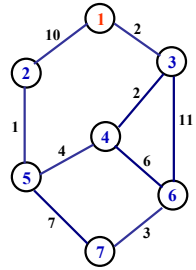
不带权的图：路径上所经过的边的数目。

带权的图：路径上所经过的边的权值之和。

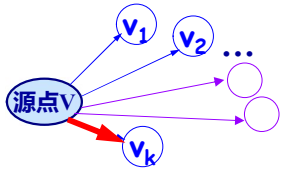
◆ 单源最短路径问题

设出发顶点为v(通常称为源点)。

1.单源点最短路径；
2.每对顶点之间的最短路径；
3.求图中第1短、第2短、...的最短路径。



求单源最短路径算法



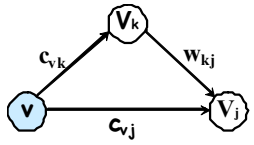
◆ **依最短路径的长度递增的次序求得各条路径**

若从源点V到顶点V_i的最短路径是所有最短路径中长度最短者。

1）长度最短的路径：在这条路径上，必定只含一条弧，并且这条弧的权值最小。如，从源点到V_k是长度最短路径。

2）长度次短的路径可能有两种情况：

- 可能是直接从源点到该点的路径(只包含一条弧)
- 或者从源点经过顶点v_k，再到达该顶点(由两条弧组成)



3）其余依次类推

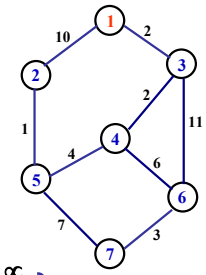
V到V_j的最小距离＝MIN(c_{vk}+w_{kj}, c_{vj})

5.1 单源最短路径迪杰斯特拉算法

◆ 图的存储

以1~n 分别代表n个顶点，采用邻接矩阵存储该图,有

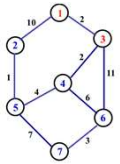
$$Weights[i][j]=\begin{cases} W_{ij} & \text{当顶点}v_i\text{到顶点}v_j\text{有边，且权为}W_{ij} \\ \infty & \text{当顶点}v_i\text{到顶点}v_j\text{无边时} \\ 0 & \text{当}v_i=v_j\text{时} \end{cases}$$



邻接矩阵

$$\begin{pmatrix} 0 & 10 & 2 & \infty & \infty & \infty & \infty \\ 10 & 0 & \infty & \infty & 1 & \infty & \infty \\ 2 & \infty & 0 & 2 & \infty & 11 & \infty \\ \infty & \infty & 2 & 0 & 4 & 6 & \infty \\ \infty & 1 & \infty & 4 & 0 & \infty & 7 \\ \infty & \infty & 11 & 6 & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & 7 & 3 & 0 \end{pmatrix}$$

5.1 单源最短路径迪杰斯特拉算法



◆ **基本思想**

设**G=(V,E)**是一个带权图，把图中顶点集合**V**分成两组：

第一组为已求出最短路径的顶点集合(用**S**表示，初始时**S**中只有一个源点，以后每求得一条最短路径**v**, …v_k，就将v_k加入到集合**S**中，直到全部顶点都加入到**S**中。)

第二组为其余未确定最短路径的顶点集合(用**U**表示)。

按最短路径长度的递增次序依次把第二组的顶点加入**S**中。在加入的过程中，总保持从源点**v**到**S**中各顶点的最短路径长度不大于从源点**v**到**U**中任何顶点的最短路径长度。

每个顶点对应一个距离：

- **S**中的顶点的距离是：从**v**到此顶点的最短路径长度，
- **U**中的顶点的距离是：从**v**到此顶点只包括以**S**中的顶点为中间顶点的当前最短路径长度。

5.1 单源最短路径迪杰斯特拉算法

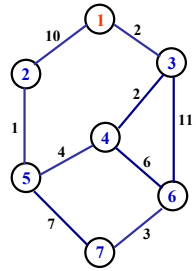
◆ **从源点到当前各顶点的最短路径长度的存储**

用数组**Sweight[i]**表示，一般情况下，

Sweight[k] = <源点到顶点 k 的弧上的权值>

或者 = <源点到S中其它顶点的路径长度>

+ <S中其它顶点到顶点 k 的弧上的权值>



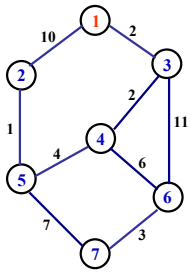
初始时，

Sweight数组的值为邻接矩阵中顶点v对应的边值信息。

$$\begin{pmatrix} 0 & 10 & 2 & \infty & \infty & \infty & \infty \\ 10 & 0 & \infty & \infty & 1 & \infty & \infty \\ 2 & \infty & 0 & 2 & \infty & 11 & \infty \\ \infty & \infty & 2 & 0 & 4 & 6 & \infty \\ \infty & 1 & \infty & 4 & 0 & \infty & 7 \\ \infty & \infty & 11 & 6 & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & 7 & 3 & 0 \end{pmatrix}$$

邻接矩阵

5.1 单源最短路径迪杰斯特拉算法



◆ 标识当前已找到最短路径的顶点

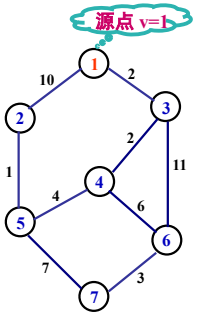
用数组**wfound [1..n]** 表示，有

$$wfound[k] = \begin{cases} 1 & \text{已经找到最短路径} \\ 0 & \text{尚未找到最短路径} \end{cases}$$

