

第5章 贪心算法

贪心算法(Greedy Algorithm)是一种构建最优解的高效算法策略。它在解决问题时,始终在每一步选择中都追求当前看来最优的局部方案,希望通过一系列局部最优的决策,最终达到整体最优。

这种策略就像人们在生活中面临选择时的思维方式:在不确定全局的情况下,人们往往会根据眼前条件做出看似合理的决定。这种“先解决当下、再看长远”的思路虽然不能保证每次都获得最优结果,但在某些问题中能奇迹般地达成最佳解。

贪心算法的启示:简单直接,有时也能高效可行。在学习过程中,当面对复杂问题不知从何下手时,不妨从“当下最清晰的部分”入手,一步步推进。只要方向正确,很多问题其实比我们想象的更容易突破。通过本章的学习,读者不仅能掌握贪心算法的核心思想与典型应用,还能体会到勇于尝试、灵活应变的重要性,在实践与思辨中实现能力的持续成长。

本章学习目标

- 理解贪心算法的基本概念。
- 掌握贪心算法适用的两个核心性质。
- 掌握典型贪心算法案例及其实现。
- 提升问题建模和算法设计能力。

5.1 贪心算法的基本要素

贪心算法通过逐步做出局部最优的选择,最终得到问题的解。它的核心思想是每一步选择当前看似最优的解,但不一定能够保证全局最优解。尽管如此,贪心算法在许多问题中能提供令人满意的解,甚至在某些情况下能得到最优解。例如,在第1章提到的找零钱问题,就是经典的贪心算法应用。

要判断一个问题是否适合用贪心算法来解决,以及能否通过贪心算法得到最优解,这个问题很难给予肯定的回答。然而,通过分析多个可用贪心算法求解的问题后,我们发现这些问题通常具有两个重要的性质:贪心选择性质和最优子结构性质。

5.1.1 贪心选择性质

贪心选择性质的核心思想是,问题的整体最优解可以通过一系列局部最优的选择来实现。具体来说,每一步选择都是在当前情况下做出的最优选择,即贪心选择,且这些局部最



视频讲解



优的选择最终能够组合成整体的最优解。这是贪心算法能够有效工作的基础,也是它与动态规划算法的主要区别。在动态规划中,每一步的决策依赖之前子问题的解,必须先解决这些子问题,再做出选择。而在贪心算法中,决策只考虑当前局部的最优解,不依赖未来的选择或已经解决的子问题的解。因此,贪心算法往往是自顶向下的逐步选择,每做出一次贪心选择,问题就简化为规模更小的子问题。

要判断一个问题是否适合用贪心算法求解,首先需要确认它是否具备贪心选择性质。通常用数学归纳法证明这一点。首先,假设问题有一个整体最优解,并尝试修改这个最优解,使其从贪心选择开始,即验证最优解的开头部分能否用贪心选择来替代。接着,通过归纳法证明每次做贪心选择后,问题会简化为规模更小的子问题,并且简化过程不会影响最优性,最终能够得到整体的最优解。



视频讲解

5.1.2 最优子结构性质

在贪心算法中,每次做出贪心选择后,问题会变得更简单,缩小为规模较小的子问题。这一过程的有效性依赖问题的最优子结构性质,即问题的最优解包含了子问题的最优解。因此,当问题具有最优子结构性质时,贪心选择能够简化问题,并通过一系列局部最优解逐步构建整体最优解。

举个例子,在经典的活动选择问题中,需要从一组活动中选出不重叠的最大数量的活动。贪心选择的策略是,每次选择结束时间最早的活动。可以证明,任何一个最优解的开始部分,都可以通过选择最早结束的活动来匹配。通过这样的选择,问题规模逐步减小,最终得到最优解。这种方法充分体现了贪心选择性质和最优子结构性质。

总的来说,贪心算法适用于那些满足贪心选择性质和最优子结构性质的问题,通过逐步做出局部最优解来实现全局最优解。在实际应用中,贪心算法不仅简单高效,尤其适用于那些可以通过局部决策逐步解决的大问题。



