第4章 贪心算法

第4章 贪心算法

顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的**局部最优**选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

第4章 贪心算法

本章主要知识点：

- 4.1 活动安排问题
- 4.2 贪心算法的基本要素
- 4.3 最优装载
- 4.4 哈夫曼编码
- 4.5 单源最短路径
- 4.6 最小生成树
- 4.7 多机调度问题
- 4.8 贪心算法的理论基础

4.1 活动安排问题

活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合，是可以用贪心算法有效求解的很好例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。

4.1 活动安排问题

设有n个活动的集合E={1,2,…,n}，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动i都有一个要求使用该资源的起始时间si和一个结束时间fi,且si <fi 。如果选择了活动i，则它在半开时间区间[si, fi)内占用资源。若区间[si, fi)与区间[sj, fj)不相交，则称活动i与活动j是相容的。也就是说，当si≥fj或sj≥fi时，活动i与活动j相容。

4.1 活动安排问题

在下面所给出的解活动安排问题的贪心算法greedySelector ：

```
public static int greedySelector(int [] s, int [] f, boolean a[])
  {
    int n=s.length-1;
    a[1]=true;
    int j=1;
    int count=1;
    for (int i=2;i<=n;i++) {
     if (s[i]>=f[j]) {
       a[i]=true;
       j=i;
       count++;
       }
     else a[i]=false;
     }
    return count;
  }
```

各活动的起始时间和结束时间存储于数组s和f中且按结束时间的非减序排列

4.1 活动安排问题

由于输入的活动以其完成时间的**非减序**排列，所以算法**greedySelector**每次总是选择**具有最早完成时间**的相容活动加入集合A中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是**使剩余的可安排时间段极大化**，以便安排尽可能多的相容活动。

算法**greedySelector**的效率极高。当输入的活动已按结束时间的非减序排列，算法只需**O(n)**的时间安排n个活动，使最多的活动能相容地使用公共资源。如果所给出的活动未按非减序排列，可以用**O(nlogn)**的时间重排。

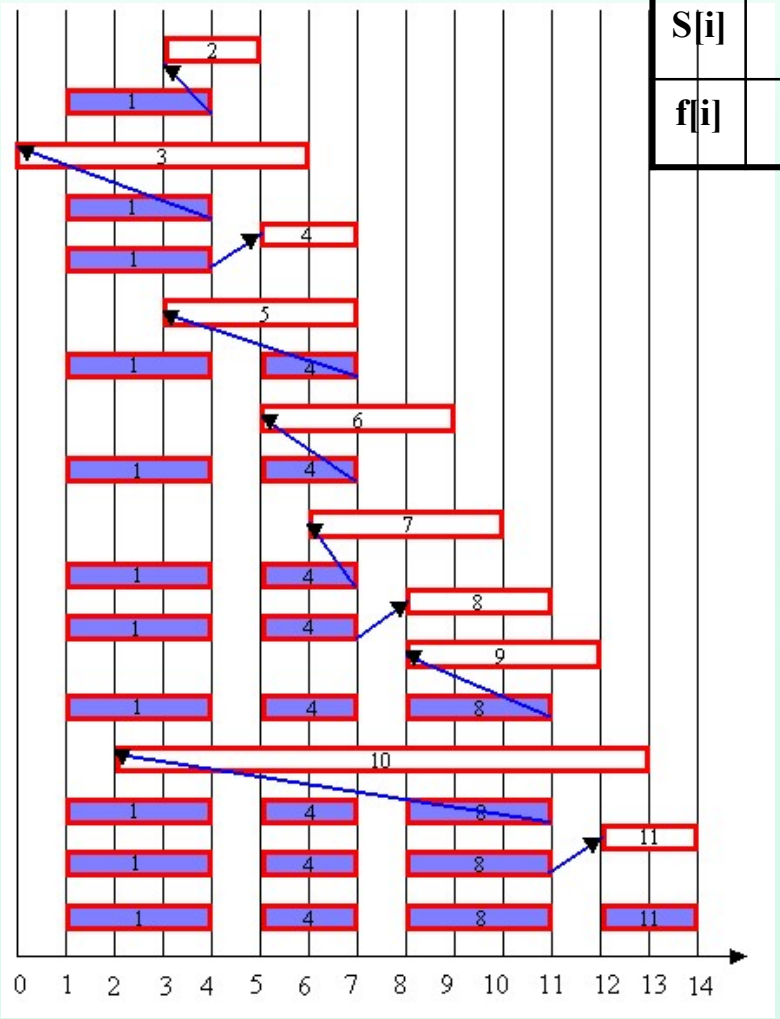
4.1 活动安排问题

例： 设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

4.1 活动安排问题

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14



算法greedySelector 的计算过程如左图所示。图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合A的活动，而空白长条表示的活动是当前正在检查相容性的活动。

4.2 贪心算法的基本要素

本节着重讨论可以用贪心算法求解的问题的一般特征。

对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢?这个问题很难给予肯定的回答。

但是，从许多可以用贪心算法求解的问题中看到这类问题一般具有2个重要的性质：**贪心选择性质**和**最优子结构性质**。

4.2 贪心算法的基本要素

1. 贪心选择性质

所谓**贪心选择性质**是指所求问题的**整体最优解**可以通过一系列**局部最优**的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

动态规划算法通常以**自底向上**的方式解各子问题，而贪心算法则通常以**自顶向下**的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

4.2 贪心算法的基本要素

2. 最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有**最优子结构性质**。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

4.2.2 活动安排问题的分析

若被检查的活动i的开始时间Si小于最近选择的活动j的结束时间fi，则不选择活动i，否则选择活动i加入集合A中。

贪心算法并不总能求得问题的**整体最优解**。但对于活动安排问题，贪心算法greedySelector却总能求得的整体最优解，即它最终所确定的相容活动集合A的规模最大。这个结论可以用数学归纳法证明。

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

1 贪心选择性质

即证明活动安排问题总存在一个最优解从贪心选择开始。

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

1 贪心选择性质

设E={1,2,…,n}为所给的活动集合。由于E中的活动按结束时间的非递减排序，故活动1具有最早完成时间。首先证明活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动1.

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

  1 贪心选择性质

  设A⊆E是所给活动安排问题的一个最优解，且A中的活动也按结束时间非递减排序，A中的第一个活动是k。

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

  1 贪心选择性质

  若k＝1，则A就是以贪心选择开始的最优解。

  若k>1，设B=A－{k}∪{1}。因为$f_1<=f_k$且A中的活动是相容的。古B中的活动也是相容的。又由于B中的活动个数与A中的活动个数相同，故A是最优的，B也是最优的。即B是以选择活动1开始的最优活动安排。

  由此可见，总存在以贪心选择开始的最优活动安排方案。

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

  2 最优子结构性质

  在作出了贪心选择，即选择了活动1后，原问题简化为对E中所有与活动1相容的活动进行活动安排的子问题。即若A是原问题的最优解，则A'=A-{1}是活动安排问题的$E'=\{i \in E: S_i \geq f_1\}$的最优解。

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

  2 最优子结构性质

  反证法：若E'中存在另一个解B'，比A'有更多的活动，则将1加入B'中产生另一个解B，比A有更多的活动。与A的最优性矛盾。

4.2.2 活动安排问题的分析

- 正确性证明（用数学归纳法证明）

  因此，每一步所作出的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。对贪心选择次数用归纳法可知，贪心算法greedySelector产生问题的最优解。

4.2 贪心算法的基本要素

3. 贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是2类算法的一个共同点。但是，对于具有**最优子结构**的问题应该选用贪心算法还是动态规划算法求解?是否能用动态规划算法求解的问题也能用贪心算法求解?下面研究2个经典的**组合优化问题**，并以此说明贪心算法与动态规划算法的主要差别。

4.2 贪心算法的基本要素

- 0-1背包问题（贪心算法不适合）

给定n种物品和一个背包。物品i的重量是Wi，其价值为Vi，背包的容量为C。应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

在选择装入背包的物品时，对每种物品i只有2种选择，即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i。

4.2 贪心算法的基本要素

- **背包问题：**

  与0-1背包问题类似，所不同的是在选择物品i装入背包时，**可以选择物品i的一部分**，而不一定要全部装入背包，$1 \le i \le n$。

这2类问题都具有**最优子结构**性质，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。

4.2贪心算法的基本要素

- 与动态规划算法比较:
  - 贪心算法通常用来于求解最优化问题，即量的最大化或最小化。
  - 然而，贪心算法不像动态规划算法，它通常包含一个用以寻找局部最优解的迭代过程。在某些实例中，这些局部最优解转变成了全局最优解，而在另外一些情况下，则无法找到最优解。

基本思想

- 贪心算法在少量计算的基础上做出正确猜想而不急于考虑以后的情况。
- 它一步步地来构筑解。
- 每一步均是建立在局部最优解的基础之上，而每一步又都扩大了部分解的规模。
- 做出的选择产生最大的直接收益而又保持可行性。
- 因为每一步的工作很少且基于少量信息，所得算法特别有效。

例子: (分数)背包问题

- 问题描述:
  - 给出n个大小为$s_1, s_2, \ldots, s_n$,值为$v_1, v_2, \ldots, v_n$的项目，并设背包容量为C,要找到非负实数$x_1, x_2, \ldots x_n$使和

$$\sum_{i=1}^{n} x_i v_i$$

  - 在约束

$$\sum_{i=1}^{n} x_i s_i \le C$$

  - 下最大

例子

- n=3, C=20, (v1,v2,v3)=(25,24,15),
  (s1,s2,s3)=(18,15,10)
- 可行方案:

(x1,x2,x3)	$\Sigma s_i x_i$	$\Sigma v_i x_i$
(1)(1/2,1/3,1/4)	16.5	24.5
(2)(1,2/15,0)	20	28.2
(3)(0,2/3,1)	20	31
(4)(0,1,1/2)	20	31.5

选择策略-1

n=3, C=20, (v1,v2,v3)=(25,24,15),

(s1,s2,s3)=(18,15,10)

- 从剩下的物品中选择最大价值的物品装入背包，得到：
- 方案(2)(1,2/15,0)　20　28.2 (非最优)
- 原因：虽然价值很大，但是容量的消耗太快

选择策略-2

n=3, C=20, (v1,v2,v3)=(25,24,15),

(s1,s2,s3)=(18,15,10)

- 从剩下的物品中选择最小尺寸的物品装入背包。
- 方案(3) (0,2/3,1)　20　31 (非最优)
- 原因：虽然容量损耗较少，但是价值增长速度太慢

选择策略-3

n=3, C=20, (v1,v2,v3)=(25,24,15),

(s1,s2,s3)=(18,15,10)

- 从剩下的物品中选择最大v_i/s_i比率的物品装入背包。

- Solution (4) (0,1,1/2)　　20　　31.5 (最优)

Greedy algorithm to solve Knapsack

- Step 1:
  - 每项计算$y_i=v_i/s_i$, 即该项值和大小的比
- Step 2:
  - 再按比值的降序来排序，从第一项开始装背包，然后是第二项，依次类推，尽可能地多放，直至装满背包。

算法Greedy-KNAPSACK

输入: 按照v(i)/s(i)比率降序排列的 n个物品的容量数组s[1..n]和物品价值数组v[1..n]; 背包的总容量C, 物品的数量n

输出: 最优贪心算法结果x[1..n]

```
for i ← 0 to n     {初始化x[i] }
      x[i] ← 0
end for
cu ← C
for i ← 0 to n    {物品按次序装入背包 }
      if s[i] > cu then
            exit
      end if
      x(i)← 1
      cu ←cu-s(i)   {剩余背包容量}
end for
if i ≤n then x(i) ←cu/s(i) end if {最后剩余背包容量中装入物品i的比率}
return x
```

