

**课程编号：20006026**

# **算法分析与设计**

**主讲教师：刘 瑶**

**电子科技大学信息与软件工程学院**

# **第6章：分支限界法**

**(Branch and Bound Method)**

# 知识要点

---

❧ 理解分支限界法的剪枝搜索策略

❧ 掌握分支限界法的算法框架

❧ 分支限界法的典型例子

⊕ 单源最短路径问题；装载问题

⊕ 布线问题；0/1背包问题

⊕ 最大团问题；旅行商问题

⊕ 电路板排列问题；批处理作业调度问题

## **6.1 分支限界法的基本概念**

# 分支限界法

---

## ❧ 分支限界法与回溯法的类似之处

- ⊕ 基本思路：在问题的解空间树上搜索问题的解

## ❧ 分支限界法与回溯法的区别

### ⊕ 求解目标不同

- 回溯法的求解目标是找出解空间树中满足约束条件的**所有解**
- 分支限界法的求解目标则是尽快找出满足约束条件的**一个解**,  
或是在满足约束条件的解中找出在某种意义下的**最优解**
- 通常用于解决离散值的最优化问题

### ⊕ 搜索方式不同

- 回溯法以**深度优先**的方式（遍历结点）搜索解空间树
- 分支限界法以**广度优先或最小耗费优先**的方式搜索解空间树

# 分支限界法

---

## ∞ 分支限界法与回溯法的区别

### ⊕ 对扩展结点的扩展方式不同

- 分支限界法中，每一个活结点只有一次机会成为扩展结点
- 活结点一旦成为扩展结点，就一次性产生其所有儿子结点

### ⊕ 存储空间的要求不同

- 分支限界法的存储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

### ⊕ 二者区别小结

- 回溯法空间效率高；分支限界法往往更“快”
- 限界函数常基于问题的目标函数，适用于求解最优化问题

# 分支限界法

∞ 分支限界法的基本思想：以广度优先或以最小耗费（最大效益）  
优先的方式搜索问题的解空间树

- ⊕ 分支限界法中，每一个活结点只有一次机会成为扩展结点
- ⊕ 活结点一旦成为扩展结点，就一次性产生其所有儿子结点
  - 其中导致不可行解或导致非最优解的儿子结点被舍弃
  - 其余儿子结点被加入活结点表PT中
  - 含义：根据限界函数估算目标函数的可能取值
  - 选取可能使目标函数取得极值的结点优先进行搜索
- ⊕ 然后从活结点表中取下一结点成为当前扩展结点
- ⊕ 重复上述结点扩展过程
  - 直至找到所需的解或活结点表为空时为止

# 分支限界法的求解步骤

---

1. 定义解空间（对解编码）

2. 确定解空间的树结构

3. 按BFS等方式搜索

- ① 每个活结点仅有一次机会变成扩展结点
- ② 由扩展结点生成一步可达的新结点
- ③ 在新结点中，删除不可能导出最优解的结点（限界策略）
- ④ 将剩余的新结点加入活动表（队列）中
- ⑤ 从活动表中选择结点再扩展（分支策略）
- ⑥ 直至活动表为空



# 两种常见的分支限界法

## ☞ 队列式分支限界法

- ⊕ 按照队列先进先出（FIFO）原则选取下一个结点为扩展结点
- ⊕ 从活结点表中取出结点的顺序与加入结点的顺序相同
- ⊕ 因此活结点表的性质与队列相同

## ☞ 优先队列分支限界法（代价最小或效益最大）

- ⊕ 每个结点都有一个对应的耗费或收益，以此决定结点的优先级
- ⊕ 从优先队列中选取优先级最高的结点成为当前扩展结点
  - 如果查找一个具有最小耗费的解
    - 则活结点表可用小顶堆来建立
    - 下一个扩展结点就是具有最小耗费的活结点
  - 如果希望搜索一个具有最大收益的解
    - 则可用大顶堆来构造活结点表
    - 下一个扩展结点是具有最大收益的活结点

## 6.2 0-1背包问题

# 解空间树的动态搜索

- ❧ 回溯求解0/1背包问题，虽剪枝减少了搜索空间，但整个搜索按深度优先机械进行，是盲目搜索（不可预测本结点以下的结点进行的如何）。
- ❧ 回溯求解TSP也是盲目的（虽有目标函数，也只有找到一个可行解后才有意义）

