第5章

回溯法——深度优先搜索

CHAPTER 5

深度优先搜索是在明确给出了图中的各顶点及边(显式图)的情况下,按照深度优先搜索的思想对图中的每个顶点进行搜索,最终得出图的结构信息。回溯法是在仅给出初始节点、目标节点及产生子节点的条件(一般由问题题意隐含给出)的情况下,构造一个图(隐式图),然后按照深度优先搜索的思想,在有关条件的约束下扩展到目标节点,从而找出问题的解。换言之,回溯法从初始状态出发,在隐式图中以深度优先的方式搜索问题的解。当发现不满足求解条件时就回溯,并尝试其他路径。通俗地讲,回溯法是一种“能进则进,进不了则换,换不了则退”的基本搜索方法。


思想引领


视频讲解

5.1 概述

1. 回溯法的算法框架及思想

(1) 回溯法的算法框架。回溯法是一种搜索方法。用回溯法解决问题时,首先应明确搜索范围,即问题所有可能解组成的范围。这个范围越小越好,且至少包含问题的一个(最优)解。为了定义搜索范围,需要明确以下 4 方面。

① 问题解的形式。回溯法希望问题的解能够表示成一个 n 元组$(x_1,x_2,\cdots,x_n)$的形式。

② 显约束。对分量 $x_i(i=1,2,\cdots,n)$的取值范围限定。

③ 隐约束。为满足问题的解而对不同分量之间施加的约束。

④ 解空间。对于问题的一个实例,解向量满足显约束的所有 n 元组构成了该实例的一个解空间。

注意:同一个问题的显约束可能有多种,相应解空间的大小就会不同,通常情况下,解空间越小,算法的搜索效率就越高。

【例 5-1】 n 皇后问题。在 $n\times n$ 的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则,皇后可以攻击与之处在同一行

或同一列或同一斜线上的棋子。换言之，n 皇后问题等价于在 $n\times n$ 的棋盘上放置 n 个皇后，任意两个皇后不同行、不同列、不同斜线。

问题分析：根据题意，首先考虑显约束为不同行的解空间。

① 问题解的形式。n 皇后问题的解表示成 n 元组 $(x_1,x_2,\cdots,x_n)$ 的形式，其中 $x_i(i=1,2,\cdots,n)$ 表示第 i 个皇后放置在第 i 行第 x_i 列的位置。

② 显约束。n 个皇后不同行。

③ 隐约束。n 个皇后不同列或不同斜线。

④ 解空间。根据显约束，第 $i(i=1,2,\cdots,n)$ 个皇后有 n 个位置可以选择，即第 i 行的 n 个列位置，即 $x_i\in\{1,2,\cdots,n\}$，显然满足显约束的 n 元组共有 n^n 种，它们构成了 n 皇后问题的解空间。

如果将显约束定义为不同行且不同列，则问题的隐约束为不同斜线，问题的解空间为第 i 个皇后不能放置在前 $i-1$ 个皇后所在的列，故第 i 个皇后有 $n-i+1$ 个位置可以选择，令 $S=\{1,2,3,\cdots,n\}$，则 $x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}$。因此，$n$ 皇后问题解空间由 $n!$ 个 n 元组组成。

显然，第二种表示方法使得问题的解空间明显变小，因此搜索效率更高。

其次，为了方便搜索，一般用树或图的形式将问题的解空间有效地组织起来。如例 5-1 的 n 皇后问题：显约束为不同行的解空间树($n=4$)，如图 5-1 所示，显约束为不同行且不同列的解空间树($n=4$)，如图 5-2 所示。树的节点代表状态，树根代表初始状态，树叶代表目标状态；从树根到树叶的路径代表放置方案；分支上的数据代表 x_i 的取值，也可以说是将第 i 个皇后放置在第 i 行，第 x_i 列的动作。

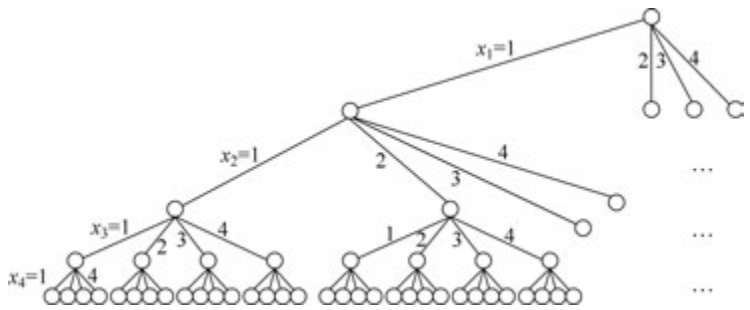

图 5-1　显约束为不同行的解空间树($n=4$)

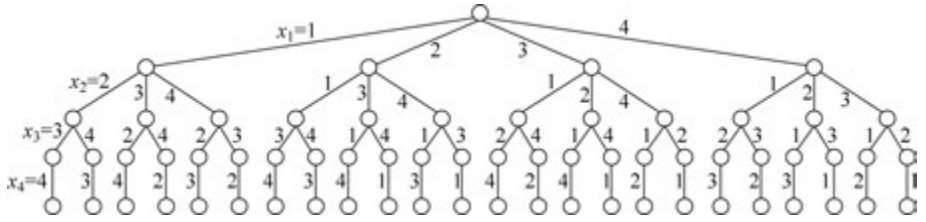

图 5-2　显约束为不同行且不同列的解空间树($n=4$)

最后，搜索问题的解空间树。在搜索的过程中，需要了解以下三个名词。

① 扩展节点：一个正在生成孩子的节点称为扩展节点。

② 活节点：一个自身已生成但其孩子还没有全部生成的节点称为活节点。

③ 死节点：一个所有孩子已经生成的节点称为死节点。

(2) 搜索思想。从根开始，以深度优先搜索的方式进行搜索。根节点是活节点并且是当前的扩展节点。在搜索过程中，当前的扩展节点沿纵深方向移向一个新节点，判断该新节点是否满足隐约束，如果满足，那么新节点成为活节点，并且成为当前的扩展节点，继续深一层的搜索；如果不满足，那么换到该新节点的兄弟节点(扩展节点的其他分支)继续搜索；如果新节点没有兄弟节点，或其兄弟节点已全部搜索完毕，那么扩展节点成为死节点，搜索回溯到其父节点处继续进行。搜索过程直到找到问题的解或根节点变成死节点为止。

从回溯法的搜索思想可知，搜索开始之前必须确定问题的隐约束。隐约束一般是考查解空间结构中的节点是否有可能得到问题的可行解或最优解。如果不可能得到问题的可行解或最优解，就不用沿着该节点的分支继续搜索，需要换到该节点的兄弟节点或回到上一层节点。也就是说，在深度优先搜索的过程中，不满足隐约束的分支被剪掉，只沿着满足隐约束的分支搜索问题的解，从而避免了无效搜索，加快了搜索速度。因此，隐约束又称为剪枝函数。隐约束(剪枝函数)一般有两种：一是判断是否能够得到可行解的隐约束，称为约束条件(约束函数)；二是判断是否有可能得到最优解的隐约束，称为限界条件(限界函数)。可见，回溯法是一种具有约束函数或限界函数的深度优先搜索方法。

总之，回溯法的算法框架主要包括以下三部分。

① 针对所给问题，定义问题的解空间。

② 确定易于搜索的解空间组织结构。

③ 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

2. 回溯法的构造实例

【例 5-2】 用回溯法解决四皇后问题。

按照回溯法的搜索思想，首先应确定根节点是活节点，并且是当前的扩展节点 R。它扩展生成一个新节点 C，若 C 不满足隐约束，则舍弃；若满足，则 C 成为活节点并成为当前的扩展节点，搜索继续向纵深处进行(此时节点 R 不再是扩展节点)。在完成对子树 C(以 C 为根的子树)的搜索之后，节点 C 变成了死节点。开始回溯到离死节点 C 最接近的活节点 R，节点 R 再次成为扩展节点。若扩展节点 R 还存在未搜索过的子节点，则继续沿 R 的下一个未搜索过的子节点进行搜索；直到找到问题的解或者根节点变成死节点为止。

用回溯法解决四皇后问题的求解过程设计如下。

第一步，定义问题的解空间。

设四皇后问题解的形式是4元组(x_1,x_2,x_3,x_4)，其中 $x_i(i=1,2,3,4)$代表第 i 个皇后放置在第 i 行第 x_i 列，x_i 的取值为1,2,3,4。

第二步，确定解空间的组织结构。

确定显约束：第 i 个皇后和第 j 个皇后不同行，即 $i \neq j$，对应的解空间的组织结构如图5-1所示。

第三步，搜索解空间。

(1) 确定约束条件。第 i 个皇后和第 j 个皇后不同列且不同斜线，即 $x_i \neq x_j$ 并且 $|i-j| \neq |x_i-x_j|$。

(2) 确定限界条件。该问题不存在放置方案是否好坏的情况，所以不需要设置限界

条件。

(3) 搜索过程，如图 5-3～图 5-8 所示。根节点 A 是活节点，也是当前的扩展节点，如图 5-3(a)所示。扩展节点 A 沿着 $x_1=1$ 的分支生成子节点 B，节点 B 满足隐约束，B 成为活节点，并成为当前的扩展节点，如图 5-3(b)所示。扩展节点 B 沿着 $x_2=1$ 和 $x_2=2$ 的分支生成的子节点不满足隐约束，舍弃；沿着 $x_2=3$ 的分支生成的子节点 C 满足隐约束，C 成为活节点，并成为当前的扩展节点，如图 5-3(c)所示。扩展节点 C 沿着所有分支生成的子节点均不满足隐约束，全部舍弃，活节点 C 变成死节点。开始回溯到离它最近的活节点 B，节点 B 再次成为扩展节点，如图 5-3(d)所示。

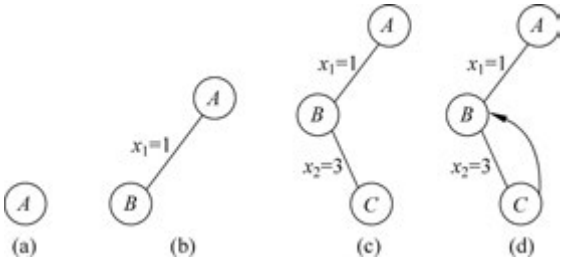

图 5-3　四皇后问题的搜索过程(一)

扩展节点 B 沿着 $x_2=4$ 的分支继续生成的子节点 D 满足隐约束，D 成为活节点，并成为当前的扩展节点，如图 5-4(a)所示。扩展节点 D 沿着 $x_3=1$ 的分支生成的子节点不满足隐约束，舍弃；沿着 $x_3=2$ 的分支生成的子节点 E 满足隐约束，E 成为活节点，并成为当前的扩展节点，如图 5-4(b)所示。扩展节点 E 沿着所有分支生成的子节点均不满足隐约束，全部舍弃，活节点 E 变成死节点。开始回溯到最近的活节点 D，D 再次成为扩展节点，如图 5-4(c)所示。扩展节点 D 沿着 $x_3=3,4$ 的分支生成的子节点均不满足隐约束，舍弃，活节点 D 变成死节点。开始回溯到最近的活节点 B，B 再次成为扩展节点，如图 5-4(d)所示。

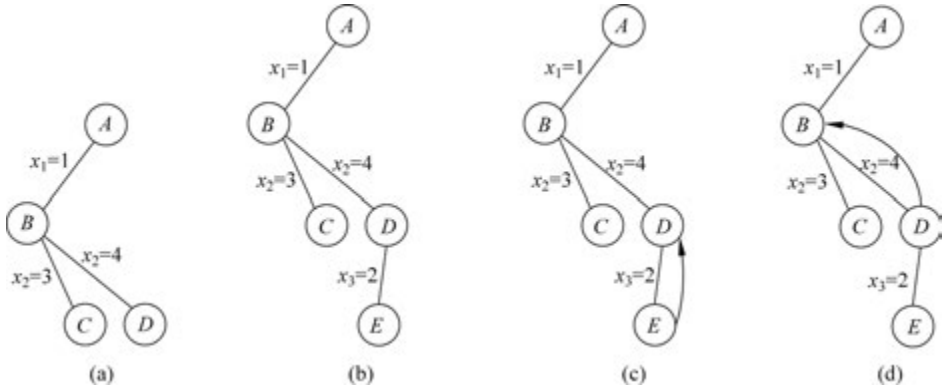

图 5-4　四皇后问题的搜索过程(二)

此时扩展节点 B 的子节点均搜索完毕，活节点 B 成为死节点。开始回溯到最近的活节点 A，节点 A 再次成为扩展节点，如图 5-5(a)所示。扩展节点 A 沿着 $x_1=2$ 的分支继续生成的子节点 F 满足隐约束，节点 F 成为活节点，并成为当前的扩展节点，如图 5-5(b)所示。扩展节点 F 沿着 $x_2=1,2,3$ 的分支生成的子节点均不满足隐约束，全部舍弃；继续沿着 $x_2=4$ 的分支生成的子节点 G 满足隐约束，节点 G 成为活节点，并成为当前的扩展节点，如

图 5-5(c)所示。

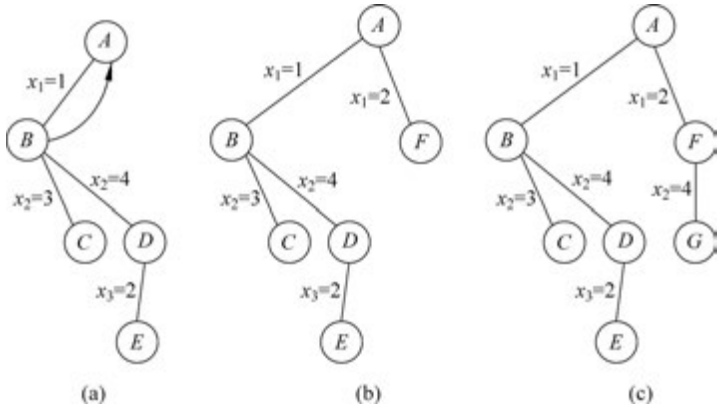

图 5-5 四皇后问题的搜索过程(三)

扩展节点 G 沿着 $x_3=1$ 的分支生成的子节点 H 满足隐约束,节点 H 成为活节点,并成为当前的扩展节点,如图 5-6(a)所示。扩展节点 H 沿着 $x_4=1,2$ 的分支生成的子节点均不满足隐约束,舍弃;沿着 $x_4=3$ 的分支生成的子节点 I 满足隐约束。此时搜索过程搜索到了叶子节点,说明已经找到一种放置方案,即(2,4,1,3),如图 5-6(b)所示。继续搜索其他放置方案,从叶子节点 I 回溯到最近的活节点 H,H 又成为当前扩展节点,如图 5-6(c)所示。

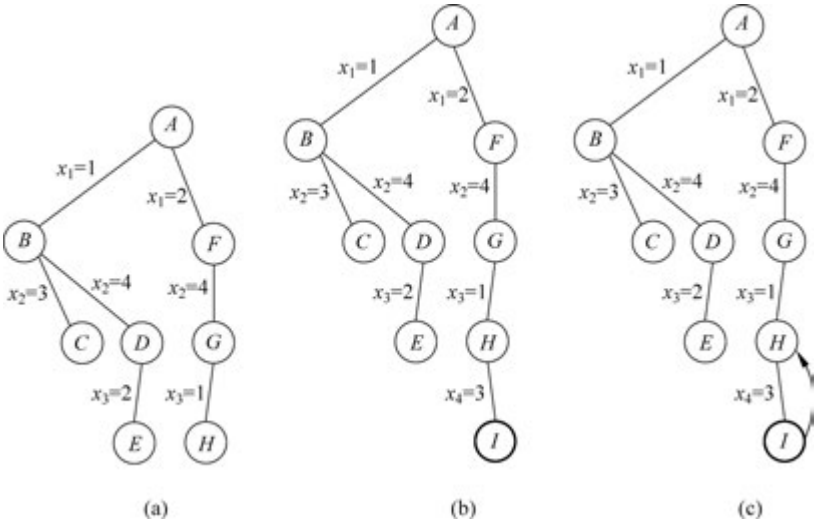

图 5-6 四皇后问题的搜索过程(四)

扩展节点 H 继续沿着 $x_4=4$ 的分支生成的子节点不满足隐约束,舍弃;此时节点 H 的 4 个分支全部搜索完毕,H 成为死节点,回溯到活节点 G,如图 5-7(a)所示。节点 G 又成为当前的扩展节点,沿着 $x_3=2,3,4$ 的分支生成的子节点均不满足隐约束,舍弃;节点 G 成为死节点,回溯到活节点 F,如图 5-7(b)所示。节点 F 的 4 个分支均搜索完毕,继续回溯到活节点 A,节点 A 再次成为当前的扩展节点,如图 5-7(c)所示。

扩展节点 A 沿着 $x_1=3,4$ 分支的扩展过程与沿着 $x_1=1,2$ 分支的扩展过程类似,这里不再详述。最终形成的树如图 5-8 所示。

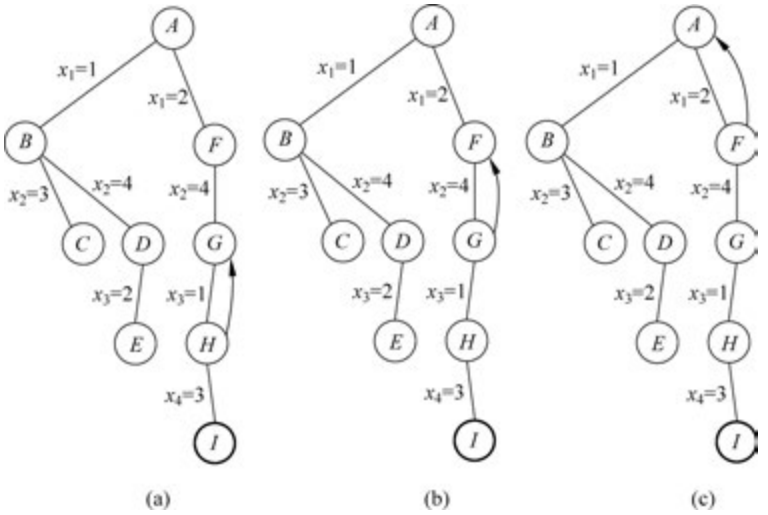

图 5-7　四皇后问题的搜索过程(五)

通常将搜索过程中形成的树状结构称为问题的搜索树。在例 5-1 中的四皇后问题对应的搜索树如图 5-8 所示。简单地讲,搜索树上的节点全部是解空间树中满足隐约束的节点,而不满足隐约束的节点被全部剪掉。

对显约束为不同行且不同列的四皇后问题的解空间树(如图 5-2 所示)进行搜索的过程与上述搜索过程类似。二者最终形成的搜索树完全相同,只有搜索过程中检查的隐约束和分支数不同,留给读者练习。

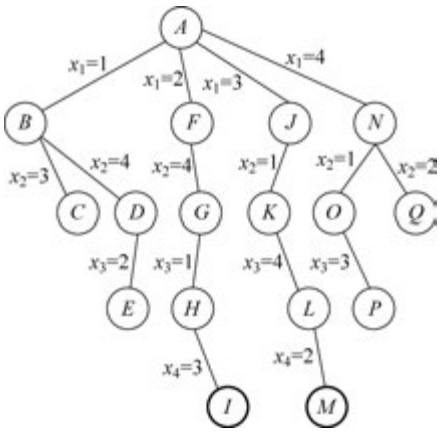

图 5-8　四皇后问题的搜索过程(六)

3. 回溯法的算法描述模式

回溯法是一种带有约束函数或限界函数的深度优先搜索方法,搜索过程是在问题的解空间树中进行的。算法描述通常采用递归技术,也可以选用非递归技术。

(1) 递归算法描述模式。递归算法描述如下:

```
def Backtrack(t):
    if (t > n):
        output(x)
    else:
        for i in range(s(n,t),e(n,t) + 1):
            x[t] = d(i)
            if (constraint(t) and bound(t)):
                Backtrack(t + 1)
```

这里,形参 t 代表当前扩展节点在解空间树中所处的层次。解空间树的根节点为第 1 层,根节点的子节点为第 2 层;以此类推,深度为 n 的解空间树的叶子节点为第 $n+1$ 层。注意:在解空间树中,节点所处的层次比该节点所在的深度大 1。解空间树中节点的深度与

