分支与限界

分支与限界法的基本思想

一、基本思想：

1、在 $e\_$ 结点估算沿着它的各儿子结点搜索时，目标函数可能取得的“界”，

2、把儿子结点和目标函数可能取得的“界”，保存在优先队列或堆中，

3、从队列或堆中选取“界”最大或最小的 $e\_$ 结点向下搜索，直到叶子结点，

4、若叶子结点的目标函数的值，是结点表中的最大值或最小值，则沿叶子结点到根结点的路径所确定的解，就是问题的最优解，由该叶子结点所确定的目标函数的值，就是解这个问题所得到的最大值或最小值

二、目标函数“界”的特性：

$(x_1)\cdots(x_1,x_2,\cdots,x_k)$ 是部分解，$bound\,(x_1),\ \cdots bound\,(x_1,\ x_2,\cdots,x_k)$ 是相应的界

1、对最小值问题，称为下界，意思是向下搜索所可能取得的值最小不会小于这些下界。

若 $X=(x_1,x_2,\cdots,x_k)$ 是所得到的部分解，满足：

$$bound\,(x_1)\le bound\,(x_1,\ x_2)\le\cdots\le bound\,(x_1,\ x_2,\cdots,x_k) \qquad (8.1.1)$$

2、对最大值问题，称为上界，意思是向下搜索所可能取得的值最大不会大于这些上界。

若 $X=(x_1,x_2,\cdots,x_k)$ 是所得到的部分解，满足：

$$bound\,(x_1)\ge bound\,(x_1,\ x_2)\ge\cdots\ge bound\,(x_1,\ x_2,\cdots,x_k)$$

三、两种分支方法：

设解向量 $X=(x_1,x_2,\cdots,x_n)$，　x_i 的取值范围为有穷集 S_i，$|S_i|=n_i$，$1\le i\le n$。

1、每棵子树都有 n_i 个分支：

最坏情况下，结点表的空间为 $O(n_1\times n_2\times\cdots\times n_n)$，

若状态空间树是完全 n 叉树，　$n_1=n_2=\cdots=n_n=n$　，结点表的空间为 $O(n^n)$。

2、每棵子树只有两个分支，　x_i 取特定值的分支、及不取特定值的分支：

状态空间树是完全二叉树，最坏情况下结点表的空间为 $O(2^n)$

货郎担问题

有向赋权图 $G=(V,E)$，顶点集 $V=(v_0,v_1,\cdots,v_{n-1})$。

c 为图的邻接矩阵，　c_{ij} 表示顶点 v_i 到顶点 v_j 的关联边的长度，又把 c 称为费用矩阵。

费用矩阵的特性及归约

l：图 G 的最短哈密尔顿回路，

$w(l)$：回路的费用。因为中的元素 c_{ij} 表示顶点 v_i 到顶点 v_j 的关联边的费用，

一、哈密尔顿回路与费用矩阵的关系：

引理 8.1 令 $G=(V,E)$ 是一个有向赋权图，l 是图 G 的一条哈密尔顿回路，c 是图 G 的费用矩阵，则回路上的边对应于费用矩阵 c 中每行每列各一个元素。

证明 图 G 有 n 个顶点，

费用矩阵第 i 行元素：顶点 v_i 到其它顶点的出边费用；

费用矩阵第 i 列元素：其它顶点到顶点 v_i 的入边费用。

l 是图 G 的一条哈密尔顿回路，

v_i 是回路中的任意一个顶点，$0 \le i \le n-1$，

v_i 在回路中只有一条出边，对应于费用矩阵中第 i 行的一个元素；

v_i 在回路中只出现一次，费用矩阵的第 i 行有且只有一个元素与其对应。

v_i 在回路中只有一条入边，费用矩阵中第 i 列也有且只有一个元素与其对应。

回路中有 n 个不同顶点，费用矩阵的每行每列都有且只有一个元素与回路中的顶点的出边与入边一一对应。

例：，图 8.1(a)中 5 城市的货郎担问题的费用矩阵，

令 $l=v_0v_3v_1v_4v_2v_0$ 是哈密尔顿回路，回路上的边对应于费用矩阵中的元素 $c_{03},c_{31},c_{14},c_{42},c_{20}$。

	0	1	2	3	4
0	∞	25	41	32	28
1	5	∞	18	31	26
2	20	16	∞	7	1
3	10	51	25	∞	6
4	23	9	7	11	∞

(a)

	0	1	2	3	4	
0	∞	0	16	7	3	lh0 = 25
1	0	∞	13	26	21	lh1 = 5
2	19	15	∞	6	0	lh2 = 1
3	4	45	19	∞	0	lh3 = 6
4	16	2	0	4	∞	lh4 = 7

(b)

	0	1	2	3	4
0	∞	0	16	3	3
1	0	∞	13	22	21
2	19	15	∞	2	0
3	4	45	19	∞	0
4	16	2	0	0	∞

ch3 = 4

(c)

图 8.1　5 城市货郎担问题的费用矩阵及其归约

二、费用矩阵的归约

1、行归约和列归约

定义 8.1 费用矩阵 c 的第 i 行（或第 j 列）中的每个元素减去一个正常数 lh_i（或 ch_j），得到一个新的费用矩阵 $\bar{c}$，使得 $\bar{c}$ 中第 i 行（或第 j 列）中的最小元素为 0，称为费用矩阵的行归约(或列归约)。称 lh_i 为行归约常数，称 ch_j 为列归约常数。

