第17章 贪婪算法

- 最优化问题
- 贪婪算法(greedy method)思想
- 应用
- **17.3.2 0/1背包问题**
- **17.3.3 拓扑排序**
- **17.3.5 单源最短路径**
- **17.3.6 最小耗费生成树**

最优化问题

- 每个最优化问题都包含一组<u>**限制条件**</u>(constraint)和一个<u>**优化函数**</u>(optimization function)；
- 符合限制条件的问题求解方案称为<u>**可行解**</u>(feasible solution)；
- 使优化函数取得最佳值的可行解称为<u>**最优解**</u>(optimal solution)。

最优化问题

- 一个有趣的例子
- 商场有一场比赛：n种商品，手推车容量c, 商品i的体积w_i，价值p_i。商品容积之和不超过手推车的容量，每种商品只能拿一件。目标是装入手推车的总价值最大，第一名免费。

装载问题

- 有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第i 个货箱的重量为w_i（$1 \le i \le n$），而货船的最大载重量为c。
- 目的是在货船上装入最多的货箱。

装载问题–最优化问题描述

- 设存在一组变量x_i，其可能取值为0或1。
  - $x_i=0$，则货箱i 将不被装上船；
  - $x_i=1$，则货箱i 将被装上船。
- 目的是找到一组x_i，
  - 限制条件$\Sigma w_i x_i \le c$ 且$x_i \in \{0, 1\}$, $1 \le i \le n$。
  - 优化函数是Σx_i
- 满足限制条件的每一组x_i都是一个可行解，
- 能使Σx_i取得最大值的方案是最优解。

例：n=8,

[w1,...w8]=[100,200,50,90,150,50,20,80],c=400。

贪婪算法思想

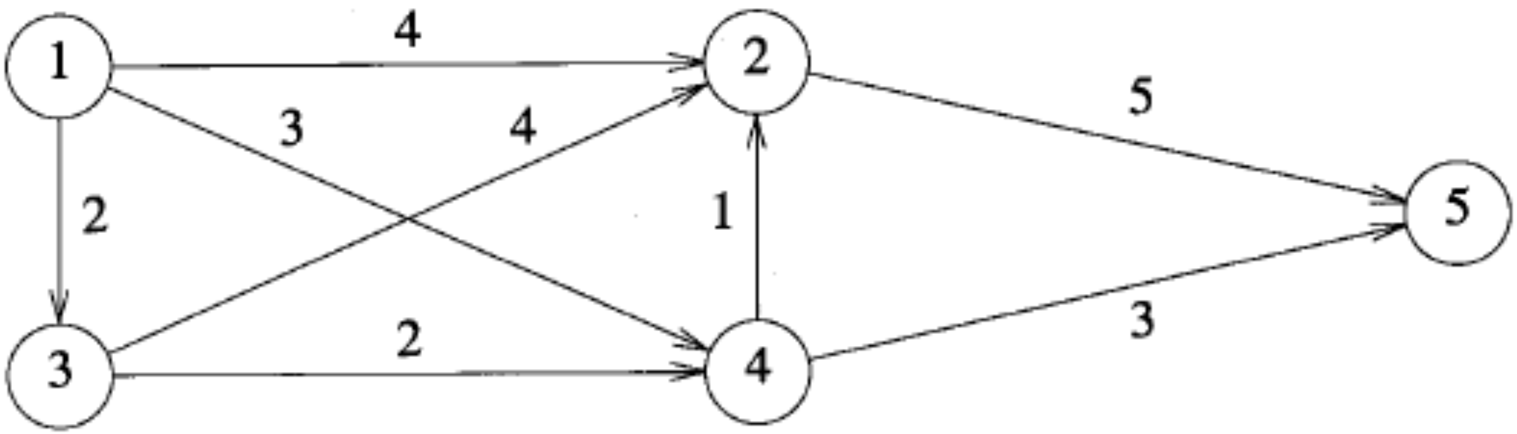
- 贪婪算法(greedy method)思想
  - 采用**逐步构造最优解**的方法。在每个阶段，都作出一个看上去最优的决策(在一定的标准下)。决策一旦作出，就不可再更改。
  - **贪婪准则:**做决策的依据.
  - **贪婪算法:**每一步,根据贪婪准则,做出一个看上去最优的决策.

货箱装船问题

- 贪婪准则：从剩下的货箱中，选择重量最小的货箱。
- 这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。
- 例
  - n=8,
  - [w1,...w8]=[100,200,50,90,150,50,20,80],c=400。
- 利用贪婪算法能产生最佳装载

最短路径

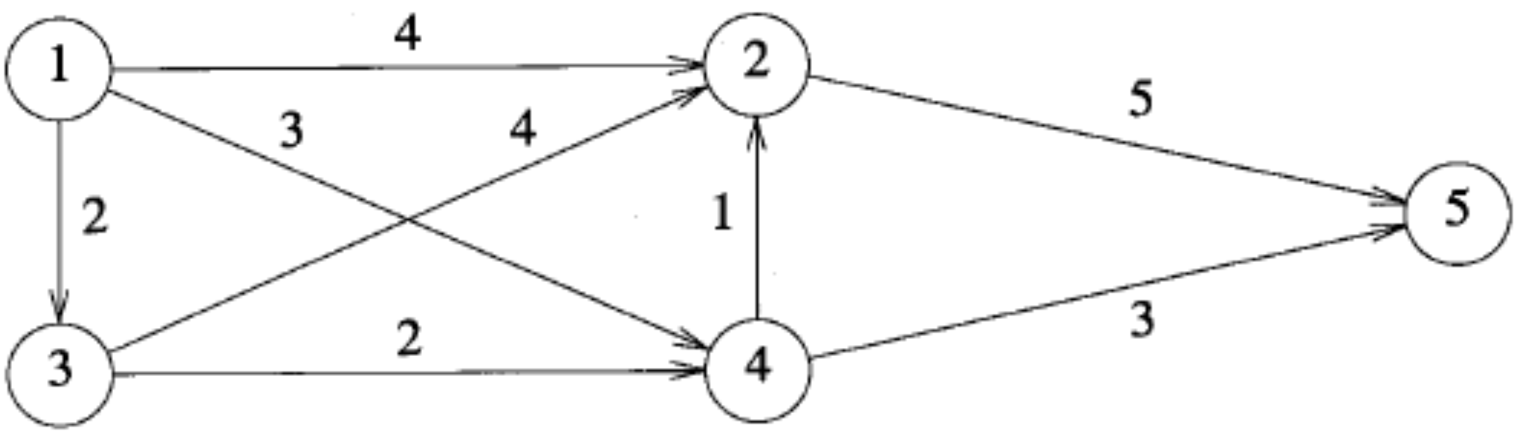
- 找一条从初始顶点s 到达目的顶点d 的最短路径



- 贪婪算法：分步构造这条路径，每一步在路径中加入一个顶点

最短路径-贪婪算法思想

- 加入下一个顶点的**贪婪准则**：
  - 假设当前路径已到达顶点q，且顶点q 并不是目的顶点d
  - 选择离q 最近且目前不在路径中的顶点



- 这种贪婪算法并不一定能获得最短路径(例1->5)

- 回忆一下，前面树的内容中哪一个算法是用到了贪心算法的思想？

17.3.2 0/1背包问题

- 在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为w_i，价值为p_i。
- 可行的背包装载：背包中物品的总重量不能超过背包的容量
  - 约束条件为$\sum w_i x_i <= c$和$x_i \in [0,1] (1 \le i \le n)$。
- 最佳装载是指所装入的物品价值最高，即
- $\sum_{i=1}^{n} p_i x_i$ 取得最大值。

