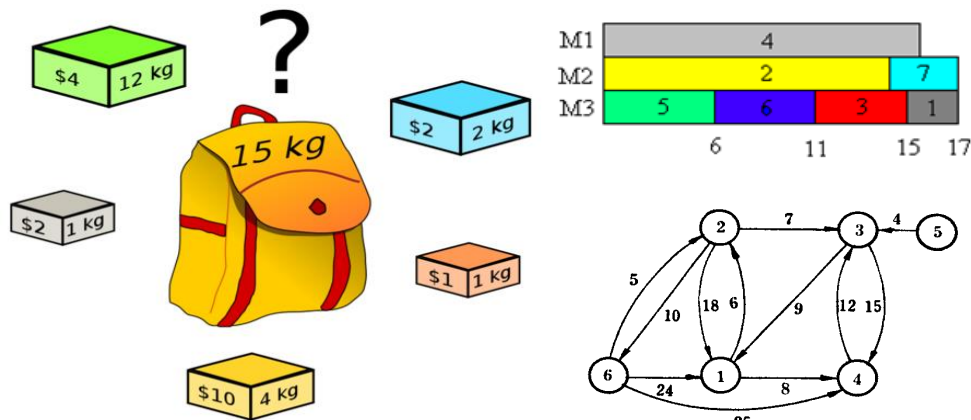


DESIGN AND ANALYSIS OF ALGORITHMS

Greedy Algorithm 贪心（贪婪）算法



盛浩

shenghao@buaa.edu.cn



问题引入

- **找零钱：** 假如售货员需要找给67美分的零钱。现在，售货员手中只有25美分、10美分、5美分和1美分的硬币。客户要求售货员尽可能的少给零钱。
- **售货员的做法：** 先找不大于67美分的最大硬币25美分硬币，再找不大于 $67 - 25 = 42$ 美分的最大硬币25美分硬币，再找不大于 $42 - 25 = 17$ 美分的最大硬币10美分硬币，再找不大于 $17 - 10 = 7$ 美分的最大硬币5美分硬币，最后售货员再找出两个1美分的硬币。



本节提纲

- 5.1 贪心算法基本思想
- 5.2 最优装载问题
- 5.3 背包问题
- 5.4 多机调度问题
- 5.5 单源点最短路径
- 5.6 最小生成树
- 5.7 哈夫曼(Huffman)编码 (略)

