数据结构与算法 (七)

张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6 （“十一五”国家级规划教材)

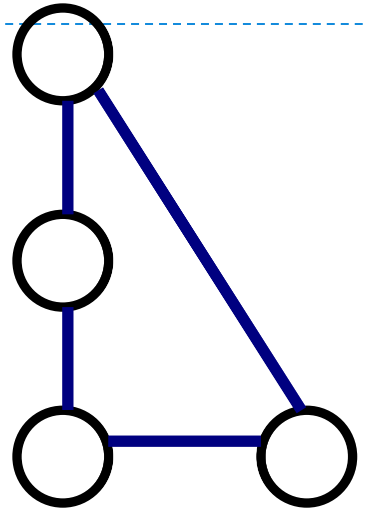
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

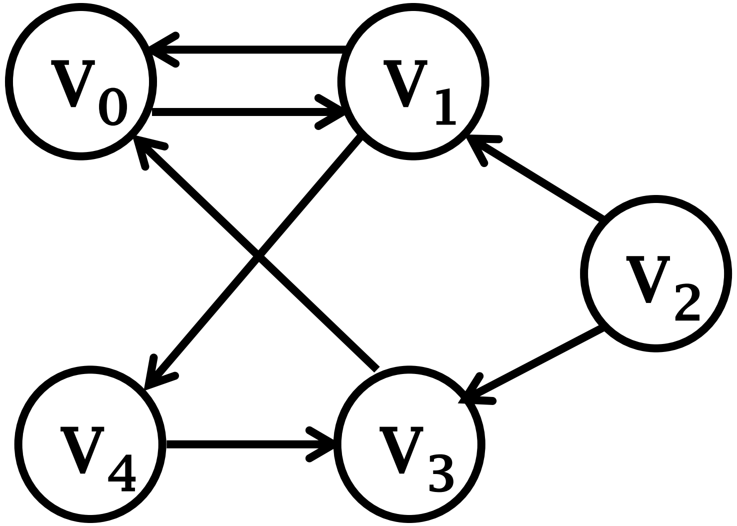
第7章　图

- 7.1　图的定义和术语
- 7.2　图的抽象数据类型
- 7.3　图的存储结构
- 7.4　图的遍历
- 7.5　最短路径
- 7.6　最小生成树

图的定义和术语

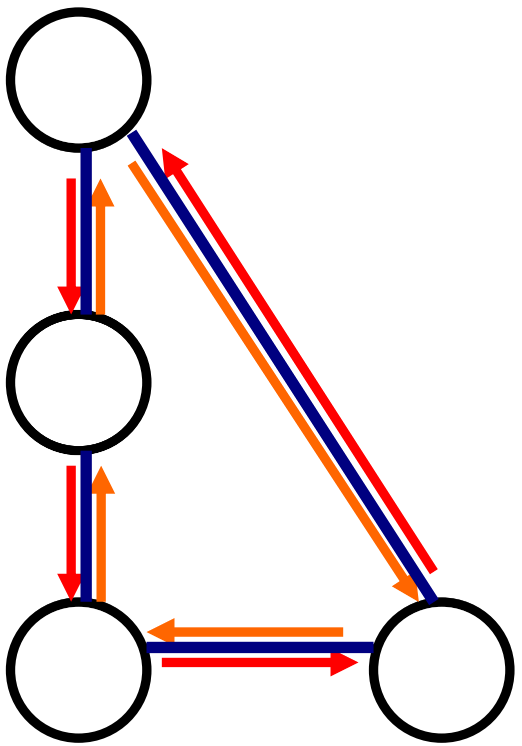
- G= (V , E) 表示
  - V 是顶点 (vertex) 集合
  - E 是边 (edge) 的集合
- 完全图 (complete graph)
- 稀疏图 (sparse graph)
  - 稀疏度（稀疏因子）
  - 边条数小于完全图的5%
- 密集图 (dense graph)





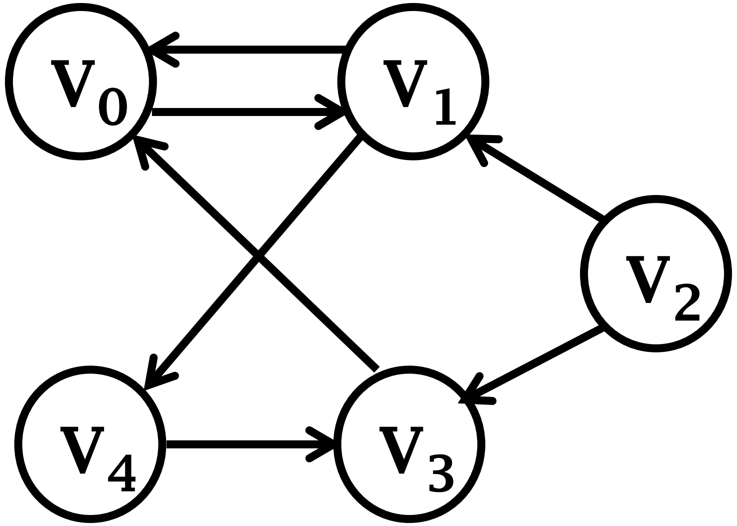
无向图

- 边涉及顶点的偶对无序
- 实际上是双通



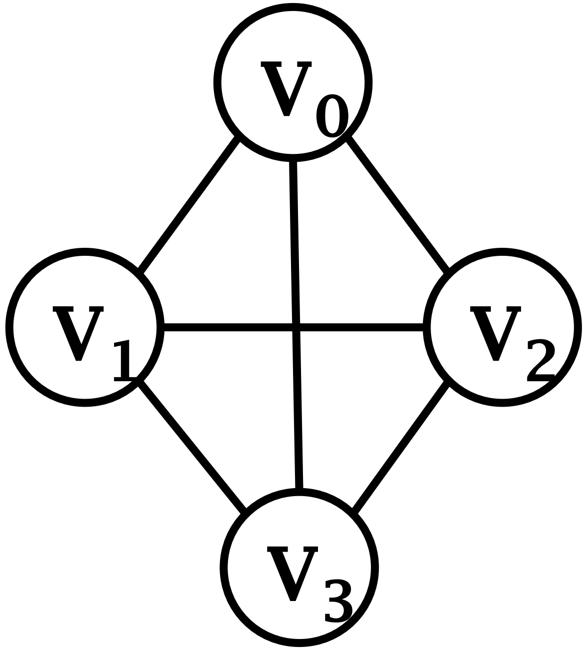
有向图

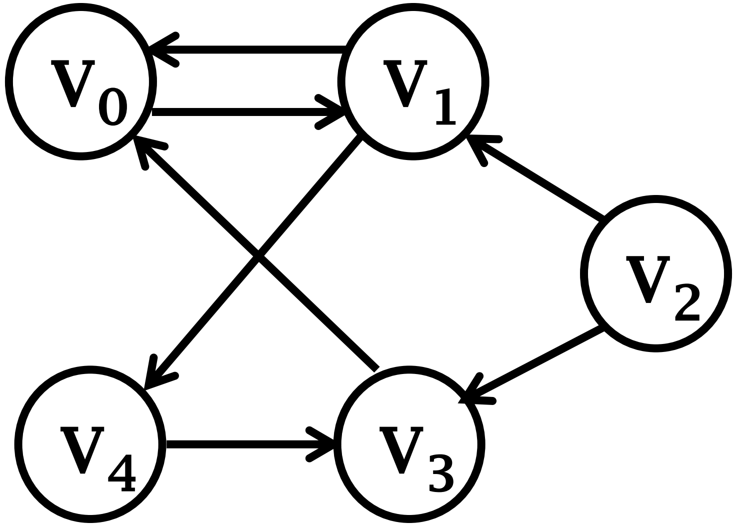
- 有向图 (directed graph 或 digraph)
  - 边涉及顶点的偶对是 **有序** 的



标号图

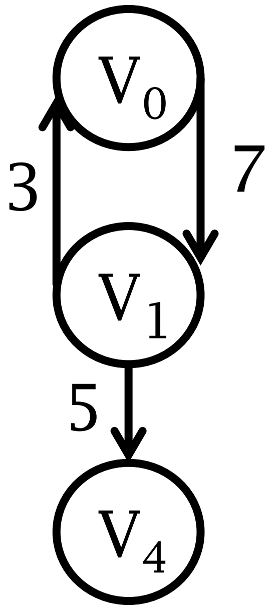
- 标号图 (labeled graph)

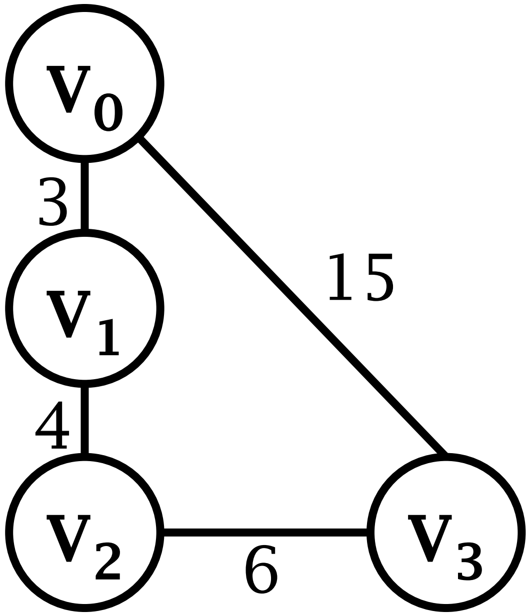




带权图

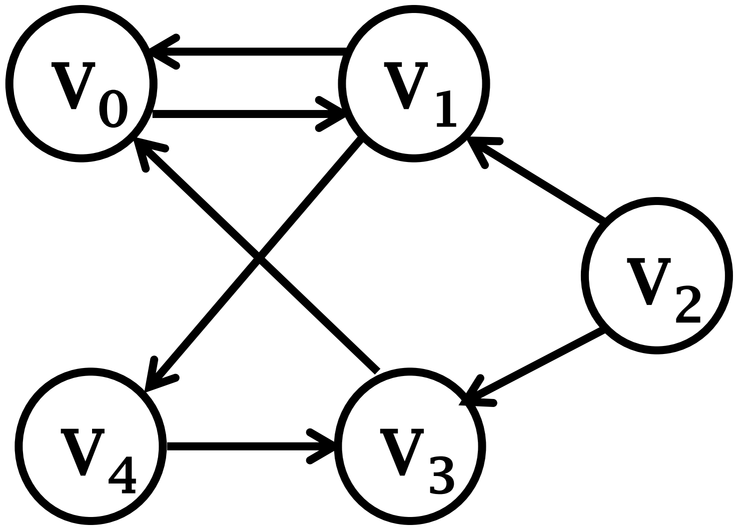
- 带权图 (weighted graph)

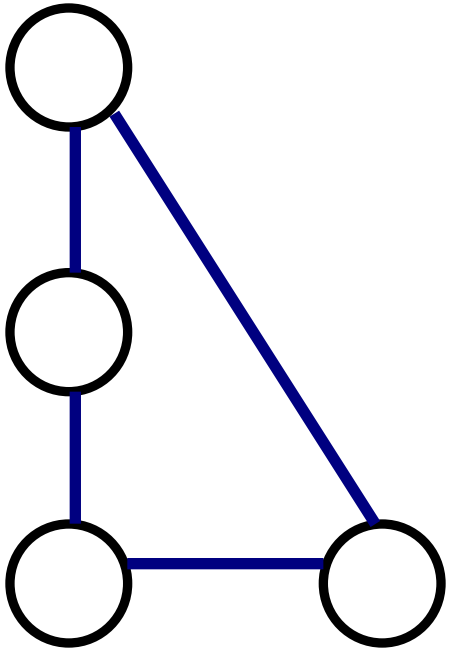




顶点的度 (degree)

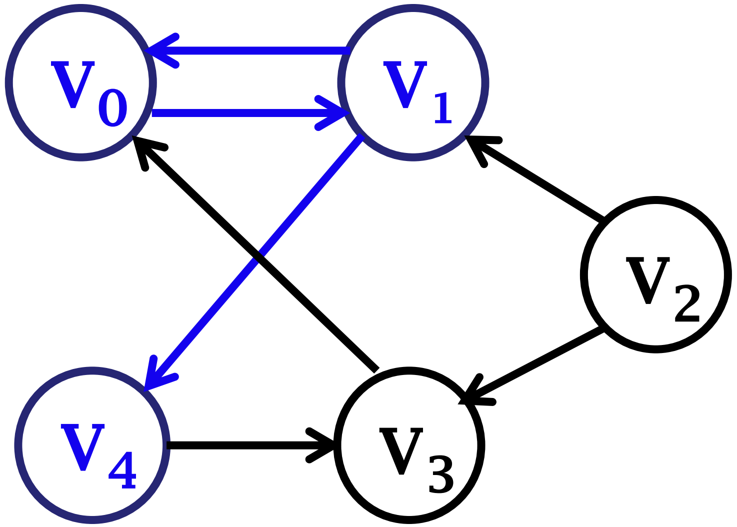
- 与该顶点相关联的边的数目
  - 入度 (in degree)
  - 出度 (out degree)

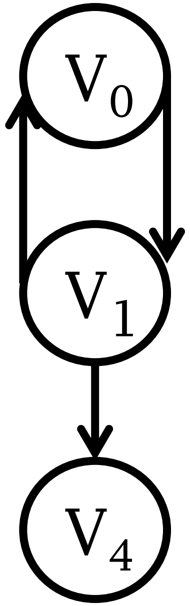




子图 (subgraph)

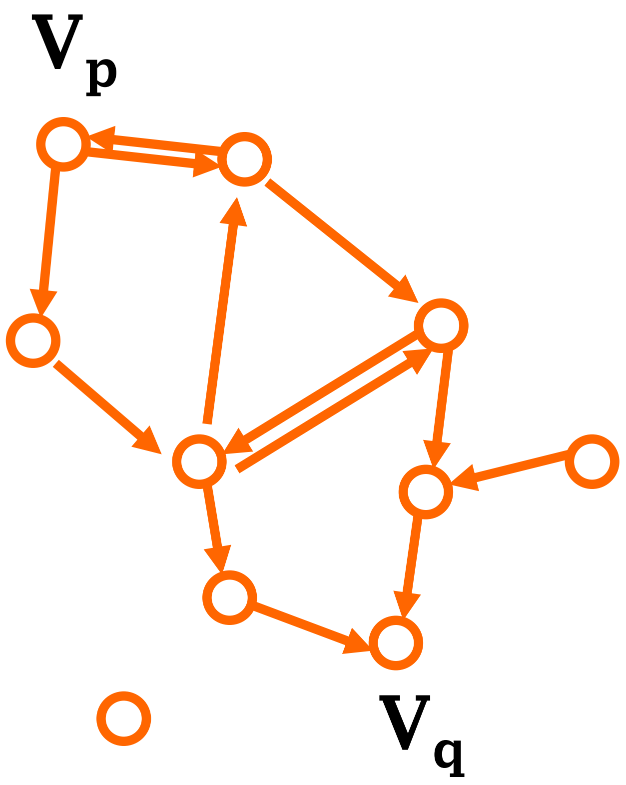
- 图G=（V，E），G’=（V’，E’)中，若 V’≤V，E’≤E，并且 E’中的边所关联的顶点都在 V’中，则称图G’是图G的 **子图**





路径 (path)

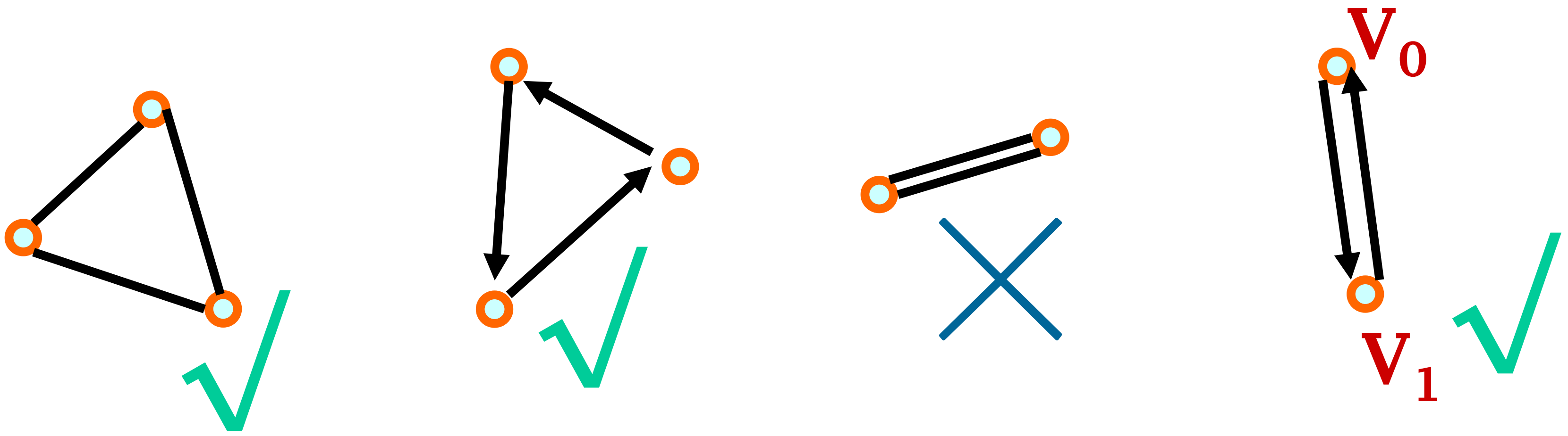
- 从顶点V_p到顶点V_q的路径
  - 顶点序列V_p，V_{i1}，V_{i2}，…，V_{in}，V_q，使得 (V_p, V_{i1})，(V_{i1}, V_{i2})，…，(V_{in}, V_q) (若对有向图，则使得$<V_p, V_{i1}>$，$<V_{i1}, V_{i2}>$，…，$<V_{in}, V_q>$) 都在 E 中
- 简单路径 (simple path)
- 路径长度 (length)



