第5章

# 回　溯　法

## 5.1　概　　述

在现实世界中，很多问题没有（至少目前没有）有效的算法，这些问题的求解只能通过蛮力穷举搜索来实现。但蛮力法需要生成并评估所有的解，执行效率低下。

回溯法（BackTracking Method）是一种深度优先搜索算法，该方法试图穷举所有可能的解，因此，回溯法能够确保获得最优解。在搜索过程中根据问题约束等规则对搜索树进行剪枝操作，跳过部分搜索区域，提高算法性能。回溯法按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再选择另外一个方向试探的技术称为回溯法，满足回溯条件的某个状态的点称为“回溯点”。回溯法把问题的解空间转化成了图或者树的结构表示，然后使用深度优先搜索策略进行遍历，遍历的过程中记录和寻找所有可行解或者最优解。

## 5.2　回溯法设计思路

复杂问题常常有很多可能解，这些可能解构成问题的解空间。确定了问题的解空间结构后，回溯法将从开始结点（根结点）出发，以深度优先的方式搜索整个解空间。搜索过程从开始结点开始搜索，将开始结点进行扩展。在当前的扩展结点处，向纵深方向搜索并移至一个新结点，这个新结点就成为一个新的活结点，并成为当前的扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前的扩展结点就成为死结点。此时应回溯至最近的一个活结点处，并使其成为当前的扩展结点。回溯法以上述工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活结点时为止。

运用回溯法解题的关键要素有以下 3 点。

（1）针对给定的问题，定义问题的解空间。

（2）确定易于搜索的解空间结构。

（3）以深度优先方式搜索解空间，并且在搜索过程中用剪枝函数避免无效搜索。

回溯法的算法框架如下。

```
void back_trace(int t) {
    if(t ⩾ n && x is better than optimal_x)          //当前解更优
        optimal_x=x;                                  //保留当前解
        update_bound(x);                              //更新剪枝函数,回溯至上一级
    else
        for(int i=s; i<=e; i++) {
            x[t]=node(i);
            if(constraint(t) && bound(t))
                back_trace(t+1);
        }
}
```

其中,t 表示递归深度,即当前扩展结点在解空间树中的深度;n 是解空间树的高度。当 t≥n 时,算法已搜索到一个叶子结点,即当前解,此时对当前解进行判断,如果比已搜索到的最优解更好,则更新最优解,同时,更新剪枝函数。已搜索到的最优解越接近问题的最优解,则后续剪枝效果越好。s 和 e 表示在当前扩展结点处子树的起始编号和终止编号;node(i)表示在当前扩展结点处的第 i 个结点;函数 constraint(t)和 bound(t)分别表示当前扩展结点处的约束函数和剪枝函数。深度优先搜索继续进行的条件是满足问题约束且该结点处不剪枝;否则,剪去相应的结点。

## 5.3　回溯法示例与过程分析

### 5.3.1　*n* 皇后问题

**例 5.1**　在 $n\times n$ 格的国际象棋棋盘上摆放 $n$ 个皇后(见图 5.1,此时 $n=8$),使其不能互相攻击,即任意两个皇后都不能处于同一行、同一列或同一斜线上,有多少种摆法?

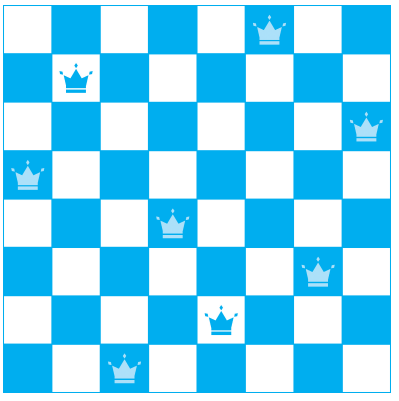

图 5.1　八皇后问题示意图

$n$ 皇后是由八皇后问题演变而来的。该问题由国际象棋棋手马克斯·贝瑟尔于 1848 年提出:在 8×8 格的国际象棋棋盘上摆放 8 个皇后,使其不能互相攻击,即任意两个皇后都不能处于同一行、同一列或同一斜线上,求有多少种摆法。高斯认为有 76 种方案。1854 年,在柏林的象棋杂志上不同的作者发表了 40 种不同的解,后来有人用图论的方法解出 92 种结果。

