第16章 回 溯

寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。理论上，当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。不过，在实际应用中，很少使用这种方法，因为候选解的数量通常都非常大（比如指数级，甚至是大数阶乘），即便采用最快的计算机也只能解决规模很小的问题。

对候选解进行系统检查的方法有多种，其中回溯和分枝定界法是比较常用的两种方法。按照这两种方法对候选解进行系统检查通常会使问题的求解时间大大减少（无论对于最坏情形还是对于一般情形）。事实上，这些方法可以使我们避免对很大的候选解集合进行检查，同时能够保证算法运行结束时可以找到所需要的解。因此，这些方法通常能够用来求解规模很大的问题。

本章集中阐述回溯方法，这种方法被用来设计货箱装船、背包、最大完备子图、旅行商和电路板排列问题的求解算法。

16.1 算法思想

回溯（backtracking）是一种系统地搜索问题解答的方法。在 5.5.6节求解迷宫老鼠问题时即采用了回溯技术。为了实现回溯，首先需要为问题定义一个解空间（ solution space），这个空间必须至少包含问题的一个解（可能是最优的）。在迷宫老鼠问题中，我们可以定义一个包含从入口到出口的所有路径的解空间；在具有 n 个对象的0/1背包问题中（见13.4节和15.2节），解空间的一个合理选择是 2^n 个长度为 n 的0/1向量的集合，这个集合表示了将0或1分配给x的所有可能方法。当 n=3时，解空间为{(0,0,0)，(0,1,0)，(0,0,1)，(1,0,0)，(0,1,1)，(1,0,1)，(1,1,0)，(1,1,1)}。

下一步是组织解空间以便它能被容易地搜索。典型的组织方法是图或树。图 16-1用图的形式给出了一个3×3迷宫的解空间。从(1,1)点到(3,3)点的每一条路径都定义了3×3迷宫解空间中的一个元素，但由于障碍的设置，有些路径是不可行的。

图16-2用树形结构给出了含三个对象的0/1背包问题的解空间。从 i 层节点到 i+1层节点的一条边上的数字给出了向量 x 中第 i 个分量的值 x_i，从根节点到叶节点的每一条路径定义了解空间中的一个元素。从根节点 A 到叶节点 H 的路径定义了解 x=[1,1,1]。根据 w 和 c 的值，从根到叶的路径中的一些解或全部解可能是不可行的。

一旦定义了解空间的组织方法，这个空间即可按深度优先的方法从开始节点进

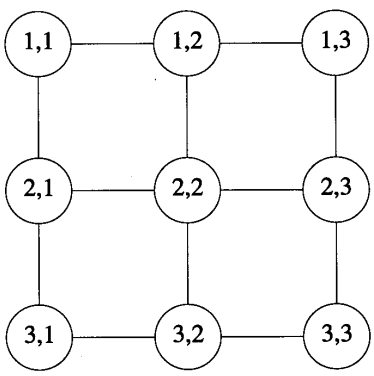

图16-1 3×3迷宫的解空间

行搜索。在迷宫老鼠问题中，开始节点为入口节点 (1,1)；在0/1背包问题中，开始节点为根节点*A*。开始节点既是一个活节点又是一个E-节点（expansion node）。从E-节点可移动到一个新节点。如果能从当前的E-节点移动到一个新节点，那么这个新节点将变成一个活节点和新的E-节点，旧的E-节点仍是一个活节点。如果不能移到一个新节点，当前的E-节点就“死”了（即不再是一个活节点），那么便只能返回到最近被考察的活节点（回溯），这个活节点变成了新的E-节点。当我们已经找到了答案或者回溯尽了所有的活节点时，搜索过程结束。

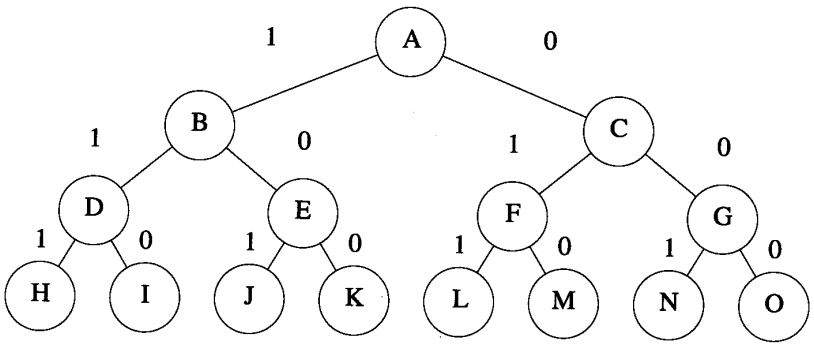

图16-2 三个对象的背包问题的解空间

例16-1 [迷宫老鼠] 考察图16-3a 的矩阵中给出的3×3的“迷宫老鼠”问题。我们将利用图16-1给出的解空间图来搜索迷宫。

从迷宫的入口到出口的每一条路径都与图 16-1中从(1,1)到(3,3)的一条路径相对应。然而，图16-1中有些从(1,1)到(3,3)的路径却不是迷宫中从入口到出口的路径。

搜索从点(1,1)开始，该点是目前唯一的活节点，它也是一个 E-节点。为避免再次走过这个位置，置*maze*(1,1)为1。从这个位置，能移动到(1,2)或(2,1)两个位置。对于本例，两种移动都是可行的，因为在每一个位置都有一个值0。假定选择移动到(1,2)，*maze*(1,2)被置为1以避免再次经过该点。迷宫当前状态如图16-3b 所示。这时有两个活节点(1,1) (1,2)。(1,2)成为E-节点。在图16-1中从当前E-节点开始有3个可能的移动，其中两个是不可行的，因为迷宫在这些位置上的值为1。唯一可行的移动是(1,3)。移动到这个位置，并置*maze*(1,3)为1以避免再次经过该点，此时迷宫状态为16-3c。图16-1中，从(1,3)出发有两个可能的移动，但没有一个是可行的。所以E-节点(1,3)死亡，回溯到最近被检查的活节点 (1,2)。在这个位置也没有可行的移动，故这个节点也死亡了。唯一留下的活节点是 (1,1)。这个节点再次变为E-节点，它可移动到(2,1)。现在活节点为(1,1)，(2,1)。继续下去，能到达点 (3,3)。此时，活节点表为 (1,1)，(2,1)，(3,1)，(3,2)，(3,3)，这即是到达出口的路径。

程序5-13是一个在迷宫中寻找路径的回溯算法。

0 0 0	1 1 0	1 1 1
0 1 1	0 1 1	0 1 1
0 0 0	0 0 0	0 0 0
a)	b)	c)

图16-3 迷宫

例16-2 [0/1背包问题] 考察如下背包问题：n=3，w=[20,15,15]，p=[40,25,25]且c=30。从根节点开始搜索图16-2中的树。根节点是当前唯一的活节点，也是E-节点，从这里能够移动到B或C点。假设移动到B，则活节点为A和B。B是当前E-节点。在节点B，剩下的容量r为10，而收益cp为40。从B点，能移动到D或E。移到D是不可行的，因为移到D所需的容量w_2为15。到E的移动是可行的，因为在这个移动中没有占用任何容量。 E变成新的E-节点。这时活节点为A,B,E。在节点E，r=10，cp=40。从E，有两种可能移动（到J 和K），到J 的移动是不可行的，而到K的移动是可行的。节点K变成了新的E-节点。因为K是一个叶子，所以得到一个可行的解。这个解的收益为cp=40。x 的值由从根到K的路径来决定。这个路径（A,B,E,K）也是此时的活节点序列。既然不能进一步扩充K，K节点死亡，回溯到E，而E也不能进一步扩充，它也死亡了。

接着，回溯到B，它也死亡了， A再次变为E-节点。它可被进一步扩充，到达节点C。此时r=30，cp=0。从C点能够移动到F或G。假定移动到F。F变为新的E-节点并且活节点为A, C, F。在F，r=15，cp=25。从F点，能移动到L或M。假定移动到L。此时r=0，cp=50。既然L是一个叶节点，它表示了一个比目前找到的最优解（即节点K）更好的可行解，我们把这个解作为最优解。节点L死亡，回溯到节点F。继续下去，搜索整棵树。在搜索期间发现的最优解即为最后的解。