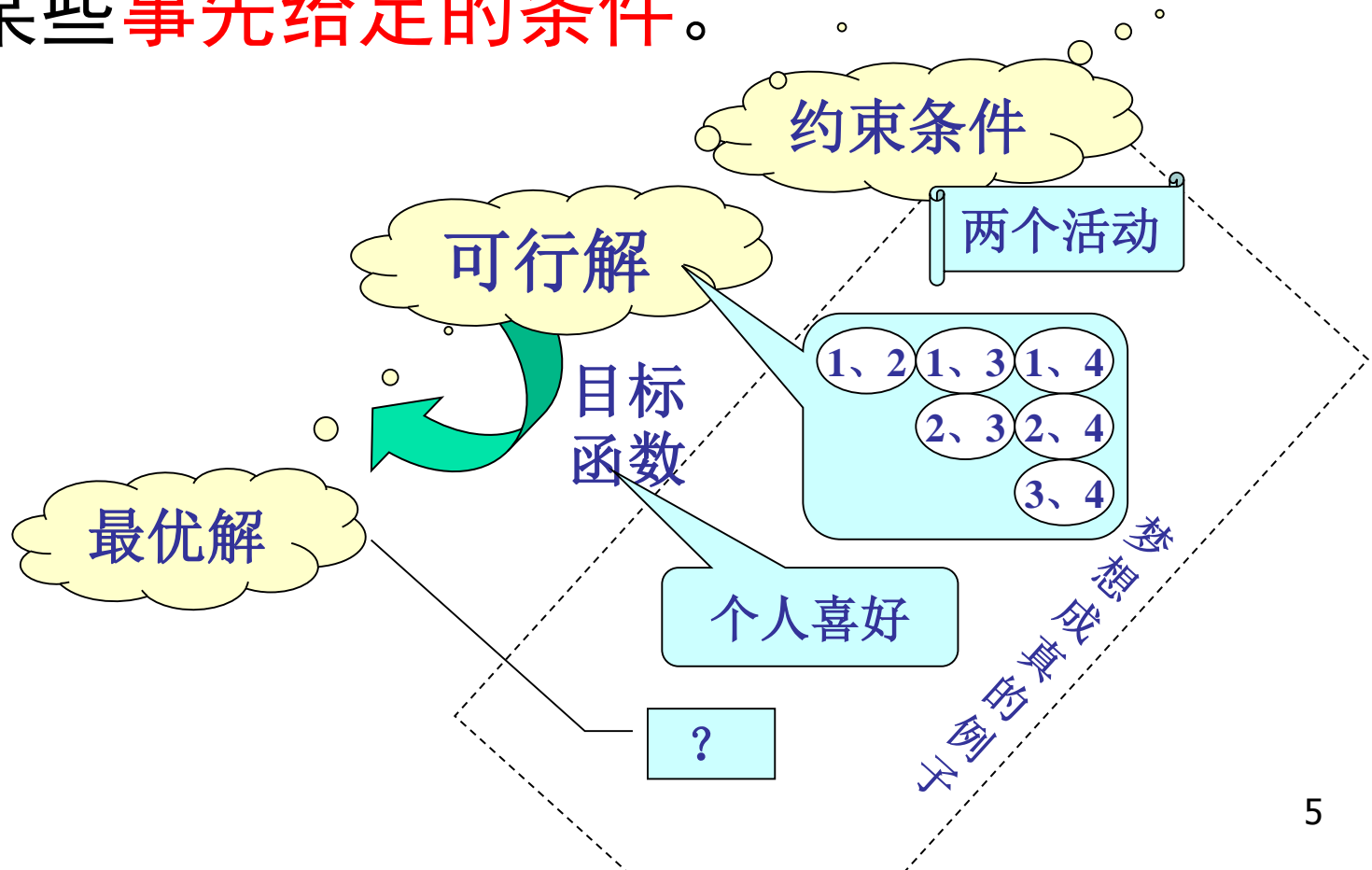


5.1 贪心算法基本思想

- 贪心算法的适用范围
- 贪心算法的基本思想
- 核心问题及求解步骤
- 贪心算法的优缺点

贪心算法的适用范围

- 有这样一类问题：它有 n 个输入，而它的解就是这 n 个输入的某个子集，这些子集**必须满足某些事先给定的条件**。





贪心算法的适用范围

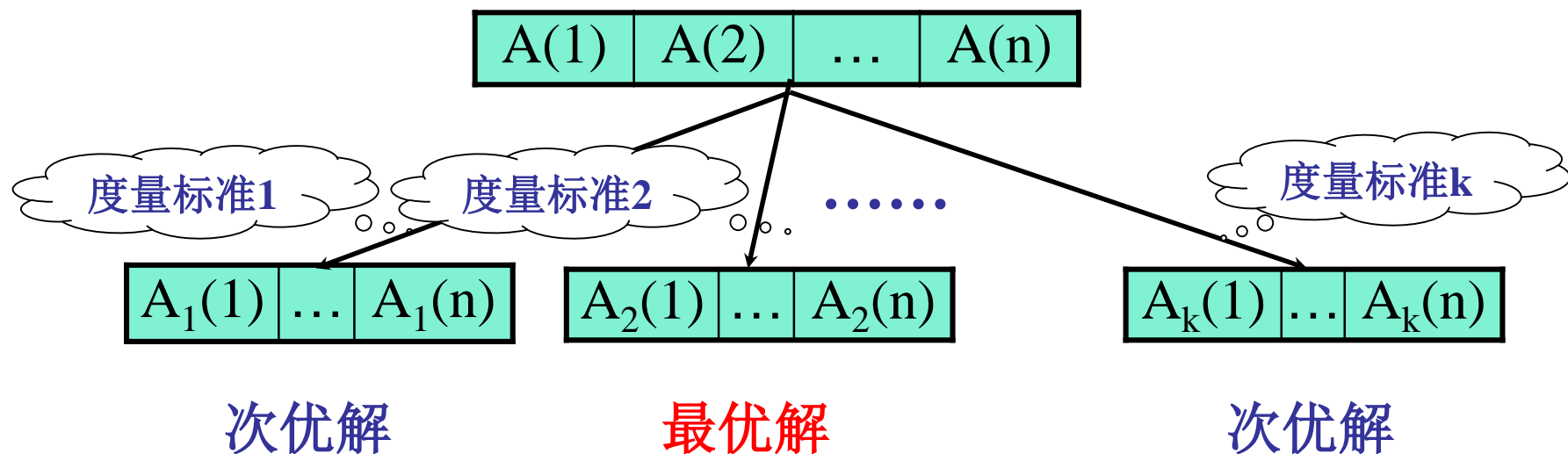
- 有这样一类问题：它有 n 个输入，而它的解就是这 n 个输入的某个子集，这些子集**必须满足某些事先给定的条件**。
- 贪心算法又叫优先策略，是一种非常简
单好用的求解最优化问题的方法，在许多方面的应用非常成功。
- **贪心算法基本要素：**
 - **贪心准则：**指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心准则来达到。
 - **最优子结构：**一个问题的最优解包含其子问题的最优解



贪心算法的基本思想

- 贪心算法是指在对问题求解时，总是做出在当前看来是最好的选择。
- 贪心算法是根据一种贪心准则 (greedy criterion) 来逐步构造问题的解的方法。在每一个阶段，都作出了相对该准则最优的决策。决策一旦作出，就不可更改。
- 由贪心算法得到的问题的解可能是最优解，也可能只是近似解。

贪心算法设计求解的核心问题



- 贪心算法设计求解的核心问题是选择能产生问题最优解的最优量度标准（**贪心准则**）。



贪心算法的求解步骤

- 贪心算法是根据具体的问题：
 - 选取一种度量标准；
 - 按此标准对 n 个输入进行排序；
 - 按该顺序一次输入一个量；
 - 如果这个输入量和当前该度量意义下的部分最优解加在一起产生一个可行解。
- 这种能够得到某种量度意义下的最优解的分级处理方法就是贪心算法。



贪心算法的缺点和优点

■ 缺点：

- 不是对所有问题都能得到整体最优解。
- 需要证明后才能真正运用到算法中。

■ 优点：

- 一旦经过证明成立后，它就是一种高效的算法。
- 策略的构造简单易行。
- 对范围相当广泛的许多问题他能产生整体最优解或者是整体最优解的近似解。



5.2 最优装载问题

■ 问题原型

假设有 n 个集装箱，其中集装箱 i 的重量为 w_i ，最优装载问题要求在装载体积不受限制的情况下，将尽可能多的集装箱装上载重量为 c 的轮船。



最优装载问题数学描述

最优装载问题可描述为：

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

$$\sum_{i=1}^n w_i x_i \leq c \quad x_i \in \{0,1\}, 1 \leq i \leq n$$

其中 $x_i = 1$ 表示装入集装箱 i , $x_i = 0$ 表示不装入集装箱 i 。

贪心算法解最优装载问题

目标是装入集装箱个数最多—贪心选择重量轻者优先

例： $c=100$ 吨

$w_1=10$ 吨 $w_2=30$ 吨 $w_3=20$ 吨 $w_4=40$ 吨

① 初始剩余载重 $g=100$ 装入个数 $k=0$

② 贪心选择 w_1

$w_1 < g$ 故 $x_1=1$ $g=g-w_1=90$ $k=1$

③ 贪心选择 w_2

$w_2 < g$ 故 $x_2=1$ $g=g-w_2=60$ $k=2$

同理，继续。得 $X=\{1, 1, 1, 1\}$

贪心算法解最优装载问题

```
template<class Type>
void Loading(int x[], Type w[], Type c, int n){
    int t[] = new int [n+1];
    Sort(w, t, n);    //按照箱子重量排序
    for (int i = 1; i <= n; i++) x[i] = 0;
    for (int i = 1; i <= n && w[t[i]] <= c; i++) {
        x[t[i]] = 1;
        c -= w[t[i]];
    }
}
```