# 解空间树的动态搜索

1. 分支限界法**首先**确定一个合理的限界函数，并根据限界函数确定目标函数的界[down, up];
2. **然后**按照广度优先策略遍历问题的解空间树，在某一支上，依次搜索该结点的所有孩子结点，分别估算这些孩子结点的目标函数的可能取值（对最小化问题，估算结点的down，对最大化问题，估算结点的up）。
3. 如果某孩子结点的目标函数值超出目标函数的界，则将其丢弃（从此结点生成的解不会比目前已得的更好），否则加入活动表等待处理。

# 0/1背包的分支限界法过程

## 问题描述

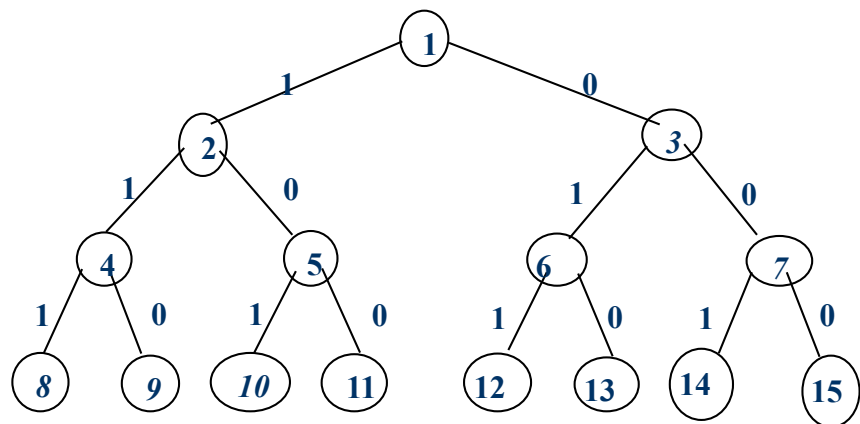
容量 $w=10$

物品	重( $w$ )	价( $v$ )	价/重( $v/w$ )
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

分析：问题的解可表示为  $n$  元向量  $\{x_1, x_2, \dots, x_n\}$ ,  $x_i \in \{0, 1\}$ , 则可用子集树表示解空间, 在树中做广度优先搜索。

约束条件:  $\sum_{i=1}^n w_i x_i \leq W$

目标函数:  $V = \max \sum_{i=1}^n v_i x_i$



• 下界  $V_{db} = 40$  (1, 0, 0, 0) — 贪心思想;

• 上界  $V_{ub} = 0 + (W - 0) \times (v_1 / w_1)$   
 $= 0 + (10 - 0) \times 10 = 100;$

目标函数的界:  
[40, 100]

限界函数:

$$ub = \underbrace{v}_{\downarrow} + \underbrace{(W - w) \times (v_{i+1} / w_{i+1})}_{\downarrow}$$

