回溯法的思想方法

问题的解空间和状态空间树

一、解空间

问题的解向量为 $X=(x_1,x_2,\cdots,x_n)$。 x_i 的取值范围为有穷集 S_i。把 x_i 的所有可能取值组合，称为问题的解空间。每一个组合是问题的一个可能解

例：0/1 背包问题， $S=\{0,1\}$，当 $n=3$ 时，0/1 背包问题的解空间是：

$$\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$$

当输入规模为 n 时，有 2^n 种可能的解。

例：货郎担问题， $S=\{1,2,\cdots,n\}$，当 $n=3$ 时， $S=\{1,2,3\}$。货郎担问题的解空间是：

$$\{(1,1,1),(1,1,2),(1,1,3),(1,2,1),(1,2,2),(1,2,3),\cdots,(3,3,1),(3,3,2),(3,3,3)\}$$

当输入规模为 n 时，它有 n^n 种可能的解。

考虑到约束方程 $x_i \neq x_j$。因此，货郎担问题的解空间压缩为：

$$\{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$$

当输入规模为 n 时，它有 $n!$ 种可能的解。

二、状态空间树：问题解空间的树形式表示

当 $n=4$ 时，货郎担问题的状态空间树。

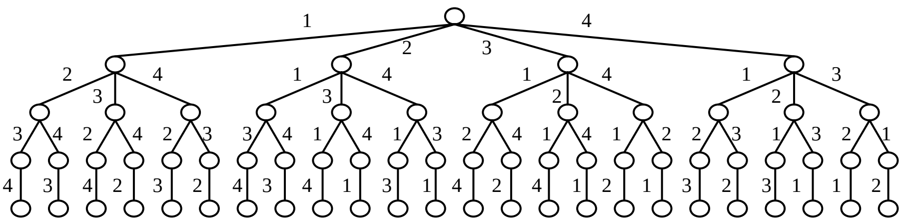

图 7.1　n=4 时货郎担问题的状态空间树

$n=4$ 时，0/1 背包问题的状态空间树

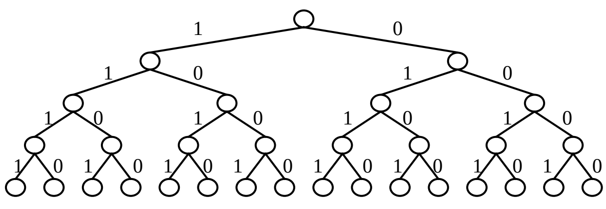

图 7.2　n=4 时背包问题的状态空间树

状态空间树的动态搜索

一、可行解和最优解

可行解：满足约束条件的解，解空间中的一个子集

最优解：使目标函数取极值（极大或极小）的可行解，一个或少数几个

例：货郎担问题，有 n^n 种可能解。$n!$ 种可行解，只有一个或几个解是最优解。

例：背包问题，有 2^n 种可能解，有些是可行解，只有一个或几个是最优解。

有些问题，只要可行解，不需要最优解，例如八后问题和图的着色问题

二、状态空间树的动态搜索

l _结点（活结点）：所搜索到的结点不是叶结点，且满足约束条件和目标函数的界，其儿子结点还未全部搜索完毕，

e _结点（扩展结点）：正在搜索其儿子结点的结点，它也是一个 l _结点；

d _结点（死结点）：不满足约束条件、目标函数、或其儿子结点已全部搜索完毕的结点、或者叶结点，。以 d _结点作为根的子树，可以在搜索过程中删除。

例 7.1　有 4 个顶点的货郎担问题，其费用矩阵如图 7.3 所示，求从顶点 1 出发，最后回到顶点 1 的最短路线。

∞	∞	1	7
8	∞	5	1
7	2	∞	6
2	5	3	∞

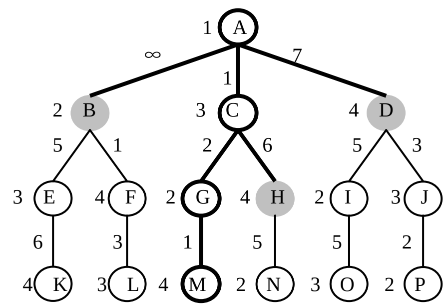

图 7.3　4 个顶点的货郎担问题的费用矩阵及搜索树

回溯法的一般性描述

