数据结构与算法
第六章　图

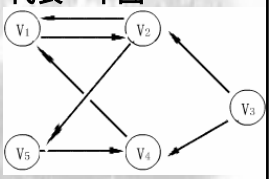
任课教员：张 铭
http://db.pku.edu.cn/mzhang/DS/
mzhang@db.pku.edu.cn
北京大学信息科学与技术学院
网络与信息系统研究所

主要内容

- 6.1 图的基本概念
- 6.2 图的抽象数据类型
- 6.3 图的存储结构
- 6.4 图的周游（深度、广度、拓扑）
- 6.5 最短路径问题
- 6.6 最小支撑树

6.1 图的基本概念

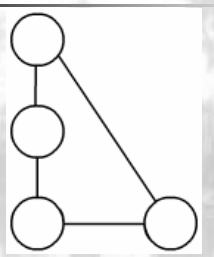
- 习惯上，常用G=（V，E）代表一个图
  - V是顶点（vertex）集合
  - E是边（edge）的集合
    - 边的始点
    - 边的终点
- 稀疏图（sparse graph）
- 密集图（dense graph）
- 完全图（complete graph）

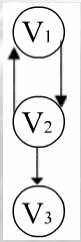


6.1 图的基本概念（续）

- 无向图（undirected graph）
  - 图中代表一条边的顶点的偶对无方向性，也即无序
- 有向图（directed graph或digraph）
  - 图中代表一条边的顶点的偶对是有序的

6.1 图的基本概念（续）

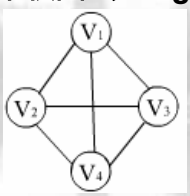


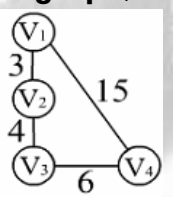


无向图示例　有向图示例

6.1 图的基本概念（续）

- 标号图（labeled graph）
- 带权图（weighted graph）





6.1 图的基本概念（续）

- 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

6.1 图的基本概念（续）

- 路径（path）
  - 在图G=(V，E)中，如果存在顶点序列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，则称从顶点V_p到顶点V_q存在一条路径。
  - 简单路径（simple path）
  - 路径长度（length）

6.1 图的基本概念（续）

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

6.1 图的基本概念（续）

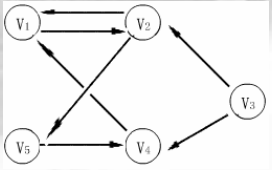
- 有根的图
  - 一个有向图中，若存在一个顶点V_0，从此顶点有路径可以到达图中其它所有顶点，则称此有向图为有根的图，V_0称作图的根。
  - 树、森林
- 连通的（connected）
  - 对无向图G=（V，E）而言，如果从V_1到V_2有一条路径（从V_2到V_1也一定有一条路径），则称V_1和V_2是连通的（connected）。
  - 强连通

6.1 图的基本概念（续）

- 连通分支或者连通分量（connected component）
  - 无向图的最大连通子图
  - 强连通分支（强连通分量）
- 网络
  - 带权的连通图
- 自由树（free tree）
  - 不带有简单回路的无向图，它是连通的，并且具有|V|-1条边

6.2 图的抽象数据类型

- 结点、边怎么表示？
- 结点：增？、删、查?、改
- 边：增、删、查?、改



6.2 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

 //返回与顶点oneVertex相关联的第一条边
 Edge FirstEdge(int oneVertex);