例：把图 8.1(a)中归约常数 $lh_0=25$，$lh_1=5$，$lh_2=1$，$lh_3=6$，$lh_4=7$。

列归约常数 $ch_3=4$，所得结果如图 8.1(c)所示。

2、归约矩阵

定义 8.2 对费用矩阵c的每一行和每一列都进行行归约和列归约，得到一个新的费用矩阵$\bar{c}$，使得$\bar{c}$中每一行和每一列至少都有一个元素为 0，称为费用矩阵的归约。矩阵$\bar{c}$称为费用矩阵c的归约矩阵。称常数h

$$h = \sum_{i=0}^{n-1} lh_i + \sum_{i=0}^{n-1} ch_i \qquad (8.2.1)$$

为矩阵c的归约常数。

例：对图 8.1(a)中的费用矩阵进行归约，得到图 8.1(c)所示归约矩阵。

归约常数h为

$$h = 25+5+1+6+7+4 = 48$$

3、归约矩阵与哈密尔顿回路的关系

定理 8.1 有向赋权图$G=(V,E)$，G的哈密尔顿回路l，G的费用矩阵c，$w(l)$是以c计算的回路费用。$\bar{c}$是c的归约矩阵，归约常数为h，$\bar{w}(l)$是以$\bar{c}$计算的回路费用，有：

$$w(l) = \bar{w}(l) + h \qquad (8.2.2)$$

证明 c_{ij}和$\bar{c}_{ij}$分别是c和$\bar{c}$的第i行第j列元素，

$c_{ij} = \bar{c}_{ij} + lh_i + ch_j \qquad i, j, 0 \le i, j \le n-1$，

$w(l)$是以c计算的哈密尔顿回路的费用，令

$$w(l) = \sum_{i,j \in l} c_{ij}$$

$\bar{w}(l)$是$\bar{c}$计算的同一条哈密尔顿回路的费用，令

$$\bar{w}(l) = \sum_{i,j \in l} \bar{c}_{ij}$$

由引理 8.1，回路上的边对应于c中每行每列各一个元素。有

$$w(l) = \sum_{i,j \in l} c_{ij} = \sum_{i,j \in l} \bar{c}_{ij} + \sum_{i=0}^{n-1} lh_i + \sum_{j=0}^{n-1} ch_j = \bar{w}(l) + h$$

定理证毕。

定理 8.2 有向赋权图$G=(V,E)$，l是G的最短哈密尔顿回路，c是G的费用矩阵，$\bar{c}$是c的归约矩阵，令$\bar{c}$是图$\bar{G}$的邻接矩阵，则l也是$\bar{G}$的最短的哈密尔顿回路。

证明 用反证法证明。

若l不是图$\bar{G}$的最短的哈密尔顿回路，

则$\bar{G}$中必存在另一条回路l^*，是$\bar{G}$中最短的哈密尔顿回路，

同时，它也是G中的一条回路。

$\bar{w}(l)$和$\bar{w}(l^*)$分别是以$\bar{c}$计算的l和l^*的费用，有：

$$\bar{w}(l) = \bar{w}(l^*) + \delta \qquad \text{其中，}\delta\text{是正整数。}$$

l^*是G的一条回路，令$w(l)$和$w(l^*)$是分别以c计算的回路l和l^*的费用。

由定理 8.1，有

$$w(l)=\bar{w}(l)+h \qquad w(l^*)=\bar{w}(l^*)+h$$

其中，h 是费用矩阵 c 的归约常数。因此

$$w(l)=\bar{w}(l)+h=\bar{w}(l^*)+\delta+h$$
$$=w(l^*)+\delta$$

l^* 是 G 中比 l 更短的哈密尔顿回路，与定理的前提相矛盾。

所以，l 也是 $\bar{G}$ 的最短的哈密尔顿回路。

界限的确定和分支的选择

先求图 G 费用矩阵 c 的归约矩阵 $\bar{c}$，得到归约常数 h

再转换为求取与 $\bar{c}$ 相对应的图 $\bar{G}$ 的最短哈密尔顿回路问题。

$w(l)$ 和 $\bar{w}(l)$ 分别是 G 和 $\bar{G}$ 的最短哈密尔顿回路的费用，

有 $w(l)=\bar{w}(l)+h$。

G 的最短哈密尔顿回路的费用，最少不会少于 h。

h 是货郎担问题状态空间树中根结点 X 的下界。$w(X)=h$

例：图 8.1(a)中归约常数 48 便是该问题的下界。该问题的最小费用不会少于 48。

8.2.2.1 界限的确定

1、搜索策略

选取沿某一边出发的路径，作为分支结点 Y；

不沿该边出发的其它所有路径集合，作为另一个分支结点 $\bar{Y}$。

2、选取沿 (i,j) 方向的路径时，结点 Y 下界 $w(Y)$ 的确定

G 的哈密尔顿回路 l，费用矩阵 c，以 c 计算的回路费用 $w(l)$。

$\bar{c}$ 是 c 的归约矩阵，归约常数为 h，以 $\bar{c}$ 计算的回路费用 $\bar{w}(l)$，

$w(l)=\bar{w}(l)+h=\bar{w}(l')+\bar{c}_{ij}+h$

1）$\bar{c}_{ji}=\infty$，处理不可能经过的边

2）矩阵降阶，删去第 i 行第 j 列所有元素，得到降阶后的矩阵 c'

3）归约 c'，得归约常数 h'，有 $\bar{w}(l')=\bar{w}(\bar{l}')+h'$

$w(l)=\bar{w}(\bar{l}')+\bar{c}_{ij}+h+h'$

$w(Y)=h+h'$

例：图 8.1(a)及图 8.1(c)的 5 城市货郎担问题的费用矩阵、及其归约矩阵。

选取从顶点 v_1 出发，沿着 (v_1,v_0) 的边前进，

则该回路的边包含费用矩阵中的 $\bar{c}_{10}$。

删去 $\bar{c}$ 中的第 1 行和第 0 列的所有元素，

素 $\bar{c}_{01}$ 置为∞。

图 8.1(c)中 5×5 的归约矩阵，降阶为图 8.2(b)所示的 4×4 的矩阵。

进一步进行归约，得到图 8.2(c)所示的归约矩阵，其归约常数为 5。

表明沿 v_1 出发，经边 (v_1, v_0) 的回路，其费用至少不会小于 48＋5 = 53。

	0	1	2	3	4
0	∞	0	16	3	3
1	0	∞	13	22	21
2	19	15	∞	2	0
3	4	45	19	∞	0
4	16	2	0	0	∞

（a）

	1	2	3	4
0	∞	16	3	3
2	15	∞	2	0
3	45	19	∞	0
4	2	0	0	∞

（b）

	1	2	3	4	
0	∞	13	0	0	lh0 = 3
2	13	∞	2	0	
3	43	19	∞	0	
4	0	0	0	∞	

ch1 = 2

（c）

图 8.2　Y 结点对费用矩阵的降阶处理

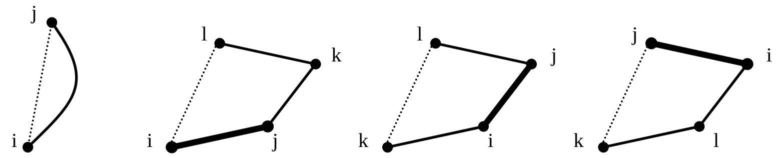
4）处理不可能经过的边：

（1） $v_i v_j$ 不和其它已经选择的边相连接，把 c_{ji} 置为∞，如图 8.3(a)所示；

（2） 和以前选择的边连接成 $v_i v_j v_k v_l$，把 c_{li} 置为∞，如图 8.3(b)所示；

（3） 和以前选择的边连接成 $v_k v_i v_j v_l$，把 c_{lk} 置为∞，如图 8.3(c)所示；

（4） 和以前选择的边连接成 $v_k v_l v_i v_j$，把 c_{jk} 置为∞，如图 8.3(d)所示；



(a)　(b)　(c)　(d)