题的解向量 $X=(x_0,x_1,\cdots,x_{n-1})$，

x_i 的取值范围 S_i， $S_i=\{a_{i.0},a_{i.1},\cdots,a_{i.m_i}\}$。

问题的解空间由笛卡尔积 $A=S_0\times S_1\times\cdots\times S_{n-1}$ 构成。

状态空间树看成为一棵高度为 n 的树，

第 0 层有 $|S_0|=m_0$ 个分支结点，构成 m_0 棵子树，每一棵子树都有 $|S_1|=m_1$ 个分支结点。

第 1 层，有 $m_0\times m_1$ 个分支结点，构成 $m_0\times m_1$ 棵子树。

第 $n-1$ 层，有 $m_0\times m_1\cdots\times m_{n-1}$ 个结点，它们都是叶子结点。

初始化令解向量 X 为空。

在第 0 层，置 $x_0=a_{0.0}$，

$m[i]$：集合 S_i 的元素个数， $|S_i|=m[i]$；

$x[i]$：解向量 X 的第 i 个分量；
$k[i]$：当前算法对集合 S_i 中的元素的取值位置。
回溯方法作如下的一般性描述：

```
void backtrack_item()
{
    initial(x);
    i = 0;  k[i] = 0;  flag = FALSE;
    while (i>=0) {
        while (k[i]<m[i]) {
            x[i] = a(i,k[i]);
            if (constrain(x)&&bound(x)) {
                if (solution(x)) {
                    flag = TRUE;   break;
                }
                else {
                    i = i + 1;  k[i] = 0;
                }
            }
            else k[i] = k[i] + 1;
        }
        if (flag) break;
        i = i – 1;
    }
    if (!flag)
        initial(x);
}
```

initial(x)把解向量初始化为空；
a(i, k[i])取 S_i 的第 $k[i]$ 个值，赋给解向量的分量 $x[i]$。
函数 constrain(x)判断解向量是否满足约束条件，如果满足，返回值为真。
bound(x)判断解向量是否满足目标函数的界，如果满足，返回值为真。
solution(x)判断解向量是否为问题的最终解，如果是，标志 *flag* 置为真，
回溯法解题时，包含下面三个步骤：

1. 对所给定的问题，定义问题的解空间；
2. 确定状态空间树的结构；
3. 用深度优先搜索方法搜索解空间，用约束方程和目标函数的界对状态空间树进行修剪，生成搜索树，取得问题的解。