层次之间的关系如图 5-9 所示。

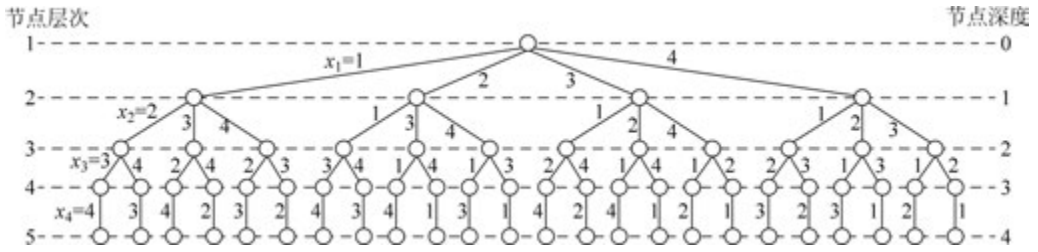

图 5-9 解空间树中节点的深度与层次关系

变量 n 代表问题的规模,同时也是解空间树的深度。注意区分树的深度和树中节点的深度两个概念,树的深度指的是树中深度最大的节点深度。$s(n,t)$代表当前扩展节点处未搜索的子树的起始编号。$e(n,t)$代表当前扩展节点处未搜索的子树的终止编号。$d(i)$代表当前扩展节点处可选的分支上的数据。x 是用来记录问题当前解的数组。constraint(t)代表当前扩展节点处的约束函数。bound(t)代表当前扩展节点处的限界函数。满足约束函数或限界函数则继续深一层次的搜索;否则,剪掉相应的子树。Backtrack(t)代表从第 t 层开始搜索问题的解。由于搜索是从解空间树的根节点开始,即从第 1 层开始搜索,因此函数调用为 Backtrack(1)。

(2) 非递归算法描述模式。非递归算法描述如下:

```
def NBacktrack():
    t = 1
    while (t > 0):
        if (s(n,t)<= e(n,t)):          # 从起始分支 s(n,t)到结束分支 e(n,t)
            for i in range(s(n,t),e(n,t) + 1):
                x[t] = d(i)
                if (constraint(t) and bound(t)):
                    if (t > n):
                        output(x)
                    else:
                        t += 1         # 更深一层
        else:
            t -= 1                     # 回溯到上一层的活节点
```

这里出现的函数和变量均和递归算法描述模式中出现的含义相同。

5.2 典型的解空间结构

5.2.1 子集树

1. 概述

子集树是使用回溯法解题时经常遇到的一种典型的解空间树。当所给的问题是从 n 个元素组成的集合 S 中找出满足某种性质的一个子集时,相应的解空间树称为子集树。此

类问题解的形式为 n 元组$(x_1,x_2,\cdots,x_n)$，分量 $x_i(i=1,2,\cdots,n)$表示第 i 个元素是否在要找的子集中。x_i 的取值为 0 或 1，$x_i=0$ 表示第 i 个元素不在要找的子集中；$x_i=1$ 表示第 i 个元素在要找的子集中。如图 5-10 所示是 $n=3$ 时的子集树。

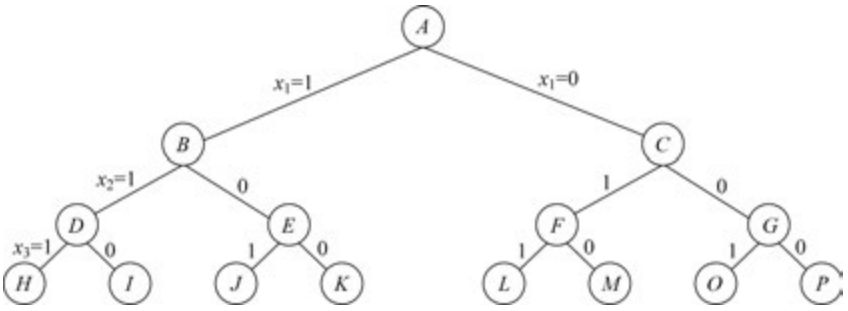

图 5-10　$n=3$ 时的子集树

子集树中所有非叶子节点均有左右两个分支，左分支为 1，右分支为 0，反之也可以。本书约定子集树的左分支为 1，右分支为 0。树中从根到叶子的路径描述了一个 n 元 0-1 向量，这个 n 元 0-1 向量表示集合 S 的一个子集，这个子集由对应分量为 1 的元素组成。如假定 3 个元素组成的集合 S 为{1,2,3}，从根节点 A 到叶节点 I 的路径描述的 n 元组为(1,1,0)，它表示 S 的一个子集{1,2}。从根节点 A 到叶节点 M 的路径描述的 n 元组为(0,1,0)，它表示 S 的另一个子集{2}。

在子集树中，树的根节点表示初始状态，中间节点表示某种情况下的中间状态，叶子节点表示结束状态。分支表示从一个状态过渡到另一个状态的行为。从根节点到叶子节点的路径表示一个可能的解。子集树的深度等于问题的规模。

解空间树为子集树的问题有很多，如下所述。

(1) 0-1 背包问题。从 n 个物品组成的集合 S 中找出一个子集，这个子集内所有物品的总重量不超过背包的容量，并且这些物品的总价值在 S 的所有不超过背包容量的子集中是最大的。显然，这个问题的解空间树是一棵子集树。

(2) 子集和问题。给定 n 个整数和一个整数 C，要求找出 n 个数中哪些数相加的和等于 C。这个问题实质上是要求从 n 个数组成的集合 S 中找出一个子集，这个子集中所有数的和等于给定的 C。因此，子集和问题的解空间树也是一棵子集树。

(3) 装载问题。n 个集装箱要装上两艘载重量分别为 c_1 和 c_2 的轮船，其中集装箱 i 的重量为 w_i，且 $\sum_{i=1}^{n} w_i \leqslant c_1+c_2$。装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这两艘轮船，如果有，找出一种装载方案。

这个问题如果有解，就采用下面的策略，可得到最优装载方案。

① 首先将第一艘轮船尽可能装满。

② 将剩余的集装箱装上第二艘轮船。如果剩余的集装箱能够全部装上船，则找到一个合理的方案，如果不能全部装上船，则不存在装载方案。

将第一艘轮船尽可能装满等价于从 n 个集装箱组成的集合中选取一个子集，该子集中集装箱重量之和小于或等于第一艘船的载重量且最接近第一艘船的载重量。由此可知，装载问题的解空间树也是一棵子集树。

(4) 最大团问题。给定一个无向图,找出它的最大团。这个问题等价于从给定无向图的 n 个顶点组成的集合中找出一个顶点子集,这个子集中的任意两个顶点之间有边相连且包含的顶点个数是所有该类子集中包含顶点个数最多的。因此,这个问题也是从整体中取出一部分,这一部分构成整体的一个子集且满足一定的特性,它的解空间树是一棵子集树。

可见,对于要求从整体中取出一部分,这一部分需要满足一定的特性,整体与部分之间构成包含与被包含的关系,即子集关系的一类问题,均可采用子集树来描述它们的解空间树。这类问题在解题时可采用统一的算法设计模式。

2. 子集树的算法描述模式

子集树的算法描述如下:

```
def Backtrack(t):
    if (t>n):
        output(x)
    if(constraint(t)):                    #左子树
        #做相关标识
        Backtrack(t+1)
        #做相关标识的反操作
    if(bound(t)):                         #右子树
        #做相关标识
        Backtrack(t+1)
        #做相关标识的反操作
```

这里,形式参数 t 表示扩展节点在解空间树中所处的层次;n 表示问题的规模,即解空间树的深度;x 是用来存放当前解的一维数组,初始化为 $x[i]=0(i=1,2,\cdots,n)$;constraint()函数为约束函数;bound()函数为限界函数。

5.2.2 排列树

1. 概述

排列树是用回溯法解题时经常遇到的第二种典型的解空间树。当所给的问题是从 n 个元素的排列中找出满足某种性质的一个排列时,相应的解空间树称为排列树。此类问题解的形式为 n 元组$(x_1,x_2,\cdots,x_n)$,分量 $x_i(i=1,2,\cdots,n)$表示第 i 个位置的元素是x_i。n 个元素组成的集合为 $S=\{1,2,\cdots,n\}$,$x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}(i=1,2,\cdots,n)$。

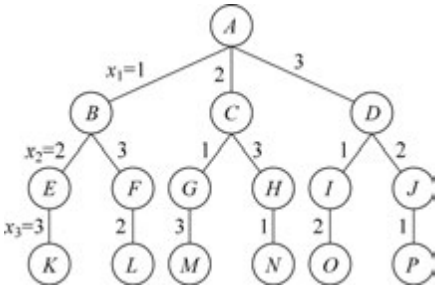

图 5-11 $n=3$ 的排列树

$n=3$ 时的排列树如图 5-11 所示。

在排列树中从根到叶子的路径描述了 n 个元素的一个排列。如 3 个元素的位置为{1,2,3},从根节点 A 到叶节点 L 的路径描述的一个排列为(1,3,2),即第 1 个位置的元素是 1,第 2 个位置的元素是 3,第 3 个位置的元素是 2;从根节点 A 到叶节点 M 的路径描述的一个排列为(2,1,3);从根节点 A 到叶节点 P 的路径描述的一个排列为(3,2,1)。

在排列树中，树的根节点表示初始状态（所有位置全部没有放置元素）；中间节点表示某种情况下的中间状态（中间节点之前的位置上已经确定了元素，中间节点之后的位置上还没有确定元素）；叶子节点表示结束状态（所有位置上的元素全部确定）；分支表示从一个状态过渡到另一个状态的行为（在特定位置上放置元素）；从根节点到叶子节点的路径表示一个可能的解（所有元素的一个排列）。排列树的深度等于问题的规模。

解空间树为排列树的问题有很多种，如下所述。

（1）n 皇后问题。满足显约束为不同行、不同列的解空间树。约定不同行的前提下，n 个皇后的列位置是 n 个列的一个排列，这个排列必须满足 n 个皇后的位置不在一条斜线上。

（2）旅行商问题。找出 n 个城市的一个排列，沿着这个排列的顺序遍历 n 个城市，最后回到出发城市，求长度最短的旅行路径。

（3）批处理作业调度问题。给定 n 个作业的集合$\{J_1, J_2, \cdots, J_n\}$，要求找出 n 个作业的一个排列，按照这个排列进行调度，使得完成时间和达到最小。

（4）圆排列问题。给定 n 个大小不等的圆$c_1, c_2, \cdots, c_n$，现要将这 n 个圆放入一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出具有最小长度的圆排列。

（5）电路板排列问题。将 n 块电路板以最佳排列方式插入带有 n 个插槽的机箱中。n 块电路板的不同排列方式对应于不同的电路板插入方案。设 $B=\{1,2,\cdots,n\}$是 n 块电路板的集合，$L=\{N_1, N_2, \cdots, N_m\}$是连接这 n 块电路板中若干电路板的 m 个连接块。N_i 是 B 的一个子集，且 N_i 中的电路板用同一条导线连接在一起。设 x 表示 n 块电路板的一个排列，即在机箱的第 i 个插槽中插入的电路板编号是 x_i。x 所确定的电路板排列 Density(x)密度定义为跨越相邻电路板插槽的最大连线数。在设计机箱时，插槽一侧的布线间隙由电路板排列的密度确定。因此，电路板排列问题要求对于给定的电路板连接条件，确定电路板的最佳排列，使其具有最小排列密度。

可见，对于要求从 n 个元素中找出它们的一个排列，该排列需要满足一定的特性这类问题，均可采用排列树来描述它们的解空间结构。这类问题在解题时可采用统一的算法设计模式。

2. 排列树的算法描述模式

排列树的算法描述如下：

```
def Backtrack(t):
    if (t > n):
        output(x)
    else:
        for i in range(t, n + 1):
            x[t], x[i]←x[i], x[t]      # x 初始化为 x[i]=i
            if (constraint(t) and bound(t)):
                Backtrack(t + 1)
            x[t], x[i]←x[i], x[t]
```

这里，形式参数 t 表示扩展节点在解空间树中所处的层次；n 表示问题的规模，即解空间

树的深度；x 是用来存储当前解的数组，初始化 $x[i]=i(i=1,2,\cdots,n)$，constraint()函数为约束函数；bound()函数为限界函数。

5.2.3 满 m 叉树

1. 概述

满 m 叉树是用回溯法解题时经常遇到的第三种典型的解空间树，也可以称为组合树。当所给问题的 n 个元素中每个元素均有 m 种选择，要求确定其中的一种选择，使得对这 n 个元素的选择结果组成的向量满足某种性质，即寻找满足某种特性的 n 个元素取值的一种组合。这类问题的解空间树称为满 m 叉树。此类问题解的形式为 n 元组$(x_1,x_2,\cdots,x_n)$，分量 $x_i(i=1,2,\cdots,n)$表示第 i 个元素的选择为 x_i。$n=3$ 时的满 $m=3$ 叉树如图 5-12 所示。

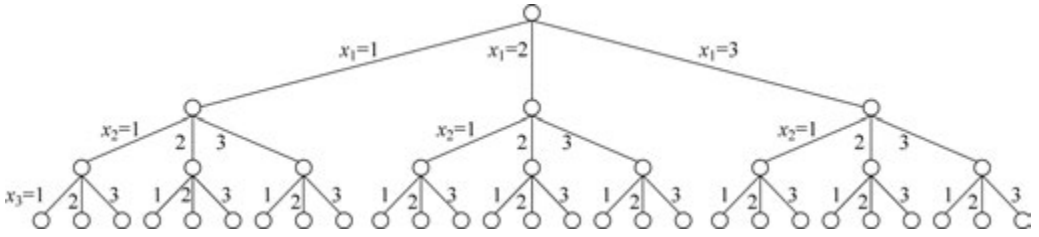

图 5-12 满 3 叉树

在满 m 叉树中从根到叶子的路径描述了 n 个元素的一种选择。树的根节点表示初始状态(任何一个元素都没有确定)，中间节点表示某种情况下的中间状态(一些元素的选择已经确定，另一些元素的选择没有确定)，叶子节点表示结束状态(所有元素的选择均已确定)，分支表示从一个状态过渡到另一个状态的行为(特定元素做何种选择)，从根节点到叶子节点的路径表示一个可能的解(所有元素的一个排列)。满 m 叉树的深度等于问题的规模 n。

解空间树为满 m 叉树的问题有很多种，如下所述。

(1) n 皇后问题。显约束为不同行的解空间树，在不同行的前提下，任何一个皇后的列位置都有 n 种选择。n 个列位置的一个组合必须满足 n 个皇后的位置不在同一列或不在同一条斜线上的性质。这个问题的解空间便是一棵满 $m(m=n)$叉树。

(2) 图的 m 着色问题。给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。如果有一种着色法使 G 中有边相连的两个顶点着不同颜色，则称这个图是 m 可着色的。图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色法。这个问题实质上是用给定的 m 种颜色给无向连通图 G 的顶点着色。每个顶点所着的颜色有 m 种选择，找出 n 个顶点着色的一个组合，使其满足有边相连的两个顶点之间所着颜色不相同。很明显，这是一棵满 m 叉树。

(3) 最小质量机器设计问题：可以看作给机器的 n 个部件找供应商，也可以看作 m 个供应商供应机器的哪个部件。如果看作给机器的 n 个部件找供应商，那么问题的实质为 n 个部件中的每个部件均有 m 种选择，找出 n 个部件供应商的一个组合，使其满足 n 个部件的总价格不超过 c 且总重量是最小的。问题的解空间是一棵满 m 叉树。如果看作 m 个供应商供应机器的哪个部件，那么问题的解空间是一棵排列树，读者可以自己思考一下原因。

可见，对于要求找出 n 个元素的一个组合，该组合需要满足一定特性这类问题，均可采

用满 m 叉树来描述它们的解空间结构。这类问题在解题时可采用统一的算法设计模式。

2. 满 m 叉树的算法描述模式

满 m 叉树的算法描述如下：

```
def Backtrack(t):
    if (t > n):
        output(x)
    else:
        for i in range(1,m + 1):
            if (constraint(t) and bound(t)):
                x[t] = i
                #做其他相关标识
                Backtrack(t + 1)
                #做其他相关标识的反操作
```

这里，形式参数 t 表示扩展节点在解空间树中所处的层次；n 表示问题的规模，即解空间树的深度；m 表示每个元素可选择的种数；x 是用来存放解的一维数组，初始化为 $x[i]=0(i=1,2,\cdots,n)$；constraint()函数为约束函数；bound()函数为限界函数。

5.3　0-1 背包问题——子集树

给定 n 种物品和一个背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 W。一种物品要么全部装入背包，要么全部不装入背包，不允许部分装入。装入背包的物品的总重量不超过背包的容量。问应如何选择装入背包的物品，使得装入背包中的物品总价值最大？

5.3.1　问题分析——解空间及搜索条件

根据问题描述可知，0-1 背包问题要求找出 n 种物品集合$\{1,2,\cdots,n\}$中的一部分物品，将这部分物品装入背包。装进去的物品总重量不超过背包的容量且价值之和最大，即找到 n 种物品集合$\{1,2,\cdots,n\}$的一个子集，这个子集中的物品总重量不超过背包的容量，且总价值是集合$\{1,2,\cdots,n\}$的所有不超过背包容量的子集中物品总价值最大的。

按照回溯法的算法框架，首先需要定义问题的解空间，然后确定解空间的组织结构，最后进行搜索。搜索前要解决两个关键问题，一是确定问题是否需要约束条件(用于判断是否有可能产生可行解)，如果需要，应如何设置？二是确定问题是否需要限界条件(用于判断是否有可能产生最优解)，如果需要，应如何设置？

1. 定义问题的解空间

0-1 背包问题是要将物品装入背包，并且物品有且只有两种状态。第 $i(i=1,2,\cdots,n)$ 种物品是装入背包能够达到目标要求，还是不装入背包能够达到目标要求呢？很显然，目前还不确定。因此，可以用变量 x_i 表示第 i 种物品是否被装入背包的行为，如果用“0”表示不被装入背包，用“1”表示装入背包，则 x_i 的取值为 0 或 1。该问题解的形式是一个 n 元组，且每个分量的取值为 0 或 1。由此可得，问题的解空间为$(x_1,x_2,\cdots,x_n)$，其中 $x_i=0$ 或

$1(i=1,2,\cdots,n)$。

2. 确定解空间的组织结构

问题的解空间描述了 2^n 种可能的解,也可以说是 n 个元素组成的集合的所有子集个数。可见,问题的解空间树为子集树。采用一棵满二叉树将解空间有效地组织起来,解空间树的深度为问题的规模 n。图 5-13 所示为 $n=4$ 时的解空间树。

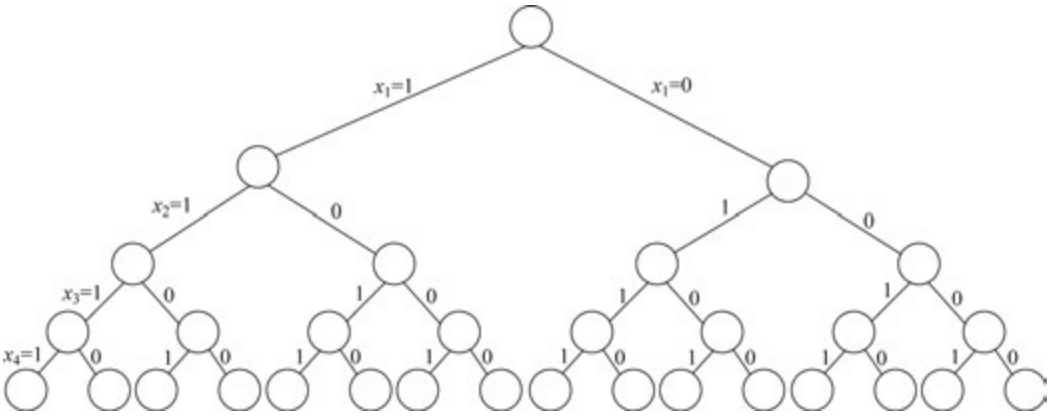

图 5-13 $n=4$ 时的解空间树

3. 搜索解空间

(1) 是否需要约束条件? 如果需要,应如何设置?