0/1背包问题

- 需求出x_i 的值
  - $x_i=1$ 表示物品i 装入背包中
  - $x_i=0$ 表示物品i 不装入背包
- 0/1背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。
- 货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

0/1背包问题贪婪策略

- 贪婪准则：从剩余的物品中，选出可以装入背包的价值最大的物品。
- 利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。
- 这种策略不能保证得到最优解。
- n=3,w=[100,10,10],p=[20,15,15],c=105。

0/1背包问题贪婪策略

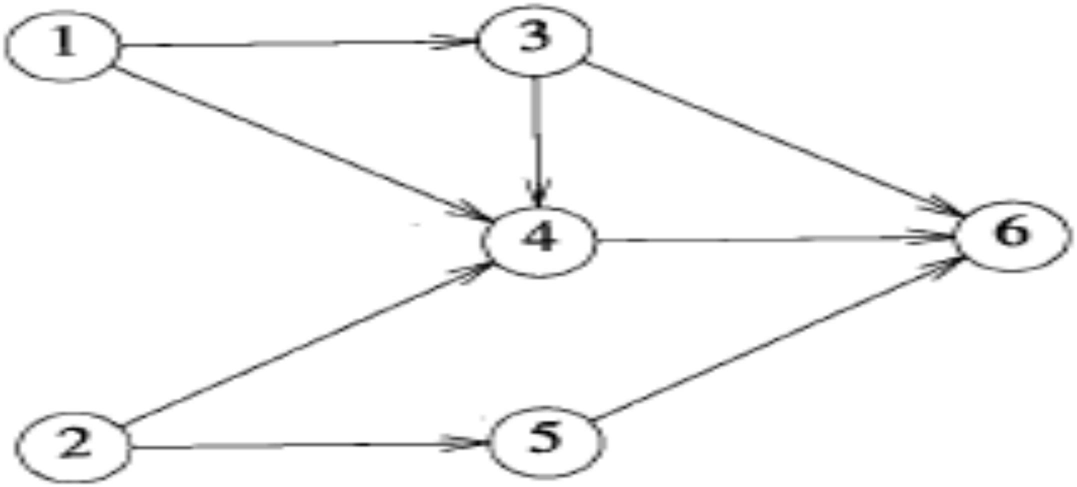
- 重量贪婪准则：从剩余的物品中，选出可以装入背包的重量最小的物品。

- 价值密度贪婪准则：从剩余的物品中，选出可以装入背包的价值密度最大的物品。

- n=3,w=[100,10,10],p=[20,15,15],c=105。

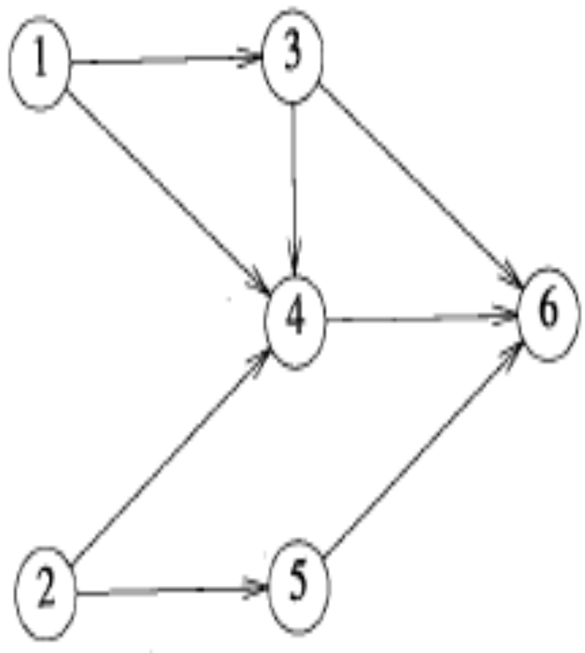
17.3.3 拓扑排序

- 一个复杂的工程通常可以分解成一组简单任务(活动)的集合，完成这些简单任务意味着整个工程的完成。
- 任务之间具有先后关系。



顶点活动网络(AOV)

- 顶点活动网络(AOV—Activity on vertex network)：任务的集合以及任务的先后顺序
  - 顶点：表示任务(活动)
  - 有向边(i, j)：表示任务间先后关系——任务j 开始前任务i 必须完成。

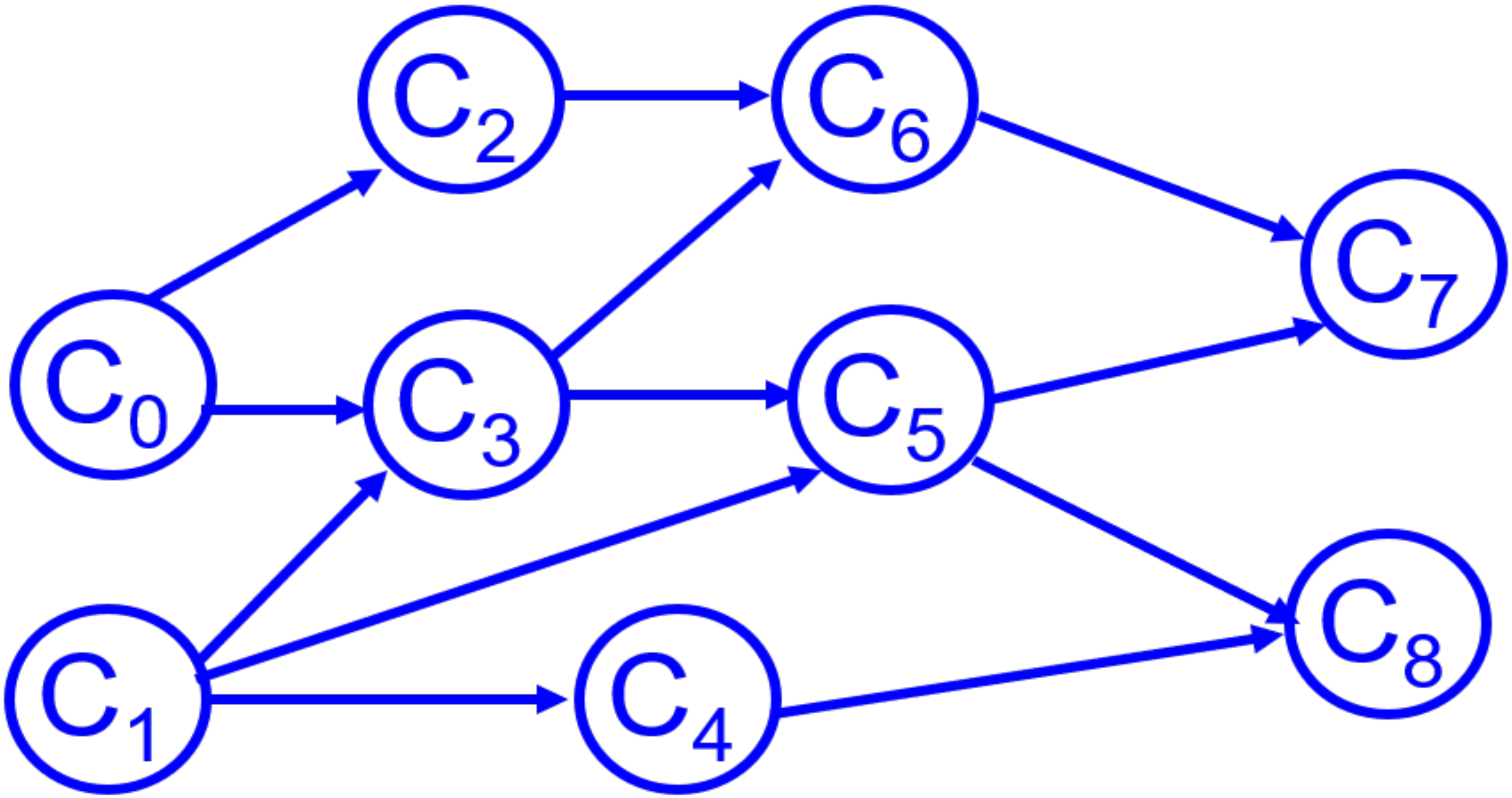


拓扑排序应用示例

- 计算机专业的学习就是一个工程，每一门课程的学习就是整个工程的一些活动(任务)。其中有些课程要求先修课程。

课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1，C_2
C_4	数据结构	C_3，C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5，C_4
C_7	操作系统	C_4，C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8