前  $i$  个物品获得的价值

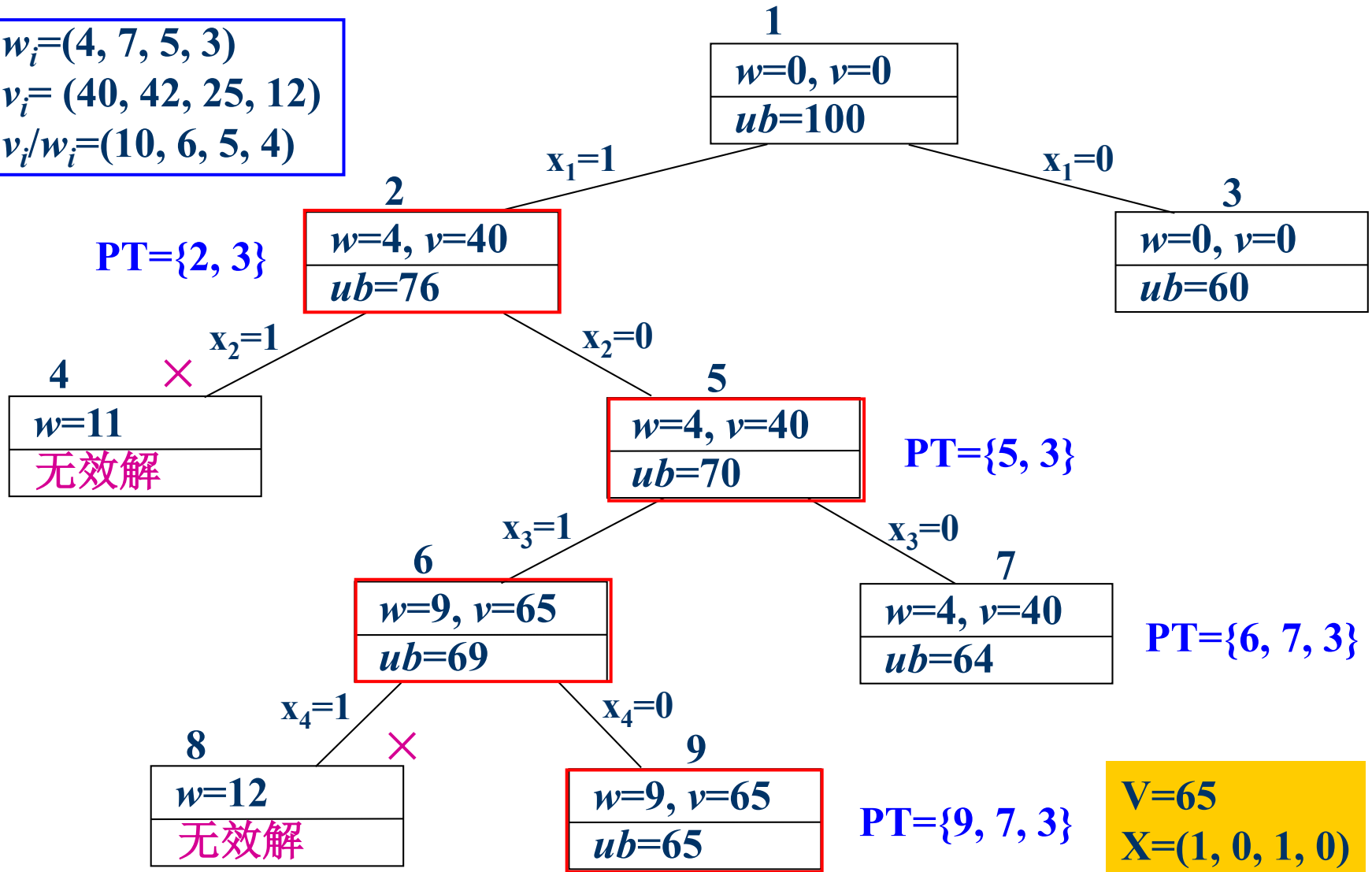
剩余容量全部装入物品  $i+1$ ,  
最多能够获得的价值

$$ub = v + (W - w) \times (v_{i+1} / w_{i+1})$$

# 分支限界法求解0/1背包问题：

目标函数范围：[40, 100]

$w_i = (4, 7, 5, 3)$   
 $v_i = (40, 42, 25, 12)$   
 $v_i / w_i = (10, 6, 5, 4)$



# 0-1背包问题

## 算法的思想

**首先，要对输入数据进行预处理，将各物品依其单位重量价值从大到小进行排列。**

**在下面描述的优先队列分支限界法中，结点的优先级由已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。**

**算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点优先队列中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶结点时为问题的最优值。**



# 0-1背包问题

## 上界函数

//  $n$ 表示物品总数,  $cleft$ 为剩余空间

while ( $i \leq n$  &&  $w[i] \leq cleft$ )

{

$cleft -= w[i];$

$b += p[i];$

$i++;$

}

if ( $i \leq n$ )  $b += p[i]/w[i] * cleft;$  // 装填剩余容量装满背包

return  $b;$

//  $w[i]$ 表示 $i$ 所占空间

//  $p[i]$ 表示 $i$ 的价值

//  $b$ 为上界函数

# 0-1背包问题

```
while (i != n+1) { // 非叶结点
    // 检查当前扩展结点的左儿子结点
    Typew wt = cw + w[i];
    if (wt <= c) { // 左儿子结点为可行结点
        if (cp+p[i] > bestp) bestp = cp+p[i];
        AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);
        up = Bound(i+1);
        // 检查当前扩展结点的右儿子结点
        if (up >= bestp) // 右子树可能含最优解
            AddLiveNode(up, cp, cw, false, i+1);

        // 取下一个扩展结点（略） }
}
```