贪心算法架构

```
Algorithm GREEDY(A,n)
    solution←Ø
    for i←1 to n do
        x←SELECT(A)
        if FEASIBLE(solution,x)
                then solution←UNION(solution,x)
        end if
    end for
return (solution)
```

4.2 贪心算法的基本要素

用贪心算法解背包问题的基本步骤：

首先计算每种物品单位重量的价值Vi/Wi，然后，依贪心选择策略，将尽可能多的**单位重量价值最高**的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

具体算法可描述如下页：

4.2 贪心算法的基本要素

```
public static float knapsack(float c,float [] w, float [] v,float [] x)
  {
    int n=v.length;
    Element [] d = new Element [n];
    for (int i = 0; i < n; i++) d[i] = new Element(w[i],v[i],i);
    MergeSort.mergeSort(d);
    int i;
    float opt=0;
    for (i=0;i<n;i++) x[i]=0;
    for (i=0;i<n;i++) {
      if (d[i].w>c) break;
      x[d[i].i]=1;
      opt+=d[i].v;
      c-=d[i].w;
      }
    if (i<n){
      x[d[i].i]=c/d[i].w;
      opt+=x[d[i].i]*d[i].v;
      }
    return opt;
  }
```

算法knapsack的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为O（nlogn）。当然，为了证明算法的正确性，还必须证明背包问题具有贪心选择性质。

4.2 贪心算法的基本要素

对于**0-1背包问题**，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用**动态规划算法**求解的另一重要特征。

实际上也是如此，动态规划算法的确可以有效地解0-1背包问题。

贪心算法特点

- 算法由一个简单的迭代过程构成，在维持可行性的前提下它选择能产生最大直接利益的项。

- 贪心算法产生全局最优解的最重要因素是选择策略。

贪心算法基本要素

- 对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢?这个问题很难给予肯定的回答。

- 但是，从许多可用贪心算法求解的问题中我们看到这类问题一般具有2个重要的性质：贪心选择性质和最优子结构性质。

贪心选择性质

- 所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。
- 动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。
- 对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

最优子结构性质

- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

- 贪心算法和动态规划算法都要求问题具有最优子结构性质，这是2类算法的一个共同点。但是，对于具有最优子结构的问题应该选用贪心算法还是动态规划算法求解？是否能用动态规划算法求解的问题也能用贪心算法求解？

4.3 最优装载

有一批集装箱要装上一艘载重量为c的轮船。其中集装箱i的重量为Wi。最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

1. 算法描述

最优装载问题可用贪心算法求解。采用重量最轻者先装的贪心选择策略，可产生最优装载问题的最优解。具体算法描述如下页。

4.3 最优装载

```
public static float loading(float c, float [] w, int [] x)
  {
    int n=w.length;
    Element [] d = new Element [n];
    for (int i = 0; i < n; i++)
      d[i] = new Element(w[i],i);
    MergeSort.mergeSort(d);
    float opt=0;
    for (int i = 0; i < n; i++) x[i] = 0;
    for (int i = 0; i < n && d[i].w <= c; i++) {
      x[d[i].i] = 1;
      opt+=d[i].w;
      c -= d[i].w;
      }
    return opt;
  }
```

其中Element类说明为参见本书P115

4.3 最优装载

2. 贪心选择性质

需要--证明最优装载问题具有贪心选择性质。

3. 最优子结构性质

需要--证明最优装载问题具有最优子结构性质。

4.3 最优装载

- 先证明贪心选择性质：设集装箱已依其重量从大到小排序，（x1,x2.....xn）是最优装载问题的一个最优解。又设k=min{i|xi=1}{1<=i<=n}.易知，如果给定的最优装载问题有解，则1<=k<=n;

(1)当$k=1$时，$(x_1,x_2,\ldots,x_n)$是满足贪心选择性质的最优解。

(2)当$k>1$时，取$y_1=1, y_k=0, y_i=x_i, 1<i\le n, i\neq k$，则

$$\sum_{i=1}^{n} w_i y_i = w_1 - w_k + \sum_{i=1}^{n} w_i x_i \le \sum_{i=1}^{n} w_i x_i \le c$$

$(y_1,y_2,\ldots,y_n)$是所给最优装载问题的可行解。

$\Rightarrow$ 又因为$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} x_i$

$\Rightarrow (y_1,y_2,\ldots,y_n)$是满足贪心选择性质的最优解。

得以证明该问题具备贪心选择性质。

4.3 最优装载

- 证明该问题具备最优子结构性质：设(x1,x2....xn)是最优装载的满足贪心选择性质的最优解，易知，x1=1，(x2,x3.....xn)是轮船载重量为c-w1，待装船集装箱为{2,3，.....n}时相应的最优装载问题的最优解。得以证明，该问题具备最优子结构性质。

4.3 最优装载

2. 贪心选择性质--

证明了最优装载问题具有贪心选择性质。

3. 最优子结构性质---

证明了最优装载问题具有最优子结构性质。

由最优装载问题的贪心选择性质和最优子结构性质，容易证明算法**loading**的正确性。

算法**loading**的主要计算量在于将集装箱依其重量从小到大排序，故算法所需的计算时间为 **O(nlogn)**。

4.4 哈夫曼编码

哈夫曼编码是广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在20%～90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用0，1串表示各字符的最优表示方式。

给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长。

1. 前缀码

对每一个字符规定一个0,1串作为其代码，并要求任一字符的代码都不是其他字符代码的前缀。这种编码称为**前缀码**。

文件编码

11001100 11001100

	a	e	t	r	t	d
频度(千字)	45	13	12	16	9	5
等长编码	000	001	010	011	100	101
变长编码1	0	001	010	110	1101	1100
变长编码2	0	101	100	111	1101	1100

前缀编码

4.4 哈夫曼编码

编码的前缀性质可以使译码方法非常简单。

表示**最优前缀码**的二叉树总是一棵**完全二叉树**，即树中任一结点都有2个儿子结点。

平均码长定义为：$B(T)=\sum_{c\in C} f(c)d_T(c)$

关于定长码与变长码的编码效率

信源空间 $\begin{bmatrix} X \\ P \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{8} \end{bmatrix}$

$$H(X) = -(1/2\log 1/2 + 1/4\log 1/4 + 1/8\log 1/8 + 1/8\log 1/8)$$

$$= 1.75\text{bit/消息}$$

信源符号个数为$n=4$，二元码符$\{0, 1\}$，码符个数为$m=2$，

K_i为信源各符号对应的码字长

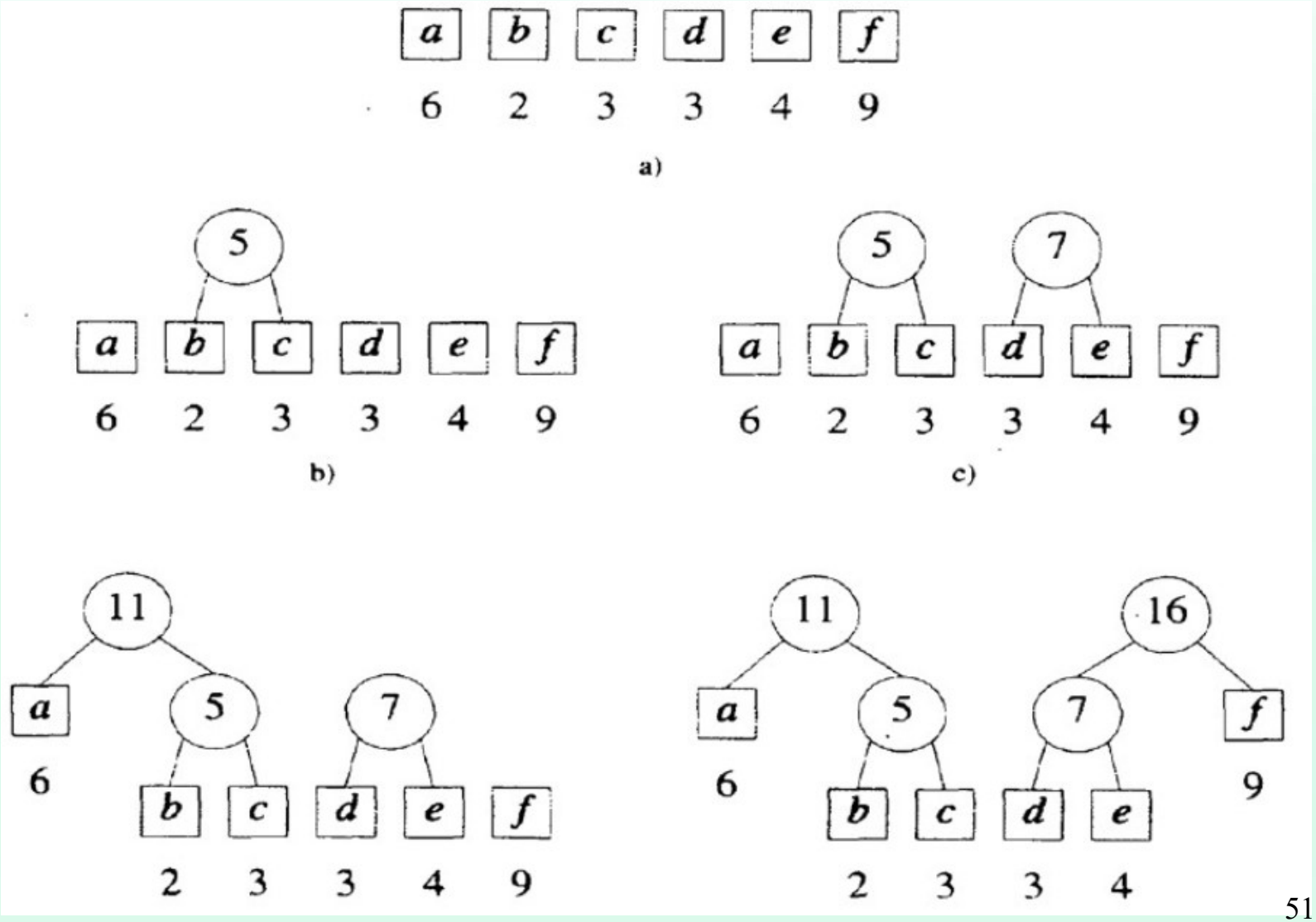
- 定长码：$K_1 = K_2 = K_3 = K_4 = 2$　　$2^{-2}+2^{-2}+2^{-2}+2^{-2}=1$

  码字：$Y_1=00$，$Y_2=01$，$Y_3=10$，$Y_4=11$

- 变长码：$K_1=1$, $K_2=2$, $K_3=3$, $K_4=3$　　$2^{-1}+2^{-2}+2^{-3}+2^{-3}=1$

  码字：$Y_1=0$，$Y_2=10$，$Y_3=110$，$Y_4=111$

4.4 哈夫曼编码


a b c d e f
6 2 3 3 4 9
a)
5
a b c d e f
6 2 3 3 4 9
b)
5 7
a b c d e f
6 2 3 3 4 9
c)
11
a 5 7
6 b c d e f
2 3 3 4 9
11 16
a 5 7 f
6 b c d e
9
2 3 3 4

4.4 哈夫曼编码

使平均码长达到最小的前缀码编码方案称为给定编码字符集C的**最优前缀码**。

2. 构造哈夫曼编码

哈夫曼提出构造最优前缀码的贪心算法，由此产生的编码方案称为**哈夫曼编码**。

哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树T。

算法以|C|个叶结点开始，执行|C| - 1次的“合并”运算后产生最终所要求的树T。

```
Public static bintree huffmanTree( float[ ]f)
{
  //生成单结点树
  int n=f.length;
  Huffman [ ]w=new Huffman[n+1];
  Bintree zero=new Bintrss( );
  for (int i=0;i<n;i++)
  {
   Bintree x=new Bintree( );
   x.makeTree(new MyInteger(i),zero,zero);
   w[i+1]=new Huffman(x,f[i]);
  }
  //建优先队列
  MinHeap H=new MinHeap( );
   H.initiaize(w,n);
  //反复合并最小频率树
  for(int i=1;i<n;i++)
  {
   Huffman x=(Huffman)H.removeMin( );
   Huffman y=(Huffman)H.removeMin( );
   Bintree z=new Bintree( );
   z.makeTree(null, x.tree, y.tree);
   Huffman t=new Huffman(z, x.weight+y.weight);
   H.put(t);
  }
  return ((Huffman)H.removeMin()).tree;
}
```

