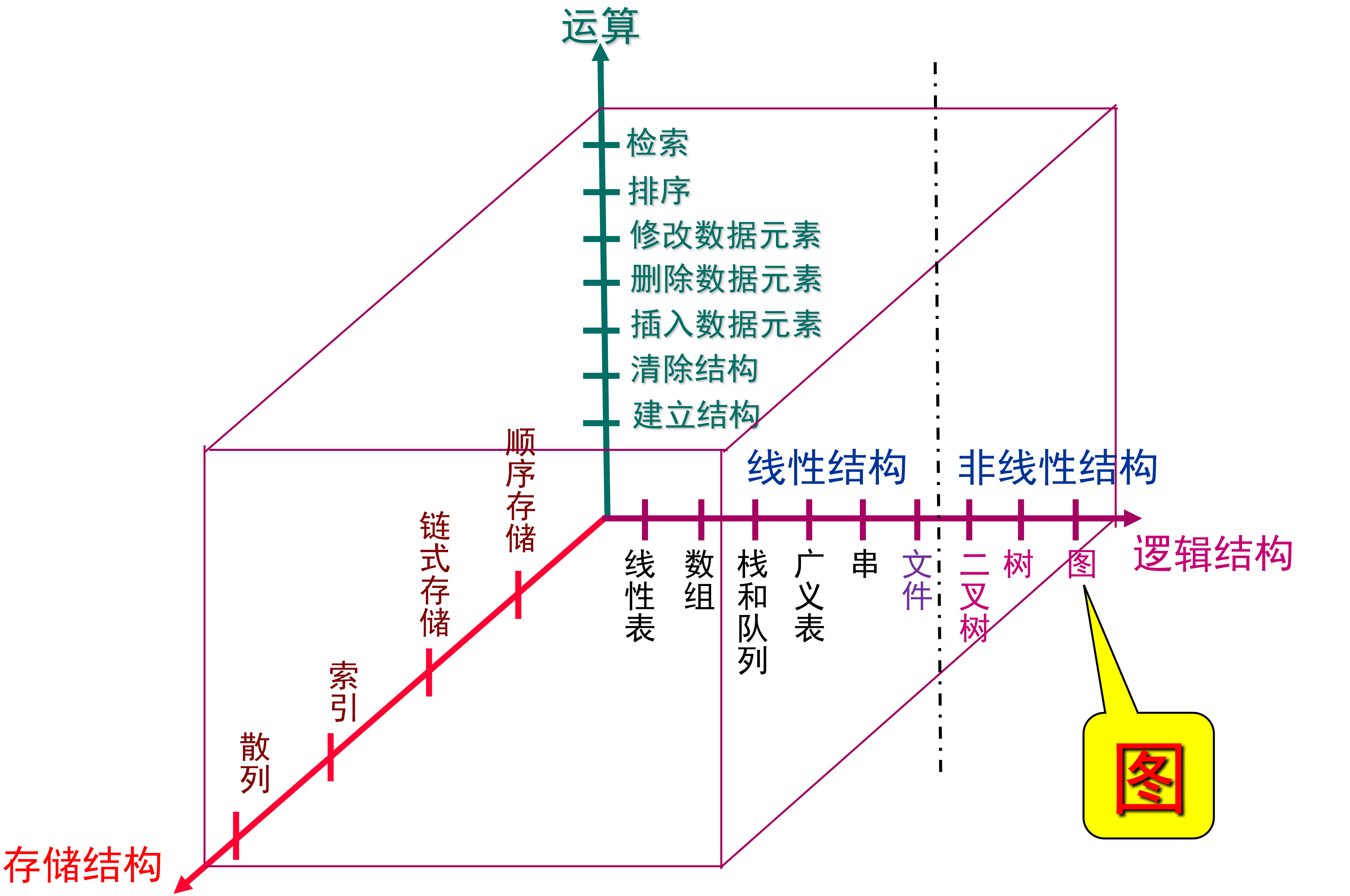
# 数据结构与程序设计
# (Data Structure and Programming)

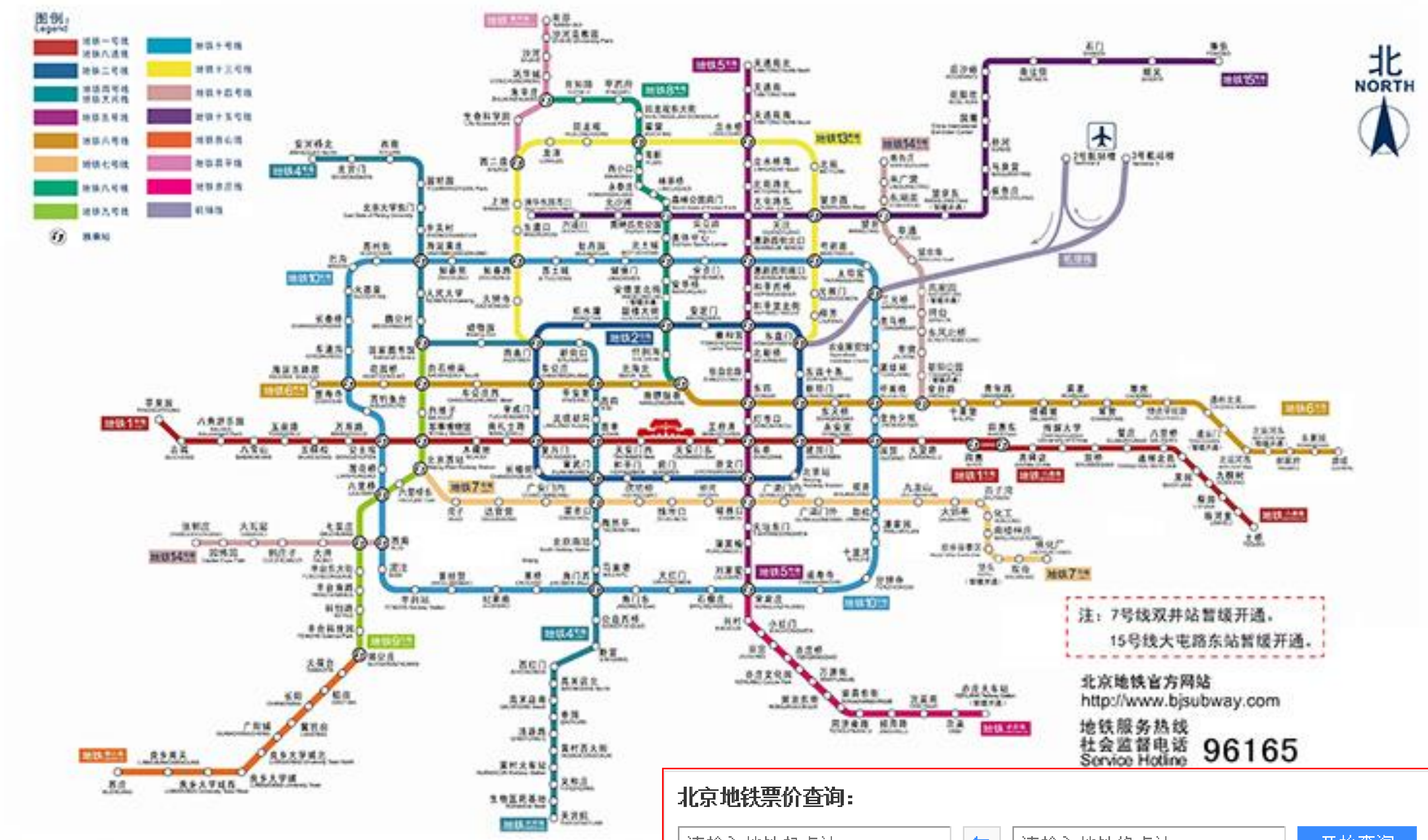
## 图
## (Graph)