 //返回与边PreEdge有相同关联顶点oneVertex的
 //下一条边
 Edge NextEdge(Edge preEdge);
```



```
 //添加一条边
 bool setEdge(int fromVertex,int toVertex,int
    weight);
 //删一条边
 bool delEdge(int fromVertex,int toVertex);
 //如果oneEdge是边则返回TRUE，否则返回FALSE
 bool IsEdge(Edge oneEdge);
 //返回边oneEdge的始点
 int FromVertex(Edge oneEdge);
 //返回边oneEdge的终点
 int ToVertex(Edge oneEdge);
 //返回边oneEdge的权
 int Weight(Edge oneEdge);
};
```


6.3 图的存储结构

- 6.3.1 图的相邻矩阵（adjacency matrix）表示法
- 6.3.2 图的邻接表（adjacency list）表示法
  - 邻接多重表(adjacency multilist)表示法

6.3.1 图的相邻矩阵（adjacency matrix）表示法

- 相邻矩阵
  - 表示顶点间相邻关系的矩阵。
  - 若G是一个具有n个顶点的图，则G的相邻矩阵是如下定义的n×n矩阵：

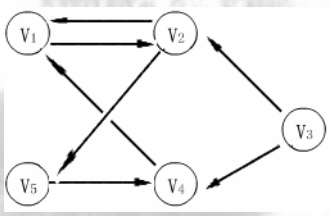
A[i,j]=1，若（V_i，V_j）（或<V_i，V_j>）是图G的边；

A[i,j]=0，若（V_i，V_j）（或<V_i，V_j>）不是图G的边。

- 相邻矩阵的空间代价为Θ(n^2)

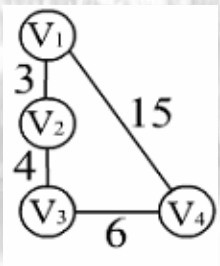
6.3.1 图的相邻矩阵（adjacency matrix）表示法（续）

$$A7=\begin{bmatrix}0&1&0&0&0\\1&0&0&0&1\\0&1&0&1&0\\1&0&0&0&0\\0&0&0&1&0\end{bmatrix}$$

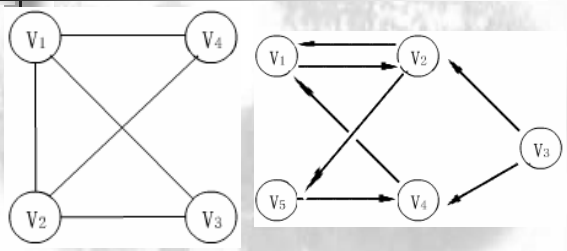


6.3.1 图的相邻矩阵（adjacency matrix）表示法（续）

$$A4=\begin{bmatrix}0&3&0&15\\3&0&4&0\\0&4&0&6\\15&0&6&0\end{bmatrix}$$

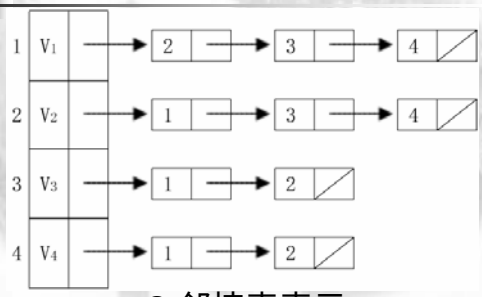


6.3.2 图的邻接表（adjacency list）表示法



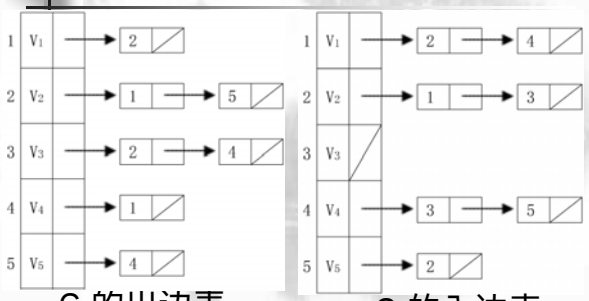
G_6 G_7

6.3.2 图的邻接表（adjacency list）表示法（续）



G_6邻接表表示

6.3.2 图的邻接表（adjacency list）表示法（续）

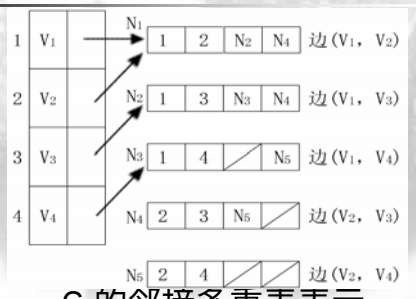


G_7的出边表 G_7的入边表

邻接多重表(adjacency multilist)

- 把邻接表表示中代表同一条边的两个表目合为一个表目
- 图的每条边只用一个多重表表目表示
  - 包括此边的两个顶点的序号
  - 两个指针（一个指针指向与第一个顶点相关联的下一条边，另一个指针指向与第二个顶点相关联的下一条边）
- 在以处理图的边为主，要求每条边处理一次的实际应用中特别有用

邻接多重表(adjacency multilist)

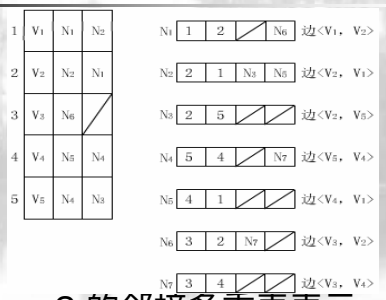


G_6的邻接多重表表示

有向图邻接多重表(adjacency multilist)

- 在顶点表中设计两个指针
  - 第一个指向以此顶点为始点的第一条边
  - 第二个指向以此顶点为终点的第一条边
- 边表
  - 第一个指针指向始点与本边始点相同的下一条边
  - 第二个指针指向终点与本边终点相同的下一条边
- 故仅用表中第一个链便得到有向图的出边表，仅用第二个链便得到有向图的入边表

邻接多重表(adjacency multilist)（续）



G_7的邻接多重表表示

6.3.2 图的邻接表（adjacency list）表示法（续）

- n个顶点m条边的无向图
  - 需用(n+2m)个存储单元
- n个顶点m条边的有向图
  - 需用(n+m)个存储单元

6.4 图的周游

- 图的周游（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，每个顶点访问一次，这叫做图的周游

6.4 图的周游（续）

- 图的周游的典型算法
  - 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
    - 为图的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；
    - 在周游的过程中，当某个顶点被访问时，其标志位就被标记为已访问。

6.4 图的周游（续）