初始时，

wfound [v]=1，wfound[k]=0 (k=1，2，…，n，且k ≠v)

例



Sweight数组

0	10	2	∝	∝	∝	∝

wfound数组

1	0	0	0	0	0	0

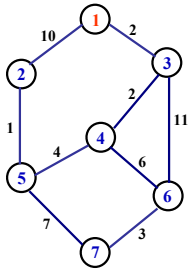
Spath数组

1	1	1				

邻接矩阵

0	10	2	∝	∝	∝	∝
10	0	∝	∝	1	∝	∝
2	∝	0	2	∝	11	∝
∝	∝	2	0	4	6	∝
∝	1	∝	4	0	∝	7
∝	∝	11	6	∝	0	3
∝	∝	∝	∝	7	3	0

记录各顶点最短路径所经过的顶点序列



用数组**Spath[1..n]** 表示，其中

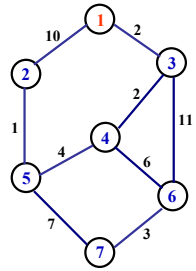
Spath[k]保存从源点到顶点v_i当前最短路径中的前一个顶点的编号，初始时，

当源点V到V_k有边时，Spath [k]={v}，

k=1，2，…，n

当源点V到V_k无边时，Spath [k]=-1

具体算法描述

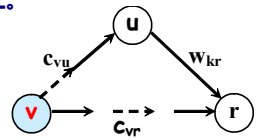


1. 确定Sweight[、wfound、Spath三个数组的初值。
2. 利用wfound数组与Sweight数组在那些尚未找到最短路径的顶点中确定一个与源点v最近的顶点u，并置wfound [u]为1。
3. 根据顶点u修改源点到所有尚未找到最短路径的顶点的路径长度，并修改相应Spath值。

   1)将源点v到顶点u的(最短)路径长度分别加到源点v通过顶点u可以直接到达、且尚未找到最短路径的那些顶点的路径长度上。

   $Sweight[[r]=MIN(c_{vu}+w_{ur},c_{vr})$

   2)若替换，则Spath[r]=u。

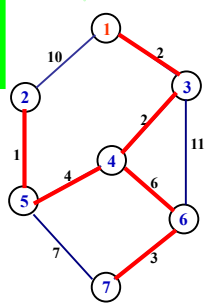


4. 重复上述过程的第2至第3步n-1次。

顶点	①	②	③	④	⑤	⑥	⑦
Sweight	0	10	2	∞	∞	∞	∞
	0	10	2	4	∞	13	∞
	0	10	2	4	8	10	∞
	0	9	2	4	8	10	15
	0	9	2	4	8	10	15
	0	9	2	4	8	10	13
	0	9	2	4	8	10	13
wfound	1	1	1	1	1	1	1
Spath	①	⑤	①	③	④	④	⑥

路径长度

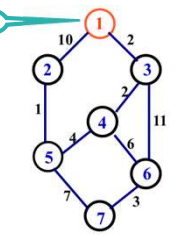
源点：v =1



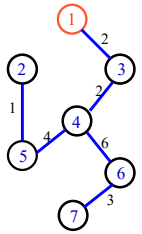
最短路径最后经由点

问题

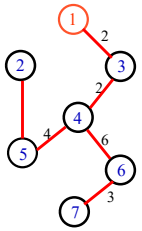
源点



最小生成树



相同



最短（路径）生成树

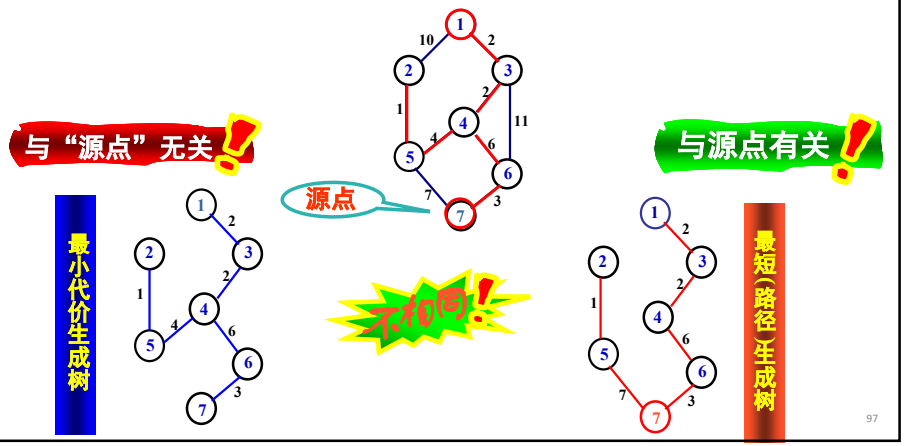
算法实现

```
Dijkstra(int v0){
    int i, j, v, minweight;
    char wfound[VNUM] = {0}; //用于标记从v0到相应顶点是否找到最短路径，0未找到，1找到
    for (i = 0; i < VNUM; i++) {
        Sweight[i] = Weights[v0][i];  Spath[i] = v0; } //初始化数组Sweight和Spath
    Sweight[v0] = 0;     wfound[v0] = 1;
    for (i = 0; i < VNUM - 1; i++) { //迭代VNUM-1次
        minweight = INFINITY;
        for (j = 0; j < VNUM; j++) //找到未标记的最小权重值顶点
            if (!wfound[j] && (Sweight[j] < minweight))  {
                v = j;  minweight = Sweight[v]; }
        wfound[v] = 1;          //标记该顶点为已找到最短路径
        for(j=0; j<VNUM; j++)//找到未标记顶点且权值大于v权值+(v,j)的权值，更新其权值
            if (!wfound[j] && (minweight + Weights[v][j] < Sweight[j])) {
                Sweight[j] = minweight + Weights[v][j];
                Spath[j] = v; //记录前驱顶点
            }
    }
}
```