下面用数组模拟棋盘,从第一行开始,依次选择位置,如果当前行中皇后位置满足条件,则选择下一行皇后的位置;如果不满足条件,那么当前位置后移一位继续搜索。

下面以四皇后为例说明利用回溯法求解 $n$ 皇后问题的过程。

步骤 1:在第 1 行放第一个皇后不受约束,可以在 4 个位置上随意放置。为了对整个解

空间进行遍历,每一个位置都不能漏掉。因此,将第 1 行的皇后放在第 1 行第 1 列的位置开始搜索。

步骤 2:当第一个皇后放在第 1 列后,根据问题约束,第 2 行的皇后只能放在第 3 列或者第 4 列,如图 5.2(a)所示。选择在第 2 行第 3 列放置第二个皇后,如图 5.2(b)所示。

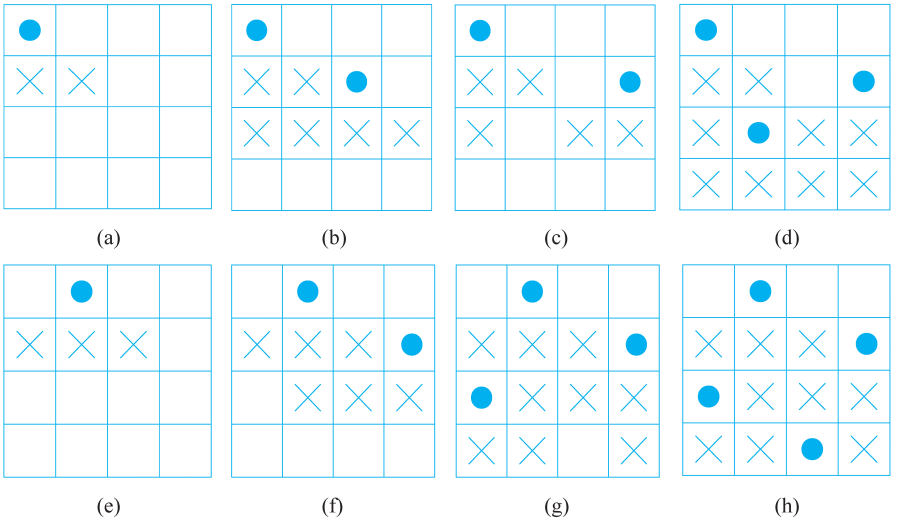


图 5.2 回溯法求解四皇后问题

步骤 3:如图 5.2(b)所示,按照以上两步放置皇后将导致第三个皇后没有合适的位置进行放置,此时,搜索将无法继续,需要回溯重新决策第二个皇后的位置,如图 5.2(c)所示。

步骤 4:根据前面的决策,第三个皇后有一个合适的位置进行放置,即第 2 列。将第 3 行第 2 列放置第三个皇后,则第四个皇后没有合适的位置放置,如图 5.2(d)所示。

步骤 5:由于第四个皇后无法放置,则回溯重新决策第三个皇后的位置,选择下一个合适的位置放置第三个皇后,由于第三行中无新的位置可选,因此,需要回溯重新决策第二个皇后的位置,但第二个皇后可能的位置都已被搜索,所以,最终回溯重新决策第一个皇后的位置,如图 5.2(e)所示。

步骤 6:第一个皇后放在第 2 列,第二个皇后可选位置为第 4 列。根据问题约束,当将第二个皇后放置在第 4 列,第三个皇后的可能位置只有第 1 列,如图 5.2(f)所示。

