第 5 章

回溯算法及分支限界算法

学习目标

☑ 掌握回溯算法的算法框架；

☑ 理解回溯算法及分支限界算法的基本思想；

☑ 掌握回溯算法及分支限界算法的异同；

☑ 能够建立子集树、排列树及满 m 叉树模型；

☑ 通过实例学习，能够基于解空间模型，运用回溯算法及分支限界算法求解步骤解决实际问题。

5.1 回溯算法

回溯算法是在仅给出初始结点、目标结点及产生子结点的条件（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结点，从而找出问题的解。换言之，回溯算法从初始状态出发，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，尝试其他路径。通俗地讲，回溯算法是一种“能进则进，进不了则换，换不了则退”的基本搜索方法。


微课视频

5.1.1 回溯算法的算法框架及思想

1. 回溯算法的算法框架及思想

回溯算法是一种搜索方法。用回溯算法解决问题时，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，需要明确以下几方面：

（1）问题解的形式：回溯算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 $x_i(i=1,2,\cdots,n)$ 取值范围的限定。

（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，不同显约束相应解空间的大小就会不同，通常情况下，解空间越小，算法的搜索效率越高。

【例 5-1】 n 皇后问题。在 $n\times n$ 格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。换句话说，n 皇后问题等价于在 $n\times n$ 格的棋盘上放置 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，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。树的结点代表状态，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。

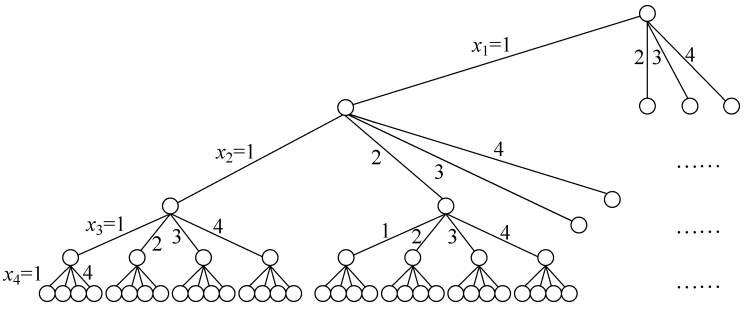


图 5-1　显约束为不同行的解空间树($n=4$)

最后，搜索问题的解空间树。在搜索的过程中，需要了解几个名词：

(1) 扩展结点：一个正在生成子树的结点称为扩展结点。

(2) 活结点：一个自身已生成但其子树还没有全部生成的结点称为活结点。

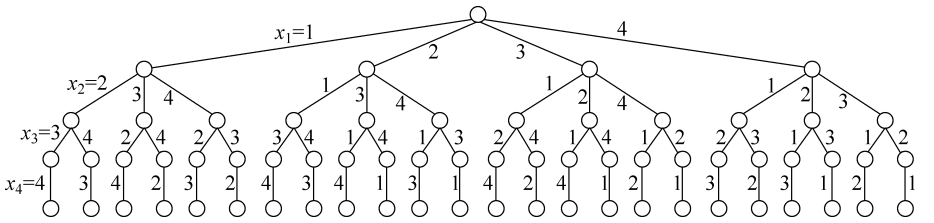


图 5-2　显约束为不同行且不同列的解空间树(n=4)

(3) 死结点：一个所有子树已经生成的结点称作死结点。

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

从回溯算法的搜索思想可知，搜索开始之前必须确定问题的隐约束。隐约束一般是考察解空间结构中的结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，就不用沿着该结点的分支继续搜索了，需要换到该结点的兄弟结点或回到上一层结点。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：一种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)；另一种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，回溯算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，回溯算法的算法框架主要包括 3 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. 回溯算法的构造实例

【例 5-2】　用回溯算法解决 4 皇后问题。

问题分析：按照回溯算法的搜索思想，首先确定根结点是活结点，并且是当前的扩展结点 R。它扩展生成一个新结点 C，如果 C 不满足隐约束，则舍弃；如果满足，则 C 成为活结点并成为当前的扩展结点，搜索继续向纵深处进行(此时结点 R 不再是扩展结点)。在完成对子树 C(以 C 为根的子树)的搜索之后，结点 C 变成了死结点。开始回溯到离死结点 C 最接近的活结点 R，结点 R 再次成为扩展结点。如果扩展结点 R 还存在未搜索过的孩子结点，则继续沿 R 的下一个未搜索过的孩子结点进行搜索；直到找到问题的解或者根结点变成死结点为止。

用回溯算法解决 4 皇后问题的求解过程设计如下：

步骤 1：定义问题的解空间。

设 4 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。

步骤 2：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。

步骤 3：搜索解空间。

步骤 3-1：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$|i-j|\neq|x_i-x_j|$。

步骤 3-2：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。

步骤 3-3：搜索过程如图 5-3～图 5-8 所示。根结点 A 是活结点，也是当前的扩展结点，如图 5-3(a)所示。扩展结点 A 沿着 $x_1=1$ 的分支生成孩子结点 B，结点 B 满足隐约束，B 成为活结点，并成为当前的扩展结点，如图 5-3(b)所示。扩展结点 B 沿着 $x_2=1$，$x_2=2$ 的分支生成的孩子结点不满足隐约束，舍弃；沿着 $x_2=3$ 的分支生成的孩子结点 C 满足隐约束，C 成为活结点，并成为当前的扩展结点，如图 5-3(c)所示。扩展结点 C 沿着所有分支生成的孩子结点均不满足隐约束，全部舍弃，活结点 C 变成死结点。开始回溯到离它最近的活结点 B，结点 B 再次成为扩展结点，如图 5-3(d)所示。

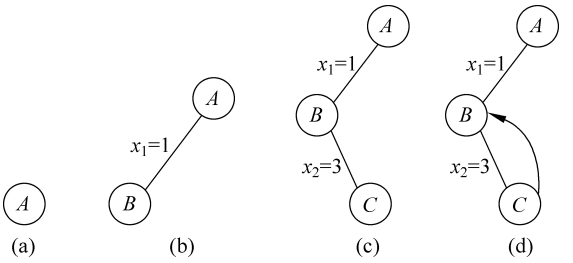


图 5-3　搜索过程 1

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

此时扩展结点 B 的孩子结点均搜索完毕，活结点 B 成为死结点。开始回溯到最近的活结点 A，结点 A 再次成为扩展结点，如图 5-5(a)所示。扩展结点 A 沿着 $x_1=2$ 的分支继续

生成的孩子结点 F 满足隐约束，结点 F 成为活结点，并成为当前的扩展结点，如图 5-5(b)所示。扩展结点 F 沿着 $x_2=1,2,3$ 的分支生成的孩子结点均不满足隐约束，全部舍弃；继续沿着 $x_2=4$ 的分支生成的孩子结点 G 满足隐约束，结点 G 成为活结点，并成为当前的扩展结点，如图 5-5(c)所示。

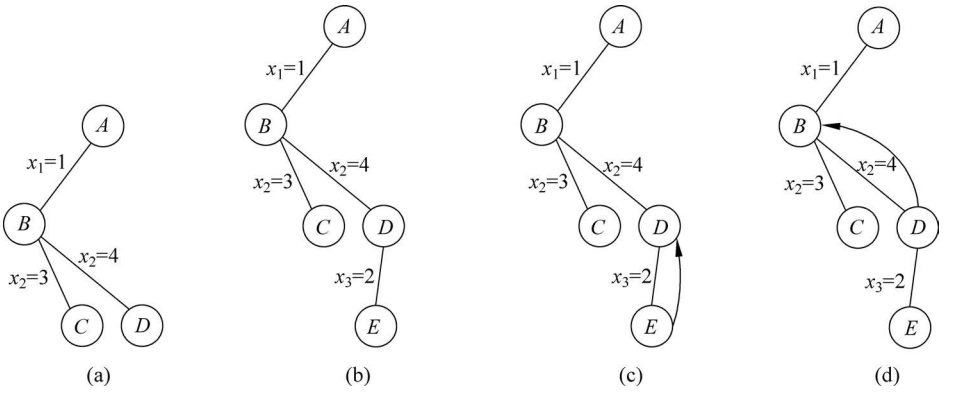


图 5-4　搜索过程 2

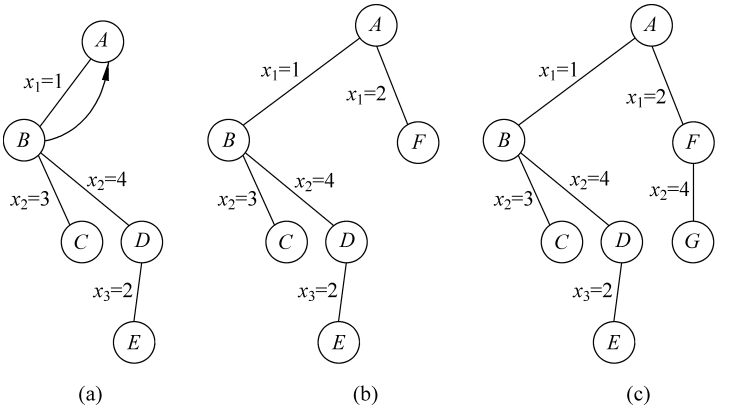


图 5-5　搜索过程 3

扩展结点 G 沿着 $x_3=1$ 的分支生成的孩子结点 H 满足隐约束，结点 H 成为活结点，并成为当前的扩展结点，如图 5-6(a)所示。扩展结点 H 沿着 $x_4=1,2$ 的分支生成的孩子结点均不满足隐约束，舍弃；沿着 $x_4=3$ 的分支生成的孩子结点 I 满足隐约束。此时搜索过程搜索到了叶子结点，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，从叶子结点 I 回溯到最近的活结点 H，H 又成为当前扩展结点，如图 5-6(c)所示。

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

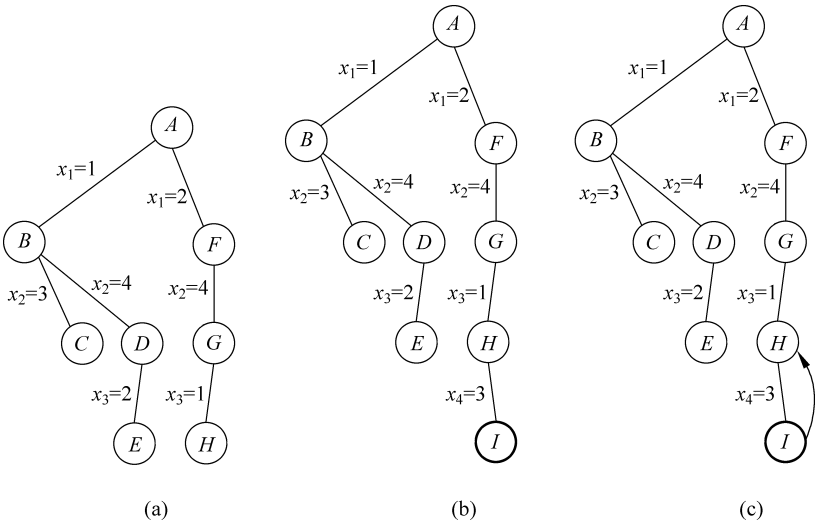


图 5-6 搜索过程 4

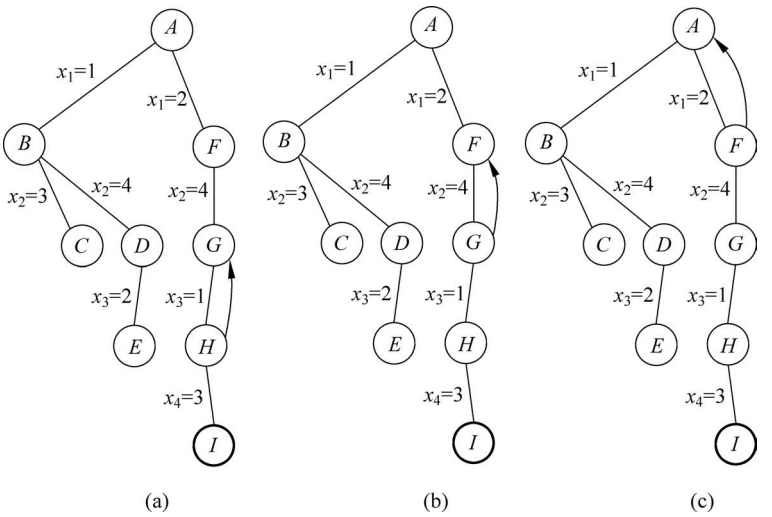


图 5-7 搜索过程 5

扩展结点 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。

通常将搜索过程中形成的树型结构称为问题的搜索树。例 5-2 4 皇后问题对应的搜索树如图 5-8 所示。简单地讲，搜索树上的结点全部是解空间树中满足隐约束的结点，而不满足隐约束的结点被全部剪掉。

对显约束为不同行且不同列的 4 皇后问题的解空间树(图 5-2)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。

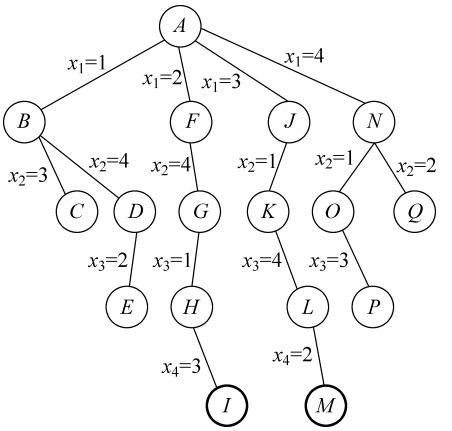


图 5-8　搜索过程 6

3. 回溯算法的算法描述模式

回溯算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。

具体描述如下。

```
void Backtrack(int t)
{
   if (t > n)          //搜索层次大于解空间树的深度,说明搜索到了叶子结点,找到了问题的一个解
     output(x);                                  //将找到的解记录输出
   else
      for (int i = s(n,t);i < = e(n,t);i++)     //检查扩展结点的每一个分支
        {
          x[t] = d(i);                           //将分支上的数据保存到记录当前解的数组 x 中
           if (constraint(t)&&bound(t))          //判断沿该分支生成的孩子结点是否满足隐约束
               Backtrack(t + 1);          //如果满足,则进入 t + 1 层的孩子结点继续搜索,递归实现
         }
}
```

这里，形参 t 代表当前扩展结点在解空间树中所处的层次。解空间树的根结点为第 1 层，根结点的孩子结点为第 2 层，以此类推，深度为 n 的解空间树的叶子结点为第 $n+1$ 层。注意：在解空间树中，结点所处的层次比该结点所在的深度大 1。解空间树中结点的深度与层次之间的关系如图 5-9 所示。

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结点的深度两个概念，树的深度指的是树中深度最大的结点深度。$s(n,t)$代表当前扩展结点处未搜索的子树的起始编号。$e(n,t)$代表当前扩展结点处未搜索的子树的终止编号。$d(i)$代表当前扩展结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)

代表当前扩展结点处的约束函数。bound(t)代表当前扩展结点处的限界函数。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。由于搜索从解空间树的根结点开始，即从第 1 层开始搜索，因此函数调用为 Backtrack(1)。

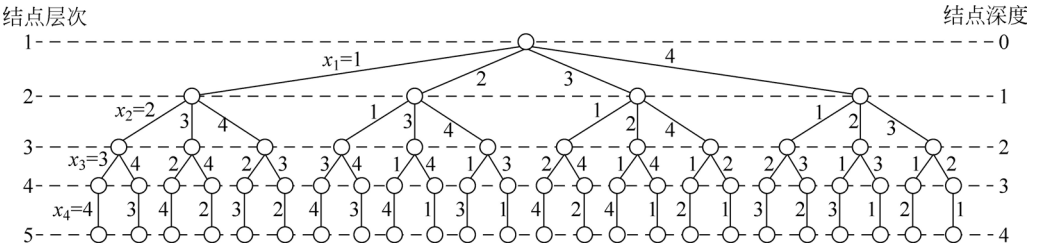


图 5-9　解空间树中结点的深度与层次关系

(2) 非递归算法描述模式。

具体描述如下：

```
void  NBacktrack()
{
  int t = 1;
  while (t > 0)
   {
    if (s(n,t)< = e(n,t))
      for (int i = s(n,t);i < = e(n,t);i++)
       {
        x[t] = d(i);
        if (constraint(t)&&bound(t))
          if (t > n)
              output(x);
          else
              t++;                          //更深一层搜索
       }                                    //end for
    else
        t--;                                //回溯到上一层的活结点
   }                                        //end while
}
```

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。

微课视频

5.1.2　子集树

微课视频

1. 概述

子集树是使用回溯算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,…,x_n)，分量 x_i(i=1,2,…,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。

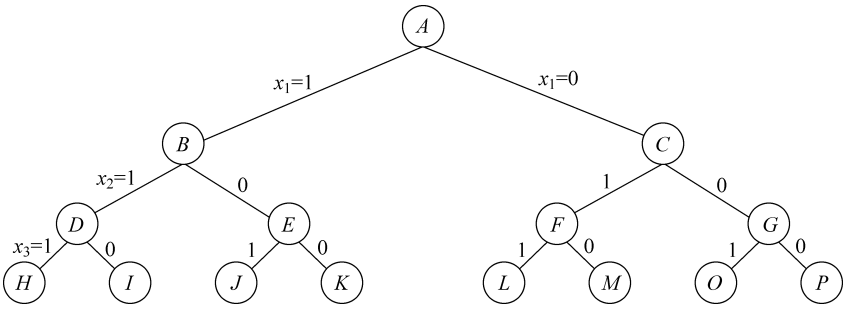


图 5-10　$n=3$ 时的子集树

子集树中所有非叶子结点均有左、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}，从根结点 A 到叶结点 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}。从根结点 A 到叶结点 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，树的根结点表示初始状态，中间结点表示某种情况下的中间状态，叶子结点表示结束状态。分支表示从一个状态过渡到另一个状态的行为。从根结点到叶子结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：

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。

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。

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，则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。

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

具体描述如下:

```
void Backtrack(int t)
{
  if (t>n)   output(x);
  if(constraint(t))                          //判断能否沿着扩展结点的左分支进行扩展
    {
    做相关标识;
    Backtrack(t+1);
    做相关标识的反操作;
  }
if(bound(t))                                 //判断能否沿着扩展结点的右分支进行扩展
  {
    做相关标识;
    Backtrack(t+1);
    做相关标识的反操作;
  }
}
```

这里,形式参数 t 表示扩展结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()为约束函数,bound()为限界函数。

3. 子集树的构造实例

【例 5-3】 0-1 背包问题。

(1) 问题描述:给定 n 种物品和一背包。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?

(2) 问题分析:根据问题描述可知,0-1 背包问题要求找出 n 种物品集合$\{1,2,3,\cdots,n\}$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$\{1,2,3,\cdots,n\}$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{1,2,3,…,n}的所有不超过背包容量的子集中物品总价值最大的。

按照回溯算法的算法框架，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，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，如何设置。

(3) 解题步骤。

步骤 1：定义问题的解空间。

0-1 背包问题是要将物品装入背包，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，($i=1,2,\cdots,n$)。

步骤 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 5-11 描述了 $n=4$ 时的解空间树。

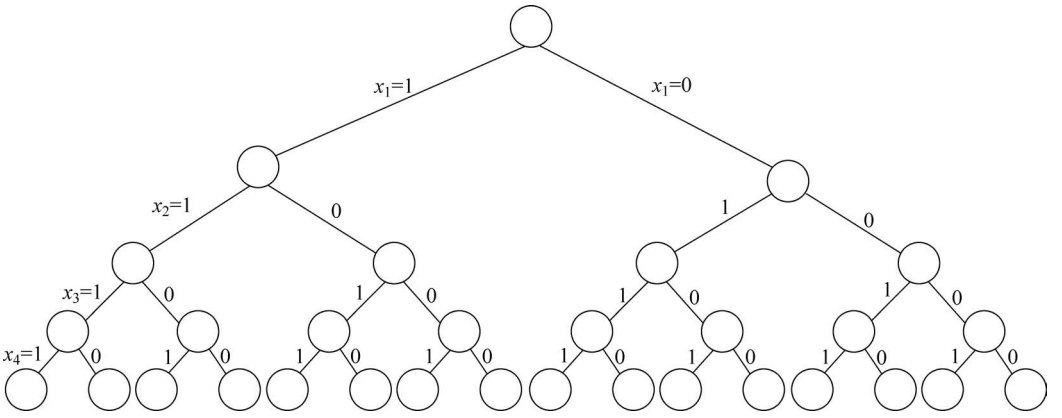


图 5-11　$n=4$ 时的解空间树

步骤 3：搜索解空间。

步骤 3-1：是否需要约束条件？如果需要，如何设置？

0-1 背包问题的解空间包含 2^n 个可能的解，是不是每一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，为不可行解；否则为可行解。搜索过程将不再搜索那些导

致不可行解的结点及其孩子结点。约束条件的形式化描述为

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

步骤 3-2：是否需要限界条件？如果需要，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，任何一个中间结点 z（中间状态）均表示从根结点到该中间结点的分支所代表的行为已经确定，从 z 到其子孙结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 的哪一个分支进行扩展，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结点 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)，则说明从中间结点 z 继续向子孙结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 的子孙结点搜索。因此，限界条件可描述为

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

步骤 3-3：搜索过程。从根结点开始，以深度优先的方式进行搜索。根结点首先成为活结点，也是当前的扩展结点。由于子集树中约定左分支上的值为“1”，因此沿着扩展结点的左分支扩展，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，即进入左孩子结点，左孩子结点成为活结点，并且是当前的扩展结点，继续向纵深结点扩展；如果不成立，则剪掉扩展结点的左分支，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。如果限界条件满足，说明有可能导致最优解，即进入右分支，右孩子结点成为活结点，并成为当前的扩展结点，继续向纵深结点扩展；如果不满足限界条件，则剪掉扩展结点的右分支，开始向最近的活结点回溯。搜索过程直到所有活结点变成死结点结束。

(4) 0-1 背包问题实例的搜索过程演示。

令 $n=4, W=7, w=(3,5,2,1), v=(9,10,7,4)$。搜索过程如图 5-12～图 5-16 所示。(注：图中结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，搜索从根结点开始，即根结点是活结点，也是当前的扩展结点。它代表初始状态，即背包是空的，如图 5-12(a)所示。扩展结点 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，因此结点 2 成为活结点，并成为当前的扩展结点。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 5-12(b)所示。扩展结点 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，故剪掉扩展结点 2 的左分支。此时，需要选择扩展结点 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，则结点 2 沿右分支扩展的结点 3 成为活结点，并成为当前的扩展结点。扩展结点 3 代表背包剩余容量为 4，背包内物品的总价值为 9，如图 5-12(c)所示。以此类推，扩展结点 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，结点 4 成为活结点，并成为当前的扩展结点。结点 4 代表背包剩余容量为 2，背包内物品总价值为 16，如图 5-12(d)所示。

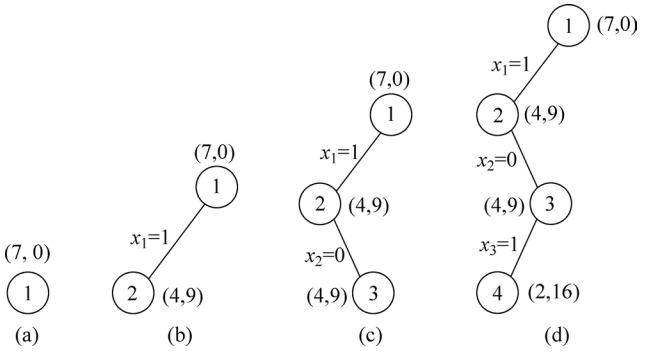


图 5-12　搜索过程 1

扩展结点 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，结点 5 满足约束条件。结点 5 已是叶子结点，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 5-13(a)所示。由于结点 5 已是叶子结点，不具备扩展能力，此时要回溯到离结点 5 最近的活结点 4，结点 4 再次成为扩展结点，如图 5-13(b)所示。扩展结点 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，故剪掉结点 4 的右分支。扩展结点 4 的左右两个分支均搜索完毕，回溯到最近的活结点 3，结点 3 再次成为扩展结点，如图 5-13(c)所示。扩展结点 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，故剪掉结点 3 的右分支。扩展结点 3 的左右两个分支均搜索完毕，回溯到最近的活结点 2，结点 2 再次成为扩展结点。扩展结点 2 的两个分支均搜索完毕，故继续回溯到结点 1，如图 5-13(d)所示。

扩展结点 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，则扩展的结点 6 成为活结点，并成为当前的扩展结点，如图 5-14(a)所示。扩展结点 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，第二种物品的重量为 5，5＜7，满足约束条件，扩展生成的结点 7 成为活结点，且是当前的扩展结点。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 5-14(b)所示。扩展结点 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，扩展生成的结点 8 成为活结点，且是当前的扩展结点。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 5-14(c)所示。