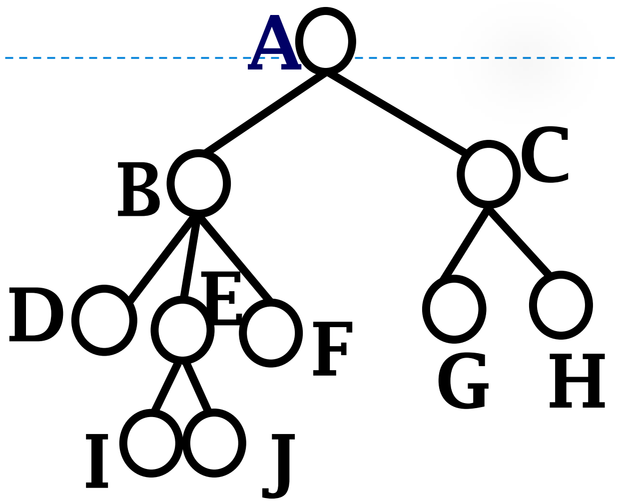
回路（cycle，也称为环）

- 简单回路 (simple cycle)
- 无环图 (acyclic graph)
  - 有向无环图（directed acyclic graph，简写为DAG）

回路（cycle，也称为环）



- 无向图中，如果两个结点之间有平行边，容易让人误看作“环”）
  - **无向图路径长度大于等于 3**
- 有向图两条边可以构成环，例如<V_0，V_1>和<V_1，V_0> 构成环

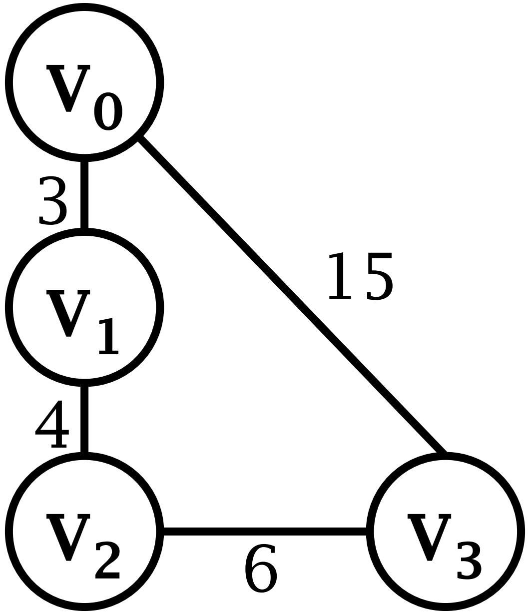


有根图

- 一个有向图中，若存在一个顶点 V_0，从此顶点有路径可以到达图中其它所有顶点，则称此有向图为有根的图，V_0 称作图的根
- 树、森林

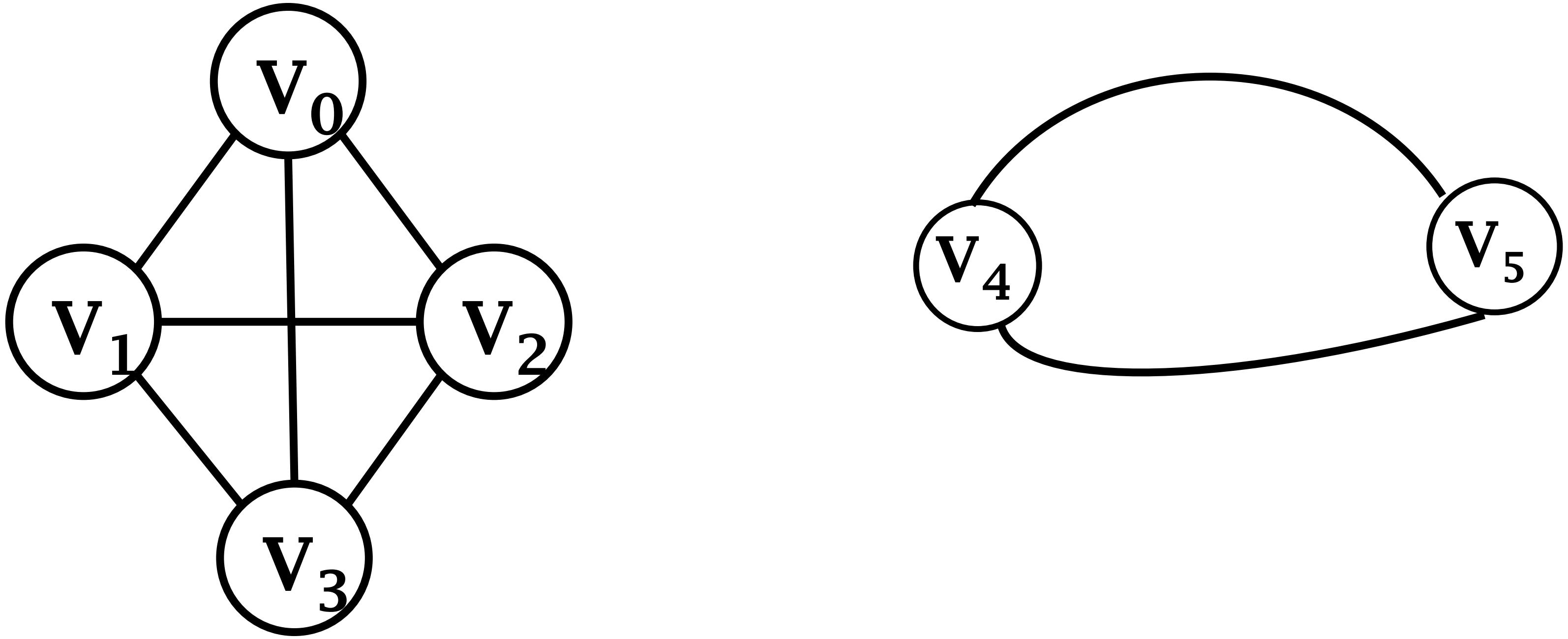
连通图

- 对无向图 G= (V，E) 而言，如果从 V_1 到 V_2 有一条路径 (从 V_2 到 V_1 也一定有一条路径) ，则称 V_1 和V_2 是连通的 (connected)



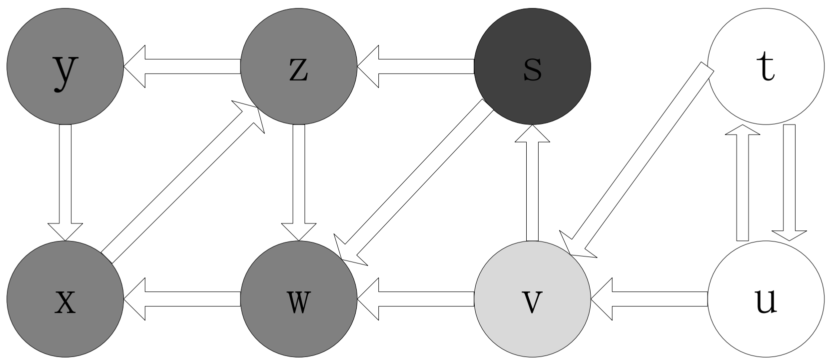
无向图连通分支(连通分量)

- 无向图的最大连通子图



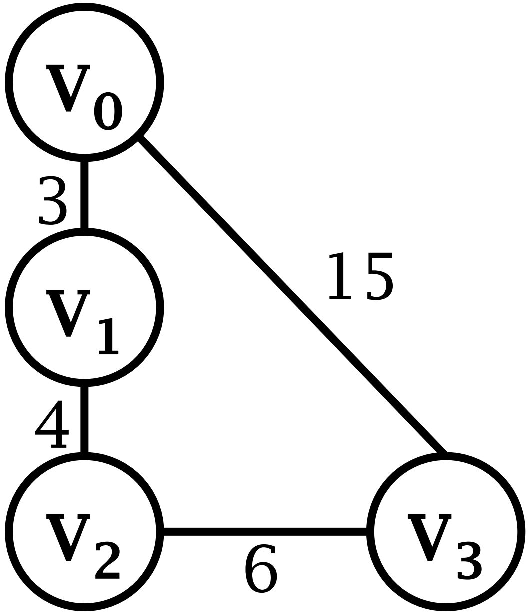
有向图的强连通分量

- 有向图 G (V,E)，如果两个顶点 v_i,v_j 间 (v_i<>v_j) 有一条从 v_i 到 v_j 的有向路径，同时还有一条从 v_j 到 v_i 的有向路径，则称两个顶点 **强连通**
- 非强连通图有向图的极大强连通子图，称为 **强连通分量** (strongly connected components)。



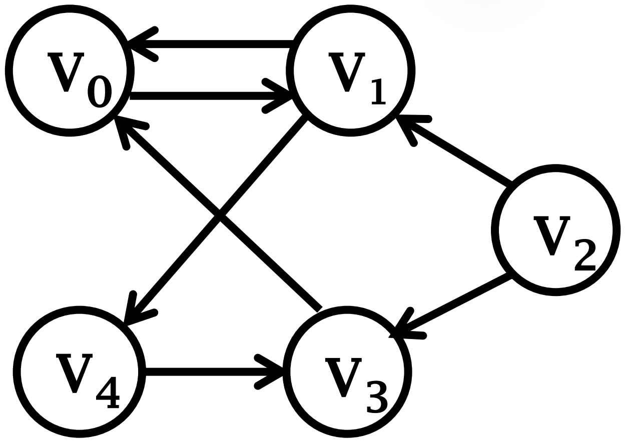
网络

- 带权的连通图



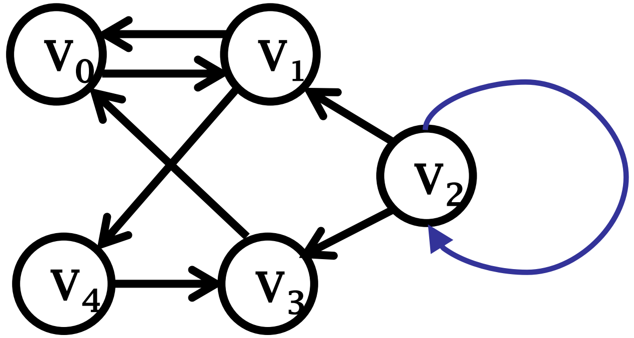
图的抽象数据类型

```
class Graph{                                // 图的ADT
public:
    int    VerticesNum();                   // 返回图的顶点个数
    int    EdgesNum();                      // 返回图的边数
    Edge FirstEdge(int oneVertex);          // 第一条关联边
    Edge NextEdge(Edge preEdge);            // 下一条兄弟边
    bool setEdge(int fromVertex,int toVertex,
           int weight);                     // 添一条边
    bool delEdge(int fromVertex,int toVertex); // 删边
    bool IsEdge(Edge oneEdge);              // 判断oneEdge是否
    int    FromVertex(Edge oneEdge); // 返回边的始点
    int    ToVertex(Edge oneEdge);          // 返回边的终点
    int    Weight(Edge oneEdge);            // 返回边的权
};
```

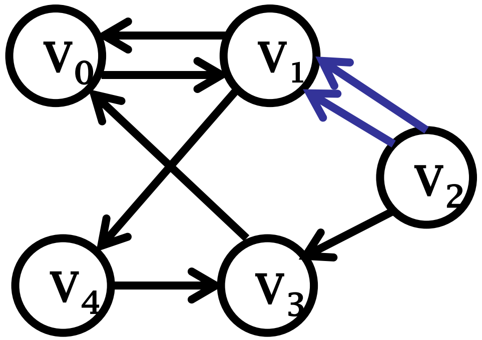


思考

- 为何不允许一条边的起点与终点都是同一个顶点？



- 是否存在多条起点与终点都相同的边？



第7章　图

- 7.1　图的定义和术语
- 7.2　图的抽象数据类型
- 7.3　图的存储结构
- 7.4　图的遍历
- 7.5　最短路径
- 7.6　最小生成树

相邻矩阵

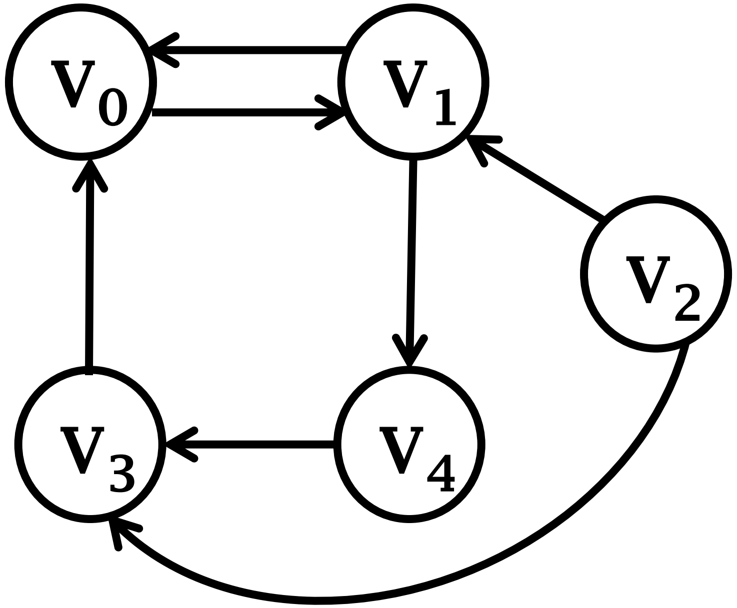
- 图的 **相邻矩阵(adjacency matrix，或邻接矩阵)**表示顶点之间的邻接关系，即有没有边
- 设 G = <V，E> 是一个有 *n* 个顶点的图，则图的相邻矩阵是一个二维数组 A[n，n]，定义如下：

$$A[i,j]=\begin{cases}1, & 若(V_i, V_j)\in E 或 \langle V_i, V_j\rangle \in E\\ 0, & 若(V_i, V_j)\notin E 或 \langle V_i, V_j\rangle \notin E\end{cases}$$

- 对于 *n* 个顶点的图，相邻矩阵的空间代价都为 $O(n^2)$，与边数无关

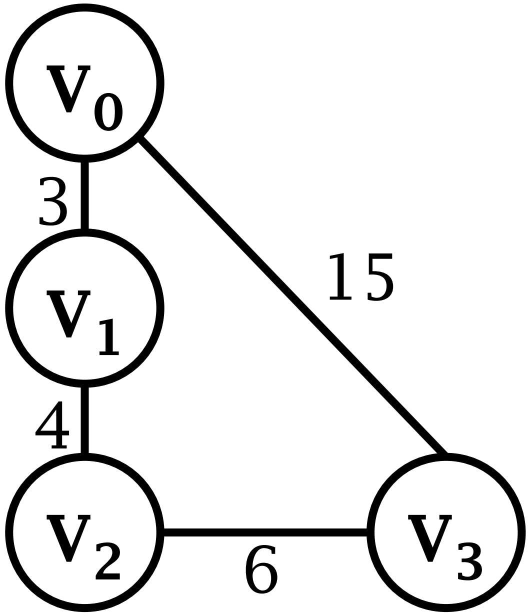
有向图的相邻矩阵

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



无向图的相邻矩阵

$$A = \begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$



相邻矩阵

```
class Edge  {                      // 边类
public:
    int from,to,weight ;           // 边的始点, 终点, 权
    Edge()  {                      // 缺省构造函数
        from = -1;  to = -1;  weight = 0;     }
    Edge(int f,int t,int w){       // 给定参数的构造函数
        from = f;   to = t;   weight = w;      }
};
class Graph  {
public:
    int numVertex;                 // 图中顶点的个数
    int numEdge;                   // 图中边的条数
    int *Mark;                     // 图的顶点访问标记
    int *Indegree;                 // 存放图中顶点的入度
};
```

- 稀疏因子
  - 在 m × n 的矩阵中，有 t 个非零元素，则稀疏因子 δ 为:

$$\delta = \frac{t}{m \times n}$$

  - 若 δ 小于 0.05 ，可认为是稀疏矩阵

邻接表

- 对于稀疏图，可以采用邻接表存储法
  - 边较少，相邻矩阵就会出现大量的零元素
  - 相邻矩阵的零元素将耗费大量的存储 **空间** 和 **时间**
- 邻接表（adjacency list）链式存储结构
  - 顶点表目有两个域：顶点数据域和指向此顶点边表指针域
  - 边表把依附于同一个顶点 v_i 的边（即相邻矩阵中同一行的非0元素）组织成一个单链表。由两个主要的域组成：
    - 与顶点 v_i 邻接的另一顶点的序号
    - 指向边表中下一个边表目的指针