例16-3 [旅行商问题] 在这个问题中，给出一个n 顶点网络（有向或无向），要求找出一个包含所有n 个顶点的具有最小耗费的环路。任何一个包含网络中所有 n 个顶点的环路被称作一个旅行（tour）。在旅行商问题中，要设法找到一条最小耗费的旅行。

图16-4给出了一个四顶点网络。在这个网络中，一些旅行如下： 1,2,4,3,1；1,3,2,4,1和1,4,3,2,1。旅行2,4,3,1,2；4,3,1,2,4和3,1,2,4,3和旅行1,2,4,3,1一样。而旅行1,3,4,2,1是旅行1,2,4,3,1的“逆”。旅行1,2,4,3,1的耗费为66；而1,3,2,4,1的耗费为25；1,4,3,2,1为59。故1,3,2,4,1是该网络中最小耗费的旅行。

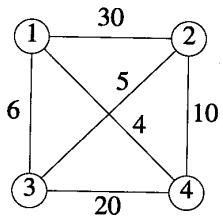

图16-4 一个四顶点网络

顾名思义，旅行商问题可被用来模拟现实生活中旅行商所要旅行的地区问题。顶点表示旅行商所要旅行的城市（包括起点）。边的耗费给出了在两个城市旅行所需的时间（或花费）。旅行表示当旅行商游览了所有城市再回到出发点时所走的路线。

旅行商问题还可用来模拟其他问题。假定要在一个金属薄片或印刷电路板上钻许多孔。孔的位置已知。这些孔由一个机器钻头来钻，它从起始位置开始，移动到每一个钻孔位置钻孔，然后回到起始位置。总共花的时间是钻所有孔的时间与钻头移动的时间。钻所有孔所需的时间独立于钻孔顺序。然而，钻头移动时间是钻头移动距离的函数。因此，希望找到最短的移动路径。

另有一个例子，考察一个批量生产的环境，其中有一个特殊的机器可用来生产 n 个不同的产品。利用一个生产循环不断地生产这些产品。在一个循环中，所有n 个产品被顺序生产出来，然后再开始下一个循环。在下一个循环中，又采用了同样的生产顺序。例如，如果这台机器被用来顺序为小汽车喷红、白、蓝漆，那么在为蓝色小汽车喷漆之后，我们又开始了新一轮循环，为红色小汽车喷漆，然后是白色小汽车、蓝色小汽车、红色小汽车，……，如此下去。一个循

环的花费包括生产一个循环中的产品所需的花费以及循环中从一个产品转变到另一个产品的花费。虽然生产产品的花费独立于产品生产顺序，但循环中从生产一个产品转变到生产另一个产品的花费却与顺序有关。为了使耗费最小化，可以定义一个有向图，图中的顶点表示产品，边<(*i*,*j*)>上的耗费值为生产过程中从产品*i* 转变到产品*j* 所需的耗费。一个最小耗费的旅行定义了一个最小耗费的生产循环。

既然旅行是包含所有顶点的一个循环，故可以把任意一个点作为起点（因此也是终点）。针对图16-4，任意选取点1作为起点和终点，则每一个旅行可用顶点序列 1, v_2,…, v_n, 1来描述，v_2, …, v_n 是(2, 3, …, *n*) 的一个排列。可能的旅行可用一个树来描述，其中每一个从根到叶的路径定义了一个旅行。图16-5给出了一棵表示四顶点网络的树。从根到叶的路径中各边的标号定义了一个旅行（还要附加1作为终点）。例如，到节点L的路径表示了旅行1,2,3,4,1，而到节点O的路径表示了旅行1,3,4,2,1。网络中的每一个旅行都由树中的一条从根到叶的确定路径来表示。因此，树中叶的数目为(*n*-1) !。

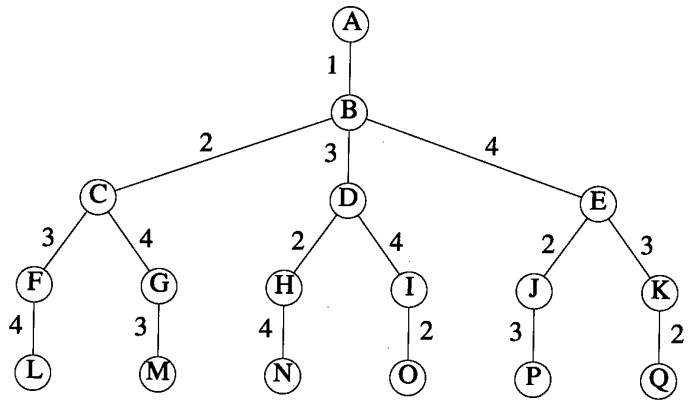

图16-5 四顶点网络的解空间树

回溯算法将用深度优先方式从根节点开始，通过搜索解空间树发现一个最小耗费的旅行。对图16-4的网络，利用图16-5的解空间树，一个可能的搜索为 ABCFL。在L点，旅行1,2,3,4,1作为当前最好的旅行被记录下来。它的耗费是 59。从L点回溯到活节点F。由于F没有未被检查的孩子，所以它成为死节点，回溯到 C点。C变为E - 节点，向前移动到G，然后是M。这样构造出了旅行1,2,4,3,1，它的耗费是66。既然它不比当前的最佳旅行好，抛弃它并回溯到 G，然后是C,B。从B点，搜索向前移动到D，然后是H,N。这个旅行1,3,2,4,1的耗费是25，比当前的最佳旅行好，把它作为当前的最好旅行。从 N点，搜索回溯到H，然后是D。在D点，再次向前移动，到达O点。如此继续下去，可搜索完整个树，得出1,3,2,4,1是最少耗费的旅行。

当要求解的问题需要根据*n* 个元素的一个子集来优化某些函数时，解空间树被称作子集树（subset tree）。所以对有*n* 个对象的0/1背包问题来说，它的解空间树就是一个子集树。这样一棵树有2^n 个叶节点，全部节点有$2^{n+1}-1$个。因此，每一个对树中所有节点进行遍历的算法都必须耗时$\Omega(2^n)$。当要求解的问题需要根据一个 *n* 元素的排列来优化某些函数时，解空间树被称作排列树（permutation tree）。这样的树有*n*! 个叶节点，所以每一个遍历树中所有节点的算法都必须耗时$\Omega(n!)$。图16-5中的树是顶点{2,3,4}的最佳排列的解空间树，顶点1是旅行的起点和终点。

通过确定一个新近到达的节点能否导致一个比当前最优解还要好的解，可加速对最优解的搜索。如果不能，则移动到该节点的任何一个子树都是无意义的，这个节点可被立即杀死，用来杀死活节点的策略称为限界函数（bounding function）。在例16-2中，可使用如下限界函数：杀死代表不可行解决方案的节点；对于旅行商问题，可使用如下限界函数：如果目前建立的部分旅行的耗费不少于当前最佳路径的耗费，则杀死当前节点。如果在图 16-4的例子中使用该限界函数，那么当到达节点I时，已经找到了具有耗费25的1,3,2,4,1的旅行。在节点I，部分旅行1,3,4的耗费为26，若旅行通过该节点，那么不能找到一个耗费小于 25的旅行，故搜索以I为根节点的子树毫无意义。

小结

回溯方法的步骤如下：

1) 定义一个解空间，它包含问题的解。

2) 用适于搜索的方式组织该空间。

3) 用深度优先法搜索该空间，利用限界函数避免移动到不可能产生解的子空间。

回溯算法的一个有趣的特性是在搜索执行的同时产生解空间。在搜索期间的任何时刻，仅保留从开始节点到当前E - 节点的路径。因此，回溯算法的空间需求为O（从开始节点起最长路径的长度）。这个特性非常重要，因为解空间的大小通常是最长路径长度的指数或阶乘。所以如果要存储全部解空间的话，再多的空间也不够用。