4.4 哈夫曼编码

在书上给出的算法huffmanTree中，编码字符集中每一字符c的频率是f(c)。**以f为键值的优先队列Q**用在**贪心选择**时有效地确定算法当前要合并的2棵具有最小频率的树。一旦2棵具有最小频率的树合并后，产生一棵新的树，其频率为合并的2棵树的频率之和，并将新树插入优先队列Q。经过n - 1次的合并后，优先队列中只剩下一棵树，即所要求的树T。

算法huffmanTree用最小堆实现优先队列Q。初始化优先队列需要O(n)计算时间，由于最小堆的removeMin和put运算均需O(logn)时间，n - 1次的合并总共需要O(nlogn)计算时间。因此，关于n个字符的哈夫曼算法的**计算时间**为O(nlogn) 。

4.4 哈夫曼编码

3. 哈夫曼算法的正确性

要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有**贪心选择性质**和**最优子结构性质**。

(1) 贪心选择性质

(2) 最优子结构性质

4.5 单源最短路径

一、问题的提法及应用背景

(1）问题的提法——给定带权有向图G=(V,E)，其中每条边的权是非负实数。另外，还给定V中的一个顶点，称为源。现在要计算从源到所有其他各顶点的最短路长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

(2）应用背景——管道铺设、线路安排、厂区布局、设备更新等。

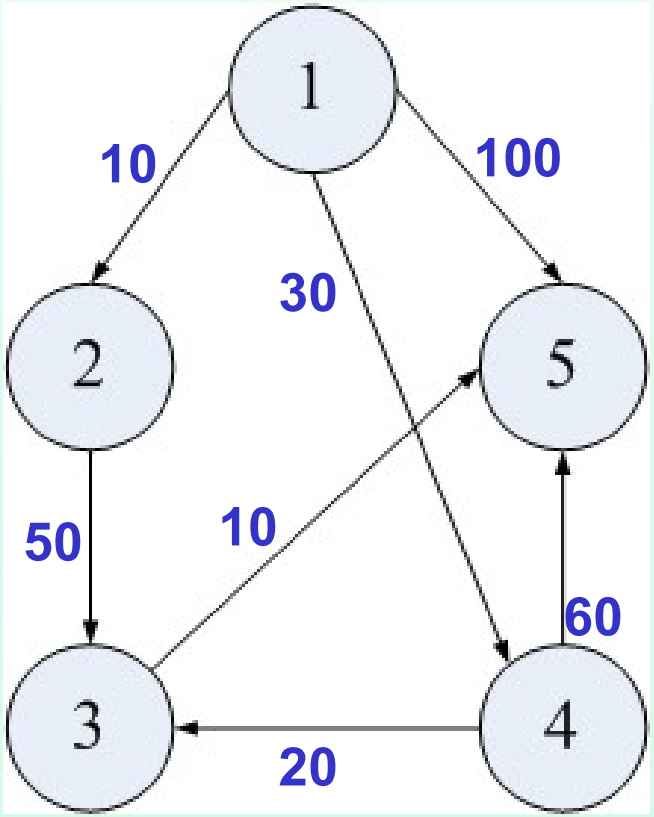
二、 Dijkstra(迪科斯彻)算法基本思想

其**基本思想**是，设置顶点集合S并不断地作**贪心选择**来扩充这个集合。一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

初始时，S中仅含有源。设u是G的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u的**特殊路径**，并用**数组dist**记录当前每个顶点所对应的最短特殊路径长度。Dijkstra算法每次从V-S中取出具有最短特殊路长度的顶点u，将u添加到S中，同时对数组dist作必要的修改。一旦S包含了所有V中顶点，dist就记录了从源到所有其他顶点之间的最短路径长度。