邻接表

- 顶点和边的信息如下所示：

顶点结点

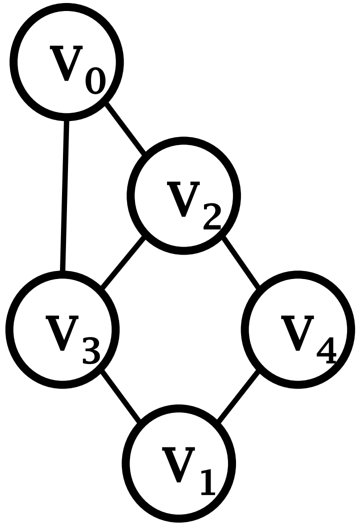
data	firstarc

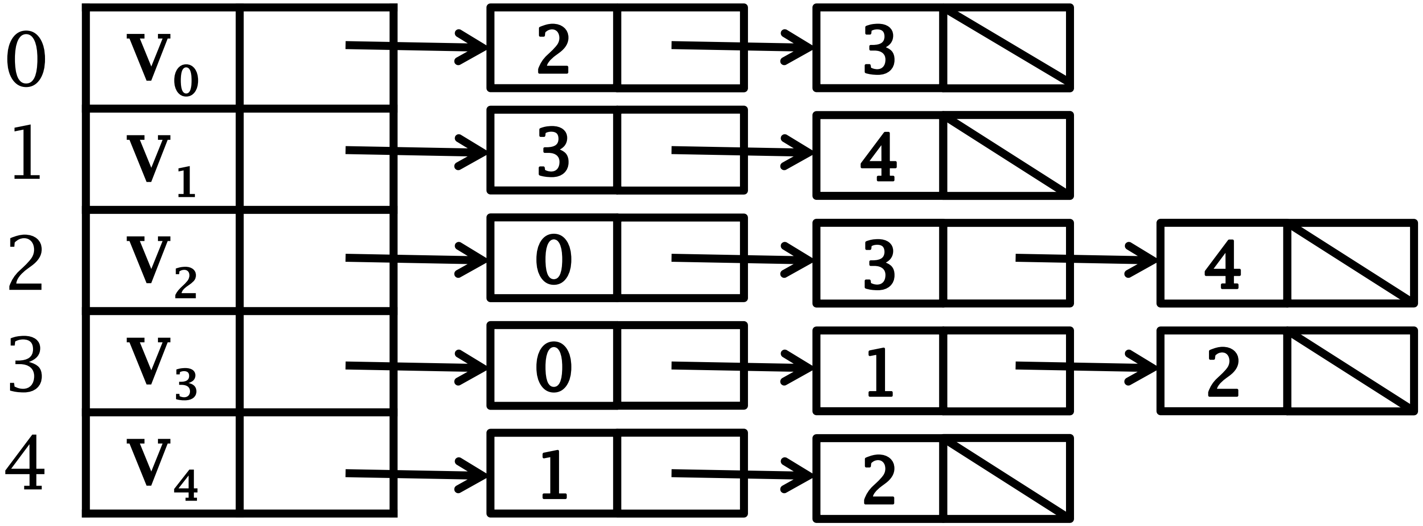
边（或弧）结点

adjvex	nextarc	Info

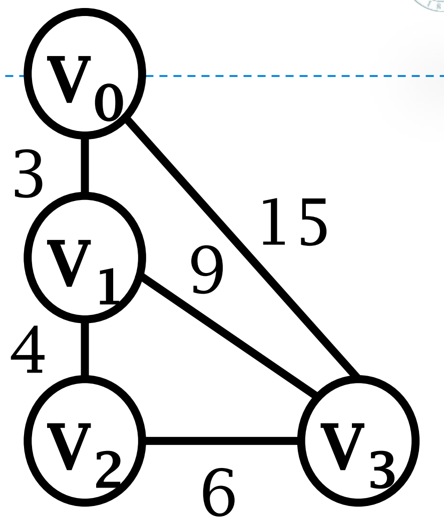
无向图的邻接表表示

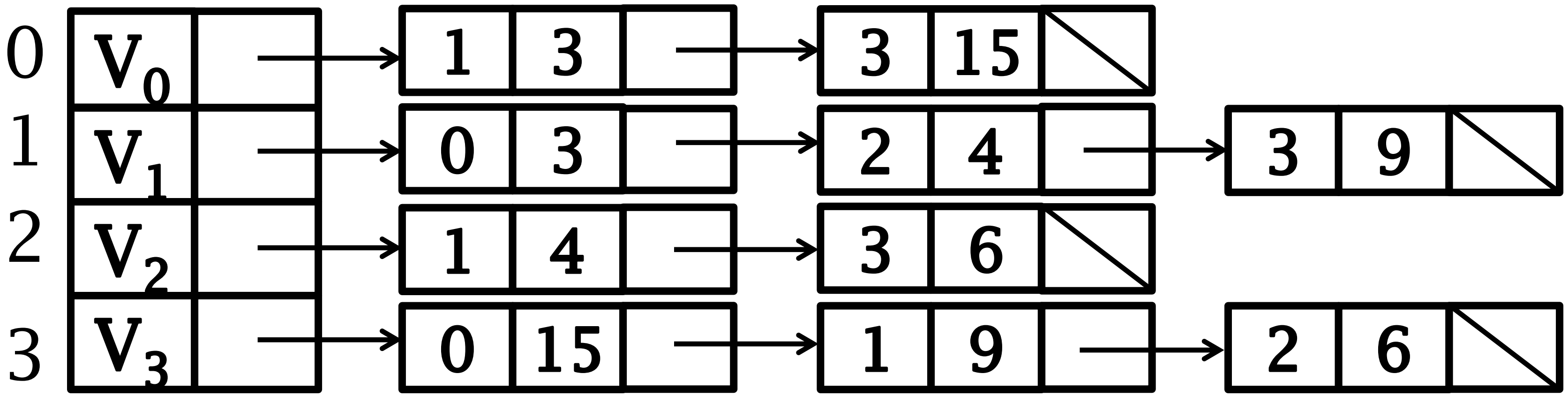
无向图同一条边在邻接表中出现两次





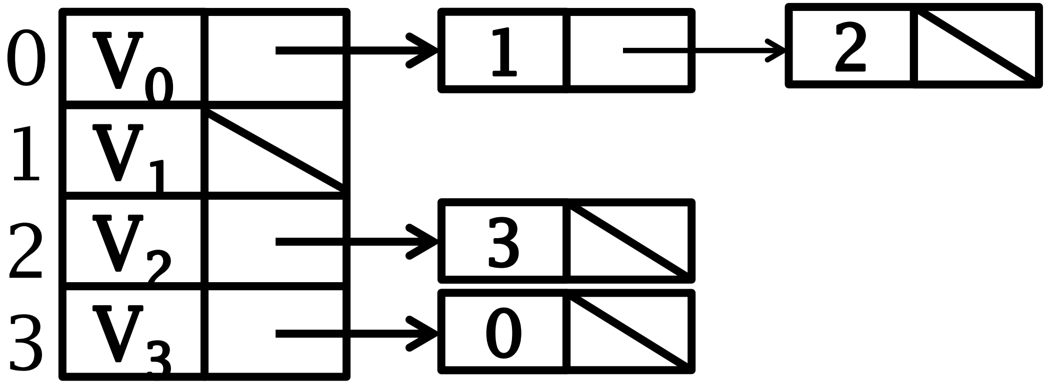
带权图的邻接表表示

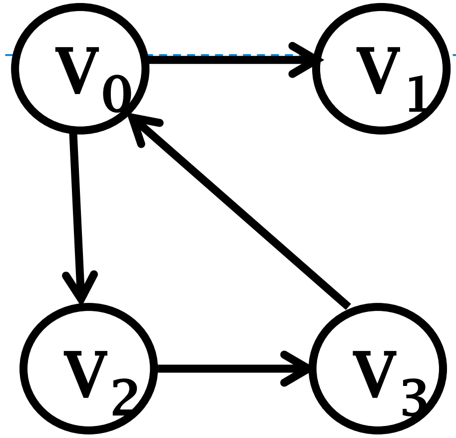




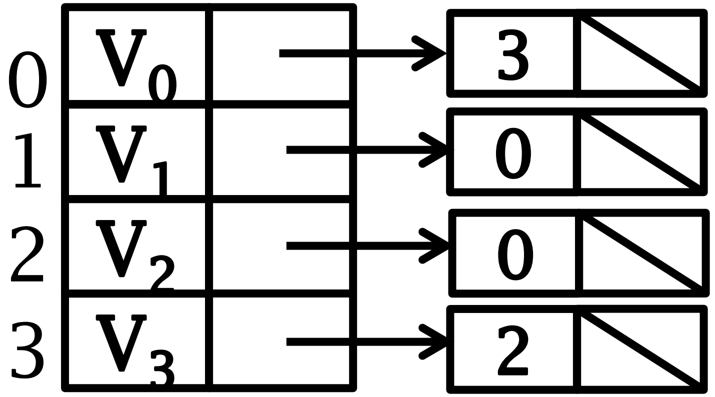
7.3 图的存储结构

有向图的邻接表（出边表）





有向图的逆邻接表（入边表）



图的邻接表空间代价

- n 个顶点 e 条边的无向图
  - 需用 $(n + 2e)$ 个存储单元
- n 个顶点 e 条边的有向图
  - 需用 $(n + e)$ 个存储单元
- 当边数 e 很小时，可以节省大量的存储空间
- 边表中表目顺序往往按照顶点编号从小到大排列

十字链表

- **十字链表 (Orthogonal List)** 可以看成是邻接表和逆邻接表的结合
- 对应于有向图的每一条弧有一个表目，共有5个域：
  - 头 headvex、尾 tailvex 、下一条共尾弧 tailnextarc；下一条共头弧 headnextarc；弧权值等 info 域
- 顶点表目由3个域组成：data 域；firstinarc 第一条以该顶点为终点的弧；firstoutarc 第一条以该顶点为始点的弧

<table>
<tr><td rowspan="2">data</td><td>firstinarc</td></tr>
<tr><td>firstoutarc</td></tr>
</table>

顶点结点

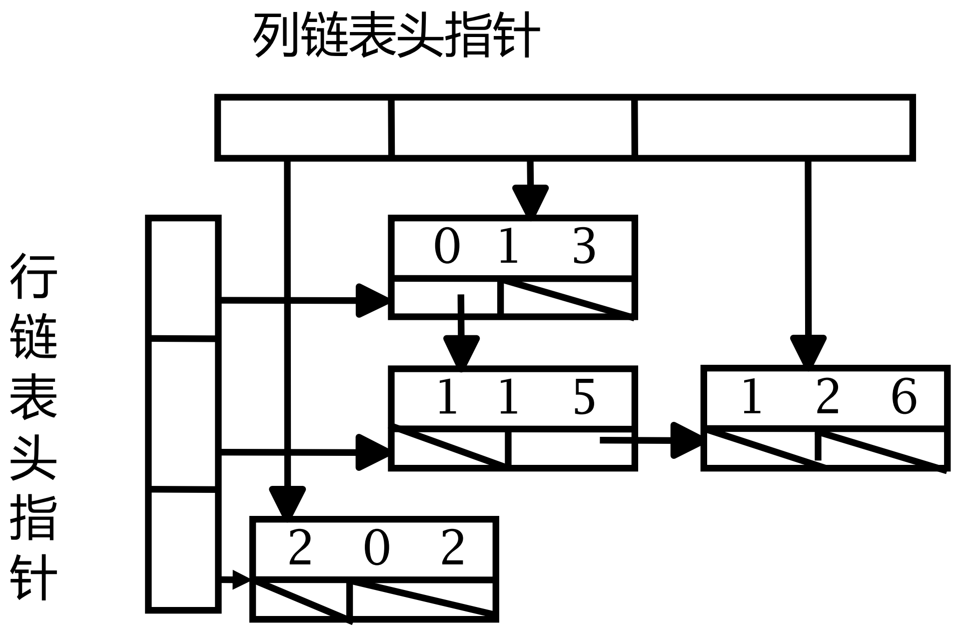
tailvex	tailnextarc	headvex	headnextarc	info

弧(有向边)结点

稀疏矩阵的十字链表

- 十字链表有两组链表组成
  - 行和列的指针序列
  - 每个结点都包含两个指针：同一行的后继，同一列的后继

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 5 & 6 \\ 2 & 0 & 0 \end{bmatrix}$$



数独 Sudoku

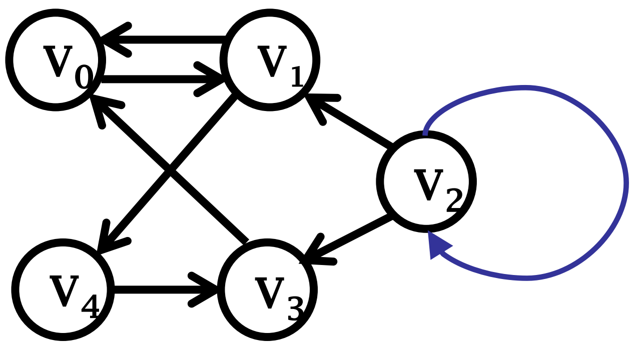
- n×n 个 n×n 的子矩阵拼接而成
  - 每行、每列的数字不重复
  - 每个子矩阵中的数字不重复

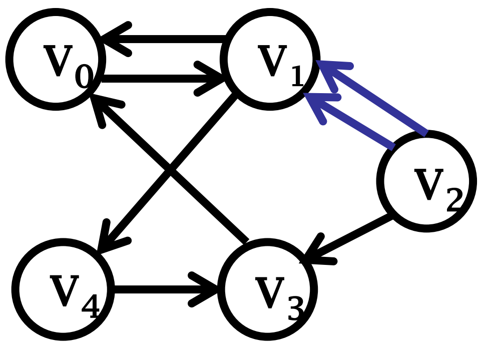
5						3		
	9		5			4		
		4				7		
	5	1		3	7	2	8	9
3		2		8		6		4
		8		5	2	1	3	7
	3	5				9		
6		9				8	2	3
	8			2	3			6

			14	13		6		1			9		5		8
			7			11	5		10	16		1			
			1			8	7		3				6		12
3	11	10	9		14					6				2	
			2	1		3		5					4		15
5	12					2	11			1	8		16		
		16	15				4		12			10		14	9
			10	15	12				2	13					11
4					6	12				7	2	16			
16	3		12			5		8				2	15		
		15		9	4			16						1	13
2		6					16		15		1	8			
	7				16					8		5	10	12	3
10		4				1		9	13			6			
			8		15	4		7	5			14			
15		1		10			8		6		16	7			

思考

- 对于以下两种扩展的复杂图结构，存储结构应做怎样的改变？





第7章　图

- 7.1　图的定义和术语
- 7.2　图的抽象数据类型
- 7.3　图的存储结构
- 7.4　图的遍历
- 7.5　最短路径
- 7.6　最小生成树

图的遍历 (graph traversal)

- 给出一个图G和其中任意一个顶点V_0，从V_0出发系统地访问G中所有的顶点，每个顶点访问而且只访问一次

- **深度优先遍历**
- **广度优先遍历**
- **拓扑排序**

图遍历的考虑

- 从一个顶点出发，试探性访问其余顶点，同时必须考虑到下列情况
  - 从一顶点出发，可能不能到达所有其它的顶点
    - 如 **非连通图**；
  - 也有可能会陷入死循环
    - 如 **存在回路的图**

解决办法

- 为每个顶点保留一个 **标志位** (mark bit)
- 算法开始时，所有顶点的标志位置零
- 在遍历的过程中，当某个顶点被访问时，其标志位就被标记为已访问

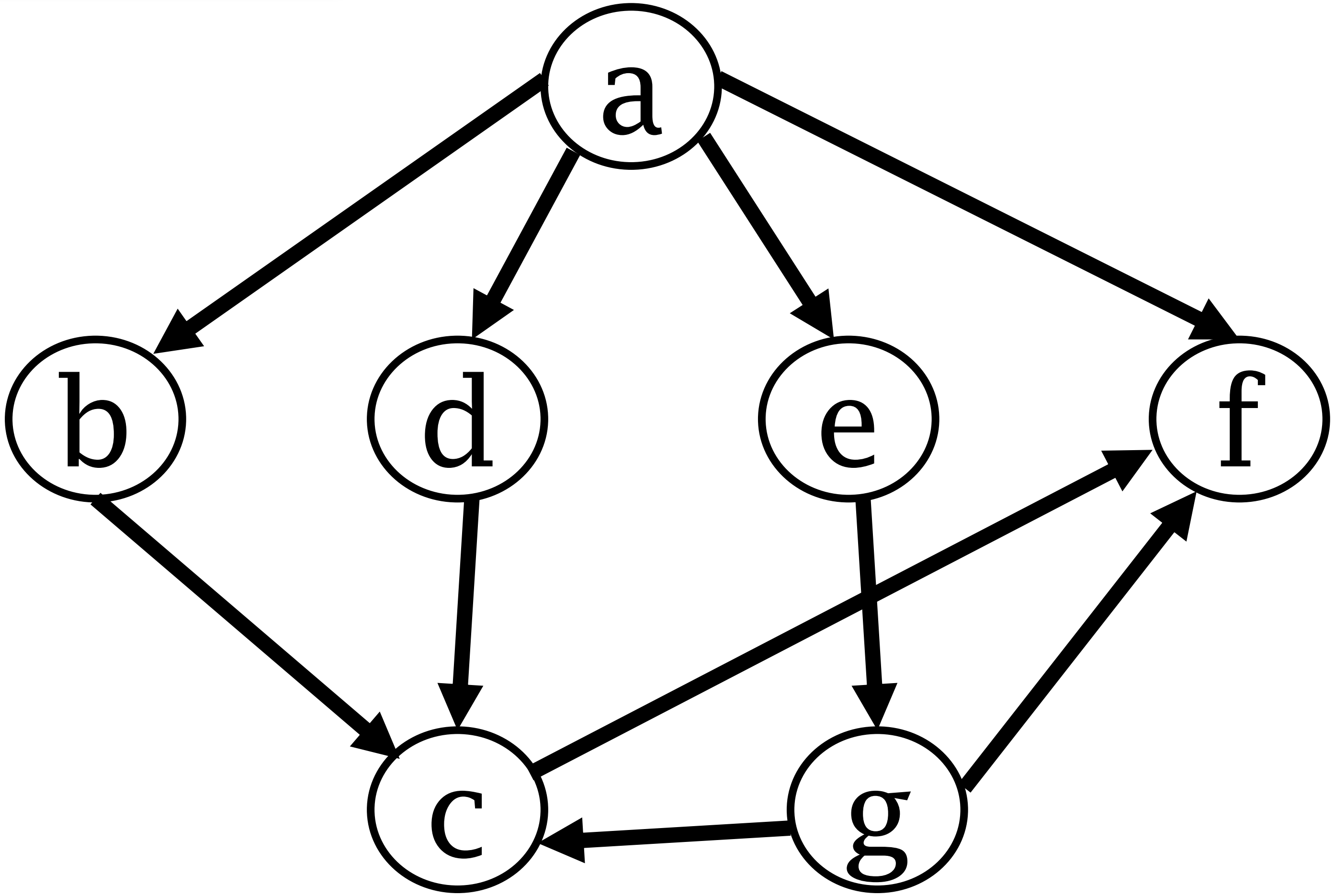
图的遍历算法框架

```
void graph_traverse(Graph& G) {
   // 对图所有顶点的标志位进行初始化
   for(int i=0; i<G.VerticesNum(); i++)
      G.Mark[i] = UNVISITED;
   // 检查图的所有顶点是否被标记过，如果未被标记，
   // 则从该未被标记的顶点开始继续遍历
   // do_traverse函数用深度优先或者广度优先
   for(int i=0; i<G.VerticesNum(); i++)
      if(G.Mark[i] == UNVISITED)
         do_traverse(G, i);
}
```