步骤 7:将第三个皇后放置于第 1 列,第四个皇后的可选位置为第 3 列。当将第四个皇后放置于第 3 列,此时,已搜索至解的叶子结点,得到第一个可行解。

步骤 8:由于第四个皇后无新的位置可以放置,回溯重新决策第三个皇后的位置;由于第三个皇后无新的位置可以放置,回溯重新决策第二个皇后的位置;同理,由于第二个皇后无新的位置放置,回溯重新决策第一个皇后的位置;第一个皇后还有第 3 列、第 4 列两个位置可以搜索,此时,将第一个皇后放置于第 3 列,重新开始搜索,寻找新的可行解。按照上述过程遍历解空间树,即可得到所有可行解。

值得注意的是,问题的求解并不需要用 $n\times n$ 的数组来表示结果,而只需要一个 $n$ 长度的数组来表示每一行的皇后位置即可,即 arr[$i$]=$k$,表示第 $i$ 行的皇后位置为 $k$。此时使用一个 arr[$n$]的数组就可以表示一个解,通过回溯可以得到所有可行解。

$n$ 皇后问题的回溯法算法如下所示。

```
//输入：n,表示皇后个数
//输出：n 皇后问题的所有可行解
步骤 1：初始化解向量 arr[i]={-1};
步骤 2：i=1;
步骤 3：while(i>=1)
    步骤 3.1 把皇后 i 摆放在下一列,即 arr[i]++,如果 arr[i]>n,则置 arr[i]=-1,i--,转步骤 3;
    步骤 3.2 考查 arr[i]位置,如果不发生冲突,则转步骤 3.3;否则转步骤 3 回溯;
    步骤 3.3 若 n 个皇后已全部摆放,输出可行解,转步骤 3 继续搜索下一个可行解,否则,i++,转步骤 3;
步骤 4：退出循环,搜索结束。
```

### 5.3.2　0-1 背包问题


0-1 背包问题

**例 5.2**　有一个背包，最多能承载 10kg，现在有 3 个物品，质量分别为[4，8，5]、价值分别为[24，40，20]，详情如表 5.1 所示。那么应该如何选择才能使得你的背包背走最多价值的物品？

表 5.1　0-1 背包问题示例

| 物　品 | 质量($w$)/kg | 价值($v$)/元 | 价值/质量($v/w$) |
|---|---|---|---|
| 1 | 4 | 24 | 6 |
| 2 | 8 | 40 | 5 |
| 3 | 5 | 20 | 4 |

**解题思路：**对于每一个物品 $i$，该物品只有选与不选两种选择，总共有 $n$ 个物品，可以顺序依次考虑每个物品，这样就形成了一棵解空间树。求解的基本思想就是遍历这棵树，以枚举所有情况，最后进行判断，如果质量不超过背包容量，且价值最大，该方案就是问题的解。

定义：$c_r$ 为背包剩余质量、$w[i]$为第 $i$ 个物品的质量、$C$ 为背包总质量、$c_v$ 为当前背包价值、$r$ 为剩余物品总价值、bestv 为当前最优解的值。

在决策第 $i$ 个物品是否选择时，当前状态的左子树表示选择装入，右子树表示选择不装入。

**剪枝策略如下。**

(1) 约束函数：如果第 $i$ 个物品加入背包使得约束条件不成立，即 $c_r<w[i]$，则无法装入第 $i$ 个物品，剪去左子树。

(2) 限界函数：如果剩余所有物品价值与当前背包价值之和小于当前的最优值，即 $c_v+r\leqslant$bestv，说明在当前状态无法搜索到更优的物品组合，放弃在右子树上进一步搜索。

表 5.1 所示问题搜索过程如下。

(1) 从根结点 A 开始按深度优先搜索。首先决策第一个物品是否装入背包。此时，$C=10$，$w[1]=4$，$r=84$，bestv$=0$，$c_v=0$。显然，$C>w[1]$，可以在左子树上进行搜索。

(2) 左子树根结点为 B，由于 $c_r=10-4=6<8$，所以第二个物品不能放入背包，因此，剪枝左子树；由于此时 bestv$=0$，可以在 B 的右子树 E 上进行搜索。