对于给定的带权连通无向图，从某源点到图中各顶点的最短路径构成的生成树是否是该图的最小生成树？

为什么？

思考：最小生成树与最短路径生成树



97

问题：北京地铁乘坐线路查询

文件bgstations.txt为数据文件，包含了北京地铁的所有线路及所有车站信息。其格式如下：

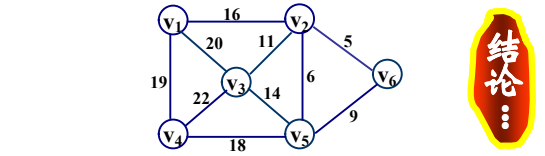
```
13
1 23
苹果园 0
古城 0
...
公主坟 1
...
四惠东 1

2 19
西直门 1
积水潭 0
...
西直门
```

说明：表明目前北京地铁共开通13条线，其中1号线有23个车站，分别 为苹果园，非换乘站；...；公主坟，换乘站....。2线共有19个站，分别为西直门，换乘站，...。

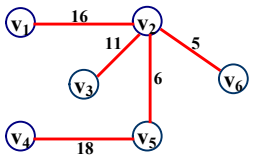
输入： *起始站：西土城*
目的站：北京西站

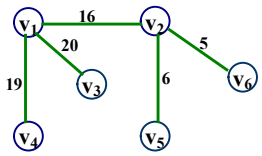
输出： *西土城-10(1)-知春路-13(2)-西直门-4(2)-国家图书馆-9(4)-北京西站*

再看一个例子



结论……





最小生成树　　源点为v_1的最短路径

98

```
13
1 23
苹果园 0
古城 0
八角游乐园 0
八宝山 0
玉泉路 0
五棵松 0
万寿路 0
公主坟 1
军事博物馆 1
木樨地 0
南礼士路 0
复兴门 1
西单 1
...
2 19
西直门 1
积水潭 0
鼓楼大街 1
...
西直门 1
...
```

```
13
1 23
苹果园 0
古城 0
...
公主坟 1
...

2 19
西直门 1
积水潭 0
...
西直门 1
```

BGvertex[512]

0	牡丹园	0
1	西土城	0
2	知春路	1
3	知春里	0
4	五道口	0
5	大钟寺	0
6	西直门	1
7	动物园	0

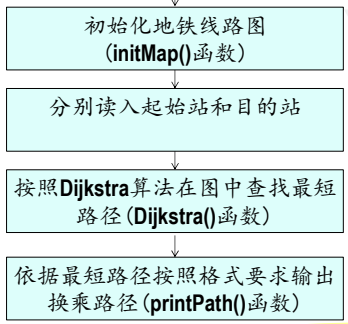
BGweights[512][512];

	0	1	2	3	4	5	6	7
0		1, 10						
1	1, 10		1, 10					
2		1, 10		1, 10	1, 13	1, 13		
3			1,10					
4			1, 13					
5			1, 13				1, 13	
6						1, 13		1, 4
7							1, 4	

100

问题分析与算法设计

主算法流程



1. 依次读入每条地铁线路的站名信息
   1.1 将站信息加入到站信息数组中（注意：站名是唯一的，每个站在该数组中的下标即为图的顶点编号）（即图的顶点数组）
   1.2 将每条线路的当前站和其前序站构成的边(v1, v2)加入到图顶点权重数组中（注：在权重数组中权重信息包括两站间的站数（缺省为1）以及所属线路）

该方法的优点是简单，缺点是所有站都在图中，性能低。

主要代码实现

```
void initMap(){
    FILE *fp;
    int i, j, snum, lno, lnum, v1, v2;
    struct station st;
    …
    fscanf(fp, "%d", &snum);
    for (i = 0; i < snum; i++)  {
        fscanf(fp, "%d %d", &lno, &lnum);    v1 = v2 = -1;
        for (j = 0; j < lnum; j++) {
            fscanf(fp, "%s %d", st.sname, &st.ischange);
            v2 = addVertex(st); //将该站加到站信息数组中，返回其下标
            if (v1 != -1) {
                BGweights[v1][v2].wei = BGweights[v2][v1].wei = 1;
                BGweights[v1][v2].lno = BGweights[v2][v1].lno = lno;
            }
            v1 = v2;
        }
    }
    fclose(fp);
}
```

```
struct weight {
    int wei; //两个站间的权重，即相差站数，缺省为1
    int lno; //两个顶点所在的线号
};
struct  station { //车站信息
    char  sname[MAXLEN];    //车站名
    int ischange;        //是否为换乘站，0-否，1-换乘
};
```

输入：
13
1 23
苹果园 0
古城 0

```
struct station BGvertex[MAXNUM];  //地铁网络图顶点数组
struct weight BGweights[MAXNUM][MAXNUM]; //网络图权重数组，邻接矩阵
int Vnum = 0;    //实际地铁总站数
```


数据结构设计

```
#define MAXNUM 512    //地铁最大站数
#define MAXLEN 16     //地铁站名的最大长度
#define INFINITY  32767

struct  station { //车站信息
    char  sname[MAXLEN];    //车站名
    int ischange;        //是否为换乘站，0-否，1-换乘
};
struct weight {
    int wei;        //两个站间的权重，即相差站数，缺省为1
    int lno;        //两个顶点所在的线号
};
struct station BGvertex[MAXNUM];        //地铁网络图顶点数组
struct weight BGweights[MAXNUM][MAXNUM];//网络图权重数组，邻接矩阵
int Vnum = 0;    //实际地铁总站数
```