深度优先遍历（depth-first search）

- 深搜（简称DFS）类似于树的先根次序遍历，尽可能先对纵深方向进行搜索
- 选取一个未访问的点 v_0 作为源点
  - 访问顶点 v_0
  - 递归地深搜遍历 v_0 邻接到的其他顶点
  - 重复上述过程直至从 v_0 有路径可达的顶点都已被访问过
- 再选取其他未访问顶点作为源点做深搜，直到图的所有顶点都被访问过

7.4 图的遍历



深度优先搜索的顺序是： a→b→c→f→d→e→g

图的深度优先遍历（DFS）算法

```
void DFS(Graph& G, int v) {     // 深度优先搜索的递归实现
   G.Mark[v] = VISITED;          // 把标记位设置为 VISITED
   Visit(G,v);                   // 访问顶点v
   for (Edge e = G.FirstEdge(v); G.IsEdge(e);
            e = G.NextEdge(e))
     if (G.Mark[G.ToVertex(e)] == UNVISITED)
       DFS(G, G.ToVertex(e));
   PostVisit(G,v);               // 对顶点v的后访问
}
```

广度优先遍历

- **广度优先搜索 (breadth-first search**，简称 **BFS)**。其遍历的过程是：
  - 从图中的某个顶点 v_0 出发
    - 访问并标记了顶点 v_0 之后
    - 一层层横向搜索 v_0 的所有邻接点
    - 对这些邻接点一层层横向搜索，直至所有由 v_0 有路径可达的顶点都已被访问过
  - 再选取其他未访问顶点作为源点做广搜，直到所有点都被访问过

图的广度优先遍历(BFS)算法

```
void BFS(Graph& G, int v)  {
    using std::queue;   queue<int> Q;      // 使用STL中的队列
    Visit(G,v);                            // 访问顶点v
    G.Mark[v] = VISITED;   Q.push(v);      // 标记,并入队列
    while (!Q.empty()) {                   // 如果队列非空
        int u = Q.front ();                // 获得队列顶部元素
        Q.pop();                           // 队列顶部元素出队
        for (Edge e = G.FirstEdge(u); G.IsEdge(e);
                e = G.NextEdge(e))         // 所有未访问邻接点入队
            if (G.Mark[G.ToVertex(e)] == UNVISITED){
                Visit(G, G.ToVertex(e));
                G.Mark[G.ToVertex(e)] = VISITED;
                Q.push(G.ToVertex(e));
            }
}}
```

图搜索的时间复杂度

- DFS 和 BFS 每个顶点访问一次，对每一条边处理一次 (无向图的每条边从两个方向处理)
  - 采用邻接表表示时，有向图总代价为 $\Theta(n + e)$，无向图为 $\Theta(n + 2e)$
  - 采用相邻矩阵表示时，处理所有的边需要 $\Theta(n^2)$ 的时间 ，所以总代价为

$$\Theta(n + n^2) = \Theta(n^2)$$

拓扑排序

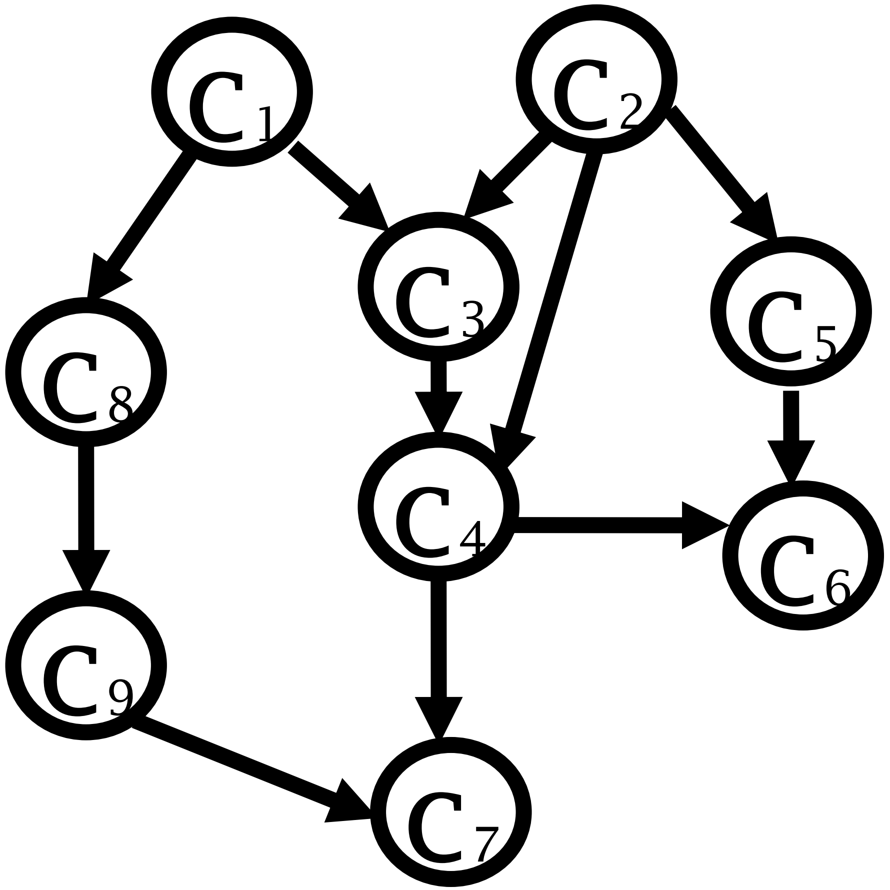
- 对于 **有向无环图** G=（V，E），V 里顶点的线性序列称作一个 **拓扑序列**，该顶点序列满足：
  - 若在有向无环图 G 中从顶点 v_i 到 v_j 有一条路径，则在序列中顶点 v_i 必在顶点 v_j 之前
- 拓扑排序 (topological sort)
  - 将一个 **有向无环图** 中所有顶点在不违反 **先决条件关系** 的前提下排成线性序列的过程称为 **拓扑排序**

7.4 图的遍历

课程代号	课程名称	先修课程
C1	高等数学	
C2	程序设计	
C3	离散数学	C1，C2
C4	数据结构	C2，C3
C5	算法分析	C2
C6	编译技术	C4，C5
C7	操作系统	C4，C9
C8	普通物理	C1
C9	计算机原理	C8

拓扑排序图例

学生课程的安排图



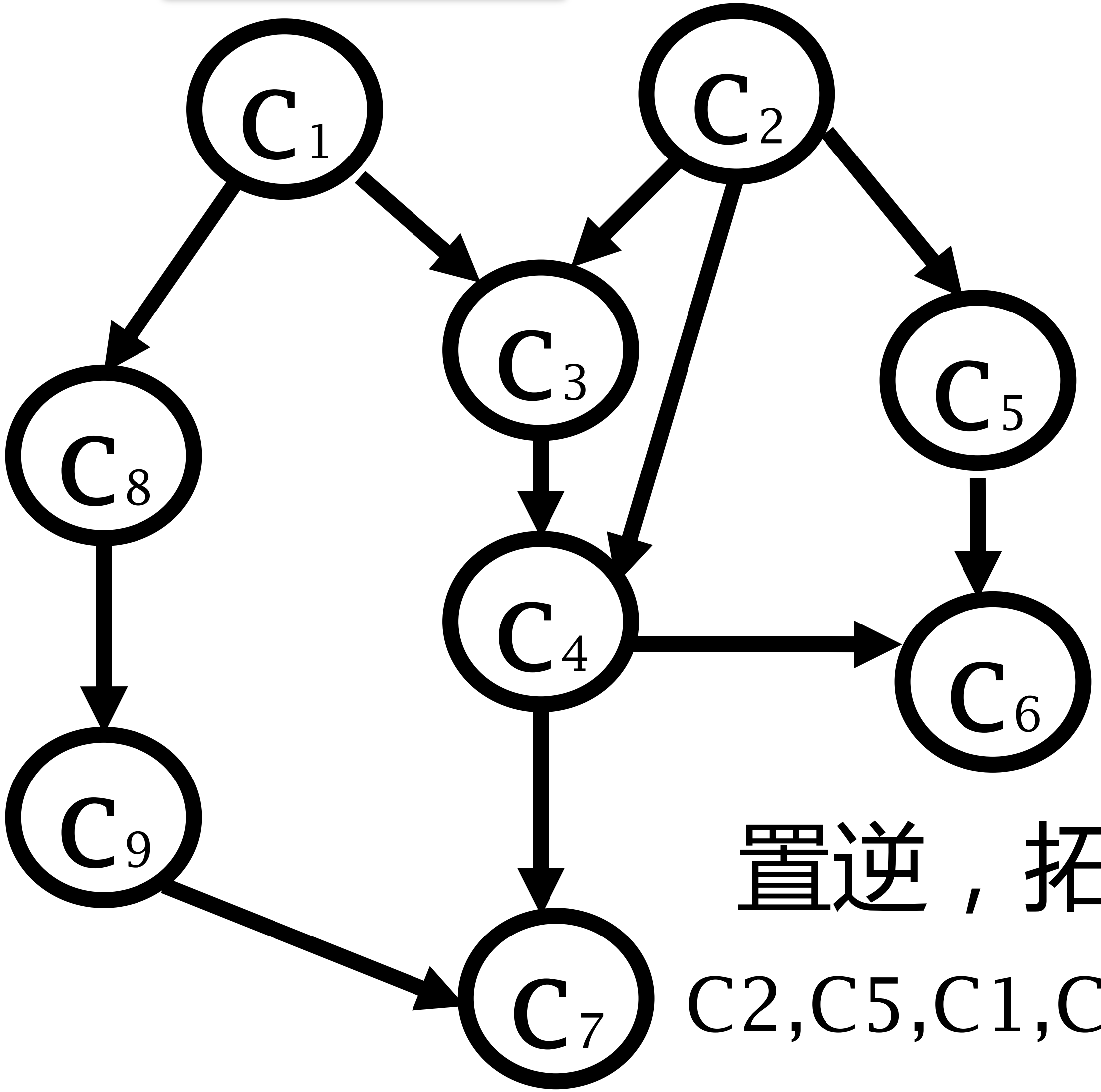
拓扑排序方法

- 任何 **有向无环图 (DAG)** ，其顶点都可以排在一个拓扑序列里，其拓扑排序的方法是：
  - (1) 从图中选择**任意**一个入度为0的顶点且输出之
  - (2) 从图中删掉此顶点及其所有的出边，将其入度减少1
  - (3) 回到第 (1) 步继续执行

用队列实现的图拓扑排序

```
void TopsortbyQueue(Graph& G) {
    for (int i = 0; i < G.VerticesNum(); i++) G.Mark[i] = UNVISITED; // 初始化
    using std::queue; queue<int> Q;              // 使用STL中的队列
    for (i = 0; i < G.VerticesNum(); i++)        // 入度为0的顶点入队
        if (G.Indegree[i] == 0)  Q.push(i);
    while (!Q.empty()) {                         // 如果队列非空
        int v = Q.front(); Q.pop();              // 获得队列顶部元素， 出队
        Visit(G,v);     G.Mark[v] = VISITED;     // 将标记位设置为VISITED
        for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e)) {
            G.Indegree[G.ToVertex(e)]--;         // 相邻的顶点入度减1
            if (G.Indegree[G.ToVertex(e)] == 0)  // 顶点入度减为0则入队
                Q.push(G.ToVertex(e));
    } }
    for (i = 0; i < G.VerticesNum(); i++)        // 判断图中是否有环
        if (G.Mark[i] == UNVISITED) {
            cout<<“ 此图有环！”;      break;
} }
```

7.4 图的遍历



按结点编号深度优先：

C6C7C4C3C9C8C1C5C2

置逆，拓扑序列为：

C2,C5,C1,C8,C9,C3,C4,C7,C6

深度优先搜索实现的拓扑排序

```
int  *TopsortbyDFS(Graph& G)  {              // 结果是颠倒的
   for(int i=0; i<G.VerticesNum(); i++)      // 初始化
      G.Mark[i] = UNVISITED;
   int *result=new int[G.VerticesNum()];
   int index=0;
   for(i=0; i<G.VerticesNum(); i++)          // 对所有顶点
      if(G.Mark[i] == UNVISITED)
         Do_topsort(G, i, result, index);     // 递归函数
   for(i=G.VerticesNum()-1; i>=0; i--)        // 逆序输出
      Visit(G, result[i]);
   return result;
}
```

拓扑排序递归函数

```
void Do_topsort(Graph& G, int V, int *result, int&
index) {
   G.Mark[V] = VISITED;
   for (Edge e = G.FirstEdge(V);
          G.IsEdge(e); e=G.NextEdge(e)) {
      if (G.Mark[G.ToVertex(e)] == UNVISITED)
          Do_topsort(G, G.ToVertex(e),
                      result, index);
   }
   result[index++]=V;  // 相当于后处理
}
```

拓扑排序的时间复杂度

- 与图的深度优先搜索方式遍历相同
  - 图的每条边处理一次
  - 图的每个顶点访问一次
- 采用邻接表表示时，为 $\Theta(n + e)$
- 采用相邻矩阵表示时，为 $\Theta(n^2)$

图算法需要考虑的问题

- 是否支持
  - 有向图、无向图
  - 有回路的图
  - 非连通图
  - 权值为负
- 如果不支持
  - 则修改方案？

递归与非递归的拓扑排序

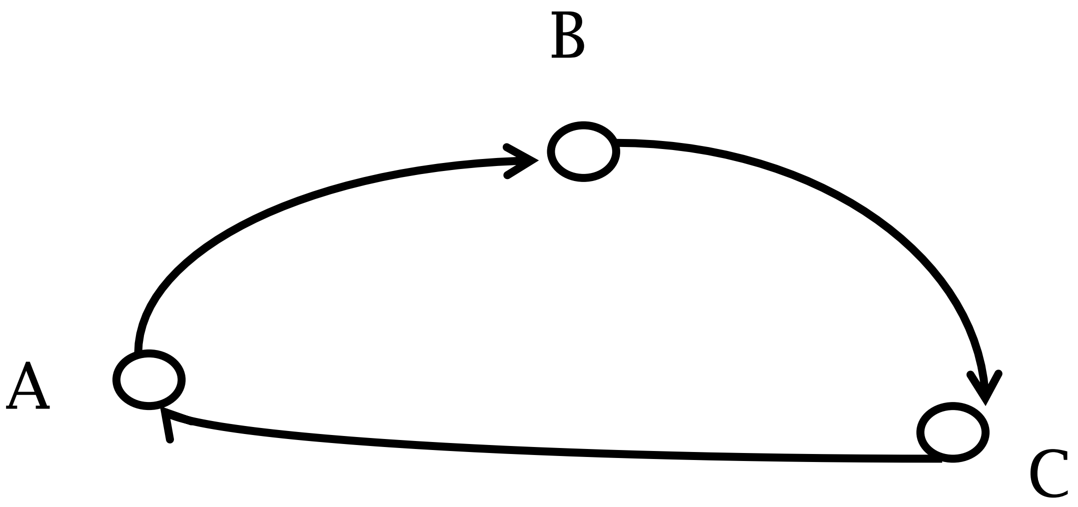
- 必须是有向图
- 必须是无环图
- 支持非连通图
- 不用考虑权值
- 回路
  - 非递归的算法，最后判断 (若还有顶点没有输出，肯定有回路)
  - 递归的算法要求判断有无回路