(3) 在右子树 E 上决策第三个物品是否放入,由于 $c_r=10-4=6>5$,因此,可以将第三个物品放入背包,进入左子树 J 搜索。由于 J 为叶子结点,说明搜索到了一个可行解,该可行解为(1,0,1),所对应的背包价值 total_val=44>bestv(初值为 0),保留当前最优解:bestv=44。

(4) 搜索到可行解后回溯至 J 的父结点 E,判断是否进入 E 的右子树搜索。此时 $c_v=24$,$r=0$,显然 $c_v+r\leqslant$bestv,所以右子树剪枝。至此,E 的左右子树搜索完毕,回溯至父结点 B;B 的左右子树已搜索完毕,回溯至父结点 A。

(5) 在结点 A 处 $c_v=0$,$r=60>$bestv,所以可以进入右子树 C 搜索。此时,$c_r>w[2]$,可以进入左子树 F 搜索。

(6) 在左子树 F 处,$c_r=10-8=2<5$,所以第三个物品不能放入背包,剪枝左子树 L,进入右子树 M 搜索。由于 M 为叶子结点,所以搜得一个新的可行解(0,1,0)。该解对应的背包价值 total_val=40<bestv,不更新 bestv。

(7) M 搜索完毕回溯至父结点 F,此时,F 左右子树已搜索完毕,回溯至父结点 C。在结点 C 处,$c_v+20<$bestv,所以右子树没有必要搜索,直接剪枝。