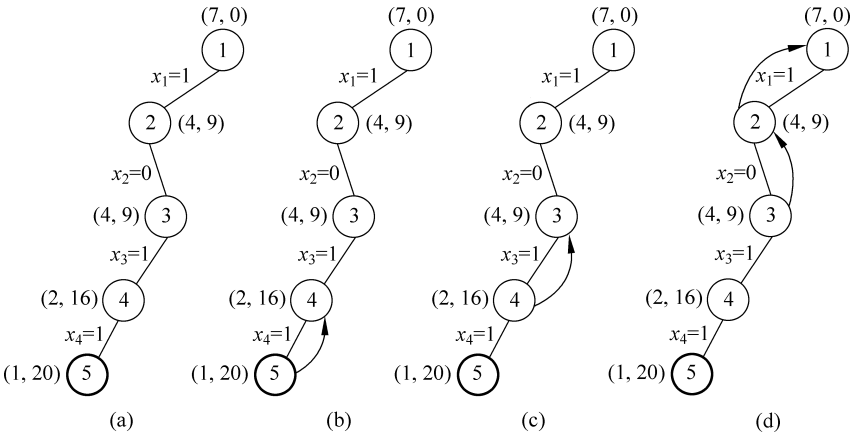


图 5-13 搜索过程 2

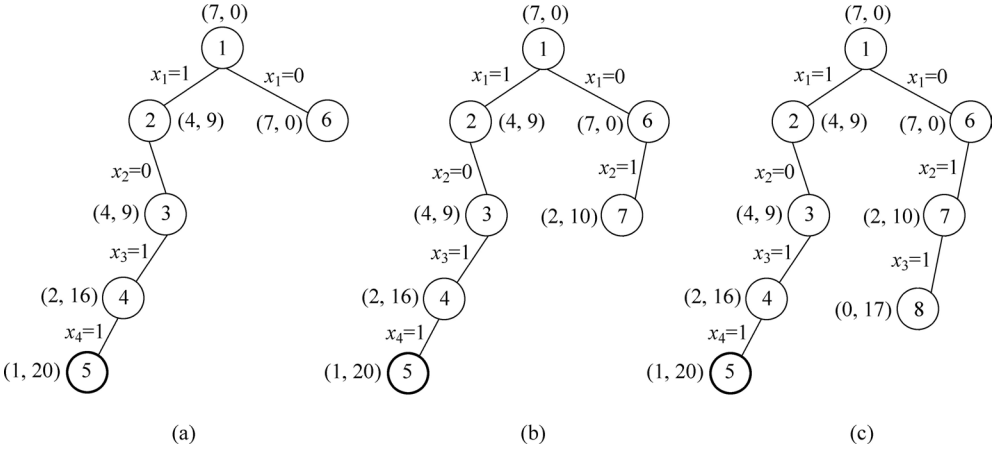


图 5-14 搜索过程 3

扩展结点 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,扩展生成的结点被剪掉。接下来沿着扩展结点 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结点也被剪掉。扩展结点 8 的所有分支均搜索完毕,回溯到最近的活结点 7,结点 7 又成为扩展结点,如图 5-15(a)所示。扩展结点 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,扩展生成的结点被剪掉。扩展结点 7 的所有分支均搜索完毕,回溯到活结点 6,结点 6 又成为扩展结点,如图 5-15(b)所示。扩展结点 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,扩展生成的结点被剪掉。扩展结点 6 的所有分支均搜索完毕,回溯到活结点 1,结点 1 又成为扩展结点,如图 5-15(c)所示。

扩展结点 1 的两个分支均搜索完毕,它成为死结点,搜索过程结束,找到的问题的解为从根结点 1 到叶子结点 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 5-16 所示。

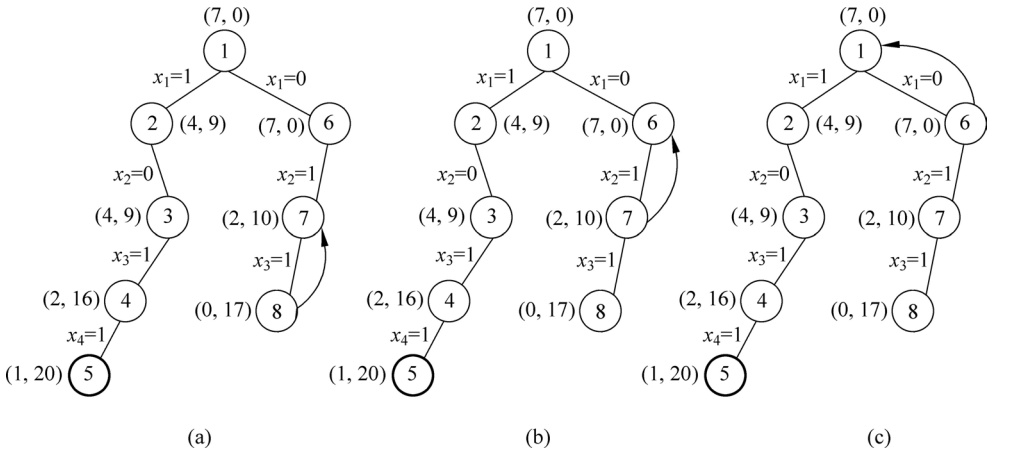


图 5-15　搜索过程 4

(5) 限界条件式(5-2)的优化。

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 种物品的最大价值(用 brp 表示)肯定小于或等于 rp，用 brp 取代 rp，则式(5-2)改写为

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

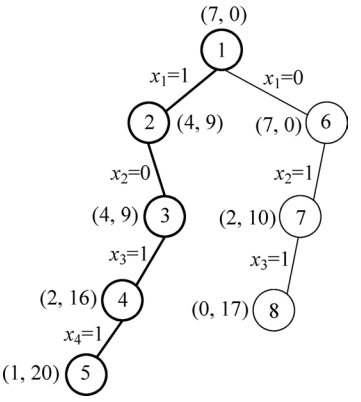


图 5-16　搜索过程 5

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结点 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 成立的可能性要小。用 cp+brp>bestp 作为限界条件，从中间结点 z 沿右分支继续向纵深搜索的可能性就小。也就是说，中间结点 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 5-12(b)中，扩展结点 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 种、第 4 种物品，背包剩余容量为 4，将背包装满装入的最大价值为第 3 种、第 4 种物品的价值之和，即 brp=11，bestp=0，cp+brp>bestp，限界条件成立，扩展的结点 3 成为活结点，并成为当前的扩展结点，继续向纵深处扩展。进行式(5-3)限界条件的搜索与式(5-2)限界条件的搜索直到图 5-13(b)(找到一个当前最优解后回溯到最近的活结点 4)均相同，其后在

式(5-3)限界条件下的搜索过程如图 5-17 所示。

扩展结点 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,扩展生成的结点被剪掉。此时,左右分支均检查完毕,开始回溯到活结点 3,结点 3 又成为扩展结点,如图 5-17(a)所示。扩展结点 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,扩展生成的结点被剪掉。此时,结点 3 的左右分支均搜索完毕,回溯到活结点 2。结点 2 的两个分支已搜索完毕,继续回溯到活结点 1,活结点 1 再次成为扩展结点,如图 5-17(b)所示。扩展结点 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 4/5,那么装进去的价值为 10×4/5=8,故 brp=7+4+8=19,cp+brp=19<bestp(20),限界条件不满足,扩展生成的结点被剪掉。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 5-17(c)所示。

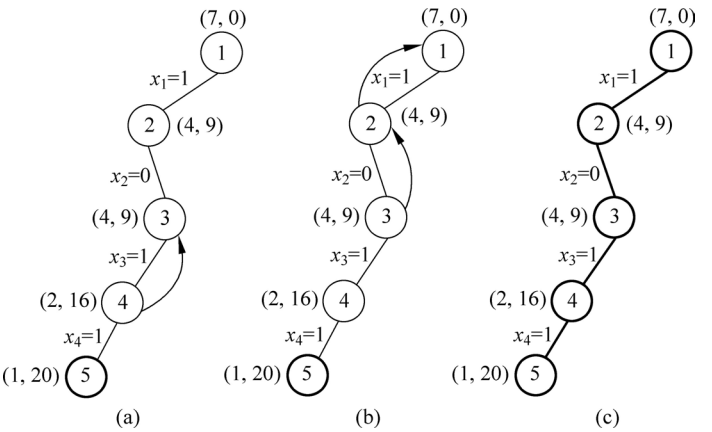


图 5-17 搜索过程

(6) 算法描述。

以下算法描述针对式(5-3)的限界条件。形式参数 t 代表当前扩展结点在解空间树中所处的层次。首先定义一个 Knap 类:

```
class Knap
{
public:
friend int Knapsack(int p[],int w[],int c,int n);
     void print()
       {
         for(int k=1;k<=n;k++)
             cout<<bestx[k]<<" ";
```

```
            cout << endl;
        }
private:
  int Bound(int i);
  void Backtrack(int t);
  int c;                                    //背包容量
  int n;                                    //物品数
  int * w;                                  //物品重量数组
  int * p;                                  //物品价值数组
  int cw;                                   //当前重量
  int cp;                                   //当前价值
  int bestp;                                //当前最优值
  int * bestx;                              //当前最优解
  int * x;                                  //当前解
};
```

类 Knap 的成员函数 Bound()用于求将剩余的物品装满剩余的背包容量时装入背包物品的最大价值。用参数 i 表示剩余物品为第 $i\sim n$ 种物品，成员函数 Bound()基于物品按照单位重量的价值由大到小排好序的序列。成员函数 Bound()的描述如下：

```
int Knap::Bound(int i)                      //计算上界
{
    int cleft = c - cw;                     //剩余容量
    int b = cp;
    //以物品单位重量价值递减序装入物品
    while(i <= n && w[i]<= cleft)
      {
          cleft -= w[i];
          b += p[i];i++;}
    //装满背包
    if(i <= n)
       b += p[i]/w[i] * cleft;
    return b;
}
```

类 Knap 的成员函数 Backtrack()用于搜索解空间树，参数 t 表示当前扩展结点在解空间树中所处的层次。函数 Backtrack()搜索解空间时，先判断是否达到叶子结点，如果到达叶子结点(即 $t>n$)，说明找到了一个当前最优解，将其记录；否则，如果没有到达叶子结点，则沿左子树扩展，此时判断是否满足约束条件，如果满足，即进行更深一层的搜索(即递归更深一层)；如果不满足，则沿着右子树扩展，此时判断是否满足限界条件，如果满足，即进行更深一层的搜索，反之则回溯到最近的活结点。算法描述如下：

```
void Knap::Backtrack(int t)
{
  if(t > n)                                 //到达叶子结点
```

```
    {
            for(int j = 1;j <= n;j++)
                 bestx[j] = x[j];
            bestp = cp;
            return;
    }
   if(cw + w[t]<= c)                          //搜索左子树
    {
        x[t] = 1;
        cw += w[t];
        cp += p[t];
        Backtrack(t + 1);
        cw -= w[t];
        cp -= p[t];
     }
   if(Bound(t + 1)> bestp)                    //搜索右子树
     { x[t] = 0; Backtrack(t + 1); }
}
```

用回溯算法求解 0-1 背包问题时，数组 x 用于记录当前解，其元素全部初始化为 0。数组 w 用于存储物品的重量，数组 p 用于存储物品的价值，背包的容量用 c 表示。为了对物品按照单位重量的价值由大到小排序，定义了 Object 类，具体定义如下：

```
class Object
{ public:
    friend int Knapsack(int p[],int w[],int c,int n);
    int operator <= (Object a)const
      { return (d >= a.d); }
  private:
    int id;                                   //物品编号
    float d;                                  //单位重量的价值
};
```

0-1 背包问题的算法首先进行初始化工作，然后对物品类对象按照单位重量的价值由大到小排序，算法描述如下：

```
int Knapsack(int p[],int w[],int c,int n)
 {                                           //初始化
    int W = 0;int P = 0;int i = 1;
    Object * Q = new Object[n];
    for(i = 1;i <= n;i++)
     {
       Q[i - 1].ID = i;
       Q[i - 1].d = 1.0 * p[i]/w[i];
       P += p[i]; W += w[i];
     }
    if(W <= c)   return P;                   //装入所有物品
```

```
    //依物品单位重量降序排序
    Sort(Q,n);                              //将数组 Q 中的元素按照单位重量的价值由大到小排序
    Knap  K;
    K.p = new int[n + 1];
    K.w = new int[n + 1];
    K.x = new int[n + 1];
    K.bestx = new int[n + 1];
   K.x[0] = 0;
   K.bestx[0] = 0;
    for(i = 1;i < = n;i++)                  //按单位重量的价值降序排列物品重量和价值
      {
        K.p[i] = p[Q[i - 1].id];
        K.w[i] = w[Q[i - 1].id];
      }
        K.cp = 0;
        K.cw = 0;
        K.c = c;
        K.n = n;
        K.bestp = 0;
  //回溯搜索
  K.Backtrack(1);                           //从根开始搜索解空间树
  K.print();                                //输出最优解
  delete [] Q;delete [] K.w;delete [] K.p;
  return K.bestp;                           //返回最优值
}
```

(7) 算法分析。

判断约束函数需要耗时 $O(1)$，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 $O(2^n)+O(n2^n)=O(n2^n)$。

(8) C++实战。

相关代码如下。

```
#include < iostream >
#include < algorithm >
using namespace std;
class Knap
{
    public:
    friend int Knapsack(int p[],int w[],int c,int n);
        void print()
          {
              for(int k = 1;k < = n;k++)
                  cout << bestx[k]<<" ";
              cout << endl;
          }
```

```
    private:
      int Bound(int i);
      void Backtrack(int t);
      int c;                                  //背包容量
      int n;                                  //物品数
      int * w;                                //物品重量数组
      int * p;                                //物品价值数组
      int cw;                                 //当前重量
      int cp;                                 //当前价值
      int bestp;                              //当前最优值
      int * bestx;                            //当前最优解
      int * x;                                //当前解
};
int Knap::Bound(int i)                        //计算上界
{
    int cleft = c - cw;                       //剩余容量
    int b = cp;
    //以物品单位重量价值递减序装入物品
    while(i <= n && w[i]<= cleft)
    {
        cleft -= w[i];
        b += p[i];i++;
    }
    //装满背包
    if(i <= n)
        b += p[i]/w[i] * cleft;
    return b;
}

void Knap::Backtrack(int t)
{
  if(t > n)                                   //到达叶子结点
   {
          for(int j = 1;j <= n;j++)
                bestx[j] = x[j];
          bestp = cp;
          return;
   }
  if(cw + w[t]<= c)                           //搜索左子树
   {
        x[t] = 1;
        cw += w[t];
        cp += p[t];
        Backtrack(t + 1);
        cw -= w[t];
        cp -= p[t];
    }
    if(Bound(t + 1)> bestp)                   //搜索右子树
    {
        x[t] = 0;
```

```
        Backtrack(t + 1);
    }
}

class Object
{
    public:
        friend int Knapsack(int p[ ],int w[ ],int c,int n);
        int operator <(const Object &a)
        { return (this -> d > a.d); }
    private:
        int id;                          //物品编号
        float d;                         //单位重量的价值
};

int Knapsack(int p[ ],int w[ ],int c,int n)
{ //初始化
    int W = 0;int P = 0;int i = 1;
    Object * Q = new Object[n];
    for(i = 1;i <= n;i++)
     {
        Q[i - 1].id = i;
        Q[i - 1].d = 1.0 * p[i]/w[i];
        P += p[i];
        W += w[i];
     }
    if(W <= c) return P;                 //装入所有物品
    for(int i = 0;i <= n;i++)
        cout << Q[i].d <<" ";
    cout << endl;
    //依物品单位重量降序排序
    sort(Q,Q + n);                       //将数组 Q 中的元素按照单位重量的价值由大到小排序
    for(int i = 0;i <= n;i++)
        cout << Q[i].d <<" ";
    cout << endl;
    Knap K;
    K.p = new int[n + 1];
    K.w = new int[n + 1];
    K.x = new int[n + 1];
    K.bestx = new int[n + 1];
    int * bestx = new int[n + 1];
    K.x[0] = 0;
    K.bestx[0] = 0;
    for(i = 1;i <= n;i++)                //按单位重量的价值降序排列物品重量和价值
    {
        K.p[i] = p[Q[i - 1].id];
        K.w[i] = w[Q[i - 1].id];
    }
    K.cp = 0;
    K.cw = 0;
```

```
    K.c = c;
    K.n = n;
    K.bestp = 0;
    //回溯搜索
    K.Backtrack(1);                    //从根开始搜索解空间树
    K.print();                         //输出最优解
    cout <<"装入的物品编号为: ";
    for(int i = 1;i <= n;i++)
        if(K.bestx[i])
            cout << Q[i - 1].id <<" ";
    cout << endl;
    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    delete [] K.x;
    delete [] K.bestx;
    delete [] bestx;
    return K.bestp;                    //返回最优值
}
int main(){
    int p[] = {0,9,10,7,4};
    int w[] = {1,3,5,2,1};
    int c = 7;
    int n = 4;
    int bestp = Knapsack(p,w,c,n);
    cout <<"最大价值为: "<< bestp << endl;
    return 0;
}
```

【例 5-4】 最大团问题。

(1) 问题描述。

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。

(2) 问题分析。

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$\{1,2,3,\cdots,n\}$的一部分顶点 V',即 n 个顶点集合$\{1,2,3,\cdots,n\}$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$\{1,2,3,\cdots,n\}$所有同类子集中包含顶点个数最多的。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。

(3) 解题步骤。

步骤 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。

步骤 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。

步骤 3：搜索解空间。

步骤 3-1：确定是否需要约束条件，如果需要，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判断是否有可能导致问题的可行解。

假设当前扩展结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。接下来沿着扩展结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 个顶点与前 $t-1$ 个顶点中在团里的那些顶点有边相连，则能放入团中，否则，就不能放入团中。因此，约束函数描述如下：

```
Bool Place(int t)
 {
  Bool OK = true;
  for (int j = 1; j < t; j++)
      if (x[j] && a[t][j] == 0)          //顶点 t 与顶点 j 不相连
          {
            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。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。

步骤 3-2：确定是否需要限界条件？如果需要，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。假设当前的扩展结点为 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 的哪一个分支进行扩展，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的顶点个数（用 cn 表示），假设从第 $t+1$ 个顶点到第 n 个顶点全部放入团，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结点 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),则说明从中间结点 z 继续向子孙结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 的子孙结点搜索。因此,限界条件可描述为:cn+fn>bestn。

步骤 3-3:搜索过程。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。以图 5-18 给定的无向图为例,按照 0-1 背包问题的搜索过程形成的搜索树如图 5-19 所示。找到问题的解是从根结点 A 到叶子结点 N 的路径(0,1,1,1,1),已在图 5-19 中用粗线画出,求得的最大团如图 5-20 所示。

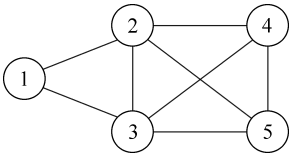


图 5-18 无向图

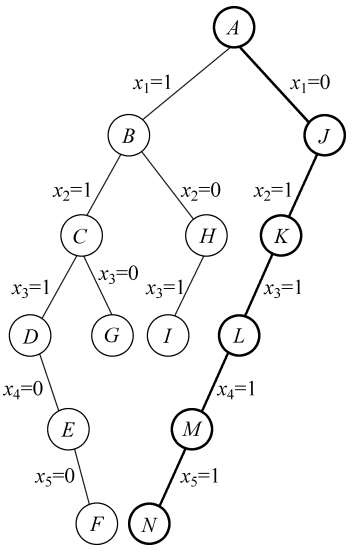


图 5-19 搜索树

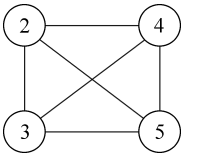


图 5-20 最大团

(4) 算法描述。

令一维数组 x 记录当前解,一维数组 bestx 记录当前最优解,变量 cn、bestn 分别记录当前已包含在团里的顶点个数和当前最优解包含的在团里的顶点个数,初始时均为 0。找出最大团的关键是判断当前顶点是否能够放入团里的约束条件,如果能放入团,就进行更深一层搜索;否则判断限界函数是否满足,如果满足,则进行更深一层搜索,反之,回溯到最近的活结点。回溯搜索的算法描述如下:

```
void Backtrack(int t)
 {
    if (t > n)                          //到达叶结点
       {
       for(int i = 1;i < = n;i++)
          bestx[i] = x[i];
       bestn = cn;
       return;
       }
    if (Place(t))                       //进入左子树
```

```
    {
        x[t] = 1;
        cn++;
        Backtrack(t + 1);
        cn -- ;
    }
  if (cn + n - t > bestn)                    //进入右子树
     {
       x[t] = 0;
       Backtrack(t + 1);
     }
}                                            //end Backtrack
```

搜索解空间树时要从根结点开始，以深度优先的方式进行。故初始化工作完成后，只需要调用 Backtrack(1)便可求得最大团。

(5) 算法分析。

判断约束函数需耗时 $O(n)$，在最坏情况下，有 2^n-1 个左子树，约束函数耗时最坏为 $O(n2^n)$。判断限界函数需要 $O(1)$时间，在最坏情况下，有 2^n-1 个右孩子结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 $O(2^n)+O(n2^n)=O(n2^n)$。

(6) C++实战。

相关代码如下。

```
#include < iostream >
using namespace std;
class Max_Clique{
    public:
        Max_Clique(int n, int cn, int bestn)
        {
            this -> n = n;
            this -> cn = cn;
            this -> bestn = bestn;
        }
        void Backtrack(int t);
        void print()
        {
            cout <<"最优解为: ";
            for(int i = 1; i < = n; i++)
                cout << bestx[i]<<" ";
            cout << endl;
        }
        bool place(int t);
        int * x;
        int ** a;
        int * bestx;
        int n;
```

```
        int cn;
        int bestn;
};
bool Max_Clique::place(int t)
{
    bool OK = true;
    for (int j = 1; j < t; j++)
        if (x[j] && a[t][j] == 0)          // 顶点 t 与顶点 j 不相连
        {
            OK = false;
            break;
        }
    return OK;
}
void Max_Clique::Backtrack(int t)
{
    if (t > n)                             // 到达叶结点
    {
        for(int i = 1;i <= n;i++)
            bestx[i] = x[i];
        bestn = cn;
        return;
    }
    if (place(t))                          // 进入左子树
    {
        x[t] = 1;
        cn++;
        Backtrack(t + 1);
        cn--;
    }
    if (cn + n - t > bestn)                // 进入右子树
    {
        x[t] = 0;
        Backtrack(t + 1);
    }
}//end Backtrack
int main(){
    cout <<"请输入图的顶点数 n:";
    int n;
    cin >> n;
    Max_Clique clique(n,0,0);
    clique.bestx = new int[n + 1];
    clique.x = new int[n + 1];
    clique.a = new int * [n + 1];
    for(int i = 0;i <= n;i++)
    {
        clique.a[i] = new int [n + 1];
    }
    for(int i = 0;i <= n;i++){
        for(int j = 0;j <= n;j++)
```

```
            cin >> clique.a[i][j];
        }
        clique.Backtrack(1);
        clique.print();
        cout <<"最大团顶点个数为: "<< clique.bestn << endl;
        delete [ ]clique.bestx;
        delete [ ]clique.x;
        delete [ ] clique.a;
    }
```

5.1.3　排列树


微课视频


微课视频


微课视频

1. 概述

排列树是用回溯算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。n 个元素组成的集合为 $S=\{1,2,\cdots,n\}$，$x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}$，$i=1,2,\cdots,n$。

$n=3$ 时的排列树如图 5-21 所示。

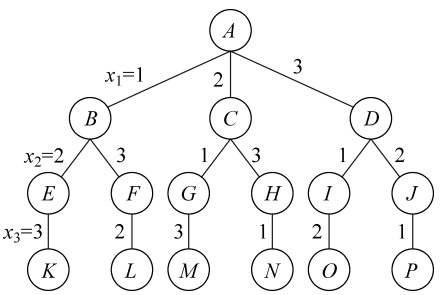


图 5-21　$n=3$ 的排列树

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$\{1,2,3\}$，从根结点 A 到叶结点 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；从根结点 A 到叶结点 M 的路径描述的一个排列为(2,1,3)；从根结点 A 到叶结点 P 的路径描述的一个排列为(3,2,1)。

在排列树中，树的根结点表示初始状态(所有位置全部没有放置元素)；中间结点表示某种情况下的中间状态(中间结点之前的位置上已经确定了元素，中间结点之后的位置上还没有确定元素)；叶子结点表示结束状态(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)；从根结点到叶子结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，例如：

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。

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。

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。

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，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。

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。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

具体描述如下：