三、算法应用

例： 对右图中的有向图，应用Dijkstra算法计算从源顶点1到其他顶点间最短路径。



dist[i] 表示当前从源到顶点i的最短特殊路径长度

a[i][j] 表示边（i,j）的权

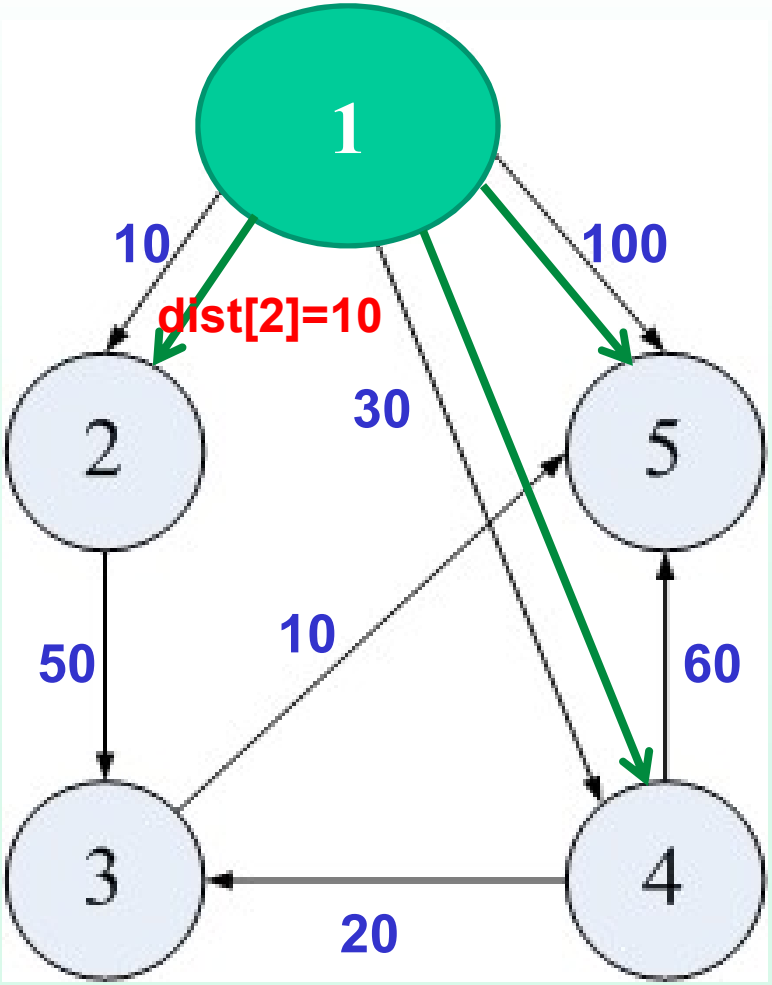
S={1}

dist[2]=0+10=10

dist[3]=0+∞=∞

dist[4]=0+30=30

dist[5]=0+100=100



min{dist[2], dist[4], dist[5]}= {10,30,100}=10

S={1, 2}　dist[2]=10

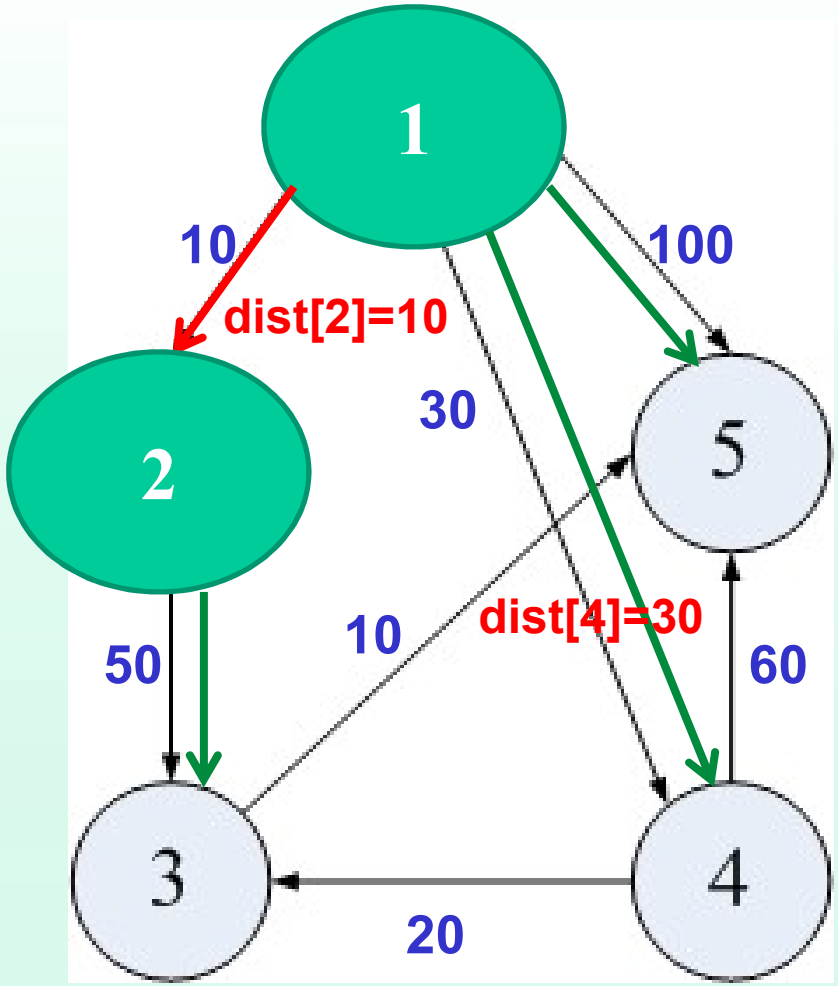
S={1, 2}

dist[2]+a[2][3]=60< ∞

dist[3]=60

dist[4]=0+30=30

dist[5]=0+100=100



min{dist[3], dist[4], dist[5]}= {60,30,100}=30

S={1, 2, 4} dist[4]=30

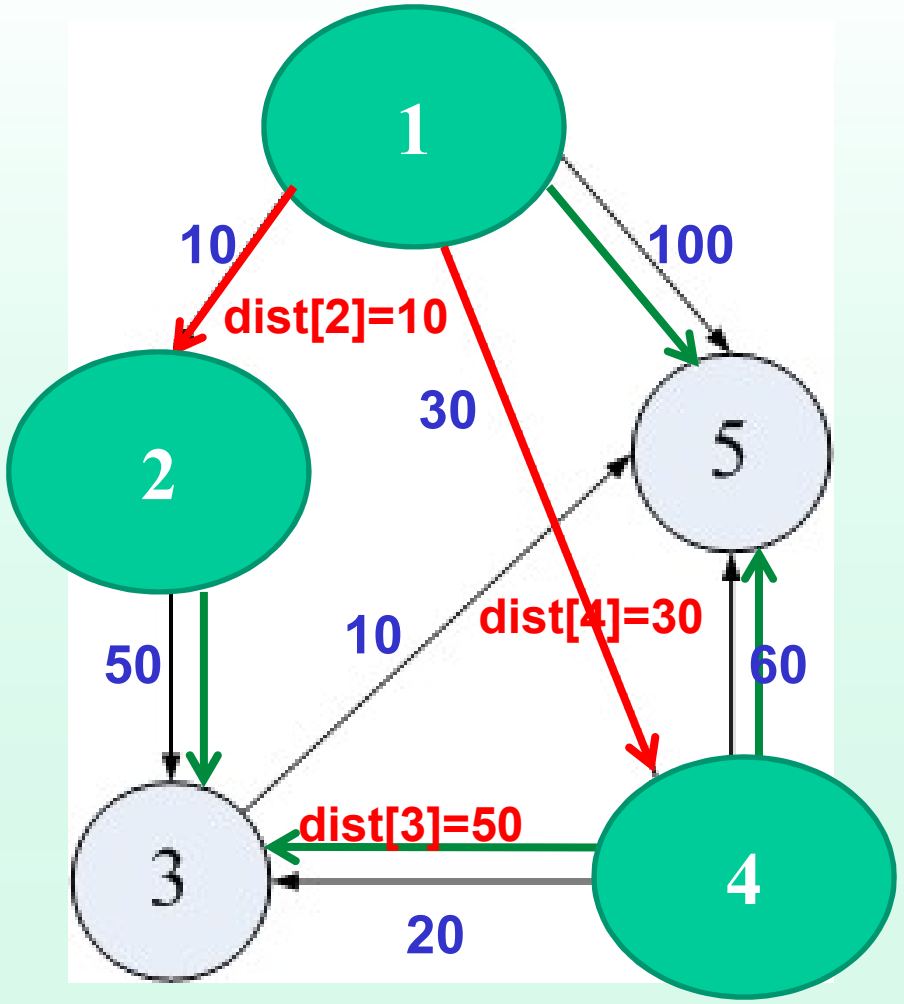
S={1, 2, 4}

dist[4]+a[4][3]=50< 60

dist[3]=50

dist[4]+c[4][5]=90<100

dist[5]=90



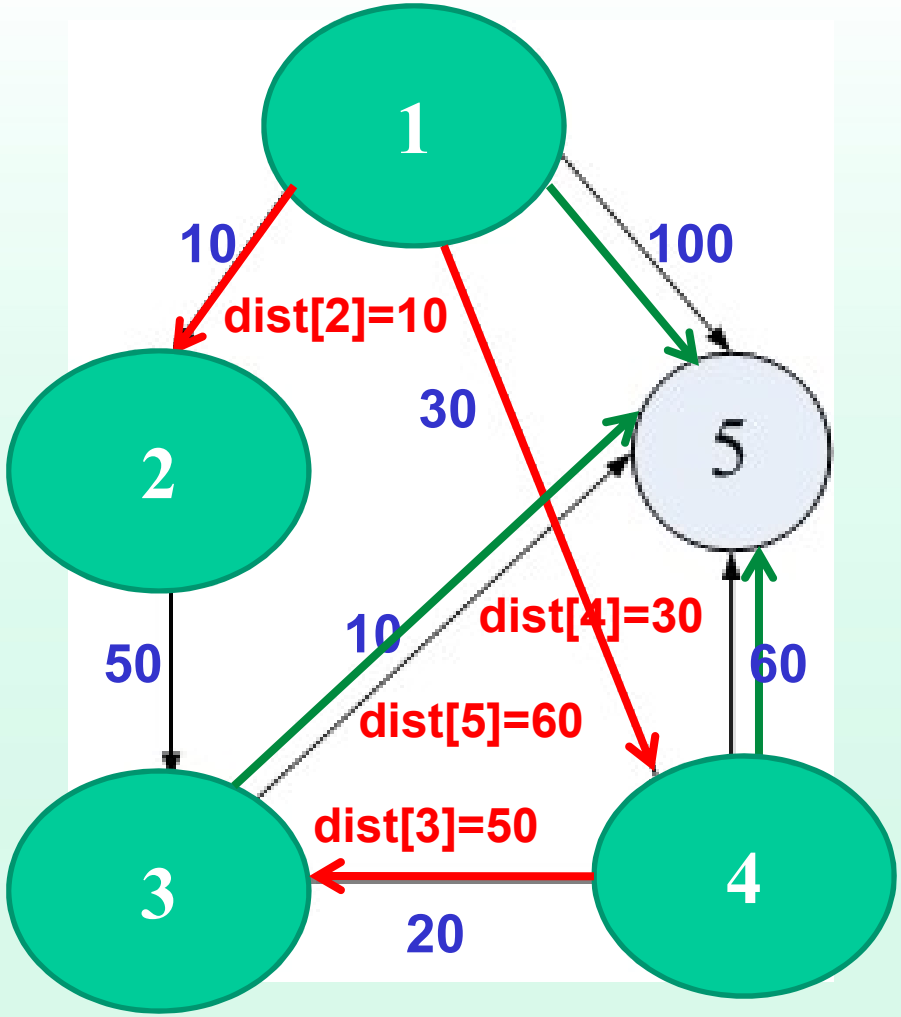
min{dist[3], dist[5]}=min{30+20,30+60} ={50,90}=50

S={1, 2, 4, 3} dist[3]=50

S={1, 2, 4, 3}

dist[3]+a[3][5]=60< 90

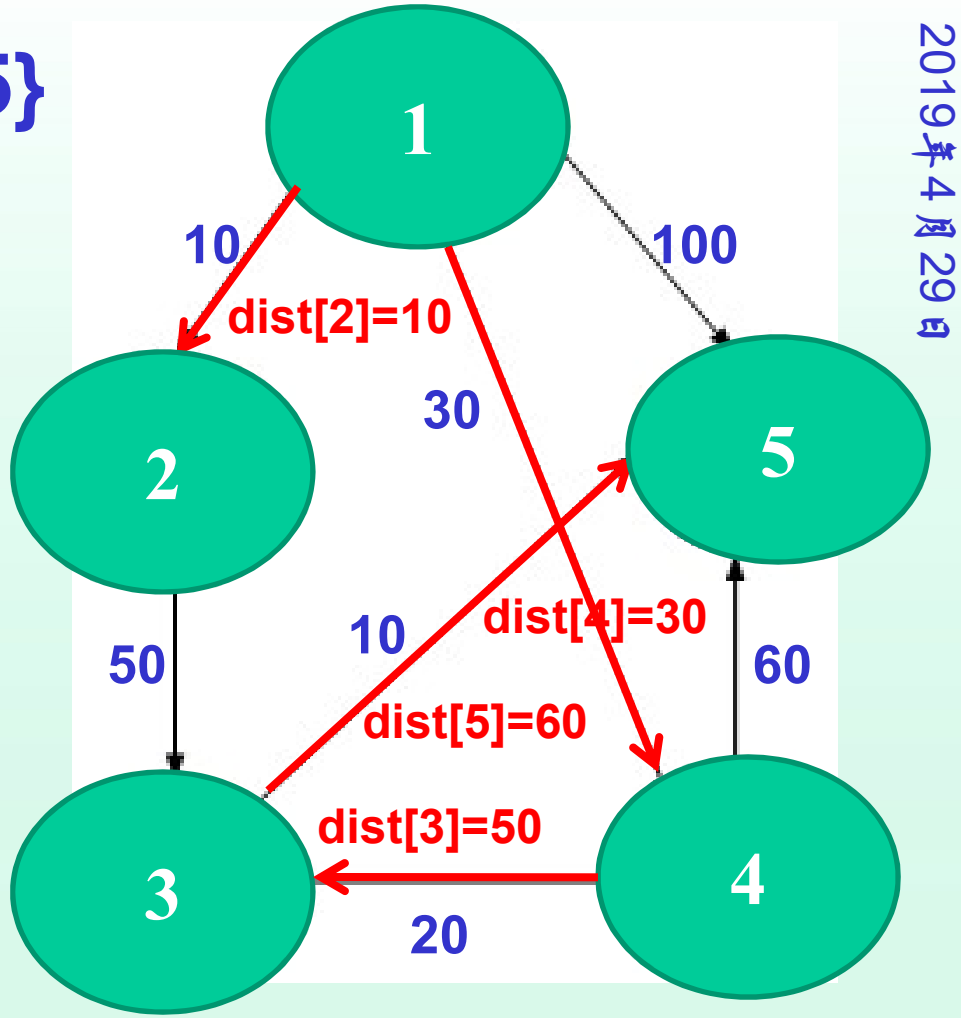
dist[5]=60



min{dist[5]}=min{50+10} ={60}=60

S={1, 2, 4, 3, 5} dist[5]=60

S={1, 2, 4, 3, 5}



源点1到其它各点的最短距离分别为：

Dist[2]=10　dist[3]=50　dist[4]=30　dist[5]=60

Dijkstra 算法的迭代过程

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

四、代码实现

```
public  static void  dijkstra (int v, float [][] a, float [] dist, int [] prev)
{
     int n=dist.length-1;
     boolean  []  s=new boolean[n+1];
    int i,j;
     if(v<1  ||  v>n)  return;
     /*初始设置*/
     for(i=1;i<=n;i++)
     {
          dist[i]=a[v][i];  /*初始时从源点到i的最短路径设为从源点到i的权值*/
          s[i]=false;  /*i不在S集合中*/
          if(dist[i]==Float.MAX_VALUE)
               prev[i]=0;  /*从源点到i没有通路*/
          else
               prev[i]=v;  /*从源点到i有通路时，最短路径前一个结点设为源点*/
     }
```

```
dist[v]=0; /*源点到源点的最短路径为0*/
s[v]=true; /*源点v在集合S中，为下面循环中源点不参加比较做准备*/
 /*中心部分*/
 for(i=1;i<n;i++)
 {
      float temp=Float.MAX_VALUE;
      int u=v;
      /*找出一个剩余结点中到源点最短的结点*/
      for(j=1;j<=n;j++)
          if((!s[j]) && (dist[j]<temp)) /*如果该点不是源点并且源点到j点路径是最短*/
          {
              u=j; /*u记录最短路径的点*/
              temp=dist[j]; /*记录源点到j点的最短路径*/
          }
          s[u]=true; /*u点是下面进行比较的点*/
```

```
/*找出通过u点是否有更短的路径*/
  for(j=1;j<=n;j++)
      if((!s[j]) && (a[u][j]<Float.MAX_VALUE))
      {
        float newdist=dist[u]+a[u][j];
        if(newdist<dist[j]) /*源点到u的路径+u到j的路径<源点到j的路径*/
          {
             dist[j]=newdist; /*更改源点到j的路径长度*/
             prev[j]=u; /*更改源点到j的最短路径中，j的前一个结点*/

          }
        }
}
```

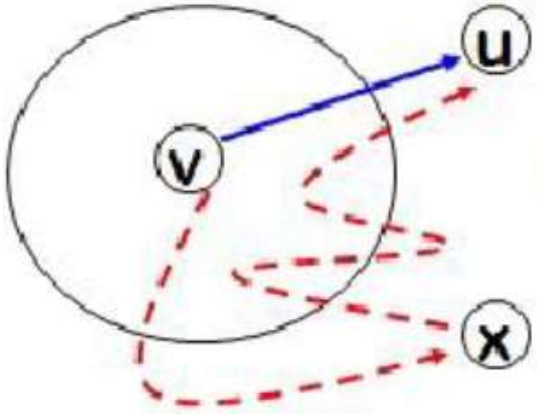
五、算法的正确性和计算复杂性

1、贪心选择性质

Dijkstra算法是应用贪心算法设计策略的又一个典型例子。它所做出的贪心选择是从V—S中选择具有最短特殊路径的顶点u，从而确定从源到u的最短路径长度dist[u]。这种贪心选择为什么能导致最优解呢?