北航计算机学院 林学练

# 数据结构的基本问题空间



# 北京地铁线路图
Beijing Subway Map

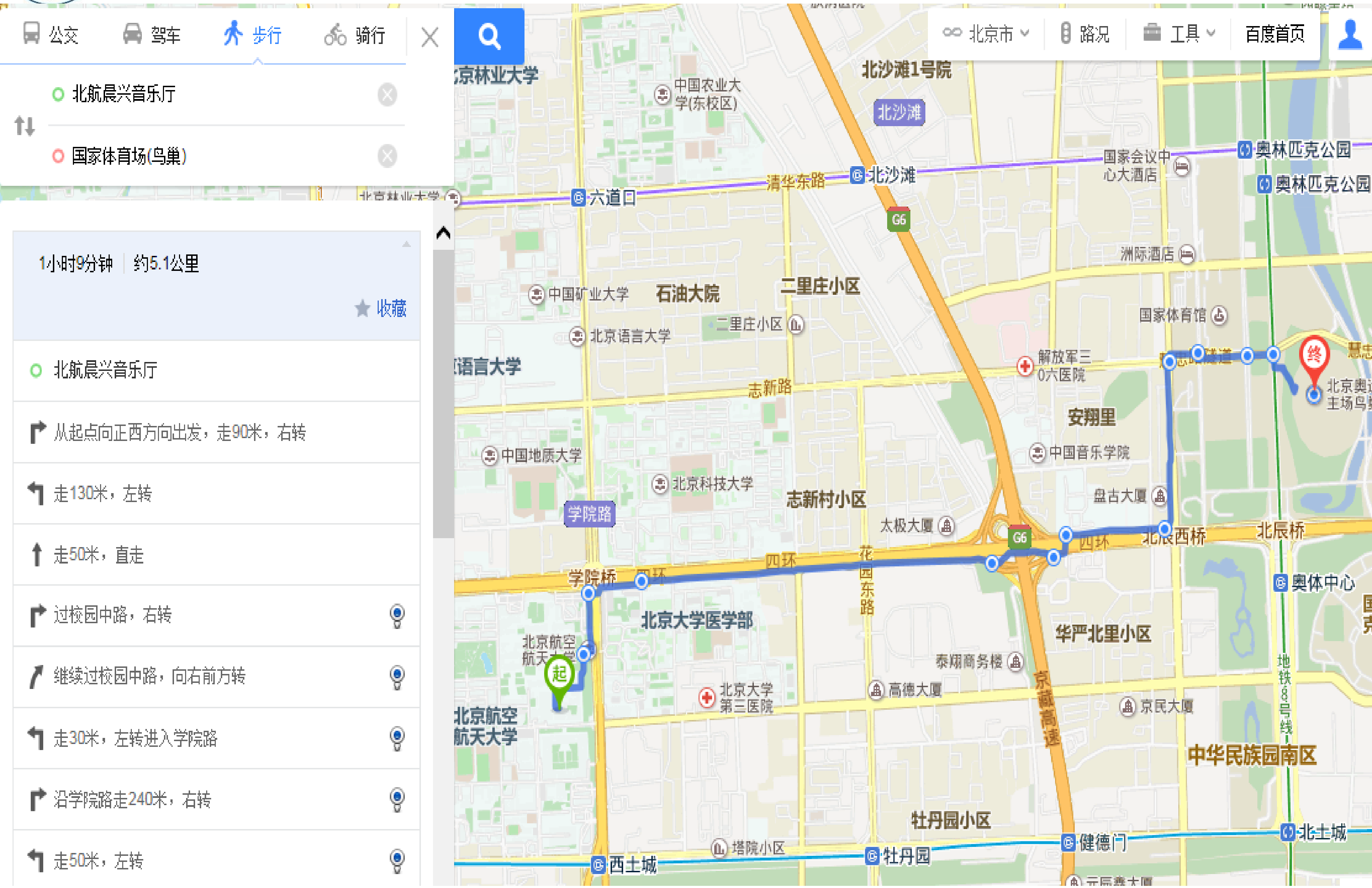



**北京地铁票价查询：**

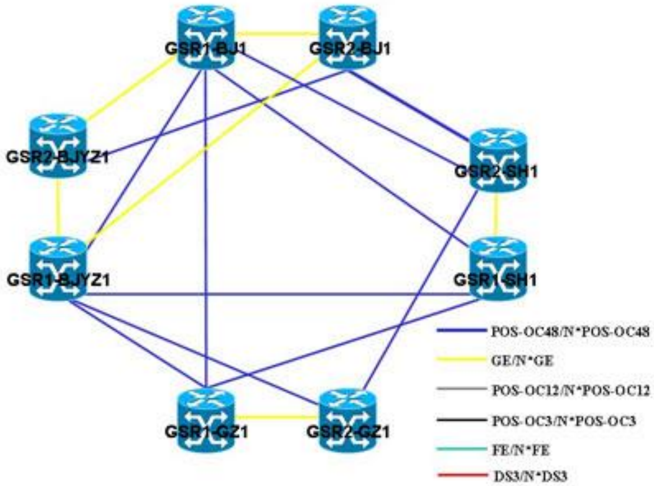
请输入地铁起点站 | 请输入地铁终点站 | 开始查询

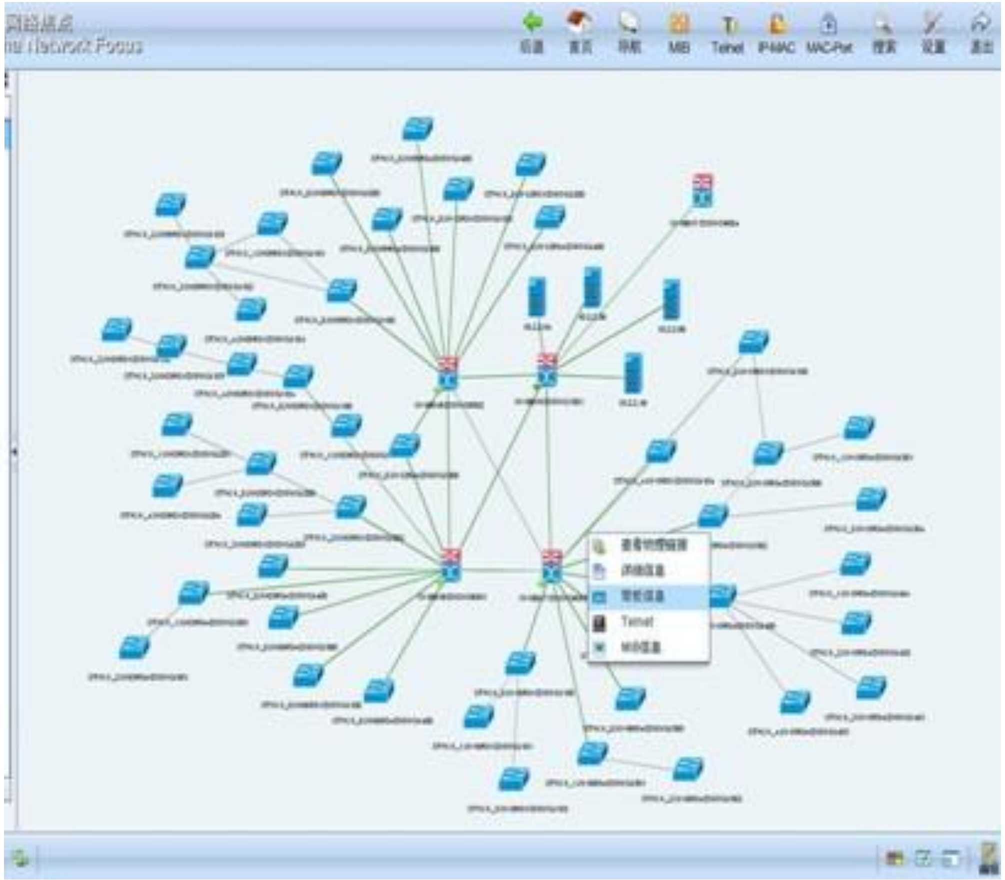
详情：起步6公里内每人次3元，6-12公里每人次4元，12-32公里每10公里加1元，32公里以上每20公里加1元，票价不封顶。（北京市发改委）

# 地图导航


公交
驾车
步行
骑行
北航晨兴音乐厅
国家体育场(鸟巢)
1小时9分钟 约5.1公里
收藏
北航晨兴音乐厅
从起点向正西方向出发，走90米，右转
走130米，左转
走50米，直走
过校园中路，右转
继续过校园中路，向右前方转
走30米，左转进入学院路
沿学院路走240米，右转
走50米，左转
北京市
路况
工具
百度首页
北沙滩1号院
北沙滩
中国农业大学(东校区)
国家会议中心大酒店
奥林匹克公园
清华东路
北沙滩
六道口
洲际酒店
中国矿业大学
石油大院
二里庄小区
北京语言大学
二里庄小区
国家体育馆
解放军三0六医院
志新路
安翔里
中国地质大学
中国音乐学院
北京科技大学
志新村小区
盘古大厦
学院路
太极大厦
北辰西桥
北辰桥
四环
学院桥
奥体中心
北京大学医学部
花园东路
华严北里小区
泰翔商务楼
高德大厦
北京大学第三医院
京民大厦
北京航空航天大学
京藏高速
中华民族园南区
地铁8号线
牡丹园小区
塔院小区
健德门
北土城
西土城
牡丹园