视频讲解

5.1.3 贪心算法与动态规划算法的差异

在贪心算法和动态规划算法这两种常用的算法策略中,它们都要求问题具有最优子结构性质,这意味着问题的最优解依赖于子问题的最优解。然而,尽管它们有这一共同点,选择使用贪心算法还是动态规划算法,还是要根据问题的具体情况来判断。下面通过两个经典的组合优化问题来探讨这两种算法的应用,帮助更好地理解它们的区别。

1. 活动选择问题(贪心算法)

假设有若干活动,每个活动都有开始时间和结束时间。目标是选择一个不重叠的活动子集,且使选择的活动数目最多。

该算法的贪心选择策略是每次选择结束时间最早的活动。这样做的原因是,选择一个结束时间早的活动可以为后续活动留下更多的时间,从而最大化可以选择的活动数量。每选择一个活动后,问题就变成了从剩下的活动中选择不重叠的活动子集。问题的最优解依赖之前的最优选择。因此,这个问题具有最优子结构性质。

在这个问题中,每次选择后剩下的问题规模变小,且每个选择只依赖当前状态,而不依赖未来的选择或以前的决策。贪心选择能够有效地解决这个问题,因为每一步的局部最优选择(选择结束时间最早的活动)最终会构成全局最优解。



2. 0-1 背包问题(动态规划算法)

0-1 背包问题用动态规划思想解决是通过状态转移方程,从小规模的子问题逐步构建出全局最优解。每个物品的选择都与背包当前的容量相关,因此动态规划通过迭代的方式,逐步解决子问题,最终得到全局最优解。该问题的关键在于每个物品的选择不仅与当前状态有关,还与之前的物品选择有关。因为背包的容量和物品的价值是相互依赖的,所以贪心算法无法保证每次选择局部最优时就能得到全局最优解。而动态规划通过构建状态表,考虑所有可能的物品选择组合,从而保证能得到全局最优解。

这两个经典问题展现了贪心算法和动态规划算法在适用场景上的不同。贪心算法适用于那些可以通过逐步做出局部最优选择,且局部最优选择能最终构成全局最优解的问题,如活动选择问题。而动态规划适用于那些问题具有较强的子问题重叠性质,并且每个决策都受之前决策影响的问题,如背包问题。

尽管这两种算法都依赖最优子结构性质,但贪心算法通过贪心选择局部最优解,动态规划则通过记忆化递归和状态转移构建全局最优解。选择使用哪种算法,关键在于问题的性质是否允许通过局部最优构建全局最优,若不允许则动态规划会是更合适的选择。

5.2 贪心算法经典案例



视频讲解

5.2.1 活动安排问题

1. 问题定义

给定一组活动集合 $S = \{1, 2, \dots, n\}$, 每个活动 i 有开始时间 s_i 和结束时间 f_i , 并且满足 $s_i < f_i$ 。两个活动 A_i 和 A_j 是相容的, 当且仅当它们的时间区间 $[s_i, f_i)$ 和 $[s_j, f_j)$ 不重叠, 即 $f_i \leq s_j$ 或 $f_j \leq s_i$ 。目标是从这些活动中选择一个最大相容的活动子集, 使得活动数量最多。

2. 问题分析

活动安排问题需要选出最大相容活动子集。如果用穷举法, 集合 $S = \{1, 2, \dots, n\}$ 的子集数量为 2^n 。这是指数级算法, 该算法随问题规模的增大, 运行时间会超出可接受范围, 因此不适用。

这里选择用贪心算法解决此问题, 关键是确定贪心选择策略。

1) 活动最早开始的顺序优先安排

可按活动开始时间的先后顺序来安排活动, 但是这样可能得不到最优解。例如, $S = \{1, 2, 3\}$, 活动时间分别为 $[s_1, f_1] = [0, 17]$, $[s_2, f_2] = [3, 5]$, $[s_3, f_3] = [8, 15]$, 如果按活动最早开始优先安排, 则只能安排活动 1, 因为活动 1 结束时, 活动 2 和活动 3 已经过了开始时间, 即活动 1 和活动 2、3 均不相容。但是 $s_3 > f_2$, 即活动 2 和活动 3 是相容的, 最大相容子集是 $A = \{2, 3\}$ 。所以不能选择这种贪心策略。