```
void Backtrack(int t)
 {
    if (t > n) output(x);
    else
      for (int i = t;i < = n;i++)
      {
          swap(x[t], x[i]);
          if (constraint(t)&&bound(t))
             Backtrack(t + 1);
          swap(x[t], x[i]);
       }
}
```

这里，形式参数 t 表示扩展结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()为约束函数，bound()为限界函数，swap()函数实现两个元素位置的交换。

3．排列树的构造实例

【例 5-5】 批处理作业调度问题。

(1) 问题描述。

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，制定出最佳作业调度方案，使其

完成时间和达到最小。

(2) 问题分析。

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。按照回溯算法的算法框架,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,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,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。

(3) 解题步骤。

步骤 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 $S=\{1,2,\cdots,n\}$,$x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}$,$i=1,2,\cdots,n$。

步骤 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。

步骤 3:搜索解空间。

步骤 3-1:由于不需要约束条件,故无须设置。

步骤 3-2: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$+\infty$。

步骤 3-3:搜索过程。扩展结点沿着某个分支扩展时需要判断限界条件,如果满足,则进入深一层继续搜索;如果不满足,则将扩展生成的结点剪掉。搜索到叶子结点时,即找到当前最优解。搜索过程直到全部活结点变成死结点为止。

(4) 批处理作业调度问题的构造实例。

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。

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

搜索过程如图 5-22～图 5-28 所示：从根结点 A 开始，结点 A 成为活结点，并且是当前的扩展结点，如图 5-22(a)所示。扩展结点 A 沿着 $x_1=1$ 的分支扩展，$F_{11}=2$，$F_{21}=3$，故 cf=3，bestf=$+\infty$，cf<bestf，限界条件满足，扩展生成的结点 B 成为活结点，并且成为当前的扩展结点，如图 5-22(b)所示。扩展结点 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=$+\infty$，cf<bestf，限界条件满足，扩展生成的结点 E 成为活结点，并且成为当前的扩展结点，如图 5-22(c)所示。扩展结点 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=$+\infty$，cf<bestf，限界条件满足，扩展生成的结点 K 是叶子结点。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 5-22(d)所示。

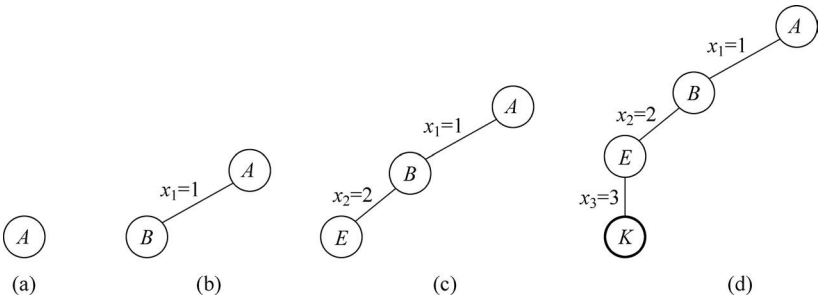


图 5-22　搜索过程 1

叶子结点 K 不具备扩展能力，开始回溯到活结点 E。结点 E 只有一个分支，且已搜索完毕，因此结点 E 成为死结点，继续回溯到活结点 B，结点 B 再次成为扩展结点，如图 5-23(a)所示。扩展结点 B 沿着 $x_2=3$ 的分支扩展，cf=10，bestf=19，cf<bestf，限界条件满足，扩展生成的结点 F 成为活结点，并且成为当前的扩展结点，如图 5-23(b)所示。扩展结点 F 沿着 $x_3=2$ 的分支扩展，cf=18，bestf=19，cf<bestf，限界条件满足，扩展生成的结点 L 是叶子结点。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 5-23(c)所示。从叶子结点 L 开始回溯到活结点 F。结点 F 的一个分支已搜索完毕，结点 F 成为死结点，回溯到活结点 B。结点 B 的两个分支已搜索完毕，回溯到活结点 A，结点 A 再次成为扩展结点，如图 5-23(d)所示。

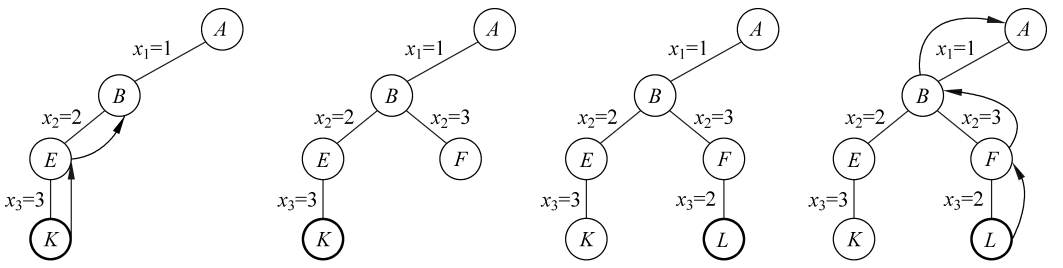


图 5-23　搜索过程 2

扩展结点 A 沿着 $x_1=2$ 的分支扩展，cf=4，bestf=18，cf<bestf，限界条件满足，扩展生成的结点 C 成为活结点，并成为当前的扩展结点，如图 5-24(a)所示。扩展结点 C 沿着 $x_2=1$ 的分支扩展，cf=10，bestf=18，cf<bestf，限界条件满足，扩展生成的结点 G 成为活结点，并成为当前的扩展结点，如图 5-24(b)所示。扩展结点 G 沿着 $x_3=3$ 的分支扩展，cf=20，bestf=18，cf>bestf，限界条件不满足，扩展生成的结点被剪掉，如图 5-24(c)所示。

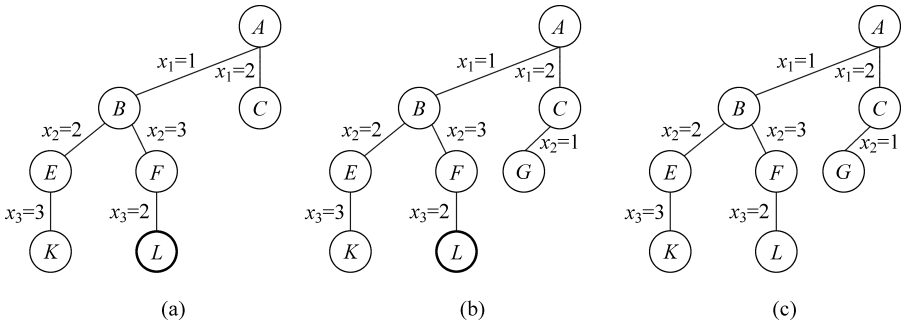


图 5-24　搜索过程 3

结点 G 的一个分支搜索完毕，结点 G 成为死结点，继续回溯到活结点 C，如图 5-25(a)所示。扩展结点 C 沿着 $x_2=3$ 的分支扩展，cf=12，bestf=18，cf<bestf，限界条件满足，扩展生成的结点 H 成为活结点，并成为当前的扩展结点，如图 5-25(b)所示。扩展结点 H 沿着 $x_3=1$ 的分支扩展，cf=21，bestf=18，cf>bestf，限界条件不满足，扩展生成的结点被剪掉。结点 H 的一个分支搜索完毕，开始回溯到活结点 C。此时，结点 C 的两个分支已搜索完毕，继续回溯到活结点 A，结点 A 再次成为当前的扩展结点，如图 5-25(c)所示。

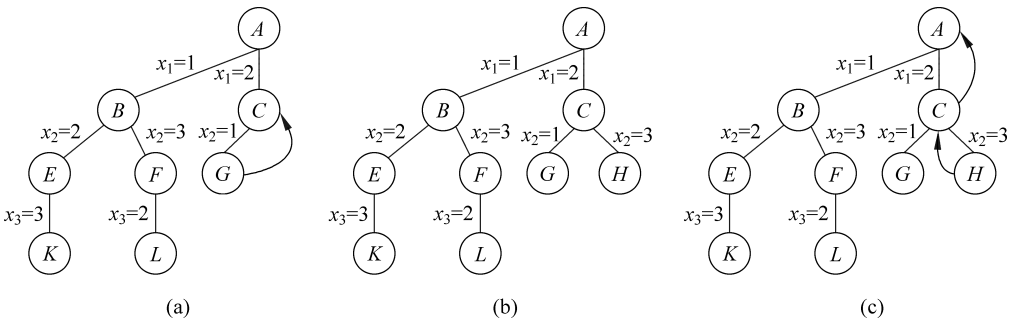


图 5-25　搜索过程 4

扩展结点 A 沿着 $x_1=3$ 的分支扩展，cf=5，bestf=18，cf<bestf，限界条件满足，扩展生成的结点 D 成为活结点，并成为当前的扩展结点，如图 5-26(a)所示。扩展结点 D 沿着 $x_2=1$ 的分支扩展，cf=11，bestf=18，cf<bestf，限界条件满足，扩展生成的结点 I 成为活结点，并成为当前的扩展结点，如图 5-26(b)所示。

扩展结点 I 沿着 $x_3=2$ 的分支扩展，cf=19，bestf=18，cf>bestf，限界条件不满足，扩展生成的结点被剪掉，回溯到活结点 D，结点 D 再次成为当前的扩展结点，如图 5-27(a)所示。

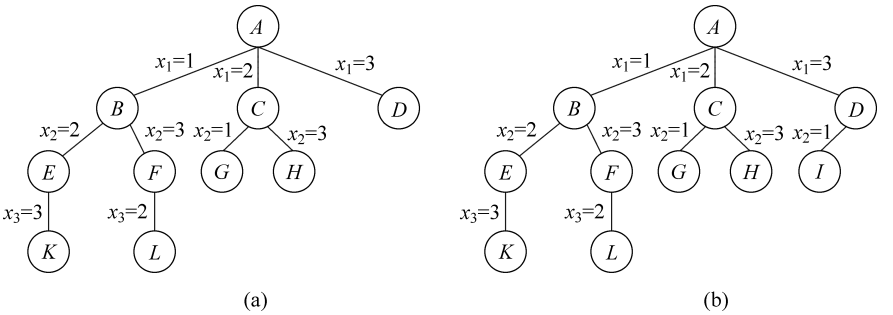


图 5-26　搜索过程 5

扩展结点 D 沿着 $x_2=2$ 的分支扩展，cf=11，bestf=18，cf<bestf，限界条件满足，扩展生成的结点 J 成为活结点，并成为当前的扩展结点，如图 5-27(b)所示。

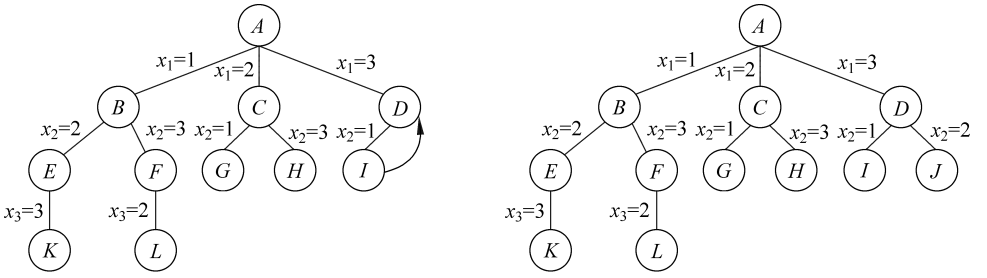


图 5-27　搜索过程 6

扩展结点 J 沿着 $x_3=1$ 的分支扩展，cf=19，bestf=18，cf>bestf，限界条件不满足，扩展生成的结点被剪掉，回溯到活结点 D，结点 D 的两个分支搜索完毕，继续回溯到活结点 A，如图 5-28(a)所示。活结点 A 的 3 个分支也已搜索完毕，结点 A 变成死结点，搜索结束。至此，找到的最优的调度方案为从根结点 A 到叶子结点 L 的路径(1，3，2)，如图 5-28(b)所示。

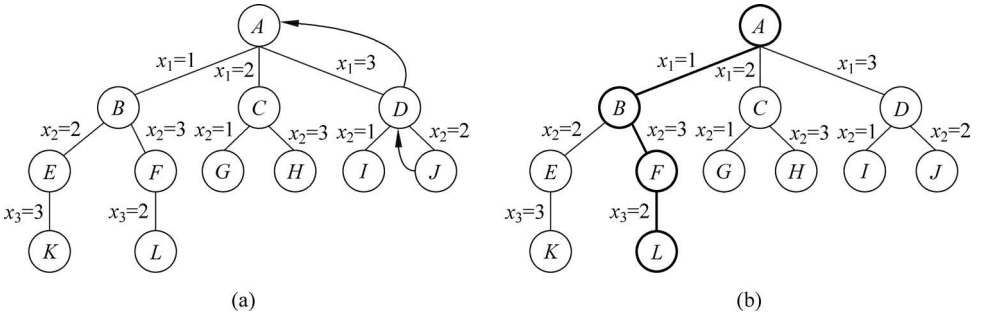


图 5-28　搜索过程 7

(5) 算法描述。

为了对解决该问题的方法进行描述，设置了数组 x、bestx 和 m，变量 $f1$、$f2$、cf 和 bestf，其中数组 x 用来记录当前调度，bestx 用来记录当前最优调度，初始时，$x[i]=i$，

bestx[i]=0,i=1,2,…,n。二维数组 m 记录各作业分别在两台机器上的处理时间；$f1$ 记录作业在第一台机器上的完成时间和，$f2$ 记录作业在第二台机器上的完成时间和，cf 记录当前在第二台机器上的完成时间和，bestf 记录当前最优调度的完成时间和。求解该问题的算法描述如下：

```
void Backtrack(int t)
{
  int tempf,j;
  if(t > n)                              //到达叶子结点
    {for(int i = 1;i < = n;i++)
        bestx[i] = x[i];
    bestf = cf;
    return;
  }//end if
for(j = t; j < = n; j++)                 //非叶子结点
  {f1 + = m[x[j]][1];tempf = f2;         //保存上一个作业在机器 2 的完成时间
    f2 = (f1 > f2?f1:f2) + m[x[j]][2]; //当前作业在机器 2 的完成时间
    cf + = f2;                           //以确定的调度作业在机器 2 上的完成时间和
    if(cf < bestf)
      {swap(x[t], x[j]);                 //交换两个元素的值
       Backtrack(t + 1);
       swap(x[t], x[j]);
      }
    f1 - = m[x[j]][1];
    cf - = f2;
    f2 = tempf;
  }//end for
}
```

求解问题时，先将相关量初始化，然后从根结点开始搜索，即 Backtrack(1)就可以求得问题的最优解。

(6) 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结点在最坏情况下有 $1+n+n(n-1)+n(n-1)(n-2)+\cdots+n(n-1)+\cdots+2\leqslant nn!$ 个，故耗时 $O(nn!)$；在叶子结点处记录当前最优解需要耗时 $O(n)$，在最坏情况下，会搜索到每一个叶子结点，叶子结点有 $n!$ 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 $O(nn!)+O(nn!)=O(nn!)=O((n+1)!)$。

(7) C++实战。

相关代码如下。

```
#include < iostream >
#include < cfloat >
#define INF DBL_MAX
using namespace std;
```

```
class Jobs_Schedule{
    public:
        Jobs_Schedule(int n,double cf,double bestf,double f1,double f2)
        {
            this->n = n;
            this->cf = cf;
            this->bestf = bestf;
            this->f1 = f1;
            this->f2 = f2;
        }
        void Backtrack(int t);
        void print()
        {
            cout<<"最优解为: ";
            for(int i=1;i<=n;i++)
                cout<<bestx[i]<<" ";
            cout<<endl;
        }
        int *x;
        double **m;                          //存储作业编号及作业在机器上的处理实践
        double f1,f2;                        //f1 为第一台机器的完成时间和,f2 为第二台机
                                             //器上的完成时间和
        int *bestx;                          //最优解
        int n;                               //作业个数
        double cf;                           //当前第二台机器上的完成时间和
        double bestf;                        //第二台机器的当前最小的完成时间和
};
void Jobs_Schedule::Backtrack(int t)
{
    int tempf,j,temp;
    if(t>n)                                  //到达叶子结点
    {
        for(int i=1;i<=n;i++)
            bestx[i]=x[i];
        bestf=cf;
        return;
    }//end if
    for(j=t; j<=n; j++)                      //非叶子结点
    {
        f1+=m[x[j]][1];
        tempf=f2;                            //保存上一个作业在机器 2 的完成时间
        f2=(f1>f2?f1:f2)+m[x[j]][2];         //当前作业在机器 2 的完成时间
        cf+=f2;                              //以确定的调度作业在机器 2 上的完成时间和
        if(cf<bestf)
          {swap(x[t], x[j]);                 //交换两个元素的值
           Backtrack(t+1);
           swap(x[t], x[j]);
          }
```

```
            f1 -= m[x[j]][1];
            cf -= f2;
        f2 = tempf;
        }//end for
    }//end Backtrack
    int main(){
        cout <<"请输入图作业数 n:";
        int n;
        cin >> n;
        Jobs_Schedule schedule(n,0,INF,0,0);
        schedule.bestx = new int[n + 1];
        schedule.x = new int[n + 1];
        schedule.m = new double * [n + 1];
        for(int i = 0;i < = n;i++)
        {
            schedule.m[i] = new double[3];
            schedule.x[i] = i;
        }
        //m 的第 0 行舍弃,第一行是第一个作业在第一台和第二台机器上的处理时间
        //m 中存储的元素为(编号,第一台机器的处理时间,第二台机器的处理时间)
        for(int i = 0;i < = n;i++){
            for(int j = 1;j < 3;j++)
                cin >> schedule.m[i][j];
            schedule.m[i][0] = i;
        }
        schedule.Backtrack(1);
        schedule.print();
        cout <<"最小完成时间和为: "<< schedule.bestf << endl;
        delete [ ]schedule.bestx;
        delete [ ]schedule.x;
        for(int i = 0;i < = n;i++)
            delete [ ] schedule.m[i];
        delete [ ] schedule.m;
    }
```

【例 5-6】 旅行商问题。

(1) 问题描述。

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,问应该怎样选择一条最短的路线?

(2) 问题分析。

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；反之，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。

(3) 解题步骤。

步骤 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 $x_1=1$，$x_i\in S-\{x_1,x_2,\cdots,x_{i-1}\}$，$i=2,\cdots,n$。

步骤 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 5-29 所示。

步骤 3：搜索解空间。

步骤 3-1：设置约束条件。

用二维数组 $\boldsymbol{g}[][]$存储无向带权图的邻接矩阵，如果 $\boldsymbol{g}[i][j]\neq\infty$表示城市 i 和城市 j 有边相连，能走通。

步骤 3-2：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，bestf 的初始值为+∞。

步骤 3-3：搜索过程。扩展结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，则进入深一层继续搜索。反之，剪掉扩展生成的结点。搜索到叶子结点时，找到当前最优解。搜索过程直到全部活结点变成死结点。

(4) 旅行商问题的构造实例。

考虑 $n=5$ 的无向带权图，如图 5-30 所示。

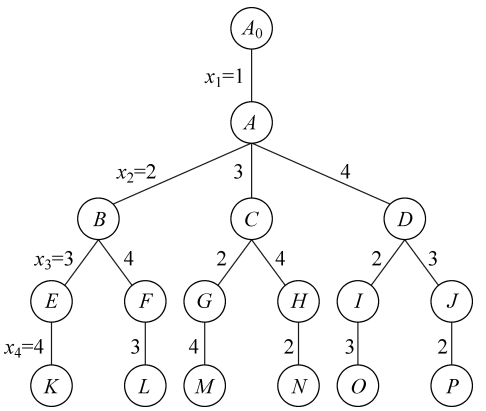


图 5-29　$n=4$ 的解空间树

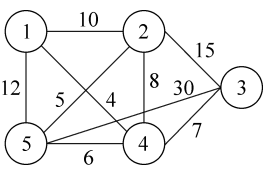


图 5-30　无向带权图

搜索过程如图 5-31～图 5-35 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，搜索从根结点 A_0 的孩子结点 A 开始，结点 A 是活结点，并且成为当前的扩展结点，如图 5-31(a)所示。扩展结点 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，扩展生成的结点 B 成为活结点，并且成为当前的扩展结点，如图 5-31(b)所示。扩展结点 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，扩展生成的结点 C 成为活结点，并且成为当前的扩展结点，如图 5-31(c)所示。扩展结点 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，扩展生成的结点 D 成为活结点，并且成为当前的扩展结点，如图 5-31(d)所示。

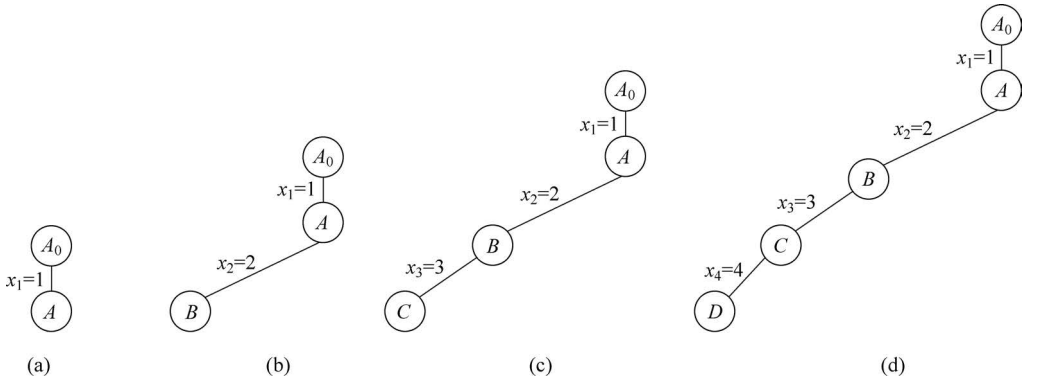


图 5-31　搜索过程 1

扩展结点 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，扩展生成的结点 E 是叶子结点。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 5-32(a)所示。接下来开始回溯到结点 D，再回溯到结点 C，C 成为当前的扩展结点，如图 5-32(b)所示。

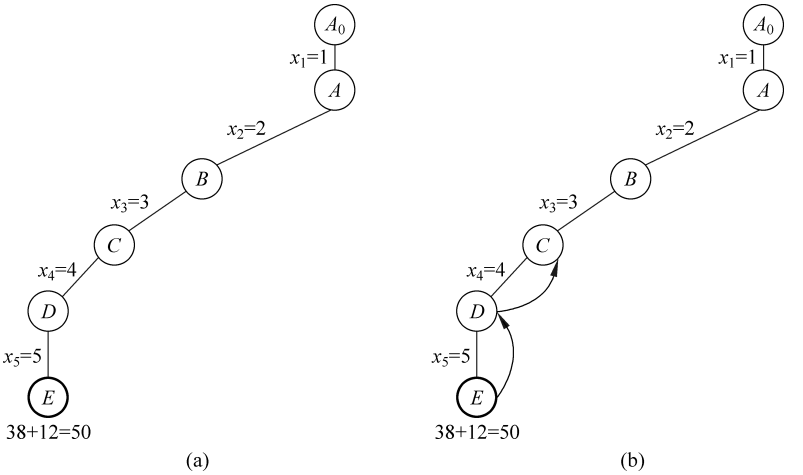


图 5-32　搜索过程 2

以此类推,第一次回溯到第二层的结点 A 时的搜索树如图 5-33 所示。结点旁边的“×”表示不能从推销货物的最后一个城市回到住地城市。

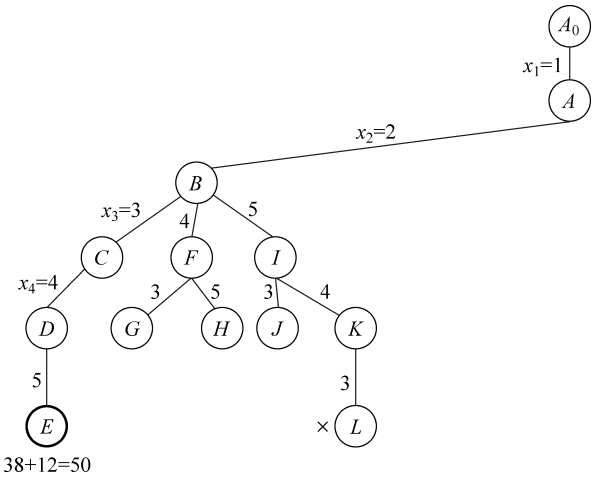


图 5-33 搜索过程 3

第二层的结点 A 再次成为扩展结点,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,扩展生成的结点被剪掉。沿着 $x_2=4$ 的分支扩展,满足约束条件和限界条件,进入其扩展的孩子结点继续搜索。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结点 A 时所形成的搜索树如图 5-34 所示。

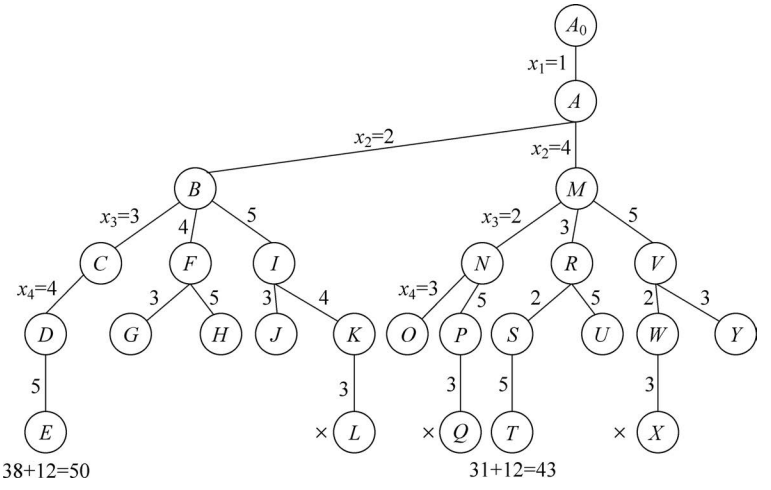


图 5-34 搜索过程 4

结点 A 沿着 $x_2=5$ 的分支扩展,满足约束条件和限界条件,进入其扩展的孩子结点继续搜索,搜索过程略。直到第三次回溯到第二层的结点 A 时所形成的搜索树如图 5-35 所示。此时,搜索过程结束,找到的最优解为图 5-35 中粗线条描述的路径(1,4,3,2,5),路径长度为 43。

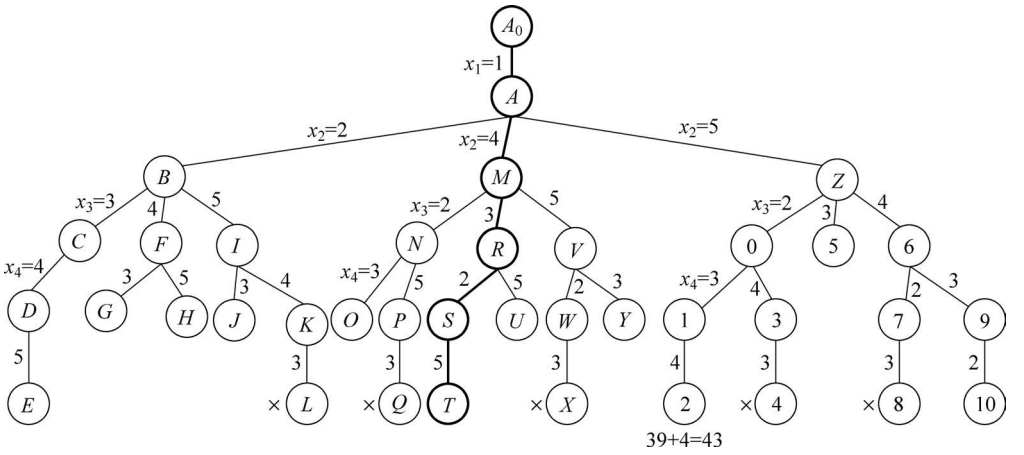


图 5-35　搜索过程 5

(5) 算法描述。

该问题的算法描述中,二维数组 g 表示图的邻接矩阵,数组 x、bestx 分别记录当前路径和当前最优路径。注意,初始时,x 数组中各元素的值和其所在的位置下标相等,bestx 中的元素全部为 0,即 $x[i]=i$,bestx$[i]=0$,$i=1,2,\cdots,n$。变量 cl 和 bestl 分别表示当前路径长度和当前最短路径的路径长度。cl=0,bestl=∞。求解该问题的算法描述如下:

```
void Traveling(int t)
 {
  if(t > n)                          //到达叶子结点
     {
  //推销货物的最后一个城市与住地城市有边相连并且路径长度比当前最优值小,说明找到了一条
  //更好的路径,记录相关信息
      if(g[x[n]][1] != ∞ && (cl + g[x[n]][1]< bestl))
        {
         for(j = 1;j < = n;j++)
                 bestx[j] = x[j];
         bestl = cl + g[x[n]][1];
        }
  }
else                                 //没有到达叶子结点
  for(j = t;j < = n;j++)             //搜索扩展结点的所有分支
    //如果第 t - 1 个城市与第 t 个城市有边相连并且有可能得到更短的路线
      if(g[x[t - 1]][x[j]] != ∞ && (cl + g[x[t - 1]][x[j]]< bestl))
         {//保存第 t 个要去的城市编号到 x[t]中,进入第 t + 1 层
          swap(x[t],x[j]);           //交换两个元素的值
          cl + = g[x[t - 1]][x[t]];
          Traveling(t + 1);          //从第 t + 1 层的扩展结点继续搜索
         //第 t + 1 层搜索完毕,回溯到第 t 层
          cl - = g[x[t - 1]][x[t]];
```