假设存在一条从源到u且长度比dist[u]更短的路。设这条路初次走出S之外到达的顶点为x∈ V—S，然后徘徊于S内外若干次，最后离开S到达u。

- 在这条路上分别记d(v,x),d(x,u)和d(v,u)为顶点v到顶点x,顶点x到顶点u，顶点v到顶点u的路长。则有：



$$d(v,x)+d(x,u)=d(v,u)\le dist[u]$$

$$d(x,u)\ge 0,$$

$$dist[x]\le d(v,x)\le dist[u]$$

dist[x]<=dist[u]与 u 是当前贪心选择矛盾！

2、最优子结构性质

要完成Dijkstra算法正确性的证明，还必须证明最优子结构性质，即算法中确定的dist[u]确实是当前从源到顶点u的最短特殊路径长度。

如果P(i,j)={Vi….Vk..Vs…Vj}是从顶点i到j的最短路径，k和s是这条路径上的一个中间顶点，那么P(k,s)必定是从k到s的最短路径。下面证明该性质的正确性。

假设P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，则有：

$$P(i,j)=P(i,k)+P(k,s)+P(s,j)$$

而P(k,s)不是从k到s的最短距离，那么必定存在另一条从k到s的最短路径P‘(k,s)，那么：

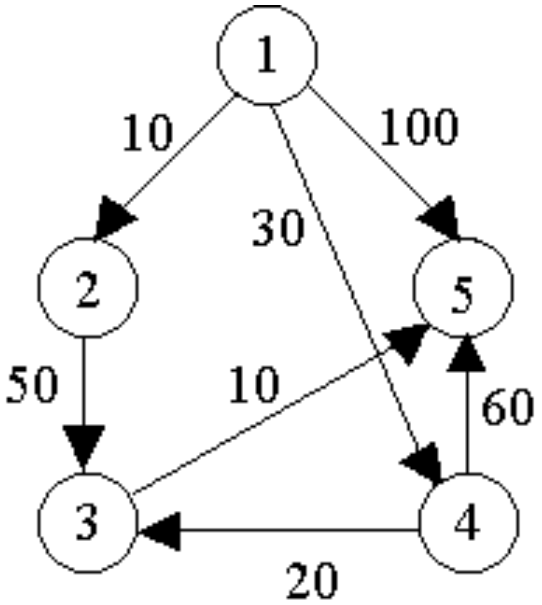
$$P'(i,j)=P(i,k)+P'(k,s)+P(s,j)<P(i,j)$$

则与P(i,j)是从i到j的最短路径相矛盾。因此该性质得证。

3、计算复杂性

对于具有n个顶点和e条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要$O(n)$时间。这个循环需要执行n-1次，所以完成循环需要$O(n^2)$时间。算法的其余部分所需要时间不超过$O(n^2)$。

Dijkstra算法的迭代过程：



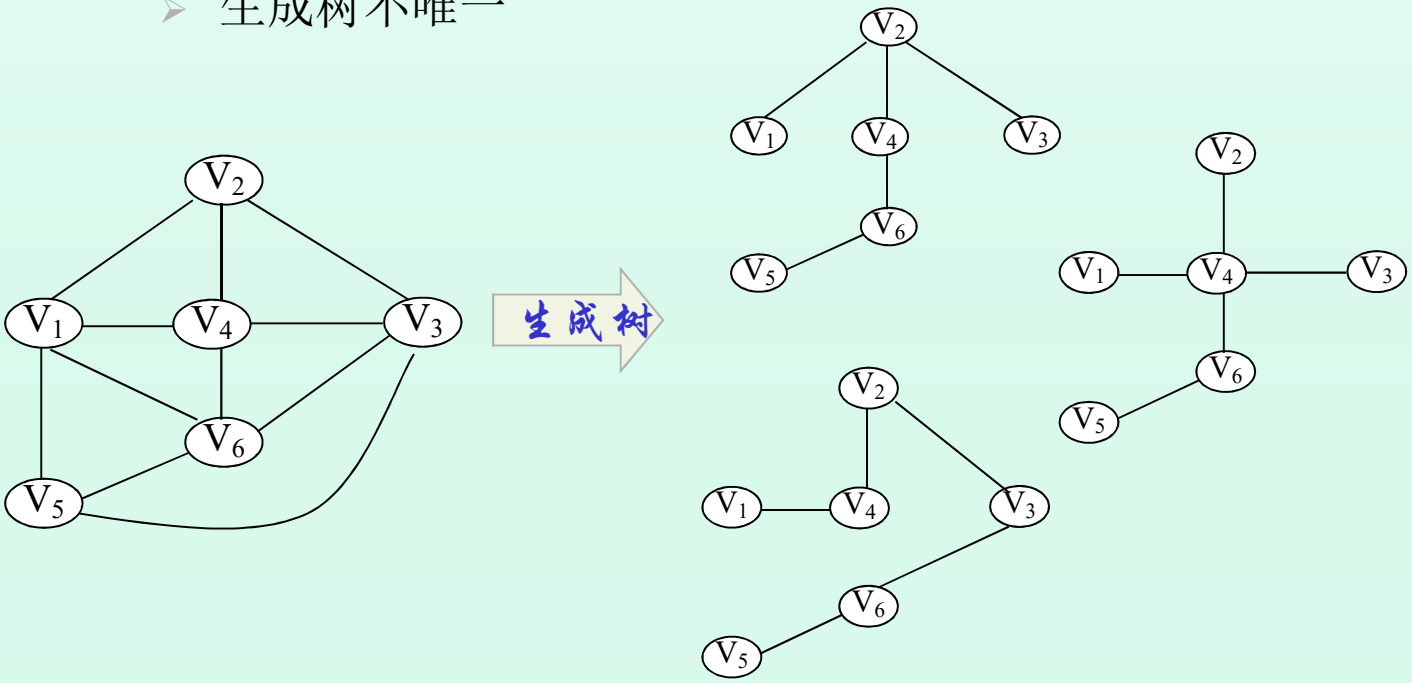
迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

4.6 最小生成树

设G =(V,E)是无向连通带权图，即一个**网络**。E中每条边(v,w)的权为c[v][w]。如果G的子图G'是一棵包含G的所有顶点的树，则称G'为G的生成树。生成树上各边权的总和称为该生成树的**耗费**。

生成树的概念

- 生成树
  - 一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的n-1条边。
  - 生成树不唯一



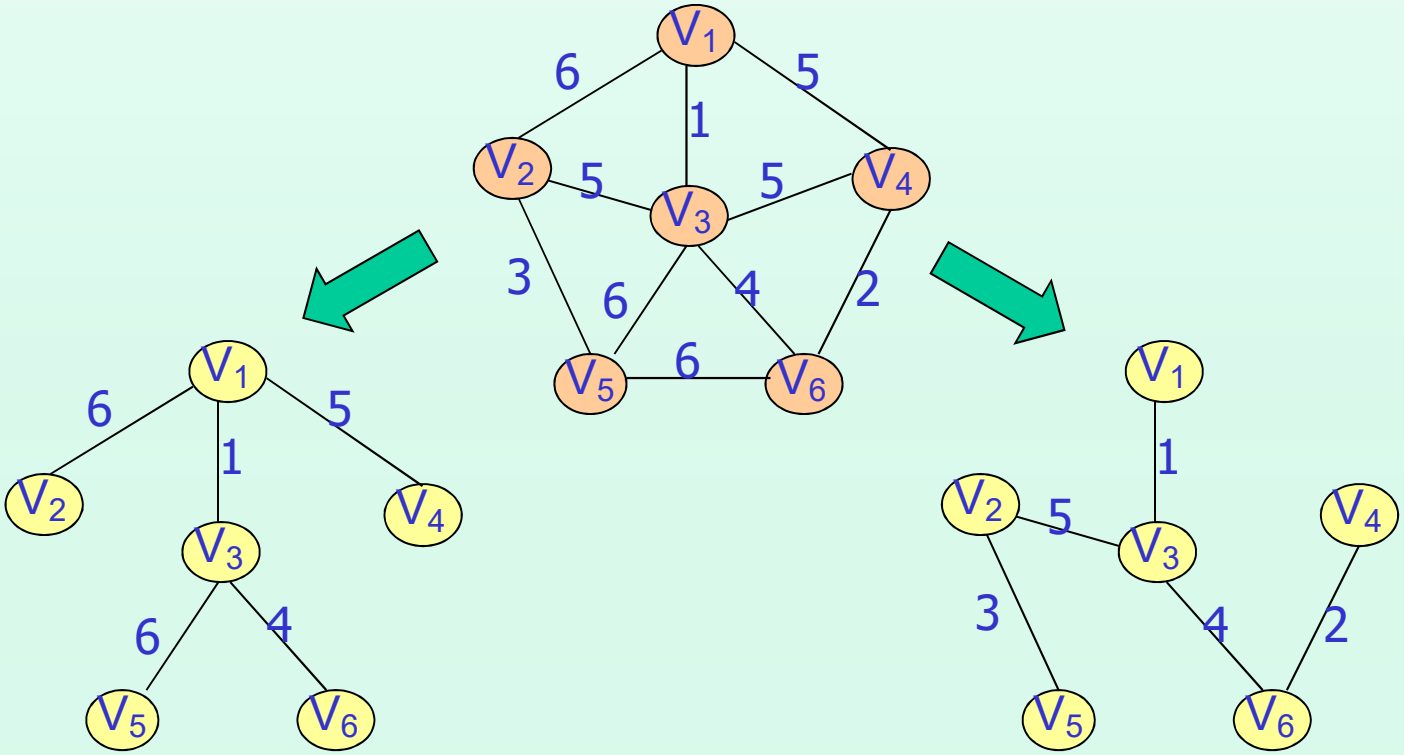
4.6 最小生成树

在G的所有生成树中，耗费最小的生成树称为G的**最小生成树**。

网络的最小生成树在实际中有广泛应用。**例如**，在设计通信网络时，用图的顶点表示城市，用边(v,w)的权c[v][w]表示建立城市v和城市w之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案。

最小(代价)生成树

- 生成树的代价等于其边上的权值之和。



4.6 最小生成树

1. 最小生成树性质

用贪心算法设计策略可以设计出构造最小生成树的有效算法。本节介绍的构造最小生成树的**Prim算法**和**Kruskal算法**都可以看作是应用贪心算法设计策略的例子。尽管这2个算法做贪心选择的方式不同，它们都利用了下面的**最小生成树性质**：

设G=(V,E)是连通带权图，U是V的真子集。如果(u,v)∈E，且u∈U，v∈V-U，且在所有这样的边中，(u,v)的权c[u][v]最小，那么一定存在G的一棵最小生成树，它以(u,v)为其中一条边。这个性质有时也称为**MST性质**。

最小生成树算法

------prim& Kruskal

最小代价生成树

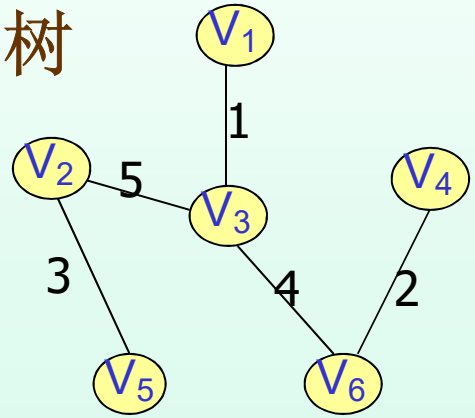
- 两种常用的构造最小生成树的方法：
  - 普里姆算法（prim）
  - 克鲁斯卡尔算法（ Kruskal）

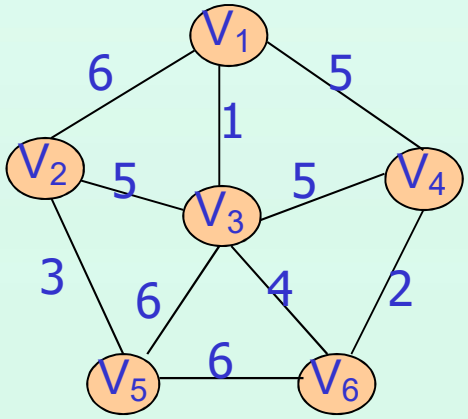
普里姆(Prim)算法

- 假设N=(V，E)是连通网，TE是N上最小生成树中边的集合。
- 算法从U={u_0}(u_0∈V)，TE={}开始，重复执行下述操作：
  - 在所有u∈U，v∈V-U的边(u，v)中找一条代价最小的边(u_0，v_0)，将其并入集合TE，同时将v_0并入U集合。
  - 当U=V则结束，此时TE中必有n-1条边，则T=(V，{TE})为**N**的最小生成树。
- 普里姆算法构造最小生成树的过程是从一个顶点U={u_0}作初态，不断寻找与U中顶点相邻且代价最小的边的另一个顶点，扩充到U集合直至U=V为止。