CNCNET骨干网超级核心节点之间互连结构图

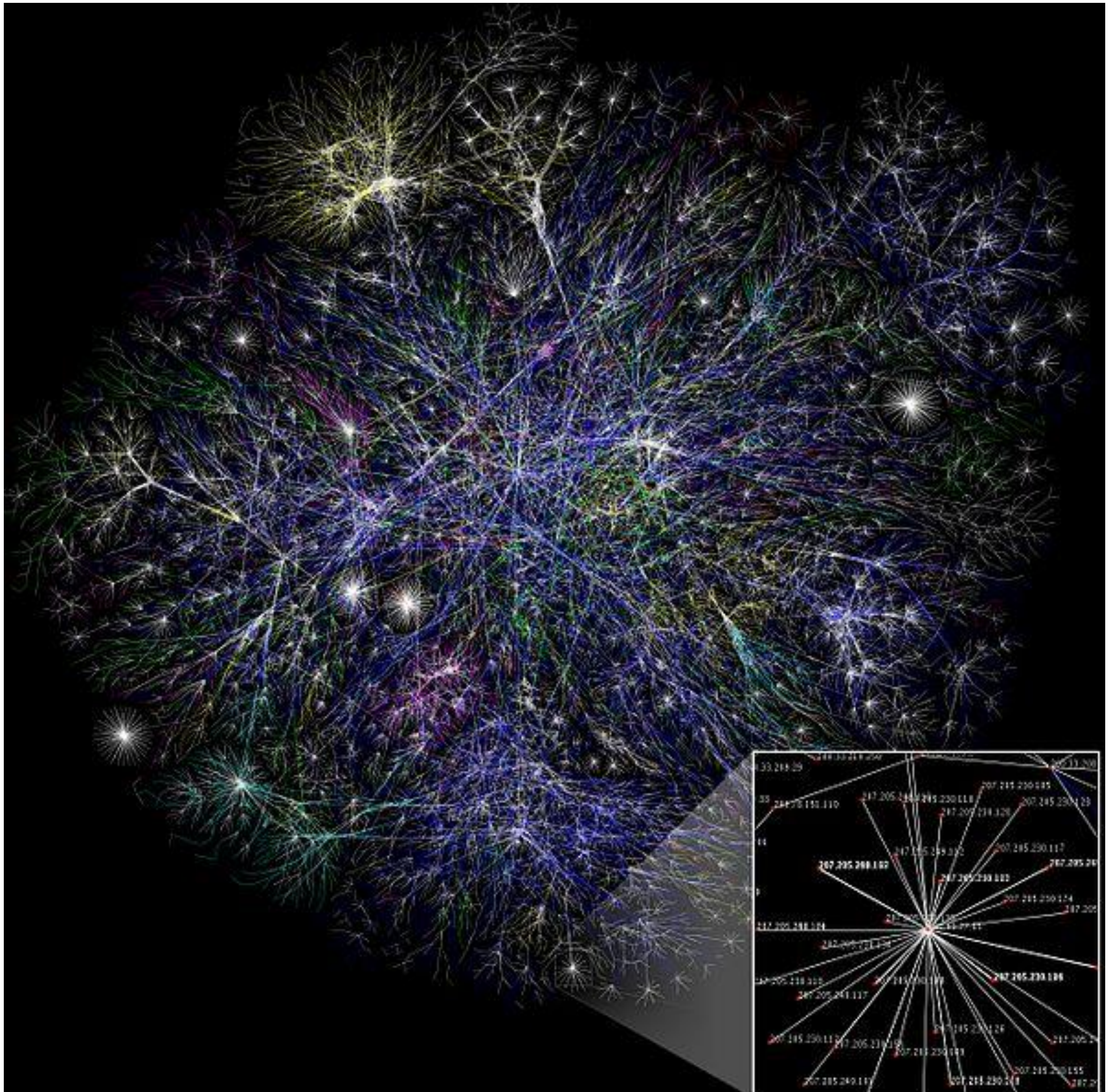


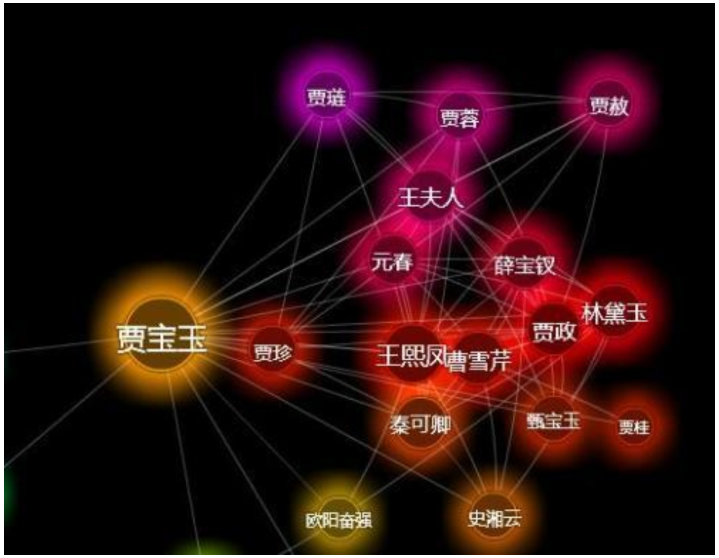
**典型问题：**

- ✓ 路由/导航
- ✓ 网络铺设
- ✓ 流量分配

# Internet Map



# Social network, reference network, infection network, …





**典型问题：**

- ✓ 学术关系
- ✓ 合作关系
- ✓ 影响力
- ✓ 社团分析
- ✓ 传播模式
- ✓ 演化模式
- ✓ ……

# 本章主要内容

6.1 图的基本概念

6.2 图的存储方法

重点

**6.3 图的遍历**

**6.4 最短路径问题**

**6.5 最小生成树**

6.6* AOV网与拓扑排序

6.7* AOE网与关键路径

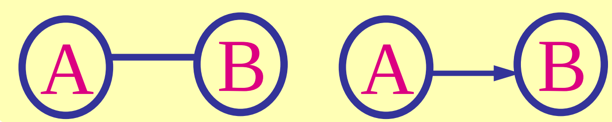
6.8* 网络流量问题

一个数据元素--

**顶点**(Vertex)


数据元素之间的关系--

**边(弧, Edge)**



# 6.1 图(Graph)的基本概念

## 一、图的定义

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,

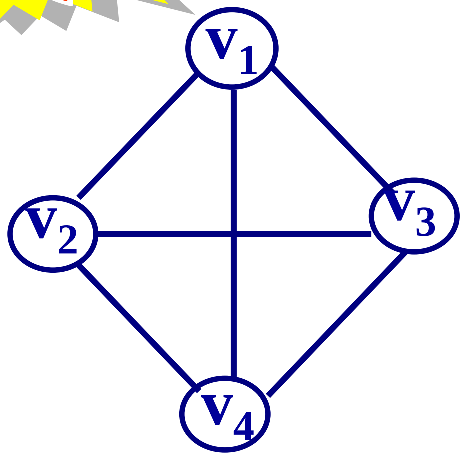
$V=\{v_1, v_2, \ldots, v_n\}$, 为顶点集合,

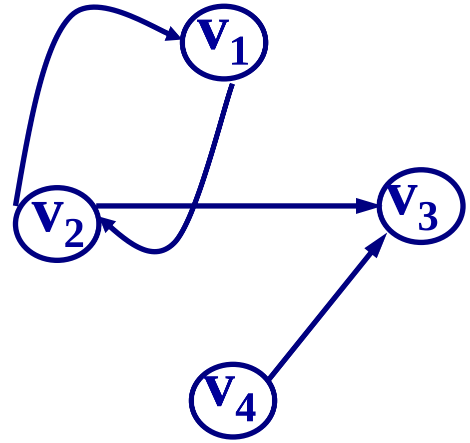
$E=\{(v_i, v_j)\}$或$E=\{<v_i, v_j>\}$, $i, j \in [1,n]$,

为关系(边或弧)的集合。

顶点偶对

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

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

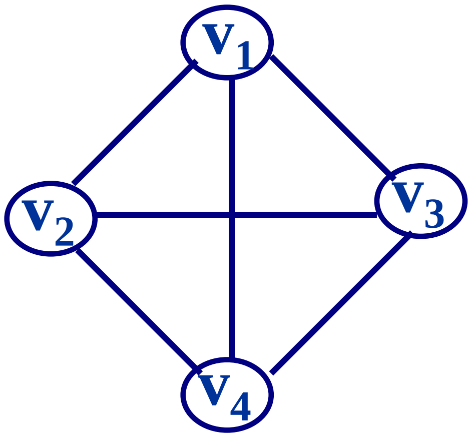
$$E_2 = \{ \langle v_1, v_2\rangle,\ \langle v_2, v_1\rangle,\ \langle v_2, v_3\rangle,\ \langle v_4, v_3\rangle \}$$

# 二、图的分类

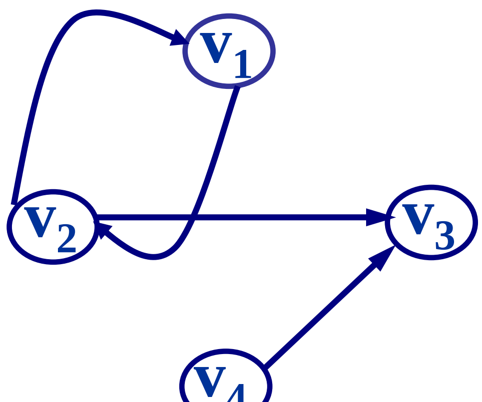
**无向图**：对于$(v_i, v_j) \in E$，必有$(v_j, v_i) \in E$，并且偶对中顶点的前后顺序无关。

**有向图**：若$\langle v_i, v_j \rangle \in E$是顶点的有序偶对。

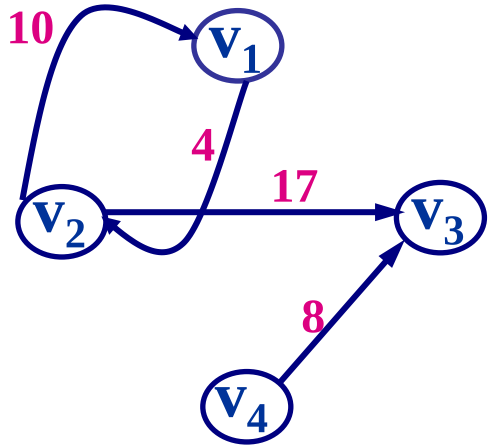
**网(络)**：与边有关的数据称为**权(weight)**，边上带权的图称为 **网络(network)** 。



无向图



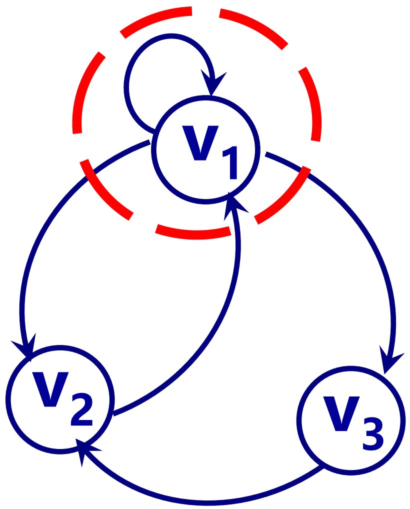
有向图



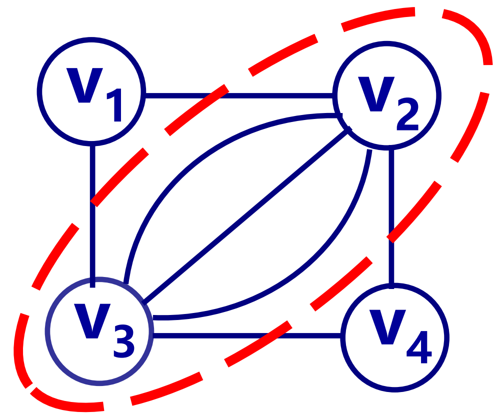
网(络)

# 本章一般不讨论的图

1. 带自身环的图



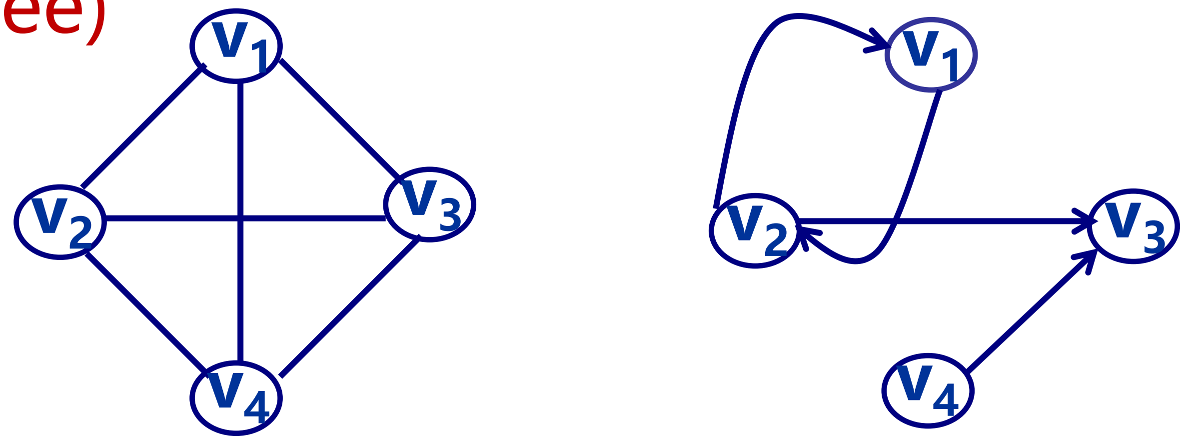

2. 多重图




本章主要讨论**简单图**

# 三、名词术语

1. 顶点的度（Degree）：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$ID(v_i)$。

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

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。

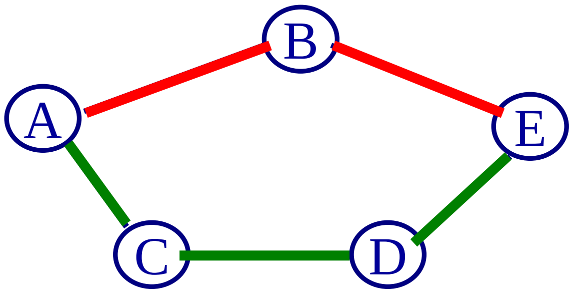
结论3 具有n个顶点的有向图最多有n(n-1)条边。

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*。

## 2. 路径和路径长度

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

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 $v_x, v_{i1}, v_{i2}, \dots, v_{im}, v_y$，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。




P(A, E): A, B, E (第1条路径)