(8) C 的左右子树搜索完毕,回溯至父结点 A。此时,A 的左右子树搜索完毕,解空间中的每一个解都得到了遍历,所搜索到的 bestv 即为问题的最优解。

0-1 背包问题的回溯法求解过程如图 5.3 所示。

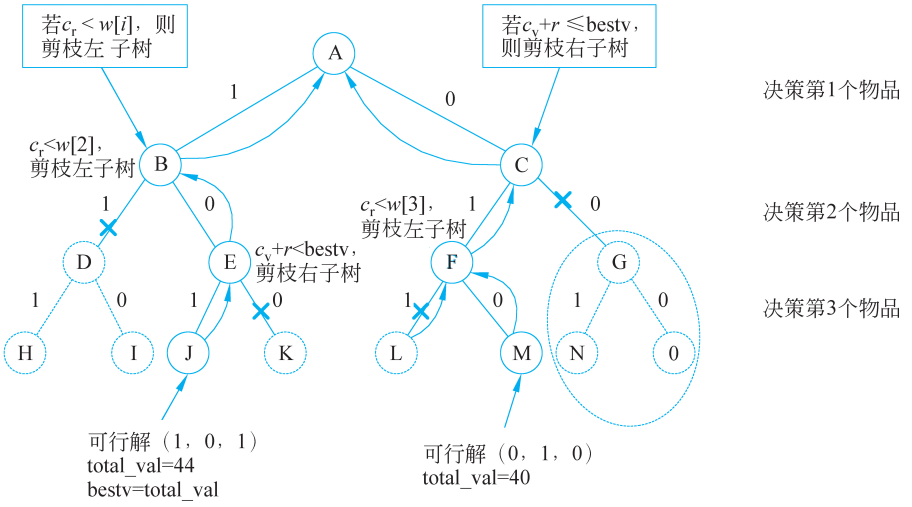


图 5.3 0-1 背包问题的回溯法求解过程

0-1 背包问题的回溯法算法如下所示。

```
//输入:背包总质量 C,物品个数 n,第 i 个物品的质量 w[i],价值 v[i]
//输出:问题的最优解选择的物品 x0[](x0[i]为 1 代表选择第 i 件物品,x0[i]为 0 代表不选择),bestv 为最优解的总价值
//x[],x[i]为 1 代表选择第 i 件物品,为 0 代表不选择
cr ← C                                                    //背包剩余质量
```

```
cv ← 0                                          //背包当前价值
r ← v[1]+v[2]+…+v[n]                            //剩余物品总价值
bestv ← 0                                       //最优解的总价值

knapwack(i,r,cv,cr)//i 为第 i 件物品,r 为剩余物品总价值,cv 为背包当前价值,cr 为背包
                   //剩余质量
{
    if cr>=0 and cv>bestv then                  //找到并更新最优解
        bestv ← cv;
        for j ← 1 to i
            x0[j] ← x[j]
        end for
        for j ← i+1 to n
            x0[j] ← 0
        end for
    end if
    if cv+r<=bestv then                         //限界函数剪枝
        return
    end if
    if cr>=w[i] then                            //只有 cr>=w[i]才需要搜索左子树
        x[i] ← 1
        knapwack(i+1,r-v[i],cv+v[i],cr-w[i]) //搜索左子树
        x[i] ← 0
    end if
    knapwack(i+1,r-v[i],cv,cr);                 //搜索右子树
}
```

### 5.3.3　图的 $m$ 着色问题

**例 5.3**　给定无向连通图 $G=(V,E)$ 和正整数 $m$，求最小的整数 $m$，使得用 $m$ 种颜色对 $G$ 中的顶点着色后任意两个相邻顶点着色不同。


图的 $m$ 着色问题

**解题思路：**问题要求任意相邻顶点颜色不同。如图 5.4(a)所示，有 A、B、C、D、E 共 5 个顶点组成的无向图 $G$，如果采用图 5.4(b)的着色方法，可以看到相邻顶点 B、C 着色相同，因此该解为错误解；如果采用图 5.4(c)的着色方法，则满足题目条件，因此该解为可行解。

给定足够多的颜色，即 $m$ 足够大，问题一定存在解。图的 $m$ 着色问题的目标是找到最小的 $m$ 使得图的着色方案可行。

对于有 $n$ 个顶点的图，每个顶点有 $m$ 种着色方案，整个问题解空间构成一棵深度为 $n$ 的 $m$ 叉树，可以利用回溯法在解空间树种遍历所有路径，搜索到最小的 $m$。由于问题的解空间树规模大，需要有效的方案进行剪枝操作，避免不必要的操作。

**剪枝策略如下。**

(1) 约束函数：图的相邻顶点颜色不能相同，如果给某个顶点着色导致相邻顶点颜色相同，则剪去该扩展分支。

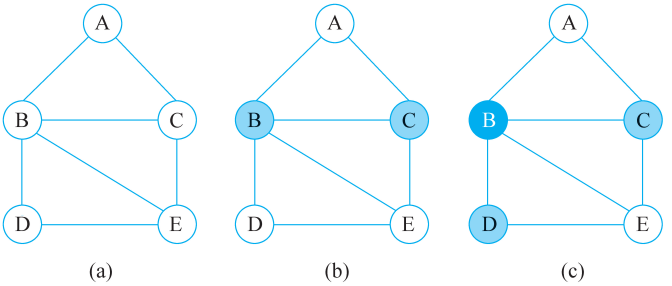


图 5.4 图的着色问题示例

(2) 限界函数：用变量 $m$ 记录当前最优解(最小值)，在搜索的过程中记录部分解中已经使用颜色数量 clr，如果 clr $\geqslant m$，则终止当前解的搜索，回溯到合适的位置再行搜索其他解。