```
            swap(x[t],x[j]);
        }
    }
```

由于旅行商从住地出发,首先推销商品的城市是住地城市,因此,求旅行商最短路径的时候,只需要从解空间树的第二层结点开始搜索就行,即 Traveling(2)。

(6) 算法分析。

判断限界函数需要 $O(1)$时间,在最坏情况下,有 $1+(n-1)+[(n-1)(n-2)]+\cdots+[(n-1)(n-2)\cdots 2]\leqslant n(n-1)!$ 个结点需要判断限界函数,故耗时 $O(n!)$;在叶子结点处记录当前最优解需要耗时 $O(n)$,在最坏情况下,会搜索到每一个叶子结点,叶子结点有 $(n-1)!$ 个,故耗时为 $O(n!)$。因此,旅行售货员问题的回溯算法所需的计算时间为 $O(n!)+O(n!)=O(n!)$。

(7) C++实战。

相关代码如下。

```
#include<iostream>
#include<cfloat>
#define INF DBL_MAX
using namespace std;
class Traving_salesman_problem{
    public:
        Traving_salesman_problem(int n,double cl,double bestl)
        {
            this->n = n;
            this->cl = cl;
            this->bestl = bestl;
        }
        void Traveling(int t)
        {
            if(t>n)                          //到达叶子结点
            {
              //推销货物的最后一个城市与住地城市有边相连并且路径长度比当前最优值小,说
              //明找到了一条更好的路径,记录相关信息
                if(g[x[n]][1] != INF && (cl+g[x[n]][1]<bestl))
                {
                    for(int j=1;j<=n;j++)
                        bestx[j]=x[j];
                    bestl=cl+g[x[n]][1];
                }
            }
            else                             //没有到达叶子结点
                for(int j=t;j<=n;j++)  //搜索扩展结点的所有分支
                //如果第 t-1 个城市与第 t 个城市有边相连并且有可能得到更短的路线
                {
                    if(g[x[t-1]][x[j]] != INF && (cl+g[x[t-1]][x[j]]<bestl))
```

```
                {//保存第 t 个要去的城市编号到 x[t]中,进入第 t+1 层
                    swap(x[t],x[j]); //交换两个元素的值
                    cl += g[x[t-1]][x[t]];
                    Traveling(t+1); //从第 t+1 层的扩展结点继续搜索
                    //第 t+1 层搜索完毕,回溯到第 t 层
                    cl -= g[x[t-1]][x[t]];
                    swap(x[t],x[j]);
                }
            }

        }
        void print()
        {
            cout <<"最优解为: ";
            for(int i=1;i<=n;i++)
                cout << bestx[i]<<" ";
            cout << endl;
        }
        int *x;
        double **g;                     //存储作业编号及作业在机器上的处理实践
        int *bestx;                     //最优解
        int n;                          //作业个数
        double cl;                      //当前第二台机器上的完成时间和
        double bestl;                   //第二台机器的当前最小的完成时间和
};

int main(){
    cout <<"请输入图作业数 n:";
    int n;
    cin >> n;
    Traving_salesman_problem tsp(n,0,INF);
    tsp.bestx = new int[n+1];
    tsp.x = new int[n+1];
    tsp.g = new double*[n+1];
    for(int i=0;i<=n;i++)
    {
        tsp.g[i] = new double[n+1];
        tsp.x[i] = i;                   //解向量初始化,必须是下标与顶点编号一致, 按照
                                        //顶点编号的顺序初始化 x
    }
    //g 为图的邻接矩阵,其第 0 行、0 列舍弃,第一行是 1 号顶点和其他顶点的邻接情况
    for(int i=0;i<=n;i++){
        for(int j=1;j<=n;j++)
            cin >> tsp.g[i][j];
        tsp.g[i][0]=0;
    }
    tsp.Traveling(2);
    tsp.print();
    cout <<"最短路径长度为: "<< tsp.bestl << endl;
    delete []tsp.bestx;
```

```
    delete []tsp.x;
    for(int i = 0;i <= n;i++)
        delete [] tsp.g[i];
    delete [] tsp.g;
}
```


微课视频


微课视频

5.1.4 满 m 叉树

1. 概述

满 m 叉树是用回溯算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 个元素中每一个元素均有 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 5-36 所示。

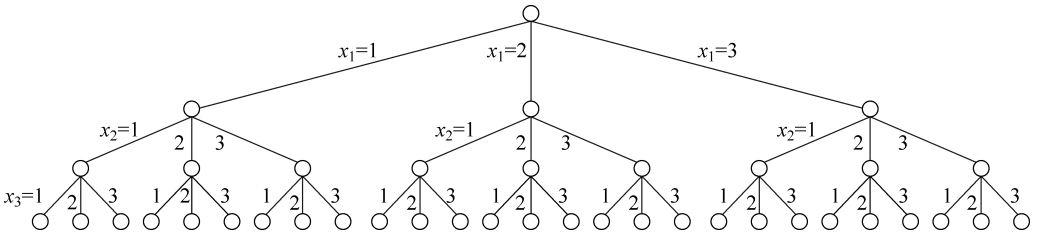


图 5-36 满 3 叉树

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。树的根结点表示初始状态(任何一个元素都没有确定)，中间结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)，叶子结点表示结束状态(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)，从根结点到叶子结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，如：

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。

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。每一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。

最小重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，则问题实质为：n 个部件中的每一个部件均有 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，则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

```
void Backtrack(int t)
 {
    if (t > n) output(x);
    else
      for (int i = 1;i < = m;i++)
         if (constraint(t)&&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;   //退回相关标识
           }
}
```

这里，形式参数 t 表示扩展结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 表示每一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()为约束函数，bound()为限界函数。

3．满 m 叉树的构造实例

【例 5-7】 图的 m 着色问题。

(1) 问题描述。

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。

(2) 问题分析。

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。根据回溯算法的算法框架，定义问题的解空间及其组织结构是很容易的。需要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。

(3) 解题步骤。

步骤 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 $S=\{1,2,\cdots,m\}$,$x_i\in S$,$i=1,2,\cdots,n$。

步骤 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。

步骤 3:搜索解空间。

步骤 3-1: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。假设当前扩展结点所在的层次为 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。

约束函数可描述为:

```
bool OK(int t)
{    for(int j = 1;j < t;j++)
      if(a[t][j])                          //a 表示邻接矩阵
         if(x[j] == x[t])                  //x 记录当前解
           return false;
   return true;
}
```

步骤 3-2:无须设置限界条件。

步骤 3-3:搜索过程。扩展结点沿着某个分支扩展时需要判断约束条件,如果满足,则进入深一层继续搜索;如果不满足,则扩展生成的结点被剪掉。搜索到叶子结点时,找到一种着色方案。搜索过程直到全部活结点变成死结点为止。

(4) 图的 m 着色问题构造实例。

给定如图 5-37 所示的无向连通图和 $m=3$。

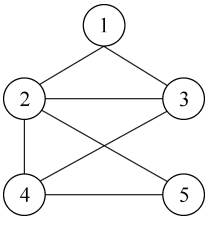


图 5-37　无向连通图

搜索过程如图 5-38～图 5-43 所示:从根结点 A 开始,结点 A 是当前的活结点,也是当前的扩展结点,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 5-38(a)所示。沿着 $x_1=1$ 分支扩展,满足约束条件,生成的结点 B 成为活结点,并成为当前的扩展结点,如图 5-38(b)所示。扩展结点 B 沿着 $x_2=1$ 分支扩展,不满足约束条件,生成的结点被剪掉。然后沿着 $x_2=2$ 分支扩展,满足约束条件,生成的结点 C 成为活结点,并成为当前的扩展结点,如图 5-38(c)所示。扩展结点 C 沿着 $x_3=1$ 和 $x_3=2$ 分支扩展,均不满足约束条件,生成的结点被剪掉。然后沿着 $x_3=3$ 分支扩展,满足约束条件,生成的结点 D 成为活结点,并成为当前的扩展结点,如图 5-38(d)所示。

扩展结点 D 沿着 $x_4=1$ 分支扩展,满足约束条件,生成的结点 E 成为活结点,并成为当前的扩展结点,如图 5-39(a)所示。扩展结点 E 沿着 $x_5=1$ 和 $x_5=2$ 分支扩展,均不满足约束条件,生成的结点被剪掉。然后沿着 $x_5=3$ 分支扩展,满足约束条件,生成的结点 F 是叶子结点。此时,找到了一种着色方案,如图 5-39(b)所示。从叶子结点 F 回溯到活结点 E,

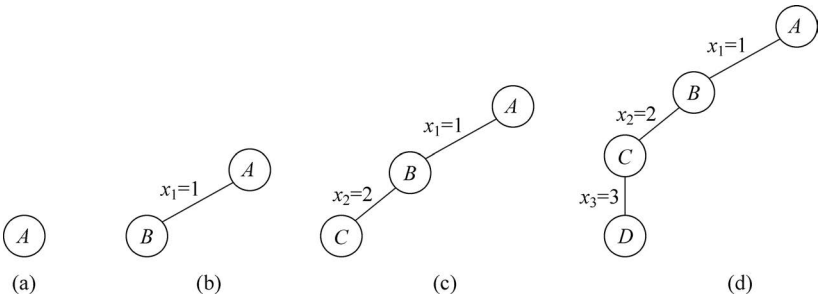


图 5-38　搜索过程 1

结点 E 的所有孩子结点已搜索完毕，因此它成为死结点。继续回溯到活结点 D，结点 D 再次成为扩展结点，如图 5-39(c)所示。

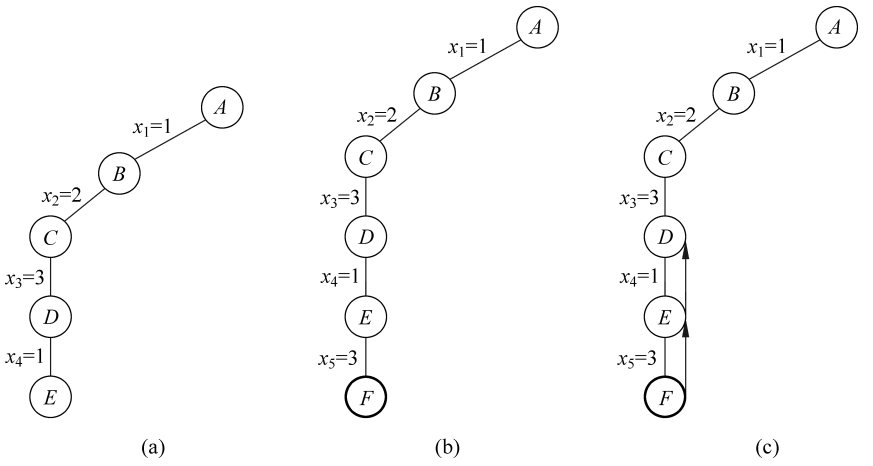


图 5-39　搜索过程 2

扩展结点 D 沿着 $x_4=2$ 和 $x_4=3$ 分支扩展，均不满足约束条件，生成的结点被剪掉。再回溯到活结点 C。结点 C 的所有孩子结点搜索完毕，它成为死结点，继续回溯到活结点 B，结点 B 再次成为扩展结点，如图 5-40(a)所示。扩展结点 B 沿着 $x_2=3$ 分支继续扩展，满足约束条件，生成的结点 G 成为活结点，并成为当前的扩展结点，如图 5-40(b)所示。扩展结点 G 沿着 $x_3=1$ 分支扩展，不满足约束条件，生成的结点被剪掉；然后沿着 $x_3=2$ 分支扩展，满足约束条件，生成的结点 H 成为活结点，并成为当前的扩展结点，如图 5-40(c)所示。

扩展结点 H 沿着 $x_4=1$ 分支扩展，满足约束条件，生成的结点 I 成为活结点，并且成为当前的扩展结点，如图 5-41(a)所示。扩展结点 I 沿着 $x_5=1$ 分支扩展，不满足约束条件，生成的结点被剪掉；然后沿着 $x_5=2$ 分支扩展，满足约束条件，J 已经是叶子结点，找到第 2 种着色方案，如图 5-41(b)所示。从叶子结点 J 回溯到活结点 I，结点 I 再次成为扩展结点，如图 5-41(c)所示。

沿着结点 I 的 $x_5=3$ 分支扩展的结点不满足约束条件，被剪掉。此时结点 I 成为死结点。继续回溯到活结点 H，结点 H 再次成为扩展结点，如图 5-42(a)所示。沿着结点 H 的

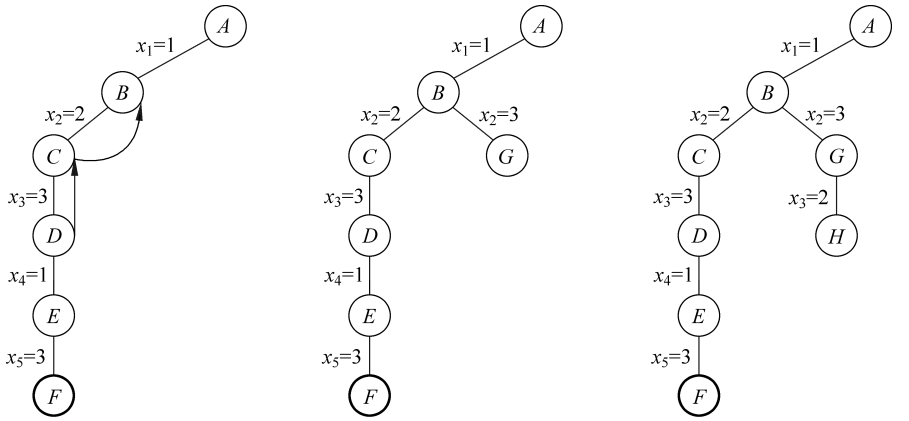


图 5-40 搜索过程 3

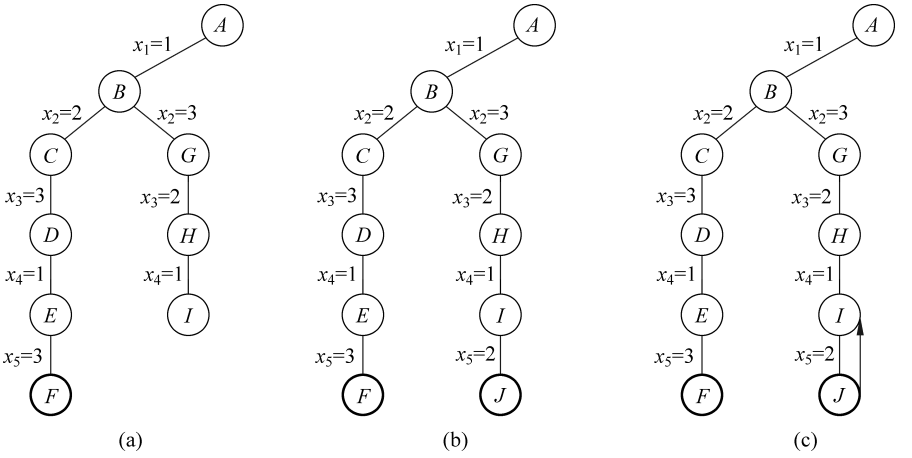


图 5-41 搜索过程 4

$x_4=2$ 和 $x_4=3$ 分支扩展的结点不满足约束条件，被剪掉。此时结点 H 成为死结点。继续回溯到活结点 G，结点 G 再次成为扩展结点，如图 5-42(b)所示。沿着结点 G 的 $x_3=3$ 分支扩展的结点不满足约束条件，被剪掉。此时结点 G 成为死结点。继续回溯到活结点 B，结点 B 的孩子结点已搜索完毕，继续回溯到结点 A，如图 5-42(c)所示。

以此类推，扩展结点 A 沿着 $x_1=2$ 和 $x_1=3$ 分支扩展的情况如图 5-43 所示。

最终找到 6 种着色方案，分别为根结点 A 到如图 5-43(b)所示的叶子结点 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) 算法描述。

在算法描述中，数组 x 记录着色方案；变量 sum 记录着色方案的种数，初始为 0；m 为给定的颜色数；n 为图的顶点个数。该算法的关键是判断当前顶点可以着哪种颜色。图的

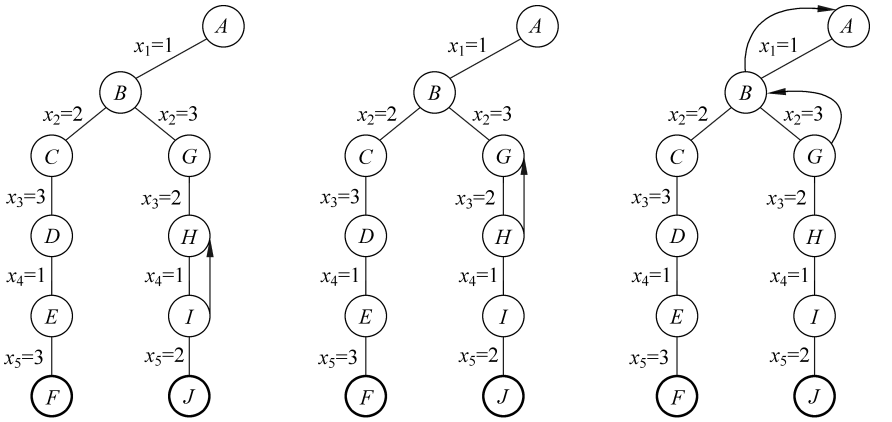


图 5-42　搜索过程 5

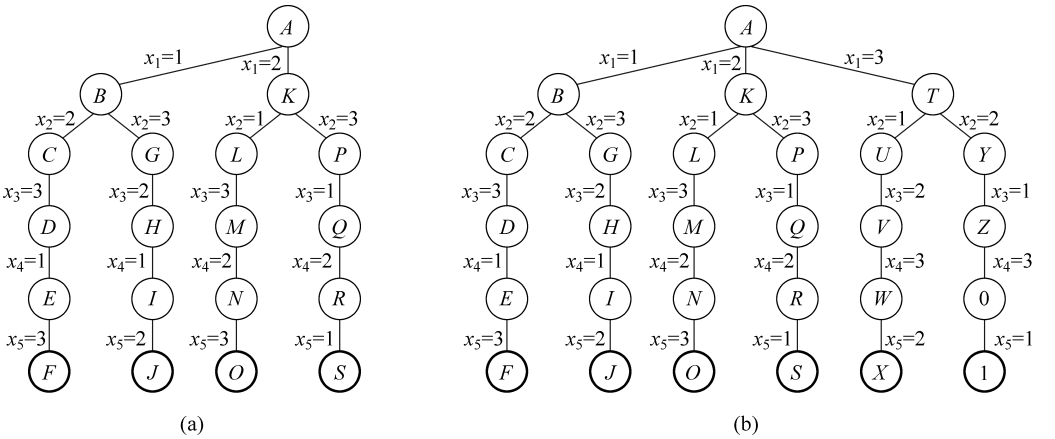


图 5-43　搜索过程 6

着色问题的算法描述如下：

```
void Backtrack(int t)                    //搜索函数
{
    if(t > n)
    {
      sum++;
      printf("第 %d 种方案: \n",sum);
      for(int i = 1;i <= n;i++)
          cout << x[i]<<"  ";
      cout << endl;
    }
    else
      for(int i = 1;i <= m;i++)
       {
```

```
            x[t] = i;
            if(OK(t))
              Backtrack(t + 1);
          }
    }
```

从根结点开始搜索着色方案,即 Backtrack(1)。

(6) 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结点在最坏情况下,有 $1+m+m^2+m^3+\cdots+m^{n-1}=(m^n-1)/(m-1)$个,故耗时 $O(nm^n)$;在叶子结点处输出着色方案需要耗时 $O(n)$,在最坏情况下,会搜索到每一个叶子结点,叶子结点有 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 $O(nm^n)+O(nm^n)=O(nm^n)$。

(7) C++实战。

相关代码如下。

```
#include<iostream>
#include<cfloat>
#define INF DBL_MAX
using namespace std;
class Graphic_m_colors{
    public:
        Graphic_m_colors(int n,int m,int sum){
            this->n = n;
            this->m = m;
            this->sum = sum;
        }
        bool OK(int t){
            for(int j = 1;j<t;j++)
                if(a[t][j])
                    if(x[j] == x[t])
                        return false;
            return true;
        }
        void Backtrack(int t)              //搜索函数
        {
            if(t>n)
            {
                sum++;
                cout<<"第"<<sum<<"种方案为: "<<endl;
                for(int i = 1;i<=n;i++)
                    cout<<x[i]<<" ";
                cout<<endl;
            }
            else
                for(int i = 1;i<=m;i++)
                {
```

```
                    x[t] = i;
                    if(OK(t))
                        Backtrack(t + 1);
                }
        }
        int * x;                        //记录一种着色方案
        int ** a;                       //图的邻接矩阵
        int n;                          //图的顶点个数
        int m;                          //颜色数
        int sum;                        //方案数
};