拓扑排序应用示例



课程学习工程图

拓扑排序问题

- 在很多条件下，任务的执行是连续进行的，需要按照一个顺序来执行。
- 任务序列？

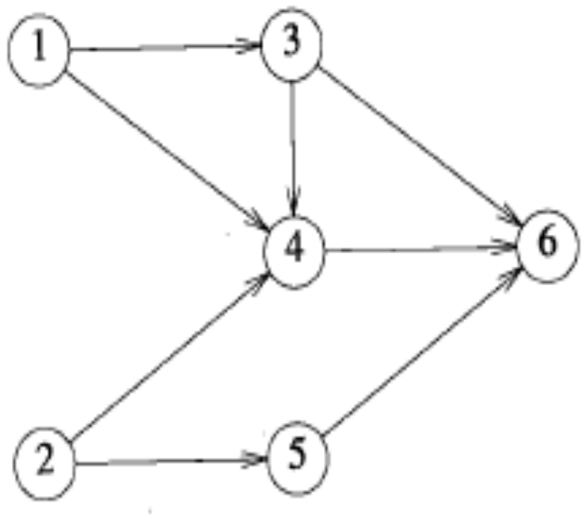
拓扑序列和拓扑排序

- **拓扑序列**(Topological orders/topological sequences):

⇒满足：对于在任务的有向图中的任一边 (i,j)，在序列中任务*i* 在任务*j* 的前面

- **拓扑排序**(Topological Sorting):

⇒根据任务的有向图建立拓扑序列的过程



– 123456
– 132456
– 215346
– 142356

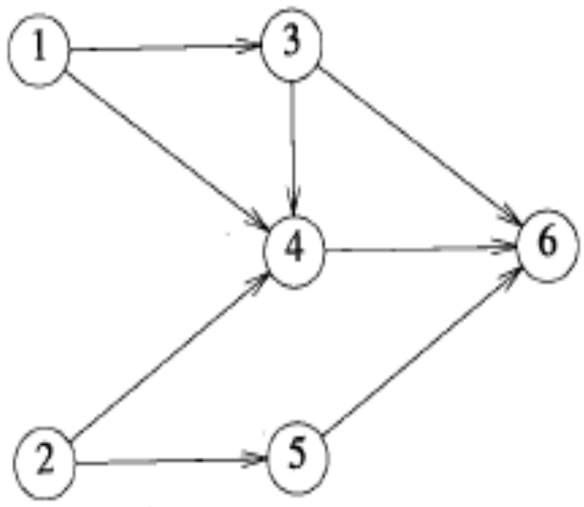
拓扑序列和拓扑排序

- **拓扑序列**(Topological orders/topological sequences):

⇒满足：对于在任务的有向图中的任一边 (i,j)，在序列中任务*i* 在任务*j* 的前面

- **拓扑排序**(Topological Sorting):

⇒根据任务的有向图建立拓扑序列的过程



- 拓扑序列:
  - 123456
  - 132456
  - 215346

拓扑排序算法

设*n*是有向图中的顶点数;

设*theOrder*是一个空序列;

While (*true*)

{设*w*是任意一个不存在入边(*v*,*w*)的顶点，其中顶点*v* 不在*theOrder*中 **意味着w入度为0吗**

如果没有这样的*w*，*break*。

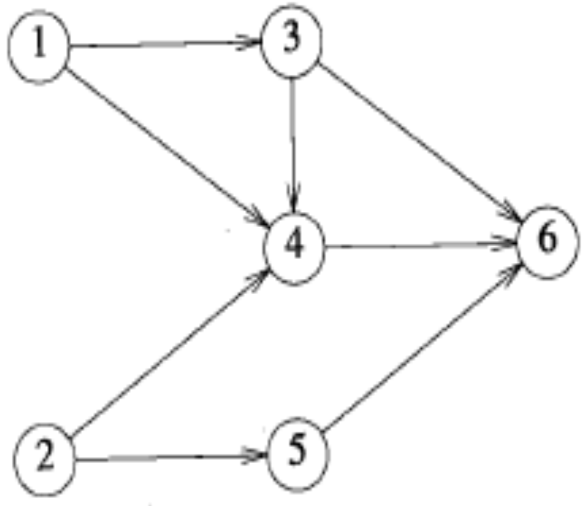
把*w*添加到*theOrder*的尾部

}

If (*theOrder*中的顶点数少于*n*) 算法失败

else *theOrder*是一个拓扑序列

拓扑排序示例



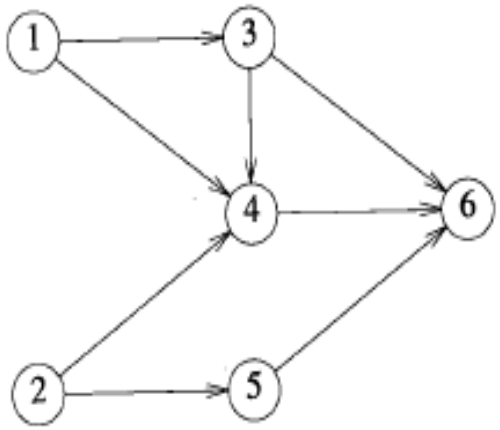
- 拓扑序列:
  - 123456
  - 251346
  - ……还有哪些?

实现

- 用一个一维数组表示theOrder
- 用一个栈来保存可加入theOrder的候选顶点
- 一个一维数组inDegree,其中indegree[j]表示不在theOrder中但邻接顶点j的顶点的数目，当indegree[j]变为0时，j变为候选节点

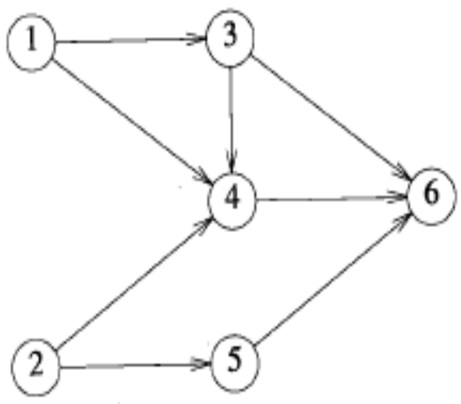
```
bool topologicalOrder(int *theOrder)
{ //求有向图中顶点的拓扑序列;如果找到了一个拓扑序列
   ，则返回true，此时，在theOrder[0:n-1]中记录拓扑序列;
   如果不存在拓扑序列，则返回false
    ……//确定图是有向图
    int n=numberofVertices();
    //计算入度
    int *inDegree = new int [n+1];
    fill(indegree+1, indegree+n+1, 0); //初始化
    for (i=1; i<=n; i++) {// i的出边
         vertexIterator<T> *ii=iterator(i);
         int u ;
         while ((u=ii->next())!=0) {
              inDegree[u]++;}
         }

    //把入度为 0 的顶点压入栈
    arrayStack<int> stack;
    for (i = 1; i <= n; i++)
         if (!inDegree[i]) stack.push(i);
```



```
// 生成拓扑序列
j = 0; // 数组theOrder 的索引
while (!stack.empty()) // 从堆栈中选择
   {int nextVertex= stack.top(); // 从栈中提取下一个顶点
    stack.pop();
    theOrder[j++] = nextVertex;
    //更新nextVertex邻接到的顶点的入度
    vertexIterator<T> *inextVertex =iterator(nextVertex);
    int u;
    while (u= inextVertex->next())!=0)
        { inDegree[u]--;
         If  (inDegree[u]==0)   stack.push(u);
        }
    }

return (j == n);
}
```



topologicalOrder的复杂性

- 使用(耗费)邻接矩阵描述：

  $\Theta(n^2)$

- 使用邻接链表描述:

  $\Theta(n+e)$

17.3.5 单源最短路径

- 带权有向图.
- 路径的长度即为此路径所经过的边的长度(耗费)之和。
- 对于给定的源顶点sourceVertex， 需找出从它到图中其他任意顶点(称为目的)的最短路径。
- 假设:
  - 边的长度(耗费) >= 0.
  - 没有路径的长度< 0.