0-1 背包问题的解空间包含 2^n 个可能的解,是不是每个可能的解描述的装入背包的物品的总重量都不超过背包的容量呢? 显然不是,这个问题存在某种或某些物品无法装入背包的情况。因此,需要设置约束条件来判断所有可能的解描述的装入背包的物品总重量是否超出背包的容量,如果超出,就为不可行解,否则为可行解。搜索过程将不再搜索那些导致不可行解的节点及其节点。约束条件的形式化描述为

$$\sum_{i=1}^{n} w_i x_i \leqslant W \tag{5-1}$$

(2) 是否需要限界条件? 如果需要,应如何设置?

0-1 背包问题的可行解可能不止一个,问题的目标是找一个所描述的装入背包的物品总价值最大的可行解,即最优解。因此,需要设置限界条件来加速找出该最优解的速度。

如何设置限界条件呢? 根据解空间的组织结构可知,任何一个中间节点 z(中间状态)均表示从根节点到该中间节点的分支所代表的行为已经确定,从 z 到其子孙节点的分支的行为是不确定的。也就是说,如果 z 在解空间树中所处的层次是 t,从第 1 种物品到第 $t-1$ 种物品的状态已经确定,接下来要确定第 t 种物品的状态。无论沿着 z 的哪个分支进行扩展,第 t 种物品的状态就确定了。那么,从第 $t+1$ 种物品到第 n 种物品的状态还不确定。这样,可以根据前 t 种物品的状态确定当前已装入背包的物品的总价值,用 cp 表示。第 $t+1$ 种物品到第 n 种物品的总价值用 rp 表示,则 cp+rp 是所有从根出发的路径中经过中间节点 z 的可行解的价值上界。如果价值上界小于或等于当前搜索到的最优解描述的装入背

包的物品总价值(用 bestp 表示,初始值为 0),就说明从中间节点 z 继续向子孙节点搜索不可能得到一个比当前更优的可行解,没有继续搜索的必要;反之,则继续向 z 的子孙节点搜索。因此,限界条件可描述为

$$cp + rp > bestp \tag{5-2}$$

5.3.2　算法设计

从根节点开始,以深度优先的方式进行搜索。根节点首先成为活节点,也是当前的扩展节点。由于子集树中约定左分支上的值为"1",因此沿着扩展节点的左分支扩展,则代表装入物品,此时,需要判断是否能够装入该物品,即判断约束条件成立与否,如果成立,就进入左子节点,左子节点成为活节点,并且是当前的扩展节点,继续向纵深节点扩展;如果不成立,就剪掉扩展节点的左分支,沿着其右分支扩展。右分支代表物品不装入背包,肯定有可能导致可行解。但是沿着右分支扩展有没有可能得到最优解呢?这一点需要由限界条件来判断。如果满足限界条件,说明有可能导致最优解,即进入右分支,右子节点成为活节点,并成为当前的扩展节点,继续向纵深节点扩展;如果不满足限界条件,则剪掉扩展节点的右分支,开始向最近的活节点回溯。搜索过程直到所有活节点变成死节点后结束。

算法伪码描述如下:

```
算法:backtrack(t)
输入:当前扩展节点的层次,根节点为 0
    if t≥n then                          //搜索到叶子节点
        if bestp < cp then
            bestp ← cp                   //记录当前最优值
            bestx ← x[:]                 //记录当前最优解
    else
        if cw + w[t] ≤ W then            //判断左分支是否满足约束条件
            x[t]←1
            cw←cw + w[t]
            cp←cp + v[t]
            rp←rp - v[t]
            backtrack(t + 1)
            cw←cw - w[t]
            cp←cp - v[t]
            rp←rp + v[t]
        if cp + rp > bestp then          //判断右分支是否满足限界条件
            x[t]←0
            backtrack(t + 1)
```

5.3.3　实例构造

令 $n=4$,$W=7$,$w=(3,5,2,1)$,$v=(9,10,7,4)$。搜索过程如图 5-14～图 5-18 所示(图中节点旁括号内的数据表示背包的剩余容量和已装入背包的物品价值)。

首先,搜索从根节点开始,即根节点是活节点,也是当前的扩展节点。它代表初始状态,即背包是空的,如图 5-14(a)所示。扩展节点 1 先沿着左分支扩展,此时需要判断式(5-1)约束条件,第一种物品的重量为 3,$3<7$,满足约束条件,因此节点 2 成为活节点,并成为当前

的扩展节点。它代表第1种物品已装入背包,背包剩余容量为4,背包内物品的总价值为9,如图5-14(b)所示。扩展节点2继续沿着左分支扩展,此时需要判断第2个物品能否装入背包,第2个物品的重量为5,背包的剩余容量为4。显然,该物品无法装入,故剪掉扩展节点2的左分支。此时,需要选择扩展节点2的右分支继续扩展,判断式(5-2)限界条件,cp=9,rp=11,bestp=0,cp+rp>bestp限界条件成立,则节点2沿右分支扩展的节点3成为活节点,并成为当前的扩展节点。扩展节点3代表背包剩余容量为4,背包内物品的总价值为9,如图5-14(c)所示。以此类推,扩展节点3沿着左分支扩展,第3种物品的重量是2,背包的剩余容量为4,满足约束条件,节点4成为活节点,并成为当前的扩展节点。节点4代表背包剩余容量为2,背包内物品总价值为16,如图5-14(d)所示。

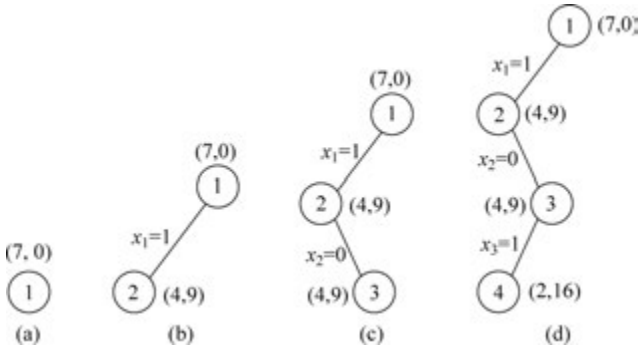

图5-14 0-1背包问题的搜索过程(一)

扩展节点4沿着左分支扩展,此时第4种物品的重量为1,背包的剩余容量为2,节点5满足约束条件。节点5已是叶子节点,故找到一个当前最优解,将其记录并修改bestp的值为当前最优解描述的装入背包的物品总价值20,如图5-15(a)所示。由于节点5已是叶子节点,不具备扩展能力,此时要回溯到离节点5最近的活节点4,节点4再次成为扩展节点,如图5-15(b)所示。扩展节点4沿着右分支继续扩展,此时要判断限界条件是否满足,cp=16,rp=0,bestp=20,cp+rp<bestp,限界条件不满足,故剪掉节点4的右分支。扩展节点4的左右两个分支均搜索完毕,回溯到最近的活节点3,节点3再次成为扩展节点,如图5-15(c)所示。扩展节点3沿着右分支继续扩展,此时要判断限界条件是否满足,cp=9,rp=4,bestp=20,cp+rp<bestp,限界条件不满足,故剪掉节点3的右分支。扩展节点3的左右两个分支均搜索完毕,回溯到最近的活节点2,节点2再次成为扩展节点。扩展节点2的两个分支均搜索完毕,故继续回溯到节点1,如图5-15(d)所示。

扩展节点1沿着右分支继续扩展,判断限界条件是否满足,cp=0,rp=21,bestp=20,cp+rp>bestp,限界条件满足,则扩展的节点6成为活节点,并成为当前的扩展节点,如图5-16(a)所示。扩展节点6沿着左分支继续扩展,判断约束条件,当前背包剩余容量为7,第2种物品的重量为5,5<7,满足约束条件,扩展生成的节点7成为活节点,并且是当前的扩展节点。此时背包的剩余容量为2,装进背包的物品总价值为10,如图5-16(b)所示。扩展节点7沿着左分支继续扩展,判断约束条件,当前背包剩余容量为2,第3种物品的重量为2,满足约束条件,扩展生成的节点8成为活节点,并且是当前的扩展节点。此时背包的剩余容量为0,装入背包的物品总价值为17,如图5-16(c)所示。

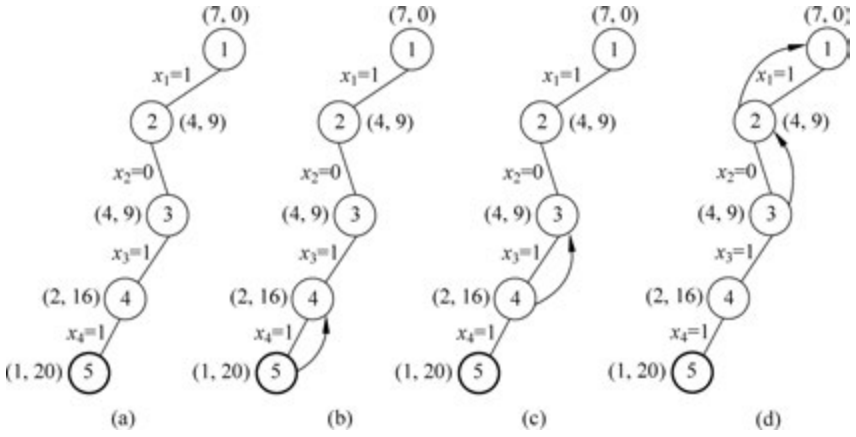

图 5-15　0-1 背包问题的搜索过程(二)

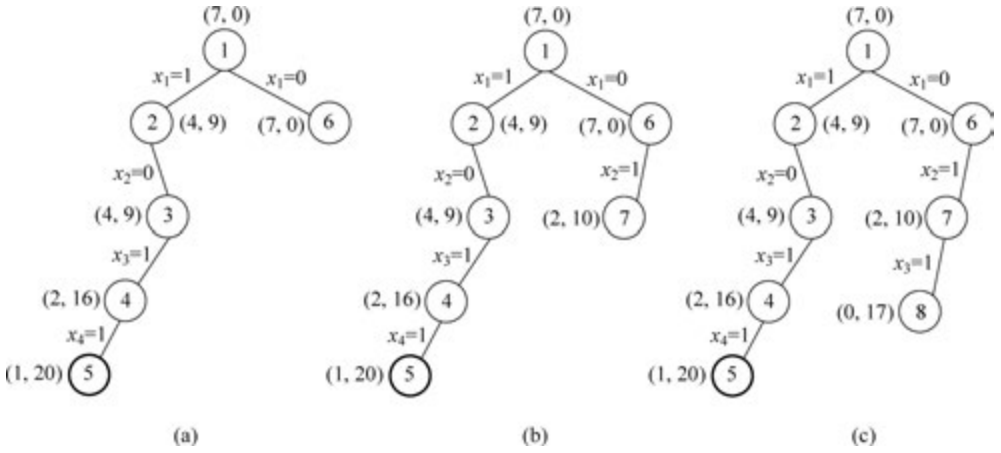

图 5-16　0-1 背包问题的搜索过程(三)

扩展节点 8 沿着左分支继续扩展,判断约束条件,当前背包剩余容量为 0,第 4 种物品的重量为 1,0＜1,不满足约束条件,扩展生成的节点被剪掉。接下来沿着扩展节点 8 的右分支进行扩展,判断限界条件,cp＝17,rp＝0,bestp＝20,cp＋rp＜bestp,不满足限界条件,沿右分支扩展生成的节点也被剪掉。扩展节点 8 的所有分支均搜索完毕,回溯到最近的活节点 7,节点 7 又成为扩展节点,如图 5-17(a)所示。扩展节点 7 沿着右分支继续扩展,判断限界条件,当前 cp＝10,rp＝4,bestp＝20,cp＋rp＜bestp,限界条件不满足,扩展生成的节点被剪掉。扩展节点 7 的所有分支均搜索完毕,回溯到活节点 6,节点 6 又成为扩展节点,如图 5-17(b)所示。扩展节点 6 沿着右分支继续扩展,判断限界条件,当前 cp＝0,rp＝11,bestp＝20,cp＋rp＜bestp,限界条件不满足,扩展生成的节点被剪掉。扩展节点 6 的所有分支均搜索完毕,回溯到活节点 1,节点 1 又成为扩展节点,如图 5-17(c)所示。

扩展节点 1 的两个分支均搜索完毕,它成为死节点,搜索过程结束,找到的问题的解为从根节点 1 到叶子节点 5 的路径(1,0,1,1),即将第 1,3,4 三种物品装入背包,装进去物品总价值为 20,如图 5-18 所示。

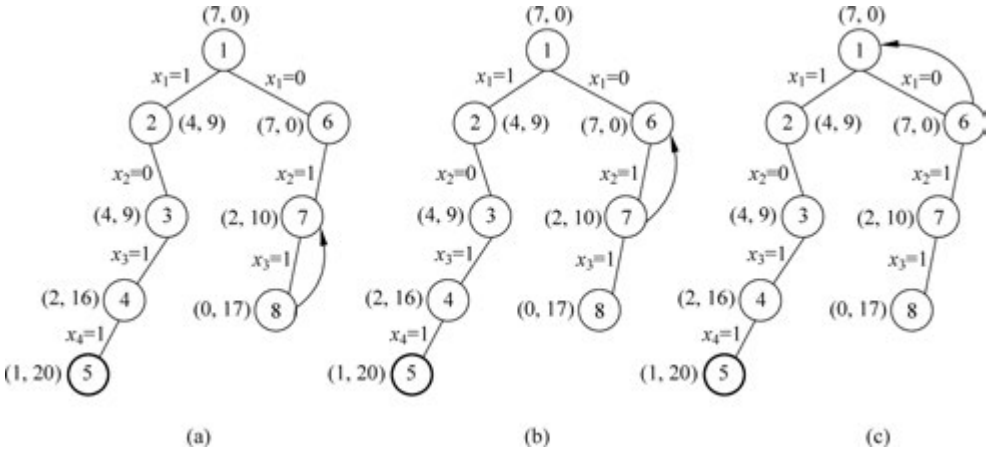

图 5-17　0-1 背包问题的搜索过程(四)

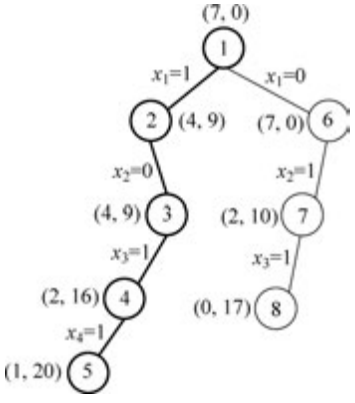

图 5-18　0-1 背包问题的搜索过程(五)

5.3.4　算法的改进

在上述限界条件中,rp 表示第 $t+1$ 种物品到第 n 种物品的总价值。事实上,背包的剩余容量不一定能够容纳从第 $t+1$ 种物品到第 n 种物品的全部物品,那么剩余容量所能容纳的从第 $t+1$ 种物品到第 n 种物品的最优值(用 brp 表示)肯定小于或等于 rp,用 brp 取代 rp,则式(5-2)改写为

$$\text{cp}+\text{brp}>\text{bestp} \tag{5-3}$$

0-1 背包问题最终不一定能够将背包装满,因此,cp+brp 同样是所有路径经过中间节点 z 的可行解的价值上界,且这个价值上界小于或等于 cp+rp。因此,表达式 cp+brp>bestp 成立的可能性比 cp+rp>bestp 成立的可能性小。用 cp+brp>bestp 作为限界条件,从中间节点 z 沿右分支继续向纵深搜索的可能性就小。也就是说,中间节点 z 的右分支剪枝的可能性就越大,搜索速度也会加快。

以式(5-3)作为限界条件的搜索过程与以式(5-2)作为限界条件的搜索过程只有在搜索右分支时进行的判断不同。在以式(5-3)作为限界条件的搜索过程中,需要求出 brp 的值,

为方便起见，事先计算出所给物品单位重量的价值$\left(\frac{9}{3},\frac{10}{5},\frac{7}{2},\frac{4}{1}\right)$。针对剩余的物品，单位重量价值大的物品优先装入背包，将背包剩余容量装满所得的价值即为 brp 的值。在图 5-14(b)中，扩展节点 2 沿右分支扩展，判断限界条件，当前 cp＝9，剩余的不确定状态的物品为第 3、4 种物品，背包剩余容量为 4，将背包装满装入的最优值为第 3、4 种物品的价值之和，即 brp＝11，bestp＝0，cp＋brp＞bestp，限界条件成立，扩展的节点 3 成为活节点，并成为当前的扩展节点，继续向纵深处扩展。式(5-3)限界条件的搜索与式(5-2)限界条件的搜索直到图 5-15(b)(找到一个当前最优解后回溯到最近的活节点 4)均相同，其后在式(5-3)限界条件下的搜索过程如图 5-19 所示。

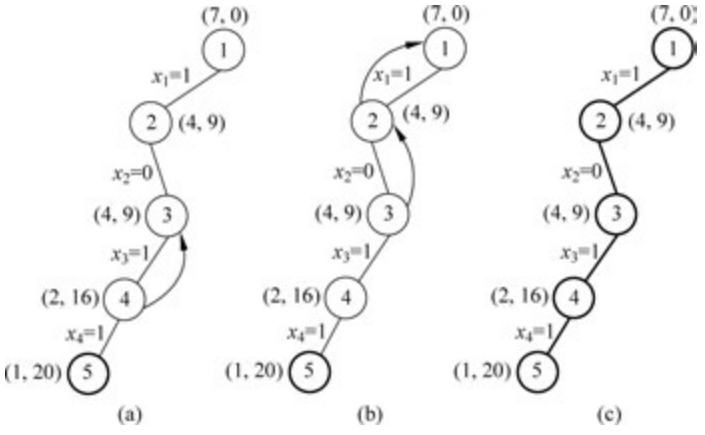

图 5-19　算法改进后的 0-1 背包问题的搜索过程

扩展节点 4 沿右分支扩展，判断限界条件，cp＝16，背包的剩余容量为 2，没有剩余物品，故 brp＝0，bestp＝20，cp＋brp＜bestp，限界条件不满足，扩展生成的节点被剪掉。此时，左右分支均检查完毕，开始回溯到活节点 3，节点 3 又成为扩展节点，如图 5-19(a)所示。扩展节点 3 沿右分支扩展，判断限界条件，cp＝9，剩余容量为 4，剩余物品为第 4 种物品，其重量为 1，能够全部装入，故 brp＝4。bestp＝20，cp＋brp＜bestp，限界条件不满足，扩展生成的节点被剪掉。此时，节点 3 的左右分支均搜索完毕，回溯到活节点 2。节点 2 的两个分支已搜索完毕，继续回溯到活节点 1，活节点 1 再次成为扩展节点，如图 5-19(b)所示。扩展节点 1 继续沿右分支扩展，判断限界条件，cp＝0，剩余容量为 7，剩余物品为第 2、3、4 种物品，按照单位重量的价值大的物品优先的原则，将第 3、4 种物品全部装入背包。此时，背包剩余容量为 4，第 2 种物品的重量为 5，无法全部装入，只需装入第 2 种物品的$\frac{4}{5}$，那么装进去的价值为 $10\times\frac{4}{5}=8$，故 brp＝7＋4＋8＝19，cp＋brp＝19＜besp(20)，不满足限界条件，扩展生成的节点被剪掉。此时，左右分支均搜索完毕，搜索过程结束，找到的当前最优解为(1，0，1，1)，最优值为 20，如图 5-19(c)所示。

5.3.5　算法分析

判断约束函数需要 $O(1)$时间，在最坏情况下有 2^n-1 个左孩子，约束函数耗时最坏为

$O(2^n)$。计算上界限界函数需要$O(n)$时间,在最坏情况下有2^n-1个右孩子需要计算上界,限界函数耗时最坏为$O(n2^n)$。0-1背包问题的回溯算法所需的计算时间为$O(2^n)+O(n2^n)=O(n2^n)$。

5.3.6 Python实践

1. 改进前的算法编码实现

首先定义一个backtrack()递归函数,用于深度优先搜索问题的解,接收当前扩展节点在子集树中所处的层次t,根节点所处的层次为0(考虑到数组下标从0开始,这里将根节点的层次记为0)。搜索过程中记录最优解bestp和最优值bestx。

