第17章 分 枝 定 界

任何美好的事情都有结束的时候。现在我们学习的是本书的最后一章。幸运的是，本章用到的大部分概念在前面各章中已作了介绍。类似于回溯法，分枝定界法在搜索解空间时，也经常使用树形结构来组织解空间（常用的树结构是第 16章所介绍的子集树和排列树）。然而与回溯法不同的是，回溯算法使用深度优先方法搜索树结构，而分枝定界一般用宽度优先或最小耗费方法来搜索这些树。本章与第16章所考察的应用完全相同，因此，可以很容易比较回溯法与分枝定界法的异同。

相对而言，分枝定界算法的解空间比回溯法大得多，因此当内存容量有限时，回溯法成功的可能性更大。

17.1 算法思想

分枝定界（branch and bound）是另一种系统地搜索解空间的方法，它与回溯法的主要区别在于对E-节点的扩充方式。每个活节点有且仅有一次机会变成 E-节点。当一个节点变为E-节点时，则生成从该节点移动一步即可到达的所有新节点。在生成的节点中，抛弃那些不可能导出（最优）可行解的节点，其余节点加入活节点表，然后从表中选择一个节点作为下一个E-节点。从活节点表中取出所选择的节点并进行扩充，直到找到解或活动表为空，扩充过程才结束。

有两种常用的方法可用来选择下一个E-节点（虽然也可能存在其他的方法）：

1) 先进先出（FIFO） 即从活节点表中取出节点的顺序与加入节点的顺序相同，因此活节点表的性质与队列相同。

2) 最小耗费或最大收益法 在这种模式中，每个节点都有一个对应的耗费或收益。如果查找一个具有最小耗费的解，则活节点表可用最小堆来建立，下一个 E-节点就是具有最小耗费的活节点；如果希望搜索一个具有最大收益的解，则可用最大堆来构造活节点表，下一个 E-节点是具有最大收益的活节点。

例17-1 [迷宫老鼠] 考察图16-3a 给出的迷宫老鼠例子和图16-1的解空间结构。使用FIFO分枝定界，初始时取（1，1）作为E-节点且活动队列为空。迷宫的位置（1,1）被置为1，以免再次返回到这个位置。（1，1）被扩充，它的相邻节点（1，2）和（2，1）加入到队列中（即活节点表）。为避免再次回到这两个位置，将位置（ 1，2）和（2，1）置为1。此时迷宫如图17-1a所示，E-节点（1，1）被删除。

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

图17-1 迷宫问题的FIFO分枝定界方法

节点（1，2）从队列中移出并被扩充。检查它的三个相邻节点（见图16-1的解空间），只有（1，3）是可行的移动（剩余的两个节点是障碍节点），将其加入队列，并把相应的迷宫位置置为1，所得到的迷宫状态如图17-1b 所示。节点（1，2）被删除，而下一个E-节点（2，1）将会被取出，当此节点被展开时，节点（3，1）被加入队列中，节点（3，1）被置为1，节点（2，1）被删除，所得到的迷宫如图17-1c 所示。此时队列中包含（1，3）和（3，1）两个节点。随后节点（1，3）变成下一个E-节点，由于此节点不能到达任何新的节点，所以此节点即被删除，节点（3，1）成为新的E-节点，将队列清空。节点（3，1）展开，（3，2）被加入队列中，而（3，1）被删除。（3，2）变为新的E-节点，展开此节点后，到达节点（3，3），即迷宫的出口。

使用FIFO搜索，总能找出从迷宫入口到出口的最短路径。需要注意的是：利用回溯法找到的路径却不一定是最短路径。有趣的是，程序6-11已经给出了利用FIFO分枝定界搜索从迷宫的（1，1）位置到（n，n）位置的最短路径的代码。

例17-2 [0/1背包问题] 下面比较分别利用FIFO分枝定界和最大收益分枝定界方法来解决如下背包问题：n=3，w=[20,15,15]，p=[40,25,25]，c=30。FIFO分枝定界利用一个队列来记录活节点，节点将按照FIFO顺序从队列中取出；而最大收益分枝定界使用一个最大堆，其中的 E-节点按照每个活节点收益值的降序，或是按照活节点任意子树的叶节点所能获得的收益估计值的降序从队列中取出。本例所使用的背包问题与例16.2相同，并且有相同的解空间树。

使用FIFO分枝定界法搜索，初始时以根节点A作为E-节点，此时活节点队列为空。当节点A展开时，生成了节点B和C，由于这两个节点都是可行的，因此都被加入活节点队列中，节点A被删除。下一个E-节点是B，展开它并产生了节点D和E，D是不可行的，被删除，而E被加入队列中。下一步节点C成为E-节点，它展开后生成节点F和G，两者都是可行节点，加入队列中。下一个E-节点E生成节点J和K，J不可行而被删除，K是一个可行的叶节点，并产生一个到目前为止可行的解，它的收益值为40。

下一个E-节点是F，它产生两个孩子L、M，L代表一个可行的解且其收益值为50，M代表另一个收益值为15的可行解。G是最后一个E-节点，它的孩子N和O都是可行的。由于活节点队列变为空，因此搜索过程终止，最佳解的收益值为50。

可以看到，工作在解空间树上的FIFO分枝定界方法非常象从根节点出发的宽度优先搜索。它们的主要区别是在FIFO分枝定界中不可行的节点不会被搜索。

最大收益分枝定界算法以解空间树中的节点A作为初始节点。展开初始节点得到节点B和C，两者都是可行的并被插入堆中，节点B获得的收益值是40（设x_1=1），而节点C得到的收益值为0。A被删除，B成为下一个E-节点，因为它的收益值比C的大。当展开B时得到了节点D和E，D是不可行的而被删除，E加入堆中。由于E具有收益值40，而C为0，因为E成为下一个E-节点。展开E时生成节点J和K，J不可行而被删除，K是一个可行的解，因此K为作为目前能找到的最优解而记录下来，然后K被删除。由于只剩下一个活节点C在堆中，因此C作为E-节点被展开，生成F、G两个节点插入堆中。F的收益值为25，因此成为下一个E-节点，展开后得到节点L和M，但L、M都被删除，因为它们是叶节点，同时L所对应的解被作为当前最优解记录下来。最终，G成为E-节点，生成的节点为N和O，两者都是叶节点而被删除，两者所对应的解都不比当前的最优解更好，因此最优解保持不变。此时堆变为空，没有下一个 E-节点产生，搜索过程终止。终止于J的搜索即为最优解。

