第七章　分枝－限界法

§1　算法基本思想

分枝限界法同回溯法类似，它也是在解空间中搜索问题的可行解或最优解，但搜索的方式不同。回溯法采用深度优先的方式，朝纵深方向搜索，直至达到问题的一个可行解，或经判断沿此路径不会达到问题的可行解或最优解时，停止向前搜索，并沿原路返回到该路径上最后一个还可扩展的节点。然后，从该节点出发朝新的方向纵深搜索。分枝限界法则采用宽度优先的方式搜索解空间树，它将活节点存放在一个特殊的表中。其策略是：***在扩展节点处，首先生成其所有的儿子节点，将那些导致不可行解或导致非最优解的儿子节点舍弃，其余儿子节点加入活节点表中。然后，从活节点表中取出一个节点作为当前扩展节点，重复上述节点扩展过程。***

从活节点表中选择下一扩展节点的不同方式导致不同的分枝限界法。最常见的有以下两种方式：

1)．队列式

这种方式是将活节点表组织成一个队列或者一个栈，并按队列的先进先出（FIFO）或后入先出（LIFO）原则选取下一个节点作为当前扩展节点。

2)．优先队列式

这种方式是将活节点表组织成一个优先队列，并按优先队列给节点规定的优先级选取优先级最高的下一个节点作为当前扩展节点。

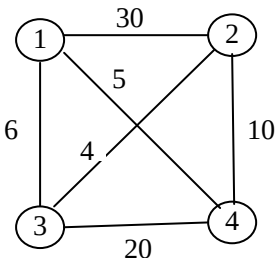
这两种方式各有优缺点，第一种方式组织活节点表简单省时，但是搜索解的路线比较单一，可能会绕道找到解；第二种方式在搜索方向上有所选择，可能会取近道找到解，但是组织节点表较复杂，有时会占用不少的时间。

队列式分枝限界法搜索解空间树的方式类似于解空间树的宽度优先搜索，不同的是队列式分枝限界法不搜索不可行节点（已经被判定不可能导致可行解或不可能导致最优解的节点）为根的子树。为达此目的，算法不把这样的节点列入活节点表。在选择当前扩展节点上是按照某种固定顺序，可称为静态选择法。

优先队列式分枝限界法的搜索方式是根据活节点表中节点的优先级确定下一个扩展节点。节点的优先级常用一个与该节点有关的数值 p 来表示。最大优先队列规定 p 值较大的节点的优先级较高。在算法实现时通常用一个最大堆来实现最大优先队列，用最大堆的 Deletemax 运算抽取堆的根节点作为当前扩展节点，体现最大效益优先的原则。类似地，最小优先队列规定 p 值较小的节点的优先级较高。在算法实现时，常用一个最小堆来实现，用最小堆的 Deletemin 运算抽取堆的根节点作为当前扩展节点，体现最小优先的原则。采用优先队列式分枝限界算法解决具体问题时，应根据问题的特点选用最大优先或最小优先队列，确定各

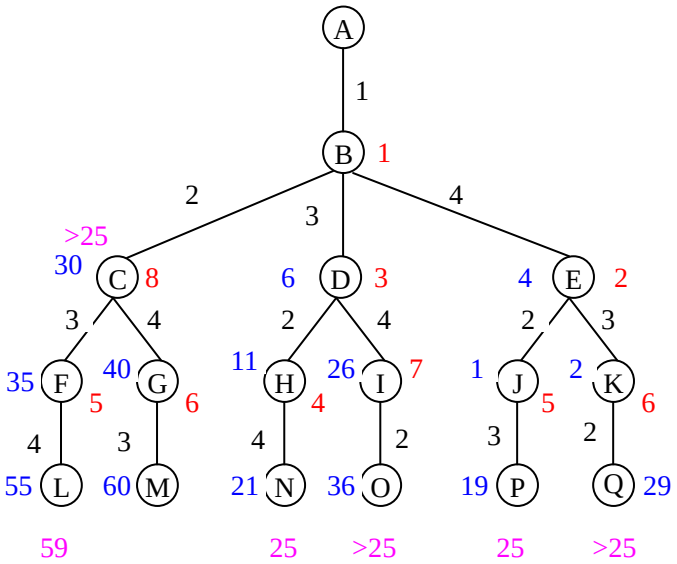
个节点的 p 值。

例7.1.1 旅行商问题，n=4，其解空间树是一棵排列树。对于如下的赋权图 G 表示的旅行商问题，我们分别说明队列式分支限界算法和优先队列分枝限界算法的执行过程。



(a) 图 G

采用优先队列式分枝限界算法搜索解空间树。蓝色表示该节点的费用；红色表示被取作扩展节点的序号；粉色相应环游的费用。



(b) 解空间树

图 7-1-1 表示旅行商问题的赋权图 G 及解空间树的搜索情况

采用**队列式分枝限界法**以排列树中的节点 B 作为初始扩展节点，此时，活节点队列为空。由于从图 G 的顶点 1 到顶点 2、3 和 4 均有边相连，B 的儿子 C、D 和 E 都是可行节点，它们被依次加入到活节点队列中。当前活节点队列中的队首节点 C 成为下一个扩展节点。由于图 G 的顶点 2 到顶点 3 和 4 有边相连，故节点 C 的二个儿子 F 和 G 均为可行节点，可以加入活节点队列。接下来，节点 D 和节点 E 相继成为扩展节点。此时活节点队列中的节点依次为 F、G、H、I、J、K。节点 F 成为下一个扩展节点，但其儿子 L 是解空间树的叶节点，我们找到了一条 Hamilton 圈（1，2，3，4，1），其费用为 59。此时记录这个目标函数值 f=59。下一个扩展节点 G 的儿子 M 也是叶节点，得到另一条 Hamilton 圈（1，2，4，3，1），其费用为 66。节点 H 成为当前扩展节点，其儿子 N 也是叶节点，得到第三条 Hamilton 圈，其费用为 25，因为它比记录中的目标函数值还小，所以修改目标函数值记录：f=25。下一个扩展节点是 I，由于从根节点到节点 I 的费用 26 已经超过目标函数的当前值，故没有必要扩展 I，以 I 为根的子树被剪掉。最后 J 和 K 被依次扩展，活节点队列成为空集，算法终止。算法搜索到的最优值是 25，相应的最优解是从根节点到节点 N 的路径