Dijkstra算法

- Dijkstra(迪克斯特拉/迪杰斯特拉)
- E. Dijkstra发明的贪婪算法：**分步求源点sourceVertex**到其它各顶点的最短路径。**每一步产生一个到达新的目的顶点**的最短路径。
- Dijkstra算法：
  - 按路径长度递增顺序产生最短路径。首先最初产生从sourceVertex到它自身的路径，这条路径没有边，其长度为0。产生下一个最短路径的贪婪准则：在目前产生的每一条最短路径中，考虑加入一条边到达未产生最短路径的顶点，再从所有这些新路径中选择最短的。

Dijkstra算法

中文维基百科条目协作计划专页已建立，欢迎报名参与！ [关闭]

艾兹赫尔·戴克斯特拉 [编辑]

维基百科，自由的百科全书

艾兹赫尔·韦伯·戴克斯特拉（荷兰语：Edsger Wybe Dijkstra，荷兰语读音：[ˈɛtsxər ˈʋibə ˈdɛikstra] 聆听，1930年5月11日－2002年8月6日），又译**艾兹赫尔·韦伯·迪杰斯特拉**，生于荷兰鹿特丹，计算机科学家，是荷兰第一位以程式为专业的科学家。[1]曾在1972年获得图灵奖，之后，他还获得1974年AFIPS Harry Goode Memorial Award、1989年ACM SIGCSE计算机科学教育教学杰出贡献奖。

2002年，在他去世前不久，艾兹赫尔获得了ACM PODC（分布式计算原理）最具影响力论文奖，以表彰他在分布式领域中关于程序计算自稳定的贡献。为了纪念他，这个每年一度奖项也在此后被更名为“Dijkstra奖”。

他曾经提出“GOTO有害论”，信号量和PV原语，解决了有趣的“哲学家就餐问题”。

目录 [隐藏]

艾兹赫尔·韦伯·戴克斯特拉
Edsger Wybe Dijkstra

出生	1930年5月11日 荷兰鹿特丹
逝世	2002年8月6日（72岁） 荷兰尼嫩
知名于	戴克斯特拉算法 结构化编程 THE 操作系统 信号量（semaphore）
奖项	图灵奖 计算机协会成员

科学生涯

生平 [编辑]

艾兹赫尔·韦伯·戴克斯特拉出生于鹿特丹，大学就读于莱顿大学，研究理论物理学。[2][3]但他很快就发现自己的兴趣是计算机科学。1980年代，担任埃因霍温理工大学教授。

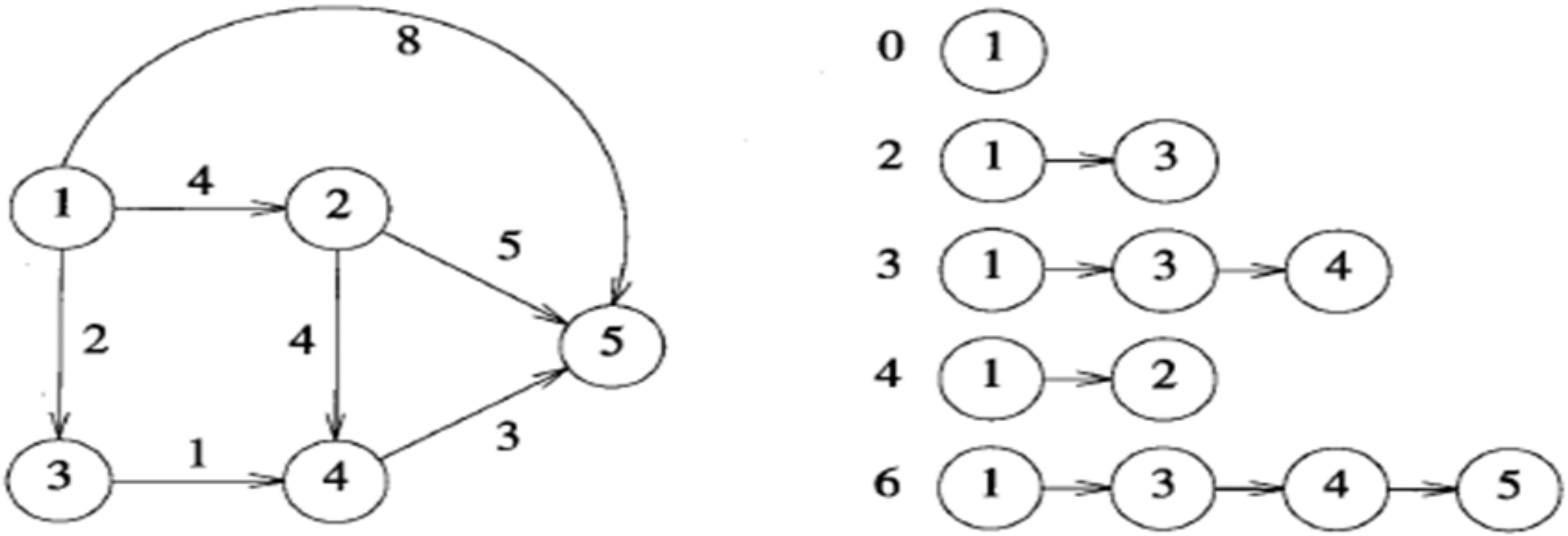
2002年8月6日，戴克斯特拉在荷兰尼嫩自己的家中与世长辞。终年72岁。

学术贡献 [编辑]

他的贡献包括：

- 提出了目前在离散数学中应用广泛的最短路径算法（Dijkstra's Shortest Path First Algorithm）
- 为解决操作系统中资源分配问题，提出银行家算法。

示例



a) b)

- 每一条路径(第1条除外)都是由一条已产生的最短路径加上一条边形成。
- 下一条最短路径总是由已产生的最短路径再扩充一条边得到的，且这条路径所到达的顶点其最短路径还未产生。

算法分析

- **distanceFromSource[i]:**
  - **在当前已产生的最短路径中加入一条边**,从而使得扩充的路径**到达顶点i的最短长度**。
- a：有向图的邻接矩阵，初始，仅有从sourceVertex到它自身的一条长度为0的路径，
  - 对于每个顶点i，distanceFromSource[i]等于a[sourceVertex][i] 。
- **表newReachableVertices:** 存储路径可到达顶点且未产生最短路径的顶点。

算法分析

- 数组**predecessor[]**:存储最短路径
  - **predecessor[i]**——从sourceVertex到达i的路径中顶点i前面的那个顶点。
  - 本例中**predecessor[1:5] = [0, 1, 1, 3, 4]**
  - 从 i = 5开始， **predecessor[5]=4, predecessor[4] =3, predecessor[3] =1 = sourceVertex,** 因此路径为**1，3，4，5**