```
//图的周游算法的实现
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周游
  //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);
}
```

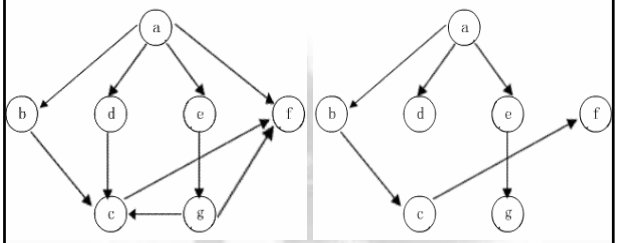

6.4 图的周游（续）

- 图的生成树
  - 图的所有顶点加上周游过程中经过的边所构成的子图称作图的生成树
- 图的生成森林

6.4.1 深度优先搜索

- 深度优先搜索（depth-first search，简称DFS）基本思想
  - 访问一个顶点V，然后访问该顶点邻接到的未被访问过的顶点V’
  - 再从V’出发递归地按照深度优先的方式周游
  - 当遇到一个所有邻接于它的顶点都被访问过了的顶点U时，则回到已访问顶点序列中最后一个拥有未被访问的相邻顶点的顶点W
  - 再从W出发递归地按照深度优先的方式周游
  - 最后，当任何已被访问过的顶点都没有未被访问的相邻顶点时，则周游结束
- 深度优先搜索树（depth-first search tree）

6.4.1 深度优先搜索（续）



深度优先搜索的顺序是a，b，c，f，d，e，g

6.4.1 深度优先搜索（续）

```
void DFS(Graph& G, int V){      //深度优先搜索算法实现
   G.Mark[V]= VISITED;  //访问顶点V，并标记其标志位
   PreVisit(G, V);             //访问V
   for(Edge e=G. FirstEdge(V); G.IsEdge(e);
        e=G. NextEdge(e))
      //访问V邻接到的未被访问过的顶点，并递归地按照
      //深度优先的方式进行周游
      if(G.Mark[G. ToVertices(e)]== UNVISITED)
         DFS(G, G. ToVertices(e));
   PostVisit(G, V);            //访问V
}
```


6.4.1 深度优先搜索（续）

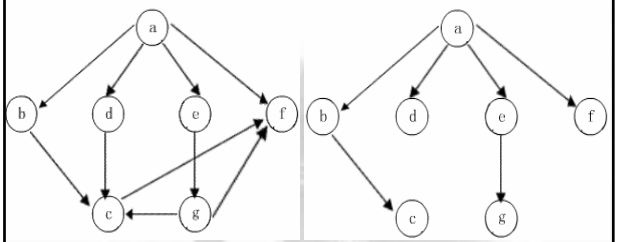
- 深度优先搜索算法的时间复杂度

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

6.4.2 广度优先搜索

- 广度优先搜索（breadth-first search，简称BFS）
  - 它的基本思想是访问顶点V_0，
  - 然后访问V_0邻接到的所有未被访问过的顶点V_{01}，V_{02}，…V_{0i}，
  - 再依次访问V_{01}，V_{02}，…V_{0i}邻接到的所有未被访问的顶点，
  - 如此进行下去，直到访问遍所有的顶点。
- 广度优先搜索树（breadth-first search tree）

6.4.2 广度优先搜索（续）



广度优先搜索的顺序是a，b，d，e，f，c，g

```
void BFS(Graph& G, int V){  //广度优先搜索
  //初始化广度优先周游要用到的队列
  using std::queue;   queue<int> Q;
  //访问顶点V，并标记其标志位，V入队
  G.Mark[V]= VISITED;   Visit(G, V);     Q.push(V);
  while(!Q.empty()) {//如果队列仍然有元素
     int V=Q.front();   Q.pop(); //顶部元素      出队
     //将与该点相邻的每一个未访问点都入队
     for(Edge e=G.FirstEdge(V);
         G.IsEdge(e);e=G.NextEdge(e))
       if (G.Mark[G.ToVertex(e)]
             == UNVISITED)  {
          G.Mark[G.ToVertex(e)]=VISITED;
          Visit(G, G.ToVertex(e));
          Q.push(G.ToVertex(e)); //入队
       } // End of if
  } // End of while
} // End of BFS
```

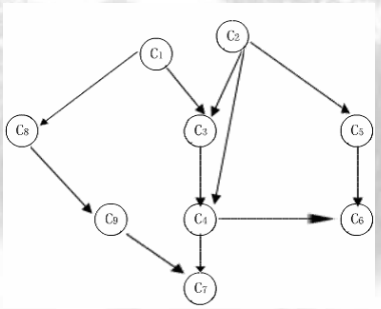

6.4.2 广度优先搜索（续）

- 广度优先搜索算法的时间复杂度与深度优先搜索算法的时间复杂度相同

6.4.3 拓扑排序（引子）

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	算法语言	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

拓扑排序问题描述

- 先决条件问题
- 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

拓扑序列定义

- 拓扑序列
  - 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，如果该顶点序列满足：
    - 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。

6.4.3 拓扑排序方法

- 任何无环有向图（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
  - （3）回到第（1）步继续执行

队列方式实现的拓扑排序

```
void TopsortbyQueue(Graph& G) {
  for(int i=0;i<G.VerticesNum();i++)
      G.Mark[i]=UNVISITED;   //初始化标记数组
 using std::queue;
 queue<int> Q;                    //初始化队列
 for(i=0; i<G.VerticesNum(); i++) {
   if (G.Indegree[i]==0)
     Q.push(i);                 //图中入度为0的顶点入队
 }
```



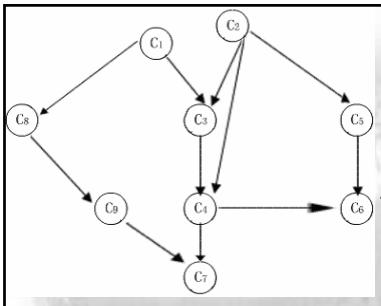
```
while (!Q.empty()) {  //如果队列中还有图顶点
  int V=Q.front();            //顶部元素
  Q.pop();                    //一个顶点出队
  Visit(G, V);                //访问该顶点
  G.Mark[V]=VISITED;
  //边e的终点的入度值减1
  for (Edge e= G.FirstEdge(V);
    G.IsEdge(e);e=G.NextEdge(e))
  {
   G.Indegree[G.ToVertex(e)]--;
   if(G.Indegree[G.ToVertex(e)]==0)
     Q.push(G.ToVertex(e)); //入度为0的点入队
  }
}
```