2) 按活动时间短的顺序优先安排

可按活动时间由短到长的顺序来安排活动。但是这样可能得不到最优解, 如 $S = \{1, 2, 3\}$, 活动时间分别为 $[s_1, f_1] = [0, 8]$, $[s_2, f_2] = [7, 10]$, $[s_3, f_3] = [9, 15]$, 排序后为活动 2, 3, 1。如果选择活动时间最短的活动 2 优先安排, 则活动 2 与活动 1、3 均不相容。然而,

由于 $s_3 > f_1$, 活动 1 和活动 3 是相容的。因此, 最大相容子集是 $A = \{1, 3\}$ 。所以不能选择这种策略。

3) 按活动最早结束的顺序优先安排

按活动最早结束的顺序优先安排, 也许可以安排更多的活动, 事实证明这一方案是可行的。下面证明这种贪心策略的正确性。

(1) 贪心选择性质。

如果活动是按结束时间从小到大进行排序的, 即有 $f_1 \leq f_2 \leq \dots \leq f_n$, 贪心算法执行到第 k 步时, 有 k 项活动已被选择, 生成一个选择序列 $i_1, i_2, \dots, i_k$ (按策略必有 $i_1 = 1$), 那么最优解 A 必然包含 $i_1, i_2, \dots, i_k$ 。

设 $S = \{1, 2, \dots, n\}$ 为所给的活动集合 (已按结束时间非减序排列), 活动 1 具有最早的完成时间。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。设 $A \subseteq S$ 是所给的活动安排问题的一个最优解, 且 A 中活动也按结束时间非减序排列, A 中的第一个活动是活动 k 。若 $k = 1$, 则 A 就是以贪心选择开始的最优解; 若 $k > 1$, 则设 $B = A - \{k\} \cup \{1\}$ 。由于 $f_1 \leq f_k$, 且 A 中活动是相容的, 故 B 中的活动也是相容的。又由于 B 中活动个数与 A 中活动个数相同, 且 A 是最优解, 故 B 也是最优解。也就是说 B 是以贪心选择活动 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。

(2) 最优子结构性质。

在做出贪心选择, 即选择活动 1 后, 原问题简化为对 S 中所有与活动 1 相容的活动进行活动安排的子问题 E' 。也就是说, 若 A 是原问题的最优解, 则 $A' = A - \{1\}$ 是活动安排问题 $E' = \{i \in S; s_i \geq f_1\}$ 的最优解。事实上, 如果能找到 E' 的一个解 B' , 它包含比 A' 更多的活动。将活动 1 加入 B' 中产生 S 的一个解 B , 它包含比 A 更多的活动。这与 A 的最优性矛盾。因此, 每一步所做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, 对贪心选择次数用数学归纳法即知, 贪心算法最终能生成原问题的最优解。

3. 算法描述

```
/* 定义活动存储结构 */
typedef struct{
    int No;                //活动序号
    int start;             //活动开始时间
    int end;               //活动结束时间
    int select;            //活动选择标识
}TIME;

void arrange(TIME t[]){
    int i, j=0;
    t[0].select=1;
    for(i=1; i<N; i++){
        if(t[i].start>t[j].end){
            t[i].select=1;
            j=i;
        }
    }
}
```



视频讲解



```

else
    t[i].select=0;
}
}

```

4. 算法分析

在 arrange 算法中,活动 i 的起始时间和结束时间分别存放在 $t[i].start$ 和 $t[i].end$ 两个变量中,算法执行是建立在所有活动按结束时间的非减序排列基础上的,如果活动未按减序排列,算法还需要额外花费 $O(n\log n)$ 的时间进行排序。