图 8.3　选择有向边时的几种可能情况

5） 父亲结点 X，下界 $w(X)$，降阶后的归约常数为 h，结点 Y 的下界为

$$w(Y) = w(X) + h \qquad (8.2.3)$$

3、不沿 (i, j) 方向的结点 $\overline{Y}$ 下界 $w(\overline{Y})$ 的确定

1）回路不包含 $v_i v_j$ 边， c_{ij} 置为∞。（不降阶）

2）$d_{ij} = \min\limits_{0 \le k \le n-1, k \ne j}\{c_{ik}\} + \min\limits_{0 \le k \le n-1, k \ne i}\{c_{kj}\}$ （8.2.4）

3）结点 $\overline{Y}$ 的下界为：

$$w(\overline{Y}) = w(X) + d_{ij} \qquad (8.2.5)$$

例：在图 8.1(a)中，根结点作为父亲结点 X，则 $w(X) = 48$。

选择边 (v_1, v_0) 向下搜索作为结点为 Y，结点为 Y 的下界为：

$$w(Y) = w(X) + h = 48 + 5 = 53$$

结点 $\overline{Y}$ 的下界为：

$$w(\overline{Y}) = w(X) + d_{ij} = 48 + 4 + 13 = 65$$

8.2.2.2 分支的选择

选择分支的思想方法：

1. 沿 $c_{ij}=0$ 的方向选择，使所选择的路线尽可能短；
2. 沿 d_{ij} 最大的方向选择，使 $w(\overline{Y})$ 尽可能大；

   令 S 是 $c_{ij}=0$ 的元素集合，D_{kl} 是 S 中使 d_{ij} 达最大的元素 d_{kl}，即：

$$D_{kl}=\max_{S}\{d_{ij}\} \tag{8.2.6}$$

   边 $v_k v_l$ 就是所选择的分支方向。

例：图 8.1(a)中的费用矩阵归约为 8.1(c)中矩阵，根结点的下界 $w(X)=48$

有 $c_{01}=c_{10}=c_{24}=c_{34}=c_{42}=c_{43}=0$，搜索方向的选择如下：

$d_{01}=3+2=5$　　$d_{10}=13+4=17$　　$d_{24}=2+0=2$

$d_{34}=4+0=4$　　$d_{42}=0+13=13$　　$d_{43}=0+2=2$

$D_{kl}=d_{10}=17$ 。

所选择的方向为边 v_1v_0，据此建立结点 Y 和 $\overline{Y}$ 。此时，

$$w(\overline{Y})=w(X)+D_{kl} \tag{8.2.7}$$

货郎担问题的求解过程

结点数据结构：

```
typedef struct node_data {
    Type   c[n][n];          /* 费用矩阵 */
    int    row_init[n];      /* 费用矩阵的当前行映射为原始行 */
    int    col_init[n];      /* 费用矩阵的当前列映射为原始列 */
    int    row_cur[n];       /* 费用矩阵的原始行映射为当前行 */
    int    col_cur[n];       /* 费用矩阵的原始列映射为当前列 */
    int    ad[n];            /* 回路顶点邻接表 */
    int    k;                /* 当前费用矩阵的阶 */
    Type   w;                /* 结点的下界 */
} NODE;
```

分支限界法求解货郎担问题的求解过程：

1. 分配堆缓冲区，初始化为空堆；
2. 建立结点 X， c 拷贝到 $X.c$， $X.k$ 初始化为 n；归约 $X.c$，计算归约常数 h，下界 $X.w=h$；初始化回路的顶点邻接表 $X.ad$；
3. 按(8.2.4)式，由 $X.c$ 中所有 $c_{ij}=0$ 的元素 c_{ij}，计算 d_{ij}；
4. 按(8.2.6)式，选取使 d_{ij} 达最大的元素 d_{kl} 作为 D_{kl}，选择边 v_kv_l 作为分支方向；
5. 建立儿子结点 $\overline{Y}$，$X.c$ 拷贝到 $\overline{Y}.c$，$X.ad$ 拷贝到 $\overline{Y}.ad$，$X.k$ 拷贝到 $\overline{Y}.k$；把 $\overline{Y}.c$ 中的 c_{kl}

置为∞，归约$\overline{Y}.c$；计算结点$\overline{Y}$的下界$\overline{Y}.w$；把结点$\overline{Y}$按$\overline{Y}.w$插入最小堆中；

6. 建立儿子结点Y， $X.c$拷贝到$Y.c$， $X.ad$拷贝到$Y.ad$， $X.k$拷贝到$Y.k$； $Y.c$的有关元素置为∞；
7. 降阶$Y.c$， $Y.k$减 1，归约降阶后的$Y.c$，按(8.2.3)式计算结点Y的下界$Y.w$；
8. 若$Y.k = 2$，直接判断最短回路的两条边，并登记于路线邻接表$Y.ad$，使$Y.k = 0$；
9. 把结点Y按$Y.w$插入最小堆中；
10. 取下堆顶元素作为结点X，若$X.k = 0$，算法结束；否则，转 3；