箱子重量不能超过
剩余的可装重量



贪心算法解最优装载问题算法分析

时间复杂性分析：排序 $O(n \log n)$ ，贪心选择 $O(n)$ ，

故时间复杂性为 $O(n \log n)$

最优解分析：

具有贪心选择性质和最优子结构性质，能得到问题的最优解。



5.3 背包问题

- 问题描述
- 背包问题实例
- 背包问题的贪心算法
- 定理1

问题描述

- **背包问题** 已知容量为M的背包和n件物品。第i件物品的重量为 w_i ，价值是 p_i 。因而将物品i的一部分 x_i 放进背包即获得 $p_i x_i$ 的价值($0 \leq x_i \leq 1, p_i > 0$)。问题是：怎样装包使所获得的价值最大。
- 即是如下的优化问题：

目标函数	$\sum_{1 \leq i \leq n} p_i x_i$	$0 \leq x_i \leq 1, p_i > 0$
约束条件	$\sum_{1 \leq i \leq n} w_i x_i \leq M$	$w_i > 0, 1 \leq i \leq n$

背包问题实例

考虑下列情况下的背包问题：

$n=3$, $M=20$, $(p_1, p_2, p_3)=(25, 24, 15)$, $(w_1, w_2, w_3)=(18, 15, 10)$

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
(1)按价值的非增次序把物品放到包里。	$(1, 2/15, 0)$	20	28.2
(2)按物品重量的非降次序把物品放到包里。	$(0, 2/3, 1)$	20	31
(3)按 p_i/w_i 比值的非增次序把物品放到包里。	$(0, 1, 1/2)$	20	31.5

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
$(1, 2/15, 0)$	20	28.2
$(0, 2/3, 1)$	20	31
$(0, 1, 1/2)$	20	31.5

$(p_1, p_2, p_3) = (25, 24, 15);$
 $(w_1, w_2, w_3) = (18, 15, 10)。$

(1) 按价值的非增次序把物品放到包里。

■ 即以目标函数为量度标准

- 该标准使得背包每装入一件物品就获得最大可能的效益值增量。
- 结果是一个次优解，原因是背包容量消耗过快。

(2) 按物品重量的非降次序把物品放到包里。

■ 以容量为量度标准

- 该标准使得背包每装入一件物品就获得最小可能的容量增量。
- 结果仍是一个次优解，原因是容量在慢慢消耗的过程中，效益值却没有迅速的增加。

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
$(1, 2/15, 0)$	20	28.2
$(0, 2/3, 1)$	20	31
$(0, 1, 1/2)$	20	31.5

$(p_1, p_2, p_3) = (25, 24, 15);$
 $(w_1, w_2, w_3) = (18, 15, 10)。$

(3) 按 p_i/w_i 比值的非增
 次序把物品放到包里。

- 选效益值和容量之比为量度标准
 - 每一次装入的物品使它占用的每一单位容量获得当前最大的单位效益
 - 结果是一个最优解，因为每一单位容量的增加将获得最大的单位效益值。

$(p_2/w_2, p_3/w_3, p_1/w_1) = (24/15, 15/10, 25/18)$

背包问题的贪心算法

先将物品按 p_i/w_i 比值的非增次序排序(降序)

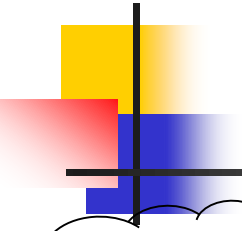
```
procedure GREEDY-KNAPSACK(P,W,M,X,n)
real P(1:n), W(1:n), X(1:n), M, cu;
integer i, n;
X ← 0
cu ← M
for i ← 1 to n do
    if (W(i) > cu) then exit endif
    X(i) ← 1
    cu ← cu - W(i)
endfor
if (i ≤ n) then X(i) ← cu/W(i)
end GREEDY-KNAPSACK
```

将解向量X初始化为0
cu为背包的剩余容量

若物品i的重量大于背包的剩余容量,则退出循环

若物品i的重量小于等于背包的剩余容量,则物品i可放入背包内

放入物品i的一部分



$(p1, p2, p3) = (25, 24, 15), (w1, w2, w3) = (18, 15, 10)。$

运行过程如下

$(p2/w2, p3/w3, p1/w1) = (24/15, 15/10, 25/18)$

输入参数 $P=(24, 15, 25); W=(15, 10, 18); M=20; n=3。$

初始: $X=(0, 0, 0); cu=20。$

for 循环部分: (1) $15 < cu$, then $x(1)=1, cu=cu-w(1)=5;$
(2) $10 > cu$, then 跳出循环;

结尾: $x(2)= cu/w(2) = 0.5。$

输入参数 $X=(1, 0.5, 0)。$

定理1

- **定理1** 如果 $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$, 则算法 **GREEDY-KNAPSACK** 对于给定的背包问题实例生成一个最优解。