5.2.2 背包问题

背包问题与 0-1 背包问题类似,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, $1 \leq i \leq n$ 。



视频讲解

1. 问题描述

给定背包容量 $C > 0$, 以及 n 个物品的重量 $w_i > 0$, 价值 $v_i > 0$, $1 \leq i \leq n$, 要求构造一个 n 维向量 $(x_1, x_2, \dots, x_n)$, 满足 $0 \leq x_i \leq 1$, $1 \leq i \leq n$, 使得 $\sum_{i=1}^n w_i x_i \leq C$ 且 $\sum_{i=1}^n v_i x_i$ 的值最大。

2. 问题分析

0-1 背包问题与背包问题这两类问题都具有最优子结构性质。对于 0-1 背包问题, 设 A 是能够装入容量为 C 的背包的具有最大价值的物品集合, 则 $A_j = A - \{j\}$ 是 $n-1$ 个物品 $1, 2, \dots, j-1, j+1, \dots, n$ 可装入容量为 $C - w_j$ 的背包的具有最大价值的物品集合。对于背包问题, 类似地, 若它的一个最优解包含物品 j , 则从该最优解中拿出所含的物品 j 的那部分重量 w , 剩余的将是 $n-1$ 个原重物品 $1, 2, \dots, j-1, j+1, \dots, n$ 以及重为 $w_j - w$ 的物品 j 中可装入容量 $C - w$ 的背包且具有最大价值的物品。

虽然这两个问题极为相似, 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。用贪心算法解背包问题的基本步骤是, 首先计算每种物品单位重量的价值 v_i/w_i , 然后依据贪心选择策略, 将尽可能多的单位重量价值最大的物品装入背包。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。重复此过程, 直到背包装满为止。

3. 算法描述

```

//定义物品结构体
typedef struct {
    int weight;
    int value;
    double ratio; //物品单位价值 (value/weight)
} Item;

double knapSack(int C, Item items[], int n) {
    heapsort(items, n); //按单位价值从高到低排序
    double totalValue = 0.0; //记录背包的总价值
}

```

```
int currentWeight = 0; //当前背包的重量
for (int i = 0; i < n; i++) {
    if (currentWeight + items[i].weight <= C) {
        //如果当前物品可以完全放入背包
        currentWeight += items[i].weight;
        totalValue += items[i].value;
    } else {
        //如果当前物品不能完全放入背包,按比例放入
        int remainingWeight = C - currentWeight;
        totalValue += items[i].ratio * remainingWeight;
        break;
    }
}
return totalValue;
}
```

4. 算法分析

knapSack 算法的主要计算时间是将各种物品按其单位重量的价值从大到小排序。因此,算法的时间复杂度是 $O(n\log n)$ 。

5. 算法证明

1) 贪心选择性质

设原问题的物品集合为 S , 最优解为 A , 假定最优解 A 的第一步不满足贪心策略, 那么设 A 中第一个加入的物品为 i , 则 i 不是单位价值最大的物品。由假设可知在 $S - A$ 中存在物品 j , 它的单位价值更大, 显然将 i 中的一部分替换为 j 后可以得到更优解, 这与 A 是最优解矛盾, 从而证明满足贪心选择性质。

2) 最优子结构性质

设原问题的物品集合为 S , 最优解为 A , 其总价值为 $\text{Value}(A)$ 且 A 中包含物品 i , 那么 $A - \{i\}$ 就是 $S - \{i\}$ 在容量为 $C - v_i$ 下的最优解。

证明: 假定 $A - \{i\}$ 不是子问题的最优解, 那么假设最优解为 A' , 满足 $\text{Value}(A') > \text{Value}(A - \{i\}) = \text{Value}(A) - \text{Value}(v_i)$, 即 A 不是 S 的最优解, 这与假设矛盾, 从而证明该问题具有最优子结构性质。

5.2.3 哈夫曼编码

哈夫曼编码是一种高效的无损数据压缩算法, 通过变长字符编码对指定字符集进行压缩, 压缩率通常可达 20%~90%。它广泛应用于文件压缩(如文本、图像、音频格式)、网络传输优化、嵌入式系统数据存储以及搜索引擎索引压缩等场景, 是数据压缩领域的重要工具。

1. 问题描述

哈夫曼编码依据字符在文件中出现的频率表构建一个用 0、1 串表示各字符的最优编码方式。假设有一个数据文件中共有 6 种不同的字符出现, 各字符出现的频率如表 5.1 所示, 共包含 100000 个字符。要用压缩的方法存储它, 如何设计编码方案使压缩后存储成本最低?