队列实现的拓扑排序讨论

- 怎么知道图中所有顶点的入度？
- 是否可以用栈来取代队列？

深度优先搜索拓扑排序讨论

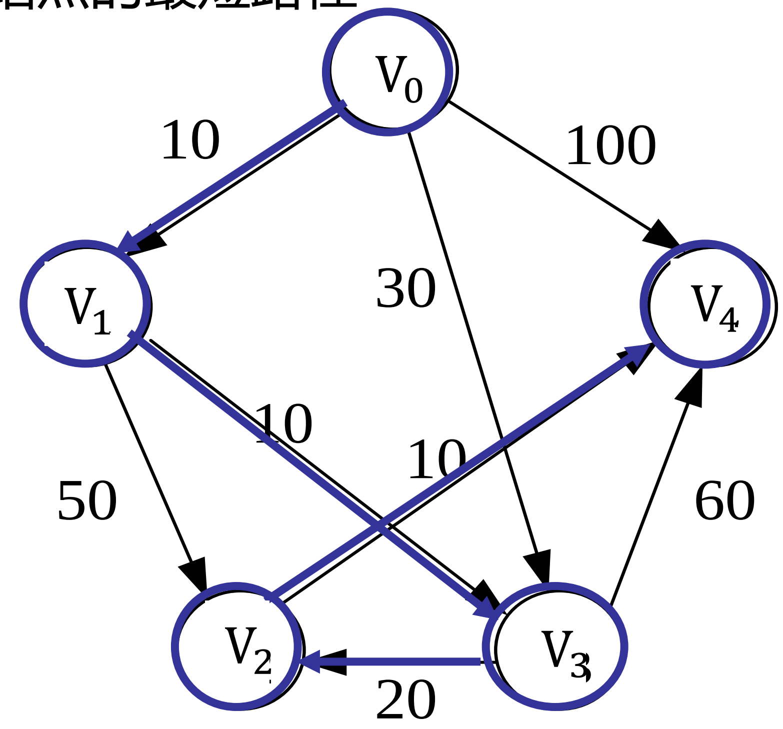
- 对于起始点是否有要求？
- 是否可以处理有环的情况？



第7章 图

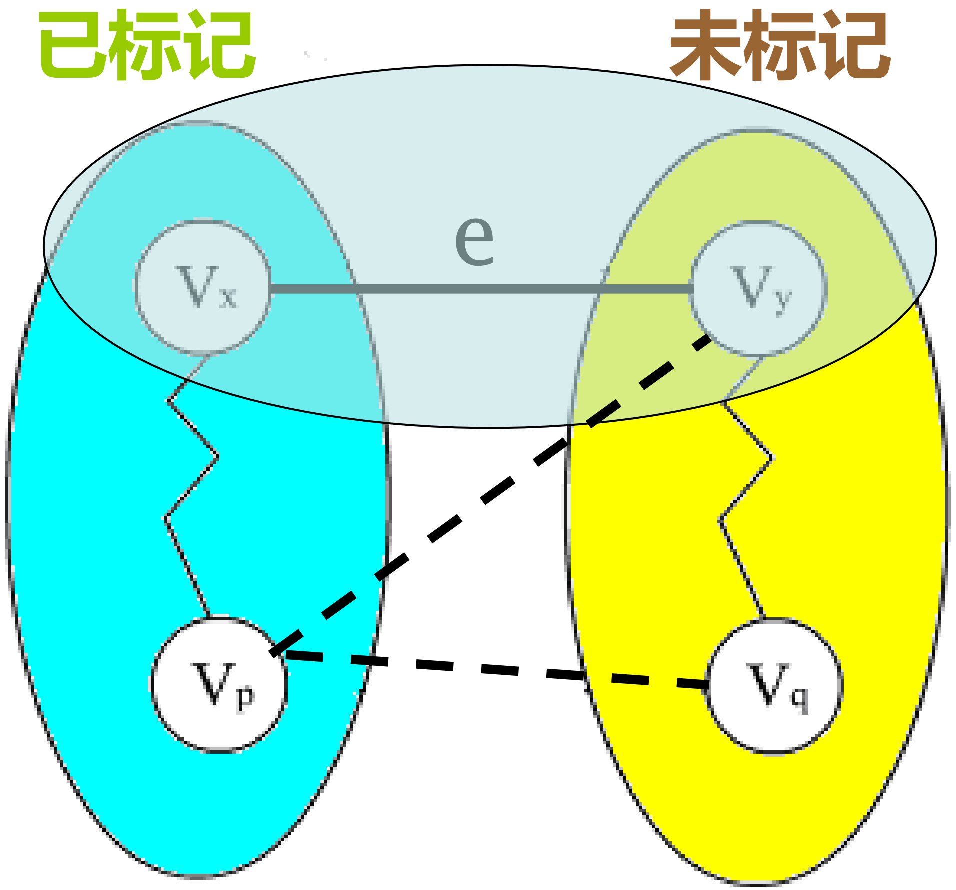
单源最短路径

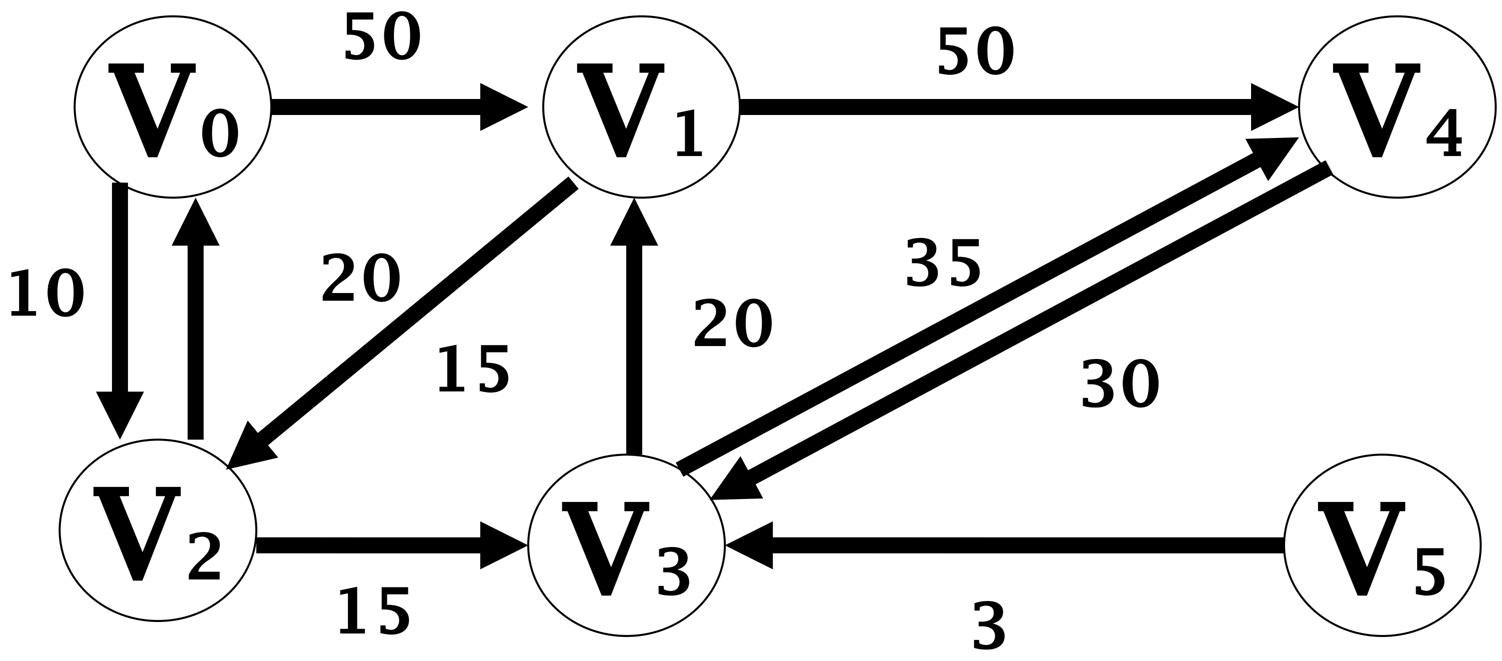
- **单源最短路径(single-source shortest paths)**
  - 给定带权图 G = <V，E>，其中每条边 (v_i, v_j) 上的权 $W[v_i, v_j]$ 是一个 **非负实数** 。计算从任给的一个源点 s 到所有其他各结点的最短路径



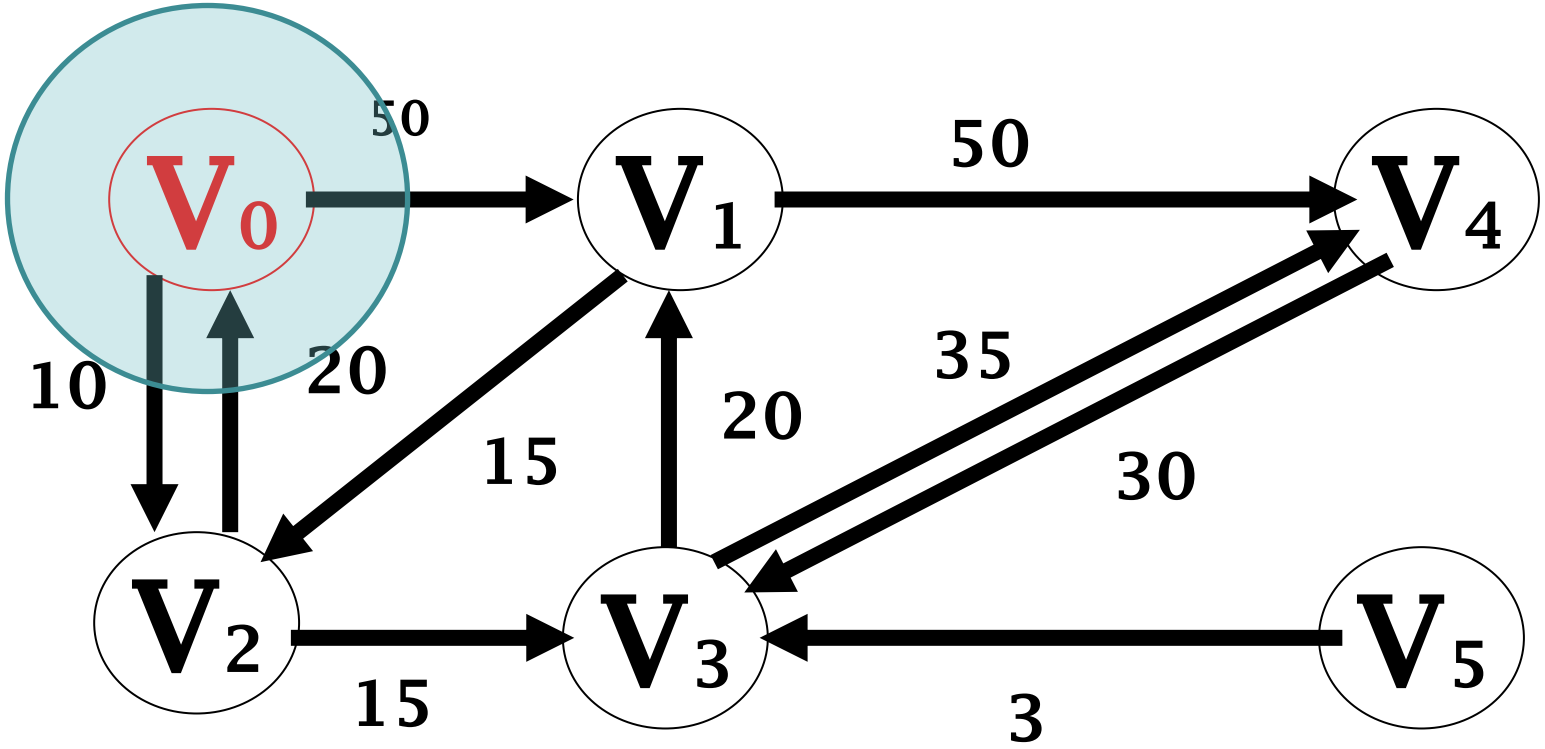
Dijkstra算法基本思想

- 把所有结点分成两组
  - 第一组 U 包括已确定最短路径的结点
  - 第二组 V-U 包括尚未确定最短路径的结点
- 按最短路径长度递增的顺序逐个把第二组的结点加到第一组中
  - 直至从 s 出发可达结点都包括进第一组中

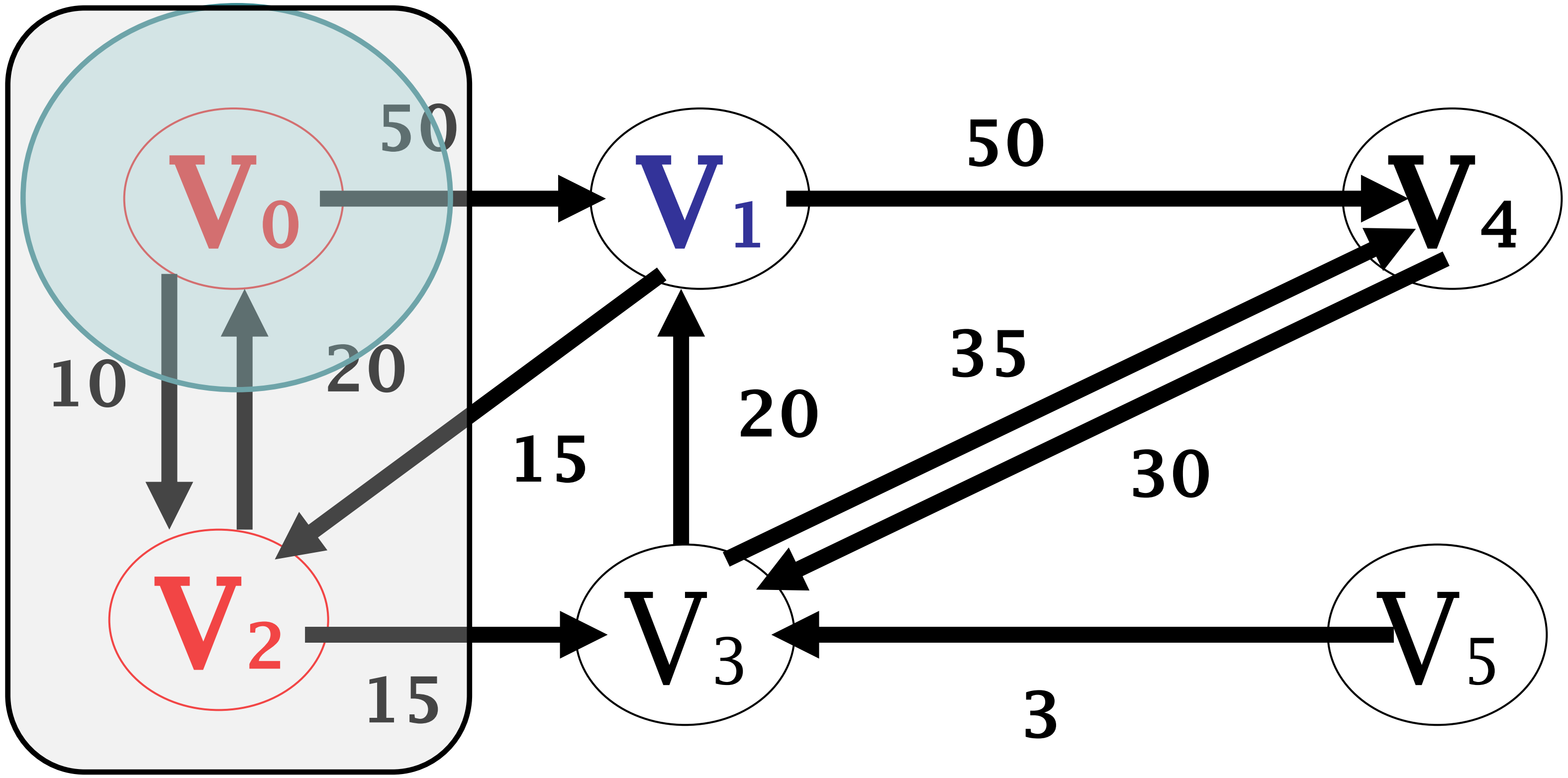




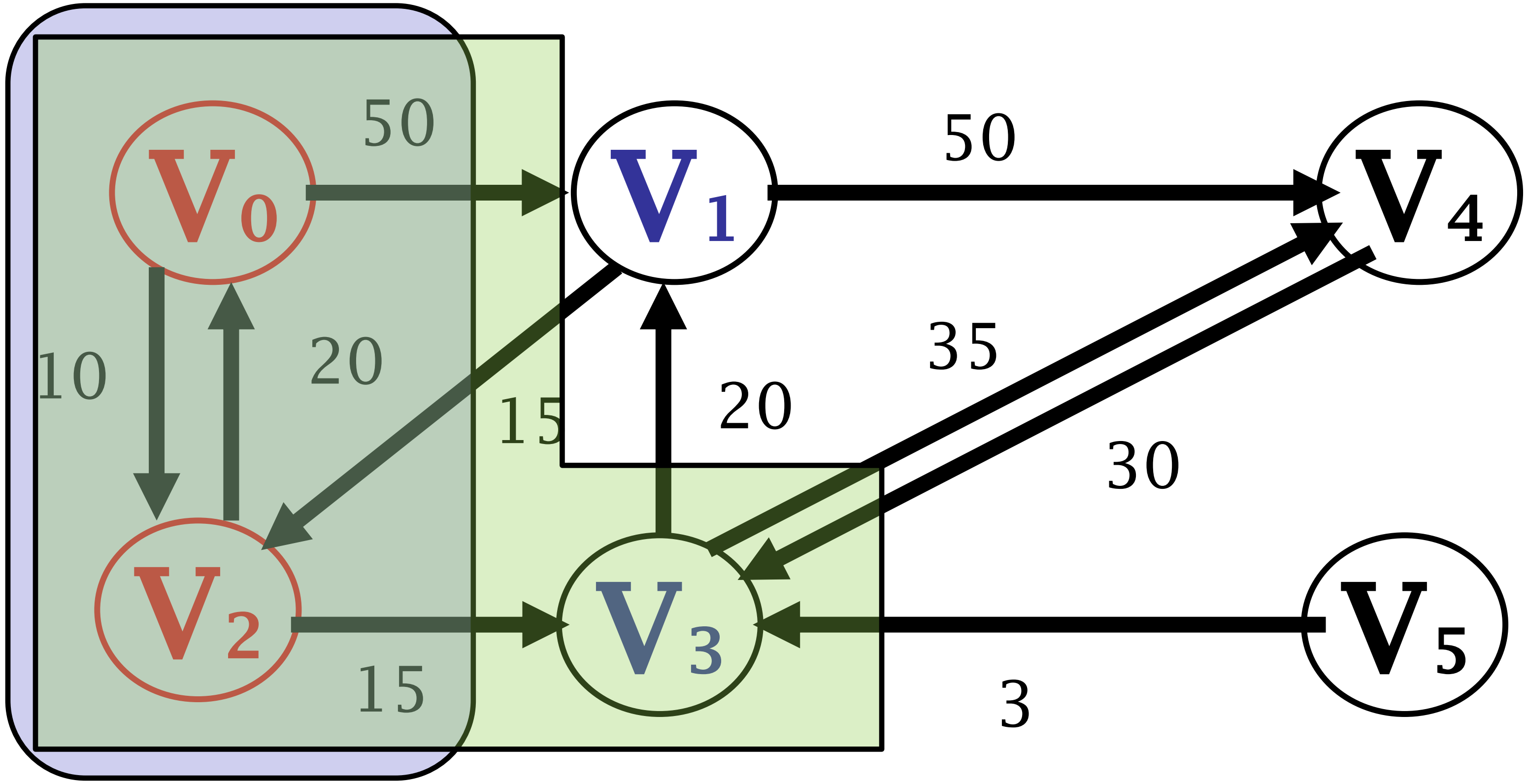
	V_0	V_1	V_2	V_3	V_4	V_5
初始状态	0 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0