改进前的算法代码如下:

```
def backtrack(t):
    global bestp,cw,cp,x,bestx,rp
    if t>=n:
        if bestp<cp:
            bestp = cp
            bestx = x[:]
    else:
        if cw + w[t]<= W:
            x[t] = 1
            cw += w[t]
            cp += v[t]
            rp -= v[t]
            backtrack(t+1)
            cw -= w[t]
            cp -= v[t]
            rp += v[t]
        if cp + rp>bestp:
            x[t]=0
            backtrack(t+1)
```

Python程序的入口是main()函数,在main()函数中,提供物品的重量w、物品的价值v和背包的容量W,并做初始化工作,调用backtrack()函数,最后将结果输出到显示器上。其代码如下:

```
if __name__ == "__main__":
    bestp=0                          #当前最优值
    cw=0                             #当前装入背包的重量
    cp=0                             #当前装入背包的价值
    bestx=None                       #记录当前最优解
    n=5                              #物品个数
    W=10                             #背包容量
    w=[2,2,6,5,4]                    #物品重量
    v=[6,3,5,4,6]                    #物品价值
    x=[0 for i in range(n)]          #当前可行解
    rp = 0                           #剩余物品的价值
    for i in range(len(v)):
        rp += v[i]
    backtrack(0)                     #从根节点开始搜索
```

```
    print("最优值为:",bestp)
    print("最优解为:",bestx)
```

输出结果为

```
最优值为: 15
最优解为: [1, 1, 0, 0, 1]
```

2. 改进后的算法编码实现

由于要将物品按照单位重量的价值非升序排列，排序后，数组下标不能表示物品编号了，所以采用三元组（物品编号、物品重量和物品价值）数组表示物品，Python 中用 list 列表 goods 存储所有物品的三元组。

首先定义一个计算价值上界的 bound()函数，接收当前剩余的起始物品位置，返回当前节点的价值上界。其代码如下：

```
def bound(i):
    global c,cw,goods,cp,n
    Wleft = c - cw
    b = cp
    while(i <= n and goods[i][1]<= Wleft):
        Wleft -= goods[i][1]
        b += goods[i][2]
        i += 1
    if(i <= n):
        b += goods[i][2]/goods[i][1] * Wleft;
    return b
```

定义一个 backtrack()递归函数，用于深度优先搜索问题的解，接收当前扩展节点的在子集树中所处的层次 t，根节点所处的层次为 0（考虑到数组下标从 0 开始，这里将根节点的层次记为 0）。搜索过程中记录最优值 bestp 和最优解 bestx。其代码如下：

```
def backtrack(t):
    global bestp,c,cw,cp,x,bestx,n,goods
    if t >= n:
        if bestp < cp:
            bestp = cp
            bestx = x[:]
    else:
        if cw + goods[t][1] <= c:      #左分支判断约束条件
            x[goods[t][0]] = 1
            cw += goods[t][1]
            cp += goods[t][2]
            backtrack(t + 1)
            cw -= goods[t][1]
            cp -= goods[t][2]
        if bound(t) > bestp:           #计算价值上界,并判断限界条件
            x[goods[t][0]] = 0
            backtrack(t + 1)
```

Python程序的入口是main()函数,在main()函数中,goods用于存储物品的三元组(物品编号、物品重量和物品价值),W为背包的容量,初始化求解中用到的其他变量,调用backtrack()函数,最后将结果输出到显示器上。其代码如下:

```
if __name__ == "__main__":
    bestp = 0                                                    #当前最优值
    cw = 0                                                       #当前重量
    cp = 0                                                       #当前价值
    bestx = None                                                 #最优解
    n = 5                                                        #问题规模
    c = 10                                                       #背包容量
    goods = [(0,2,6),(1,2,3),(2,6,5),(3,5,4),(4,4,6)]            #记录物品的编号、重量、价值
    x = [0 for i in range(n)]                                    #当前解
    goods.sort(key = lambda x:x[2]/x[1],reverse = True)          #按照物品的单位重量的价值由
                                                                 #大到小排序
    backtrack(0)
    print("最优值为:",bestp)
    print("最优解为:",bestx)
```

输出结果为

```
最优值为: 15
最优解为: [1, 1, 0, 0, 1]
```

5.4 最大团问题——子集树

给定无向图$G=(V,E)$。如果$U\subseteq V$,且对任意$u,v\in U$,有$(u,v)\in E$,则称U是G的完全子图。G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中。G的最大团是指G中所含顶点数最多的团。最大团问题就是要求找出无向图G的包含顶点个数最多的团。

5.4.1 问题分析——解空间及搜索条件

根据问题描述可知,最大团问题就是要求找出无向图$G=(V,E)$的n个顶点集合$\{1,2,\cdots,n\}$的一部分顶点V',即n个顶点集合$\{1,2,\cdots,n\}$的一个子集,这个子集中的任意两个顶点在无向图G中都有边相连,且包含顶点个数是n个顶点集合$\{1,2,\cdots,n\}$所有同类子集中最多的。显然,问题的解空间是一棵子集树,解决方法与解决0-1背包问题类似。

1. 定义问题的解空间

问题解的形式为n元组,每个分量的取值为0或1,即问题的解是一个n元0-1向量。具体形式为$(x_1,x_2,\cdots,x_n)$,其中$x_i=0$或$1(i=1,2,\cdots,n)$。$x_i=1$表示图G中第i个顶点在团里,$x_i=0$表示图G中第i个顶点不在团里。

2. 确定解空间的组织结构

解空间是一棵子集树,树的深度为n。

3. 搜索解空间

(1) 确定是否需要约束条件？如果需要，应如何设置？

最大团问题的解空间包含 2^n 个子集，这些子集中存在集合中的某两个顶点没有边相连的情况。显然，这种情况下的可能解不是问题的可行解，故需要设置约束条件来判断是否有可能求得问题的可行解。

假设当前扩展节点处于解空间树的第 t 层，那么从第 1 个顶点到第 $t-1$ 个顶点的状态(有没有在团里)已经确定。接下来沿着扩展节点的左分支进行扩展，此时需要判断是否将第 t 个顶点放入团里。只要第 t 个顶点与第 $t-1$ 个顶点中的在团里的顶点有边相连，就能放入团中；否则，就不能放入团中。因此，约束函数描述如下：

```
def place(t):
    global x
    global a
    OK = True
    for j in range(t):
        if x[j] and a[t][j] == 0:
            OK = False
            break
    return OK
```

其中，形式参数 t 表示第 t 个顶点；place(t)用来判断第 t 个顶点能否放入团里；二维数组 a 是图 G 的邻接矩阵；一维数组 x 记录当前解。搜索到第 t 层时，从第 1 个顶点到第 $t-1$ 个顶点的状态存放在 $x[1:t-1]$中。

(2) 确定是否需要限界条件？如果需要，应如何设置？

最大团问题的可行解可能不止一个，问题的目标是找一个包含的顶点个数最多的可行解，即最优解。因此，需要设置限界条件来加速寻找该最优解的速度。

如何设置限界条件呢？与 0-1 背包问题类似。假设当前的扩展节点为 z，如果 z 处于第 t 层，从第 1 个顶点到第 $t-1$ 个顶点的状态已经确定，接下来要确定第 t 个顶点的状态，无论沿着 z 的哪个分支进行扩展，第 t 个顶点的状态就确定了。那么，从第 $t+1$ 个顶点到第 n 个顶点的状态还不确定。这样，可以根据前 t 个顶点的状态确定当前已放入团内的顶点个数(用 cn 表示)，假想从第 $t+1$ 个顶点到第 n 个顶点全部放入团内，放入的顶点个数(用 fn 表示)fn$=n-t$，则 cn+fn 是所有从根出发的路径中经过中间节点 z 的可行解所包含顶点个数的上界。如果 cn+fn 小于或等于当前最优解包含的顶点个数(用 bestn 表示，初始值为 0)，就说明从中间节点 z 继续向子孙节点搜索不可能得到一个比当前更优的可行解，没有继续搜索的必要；反之，则继续向 z 的子孙节点搜索。因此，限界条件可描述为 cn+fn$>$bestn。

5.4.2　算法设计

最大团问题的搜索和 0-1 背包问题的搜索相似，只是进行判断的约束条件和限界条件不同而已。

算法伪码描述如下：

```
算法:backtrack(t)
输入:根节点的层次 t
输出:全局变量 bestx,bestn 记录最优解和最优值
    if (t > n) then
        bestx←x[:]                    //记录当前最优解
        bestn←cn                      //记录当前最优值
        return
    if place(t - 1) then              //判断 t—1 号点是否能放到当前表示团的点集,即约
                                      //束条件
        x[t - 1]←1
        cn ← cn + 1
        backtrack(t + 1)
        cn ← cn - 1
    if (cn + n - t > bestn) then      //判断当前节点有没有可能导出最优解,即限界条件
        x[t - 1]←0
        Backtrack(t + 1)
```

5.4.3 实例构造

以图 5-20 所示的给定的无向图为例,最大团问题的搜索过程如图 5-21～图 5-27 所示。

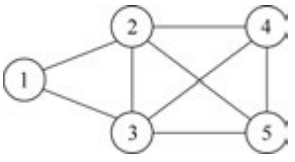

图 5-20 无向图

首先,搜索从根节点开始,即根节点是活节点,也是当前的扩展节点。它代表初始状态,即最大团集合当前是空集,如图 5-21(a)所示。扩展节点 A 先沿着左分支扩展,此时需要判断约束条件,即 1 号点能不能放入最大团集合。由于当前集合为空集,满足约束条件,因此节点 B 成为活节点,并成为当前的扩展节点,它代表 1 号顶点已经加入最大团集合,如图 5-21(b)所示。扩展节点 B 继续沿着左分支扩展,此时需要判断 2 号点能否加入最大团集合,2 号点和最大团集合中的 1 号点有边相连,满足约束条件,因此节点 C 成为活节点,并成为当前的扩展节点,代表将 2 号点加入最大团集合,如图 5-21(c)所示。扩展节点 C 沿着左分支扩展,3 号点与最大团集合中的 1、2 号点都有边相连,满足约束条件,因此节点 D 成为活节点,并成为当前的扩展节点,节点 D 代表 3 号点加入最大团集合,如图 5-21(d)所示。

扩展节点 D 沿着左分支扩展,4 号点与最大团集合中的 1 号点没有边相连,不满足约束条件,D 的左孩子被剪掉。沿着 D 右分支扩展,判断是否满足限界条件,当前最大团集合已经 3 个点,D 的右分支代表 4 号点不加入最大团集合,剩余没判断是否加入最大团集合的只有 5 号点,cn(3)+rn(1)>bestn(0),满足限界条件,因此节点 D 的右子节点 E 成为活节点,并成为当前的扩展节点,节点 E 代表 4 号点不加入最大团集合,如图 5-22(a)所示。

扩展节点 E 沿着左分支扩展,5 号点与最大团集合中的 1 号点没有边相连,不满足约束条件,E 的左孩子被剪掉。E 的右孩子判断限界条件,当前最大团集合已经 3 个点,E 的右分支代表 5 号点不加入最大团集合,剩余没判断是否加入最大团集合的点不存在,cn(3)+rn(0)>bestn(0),满足限界条件,因此节点 E 的右子节点 F 成为活节点,并成为当前的扩展节点。由于节点 E 已经是叶子节点,故找到了当前最优解,集合中含有 1、2、3 节点的团,bestn=3,如图 5-22(b)所示。

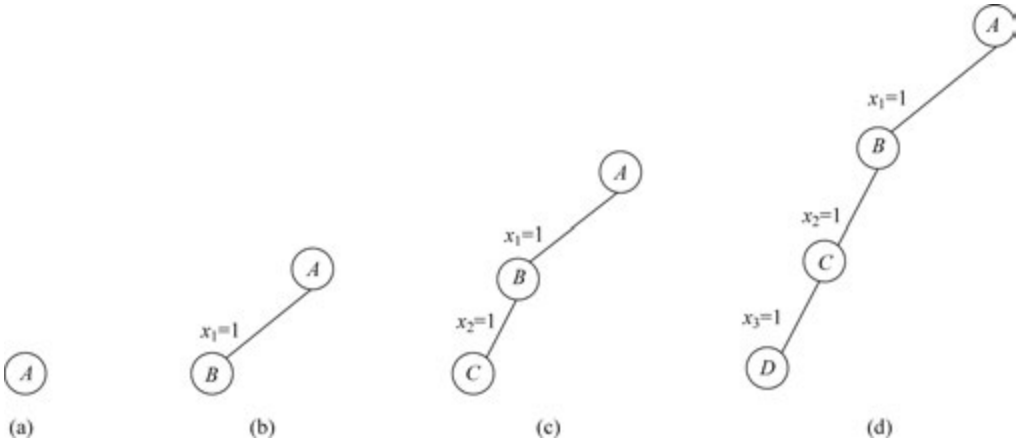

图 5-21　最大团问题的搜索过程(一)

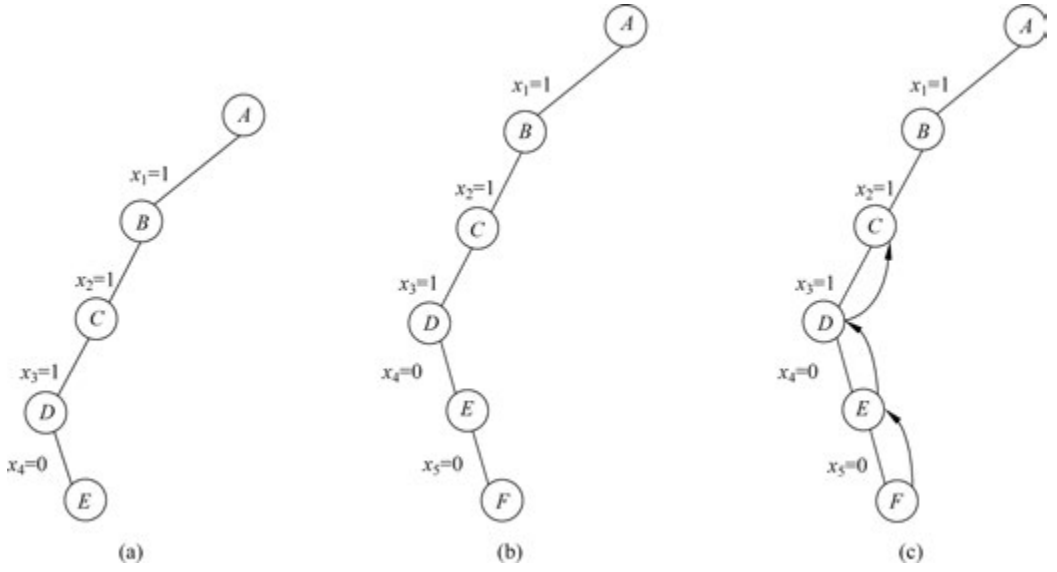

图 5-22　最大团问题的搜索过程(二)

开始回溯,节点 F 回溯到节点 E,E 的两个分支扩展完毕,E 成为死节点。节点 E 回溯到节点 D,D 的两个分支扩展完毕,D 成为死节点。继续回溯,节点 D 回溯到节点 C,如图 5-22(c)所示。

节点 C 的右分支还没有扩展,开始扩展 C 的右分支,判断限界条件:cn(2)+rn(2)>bestn(3),满足限界条件,节点 G 成为活节点,并成为当前的扩展节点,如图 5-23(a)所示。开始扩展 G 的左分支,判断约束条件,4 号点与最大团集合中的 1 号点没有边相连,不满足约束条件,G 的左孩子被剪掉。扩展 G 的右分支,判断限界条件:cn(2)+rn(1)=bestn(3),不满足限界条件,G 的右孩子也被剪掉,节点 G 成为死节点。算法开始回溯节点 C,C 的两个分支扩展完毕,C 成为死节点,继续回溯到节点 B,如图 5-23(b)所示。沿着节点 B 的右分支扩展,判断限界条件:cn(1)+rn(3)>bestn(3),满足限界条件,节点 H 成为活节点,并成为当前的扩展节点,如图 5-23(c)所示。

开始扩展 H 的左分支,判断约束条件,3 号点与最大团集合中的 1 号点有边相连,满足

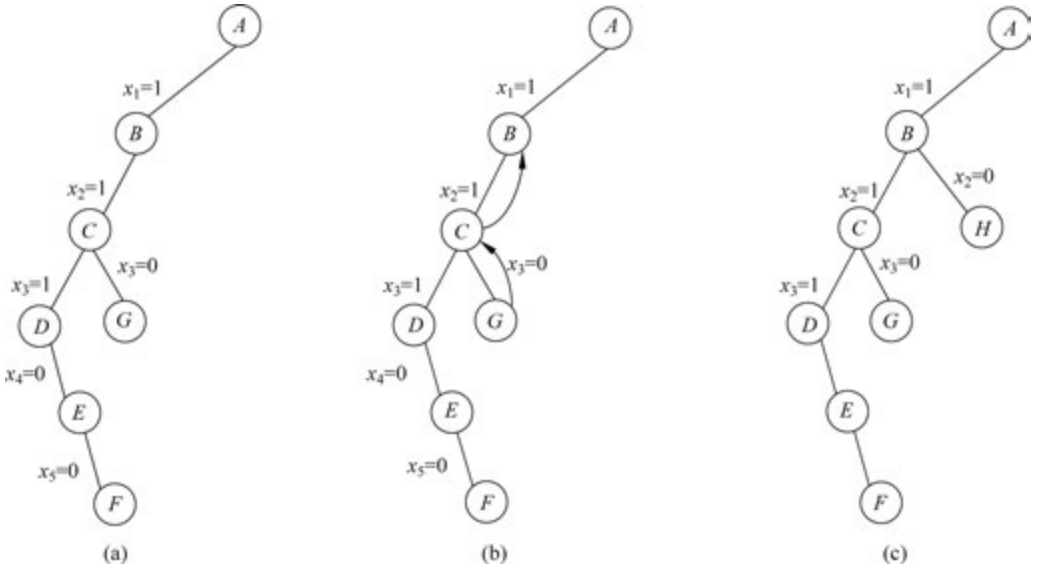

图 5-23 最大团问题的搜索过程(三)

约束条件,H 的左子节点 I 成为活节点,并成为当前的扩展节点,如图 5-24(a)所示。扩展 I 的左分支,判断约束条件,4 号点与最大团集合中的 1 号点没有边相连,不满足约束条件,I 的左孩子被剪掉。扩展 I 的右分支,判断限界条件 cn(2)+rn(1)=bestn(3),不满足限界条件,I 的右孩子也被剪掉,节点 I 成为死节点。算法开始回溯节点 H,如图 5-24(b)所示。扩展 H 的右分支,判断限界条件:cn(1)+rn(2)=bestn(3),不满足限界条件,H 的右分支被剪掉。算法开始回溯节点 B,节点 B 的两个分支都扩展完毕,成为死节点,继续回溯到节点 A,如图 5-24(c)所示。

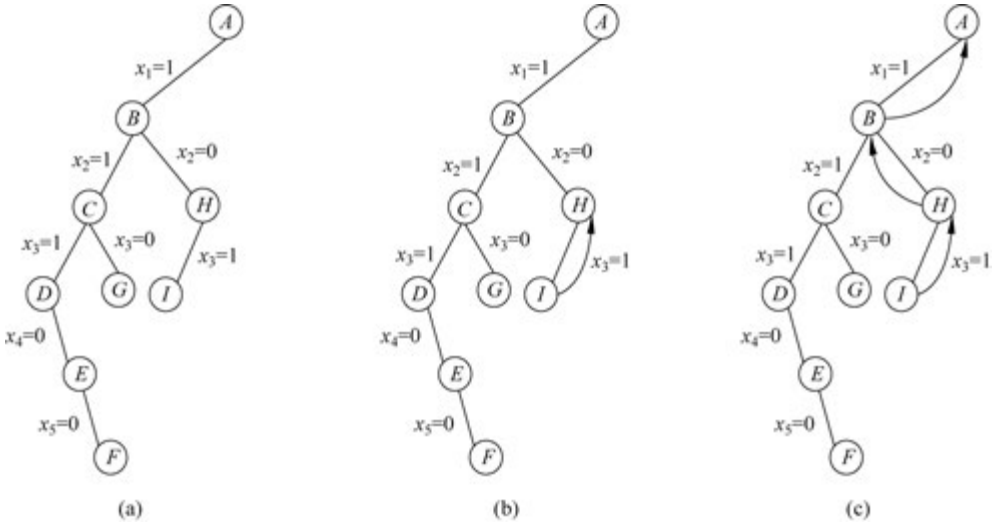

图 5-24 最大团问题的搜索过程(四)