证明: 设 $X=(x_1, \dots, x_n)$ 是算法所生成的解。

1. 如果所有的 x_i 等于1, 显然这个解就是最优解。
2. 否则, 设 j 是使 $x_j \neq 1$ 的最小下标, 由算法可知:
 - 对于 $1 \leq i < j$, $x_i = 1$;
 - 对于 $j < i \leq n$, $x_i = 0$;
 - 对于 $i = j$, $0 \leq x_j < 1$ 。



3. 若 X 不是最优解, 则必存在一个最优解

$Y = (y_1, \dots, y_n)$, 使得 $\sum p_i y_i > \sum p_i x_i$ 。

假定 $\sum w_i y_i = M$, 设 k 是使得 $y_k \neq x_k$ 的最小下标,
则可以推出 $y_k < x_k$ 成立。

三种可能发生的情况:
(1). $k < j$; (2). $k = j$; (3). $k > j$

4. 现在, 假定把 y_k 增加到 x_k , 那么必须从 (y_{k+1}, \dots, y_n) 中减去同样多的量, 使得所用的总容量仍为 M 。这导致一个新的解 $Z = (z_1, \dots, z_n)$, 其中 $z_i = x_i, 1 \leq i \leq k$ 。
并且 $\sum_{k < i \leq n} w_i (y_i - z_i) = w_k (z_k - y_k)$

5. 因此, 对于Z有

$$\begin{aligned}\sum_{1 \leq i \leq n} p_i z_i &= \sum_{1 \leq i \leq n} p_i y_i + (z_k - y_k) p_k - \sum_{k < i \leq n} (y_i - z_i) p_i \\&= \sum_{1 \leq i \leq n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k < i \leq n} (y_i - z_i) w_i p_i / w_i \\&\geq \sum_{1 \leq i \leq n} p_i y_i + [(z_k - y_k) w_k - \sum_{k < i \leq n} (y_i - z_i) w_i] p_k / w_k \\&= \sum_{1 \leq i \leq n} p_i y_i\end{aligned}$$

所以 $\sum p_i z_i \geq \sum p_i y_i$ 成立

元素按 p_i/w_i 非增次序排列。

6. 若 $\sum p_i z_i > \sum p_i y_i$, 则Y不可能是最优解。

所以 $\sum p_i z_i = \sum p_i y_i$,

如果Z=X, 则X就是最优解;

否则Z≠X, 则重复上面的讨论, 每次Y中少一个和X不同的值, 最终把Y转换成X, 从而证明了X也是最优解, 证毕。



0-1背包问题

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。



5.4 多机调度问题

■ 问题原型

设有 n 个独立的不可拆分作业 $\{1, 2, \dots, n\}$ ，由 m 台相同的机器进行加工处理。作业 i 所需的处理时间为 t_i 。约定，每个作业均可在任何一台机器上加工处理，但在完工前不可中断处理。

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。



贪心算法解多机调度问题

调度时间最短—贪心选择作业执行时间大者，
将其调度到可最早开始加工它的机器上。



贪心算法解多机调度问题

例： 设7个作业的执行时间如下表，将在3台机器上加工处理。请给出它们的调度安排。

i	1	2	3	4	5	6	7
t_i	7	3	4	5	3	8	3



贪心算法解多机调度问题

```
float loading(float c, float [] w, int [] x)
{
    int n=w.length;
    for (int i = 0; i < n; i++)
        d[i] = (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;
}
```



解： 设分别用 $T[3]$ 表示3台机器的最早开始时间。

初始： $T[0]=T[1]=T[2]=0$

作业排序得： 6 1 4 3 2 5 7

1. 选作业6，调度至机器0， $T[0]=8$
2. 选作业1，调度至机器1， $T[1]=7$
3. 选作业4，调度至机器2， $T[2]=5$
4. 选作业3，调度至机器2， $T[2]=9$
5. 选作业2，调度至机器1， $T[1]=10$
6. 选作业5，调度至机器0， $T[0]=11$
7. 选作业7，调度至机器2， $T[2]=12$



调度示意图如下所示：

P_0	6 (8)		5 (3)		
P_1	1 (7)		2 (3)		
P_2	4 (5)		3 (4)		7 (3)

贪心算法解多机调度问题算法分析

时间复杂性分析： $O(n \log n + n \log m)$

多机问题不一定得到最优解：不具有贪心选择性质和最优子结构。

多机调度是NP完全问题，贪心算法得到其近似解。

上例的一种最优调度示意图如下所示：

P_0	6(8)		5(3)
P_1	1(7)		3(4)
P_2	4(5)	2(3)	7(3)



5.5 单源最短路问题

一、问题

设给定带权有向图 $G=(V,E)$ ，是一个有向图，图中每一条边都有一个非负长度。单源最短路径问题就是要求出从图中一个指定的点 s （称为源点）到其它每个点的长度最短的路径。这里路的长度是指路上各边**权之和**。



5.5 单源最短路问题

二、算法基本思想

Dijkstra算法是解单源最短路径问题的贪心算法。

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。



5.5 单源最短路问题

二、算法基本思想

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 $\lambda[]$ 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 $\lambda[]$ 作必要的修改。一旦 S 包含了所有 V 中顶点， $\lambda[]$ 就记录了从源到所有其它顶点之间的最短路径长度。