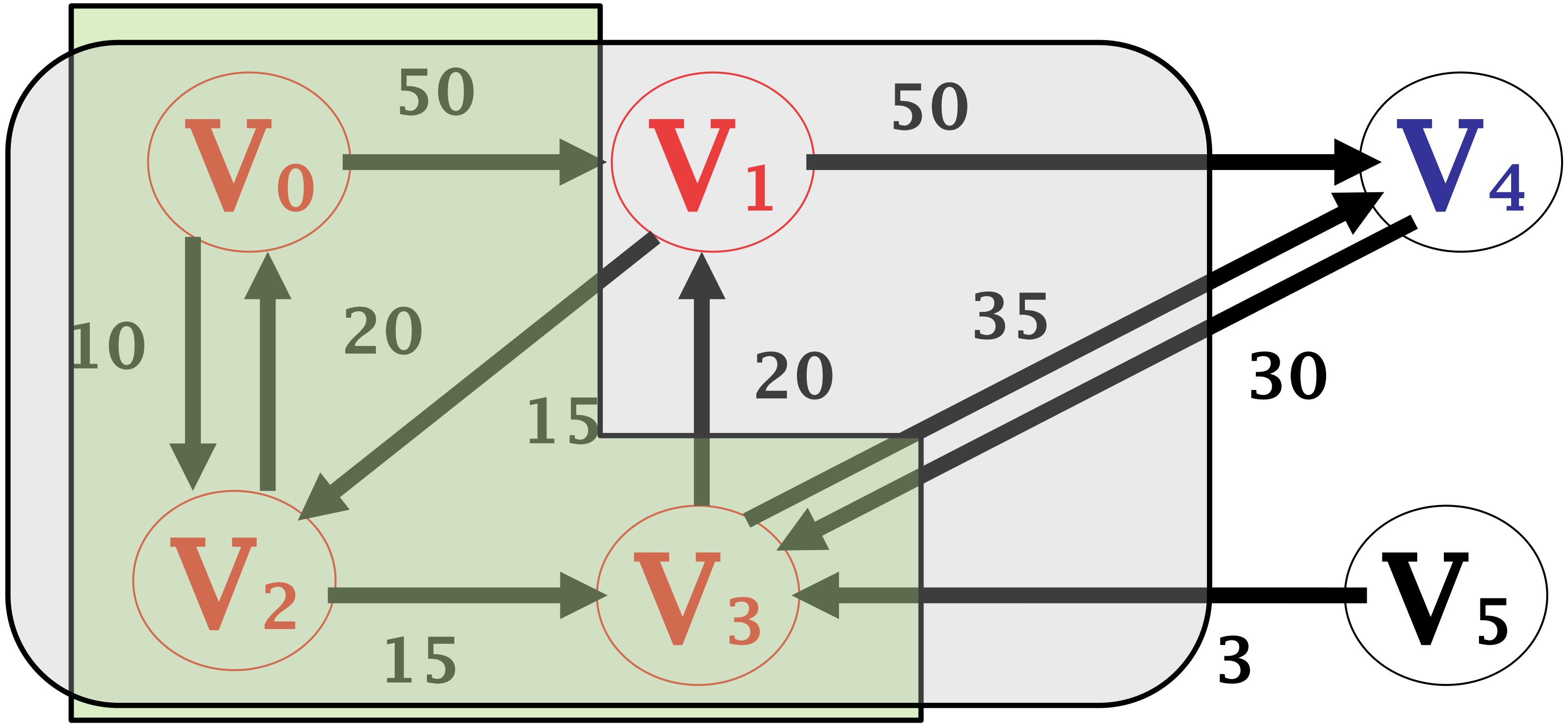
	V_0	V_1	V_2	V_3	V_4	V_5
初始	0 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0
V_0进入第一组	0 Pre:0	50 Pre:0	10 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0



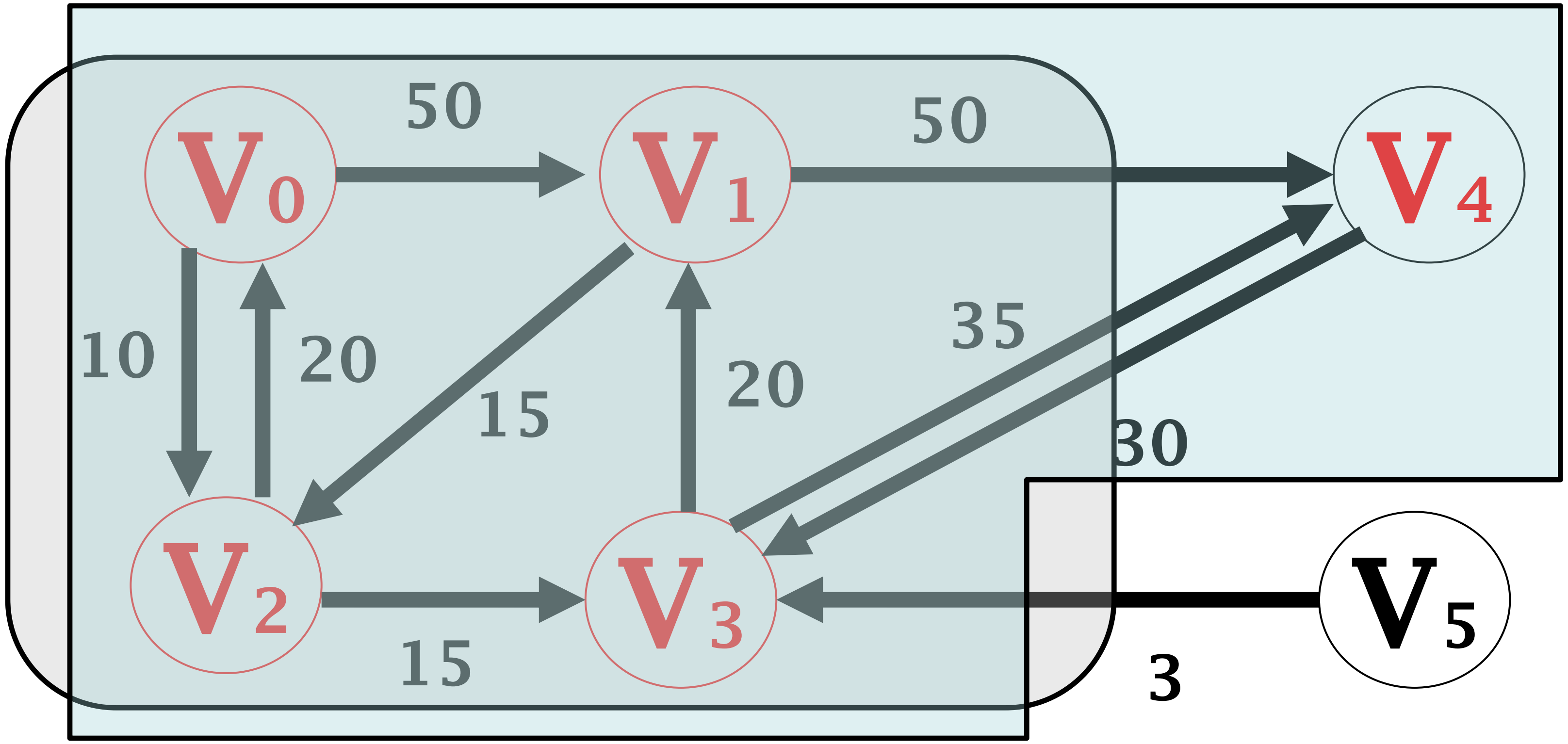
	V_0	V_1	V_2	V_3	V_4	V_5
V_2进入之前	0 Pre:0	50 Pre:0	10 Pre:0	∞ Pre:0	∞ Pre:0	∞ Pre:0
V_2进入第一组	0 Pre:0	50 Pre:0	10 Pre:0	25 Pre:2	∞ Pre:0	∞ Pre:0



	V_0	V_1	V_2	V_3	V_4	V_5
V_3进入之前	0 Pre:0	50 Pre:0	10 Pre:0	25 Pre:2	∞ Pre:0	∞ Pre:0
V_3进入第一组	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0



	V_0	V_1	V_2	V_3	V_4	V_5
V_1进入之前	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0
V_1进入第一组	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0



	V_0	V_1	V_2	V_3	V_4	V_5
V_4进入之前	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0
V_4进入第一组	0 Pre:0	45 Pre:3	10 Pre:0	25 Pre:2	60 Pre:3	∞ Pre:0

Dijkstra单源最短路径迭代过程

步数	S	v_0	v_1	v_2	v_3	v_4
初始	$\{v_0\}$	Length:0 pre:0	length:<u>50</u> pre:0	length:<u>10</u> pre:0	length:∞ pre:0	length:∞ pre:0
1	$\{v_0, v_2\}$	Length:0 pre:0	length:50 pre:0	length:10 pre:0	length:<u>25</u> pre:2	length:∞ pre:0
2	$\{v_0, v_2, v_3\}$	Length:0 pre:0	length: <u>45</u> pre:3	length:10 pre:0	length:25 pre:2	length:<u>60</u> pre:3
3	$\{v_0, v_2, v_3, v_1\}$	Length:0 pre:0	length: 45 pre:3	length:10 pre:0	length:25 pre:2	length:60 pre:3
4	$\{v_0, v_2, v_3, v_1, v_4\}$	Length:0 pre:0	length: 45 pre:3	length:10 pre:0	length:25 pre:2	length:60 pre:3

Dijkstra单源最短路径算法

```
class Dist {                    // Dist类，用于保存最短路径信息
public:
    int index;                  // 结点的索引值，仅Dijkstra算法用到
    int length;                 // 当前最短路径长度
    int pre;                    // 路径最后经过的结点
};
void Dijkstra(Graph& G, int s, Dist* &D) {        // s是源点
    D = new Dist[G. VerticesNum()];               // 记录当前最短路径长度
    for (int i = 0; i < G.VerticesNum(); i++) {   // 初始化
        G.Mark[i] = UNVISITED;
        D[i].index = i;  D[i].length = INFINITE;  D[i].pre = s;
    }
    D[s].length = 0;                    // 源点到自身的路径长度置为0
    MinHeap<Dist> H(G. EdgesNum());     // 最小值堆用于找出最短路径
    H.Insert(D[s]);
```

7.5 最短路径

```
for (i = 0; i < G.VerticesNum(); i++) {
    bool FOUND = false;
    Dist d;
    while (!H.isEmpty())  {
        d = H.RemoveMin();                          //获得到s路径长度最小的结点
        if (G.Mark[d.index] == UNVISITED) {   //如果未访问过则跳出循环
            FOUND = true;   break;
    } }
    if  (!FOUND)  break;         // 若没有符合条件的最短路径则跳出本次循环
    int v = d.index;
    G.Mark[v] = VISITED;         // 将标记位设置为 VISITED
    for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e)) // 刷新最短路
        if (D[G.ToVertex(e)].length > (D[v].length+G.Weight(e))) {
            D[G.ToVertex(e)].length = D[v].length + G.Weight(e);
            D[G.ToVertex(e)].pre = v;
            H.Insert(D[G.ToVertex(e)]);
    } }
}
```

Dijkstra算法时间代价分析

- 每次改变D[i].length
  - 不删除，添加一个新值(更小的)，作为堆中新元素。旧值被找到时，该结点一定被标记为VISITED，从而被忽略
- 在最差情况下，它将使堆中元素数目由$\Theta(|V|)$增加到$\Theta(|E|)$，总的时间代价$\Theta((|V|+|E|) \log|E|)$

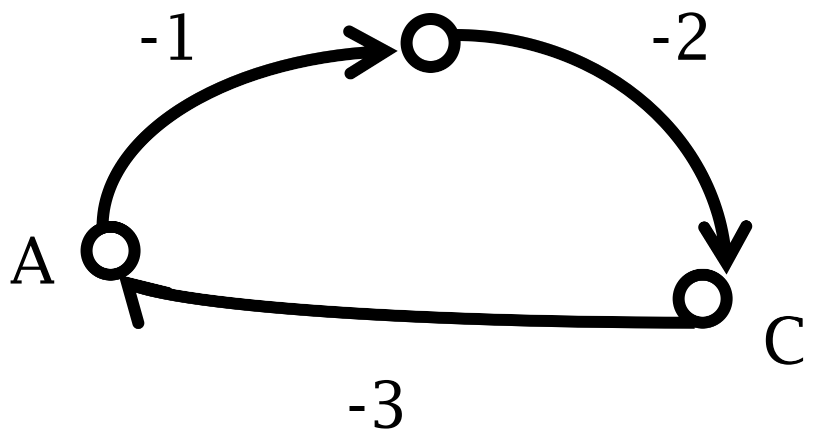
Dijkstra算法

- 是否支持
  - 有向图、无向图
  - 非连通
  - 有回路的图
  - 权值为负
- 如果不支持
  - 则修改方案？

- 针对有向图 (且“有源”)
  - 若输入无向图？
  - 照样能够处理 (边都双向)
- 对非连通图，有不可达
  - 没有必要修改
- 支持回路
- 支持负权值？

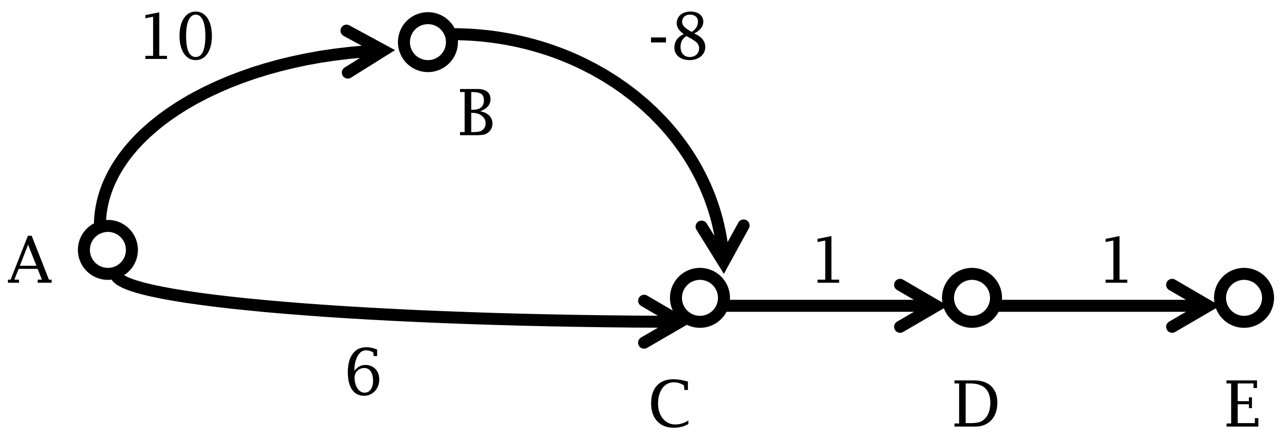
Dijkstra算法不支持负权值

- 如果存在总权值为负的回路，则将出现权值为 -∞ 的情况



如果不存在负权回路呢？
Dijkstra算法不受负权边的影响吗？

- 即使不存在负的回路，也可能有在后面出现的负权值，从而导致整体计算错误
- 主要原因是 Dijkstra 算法是贪心法，当作最小取进来后，不会返回去重新计算



7.5 最短路径

- 持负权值的最短路径算法
  - Bellman - Ford 算法
    - 参考书 MIT “Introduction to Algorithms”
  - SPFA 算法

每对结点间的最短路径

- 还能用 Dijkstra 算法吗？
- 以每个结点为起点，调用 n 次 Dijkstra 算法

```
void Dijkstra_P2P(Graph& G) {
        Dist **D=new Dist *[G.VerticesNum()];
        for(i=0; i<G.VerticesNum(); i++)
                Dijkstra(Graph& G, i, D[i]);
}
```

Floyd算法求每对结点之间的最短路径

- 用相邻矩阵 adj 来表示带权有向图
- 基本思想：
  - 初始化 $adj^{(0)}$ 为相邻矩阵 adj
  - 在矩阵 $adj^{(0)}$上做 n 次迭代，递归地产生一个矩阵序列$adj^{(1)}$，… ，$adj^{(k)}$，… ，$adj^{(n)}$
  - 其中经过第k次迭代，$adj^{(k)}[i，j]$ 的值等于从结点 v_i 到结点 v_j 路径上所经过的结点序号不大于 k 的最短路径长度
- 动态规划法