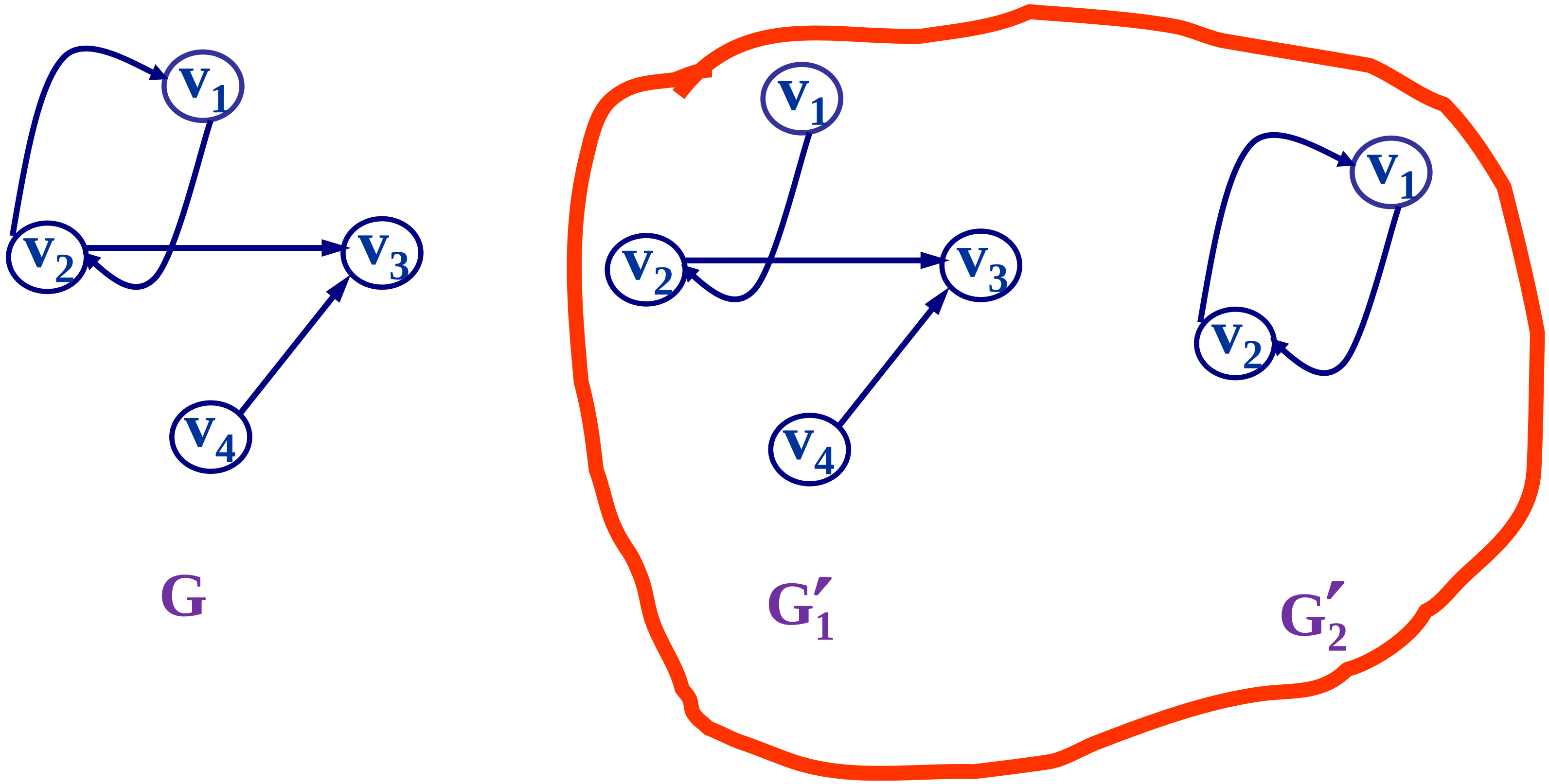
A, C, D, E (第2条路径)

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。

# 3. 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。

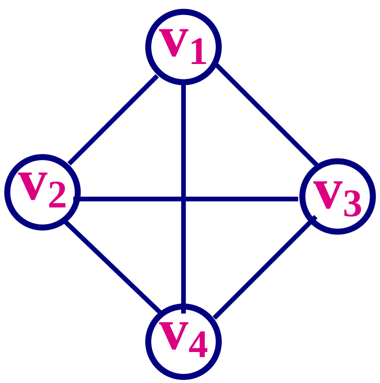


# 4. 图的连通(Connected)

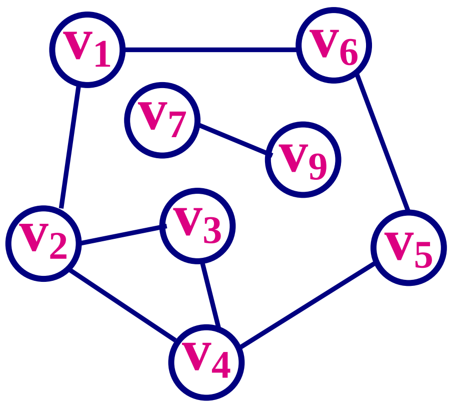
## (1) 无向图的连通

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。

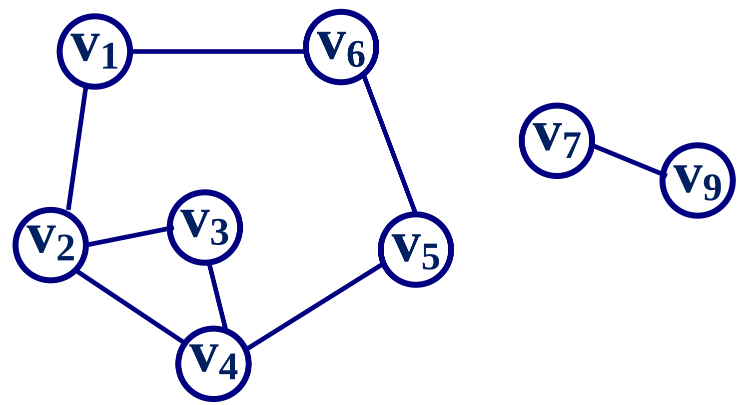
**连通分量**——无向图中的极大连通子图。




连通图




非连通图



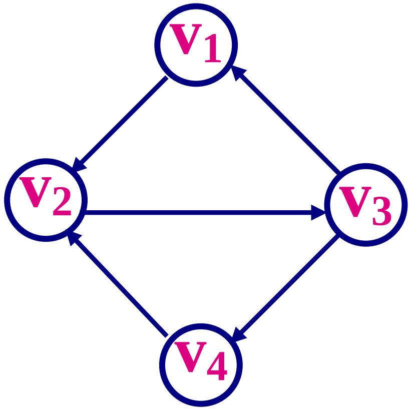

2个连通分量

# (2) 有向图(Directed Graph)的连通

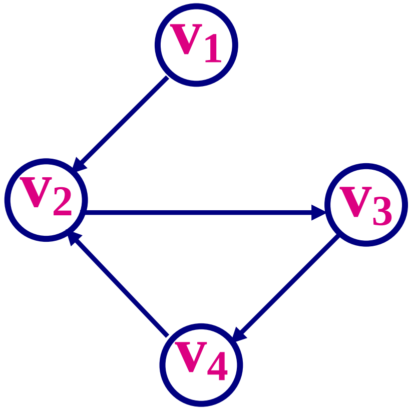
若有向图中顶点$v_i$到$v_j$有路径，并且顶点$v_j$到$v_i$也有路径，则称顶点$v_i$与$v_j$是连通的。