例 8.1 求解图 8.1(a)所示的 5 城市货郎担问题。

该问题的求解过程如图 8.4 所示，过程如下：

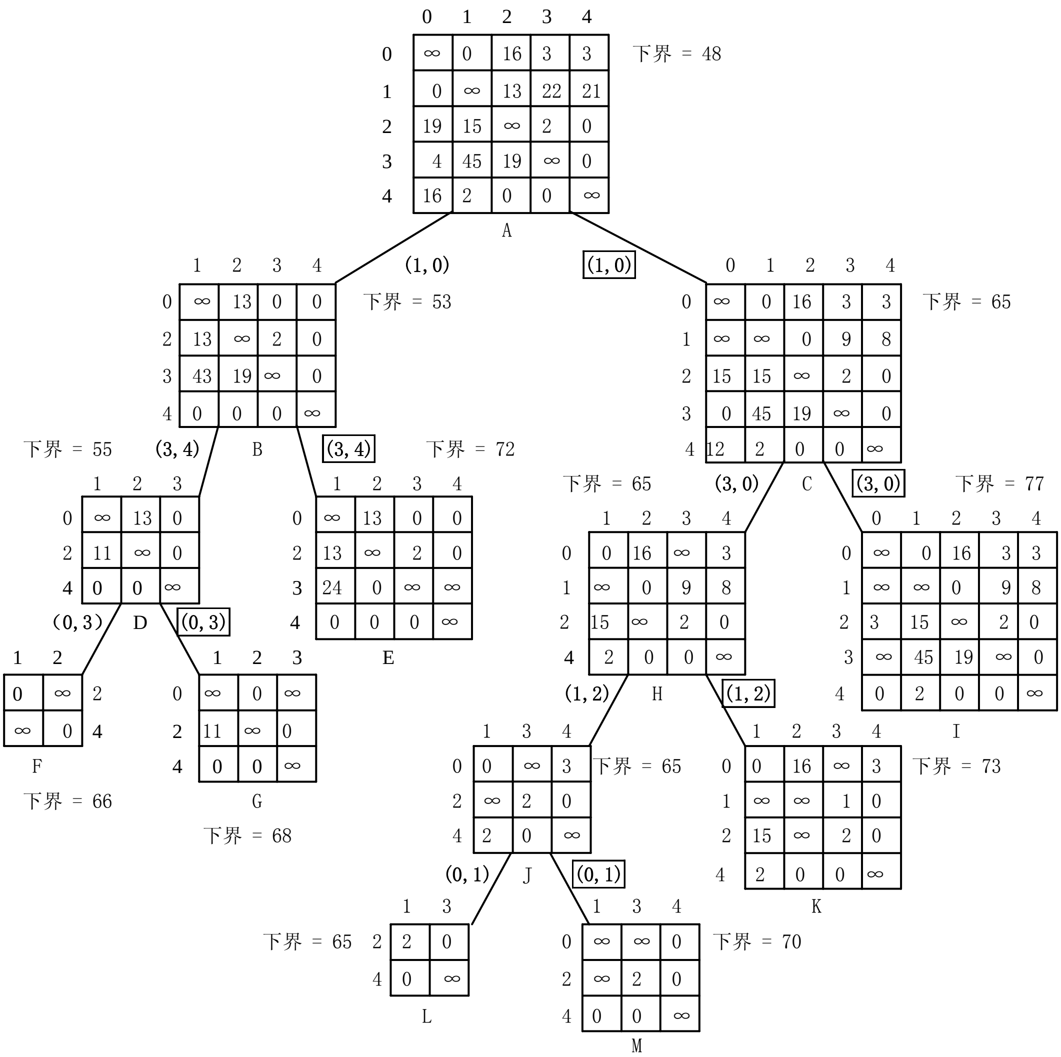


图 8.4 用分支限界法解 5 城市货郎担问题的过程

几个辅助函数的实现

数据结构：

```
typedef struct node_data {
    Type  c[n][n];          /* 费用矩阵 */
    int   row_init[n];      /* 费用矩阵的当前行(下标)映射为原始行(内容) */
    int   col_init[n];      /* 费用矩阵的当前列(下标)映射为原始列(内容)*/
    int   row_cur[n];       /* 费用矩阵的原始行(下标)映射为当前行(内容) */
    int   col_cur[n];       /* 费用矩阵的原始列(下标)映射为当前列(内容) */
    int   ad[n];            /* 回路顶点邻接表 */
    int   k;                /* 当前费用矩阵的阶 */
    Type  w;                /* 结点的下界 */
} NODE;
NODE      *xnode;           /* 父亲结点指针 */
NODE      *ynode;           /* 儿子结点指针 */
NODE      *znode;           /* 儿子结点指针 */
int       n_heap;           /* 堆元素个数 */
typedef struct {            /* 堆结构数据 */
    NODE  *p;               /* 指向结点元素的指针*/
    Type  w;                /* 所指向结点的下界,堆元素的关键字 */
} HEAP;
```

$ad[i]$：与顶点 i（出）相邻接的顶点（入）序号。
例：的回路由边 v_3v_0、v_1v_2、v_2v_4、v_0v_1、v_4v_3 组成，数组 ad 中的登记情况：

	0	1	2	3	4
ad	1	2	4	0	3

图 8.5　回路顶点邻接表的登记情况

算法中使用下面的几个函数：

Type row_min(NODE * node , int row, Type &second)；　计算费用矩阵行的最小值
Type col_min(NODE * node , int col, Type &second)；　计算费用矩阵列的最小值
Type array_red(NODE * node)；　归约 *node* 所指向结点的费用矩阵
Type edge_sel(NODE * node, int &vk, int &vl)；　计算 D_{kl}，选择搜索分支的边
void del_rowcol(NODE * node, int vk, int vl)；　删除费用矩阵第 *vk* 行、*vl* 列
void edge_byp(NODE * node, int vk, int vl)；　登记回路顶点邻接表，旁路有

关的边

NODE * initial（Type c[ ][ ], int n）；　初始化

1、row_min（NODE * node , int row, Type &second）函数返回*node*所指向结点的费用矩阵中第*row*行的最小值，次小值回送于引用变量sec*ond* 。

```
Type row_min(NODE *node,int row,Type &second)
{
   Type temp;
   int i;
   if (node->c[row][0]<node->c[row][1]) {
      temp = node->c[row][0];   second = node->c[row][1];
   }
   else {
      temp = node->c[row][1];   second = node->c[row][0];
   }
   for (i=2;i<node->k;i++) {
      if (node->c[row][i]<temp) {
         second = temp;   temp = node->c[row][i];
      }
      else if (node->c[row][i]<second)
         second = node->c[row][i];
   }
   return temp;
}
```