5.5 单源最短路问题

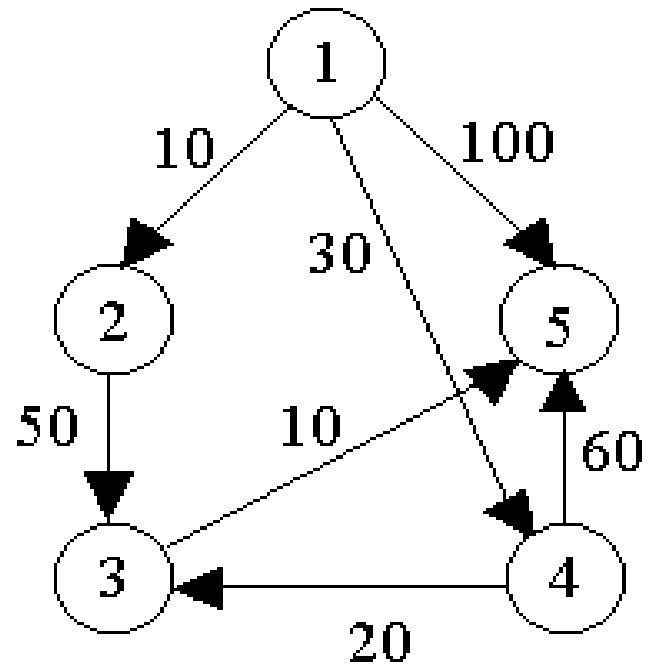
二、算法基本思想

- (1) $X \leftarrow \{1\}; Y \leftarrow V - \{1\}$
- (2) 对于每个点 $v \in Y$ ，若有一条从1到 v 的边，则令 $\lambda[v]$ = 该边的边长，否则令 $\lambda[v] = \infty$ 。令 $\lambda[1] = 0$
- (3) while $Y \neq \{\}$
- (4) 令 $y \in Y$ 为 $\lambda[\]$ 中最小的 y
- (5) 将 y 从 Y 移到 X 中
- (6) 更新 Y 中与 y 相连的点 $\lambda[\]$ 值
- (7) end while.

5.5 单源最短路问题

三、实例

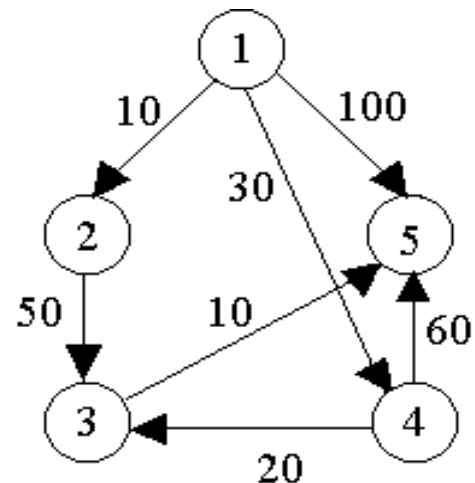
例如，对右图中的有向图，应用Dijkstra算法计算从源顶点1到其它顶点间最短路径的过程列在下页的表中。



5.5 单源最短路径问题

三、实例

Dijkstra算法的迭代过程：



迭代	S	u	$\lambda[2]$	$\lambda[3]$	$\lambda[4]$	$\lambda[5]$
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

DijkstraPaths(v, COST, Dist, n) //G 是具有 n 个顶点 {1, 2, ..., n}

//的有向图, v 是 G 中取定的顶点, COST 是 G 的邻接矩阵

boolean S[1:n]; **real** COST[1:n, 1:n], Dist[1:n];

integer u, v, n, num, i, w;

1 **for** i **from** 1 **to** n **do** //将集合 S 初始化为空

2 S(i)=0; Dist(i)=COST(v, i);

3 **endfor**

4 S(v)=1; Dist(v)=0; //首先将节点 v 记入 S

5 **for** num **from** 2 **to** n **do** //确定由节点 v 出发的 n-1 条路

6 选取顶点 w 使得 $Dist(w) = \min_{S(u)=0} \{Dist(u)\}$;

7 S(w)=1;

8 **while** S(u)=0 **do** //修改 S 外节点通过 S 到达 v 的最短距离

9 Dist(u)=min{Dist(u), Dist(w)+COST(w, u)};

10 **endwhile**

11 **endfor**

12 **end** DijkstraPath



5.6 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**。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。



5.6 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 $(u,v) \in E$ ，且 $u \in U$ ， $v \in V-U$ ，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**。



5.6 最小生成树

2. Prim算法

设 $G=(V,E)$ 是连通带权图， $V=\{1,2,\dots,n\}$ 。

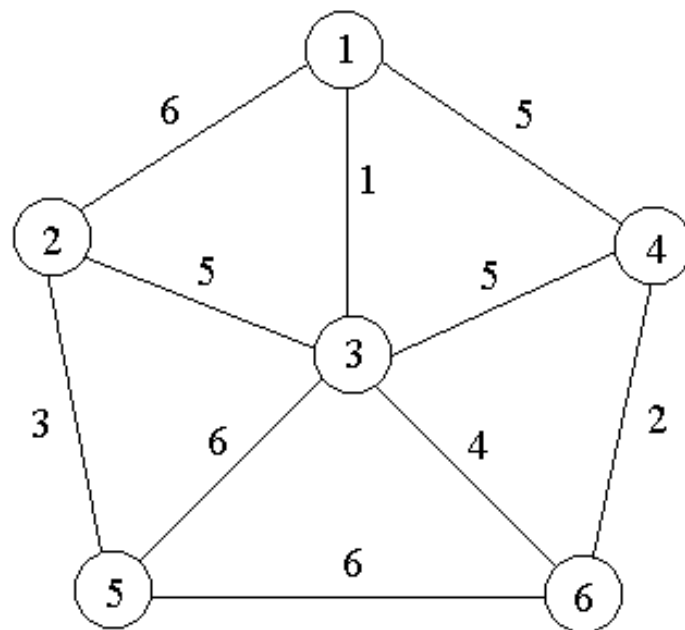
构造 G 的最小生成树的Prim算法的基本思想是：首先置 $S=\{1\}$ ，然后，只要 S 是 V 的真子集，就作如下的贪心选择：选取满足条件 $i \in S$ ， $j \in V-S$ ，且 $c[i][j]$ 最小的边，将顶点 j 添加到 S 中。这个过程一直进行到 $S=V$ 时为止。