开始扩展 A 的右分支,判断限界条件 cn(0)+rn(4)>bestn(3),满足限界条件,A 的右孩子 J 成为活节点,并成为当前的扩展节点,如图 5-25(a)所示。扩展 J 的左分支,判断约

束条件，当前最大团集合是空集，满足约束条件，J 的左子节点 K 成为活节点，并成为当前的扩展节点，2 号点加入最大团集合，如图 5-25(b)所示。扩展 K 的左分支，判断约束条件，3 号点与最大团集合中的 2 号点有边相连，满足约束条件，K 的左子节点 L 成为活节点，并成为当前的扩展节点，3 号点加入最大团集合，如图 5-25(c)所示。

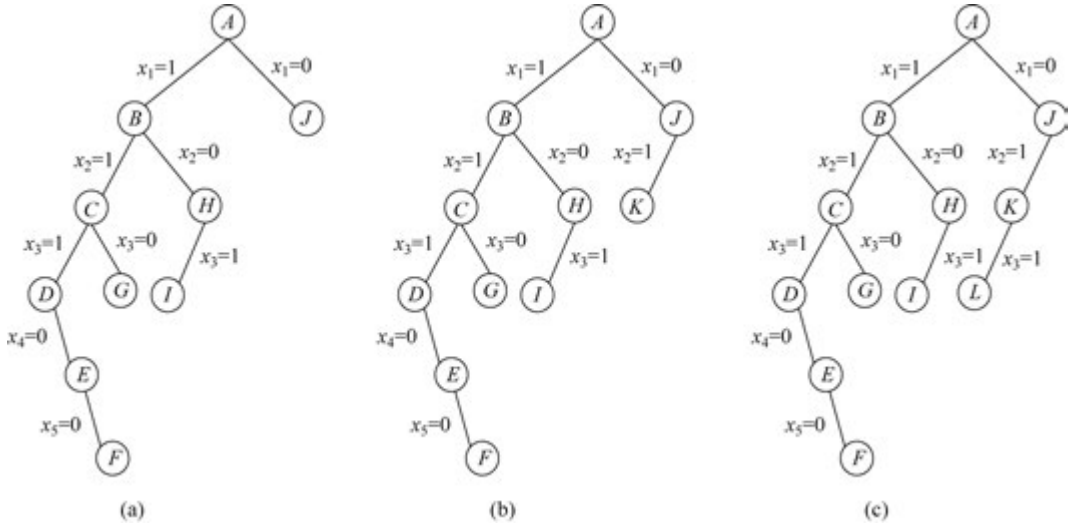

图 5-25　最大团问题的搜索过程(五)

扩展 L 的左分支，判断约束条件，4 号点与最大团集合中的 2、3 号点有边相连，满足约束条件，L 的左子节点 M 成为活节点，并成为当前的扩展节点，4 号点加入最大团集合，如图 5-26(a)所示。扩展 M 的左分支，判断约束条件，5 号点与最大团集合中的 2、3、4 号点有边相连，满足约束条件，M 的左子节点 N 成为活节点，并成为当前的扩展节点，5 号点加入最大团集合，如图 5-26(b)所示。开始回溯，节点 N 回溯到 M，扩展 M 的右分支，判断限界条件：cn(3)＋rn(0)＜bestn(4)，不满足限界条件，M 的右分支被剪掉。继续回溯到节点

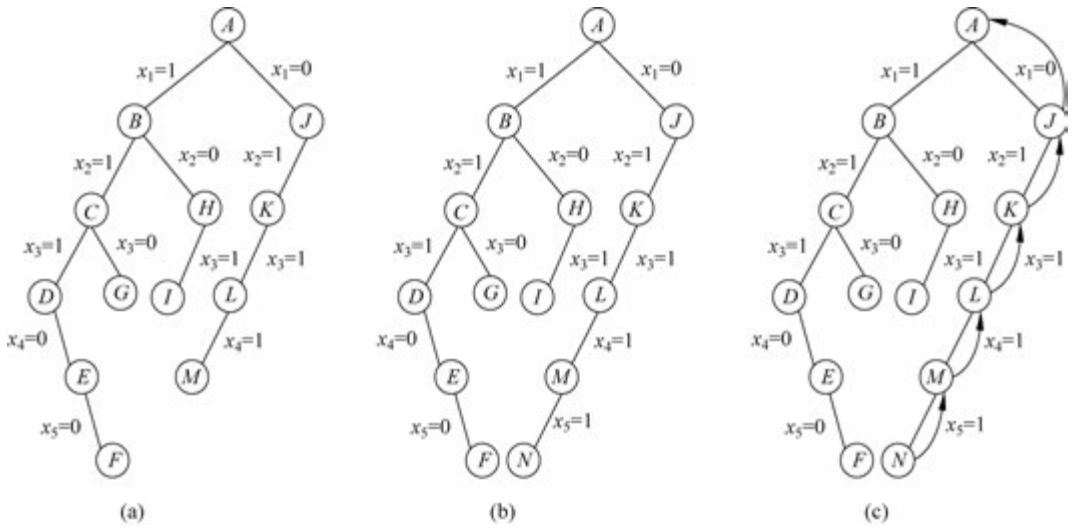

图 5-26　最大团问题的搜索过程(六)

L,扩展 L 的右分支,判断限界条件:cn(2)+rn(1)<bestn(4),不满足限界条件,L 的右分支被剪掉。继续回溯到节点 K,扩展 K 的右分支,判断限界条件:cn(1)+rn(2)<bestn(4),不满足限界条件,K 的右分支被剪掉。继续回溯到节点 J,扩展 J 的右分支,判断限界条件:cn(0)+rn(3)<bestn(4),不满足限界条件,J 的右分支被剪掉。继续回溯到节点 A,如图 5-26(c)所示。

A 的两个分支都搜索完毕,算法结束,形成的搜索树如图 5-27 所示。

所找到的问题的解是从根节点 A 到叶子节点 N 的路径(0,1,1,1,1),已在图 5-27 中用粗实线画出,求得的最大团如图 5-28 所示。

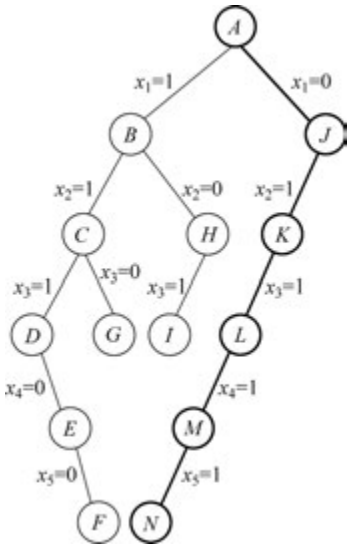

图 5-27　最大团问题的搜索树

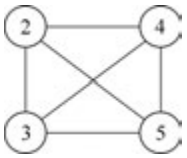

图 5-28　求得的最大团

5.4.4　算法分析

判断约束函数耗时 $O(n)$,在最坏情况下有 2^n-1 个左孩子,耗时最坏为 $O(n2^n)$。判断限界函数需要 $O(1)$时间,在最坏情况下有 2^n-1 个右子节点需要判断限界函数,耗时最坏为 $O(2^n)$。因此,最大团问题的回溯算法所需的计算时间为 $O(2^n)+O(n2^n)=O(n2^n)$。

5.4.5　Python 实践

首先定义一个 place()函数,用于判断指定的点是否能加入最大团集合中。该函数接收待判定的点,返回 True 或 False,True 表示指定点能放入最大团集合,False 表示指定点不能放入最大团集合。其代码如下:

```
def place(t):
    global x
    global a
    OK = True
    for j in range(t):
        if x[j] and a[t][j] == 0:
```

```
            OK = False
            break
    return OK
```

定义一个递归深度优先搜索的 backtrack()函数,搜索最优解。代码如下:

```
def backtrack(t):
    global cn,bestn,n,bestx,x
    if (t > n):
        bestx = x[:]
        bestn = cn
        return
    if place(t - 1):
        x[t - 1] = 1
        cn += 1
        backtrack(t + 1)
        cn -= 1
    if (cn + n - t > bestn):
        x[t - 1] = 0
        backtrack(t + 1)
```

Python 程序的入口是 main()函数。在 main()函数中,用邻接矩阵存储给定的图 G,调用 backtrack()函数,求最优解和最优值,最后将结果输出到显示器上。其代码如下:

```
if __name__ == "__main__":
    a = [[0,1,1,0,0],[1,0,1,1,1],[1,1,0,1,1],[0,1,1,0,1],[0,1,1,1,0]]
    n = len(a)
    x = [i for i in range(n)]
    bestx = None
    bestn = cn = 0
    backtrack(1)                          #根节点的层次为 1
    print("最大团点个数:", bestn)
    print("最大团为:", bestx)
```

输出结果为

```
最大团点个数: 4
最大团为: [0, 1, 1, 1, 1]
```

5.5　批处理作业调度问题——排列树

给定 n 个作业的集合$\{J_1,J_2,\cdots,J_n\}$。每个作业必须先由机器 1 处理,再由机器 2 处理。作业 J_i 需要机器 j 的处理时间为 t_{ji}。对于一个确定的作业调度,设 F_{ji} 是作业 J_i 在机器 j 上完成处理的时间。所有作业在机器 2 上完成处理的时间和称为该作业调度的完成时间和。批处理作业调度问题要求对于给定的 n 个作业,制订出最佳作业调度方案,使其完成时间和最小。

5.5.1 问题分析——解空间及搜索条件

根据问题描述可知,批处理作业调度问题要求找出 n 个作业$\{J_1,J_2,\cdots,J_n\}$的一个排列,按照这个排列的顺序进行调度,使得完成 n 个作业的完成时间和最小。按照回溯法的算法框架,首先需要定义问题的解空间,然后确定解空间的组织结构,最后进行搜索。搜索前要解决两个关键问题:一是确定问题是否需要约束条件(判断是否有可能产生可行解的条件),如果需要,应如何设置,由于作业的任何一种调度次序不存在无法调度的情况,均是合法的,因此,任何一个排列都表示问题的一个可行解,故不需要约束条件;二是确定问题是否需要限界条件,如果需要,应如何设置,在 n 个作业的 $n!$ 种调度方案(排列)中,存在完成时间和多与少的情况,该问题要求找出完成时间和最少的调度方案,因此,需要设置限界条件。

1. 确定问题的解空间

批处理作业调度问题解的形式为$(x_1,x_2,\cdots,x_n)$,分量 $x_i(i=1,2,\cdots,n)$表示第 i 个要调度的作业编号。设 n 个作业组成的集合为 $S=\{1,2,\cdots,n\}$,$x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}(i=1,2,\cdots,n)$。

2. 解空间的组织结构

解空间的组织结构是一棵排列树,树的深度为 n。

3. 搜索解空间

(1) 由于不需要约束条件,故无须设置。

(2) 设置限界条件。

用 cf 表示当前已完成调度的作业所用的时间和,用 bestf 表示当前找到的最优调度方案的完成时间和。显然,继续向纵深处搜索时,cf 不会减少,只会增加。因此当 cf≥bestf 时,没有继续向纵深处搜索的必要。限界条件可描述为 cf<bestf,cf 的初始值为 0,bestf 的初始值为$+\infty$。

5.5.2 算法设计

扩展节点沿着某个分支扩展时需要判断限界条件,如果满足,就进入深一层继续搜索;如果不满足,就将扩展生成的节点剪掉。搜索到叶子节点时,即找到当前最优解。搜索过程直到全部活节点变成死节点为止。

算法伪码描述如下:

```
算法:backtrack(t)
输入:扩展节点的层次 i,根节点层次为 1
输出:最优解 bestx 和最优值 bestf
    if (t > n):
        bestx←x[:]
        bestf←f
```

```
else:
    for j←t to n－1:
        f1←f1 + M[x[j]][1]
        k←t－1
        f2[t]←max(f2[k],f1) + M[x[j]][2]
        f←f + f2[t]
        if (f < bestf):
            x[t]↔x[j]
            Backtrack(t + 1)
            x[t]↔x[j]
        f1←f1－M[x[j]][1]
        f←f－f2[t]
```

5.5.3　实例构造

考虑 $n=3$ 的实例，每个作业在两台机器上的处理时间如表 5-1 所示。

表 5-1　作业在两台机器上的处理时间

作　　业	机　器　1	机　器　2
J_1	2	1
J_2	3	1
J_3	2	3

注：行分别表示作业 J_1、J_2 和 J_3；列分别表示机器 1 和机器 2。表中数据表示 t_{ji}，即作业 J_i 需要机器 j 的处理时间。

搜索过程如图 5-29～图 5-35 所示。从根节点 A 开始，节点 A 成为活节点，并且是当前的扩展节点，如图 5-29(a)所示。扩展节点 A 沿着 $x_1=1$ 的分支扩展，$F_{11}=2$，$F_{21}=3$，故cf=3，bestf=＋∞，cf<bestf，限界条件满足，扩展生成的节点 B 成为活节点，并且成为当前的扩展节点，如图 5-29(b)所示。扩展节点 B 沿着 $x_2=2$ 的分支扩展，$F_{12}=5$，$F_{22}=6$，故 cf＝$F_{21}+F_{22}=9$，bestf＝＋∞，cf<bestf，限界条件满足，扩展生成的节点 E 成为活节点，并且成为当前的扩展节点，如图 5-29(c)所示。扩展节点 E 沿着 $x_3=3$ 的分支扩展，$F_{13}=7$，$F_{23}=10$，故 cf＝$F_{21}+F_{22}+F_{23}=19$，bestf＝＋∞，cf<bestf，限界条件满足，扩展生成的节点 K 是叶子节点。此时，找到当前最优的一种调度方案(1，2，3)，同时修改 bestf＝19，如图 5-29(d)所示。

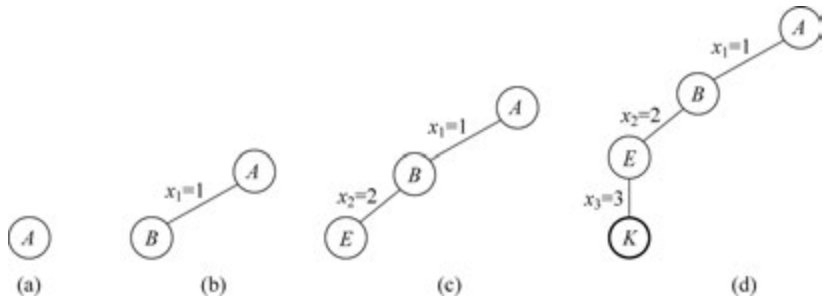

图 5-29　批处理作业调度问题的搜索过程(一)

叶子节点 K 不具备扩展能力，开始回溯到活节点 E。节点 E 只有一个分支，且已搜

索完毕,因此节点 E 成为死节点,继续回溯到活节点 B,节点 B 再次成为扩展节点,如图 5-30(a)所示。扩展节点 B 沿着 $x_2=3$ 的分支扩展,cf=10,bestf=19,cf<bestf,限界条件满足,扩展生成的节点 F 成为活节点,并且成为当前的扩展节点,如图 5-30(b)所示。扩展节点 F 沿着 $x_3=2$ 的分支扩展,cf=18,bestf=19,cf<bestf,限界条件满足,扩展生成的节点 L 是叶子节点。此时,找到比先前更优的一种调度方案(1,3,2),修改 bestf=18,如图 5-30(c)所示。从叶子节点 L 开始回溯到活节点 F。节点 F 的一个分支已搜索完毕,节点 F 成为死节点,回溯到活节点 B。节点 B 的两个分支已搜索完毕,回溯到活节点 A,节点 A 再次成为扩展节点,如图 5-30(d)所示。

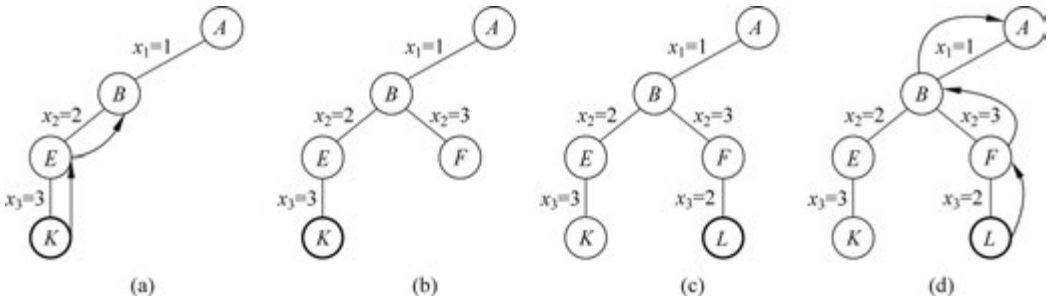

图 5-30 批处理作业调度问题的搜索过程(二)

扩展节点 A 沿着 $x_1=2$ 的分支扩展,cf=4,bestf=18,cf<bestf,限界条件满足,扩展生成的节点 C 成为活节点,并且成为当前的扩展节点,如图 5-31(a)所示。扩展节点 C 沿着 $x_2=1$ 的分支扩展,cf=10,bestf=18,cf<bestf,限界条件满足,扩展生成的节点 G 成为活节点,并且成为当前的扩展节点,如图 5-31(b)所示。扩展节点 G 沿着 $x_3=3$ 的分支扩展,cf=20,bestf=18,cf>bestf,限界条件不满足,扩展生成的节点被剪掉,如图 5-31(c)所示。

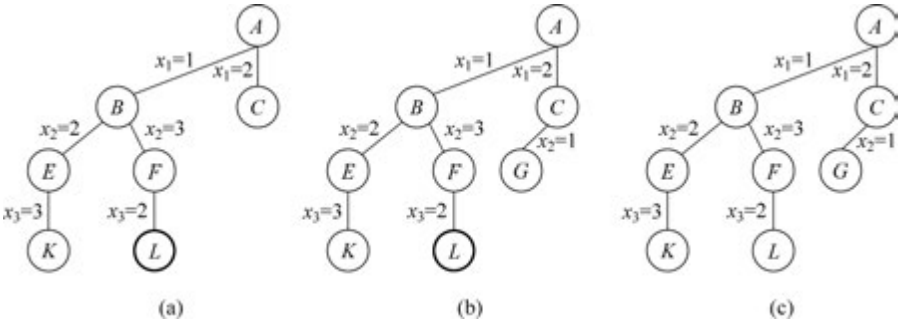

图 5-31 批处理作业调度问题的搜索过程(三)

节点 G 的一个分支搜索完毕,节点 G 成为死节点,继续回溯到活节点 C,如图 5-32(a)所示。扩展节点 C 沿着 $x_2=3$ 的分支扩展,cf=12,bestf=18,cf<bestf,限界条件满足,扩展生成的节点 H 成为活节点,并且成为当前的扩展节点,如图 5-32(b)所示。扩展节点 H 沿着 $x_3=1$ 的分支扩展,cf=21,bestf=18,cf>bestf,限界条件不满足,扩展生成的节点被剪掉。节点 H 的一个分支搜索完毕,开始回溯到活节点 C。此时,节点 C 的两个分支已搜索完毕,继续回溯到活节点 A,节点 A 再次成为当前的扩展节点,如图 5-32(c)所示。

扩展节点 A 沿着 $x_1=3$ 的分支扩展,cf=5,bestf=18,cf<bestf,限界条件满足,扩展

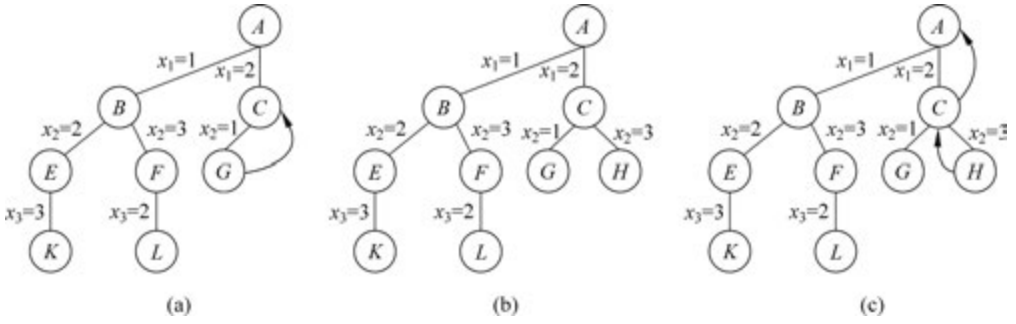

图 5-32　批处理作业调度问题的搜索过程(四)