犹如在回溯方法中一样，可利用一个定界函数来加速最优解的搜索过程。定界函数为最大收益设置了一个上限，通过展开一个特殊的节点可能获得这个最大收益。如果一个节点的定界

函数值不大于目前最优解的收益值，则此节点会被删除而不作展开，更进一步，在最大收益分枝定界方法中，可以使节点按照它们收益的定界函数值的非升序从堆中取出，而不是按照节点的实际收益值来取出。这种策略从可能到达一个好的叶节点的活节点出发，而不是从目前具有较大收益值的节点出发。

例17-3 [旅行商问题] 对于图16-4的四城市旅行商问题，其对应的解空间为图16-5所示的排列树。FIFO分枝定界使用节点B作为初始的E-节点，活节点队列初始为空。当B展开时，生成节点C、D和E。由于从顶点1到顶点2，3，4都有边相连，所以C、D、E三个节点都是可行的并加入队列中。当前的E-节点B被删除，新的E-节点是队列中的第一个节点，即节点C。因为在图16-4中存在从顶点2到顶点3和4的边，因此展开C，生成节点F和G，两者都被加入队列。下一步，D成为E-节点，接着又是E，到目前为止活节点队列中包含节点F到K。

下一个E-节点是F，展开它得到了叶节点L。至此找到了一个旅行路径，它的开销是59。展开下一个E-节点G，得到叶节点M，它对应于一个开销为66的旅行路径。接着H成为E-节点，从而找到叶节点N，对应开销为25的旅行路径。下一个E-节点是I，它对应的部分旅行1-3-4的开销已经为26，超过了目前最优的旅行路径，因此，I不会被展开。最后，节点J，K成为E-节点并被展开。经过这些展开过程，队列变为空，算法结束。找到的最优方案是节点N所对应的旅行路径。

如果不使用FIFO方法，还可以使用最小耗费方法来搜索解空间树，即用一个最小堆来存储活节点。这种方法同样从节点B开始搜索，并使用一个空的活节点列表。当节点B展开时，生成节点C、D和E并将它们加入最小堆中。在最小堆的节点中，E具有最小耗费（因为1-4的局部旅行的耗费是4），因此成为E-节点。展开E生成节点J和K并将它们加入最小堆，这两个节点的耗费分别为14和24。此时，在所有最小堆的节点中，D具有最小耗费，因而成为E-节点，并生成节点H和I。至此，最小堆中包含节点C、H、I、J和K，H具有最小耗费，因此H成为下一个E-节点。展开节点E，得到一个完整的旅行路径1-3-2-4-1，它的开销是25。节点J是下一个E-节点，展开它得到节点P，它对应于一个耗费为25的旅行路径。节点K和I是下两个E-节点。由于I的开销超过了当前最优的旅行路径，因此搜索结束，而剩下的所有活节点都不能使我们找到更优的解。

对于例17-2的背包问题，可以使用一个定界函数来减少生成和展开的节点数量。这种函数将确定旅行的最小耗费的下限，这个下限可通过展开某个特定的节点而得到。如果一个节点的定界函数值不能比当前的最优旅行更小，则它将被删除而不被展开。另外，对于最小耗费分枝定界，节点按照它在最小堆中的非降序取出。

在以上几个例子中，可以利用定界函数来降低所产生的树型解空间的节点数目。当设计定界函数时，必须记住主要目的是利用最少的时间，在内存允许的范围内去解决问题。而通过产生具有最少节点的树来解决问题并不是根本的目标。因此，我们需要的是一个能够有效地减少计算时间并因此而使产生的节点数目也减少的定界函数。

回溯法比分枝定界在占用内存方面具有优势。回溯法占用的内存是O（解空间的最大路径长度），而分枝定界所占用的内存为O（解空间大小）。对于一个子集空间，回溯法需要$\Theta(n)$的内存空间，而分枝定界则需要$O(2^n)$的空间。对于排列空间，回溯需要$\Theta(n)$的内存空间，分枝定界需要$O(n!)$的空间。虽然最大收益（或最小耗费）分枝定界在直觉上要好于回溯法，并且在许多情况下可能会比回溯法检查更少的节点，但在实际应用中，它可能会在回溯法超出允许的时间限制之前就超出了内存的限制。

练习

1. 假定在一个LIFO分枝定界搜索中，活节点列表的行为与堆栈相同，请使用这种方法来解决例17-2的背包问题。LIFO分枝定界与回溯有何区别？

2. 对于如下0/1背包问题：n=4, p=[4,3,2,1], w=[1,2,3,4], c=6。

1) 画出有四个对象的背包问题的解空间树。

2) 像例17-2那样，描述用FIFO分枝定界法解决上述问题的过程。

3) 使用程序16-6的Bound函数来计算子树上任一叶节点可能获得的最大收益值，并根据每一步所能得到的最优解对应的定界函数值来判断是否将节点加入活节点列表中。解空间中哪些节点是使用以上机制的FIFO分枝定界方法产生的？

4) 像例17-2那样，描述用最大收益分枝定界法解决上述问题的过程。

5) 在最大收益分枝定界中，若使用3）中的定界函数，将产生解空间树中的哪些节点？

17.2 应用

17.2.1 货箱装船

1. FIFO分枝定界

16.2.1节的货箱装船问题主要是寻找第一条船的最大装载方案。这个问题是一个子集选择问题，它的解空间被组织成一个子集树。对程序16-1进行改造，即得到程序17-1中的FIFO分枝定界代码。程序17-1只是寻找最大装载的重量。

程序17-1 货箱装船问题的FIFO分枝定界算法

```
template<class T>
void AddLiveNode(LinkedQueue<T> &Q, T wt,
                 T& bestw, int i, int n)
{// 如果不是叶节点，则将节点权值wt加入队列Q
  if (i == n) {// 叶子
    if (wt > bestw) bestw = wt;}
  else Q.Add(wt); // 不是叶子
}

template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载值
 // 使用FIFO分枝定界算法
  // 为层次1 初始化
  LinkedQueue<T> Q;  // 活节点队列
  Q.Add(-1);         //标记本层的尾部
  int i = 1;         // E-节点的层
  T Ew = 0,          // E-节点的权值
    bestw = 0;       // 目前的最优值

  // 搜索子集空间树
  while (true) {
```

```
        // 检查E-节点的左孩子
        if (Ew + w[i] <= c) // x[i] = 1
          AddLiveNode(Q, Ew + w[i], bestw, i, n);

        // 右孩子总是可行的
        AddLiveNode(Q, Ew, bestw, i, n); // x[i] = 0

        Q.Delete(Ew);    // 取下一个E-节点
        if (Ew == -1) {  // 到达层的尾部
          if (Q.IsEmpty()) return bestw;
          Q.Add(-1);    //添加尾部标记
          Q.Delete(Ew); // 取下一个E-节点
          i++;}         // Ew的层
        }
    }
```