练习

1. 考察如下0/1背包问题：n=4，w=[20,25,15,35]，p=[40,49,25,60]，c=62。

1) 画出该0/1背包问题的解空间树。

2) 对该树运用回溯算法（利用给出的 *ps*,*ws*,*c*值），依回溯算法遍历节点的顺序标记节点。确定回溯算法未遍历的节点。

2. 1) 当n=5时，画出旅行商问题的解空间树。

2) 在该树上，运用回溯算法（使用图16-6的例子）。依回溯算法遍历节点的顺序标记节点。确定未被遍历的节点。

3. 每周六，Mary 和Joe 都在一起打乒乓球。她们每人都有一个装有 120个球的篮子。这样一直打下去，直到两个篮子为空。然后她们需要从球桌周围拾起 240个球，放入各自的篮子。Mary 只拾她这边的球，而 Joe 拾剩下的球。描述如何用旅行商问题帮助 Mary 和 Joe 决定她们拾球的顺序以便她们能走最少的路径。

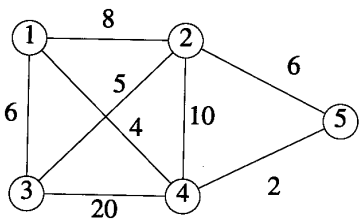

图16-6 练习2的实例

16.2 应用

16.2.1 货箱装船

1. 问题

在13.3节中，考察了用最大数量的货箱装船的问题。现在对该问题做一些改动。在新问题中，有两艘船，n 个货箱。第一艘船的载重量是c_1，第二艘船的载重量是c_2，w_i 是货箱i 的重量

且$\sum_{i=1}^{n} w_i \le c_1+c_2$。我们希望确定是否有一种可将所有 n 个货箱全部装船的方法。若有的话，找出该方法。

例16-4 当n=3，$c_1=c_2$=50，w=[10,40,40] 时，可将货箱1,2装到第一艘船上，货箱3装到第二艘船上。如果w=[20,40,40]，则无法将货箱全部装船。

当$\sum_{i=1}^{n} w_i = c_1+c_2$时，两艘船的装载问题等价于子集之和（sum-of-subset）问题，即有n个数字，要求找到一个子集（如果存在的话）使它的和为c_1。当$c_1=c_2$且$\sum_{i=1}^{n} w_i = 2c_1$时，两艘船的装载问题等价于分割问题（partition problem），即有n个数字a_i,$(1 \le i \le n)$，要求找到一个子集（若存在的话），使得子集之和为$\left(\sum_{i=1}^{n} a_i\right)/2$。分割问题和子集之和问题都是NP - 复杂问题。而且即使问题被限制为整型数字，它们仍是NP - 复杂问题。所以不能期望在多项式时间内解决两艘船的装载问题。

当存在一种方法能够装载所有n个货箱时，可以验证以下的装船策略可以获得成功：1) 尽可能地将第一艘船装至它的重量极限；2) 将剩余货箱装到第二艘船。为了尽可能地将第一艘船装满，需要选择一个货箱的子集，它们的总重量尽可能接近c_1。这个选择可通过求解0/1背包问题来实现，即寻找$\max\left(\sum_{i=1}^{n} w_i x_i\right)$，其中$\sum_{i=1}^{n} w_i x_i \le c_1$，$x_i \in \{0,1\}$，$1 \le i \le n$。

当重量是整数时，可用15.2节的动态规划方法确定第一艘船的最佳装载。用元组方法所需时间为$O(\min\{c_1, 2^n\})$。可以使用回溯方法设计一个复杂性为$O(2^n)$的算法，在有些实例中，该方法比动态规划算法要好。

2. 第一种回溯算法

既然想要找到一个重量的子集，使子集之和尽量接近c_1，那么可以使用一个子集空间，并将其组织成如图16-2那样的二叉树。可用深度优先的方法搜索该解空间以求得最优解。使用限界函数去阻止不可能获得解答的节点的扩张。如果 Z是树的j+1层的一个节点，那么从根到O的路径定义了x_i（$1 \le i \le j$）的值。使用这些值，定义cw（当前重量）为$\sum_{i=1}^{n} w_i x_i$。若$cw > c_1$，则以O为根的子树不能产生一个可行的解答。可将这个测试作为限界函数。当且仅当一个节点的 cw值大于c_1时，定义它是不可行的。

例16-5 假定n=4，w=[8,6,2,3]，c_1=12。解空间树为图16-2的树再加上一层节点。搜索从根A开始且cw=0。若移动到左孩子B则cw=8，$cw \le c_1$=12。以B为根的子树包含一个可行的节点，故移动到节点B。从节点B不能移动到节点D，因为$cw+w_2 > c_1$。移动到节点E，这个移动未改变cw。下一步为节点J，cw=10。J的左孩子的cw值为13，超出了c_1，故搜索不能移动到J的左孩子。可移动到J的右孩子，它是一个叶节点。至此，已找到了一个子集，它的 cw=10。x_i 的值由从A到J的右孩子的路径获得，其值为[1,0,1,0]。

回溯算法接着回溯到J，然后是E。从E，再次沿着树向下移动到节点K，此时cw=8。移动到它的左子树，有cw=11。既然已到达了一个叶节点，就看是否cw的值大于当前的最优cw 值。结果确实大于最优值，所以这个叶节点表示了一个比 [1,0,1,0]更好的解决方案。到该节点的路径决定了x 的值[1,0,0,1]。

从该叶节点，回溯到节点K，现在移动到K的右孩子，一个具有cw=8的叶节点。这个叶节点中没有比当前最优cw 值还好的cw 值，所以回溯到K,E,B直到A。从根节点开始，沿树继续向

下移动。算法将移动到C并搜索它的子树。

当使用前述的限界函数时，便产生了程序16-1所示的回溯算法。函数MaxLoading返回≤c的最大子集之和，但它不能找到产生该和的子集。后面将改进代码以便找到这个子集。MaxLoading调用了一个递归函数maxLoading，它是类Loading的一个成员，定义Loading类是为了减少MaxLoading中的参数个数。maxLoading(1) 实际执行解空间的搜索。MaxLoading(i) 搜索以i层节点（该节点已被隐式确定）为根的子树。从根到该节点的路径定义的子解答有一个重量值cw，目前最优解答的重量为bestw，这些变量以及与类Loading的一个成员相关联的其他变量，均由MaxLoading初始化。

程序16-1 第一种回溯算法

```
template<class T>
class Loading {
  friend MaxLoading(T [], T, int);
  private:
    void maxLoading(int i);
    int n;      // 货箱数目
    T *w,       // 货箱重量数组
      c,        // 第一艘船的容量
      cw,       // 当前装载的重量
      bestw;    // 目前最优装载的重量
};

template<class T>
void Loading<T>::maxLoading(int i)
{// 从第 i 层节点搜索
  if (i > n) {// 位于叶节点
    if (cw > bestw) bestw = cw;
    return;}
  // 检查子树
  if (cw + w[i] <= c) {// 尝试x[i] = 1
    cw += w[i];
    maxLoading(i+1);
    cw -= w[i];}
  maxLoading(i+1);// 尝试x[i] = 0
}

template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载的重量
  Loading<T> X;
  // 初始化X
  X.w = w;
  X.c = c;
  X.n = n;
  X.bestw = 0;
  X.cw = 0;

  // 计算最优装载的重量
```

```
    X.maxLoading(1);
    return X.bestw;
}
```

如果i＞n，则到达了叶节点。被叶节点定义的解答有重量 cw，它一定≤c，因为搜索不会移动到不可行的节点。若 cw>bestw，则目前最优解答的值被更新。当 i≤n 时，我们处在有两个孩子的节点Z上。左孩子表示x[i]=1的情况，只有cw+w[i]≤c 时，才能移到这里。当移动到左孩子时，cw 被置为cw+w[i]，且到达一个i+1层的节点。以该节点为根的子树被递归搜索。当搜索完成时，回到节点Z。为了得到Z的cw 值，需用当前的cw 值减去w[i]，Z的右子树还未搜索。既然这个子树表示x[i]=0的情况，所以无需进行可行性检查就可移动到该子树，因为一个可行节点的右孩子总是可行的。