Dijkstra算法的伪代码-1/2

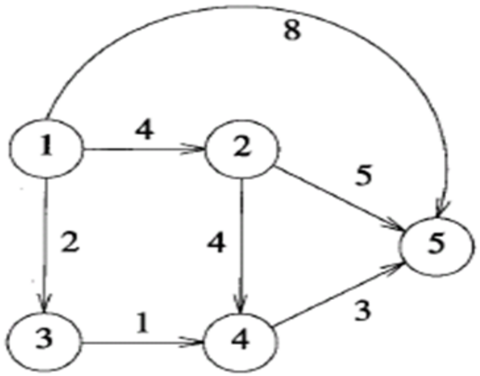
1)初始化

distanceFromSource[i] = a[sourceVertex][i](1≤i≤n)
，

predecessor[i] = sourceVertex, //邻接于sourceVertex的顶点

predecessor[sourceVertex] = 0；

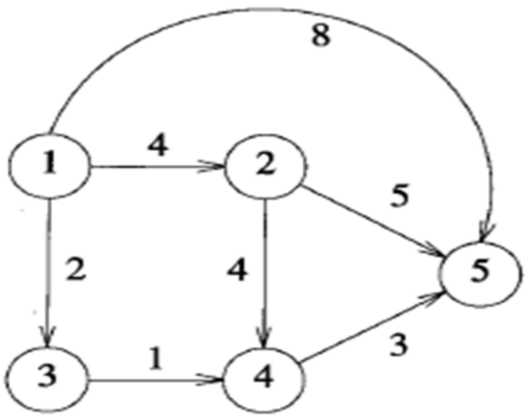
predecessor[i] = -1; //其余的顶点



创建一个表**newReachableVertices**，保存所有**predecessor**[i]>0的顶点。

2) 当**newReachableVertices**为空时,算法停止,否则转至3);

Dijkstra算法的伪代码-2/2



3) 从**newReachableVertices**中选择并删除
distanceFromSource值最小的顶点i。

4) 对于所有邻接于顶点i的顶点j,
更新**distanceFromSource[j]**值为
min{distanceFromSource[j],
distanceFromSource[i] +a[i][j]}；
若distanceFromSource[j]改变，
则 置predecessor[j]=i，
而且，若j没有在表newReachableVertices中，
则将j 加入newReachableVertices。

复杂性分析

- 读P435程序17-3
- 在**newReachableVertices**中选择**distanceFromSource**值最小的顶点**i**：O(n)
- 更新邻接自顶点**i** 的顶点的 **distanceFromSource**值和 **predecessor**值
  - 使用邻接表：O(顶点**i** 的出度) .
  - 使用耗费邻接矩阵：O(n) .
- 总的时间复杂性： $O(n^2)$.

17.3.6 最小耗费生成树

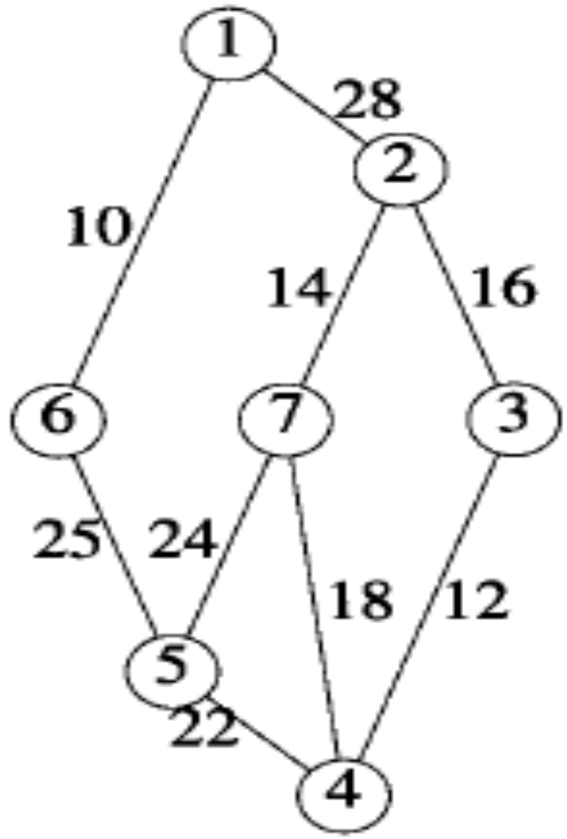
- 例[最小代价通讯网络]
  - 城市及城市之间所有可能的通信连接可被视作一个无向图，图的每条边都被赋予一个权值，权值表示建成由这条边所表示的通信连接所要付出的代价。
- 可行解：包含图中所有顶点（城市）的连通子图。设所有的权值都非负，则所有可能的可行解可表示成无向图的一组生成树。
- 最优解：具有最小代价的生成树。
- 限制条件：所有的边构成一个生成树。
- 优化函数：子集中所有边的权值之和。

- 最小耗费生成树(最小代价生成树/最小生成树)
- 具有*n*个顶点的**无向(连通)网络***G*的每个生成树刚好具有*n*-1条边。
- 最小耗费生成树问题是用某种方法**选择*n*-1条边**使它们形成*G*的**最小生成树**。
- 三种求解最小生成树的贪婪策略是:
  - **Kruskal (克鲁斯卡尔)算法**
  - **Prim(普里姆)算法**
  - Sollin算法

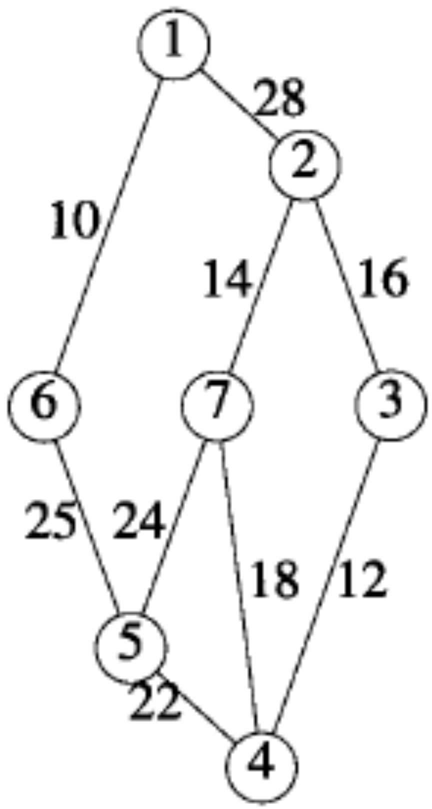
Kruskal算法

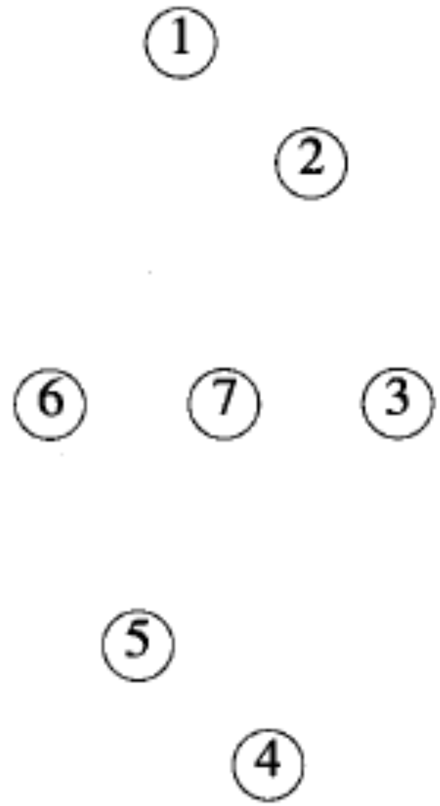
- Kruskal算法思想：
- 开始，初始化含有 <u>n个顶点</u> <u>0条边</u>的森林.
- Kruskal算法所使用的贪婪准则是：从剩下的边中选择一条<u>**不会产生环路**</u>的<u>**具有最小耗费**</u>的**边**加入已选择的边的集合中。
  - Kruskal算法分*e*步(*e*是网络中边的数目)。
  - 按<u>耗费递增的顺序</u>来考虑这*e*条边，<u>每次考虑一条边</u>。
  - 当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