最短路径组合情况分析

由于第 k 次迭代时已求得矩阵 $adj^{(k-1)}$，那么从结点 v_i 到 v_j 中间结点的序号不大于 k 的最短路径有两种情况：、

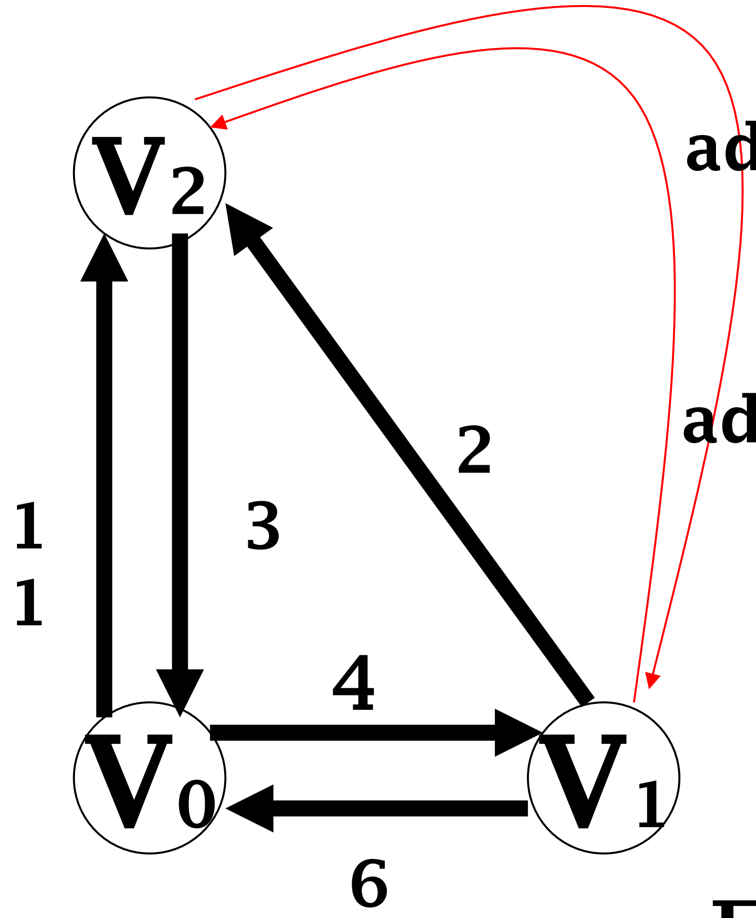
- 一种是中间不经过结点 v_k，那么此时就有

$$adj^{(k)}[i, j] = adj^{(k-1)}[i, j]$$

- 另中间经过结点 v_k，此时 $adj^{(k)}[i, j] < adj^{(k-1)}[i, j]$，那么这条由结点 v_i 经过 v_k 到结点 v_j 的中间结点序号不大于k的最短路径由两段组成

$$adj^{(k)}[i, j] = adj^{(k-1)}[i, k] + adj^{(k-1)}[k, j]$$

7.5 最短路径



$$adj = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & -1 & 2 \end{pmatrix}$$

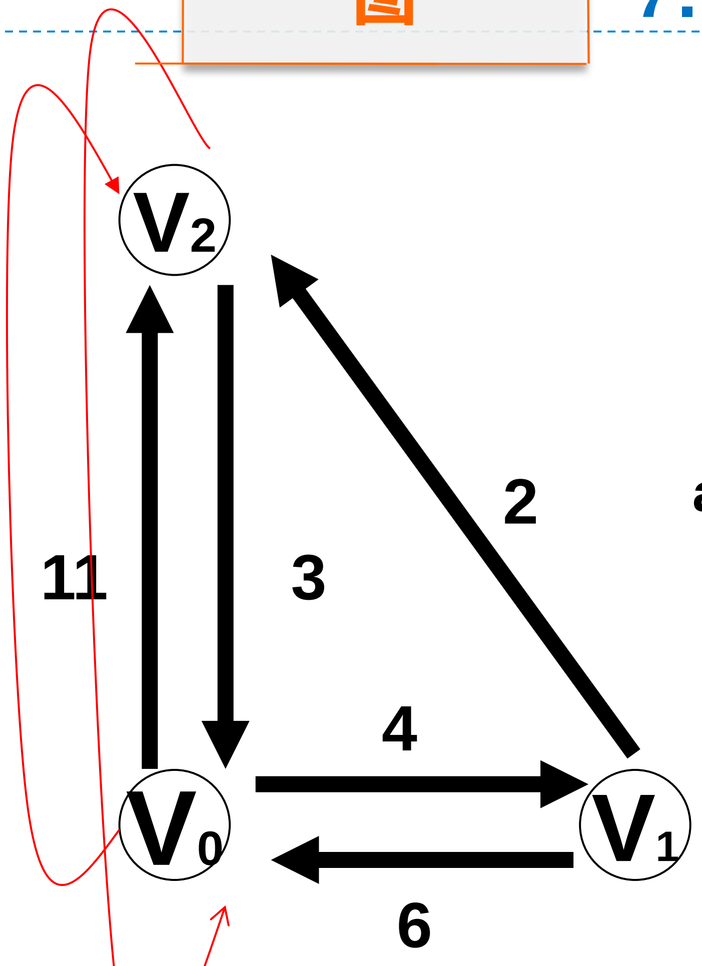
$$adj^{(0)} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$2 < 6+11$

$\infty > 3+4$

Floyd算法演示动画

7.5 最短路径



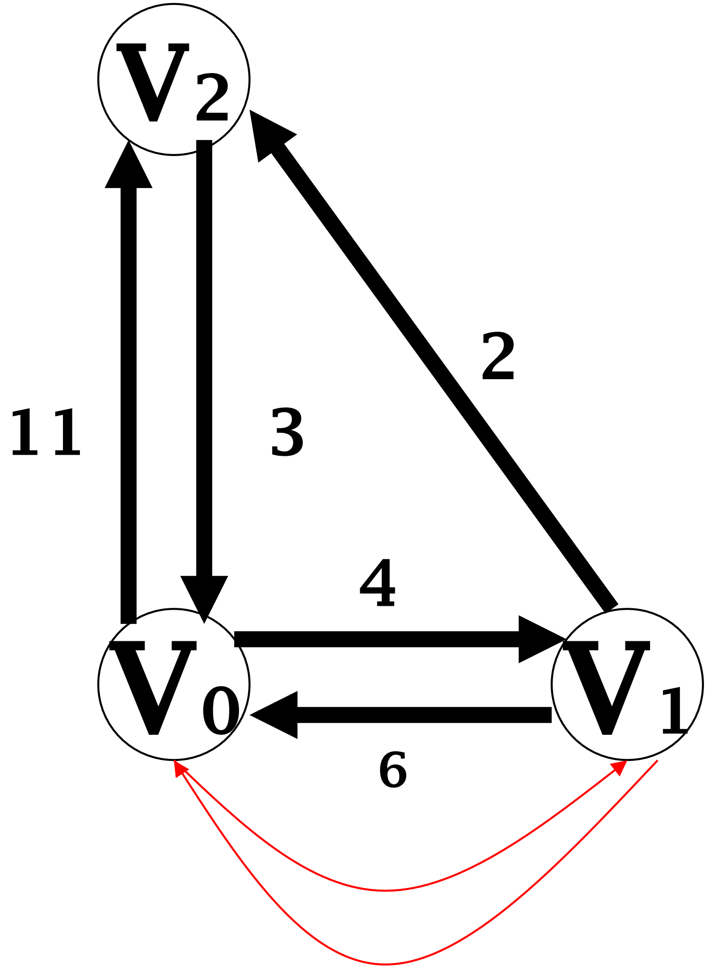
$$adj = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \boxed{\infty} & 0 \end{pmatrix} \qquad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & -1 & 2 \end{pmatrix}$$

$$adj^{(0)} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \qquad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$$adj^{(1)} = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \qquad Path = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$11 > 4+2$

$3 < \infty(V_2V_1, V_1V_0)$



$4 < \infty(V_0V_2,\ V_2V_1)$

$6 > 2+3$

$$adj = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \boxed{\infty} & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 2 & -1 & 2 \end{pmatrix}$$

$$adj^{(0)} = \begin{pmatrix} 0 & 4 & \boxed{11} \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$$adj^{(1)} = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 2 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

$$adj^{(2)} = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \quad Path = \begin{pmatrix} 0 & 0 & 1 \\ 2 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

每对结点之间最短路径的Floyd算法

```
void Floyd(Graph& G, Dist** &D)  {
    int i,j,v;
    D = new Dist*[G.VerticesNum()];              // 申请空间
    for (i = 0; i < G.VerticesNum(); i++)
        D[i] = new Dist[G.VerticesNum()];
    for (i = 0; i < G.VerticesNum(); i++)        // 初始化数组D
        for (j = 0; j < G.VerticesNum(); j++) {
            if (i == j) {
                D[i][j].length = 0;
                D[i][j].pre = i;
            } else {
                D[i][j].length = INFINITE;
                D[i][j].pre = -1;
            }
        }
```

7.5 最短路径

```
for (v = 0; v < G.VerticesNum(); v++)
    for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e)) {
        D[v][G.ToVertex(e)].length = G.Weight(e);
        D[v][G.ToVertex(e)].pre = v;
    }
// 加入新结点后，更新那些变短的路径长度
for (v = 0; v < G.VerticesNum(); v++)
    for (i = 0; i < G.VerticesNum(); i++)
        for (j = 0; j < G.VerticesNum(); j++)
            if (D[i][j].length > (D[i][v].length+D[v][j].length)) {
                D[i][j].length = D[i][v].length+D[v][j].length;
                D[i][j].pre = D[v][j].pre;
            }
}
```

Floyd算法的时间复杂度

- 三重for循环
  - 复杂度是$\Theta(n^3)$

讨论：Dijkstra 找最小 Dist 值

- 如果不采用最小堆，而采用每次遍历的方式寻找最小值，与用最小堆实现的 Dijkstra 相比，时间效率如何？

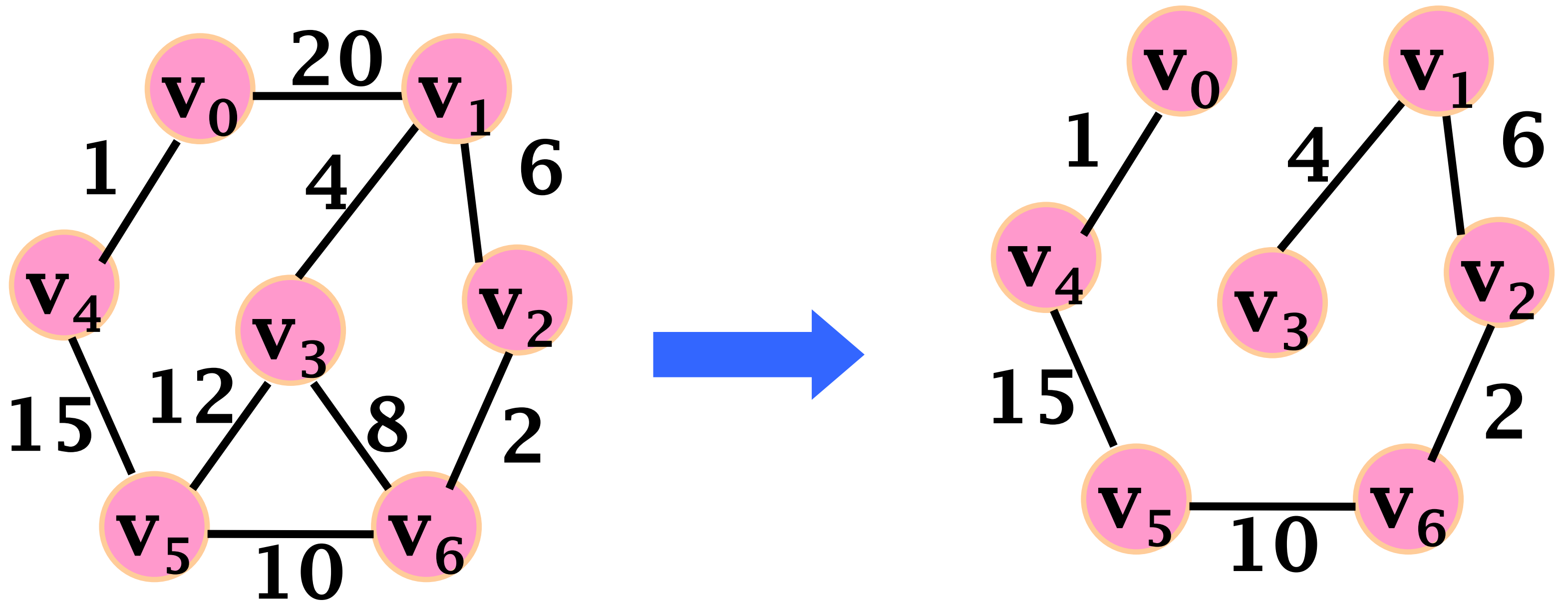
讨论：Floyd算法保持 pre 的方式

- 将“D[i][j].pre= D[v][j].pre” 改为 “ D[i][j].pre = v” 是否可以？
  - 上述两种方案不影响 D[i][j].length 的求解
  - 对于恢复最短路径，策略有何不同？那种更优？

第7章　图

7.6 最小生成树

- 图 G 的生成树是一棵包含 G 的所有顶点的树，树上所有权值总和表示代价，那么在 G 的所有的生成树中
- 代价最小的生成树称为图 G 的 **最小生成树** **(minimum-cost spanning tree**，简称 **MST)**

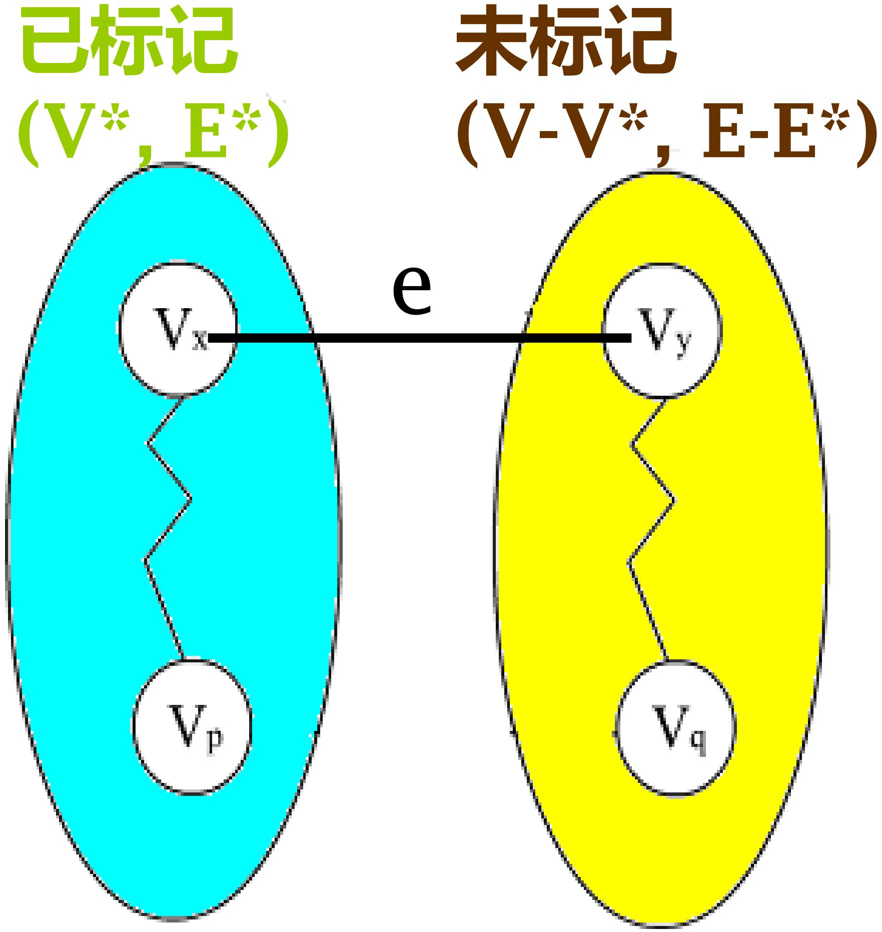


7.6.1 Prim 算法

- 与 Dijkstra 算法类似——也是贪心法
  - 从图中任意一个顶点开始 (例如 v_0)，首先把这个顶点包括在 MST，U = (V*, E*) 里
    - 初始 V*= {v_0}，E* = {}
  - 然后在那些其一个端点已在 MST 里，另一个端点**还不是 MST里的边**，找权最小的一条边 (v_p, v_q) ，并把此v_q 包括进 MST 里……
  - 如此进行下去，每次往 MST 里加一个顶点和一条权最小的边，直到把所有的顶点都包括进 MST 里
- 算法结束时 V* = V，E* 包含了 G 中的 n-1 条边