运行时间：$O(n)$ 。

工作单元个数：$\Theta(1)$ 。

2、Type col_min（NODE * node , int col, Type &second）返回*node*所指向的结点的费用矩阵中第*col*列的最小值，次小值回送于引用变量sec*ond* 。

3、Type array_red(NODE *node)归约*node*所指向的结点的费用矩阵，返回值为归约常数

```
Type array_red(NODE *node)
{
   int i,j;
   Type temp,temp1,sum = 0;
   for (i=0;i<node->k;i++) {             /* 行归约 */
      temp = row_min(node,i,temp1);      /* 行归约常数 */
      for (j=0;j<node->k;j++)
```

```
            node->c[i][j] -= temp;
        sum += temp;                          /* 行归约常数累计 */
    }
    for (j=0;j<node->k;j++) {                 /* 列归约 */
        temp = col_min(node,j,temp1);         /* 列归约常数 */
        for (i=0;i<node->k;i++)
            node->c[i][j] -= temp;
        sum += temp;                          /* 列归约常数累计 */
    }
    return sum;                               /* 返回归约常数*/
}
```

运行时间：$O(n^2)$ 时间。

工作单元个数：$\Theta(1)$ 。

4、函数 edge_sel 计算 D_{kl}，选择搜索分支的边。返回 D_{kl} 的值，出边顶点序号 vk 和入边顶点序号 vl 。

```
Type edge_sel(NODE * node,int &vk,int &vl)
{
    int i,j;
    Type temp,d = 0;
    Type *row_value = new Type[node->k];
    Type *col_value = new Type[node->k];
    for (i=0;i<node->k;i++)                   /* 每一行的次小值 */
        row_min(node,i,row_value[i]);
    for (i=0;i<node->k;i++)                   /* 每一列的次小值 */
        col_min(node,i,col_value[i]);
    for (i=0;i<node->k,i++) {                 /* 对费用矩阵所有值为0的元素*/
        for (j=0;j<node->k;j++) {             /* 计算相应的 temp 值 */
            if (node->c[i][j]==0) {
                temp = row_value[i] + col_value[j];
                if (temp>d) {                 /* 求最大的 temp 值于 d */
                    d = temp;   vk = i;   vl = j;
                }                             /* 保存相应的行、列号 */
            }
        }
    }
    delete row_value;
    delete col_value;
```

```
    return d;
}
```

运行时间：$O(n^2)$时间。

工作单元：$O(n)$。

5、函数 del_rowcol 删除费用矩阵当前第 vk 行、第 vl 列的所有元素

```
void del_rowcol(NODE *node,int vk,int vl)
{
    int i,j,vk1,vl1;
    for (i=vk;i<node->k-1;i++)          /* 元素上移 */
        for (j=0;j<vl;j++)
            node->c[i][j] = node->c[i+1][j];
    for (j=vl;j<node->k-1;j++)          /* 元素左移 */
        for (i=0;i<vk;i++)
            node->c[i][j] = node->c[i][j+1];
    for (i=vk;i<node->k-1;i++)          /* 元素上移及左移 */
        for (j=vl;j<node->k-1;j++)
            node->c[i][j] = node->c[i+1][j+1];
    vk1 = node->row_init[vk];           /* 当前行 vk 转换为原始行 vk1*/
    node->row_cur[vk1] = -1;            /* 原始行 vk1 置删除标志 */
    for (i= vk1+1;i<n;i++)      /*vk1 之后的原始行,其对应的当前行号减 1*/
        node->row_cur--;
    vl1 = node->col_init[vl];           /* 当前列 vl 转换为原始列 vl1*/
    node->col_cur[vl1] = -1;            /* 原始列 vk1 置删除标志 */
    for (i=vl1+1;i<n;i++)       /* vl1 之后的原始列,其对应的当前列号减 1*/
        node->col-cur--;
    for (i=vk;i<node->k-1;i++)     /* 修改 vk 及其后的当前行的对应原始行号*/
        node->row_init[i] = node->row_init[i+1];
    for (i=vl;i<node->k-1;i++)/* 修改 vl 及其后的当前列的对应原始列号 */
        node->col_init[i] = node->col_init[i+1];
    node->k--;                  /* 当前矩阵的阶数减 1 */
}
```

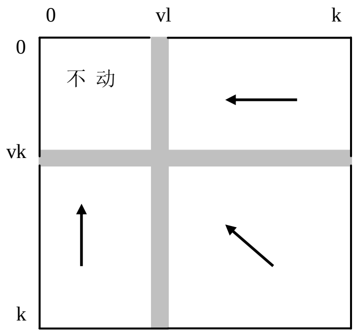


图 8.5　矩阵降阶时元素的移动过程

运行时间：$O(n^2)$ 时间。

工作单元个数：$\Theta(1)$ 。

6、函数 edge_byp 把 *vk* 行、*vl* 列所表示的边，登记到回路顶点邻接表，旁路矩阵中有关的边：

```
void edge_byp(NODE *node,int vk,int vl)
{
    int i,j,k,l;
    vk = row_init[vk];              /* 当前行号转换为原始行号 */
    vl = col_init[vl];              /* 当前列号转换为原始列号 */
    node->ad[vk] = vl;              /* 登记回路顶点邻接表*/
    for (i=0;i<n;i++) {             /* 检索顶点邻接表 */
        j = i;
        while(node->ad[j]!=-1)      /* 检查从顶点 i 开始的通路 */
            j = node->ad[j];
        if (i!=j) {                 /* 存在一条起点为 i 终点为 j 的通路 */
            l = node->row_cur[j];   /* j 转换为当前行号 l */
            k = node->col_cur[i];   /* i 转换为当前列号 k */
            if ((k>0)&&(l>0))       /* 当前行、列号均处于当前矩阵中 */
                node->c[l][k] = MAX_VALUE_OF_TYPE;
        }                           /* 相应元素置为无限大,旁路相应的边*/
    }
}
```

运行时间：$O(n^2)$ 时间。

工作单元个数：$\Theta(1)$ 。

初始化函数 NODE * initial（Type c[ ][ ], int n）叙述如下：

```
NODE *initial(Type c[][],int n)
{
    int i,j;
    NODE *node = new NODE;          /* 分配结点缓冲区 */
    for (i=0;i<n;i++)               /* 拷贝费用矩阵的初始数据 */
        for (j=0;j<n;j++)
            node->c[i][j] = c[i][j];
    for (i=0;i<n;i++) {             /* 建立费用矩阵原始行、列号与 */
        node->row_init[i] = i;      /* 初始行、列号的初始对应关系 */
        node->col_init[i] = i;
        node->row_cur[i] = i;
        node->col_cur[i] = i;
    }
    for (i=0;i<n;i++)               /* 回路顶点邻接表初始化为空 */
        node->ad[i] = -1;
    node->k = n;
    return node;                    /* 返回结点指针 */
}
```