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。

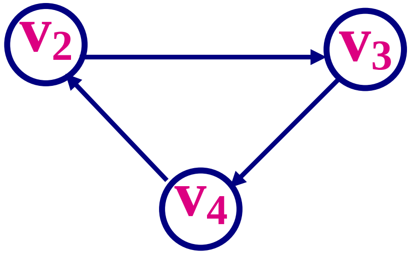
**强连通分量** —— 有向图中的**极大**强连通子图。




(强)连通图



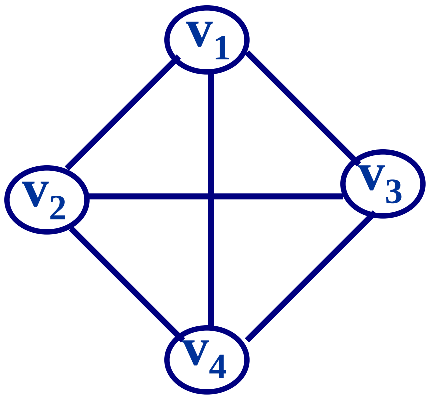

非连通图




强连通分量

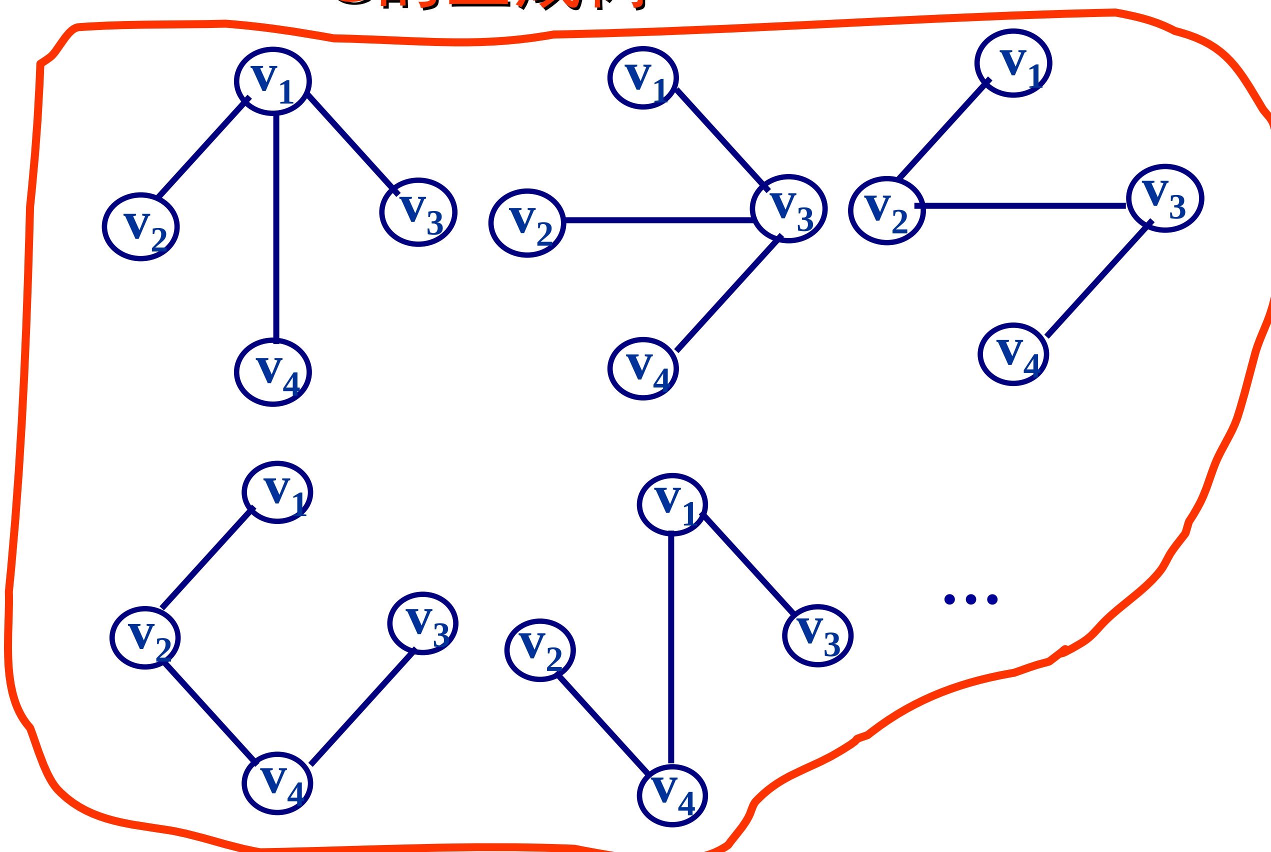
# 5. 生成树（Spanning Tree）

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。




G

G的生成树



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。

**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个顶点的图中只要少于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

# 6.2 图的存储方法

**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"

# 一、邻接矩阵存储方法

- 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0, 1, 2, 3,…,此数组可略）。

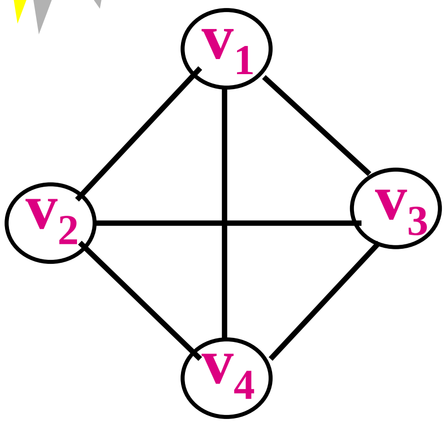
2. 定义一个二维数组**A[0..n-1,0..n-1]**存放图中所有顶点之间关系的信息（该数组被称为**邻接矩阵**）,有

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

对于带权的图，有

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

# 例

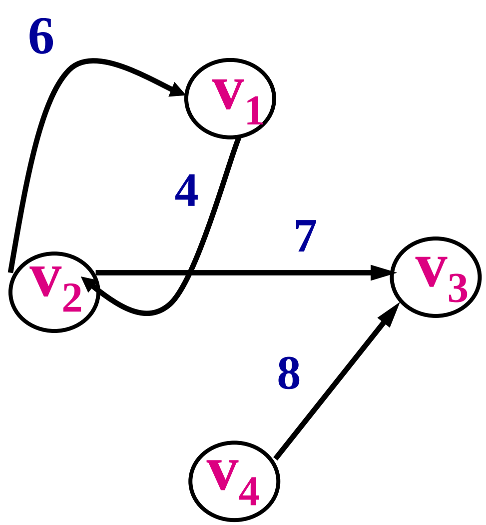



**VERTEX1[0..3]**

| | |
|---|---|
| 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 | $v_4$ |

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

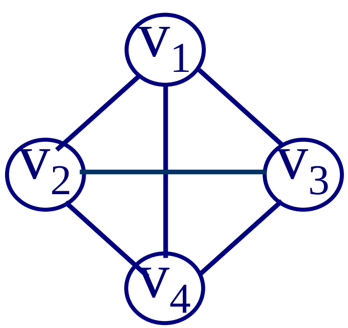
# 特点

(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**。

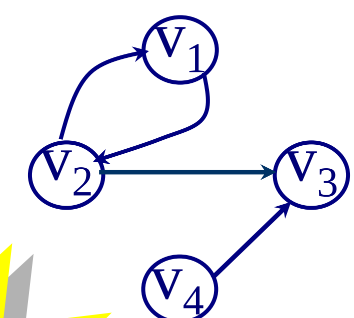



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

| | | | |
|---|---|---|---|
| $v_1$ | $v_2$ | $v_3$ | $v_4$ |

(无向图)

空间复杂度 $O(n^2)$




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

| | | | |
|---|---|---|---|
| $v_1$ | $v_2$ | $v_3$ | $v_4$ |

(有向图)

# 邻接矩阵实现

```
#define MaxV  <最大顶点个数>

Vertype  Vertex[MaxV];

Edge  G[MaxV][MaxV];
```

顶点信息数组

邻接矩阵

# 三元组(稀疏图)

```
#define MaxV   <最大顶点个数>
#define MaxE   <最大边数>

Vertype  Vertex[MaxV];

struct  edge{
        int v1, v2;
        int  weight;
} E[MaxE];
```

顶点信息数组

定义边类型

边集数组

# 二、邻接表存储方法

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结点的地址。

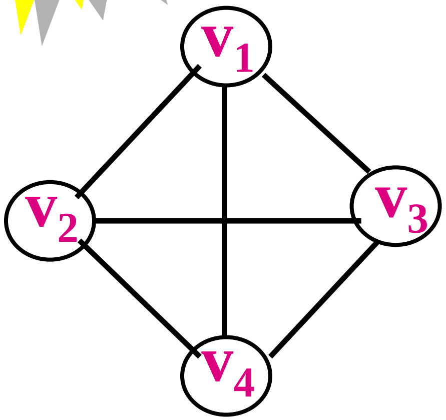
n个头结点构成一数组结构。

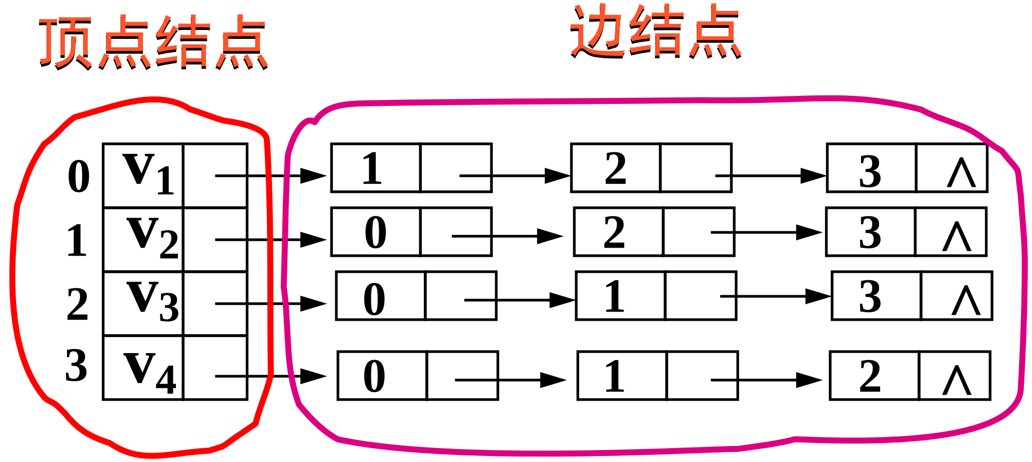
2. 第i个链表中的每一个链结点(称之为 **边结点** )表示以第i 个顶点为 **出发点** 的一条边;边结点的构造为

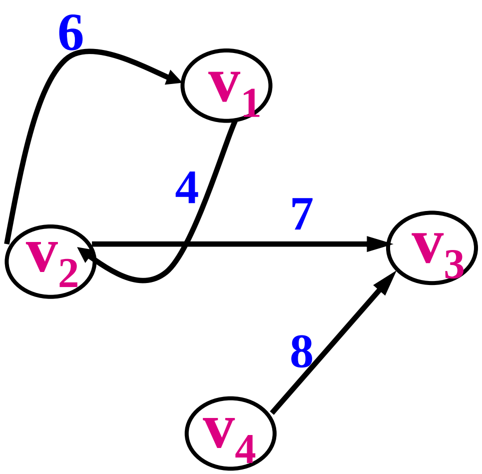
| adjvex | weight | next |
|---|---|---|

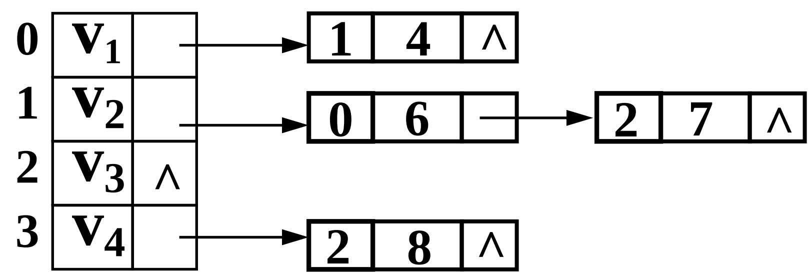
其中,next 域为指针域;
weight 域为权值域(若图不带权,则无此域);
adjvex 域存放以第i个顶点为出发点的一条边,
保存另一端点在头结点数组中的位置。