（1，3，2，4，1）

采用**优先队列式分枝限界法**，用一个最小堆存储活节点表，优先级函数值是

节点的当前费用。算法还是从排列树的节点 B 和空队列开始。节点 B 被扩展后，它的 3 个儿子 C、D 和 E 被依次插入最小堆中。此时，由于 E 是堆中具有最小当前费用（为 4）的节点，所以处于堆顶的位置，它自然成为下一个扩展节点。节点 E 被扩展后，其儿子 J 和 K 被插入当前堆中，它们的费用分别为 14 和 24。此时，堆顶元素是 D，它成为下一个扩展节点。它的 2 个儿子 H 和 I 被插入堆中。此时，堆中元素是节点 C、H、I、J、K。在这些节点中，H 具有最小费用，成为下一个扩展节点。扩展节点 H 后得到一条 Hamilton 圈（1，3，2，4，1），相应的费用为 25。接下来，节点 J 成为扩展节点，并由此得到一条 Hamilton 圈（1，4,2,3,1),费用仍为 25。此后的两个扩展节点是 K 和 I。由节点 K 得到的 Hamilton 圈的费用高于当前所知最小费用，节点 I 当前的费用已经高于当前所知最小费用，因而，它们都不能得到最优解。最后，优先队列为空，算法终止。

对于优化问题，要记录一个到目前已经取得的最优可行解及对应的目标函数值，这个记录要根据最优的原则更新。无论采用队列式还是优先队列式搜索，常常用目标函数的一个动态界（函数）来剪掉不必要搜索的分枝。

对于最大值优化问题，常引用一个可能获得的目标函数值的一个上界 CUB（经此节点可能达到的最大“效益值”)。如果当前扩展节点的儿子节点处的动态上界 CUB 小于目前所取得的目标函数值 prev，则该儿子节点不被放入活节点表。实际上相当于剪掉了解空间树中以该节点为根的子树。

对于最小值优化问题，常引用一个可能出现的目标函数值的一个动态下界 CLB（经此节点可能出现的最小“成本”），如果当前扩展节点的儿子节点处的动态下界 CLB 大于目前所取得的目标函数值 prev，则该节点不被放入活节点队列。

对于只是找可行解的问题，我们可以考虑如何降低搜索成本。常引用一个可能需要的成本的动态下界 CLB，如果当前扩展节点的儿子节点处的动态下界 CLB 大于目前所知道的到达最小成本答案节点所需要的成本 prev，则该节点不放入活节点表。

上述动态界称为剪枝函数，采用剪枝函数可以减少活节点数，降低搜索过程的复杂度。另一方面，对于采用优先队列式分支限界算法，选择当前扩展节点应该朝着效率高的方向努力。对于只是找可行解的问题，给每个进入活节点表的节点一个成本最大下界估值ĉ(X)，当前扩展节点就选活节点表中ĉ(X)最小的。对于最小化问题也同样处理，这里的成本可以采用目标值。对于最大化问题，给每个进入活节点表的节点一个“效益”最小上界估值ĉ(X)，当前扩展节点就选活节点表中ĉ(X)最大的。基于这种考虑，第四节将提出统一的 LC－分枝限界算法。

§2 0/1 背包问题的分枝－限界法

用优先队列式分枝限界法解决 0/1 背包问题(作为最大优化问题)，需要确定以下四个问题：

i. 解空间树中节点的结构；
ii. 如何生成一个给定节点的儿子节点；
iii. 如何组织活节点表；
iv. 如何识别答案节点。

采用完整的二叉树作为解空间树，放在活节点表中的每个节点具有 6 个信息段：

Parent、Level、Tag、CC、CV、CUB

其中 Parent 是节点 X 的父亲节点连接指针；Level 标出节点 X 在解空间树中的深度，通过置 $X_{Level(X)+1}=1$ 表示生成 X 的左儿子，置 $X_{Level(X)+1}=0$ 表示生成 X 的右儿子；信息段 Tag 用来输出最优解各个分量 x_i 的值；信息段 CC 记录背包在节点 X 处的可用空间(即剩余空间)，在确定 X 左儿子的可行性时用；CV 记录在节点 X 处背包中已装物品的价值(或效益值)，等于 $\sum_{1\le i\le Level(X)} p_i x_i$；信息段 CUB 用来存放节点 X 的 Pvu 值。这里，Pvu 表示在节点 X 所表示的状态下，可行解所能达到的可能价值的一个上界。也即是说，当 $x_1,\cdots,x_{l-1},x_l$ 的值确定后，可行解 $x_1,\cdots,x_l,x_{l+1},\cdots,x_n$ 所能达到的效益值的上界。类似地，当 $x_1,\cdots,x_{l-1},x_l$ 的值确定后，可行解 $x_1,\cdots,x_l,x_{l+1},\cdots,x_n$ 所能达到的最大效益值记做 Pvl。prev 是到目前为止能够探测到解的目标值的一个最大下界。如果节点 X 满足 Pvu<prev，则应该杀死节点 X（不放入节点表)。但是，当 Pvu = prev 时要谨慎处理。因为 prev 可能是某个答案节点的目标值，此时，由 Pvu=prev 可知沿此线路从节点 X 继续搜索下去不会得到更好的解，可以杀死节点 X。但是，prev 也可能只是一个纯粹的最大下界，此时 Pvu(X)=prev=Pvl(Y)，其中 Y 是已经放在节点表中的某个节点。因而，如果我们在一般节点 Z 处更新 prev 值时用 prev:=max(prev,Pvl(Z)) 这种做法，则 prev 的值可能是还没有产生具有这个目标值的解之前就知道了。此时如果在 Pvu(X)=prev 时贸然杀死节点 X 可能导致具有 prev 目标值的解夭折。为了区分上述两种情况，我们可以引进一个充分小的数 e，在一般节点 Z 处更新 prev 值时采取 prev:=max(prev,Pvl(Z)-e) 这样的做法，那么，当出现 Pvu=prev 时，就知道 prev 是某个答案节点的目标值，可以杀死该节点。但是，为了不影响活节点的优先级，e 必须满足 Pvl(Z)<Pvl(Y)⇒Pvl(Z)<Pvl(Y)-e。采用上述处理后，我们可以使用规则：

当 Pvu ≤ prev 时杀死节点 X

可以杀死更多不必要搜索的节点。Pvu(X)可以作为优先级函数，在活节点表中，选择 Pvu 值大的节点作为当前扩展节点。关于 Pvl(X)和 Pvu(X)的计算将由一个子程序给出。

作为求最大值优化问题处理的优先队列式分枝限界法解 0/1 背包问题的程序 LFKNAP 采用了六个子程序：LUBound、NewNode、Finish、Init、GetNode 和 Largest。子程序 LUBound 计算 Pvl 和 Pvu 之用；NewNode 生成一个具有六个信息段的节点，给各个信息段置入适当的值，并将此节点加入节点表；Finish 打印出最优解的值和此最优解中 $x_i=1$ 的物品；Init 对可用节点表和活节点表置初值；GetNode 在程序开始取一个可用节点；Largest 在活节点表中取一个具有最大 Pvu 值的节点作为当前扩展节点。