主要代码实现

```
void Dijkstra(int v0, int v1, int spath[]){
    int i, j, v, minweight;
    char wfound[MAXNUM] = {0}; //用于标记从v0到相应顶点是否找到最短路径，0未找到，1找到
    int sweight[MAXNUM] = {0};
    for (i = 0; i < Vnum; i++){
        sweight[i] = BGweights[v0][i].wei; //初始化数组Sweight
        spath[i] = v0;                     //初始化数组spath
    }
    sweight[v0] = 0;    wfound[v0] = 1;
    for (i = 0; i < Vnum - 1; i++)  { //迭代VNUM-1次
        minweight = INFINITY;
        for (j = 0; j < Vnum; j++) //找到未标记的最小权重值顶点
            if (!wfound[j] && (sweight[j] < minweight)) {
                v = j; minweight = sweight[v];  }
        wfound[v] = 1; //标记该顶点为已找到最短路径
        if (v == v1)  return;                //找到(v0,v1)最短路径，返回
        for (j = 0; j < Vnum; j++) //找到未标记顶点且其权值大于v的权值+(v,j)的权值，更新其权值
          if(!wfound[j]&&(BGweights[v][j].lno>0)&&(minweight+BGweights[v][j].wei<sweight[j])){
            sweight[j] = minweight + BGweights[v][j].wei;
            spath[j] = v; //记录前驱顶点
        }
    } }
```

思考一下如何计算多种换乘？（即存在多个最短路径）

输入：
起始站：西土城
目的站：北京西站

主要代码实现

```
void printPath(int v0, int v1, int spath[]){
    char path[80] = {0}, buf[80];
    //bcount为从v0到v1最短路径的乘坐站数，sc为乘坐某一线路的站数
    int board[80], bcount = 0, line = -1, sc = 0;
    int i;
    do { //获得乘坐站序列，该序列为倒的，即起始站在最后
        board[bcount++] = v1;
    } while ((v1 = spath[v1]) != v0);
    board[bcount++] = v0;
    line = BGweights[board[bcount - 1]][board[bcount - 2]].lno;
    sprintf(buf, "%s - %d(", BGvertex[board[bcount - 1]].sname, line);
    strcpy(path, buf);    sc = 1;
    for (i = bcount - 2; i > 0; i--, sc++)
        if (BGweights[board[i]][board[i - 1]].lno != line)    {
            line = BGweights[board[i]][board[i - 1]].lno;
            sprintf(buf, "%d) - %s - %d(", sc, BGvertex[board[i]].sname, line);
            strcat(path, buf);  sc = 0;
        }
    sprintf(buf, "%d) - %s\n", sc, BGvertex[board[i]].sname);
    strcat(path, buf);
    puts(path);
}
```

输出：*西土城-10(1)-知春路-13(2)-西直门-4(2)-国家图书馆-9(4)-北京西站*

更多的最短路径算法*

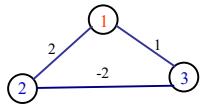
- 除Dijkstra算法，其他最短路径算法
  - ✓Floyd算法（Floyd-Warshall算法）
    - 非常简洁，特别适用于计算所有顶点间的最短路径问题
    - 算法复杂度为$O(n^3)$
  - ✓Bellman-Ford算法
    - 从DJ算法引申出来的，可以解决带有负权边的最短路径问题
  - ✓SPFA算法
    - 可以解决负权边的问题，且复杂度比Ford低
    - 可以看做是dj算法和BFS算法
  - ✓A*算法

在实际应用时，可事先采用Floyd算法计算出地铁所有站间的最短路径，这样每次查询时不用重新计算，直接显示就行。

Dijkstra算法的局限性*

对于带有负权值的图

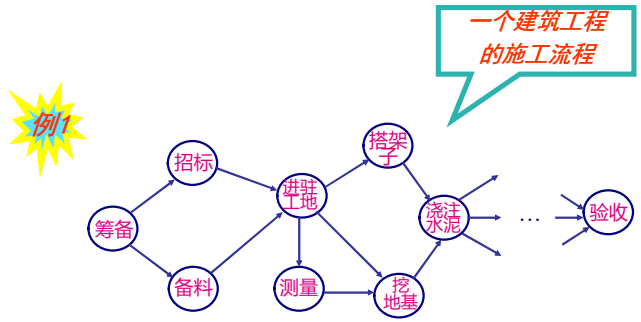
- 考虑从1-3的最短路径
  - ✓1->3：路径长度1，Dijkstra输出结果
  - ✓1->2->3：路径长度0，更短！
- 问题所在？
  - ✓Dijkstra算法要求已经确定的最短路径在后续算法执行时不会被修改。即路径长度要单调递增。
  - ✓解决方法：re-weighting，每条路径增加2？仍然不解决问题！
  - ✓如果**存在负回路**，则最短路径可以任意负下去！
  - ✓直接求s-t的简单最短路径是难解问题！
  - ✓Bellman-Ford algorithm（无负回路）