执行时间：$O(n^2)$。

不把结点缓冲区所需存储空间包括在内，工作单元个数是$\Theta(1)$。

货郎担问题分支限界算法的实现

算法 8.1 货郎担问题的分支限界算法

输入：城市顶点的邻接矩阵 c[][],顶点个数 n

输出：最短路线费用 w 及回路的邻接顶点表 ad[]

```
template <class Type>
Type traveling_salesman(Type c[][],int n,int ad[])
{
    int i,j,vk,vl,n_heap = 0;
    Type d,w;
    NODE *xnode,*ynode,*znode;
    HEAP *heap = new HEAP[n*n];     /* 分配堆的缓冲区 */
    HEAP x,y,z;                     /* x,y,z 结点的堆元素 */
    xnode = initial(c,n);           /* 初始化父亲结点—-x 结点 */
    xnode->w = array_red(xnode);    /* 归约费用矩阵 */
    while (xnode->k!=0) {
        d = edge_sel(xnode,vk,vl);  /* 选择分支方向并计算 D_kl */
```

```
        znode = new NODE;           /* 建立分支结点—-z 结点(右儿子结点) */
        *znode = *xnode;            /* x 结点数据拷贝到 z 结点 */
        znode->c[vk][vl] = MAX_VALUE_OF_TYPE; /* 旁路 z 结点的边 */
        array_red(znode);           /* 归约 z 结点费用矩阵 */
        znode->w = xnode->w + d;    /* 计算 z 结点的下界 */
        z.w = znode->w;             /* 构造 z 结点的堆元素 */
        z.p = znode;
        insert(heap,n_heap,z);      /* z 结点插入堆中 */
        ynode = new NODE;           /* 建立分支结点—-y 结点(左儿子结点) */
        *ynode = *xnode;            /* x 结点数据拷贝到 y 结点 */
        edge_byp(ynode,vk,vl);      /* 登记回路邻接表,旁路有关的边 */
        del_rowcol(ynode,vk,vl);    /* 删除 y 结点费用矩阵当前 vk 行 vl 列*/
        ynode->w = array_red(xnode);    /* 归约 y 结点费用矩阵 */
        ynode->w += xnode->w;           /* 计算 y 结点的下界 */
        y.w = ynode->w;                 /* 构造 y 结点的堆元素 */
        y.p = ynode;
        if (ynode->k==2) {              /* 费用矩阵只剩 2 阶 */
            if (ynode->c[0][0]==0) {    /* 登记最后的两条边 */
                ynode->ad[ynode->row_init[0]] = ynode->col_init[0];
                ynode->ad[ynode->row_init[1]] = ynode->col_init[1];
            }
            else {
                ynode->ad[ynode->row_init[0]] = ynode->col_init[1];
                ynode->ad[ynode->row_init[1]] = ynode->col_init[0];
            }
            ynode->k = 0;
        }
        insert(heap,n_heap,y);          /* y 结点插入堆中 */
        delete xnode;                   /* 释放没用的 x 结点缓冲区 */
        x = delete_min(heap,n_heap);    /* 取下堆顶元素作为 x 结点*/
        xnode = x.p;
    }
    w = xnode->w                        /* 保存最短路线费用 */
    for (i=0;i<n;i++)                   /* 保存路线的顶点邻接表 */
        ad[i] = xnode->ad[i];
    delete xnode;                       /* 释放 x 结点缓冲区*/
    for (i=1;i<=n_heap;i++)             /* 释放堆的缓冲区*/
        delete heap[i].p;
    delete heap;
```

```
52.     return w;                               /* 回送最短路线费用 */
53. }
```

算法的时间花费估计如下：

第 9 行初始化父亲结点，第 10 行归约父亲结点费用矩阵，都需 $O(n^2)$ 时间。

第 11 行开始的 while 循环，在最坏情况下，循环体执行 2^n 次。

在 while 循环内部：

12 行选择分支方向，需 $O(n^2)$ 时间。

14 行把 x 结点数据拷贝到 z 结点， 16 行归约 z 结点费用矩阵，都需 $O(n^2)$ 时间。

20 行把 z 结点插入堆中，在最坏情况下，有 2^n 个结点，需 $O(\log 2^n) = O(n)$ 时间。

22 行把 x 结点数据拷贝到 y 结点，需 $O(n^2)$ 时间。

23 行登记回路邻接表，旁路有关的边， 24 行删除 y 结点费用矩阵当前 vk 行 vl 列，25 行归约 y 结点费用矩阵，这些操作都需 $O(n^2)$ 时间。

40 行把 y 结点插入堆中， 42 行删除堆顶元素，都需 $O(\log 2^n) = O(n)$ 时间。

其余花费为 $O(1)$ 时间。

整个 while 循环，在最坏情况下需 $O(n^2 2^n)$ 。

第 46 行的 for 循环保存路线的顶点邻接表于数组 ad 需 $O(n)$ 时间。

第 49 行释放堆的缓冲区，在最坏情况下，需 $O(n)$ 时间。

算法的运行时间： $O(n^2 2^n)$ 。

算法所需要的空间：

每个结点需要 $O(n^2)$ 空间存放费用矩阵，共有 2^n 个结点，需 $O(n^2 2^n)$ 空间。

0/1 背包问题

分支限界法解 0/1 背包问题的思想方法和求解过程

n 个物体重量分别为 $w_0, w_1, \cdots, w_{n-1}$ ，价值分别为 $p_0, p_1, \cdots, p_{n-1}$ ，背包载重量为 M

物体按价值重量比递减的顺序，排序后物体序号的集合为 $S = \{0, 1, \cdots, n-1\}$ 。

S_1 ：选择装入背包的物体集合，

S_2 ：不选择装入背包的物体集合，

S_3 ：尚待选择的物体集合。

$S_1(k)$ 、 $S_2(k)$ 、 $S_3(k)$ ：搜索深度为 k 时的三个集合中的物体。开始时，

$$S_1(0) = \varphi \qquad S_2(0) = \varphi \qquad S_3(0) = S = \{0, 1, \cdots, n-1\}$$