```
      for(i=0; i<G.VerticesNum(); i++)
        if(G.Mark[i]==UNVISITED)
        {
          Print("图有环");    //图有环
          break;
        }
    }   // End of TopsortbyQueue()
```


队列实现的拓扑排序讨论

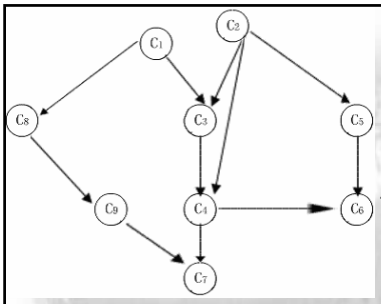
- 怎么知道图中所有顶点的入度？
- 是否可以用栈来取代队列？



按从左到右观察后处理型深度优先:

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

置逆,则为拓扑序列：C2,C5, C1,C3,C4, C6, C8,C9,C7



按结点编号后处理型深度优先:

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

拓扑序列：C2,C5,C6,C1, C8,C9, C3,C4,C7

深度优先搜索实现的拓扑排序

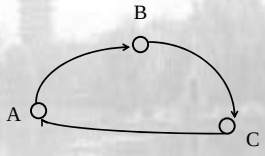
```
int *TopsortbyDFS(Graph& G) {    //结果是颠倒的
  for(int i=0; i<G.VerticesNum(); i++) //初始化
    G.Mark[i]=UNVISITED;
  int *result=new int[G.VerticesNum()];
  int tag=0;
  for(i=0; i<G.VerticesNum(); i++) //对所有顶点
    if(G.Mark[i]== UNVISITED)
      Do_topsort(G, i, result, tag);  //调用递归函数
  for(i=G.VerticesNum()-1;i>=0;i--) //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&
   tag) {
  G.Mark[V] = VISITED;
  for (Edge e = G.FirstEdge(V);
       G.IsEdge(e);e=G.NextEdge(e)) {
   //访问V邻接到的所有未被访问过的顶点
   if(G.Mark[G.ToVertex(e)]== UNVISITED)
      Do_topsort(G, G.ToVertex(e), result, tag);
  }
  // 相当于后处理
  result[tag++]=V;
}
```


深度优先搜索拓扑排序讨论

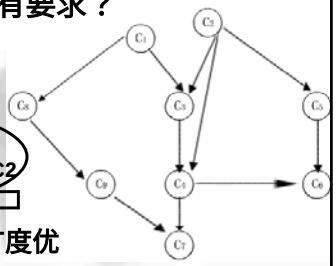
- 是否可以处理有环的情况？



深度优先搜索拓扑排序讨论

- 对于起始点是否有要求？

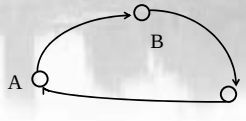
从C8开始
C7 C9 C8 C4 C3 C1 C6 C5 C2



- 是否可以修改为广度优先？

针对有回路的拓扑排序

- 解决的方法：
  - 将标志位设为三种情况：unvisit,visit和pushed(表明已放入结果数组)
  - 当遇到一个结点，从它所能邻接到的结点中若有已被标志为visit但不是pushed 的时，表明出现了环，即打印“存在环”即可