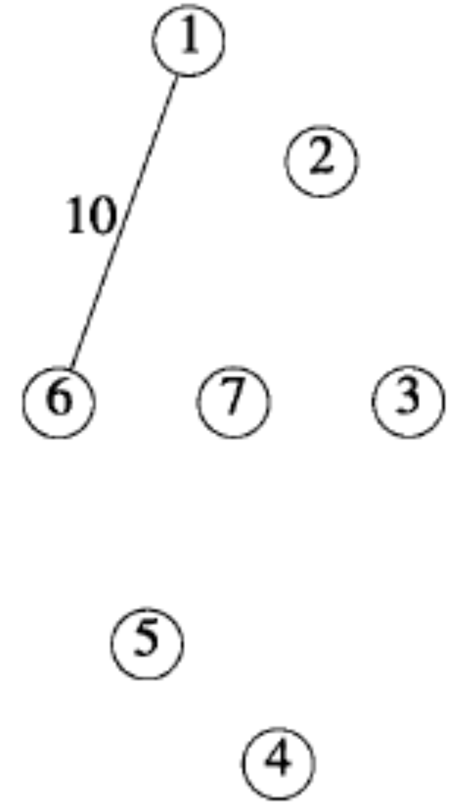
Kruskal算法示例



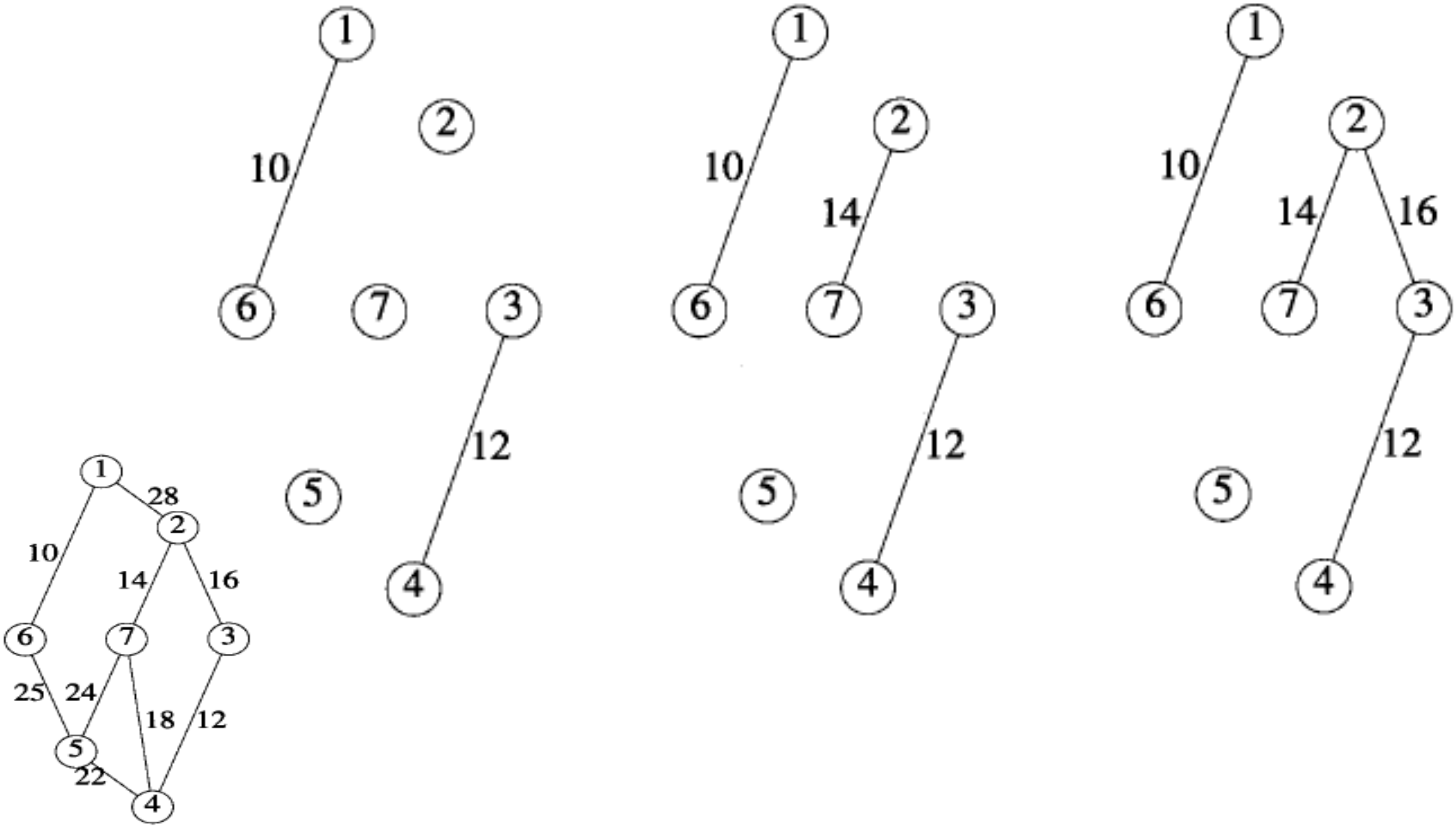
Kruskal算法示例



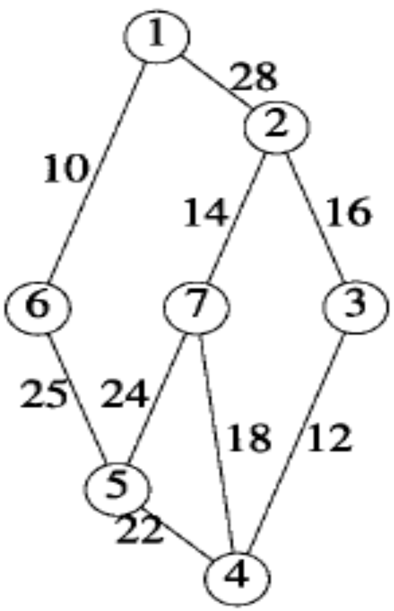


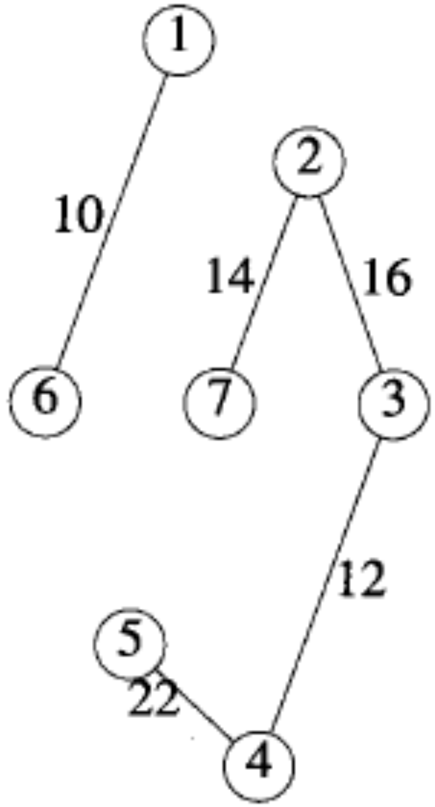


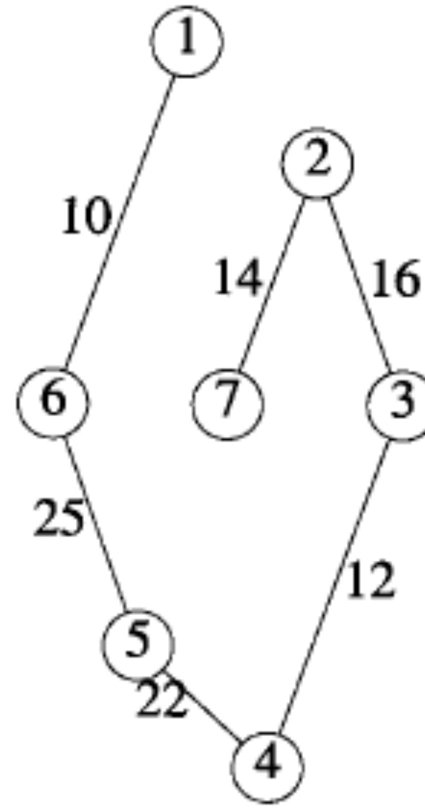
Kruskal算法示例



Kruskal算法示例







Kruskal算法的伪代码

- //在一个具有*n*个顶点的网络中找到一棵最小生成树
- 令*T*为所选边的集合，初始化T =∅
- 令E为网络中边的集合
- *While* (E≠∅ && |T| ≠ n-1){
  - 令(*u*,*v*)为*E*中代价最小的边
  - *E*=*E*-{(*u*,*v*) } //从*E*中删除边
  - *If* ((*u*,*v*)加入*T*中不会产生环路) 将(*u*,*v*)加入*T*
  - }
- *if* (|T| = = n-1) T是最小耗费生成树
- *Else* 网络不是互连的，不能找到生成树

•Kruskal算法的正确性证明 见P438

数据结构的选择及复杂性分析

- 边集 E：使用边的最小堆(小根堆)描述边集 E.
- 被选择的边的集合T：用数组spanningTreeEdges来实现

将边(u， v)加入T是否产生环路?

- 边的集合T与G中的顶点一起定义了一个由至多n个连通子图(树)构成的图。
- 当u 和 v处于同一子图时，将边 (u, v) 加入T会产生环路
- 当u 和 v处于不同的子图中时，将边 (u, v) 加入T则不会产生环路

将边(u， v)加入T是否产生环路?

- 用子图中的顶点集合来描述每个连通子图(树)，这些顶点集合没有公共顶点。
- 两个顶点在同一个子图中 *当且仅当* 它们在同一个顶点集合中

- **两个顶点是否在同一个集合中：**可利用并查集中的find操作实现
- ***在T中加一条边，两个子图被合并：可***利用并查集中的unite操作实现

复杂性分析

- 使用并查集(11.9.2).
  - 初始化.： O(n) .
  - find操作的次数最多为2*e*，Unite操作的次数最多为*n*-1(若网络是连通的，则刚好是*n*-1次)。
  - 比O(n+e)稍大一点。

- 使用边的最小堆，按耗费递增的顺序来考虑*e*条边： O(eloge).

- Kruskal算法的渐进复杂性： O(n+eloge).

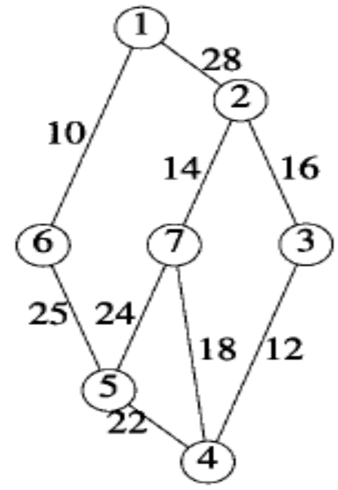
Graph::Kruskal 1/3

```
bool kruskal(weightedEdge<T> *spanningTreeEdges)
{//使用Kruskal算法寻找最小代价(耗费)生成树
 //如果不连通则返回false;
 //如果连通,则在spanningTreeEdges[0:n-2]中
 //返回最小生成树