程序 7-2-1　0/1 背包问题的优先队列式分枝限界算法

```
    proc LFKNAP(P,W,M,N,e)//假定物品的排列顺序遵循
        //P[i]/W[i]≥P[i+1]/W[i+1];
        real P[1..N],W[1..N],M,CL,Pvl,Pvu,cap,cv,prev;
        integer ANS, X, N;
1.      Init; //初始化可用节点表及活节点表
2.      GetNode(E); //生成根节点
3.      Parent(E):=0; Level(E):=0; CC(E):=M; CV(E)=0;
4.      LUBound(P,W,M,0,N,1,Pvl,Pvu);
5.      prev:=Pvl-e; CLB(E):=Pvl; CUB(E):=Pvu; Tag(E):=0;
6.      Loop
7.        i:=Level(E)+1, cap:=CC(E), cv:=CV(E);
8.        case:
9.        i=N+1: //解节点
10.         if cv>prev then
11.           prev:=cv; ANS:=E;
12.         end{if}
13.       else: //E 是内部节点，有两个儿子
14.         if cap≥W[i] then //左儿子可行
15.           NewNode(E,i,1,cap-W[i],cv+P[i],CLB(E));
17.         end{if}
18.         LUBound(P,W,cap,cv,N,i+1,Pvl,Pvu);
19.         if Pvu>prev then //右儿子会活
```

```
20.            NewNode(E,i,0,cap,cv,Pvl);
21.            prev:=max(prev,Pvl-e);
22.         end{if}
23.       end{case}
24.       if 不再有活节点 then exit; end{if}
25.       Largest(E); //取一个活节点作为当前扩展节点
26.     until CUB(E)≤prev
27.     Finish(cv,ANS,N);
28.  end{LFKNAP}
```

算法中有两点值得注意：

1). 第 6～26 行的循环依次检查所生成的每个节点。此循环在以下两种情况下终止：或者活节点队列为空，或者为了扩展而选择的节点 E（扩展节点）满足 CUB(E)≤prev.在后一种情况下，由扩展节点的选法可知，对所有的扩展节点 X 均有 CUB(X)≤CUB(E)≤ prev，因而它们都不能导致其值比 prev 更大的解。

2). 在左儿子 X 可行的情况下，由 LUBound 算出它的上界、下界与 E 的相同，因而可不调用 LUBound 再计算一次。因为 CUB(E)>prev，所以将 X 加入活节点表。但是右儿子节点 Y 则不同，需要调用函数 LUBound 来获取 CUB(Y)= Pvu.如果 Pvu≤prev,则杀死节点 Y(即，不放在节点表中)。否则，将节点 Y 加入活节点表，并修改 prev 的值（第 21 行)。以下附上前面提到的几个子程序。

程序 7-2-2 计算节点状态下的可能取得最大效益值的上、下界

```
LUBound(P,W,cap,cv,N,k,Pvl,Pvu)// k 为当前节点的级（level），cap
  //是背包当前的剩余容量，cv 是当前背包中物品的总价值（已取得的效
  //益值），还有物品 k,…,N 要考虑
 Real rw; //随时记录本函数执行过程中是背包的剩余容量
 Pvl:=cv; rw:=cap;
 for i from k+1 to N do
   if rw<W[i] then Pvu:=Pvl+rw*P[i]/W[i];
     // 从第 k+1 件到第 N 件至少有一件物品不能装进背包的情形
     for j from i+1 to N do
       if rw≥W[j] then
           rw:=rw-W[j]; Pvl:=Pvl+P[j];
       end{if}
     end{for}
```

```
        return //此时 Pvl < Pvu
      end{if}
      rw:=rw-W[i]; Pvl:=Pvl+P[i];
    end{for}
    Pvu:=Pvl; // 从第 k+1 件物品到第 N 件物品都能装进背包的情形出现,
end{LUBound}
```

程序 7-2-3　程序生成新节点算法

```
NewNode(par,lev,t,cap,cv,ub) //生成一个新节点 J，并把它加到活节点表
  GetNode(J);
  Parent(J):=par; Level(J):=lev; Tag(J):=t;
  CC(J):=cap; CV(J):=cv; CUB(J):=ub;
  Add(J);
end{NewNode}
```

程序 7-2-4　打印答案程序

```
Finish(CV,ANS,N)//输出解
 real CV; global Tag,Parent;
 print( 'OBJECTS IN KNAPSACK ARE' )
   for j from N by -1 to 1 do
     if Tag(ANS)=1 then
       print(j);
     end{if}
     ANS:=Parent(ANS);
   end{for}
 end{Finish}
```

例子　已知 n=4, P=(10,10,12,18), W=(2,4,6,9), M=15. 试绘出算法 LFKNAP 求最优解的检索过程。

解：如图 7-2-1 所示，用红色数字标识解空间树中的各个节点。首先计算出节点 1 的上下界估值：Pvu(1)=38，Pvl(1)=32，并令 prev=32-e；因为 32>prev，此时，该节点的两个儿子节点 2，3 进入活节点表。节点 2 的上界估值最大，选为当前扩展节点。节点 2 的左儿子节点 4 的上下界估值与其父亲一致，右儿子节点 5 的上下界估值为：Pvu(3)=36，Pvl(3)=22。因为 36>prev，节点 4 进入活节

点表。在活节点表中，Pvu(4)=38 最大，节点 4 选作当前扩展节点。左儿子节点 6 的上下界估值与父亲节点相同；右儿子 7 的上下界估值为：Pvu(7)=38，Pvl(7)=38，因而 prev:=max{prev,38-e}=38-e。这两个节点都进入活节点表。选择节点 6 作为当前扩展节点。因为它的左儿子节点生成会破坏约束条件，所以不能放进活节点表；右儿子节点是答案节点，但获得的目标值 32 低于 prev 值，也不能进入活节点表。再选择节点 7 作为当前扩展节点，它的左儿子结点是答案结点，获得目标值 38，更新 Prev:=38。它的右儿子结点的上下界估值都是 20，因而不能进入活节点表。注意活节点表中的其它节点的上界估值 Pvu 都小于 Prev，所以不必再扩展了。程序结束，得到最优解(1,1,0,1)，最优目标值为 38。

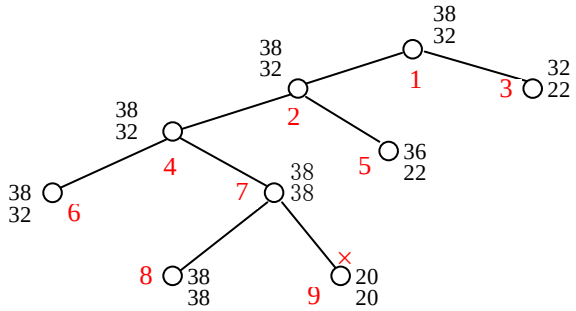
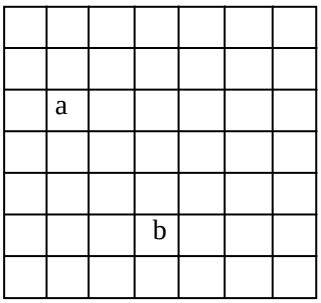