```
//深度优先搜索方式实现拓扑排序
void Do_topsort_Circle(Graph& G, int V,int
    *result,int& tag)  {
    if(G.Mark[V]== VISITED)  {
        cout<<"此图有环！"<<endl;       //图有环
        tag = CIRCLED;   return;
    }
    G.Mark[V]= VISITED;
    for   (Edge e= G.FirstEdge(V);
           G.IsEdge(e);e=G.NextEdge(e))
    {    //访问V邻接到的所有未被访问过的顶点
        if ((G.Mark[G.ToVertex(e)] != PUSHED)
            && (tag == NOTCIRCLED) )
        //递归调用
        Do_topsort_Circle(G, G.ToVertex(e),result,tag);
    }
    result[tag++]=V;    G.Mark[V] = PUSHED;
}
```



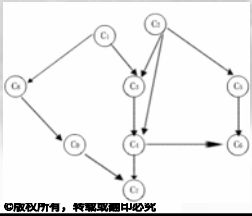
```
//深度优先搜索方式实现的拓扑排序,结果是颠倒的
void TopsortbyDFS_Circle(Graph& G)
{
    //对图所有顶点的标志位进行初始化
    for(int i=0; i<G.VerticesNum(); i++)
        G.Mark[i]=UNVISITED;
    int *result=new int[G.VerticesNum()];
    int tag=NOTCIRCLED;
    //对图的所有顶点进行处理
    for (i=0; i<G.VerticesNum(); i++)
        if ((G.Mark[i]== UNVISITED)
            && (tag == NOTCIRCLED))
        Do_topsort_Circle(G,i,result,tag);   //调用递归
    if  (tag == NOTCIRCLED)
    for (i=G.VerticesNum()-1;i>=0;i--)  //逆序输出
        Visit(G, result[i]);
}
```


拓扑排序的时间复杂度

- 与图的深度优先搜索方式周游相同
  - 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Θ(|V|+|E|)
- 采用相邻矩阵表示时，为Θ(|V|²)
- 在有环的情况下会提前退出，从而可能没处理完所有的边和顶点

队列拓扑排序计算入度

- 与图的深度优先搜索方式周游相同
  - 当采用相邻矩阵时，需要 Θ (|V|²)
  - 当采用邻接表时，为Θ(|V|+|E|)



递归与非递归的拓扑排序

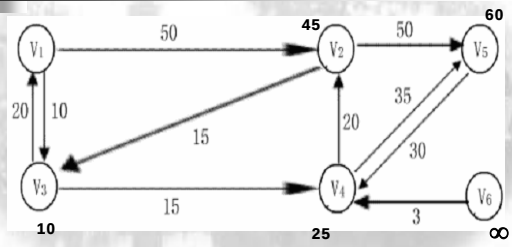
- 是否支持
  - 有向图、无向图
  - 有回路的图
  - 非连通图
  - 权值为负
- 如果不支持
  - 则修改方案？

- 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
  - 递归的算法要求判断有无回路
  - 非递归的算法，最后判断（若还有顶点没有输出，肯定有回路）

6.5 最短路径问题

- 6.5.1 单源最短路径：Dijstra算法
  - 指的是对已知图G=（V，E），给定源顶点s∈V，找出s到图中其它各顶点的最短路径。
- 6.5.2 每对顶点间的最短路径：Floyd算法
  - 指的是对已知图G=（V，E），任意的顶点V_i，V_j∈V，找出从V_i到V_j的最短路径

单源最短路径图例



求上图中顶点V_1到其它各顶点的最短路径

Dijstra算法基本思想

- 把图中所有顶点分成两组
  - 第一组包括已确定最短路径的顶点
  - 第二组包括尚未确定最短路径的顶点；
- 按最短路径长度递增的顺序逐个把第二组的顶点加到第一组中去
  - 直至从s出发可以到达的所有顶点都包括进第一组中。

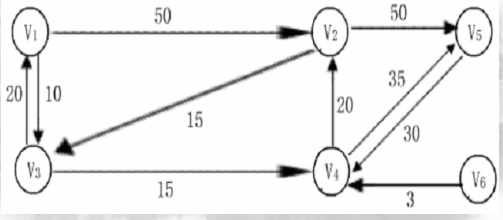
- 在这个过程中，总保持从s到第一组各顶点的最短路径长度都不大于从s到第二组的任何顶点的最短路径长度，而且，每个顶点都对应一个距离值：
  - 第一组的顶点对应的距离值就是从s到该顶点的最短路径长度
  - 第二组的顶点对应的距离值是从s到该顶点的只包括第一组的顶点为中间顶点的最短路径长度

Dijkstra算法的具体做法

- 一开始第一组只包括顶点s，第二组包括其它所有顶点；
- s对应的距离值为0，而第二组的顶点对应的距离值这样确定：
  - 若图中有边<s，V_i>或者（s，V_i），则V_i的距离值为此边所带的权，否则V_i的距离值为∞。
- 然后，每次从第二组的顶点中选一个其距离值为最小的顶点V_m加入到第一组中；

- 每往第一组加入一个顶点V_m，就要对第二组的各顶点的距离值进行一次修正：
  - 若加进V_m做中间顶点，使从s到V_i的最短路径比不加V_m的为短，则需要修改V_i的距离值。
- 修改后再选距离值最小的顶点加入到第一组中
- ……
- 如此进行下去，直到图的所有顶点都包括在第一组中或者再也没有可加入到第一组的顶点存在。

单源最短路径运算示意



求上图中顶点V_1到其它各顶点的最短路径

	V_1	V_2	V_3	V_4	V_5	V_6
初始状态	length:0 pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1
V_1进入第一组	length:0 pre:1	length:50 pre:1	length:10 pre:1	length:∞ pre:1	length:∞ pre:1	length:∞ pre:1
V_3进入第一组	length:0 pre:1	length:50 pre:1	length:10 pre:1	length:25 pre:3	length:∞ pre:1	length:∞ pre:1
V_4进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length:∞ pre:1
V_2进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length:∞ pre:1
V_5进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length:∞ pre:1

用Dijkstra算法的处理过程，源顶点为V_1

- Dist类可以如下定义：

```
Class Dist{
   int length;    //与源s的距离
   int pre;       //前面的顶点
};
```


Dijkstra算法

```
Dist *Dijkstra(Graph& G,int s) {
  Dist *D=new Dist[G.VerticesNum()];

  //初始化Mark数组、D数组
  //minVertex函数中会用到Mark数组的信息
  for(int i=0;i<G.VerticesNum();i++){
    G.Mark[i]=UNVISITED;
    D[i].length= INFINITY;
    D[i].pre=s;
  }
  D[s].length=0;
```