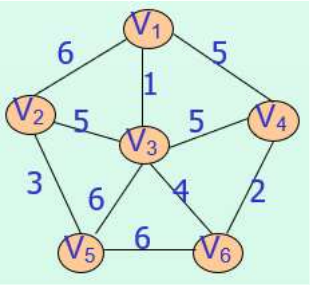
最小代价生成树

- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止

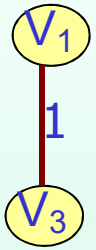




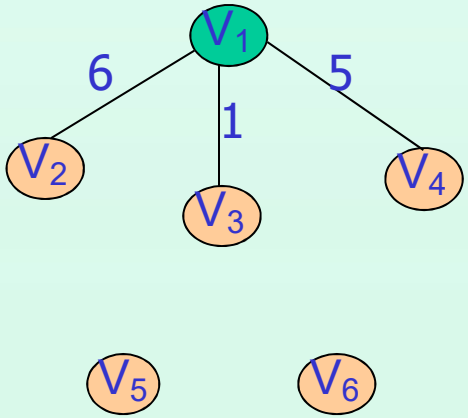
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$
(4)	$\{V_1, V_3, V_6, V_4, V_2\}$	$\{V_5\}$
(5)	$\{V_1, V_3, V_6, V_4, V_2, V_5\}$	$\{\}$



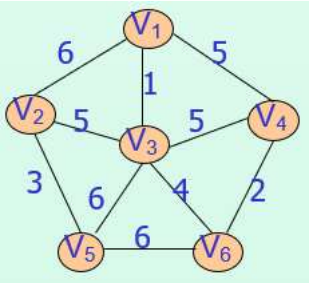
最小代价生成树



- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止

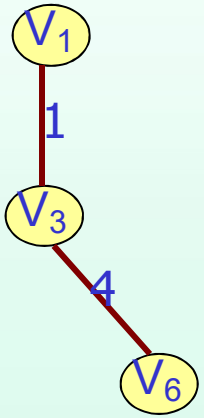


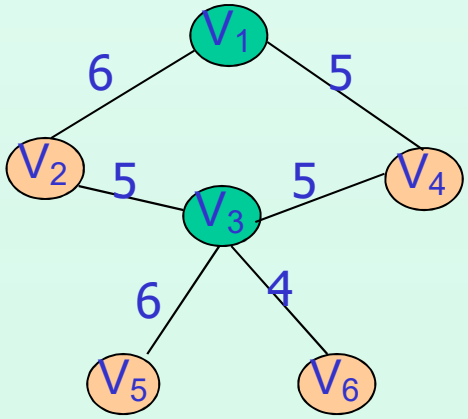
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$



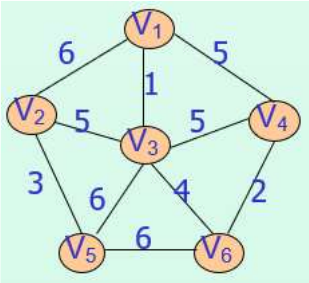
最小代价生成树

- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止



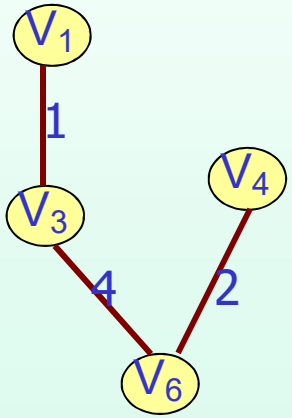


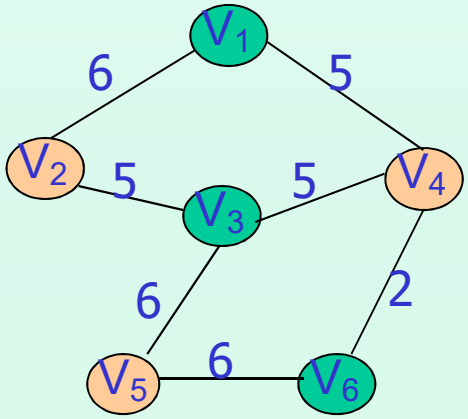
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$



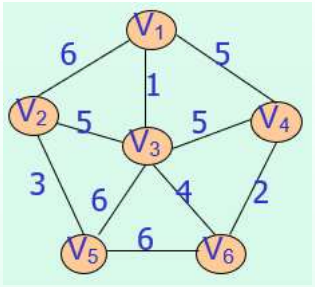
最小代价生成树

- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止



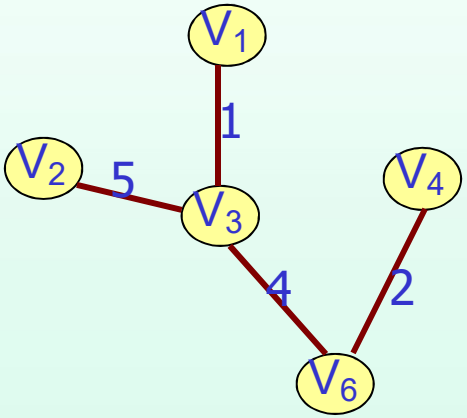


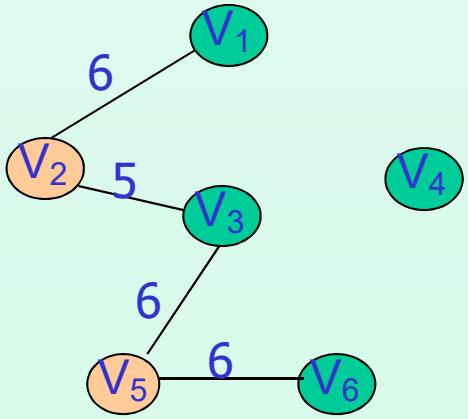
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$



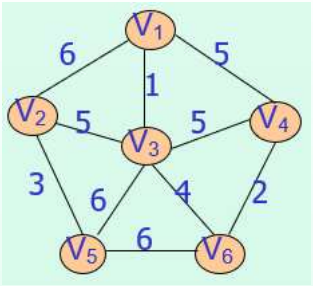
最小代价生成树

- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止



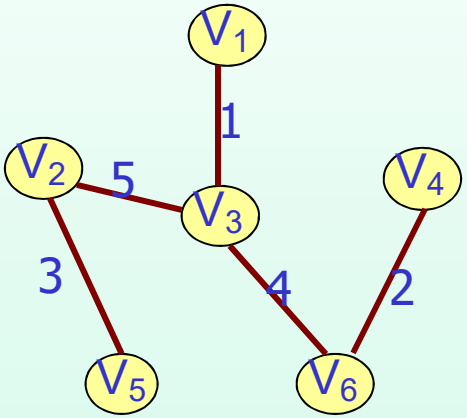


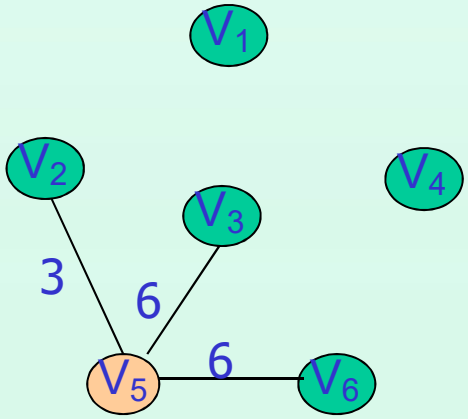
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$
(4)	$\{V_1, V_3, V_6, V_4, V_2\}$	$\{V_5\}$



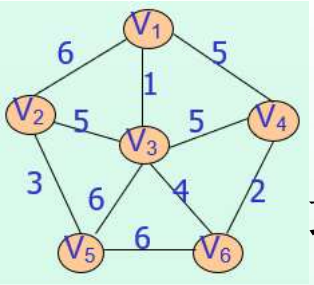
最小代价生成树

- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止



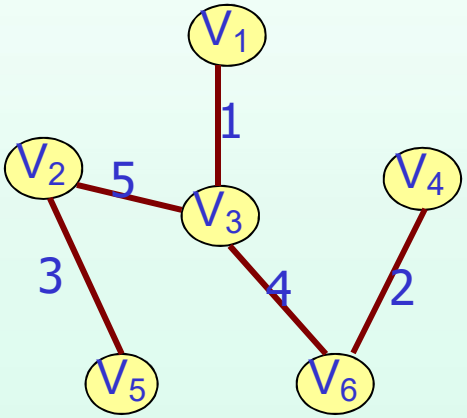


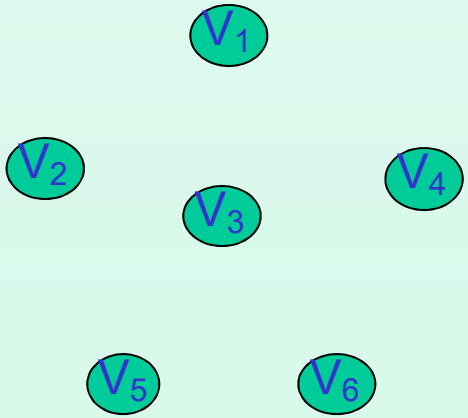
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$
(4)	$\{V_1, V_3, V_6, V_4, V_2\}$	$\{V_5\}$
(5)	$\{V_1, V_3, V_6, V_4, V_2, V_5\}$	$\{\}$



最小代价生成树

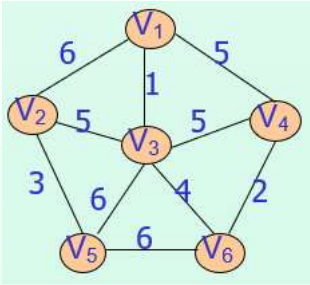
- 普里姆算法求最小生成树：从生成树中只有一个顶点开始，到顶点全部进入生成树为止