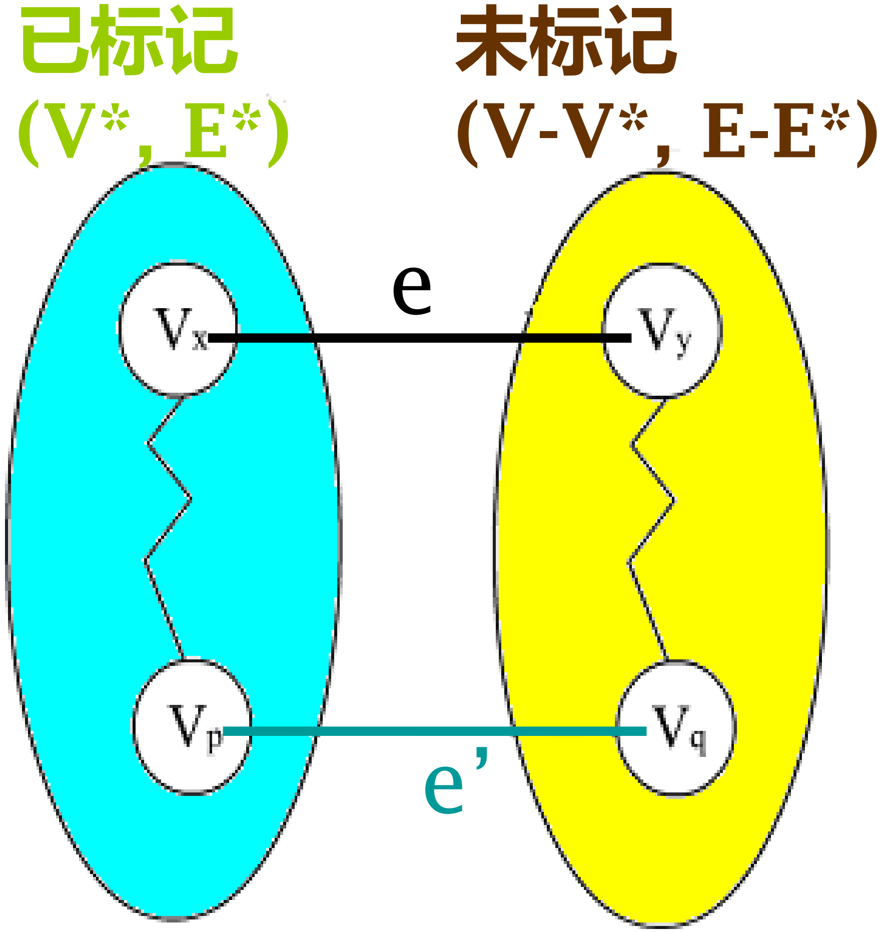
Prim 算法的 MST 性质

- 设 T(V*，E*) 是一棵正在构造的生成树
- E 中有边 e= (v_x，v_y)，其中v_x∈v*，v_y不属于V*
  - e 的权 w(e) 是所有一个端点在V* 里，另一端不在V* 里的边的权中最小者
- 则一定存在 G 的一棵包括 T 的 MST 包括边 e= (v_x，v_y)



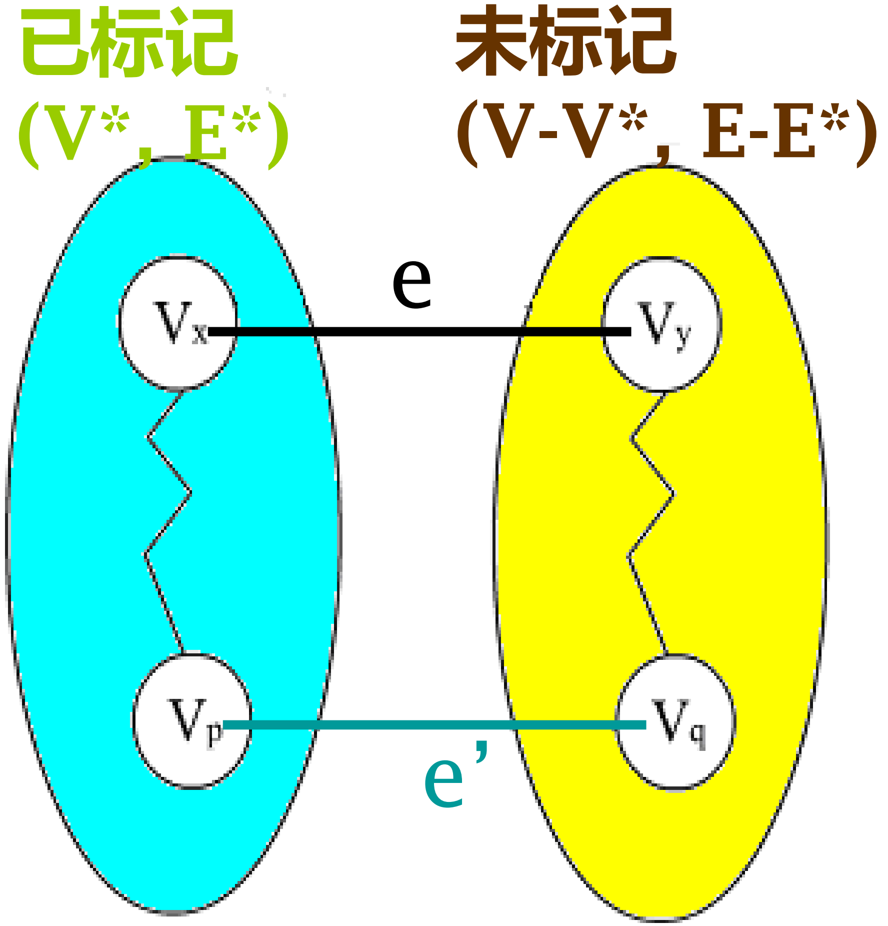
反证法证明 MST 性质

- 用反证法
  - 设G的任何一棵包括 T 的 MST 都不包括 e= (v_x , v_y) ，且设 T’ 是一棵这样的 MST
  - 由于 T’ 是连通的，因此有从 v_x 到 v_y 的路径 v_x , … , v_y



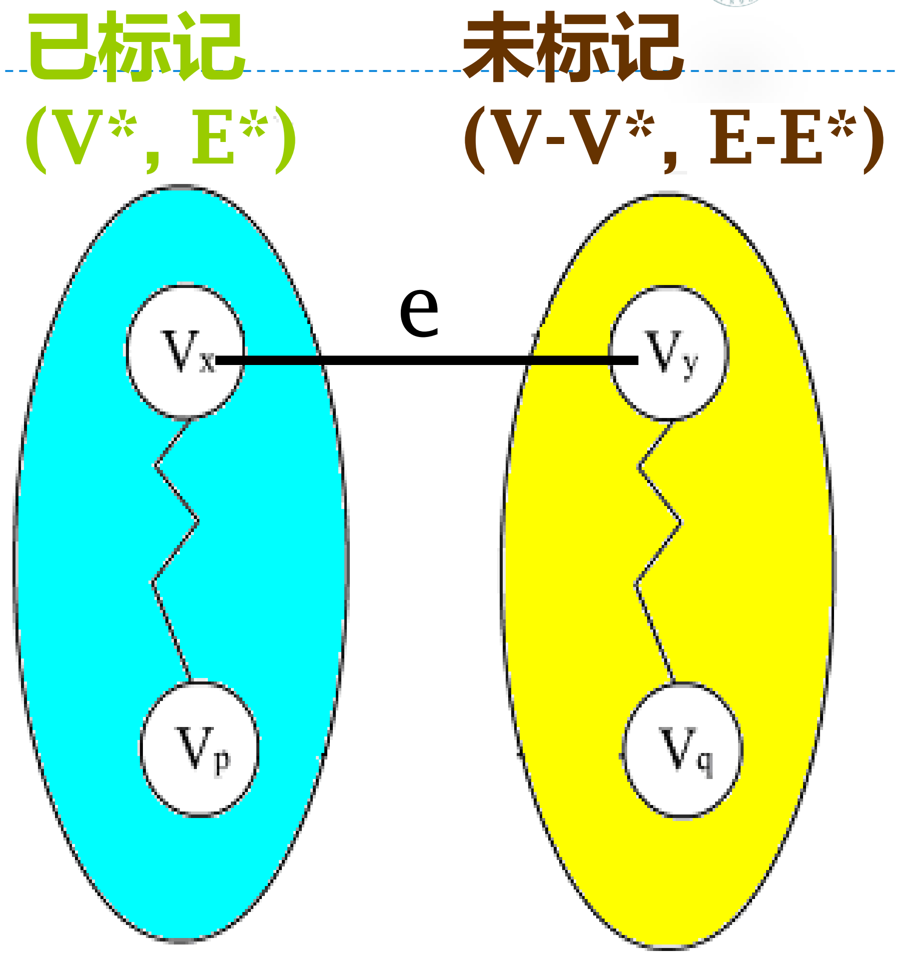
反证法证明 MST 性质 (续)

- 把边 e= (V_x , V_y) 加进树 T’，得到一个回路 v_x , … , v_y , v_x
- 上述路径 v_x , … , v_y 中必有边 e’= (v_p , v_q)，其中$v_p \in V^*$, v_q 不属于 V*，由条件知边的权 w(e’)≥w(e)，从回路中去掉边 e’
- 回路打开，成为另一棵生成树 T”，T” 包括边 e= (v_x , v_y)，且各边权的总和不大于 T’ 各边权的总和



反证法证明 MST 性质 (续)

- 因此 T” 是一棵包括边 e 的 MST，与假设矛盾，即证明了我们的结论
- 也证明了 Prim 算法构造 MST 的方法是正确的
  - 因为我们是从 T 包括任意一个顶点和0条边开始，每一步加进去的都是 MST 中应包括的边，直至最后得到 MST



Prim 算法

```
void Prim(Graph& G, int s, Edge* &MST) {          // s是始点，MST存边
    int MSTtag = 0;                                // 最小生成树的边计数
    MST = new Edge[G.VerticesNum()-1];             // 为数组MST申请空间
    Dist *D;
    D = new Dist[G. VerticesNum()];                // 为数组D申请空间
    for (int i = 0; i < G.VerticesNum(); i++) {    // 初始化Mark和D数组
        G.Mark[i] = UNVISITED;
        D[i].index = i;
        D[i].length = INFINITE;
        D[i].pre = s;                              // D[i].pre = -1 呢？
    }
    D[s].length = 0;
    G.Mark[s]= VISITED;                            // 开始顶点标记为VISITED
    int v = s;
```

```
for (i = 0; i < G.VerticesNum()-1; i++) {
    // 因为v的加入，需要刷新与v相邻接的顶点的D值
    for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e))
        if (G.Mark[G.ToVertex(e)] != VISITED &&
        (D[G.ToVertex(e)].length > e.weight)) {
            D[G.ToVertex(e)].length = e.weight;
            D[G.ToVertex(e)].pre = v;
        }
    v = minVertex(G, D);                    // 在D数组中找最小值记为v
    if (v == -1) return;                    // 非连通，有不可达顶点
    G.Mark[v] = VISITED;                    // 标记访问过
    Edge edge(D[v].pre, D[v].index, D[v].length); // 保存边
    AddEdgetoMST(edge, MST, MSTtag++);          // 将边加入MST
  }
}
```

在 Dist 数组中找最小值

```
int minVertex(Graph& G, Dist* & D)  {
    int i, v = -1;
    int MinDist = INFINITY;
    for (i = 0; i < G.VerticesNum(); i++)
        if ((G.Mark[i] == UNVISITED) && (D[i] < MinDist)){
            v = i;          // 保存当前发现的最小距离顶点
            MinDist = D[i];
        }
    return v;
}
```

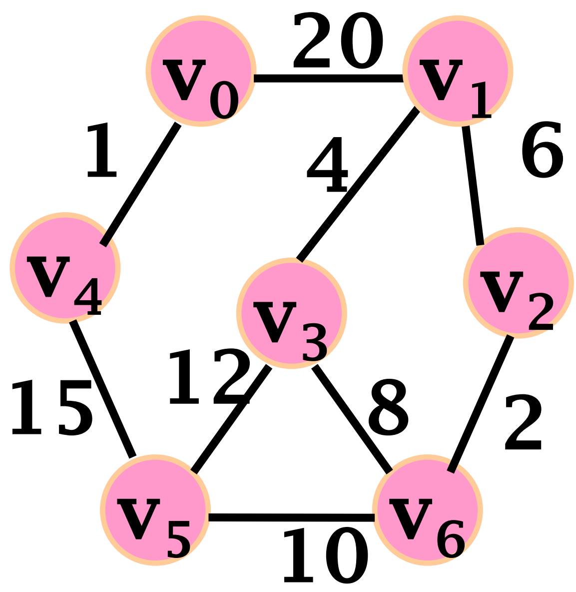
Prim 算法时间复杂度

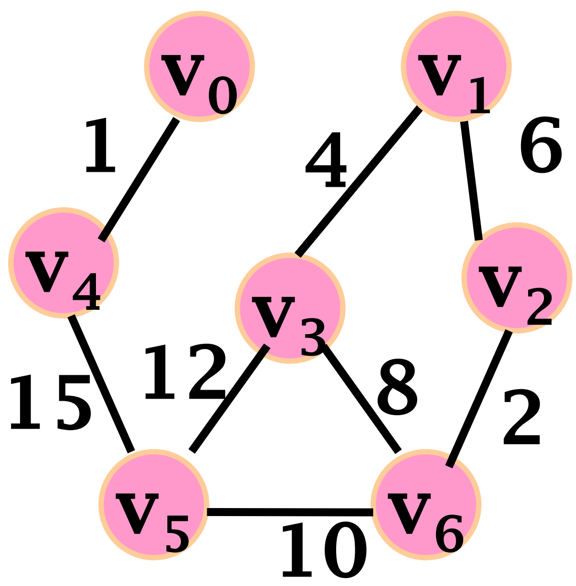
- Prim 算法非常类似于 Dijkstra 算法
  - Prim 算法框架与 Dijkstra 算法相同
  - Prim 算法中的距离值不需要累积，直接用最小边
- 本算法通过直接比较 D 数组元素，确定代价最小的边就需要总时间$O(n^2)$；取出权最小的顶点后，修改 D 数组共需要时间$O(e)$，因此共需要花费$O(n^2)$ 的时间
  - 算法适合于稠密图
  - 对于稀疏图，可以像 Dijkstra 算法那样用堆来保存距离值

7.6.2 Kruskal 算法

- 首先将 G 中的 n 个顶点看成是独立的 n 个连通分量，这时的状态是有 n 个顶点而无边的森林，可以记为 T = <V，{}>
- 然后在 E 中选择代价最小的边，如果该边依附于两个不同的连通分支，那么将这条边加入到 T 中，否则舍去这条边而选择下一条代价最小的边
- 依此类推，直到 T 中所有顶点都在同一个连通分量中为止，此时就得到图 G 的一棵最小生成树

最小生成树 Kruskal 算法





Kruskal 最小生成树算法

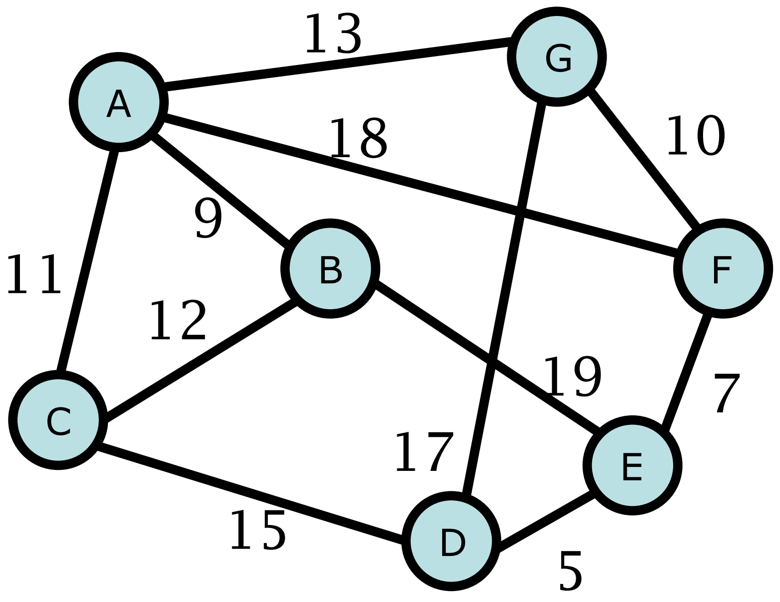
```
void Kruskal(Graph& G, Edge* &MST)  {      // MST存最小生成树的边
    ParTree<int> A(G.VerticesNum());       // 等价类
    MinHeap<Edge> H(G.EdgesNum());         // 最小堆
    MST = new Edge[G.VerticesNum()-1];     // 为数组MST申请空间
    int MSTtag = 0;                        // 最小生成树的边计数
    for (int i = 0; i < G.VerticesNum(); i++)  // 将所有边插入最小堆H中
        for (Edge e = G. FirstEdge(i); G.IsEdge(e); e = G. NextEdge(e))
            if (G.FromVertex(e) < G.ToVertex(e))// 防重复边
                H.Insert(e);
    int EquNum = G.VerticesNum();          // 开始有n个独立顶点等价类
```

7.6 最小生成树

```
while (EquNum > 1) {                    // 当等价类的个数大于1时合并等价类
    if (H.isEmpty()) {
        cout << "不存在最小生成树." <<endl;
        delete [] MST;
        MST = NULL;                     // 释放空间
        return;
    }
    Edge e = H.RemoveMin();             // 取权最小的边
    int from = G.FromVertex(e);         // 记录该条边的信息
    int to = G.ToVertex(e);
    if (A.Different(from,to)) {         // 边e的两个顶点不在一个等价类
        A.Union(from,to);               // 合并边的两个顶点所在的等价类
        AddEdgetoMST(e,MST,MSTtag++); // 将边e加到MST
        EquNum--;                       // 等价类的个数减1
    }
  }
}
```

7.6 最小生成树

算法演示



~~1~~	~~4~~	~~1~~	~~4~~		~~4~~	~~5~~4
A	B	C	D	E	F	G
0	1	2	3	4	5	6

- (D,E,5)
- (F,E,7)
- (A,B,9)
- (F,G,10)
- (A,C,11)
- (B,C,12)
- (A,G,13)
- (C,D,15)

…………

Kruskal 算法代价

- 使用了路径压缩，Different() 和 Union() 函数几乎是常数
- 假设可能对几乎所有边都判断过了
  - 则最坏情况下算法时间代价为 Θ (elog e)，即堆排序的时间
- 通常情况下只找了略多于 n 次，MST 就已经生成
  - 时间代价接近于 Θ (nlog e)

讨论

- 最小生成树是否唯一？
  - 不一定
  - 试设计算法生成所有的最小生成树
- 如果边的权都不相等
  - 一定是唯一的

数据结构与算法

谢谢聆听

国家精品课 “数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6。 “十一五”国家级规划教材