分支限界搜索过程

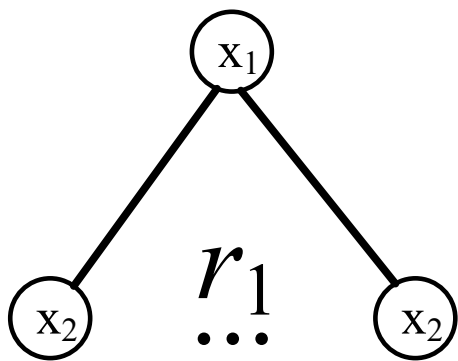
# 0/1背包的分支限界法过程

## 总结

**从0/1背包问题的搜索过程可看出：与回溯法相比，分支限界法可根据限界函数不断调整搜索方向，选择最可能得最优解的子树优先进行搜索→找到问题的解。**

# 分支限界法的设计思路

设求解**最大化**问题，解向量为 $X=(x_1, \dots, x_n)$ ， $x_i$ 的取值范围为 $S_i$ ， $|S_i|=r_i$ 。在使用分支限界搜索问题的解空间树时，先根据限界函数估算目标函数的界 $[down, up]$ ，然后从根结点出发，扩展根结点的 $r_1$ 个孩子结点，从而构成分量 $x_1$ 的 $r_1$ 种可能的取值方式。