注意：解空间树未被maxLoading显示构造。函数maxLoading在它到达的每一个节点上花费$\Theta(1)$时间。到达的节点数量为 $O(2^n)$，所以复杂性为 $O(2^n)$。这个函数使用的递归栈空间为$\Theta(n)$。

3. 第二种回溯方法

通过不移动到不可能包含比当前最优解还要好的解的右子树，能提高函数 maxLoading的性能。令bestw为目前最优解的重量，Z为解空间树的第i层的一个节点，cw的定义如前。以Z为根的子树中没有叶节点的重量会超过cw+r，其中$r=\sum_{j=i+1}^{n} w[j]$为剩余货箱的重量。因此，当cw+r≤bestw时，没有必要去搜索Z的右子树。

例16-6 令n, w, c_1 的值与例16-5中相同。用新的限界函数，搜索将像原来那样向前进行直至到达第一个叶节点J（它是J的右孩子）。bestw 被置为10。回溯到E，然后向下移动到K的左孩子，此时bestw被更新为11。我们没有移动到 K的右孩子，因为在右孩子节点 cw=8，r=0，cw+r≤bestw。回溯到节点A。同样，不必移动到右孩子C，因为在C点cw=0，r=11且cw+r≤bestw。

加强了条件的限界函数避免了对A的右子树及K的右子树的搜索。

当使用加强了条件的限界函数时，可得到程序 16-2的代码。这个代码将类型为 T的私有变量r 加到了类Loading的定义中。新的代码不必检查是否一个到达的叶节点有比当前最优解还优的重量值。这样的检查是不必要的，因为加强的限界函数不允许移动到不能产生较好解的节点。因此，每到达一个新的叶节点就意味着找到了比当前最优解还优的解。虽然新代码的复杂性仍是$O(2^n)$，但它可比程序16-1少搜索一些节点。

程序16-2 程序16-1的优化

```
template<class T>
void Loading<T>::maxLoading(int i)
{// // 从第 i 层节点搜索
  if (i > n) {// 在叶节点上
    bestw = cw;
    return;}
  // 检查子树
  r -= w[i];
  if (cw + w[i] <= c) {//尝试x[i] = 1
    cw += w[i];
    maxLoading(i+1);
```

```
    cw -= w[i];}
  if (cw + r > bestw) //尝试x[i] = 0
    maxLoading(i+1);
  r += w[i];
}

template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载的重量
  Loading<T> X;
  // 初始化 X
  X.w = w;
  X.c = c;
  X.n = n;
  X.bestw = 0;
  X.cw = 0;
  // r的初始值为所有重量之和
  X.r = 0;
  for (int i = 1; i <= n; i++)
    X.r += w[i];

  // 计算最优装载的重量
  X.maxLoading(1);
  return X.bestw;
}
```

4. 寻找最优子集

为了确定具有最接近c 的重量的货箱子集，有必要增加一些代码来记录当前找到的最优子集。为了记录这个子集，将参数bestx 添加到Maxloading 中。bestx 是一个整数数组，其中元素可为0或1，当且仅当bestx[i]=1时，货箱i 在最优子集中。新的代码见程序16-3。

程序16-3 给出最优装载的代码

```
template<class T>
void Loading<T>::maxLoading(int i)
{//从第 i 层节点搜索
  if (i > n) {// 在叶节点上
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestw = cw; return;}
  // 检查子树
  r -= w[i];
  if (cw + w[i] <= c) {//尝试 x[i] = 1
    x[i] = 1;
    cw += w[i];
    maxLoading(i+1);
    cw -= w[i];}
  if (cw + r > bestw) {//尝试x[i] = 0
    x[i] = 0;
    maxLoading(i+1);}
```

```
    r += w[i];
  }

  template<class T>
  T MaxLoading(T w[], T c, int n, int bestx[])
  {// 返回最优装载及其值
    Loading<T> X;
    // 初始化 X
    X.x = new int [n+1];
    X.w = w;
    X.c = c;
    X.n = n;
    X.bestx = bestx;
    X.bestw = 0;
    X.cw = 0;
    // r的初始值为所有重量之和
    X.r = 0;
    for (int i = 1; i <= n; i++)
      X.r += w[i];
    X.maxLoading(1);
    delete [] X.x;
    return X.bestw;
  }
```

这段代码在Loading中增加了两个私有数据成员： x 和bestx。这两个私有数据成员都是整型的一维数组。数组x 用来记录从搜索树的根到当前节点的路径（即它保留了路径上的 x_i 值），bestx记录当前最优解。无论何时到达了一个具有较优解的叶节点， bestx 被更新以表示从根到叶的路径。为1的x_i 值确定了要被装载的货箱。数据x 的空间由MaxLoading 分配。

因为bestx 可以被更新$O(2^n)$次，故maxLoading 的复杂性为$O(n2^n)$。使用下列方法之一， 复杂性可降为$O(2^n)$：

1) 首先运行程序16-2的代码，以决定最优装载重量，令其为 W。然后运行程序16-3的一个修改版本。该版本以bestw=W开始运行，当cw+r≥bestw时搜索右子树，第一次到达一个叶节点时便终止（即i>n）。

2) 修改程序16-3的代码以不断保留从根到当前最优叶的路径。尤其当位于 i 层节点时，则到最优叶的路径由x[j]（$1 \le j < i$）和bestx[j]（j≤i≤n）给出。按照这种方法，算法每次回溯一级，并在bestx中存储一个x[i]。由于算法回溯所需时间为$O(2^n)$，因此额外开销为$O(2^n)$。

5. 一个改进的迭代版本

可改进程序16-3的代码以减少它的空间需求。因为数组x 中记录可在树中移动的所有路径，故可以消除大小为$\Theta(n)$的递归栈空间。如例16-5所示，从解空间树的任何节点，算法不断向左孩子移动，直到不能再移动为止。如果一个叶子已被到达，则最优解被更新。否则，它试图移动到右孩子。当要么到达一个叶节点，要么不值得移动到一个右孩子时，算法回溯到一个节点，条件是从该节点向其右孩子移动有可能找到最优解。这个节点有一个特性，即它是路径中具有x[i]=1的节点中离根节点最近的节点。如果向右孩子的移动是有效的，那么就这么做，然后再完成一系列向左孩子的移动。如果向右孩子的移动是无效的，则回溯到 x[i]=1的下一个节点。该算法遍历树的方式可被编码成与程序 16-4一样的迭代（即循环）算法。不像递归代码，这种

代码在检查是否该向右孩子移动之前就移动到了右孩子。如果这个移动是不可行的，则回溯。迭代代码的时间复杂性与程序16-3一样。

程序16-4 迭代代码

```
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最佳装载及其值
 // 迭代回溯程序
  // 初始化根节点
  int i = 1;  // 当前节点的层次
  // x[1:i-1] 是到达当前节点的路径
  int *x = new int [n+1];
  T bestw = 0,  // 迄今最优装载的重量
    cw = 0,     // 当前装载的重量
    r = 0;      // 剩余货箱重量的和
  for (int j = 1; j <= n; j++)
    r += w[j];

  // 在树中搜索
  while (true) {
    // 下移，尽可能向左
    while (i <= n && cw + w[i] <= c) {
      // 移向左孩子
      r -= w[i];
      cw += w[i];
      x[i] = 1;
      i++;
      }

    if (i > n) {// 到达叶子
      for (int j = 1; j <= n; j++)
        bestx[j] = x[j];
      bestw = cw;}
    else {// 移向右孩子
        r -= w[i];
        x[i] = 0;
        i++;}

    // 必要时返回
    while (cw + r <= bestw) {
      // 本子树没有更好的叶子，返回
      i--;
      while (i > 0 && !x[i]) {
        // 从一个右孩子返回
        r += w[i];
        i--;
        }

      if (i == 0) {delete [] x;
```

```
                return bestw;}

            // 移向右子树
            x[i] = 0;
            cw -= w[i];
            i++;
            }
          }
      }
```