其中函数MaxLoading 在解空间树中进行分枝定界搜索。链表队列 Q用于保存活节点，其中记录着各活节点对应的权值。队列还记录了权值 -1，以标识每一层的活节点的结尾。函数AddLiveNode 用于增加节点（即把节点对应的权值加入活节点队列），该函数首先检验i（当前E-节点在解空间树中的层）是否等于 n，如果相等，则已到达了叶节点。叶节点不被加入队列中，因为它们不被展开。搜索中所到达的每个叶节点都对应着一个可行的解，而每个解都会与目前的最优解来比较，以确定最优解。如果 $i<n$，则节点i 就会被加入队列中。

MaxLoading函数首先初始化i=1（因为当前E-节点是根节点），bestw=0（目前最优解的对应值），此时，活节点队列为空。下一步， A-1被加入队列以说明正处在第一层的末尾。当前E-节点对应的权值为E_w。在while 循环中，首先检查节点的左孩子是否可行。如果可行，则调用AddLiveNode，然后将右孩子加入队列（此节点必定是可行的），注意到AddliveNode可能会失败，因为可能没有足够的内存来给队列增加节点。 AddLiveNode并没有去捕获Q.Add中的NoMem异常，这项工作留给用户完成。

如果E-节点的两个孩子都已经被生成，则删除该 E-节点。从队列中取出下一个 E-节点，此时队列必不为空，因为队列中至少含有本层末尾的标识-1。如果到达了某一层的结尾，则从下一层寻找活节点，当且仅当队列不为空时这些节点存在。当下一层存在活节点时，向队列中加入下一层的结尾标志并开始处理下一层的活节点。

MaxLoading函数的时间和空间复杂性都是$O(2^n)$。

2. 改进

我们可以尝试使用程序16-2的优化方法改进上述问题的求解过程。在程序 16-2中，只有当右孩子对应的重量加上剩余货箱的重量超出 bestw时，才选择右孩子。而在程序 17-1中，在i变为n之前，bestw的值一直保持不变，因此在i等于n之前对右孩子的测试总能成功，因为bestw=0且r>0。当i等于n时，不会再有节点加入队列中，因此这时对右孩子的测试不再有效。

如想要使右孩子的测试仍然有效，应当提早改变 bestw的值。我们知道，最优装载的重量是子集树中可行节点的重量的最大值。由于仅在向左子树移动时这些重量才会增大，因此可以在每次进行这种移动时改变 bestw的值。根据以上思想，我们设计了程序 17-2。当活节点加入队列时，wt不会超过bestw，故bestw不用更新。因此用一条直接插入MaxLoading的简单语句取代了函数AddLiveNode。

程序17-2 对程序17-1改进之后

```
template<class T>
T MaxLoading(T w[], T c, int n)
{// 返回最优装载值
 // 使用FIFO分枝定界算法
  // 为层1初始化
  LinkedQueue<T> Q;        // 活节点队列
  Q.Add(-1);               //标记本层的尾部
  int i = 1;               // E-节点的层
  T Ew = 0,                // E-节点的重量
    bestw = 0;             // 目前的最优值
    r = 0;                 // E-节点中余下的重量
  for (int j = 2; j <= n; j++)
    r += w[i];

  // 搜索子集空间树
  while (true) {
    // 检查E-节点的左孩子
    T wt = Ew + w[i];  // 左孩子的权值
    if (wt <= c) {     // 可行的左孩子
      if (wt > bestw) bestw = wt;
      // 若不是叶子，则添加到队列中
      if (i < n) Q.Add(wt);}

    // 检查右孩子
    if (Ew + r > bestw && i < n)
       Q.Add(Ew);   // 可以有一个更好的叶子

    Q.Delete(Ew);     // 取下一个 E-节点
    if (Ew == -1) {   // 到达层的尾部
      if (Q.IsEmpty()) return bestw;
      Q.Add(-1);     //添加尾部标记
      Q.Delete(Ew);  // 取下一个 E-节点
      i++;           // E-节点的层
       r -= w[i];}    // E-节点中余下的重量
    }
}
```

3. 寻找最优子集

为了找到最优子集，需要记录从每个活节点到达根的路径，因此在找到最优装载所对应的叶节点之后，就可以利用所记录的路径返回到根节点来设置 x的值。活节点队列中元素的类型是QNode(见程序17-3)。这里，当且仅当节点是它的父节点的左孩子时， LChild为true。

程序17-3 类QNode

```
template<class T>
class  QNode {
  private:
```

```
    QNode *parent; // 父节点指针
    bool LChild;   // 当且仅当是父节点的左孩子时，取值为true
    T weight;      // 由到达本节点的路径所定义的部分解的值
};
```

程序17-4是新的分枝定界方法的代码。为了避免使用大量的参数来调用 AddLiveNode，可以把该函数定义为一个内部函数。使用内部函数会使空间需求稍有增加。此外，还可以把AddLiveNode和MaxLoading定义成类成员函数，这样，它们就可以共享诸如Q,i,n,bestw,E,bestE和bestw 等类成员。

程序17-4并未删除类型为QNode的节点。为了删除这些节点，可以保存由 AddLiveNode创建的所有节点的指针，以便在程序结束时删除这些节点。

程序17-4 计算最优子集的分枝定界算法

```
template<class T>
void AddLiveNode(LinkedQueue<QNode<T>*> &Q, T wt, int i, int n, T bestw, QNode<T> *E,
    QNode<T> *&bestE, int bestx[], bool ch)
{// 如果不是叶节点，则向队列Q中添加一个i 层、重量为 wt的活节点
 // 新节点是E的一个孩子。当且仅当新节点是左孩子时， ch为true。
 // 若是叶子，则ch取值为bestx[n]
  if (i == n) {// 叶子
    if (wt == bestw) {
      // 目前的最优解
      bestE = E;
      bestx[n] = ch;}
    return;}

  // 不是叶子，添加到队列中
  QNode<T> *b;
  b = new QNode<T>;
  b->weight = wt;
  b->parent = E;
  b->LChild = ch;
  Q.Add(b);
}

template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最优装载值，并在bestx中返回最优装载
// 使用FIFO分枝定界算法
  // 初始化层1
  LinkedQueue<QNode<T>*> Q;     // 活节点队列
  Q.Add(0);                     // 0 代表本层的尾部
  int i = 1;                    // E - 节点的层
  T Ew = 0,                     // E - 节点的重量
    bestw = 0;                  // 迄今得到的最优值
    r = 0;                      // E - 节点中余下的重量
  for (int j = 2; j <= n; j++)
```

```
    r += w[i];
  QNode<T> *E = 0,                    // 当前的 E-节点
        *bestE;                       // 目前最优的 E-节点

  // 搜索子集空间树
  while (true) {
    // 检查E-节点的左孩子
    T wt = Ew + w[i];
    if (wt <= c) {// 可行的左孩子
      if (wt > bestw) bestw = wt;
      AddLiveNode(Q, wt, i, n, bestw, E, bestE, bestx, true);}

    // 检查右孩子
    if (Ew + r > bestw) AddLiveNode(Q, Ew, i, n, bestw, E, bestE, bestx, false);

    Q.Delete(E);                      // 下一个E-节点
    if (!E) {                         // 层的尾部
      if (Q.IsEmpty()) break;
      Q.Add(0);                       // 层尾指针
      Q.Delete(E);                    // 下一个E-节点
      i++;                            // E-节点的层次
      r -= w[i];}                     // E-节点中余下的重量

    Ew = E->weight;                   // 新的E-节点的重量
    }

  // 沿着从bestE到根的路径构造x[ ]， x[n]由 AddLiveNode来设置
  for (j = n - 1; j > 0; j--) {
    bestx[j] = bestE->LChild;  // 从bool转换为int
    bestE = bestE->parent;
    }

  return bestw;
}
```