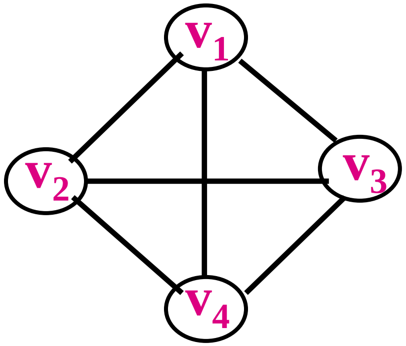
例

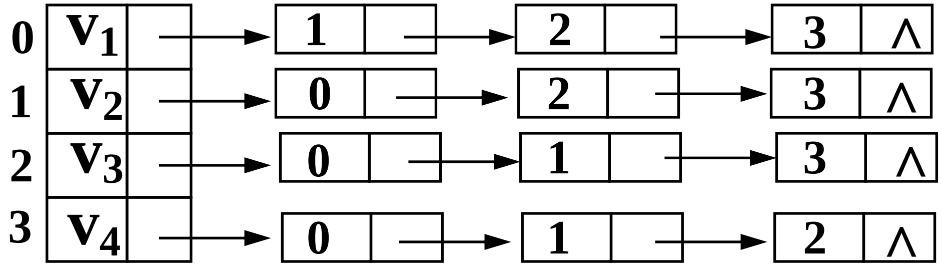

v1
v2
v3
v4

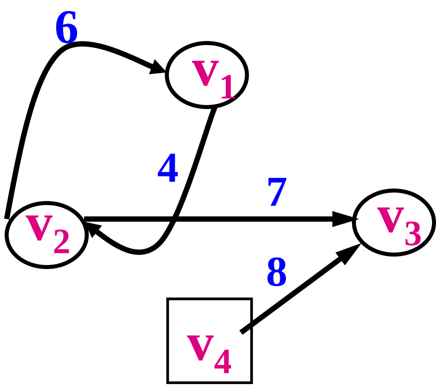

顶点结点
边结点
0 v1 1 2 3 ∧
1 v2 0 2 3 ∧
2 v3 0 1 3 ∧
3 v4 0 1 2 ∧

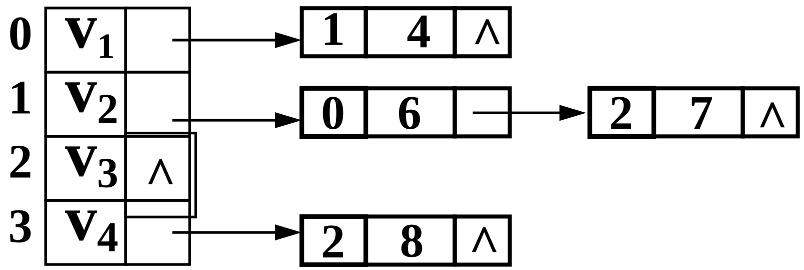

6
4
7
8
v1
v2
v3
v4


0 v1 1 4 ∧
1 v2 0 6 2 7 ∧
2 v3 ∧
3 v4 2 8 ∧



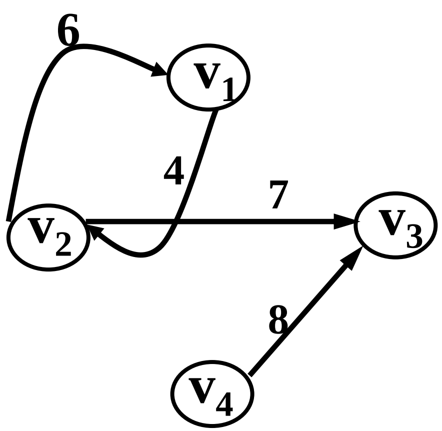

**特点**

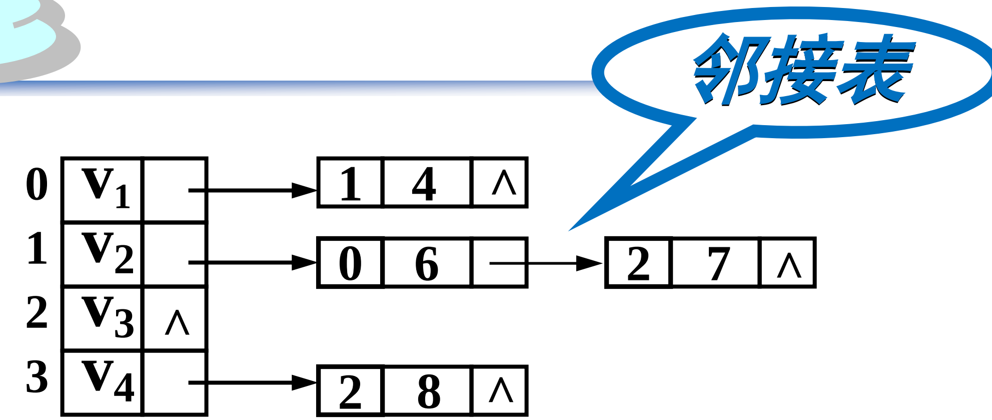
(1)无向图的第i个链表中边结点个数是第i个顶点**度数**。

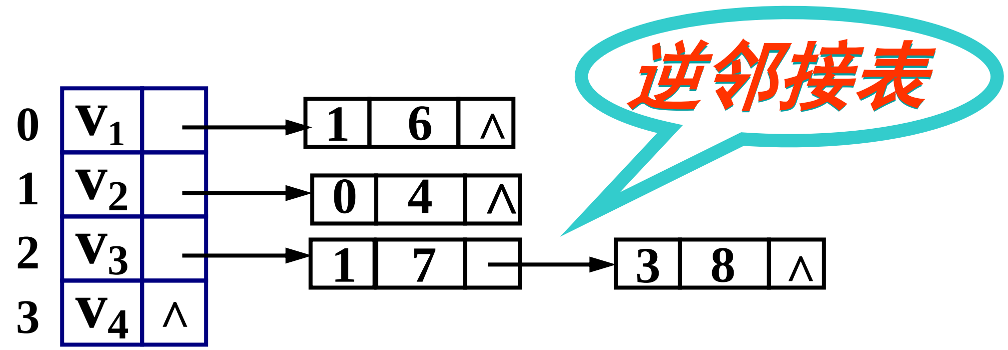
(2)有向图的第i个链表中边结点个数是第i个顶点的**出度**。

(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。

# 关于逆邻接表



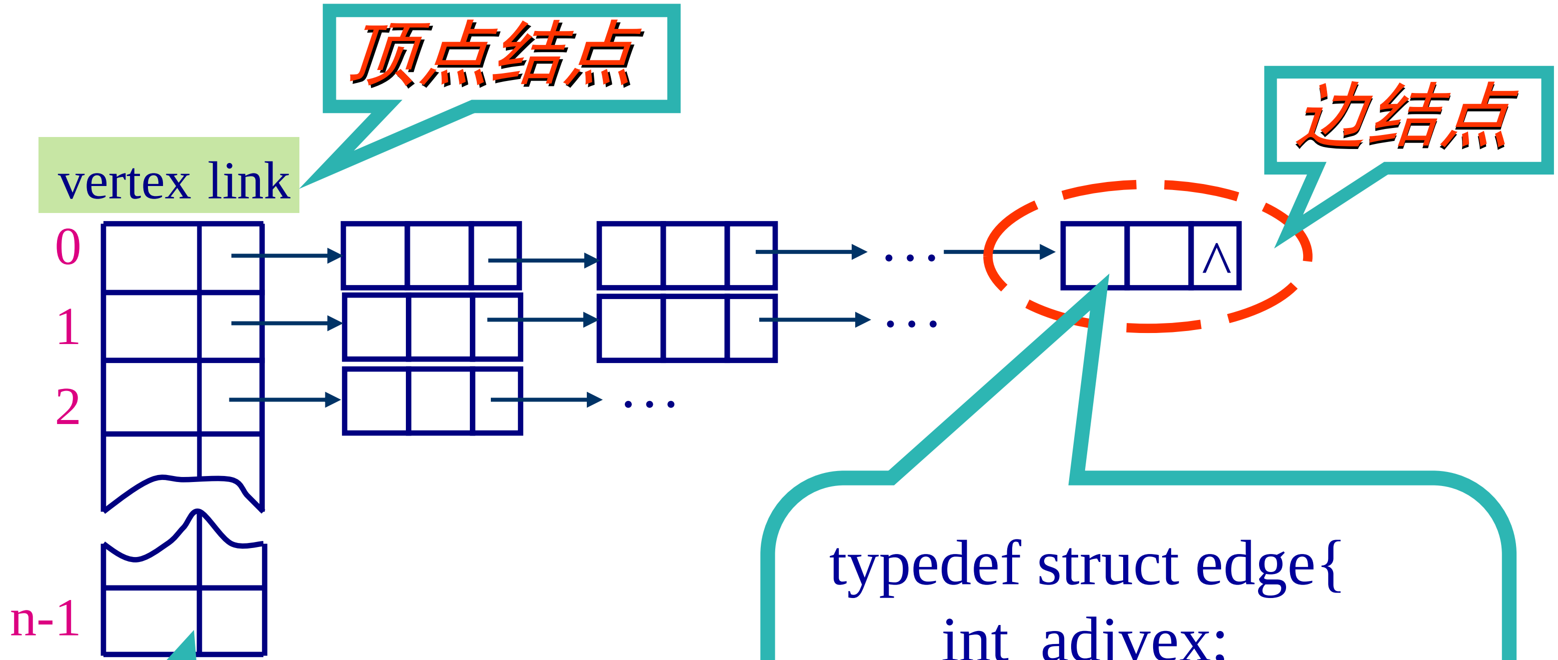




**对于邻接表**

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




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

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

# 邻接表

```
#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

# 延伸阅读*

三. 有向图的十字链表存储方法

（略）

四. 无向图的多重邻接表存储方法

（略）

# 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 |

若有如下输入：

8

0 2 4 … −1

1 3 6 8 … −1

…

第1行为无向图顶点个数，从第2行开始：每行第1个数为顶点序号，第2个数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：

```
//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;
}
```

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