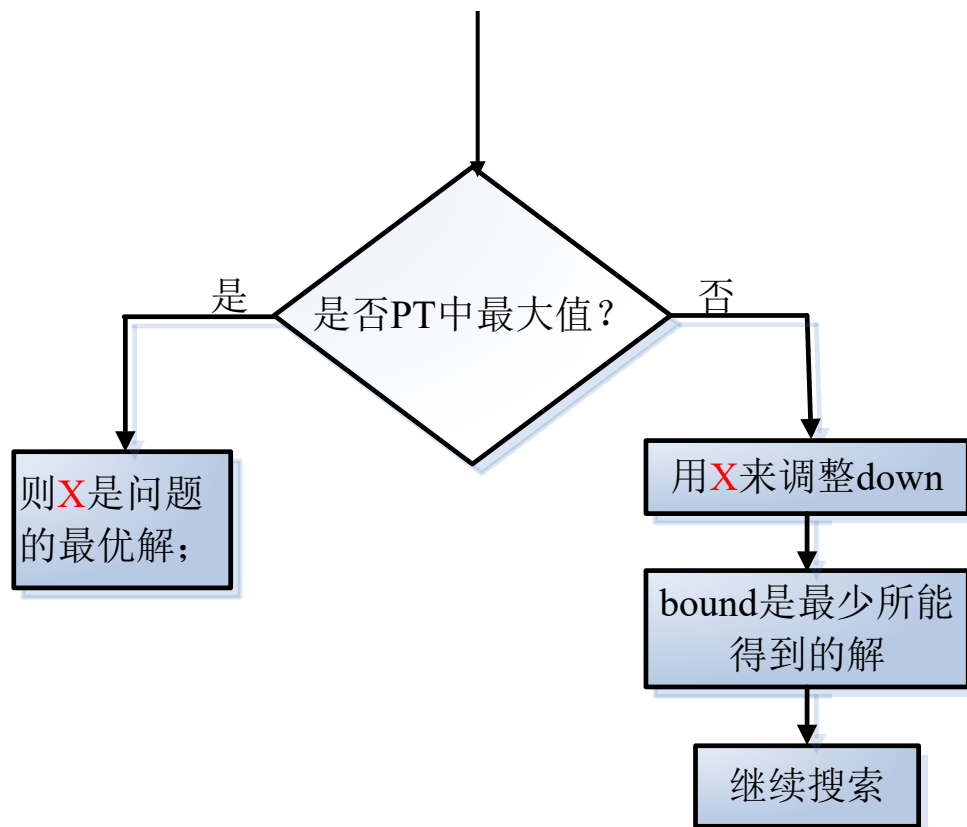


对这 $r_1$ 个孩子结点分别估算可能的目标函数 $bound(x_1)$ ，其含义：以该结点为根的子树所有可能的取值不大于 $bound(x_1)$ ，即：

$$bound(x_1) \geq bound(x_1, x_2) \geq \dots \geq bound(x_1, \dots, x_n)$$

# 分支限界法的设计思路

- ⌘ 若某孩子结点的目标函数值小于目标函数的下界，则将该孩子结点丢弃；否则，将该孩子结点保存在待处理结点表PT中。
- ⌘ 再取PT表中目标函数极大值结点作为扩展的根结点，重复上述步骤。
- ⌘ 直到一个叶子结点时的可行解  $X=(x_1, \dots, x_n)$ ，及目标函数值  $bound(x_1, \dots, x_n)$ 。



# 小结

## 分支限界法的一般过程：

1. 根据限界函数确定目标函数的界[down, up];
2. 将待处理结点表PT初始化为空;
3. 对根结点的每个孩子结点x执行下列操作
  - 3.1 估算结点x的目标函数值value;
  - 3.2 若( $value \geq down$ ), 则将结点x加入表PT中;
4. 循环直到某个叶子结点的目标函数值在表PT中最大
  - 4.1  $i$ =表PT中值最大的结点;
  - 4.2 对结点i的每个孩子结点x执行下列操作
    - 4.2.1 估算结点x的目标函数值value;
    - 4.2.2 若( $value \geq down$ ), 则将结点x加入表PT中;
    - 4.2.3 若(结点x是叶子结点且结点x的value值在表PT中最大), 则将结点x对应的解输出, 算法结束;
    - 4.2.4 若(结点x是叶子结点但结点x的value值在表PT中不是最大), 则令 $down = value$ , 并且将表PT中所有小于value的结点删除;