4. 最大收益分枝定界

在对子集树进行最大收益分枝定界搜索时，活节点列表是一个最大优先级队列，其中每个活节点x都有一个相应的重量上限（最大收益）。这个重量上限是节点x 相应的重量加上剩余货箱的总重量，所有的活节点按其重量上限的递减顺序变为 E-节点。需要注意的是，如果节点 x 的重量上限是x.uweight，则在子树中不可能存在重量超过 x.uweight 的节点。另外，当叶节点对应的重量等于它的重量上限时，可以得出结论：在最大收益分枝定界算法中，当某个叶节点成为E-节点并且其他任何活节点都不会帮助我们找到具有更大重量的叶节点时，最优装载的搜索终止。

上述策略可以用两种方法来实现。在第一种方法中，最大优先级队列中的活节点都是互相独立的，因此每个活节点内部必须记录从子集树的根到此节点的路径。一旦找到了最优装载所对应的叶节点，就利用这些路径信息来计算 x 值。在第二种方法中，除了把节点加入最大优先

队列之外，节点还必须放在另一个独立的树结构中，这个树结构用来表示所生成的子集树的一部分。当找到最大装载之后，就可以沿着路径从叶节点一步一步返回到根，从而计算出 x 值。本书使用第二种方法，第一种方法的实现留作习题。

最大优先队列可用HeapNode 类型的最大堆来表示（见程序17-5）。uweight 是活节点的重量上限，level 是活节点所在子集树的层，ptr 是指向活节点在子集树中位置的指针。子集树中节点的类型是bbnode（见程序17-5）。节点按uweight值从最大堆中取出。

程序17-5 bbnode 和HeapNode 类

```
class bbnode {
  private:
    bbnode *parent; // 父节点指针
    bool LChild;  // 当且仅当是父节点的左孩子时，取值为true
};

template<class T>
class HeapNode {
  public:
    operator T () const {return uweight;}
  private:
    bbnode *ptr;  // 活节点指针
    T uweight;   // 活节点的重量上限
    int level;   // 活节点所在层
};
```

程序17-6中的函数AddLiveNode用于把bbnode类型的活节点加到子树中，并把HeapNode类型的活节点插入最大堆。AddLiveNode必须被定义为bbnode和HeapNode的友元。

程序17-6

```
template<class T>
void AddLiveNode(MaxHeap<HeapNode<T> > &H, bbnode *E, T wt, bool ch, int lev)
{// 向最大堆H中增添一个层为lev上限重量为wt的活节点
 // 新节点是 E的一个孩子
 // 当且仅当新节点是左孩子ch 为true
  bbnode *b = new bbnode;
  b->parent = E;
  b->LChild = ch;
  HeapNode<T> N;
  N.uweight = wt;
  N.level = lev;
  N.ptr = b;
  H.Insert(N);
}

template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{// 返回最优装载值，最优装载方案保存于bestx
```

```
// 使用最大收益分枝定界算法
  // 定义一个最多有1000个活节点的最大堆
  MaxHeap<HeapNode<T> > H(1000);

  // 第一剩余重量的数组
  // r[j] 为 w[j+1:n]的重量之和
  T *r = new T [n+1];
  r[n] = 0;
  for (int j = n-1; j > 0; j--)
    r[j] = r[j+1] + w[j+1];

  // 初始化层1
  int i = 1;              // E-节点的层
  bbnode *E = 0;          // 当前E-节点
  T Ew = 0;               // E-节点的重量

  // 搜索子集空间树
  while (i != n+1) {// 不在叶子上
    // 检查E-节点的孩子
    if (Ew + w[i] <= c) {// 可行的左孩子
      AddLiveNode(H, E, Ew+w[i]+r[i], true, i+1);}
    // 右孩子
    AddLiveNode(H, E, Ew+r[i], false, i+1);

    // 取下一个E-节点
    HeapNode<T> N;
    H.DeleteMax(N); // 不能为空
    i = N.level;
    E = N.ptr;
    Ew = N.uweight - r[i-1];
    }

  // 沿着从E-节点E到根的路径构造bestx[]
  for (int j = n; j > 0; j--) {
    bestx[j] = E->LChild;  // 从bool转换为int
    E = E->parent;
    }

  return Ew;
}
```

函数MaxLoading（见程序17-6）首先定义了一个容量为1000的最大堆，因此，可以用它来解决优先队列中活节点数在任何时候都不超过1000的装箱问题。对于更大型的问题，需要一个容量更大的最大堆。接着，函数MaxLoading初始化剩余重量数组r。第i+1层的节点（即x[1:i]的值都已确定）对应的剩余容器总重量可以用如下公式求出：$r[i]=\sum_{j=i+1}^{n} w[j]$。变量E指向子集树中的当前E-节点，Ew是该节点对应的重量，i是它所在的层。初始时，根节点是E-节点，因此取i=1, Ew=0。由于没有明确地存储根节点，因此E的初始值取为0。

while循环用于产生当前E-节点的左、右孩子。如果左孩子是可行的（即它的重量没有超

出容量），则将它加入到子集树中并作为一个第i+1层节点加入最大堆中。一个可行的节点的右孩子也被认为是可行的，它总被加入子树及最大堆中。在完成添加操作后，接着从最大堆中取出下一个E-节点。如果没有下一个E-节点，则不存在可行的解。如果下一个E-节点是叶节点（即是一个层为n+1的节点），则它代表着一个最优的装载，可以沿着从叶到根的路径来确定装载方案。