16.2.2 0/1背包问题

0/1背包问题是一个NP - 复杂问题，为了解决该问题，在13.4节采用了贪婪算法，在15.2节又采用了动态规划算法。在本节，将用回溯算法解决该问题。既然想选择一个对象的子集，将它们装入背包，以便获得的收益最大，则解空间应组织成子集树的形状（如图 16-2所示）。该回溯算法与16.2节的装载问题很类似。首先形成一个递归算法，去找到可获得的最大收益。然后，对该算法加以改进，形成代码。改进后的代码可找到获得最大收益时包含在背包中的对象的集合。

与程序16-2一样，左孩子表示一个可行的节点，无论何时都要移动到它；当右子树可能含有比当前最优解还优的解时，移动到它。一种决定是否要移动到右子树的简单方法是令 r 为还未遍历的对象的收益之和，将 r 加到 cp（当前节点所获收益）之上，若（$r+cp$）≤$bestp$（目前最优解的收益），则不需搜索右子树。一种更有效的方法是按收益密度 p_i/w_i 对剩余对象排序，将对象按密度递减的顺序去填充背包的剩余容量，当遇到第一个不能全部放入背包的对象时，就使用它的一部分。

例16-7 考察一个背包例子：n=4，c=7，p=[9,10,7,4]，w=[3,5,2,1]。这些对象的收益密度为[3,2,3.5,4]。当背包以密度递减的顺序被填充时，对象 4首先被填充，然后是对象3、对象1。在这三个对象被装入背包之后，剩余容量为 1。这个容量可容纳对象2的0.2倍的重量。将0.2倍的该对象装入，产生了收益值2。被构造的解为x=[1,0.2,1,1]，相应的收益值为22。尽管该解不可行（x_2是0.2，而实际上它应为0或1），但它的收益值22一定不少于要求的最优解。因此，该0/1背包问题没有收益值多于22的解。

解空间树为图16-2再加上一层节点。当位于解空间树的节点 B时，x_1=1，目前获益为cp=9。该节点所用容量为 cw=3。要获得最好的附加收益，要以密度递减的顺序填充剩余容量 $cleft$=c-cw=4。也就是说，先放对象 4，然后是对象3，然后是对象2的0.2倍的重量。因此，子树A的最优解的收益值至多为22。

当位于节点C时，cp=cw=0，$cleft$=7。按密度递减顺序填充剩余容量，则对象 4和3被装入。然后是对象2的0.8倍被装入。这样产生出收益值 19。在子树C中没有节点可产生出比 19还大的收益值。

在节点E，cp=9，cw=3，$cleft$=4。仅剩对象3和4要被考虑。当对象按密度递减的顺序被考虑时，对象4先被装入，然后是对象 3。所以在子树E中无节点有多于cp+4+7=20的收益值。如果已经找到了一个具有收益值20或更多的解，则无必要去搜索E子树。

一种实现限界函数的好方法是首先将对象按密度排序。假定已经做了这样的排序。定义类Knap（见程序16-5）来减少限界函数Bound（见程序16-6）及递归函数Knapsack（见程序16-7）

的参数数量，该递归函数用于计算最优解的收益值。参数的减少又可引起递归栈空间的减少以及每一个Knapsack的执行时间的减少。注意函数Knapsack和函数maxLoading（见程序16-2）的相似性。同时注意仅当向右孩子移动时，限界函数才被计算。当向左孩子移动时，左孩子的限界函数的值与其父节点相同。

程序16-5 Knap类

```
template<class Tw, class Tp>
class Knap {
  friend Tp Knapsack(Tp *, Tw *, Tw, int);
  private:
    Tp Bound(int i);
    void Knapsack(int i);
    Tw c;       //背包容量
    int n;      // 对象数目
    Tw *w;      // 对象重量的数组
    Tp *p;      // 对象收益的数组
    Tw cw;      // 当前背包的重量
    Tp cp;      // 当前背包的收益
    Tp bestp;   // 迄今最大的收益
};
```

程序16-6 限界函数

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>::Bound(int i)
{// 返回子树中最优叶子的上限值Return upper bound on value of
 // best leaf in subtree.
  Tw cleft = c - cw;  // 剩余容量
  Tp b = cp;          // 收益的界限
  // 按照收益密度的次序装填剩余容量
  while (i <= n && w[i] <= cleft) {
    cleft -= w[i];
    b += p[i];
    i++;
    }

  // 取下一个对象的一部分
  if (i <= n) b += p[i]/w[i] * cleft;
  return b;
}
```

程序16-7 0/1背包问题的迭代函数

```
template<class Tw, class Tp>
void Knap<Tw, Tp>::Knapsack(int i)
{// 从第 i 层节点搜索
  if (i > n) {// 在叶节点上
    bestp = cp;
```

```
      return;}
    // 检查子树
    if (cw + w[i] <= c) {//尝试x[i] = 1
      cw += w[i];
      cp += p[i];
      Knapsack(i+1);
      cw -= w[i];
      cp -= p[i];}
    if (Bound(i+1) > bestp) // 尝试x[i] = 0
      Knapsack(i+1);
  }
```

在执行程序16-7的函数Knapsack之前，需要按密度对对象排序，也要确保对象的重量总和超出背包的容量。为了完成排序，定义了类 Object（见程序16-8）。注意定义操作符 <=是为了使归并排序程序（见程序14-3）能按密度递减的顺序排序。

程序16-8 Object类

```
class Object {
  friend int Knapsack(int *, int *, int, int);
  public:
    int operator<=(Object a) const
    {return (d >= a.d);}
  private:
    int ID;  // 对象号
    float d; // 收益密度
};
```

程序16-9首先验证重量之和超出背包容量，然后排序对象，在执行 Knap::Knapsack之前完成一些必要的初始化。 Knap:Knapsack的复杂性是$O(n2^n)$，因为限界函数的复杂性为 $O(n)$，且该函数在$O(2^n)$个右孩子处被计算。

程序16-9 程序16-7的预处理代码

```
template<class Tw, class Tp>
Tp Knapsack(Tp p[], Tw w[], Tw c, int n)
{// 返回最优装包的值
  // 初始化
  Tw W = 0;  // 记录重量之和
  Tp P = 0;  // 记录收益之和
  // 定义一个按收益密度排序的对象数组
  Object *Q = new Object [n];

  for (int i = 1; i <= n; i++) {
    // 收益密度的数组
    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;  // 可容纳所有对象

    MergeSort(Q,n);   // 按密度排序

    // 创建Knap的成员
    Knap<Tw, Tp> K;
    K.p = new Tp [n+1];
    K.w = new Tw [n+1];
    for (i = 1; i <= n; i++) {
      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;