int main(){
    cout <<"请输入图的顶点数 n 和颜色数 m:";
    int n,m;
    cin >> n >> m;
    Graphic_m_colors gmc(5,3,0);
    gmc.x = new int[n + 1];
    gmc.a = new int * [n + 1];
    for(int i = 0;i < = n;i++)
        gmc.a[i] = new int [n + 1];
    //g 为图的邻接矩阵,其第 0 行、0 列舍弃,从第一行是 1 号顶点和其他顶点的连接边及边权情况
    for(int i = 0;i < = n;i++){
        for(int j = 1;j < = n;j++)
            cin >> gmc.a[i][j];
        gmc.a[i][0] = 0;
    }
    gmc.Backtrack(1);
    cout <<"共: "<< gmc.sum <<"种着色方案."<< endl;
    delete [ ]gmc.x;
    for(int i = 0;i < = n;i++)
        delete [ ] gmc.a[i];
    delete [ ] gmc.a;
}
```

【例 5-8】 最小重量机器设计问题。

(1) 问题描述。

设某一机器由 n 个部件组成,每一个部件可以从 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 的重量,c_{ij} 是相应的价格。试设计一个算法,给出总价格不超过 c 的最小重量机器设计。

(2) 问题分析。

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?要求找出总重量最小的组合,故需要设置限界条件。

(3) 解题步骤。

步骤 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 $S=\{1,2,\cdots,m\}$，$x_i\in S$，$i=1,2,\cdots,n$。

步骤 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。

步骤 3：搜索解空间。

步骤 3-1：设置约束条件。约束条件设置为 $\sum_{i=1}^{n} c_{ix_i} \leqslant c$。

步骤 3-2：设置限界条件。

假设当前扩展结点所在的层次为 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重量，则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。当前最优重量用 bestw 表示，初始值为 $+\infty$，限界条件可描述为：cw＜bestw。

步骤 3-3：搜索过程。与图的 m 着色问题相同。

(4) 最小重量机器设计问题的构造实例。

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 的重量；c_{ij} 表示从供应商 j 处购得的部件 i 的价格。

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

表 5-2 部件的重量表

部 件	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

部 件	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

搜索过程如图 5-44～图 5-49 所示。(注：图中结点旁括号内的数据为已选择部件的重量之和和价格之和。)

从根结点 A 开始进行搜索，A 是活结点且是当前的扩展结点，如图 5-44(a)所示。扩展结点 A 沿 $x_1=1$ 分支扩展，cc=1$\leqslant c$，满足约束条件；cw=1，bestw=∞，cw＜bestw，满足限界条件。扩展生成的结点 B 成为活结点，并成为当前的扩展结点，如图 5-44(b)所示。扩展结点 B 沿 $x_2=1$ 分支扩展，cc=6$\leqslant c$，满足约束条件；cw=4，bestw=∞，cw＜bestw，满足限界条件。扩展生成的结点 C 成为活结点，并成为当前的扩展结点，如图 5-44(c)所示。扩展结点 C 沿 $x_3=1$ 分支扩展，cc=8＞c，不满足约束条件，扩展生成的结点被剪掉。然后沿 $x_3=2$ 分支扩展，cc=7$\leqslant c$，满足约束条件；cw=7，bestw=∞，cw＜bestw，满足限界条件。扩展生成的结点 D 已经是叶子结点，找到了当前最优解，最优重量为 7，将 bestw 修改

为 7，如图 5-44(d)所示。

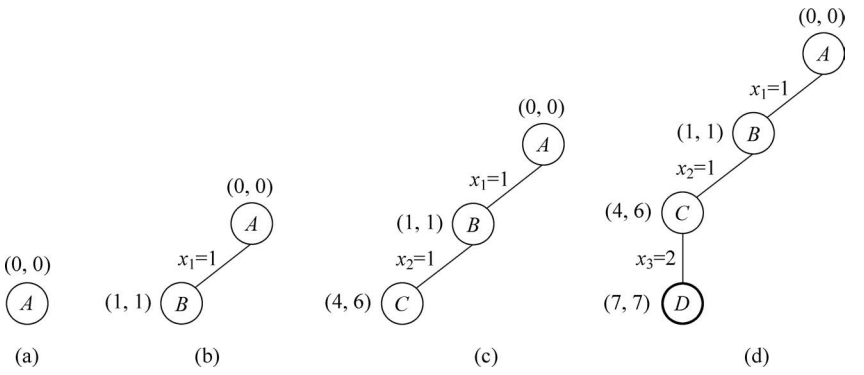


图 5-44　搜索过程 1

从叶子结点 D 回溯到活结点 C，活结点 C 再次成为当前的扩展结点。沿着它的 $x_3=3$ 分支扩展，不满足约束条件，扩展生成的结点被剪掉。继续回溯到活结点 B，结点 B 成为当前的扩展结点，如图 5-45(a)所示。扩展结点 B 沿 $x_2=2$ 分支扩展，cc=5$\leqslant c$，满足约束条件；cw=3，bestw=7，cw<bestw，满足限界条件。扩展生成的结点 E 成为活结点，并成为当前的扩展结点，如图 5-45(b)所示。扩展结点 E 沿 $x_3=1$ 分支扩展，cc=7$\leqslant c$，满足约束条件；cw=5，bestw=7，cw<bestw，满足限界条件。扩展生成的结点 F 已经是叶子结点，找到了当前最优解，最优重量为 5，将 bestw 修改为 5，如图 5-45(c)所示。

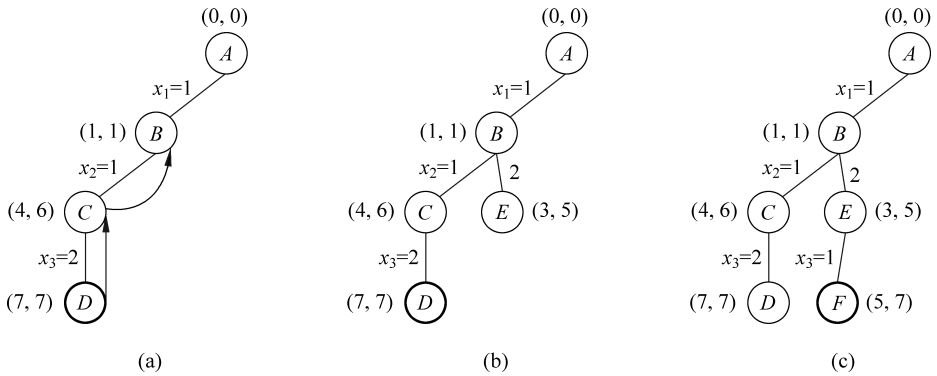


图 5-45　搜索过程 2

从叶子结点 F 回溯到活结点 E，沿着它的 $x_3=2$ 和 $x_3=3$ 分支扩展，均不满足限界条件，扩展生成的结点被剪掉。继续回溯到活结点 B，结点 B 成为当前的扩展结点，如图 5-46(a)所示。扩展结点 B 沿 $x_2=3$ 分支扩展，cc=3$\leqslant c$，满足约束条件；cw=2，bestw=5，cw<bestw，满足限界条件。扩展生成的结点 G 成为活结点，并成为当前的扩展结点，如图 5-46(b)所示。扩展结点 G 沿 $x_3=1$ 分支扩展，cc=5$\leqslant c$，满足约束条件；cw=4，bestw=5，cw<bestw，满足限界条件。扩展生成的结点 H 已经是叶子结点，找到了当前最优解，其重量为 4，bestw 修改为 4，如图 5-46(c)所示。

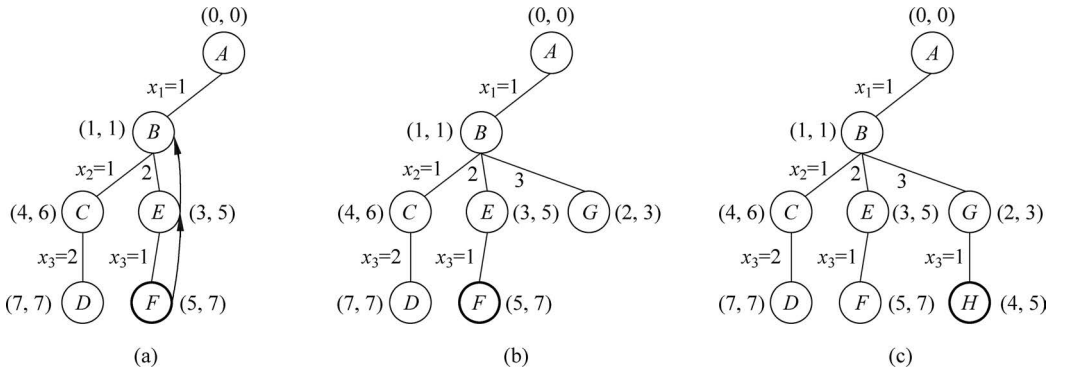


图 5-46 搜索过程 3

从叶子结点 H 回溯到活结点 G，沿着它的 $x_3=2$ 和 $x_3=3$ 分支扩展，均不满足限界条件，扩展生成的结点被剪掉。继续回溯到活结点 B，结点 B 的 3 个分支均搜索完毕，继续回溯到活结点 A，结点 A 成为当前的扩展结点，如图 5-47(a)所示。扩展结点 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。扩展生成的结点 I 成为活结点，并成为当前的扩展结点，如图 5-47(b)所示。

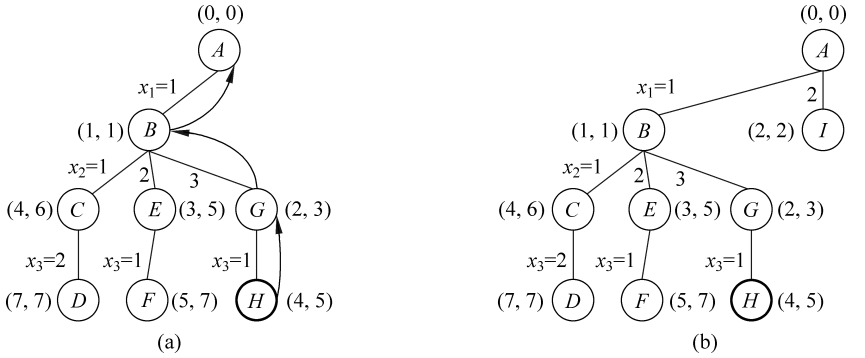


图 5-47 搜索过程 4

扩展结点 I 沿 $x_2=1$ 和 $x_2=2$ 分支扩展，不满足限界条件，扩展生成的结点被剪掉。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。扩展生成的结点 J 成为活结点，并成为当前的扩展结点，如图 5-48(a)所示。扩展结点 J 沿 $x_3=1$、$x_3=2$ 和 $x_3=3$ 分支扩展，均不满足限界条件，扩展生成的结点被剪掉。开始回溯到活结点 I。结点 I 的 3 个分支已搜索完毕，继续回溯到活结点 A，结点 A 再次成为扩展结点，如图 5-48(b)所示。

扩展结点 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。扩展生成的结点 K 成为活结点，并成为当前的扩展结点，如图 5-49(a)所示。扩展结点 K 沿 $x_2=1$、$x_2=2$ 和 $x_2=3$ 分支扩展，均不满足限界条件，扩展生成的结点被剪掉。开始回溯到活结点 A。此时，结点 A 的 3 个分支均搜索完毕，搜索结束。找到了问题的最优解为从根结点 A 到叶子结点 H 的路径(1,3,1)，最优重量为 4，如图 5-49(b)所示。

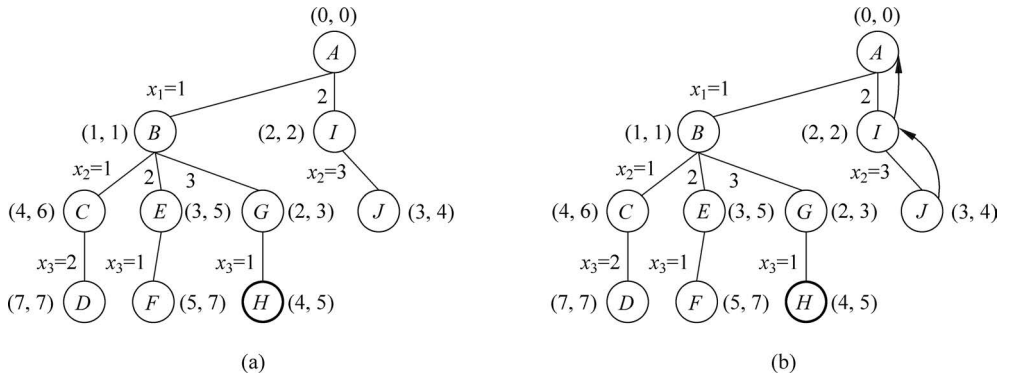


图 5-48　搜索过程 5

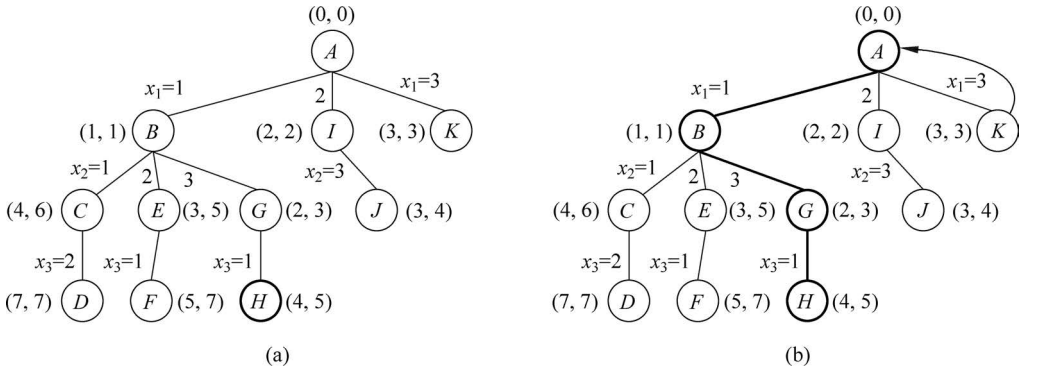


图 5-49　搜索过程 6

（5）算法描述。

最小重量机器设计问题涉及部件个数、部件重量、价格、供应商信息，用数组 w，c 分别存储部件的重量和价格，数组 x、bestx 分别记录当前解和当前最优解，变量 cw、cc、bestw 分别记录当前重量、当前花费、当前最优重量。将算法中用到的数据及对数据的操作定义为一个类，具体描述如下：

```
class MinMachine
{
 int n;                                  //部件个数
 int m;                                  //供应商个数
 double COST;                            //题目中的 C
 double cw;                              //当前的重量
 double cc;                              //当前花费
 double bestw;                           //当前最小重量
 int * bestx; int * x;
 double ** w;
 double ** c;
public:
 MinMachine();                           //构造函数
```

```
    void Backtrack(int i);                    //回溯搜索
    void print();                             //输出最优解和最优重量最优价格
    ~MinMachine();                            //析构函数,释放 new 动态开辟的空间
};
```

类 MinMachine 的构造函数用于对数据成员的初始化,其描述如下:

```
MinMachine::MinMachine()
 {
   cw = 0;                                    //当前的重量
   cc = 0;                                    //当前花费
   bestw = + ∞;                               //当前最小重量
   cout <<"请输入部件个数: "; cin >> n;
   cout <<"请输入供应商个数: "; cin >> m;
   cout <<"请输入总价格不超过: "; cin >> COST;
   w = new double * [n + 1];
   c = new double * [n + 1];
   for(int i = 0;i < = n;i++)
   {
       w[i] = new double[m + 1];
       c[i] = new double[m + 1];
       bestx = new int[n + 1];
       x = new int[n + 1];
   }
   for(int j = 1;j < = m;j++)
     for(i = 1;i < = n;i++)
       {
           cout <<"请输入第 "<< j <<" 个供应商的第 "<< i <<" 个部件的重量: ";
           cin >> w[i][j];
           cout <<"请输入第 "<< j <<" 个供应商的第 "<< i <<" 个部件的价格: ";
           cin >> c[i][j];
           if(w[i][j]< 0 || c[i][j]< 0)
             {cout <<"重量或价钱不能为负数!\n";i = i - 1;}
       }
}
```

类 MinMachine 的成员函数 Backtrack()用于搜索解空间树。搜索问题的解时,从根结点开始,给形式参数 t 传递 1。搜索过程中,沿其中一个分支扩展,判断约束条件和限界条件是否满足,如果满足,则更深一层搜索;如果不满足,则换其他分支继续搜索;如果没有其他分支,则回溯到最近的活结点继续搜索。其描述如下:

```
void MinMachine::Backtrack(int t)
{
    if(t > n)                                 //到达叶子结点
     { bestw = cw;
           for(int j = 1;j < = n; j++)        //保存当前最优解
             bestx[j] = x[j];
```

```
            return;
        }
    for(int j = 1; j <= m; j++)                    //非叶子结点,依次搜索每一个供应商
        { x[t] = j;
          if(cc + c[t][j]<= COST && cw + w[t][j]< bestw)  //判断约束条件和限界条件
           {
             cc + = c[t][j];cw + = w[t][j];
           Backtrack(t + 1); cc - = c[t][j];cw - = w[t][j];
           }
        }
}
```

类 MinMachine 的成员函数 print()用于求解满足条件的问题的最优解并输出问题,即输出最优解及其对应的最小重量和所花费的钱数。其描述如下:

```
void MinMachine::print()
{
  double Totalc = 0;                          //用于记录最优解所花费的钱数
  Backtrack(1);                               //从根结点开始搜索解空间树,找出满足条件的解
  cout <<"\n 最小重量机器的重量是: "<< bestw << endl;
  for(int k = 1; k <= n; k++)
   {
    cout <<" 第 "<< k <<" 部件来自供应商 "<< bestx[k]<<"\n";
    Totalc += c[k][bestx[k]];
   }
cout <<"\n 该机器的总价钱是: "<< Totalc << endl;
}
```

类 MinMachine 的析构函数~MinMachine()用于释放 new 动态开辟的空间。其描述如下:

```
MinMachine::~MinMachine()
{
    for(int i = 0;i <= n;i++)
    {
        delete[ ]w[i];
        delete[ ]c[i];
    }
    delete [ ]w;
    delete [ ]c;
    delete [ ]bestx;
    delete [ ]x;
}
```

(6) 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结点在最坏情况下,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

结点处记录当前最优方案需要耗时 $O(n)$，在最坏情况下，会搜索到每一个叶子结点，叶子结点有 m^n 个，故耗时为 $O(nm^n)$。最小重量机器设计问题的回溯算法所需的计算时间为 $O(m^{n-1})+O(nm^n)=O(nm^n)$。

(7) C++实战。

相关代码如下。

```
#include<iostream>
#include<cfloat>
#define INF DBL_MAX
using namespace std;
class MinMachine
{
        public:
        int n;                          //部件个数
        int m;                          //供应商个数
        double COST;                    //题目中的 C
        double cw;                      //当前的重量
        double cc;                      //当前花费
        double bestw;                   //当前最小重
        int * bestx;
        int *x;
        double **w;
        double **c;
    public:
        MinMachine();                   //构造函数
        void Backtrack(int i);          //回溯搜索
        void print();                   //输出最优解和最优重量最优价格
        ~MinMachine();                  //析构函数,释放 new 动态开辟的空间
};
MinMachine::MinMachine()
{
    cw=0;                               //当前的重量
    cc=0;                               //当前花费
    bestw=INF;                          //当前最小重量
    cout<<"请输入部件个数:";cin>>n;
    cout<<"请输入供应商个数:";cin>>m;
    cout<<"请输入总价格不超过:";cin>>COST;
    w = new double *[n+1];
    c = new double *[n+1];
    for(int i=0;i<=n;i++)
    {
        w[i] = new double[m+1];
        c[i] = new double[m+1];
        bestx = new int[n+1];
        x = new int[n+1];
    }
    for(int j=1;j<=m;j++)
       for(int i=1;i<=n;i++)
```

```
        {
            cout <<"请输入第 "<< j <<" 个供应商的第 "<< i <<" 个部件的重量: ";
            cin >> w[i][j];
            cout <<"请输入第 "<< j <<" 个供应商的第 "<< i <<" 个部件的价格: ";
            cin >> c[i][j];
            if(w[i][j]< 0 || c[i][j]< 0)
            {
                cout <<"重量或价钱不能为负数!\n";
                i = i - 1;
            }
        }
}
void MinMachine::Backtrack(int t)
{
    if(t > n)                          //到达叶子结点
    {
        bestw = cw;
        for(int j = 1;j < = n; j++)    //保存当前最优解
            bestx[j] = x[j];
        return;
    }
    for(int j = 1; j < = m; j++)       //非叶子结点,依次搜索每一个供应商
    {
        x[t] = j;
        if(cc + c[t][j]< = COST && cw + w[t][j]< bestw)          //判断约束条件和限界条件
        {
            cc += c[t][j];cw += w[t][j];
            Backtrack(t + 1);
            cc -= c[t][j];
            cw -= w[t][j];
        }
    }
}
void MinMachine::print()
{
    double Totalc = 0;                 //用于记录最优解所花费的钱数
    Backtrack(1);                      //从根结点开始搜索解空间树,找出满足条件的解
    cout <<"\n 最小重量机器的重量是: "<< bestw << endl;
    for(int k = 1; k < = n; k++)
        cout <<" 第 "<< k <<" 部件来自供应商 "<< bestx[k]<<"\n";
    cout <<"\n 该机器的总价钱是: "<< Totalc << endl;
}
MinMachine::～MinMachine()
{
    for (int i = 0;i < = n;i++)
    {
        delete [ ]w[i];
        delete [ ]c[i];
    }
    delete [ ]w;
```

```
    delete []c;
    delete []bestx;
    delete []x;
}
int main(){
    MinMachine min_machine;
    min_machine.print();                    //从根结点开始搜索解空间树,找出满足条件的解
}
```

5.2 分支限界算法


微课视频

5.2.1 分支限界算法的基本思想

分支限界算法类似于回溯算法,也是一种在问题的解空间树中搜索问题解的算法,它常以宽度优先或以最小耗费(最大效益)优先的方式搜索问题的解空间树。分支限界算法首先将根结点加入活结点表(用于存放活结点的数据结构),接着从活结点表中取出根结点,使其成为当前扩展结点,一次性生成其所有孩子结点,判断孩子结点是舍弃还是保留,舍弃那些导致不可行解或导致非最优解的孩子结点,其余的被保留在活结点表中。再从活结点表中取出一个活结点作为当前扩展结点,重复上述扩展过程,一直持续到找到所需的解或活结点表为空时为止。由此可见,每一个活结点最多只有一次机会成为扩展结点。

可见,分支限界算法搜索过程的关键在于判断孩子结点是舍弃还是保留。因此,在搜索之前要设定孩子结点是舍弃还是保留的判断标准,这个判断标准与回溯算法搜索过程中用到的约束条件和限界条件含义相同。活结点表的实现通常有两种方法:一是先进先出队列,二是优先级队列,它们对应的分支限界算法分别称为队列式分支限界算法和优先队列式分支限界算法。

队列式分支限界算法按照队列先进先出(FIFO)的原则选取下一个结点作为当前扩展结点。优先队列式分支限界算法按照规定的优先级选取队列中优先级最高的结点作为当前扩展结点。优先队列一般用二叉堆来实现:最大堆实现最大优先队列,体现最大效益优先;最小堆实现最小优先队列,体现最小费用优先。

分支限界算法的一般解题步骤为:

(1) 定义问题的解空间。

(2) 确定问题的解空间组织结构(树或图)。

(3) 搜索解空间。搜索前要定义判断标准(约束函数或限界函数),如果选用优先队列式分支限界算法,则必须确定优先级。

5.2.2 0-1 背包问题

分别用队列式分支限界算法和优先队列式分支限界算法解 0-1 背包问题:$n=4$,$w=$

$[3,5,2,1]$，$v=[9,10,7,4]$，$C=7$。

1. 求解步骤

步骤 1：定义问题的解空间。

该实例的解空间为(x_1,x_2,x_3,x_4)，$x_i=0$ 或 $1(i=1,2,3,4)$。

步骤 2：确定问题的解空间组织结构。

该实例的解空间是一棵子集树，深度为 4。

步骤 3：搜索解空间。

根据采用的搜索方法不同定义合适的约束条件和限界条件，然后开始搜索。初始时将根结点放入活结点表中。从活结点表中取出一个活结点作为当前的扩展结点，一次性生成扩展结点的所有孩子结点，判断是否满足约束条件和限界条件，如果满足，则将其插入活结点表；反之则舍弃。搜索过程直到找到问题的解或活结点表为空为止。

2. 0-1 背包问题实例的搜索过程演示

(1) 队列式分支限界算法。

定义约束条件为$\sum_{i=1}^{n} w_i x_i \leqslant C$，限界条件为 cp+rp>bestp。其中，cp 表示当前已装入背包的物品总价值，初始值为 0；rp 表示剩余物品的总价值，初始值为所有物品的价值之和；bestp 表示当前最优解，初始值为 0，当 cp>bestp 时，更新 bestp 为 cp。

采用队列式分支限界算法对该实例的搜索过程如图 5-50～图 5-53 所示。(注：图中深色结点表示死结点，已不在活结点表中；结点旁括号内的数据表示背包的剩余容量、已装入背包的物品价值。)

初始时，将根结点 A 插入活结点表中，结点 A 是唯一的活结点，如图 5-50(a)所示。从活结点表中取出 A，结点 A 是当前的扩展结点，一次性生成它的两个孩子结点 B 和 C，结点 B 满足约束条件，将 bestp 改写为结点 B 的 cp，即 bestp=9；对于结点 C，由于 cp=0，rp=21，bestp=9，满足限界条件，依次将 B 和 C 插入活结点表，如图 5-50(b)所示。再从活结点表中取出一个活结点 B 作为当前的扩展结点，一次性生成 B 的两个孩子结点，左孩子结点不满足约束条件，舍弃；对于右孩子结点 D，由于 cp=9，rp=11，bestp=9，满足限界条件，将结点 D 保存到活结点表中，如图 5-50(c)所示。

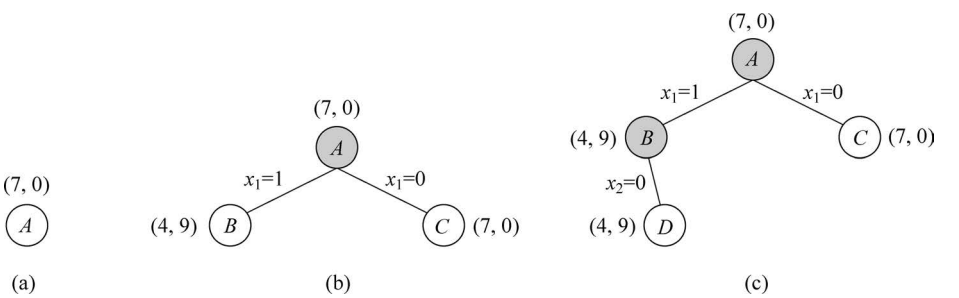


图 5-50　搜索过程 1

重复上述过程,从活结点表中取出 C,结点 C 是当前的扩展结点,一次性生成它的两个孩子结点 E 和 F,结点 E 满足约束条件,将 bestp 改写为结点 E 的 cp,bestp=10。由于 cp=0,rp=11,bestp=10,结点 F 满足限界条件。依次将 E 和 F 插入活结点表中,如图 5-51(a)所示。从活结点表中取出 D,结点 D 是当前的扩展结点,一次性生成它的两个孩子结点 G 和 H,结点 G 满足约束条件,将 G 插入活结点表,bestp 改写为结点 G 的 cp,bestp=16。由于 cp=9,rp=4,bestp=16,结点 H 不满足限界条件,舍弃,如图 5-51(b)所示。

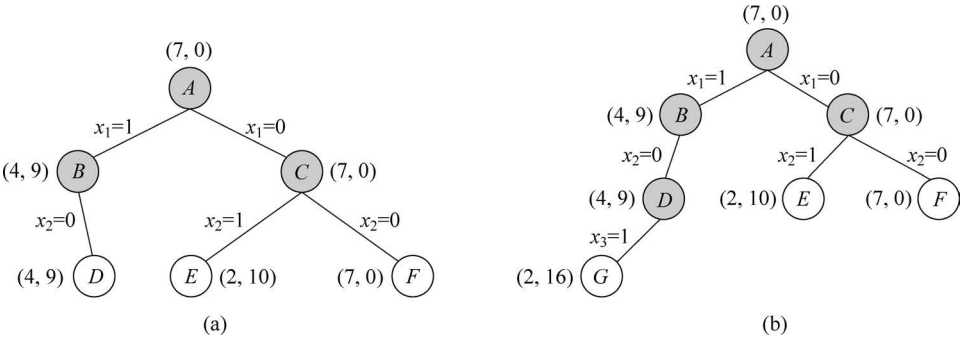


图 5-51 搜索过程 2

从活结点表中取出 E,结点 E 是当前的扩展结点,一次性生成它的两个孩子结点 I 和 J,结点 I 满足约束条件。将其插入活结点表,修改 bestp=17。对于结点 J,由于 cp=10,rp=4,bestp=17,不满足限界条件,舍弃,如图 5-52(a)所示。从活结点表中取出 F,结点 F 是当前的扩展结点,一次性生成它的两个孩子结点 K 和 L,结点 K 满足约束条件,插入活结点表。由于 cp=0,rp=4,bestp=17,结点 L 不满足限界条件,舍弃,如图 5-52(b)所示。

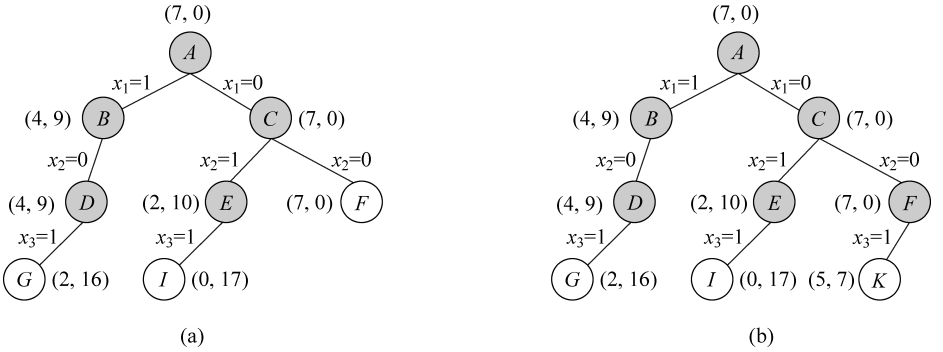


图 5-52 搜索过程 3

从活结点表中取出 G,结点 G 是当前的扩展结点,一次性生成它的两个孩子结点 M 和 N,左孩子结点 M 满足约束条件且已经是叶子结点,此时找到了当前最优解,将 M 暂存临时变量 bestJ 中,修改 bestp=20,右孩子结点 N 不满足限界条件,舍弃。如图 5-53(a)所示。从活结点表中取出活结点 I,它扩展生成的孩子结点不满足约束条件或限界条件,舍弃。再取一个活结点 K,它扩展生成的左孩子结点 O 满足约束条件且已是叶子结点,又找到了一个解,由于该解对应的 cp<bestp,故不保存;K 扩展生成的右孩子不满足限界条件,舍弃。

此时活结点表为空，算法结束，找到了问题的最优解，即从根结点 A 到叶子结点 M 的路径（粗线条表示的路径）(1,0,1,1,)，最优值为 20，如图 5-53(b)所示。

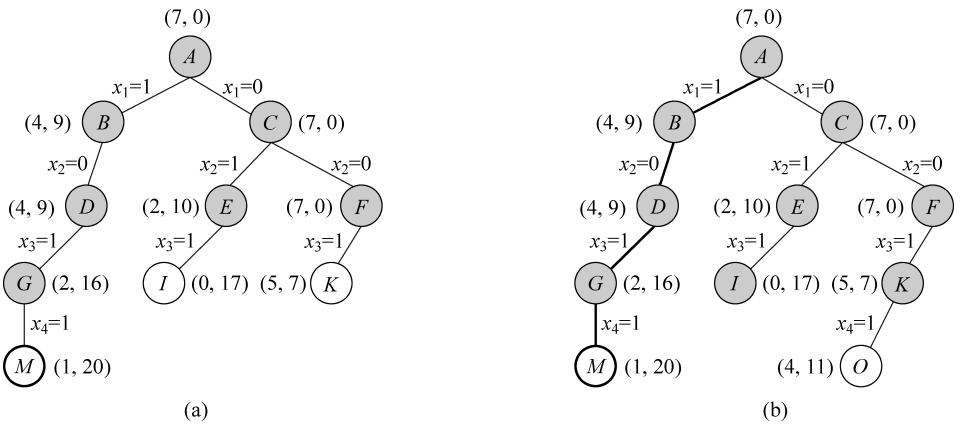


图 5-53　搜索过程 4

（2）优先队列式分支限界算法。

优先级定义为：活结点代表的部分解所描述的装入背包的物品价值上界，该价值上界越大，优先级越高。活结点的价值上界 up＝活结点的 cp＋剩余物品装满背包剩余容量的最大价值 r′p。

约束条件：与队列式分支限界算法相同。限界条件：up＝cp＋r′p＞bestp。

采用优先队列式分支限界算法对上述实例的搜索过程如图 5-54、图 5-55 所示。

初始时，将根结点 A 插入活结点表，结点 A 是唯一的活结点，如图 5-54(a)所示。从活结点表中取出 A，结点 A 是当前的扩展结点，一次性生成它的两个孩子结点 B 和 C，结点 B 满足约束条件，将结点 B 插入活结点表，其 $up=9+4+7+\frac{1}{5}\times 10=22$，bestp＝cp＝9；结点 C 的 $up=0+4+7+\frac{4}{5}\times 10=19$，满足限界条件，将 C 插入活结点表，如图 5-54(b)所示。从活结点表中取出一个优先级最高的活结点 B 作为当前的扩展结点，一次性生成 B 的两个孩子结点，左孩子结点不满足约束条件，舍弃；右孩子结点 D 的 up＝9＋4＋7＝20，满足限界条件，将结点 D 保存到活结点表中，如图 5-54(c)所示。从活结点表中取出优先级最高的活结点 D，结点 D 是当前的扩展结点，一次性生成它的两个孩子结点 E 和 F，结点 E 满足约束条件，其 bestp＝16，up＝16＋4＝20，将 E 插入活结点表；结点 F 的 up＝9＋4＝11，不满足限界条件，舍弃，如图 5-54(d)所示。

从活结点表中取出优先级最高的活结点 E，结点 E 是当前的扩展结点，一次性生成它的两个孩子结点，左孩子结点 G 满足约束条件且是叶子结点，其 bestp＝20，up＝20，将 G 插入活结点表；右孩子结点的 up＝16，不满足限界条件，舍弃，如图 5-55(a)所示。从活结点表取出优先级最高的活结点 G，由于 G 已经是叶子结点，搜索结束，找到了问题的最优解，即从根结点到叶结点的路径(1,0,1,1)，bestp＝20。如图 5-55(b)中粗线条所示。

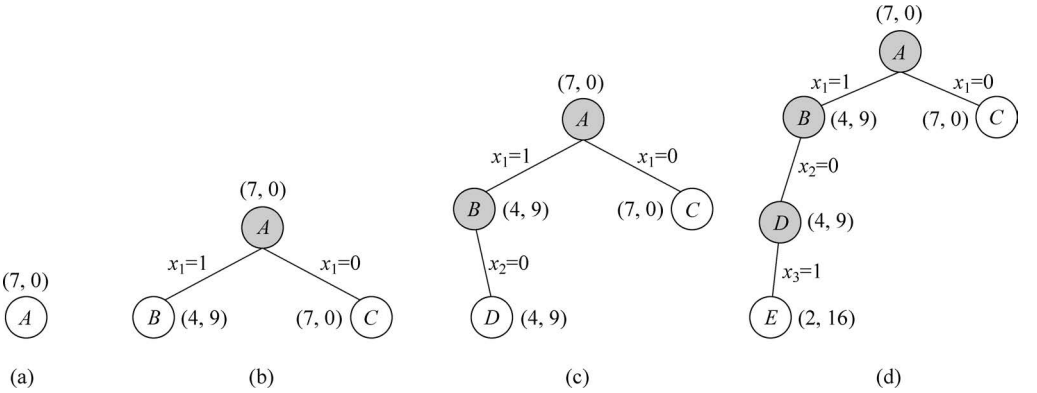


图 5-54 搜索过程 1

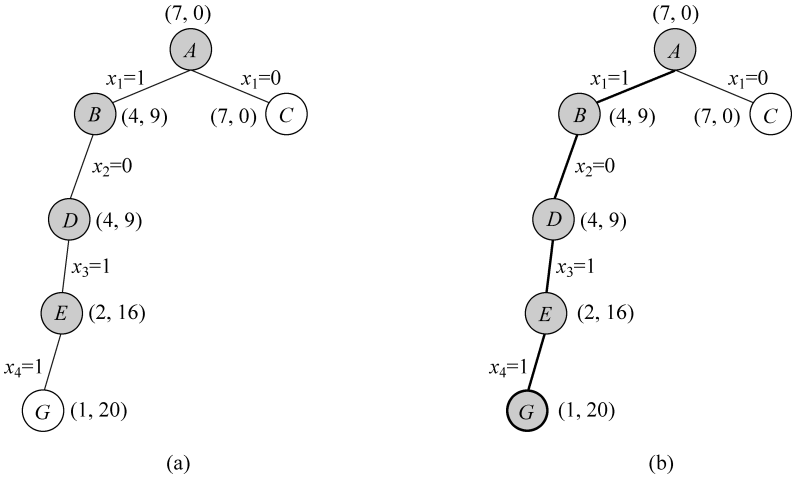


图 5-55 搜索过程 2

思考：如果将队列式分支限界算法中的限界条件改为 cp＋r′p＞bestp，搜索树有何变化？同样，如果将优先队列式分支限界算法中的限界条件改为 cp＋rp＞bestp，搜索树又有何变化？如果选用活结点的 cp 作为优先级呢？这些情况下的搜索树留给读者练习。

3．算法描述

注：针对上述优先队列式分支限界算法进行算法描述。

采用最大堆来实现活结点优先队列，堆中元素 N 的优先级由 uprofit 给出。算法采用的数据结构包括子集树结点类型，最大堆结点类型和最大堆结构体类型，具体描述如下：

(1) 子集树结点类。