5. 说明

1) 使用最大堆来表示活节点的最大优先队列时，需要预测这个队列的最大长度（程序17-6中是1000）。为了避免这种预测，可以使用一个基于指针的最大优先队列来取代基于数组的队列，这种表示方法见9.4节的左高树。

2) bestw表示当前所有可行节点的重量的最大值，而优先队列中可能有许多其uweight不超过bestw的活节点，因此这些节点不可能帮助我们找到最优的叶节点，这些节点浪费了珍贵的队列空间，并且它们的插入/删除动作也浪费了时间，所以可以将这些节点删除。有一种策略可以减少这种浪费，即在插入某个节点之前检查是否有uweight<bestw。然而，由于bestw在算法执行过程中是不断增大的，所以目前插入的节点在以后并不能保证uweight<bestw。另一种更好的方法是在每次bestw增大时，删除队列中所有uweight<besew的节点。这种策略要求删除具有最小uweight的节点。因此，队列必须支持如下的操作：插入、删除最大节点、删除最小节点。这种优先队列也被称作双端优先队列（double-ended priority queue）。这种队列的数据结构描述见第9章的参考文献。

17.2.2 0/1背包问题

0/1背包问题的最大收益分枝定界算法可以由程序16-6发展而来。可以使用程序16-6的Bound函数来计算活节点N的收益上限up,使得以N为根的子树中的任一节点的收益值都不可能超过uprofit。活节点的最大堆使用uprofit作为关键值域，最大堆的每个入口都以HeapNode作为其类型，HeapNode有如下私有成员：uprofit, profit, weight，level，ptr，其中level和ptr的定义与装箱问题（见程序17-5）中的含义相同。对任一节点N，N.profit是N的收益值，N uprofit是它的收益上限，N. weight是它对应的重量。bbnode类型如程序17-5中的定义，各节点按其uprofit值从最大堆中取出。

程序17-7使用了类Knap，它类似于回溯法中的类Knap（见程序16-5）。两个Knap版本中数据成员之间的区别见程序17-7：1) bestp 不再是一个成员；2) bestx 是一个指向int 的新成员。新增成员的作用是：当且仅当物品j 包含在最优解中时，bestx[j]=1。函数AddLiveNode用于将新的bbnode类型的活节点插入子集树中，同时将HeapNode类型的活节点插入到最大堆中。这个函数与装箱问题（见程序17-6）中的对应函数非常类似，因此相应的代码被省略。

程序17-7 0/1背包问题的最大收益分枝定界算法

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>::MaxProfitKnapsack()
{// 返回背包最优装载的收益
 // bestx[i] = 1 当且仅当物品 i 属于最优装载
 // 使用最大收益分枝定界算法
   // 定义一个最多可容纳1000个活节点的最大堆
   H = new MaxHeap<HeapNode<Tp, Tw> > (1000);

   // 为bestx分配空间
```

```
    bestx = new int [n+1];

    // 初始化层1
    int i = 1;
    E = 0;
    cw = cp = 0;
    Tp bestp = 0;      // 目前的最优收益
    Tp up = Bound(1);  // 在根为E的子树中最大可能的收益

    // 搜索子集空间树
    while (i != n+1) { // 不是叶子
      // 检查左孩子
      Tw wt = cw + w[i];
      if (wt <= c) {// 可行的左孩子
        if (cp+p[i] > bestp) bestp = cp+p[i];
        AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);}
        up = Bound(i+1);

      // 检查右孩子
      if (up >= bestp) // 右孩子有希望
          AddLiveNode(up, cp, cw, false, i+1);

      // 取下一个E-节点
      HeapNode<Tp, Tw> N;
      H->DeleteMax(N); // 不能为空
      E = N.ptr;
      cw = N.weight;
      cp = N.profit;
      up = N.uprofit;
      i = N.level;
      }

    // 沿着从E-节点E到根的路径构造bestx[]
    for (int j = n; j > 0; j--) {
      bestx[j] = E->LChild;
      E = E->parent;
      }

    return cp;
  }
```

函数MaxProfitKnapsack在子集树中执行最大收益分枝定界搜索。函数假定所有的物品都是按收益密度值的顺序排列，可以使用类似于程序 16-9中回溯算法所使用的预处理代码来完成这种排序。函数MaxProfitKnapsack首先初始化活节点的最大堆，并使用一个数组 bestx来记录最优解。由于需要不断地利用收益密度来排序，物品的索引值会随之变化，因此必须将MaxProfitKnapsack所生成的结果映射回初始时的物品索引。可以用 Q的ID域来实现上述映射（见程序16-9）。

在函数MaxProfitKnapSack中，E是当前E-节点，cw是节点对应的重量，cp是收益值，up是以E为根的子树中任一节点的收益值上限。while循环一直执行到一个叶节点成为E-节点为止。

由于最大堆中的任何剩余节点都不可能具有超过当前叶节点的收益值，因此当前叶即对应了一个最优解。可以从叶返回到根来确定这个最优解。

MaxProfitKnapsack中while循环的结构很类似于程序17-6的while循环。首先，检验E-节点左孩子的可行性，如它是可行的，则将它加入子集树及活节点队列（即最大堆）；仅当节点右孩子的Bound值指明有可能找到一个最优解时才将右孩子加入子集树和队列中。

17.2.3 最大完备子图

16.2.3节完备子图问题的解空间树也是一个子集树，故可以使用与装箱问题、背包问题相同的最大收益分枝定界方法来求解这种问题。解空间树中的节点类型为 bbnode，而最大优先队列中元素的类型则是CliqueNode。CliqueNode有如下域：cn（该节点对应的完备子图中的顶点数目），un（该节点的子树中任意叶节点所对应的完备子图的最大尺寸），level（节点在解空间树中的层），cn（当且仅当该节点是其父节点的左孩子时， cn为1），ptr（指向节点在解空间树中的位置）。un的值等于cn+n-leve+1。因为根据un 和cn（或level）可以求出level（或cn），所以可以去掉cn或level域。当从最大优先队列中选取元素时，选取的是具有最大 un值的元素。在程序17-8中，CliqueNode包含了所有的三个域：cn，un 和level，这样便于尝试为un赋予不同的含义。函数AddCliqueNode用于向生成的子树和最大堆中加入节点，由于其代码非常类似于装箱和背包问题中的对应函数，故将它略去。

函数BBMaxClique在解空间树中执行最大收益分枝定界搜索，树的根作为初始的 E-节点，该节点并没有在所构造的树中明确存储。对于这个节点来说，其 cn 值（E-节点对应的完备子图的大小）为0，因为还没有任何顶点被加入完备子图中。 E-节点的层由变量i 指示，它的初值为1，对应于树的根节点。当前所找到的最大完备子图的大小保存在 bestn中。

在while 循环中，不断展开E-节点直到一个叶节点变成E-节点。对于叶节点，un＝cn。由于所有其他节点的un 值都小于等于当前叶节点对应的un 值，所以它们不可能产生更大的完备子图，因此最大完备子图已经找到。沿着生成的树中从叶节点到根的路径，即可构造出这个最大完备子图。

为了展开一个非叶E-节点，应首先检查它的左孩子，如果左孩子对应的顶点 i与当前E-节点所包含的所有顶点之间都有一条边，则 i 被加入当前的完备子图之中。为了检查左孩子的可行性，可以沿着从E-节点到根的路径，判断哪些顶点包含在 E-节点之中，同时检查这些顶点中每个顶点是否都存在一条到i 的边。如果左孩子是可行的，则把它加入到最大优先队列和正在构造的树中。下一步，如果右孩子的子树中包含最大完备子图对应的叶节点，则把右孩子也加入。

由于每个图都有一个最大完备子图，因此从堆中删除节点时，不需要检验堆是否为空。仅当到达一个可行的叶节点时， while循环终止。

程序17-8 最大完备子图问题的分枝定界算法