例 5.3 所示问题解空间树搜索过程如图 5.5 所示。

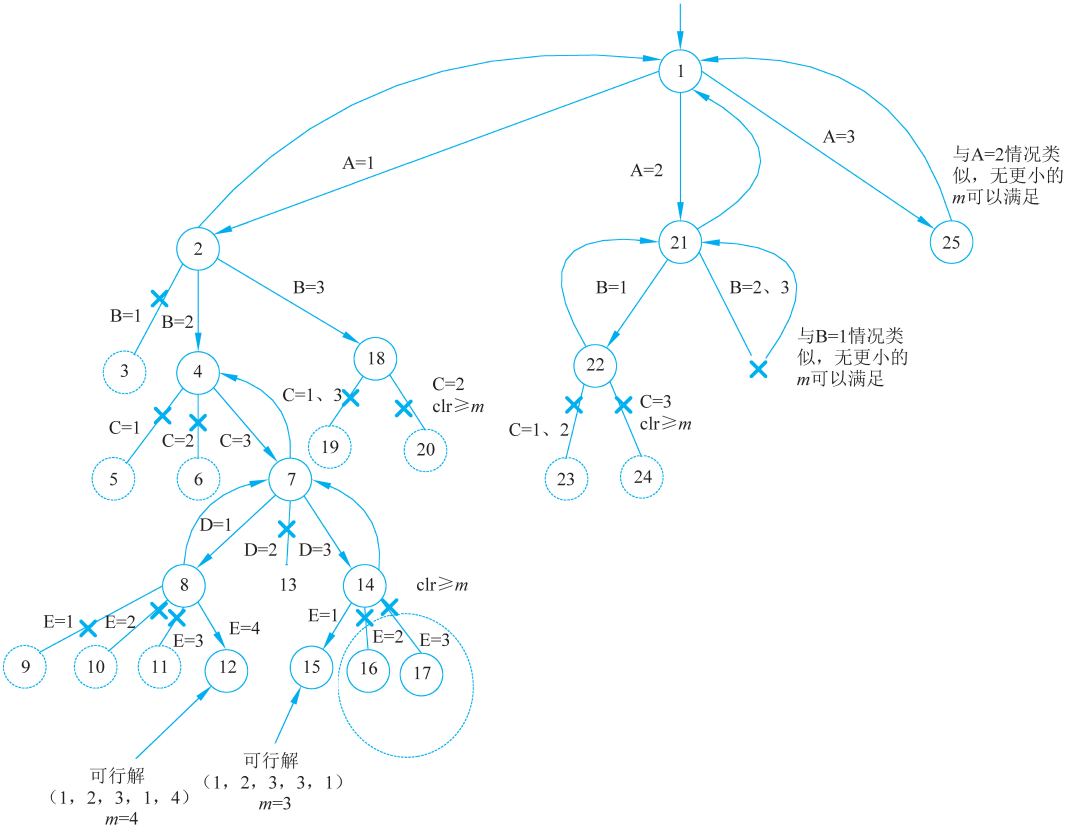


图 5.5 图的 $m$ 着色问题解空间树搜索过程

对于 $n=5$ 的图 5.4 所示问题，求最小颜色数 $m$ 的过程如下。

(1) 搜索树从状态 1 开始，为顶点 A 赋 1 号颜色，此时没有颜色冲突，满足问题约束，转入状态 2。

(2) 为顶点 B 赋 1 号色，与 A=1 冲突，所以该分支被剪枝；接着为 B 赋 2 号色，没有颜

色冲突，满足问题约束，转入状态 4。

(3) 为顶点 C 赋 1、2 号色，分别与 A=1、B=2 冲突，所以只能为 C 赋 3 号色，没有颜色冲突，满足问题约束，转入状态 7。

(4) 为顶点 D 赋 1 号色，没有颜色冲突，满足问题约束，转入状态 8。

(5) 为顶点 E 赋 1、2、3 号色，分别与 B=2、C=3、D=1 冲突，所以只能为 E 赋 4 号色。此时，搜索已到达解空间树的叶子结点，着色方案(1,2,3,1,4)是一个可行方案，总共用了 $m=4$ 种颜色。

(6) 为了遍历解空间树，从状态 12 回溯至状态 8，状态 8 下的 4 种选择已遍历完毕，继续回溯至状态 7，为顶点 D 赋 2 号色，与 B=2 的方案冲突，所以该分支被剪切，继续为顶点 D 赋 3 号色，满足问题约束，转入状态 14。

(7) 在状态 14 结点为顶点 E 赋 1 号色，满足问题约束，此时搜索到达解空间树的叶子结点，着色方案(1,2,3,3,1)是一个可行解，总共用了 $m=3$ 种颜色。