生成的节点 D 成为活节点,并且成为当前的扩展节点,如图 5-33(a)所示。扩展节点 D 沿着 $x_2=1$ 的分支扩展,cf=11,bestf=18,cf<bestf,限界条件满足,扩展生成的节点 I 成为活节点,并且成为当前的扩展节点,如图 5-33(b)所示。

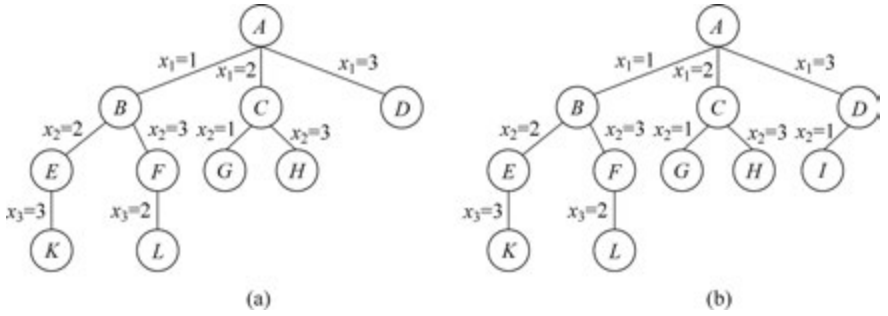

图 5-33　批处理作业调度问题的搜索过程(五)

扩展节点 I 沿着 $x_3=2$ 的分支扩展,cf=19,bestf=18,cf>bestf,限界条件不满足,扩展生成的节点被剪掉,开始回溯到活节点 D,节点 D 再次成为当前的扩展节点,如图 5-34(a)所示。扩展节点 D 沿着 $x_2=2$ 的分支扩展,cf=11,bestf=18,cf<bestf,限界条件满足,扩展生成的节点 J 成为活节点,并且成为当前的扩展节点,如图 5-34(b)所示。

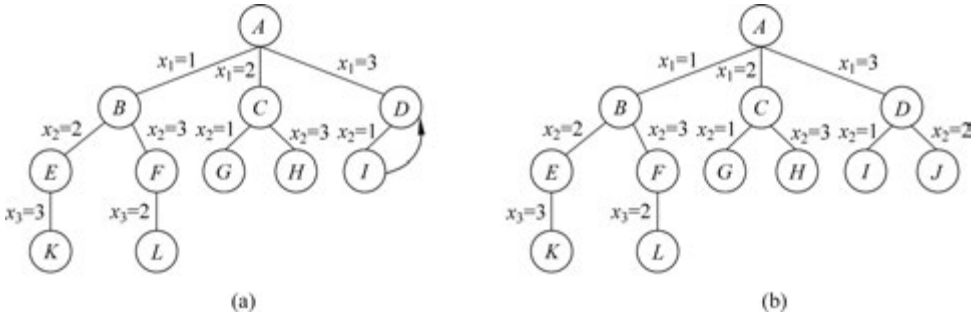

图 5-34　批处理作业调度问题的搜索过程(六)

扩展节点 J 沿着 $x_3=1$ 的分支扩展,cf=19,bestf=18,cf>bestf,限界条件不满足,扩展生成的节点被剪掉,开始回溯到活节点 D,节点 D 的两个分支搜索完毕,继续回溯到活节点 A,如图 5-35(a)所示。活节点 A 的三个分支也已搜索完毕,节点 A 变成死节点,搜索结束。至此,找到的最优的调度方案为从根节点 A 到叶子节点 L 的路径(1, 3, 2),如图 5-35(b)所示。

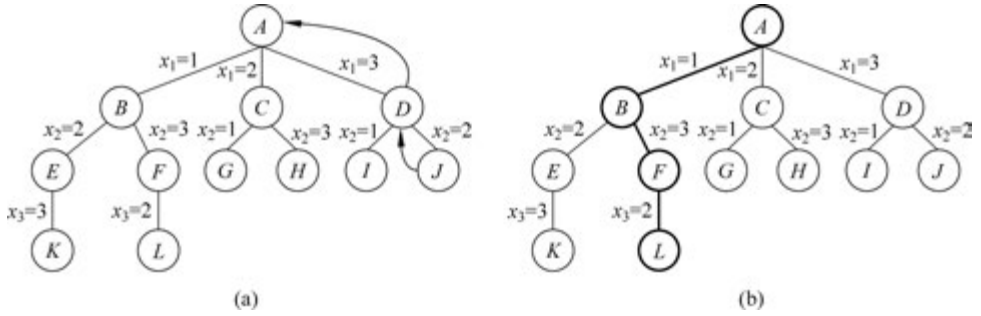

图 5-35　批处理作业调度问题的搜索过程(七)

5.5.4　算法分析

计算限界函数需要 $O(1)$时间,需要判断限界函数的节点在最坏情况下有 $1+n+n(n-1)+n(n-1)(n-2)+\cdots+n(n-1)+\cdots+2\leqslant nn!$个,故耗时 $O(nn!)$;在叶子节点处记录当前最优解耗时 $O(n)$,在最坏情况下会搜索到每一个叶子节点,叶子节点有 $n!$个,故耗时为 $O(nn!)$。因此,批处理作业调度问题的回溯算法所需的计算时间为 $O(nn!)+O(nn!)=O(nn!)$。

5.5.5　Python 实践

定义一个深度优先搜索的 backtrack()函数,搜索最优解。其代码如下:

```
def backtrack(t):
    global f1,f2,f,x,M,bestf,bestx,n
    if (t > n):                                    #搜索到叶子节点
        bestx = x[:]                               #记录当前最优解
        bestf = f                                  #记录当前最优值
    else:
        for j in range(t,n):
            f1 += M[x[j]][1]                       #计算第一台机器上的完成时间
            k = t-1                                #t 号作业的前一个作业
            f2[t] = max(f2[k],f1) + M[x[j]][2]     #计算第二台机器上的完成时间
            f += f2[t]                             #计算第二台机器上的完成时间和
            if (f < bestf):                        #判断限界条件
                x[t],x[j] = x[j],x[t]
                backtrack(t+1)                     #递归搜索
                x[t],x[j] = x[j],x[t]
            f1 -= M[x[j]][1]
            f -= f2[t]
```

Python 程序的入口是 main()函数。在 main()函数中,给定各作业在两台机器上的处理时间 M,初始化相关辅助变量,调用 backtrack()函数,求最优解和最优值,最后将结果输出到显示器上。其代码如下:

```
if __name__ == "__main__":
    import sys
    M = [[0,0,0],[0,2,1],[0,3,1],[0,2,3]]   #牺牲第 0 行第 0 列,从下标 1 开始有效
```

```
f = 0                                    # 记录完成时间和
f1 = 0                                   # 记录第一台机器的完成时间和
n = len(M)
f2 = [i for i in range(n)]               # 记录各作业在第二台机器上的完成时间和
x = [i for i in range(n)]
bestf = sys.maxsize                      # 记录最优值,初始化为无穷大
bestx = None                             # 记录最优解
backtrack(1)
print("最优解为:" ,bestx)
print("最优值为:", bestf)
```

输出结果为

```
最优解为: [0, 1, 3, 2]
最优值为: 18
```

5.6　旅行商问题——排列树

设有 n 个城市组成的交通图，一个售货员从住地城市出发，到其他城市各一次去推销货物，最后回到住地城市。假定任意两个城市 i、j 之间的距离 d_{ij}（$d_{ij}=d_{ji}$）是已知的，应该怎样选择一条最短的路线？

5.6.1　问题分析——解空间及搜索条件

旅行商问题给定 n 个城市组成的无向带权图 $G=(V,E)$，顶点代表城市，权值代表城市之间的路径长度。要求找出以住地城市开始的一个排列，按照这个排列的顺序推销货物，所经路径长度是最短的。问题的解空间是一棵排列树。显然，对于任意给定的一个无向带权图，存在某两个城市（顶点）之间没有直接路径（边）的情况。也就是说，并不是任何一个以住地城市开始的排列都是一条可行路径（问题的可行解），因此需要设置约束条件，判断排列中相邻两个城市之间是否有边相连，有边相连则能走通，否则就不是可行路径。另外，在所有可行路径中要找一条最短的路线，因此需要设置限界条件。

1. 定义问题的解空间

旅行商问题的解空间形式为 n 元组（$x_1,x_2,\cdots,x_n$），分量 x_i（$i=1,2,\cdots,n$）表示第 i 个去推销货物的城市号。假设住地城市编号为城市 1，其他城市顺次编号为 2，3，…，n。n 个城市组成的集合为 $S=\{1,2,\cdots,n\}$。由于住地城市是确定的，因此 x_1 的取值只能是住地城市，即 $x_1=1$，$x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}$（$i=2,3,\cdots,n$）。

2. 确定解空间的组织结构

该问题的解空间是一棵排列树，树的深度为 n。$n=4$ 的旅行商问题的解空间树如图 5-36 所示。

3. 搜索解空间

(1) 设置约束条件。用列表 $g[\,][\,]$存储无向带权图的邻接矩阵,如果 $g[i][j]\neq\infty$表示城市 i 和城市 j 有边相连,能走通。

(2) 设置限界条件。用 cl 表示当前已走过的城市所用的路径长度,用 bestl 表示当前找到的最短路径的路径长度。显然,继续向纵深处搜索时,cl 不会减少,只会增加。因此当 cl≥bestl 时,没有继续向纵深处搜索的必要。限界条件可描述为 cl<bestl,cl 的初始值为 0,bestl 的初始值为+∞。

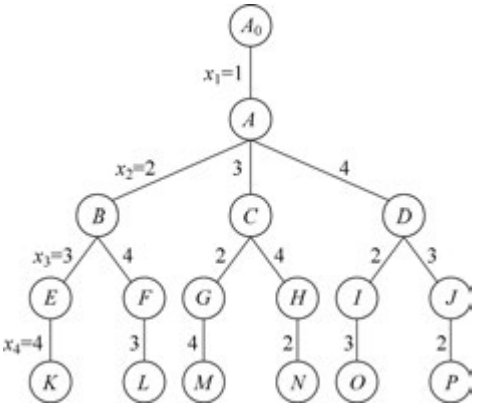

图 5-36 $n=4$ 的解空间树

5.6.2 算法设计

扩展节点沿着某个分支扩展时需要判断约束条件和限界条件,如果两者都满足,就进入深一层继续搜索;否则,剪掉扩展生成的节点。搜索到叶子节点时,找到当前最优解。搜索过程直到全部活节点变成死节点。

算法伪码描述如下:

```
算法:backtrack(t):
输入:节点所处的层次,根节点层次为 1
输出:最优解和最优值
    g_n ← n - 1
    if (t == g_n) then                                          //g_n 是问题的规模
        if (a[x[g_n - 1]][x[g_n]] != NoEdge and a[x[g_n]][1] != NoEdge and (cc + a[x[g_n -
1]][x[g_n]] + a[x[g_n]][1] < bestc or bestc == NoEdge)) then  //判断约束条件和限界条件
            bestx ← x[:]                                        //记录当前最优解
            bestc ← cc + a[x[g_n - 1]][x[g_n]] + a[x[g_n]][1]//记录当前最优值
    else
        for j ← i to n - 1 do                                   //控制搜索扩展节点的所有
                                                                //分支
            if (a[x[i - 1]][x[j]] != NoEdge and (cc + a[x[i - 1]][x[i]] < bestc or bestc ==
NoEdge)) then                                                   //判断约束条件和限界条件
                x[i]↔x[j]
                cc←cc + a[x[i - 1]][x[i]]
                backtrack(i + 1)
                cc←cc - a[x[i - 1]][x[i]]
                x[i]↔x[j]
```

5.6.3 实例构造

考虑 $n=5$ 的无向带权图,如图 5-37 所示。

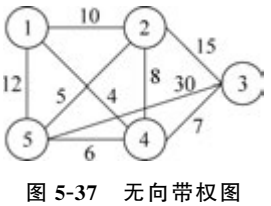

图 5-37 无向带权图

搜索过程如图 5-38～图 5-42 所示。由于排列的第一个元素已经确定,即推销员的住地城市 1,搜索从根节点 A_0 的子节点 A 开始,节点 A 是活节点,并且成为当前的扩展节点,如图 5-38(a)所

示。扩展节点 A 沿着 $x_2=2$ 的分支扩展，城市 1 和城市 2 有边相连，约束条件满足；cl=10，bestl=∞，cl<bestl，限界条件满足，扩展生成的节点 B 成为活节点，并且成为当前的扩展节点，如图 5-38(b)所示。扩展节点 B 沿着 $x_3=3$ 的分支扩展，城市 2 和城市 3 有边相连，约束条件满足；cl=25，bestl=∞，cl<bestl，限界条件满足，扩展生成的节点 C 成为活节点，并且成为当前的扩展节点，如图 5-38(c)所示。扩展节点 C 沿着 $x_4=4$ 的分支扩展，城市 3 和城市 4 有边相连，约束条件满足；cl=32，bestl=∞，cl<bestl，限界条件满足，扩展生成的节点 D 成为活节点，并且成为当前的扩展节点，如图 5-38(d)所示。

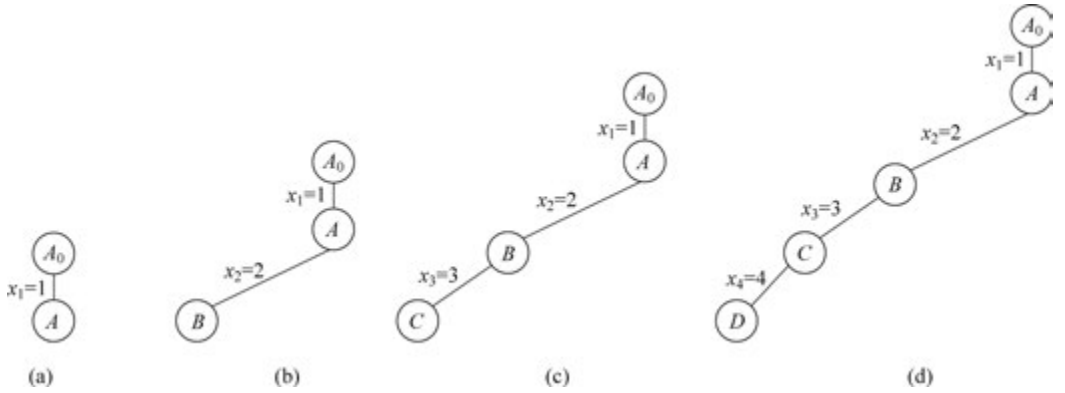

图 5-38　旅行商问题的搜索过程(一)

扩展节点 D 沿着 $x_5=5$ 的分支扩展，城市 4 和城市 5 有边相连，约束条件满足；cl=38，bestl=∞，cl<bestl，限界条件满足，扩展生成的节点 E 是叶子节点。由于城市 5 与住地城市 1 有边相连，故找到一条当前最优路径(1,2,3,4,5)，其长度为 50，修改 bestl=50，如图 5-39(a)所示。接下来开始回溯到节点 D，再回溯到节点 C，C 成为当前的扩展节点，如图 5-39(b)所示。

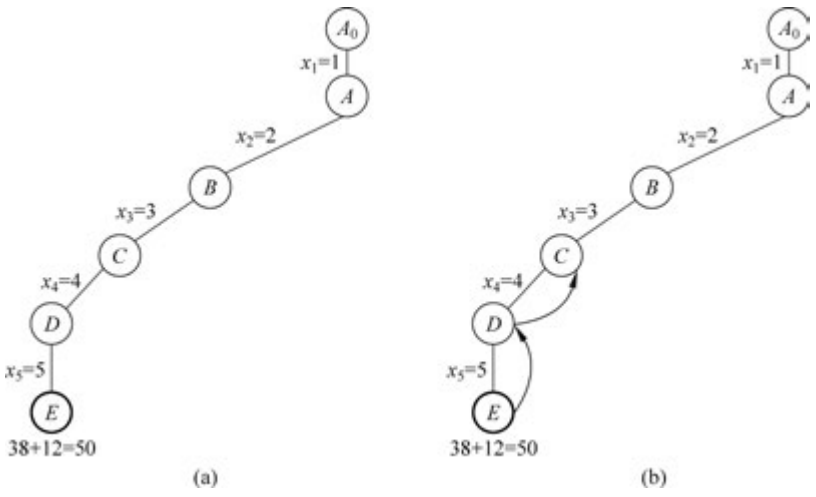

图 5-39　旅行商问题的搜索过程(二)

以此类推，第一次回溯到第二层的节点 A 时的搜索树如图 5-40 所示。节点旁边的“×”表示不能从推销货物的最后一个城市回到住地城市。

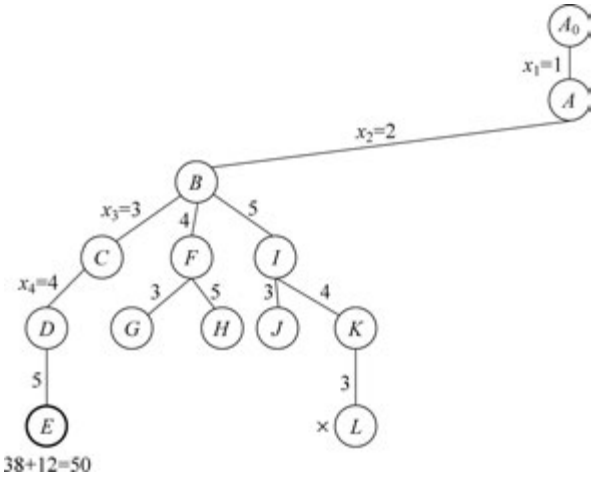

图 5-40　旅行商问题的搜索过程(三)

第二层的节点 A 再次成为扩展节点,开始沿着 $x_2=3$ 的分支扩展,城市 1 和城市 3 之间没有边相连,不满足约束条件,扩展生成的节点被剪掉。沿着 $x_2=4$ 的分支扩展,满足约束条件和限界条件,进入其扩展的子节点继续搜索。搜索过程略。此时,找到当前最优解(1,4,3,2,5),路径长度为 43。直到第二次回溯到第二层的节点 A 时所形成的搜索树如图 5-41 所示。

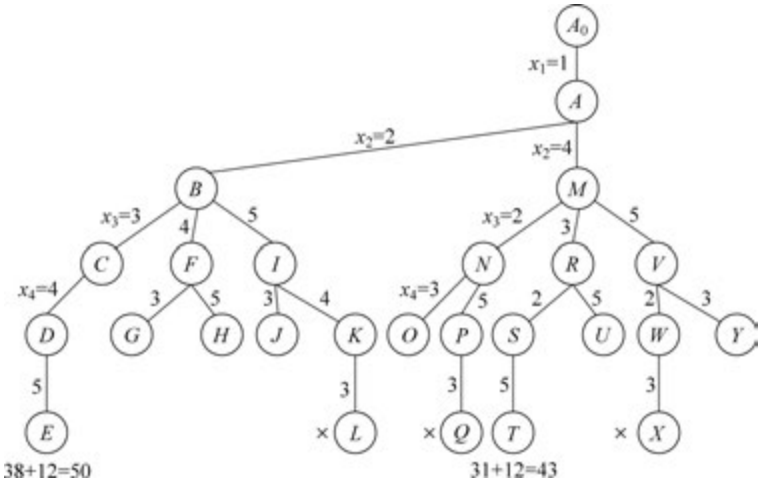

图 5-41　旅行商问题的搜索过程(四)

节点 A 沿着 $x_2=5$ 的分支扩展,满足约束条件和限界条件,进入其扩展的子节点继续搜索,搜索过程略。直到第三次回溯到第二层的节点 A 时所形成的搜索树如图 5-42 所示。此时,搜索过程结束,找到的最优解为图 5-42 中粗线条描述的路径(1,4,3,2,5),路径长度为 43。