```
void createGraph(VLink graph[]){
   int i,n,v1,v2;
   scanf("%d",&n);  //读入顶点个数
   for(i=0; i<n; i++){  //读入一行数据
      scanf("%d %d",&v1, &v2);//读顶点和第一条边
      while(v2 != -1){
         graph[v1].link=insertEdge(graph[v1].link, v2);
         graph[v2].link=insertEdge(graph[v2].link, v1);
         scanf("%d",&v2); //读一条边
      }
   }
}
```

邻接矩阵：

graph[v1][v2]= graph[v2][v1]= 1;

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

# 6.3 图的遍历

以无向图为例

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且仅访问一次，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** 。

基于遍历的方法可以求解多数问题：

利用图的遍历

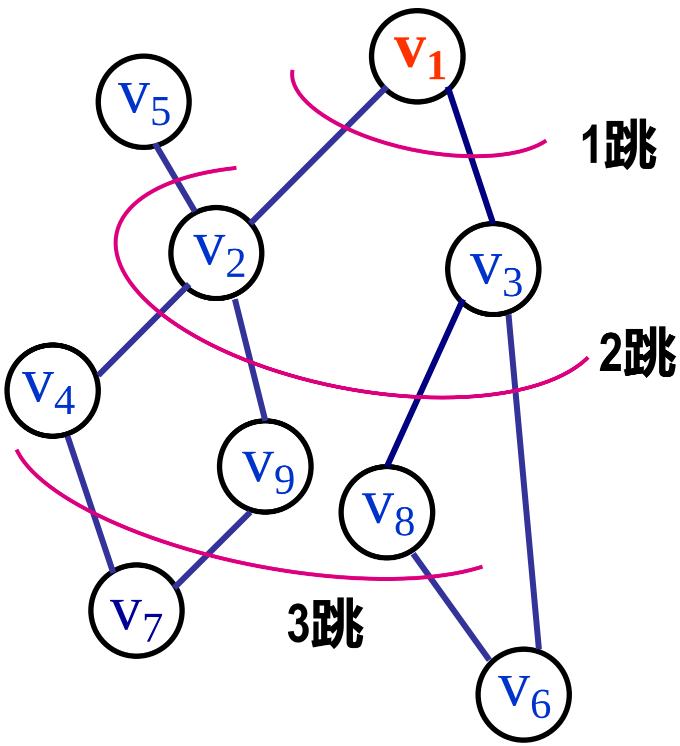
- 确定图中满足条件的顶点；
- 求解图的连通性问题，如求分量；
- 判断图中是否存在回路；
- 找出点S和D之间的路径

…

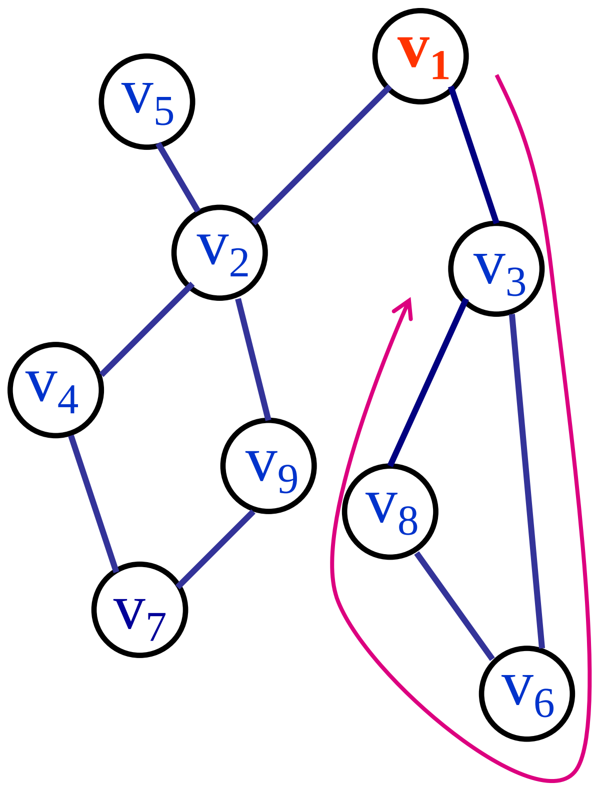
注：通用的方法往往不是高效的

# 两种遍历方式

## 一、广度优先遍历
(Breadth First Search,BFS)




## 二、深度优先遍历
(Depth First Search,DFS)

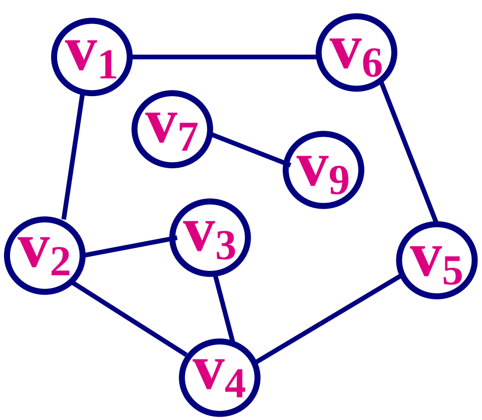


# 一、广度优先遍历(Breadth First Search,BFS)

类似于树/二叉树的层次遍历

【原则】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， 直到图中与v 相通的所有顶点都被访问,

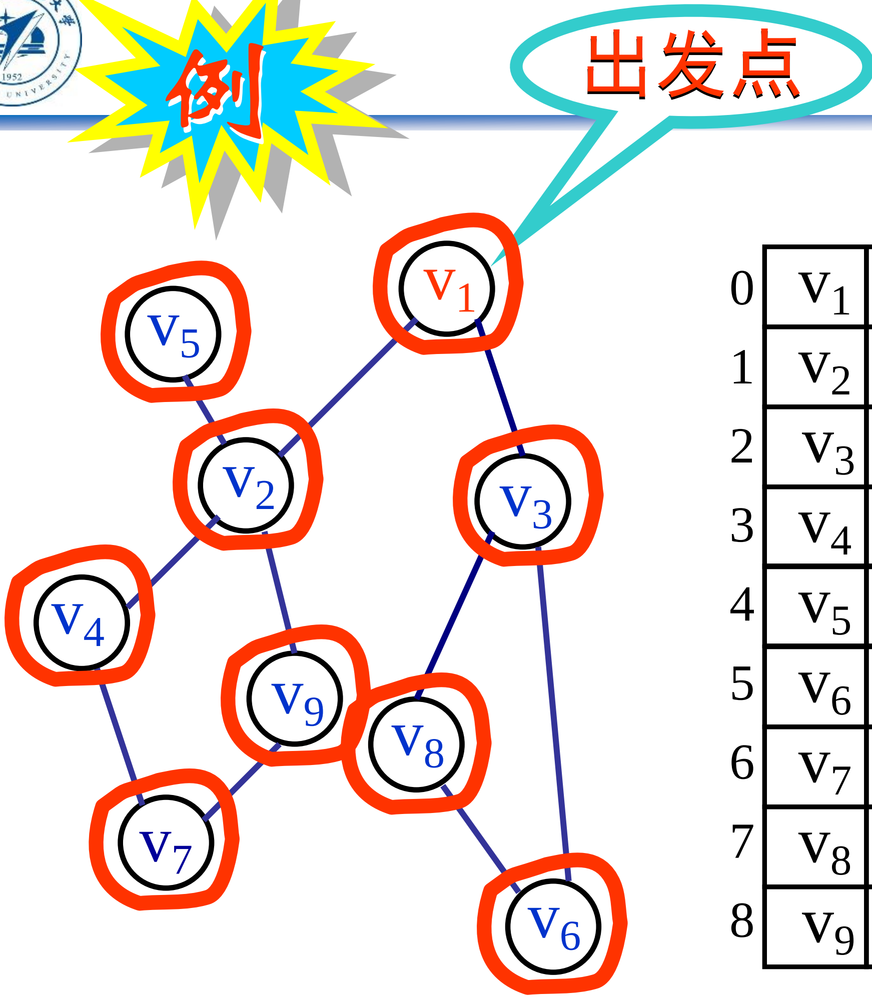
若此时图中还有未被访问过的顶点, 则从另一个未被访问过的顶点出发重复上述过程,直到遍历全图。




*完成所有连通分量的遍历*

*完成一个连通分量的遍历*

例




| | | | | | | |
|---|---|---|---|---|---|---|
| 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 ∧ | | |

队列

$V_1$ $V_2$ $V_3$ $V_4$ $V_5$ $V_9$ $V_6$ $V_8$ $V_7$

遍历序列

# 如何确保顶点不被重复访问？

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

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

回顾：树的层次遍历有与visited等价的数组吗？

```
int  Visited[N]={0}; //标识顶点是否被访问守，N为顶点数
void  travelBFS(VLink  G[ ], int n){
    int i;
    //初始化
    for(i=0; i<n; i++)
        Visited[i] = 0 ;
    //遍历每个连通分量
    for(i=0; i<n; i++)
        if( !Visited[i] )
            BFS(G, i);
}
```

```
void BFS(VLink  G[ ], int v){ //遍历一个连通分量
    ELink  *p;
    VISIT(G, v); //自定义函数，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;
                enQueue(G, p->adjvex);
            }
    }
}
```

保证每个顶点
只被访问一次

## 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)，因而算法的时间复杂度为$O(n^2)$。

# 二、深度优先遍历(Depth First Search,DFS)

类似于树/二叉树的前序遍历

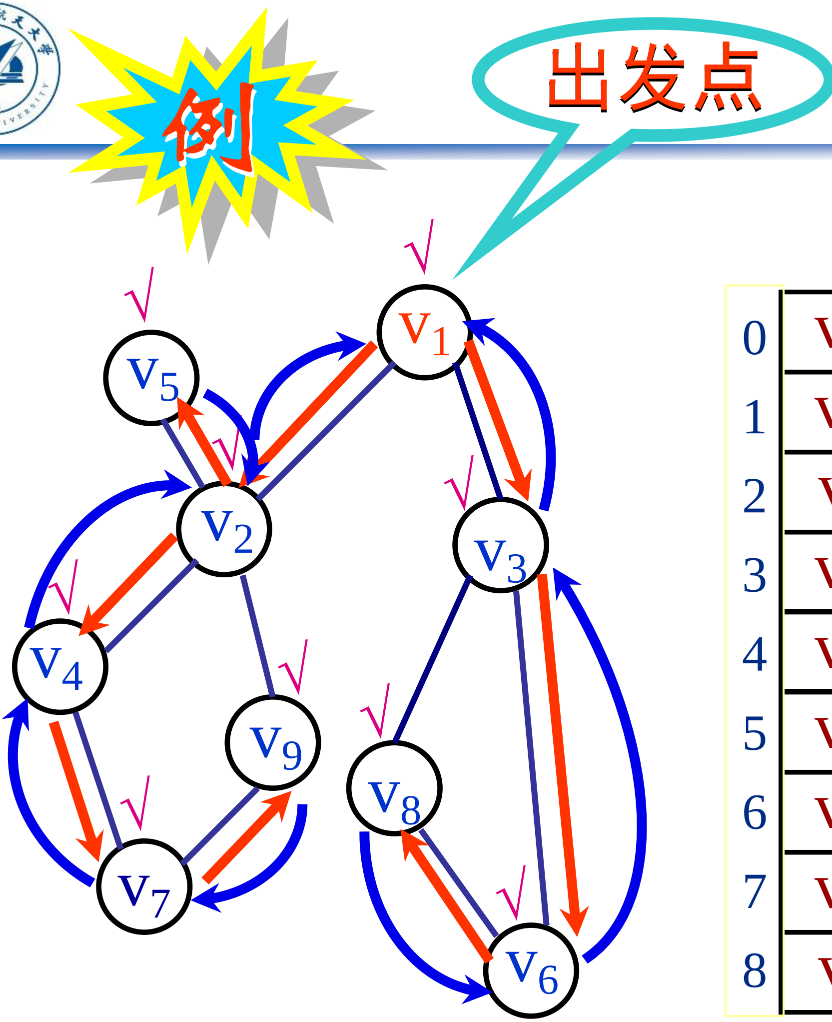
**原则** 从图中某个指定的顶点v出发,先访问顶点v,然后从顶点v未被访问过的某个邻接点出发,继续进行深度优先遍历,直到图中与v相通的所有顶点都被访问;

递归过程

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*

## 例




| | | | | | |
|---|---|---|---|---|---|
| 0 | V1 | → 1 | → 2 ∧ | | |
| 1 | V2 | → 0 | → 3 | → 4 | → 8 ∧ |
| 2 | V3 | → 0 | → 5 | → 7 ∧ | |
| 3 | V4 | → 1 | → 6 ∧ | | |
| 4 | V5 | → 1 ∧ | | | |
| 5 | V6 | → 2 | → 7 ∧ | | |
| 6 | V7 | → 3 | → 8 ∧ | | |
| 7 | V8 | → 2 | → 5 ∧ | | |
| 8 | V9 | → 1 | → 6 ∧ | | |

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

# 如何确保顶点不被重复访问?

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:

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

```
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);
}
```

```
/*遍历一个连通子图*/
void DFS(VLink  G[ ], int v){
    ELink  *p;
    VISIT(G, v);    //访问某顶点
    Visited[v] = 1; //标识某顶点被访问过
    for(p = G[v].link; p !=NULL;  p=p->next)
        if( !Visited[p->adjvex] )
            DFS(G, p->adjvex);
}
```

**树深度优先遍历算法**