```
class bbnode
{
 public:
```

```
  friend class Knap;
  friend int Knapsack(int p[], int w[], int c, int n);
  private:
  bbnode   * parent;                    //指向父结点的指针,构造最优解使用
  bool  LChild;                         //左孩子结点标志,"1"表示左孩子,"0"表示不是左孩子
};
```

(2) 最大堆结点类。

```
class HeapNode
{
   public:
     friend class Knap;
     operator int() const {return uprofit;}
     int uprofit,                       //结点的价值上界
     int profit;                        //结点所对应的价值
     int weight;                        //结点所对应的重量
     int level;                         //活结点在子集树中所处的层序号
     bbnode * ptr;                      //指向活结点在子集树中相应结点的指针
};
```

(3) 堆类型的结构体。

```
typedef struct
{
  int capacity;
  int size;
  HeapNode * Elem;
}Heap, * MaxHeap;
```

定义了堆类型的结构体以后,要对堆进行初始化,即确定堆中最多允许存放的元素个数并开辟足够的空间,当前无任何元素。堆结构体的初始化如下:

```
MaxHeap initH(int maxElem)
{
    MaxHeap H;
    H = new Heap;
    H -> capacity = maxElem;            //堆中最多允许存放的元素个数
    H -> Elem = new HeapNode[maxElem];  //开辟足够的空间
    H -> size = 0;                      //堆中实际存放的元素个数
    return H;
}
```

对堆结构的操作,主要是将一个元素插入堆或从堆中删除并调整成新堆,堆的插入和删除操作描述如下:

(1) 插入堆结点的描述。

```
void InsertH(HeapNode x, MaxHeap H)
{
    if(H->size>=H->capacity){
        cout<<"堆已满,插入失败"<<endl;
        return;
    }
    H->Elem[++H->size]=x;
    int sindex=H->size/2;
    while(sindex>=1){
        if(H->Elem[H->size].uprofit>H->Elem[sindex].uprofit)
            swap(H->Elem[H->size],H->Elem[sindex]);
        else
            break;
        sindex = sindex/2;
    }
}
```

(2) 删除堆结点的算法描述。

```
HeapNode  DeleteMaxH(MaxHeap H)
{   HeapNode HeapTop;
    HeapTop=H->Elem[1];
    H->Elem[1]=H->Elem[H->size--];
    int m=0; int i=1,i1,i2;
    do{
       m=i,i1=2*m,i2=2*m+1;
       if((i1<=H->size)&&(H->Elem[i1].uprofit>H->Elem[i]).uprofit)i=i1;
       if((i2<=H->size)&&(H->Elem[i2].uprofit>H->Elem[i]).uprofit)i=i2;
       if(i!=m)
           swap(H->Elem[i],H->Elem[m]);
       }while(i!=m)
    return HeapTop;
}
```

算法中用到的类 Knap 与用回溯算法解该问题时用到的类 Knap 很相似,它们的主要区别在于本节的类 Knap 中没有成员函数 Backtrack,而增加了新的成员函数 AddLiveNode,用于将活结点插入最大堆;成员函数 MaxKnapsack 用于求问题的最大价值和最优解。

```
class Knap
{
  public:
    friend int Knapsack(int p[], int w[], int c, int n, int bestx[]);
    int MaxKnapsack();
  private:
    MaxHeap H; int Bound(int i);
```

```
    void AddLiveNode(int up, int cp, int cw, bool ch, int level);
    bbnode * E;                          //指向扩展结点的指针
     int c;                              //背包容量
     int n;                              //物品数
     int * w;                            //物品重量数组
     int * p;                            //物品价值数组
     int cw;                             //当前装包重量
     int cp;                             //当前装包价值
     int bestp;                          //最大价值
     int * bestx;                        //最优解
};
```

类 Knap 的成员函数 Bound 与回溯算法解决该问题时的相同，具体描述参见 5.1.2 节的详细描述。

成员函数 AddLiveNode 主要负责将活结点插入活结点表，调用插入堆结点的操作完成。具体描述如下：

```
void Knap::AddLiveNode(int up, int cp, int cw, bool ch, int lev)
{
  //将一个新的活结点插入子集树和最大堆 H
  bbnode * b = new bbnode;
   b->parent = E;
   b->LChild = ch;
    HeapNode N;
    N.uprofit = up;
    N.profit = cp;
    N.weight = cw;
    N.level = lev;
    N.ptr = b;
    InsertH(N, H);
}
```

类 Knap 的成员函数 MaxKnapsack 根据分支限界法的搜索思想搜索解空间树，找出问题的最优解和最优值(最大价值)。具体描述如下：

```
int Knap::MaxKnapsack()
{
     //定义最大堆的容量为 1000
     H = initH(1000); bestx = new int[n + 1];
     int i = 1; E = 0; cw = cp = 0; int bestp = 0;          //当前最优值
     int up = Bound(1);                                     //价值上界
     while(i!= n + 1)                                       //搜索子集空间树
      {                                                     //检查当前扩展结点的左孩子结点
        if(cw + w[i]<= c)                                   //左孩子结点为可行结点
        {
          if(cp + p[i]> bestp)   bestp = cp + p[i];
          AddLiveNode(up, cp + p[i], cw + w[i], true, i + 1);
```

```
        }
      up = Bound(i + 1);
      if(up >= bestp)  AddLiveNode(up, cp, cw, false, i + 1); //检查当前扩展结点的右孩子结点
       //取下一扩展结点
        HeapNode N;
        N = DeleteMaxH(H);
        E = N.ptr; cw = N.weight; cp = N.profit; up = N.uprofit;
        i = N.level;
      }//end while
      for(i = n;i > 0;i -- )                          //构造当前最优解
        { bestx[i] = E -> LChild; E = E -> parent; }
      return cp;
}
```

类的友元函数 Knapsack 完成对输入数据的预处理。其主要任务是对相关变量的初始化、将各物品按照单位重量的价值由大到小排序,然后调用 MaxKnapsack 完成对解空间树的优先队列式分支限界搜索。具体实现与 5.2.2 节中的 Knapsack 类似,请读者参阅 5.2.2 节的 Knapsack 函数的描述自行设计。

4. C++实战

相关代码如下。

```
#include < iostream >
#include < algorithm >
using namespace std;
//子集树结点类
class bbnode
{
    public:
        friend class Knap;
        friend int Knapsack(int p[], int w[], int c, int n);
        private:
        bbnode * parent;          //指向父结点的指针,构造最优解使用
        bool LChild;              //左孩子结点标志,1 表示左孩子结点,0 表示不是左孩子结点
};
//最大堆结点类
class HeapNode
{
    public:
        friend class Knap;
        operator int() const {return uprofit;}
        int uprofit;              //结点的价值上界
        int profit;               //结点所对应的价值
        int weight;               //结点所对应的重量
        int level;                //活结点在子集树中所处的层序号
        bbnode * ptr;             //指向活结点在子集树中相应结点的指针
};
//堆类型的结构体
```

```
typedef struct
{
    int capacity;
    int size;
    HeapNode * Elem;
}Heap, * MaxHeap;
//堆结构的初始化函数
MaxHeap initH(int maxElem)
{
    MaxHeap H;
    H = new Heap;
    H->capacity = maxElem;                    //堆中最多允许存放的元素个数
    H->Elem = new HeapNode[maxElem];          //开辟足够的空间
    H->size = 0;                              //堆中实际存放的元素个数
    return H;
}
//堆插入操作
void InsertH(HeapNode x, MaxHeap H)
{
    if(H->size >= H->capacity){
        cout <<"堆已满,插入失败"<< endl;
        return;
    }
    H->Elem[++H->size] = x;
    int sindex = H->size/2;
    while(sindex >= 1){
        if(H->Elem[H->size].uprofit > H->Elem[sindex].uprofit)
            swap(H->Elem[H->size],H->Elem[sindex]);
        else
            break;
        sindex = sindex/2;
    }
}
//堆删除操作
HeapNode DeleteMaxH(MaxHeap H)
{
    HeapNode HeapTop;
    HeapTop = H->Elem[1];
    H->Elem[1] = H->Elem[H->size-- ];
    int m = 0;
    int i = 1,i1,i2;
    do{
        m = i,i1 = 2 * m,i2 = 2 * m + 1;
        if((i1 <= H->size)&&(H->Elem[i1].uprofit > H->Elem[i].uprofit))
            i = i1;
        if((i2 <= H->size)&&(H->Elem[i2].uprofit > H->Elem[i].uprofit))
            i = i2;
        if(i!= m)
            swap(H->Elem[i],H->Elem[m]);
    }while(i!= m);
```

```
    return HeapTop;
}
class Knap
{
    public:
        friend int Knapsack(int p[], int w[], int c, int n);
        int MaxKnapsack();
        void print()
            {
                cout <<"按照单位重量的价值降序排列的最优解为: ";
                for(int k = 1;k < = n;k++)
                    cout << bestx[k]<<" ";
                cout << endl;
            }
    private:
        MaxHeap H;
        int Bound(int i);
        void AddLiveNode(int up, int cp, int cw, bool ch, int level);
        bbnode * E;                          //指向扩展结点的指针
        int c;                               //背包容量
        int n;                               //物品数
        int * w;                             //物品重量数组
        int * p;                             //物品价值数组
        int cw;                              //当前装包重量
        int cp;                              //当前装包价值
        int bestp;                           //最大价值
        int * bestx;                         //最优解
};

int Knap::Bound(int i)                       //计算上界
{
    int cleft = c - cw;                      //剩余容量
    int b = cp;
    //以物品单位重量价值递减序装入物品
    while(i < = n && w[i]< = cleft)
    {
        cleft -= w[i];
        b += p[i];i++;
    }
    //装满背包
    if(i < = n)
        b += p[i]/w[i] * cleft;
    return b;
}

void Knap::AddLiveNode(int up, int cp, int cw, bool ch, int lev)
{
    //将一个新的活结点插入子集树和最大堆 H 中
    bbnode * b = new bbnode;
    b -> parent = E;
```

```
    b->LChild = ch;
    HeapNode N;
    N.uprofit = up;
    N.profit = cp;
    N.weight = cw;
    N.level = lev;
    N.ptr = b;
    InsertH(N, H);
}

int Knap::MaxKnapsack()
{
    //定义最大堆的容量为 1000
    H = initH(1000);
    bestx = new int[n + 1];
    int i = 1;
    E = 0;
    cw = cp = 0;
    bestp = 0;                                  //当前最优值
    int up = Bound(1);                          //价值上界
    while(i!= n + 1)                            //搜索子集空间树
    { //检查当前扩展结点的左孩子结点
        if(cw + w[i]<= c)                       //左孩子结点为可行结点
        {
            if(cp + p[i]> bestp)
                bestp = cp + p[i];
            AddLiveNode(up, cp + p[i], cw + w[i], true, i + 1);
        }
        up = Bound(i + 1);
        if(up >= bestp)
            AddLiveNode(up, cp, cw, false, i + 1);          //检查当前扩展结点的右孩子结点
        //取下一扩展结点
        HeapNode N;
        N = DeleteMaxH(H);
        E = N.ptr;
        cw = N.weight;
        cp = N.profit;
        up = N.uprofit;
        i = N.level;
    }//end while
    for(i = n;i > 0;i -- )                      //构造当前最优解
    {
        bestx[i] = E->LChild;
        E = E->parent;
    }
    return cp;
}
class Object
{
    public:
```

```
        friend int Knapsack(int p[],int w[],int c,int n);
        int operator<(const Object &a)
        { return(this->d>a.d); }
    private:
        int id;                                  //物品编号
        float d;                                 //单位重量的价值
};

int Knapsack(int p[],int w[],int c,int n)
{ //初始化
    int W=0;int P=0;int i=1;
    Object *Q=new Object[n];
    for(i=1;i<=n;i++)
    {
        Q[i-1].id=i;
        Q[i-1].d=1.0*p[i]/w[i];
        P+=p[i];
        W+=w[i];
    }
    if(W<=c) return P;                           //装入所有物品
    //依物品单位重量降序排序
    sort(Q,Q+n);                                 //将数组Q中的元素按照单位重量的价值由大到小排列
    Knap K;
    K.p=new int[n+1];
    K.w=new int[n+1];
    K.bestx=new int[n+1];
    int *bestx = new int[n+1];
    K.bestx[0]=0;
    for(i=1;i<=n;i++)                            //按单位重量的价值降序排列物品重量和价值
    {
        K.p[i]=p[Q[i-1].id];
        K.w[i]=w[Q[i-1].id];
    }
    K.cp=0;
    K.cw=0;
    K.c=c;
    K.n=n;
    K.bestp=0;
    //分支限界算法搜索
    K.bestp = K.MaxKnapsack();
    K.print();                                   //输出最优解
    cout<<"装入的重量为:"<<K.cw<<endl;
    cout<<"装入的物品编号为:";
    for(int i=1;i<=n;i++)
        if(K.bestx[i])
            cout<<Q[i-1].id<<" ";
    cout<<endl;
    delete[] Q;
    delete[] K.w;
    delete[] K.p;
```