```
int AdjacencyGraph::BBMaxClique(int bestx[])
{// 寻找一个最大完备子图的最大收益分枝定界程序
  // 定义一个最多可容纳1000个活节点的最大堆
  MaxHeap<CliqueNode> H(1000);

  // 初始化层1
  bbnode *E = 0;  // 当前的E-节点为根
```

```
    int i = 1,      // E-节点的层
        cn = 0,     // 完备子图的大小
        bestn = 0;  // 目前最大完备子图的大小

    // 搜索子集空间树
    while (i != n+1) {// 不是叶子
      // 在当前完备子图中检查顶点 i 是否与其它顶点相连
      bool OK = true;
      bbnode *B = E;
      for (int j = i - 1; j > 0; B = B->parent, j--)
        if (B->LChild && a[i][j] == NoEdge) {
          OK = false;
          break;}

      if (OK) {// 左孩子可行
        if (cn + 1 > bestn) bestn = cn + 1;
        AddCliqueNode(H, cn+1, cn+n-i+1, i+1, E, true);}
      if (cn + n - i >= bestn)
        // 右孩子有希望
        AddCliqueNode(H, cn, cn+n-i, i+1, E, false);

      // 取下一个E-节点
      CliqueNode N;
      H.DeleteMax(N); // 不能为空
      E = N.ptr;
      cn = N.cn;
      i = N.level;
      }

    // 沿着从E到根的路径构造bestx[]
    for (int j = n; j > 0; j--) {
      bestx[j] = E->LChild;
      E = E->parent;
      }

    return bestn;
  }
```

17.2.4 旅行商问题

旅行商问题的介绍见16.2.4节，它的解空间是一个排列树。与在子集树中进行最大收益和最小耗费分枝定界搜索类似，该问题有两种实现的方法。第一种是只使用一个优先队列，队列中的每个元素中都包含到达根的路径。另一种是保留一个部分解空间树和一个优先队列，优先队列中的元素并不包含到达根的路径。本节只实现前一种方法。

由于我们要寻找的是最小耗费的旅行路径，因此可以使用最小耗费分枝定界法。在实现过程中，使用一个最小优先队列来记录活节点，队列中每个节点的类型为 MinHeapNode。每个节点包括如下区域：x（从1到n的整数排列，其中x[0]=1），s（一个整数，使得从排列树的根

节点到当前节点的路径定义了旅行路径的前缀 x[0:s]，而剩余待访问的节点是 x[s+1:n-1])，cc（旅行路径前缀，即解空间树中从根节点到当前节点的耗费），lcost（该节点子树中任意叶节点中的最小耗费），rcost（从顶点 x[s:n-1]出发的所有边的最小耗费之和）。当类型为MinHeapNode(T)的数据被转换成为类型T时，其结果即为lcost的值。分枝定界算法的代码见程序17-9。

程序17-9首先生成一个容量为1000的最小堆，用来表示活节点的最小优先队列。活节点按其lcost值从最小堆中取出。接下来，计算有向图中从每个顶点出发的边中耗费最小的边所具有的耗费MinOut。如果某些顶点没有出边，则有向图中没有旅行路径，搜索终止。如果所有的顶点都有出边，则可以启动最小耗费分枝定界搜索。根的孩子（图 16-5的节点B）作为第一个E-节点，在此节点上，所生成的旅行路径前缀只有一个顶点 1，因此s=0, x[0]=1, x[1:n-1]是剩余的顶点（即顶点 2，3，…，n）。旅行路径前缀 1的开销为 0，即cc=0，并且，rcost= $\sum_{i=1}^{n}$MinOut[i]。在程序中，bestc 给出了当前能找到的最少的耗费值。初始时，由于没有找到任何旅行路径，因此bestc的值被设为NoEdge。

程序17-9 旅行商问题的最小耗费分枝定界算法

```
template<class T>
T AdjacencyWDigraph<T>::BBTSP(int v[])
{// 旅行商问题的最小耗费分枝定界算法
   // 定义一个最多可容纳1000个活节点的最小堆
  MinHeap<MinHeapNode<T> > H(1000);

  T *MinOut = new T [n+1];
  // 计算MinOut[i] = 离开顶点i的最小耗费边的耗费
  T MinSum = 0;  // 离开顶点i的最小耗费边的数目
  for (int i = 1; i <= n; i++) {
    T Min = NoEdge;
    for (int j = 1; j <= n; j++)
      if (a[i][j] != NoEdge &&
            (a[i][j] < Min || Min == NoEdge))
        Min = a[i][j];
    if (Min == NoEdge) return NoEdge; // 此路不通
    MinOut[i] = Min;
    MinSum += Min;
    }

  // 把 E-节点初始化为树根
  MinHeapNode<T> E;
  E.x = new int [n];
  for (i = 0; i < n; i++)
    E.x[i] = i + 1;
  E.s = 0;          // 局部旅行路径为 x[1:0]
  E.cc = 0;         // 其耗费为0
  E.rcost = MinSum;
  T bestc = NoEdge;  // 目前没有找到旅行路径
```