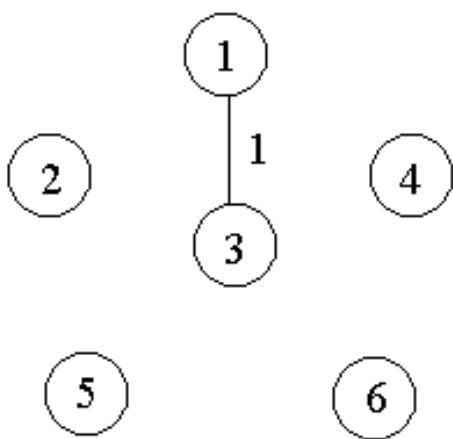
在这个过程中选取到的所有边恰好构成 G 的一棵最小生成树。

5.6 最小生成树

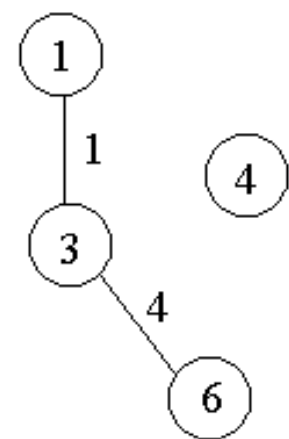
利用最小生成树性质和数学归纳法容易证明，上述算法中的**边集合T始终包含G的某棵最小生成树中的边**。因此，在算法结束时，T中的所有边构成G的一棵最小生成树。

例如，对于右图中的带权图，按**Prim算法**选取边的过程如下页图所示。

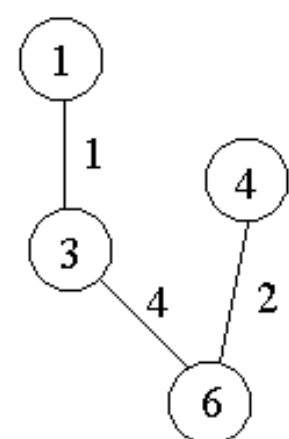




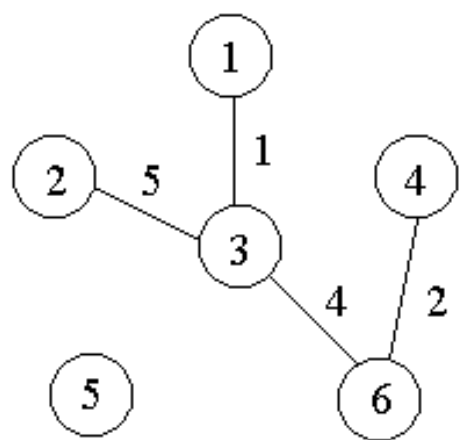
(a)



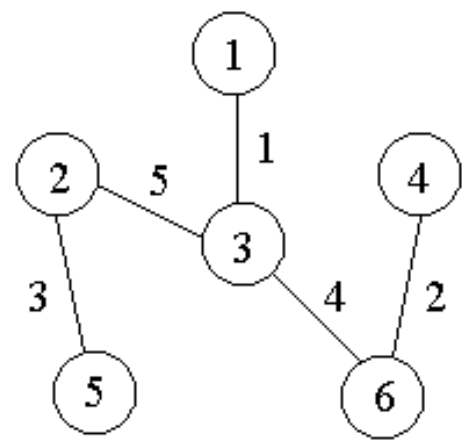
(b)



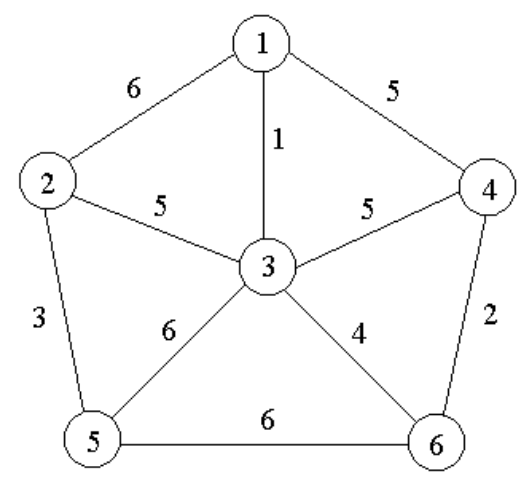
(c)



(d)



(e)





5.6 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 $i \in S, j \in V-S$ ，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 $(j, \text{closest}[j])$ ，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 $O(n^2)$



Prim算法的基本思想

- (1) $T \leftarrow \{\}; X \leftarrow \{1\}; Y \leftarrow V - \{1\}$
- (2) while $Y \neq \{\}$
- (3) 设 (x,y) 是一条满足 $x \in X$ 且 $y \in Y$ 的权值最小的边
- (4) $X \leftarrow X \cup \{y\}$
- (5) $Y \leftarrow Y - \{y\}$
- (6) $T \leftarrow T \cup \{(x,y)\}$
- (7) end while 。