```
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]);
      }
}
```

图和树的深度优先遍历算法对比

# 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)，因而算法的时间复杂度为$O(n^2)$。

# DFS与BFS

**一、广度优先遍历**
**(Breadth First Search,BFS)**

**二、深度优先遍历**
**(Depth First Search,DFS)**

DFS与BFS本质上都是“穷举”所有可能，算法时间复杂度是一样的，在很多场景下可以替换使用；不同之处主要在于对顶点的访问的顺序不同，以及得到解的早晚不同。

**例如**：找到一条从源点S到终点D的路径。

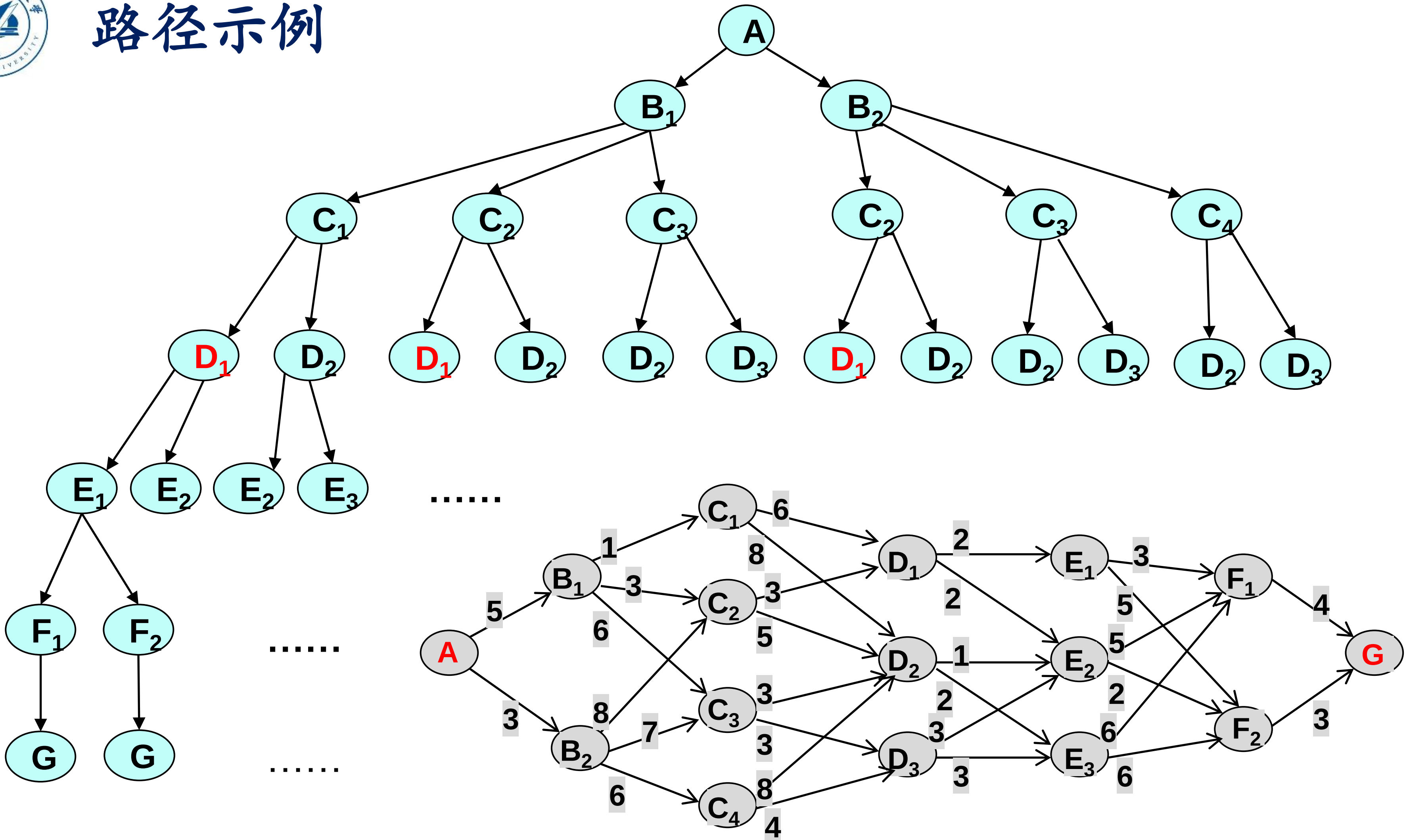
**注：这里的DFS与BFS“遍历”算法都要求每个结点仅被访问一次，实际上有些应用需多次访问某结点。**

# 问题：路径搜索

【背景】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获胜。

【问题】给定一线路图，编程计算从起始点至终点共有**多少条**路径，并输出相关路径信息。

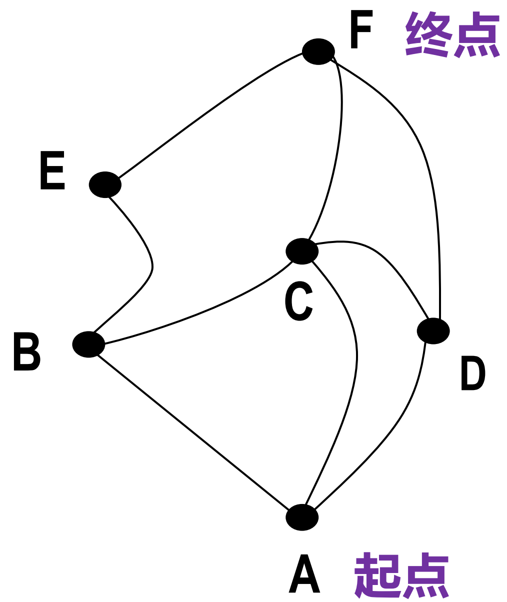
# 路径示例



针对该图：实际上共有2 * 3 * 2 * 2 * 2 * 1=48 条有效的路线，对应一棵有48个叶子节点的树，对应48条路径

# 问题：路径搜索 – 问题描述及分析

- **利用遍历寻找路径**：给定**起点**（如图中A点），对图进行遍历，并在遍历图的过程中找到到达**终点**（如图中F点）的所有路径。
  - 可以是有向图或无向图
- **要点**：顶点可能存在于多条路径中，如ACF,ADCF,ABCF,ABCDF

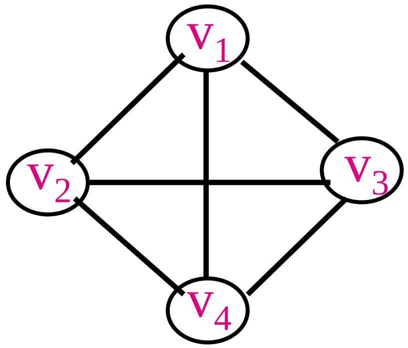


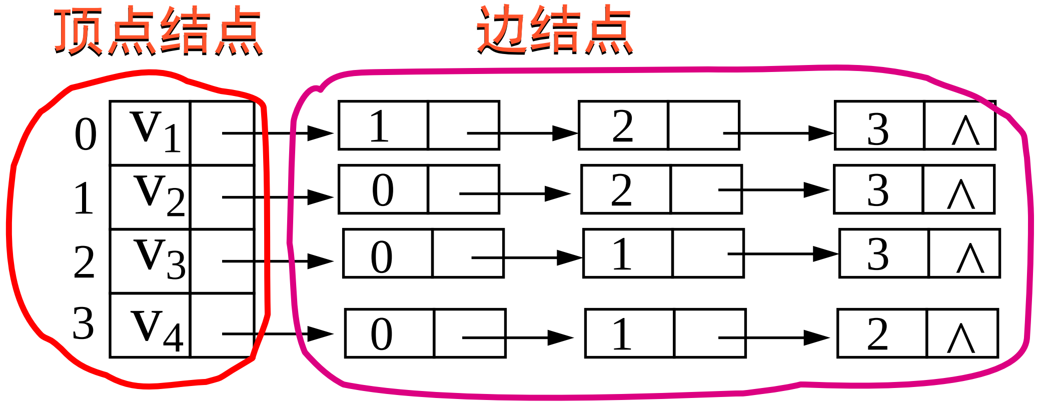

该问题中，一条路径中每个顶点只出现一次；但是顶点可能出现在不同路径中，遍历时需允许顶点被多次访问！

# 问题：独立路径计算－数据结构设计

采用邻接表来存储图

```
#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 paths[MAXSIZE];               //记录一条独立路径
char  Visted[MAXSIZE] = {0};   //标识相应顶点是否被访问（一条路径中）
```



```
int Vs, Vd; //路径起点和终点
int main(){
    int vn, en, i;  //顶点数， 边数; 循环控制变量i
    int  eno, v1,v2;  // 边序号， 边起点， 边终点
    scanf("%d %d", &vn, &en); //顶点数， 边数
    for(i=0; i<en; i++){
        scanf("%d %d %d", &eno, &v1, &v2);
        G[v1].link = insertEdge(G[v1].link,v2, eno);
        G[v2].link = insertEdge(G[v2].link,v1, eno);
    }
    Vs = 0; Vd = vn -1;
    eDFS(G, Vs, 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
```

第一行：顶点数、边数
第二行起的每行：
边序号、边起点、终点

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

```
/*基于DFS的独立路径查找算法*/
void eDFS(VLink  G[ ], int v, int level){
    struct edge  *p;
    if(v == Vd)  { printPath(level); return; } //Vd是终点

    Visited[v] = 1;
    for(p=G[v].link; p!= NULL; p=p->next)
        if( !Visited[p->adjvex]){
            Paths[level] = p->eno; //记录路径信息
            eDFS(p->adjvex, level+1);
        }
    Visited[v] = 0;
}
```

用来避免环路！

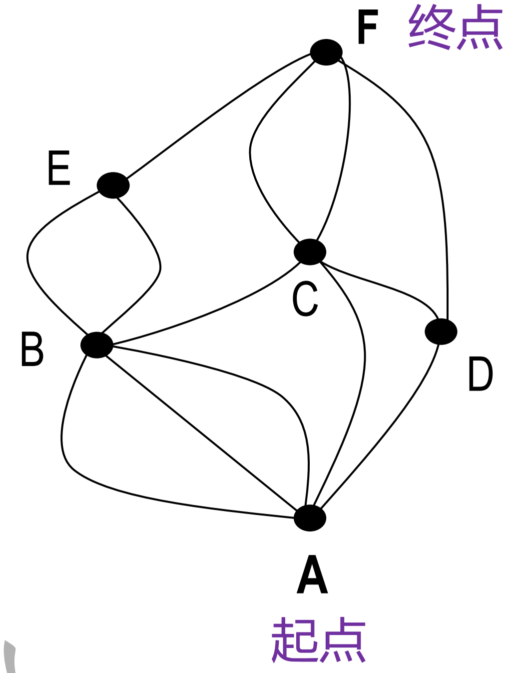
允许顶点被(不同路径)重复访问

Q1：需要记录什么？Q2：既然允许顶点被多次访问，那么可否去掉 visited[]？

# 扩展问题：独立路径计算*

- **利用遍历寻找独立路径**：给定**起点**（如图中A点），对图进行遍历，并在遍历图的过程中找到到达**终点**（如图中F点）的所有路径。
  - 可以是有向图或无向图
  - 允许顶点对之间存在多条边，如A和B之间有三条边

对于本问题，用什么数据结构更合适？
为什么用邻接表比邻接矩阵更合适？




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。