(8) 从状态 15 回溯至状态 14，此时部分解为(1,2,3,3)，已用了 clr=3 种颜色，继续在状态 14 下探索 E 的着色方案并不能优化 $m$ 的值，所以这些虚线内所有分支都被剪枝，以减少搜索空间。

(9) 从状态 14 回溯至状态 7，此时部分解为(1,2,3)，已用了 clr=3 种颜色，所以状态 7 下其他分支被剪枝；继续回溯至状态 4，此时部分解为(1,2)，使用颜色数量为 clr=2，$m$ 存在优化空间，但此时状态 4 已遍历了 $m=3$ 种颜色，所以只能回溯至状态 2。

(10) 在状态 2 下可以为顶点 B 赋 3 号色转入状态 18，在状态 18 下为顶点 C 赋 1 或者 3 号色均不满足问题约束；因此，只能为顶点 C 赋 2 号色，但此时部分解已使用了 clr=3 种颜色，无法优化 $m$，所以这些分支都被剪枝。

(11) 此时，A=1 的分支全部搜索完毕，回溯至状态 1 并赋 A=2 转入状态 21 继续进行搜索，由于状态 21 下的分支不能满足问题约束或者无法优化 $m$ 都被剪枝，回溯至状态 1 并赋 A=3 转入状态 25 继续进行搜索，具体过程与 A=2 类似，其扩展结点先后被剪枝。

经过搜索，可得 $m$ 的最优解为 3，即至少需要 3 种颜色才能为图 5.4 所示的无向图进行着色且相邻顶点颜色不同。

搜索图的 $m$ 着色问题的最小 $m$ 值伪代码如下。

```
//color[]表示可行解,color[i]表示为第 i 个顶点赋的颜色号
m←∞
search_min_clr(i){
    clr ← cnt_clr(color[1..i -1])   //cnt_clr 用于统计部分解中颜色数量
    if clr>=m then                  //部分解使用颜色数量大于或等于最优解 m
        return                      //剪枝回溯至上一个状态
    end if
    if i>n and clr<m then           //涂色完成且使用颜色小于最优解,记录最优解至 m 中
        m ← clr
        return                      //回溯至上一个状态
    end if
    for j ← 1 to m
```

```
            color[i] ← j
            if judge(i) then            //judge 用于判断任何相邻顶点颜色是否不同,否则剪枝
                search_min_clr(i+1)     //搜索下一个顶点颜色
            end if
            color[i] ←0
        end for
    }
```

### 5.3.4 批处理作业调度问题

**例 5.4** 设有 $n$ 个作业{1,2,…,$n$}要在两台机器上处理,每个作业要先在机器 1 上处理,再在机器 2 上处理,请给出一种作业调度方案,使得所有任务完成的总时间最短。

**解题思路:** 问题的目标是求批处理作业调度完成的最短时间,基本策略是尽量保证每台机器的空闲时间最小。显然,第一台机器在调度作业时不受加工顺序制约,所以,第一台机器不能空闲,否则会推迟作业在第二台机器上的开始加工时间。

以表 5.2 所示问题说明不同的作业调度顺序会影响完成的时间。

表 5.2 批处理作业调度问题示例

| 作　　业 | 机器 1 加工时间 | 机器 2 加工时间 |
|---|---|---|
| $J_1$ | 2 | 1 |
| $J_2$ | 4 | 2 |
| $J_3$ | 3 | 3 |

#### 1. 调度方案一:$J_1 \rightarrow J_2 \rightarrow J_3$

从图 5.6 可知,完成所有任务所需时间为 12。

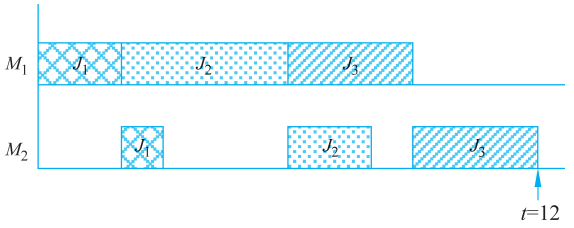


图 5.6 调度方案一完成任务所需时间

#### 2. 调度方案二:$J_3 \rightarrow J_1 \rightarrow J_2$

从图 5.7 可知完成所有任务所需时间为 11。

显然,调度顺序决定最终完成工作的时间。为得到最优调度顺序,需要搜索所有可能的调度方案。设共有 $n$ 个作业,未调度任何作业为状态 1,在状态 1 下可以调度任一作业,当确定状态 1 所调度的作业后,下一级结点只能从剩下的 $n-1$ 个作业中选择,因此,解空间树是一棵排列树,如图 5.8 所示。