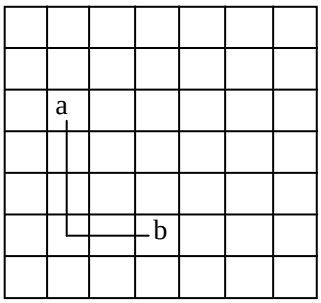
图 7-2-1 LCKNAP 的搜索过程

§3 电路板布线问题

印刷电路板将布线区域分成 n×m 个方格(阵列)，如下图(a).精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时，电路只能沿直线或直角布线，如下图(b)所示。为了避免线路相交，已布了线的方格做了封锁标记，其它线路不允许穿过被封锁的方格。



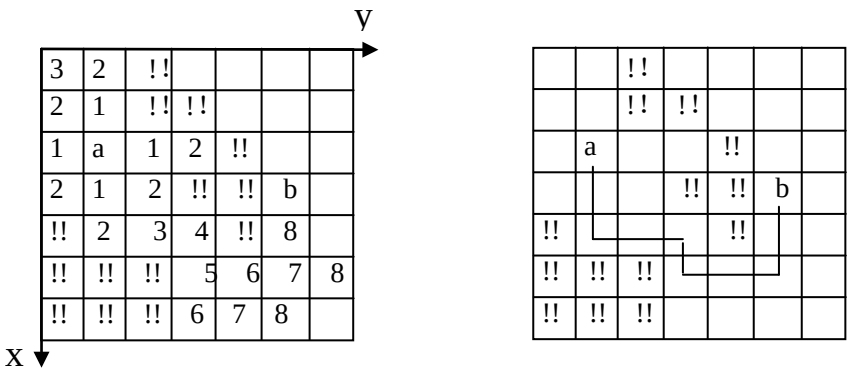
图（a）



图（b）

采用队列式(FIFO)分枝限界法解此问题，它的解空间树是一个多叉树。首先结点 a 作为第一个扩展结点。与该扩展结点相邻并且可达的方格成为可行结点被加入到活结点队列，这些结点被加入队列的顺序是：右、下、左、上。将这些

方格标记为 1，它表示从起始方格 a 到这些方格的距离为 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，并将与扩展结点相邻且未标记过的方格标记为 2，并以右、下、左、上的顺序将这些结点存放到活结点队列中。这个过程一直持续到算法搜索到目标方格 b 或者活结点队列为空时停止。注意到，搜索过程在遇到标记封锁的方格时，自动放弃此方格。下图是在 7×7 方格阵列中布线的例子。其中，起始点的位置是 a=(3,2)，目标位置是 b=(4，6).有!! 号的方格表示被封锁。搜索过程如下图(c)所示。



本例中，a 到 b 的最短距离是 9。要构造出与最短距离相应的最短路径，需从目标方格开始向起始方格回溯，逐步构造出最优解。每次向比当前方格标号小 1 的相邻方格移动，直至到达起始方格为止。图(d)给出了该例子的最短路径，它是从目标方格 b 移动到(5,6)，然后移至(6,6)，…,最终移至起始方格 a。

用 C++语言实现算法时，首先定义一个表示电路板上方格位置的类 Position，它有两个私有成员 row 和 col，分别表示方格所在的行和列。在电路板的任一方格处，布线可沿右、下、左、上四个方向进行，分别记为移动 0，1，2，3。offset[0].row=0 and offset[0].col=1 表示向右前进一步；offset[3].row=-1 and offset[3].col=0 表示向上前进一步。类似地讨论向下和向左前进的情况。一般用一个二维数组 grid 表示所给的方格阵列。初始时，grid[i][j]=0,表示该方格允许布线，而 grid[i][j]=1 表示该方格不允许布线（有封锁标记）。为了便于处理边界方格的情况，算法对所给的方格阵列的四周设置一道“围墙”，即增设标记为 1 的附加方格。

算法开始时测试初始方格与目标方格是否相同。若相同，则不必计算，直接返回最短距离 0。否则，算法设置方格阵列的围墙，初始化位移矩阵 offset。算法将起始位置的距离记为 2，这是因为数字 0 和 1 用于表示方格的开放或封闭状态，因而将所有的距离值都加 2。实际距离应为标记距离减 2。算法从起始位置 start 开始，首先标记所有标记距离为 3 的方格，并存入活结点队列，然后依次标记所有标记距离为 4，5，… 的方格，直至到达目标方格 finish 或活结点队

列为空时为止。

程序 7-3-1 布线问题的队列式分枝限界算法

```
bool FindPath(Position start,Position finish,int& PathLen,
              Position * &path)
{//计算从起点位置 start 到目标位置 finish 的最短布线路径，
 //找到最短布线路径则返回 true，否则返回 false
  if((start.row==finish.row) && (start.col==finish.col)
    {PathLen=0; return true;} //start=finish
 //设置方格阵列“围墙”
 for(int i=0; i<= m+1; i++)
   grid[0][i]=grid[n+1][i]=1; //顶部和底部
 for(int i=0; i<= n+1; i++)
   grid[i][0]=grid[i][m+1]=1; //左翼和右翼
//初始化相对位移
Position offset[4];
offset[0].row=0; offset[0].col=1;//右
offset[1].row=1; offset[1].col=0;//下
offset[2].row=0; offset[2].col=-1;//左
offset[3].row=-1; offset[3].col=0;//上
int NumOfNbrs=4;//相邻方格数
Position here,nbr;
here.row=start.row;
here.col=start.col;
grid[start.row][start.col]=2;
//标记可达方格位置
LinkedQueue<Position> Q;
do {//标记相邻可达方格
  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]==0){
     //该方格未被标记
        grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
        if((nbr.row==finish.row) &&
```

```
            (nbr.col==finish.col)) break; //完成布线
          Q.Add(nbr);}
       }
      //是否到达目标位置 finish?
      if((nbr.row==finish.row)
         (nbr.col==finish.col)) break;//完成布线
      //活结点队列是否非空?
      if(Q.IsEmpty()) return false;//无解
      Q.Delete(here);//取下一个扩展结点
   }while(true);
   //构造最短布线路径
   PathLen=grid[finish.row][finish.col]-2;
   path=new Position[PathLen];
   //从目标位置 finish 开始向起始位置回溯
   here=finish;
   for(int j=PathLen-1; j>=0; j--){
      path[j]=here;
      //找前驱位置
      for(int i=0; i<NumOfNbrs; i++){
         nbr.row=here.row+offset[i].row;
         nbr.col=here.col+offset[i];
         if(grid[nbr.row][nbr.col]==j+2) break;
       }
       here=nbr;//向前移动
     }
     return true;
   }
```

由于每个方格成为活结点进入活结点队列最多 1 次，活结点队列中最多只处理 $O(mn)$个活结点。扩展每个活结点需要 $O(1)$时间，因此共耗时 $O(mn)$。构造相应的最短距离需要 $O(L)$时间，其中，L 是最短路径的长度。