```
for(i=0;i<G.VerticesNum();i++){
 int v=minVertex(G,D);    //找到距离s最小的顶点
 if(D[v].length==INFINITY)
    break;
 G.Mark[v]=VISITED;       //把该点加入已访问组
 Visit(G,v);              //打印输出
 //刷新D与v相邻的点的Dist值
 for(Edge e=G.FirstEdge(v);
     G.IsEdge(e);e=G.NextEdge(e)) {
     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;
     }
 }
}
return D;
}
```


Dijstra算法时间代价分析

- minVertex()函数通过两两比较来扫描D数组
- for(i=0;i<G.VerticesNum();i++)循环|V|次
  - 每次minVertex()扫描需要进行|V|次，共$|V|^2$
  - 而在更新D值处总共扫描|E|次，$\Theta(|V|+|E|)$
- 总时间代价为$\Theta(|V|^2)$

minVertex()函数采用最小堆

- 每次改变D[i].length
- 先删除再重新插入
- 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，该顶点一定被标记为VISITED，从而被忽略

不删除旧值的缺点

- 在最差情况下，它将使堆中元素数目由Θ(|V|)增加到Θ(|E|)
- 总的时间代价
  - Θ((|V|+|E|) log|E|)
  - 因为处理每条边时都必须对堆进行一次重排

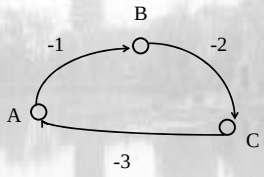
Dijstra算法

- 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？

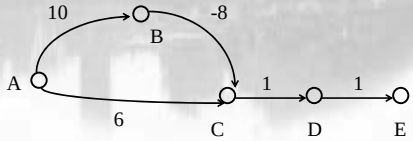
- 针对有向图（且要求“有源”）
  - 若输入无向图？
  - 照样能够处理（每条边都双向）
- 不支持非连通图
  - 没有必要修改
- 支持回路
- 支持负权值？

Dijstra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



- 即使不存在负的回路，也将出现后出现的负权值，从而导致整体计算错误
- 主要原因是Dijstra算法是贪心法，当作最小取进来后，不会返回去重新计算



- 支持负权值的最短路径算法
  - 参考书MIT《Introduction to Algorithms》

6.5.2 每对顶点间的最短路径

- 以每个顶点为起点，调用|V|次Dijstra算法

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

Dijkstra算法

```
Dist *Dijkstra(Graph& G,int s) {
  Dist *D=new Dist[G.VerticesNum()];

  //初始化Mark数组、D数组
  //minVertex函数中会用到Mark数组的信息
  for(int i=0;i<G.VerticesNum();i++){
    G.Mark[i]=UNVISITED;
    D[i].length= INFINITY;
    D[i].pre=s;
  }
  D[s].length=0;
```



```
for(i=0;i<G.VerticesNum();i++){
  int v=minVertex(G,D);    //找到距离s最小的顶点，可以用堆
  if(D[v].length==INFINITY)
      break;
  G.Mark[v]=VISITED;       //把该点加入已访问组
  Visit(G,v);              //打印输出
  //刷新D与v相邻的点的Dist值
  for(Edge e=G.FirstEdge(v);
      G.IsEdge(e);e=G.NextEdge(e)) {
      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;
        // 如果用最小堆，还需要 H.Insert(D[G.ToVertex(e)]);
      }
  }
}
return D;
}
```


Dijkstra算法（堆实现）

```
Dist *Dijkstra(Graph& G,int s) {
  Dist *D=new Dist[G.VerticesNum()];
  Dist tmp;
  //初始化Mark数组、D数组
  for(int i=0;i<G.VerticesNum();i++){
    G.Mark[i]=UNVISITED;
    D[i].length= INFINITY;   D[i].pre=s;
  }
  D[s].length=0;
  MinHeap<Dist>  H(G. EdgesNum());    //最小值堆
  H.Insert(D[s]);
  for(i=0;i<G.VerticesNum();i++)    {
    do {
       if (!H. RemoveMin(tmp))  return D; //非连通
       v = tmp.index;
    } while (G.Mark[d.index]==VISITED);
  //注意，修改堆中函数返回类型bool RemoveMin(T& node);
```



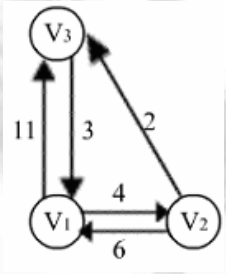
```
    if(D[v].length==INFINITY)   //非连通
        break;
    G.Mark[v]=VISITED;       //把该点加入已访问组
    Visit(G,v);              //打印输出
    //刷新D与v相邻的点的Dist值
    for(Edge e=G.FirstEdge(v);
        G.IsEdge(e);e=G.NextEdge(e))
        if (D[G.ToVertex(e)].length >
                    D[v].length+G.Weight(e) ){
          D[G.ToVertex(e)].length=D[v].length
                                  + G.Weight(e);
          D[G.ToVertex(e)].pre=v;
          // 如果用最小堆，还需要
       H.Insert(D[G.ToVertex(e)]);
        }