```
    delete [ ] K.bestx;
    delete [ ] bestx;
    return K.bestp;                          //返回最优值
}
int main(){
    int p[ ] = {0,9,10,7,4};
    int w[ ] = {1,3,5,2,1};
    int c = 7;
    int n = 4;
    int bestp = Knapsack(p,w,c,n);
    cout <<"最大价值为: "<< bestp << endl;
    return 0;
}
```

5.2.3　旅行商问题


微课视频

旅行商问题(TSP)的解空间和解空间组织结构已在 5.1 节的例 5-6 中详细分析过。在此基础上,讨论如何用分支限界算法进行搜索。

考虑 $n=4$ 的实例,如图 5-56 所示,城市 1 为售货员所在的住地城市。

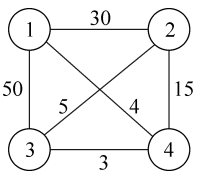


图 5-56　无向连通图

对于该实例,简单做如下分析:

(1) 问题的解空间(x_1,x_2,x_3,x_4),其中令 $S=\{1,2,3,4\}$,$x_1=1$,$x_2\in S-\{x_1\}$,$x_3\in S-\{x_1,x_2\}$,$x_4\in S-\{x_1,x_2,x_3\}$。

(2) 解空间的组织结构是一棵深度为 4 的排列树。

(3) 搜索:设置约束条件 $\boldsymbol{g}[i][j]!=\infty$,其中 $1\leqslant i\leqslant 4$,$1\leqslant j\leqslant 4$,$\boldsymbol{g}$ 是该图的邻接矩阵;设置限界条件:cl<bestl,其中 cl 表示当前已经走的路径长度,初始值为 0;bestl 表示当前最短路径长度,初始值为∞。

搜索过程:

① 队列式分支限界算法对该实例的搜索过程如图 5-57～图 5-59 所示(注:图中结点旁的数据为 cl 的值)。

由于 x_1 的取值是确定的,所以从根结点 A_0 的孩子结点 A 开始搜索即可。将结点 A 插入活结点表,结点 A 是活结点并且是当前的扩展结点,如图 5-57(a)所示。从活结点表中取出活结点 A 作为当前的扩展结点,一次性生成它的 3 个孩子结点 B、C、D,均满足约束条件和限界条件,依次插入活结点表,结点 A 变成死结点,如图 5-57(b)所示。从活结点表中取出活结点 B 作为当前的扩展结点,一次性生成它的两个孩子结点 E、F,均满足约束条件和限界条件,依次插入活结点表,结点 B 变成死结点,如图 5-57(c)所示。

从活结点表中取出活结点 C 作为当前的扩展结点,一次性生成它的两个孩子结点 G、H,均满足约束条件和限界条件,依次插入活结点表,结点 C 变成死结点,如图 5-58(a)所示。从活结点表中取出活结点 D 作为当前的扩展结点,一次性生成它的两个孩子结点 I、J,均满足约束条件和限界条件,依次插入活结点表,结点 D 变成死结点,如图 5-58(b)所示。

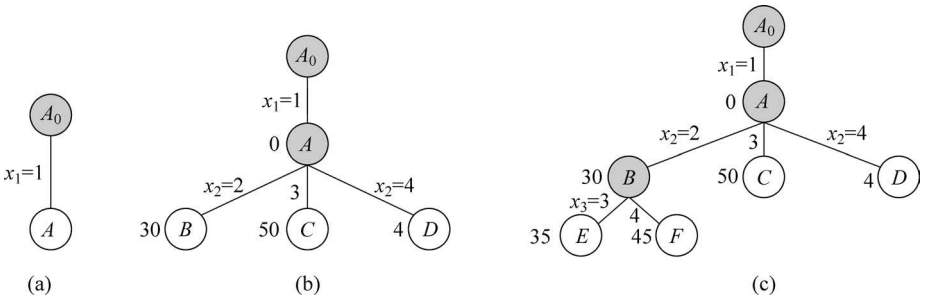


图 5-57 搜索过程 1

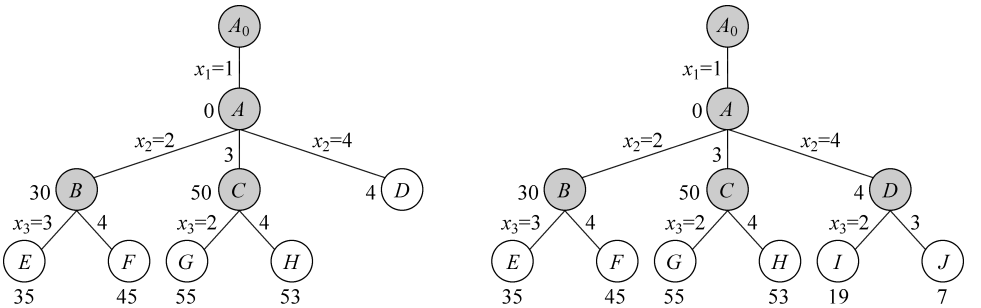


图 5-58 搜索过程 2

从活结点表中取出活结点 E 作为当前的扩展结点，一次性生成它的一个孩子结点 K，满足约束条件和限界条件，结点 K 已经是叶子结点，且顶点 4 与住地城市 1 有边相连，说明已找到一个当前最优解，记录该结点，最短路径长度为 42，修改 bestl＝42，如图 5-59(a)所示。从活结点表中依次取出活结点 F、G、H、I、J，一次性生成它们的孩子结点，均不满足限界条件，舍弃，这些结点变成死结点。此时，活结点表为空，算法结束，找到的最优解是从根结点到叶子结点 K 的路径(1,2,3,4)，路径长度为 42，如图 5-59(b)所示。

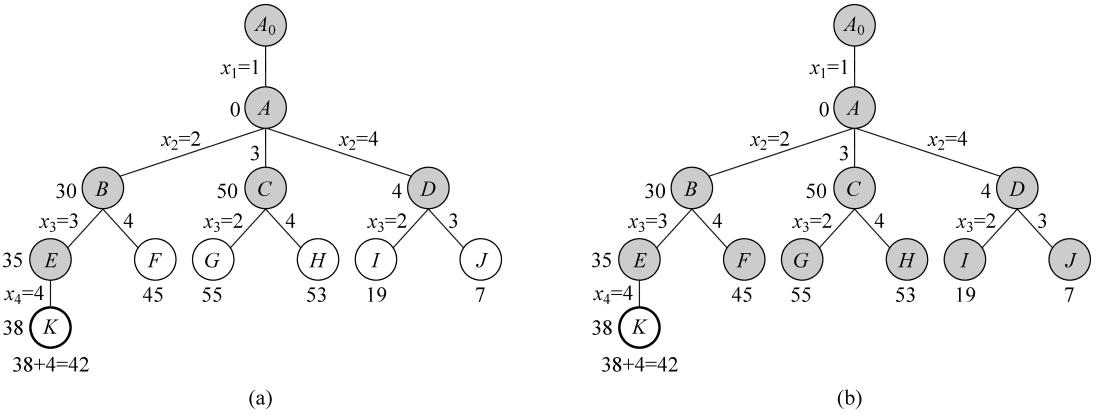


图 5-59 搜索过程 3

② 优先队列式分支限界算法对该实例的搜索过程如图 5-60～图 5-62 所示(注：结点旁的数据为 cl 的值)。

优先级定义为：活结点所对应的已经走过的路径长度 cl，长度越短，优先级越高。

从结点 A 开始，结点 A 插入活结点表，结点 A 是活结点并且是当前的扩展结点，如图 5-60(a)所示。从活结点表中取出活结点 A 作为当前的扩展结点，一次性生成它的 3 个孩子结点 B、C、D，均满足约束条件和限界条件，依次插入活结点表，结点 A 变成死结点，如图 5-60(b)所示。从活结点表中取出优先级最高的活结点 D 作为当前的扩展结点，一次性生成它的两个孩子结点 E、F，均满足约束条件和限界条件，依次插入活结点表，结点 D 变成死结点，如图 5-60(c)所示。

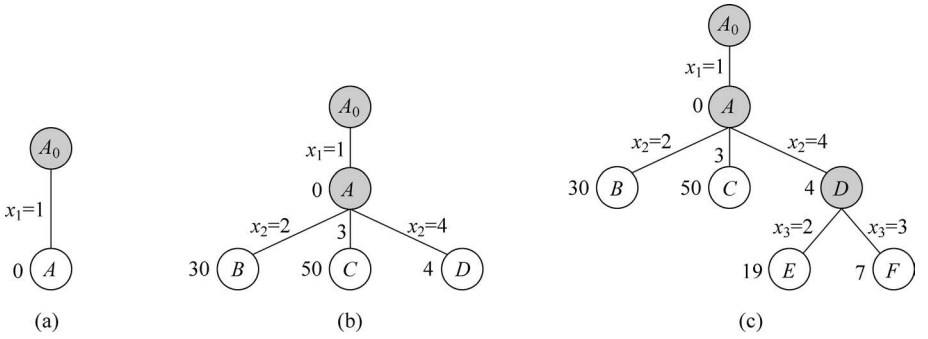


图 5-60　搜索过程 1

从活结点表中取出优先级最高的活结点 F 作为当前的扩展结点，一次性生成它的一个孩子结点 G，满足约束条件和限界条件将 G 插入活结点表。由于结点 G 已经是叶子结点，此时找到了当前最优解，最短路径长度为 42，修改 bestl＝42，如图 5-61(a)所示。从活结点表中取出优先级最高的活结点 E 作为当前的扩展结点，一次性生成它的一个孩子结点，不满足限界条件，舍弃，结点 E 变成死结点，如图 5-61(b)所示。

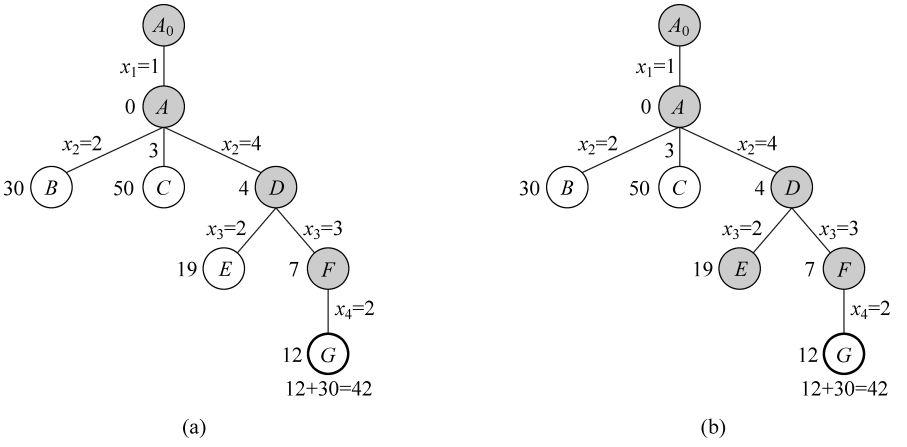


图 5-61　搜索过程 2

从活结点表中取出优先级最高的活结点 B 作为当前的扩展结点，一次性生成它的两个孩子结点 H、I，结点 H 满足约束条件和限界条件，将其插入活结点表；结点 I 不满足限界条件，舍弃。结点 B 变成死结点，如图 5-62(a)所示。从活结点表中取出优先级最高的活结点 H 作为当前的扩展结点，生成的孩子结点不满足限界条件，舍弃，结点 H 变成死结点。再从活结点表中取出优先级最高的活结点 G 作为当前的扩展结点，G 已经是叶子结点，此时找到问题的最优解，算法结束，找到问题的最优解是从根结点 A_0 到叶子结点 G 的最短路径(1,4,3,2)，最短路径长度为 42，如图 5-62(b)中粗线所示。

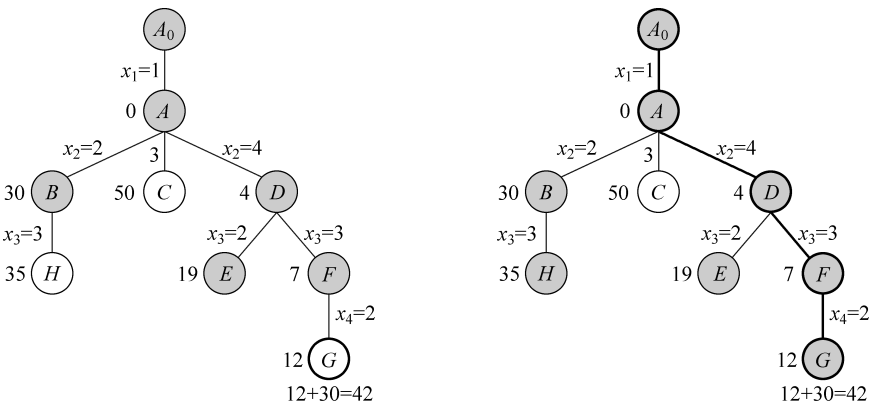


图 5-62　搜索过程 3

(4) 算法优化。

根据题意，每个城市各去一次销售商品。因此，可以估计路径长度的下界(用 zl 表示)，初始时，zl 等于图中每个顶点权最小的出边权之和。随着搜索的深入，可以估计剩余路径长度的下界(用 rl 表示)。故可以考虑用 zl(zl＝当前路径长度 cl＋剩余路径长度的下界 rl)作为活结点的优先级，同时将限界条件优化为 zl＝cl＋rl＞bestl，cl 的初始值为 0，rl 初始值为每个顶点权最小的出边权之和。那么，按照该限界条件，上述搜索过程形成的搜索树如图 5-63 所示。

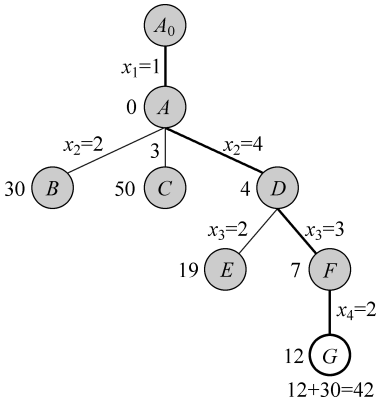


图 5-63　优化条件下的搜索树

(5) 算法描述。

根据上述讨论，优先队列式分支限界算法搜索解空间树时，需要设置优先级和判断孩子结点是舍弃还是保留的判断标准。在具体实现时，用邻接矩阵表示所给的图 G，在类 Traveling 中用二维数组 a 存储图 G 的邻接矩阵，cl 和 bestl 分别表示当前已走的路径长度和当前最短路径长度。类 Traveling 的定义如下：

```
class Traveling
{
  public:
    Traveling(int n,double ** a,double cl,double bestl)
    {
      this -> n = n;
      this -> a = a;
      this -> cl = cl;
      this -> bestl = bestl;
    }
    double  TSP();
    double ShortestEdge(double * Minout,double & MinSum);
    void print(){
        for(int i = 1;i < = n;i++)
            cout << bestx[i]<<"  ";
        cout << end;
    }
  private:
    int n * bestx;                    //图 G 的顶点数
    double ** a,  cl,  bestl;
    //a 表示邻接矩阵,cl 表示当前路径长度,bestl 表示当前最小路径长度
  };
```

根据旅行商问题的特点,每个城市去且仅去一次,可设置该问题的优先级为子树的最短路径长度,其长度越短,优先级越高,故优先队列采用最小堆来实现。最小堆结点结构的定义如下:

```
class MinHeapNode
{
  friend class Traveling;
  public:
      double zl,  cl, rl; //zl 表示子树路径长度的下界,cl 表示当前路径长度,rl 表示
                          //x[s + 1:n]中顶点最短出边路径长度和
      int s, * x, n;      //s 表示根结点到当前结点的路径为 x[1:s - 1],x 表示需要进一步搜索
                          //的结点为 x[s:n],n 表示总结点数
  };
  //定义按照 zl 构建小顶堆
  struct Cmp{
      bool operator()(MinHeapNode a,MinHeapNode b){
      return a.zl > b.zl;
      }
  }
```

类 Traveling 的成员函数 TSP 中,首先创建一个最小堆,找出所有顶点出边最短路径长度并用 Minout 记录,其中 Minout[i]记录了第 i 个顶点的最短出边权;然后计算并用变量 MinSum 记录所有最短出边的路径长度和。数组 x 记录最短路径,初始化为 $x[i]=i$,$i=$

1,2,…,n。由于 x_1 的取值必为住地城市,故搜索最短路径时只需要从解空间树的第二层开始搜索,即 s 初始化为 2。

① 计算每个顶点的最短出边及它们的长度之和。

```
double Traveling::ShortestEdge(double * Minout,double & MinSum)
{
 for(int i = 1;i < = n;i++)                    //计算最小出边及其路径长度和
    {  double Min = ∞;
    for(int j = 1;j < = n;j++)
        if(a[i][j]!= ∞ && (a[i][j]< Min))  Min = a[i][j];
    if(Min == ∞)   return  Min;                //无回路
      Minout[i] = Min;MinSum + = Min;
    }
}
return 1;
```

② TSP 算法描述。

```
double Traveling::TSP(int v[])                 //求解旅行售货员的优先队列式分支限界算法
  {
   priority - queue < MinHeapNode,vector < MinHeapNode >,Cmp > H;
   double MinSum,Minout[n + 1];
   bestx = new int[n + 1];
   if(ShortestEdge(Minout,MinSum,n) == ∞)  return - 1;
   else
       { MinHeapNode E;E.x = new int[n + 1];
         for(i = 1;i < = n;i++)  E.x[i] = i;
         E.s = 2;E.n = n;                      //需记录总结点数,方可分配空间
         E.cl = 0;E.rl = MinSum;double bestl = ∞;
         H.push(E);
       //搜索排列树
       while(E.s < = n&&!H.empty())            //非叶结点
         {
         E = H.top();                          //取下一活结点扩展 H.pop();
          if(E.s == n)                         //当前扩展结点是叶结点的父结点
             {if(a[E.x[n - 1]][E.x[n]]!= ∞ && a[E.x[n]][1]!= ∞ &&
               (E.cl + a[E.x[n - 1]][E.x[n]] + a[E.x[n]][1]< bestl))
                {//再加两条边构成回路,判断回路是否优于当前最优解
                 cl = E.cl + a[E.x[n - 1]][E.x[n]] + a[E.x[n]][1];
                 if(E.cl < best)  {
                      bestl = E.cl;
                      for(int i = 1;i < = n;i++)
                          bestx[i] = E.x[i];  }
                }}
       else{                                   //非叶结点的父结点时,产生当前扩展结点的所有子结点
       for(i = E.s;i < = n;i++)
         if(a[E.x[E.s - 1]][E.x[i]]!= ∞)   //可行子结点
           { double cl = E.cl + a[E.x[E.s - 1]][E.x[i]];double rl = E.rl - MinOut[E.x[E.s - 1]];
```

```
            Type b = cl + rl;                    //子结点的下界
            if(b < bestl)                        //子树可能含有最优解,结点插入最小堆
                {MinHeapNode  N;N.x = new int[n + 1];
                for(j = 1;j < = n;j++) N.x[j] = E.x[j];
                N.x[E.s] = E.x[i];N.x[i] = E.x[E.s];N.cl = cl;N.s = E.s + 1;
                N.n = n; N.zl  = b;N.key = N.zl; N.rl = rl;H.push(N);
                }
            }
        }//end else
    }//end while(E.s < = n)
      if(bestl == ∞)  return - 1;            //无回路
         return bestl;
    }//end else
}
```

(6) C++实战。

相关代码如下。

```
#include < iostream >
#include < cfloat >
#include < algorithm >
#include < queue >
#define INF DBL_MAX
using namespace std;
class Traveling
{
    public:
        Traveling(int n,double ** a,double cl,double bestl)
        {
            this - > n = n;
            this - > a = a;
            this - > cl = cl;
            this - > bestl = bestl;
        }
        double TSP();
        double ShortestEdge(double * Minout,double & MinSum);
        void print(){
            for(int i = 1;i < = n;i++)
                cout << bestx[i]<<" ";
            cout << endl;
        }
    private:
        int n;                              //图 G 的顶点数
        double ** a,cl,bestl;               //a 表示邻接矩阵,cl 表示当前路径长度,bestl 表示当
                                            //前最小路径长度
        int * bestx;
};
//最小堆结构的类
class MinHeapNode
```

```
{
    friend class Traveling;
    public:
        double zl,cl,rl;                    //zl 表示子树路径长度的下界,cl 表示当前路径长度,
                                            //rl 表示 x[s+1:n]中顶点最小出边路径长度和
        int s, *x, n;                       //s 表示根结点到当前结点的路径为 x[1:s-1],x 表示
                                            //需要进一步搜索的结点为 x[s:n],n 表示总结点数
};
//定义按照 zl 构建小顶堆
struct Cmp{
    bool operator()(MinHeapNode a,MinHeapNode b){
    return a.zl > b.zl;
    }
};
//计算每个顶点的最短出边及它们的长度之和
double Traveling::ShortestEdge(double *MinOut,double &MinSum)
{
    for(int i=1;i<=n;i++)                   //计算最小出边及其路径长度和
    {
        double Min=INF;
        for(int j=1;j<=n;j++)
            if(a[i][j]!=INF && (a[i][j]<Min))
                Min=a[i][j];
        if(Min == INF)
            return Min;                     //无回路
        MinOut[i]=Min;
        MinSum+=Min;
    }
    return 1;
}
//解旅行售货员的优先队列分支限界算法
double Traveling::TSP()
{
    priority_queue<MinHeapNode,vector<MinHeapNode>,Cmp> H;
    double MinSum,MinOut[n+1];
    bestx = new int[n+1];
    if(ShortestEdge(MinOut,MinSum)==INF)
        return -1;
    else
    {
        MinHeapNode E;
        E.x=new int[n+1];
        for(int i=1;i<=n;i++)
            E.x[i]= i;
        E.s=2;
        E.n=n;                              //需记录总结点数,方可分配空间
        E.cl=0;
        E.rl=MinSum;
        double bestl=INF;
        H.push(E);
```

```
    //搜索排列树
        while(E.s<=n && !H.empty())   //非叶结点
        {
            E = H.top();              //取下一活结点扩展
            H.pop();
            if(E.s==n)                //当前扩展结点是叶结点的父结点
            {
                if(a[E.x[n-1]][E.x[n]]!= INF && a[E.x[n]][1]!= INF && (E.cl+a[E.x[n-1]]
[E.x[n]]+a[E.x[n]][1]<bestl))
                {
                    //再加2条边构成回路,判断回路是否优于当前最优解
                    E.cl=E.cl+a[E.x[n-1]][E.x[n]]+a[E.x[n]][1];
                    if(E.cl<bestl)
                    {
                        bestl = E.cl;
                        for(int i=1;i<=n;i++)
                        bestx[i] = E.x[i];

                    }
                }
            }
            else
            { //非叶结点的父结点时,产生当前扩展结点的所有子结点
                for(int i=E.s;i<=n;i++)
                    if(a[E.x[E.s-1]][E.x[i]]!= INF)  //可行子结点
                    {
                        double cl=E.cl+a[E.x[E.s-1]][E.x[i]];
                        double rl=E.rl-MinOut[E.x[i]];
                        double b=cl+rl;              //子结点的下界
                        if(b<bestl)                  //子树可能含有最优解,结点插入最小堆
                        {
                            MinHeapNode N;
                            N.x=new int[n+1];
                            for(int j=1;j<=n;j++)
                                N.x[j]=E.x[j];
                            N.x[E.s]=E.x[i];
                            N.x[i]=E.x[E.s];
                            N.cl=cl;
                            N.s=E.s +1;
                            N.n=n;
                            N.zl=b;
                            N.rl=rl;
                            H.push(N);
                        }
                    }
            }//end else
        }//end while(E.s<=n)
          if(bestl==INF)
              return -1;                             //无回路
          return bestl;
```

```
    }//end else
}
int main(){
    cout <<"请输入城市个数 n: ";
    int n;
    cin >> n;
    double **a = new double*[n+1];
    for(int i=0;i<=n;i++)
        a[i] = new double[n+1];
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            cin >> a[i][j];
    Traveling traveling(n,a,0,INF);
    double bestl = traveling.TSP();
    cout <<"最短路径为: "<< bestl << endl;
    cout <<"最优解为: "<< endl;
    traveling.print();
    return 0;
}
```


微课视频

5.2.4 布线问题

1. 问题描述

布线问题就是在 $N \times M$ 的方格阵列中，指定一个方格的中点为 a，另一个方格的中点为 b，如图 5-64 所示，要求找出 a 到 b 的最短布线方案(即最短路径)。布线时只能沿直线或直角，不能走斜线。黑色的单元格代表不可以通过的封锁方格。

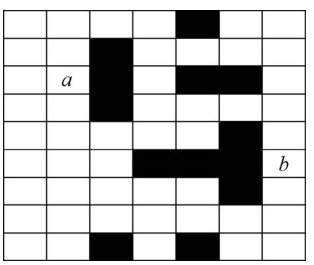


图 5-64　9×7 阵列

2. 问题分析

将方格抽象为顶点，中心方格和相邻 4 个方向(上、下、左、右)能通过的方格用一条边连起来。这样，可以把问题的解空间定义为一个图。

该问题是特殊的最短路径问题，特殊之处在于用布线走过的方格数代表布线的长度，也就是说，布线时每布一个方格，布线长度累加 1。由问题描述可知，从 a 点开始布线，只能朝上、下、左、右 4 个方向进行布线，并且遇到以下几种情况均不能布线：封锁方格、超出方格阵列的边界、已布过线的方格。把能布线的方格插入活结点表，然后从活结点表中取出一个活结点作为当前扩展结点继续扩展，搜索直到找到问题的目标点或活结点表为空为止。采用队列式分支限界算法。

3. 解题步骤

搜索从起始点 a 开始，到目标点 b 结束。约束条件为有边相连且未曾布线。

搜索过程：从 a 开始将其作为第一个扩展结点，沿 a 的上、下、左、右 4 个方向的相邻结点扩展。判断约束条件是否成立，如果成立，则放入活结点表，并将这些方格标记为 1。接

着从活结点队列中取出队首结点作为下一个扩展结点，并将与当前扩展结点相邻且未标记过的方格记为 2。以此类推，直到算法搜索到目标方格或活结点表为空为止。目标方格里的数据表示布线长度。

构造最优解过程。从目标点开始，沿着上、下、左、右 4 个方向。判断如果某个方向方格里的数据比扩展结点方格里的数据小 1，则进入该方向方格，使其成为当前的扩展结点。以此类推，搜索过程一直持续到起始点。如图 5-64 所示实例的搜索过程为：结点 a 的扩展情况如图 5-65(a)所示，方格为 1 的结点扩展情况如图 5-65(b)所示，以此类推，搜索到目标点时的情况如图 5-65(c)所示，构造最优解的过程(上、下、左、右)如图 5-65(d)所示。

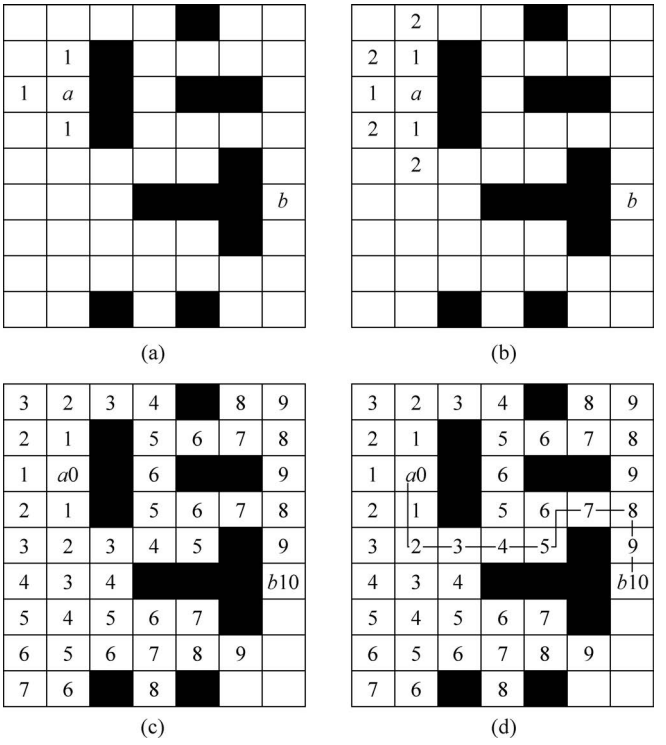


图 5-65　构造最优解

4. 算法描述

在实现布线问题的算法时，需要存储方格阵列、封锁标记、布线的起点和终点位置、4 个方向的相对位置、边界标识等。在下面的算法描述中，用二维数组 grid 表示给定方阵阵列，grid[i][j]等于－1，表示第 i 行、第 j 列的方格未曾布线；grid[i][j]大于或等于 0，表示第 i 行、第 j 列的方格已布线；grid[i][j]＝－2，表示第 i 行、第 j 列的方格已被封锁。为了便于判断是否到达阵列边界，在方格阵列的外围加了一堵“围墙”，其值也为－2，即布线不能通过。

```
typedef struct
{ int row;int col;}Position;
```

```
bool findpath(Position start, Position finish, int&PathLen,Position * &path)
{  if ((start.row == finish.row)&&(start.col == finish.col))     //起点与终点相同,不用布线
       { pathLen = 0;return true;}
   for(int i = 0;i <= m + 1;i++)          //方格阵列的上、下"围墙"
   grid[0][i] = grid[n + 1][i] = -2;
for(int i = 0; i <= n + 1;i++)            //方格阵列的左、右"围墙"
   grid[i][0] = grid[i][n + 1] = -2;
//初始化四个扩展方向相对位置
 Position offset[4];
 offset[0].row = 0;offset[0].col = 1;offset[1].row = 1;offset[1].col = 0;
 offset[2].row = 0;offset[2].col = -1;offset[3].row = -1;offset[3].col = 0;
 int NumofNbrs = 4                       //4 个方向
 Position here, nbr;                     //here 记录当前扩展方格,nbr 记录扩展方向的方格
 here.row = start.row;here.col = start.col;grid[start.row][start.col] = 0
 LinkedQueue < Position > Q;
 do {
      for(int i = 0; i < Numofnbrs;i++)  //沿着扩展结点的右、下、左、上 4 个方向扩展
        {
          nbr.row = here.row + offset[i].row;nbr.col = here.col + offset[i].col;
          if(grid[nbr.row][nbr.col] == -1) //如果这个方格还没有扩展
              grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
          if((nbr.row == finish.row)&&(nbr.col == finish.col))  break;
            //如果到达终点结束
              Q.Add(nbr);                  //将此邻结点放入队列
        }
      if((nbr.row == finish.row)&&(nbr.col == finish.col))  break;     //完成布线
      if(Q.Isempty()) return;              //如果扩展结点都用完了,还到不了终点,则无解
        Q.Delete(here);                    //从队列中取下一个扩展结点
     } while(true)
//逆向构造最短布线方案
PathLen = grid[finish.row][finish.col];path = new Position[Pathlen];
here = finish;
for(int j = PathLen - 1;j > = 0;j -- )
  {
    path[j] = here;
    for(int i = 0;i < Numofnbrs;i++)
      { nbr.row = here.row + offset[i].row;nbr.col = here.col + offset[i].col;
        if (grid[nbr.row][nbr.col] == j) break;
       }
    here = nbr;                            //往回推进
  }
  return true;
}
```

5. C++实战

相关代码如下。

```
#include < iostream >
#include < queue >
```

```
using namespace std;
typedef struct{
    int row;
    int col;
}Position;
bool findpath(int n,int m,int **grid,Position start, Position finish, int&PathLen,Position
*&path)
{
    if ((start.row==finish.row)&&(start.col==finish.col))     //起点与终点相同,不用布线
    {
        PathLen=0;
        return true;
    }
    for(int i=0;i<=m+1;i++)                  //方格阵列的上、下"围墙"
        grid[0][i]=grid[n+1][i]=-2;
    for(int i=0;i<=n+1;i++)                  //方格阵列的左、右"围墙"
        grid[i][0]=grid[i][m+1]=-2;
    //初始化4个扩展方向相对位置
    Position offset[4];
    offset[0].row=0;
    offset[0].col=1;
    offset[1].row=1;
    offset[1].col=0;
    offset[2].row=0;
    offset[2].col=-1;
    offset[3].row=-1;
    offset[3].col=0;
    int Numofnbrs=4;                         //4个方向
    Position here, nbr;                      //here记录当前扩展方格,nbr记录扩展方向的方格
    here.row=start.row;
    here.col=start.col;
    grid[start.row][start.col]=0;
    queue<Position>Q;
    do
    {
        for(int i=0; i<Numofnbrs;i++)        //沿着扩展结点的右、下、左、上4个方向扩展
        {
            nbr.row=here.row+offset[i].row;
            nbr.col=here.col+offset[i].col;
            if(grid[nbr.row][nbr.col]==-1)//如果这个方格还没有扩展
            {
                grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
                Q.push(nbr);                 //将此邻结点放入队列
            }
            if((nbr.row==finish.row)&&(nbr.col==finish.col))
                break;                       //如果到达终点结束
        }
        if((nbr.row==finish.row)&&(nbr.col==finish.col))
            break;                           //完成布线
        if(Q.empty())
```