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*

6. AOV网与拓扑排序*

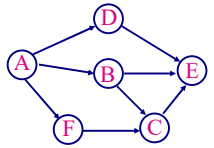


6.1 AOV定义

◆ **以顶点表示活动，以有向边表示活动之间的优先关系的有向图称为顶点表示活动的网(Activity On Vertex Network),简称AOV网。**

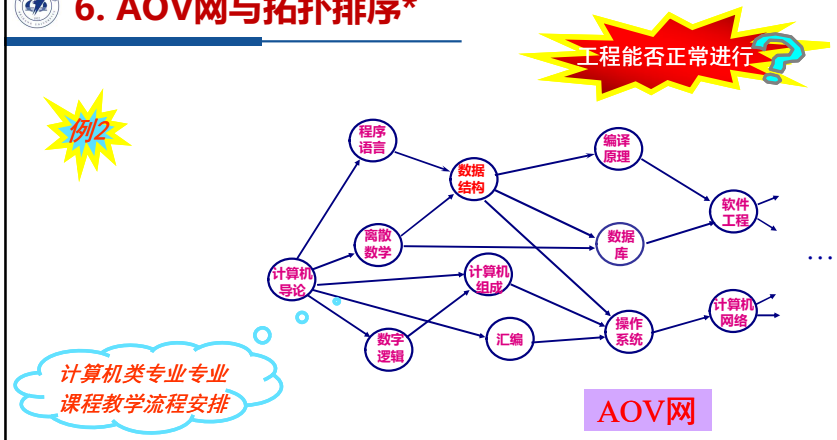
在AOV网中，若顶点i到顶点j之间有路径,则称顶点i为顶点j的前驱，顶点j为顶点i的后继；

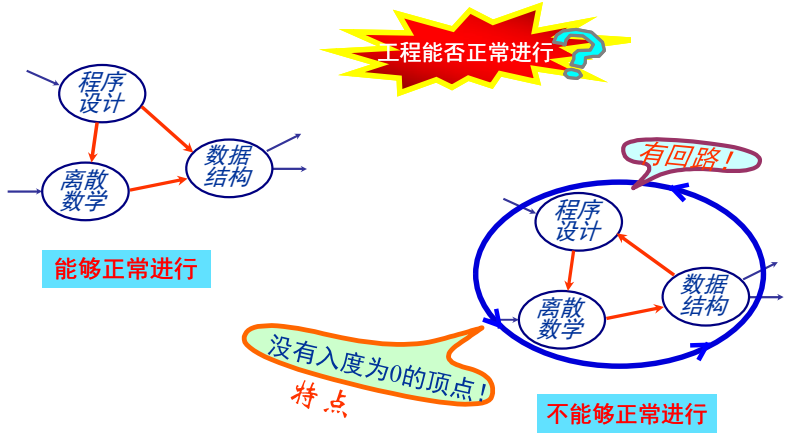
若顶点i到顶点j之间为一条有向边，则称顶点i为顶点j的直接前驱，顶点j为顶点i的直接后继。



一个AOV网

6. AOV网与拓扑排序*





6.2 拓扑排序

◆ **定义**

检测工程能否正常进行，首先要判断对应的AOV网中是否存在回路，达到该目的最有效的方法之一是对AOV网构造其顶点的拓扑序列，即对AOV网进行 **拓扑排序** 。

离散数学：由某个集合上的一个偏序得到该集合上的一个全序的操作称为拓扑排序

设G=(V,E)是一个具有n个顶点的有向图，V中的顶点序列v1,v2,…,vn，满足若从顶点vi到vj有一条路径，则在顶点序列中顶点vi必在顶点vj之前，则称这样的顶点序列为一个拓扑序列。构造拓扑序列的过程就是拓扑排序。

6.2 拓扑排序

◆ **拓扑排序原则**

构造AOV网的一个顶点序列，使得该顶点序列满足下列条件：

1. 若在AOV网中,顶点i 优先于顶点j,则在该序列中顶点i 仍然优先于顶点j；
2. 若在AOV网中，顶点i 与顶点j之间不存在优先关系,则在该序列中**建立它们的优先关系**,即顶点i优先于顶点j，或者顶点j 优先于顶点i；
3. 若能构造出这样的拓扑序列，则拓扑序列包含AOV网的全部顶点，**说明AOV网中没有回路**

若构造不出这样的序列，说明AOV网中存在回路

6.2 拓扑排序

◆ **拓扑排序方法**

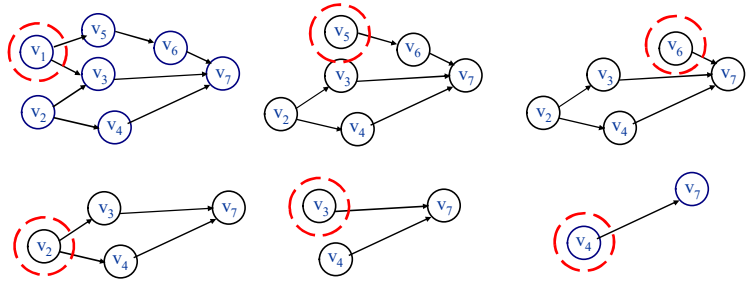
入度为0

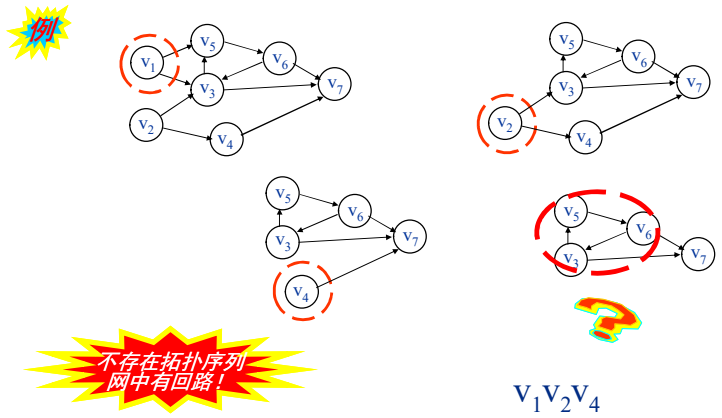
1. 从AOV网中任意选择一个没有前驱的顶点；
2. 从AOV网中去掉该顶点以及以该顶点为出发点的所有边；
3. 重复上述过程,直到AOV网中的所有顶点都被去掉,或者AOV网中还有顶点，但不存在入度为0 的顶点。