0/1 背包问题

不需把背包的载重量划分为 m 等分、物体的重量是背包载重量 m 等分的整数倍的限制。

回溯法解 0/1 背包问题的求解过程

一、解空间和状态可树

n 个物体 v_i，重量 w_i、价值 p_i，$0 \le i \le n-1$，背包的载重量 M。

x_i：物体 v_i 被装入背包的情况，$x_i = 0,1$。

约束方程和目标函数：

$$\sum_{i=1}^{n} w_i x_i \le M \qquad (7.5.1)$$

$$optp = \max \sum_{i=1}^{n} p_i x_i \qquad (7.5.2)$$

解向量：$X = (x_0, x_1, \cdots, x_{n-1})$，

状态空间树：高度为 n 的完全二叉树，其结点总数有 $2^{n+1}-1$ 个。

根结点到叶结点的路径，是问题的可能解。

假定：第 i 层的左儿子子树，物体 v_i 被装入背包的情况；右儿子子树，物体 v_i 未被装入背包的情况。

二、求解过程

初始化：目标函数上界为 0，物体按价值重量比的非增顺序排序，

搜索过程：尽量沿左儿子结点前进，当不能沿左儿子继续前进时，就得到问题的一个部分解，并把搜索转移到右儿子子树。

估计由部分解所能得到的最大价值，

估计值高于当前上界：继续由右儿子子树向下搜索，扩大部分解，直到找到可行解；

保存可行解，用可行解的值刷新目标函数的上界，向上回溯，寻找其它可行解；

若估计值小于当前上界：丢弃当前正在搜索的部分解，向上回溯。

三、部分解的最大估价值

假定，当前部分解是 $\{x_0, x_1, \cdots, x_{k-1}\}$，同时，有：

$$\sum_{i=0}^{k-1} x_i w_i \le M \quad 且 \quad \sum_{i=0}^{k-1} x_i w_i + w_k > M \qquad (7.5.3)$$

将得到部分解 $\{x_0, x_1, \cdots, x_k\}$，其中，$x_k = 0$。由这个部分解继续向下搜索，将有：

$$\sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i \le M \quad 且 \quad \sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i + w_{k+m} > M \qquad (7.5.4)$$

$m=1,2,\cdots,n-k-1$，当 $m=1$ 时，表示继续装入物体 v_{k+1}，仍然将超过背包的载重量。

能够找到的可能解的最大值不会超过：

$$\sum_{i=0}^{k} x_i p_i + \sum_{i=k+1}^{k+m-1} x_i p_i + (M - \sum_{i=0}^{k-1} x_i w_i - \sum_{i=k+1}^{k+m-1} x_i w_i) \times p_{k+m} / w_{k+m} \qquad (7.5.5)$$

四、回溯的两种情况

当估计值小于目标函数的上界（它是已经得到的可行解中的最大值），向上回溯：

当前的结点是左儿子分支结点，就转而搜索相应的右儿子分支结点；

当前的结点是右儿子分支结点，就沿右儿子分支结点向上回溯，直到左儿子分支结点为止，然后，再转而搜索相应的右儿子分支结点。

五、步骤

w_cur：部分解中装入背包物体的总重量

p_cur：部分解中装入背包物体的总价值；

p_est：部分解可能达到的最大估计值；

p_total：当前搜索到的所有可行解中的最大价值，目标函数的上界。

x_k：部分解的第 k 个分量

y_k：部分解的第 k 个分量的拷贝

k：搜索深度。

回溯法解 0/1 背包问题的步骤：

1. 物体按价值重量比的非增顺序排序；
2. w_cur、p_cur、p_total、k 初始化为 0，部分解初始化为空；
3. 按（7.5.4）和（7.5.5）式估计从当前的部分解可取得的最大价值 p_est；
4. 如果 $p\_est > p\_total$，转 5；否则转 8；
5. 从 v_k 开始把物体装入背包，直到没有物体可装、或装不下物体 v_i 为止，生成部分解 $y_k,\cdots,y_i$，$k \le i < n$；刷新 p_cur；
6. 如果 $i \ge n$，得到新的可行解，所有 y_i 拷贝到 x_i，$p\_total = p\_cur$；令 $k = n$，转 3，以便回溯搜索其它的可能解；
7. 否则，得到一个部分解，令 $k = i+1$，舍弃物体 v_i，从物体 v_{i+1} 继续装入，转 3；
8. 当 $i \ge 0$ 并且 $y_i = 0$，执行 $i = i-1$，直到条件不成立；即沿右儿子分支结点方向向上回溯，直到左儿子分支结点；
9. 如果 $i < 0$，算法结束；否则，转 10；
10. 令 $y_i = 0$，$w\_cur = w\_cur - w_i$，$p\_cur = p\_cur - p_i$，$k = i+1$，转 3；从左儿子分支结点转移到相应的右儿子分支结点，继续搜索其它的部分解或可能解；