5.6.4　算法分析

判断限界函数需要 $O(1)$时间,在最坏情况下有 $1+(n-1)+[(n-1)(n-2)]+\cdots+[(n-1)(n-2)\cdot\cdots\cdot 2]\leqslant n(n-1)!$个节点需要判断限界函数,故耗时 $O(n!)$;在叶子节

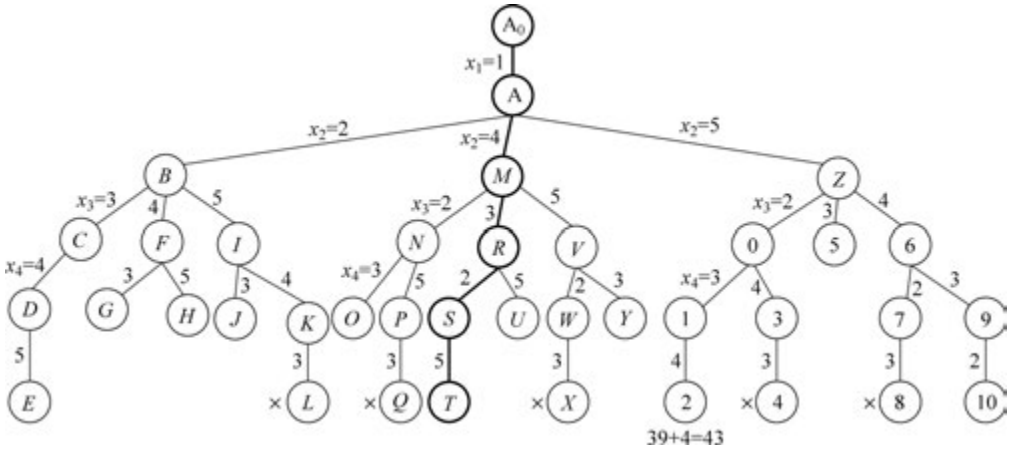

图 5-42　旅行商问题的搜索过程(五)

点处记录当前最优解耗时 $O(n)$,在最坏情况下会搜索到每个叶子节点,叶子节点有 $(n-1)!$ 个,故耗时为 $O(n!)$。因此,旅行商问题的回溯算法所需的计算时间为 $O(n!)+O(n!)=O(n!)$。

5.6.5　Python 实践

定义一个深度优先搜索的 backtrack()函数,搜索最优解。其代码如下:

```
def backtrack(t):
    global n,a,x,bestc,NoEdge,cc,bestx
    g_n = n-1
    if (t == g_n):                                  #g_n 是问题的规模
        if (a[x[g_n-1]][x[g_n]] != NoEdge and a[x[g_n]][1] != NoEdge and (cc + a[x[g_n-
1]][x[g_n]] + a[x[g_n]][1] < bestc or bestc == NoEdge)):
            bestx = x[:]
            bestc = cc + a[x[g_n-1]][x[g_n]] + a[x[g_n]][1]
    else:
        for j in range(t,n):
            if (a[x[t-1]][x[j]] != NoEdge and (cc + a[x[t-1]][x[t]] < bestc or bestc ==
NoEdge)):
                x[t], x[j] = x[j], x[t]
                cc += a[x[t-1]][x[t]]
                backtrack(t+1)
                cc -= a[x[t-1]][x[t]]
                x[t], x[j] = x[j], x[t]
```

Python 程序的入口是 main()函数。在 main()函数中,用邻接矩阵 ***a*** 存储无向带权图,牺牲 0 行 0 列位置的存储单元,下标从 1 开始有效。初始化相关辅助变量,调用 backtrack()函数,求最优解和最优值,最后将结果输出到显示器上,最优解数组 bestx 的 0 号存储单元舍弃,下标也从 1 开始有效。其代码如下:

```
if __name__ == "__main__":
    import sys
    NoEdge = sys.maxsize
```

```
    a = [[0,0,0,0,0,0],[0,NoEdge,10,NoEdge,4,12],[0,10,NoEdge,15,8,5],[0,NoEdge,15,
NoEdge,7,30],[0,4,8,7,NoEdge,6],[0,12,5,30,6,NoEdge]]
    n = len(a)
    x = [i for i in range(n)]
    bestx = None
    bestc = NoEdge
    cc = 0
    backtrack(2)                                    #第一个城市固定,所以从第二层开始搜索
    print("最短路径长度为:", bestc)
    print("最短路径为:", bestx)
```

输出结果为

```
最短路径长度为: 43
最短路径为: [0, 1, 4, 3, 2, 5]
```

5.7 图的 m 着色问题——满 m 叉树

给定无向连通图 $G=(V,E)$ 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。如果有一种着色法使 G 中有边相连的两个顶点着不同颜色，则称这个图是 m 可着色的。图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色方法。

5.7.1 问题分析——解空间及搜索条件

该问题中每个顶点所着的颜色均有 m 种选择，n 个顶点所着颜色的一个组合是一个可能的解。根据回溯法的算法框架，定义问题的解空间及其组织结构是很容易的。需不需要设置约束条件和限界条件呢？从给定的已知条件来看，无向连通图 G 中假设有 n 个顶点，它肯定至少有 $n-1$ 条边，有边相连的两个顶点所着颜色不相同，n 个顶点所着颜色的所有组合中必然存在不是问题着色方案的组合，因此需要设置约束条件；而针对所有可行解(组合)，不存在可行解优劣的问题，所以不需要设置限界条件。

1. 定义问题的解空间

图的 m 着色问题的解空间形式为 $(x_1,x_2,\cdots,x_n)$，分量 $x_i(i=1,2,\cdots,n)$ 表示第 i 个顶点着第 x_i 号颜色。m 种颜色的色号组成的集合为 $S=\{1,2,\cdots,m\}$，$x_i\in S(i=1,2,\cdots,n)$。

2. 确定解空间的组织结构

问题的解空间组织结构是一棵满 m 叉树，树的深度为 n。

3. 搜索解空间

(1) 设置约束条件。当前顶点要和前面已确定颜色且有边相连的顶点所着颜色不相同。假设当前扩展节点所在的层次为 t，则下一步扩展就是要判断第 t 个顶点着什么颜色，

第 t 个顶点所着的颜色要与已经确定所着颜色的第 1～($t-1$)个顶点中与其有边相连的颜色不相同。

约束函数代码如下：

```
def Ok(k):
    for j in range(1,k):
        if ((a[k][j] == 1) and (x[j] == x[k])):
            return False
    return True
```

(2) 无须设置限界条件。

5.7.2　算法设计

扩展节点沿着某个分支扩展时需要判断约束条件，如果满足，就进入深一层继续搜索；如果不满足，就剪掉扩展生成的节点。搜索到叶子节点时，找到一种着色方案。搜索过程直到全部活节点变成死节点为止。

算法伪码描述如下：

```
算法:backtrack(t)
输入:扩展节点的层次,根节点层次为 1
输出:最优解和最优值
    if (t > n) then
        sum1 ← sum + 1                          //着色方案数
        用 colors 记录着色方案
    else
        for i←1 to m do                         //循环 m 种颜色
            x[t]←i
            if (Ok(t)) then                     //判断约束条件
                backtrack(t + 1)
```

5.7.3　实例构造

给定如图 5-43 所示的无向连通图且 $m=3$。

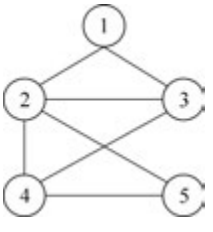

图 5-43　无向连通图

图的 m 着色问题的搜索过程如图 5-44～图 5-49 所示。从根节点 A 开始，节点 A 是当前的活节点，也是当前的扩展节点，它代表的状态是给定无向连通图中任何一个顶点还没有着色，如图 5-44(a)所示。沿着 $x_1=1$ 分支扩展，满足约束条件，生成的节点 B 成为活节点，并且成为当前的扩展节点，如图 5-44(b)所示。扩展节点 B 沿着 $x_2=1$ 分支扩展，不满足约束条件，生成的节点被剪掉。然后沿着 $x_2=2$ 分支扩展，满足约束条件，生成的节点 C 成为活节点，并且成为当前的扩展节点，如图 5-44(c)所示。扩展节点 C 沿着 $x_3=1,2$ 分支扩展，均不满足约束条件，生成的节点被剪掉。然后沿着 $x_3=3$ 分支扩展，满足约束条件，生成的节点 D 成为活节点，并且成为当前的扩展节点，如图 5-44(d)所示。

扩展节点 D 沿着 $x_4=1$ 分支扩展，满足约束条件，生成的节点 E 成为活节点，并且成

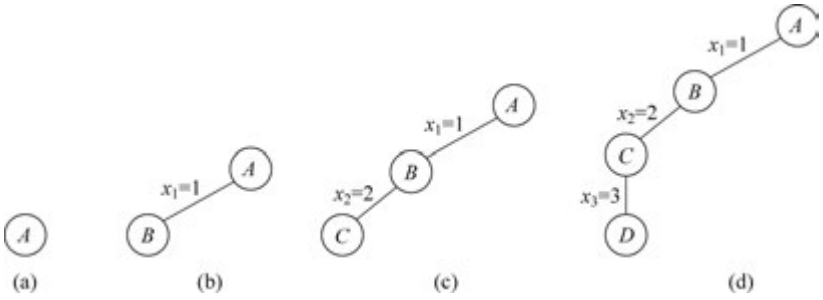

图 5-44 图的 *m* 着色问题的搜索过程(一)

为当前的扩展节点,如图 5-45(a)所示。扩展节点 E 沿着 $x_5=1,2$ 分支扩展,均不满足约束条件,生成的节点被剪掉。然后沿着 $x_5=3$ 分支扩展,满足约束条件,生成的节点 F 是叶子节点。此时,找到了一种着色方案,如图 5-45(b)所示。从叶子节点 F 回溯到活节点 E,节点 E 的所有子节点已搜索完毕,因此它成为死节点。继续回溯到活节点 D,节点 D 再次成为扩展节点,如图 5-45(c)所示。

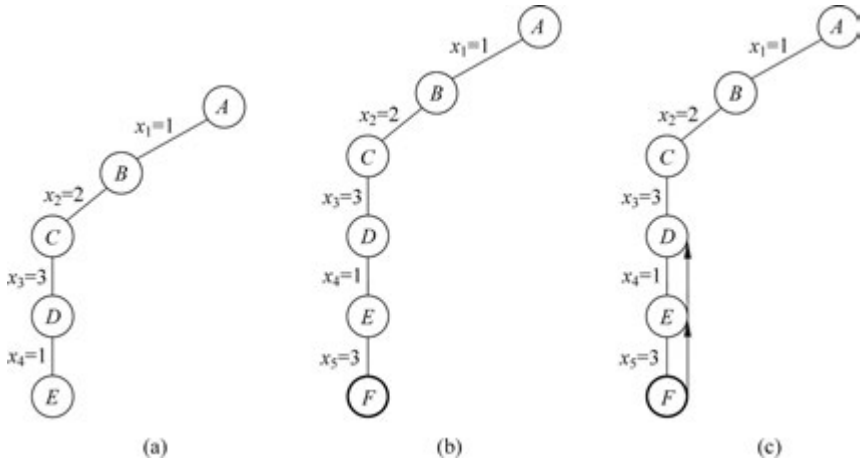

图 5-45 图的 *m* 着色问题的搜索过程(二)

扩展节点 D 沿着 $x_4=2$ 和 $x_4=3$ 分支扩展,均不满足约束条件,生成的节点被剪掉,再回溯到活节点 C。节点 C 的所有子节点搜索完毕,它成为死节点,继续回溯到活节点 B,节点 B 再次成为扩展节点,如图 5-46(a)所示。扩展节点 B 沿着 $x_2=3$ 分支继续扩展,满足约束条件,生成的节点 G 成为活节点,并且成为当前的扩展节点,如图 5-46(b)所示。扩展节点 G 沿着 $x_3=1$ 分支扩展,不满足约束条件,生成的节点被剪掉;然后沿着 $x_3=2$ 分支扩展,满足约束条件,生成的节点 H 成为活节点,并且成为当前的扩展节点,如图 5-46(c)所示。

扩展节点 H 沿着 $x_4=1$ 分支扩展,满足约束条件,生成的节点 I 成为活节点,并且成为当前的扩展节点,如图 5-47(a)所示。扩展节点 I 沿着 $x_5=1$ 分支扩展,不满足约束条件,生成的节点被剪掉;然后沿着 $x_5=2$ 分支扩展,满足约束条件,J 已经是叶子节点,找到第 2 种着色方案,如图 5-47(b)所示。从叶子节点 J 回溯到活节点 I,节点 I 再次成为扩展节点,如图 5-47(c)所示。

沿着节点 I 的 $x_5=3$ 分支扩展的节点不满足约束条件,被剪掉。此时节点 I 成为死节

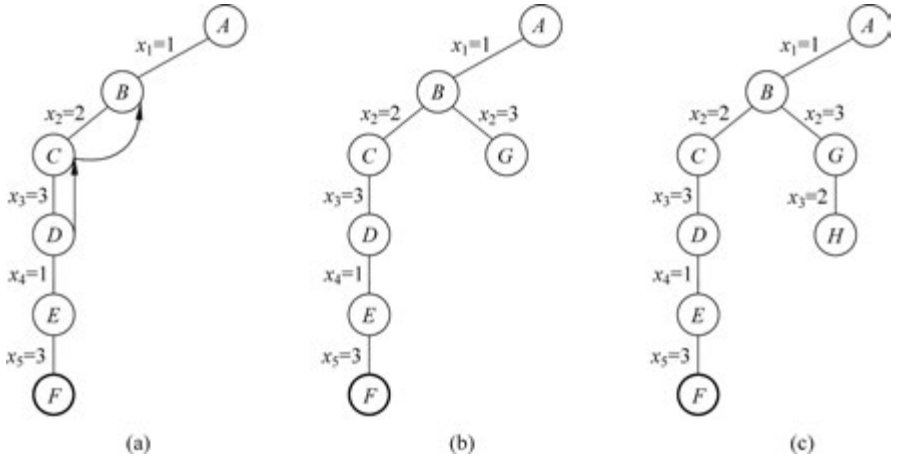

图 5-46　图的 *m* 着色问题的搜索过程(三)

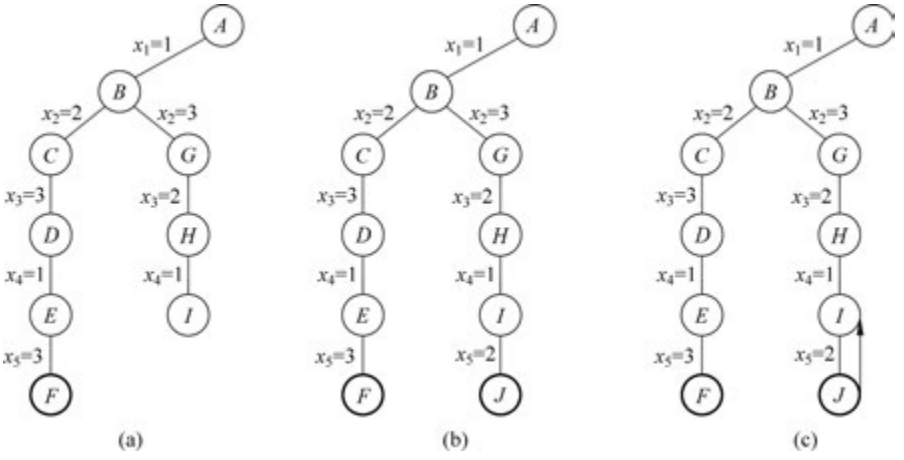

图 5-47　图的 *m* 着色问题的搜索过程(四)

点。继续回溯到活节点 H,节点 H 再次成为扩展节点,如图 5-48(a)所示。沿着节点 H 的 $x_4=2,3$ 分支扩展的节点不满足约束条件,被剪掉。此时节点 H 成为死节点。继续回溯到活节点 G,节点 G 再次成为扩展节点,如图 5-48(b)所示。沿着节点 G 的 $x_3=3$ 分支扩展的节点不满足约束条件,被剪掉。此时节点 G 成为死节点。继续回溯到活节点 B,节点 B 的子节点已搜索完毕,继续回溯到节点 A,如图 5-48(c)所示。

以此类推,扩展节点 A 沿着 $x_1=2,3$ 分支扩展的情况如图 5-49 所示。

最终找到 6 种着色方案,分别为根节点 A 到如图 5-49(b)所示的叶子节点 F、J、O、S、X、I 的路径,即(1,2,3,1,3)、(1,3,2,1,2)、(2,1,3,2,3)、(2,3,1,2,1)、(3,1,2,3,2)和(3,2,1,3,1)。

5.7.4　算法分析

计算限界函数需要 $O(n)$时间,需要判断限界函数的节点在最坏情况下有 $1+m+m^2+m^3+\cdots+m^{n-1}=(m^n-1)/(m-1)$个,故耗时 $O(nm^n)$;在叶子节点处输出着色方案耗时

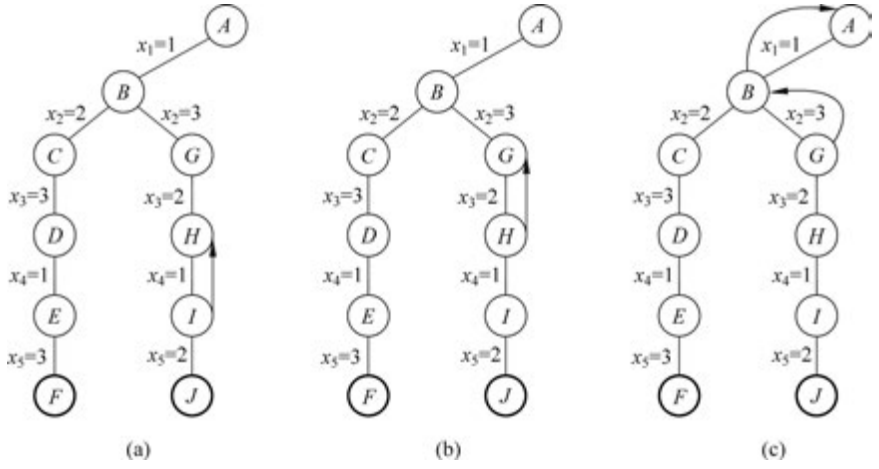

图 5-48 图的 *m* 着色问题的搜索过程(五)

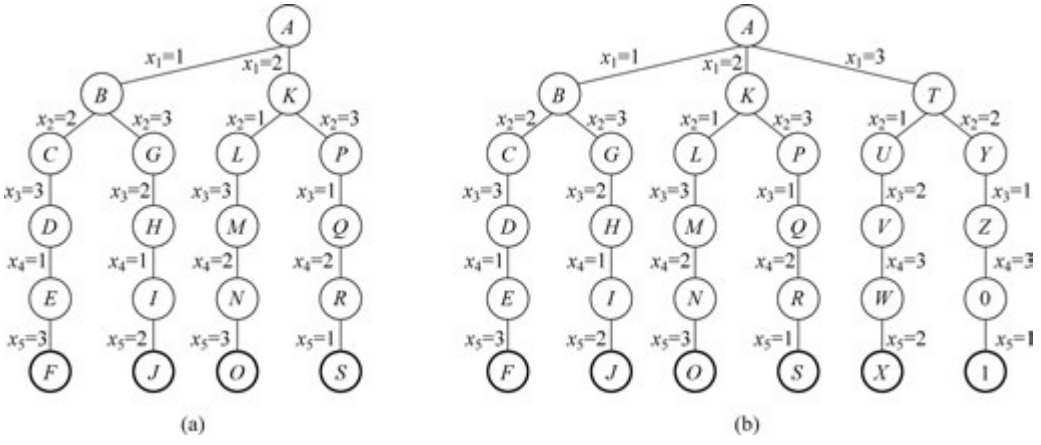

图 5-49 图的 *m* 着色问题的搜索过程(六)

$O(n)$,在最坏情况下会搜索到每个叶子节点,叶子节点有 m^n 个,故耗时为 $O(nm^n)$。图的 m 着色问题的回溯算法所需的计算时间为 $O(nm^n)+O(nm^n)=O(nm^n)$。