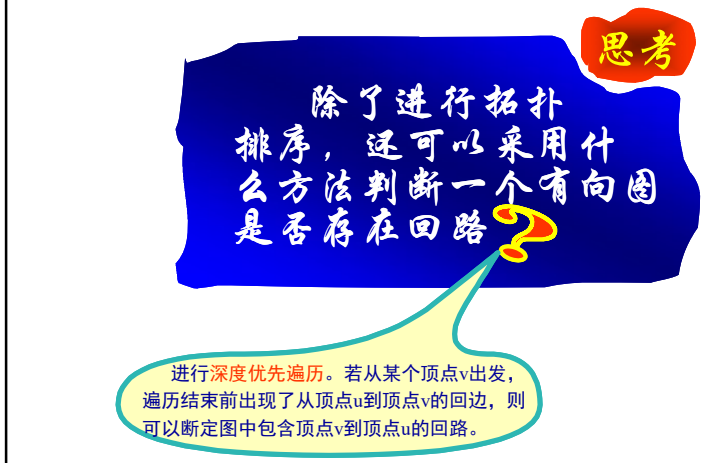
前者说明AOV网中无回路，
后者说明AOV网中存在回路。

例

拓扑序列不一定惟一







6.2 拓扑排序

采用邻接表存储

◆ **拓扑排序方法的用自然语言描述**

1. 首先建立一个入度为0的顶点栈，将网中所有入度为0的顶点分别进栈。
2. 当堆栈不空时，反复执行以下动作：
   - 从顶点栈中退出一个顶点，并输出它；
   - 从AOV网中删去该顶点以及以它发出的所有边，并分别将这些边的终点的入度减1；
   - 若此时边的终点的入度为0，则将该终点进栈；
3. 若输出的顶点个数少于AOV网中的顶点个数，则网中存在回路，否则，说明该网中不存在回路。

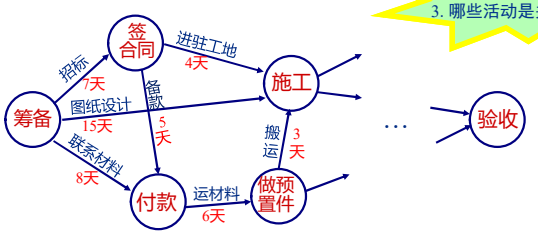
请同学们自学该算法的C语言描述

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*

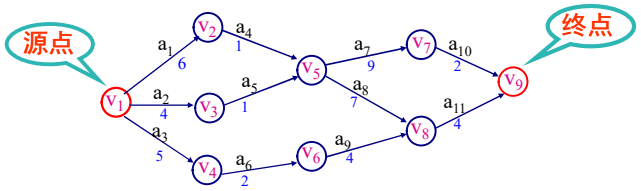
7. AOE网与关键路径*

更关心
1. 每个活动持续多少时间？
2. 完成整个工程至少需要多少时间？
3. 哪些活动是关键活动？



事件　活动

AOE网



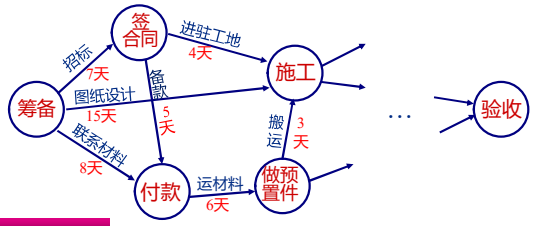
AOE网

网中无回路

正常情况下，AOE网中只有一个入度为0的顶点，称之为**源点**；有一个出度为0 的顶点，称之为**终点**。

7.1 AOE网的定义及特点

◆ AOE(Activity On Edge)网为一个带权的有向无环图，其中，以顶点表示**事件**，有向边表示**活动**，边上的权值表示活动持续的**时间**。