例 7.4 有载重量 $M = 50$ 的背包，物体重量分别为 5, 15, 25, 27, 30，物体价值分别为 12, 30, 44, 46, 50。求最优装入背包的物体及价值。

1. p_total 为 0，计算 $p\_est = 94.5$，大于 p_total，生成结点 1, 2, 3, 4，部分解 (1, 1, 1, 0)；
2. 在结点 4 计算 $p\_est = 94.3$，大于 p_total，继续向下搜索生成结点 5，得到价值为 86 的可行解 (1, 1, 1, 0, 0)，保存在解向量 X 中，p_total 更新为 86；

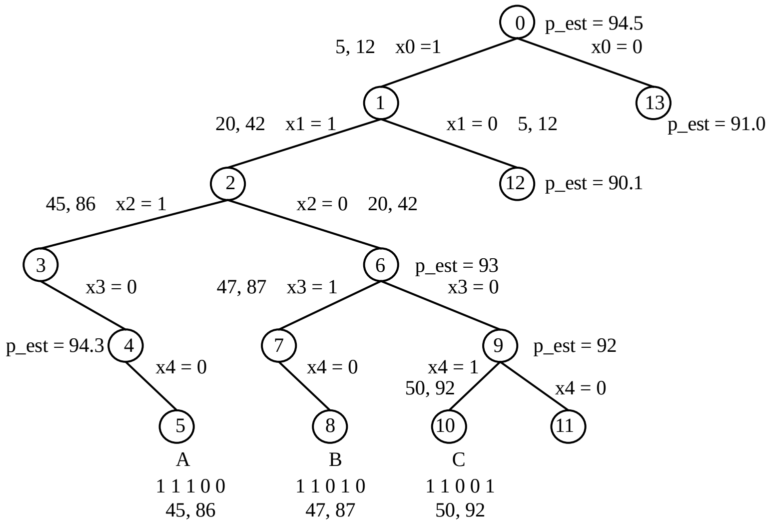

图 7.13 例 7.4 中 0/1 背包问题的搜索树

3. 由叶结点 5 继续搜索时，p_est 被置为 86，不大于 p_total 的值，因此，沿右儿子分支结点回溯到左儿子分支结点 3，生成右儿子分支结点 6，得到部分解 (1, 1, 0)；
4. 在结点 6 计算 $p\_est = 93$，大于 p_total，因此，生成结点 7, 8, ，得到价值为 87 的可行解 (1, 1, 0, 1, 0)，更新解向量 X，p_total 更新为 87；
5. 由叶结点 8 继续搜索时，p_est 被置为 87，不大于 p_total 的值，因此，沿右儿子分支结点回溯到左儿子分支结点 7，生成右儿子分支结点 9，得到部分解 (1, 1, 0, 0)；
6. 在结点 9 计算 $p\_est = 92$，大于 p_total，生成结点 10，得到价值为 92 的可行解 (1, 1, 0, 0, 1)，更新解向量 X，p_total 更新为 92；
7. 由叶结点 10 继续搜索时，p_est 被置为 92，不大于 p_total 的值，因此，进行回溯，因为结点 10 是左儿子结点，生成右儿子结点 11，得到可行解 (1, 1, 0, 0, 0)；
8. 由叶结点 11 继续搜索时，p_est 被置为 42，不大于 p_total 的值，因此，沿右儿子分支结点回溯到左儿子分支结点 2，生成右儿子分支结点 12，得到部分解 (1, 0)；
9. 在结点 12 计算 $p\_est = 90.1$，小于 p_total，因此，回溯到左儿子分支结点 1，生成右儿子分支结点 13，得到部分解 (0)；
10. 在结点 13 计算 $p\_est = 91.0$，小于 p_total，因此，向上回溯到根结点 0，结束算法。最后，由向量 X 中的内容，得到最优解 (1, 1, 0, 0, 1)，从 p_total 中得到最大价值 92。