    // 寻找最优收益
    K.Knapsack(1);

    delete [] Q;
    delete [] K.w;
    delete [] K.p;
    return K.bestp;
  }
```

16.2.3 最大完备子图

令U为无向图G的顶点的子集，当且仅当对于U中的任意点u 和v，(u,v)是图G的一条边时，U定义了一个完全子图（complete subgraph）。子图的尺寸为图中顶点的数量。当且仅当一个完全子图不被包含在G的一个更大的完全子图中时，它是图 G的一个完备子图。最大的完备子图是具有最大尺寸的完备子图。

例16-8 在图16-7a 中，子集{1,2}定义了一个尺寸为2的完全子图。这个子图不是一个完备子图，因为它被包含在一个更大的完全子图{1,2,5}中。{1,2,5}定义了该图的一个最大的完备子图。点集{1,4,5}和{2,3,5}定义了其他的最大的完备子图。

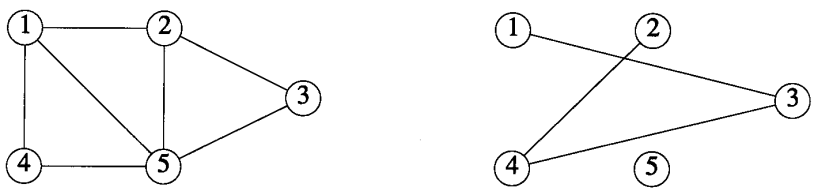

图16-7 图及其补图

a) 图G b) 补图 $\overline{G}$

当且仅当对于U中任意点u 和v，(u,v) 不是G的一条边时，U定义了一个空子图。当且仅当一个子集不被包含在一个更大的点集中时，该点集是图 G的一个独立集（independent set），同时它也定义了图G的空子图。最大独立集是具有最大尺寸的独立集。对于任意图 G，它的补图（complement）$\overline{G}$ 是有同样点集的图，且当且仅当（u,v）不是G的一条边时，它是$\overline{G}$ 的一条边。

例16-9 图16-7b 是图16-7a 的补图，反之亦然。{2,4}定义了16-7a 的一个空子图，它也是该图的一个最大独立集。虽然{1,2}定义了图16-7b 的一个空子图，它不是一个独立集，因为它被包含在空子图{1,2,5}中。{1,2,5}是图16-7b 中的一个最大独立集。

如果U定义了G的一个完全子图，则它也定义了$\overline{G}$ 的一个空子图，反之亦然。所以在 G的完备子图与$\overline{G}$ 的独立集之间有对应关系。特别的， G的一个最大完备子图定义了$\overline{G}$ 的一个最大独立集。

最大完备子图问题是指寻找图G的一个最大完备子图。类似地，最大独立集问题是指寻找图G的一个最大独立集。这两个问题都是 NP - 复杂问题。当用算法解决其中一个问题时，也就解决了另一个问题。例如，如果有一个求解最大完备子图问题的算法，则也能解决最大独立集问题，方法是首先计算所给图的补图，然后寻找补图的最大完备子图。

例16-10 假定有一个n 个动物构成的集合。可以定义一个有n 个顶点的相容图（compatibility graph）G。当且仅当动物u 和v 相容时，（u,v）是G的一条边。G的一个最大完备子图定义了相互间相容的动物构成的最大子集。

15.2节考察了如何找到一个具有最大尺寸的互不交叉的网组的集合问题。可以把这个问题看作是一个最大独立集问题。定义一个图，图中每个顶点表示一个网组。当且仅当两个顶点对应的网组交叉时，它们之间有一条边。所以该图的一个最大独立集对应于非交叉网组的一个最大尺寸的子集。当网组有一个端点在路径顶端，而另一个在底端时，非交叉网组的最大尺寸的子集能在多项式时间（实际上是 $\Theta(n^2)$）内用动态规划算法得到。当一个网组的端点可能在平面中的任意地方时，不可能有在多项式时间内找到非交叉网组的最大尺寸子集的算法。

最大完备子图问题和最大独立集问题可由回溯算法在 $O(n2^n)$时间内解决。两个问题都可使用子集解空间树（如图 16-2所示）。考察最大完备子图问题，该递归回溯算法与程序 16-3非常类似。当试图移动到空间树的 i 层节点Z的左孩子时，需要证明从顶点 i 到每一个其他的顶点j（x_j =1且j 在从根到Z的路径上）有一条边。当试图移动到Z的右孩子时，需要证明还有足够多的顶点未被搜索，以便在右子树有可能找到一个较大的完备子图。

回溯算法可作为类AdjacencyGraph（见程序12-4）的一个成员来实现，为此首先要在该类中加入私有静态成员 x（整型数组，用于存储到当前节点的路径），bestx（整型数组，保存目前的最优解），bestn（bestx中点的数量），cn（x 中点的数量）。所以类AdjacecyGraph的所有实例都能共享这些变量。

函数maxClique（见程序16-10）是类AdjacencyGraph的一个私有成员，而MaxClique是一个共享成员。函数maxClique对解空间树进行搜索，而MaxCique初始化必要的变量。Maxclique(v)的执行返回最大完备子图的尺寸，同时它也设置整型数组 v，当且仅当顶点i 不是所找到的最大完备子图的一个成员时， v[i]=0。

程序16-10 最大完备子图

```
void AdjacencyGraph::maxClique(int i)
```

```
{// 计算最大完备子图的回溯代码
  if (i > n) {// 在叶子上
    // 找到一个更大的完备子图，更新
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestn = cn;
    return;}

  // 在当前完备子图中检查顶点i是否与其它顶点相连
  int OK = 1;
  for (int j = 1; j < i; j++)
    if (x[j] && a[i][j] == NoEdge) {
      // i 不与 j 相连
      OK = 0;
      break;}

  if (OK) {// 尝试 x[i] = 1
    x[i] = 1;  // 把 i 加入完备子图
    cn++;
    maxClique(i+1);
    x[i] = 0;
    cn--;}

  if (cn + n - i > bestn) {// 尝试 x[i] = 0
    x[i] = 0;
    maxClique(i+1);}
}

int AdjacencyGraph::MaxClique(int v[])
{// 返回最大完备子图的大小
 // 完备子图的顶点放入 v[1:n]
  // 初始化
  x = new int [n+1];
  cn = 0;
  bestn = 0;
  bestx = v;

 // 寻找最大完备子图
  maxClique(1);

  delete [] x;
  return bestn;
}
```

16.2.4 旅行商问题

旅行商问题（例 16.3）的解空间是一个排列树。这样的树可用函数 Perm(见程序1-10)搜索，并可生成元素表的所有排列。如果以 x=[1, 2, …, n] 开始，那么通过产生从 x_2 到x_n 的所有排列，可生成 n 顶点旅行商问题的解空间。由于 Perm产生具有相同前缀的所有排列，因此

可以容易地改造 Perm，使其不能产生具有不可行前缀（即该前缀没有定义路径）或不可能比当前最优旅行还优的前缀的排列。注意在一个排列空间树中，由任意子树中的叶节点定义的排列有相同的前缀（如图 16-5所示）。因此，考察时删除特定的前缀等价于搜索期间不进入相应的子树。

旅行商问题的回溯算法可作为类 AdjacencyWDigraph（见程序12-1）的一个成员。在其他例子中，有两个成员函数： tSP和TSP。前者是一个保护或私有成员，后者是一个共享成员。函数G.TSP(v)返回最少耗费旅行的花费，旅行自身由整型数组 v 返回。若网络中无旅行，则返回NoEdge。tSP在排列空间树中进行递归回溯搜索， TSP是其一个必要的预处理过程。 TSP假定x（用来保存到当前节点的路径的整型数组），bestx（保存目前发现的最优旅行的整型数组），cc（类型为T的变量，保存当前节点的局部旅行的耗费），bestc（类型为T的变量，保存目前最优解的耗费）已被定义为 AdjacencyWDigraph中的静态数据成员。 TSP见程序16-11。tSP(2)搜索一棵包含x[2:n]的所有排列的树。

程序16-11 旅行商回溯算法的预处理程序

```
template<class T>
T AdjacencyWDigraph<T>::TSP(int v[])
{// 用回溯算法解决旅行商问题
 // 返回最优旅游路径的耗费，最优路径存入 v[1:n]
   //初始化
   x = new int [n+1];
   // x 是排列
   for (int i = 1; i <= n; i++)
      x[i] = i;
   bestc = NoEdge;
   bestx = v;  // 使用数组 v来存储最优路径
   cc = 0;