视频讲解



视频讲解

表 5.1 字符出现的频率表

字 符	出 现 次 数	出 现 频 率	定 长 码	变 长 码	
				方 案 1	方 案 2
A	45000	0.45	000	0	0
B	13000	0.13	001	1	101
C	12000	0.12	010	10	100
D	16000	0.16	011	11	111
E	9000	0.09	100	100	1101
F	5000	0.05	101	101	1100

数据压缩有多种方法,0、1 码串是一种二进制压缩方法,即每个字符用唯一的 0、1 串表示。若每个字符使用固定长度二进制编码(定长码),则表示 6 个不同的字符需要的码长至少是 3,如表 5.1 中定长码一列。如果文件中有 100000 个字符,则使用定长码对整个文件进行编码需要 300000 位。能否用更短的编码实现呢?

变长码可以大幅缩短文件总码长,如表 5.1 给出了两种变长码编码方案:普通变长码和前缀码,这两种方案中每个字符编码长度不相等。对同样的文件,第一种变长码编码方案需要 156000 位,第二种需要 224000 位,两种方案对文件都有较大压缩,而且第一种压缩更大。

当文件采用编码方式存储,那么在读取时需要进行译码,如果编码不恰当就会引起译码歧义。例如,给定一串编码:01011101,如果采用表 5.1 变长码的方案 1 进行译码,那么可以有多种译码结果:

分解为: 0, 1, 0, 1, 1, 1, 0, 1 译为: ABABBBAB

分解为: 0, 10, 11, 101 译为: ACDF

分解为: 0, 101, 11, 0, 1 译为: AFDAB

.....

由于译码歧义,因此方案 1 不能用。那么什么样的变长码能够实现译码唯一性呢? 前缀码是一种特殊的编码方式,它的特点是:没有任何一个码字是另一个码字的前缀。也就是说,在前缀码中,没有一个编码会出现在另一个编码的开始部分。

前缀码的无前缀特性保证了它具有解码的唯一性,也使译码方法变的简单。在解码时,可以从编码的开始逐个比对,一旦遇到一个完整的码字,就能确认这个码字对应的字符,然后继续解码,直到所有的数据都被正确还原。

使用表 5.1 的方案 2 同样对 01011101 进行译码,读出的第一个字符 0 和字符 A 的编码对应,因此译为 A。原因是其他字符的编码都不是 0 开始的编码。接下来读字符 1,由于 BCDEF 的编码都由 1 开始,因此继续向下读 0,对应字符 BC 都含 10,所以继续向下读,当读出 1 时,只有字符 B 的编码是 101,因此将 101 译成 B,以此类推,即可准确译出这串编码对应的字符为 ABE。

前缀码的存储通常采用二叉树结构。在表示前缀码的二叉树中,树叶代表给定的字符,对应这个字符的前缀码是从树根到代表该字符的树叶的一条路径。如图 5.1 所示,规定每

个结点通向左孩子的边记作 0,通向右孩子的边记作 1。因此,A 的编码是 0,B 的编码是 101……

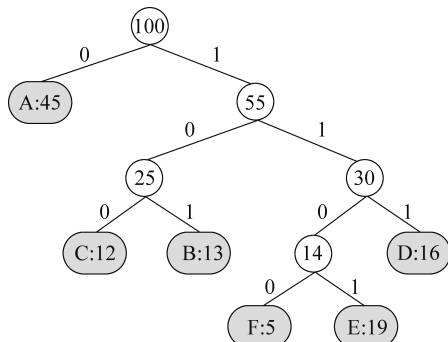


图 5.1 字符前缀码对应的二叉树

设 $C = \{x_1, x_2, \dots, x_n\}$ 是 n 个字符的集合,字符 x_i 的频率是 $f(x_i) (i = 1, 2, \dots, n)$, $d(x_i)$ 表示字符 x_i 使用的编码位数,即码长。那么存储一个字符所使用的二进制位数的平均值为