      if (directed() || !weighted())  throw…….;

      int n = numberOfVertices();
      int e = numberOfEdges();
```

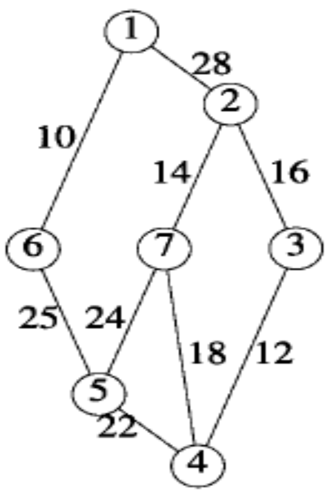
Graph::kruskal 2/3

```
// 建立图中的边数组
 weightedEdge<T> *edge = new weightedEdge<T> [e + 1];
  int k = 0;        // edge[]的索引(游标)
  for (int i = 1; i <= n; i++)
     {//获得所有关联至顶点 i的边
        vertexIterator<T> *ii = iterator(i);
         int j;
         T w;
         while ((j = ii->next(w)) != 0)
            if (i < j)  // 加入到边数组
               edge[++k] = weightedEdge<int> (i, j, w);
     }
   // 数组edge[1:e]初始化为最小堆(小根堆)
  minHeap<weightedEdge<T> > heap(1);
  heap.initialize(edge, e);
  fastUnionFind uf(n); //建立n个元素的并查集对象
```



Graph::kruskal 3/3

```
//按照权值的递增顺序来抽取边,然后决定选入或舍弃
k = 0;  //作为spanningTreeEdges中的游标
while (e > 0  && k < n - 1)
   {//生成树未完成 并且 尚有剩余边
    weightedEdge<T> x =  heap.top();
     heap.pop();
     e--;
     int a = uf.find(x.vertex1());
     int b = uf.find(x.vertex2());
     if  (a != b)
         {// 选择边x
           spanningTreeEdges[k++] =;
           uf.unite(a,b); x
         }
     }
        return (k == n - 1);
     }
```



Prim 算法

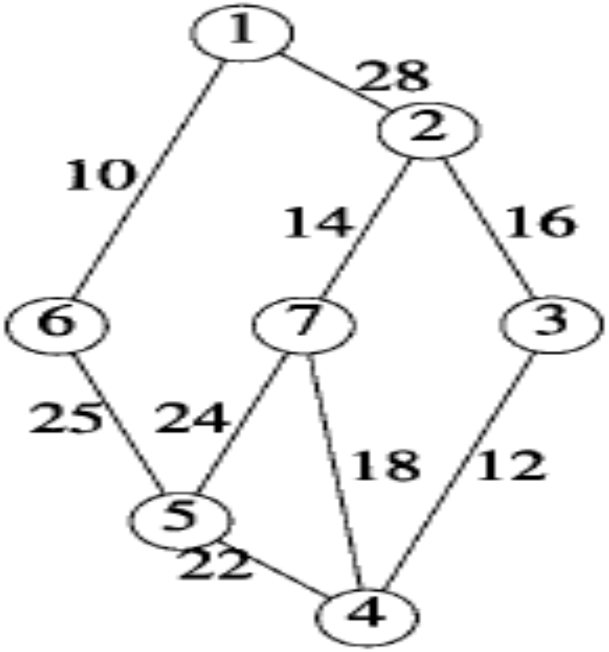
- Prim (普里姆)算法思想：
  - 从具有一个**单一顶点**(可以是原图中任意一个顶点)的树*T*开始
  - 重复地加一条边和一个顶点.
    - 往*T*中加入一条代价最小的边(u,v) 使 T∪{(u, v)}仍是一棵树.
    - ⇒对于边(u,v)， u 、 v 中正好有一个顶点位于*T*中.
  - 直到 T 中包含 n-1 条边为止.