   // 搜索x[2:n]的各种排列
   tSP(2);

   delete [] x;
   return bestc;
}
```

函数tSP见程序16-12。它的结构与函数Perm相同。当i=n 时，处在排列树的叶节点的父节点上，并且需要验证从x[n-1] 到x[n] 有一条边，从x[n] 到起点x[1] 也有一条边。若两条边都存在，则发现了一个新旅行。在本例中，需要验证是否该旅行是目前发现的最优旅行。若是，则将旅行和它的耗费分别存入bestx与bestc中。

当i＜n 时，检查当前i-1 层节点的孩子节点，并且仅当以下情况出现时，移动到孩子节点之一：1) 有从x[i-1] 到x[i] 的一条边（如果是这样的话， x[1:i]定义了网络中的一条路径）；2) 路径x[1:i] 的耗费小于当前最优解的耗费。变量cc 保存目前所构造的路径的耗费。

每次找到一个更好的旅行时，除了更新 bestx 的耗费外， tSP需耗时O((*n*-1)!)。因为需发生O((*n*-1)!) 次更新且每一次更新的耗费为 $\Theta(n)$ 时间，因此更新所需时间为O(*n**(*n*-1)!)。通过使用加强的条件（练习16），能减少由tSP搜索的树节点的数量。

程序16-12 旅行商问题的迭代回溯算法

```
void AdjacencyWDigraph<T>::tSP(int i)
{// 旅行商问题的回溯算法
  if (i == n) {// 位于一个叶子的父节点
    // 通过增加两条边来完成旅行
    if (a[x[n-1]][x[n]] != NoEdge &&
      a[x[n]][1] != NoEdge &&
      (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc ||
      bestc == NoEdge)) {// 找到更优的旅行路径
      for (int j = 1; j <= n; j++)
        bestx[j] = x[j];
      bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}
    }
  else {// 尝试子树
    for (int j = i; j <= n; j++)
      //能移动到子树x[j]吗?
      if (a[x[i-1]][x[j]] != NoEdge &&
          (cc + a[x[i-1]][x[i]] < bestc ||
           bestc == NoEdge)) {//能
        // 搜索该子树
        Swap(x[i], x[j]);
        cc += a[x[i-1]][x[i]];
        tSP(i+1);
        cc -= a[x[i-1]][x[i]];
        Swap(x[i], x[j]);}
    }
}
```

16.2.5 电路板排列

在大规模电子系统的设计中存在着电路板排列问题。这个问题的经典形式为将n个电路板放置到一个机箱的许多插槽中，(如图16-8所示)。n个电路板的每一种排列定义了一种放置方法。令$B=\{b_1, \cdots, b_n\}$表示这n个电路板。m个网组集合$L=\{N_1, \cdots, N_m\}$由电路板定义，N_i是B的子集，子集中的元素需要连接起来。实际中用电线将插有这些电路板的插槽连接起来。

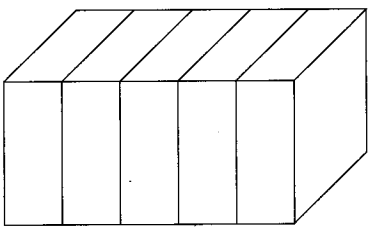

图16-8 电路板及插槽

例16-11 令n=8, m=5。集合B和L如下：

$B=\{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$

$L=\{N_1, N_2, N_3, N_4, N_5\}$

$N_1=\{b_4, b_5, b_6\}$

$N_2=\{b_2, b_3\}$

$N_3=\{b_1, b_3\}$

$N_4=\{b_3, b_6\}$

$N_5=\{b_7, b_8\}$

图16-9给出了电路板的一个可能的排列。边表示在电路板之间的连线。

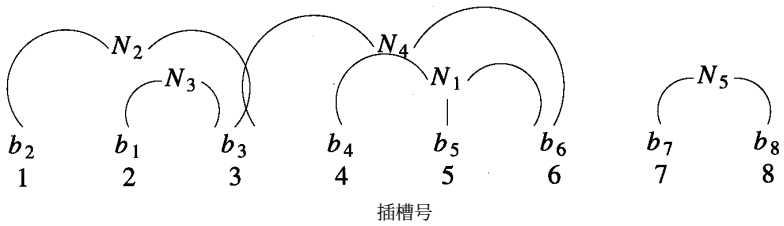

图16-9 电路板布线

令x 为电路板的一个排列。电路板x_i 被放置到机箱的插槽i 中。$density(x)$ 为机箱中任意一对相邻插槽间所连电线数目中的最大值。对于图 16-9中的排列，$density$为2。有两根电线连接了插槽2和3，插槽4和5，插槽5和6。插槽6和7之间无电线，余下的相邻插槽都只有一根电线。

板式机箱被设计成具有相同的相邻插槽间距，因此这个间距决定了机箱的大小。该间距必须保证足够大以便容纳相邻插槽间的连线。因此这个距离（继而机箱的大小）由 $density(x)$ 决定。

电路板排列问题的目标是找到一种电路板的排列方式，使其有最小的 $density$。既然该问题是一个NP - 复杂问题，故它不可能由一个多项式时间的算法来解决，而象回溯这样的搜索方法则是解决该问题的一种较好方法。回溯算法为了找到最优的电路板排列方式，将搜索一个排列空间。

用一个$n\times m$ 的整型数组B表示输入，当且仅当N_j 中包含电路板b_i 时，B[i][j]=1。令total[j]为N_j 中电路板的数量。对于任意部分的电路板排列 x[1:i]，令now[j]为既在x[1:i]中又被包含在N_j 中的电路板的数量。当且仅当 now[j]>0且now[j]≠total[j]时，N_j 在插槽i 和i+1之间有连线。插槽i 和i+1间的线密度可利用该测试方法计算出来。在插槽 k和k+1(1≤k≤i) 间的线密度的最大值给出了局部排列的密度。

为了实现电路板排列问题的回溯算法，使用了类 Board（见程序16-13）。程序16-14给出了私有方法BestOrder，程序16-15给出了函数ArrangeBoards。ArrangeBoards 返回最优的电路板排列密度，最优的排列由数组bestx 返回。

ArrangeBoards 创建类Board 的一个成员x 并初始化与之相关的变量。尤其是total 被初始化以使total[j] 等于N_j 中电路板的数量。now[1:n] 被置为0，与一个空的局部排列相对应。调用 x.BestOrder(1,0) 搜索x[1:n] 的排列树，以从密度为 0的空排列中找到一个最优的排列。通常，x.BestOrder(i,cd) 寻找最优的局部排列x[1:i-1]，该局部排列密度为cd。

函数BestOrder（见程序16-14）和程序16-12有同样的结构，它也搜索一个排列空间。当i=n 时，表示所有的电路板已被放置且cd 为排列的密度。既然这个算法只寻找那些比当前最优排列还优的排列，所以不必验证cd 是否比beste 要小。当i<n 时，排列还未被完成。x[1:i-1] 定义了当前节点的局部排列，而 cd 是它的密度。这个节点的每一个孩子通过在当前排列的末端增加一个电路板而扩充了这个局部排列。对于每一个这样的扩充，新的密度 ld被计算，且只有ld<bestd 的节点被搜索，其他的节点和它们的子树不被搜索。

在排列树的每一个节点处，函数BestOrder花费$\Theta(m)$的时间计算每一个孩子节点的密度。所以计算密度的总时间为$O(mn!)$。此外，产生排列的时间为$O(n!)$ 且更新最优排列的时间为$O(mn)$。

注意每一个更新至少将bestd的值减少1，且最终bestd≥0。所以更新的次数是O(*m*)。BestOrder的整体复杂性为O(*mn*!)。

程序16-13 Board的类定义

```
class Board {
  friend ArrangeBoards(int**, int, int, int []);
  private:
    void BestOrder(int i, int cd);
    int *x,          // 到达当前节点的路径
        *bestx,      // 目前的最优排列
        *total,      // total[j] = 带插槽j的板的数目
        *now,        // now[j] = 在含插槽j的部分排列中的板的数目
        bestd,       // bestx的密度
        n,           //板的数目
        m,           // 插槽的数目
        **B;         // 板的二维数组
};
```

程序16-14 搜索排列树

```
void Board::BestOrder(int i, int cd)
{// 按回溯算法搜索排列树
  if (i == n) {
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestd = cd;}
  else // 尝试子树
    for (int j = i; j <= n; j++) {
      // 用x[j] 作为下一块电路板对孩子进行尝试
      // 在最后一个插槽更新并计算密度
      int ld = 0;
      for (int k = 1; k <= m; k++) {
        now[k] += B[x[j]][k];
        if (now[k] > 0 && total[k] != now[k])
          ld++;
        }