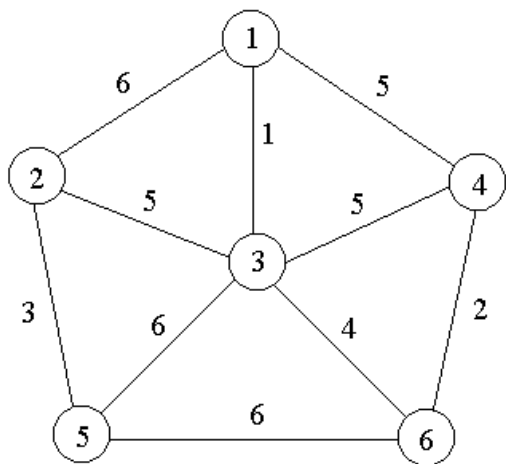
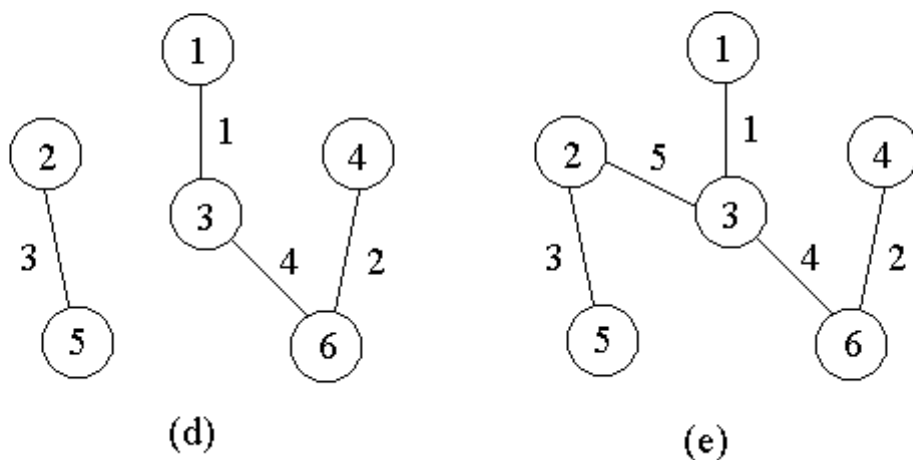
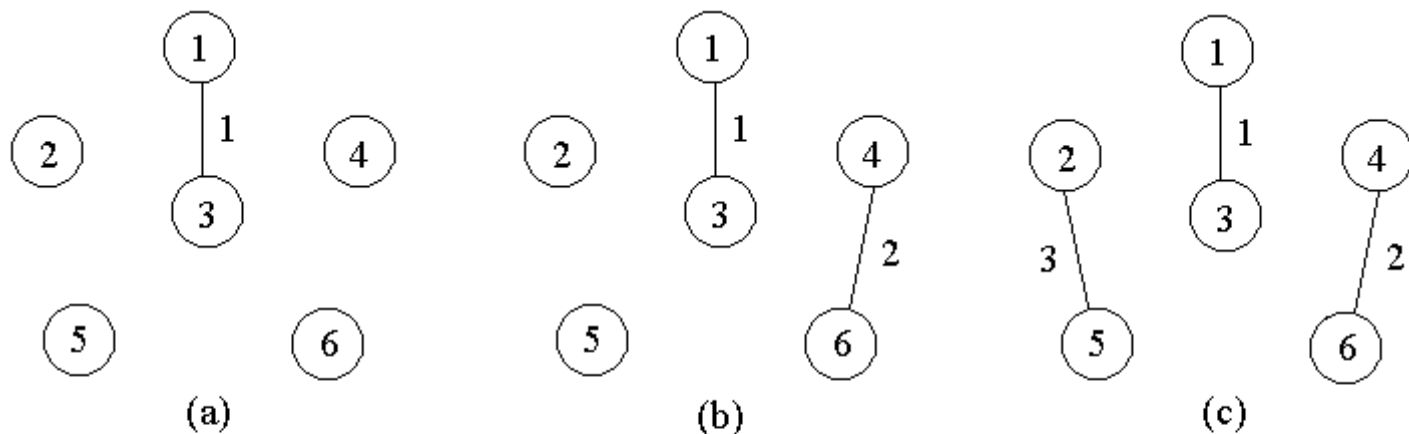
5.6 最小生成树

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。

5.6 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。





5.7 哈夫曼(Huffman)编码

- 哈夫曼编码是用于数据文件压缩的一个十分有效的编码方法，其压缩率通常在20%~90%之间。哈夫曼编码算法使用一个字符在文件中出现的频率表来建立一个0,1串表示各个字符的最优表示方式。



5.7 哈夫曼(Huffman)编码

- 具有100,000个字符文件中出现的6个不同的字符的出现频率统计。

不同的字符	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100



5.7 哈夫曼(Huffman)编码

- 如果采用定长码，则每个字符至少需用三位，该文件总共需要300,000位；如果采用上面所列变长码，则该文件需用 $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224,000$ 位，总码长减少了25%。

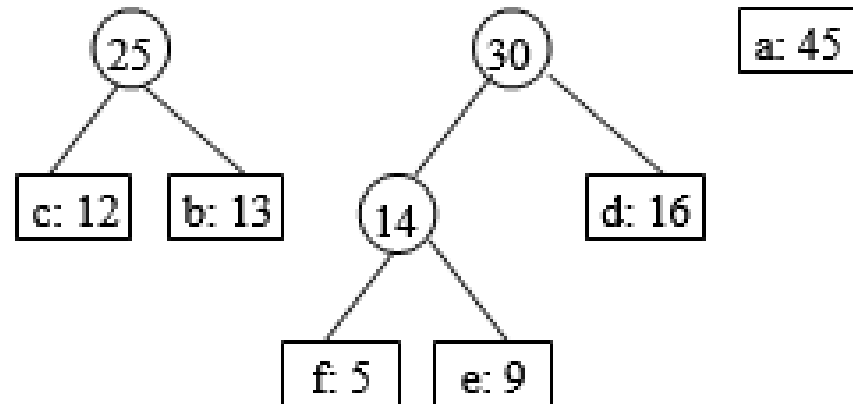
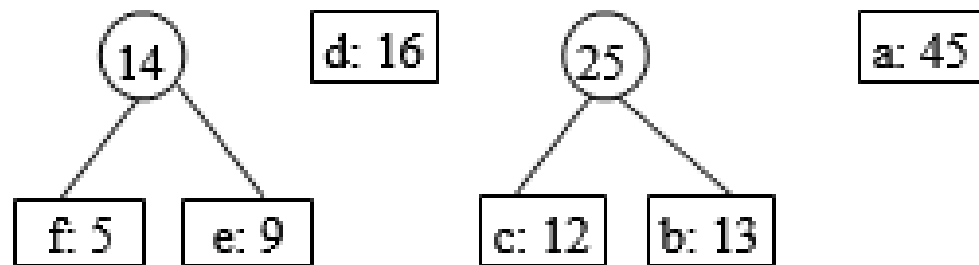
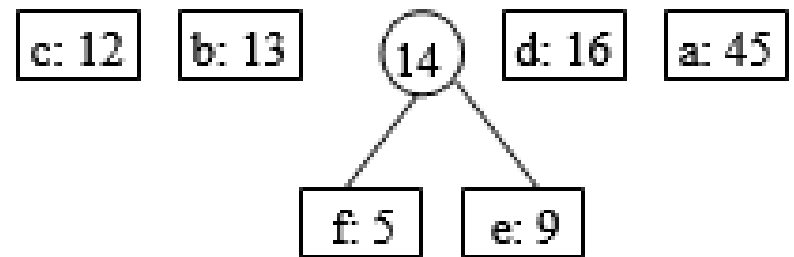