   } // End of for(i=0;i<G.VerticesNum();i++)
   return D;
   }
```


Floyd算法算法思想

- 假设用相邻矩阵adj表示图
- Floyd算法递归地产生一个矩阵序列 $adj^{(0)}$，$adj^{(1)}$，…，$adj^{(k)}$，…，$adj^{(n)}$
- $adj^{(k)}[i，j]$等于从顶点V_i到顶点V_j中间顶点序号不大于k的最短路径长度

- 假设已求得矩阵$adj^{(k-1)}$，那么从顶点V_i到顶点V_j中间顶点的序号不大于k的最短路径有两种情况：
  - 一种是中间不经过顶点V_k，那么就有 $adj^{(k)}[i，j]=adj^{(k-1)}[i，j]$
  - 另一种是中间经过顶点V_k，那么
    - $adj^{(k)}[i，j]< adj^{(k-1)}[i，j]$
    - 且$adj^{(k)}[i，j]= adj^{(k-1)}[i，k]+ adj^{(k-1)}[k，j]$

Floyd算法图示



$adj^{(0)}=\begin{bmatrix}0&4&11\\6&0&2\\3&\infty&0\end{bmatrix}$ $path=\begin{bmatrix}1&1&1\\2&2&2\\3&0&3\end{bmatrix}$

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

$adj^{(2)}=\begin{bmatrix}0&4&6\\6&0&2\\3&7&0\end{bmatrix}$ $path=\begin{bmatrix}1&1&2\\2&2&2\\3&1&3\end{bmatrix}$

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

Floyd算法

```
void Floyd(Graph& G){
   int i,j,k;                  //i,j,k作为计数器
  Dist **D=new Dist*[G.VerticesNum()];
  for (i=0; ;i<G.VerticesNum();i++)
     D[i]=new Dist[G.VerticesNum()];
  // 建立直接边， adj(0)
  for(i=0; i<G.VerticesNum(); i++)
     for(j=0; i<G.VerticesNum(); j++) {
        D[i][j].length = G.Weight(i, j);
        D[i][j].pre = i;
     }
```



```
  //如果两个顶点间的最短路径经过顶点k，则有
  //D[i][j].length>(D[i][k].length+D[k][j].length
  for(k=0;k<G.VerticesNum();k++)
    for(i=0;i<G.VerticesNum();i++)
      for(j=0;j<G.VerticesNum();j++)
       if (D[i][j].length
             > D[i][k].length+D[k][j].length){
          D[i][j].length=D[i][k].length+D[k][j].length;
          D[i][j].pre = k;
       }
    for(i=0;i<G.VerticesNum();i++)
      for(j=0;j<G.VerticesNum();j++)
        Visit(D[i][j].length, D[i][j].pre);
    for(i=0;i<G.VerticesNum();i++)
       delete [] D[i];
     delete [] D;
}
```


Floyd算法的时间复杂度

- 三重for循环
  - 复杂度是Θ(|V|³)

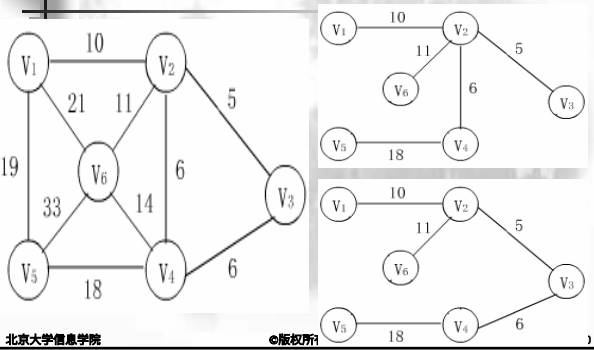
6.6 最小支撑树

- 最小支撑树（minimum-cost spanning tree，简称MST）
  - 对于带权的连通无向图G，其最小支撑树是一个包括G的所有顶点和部分边的图，这部分的边满足下列条件：

（1）这部分的边能够保证图是连通的；

（2）这部分的边，其权的总和最小。

6.6 最小支撑树（续）



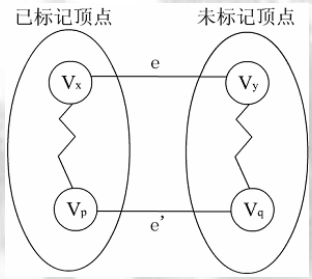
6.6.1 Prim算法

- Prim算法：与Dijstra算法类似
- 具体操作是：
  - 从图中任意一个顶点开始，首先把这个顶点包括在MST里，
  - 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，并把这条边和其不在MST里的那个端点包括进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。

证明用Prim算法构造的是MST

- 首先证明这样一个结论（MST性质）：
  - 设T(V*，E*)是连通无向图G=(V，E)的一棵正在构造的生成树，又E中有边e=(V_x，V_y)，其中V_x∈V*，V_y不属于V*，且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)。

6.6.1 Prim算法证明的图示



Prim算法证明过程的图示

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∈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) { //最小支撑树的Prim算法
  int *D=new int[G. VerticesNum()]; //记录顶点距离
  //记录顶点相关联的顶点
  int *V=new int[G. VerticesNum()];
  int MSTtag=0;              //最小支撑树边的标号
  Edge *MST=new Edge[G.VerticesNum()-1];
  for(int i=0;i<G.VerticesNum();i++) {
     //初始化Mark数组、距离数组
     G.Mark[i]=UNVISITED;
     D[i]=INFINITY;
  }
  //选定的开始顶点的距离置为0
  D[s]=0;
```



```
  for(i=0; i< G. VerticesNum(); i++)  {
     //寻找下一条权最小的边
     int v=minVertex(G, D);
     //在顶点v的标志位上做已访问的标记
     G.Mark[v]=VISITED;
     //不存在最小支撑树
     if (D[v]==INFINITY)  {
        Print("不存在最小支撑树。"); return; }
     if (v!=s)          //将边(V[v]，v)加到MST中
        AddEdgetoMST(V[v], v,MST,MSTtag++);
     for(Edge e= G. FirstEdge(v); G.IsEdge(e);e=G.
                    NextEdge(e))
        if(D[G. ToVertex(e)]>G. Weight(e))  {
           //修改顶点G. ToVertices(e)的距离值，关联点
           D[G. ToVertex(e)]=G. Weight(e);
           V[G. ToVertex(e)]=v;
        }
  } End of for(i=0; i< G. VerticesNum(); i++)
```



```
   //输出
   for(i=0;i<G.VerticesNum()-1;i++)
      Print(MST[i].from,MST[i].to);
   delete [] D;
   delete [] V;
   delete [] MST;
}
void AddEdgetoMST(int i,int j,Edge *MST,int tag)
{
   MST[tag].from=i;
   MST[tag].to=j;
}
```

以上代码中的minVertex函数，可用最小堆等方式实现.