$$B = \sum_{i=1}^n f(x_i) d(x_i)$$

由于表示字符的前缀码是二叉树从根到树叶的路径长度,即树叶的深度,那么存储一个字符的平均二进制位数恰好就是这棵树在给定频率下的平均深度,也称为这棵树的权。因此,图 5.1 所示二叉树的一个字符平均需要的二进制位数为

$$B = 0.45 \times 1 + (0.12 + 0.13 + 0.16) \times 3 + (0.09 + 0.05) \times 4 = 0.45 + 1.23 + 0.56 = 2.24$$

不难看出,对应同一组频率可以构造出不同的二叉树,这些二叉树所对应的前缀码的平均字符占用的位数也不一样,占用位数越少,压缩效率越高。压缩效率最高的情况,即每个字符平均使用二进制位数最少的前缀码,称为最优前缀码。根据上面的分析,在给定频率下,对应最优前缀码的二叉树应该是平均深度最小(即权最小)的二叉树。如果叶结点数 n 恰好是 2^k ,且每个字符的频率都是 $1/n$,那么这棵树应该是一棵均衡的二叉树,每片树叶都分布在第 k 层上。但是,对于任意给定的 n 个字符频率 $f(x_1), f(x_2), \dots, f(x_n)$,如何构造一棵对应最优前缀码的二叉树?

2. 问题定义

给定字符集 $C = \{x_1, x_2, \dots, x_n\}$ 和每个字符的频率 $f(x_i) (i = 1, 2, \dots, n)$,如何求关于 C 的最优前缀码?

3. 问题分析

一个著名的构造最优前缀码的贪心算法是哈夫曼(Huffman)算法,由该算法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用于表示最优前缀码的二叉树 T 。哈夫曼算法从 $|C|$ 个叶结点开始,执行 $|C| - 1$ 次“合并”运算后产生最终所需的树 T 。

具体步骤如下。

(1) 将集合 C 中字符对应的频率看成 n 棵只有根结点的二叉树。

(2) 在这些二叉树中,选取两棵根结点权值最小的树作为左、右子树,构造一棵新的二叉树,新二叉树根结点的权值是其左、右子树根结点权值之和。



视频讲解



(3) 从原有的 n 棵二叉树集合中删除选取的两棵树,并将新构造的二叉树加入集合中。

(4) 重复步骤(2)和步骤(3),直到集合中只剩下一棵树,这棵树即为哈夫曼树。

用哈夫曼算法构造出的二叉树叫作哈夫曼树。根据哈夫曼树生成的编码称为哈夫曼编码。

下面以表 5.1 中的字符频率为例描述构建哈夫曼树和哈夫曼编码产生的过程。

初始状态,构造 n 棵只有根结点的二叉树,如图 5.2 所示。

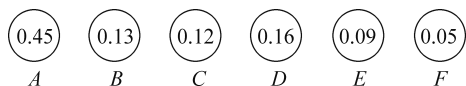


图 5.2 初始化,构建只有根结点的二叉树集合

从这些二叉树中选出权值最小的两棵树 E 和 F ,构建一棵新二叉树,新二叉树根结点权重为 0.14。将新二叉树加入集合,删除 E 和 F 两棵二叉树,如图 5.3 所示。