Prim算法的伪代码

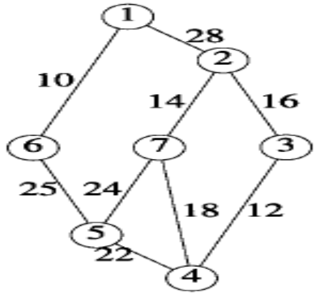
//假设网络中至少具有一个顶点
设 T 为所选择的边的集合，初始化 $T=\varnothing$
设 TV 为已在树中的顶点的集合，置 $TV=\{1\}$
令 E 为网络中边的集合
While $(E\neq\varnothing)$ & & $(|T|\neq n\text{-}1)$ {
    令(u, v)为最小代价边，其中$u \in TV, v \notin TV$
    If (没有这种边) break
    $E=E - \{(u,v)\}$ / /从E中删除此边
    在 T 中加入边(u, v)
    }
if $(|T|==n\text{-}1)$ T是一棵最小生成树
else 网络是不连通的，没有最小生成树

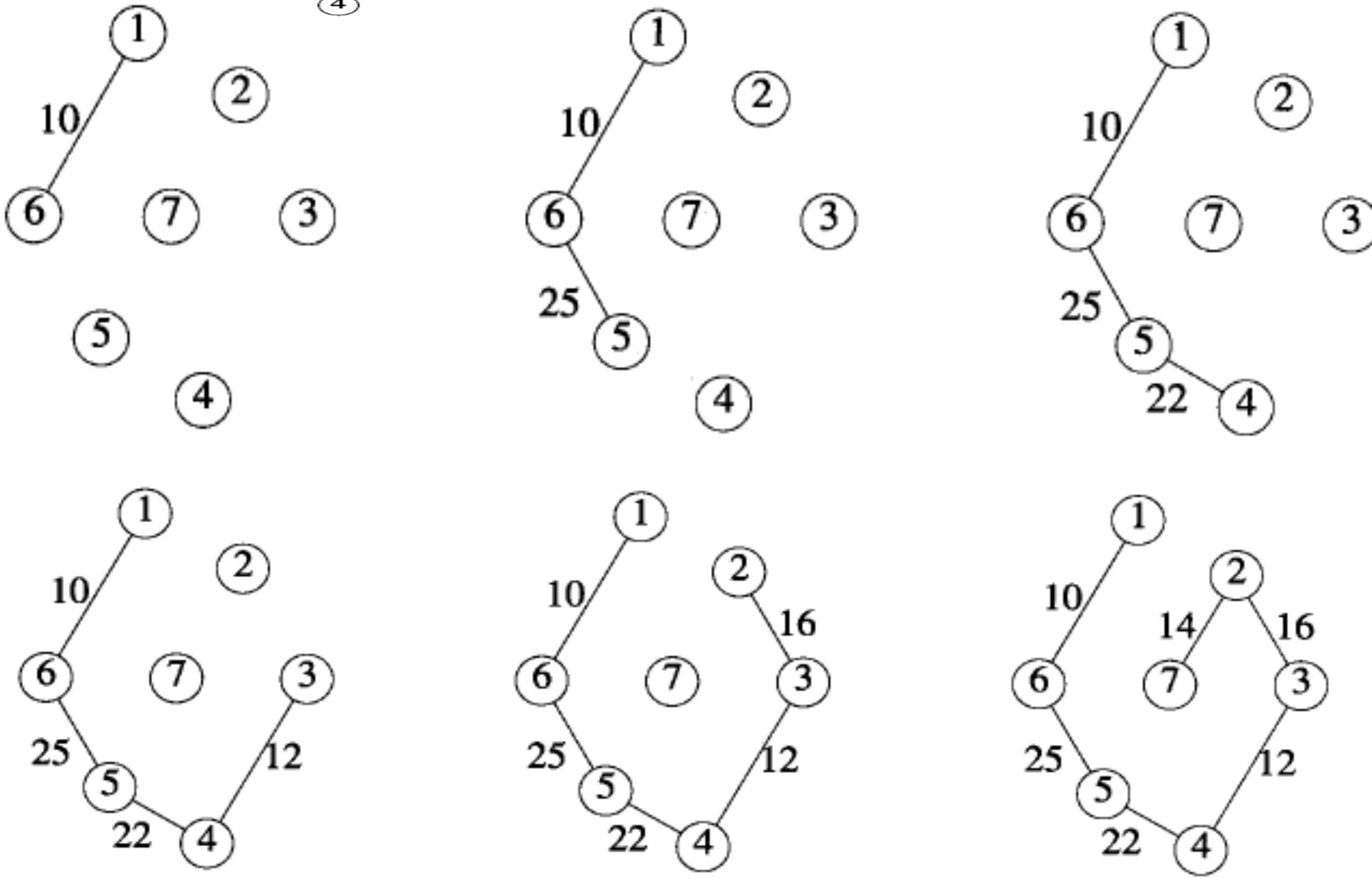
时间复杂性：$O(n^2)$

Prim算法示例



Prim算法示例

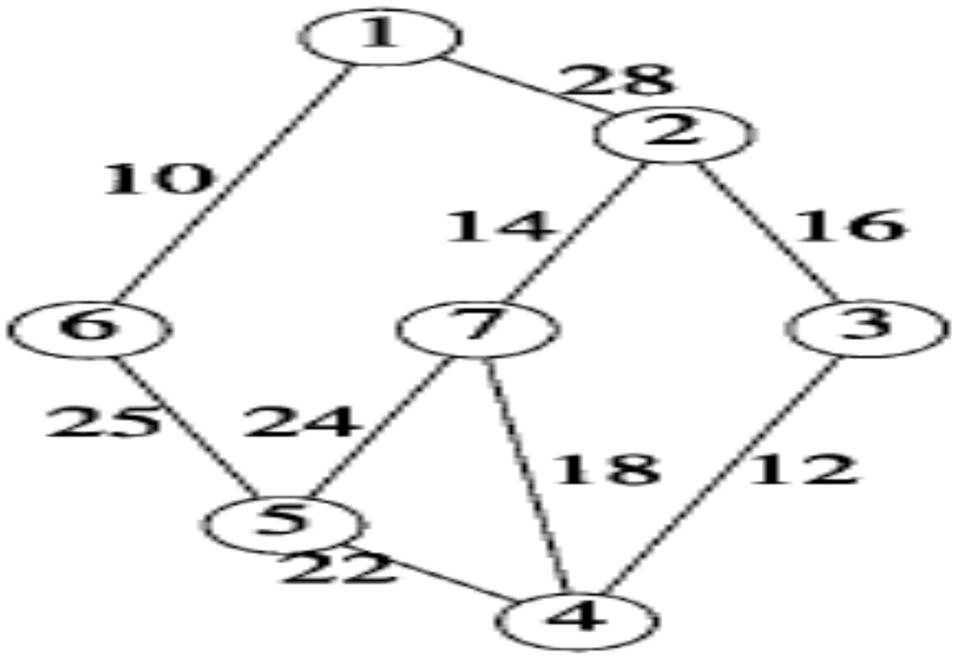




Sollin算法

- Sollin算法思想.
  - 从含有 n个顶点的森林开始.
  - 每一步中为森林中的每棵树选择一条边，这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。
    - 一个森林中的两棵树可选择同一条边。
    - 当有多条边具有相同的耗费时，两棵树可选择与它们相连的不同的边。
    - 丢弃重复的边和构成环路的边。
  - 直到仅剩下一棵树或没有剩余的边可供选择时算法终止。

Sollin算法示例



作业

- P444. 29