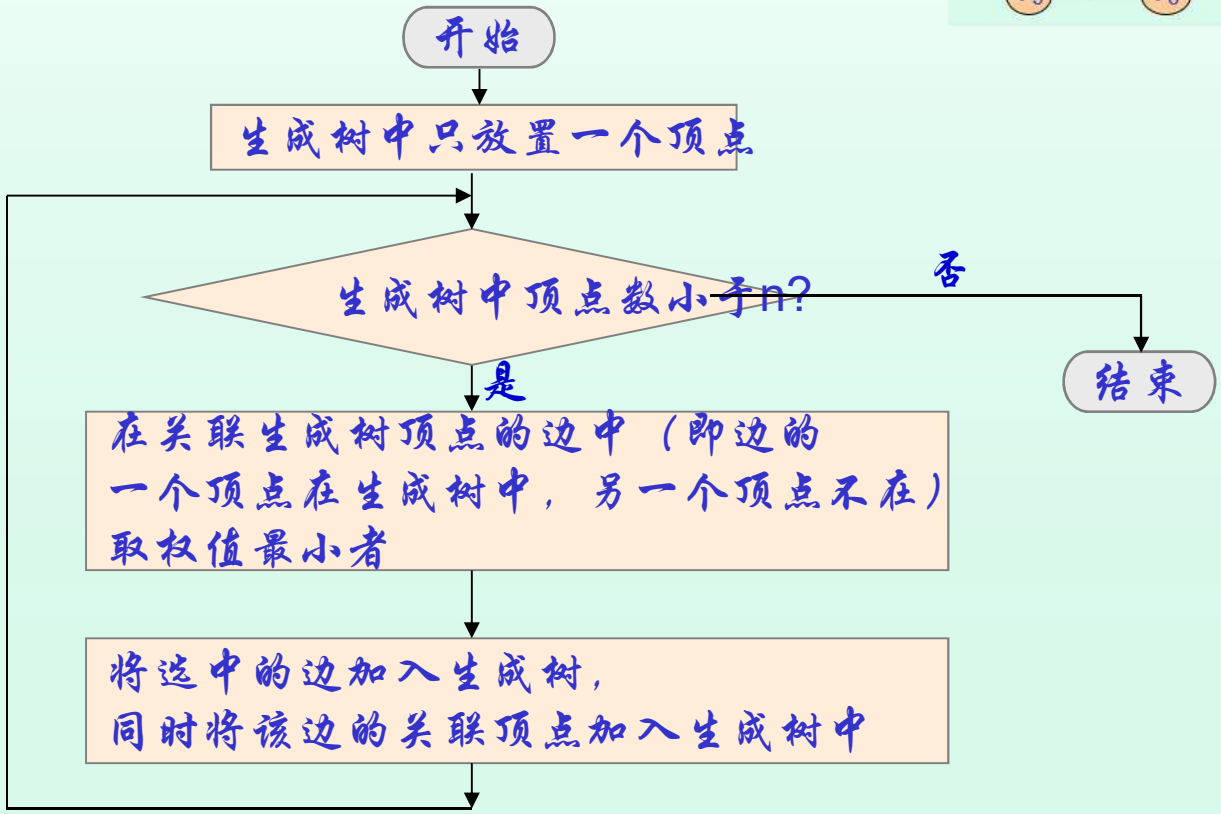




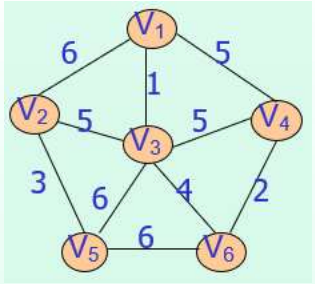
步骤	U	V-U
(0)	{V_1}	{V_2, V_3, V_4, V_5, V_6}
(1)	{V_1, V_3}	{V_2, V_4, V_5, V_6}
(2)	{V_1, V_3, V_6}	{V_2, V_4, V_5}
(3)	{V_1, V_3, V_6, V_4}	{V_2, V_5}
(4)	{V_1, V_3, V_6, V_4, V_2}	{V_5}
(5)	{V_1, V_3, V_6, V_4, V_2, V_5}	{ }

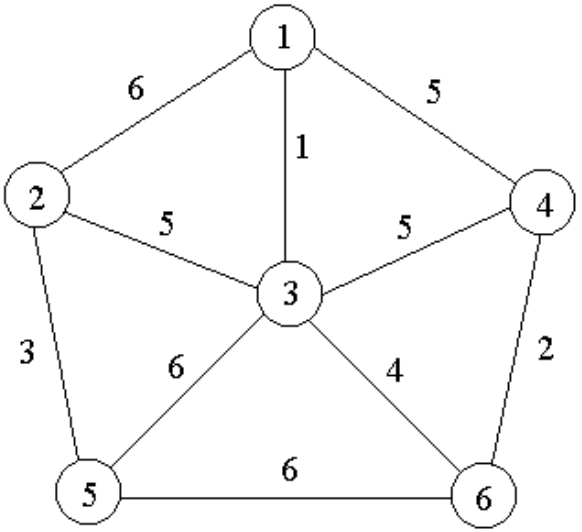
普里姆(Prim)算法

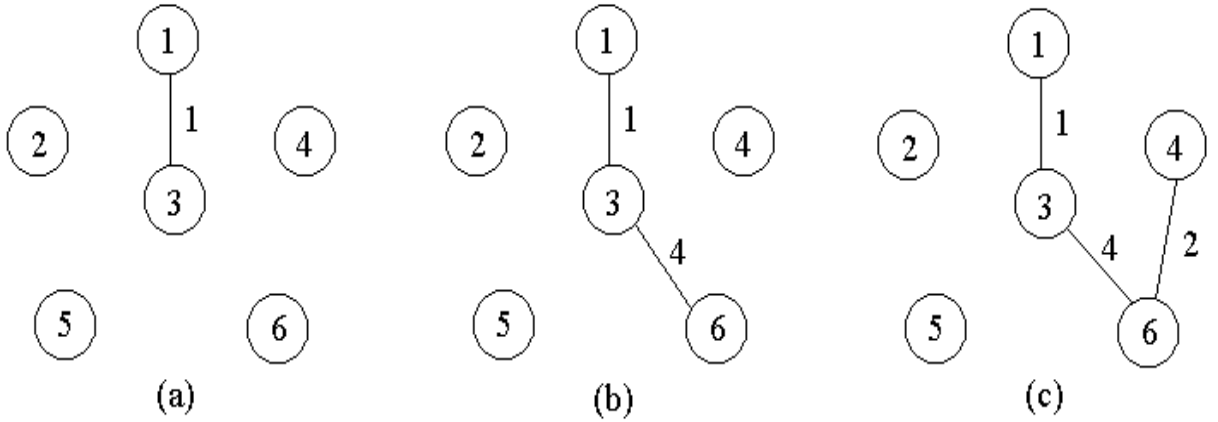

V1
V2
V3
V4
V5
V6
6
5
1
5
5
3
6
4
2
6

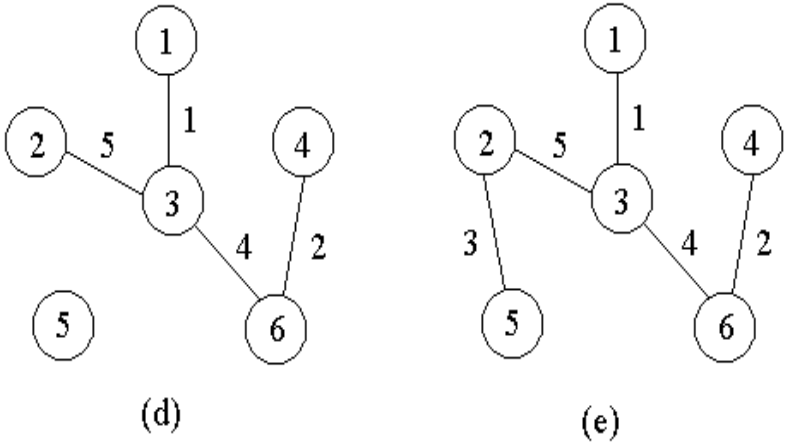

开始
生成树中只放置一个顶点
生成树中顶点数小于n?
否
是
结束
在关联生成树顶点的边中（即边的一个顶点在生成树中，另一个顶点不在）取权值最小者
将选中的边加入生成树，同时将该边的关联顶点加入生成树中

4.6 最小生成树


V1
V2
V3
V4
V5
V6
6
5
1
5
5
3
6
4
2
6


1
2
3
4
5
6
6
5
1
5
5
3
6
4
2
6


1
2
3
4
5
6
1
(a)
1
2
3
4
5
6
1
4
(b)
1
2
3
4
5
6
1
4
2
(c)


1
2
3
4
5
6
5
1
4
2
(d)
1
2
3
4
5
6
5
1
3
4
2
(e)

4.6 最小生成树

在上述Prim算法中，还应当考虑**如何有效地找出满足条件i∈S,j∈V-S，且权c[i][j]最小的边(i,j)**。实现这个目的的较简单的办法是设置2个数组closest和lowcost。

在Prim算法执行过程中，先找出V-S中使lowcost值最小的顶点j，然后根据数组closest选取边(j,closest[j])，最后将j添加到S中，并对closest和lowcost作必要的修改。

用这个办法实现的Prim算法所需的**计算时间**为 $O(n^2)$

4.6 最小生成树

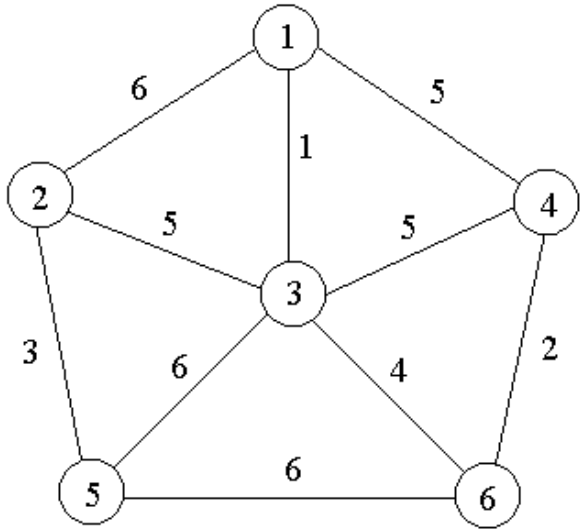
3. Kruskal算法

- Kruskal算法构造G的最小生成树的**基本思想**是，首先将G的n个顶点看成n个孤立的连通分支。将所有的边按权从小到大排序。

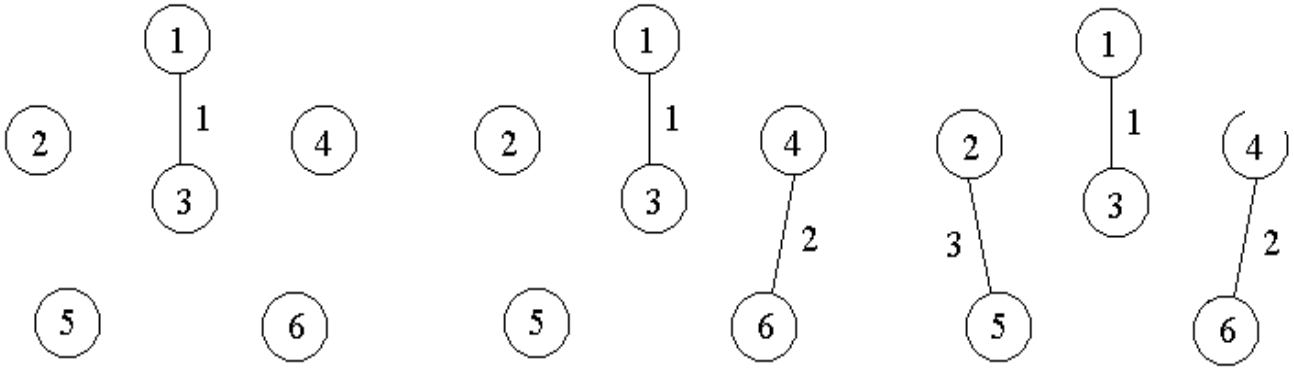
- 然后从第一条边开始，依边权递增的顺序查看每一条边，

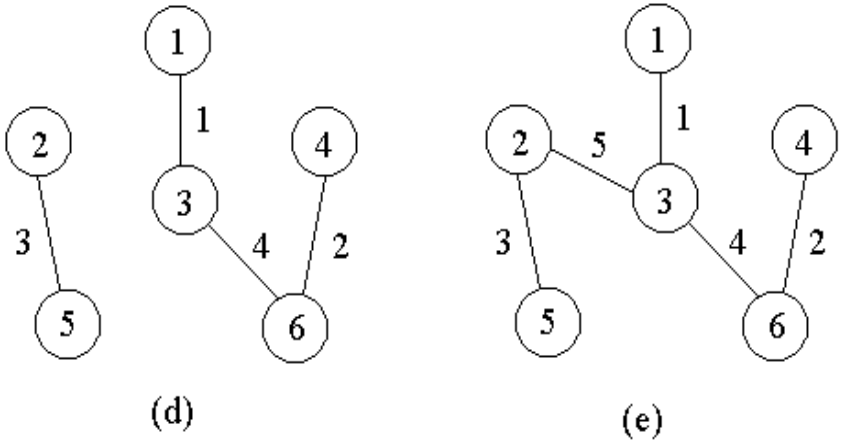
- 并按下述方法连接2个不同的连通分支：当查看到第k条边(v,w)时，如果端点v和w分别是当前2个不同的连通分支T1和T2中的顶点时，就用边(v,w)将T1和T2连接成一个连通分支，然后继续查看第k+1条边；如果端点v和w在当前的同一个连通分支中，就直接再查看第k+1条边。这个过程一直进行到只剩下一个连通分支时为止。