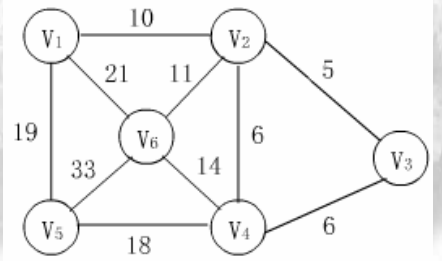
6.6.1 Prim算法（续）

- Prim算法与Dijkstra算法十分相似，唯一区别是：
  - Prim算法要寻找的是离已加入顶点距离最近的顶点
  - 而不是寻找离固定顶点距离最近的顶点
- 所以其时间复杂度分析与Dijkstra算法相同。

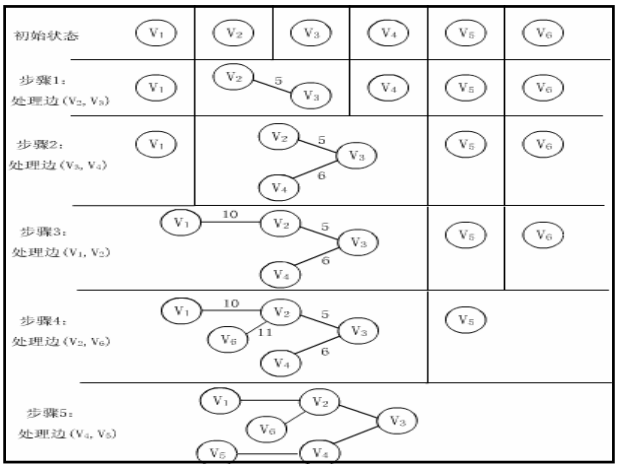
6.6.2 Kruskal算法

- Kruskal算法的基本思想是：
  - 对于图G=（V，E），开始时，将顶点集分为|V|个等价类，每个等价类包括一个顶点；
  - 然后，以权的大小从小到大为顺序处理各条边，如果某条边连接两个不同等价类的顶点，则这条边被添加到MST，两个等价类被合并为一个；
  - 反复执行此过程，直到只剩下一个等价类。

6.6.2 Kruskal算法（续）



对上图用Kruskal算法，其处理过程见下图



Kruskal算法

```
void Kruskal(Graph& G) {
   Gentree A(G.VerticesNum()); //等价类
   //记录图的所有边
   Edge *E=new Edge[G.EdgesNum()];
   //最小支撑树
   Edge *MST=new Edge[G.VerticesNum()-1];
   int MSTtag=0, edgecount=0;
   //将图的所有边记录在数组E中
   for(int i=0; i<G.VerticesNum(); i++) {
        for(Edge e= G. FirstEdge(i);
            G.IsEdge(e);e=G. NextEdge(e))
          if(G.FromVertex(e)< G.ToVertex(e))
              E[edgecount++]=e;
   }
   //最小堆（min-heap）
   minheap<Edge> H(E,edgecount, edgecount);
```



```
   int EquNum=G.VerticesNum(); //|V|个等价类
   while (EquNum>1)    {  //合并等价类
        Edge e=H.removemin();   //获得权最小的边
        if (e.weight>=INFINITY)          {
              Print("不存在最小支撑树.");
              delete [] MST；  return;
        }
        //记录该条边的信息
        int from=G.FromVertex(e), to= G.ToVertex(e);
        if(A.differ(from,to)) {  //两个顶点不在一个等价类
            //将边e的两个顶点所在的两个等价类合并为一个
            A.UNION(from,to);
            AddEdgetoMST(from,to,MST,MSTtag++);
            EquNum--;   //将等价类的个数减1
        }
   }
   for(i=0;i<G.VerticesNum()-1;i++)
     Print(MST[i].from,   MST[i].to);
   }
```


Kruskal算法代价

- 使用了路径压缩，differ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（|E|log |E|），即堆排序的时间
- 通常情况下只找了接近顶点数目那么多边的情况下，MST就已经生成，时间代价接近于Θ（|V|log |E|）

Prim算法、Kruskal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？
- 针对无向图（若算法的输入是有向图呢？）
  - Prim能给出一个解
  - 修改后Kruskal也给出解
  - 但解无意义
- 都不支持非连通图
  - 可以修改
- 支持负权值
  - 假设最小的负权值为Neg
  - 所有边权值+Neg
- 都支持回路

总结

- 6.1 图的基本概念
- 6.2 图的抽象数据类型
- 6.3 图的存储结构
- 6.4 图的周游
- 6.5 最短路径问题
- 6.6 最小支撑树