§4　优先级的确定与 LC-检索

对于优先队列式分枝限界法，节点优先级的确定直接影响着算法性能。我们当然希望具有下列特征的活节点 X 成为当前扩展节点：

1).以 X 为根的子树中含有问题的答案节点；

2).在所有满足条件 1)的活节点中，X 距离答案节点“最近”。

但是，要给可能导致答案节点的活节点附以优先级，必须要附加若干计算工作，即要付出代价。对于任一节点，与之相关的搜索成本可以使用两种标准来度量：

(i).在生成一个答案节点之前，子树 X 需要生成的节点数；

(ii).以 X 为根的子树中，离 X 最近的那个答案节点到 X 的路径长度。

容易看出，如果使用度量(i)，则对于每一种分枝限界算法，总是本着生成最小数目的节点去做；如果采用度量(ii),则要成为扩展节点的节点只是那些由根到最近的那个答案节点路径上的那些节点。用 c(•)表示最小搜索成本函数，它可以递归地定义如下：

a).如果 X 是答案节点，则 c(X)是解空间树中由根节点到 X 的搜索成本(即所用的代价，如深度、其它计算复杂度等)；

b).如果 X 不是答案节点，而且以 X 为根的子树中不含答案节点，则 c(X)应该定义为一个充分大的数，不妨以 ∞ 表示之；

c).如果 X 不是答案节点，但是以 X 为根的子树中含答案节点，则 c(X)应该是具有最小成本的答案节点的成本。

如果以最小成本函数作为优先级函数，则优先队列式分枝限界法将以最少的搜索成本找到问题的解。然而，这样的最小搜索成本函数往往是不易得到的，有时可能需要不亚于求解原问题的计算工作量。实际问题中都是采用一个成本估计函数 ，它由两部分决定：解空间树的根节点到 X 的搜索成本 f(X)，以及由 X 到答案节点的搜索成本 g(X)。

$$(X)= f(X)+g(X),$$

称为成本估计函数。

根据成本估计函数，选择下一个扩展节点的策略总是选取 (•)值最小的活节点作为当前扩展节点。这种搜索策略称为最小成本检索(Least Cost Search)，简称 LC-检索。如果取 g=0 且 f(X)等于 X 在解空间树中的深度，则 LC-检索即是宽度优先搜索(BFS)；如果 f=0，而且 g 满足：

$$Y \text{ 是 } X \text{ 的儿子} \Rightarrow g(Y) \le g(X) \qquad (7.4.1)$$

则 LC-检索即是深度优先搜索(DFS)。在以后所提到的成本函数 中，函数 g 都满足(7.4.1)式。

例子 7.4.1 15 迷问题

在一个 4×4 的棋盘上排列 15 块号牌，如图 7.4.1(a)。其中会出现一个空格。棋盘上号牌的一次合法移动是指将与空格相邻的号牌的一块号牌移入空格。15 迷问题要求通过一系列合法移动，将号牌的初始排列转换成自然排列，如图 7.4.1(b).

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

图 (a)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

图 (b)

	#		#
#		#	
	#		#
#		#	

图 (c)

如果将棋盘的各个位置按照号牌的自然排列发给序号，右下角发给 16 号。则号牌的每一种排列都可以看作是 1，…，15，16 这 16 个数的排列，其中，16 的位置代表空格。如图 7.4.1(a)对应的排列是

(1, 3, 4, 15, 2, 16, 5, 12, 7, 6, 11, 14, 8, 9, 10, 13)

事实上，不是号牌的每一种排列都能够经过一系列合法移动转换成自然排列的。用 Less(i)记排列 P 中位于 i 后面且号码比 i 小的号牌个数，则排列 P 可以经一系列合法移动转换成自然排列的充要条件是

$$\sum_{1\le i\le 16} Less(i)+\tau_0 \quad \text{是偶数} \qquad (7.4.2)$$

其中，当空格在图 7.4.1(c)的某个＃位置时，τ_0=1；否则τ_0=0. 以后记

$$\tau(P)=\sum_{1\le i\le 16} Less(i)$$

称为排列 P 的逆序数。例如，图 7.4.1(a)中排列的逆序数为 37，τ_0=0, 所以图 7.4.1(a)中排列不能经一系列合法移动转换成自然排列。

在处理实际问题中，一般是根据具体问题的特性，确定成本估价函数

(X)= f(X)+g(X)

中的函数 f(X)和 g(X). 在本例中，我们令 f(x)是由根到节点 X 的路径的长度，g(X)是以 X 为根的子树中，由 X 到目标状态(自然排列)的一条最短路径长度的估计值。为此，g(X)应是把状态 X 转换成目标状态所需的最少移动数目。对它的一种可能的选择是

g(X)= 排列 X 的不在自然位置的号牌的数目　　(7.4.3)

图 7.4.1(a)的排列中，不在自然位置的牌号分别是：3，4，15，2，5，12，7，6，14，8，9，10，13，共 13 个。g(X)=13。将图 7.4.1(a)中的排列转换成自然排列至少需要 13 次合法的移动。可见，成本估价函数 (X)是函数 c(X)的下界。以这样成本估价函数为优先级，则 15 迷问题解空间树的搜索过程可以从本章附页“15 迷问题解空间树”的图(a)中看出。

首先以节点 1 作为扩展节点，生成 4 个儿子：2，3，4，5。这 4 个儿子的成本估价分别为： (2)=1+4； (3)=1+4； (4)=1+2； (5)=1+4. 因而节点 4 成为当前扩展节点，生成 4 个儿子，但有一个儿子同其祖父相同被舍弃。剩下的 3 个儿

子的成本估值分别为 (10)=2+1； (11)=2+3； (12)=2+3，此时，活节点表中共有 6 个节点：2，3，5，10，11，12，其中节点 10 的成本估值最小。故以节点 10 作为新的扩展节点。它能生成 3 个儿子，其中有一个因同其祖父相同而被舍弃。剩下的 2 个儿子是节点 22 和 23。但节点 23 所表示的排列是自然的，至此得到可行解。

如果采用队列式分枝限界法，即采用宽度优先搜索，势必将图 7.4.2(a)中的所有状态全部搜索。如果采用深度优先搜索，则图 7.4.2(b)中也只给出了一部分搜索步骤，而且节点 11 表示的号牌排列与自然排列还相距甚远。由此看出选择合适的优先级函数对于分枝限界法效率的影响甚大。

值得注意的是按照成本估价函数 (X)确定的优先级进行搜索，所得到的答案节点未必是最小成本答案节点。例如，在后面的图 7-4-3 所示的解空间树中，

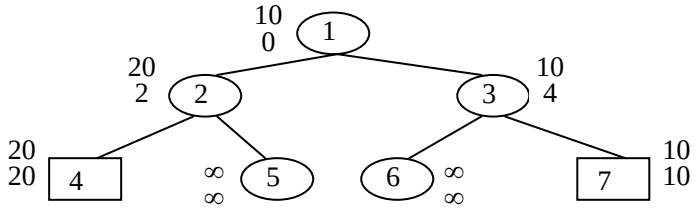


图 7-4-3 一棵假想的解空间树

每个节点 X 旁有两个数：上面的是最小成本 c(X),下面的是最小成本估价 (X)。整个解空间树中有两个答案节点。如果按照估价函数进行搜索，根节点 1 首先成为扩展节点，然后是节点 2，在扩展节点 2 得到一个答案节点 4，它是节点 2 的一个儿子。这个答案节点的成本是 20，比另一个答案节点 7 的成本大。