一、分支的选择及处理

s ：比值 p_i / w_i 最大的物体序号， $s \in S_3$ 。

把物体 s 装入背包的分支结点，不把物体 s 装入背包的分支结点。

s 就是集合 $S_3(k)$ 中的第一个元素。

搜索深度为 k 时，物体 s 的序号就是集合 S 中的元素 k 。

物体 s 装入背包的分支结点作如下处理：

$$S_1(k+1) = S_1(k) \cup \{k\}$$
$$S_2(k+1) = S_2(k)$$
$$S_3(k+1) = S_3(k) - \{k\}$$

不把物体 s 装入背包的分支结点则做如下处理：

$$S_1(k+1) = S_1(k)$$
$$S_2(k+1) = S_2(k) \cup \{k\}$$
$$S_3(k+1) = S_3(k) - \{k\}$$

二、上界的确定

$b(k)$：搜索深度为 k 时，分支结点的背包中物体的价值上界

$S_3(k) = \{k, k+1, \cdots, n-1\}$。若：

$$M < \sum_{i \in S_1(k)} w_i \qquad 令 \qquad b(k) = 0 \qquad （8.3.1）$$

若：

$$M = \sum_{i \in S_1(k)} w_i + \sum_{i=k}^{l-1} w_i + x \cdot w_l \qquad 0 \le x < 1, k < l, k \in S_3(k), l \in S_3(k)$$

令：

$$b(k) = \sum_{i \in S_1(k)} p_i + \sum_{i=k}^{l-1} p_i + x \cdot p_l \qquad （8.3.2）$$

三、求解步骤

1. 把物体按价值重量比递减顺序排序；
2. 建立根结点 X ，令 $X.b = 0$ ， $X.k = 0$ ， $X.S_1 = \varphi$ ， $X.S_2 = \varphi$ ， $X.S_3 = S$ ；
3. 若 $X.k = n$ ，算法结束， $X.S_1$ 即为装入背包中的物体， $X.b$ 即为装入背包中物体的最大价值；否则，转 4；
4. 建立结点 Y ， $Y.S_1 = X.S_1 \cup \{X.k\}$ ，$Y.S_2 = X.S_2$ ，$Y.S_3 = X.S_3 - \{X.k\}$ ，$Y.k = X.k + 1$；按（8.3.1）、（8.3.2）式计算 $Y.b$ ；把结点 Y 按 $Y.b$ 插入堆中；
5. 建立结点 Z ， $Z.S_1 = X.S_1$ ，$Z.S_2 = X.S_2 \cup \{X.k\}$ ，$Z.S_3 = X.S_3 - \{X.k\}$ ，$Z.k = X.k + 1$；按（8.3.1）、（8.3.2）式计算 $Z.b$；把结点 Z 插入堆中；
6. 取下堆顶元素于结点 X ，转 3；

例 8.2　有 5 个物体，重量分别为 8, 16, 21, 17, 12，价值分别为 8, 14, 16, 11, 7，背包载重量为 37，求装入背包的物体及其价值。