```
// 搜索排列树
while (E.s < n - 1) {// 不是叶子
  if (E.s == n - 2) {// 叶子的父节点
    // 通过添加两条边来完成旅行
    // 检查新的旅行路径是不是更好
    if (a[E.x[n-2]][E.x[n-1]] != NoEdge && a[E.x[n-1]][1] != NoEdge && (E.cc +
       a[E.x[n-2]][E.x[n-1]] + a[E.x[n-1]][1] < bestc || bestc == NoEdge)) {
       // 找到更优的旅行路径
       bestc = E.cc + a[E.x[n-2]][E.x[n-1]] + a[E.x[n-1]][1];
       E.cc = bestc;
       E.lcost = bestc;
       E.s++;
       H.Insert(E);}
    else delete [] E.x;}
  else {// 产生孩子
    for (int i = E.s + 1; i < n; i++)
      if (a[E.x[E.s]][E.x[i]] != NoEdge) {
        // 可行的孩子, 限定了路径的耗费
        T cc = E.cc + a[E.x[E.s]][E.x[i]];
        T rcost = E.rcost - MinOut[E.x[E.s]];
        T b = cc + rcost;  //下限
        if (b < bestc || bestc == NoEdge) {
          // 子树可能有更好的叶子
          // 把根保存到最大堆中
          MinHeapNode<T> N;
          N.x = new int [n];
          for (int j = 0; j < n; j++)
            N.x[j] = E.x[j];
          N.x[E.s+1] = E.x[i];
          N.x[i] = E.x[E.s+1];
          N.cc = cc;
          N.s = E.s + 1;
          N.lcost = b;
          N.rcost = rcost;
          H.Insert(N);}
        } // 结束可行的孩子
    delete [] E.x;}  // 对本节点的处理结束

  try {H.DeleteMin(E);}       // 取下一个E-节点
  catch (OutOfBounds) {break;} // 没有未处理的节点
  }

if (bestc == NoEdge) return NoEdge; // 没有旅行路径
// 将最优路径复制到 v[1:n] 中
for (i = 0; i < n; i++)
  v[i+1] = E.x[i];

while (true) {// 释放最小堆中的所有节点
```

```
    delete [] E.x;
    try {H.DeleteMin(E);}
    catch (OutOfBounds) {break;}
    }

  return bestc;
}
```

while 循环不断地展开E-节点，直到找到一个叶节点。当s=n-1时即可说明找到了一个叶节点。旅行路径前缀是x[0:n-1]，这个前缀中包含了有向图中所有的n个顶点。因此s=n-1的活节点即为一个叶节点。由于算法本身的性质，在叶节点上lcost和cc恰好等于叶节点对应的旅行路径的耗费。由于所有剩余的活节点的lcost值都大于等于从最小堆中取出的第一个叶节点的lcost值，所以它们并不能帮助我们找到更好的叶节点，因此，当某个叶节点成为E-节点后，搜索过程即终止。

while循环体被分别按两种情况处理，一种是处理s=n-2的E-节点，这时，E-节点是某个单独叶节点的父节点。如果这个叶节点对应的是一个可行的旅行路径，并且此旅行路径的耗费小于当前所能找到的最小耗费，则此叶节点被插入最小堆中，否则叶节点被删除，并开始处理下一个E-节点。

其余的E-节点都放在while循环的第二种情况中处理。首先，为每个E-节点生成它的两个子节点，由于每个E-节点代表着一条可行的路径x[0:s]，因此当且仅当< x[s],x[i] >是有向图的边且x[i]是路径x[s+1:n-1]上的顶点时，它的子节点可行。对于每个可行的孩子节点，将边< x[s],x[i] >的耗费加上E.cc即可得到此孩子节点的路径前缀(x[0:s]，x[i])的耗费cc。由于每个包含此前缀的旅行路径都必须包含离开每个剩余顶点的出边，因此任何叶节点对应的耗费都不可能小于cc加上离开各剩余顶点的出边耗费的最小值之和，因而可以把这个下限值作为E-节点所生成孩子的lcost值。如果新生成孩子的lcost值小于目前找到的最优旅行路径的耗费bestc，则把新生成的孩子加入活节点队列（即最小堆）中。

如果有向图没有旅行路径，程序17-9返回NoEdge；否则，返回最优旅行路径的耗费，而最优旅行路径的顶点序列存储在数组v中。

17.2.5 电路板排列

电路板排列问题（16.2.5节）的解空间是一棵排列树，可以在此树中进行最小耗费分枝定界搜索来找到一个最小密度的电路板排列。我们使用一个最小优先队列，其中元素的类型为BoardNode，代表活节点。BoardNode类型的对象包含如下域：x（电路板的排列），s（电路板x[1:s]）依次放置在位置1到s上），cd（电路板排列x[1:s]的密度，其中包括了到达x[s]右边的连线），now（now[j]是排列x[1:s]中包含j的电路板的数目）。当一个BoardNode类型的对象转换为整型时，其结果即为对象的cd值。代码见程序17-10。

程序17-10 电路板排列问题的最小耗费分枝定界算法

```
int BBArrangeBoards(int **B, int n, int m, int* &bestx)
{// 最小耗费分枝定界算法, m个插槽, n块板
  MinHeap<BoardNode> H(1000);  // 容纳活节点
  // 初始化第一个E节点、 total和bestd
  BoardNode E;
```

```
E.x = new int [n+1];
E.s = 0;   // 局部排列为 E.x[1:s]
E.cd = 0;  // E.x[1:s]的密度
E.now = new int [m+1];
int *total = new int [m+1];
// now[i] = x[1:s]中含插槽i的板的数目
// total[i] = 含插槽i的板的总数目
for (int i = 1; i <= m; i++) {
  total[i] = 0;
  E.now[i] = 0;
  }
for (i = 1; i <= n; i++) {
  E.x[i] = i; // 排列为 12345...n
  for (int j = 1; j <= m; j++)
    total[j] += B[i][j]; // 含插槽 j的板
  }
int bestd = m + 1;      //目前的最优密度
bestx = 0;              // 空指针

do {// 扩展 E节点
  if (E.s == n - 1) {// 仅有一个孩子
    int ld = 0; // 最后一块板的局部密度
    for (int j = 1; j <= m; j++)
      ld += B[E.x[n]][j];
    if (ld < bestd) {// 更优的排列
      delete [] bestx;
      bestx = E.x;
      bestd = max(ld, E.cd);
      }
    else delete [] E.x;
    delete [] E.now;}

  else {// 生成 E-节点的孩子
  for (int i = E.s + 1; i <= n; i++) {
    BoardNode N;
    N.now = new int [m+1];
    for (int j = 1; j <= m; j++)
      // 在新板中对 插槽计数
      N.now[j] = E.now[j] + B[E.x[i]][j];
    int ld = 0; // 新板的局部密度
    for (j = 1; j <= m; j++)
      if (N.now[j] > 0 && total[j] != N.now[j]) ld++;
    N.cd = max(ld, E.cd);
    if (N.cd < bestd) {// 可能会引向更好的叶子
       N.x = new int [n+1];
       N.s = E.s + 1;
       for (int j = 1; j <= n; j++)
         N.x[j] = E.x[j];
       N.x[N.s] = E.x[i];
```