定理 7.4.1 在有限的解空间树中，如果对每对节点 X 和 Y 都有

“c(X) ＜ c(Y)” ＝＞ “ (X) ＜ (Y)”

则按照最小成本估计函数搜索能够达到最小成本答案节点。

一般情况下，不易找到定理 7.4.1 中要求的成本估计函数。但是，对于成本估计函数一般有一个基本要求：

(X)≤ c(X)，对任意节点 X； (7.4.4)

(X)＝c(X)，当 X 是答案节点时。 (7.4.5)

在以下给出的算法中，要求搜索一直继续到一个答案节点变成扩展节点为止。

程序 7-4-1 搜索最小成本答案节点的 LC－检索

```
LCSearch(T,  ) //T 是问题的解空间树
```

```
1    E:=T; //第一个扩展节点
2    置活节点表为空;
3    loop
4      if E 是答案节点 then
5        输出从E到T的路径; return;
6      end{if}
7      for E的每个儿子X  do
8        Add(X); Parent(X):=E;
9      end{for}
10     if 不再有活节点  then
11       print('no answer node'); stop;
12     end{if}
13     Least(E);
14   endloop
15 end{LeastCost}
```

其中，Least(X)是从活节点表中找一个具有最小　值的活节点，并从活节点表中删除该节点，再将该节点赋给变量X；Add(X)是将新的活节点加到活节点表中。Parent(X)将活节点X与它的父亲相连接。在算法LCSearch中，由于答案节点E是扩展节点，所以，对于活节点表中的每个节点P均有　(E)≤　(P)。再由假设　(E)=c(E)，　(P) ≤ c(P)，得c(E)≤ c(P)，即E是一个最小成本答案节点。

综上所述，采用优先队列式分枝限界法，其搜索解空间树的算法主要决定于三个因素：约束函数、限界函数和优先级函数。前两项主要是限制被搜索的节点数量，而优先级函数则是用来确定搜索的方向。但是，在处理实际问题中，未必把这三种函数全部列出来，特别是优先级函数往往同限界函数合为一体。以下给出处理最小值最优问题的LC-分枝限界法的一般流程，这里“成本”就采用问题解的目标值。

程序7-4-2　最小化问题的LC-分枝限界算法

```
proc LCBB(T,  ,u,ε,cost) //假定解空间树T包含一个解节点X且
  // (X) ≤ c(X)≤ u(X)。c(X)是最小成本函数，一般都是未知的;  (X)是
  //最小成本估价函数，  u(X)是成本上界估计函数; cost(X)是X所对应的
  //解的成本。ε 是一个充分小的正数。
1    E:=T; Parent(E):=0;
2    if T 是解节点 then U:=min(cost(T),u(T)+ ε); ans:=T;
```

```
3     else U:=u(T)+ε; ans:=0; // 将活节点表初始化为空集;
4   end{if}
5   loop
6     for E 的每个儿子节点 X do
7       if  (X)<U && X 是一个可行节点 then
8          Add(X); Parent(X):=E;  //把 X 加入活节点 表
9          case:
10          X 是解节点 && cost(X)< U:
11            U:=min(cost(X),u(X)); ans:=X;
12          u(X) < U:
13            U:=u(X);
14         end{case}
15      end{if}
16    end{for}
17    if 不再有活节点 or 下一个扩展节点满足  ≥ U then
18      print( 'least cost=' ,U);
19      while ans≠0 do
20        print(ans); ans:=Parent(ans);
21      end{while}
22      return;
23    end{if}
24    Least(E);
25  end{loop}
27 end{LCBB}
```

这里我们将选择扩展节点的任务交给函数 Least(E)。所谓的“成本”是指目标函数的值。所以，当 X 是可行解的答案节点时，c(X)表示该可行解的目标函数值 cost(X)；当 X 是不可行节点时，c(X)=∞；当 X 是可行节点但不是答案节点时，c(X)表示以 X 为根的子树中**最小成本答案节点**的成本。此时，成本估价函数 (X)即变成对于节点 X 的目标函数值的一个估计。我们要求 (X) ≤ c(X)。算法中的 U 是一个上界值，开始时，U 可以取为一个充分大的数。**例如**，假定一个可行解需要确定 n 个数 $x_1,x_2,\cdots,x_n$，而每个取值 x_i 会使得成本开销增加 c_ix_i，于是，一个可行解 X=$(x_1,x_2,\cdots,x_n)$的总开销为 $c(X)=\sum_{1\le i\le n}c_ix_i$。

若当前只确定了前 k 个数 $x_1,x_2,\cdots,x_k$，用节点 X 表示当前之状态，则以这 k

个数为部分解的可行解的开销一定不小于 $\hat{c}(X)=\sum_{1\le i\le k}c_i x_i$，不大于

$$u(X)=\sum_{k+1\le j\le n}c_j\overline{x}_j+\sum_{1\le i\le k}c_i x_i$$

其中 $\overline{x}_j$ 是 x_j 可取的最大值。这样选取的函数 (X)和 u(X)就会满足上面我们提出的要求：

$$(X)\ \le\ c(X)\le u(X)$$

从上面的例子可以看出：如果 X 是成本值为 cost(X)的解节点，且 cost(X)< U,则可用 min{cost(X),u(X)}更新 U 的值；或者 X 不是解节点，但 u(X)< U，可用 u(X)更新 U 的值。如果 U 是某个可行解（或答案节点）的成本值（开销），则当 (X)≥U 时，可以杀死节点 X。但是，当 U 只是一个简单的上界，不是某个可行解的成本值时，应该谨慎考虑 (X)=U 的情况。因为，如果 X 不是答案节点，则 X 的某个子孙 Y 可能是答案节点，以 X 为根的子树中含有答案节点。如果前面盲目地杀死节点 X，则可能丢失一些最小成本答案节点。所以，在这种情况下，不能杀死节点 X。为此，在用 u(X)更新 U 值时，常常多加一个微量ε（ε的选取应当满足：“u(X)<u(Y)⇒ u(X)+ ε < u(Y)”，即用 u(X)+ ε 更新 U。于是，U 的更新可以描述为：设 E 是当前扩展节点，X 是 E 的一个儿子节点:

(1).当 X 是一个解节点，而且 cost(X)< U 时，可用 min(cost(X),u(X)+ ε)来更新 U;