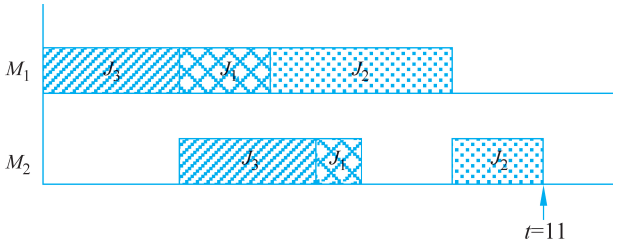


图 5.7　调度方案二完成任务所需时间

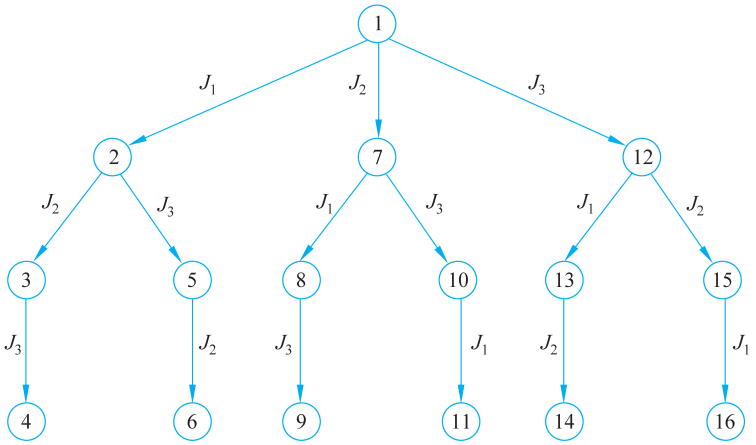


图 5.8　解空间树

图 5.8 所示的排列树中包含了 3 种作业的全排列，只要搜索整棵解空间树，一定可以搜索到最优调度方案。显然，全排列树随着问题规模增加解空间急剧膨胀，是一个典型的 NP 难问题。因此，回溯法只适用于求解小规模的问题，而且需要对解空间树进行剪枝以提高搜索速度。

定义 $a[i]$为第 $i$ 个作业在第一台机器上的加工时间，$b[i]$为第 $i$ 个作业在第二台机器上的加工时间，m1[1..$n$]为机器 1 上已调度作业的结束时间，m2[1..$n$]为机器 2 上已调度作业结束的结束时间，$x[1..n]$表示 $n$ 个作业的调度方案。根据图 5.9 可得作业在两台机器上的结束时间。

根据图 5.9 可知第 $k$ 个作业在机器 1、机器 2 上的结束时间分别为

$$\text{m1}[k] = \text{m1}[k-1] + a[x[k]]$$

$$\text{m2}[k] = \max(\text{m1}[k], \text{m2}[k-1]) + b[x[k]]$$

设当前搜索到的最优调度所需时间为 $t_{best}$，当前搜索调度第 $k$ 个作业，剩余作业在机器 2 上加工时间为 sum，则可按如下规则剪枝。

剪枝规则：如果 m2[$k$]+sum$\geqslant t_{best}$，则剪枝，如图 5.10 所示。

批处理作业调度问题的回溯算法可表示如下。

```
//输入：第 i 个作业在机器 1 上的处理时间 a[i],在机器 2 上的处理时间 b[i]
//输出：最优调度序列 x0[1..n]
//x[1..n],当前调度序列
//f[1..n],f[i]为 1 代表第 i 个作业已被处理,为 0 代表未被处理
```