      // 更新 ld 为局部排列的总密度
      if (cd > ld) ld = cd;

      // 仅当子树中包含一个更优的排列时，搜索该子树
      if (ld < bestd) {// 移动到孩子
        Swap(x[i], x[j]);
        BestOrder(i+1, ld);
        Swap(x[i], x[j]);}

      // 重置
      for (k = 1; k <= m; k++)
        now[k] -= B[x[j]][k];
```

```
        }
    }
```

程序16-15 BestOrder(程序16-14)的预处理代码

```
int ArrangeBoards(int **B, int n, int m, int bestx[ ])
{// 返回最优密度
 // 在bestx中返回最优排列
   Board X;
   // 初始化X
   X.x = new int [n+1];
   X.total = new int [m+1];
   X.now = new int [m+1];
   X.B = B;
   X.n = n;
   X.m = m;
   X.bestx = bestx;
   X.bestd = m + 1;

   // 初始化total和now
   for (int i = 1; i <= m; i++) {
      X.total[i] = 0;
      X.now[i] = 0;
      }

   // 初始化x并计算 total
   for (i = 1; i <= n; i++) {
      X.x[i] = i;
      for (int j = 1; j <= m; j++)
         X.total[j] += B[i][j];
      }

   // 寻找最优排列
   X.BestOrder(1,0);

   delete [] X.x;
   delete [] X.total;
   delete [] X.now;
   return X.bestd;
```

练习

4. 证明在两船装载问题中，只要存在一种方法能装载所有货箱，则通过尽可能装满第一艘船就可找到一种可行的装载方法。

5. 运行程序16-3和16-4的代码，测试它们的相对运行时间。

6. 运用16.2.1节中第4小节的方法1) 来更新程序16-3，使其能得到时间复杂性$O(2^n)$。

7. 运用16.2.1节中第4小节的方法2) 来修改程序16-3，使其运行时间减少至$O(2^n)$。

8. 为子集之和问题写一个递归回溯算法。注意，只要找到一个子集，其和为 c_1，则可终止

程序运行。没有必要记住目前的最优解。代码不应使用像程序 16-3中的数组x。在找到和为c_1的子集之后展开递归，可以重构出最优解。

9. 优化程序16-7和16-9以便能产生出与背包问题最优解相对应的一个0/1数组x。

10. 用迭代回溯算法求解0/1背包问题。该算法与程序16-4类似。可以修改Knap::Bound，使其返回被装入背包的最后一个对象i，这样可避免根据Bound重新向左移动而可直接移动到最左节点（原先由Bound确定）。

11. 编写程序16-10（与程序16-4相对应）的迭代版本并比较这两个版本。

12. 改写程序16-10，使其首先按度的递减次序来排列各顶点。你认为该版本比程序 16-10好吗？

13. 编写一个求解最大独立集问题的回溯算法。

14. 重写最大完备子图代码（见程序 16-10），把它作为类 UNetwork的成员。对于类ADjacencyGraph, AdjacencyWGraph, LinkedGraph和LinkedWGraph（见12.7节）的成员，该代码同样有效。

15. 令G为一个n 顶点的有向图，Max_i 为从顶点i 出发的具有最大耗费的边的耗费

1) 证明旅行商的每一个旅行有一个小于 $\sum_{i=1}^{n} Max_i$ +1的耗费。

2) 使用上述界限作为bestc的初始值。重写TSP和tSP，尽可能简化它们。

16. 令G为具有n 个顶点的有向图，$MinOut_i$ 为从顶点i 出发的具有最小耗费的边的耗费

1) 证明具有前缀x_1 到x_i 的旅行商的所有旅行耗费至少为 $\sum_{j=2}^{i} A(x_{j-1}, x_j) + \sum_{y=i}^{n} MinOut_{xj}$ 其中$A(u,v)$是边(u,v)的耗费。

2) 在程序16-12中，使用

```
if (a [x [i-1] ] [x [j] ] ! = NoEdge &&
(cc + a [x [i-1] ] [x [i] ] < bestc | |
bestc == NoEdge) )
```

来决定何时移动到一个孩子节点。要求使用 1) 的结果得到一个更强的条件。第一个和可根据cc 计算出来，通过用一个新变量r 保留不在当前路径中的顶点的MinOut[i]的和，可以很容易地计算出第二个和。

3) 测试tSP的新版本。与程序16-12比较，它访问了排列树的多少节点？

17. 考察电路板排列问题。N_i 的长度为N_i 中第一块和最后一块电路板间的距离。对于图 16-9，N_4中第一个电路板在插槽3中，最后一个电路板在插槽6中，则N_4 的长度为3。N_2 的长度为2。N_i 最大值为3。编写一个回溯算法以找到具有最小的最大长度的板排列。试测试代码的正确性。

18. [顶点覆盖] 令G为一个无向图。当且仅当对于G中的每一条边(u,v)，u 或v 或u,v 在U中时，G的顶点子集U是一个顶点覆盖（vertex cover）。U中顶点的数量是覆盖的大小。在图16-7a中，{1,2,5}是大小为3的一个顶点覆盖。编写一个回溯算法寻找具有最小尺寸的顶点覆盖。算法的复杂性是多少？

19. [简易最大切割] 令G是一个无向图。U是G中顶点的任意子集。V是G余下的点的集合。一个端点在U中，另一个端点在V中的边的数量是U所定义的切割（cut）的大小。编写一个回溯算法，寻找最大切割的大小和相应的U。算法的复杂性是多少？

20. [机器设计] 某机器由n 个部件组成，每一个部件可从3个投资者那里获得。令w_{ij} 是从投资者j 那里得到的零件i 的重量，c_{ij} 则为该零件的耗费。编写一个回溯算法，找出耗费不超过 c 的机器构成方案，使其重量最少。算法的复杂性是多少？

21. [网络设计] 一个汽油传送网络可由加权的有向无环图G表示。G中有一个称为原点的顶点S。从S出发，汽油被输送到图中的其他顶点。S的入度为0，每一条边上的权给出了它所连接的两点间的距离。通过网络输送汽油时，压力的损失是所走距离的函数。为了保证网络的正常运转，在网络传输中必须保证最小压力P_{min}。为了维持这个最小压力，可将压力放大器放在网络中的一些或全部顶点。压力放大器可将压力恢复至最大可允许的量级P_{max}。令d为汽油在压力由P_{max}降为P_{min}时所走的距离。在设置信号放大器问题中，需要放置最少数量的放大器，以便在遇到一个放大器之前汽油所走的距离不超过d。编写一个回溯算法来求解该问题。算法的复杂性是多少？

22. [n皇后问题] 在n皇后问题中，我们希望在$n \times n$的棋盘上找到一个n皇后的放置方法以便任意两个皇后之间不冲突。当且仅当两个皇后在相同的排、列、对角线或反对角线上时，她们之间将发生冲突。假定在任何可行的解决方案中，皇后i被放置在棋盘的第i排。所以只对决定每一个皇后所在的列感兴趣。令c_i为皇后i所处的列。如果任意两个皇后不冲突，则[c_1, …, c_n]是[1, 2, …, n]的一个排列。n皇后问题的解空间因此被限制到[1, 2, …, n]的所有排列中。

1) 将n皇后的解空间组织成一棵树。

2) 编写一个回溯算法，搜索n皇后问题的可行排列。

*23. 编写一个函数，使用回溯算法来搜索一个子集空间树，该树为一个二叉树。函数中的参数应包含如下函数：确定一个节点是否可行的函数，计算该节点的界限值的函数，决定界限是否优于另一个值的函数等。用0/1背包问题来测试程序。

*24. 使用排列空间树来完成练习23。

*25. 编写一个函数，用回溯法搜索一个解空间。函数中的参数应包括下列函数：产生节点的下一个孩子的函数，决定下一个孩子是否是可行的函数，计算该节点界限的函数，决定该界限值是否优于另一个值的函数等。用0/1背包问题来测试程序。