5.7 哈夫曼(Huffman)编码

- 哈夫曼提出了一种构造最优前缀编码的贪心算法，称为哈夫曼算法。该算法以自底向上的方式构造表示最优前缀编码的二叉树T。使平均码长达到最小的前缀编码方案。

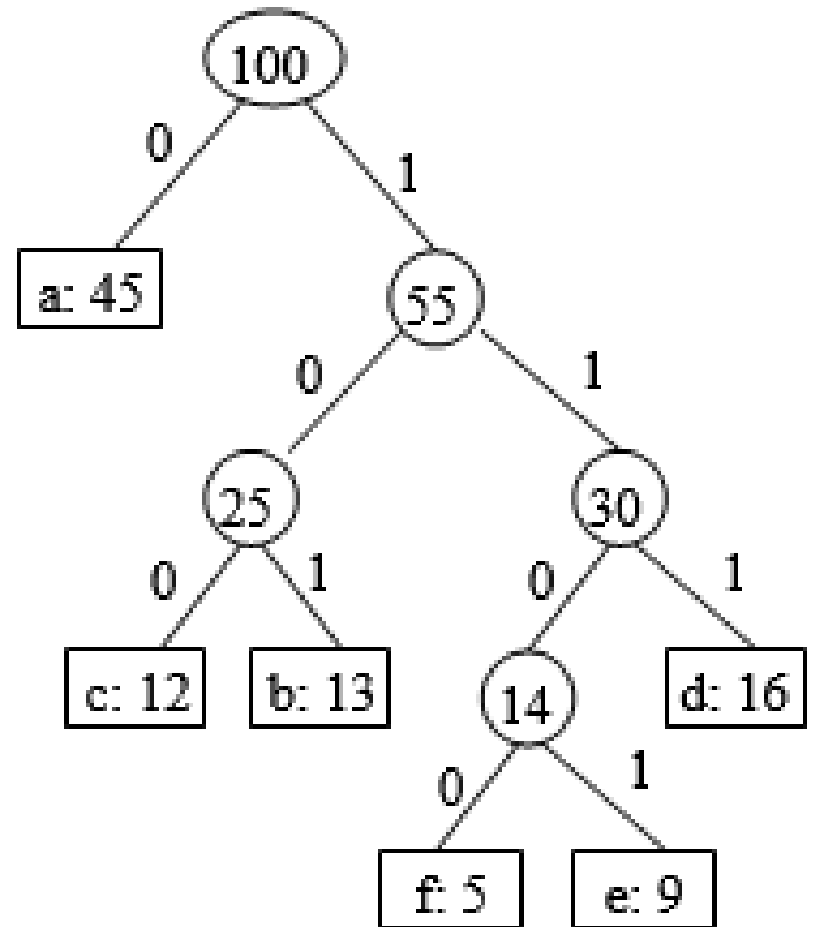
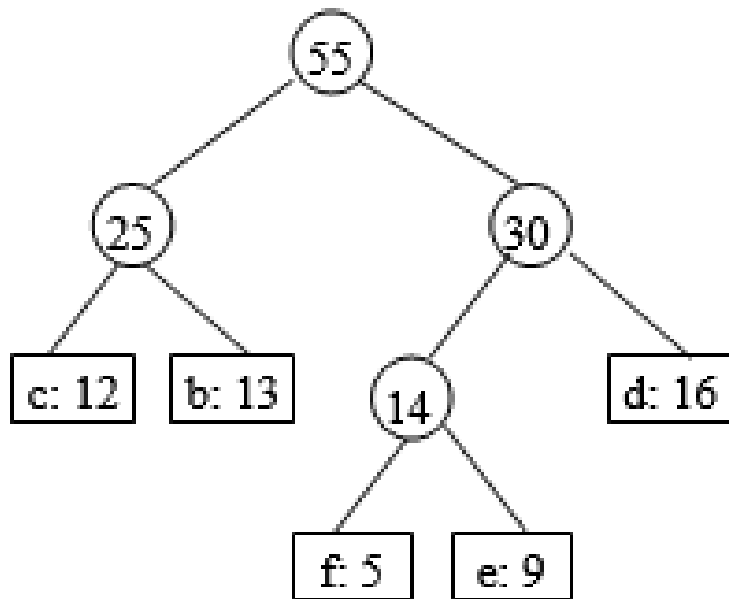
5.7 哈夫曼(Huffman)编码

f: 5 e: 9 d: 16 c: 12 b: 13 a: 45



5.7 哈夫曼(Huffman)编码

a: 45





END

课后思考题：

用哈夫曼编码完成对一个文件的压缩算法。
词频部分可以用统计规律。