假定，物体序号分别为 0, 1, 2, 3, 4。最后得到的解是 $S_1 = \{1, 2\}$ ，最大价值是 30。

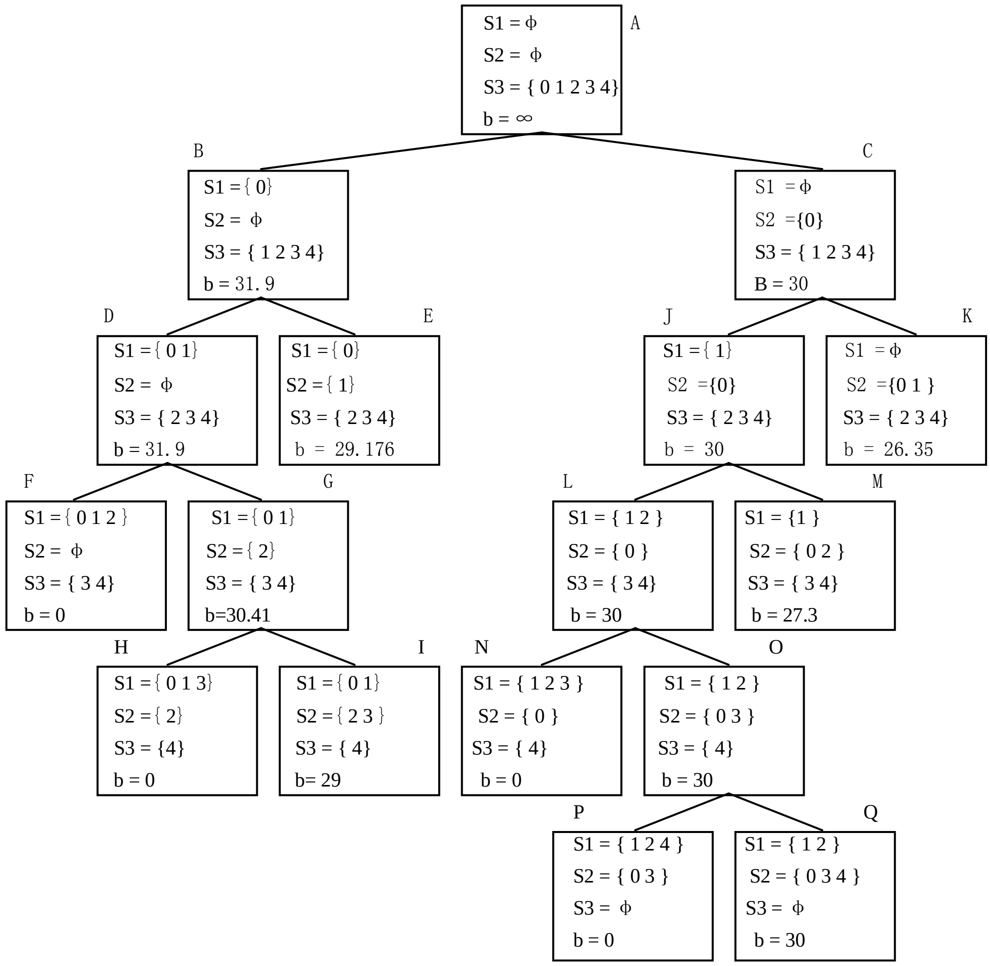


图 8.6　0/1 背包问题分支限界法的求解过程

0/1 背包问题分支限界算法的实现

数据结构：

```
typedef struct {
    float     w;          /* 物体重量 */
    float     p;          /* 物体价值 */
    float     v;          /* 物体的价值重量比 */
    int       num;        /* 物体排序前的初始序号 */
} OBJECT;
OBJECT   ob[n];
float    M;               /* 背包载重量 */
```

```
BOOL      x[n];           /* 最优装入背包的物体 */

typedef  struct {
    BOOL       s1[n];      /* 当前集合 S1 中的物体 */
    int        k;          /* 当前结点的搜索深度 */
    float      b;          /* 当前结点的价值上界 */
    float      w;          /* 当前集合 S1 中的物体重量 */
    float      p;          /* 当前集合 S1 中的物体价值 */
} KNAPNODE；

typedef  struct {
    KNAPNODE   *p;         /* 指向结点的数据 */
    float      b;          /* 所指向结点的上界,堆元素的关键字 */
} HEAP;
```

使用 bound 函数来计算分支结点的上界。bound 函数叙述如下：

```
void bound(KNAPNODE *node,float M,OBJECT ob[],int n)
{
    int i = node->k;
    float w = node->w;
    float p = node->p;
    if (node->w>M)                         /* 物体重量超过背包载重量 */
        node->b = 0;                       /* 上界置为 0 */
    else {                                 /* 否则,确定背包的剩余载重量 */
        while (w+ob[i].w<=M)&&(i<n) {      /* 以及继续装入可得到的最大价值 */
            w += ob[i].w;
            p += ob[i++].p;
        }
        if (i<n)
            node->b = p + (M – w) * ob[i].p / ob[i].w;
        else
            node->b = p;
    }
}
```

这个函数的执行时间，在最好的情况下是 $O(1)$ 时间，在最坏的情况下是 $O(n)$ 时间。这样，0/1 背包问题分支限界算法，可叙述如下：

算法 8.2 用分支限界方法实现 0/1 背包问题

输入：包含 n 个物体的重量和价值的数组 ob[],背包载重量 M

输出：最优装入背包的物体 obx[],装入背包的物体最优价值 v

```
float knapsack_bound(OBJECT ob[],float M,int n,int obx[])
{
    int i,j,k = 0;                    /* 堆中元素个数的计数器初始化为 0 */
    float v;
    KNAPNODE *xnode,*ynode,*znode;
    HEAP x,y,z,*heap;
    heap = new HEAP[n*n];             /* 分配堆的存储空间 */
    for (i=0;i<n;i++) {
        ob[i].v = ob[i].p / ob[i].w;  /* 计算物体的价值重量比 */
        ob[i].num = i;                /* 物体排序前的原始序号 */
    }
    merge_sort(ob,n);                 /* 物体按价值重量比排序 */
    xnode = new KNAPNODE;             /* 建立父亲结点 x */
    for (i=0;i<n;i++)                 /* 结点 x 初始化 */
        xnode->s1[i] = FALSE;
    xnode->k = 0;
    xnode->w = 0;
    xnode->p = 0;
    while (xnode->k<n) {
        ynode = new KNAPNODE;         /* 建立结点 y */
        *ynode = *xnode;              /* 结点 x 的数据拷贝到结点 y */
        ynode->s1[ynode->k] = TRUE;    /* 装入第 k 个物体 */
        ynode->w += ob[ynode->k].w;   /* 背包中物体重量累计 */
        ynode->p += ob[ynode->k].p;   /* 背包中物体价值累计 */
        ynode->k++;                   /* 搜索深度加 1 */
        bound(ynode,M,ob,n);          /* 计算结点 y 的上界 */
        y.b = ynode->b;
        y.p = ynode;
        insert(heap,y,k);             /* 结点 y 按上界之值插入堆中*/
        znode = new KNAPNODE;         /* 建立结点 z */
        *znode = *xnode;              /* 结点 x 的数据拷贝到结点 z */
        znode->k++;                   /* 搜索深度加 1 */
        bound(ynode,M,ob,n);          /* 计算结点 z 的上界 */
        z.b = znode->b;
        z.p = znode;
        insert(heap,z,k);             /* 结点 z 按上界之值插入堆中*/
```

```
37.         delete xnode;                   /* 释放结点 x 的缓冲区 */
38.         x = delete_max(heap,k);         /* 取下堆顶元素作为新的父亲结点*/
39.         xnode = x.p;
40.     }
41.     v = xnode->p;
42.     for (i=0;i<n;i++) {                 /* 取装入背包中物体在排序前的序号*/
43.         if (xnode->sl[i]) obx[i] = ob[i].num;
44.         else obx[i] = -1;
45.     }
46.     delete xnode;                       /* 释放 x 结点缓冲区*/
47.     for (i=1;i<=k;i++)                  /* 释放堆中结点的缓冲区*/
48.         delete heap[i].p;
49.     delete heap;                        /* 释放堆的缓冲区 */
50.     return v;                           /* 回送背包中物体的价值 */
51. }
```

算法的时间复杂性估计：

第 8~12 行中，执行排序算法需要花费 $O(n\log n)$；

第 13~18 行对父亲结点进行初始化，需 $O(n)$ 时间；

第 19~40 行的 while 循环，循环体在最坏情况下，可能执行 2^n 次；

　21 行和 31 行拷贝结点中的数据，需花费 $O(n)$ 时间；

　26 行和 33 行计算上界的工作，需花费 $O(n)$ 时间；

　29、36、38 行的堆的操作，需花费 $O(\log n)$；

　其余花费 $O(1)$ 时间。

第 19~40 行的 while 循环，在最坏情况下，需花费 $O(n2^n)$ 时间。

第 42~45 行，把在数组 *obx* 中构成解向量，需 $O(n)$ 时间；

第 46~50 行释放堆及存放结点的存储空间，在最坏情况下，需 $O(n)$ 时间。

算法需花费 $O(n2^n)$ 时间。

每一个结点需 $O(n)$ 空间，在最坏情况下，有 2^n 个结点，因此，空间复杂性也是 $O(n2^n)$。