状态空间树的 63 个结点
被访问的结点数为 14 个。

回溯法解 0/1 背包问题算法的实现

数据结构和变量：

```
typedef struct {
    float     w;          /* 物体重量 */
    float     p;          /* 物体价值 */
    float     v;          /* 物体的价值重量比 */
} OBJECT;
OBJECT    ob[n];
float     M;              /* 背包载重量 */
int       x[n];           /* 可能的解向量 */
int       y[n];           /* 当前搜索的解向量 */
float     p_est;          /* 当前搜索方向装入背包物体的估计最大价值 */
float     p_total;        /* 装入背包的物体的最大价值的上界 */
float     w_cur;          /* 当前装入背包的物体的总重量 */
float     p_cur;          /* 当前装入背包的物体的总价值 */
```

0/1 背包问题的回溯算法：

算法 7.4 0/1 背包问题的回溯算法
输入：背包载重量 M,问题个数 n,存放物体的价值和重量的结构体数组 Ob[]
输出：0/1 背包问题的最优解 x[]

```
float knapsack_back(OBJECT ob[],float M,int n,BOOL x[])
{
    int i,k;
    float w_cur,p_total,p_cur,w_est,p_est;
    BOOL *y = new BOOL[n];
    for (i=0;i<=n;i++) {               /* 计算物体的价值重量比 */
        ob[i].v = ob[i].p / ob[i].w;
        y[i] = FALSE;                  /* 当前的解向量初始化 */
    }
    merge_sort(ob,n);                  /* 物体按价值重量比的非增顺序排序*/
    w_cur = p_cur = p_total = 0;       /* 当前背包中物体的价值重量初始化*/
    k = 0;                             /* 已搜索到的可能解的总价值初始化*/
    while (k>=0) {
```

```
    w_est = w_cur;   p_est = p_cur;
    for (i=k;i<n;i++) {            /* 沿当前分支可能取得的最大价值 */
        w_est = w_est + ob[i].w;
        if (w_est<M) {
            p_est = p_est + ob[i].p;
        else {
            p_est = p_est + ((M – w_est + ob[i].w) / ob[i].w) * ob[i].p;
            break;
        }
    }
    if (p_est>p_total) {                   /* 估计值大于上界 */
        for (i=k;i<n;i++) {
            if (w_cur+ob[i].w<=M) {        /* 可装入第 i 个物体 */
                w_cur = w_cur + ob[i].w;
                p_cur = p_cur + ob[i].p;
                y[i] = TRUE;
            }
            else {
                y[i] = FALSE;   break;     /* 不能装入第 i 个物体 */
            }
        }
        if (i>=n) {                        /* n 个物体已全部装入 */
            if (p_cur>p_total) {
                p_total = p_cur;   k = n;  /* 刷新当前上限 */
                for (i=0;i<n;i++)          /* 保存可能的解 */
                    x[i] = y[i];
            }
        }
        else k = i + 1;                /* 继续装入其余物体 */
    }
    else {                             /* 估计价值小于当前上限*/
        while ((i>=0)&&(y[i]==0)       /* 沿着右分支结点方向回溯*/
            i = i – 1;                 /* 直到左分支结点 */
        if (i<0) break;                /* 已到达根结点,算法结束 */
        else {
            w_cur = w_cur – ob[i].w;   /* 修改当前值 */
            p_cur = p_cur – ob[i].p;
            y[i] = FALSE;   k = i + 1; /* 搜索右分支子树 */
        }
```

```
        }
    }
    delete y;
    return p_total;
}
```

工作空间为$\Theta(n)$。

算法在最坏情况下所花费的时间：$O(n2^n)$

合并排序，需花费$\Theta(n\log n)$时间；

在最坏情况下，状态空间树有$2^{n+1}-1$个结点，有$O(2^n)$儿子结点，

每个右儿子结点都需估计继续搜索可能取得的目标函数的最大价值，每次估计时间需花费$O(n)$时间，因此，右儿子结点需花费$O(n2^n)$时间，