4. 6 最小生成树



例如，对前面的连通带权图，按Kruskal算法顺序得到的最小生成树上的边如下图所示。





4.6 最小生成树

关于**集合的一些基本运算**可用于实现Kruskal算法。

按权的递增顺序查看等价于对**优先队列**执行**removeMin**运算。可以用**堆**实现这个优先队列。

对一个由连通分支组成的集合不断进行修改，需要用到抽象数据类型**并查集UnionFind**所支持的基本运算。

当图的边数为e时，Kruskal算法所需的**计算时间**是$O(e\log e)$当 $e=\Omega(n^2)$时，Kruskal算法比Prim算法差，但当 $e=O(n)$时，Kruskal算法却比Prim算法好得多。

4.7 多机调度问题

多机调度问题要求给出一种作业调度方案，使所给的n个作业在尽可能短的时间内由m台机器加工处理完成。

约定，每个作业均可在任何一台机器上加工处理，但未完工前不允许中断处理。作业不能拆分成更小的子作业。

这个问题是**NP完全问题**，到目前为止还没有有效的解法。对于这一类问题,用**贪心选择策略**有时可以设计出较好的近似算法。

4.7 多机调度问题

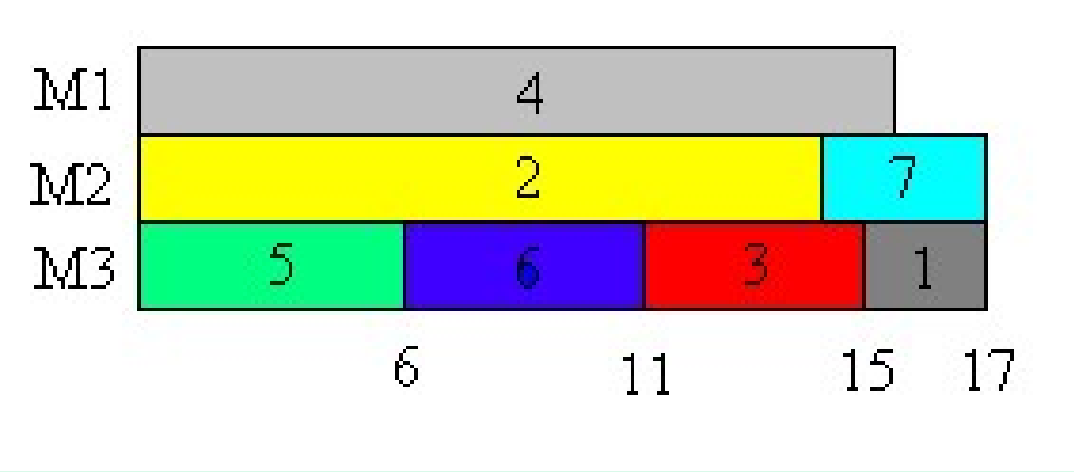
采用**最长处理时间作业优先**的贪心选择策略可以设计出解多机调度问题的较好的近似算法。

按此策略，当$n \le m$时，只要将机器i的[0, ti]时间区间分配给作业i即可，算法只需要**O(1)**时间。

当$n > m$时，首先将n个作业依其所需的处理时间从大到小排序。然后依此顺序将作业分配给空闲的处理机。算法所需的计算时间为**O(nlogn)**。

4.7 多机调度问题

例如，设7个独立作业{1,2,3,4,5,6,7}由3台机器M1，M2和M3加工处理。各作业所需的处理时间分别为{2,14,4,16,6,5,3}。按算法**greedy**产生的作业调度如下图所示，所需的加工时间为17。



第四章
授课内容-结束

4.8 贪心算法的理论基础（了解）

借助于**拟阵**工具，可建立关于贪心算法的较一般的理论。这个理论对**确定何时使用贪心算法**可以得到问题的整体最优解十分有用。

1. 拟阵

拟阵M定义为满足下面3个条件的有序对(S,I)：

(1)S是非空有限集。

(2)I是S的一类具有遗传性质的独立子集族，即若$B \in I$，则B是S的独立子集，且B的任意子集也都是S的独立子集。空集$\varnothing$必为I的成员。

(3)I满足交换性质，即若$A \in I, B \in I$且$|A|<|B|$，则存在某一元素$x \in B-A$，使得$A \cup \{x\} \in I$。

4.8 贪心算法的理论基础

例如，设S是一给定矩阵中行向量的集合，I是S的线性独立子集族，则由线性空间理论容易证明(S,I)是一拟阵。拟阵的另一个例子是无向图G=(V,E)的图拟阵 $M_G=(S_G,I_G)$

给定拟阵M=(S,I)，对于I中的独立子集A∈ I，若S有一元素x∉ A，使得将x加入A后仍保持独立性，即A∪{x} ∈ I,则称x为A的**可扩展元素**。

当拟阵M中的独立子集A没有可扩展元素时，称A为**极大独立子集**。

4.8 贪心算法的理论基础

下面的关于**极大独立子集**的性质是很有用的。

定理4.1：拟阵M中所有极大独立子集大小相同。

这个定理可以用反证法证明。

若对拟阵M=(S,I)中的S指定权函数W，使得对于任意x∈ S，有W(x)>0，则称拟阵M为**带权拟阵**。依此权函数，S的任一子集A的权定义为 $W(A)=\sum_{x\in A}W(x)$ 。

2. 关于带权拟阵的贪心算法

许多可以用贪心算法求解的问题可以表示为求带权拟阵的**最大权独立子集问题**。

4.8 贪心算法的理论基础

给定带权拟阵M=(S,I)，确定S的独立子集A∈I使得W(A)达到最大。这种使W(A)最大的独立子集A称为拟阵M的**最优子集**。由于S中任一元素x的权W(x)是正的，因此，**最优子集也一定是极大独立子集**。

例如，在最小生成树问题可以表示为确定带权拟阵M_G的最优子集问题。求带权拟阵的最优子集A的算法可用于解最小生成树问题。

下面给出求**带权拟阵最优子集**的贪心算法。该算法以具有正权函数W的带权拟阵M=(S,I)作为输入，经计算后输出M的最优子集A。

4.8 贪心算法的理论基础

- Set **greedy** (M,W)
- {A=∅;
- 将S中元素依权值W（大者优先）组成优先队列;
- while (S!=∅) {
- S.removeMax(x);
- if (A∪{x}∈I) A=A∪{x};
- }
- return A
- }

4.8 贪心算法的理论基础

算法**greedy**的计算时间复杂性为$O(n\log n + nf(n))$

引理4.2(拟阵的贪心选择性质)

设M=(S,I)是具有权函数W的带权拟阵，且S中元素依权值从大到小排列。又设x∈ S是S中第一个使得{x}是独立子集的元素，则存在S的最优子集A使得x∈ A。

算法**greedy**在以贪心选择构造最优子集A时，首次选入集合A中的元素x是单元素独立集中具有最大权的元素。此时可能已经舍弃了S中部分元素。可以证明这些被舍弃的元素不可能用于构造最优子集。

4.8 贪心算法的理论基础

引理4.3： 设M=(S,I)是拟阵。若S中元素x不是空集的可扩展元素，则x也不可能是S中任一独立子集A的可扩展元素。

引理4.4(拟阵的最优子结构性质)

设x是求带权拟阵M = (S，I)的最优子集的贪心算法**greedy**所选择的S中的第一个元素。那么，原问题可简化为求带权拟阵M'=(S',I')的**最优子集**问题，其中：

$$S'=\{y \mid y \in S \text{且} \{x,y\} \in I\}$$

$$I'=\{B \mid B \subseteq S-\{x\} \text{且} B \cup \{x\} \in I\}$$

M'的权函数是M的权函数在S'上的限制(称M'为M关于元素x的**收缩**)。

4.8 贪心算法的理论基础

定理4.5(带权拟阵贪心算法的正确性)

设M = (S,I)是具有权函数W的带权拟阵，算法greedy返回M的最优子集。

3. 任务时间表问题

给定一个**单位时间任务**的有限集S。关于S的一个**时间表**用于描述S中单位时间任务的执行次序。时间表中第1个任务从时间0开始执行直至时间1结束，第2个任务从时间1开始执行至时间2结束，…，第n个任务从时间n-1开始执行直至时间n结束。

4.8 贪心算法的理论基础

具有**截止时间**和**误时惩罚**的单位时间任务时间表问题可描述如下。

(1) n个单位时间任务的集合S={1,2,…,n};

(2) 任务i的截止时间 ,d_i≤i≤n,1≤ ≤n, d_i即要求任务i在时间 之前结束;d_i

(3) 任务i的误时惩罚 w_i≤i≤n,即任务i未在时间 之前结束将招致的 惩罚;w_i若按时完成则无惩罚。

任务时间表问题要求确定S的一个时间表（最优时间表）使得总误时惩罚达到最小。

4.8 贪心算法的理论基础

这个问题看上去很复杂，然而借助于**拟阵**，可以用**带权拟阵的贪心算法**有效求解。

对于一个给定的S的时间表，在截止时间之前完成的任务称为**及时任务**，在截止时间之后完成的任务称为**误时任务**。

S的任一时间表可以调整成**及时优先形式**，即其中所有及时任务先于误时任务，而不影响原时间表中各任务的及时或误时性质。

类似地，还可将S的任一时间表调整成为**规范形式**，其中及时任务先于误时任务，且及时任务依其截止时间的非减序排列。

4.8 贪心算法的理论基础

首先可将时间表调整为及时优先形式，然后再进一步调整及时任务的次序。

任务时间表问题**等价于**确定最优时间表中**及时任务子集A**的问题。一旦确定了及时任务子集A，将A中各任务依其截止时间的非减序列出，然后再以任意次序列出误时任务，即S-A中各任务，由此产生S的一个规范的最优时间表。

对时间t=1,2,…,n，**设** $N_t(A)$是任务子集A中所有截止时间是t或更早的任务数。考察任务子集A的独立性。

4.8 贪心算法的理论基础

引理4.6： 对于S的任一任务子集A，下面的各命题是等价的。

(1) 任务子集A是独立子集。

(2) 对于t=1,2,…,n， $N_t(A) \le t$。

(3) 若A中任务依其截止时间非减序排列，则A中所有任务都是及时的。

任务时间表问题要求使总误时惩罚达到最小，这等价于使任务时间表中的及时任务的惩罚值之和达到最大。下面的**定理**表明可用带权拟阵的贪心算法解任务时间表问题。

4.8 贪心算法的理论基础

定理4.7：设S是带有截止时间的单位时间任务集，I是S的所有独立任务子集构成的集合。则有序对(S,I)是拟阵。

由**定理4.5**可知，用带权拟阵的贪心算法可以求得最大权(惩罚)独立任务子集A，以A作为最优时间表中的及时任务子集，容易构造最优时间表。

任务时间表问题的贪心算法的**计算时间复杂性**是$O(n\log n+nf(n))$其中f(n)是用于检测任务子集A的独立性所需的时间。用引理4.6中性质(2)容易设计一个$O(n)$时间算法来检测任务子集的独立性。因此，整个算法的**计算时间**为$O(n^2)$具体算法**greedyJob**可描述如P130。

4.8 贪心算法的理论基础

用抽象数据类型并查集**UnionFind**可对上述算法作进一步改进。如果不计预处理的时间，改进后的算法**fasterJob**所需的**计算时间**为$O(n\log^* n)$。