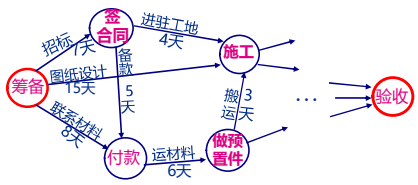
一个AOE网

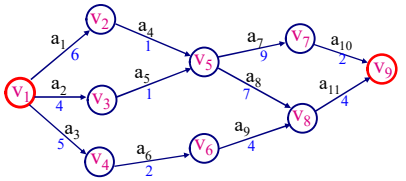
7.1 AOE网的定义及特点

◆ **AOE网的特点**

✓ 只有在某个顶点所代表的事件发生以后,该顶点引发的活动才能开始。

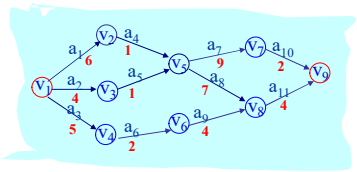
✓ 进入某事件的所有边代表的活动都已完成,该顶点代表的事件才能发生。





7.2 AOE网的存储方法

◆ **采用邻接矩阵存储方法**



7.3 关键路径概念

◆ **关键路径的定义**

- ✓ 从源点到终点的路径中具有最大长度的路径为关键路径；
- ✓ 关键路径上的活动称为关键活动。

◆ **关键路径的特点**

1) 关键路径的长度(路径上的边的权值之和)为完成整个工程所需要的最短时间。

2) 关键路径的长度变化(即任意关键活动的权值变化)将影响整个工程的进度，而其他非关键活动在一定范围内的变化不会影响工期。

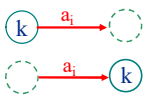
7.4 求关键路径思路

缓冲时间/松弛时间/时间余量

e[i] — 活动a_i的最早开始时间；
l[i] — 活动a_i的最晚开始时间；
若l[i]–e[i]=0，则说明活动a_i为一个关键活动。

ee[k]— 事件k 的最早发生时间

le[k] — 事件k 的最晚发生时间



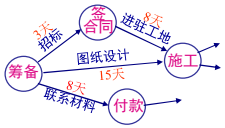
结 论

事件k的最早发生时间ee[k] ⇒ 活动a_i的最早开始时间e[i]

事件k的最晚发生时间le[k] ⇒ 活动a_i的最晚开始时间l[i]

a_i为关键活动 ⇒ 求e[i]=l[i]

计算事件k的最早发生时间ee[k]



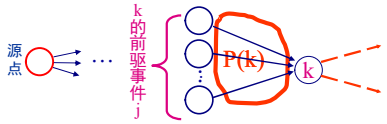
事件k的最早发生时间决定了由事件k出发的所有活动的最早开始时间;该时间是指从源点到顶点(事件)k的最大路径长度。

计算方法

ee[0] = 0

ee[k] = MAX { ee[j] + <j,k>的权}

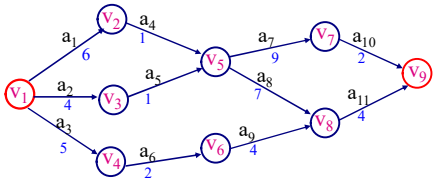
<j,k>∈P(k)

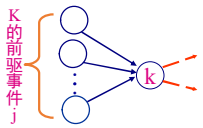


计算事件k的最早发生时间ee[k]

ee[k]：

0	6	4	5	7	7	16	14	18
v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9





计算方法

ee[0] = 0

ee[k] = MAX{ee[j] + <j, k>的权}

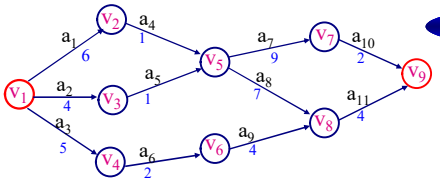
<j,k>∈P(k)

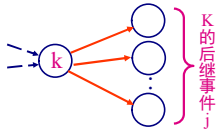
计算事件k的最晚发生时间le[k]

le[k]：

0	6	6	8	7	10	16	14	18
v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

从后向前反推计算





计算方法

le[n-1] = ee[n-1]

le[k] = MIN{ le[j] – <k,j>的权}

<k,j>∈S(k)

计算事件k的最晚发生时间le[k]

事件k的最晚发生时间是指不影响整个工期的前提下事件k必须发生的最晚时间，它必须保证从事件k发出的所有活动的终点事件的最迟发生时间。

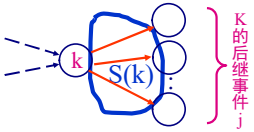
k的后继事件

计算方法　从后向前反推计算

le[n-1] = ee[n-1]

le[k] = MIN{ le[j]– <k,j>的权 }

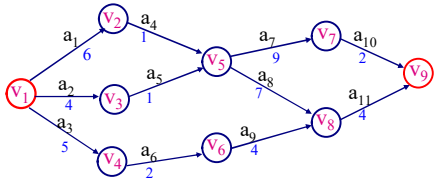
<k,j>∈S(k)



计算活动 i 的最早开始时间e[i]

活动i的最早开始时间实际上是事件k发生的最早时间，即只有事件k发生，活动i才能开始。

计算方法　e[i] = ee[k]



计算活动 i 的最早开始时间e[i]

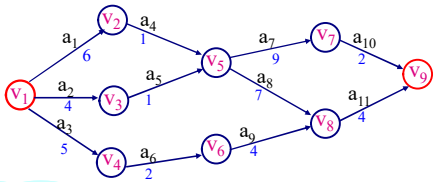
ee[k]:

0	6	4	5	7	7	16	14	18
v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

（事件的最早发生时间）

e[i] :

0	0	0	6	4	5	7	7	7	16	14
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}



k 活动i j

计算方法 e[i] = ee[k]

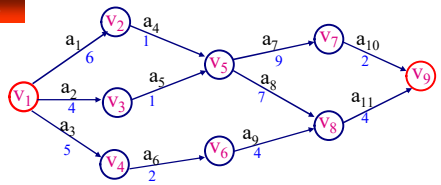
133

计算活动 i 的最晚开始时间l[i]

活动i的最晚开始时间是指不推迟整个工期的前提下活动i开始的最晚时间。

k 活动 i j

计算方法 l[i] = le[j]-<k,j>的权



134

计算活动 i 的最晚开始时间l[i]

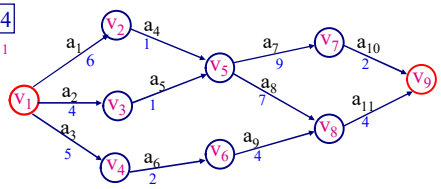
le[j] :