(2).当 X 不是解节点，而且 u(X)+ ε <U 时，直接用 u(X)+ ε 更新 U。

§5　旅行商问题

本节采用最小成本分枝限界算法求解旅行商问题。设 G=(V,E)是旅行商问题实例中定义的有向图，边(i,j)的成本为 c_{ij}，若(i,j)∉E，则 $c_{ij}=\infty$，图 G 的顶点数为|V|=n。不失一般性，假设每次环游都是从顶点 1 开始和结束：(1,i_1,…,i_{n-1},1)。其状态空间树中每条从根节点到叶节点的一条路径唯一代表一条 Hamilton 回路，其成本是该回路上所有有向边的代价之和，这样的环游有(n-1)!个，如第六章第一节所指出的，它的状态空间树是一棵排列树。使用最小成本分枝限界算法，需要定义状态空间树中各节点的成本函数 c(X)，和上下界函数 u(X)和ĉ(X)。成本函数可以如下定义：

$$C(X)=\begin{cases}\text{从根到 X 的路径所定义的回路成本，当 X 是叶节点时；}\\ \text{子树 X 中一个最小成本叶节点的成本，当 X 不是叶节点时。}\end{cases}$$

定义ĉ(X)为根节点到节点 X 的路径的成本，显然有ĉ(X)≤C(X)，但是，当 X 是叶节点时，ĉ(X)≠C(X)。实际上，我们可以用简约成本矩阵获得更好的ĉ(X)，它满

足ĉ(X)≤C(X)，而且在 X 是叶节点时，ĉ(X)=C(X)。矩阵 A 称为简约的，如果它的每个元素都是非负的，而且它的每一行（列）都至少有一个元素是 0。我们可以将旅行商问题的成本矩阵(即表示它的有向赋权图的邻接矩阵)约化成简约矩阵：先将矩阵的每一行各元素减去该行的最小元素，再将所得矩阵的每一列各元素减去该列的最小元素。因为有向图的一条有向 hamilton 回路含有每个顶点 i 的恰好一条出边(i,j)和一条入边(k,i)，从成本矩阵的一列或者一行的每一个项减去一个常数 t，就会使得每条回路成本正好减少 t。这不改变最低成本回路，即最小成本回路仍然是最小成本回路。如果 i 行使用的减数是 r_i，j 列使用的减数是 c_j，则旅行商问题的最小回路成本不会低于这些数的和：

$$\hat{c} = \sum_{i=1}^{n} r_i + \sum_{j=1}^{n} c_j$$

因此，$\hat{c}$ 可以作为状态空间树中根节点的下界估值，上述简约矩阵称为根节点的简约矩阵。对于其它的节点 S，设 R 是它的父亲节点，状态空间树中边(R,S)对应 Hamilton 回路中包含的边(i,j)。如果 S 不是叶节点，S 的简约矩阵 A_S 可以通过修简 R 的简约矩阵 A_R 而得：(1)将 A_R 中 i 行和 j 列的所有项都改为∞，防止任何其它离开顶点 i 的边，进入顶点 j 的边的使用。(2)将 A_R 的(j,1)元素置为∞，防止使用边(j,1)。(3)约简经过(1)、(2)两步操作后得到的矩阵。上面三步操作后得到的矩阵就作为节点 S 的简约矩阵，并且给出节点 S 的下界估值如下：

$$\hat{c}(S) = \hat{c}(R) + A_R(i,j) + \tilde{c}$$

其中，$A_R(i,j)$ 是矩阵 A_R 的 (i,j) 元素，$\tilde{c}$ 是约简步骤(3)施行时减掉的总数。如果 S 是叶节点，则直接计算 $\hat{c}(S) = c(S)$ 即可，不必计算伴随矩阵。

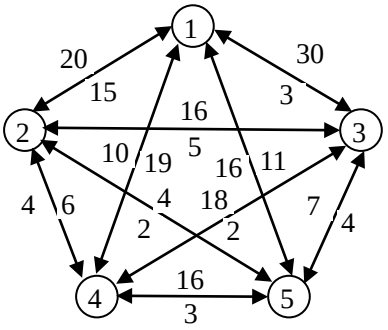
每个上界估值可用 u(x)=∞。

考察下面成本矩阵 A 描述的旅行商问题，对应有向赋权图如图 7-5-1 所示，它的简约矩阵为 A(1)。采用最小成本分枝限界算法，搜索最优解的过程如下，产生的状态空间树由图 7-5-2 表示。

$$A = \begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix} \qquad A(1) = \begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$$

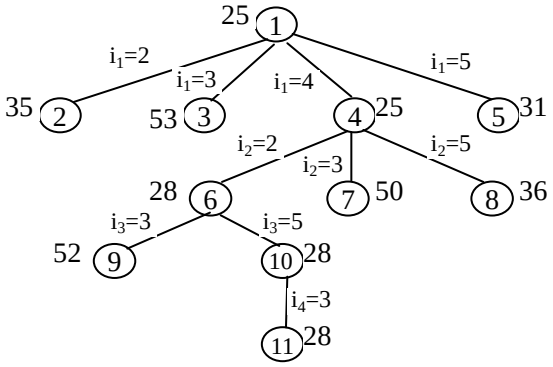
令 Prev 表示目前所知的最小回路成本，初始时 Prev=∞。根节点 1 的简约矩阵就是成本矩阵的简约矩阵 A(1)，而ĉ(1)=25。从根节点作为可扩展节点开始，

依次序产生节点 2,3,4,5。对应这些节点的简约矩阵分别是 A(i),i=2,3,4,5。例如,简约矩阵 A(3)对应的路径是有向图的顶点 1 到顶点 3,该矩阵是这样获得的:将矩阵 A(1)的第 1 行、第 3 列所有项及(3,1)项都置为∞,这样得到的矩阵不是简约的,但只有第 1 列没有零,将第 1 列的每项都减去 11,得到简约矩阵 A(3)。此时,节点 3 的成本下界估值为ĉ(3)=25+17+11=53。



由小到大权值标于外侧,由大到小标于内侧

图 7-5-1 有五个顶点的赋权有向图



节点外面的数字是ĉ 值

图 7-5-2 用 LC-搜索产生的状态空间树

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{pmatrix}$$

A(2),路径 1,2

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{pmatrix}$$

A(3),路径 1,3

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

A(4),路径 1,4

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{pmatrix}$$

A(5),路径 1,5

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

A(6),路径 1,4,2

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{pmatrix}$$

A(7),路径 1,4,3

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

A(8),路径 1,4,5

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{pmatrix}$$

A(9),路径 1,4,2,3

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

A(10),路径 1,4,2,5

其它三个节点的成本下界估值分别为:ĉ(2)=35, ĉ(4)=25, ĉ(5)=31。节点 4 选为当前扩展节点,产生子节点 6,7,8,它们的成本下界估值分别为:ĉ(6)=28,