5.7.5 Python 实践

1. 数据结构选择

选用邻接矩阵 ***a*** 存储图 G,用列表存储所有的着色方案,用一维列表存储当前着色方案。

2. 编码实现

首先定义 ok()函数,接收待着色的第 k 号顶点编号,输出是否都能为该点着相应的颜色 $x[k]$。True 表示 k 号顶点能着 $x[k]$号色; False 表示 k 号顶点不能着 $x[k]$号色。其代码如下:

```
#牺牲下标为 0 的单元
def ok(k):
    for j in range(1,k):
        if ((a[k][j] == 1) and (x[j] == x[k])):
            return False
    return True
```

定义一个深度优先搜索的 backtrack()函数,搜索所有可能的着色方案,并统计着色方案数。其代码如下:

```
def backtrack(t):
    global colors,x,sum1,n,m,a
    if (t > n):                          #搜索到叶子节点
        sum1 += 1
        colors.append(x[:])
    else:
        for i in range(1,m + 1):         #搜索中间节点的每个分支,即尝试着任何一种颜色
            x[t] = i
            if (Ok(t)):
                backtrack(t + 1)
```

Python 程序的入口是 main()函数。在 main()函数中,用邻接矩阵 ***a*** 存储无向图,牺牲 0 行 0 列位置的存储单元,下标从 1 开始有效。初始化相关辅助变量,调用 backtrack()函数,求所有可能的着色方案,最后将结果打印输出到显示器上。输出结果中,0 号存储单元数据无效,从下标 1 开始有效。其代码如下:

```
if __name__ == "__main__":
    a = [[0,0,0,0,0,0],[0,0,1,1,0,0],[0,1,0,1,1,1],[0,1,1,0,1,0],[0,0,1,1,0,1],[0,0,1,
0,1,0]]
    n = len(a) - 1
    m = 3
    sum1 = 0
    colors = []
    x = [0 for i in range(n + 1)]
    backtrack(1)
    for i in range(len(colors)):
        print(colors[i])
    print("共有:" + str(sum1) + "种着色方案")
```

输出结果为

```
[0, 1, 2, 3, 1, 3]
[0, 1, 3, 2, 1, 2]
[0, 2, 1, 3, 2, 3]
[0, 2, 3, 1, 2, 1]
[0, 3, 1, 2, 3, 2]
[0, 3, 2, 1, 3, 1]
共有: 6 种着色方案
```

5.8 最小质量机器设计问题——满 m 叉树

设某种机器由 n 个部件组成,每个部件可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的质量,c_{ij} 是相应的价格。试设计一个算法,给出总价格不超过 c 的最小质量机器设计。

5.8.1 问题分析——解空间及搜索条件

该问题实质上是为机器部件选供应商。机器由 n 个部件组成,每个部件有 m 个供应商可以选择,要求找出 n 个部件供应商的一个组合,使其满足 n 个部件总价格不超过 c 且总重量是最小的。显然,这个问题存在 n 个部件供应商的组合不满足总价格不超过 c 的条件,因此需要设置约束条件;在 n 个部件供应商的组合满足总价格不超过 c 的前提下,哪个组合的总重量最小呢?问题要求找出总重量最小的组合,故需要设置限界条件。

1. 定义问题的解空间

该问题的解空间形式为$(x_1,x_2,\cdots,x_n)$,分量 $x_i(i=1,2,\cdots,n)$表示第 i 个部件从第 x_i 个供应商处购买。m 个供应商的集合为 $S=\{1,2,\cdots,m\}$,$x_i\in S(i=1,2,\cdots,n)$。

2. 确定解空间的组织结构

问题解空间的组织结构是一棵满 m 叉树,树的深度为 n。

3. 搜索解空间

(1) 设置约束条件。约束条件设置为 $\sum_{i=1}^{n} c_{ix_i} \leqslant c$。

(2) 设置限界条件。假设当前扩展节点所在的层次为 t,则下一步扩展就是要判断第 t 个零件从哪个供应商处购买。如果第 $1\sim t$ 个部件的重量之和大于或等于当前最优值,就没有继续深入搜索的必要。因为,再继续深入搜索也不会得到比当前最优解更优的一个解。令第 $1\sim t$ 个部件的重量之和用 $\mathrm{cw}=\sum_{i=1}^{t} w_{ix_i}$ 表示,价格之和用 cc 表示,二者初始值均为 0。当前最优值用 bestw 表示,初始值为$+\infty$,限界条件可描述为 cw<bestw。

5.8.2 算法设计

与图的 m 着色问题相同。

算法伪码描述如下:

```
算法:backtrack(t)
输入:扩展节点所处层次,根节点层次为 1
输出:最小质量机器设计问题最优方案 bestx 及最优值 bestw
    if(t > n) then                                    //搜索到解空间树的叶子节点
```

```
        bestw ← cw
        bestx ← x[:]
        return
    for j←1 to m do                                          //循环 m 个供应商
        x[t]←j                                               //第 t 个部件从 j 供应商处购买
        if(cc + c[t][j]≤COST and cw + w[t][j]< bestw) then   //判断约束条件和限界条件
            cc←cc + c[t][j]
            cw←cw + w[t][j]
            backtrack(t + 1)
            cc←cc - c[t][j]
            cw←cw - w[t][j]
```

5.8.3　实例构造

考虑 $n=3, m=3, c=7$ 的实例。部件的质量如表 5-2 所示，价格如表 5-3 所示。

表 5-2　部件的质量表

	供应商 1	供应商 2	供应商 3
部件 1	1	2	3
部件 2	3	2	1
部件 3	2	3	2

表 5-3　部件的价格表

C_{ij}	供应商 1	供应商 2	供应商 3
部件 1	1	2	3
部件 2	5	4	2
部件 3	2	1	2

注：行分别表示部件 1、2 和 3；列分别表示供应商 1、2 和 3；表中数据表示 w_{ij}：从供应商 j 处购得的部件 i 的质量；c_{ij} 表示从供应商 j 处购得的部件 i 的价格。

最小质量机器设计问题的搜索过程如图 5-50～图 5-55 所示(图中节点旁括号内的数据为已选择部件的重量之和、价格之和)。

从根节点 A 开始进行搜索，A 是活节点且是当前的扩展节点，如图 5-50(a)所示。扩展节点 A 沿 $x_1=1$ 分支扩展，cc=1≤c，满足约束条件；cw=1，bestw=∞，cw<bestw，满足限界条件。扩展生成的节点 B 成为活节点，并且成为当前的扩展节点，如图 5-50(b)所示。扩展节点 B 沿 $x_2=1$ 分支扩展，cc=6≤c，满足约束条件；cw=4，bestw=∞，cw<bestw，满足限界条件。扩展生成的节点 C 成为活节点，并且成为当前的扩展节点，如图 5-50(c)所示。扩展节点 C 沿 $x_3=1$ 分支扩展，cc=8>c，不满足约束条件，扩展生成的节点被剪掉。然后沿 $x_3=2$ 分支扩展，cc=7≤c，满足约束条件；cw=7，bestw=∞，cw<bestw，满足限界条件。扩展生成的节点 D 已经是叶子节点，找到了当前最优解，最优值为 7，将 bestw 修改为 7，如图 5-50(d)所示。

从叶子节点 D 回溯到活节点 C，活节点 C 再次成为当前的扩展节点。沿着它的 $x_3=3$ 分支扩展，不满足约束条件，扩展生成的节点被剪掉。继续回溯到活节点 B，节点 B 成为当前的扩展节点，如图 5-51(a)所示。扩展节点 B 沿 $x_2=2$ 分支扩展，cc=5≤c，满足约束条件；cw=3，bestw=7，cw<bestw，满足限界条件。扩展生成的节点 E 成为活节点，并且成为当前的扩展节点，如图 5-51(b)所示。扩展节点 E 沿 $x_3=1$ 分支扩展，cc=7≤c，满足约束条件；cw=5，bestw=7，cw<bestw，满足限界条件。扩展生成的节点 F 已经是叶子节点，找到了当前最优解，最优值为 5，将 bestw 修改为 5，如图 5-51(c)所示。

从叶子节点 F 回溯到活节点 E，沿着它的 $x_3=2,3$ 分支扩展，均不满足限界条件，扩展生成的节点被剪掉。继续回溯到活节点 B，节点 B 成为当前的扩展节点，如图 5-52(a)所示。

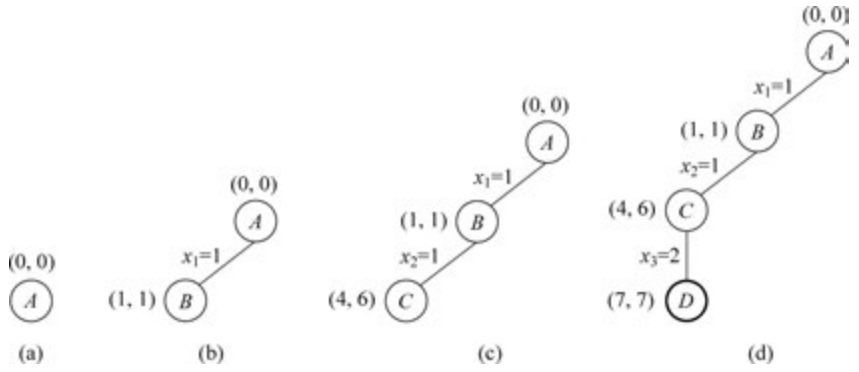

图 5-50　最小质量机器设计问题的搜索过程(一)

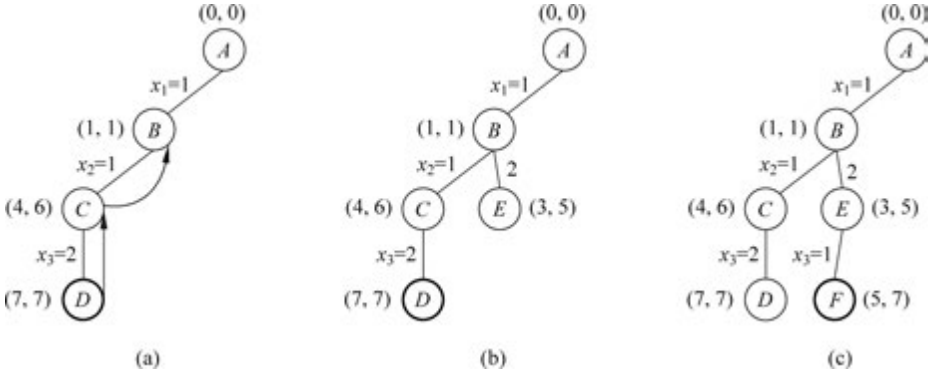

图 5-51　最小质量机器设计问题的搜索过程(二)

扩展节点 B 沿 $x_2=3$ 分支扩展，cc=3≤c，满足约束条件；cw=2，bestw=5，cw<bestw，满足限界条件。扩展生成的节点 G 成为活节点，并且成为当前的扩展节点，如图 5-52(b)所示。扩展节点 G 沿 $x_3=1$ 分支扩展，cc=5≤c，满足约束条件；cw=4，bestw=5，cw<bestw，满足限界条件。扩展生成的节点 H 已经是叶子节点，找到了当前最优解，其质量为 4，bestw 修改为 4，如图 5-52(c)所示。

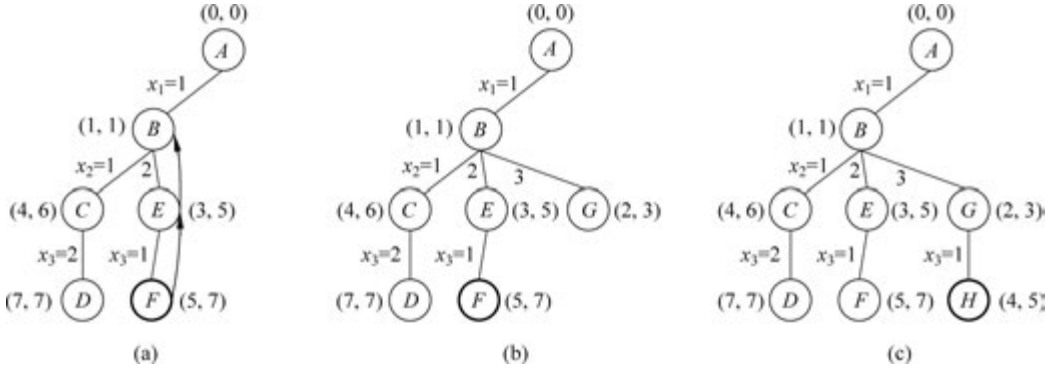

图 5-52　最小质量机器设计问题的搜索过程(三)

从叶子节点 H 回溯到活节点 G，沿着它的 $x_3=2$，3 分支扩展，均不满足限界条件，扩展生成的节点被剪掉。继续回溯到活节点 B，节点 B 的三个分支均搜索完毕，继续回溯到活

节点 A，节点 A 成为当前的扩展节点，如图 5-53(a)所示。扩展节点 A 沿 $x_1=2$ 分支扩展，cc=2≤c，满足约束条件；cw=2，bestw=4，cw<bestw，满足限界条件。扩展生成的节点 I 成为活节点，并且成为当前的扩展节点，如图 5-53(b)所示。

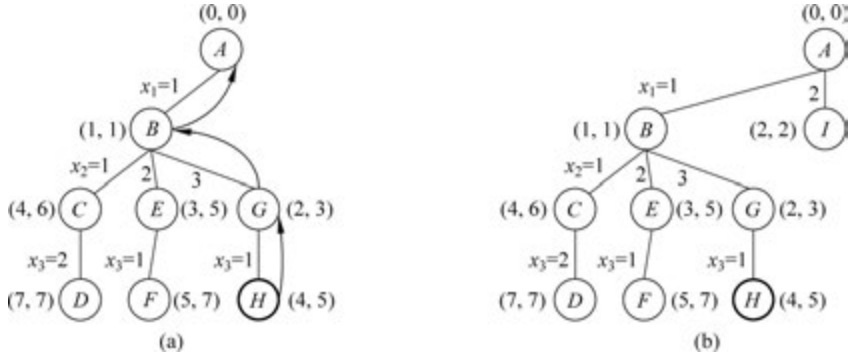

图 5-53　最小质量机器设计问题的搜索过程(四)

扩展节点 I 沿 $x_2=1,2$ 分支扩展，不满足限界条件，扩展生成的节点被剪掉。沿着 $x_2=3$ 分支扩展，cc=4≤c，满足约束条件；cw=3，bestw=4，cw<bestw，满足限界条件。扩展生成的节点 J 成为活节点，并且成为当前的扩展节点，如图 5-54(a)所示。扩展节点 J 沿 $x_3=1,2,3$ 分支扩展，均不满足限界条件，扩展生成的节点被剪掉。开始回溯到活节点 I。节点 I 的三个分支已搜索完毕，继续回溯到活节点 A，节点 A 再次成为扩展节点，如图 5-54(b)所示。

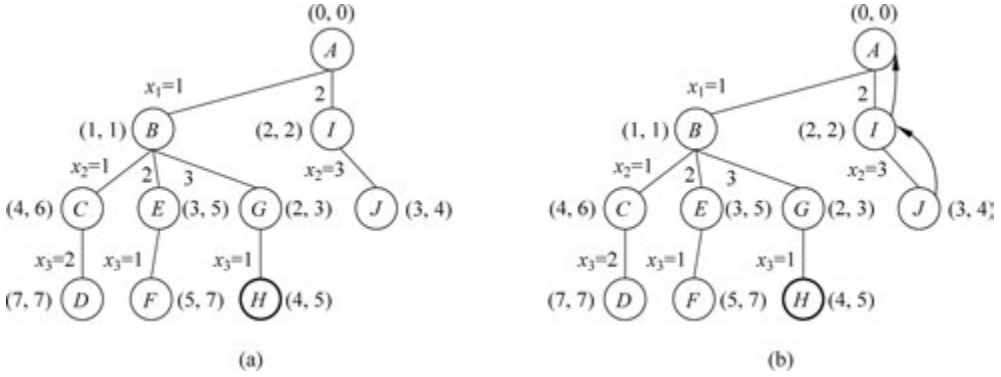

图 5-54　最小质量机器设计问题的搜索过程(五)

扩展节点 A 沿 $x_1=3$ 分支扩展，cc=3≤c，满足约束条件；cw=3，bestw=4，cw<bestw，满足限界条件。扩展生成的节点 K 成为活节点，并且成为当前的扩展节点，如图 5-55(a)所示。扩展节点 K 沿 $x_2=1,2,3$ 分支扩展，均不满足限界条件，扩展生成的节点被剪掉。开始回溯到活节点 A。此时，节点 A 的三个分支均搜索完毕，搜索结束。找到了问题的最优解为从根节点 A 到叶子节点 H 的路径(1,3,1)，最优值为 4，如图 5-55(b)所示。

5.8.4　算法分析

计算约束函数和限界函数需要 $O(1)$时间，需要判断约束函数和限界函数的节点在最坏情况下有 $1+m+m^2+m^3+\cdots+m^{n-1}=(m^n-1)/(m-1)$个，故耗时 $O(m^{n-1})$；在叶子

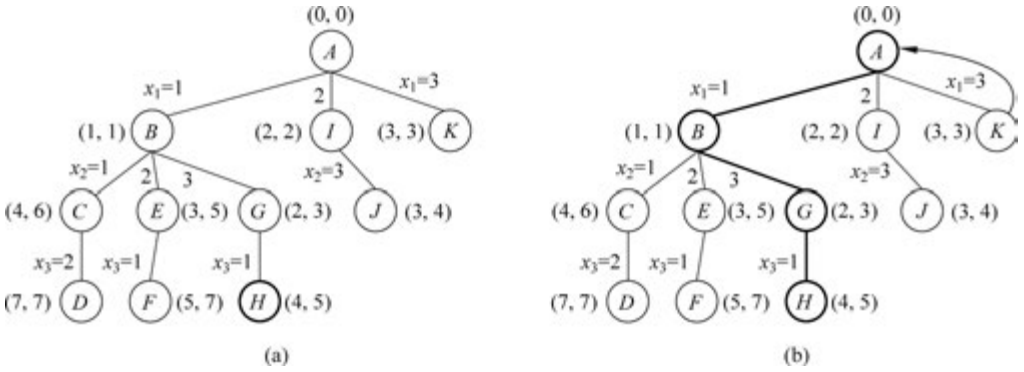

图 5-55 最小质量机器设计问题的搜索过程(六)

节点处记录当前最优方案耗时 $O(n)$,在最坏情况下会搜索到每个叶子节点,叶子节点有 m^n 个,故耗时为 $O(nm^n)$。最小质量机器设计问题的回溯算法所需的计算时间为 $O(m^{n-1})+O(nm^n)=O(nm^n)$。

5.8.5 Python 实践

定义一个深度优先搜索的 backtrack()函数,搜索最小质量机器设计方案,记录最优值及所需费用。其代码如下:

```
def backtrack(t):
    global bestw,cw,bestx,x,COST,cc,w,c,Total_cost
    if(t>n):
        Total_cost = cc
        bestw = cw
        bestx = x[:]
        return
    for j in range(1,m+1):
        x[t] = j
        if(cc+c[t][j]<=COST and cw+w[t][j]<bestw):
            cc+=c[t][j]
            cw+=w[t][j]
            backtrack(t+1)
            cc-=c[t][j]
            cw-=w[t][j]
```

Python 程序的入口是 main()函数。在 main()函数中,用二维列表 c 存储各供应商供应各部件的费用,二维列表 w 存储各供应商供应各部件的质量,牺牲 0 行 0 列位置的存储单元,下标从 1 开始有效。初始化相关辅助变量,调用 backtrack()函数,求最小质量机器设计方案,最后将结果输出到显示器上。输出结果中,0 号存储单元数据无效,从下标 1 开始有效。其代码如下:

```
if __name__ == "__main__":
    import sys
    COST = 7
    w = [[0,0,0,0],[0,1,2,3],[0,3,2,1],[0,2,3,2]]
    c = [[0,0,0,0],[0,1,2,3],[0,5,4,2],[0,3,1,2]]
```

```
n = 3
bestw = sys.maxsize
cw = 0
cc = 0
Total_cost = 0
m = 3
bestx = [0 for i in range(n + 1)]
x = [0 for i in range(n + 1)]
backtrack(1)
print("设计方案为:" ,bestx)
print("最优值为:" ,bestw)
print("所需费用为:" ,Total_cost)
```

输出结果为

```
设计方案为: [0, 1, 3, 1]
最优值为: 4
所需费用为: 6
```

习题 5