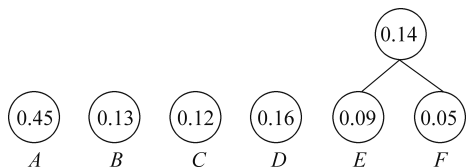


图 5.3 合并 E 和 F ,生成新的二叉树集合

再选取权值最小的两棵树 B 和 C ,构造新二叉树,二叉树根结点权值为 0.25。将新二叉树加入集合,删除 B 和 C 两棵二叉树,如图 5.4 所示。

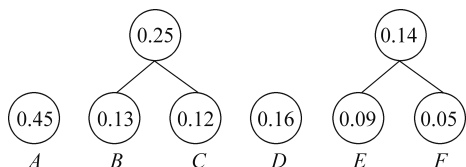


图 5.4 合并 B 和 C

重复上述过程,如图 5.5 和图 5.6 所示。最后生成如图 5.7 所示的哈夫曼树。

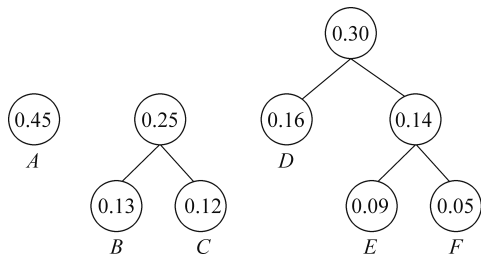


图 5.5 选择根结点权值最小的两棵二叉树继续合并(0.16 和 0.14)

构建完哈夫曼树,从根结点开始,沿着哈夫曼树的左、右分支进行编码,将左分支编为 0,右分支编为 1,可以得到叶结点对应字符的最优前缀码。

从构建好的哈夫曼树可以看出,使用频率高的字符用较短编码,而使用频率低的字符用

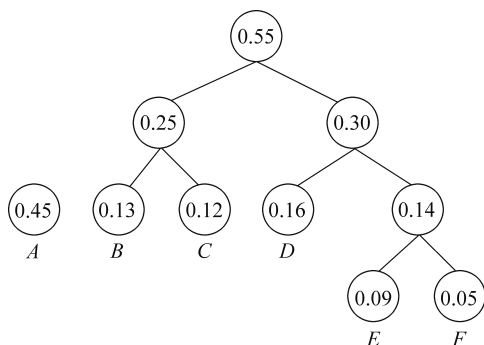


图 5.6 选择根结点权值最小的两棵二叉树继续合并(0.25 和 0.30)

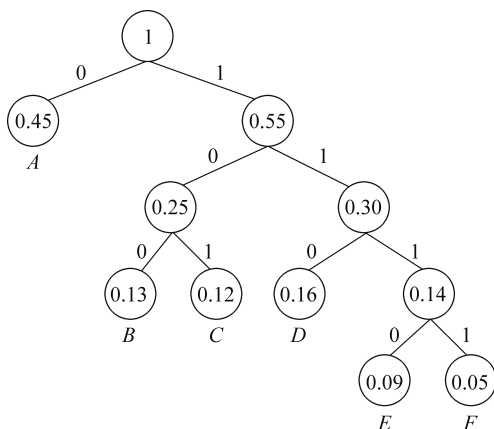


图 5.7 结果哈夫曼树

较长编码。采用生成的最优前缀码编码方案时,整个文件的总码长为 $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224000$ 位。它比用定长码方案的总码长减少了约 25%。

4. 算法描述



视频讲解

```
//定义哈夫曼树的结点
typedef struct{
    double weight;           //频率
    double lchild, rchild, parent; //左、右子结点和父结点编号
}huffNode;
//选择两个最小的权重且 parent==0 的结点
void select(huffNode t[], int n, int * s1, int * s2){
    * s1= * s2=-1;           //初始化为-1,表示没有找到有效的结点
    double min1=1e10;        //存储频率中的最小值
    double min2=1e10;        //存储频率中的次小值
    for(i=1; i<n; ++i)
        if(t[i].parent==0) //表示该结点没有父结点,是一个叶结点
            if (t[i].weight<min1){ //当前结点的权重小于最小值
                min2=min1; //将原最小值赋给次小值
            }
            else if (t[i].weight<min2)
                min2=t[i].weight;
    * s1=i;
    * s2=i;
}
```