```
            return 0;                          //如果扩展结点用完,还到不了终点,则无解
        here = Q.front();                      //从队列中取下一个扩展结点
        Q.pop();
    } while(true);
    //逆向构造最短布线方案
    PathLen = grid[finish.row][finish.col];
    path = new Position[PathLen];
    here = finish;
    for(int j = PathLen - 1;j >= 0;j -- )
    {
        path[j] = here;
        for(int i = 0;i < Numofnbrs;i++)
        {
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if (grid[nbr.row][nbr.col] == j)
                break;
        }
        here = nbr;                            //往回推进
    }
    return true;
}
int main(){
    cout <<"请输入棋盘的行列数 n,m:";
    int n,m;
    cin >> n >> m;
    //n = 9, m = 7; 对应图 5-64 中的 9 × 7 阵列
    int ** grid = new int *[n + 2];
    for(int i = 0;i <= n + 1;i++)
        grid[i] = new int[m + 2];
    //初始化为 - 1,标识未访问过的方格
    for(int i = 0;i <= n + 1;i++)
        for(int j = 0;j <= m + 1;j++)
            grid[i][j] = - 1;
    //设置障碍物,对应图 5-64 中的 9 × 7 阵列
    grid[1][5] = grid[2][3] = grid[3][3] = grid[3][5] = grid[3][6] = - 2;
    grid[4][3] = grid[5][6] = grid[6][4] = grid[6][5] = grid[6][6] = - 2;
    grid[7][6] = grid[9][3] = grid[9][5] = - 2;
    Position start,finish;
    cout <<"请输入起点: ";
    cin >> start.row >> start.col;
    cout <<"请输入终点: ";
    cin >> finish.row >> finish.col;
    int PathLen = 0;
    Position *path;
    bool res = findpath(n,m,grid,start,finish,PathLen,path);
    if(res)
    {
        cout <<"最短路径长度为: "<< PathLen << endl;
        cout <<"最短路径为: "<< endl;
```

```
            cout << start.row <<" "<< start.col << endl;
            for(int i = 0;i < PathLen;i++)
                cout << path[i].row <<" "<< path[i].col << endl;
        }
        else
            cout <<"start 到 finish 走不通!"<< endl;
        return 0;
    }
```

5.2.5　分支限界算法与回溯算法的比较

通过以上几小节的学习，容易得知分支限界算法与回溯算法类似。

1. 相同点

(1) 均需要先定义问题的解空间，确定的解空间组织结构一般都是树或图。

(2) 在问题的解空间树上搜索问题解。

(3) 搜索前均需确定判断条件，该判断条件用于判断扩展生成的结点是否为可行结点。

(4) 搜索过程中必须判断扩展生成的结点是否满足判断条件，如果满足，则保留该扩展生成的结点，否则舍弃。

2. 不同点

(1) 在一般情况下，分支限界算法与回溯算法的求解目标不同。回溯算法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界算法的求解目标则是找出满足约束条件的一个解。换言之，分支限界算法是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

(2) 由于求解目标不同，导致分支限界算法与回溯算法在解空间树上的搜索方式也不相同。回溯算法以深度优先的方式搜索解空间树，而分支限界算法则以宽度优先或以最小耗费(最大效益)优先的方式搜索解空间树。

(3) 由于搜索方式不同，直接导致当前扩展结点的扩展方式也不相同。在分支限界算法中，当前扩展结点一次性生成所有的孩子结点，舍弃那些导致不可行解或导致非最优解的孩子结点，其余孩子结点被加入活结点表，而后自己变成死结点。因此，每一个活结点最多只有一次机会成为扩展结点。在回溯算法中，当前扩展结点选择其中某一个孩子结点进行扩展，如果扩展的孩子结点是可行结点，则进入该孩子结点继续搜索，等到以该孩子结点为根的子树搜索完毕，则回溯到最近的活结点继续搜索。因此，每一个活结点有可能多次成为扩展结点。

在解决实际问题时，有些问题用回溯算法或分支限界算法解决效率都比较高，但是有些用分支限界算法解决比较好，而有些用回溯算法解决比较好。如：

(1) 一个比较适合采用回溯算法解决的问题——n 皇后问题。

n 皇后问题的解空间可以组织成一棵排列树，问题的解与解之间不存在优劣差异。直到搜索到叶结点时才能确定出一组解。如果用回溯算法可以系统地搜索 n 皇后问题的全

部解，而且由于解空间树是排列树的特性，代码的编写十分容易。在最坏的情况下，堆栈的深度不会超过 n。如果采取分支限界算法，在解空间树的第一层就会产生 n 个活结点，如果不考虑剪枝，将在第二层产生 $n\times(n-1)$个活结点，如此下去对队列空间的要求太高。

另外，n 皇后问题不适合使用分支限界算法处理的根源在于 n 皇后问题需要找出所有解的组合，而不是某种最优解(事实上也没有最优解可言)。

(2) 一个既可以采用回溯算法也可以采用分支限界算法解决的问题——0-1 背包问题。

0-1 背包问题的解空间树是一棵子集树，问题的解要求具有最优性质。如果采用回溯算法解决这个问题，可采用如下的搜索策略：只要一个结点的左孩子结点是一个可行结点就搜索其左子树；而对于右子树，用贪心算法构造一个上界函数，这个函数表明这个结点的子树所能达到的最大价值，只有在这个上界函数的值超过当前最优解时才进行搜索。随着搜索进程的推进，最优解不断得到加强，对搜索的限制就越来越严格。如果采用优先队列式分支限界算法解决这个问题，同样需要用到贪心算法构造的上界函数。不同的是，这个上界函数的作用不仅仅在于判断是否进入一个结点的子树继续搜索，还用作一个活结点的优先队列的优先级，这样一旦有一个叶结点成为扩展结点，就表明已经找到了最优解。

可以看出，用两种方法处理 0-1 背包问题都有一定的可行性，相比之下回溯算法的思路容易理解一些。但是这是一个寻找最优解的问题，由于采用了优先队列处理，不同的结点没有相互之间的牵制和联系，用分支限界算法处理效果一样很好。

(3) 一个比较适合采用分支限界算法解决的问题——布线问题。

布线问题的解空间是一个图，适合采用队列式分支限界算法来解决。从起始位置 a 开始将它作为第一个扩展结点。与该结点相邻并且可达的方格被加入活结点表，并将这些方格标记为 1，表示它们到 a 的距离为 1。接着从活结点队列中取出队首元素作为下一个扩展结点，并将与当前扩展结点相邻且未标记过的方格标记为 2，并加入活结点表。这个过程一直继续到算法搜索到目标方格 b 或活结点表为空时为止(表示没有通路)。如果采用回溯算法，这个解空间需要搜索完毕才确定最短布线方案，效率很低。

请读者考虑一下，最大团问题、单源最短路径问题、符号三角形问题、图的 m 着色问题等适合采用回溯算法还是分支限界算法来求解。

拓展知识：蚁群算法

群体智能(Swarm Intelligence，SI)是一种人工智能技术，主要探讨由多个简单个体构成的群体的集体行为，这些个体之间相互作用，个体与环境之间也相互影响。尽管没有集中控制机制来指导个体的行为，个体之间的局部交互也能够导致某一社会模式的出现。自然界中诸如此类的现象很多，如蚁群、鸟群、兽群、蜂群等，由这种自然现象引发的“群类算法”，如蚁群算法、粒子群算法能够成功地解决现实中的优化问题。SI 与其他进化算法有共同之处，都是基于种群的，系统从一个由多个个体(潜在解)组成的种群开始，这些个体模仿昆虫或动物的社会行为来代代繁殖，以寻求最优。不同于其他进化算法的是，群体智能模式不使

用交叉、变异这些进化算子，个体只是根据自身与群体中其他个体、与周围环境的关系来不断更新，以求得最优。

蚁群算法是模拟自然界蚂蚁觅食过程的一种分布式、启发式群体智能算法，最早于1991年由学者Colorni、Dorigo和Maniezzo提出，用于求解复杂的组合优化问题，如旅行商问题(TSP)、加工调度问题(JSSP)、图着色问题(GCP)等。

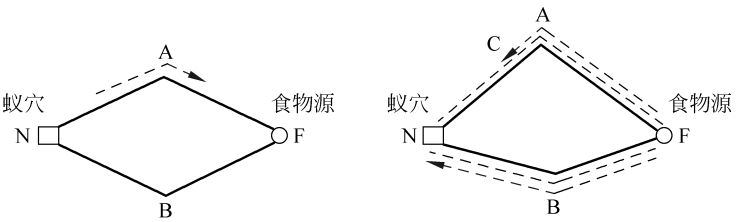
像蚂蚁这类群居昆虫，虽然单个个体的行为极为简单，但由这样的个体组成的蚁群却表现出极其复杂的行为，能够完成复杂的任务，不仅如此，蚂蚁还能适应环境的变化，如在运动的路线上遇到障碍物时，蚂蚁能够很快重新找到最优路径。那么蚁群是如何寻找最优路径的呢？

1. 蚂蚁觅食过程中的最短路径搜索策略

Colorni、Dorigo和Maniezzo三位学者在对大自然中真实蚁群的集体行为进行研究的过程中发现：蚂蚁这类群居昆虫虽然没有视觉，却能找到由蚁巢到食物源的最短路径。再经过进一步的大量细致观察研究发现，蚂蚁个体之间通过一种称为信息素(pheromone)的物质进行信息传递，信息素中记录了食物源的远近与食物量的多少，蚂蚁在运动过程中，能够在它所经过的路径上留下该物质，而其他蚂蚁通过触角能够检测识别到这种信息素，并能够感知这种物质的强度，以此指导自己的运动方向，蚂蚁倾向于朝着该物质强度高的方向移动。因此，由大量蚂蚁组成的蚁群的集体行为便表现出一种信息正反馈现象：某一路径上走过的蚂蚁越多，则后来者选择该路径的概率就越大。蚂蚁个体之间就是通过这种信息的交流达到搜索食物的目的。蚁群算法模拟真实蚁群的协作过程，算法由许多蚂蚁共同完成，每只蚂蚁在候选解的空间中独立地搜索解，并在所寻得的解上留下一定的信息素，解的性能越好蚂蚁留在其上的信息素越大，信息素越大的解被选择的可能性就越大。在算法的初级阶段所有解上的信息素是相同的，随着算法的推进，较优解上的信息素增加，算法渐渐收敛，即最终找到一条最短的路线，此后所有的蚂蚁都将走这条最短路径到达食物源。

如图5-66(a)所示，假设从蚁穴到食物源有两条等长路线NAF和NBF(NAF＝NBF)。开始时，两条路线上都没有信息素，各个蚂蚁将随机选择其中一条路线，并沿途散播信息素；随时间的推移，各路线会挥发掉部分信息素，也不断地增加新的蚂蚁带来的信息素，这是一个正反馈的过程；后来的蚂蚁再选择路线时，浓度较高的路线被选择的概率较大；一段时间后，越来越多的蚂蚁会选择同一条路线，而另一条路线上的蚂蚁数量越来越少，且其上的信息素逐渐挥发殆尽。

如图5-66(b)所示，对于两条不等长的路线NAF和NBF(NAF＞NBF)来说，开始时，两条路线上都没有信息素，各蚂蚁是随机选择其中一条路线，即有一些走NAF路线，另一些走NBF路线，并沿途散播信息素，两条路线上的蚂蚁数大致相等；假设蚂蚁的行走速度相同，则选择走NBF路线(较短路线)的蚂蚁比选择走NAF路线(较长路线)的蚂蚁先到达食物源F；当走NBF路线的蚂蚁返回到蚁穴时，走NAF路线的蚂蚁仍在途中C点处，即2NBF＝NAF＋FAC，可以看出，线段NC上的信息素要少于别处；下次蚂蚁再选择路线时，会以较高概率走较短路径，这使得较长路线上的信息素浓度越来越低，较短路线上的信息素



(a) 在两条相等路线中选择　　(b) 在两条不相等路线中选择

图 5-66　蚂蚁觅食时最短路径选择

浓度越来越高；一段时间后，所有的蚂蚁都将选择较短的路线。

蚁群算法就是从蚂蚁觅食时寻找最短路径的现象中得到启示而设计的，由计算机编程实现的分布式并行搜索策略。蚂蚁通过别的蚂蚁留下来的信息素的强弱作为自己选择路径的参数，信息素越强的路径被选择的可能性越大。信息素的更新策略是越好的路径上获得的信息素越多，通过这个正反馈来寻找更好的路径，这是蚁群算法工作的基本原理。单个蚂蚁的规则相当简单，但是通过蚂蚁群体的协同工作，产生对复杂环境的认知，以实现对解空间进行有效的搜索。

2．蚁群算法的基本步骤

上述介绍的只是有关蚁群算法的简单原理，当然要设计出切实可行的算法并建立模型需要将模型进一步精确，如要计算信息素的挥发(即信息素的浓度将随时间而逐步降低)等。

据蚁群算法的基本原理，人们设计了一个寻找最优路径的蚁群算法：

(1) 一群蚂蚁随机从出发点出发，遇到食物，衔住食物，沿原路返回。

(2) 蚂蚁在往返途中，在路上留下信息素标志。

(3) 信息素随时间逐渐蒸发(一般可用负指数函数来表示)。

(4) 由蚁穴出发的蚂蚁，其选择路径的概率与各路径上的信息素浓度成正比。

利用同样原理可以描述蚁群进行多食物源的寻食情况。

3．基本蚁群算法模型

因为蚂蚁觅食的过程与旅行商问题非常相似，下面通过求解 n 个城市的 TSP 为例来说明基本的蚁群算法模型，其他问题可以对此模型稍做修改便可应用。

首先设在 TSP 中城市 i 与城市 j 之间的距离为 d_{ij}，m 为蚁群中蚂蚁的数量，$b_i(t)$表示 t 时刻位于城市 i 的蚂蚁数量，则有 $m=\sum_{i=1}^{n}b_i(t)$。$\tau_{ij}(t)$表示 t 时刻弧(i,j)上的信息素量。初始时刻各弧上的信息素量相等，$\tau_{ij}(0)=C$，C 为常数。蚂蚁 k 在运动过程中，根据各弧上的信息素量来决定移动的方向，$p_{ij}^{k}(t)$表示在 t 时刻蚂蚁 k 由点 i 向点 j 移动的概率，则有

$$p_{ij}^{k}(t)=\begin{cases}\dfrac{\tau_{ij}^{\alpha}(t)\eta_{ij}^{\beta}(t)}{\sum\limits_{S\in J_k(i)}\tau_{is}^{\alpha}(t)\eta_{is}^{\beta}(t)}, & \text{若 } j\in J_k(i)\\ 0, & \text{其他}\end{cases}\tag{5-4}$$

其中，$J_k(i)$表示城市 i 上的蚂蚁 k 下一步允许选择的城市集合。α,β 分别表示蚂蚁在移动过程中所积累的信息素 $\tau_{ij}(t)$及启发式因子 $\eta_{ij}(t)$在蚂蚁择路时的重要程度。η_{ij} 表示由城市 i 到城市 j 的期望值，可模拟某种启发式算法具体确定。另外，蚁群算法还具有记忆功能，用 tab $u_k(k=1,2,\cdots,m)$记录蚂蚁 k 当前所走过的城市，集合 tab u_k 随进化过程做动态调整。随时间的推移，以前留下的信息素逐渐挥发，用参数 $1-\rho$ 表示信息素挥发程度，经过 l 个时刻，蚂蚁完成一次循环，各弧上的信息素量做以下调整

$$\tau_{ij}(t+l)=\rho\cdot\tau_{ij}(t)+\Delta\tau_{ij} \tag{5-5}$$

$$\Delta\tau_{ij}=\sum_{k=1}^{m}\Delta\tau_{ij}^{k} \tag{5-6}$$

$\Delta\tau_{ij}^{k}$ 表示第 k 只蚂蚁在本次循环中留在弧$<i,j>$上的信息素量，$\Delta\tau_{ij}$ 表示本次循环中弧$<i,j>$上的信息素的总增量，则有

$$\Delta\tau_{ij}^{k}=\begin{cases}\dfrac{Q}{L_k}, & \text{若第 } k \text{ 只蚂蚁在本次循环中经过弧}<i,j>\\ 0, & \text{其他}\end{cases} \tag{5-7}$$

其中，Q 是常数，L_k 表示第 k 只蚂蚁在本次循环中所走路径的总长度。

此模型中，参数 Q，C，α，β，ρ 通常由实验确定其最佳值。

蚁群算法解 TSP 的主要步骤描述如下：

步骤 1：迭代次数 nc=0；各 τ_{ij} 和 $\Delta\tau_{ij}$ 初始化；将 m 个蚂蚁置于 n 个顶点上。

步骤 2：将各蚂蚁的初始出发点置于当前解路线集中；对每个蚂蚁 $k(k=1,2,\cdots,m)$，按式(5-4)的概率 p_{ij}^{k} 移至下一顶点 j；将顶点 j 置于当前解路线集中。

步骤 3：计算各蚂蚁的目标函数值 Z_k；记录当前最好解。

步骤 4：按更新式(5-5)修改轨迹强度。

步骤 5：对各弧(i,j)，置 $\Delta\tau_{ij}=0$，nc=nc+1。

步骤 6：若 nc＜预定迭代次数且无退化行为(即找到的都是相同解)，则转步骤 2；否则，算法结束。

算法的时间复杂度为 $O(\text{nc}\cdot m\cdot n^2)$。就 TSP 而言，经验结果是，当 m 约等于 n 时，效果最佳，此时的时间复杂度为 $O(\text{nc}\cdot n^3)$。

4. 蚁群算法的特点

蚁群算法是继模拟退火算法、遗传算法、禁忌搜索算法、人工神经网络算法等搜索算法以后的又一种应用于组合优化问题的随机搜索算法。众多的研究结果已经证明，蚁群算法具有较强的发现较好解的能力，这是因为该算法不仅利用了正反馈原理(在一定程度上可以加快进化过程)，而且它本质上是一种并行算法，不同个体之间不断地进行信息的交流和传递，从而能够相互协作，有利于发现较好的解。该算法可以解释为一种特殊的强化学习(Reinforcement Learning)算法。

从查新情况分析，研究和应用蚁群算法主要集中在比利时、意大利、英国等欧洲国家，日本和美国也是近些年开始启动研究的。国内则于 1998 年末到 1999 年初才开始有少量的公

开报道和研究成果,多局限于 TSP。

该算法具有如下的优点:

(1) 自组织性。算法初期,单个的蚂蚁无序地寻找解,经过一段时间的演化,蚂蚁间通过信息素的作用,自发地越来越趋向于寻找到接近最优解的一些解,是个从无序到有序的过程。

(2) 较强的鲁棒性。对蚁群算法模型稍加修改,便可以应用于其他问题。

(3) 本质上并行。蚁群在问题空间的多点同时开始进行独立的解搜索,增强了算法的全局搜索能力。

(4) 正反馈。蚁群能够最终找到最短路径,直接依赖于最短路径上信息素的堆积,而信息素的堆积却是一个正反馈的过程。

(5) 易于与其他方法结合。蚁群算法很容易与多种启发式算法结合,以改善算法的性能。

蚁群算法也存在一些缺陷:

(1) 需要较长的搜索时间。当群体规模较大时,很难在较短的时间内从大量杂乱无章的路径中找出一条较好的路径。

(2) 容易出现停滞现象(Stagnation Behavior),即在搜索进行到一定程度后,所有个体所发现的解完全一样,以致不能对解空间进一步进行搜索,不利于发现更好的解。

许多研究者已经注意到上述两个问题,并提出了一些改善措施,如 M. Dorigo 提出蚁量系统(Ant-Quantity System),Thomas 等提出最大最小蚁群系统(Max-Min Ant System,MMAS),郝晋等提出具有扰动特性的蚁群系统(Stochastic Distributed Ant System,SDAS),陈烨提出带杂交算子的蚁群算法,等等。

5. 蚁群算法的应用情况

由于蚁群算法不依赖于问题的具体领域,所以在很多学科有广泛的应用。例如:

(1) 函数优化。蚁群算法应用的领域之一,也是评价蚁群算法性能的主要方法。

(2) 组合优化。用于二次分配问题、背包问题、TSP、加工车间(job-shop)问题等。

(3) 人工生命。蚁群算法在人工生命及复杂系统的模拟设计等方面应用前景广阔。

(4) 机器学习。基于蚁群算法的机器学习在许多领域得到了应用。例如,利用蚁群算法进行神经网络训练、神经网络的优化设计等。

此外,蚁群算法在集成电路布线、数据挖掘、生产调度等方面也有广泛应用。

特别需要指出的是:由于蚁群算法在求解复杂组合优化问题方面具有并行化、正反馈、鲁棒性强等先天优越性,所以在解决一些组合优化问题时所取得的结果无论是在解的质量上,还是在收敛速度上都优于或至少等效于模拟退火及其他启发式算法。

本章习题

5-1 叙述回溯算法和分支限界算法的思想。

5-2 试说明回溯算法与分支限界算法的异同点。

5-3　部落卫队问题。原始部落中的居民们为了争夺有限的资源，经常发生冲突。几乎每个居民都有仇敌。部落酋长为了组织一支保卫部落的队伍，希望从部落的居民中选出最多的居民入伍，并保证队伍中任何两个人都不是仇敌。给定部落中居民间的仇敌关系，编程计算组成部落卫队的最佳方案。

5-4　子集和问题。该问题的一个实例为$<s,t>$，其中$s=\{x_1,x_2,\cdots,x_n\}$是一个正整数的集合，t是一个正整数，判定是否存在s的一个子集s_1，使得s_1的元素和等于t。

5-5　有一批共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$\sum_{i=1}^{n}w_i\leqslant c_1+c_2$。要求确定是否可以将这个集装箱装上这两艘轮船。如果有，找出一种合理的装载方案。

5-6　给定n个大小不等的圆$c_1,c_2,\cdots,c_n$，现要将这n个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。要求从n个圆的所有排列中找出有最小长度的圆排列。例如，当$n=3$，且给定的3个圆的半径分别为1,1,2时，这3个圆的最小长度的圆排列如图5-67所示。其最小长度为$2+4\sqrt{2}$。

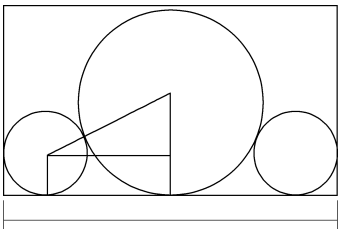


图5-67　题5-6

5-7　运动员最佳匹配问题。羽毛球队有男女运动员各n人。给定两个$n\times n$矩阵$\boldsymbol{P}$和$\boldsymbol{Q}$。$\boldsymbol{P}[i][j]$是男运动员i和女运动员j配对组成混合双打的男运动员竞赛优势；$\boldsymbol{Q}[i][j]$是女运动员i和男运动员j配合的女运动员竞赛优势。由于技术配合和心理状态等各种因素影响，$\boldsymbol{P}[i][j]$不一定等于$\boldsymbol{Q}[j][i]$。男运动员i和女运动员j配对组成混合双打的男女双方竞赛优势为$\boldsymbol{P}[i][j]\times\boldsymbol{Q}[j][i]$。设计一个算法，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

5-8　最小长度电路板排列问题。最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将n块电路板以最佳排列方案插入带有n个插槽的机箱。n块电路板的不同的排列方式对应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。集合$L=\{N_1,N_2,\cdots,N_m\}$是n块电路板的m个连接块。其中每个连接块N_i是B的一个子集，且N_i中的电路板用同一根导线连接在一起。

5-9　设有n个顾客同时等待一项服务。顾客i需要的服务时间为T_i，其中$1\leqslant i\leqslant n$。共有s处可以提供此项服务。应如何安排n个顾客的服务次序才能使平均等待时间达到最小？平均等待时间是n个顾客等待服务时间的总和除以n。

5-10　整数变换问题。关于整数i的变换f和g定义如下：$f(i)=3i$，$g(i)=\lfloor i/2\rfloor$。试设计一个算法，对于给定的两个整数n和m，用最少的f和g变换次数将n变换为m。例如，可以用4次变换将整数15变换为整数4：4=gfgg(15)。当整数n不可能变换为整数m时，算法应如何处理？

5-11　推箱子问题。码头仓库是划分为$n\times m$个格子的矩形阵列。有公共边的格子是相邻格子。当前仓库中有的格子是空闲的，有的格子则已经堆放了沉重的货物。由于堆放

的货物很重,单凭仓库管理员的力量是无法移动的。仓库管理员有一项任务,要将一个小箱子推到指定的格子上。管理员可以在仓库中移动,但不能跨过已经堆放了货物的格子。管理员站在与箱子相对的空闲格子上时,可以做一次推动,把箱子推到相邻的空闲格子。只能向管理员的对面方向推箱子。由于要推动的箱子很重,仓库管理员希望尽量减少推箱子的次数。请设计一算法,使仓库管理员推箱子的次数最少。

5-12 喷漆机器人。F大学开发出一种喷漆机器人Rob,能用指定颜色给一块矩形材料喷漆。Rob每次拿起一种颜色的喷枪,为指定颜色的小矩形区域喷漆。喷漆工艺要求:一个小矩形区域只能在所有紧靠它上方的矩形区域都喷过漆后,才能开始喷漆,且小矩形区域开始喷漆后必须一次性喷完,不能只喷一部分。为Rob编写一个自动喷漆程序,使Rob拿起喷枪的次数最少。