0	6	6	8	7	10	16	14	18
v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9

（事件的最晚发生时间）

l[i] :

0	2	3	6	6	8	7	7	10	16	14
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}



k 活动i j

计算方法 l[i] = le[j]-<k,j>的权

135

求出关键活动与关键路径

计算方法：l[i] = e[i]

a_i是关键活动

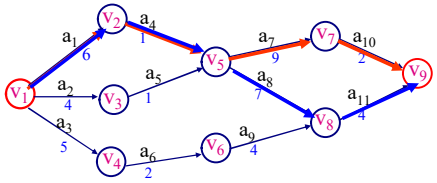
e[i] :

0	0	0	6	4	5	7	7	7	16	14
0	1	2	3	4	5	6	7	8	9	10

l[i] :

0	2	3	6	6	8	7	7	10	16	14
0	1	2	3	4	5	6	7	8	9	10

关键活动 a_1 a_4 a_7 a_8 a_{10} a_{11}



关键路径的长度为18个时间单位。

136

延伸阅读*：

上面我们介绍了*关键活动*和*关键路径的计算算法*的基本原理，请同学自学算法的C实现。

内容提要

8. 网络流量问题*

◆ **相关概念**

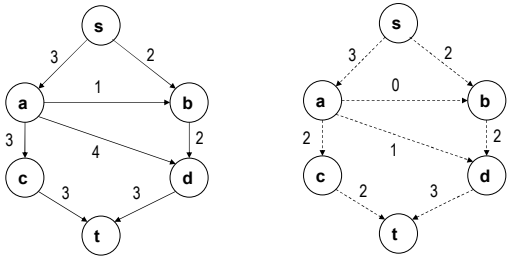
设给定边容量为$c_{v,w}$的有向图G=(V,E)（容量可以表示通过一个管道的水、电、交通、网络等流量）。有两个顶点，一个是s称为**源点**(source)，一个是t称为**汇点**(sink)。对于任一条边(v,w)，最多有“流”的$c_{v,w}$个单位（容量）可以通过。**在既不是源点s又不是汇点t的任一顶点v，总的进入流必须等于总的发出的流。**每条边上的流满足下面两个条件：

- 通过边的流不能大于边的容量**（容量约束）**
- 到达顶点v的流的总和与从v流出的总合相同，其中v不是源点或汇点。**（流守恒）**

最大流问题：确定从s到t可以通过的最大流量。

8. 网络流量问题*

对于下（左）图，其最大流是5，如右图所示。

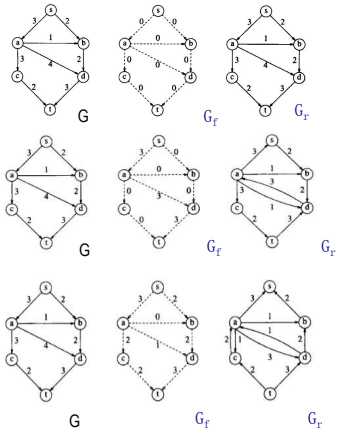


8. 网络流量问题*

◆ 最大流算法原理

算法设有3个图（原图G、流图G_f、残余图G_r），在其上分阶段进行。G_f表示在算法的任意阶段已经达到的流，算法终止时其包含最大流；G_r称为残余图（residual graph），它表示每条边还能再添加上多少流（即还残余多少流），对于G_r中每条边（称为残余边，residual edge）可以从其容量中减去当前流来计算其残余流。

1. 初始时G_f所有边都没有流（流为0），G_r与G相同；
2. 每个阶段，先从G_r中找一条从s到t的路径（称为增长路径augmenting path）；
   a. 将该路径上最小边的流量作为整个路径的流（权），并将路径加至流图G_f中；
   b. 将该权值路径从G_r中减去，若某条边权值为0，则从G_r中除去；
   c. 将具有该权的反向路径加到G_r中；
   d. 重新执行步骤2，直到G_r中无从s到t的路径；
3. 将G_f中顶点t的每条入边流值相加得到最大流。



◆ 算法原理示例

原图G、流图G_f和残余图G_r初始状态

阶段一：
1. 在G_r中找到一条路径s-a-d-t
2. 将该路径中最小边流3作为整个路径流，并将该路径及其流加到G_f中
3. 在G_r中减去该路径及其流，若某边流为0，则从图中删除
4. 将该路径反向加到G_r中

阶段二：
1. 在G_r中找到一条路径s-b-d-a-c-t
2. 将该路径中最小边流2作为整个路径流，并将该路径及其流加到G_f中
3. 在G_r中减去该路径及其流，若某边流为0，则从图中删除
4. 将该路径反向加到G_r中

由于G_r中已没有从s至t的路径，算法结束。此时G_f中t的入流之和即为**最大流**。

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*

143

小结：图

◆ **图的基本概念**
- ✓ 顶点和边
- ✓ 有向图、无向图、网
- ✓ 度、路径、路径长度
- ✓ 子图、图的连通、生成树

◆ **图的存储方法**
- ✓ 邻接矩阵
- ✓ 邻接表

◆ **图的遍历**
- ✓ 深度优先遍历（DFS）
- ✓ 广度优先遍历（BFS）

◆ **最小生成树**
- ✓ 生成树、最小生成树
- ✓ 普里姆(Prim)算法
- ✓ 克鲁斯卡尔(Kruskal)算法

◆ **最短路径问题**
- ✓ 路径、路径长度、单源最短路径
- ✓ Dijkstra算法

◆ **图的应用**
- ✓ AOV网与拓扑排序*
- ✓ AOE网与关键路径*
- ✓ 网络流量问题*

144