ĉ(7)=50，ĉ(8)=36。现在活节点表中有节点 2,3,5,6,7,8，节点 6 被选为当前扩展节点，生成其子节点 9,10，它们的成本下界估值分别是ĉ(9)=52，ĉ(10)=28，活节点表中的节点 10 称为当前扩展节点。它只有一个子节点 11，而且是解节点，其成本下界估值即是其成本值ĉ(11)=c(11)=28。更新 Prev 的值：Prev:=28。注意到目前活节点表中所有节点的成本下界估值都大于 Prev，所以，不需要继续扩展，程序执行至此结束。得到该旅行商问题的最低成本环游：(1,4,2,5,3,1)，成本为 28。

习题 七

1. 假设对称旅行商问题的邻接矩阵如图 1 所示，试用优先队列式分枝限界算法给出最短环游。画出解空间树的搜索图，并说明搜索过程。

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ & \infty & 16 & 4 & 2 \\ & & \infty & 6 & 7 \\ & & & \infty & 12 \\ & & & & \infty \end{pmatrix}$$

图 1 邻接矩阵

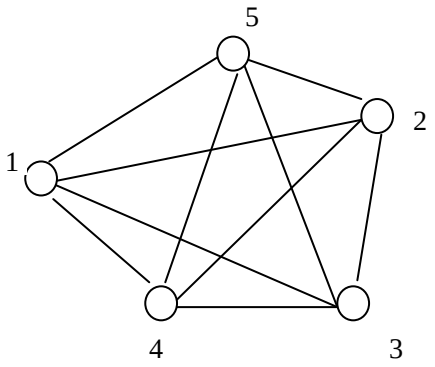
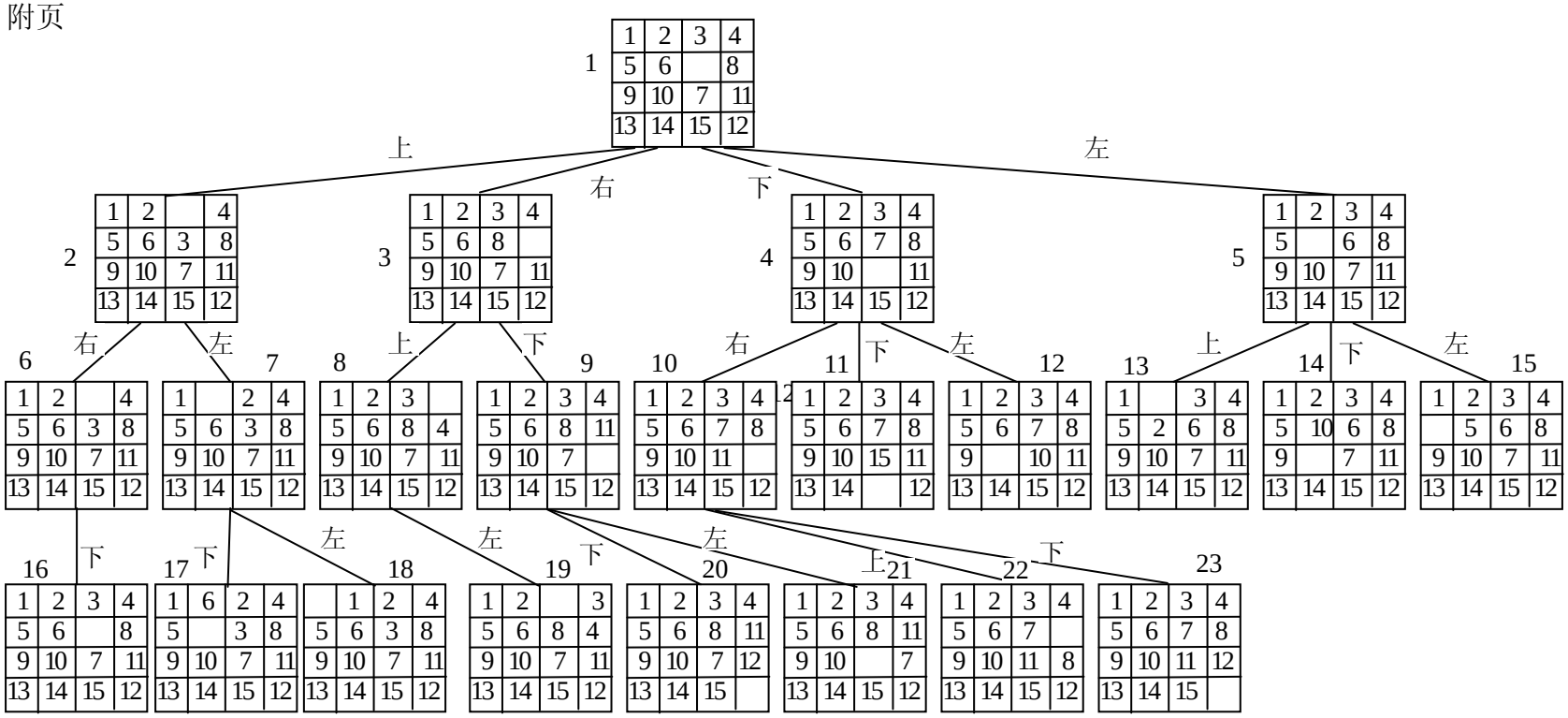


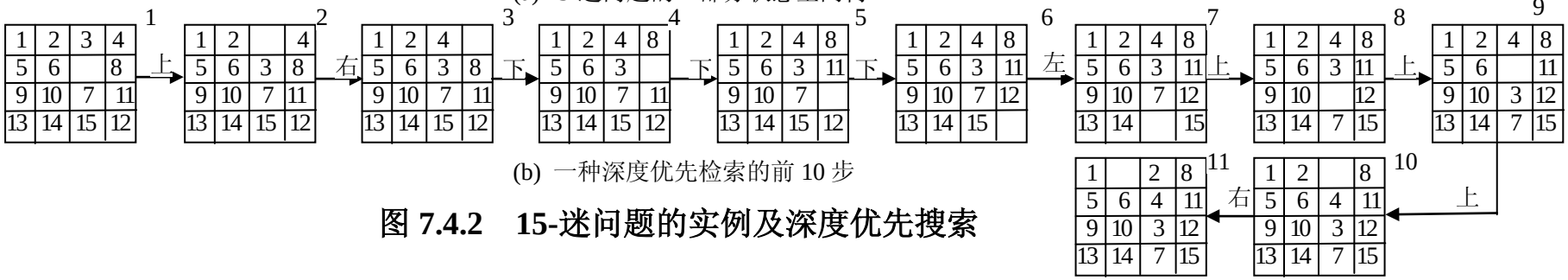
图 2 旅行商问题

2. 试写出 0/1 背包问题的队列式分枝限界算法程序，并找一个物品个数是 16 的例子检验程序的运行情况。
3. 最佳调度问题:假设有 n 个任务要由 k 个可并行工作的机器来完成，完成任务需要的时间为 t_i。试设计一个分枝限界算法，找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最短。

附页



(a) 15-迷问题的一部分状态空间树



(b) 一种深度优先检索的前 10 步

图 7.4.2　15-迷问题的实例及深度优先搜索