```
            N.x[i] = E.x[N.s];
            H.Insert(N);}
          else delete [] N.now;}

          delete [] E.x;} // 处理完当前E-节点

        try {H.DeleteMin(E);} // 下一个E-节点
        catch (OutOfBounds) {return bestd;}  //没有E-节点
        } while (E.cd < bestd);

      // 释放最小堆中的所有节点
      do {delete [] E.x;
         delete [] E.now;
         try {H.DeleteMin(E);}
         catch (...) {break;}
        } while (true);

      return bestd;
    }
```

程序17-10首先初始化E-节点为排列树的根，此节点中没有任何电路板，因此有 s=0, cd=0, now[i]=0（1≤i≤n），x是整数1到n 的任意排列。接着，程序生成一个整型数组 total，其中total[i] 的值为包含i 的电路板的数目。目前能找到的最优的电路板排列记录在数组bestx 中，对应的密度存储在bestd 中。程序中使用一个do-while 循环来检查每一个E-节点，在每次循环的尾部，将从最小堆中选出具有最小cd 值的节点作为下一个E-节点。如果某个E-节点的cd 值大于等于bestd，则任何剩余的活节点都不能使我们找到密度小于bestd的电路板排列，因此算法终止。

do-while循环分两种情况处理E-节点，第一种是处理s=n-1时的情况，此种情况下，有n-1个电路板被放置好， E-节点即解空间树中的某个叶节点的父节点。节点对应的密度会被计算出来，如果需要，bested 和bestx 将被更新。

在第二种情况中，E-节点有两个或更多的孩子。每当一个孩子节点N生成时，它对应的部分排列(x[1:s+1])的密度N.cd就会被计算出来，如果N.cd<bestd,则N被存放在最小优先队列中；如果N.cd≥bestd，则它的子树中的所有叶节点对应的密度都满足 density≥bestd，这就意味着不会有优于bestx的排列。

练习

3. 在程序17-4中增加代码，将指向由函数AddLiveNode生成的节点的指针存储在一个链表队列中。MaxLoading必须利用这些指针信息在程序终止之前删除所有生成的节点。

4. 本节所使用的AddLiveNode函数直到程序终止前才删除所生成的节点。实际上，没有活动孩子且不产生叶节点的那些节点都可以被立即删除。类似地，在第 n 层节点中，若节点没有重量为bestw的孩子，则可以立即删除该节点。讨论怎样尽快删除不需要的节点。描述实现这种方法时所涉及的时间/空间变化。你推荐使用上述方法吗？

5. 在程序17-6中，定义一个bestw来记录目前生成的可行节点所对应的重量的最大值。修改程序17-6，使得如果活节点的重量大于等于 bestw，则将它加入子集树及最大堆中。此外，还必须增加初始化和更新bestw的代码。

6. 只使用一个最大优先队列，来实现用最大收益分枝定界方法求解货箱装船问题，即不要使用程序17-6中所用到的部分解空间树，而在每个优先队列的节点中都加入通向根节点的路径信息。

7. 修改程序17-6，把删除bbnode类型和HeapNode类型节点的任务放在程序结尾处。

8. 只使用一个最大优先队列，利用最大收益分枝定界法求解 0/1背包问题，即不必保存一个部分解空间树，所有优先队列中的节点都记录着通往根节点的路径。

9. 修改程序17-7，使得删除bbnode和HeapNode类型的节点的任务放在程序的结尾处执行。

10. 1) 程序17-8中，若右孩子的un值大于等于bestn，则将它加入最大堆中，如果将条件设为un>bestn，程序能否正确执行呢？为什么？

2) 程序是否将un≥bestn的左孩子加入最大堆中？

3) 修改程序，使得只将un>bestn的节点加入到最大堆和生成的解空间树中。

11. 考察最大完备子图问题的解空间树。对于任意层（第 i层）的子树中的节点 x，令 MinDegree(x) 为x 所包含的顶点的度的最小值。

1) 证明任何以x为根的子树的叶节点都不可能表示一个尺寸超过 X.un＝min{ X.cn+n-i+1, MinDegree(X)+1 }的完备子图。

2) 使用以上X.un的定义重写BBMaxClique。

3) 比较两种BBMaxClique版本在运行时间及产生解空间树节点的数目上的不同。

12. 只使用最大优先队列，实现最大完备子图问题的最大收益分枝定界算法。即：不必保存一个部分解空间树，而在每一个最大优先队的节点内包含通向根的路径。

13. 修改程序17-8，使得删除bbnode和CliqueNode类型的节点的工作放在程序结尾处执行。

14. 修改程序17-9，使得s=n-2的节点不进入优先队列，并且，将当前最优排列放在数组bestp中。当下一个E-节点的lcostbestc 时，算法终止。

15. 使用指向父节点的指针来实现部分解空间树，并使用包含 lcost，cc，rcost和ptr（指向解空间树中对应节点的指针）域的优先队列来实现程序 17-9。

16. 写出用FIFO分枝定界方法求解电路板排列问题的代码。代码必须输出最优电路板排列的排列次序及对应的密度。使用合适的数据来测试代码的正确性。

17. 用FIFO分枝定界方法来搜索一种电路板的排列，使得最长的网组的长度最小（参见 16章练习17）。

18. 使用最小耗费分枝定界法来完成练习17。

19. 用最小耗费分枝定界算法求解16章练习18的顶点覆盖问题。

20. 用最大收益分枝定界算法求解16章练习19的简易最大切割问题。

21. 用最小耗费分枝定界算法求解16章练习20的机器设计问题。

22. 用最小耗费分枝定界算法求解16章练习21的网络设计问题。

23. 用FIFO分枝定界算法求解16章练习22的n-皇后放置问题。

* 24. 用FIFO分枝定界完成16章练习23。

* 25. 用FIFO分枝定界完成16章练习24。

* 26. 用FIFO分枝定界完成16章练习25。

* 27. 用最小耗费分枝定界完成16章练习23。

* 28. 用最小耗费分枝定界完成16章练习24。

* 29. 用最小耗费分枝定界完成16章练习25。

* 30. 用任意的分枝定界方法完成16章的练习25。在本练习中，必须把增加活节点的函数以及选择下一个E-节点的函数作为函数的参数。