## 应用分支限界法的其他关键问题：

⊕ 如何确定最优解中的各个分量？

- 对每个扩展结点保存根结点到该结点的路径；

例如，0/1背包问题。将部分解 $(x_1, \dots, x_i)$ 和该部分解的目标函数的上界值都存储在待处理结点表PT中，在搜索过程中表PT的状态如下：

结点2	结点3	
(1)76	(0)60	

(a) 扩展根结点后表PT状态

结点5	结点3	
(1,0)70	(0)60	

(b) 扩展结点2后表PT状态

结点6	结点7	结点3
(1,0,1)69	(1,0,0)64	(0)60

(c) 扩展结点5后表PT状态

结点9	结点7	结点3
(1,0,1,0)65	(1,0,0)64	(0)60

(d) 扩展结点6后表PT状态，最优解为(1,0,1,0)65

## ∞ 分支限界法与回溯法的区别:

### ⊕ 求解目标不同:

- 回溯法——找出满足约束条件的所有解
- 分支限界法——找出满足条件的一个解，或某种意义下的最优解

### ⊕ 搜索方式不同:

- 回溯法——深度优先
- 分支限界法——广度优先或最小耗费优先

⊕ 此外，在分支限界法中，每一个活结点**只有一次**机会成为扩展结点。

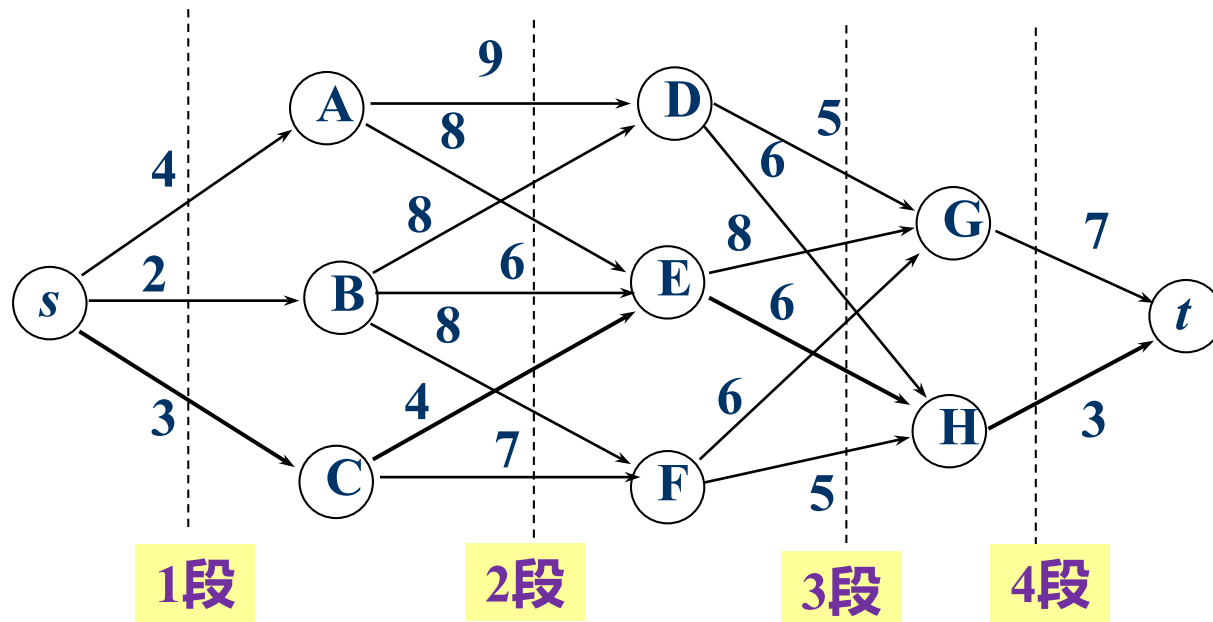


## 6.3 单源路径问题

# 单源最短路径问题

## 1. 问题描述:

- 在有向图G中，每一边都有一个非负边权。要求图G的从源顶点s到目标顶点t之间的最短路径。



分析：采用**优先队列**式分支限界法，并用一**极小堆**来存储活结点表。其优先级是结点所对应的当前路长

⊕ 解向量:  $X=(s, x_2, \dots, t)$ ,  $s$  和  $t$  分别为起点和终点

⊕ 约束条件:

- 显式:  $x_i=A, B, \dots (i=2, \dots, n)$

- 隐式:  $c_{ij} \neq \infty$

⊕ 目标函数:  $cost(i)=\min \{c_{ij}+cost(j)\}$

( $i \leq j \leq n$  且顶点  $j$  是  $i$  的邻接点)

- 下界: 把每一段最小的代价相加,  $2+4+5+3=14$

- 上界:  $2+6+6+3=17$  ( $s \rightarrow B \rightarrow E \rightarrow H \rightarrow t$ )

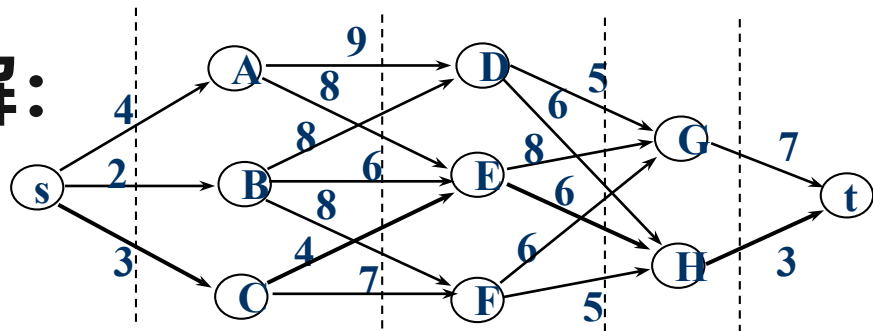
⊕ 限界函数: 假设已经确定了  $i$  段 ( $1 \leq i \leq k$ ), 其路径为  $(r_1, r_2, \dots, r_i, r_{i+1})$

$$db = \sum_{j=1}^i c[r_j][r_{j+1}] + \boxed{\min_{\langle r_{i+1}, v_p \rangle \in E} \{c[r_{i+1}][v_p]\}} + \sum_{j=i+2}^k \text{第 } j \text{ 段的最短边}$$

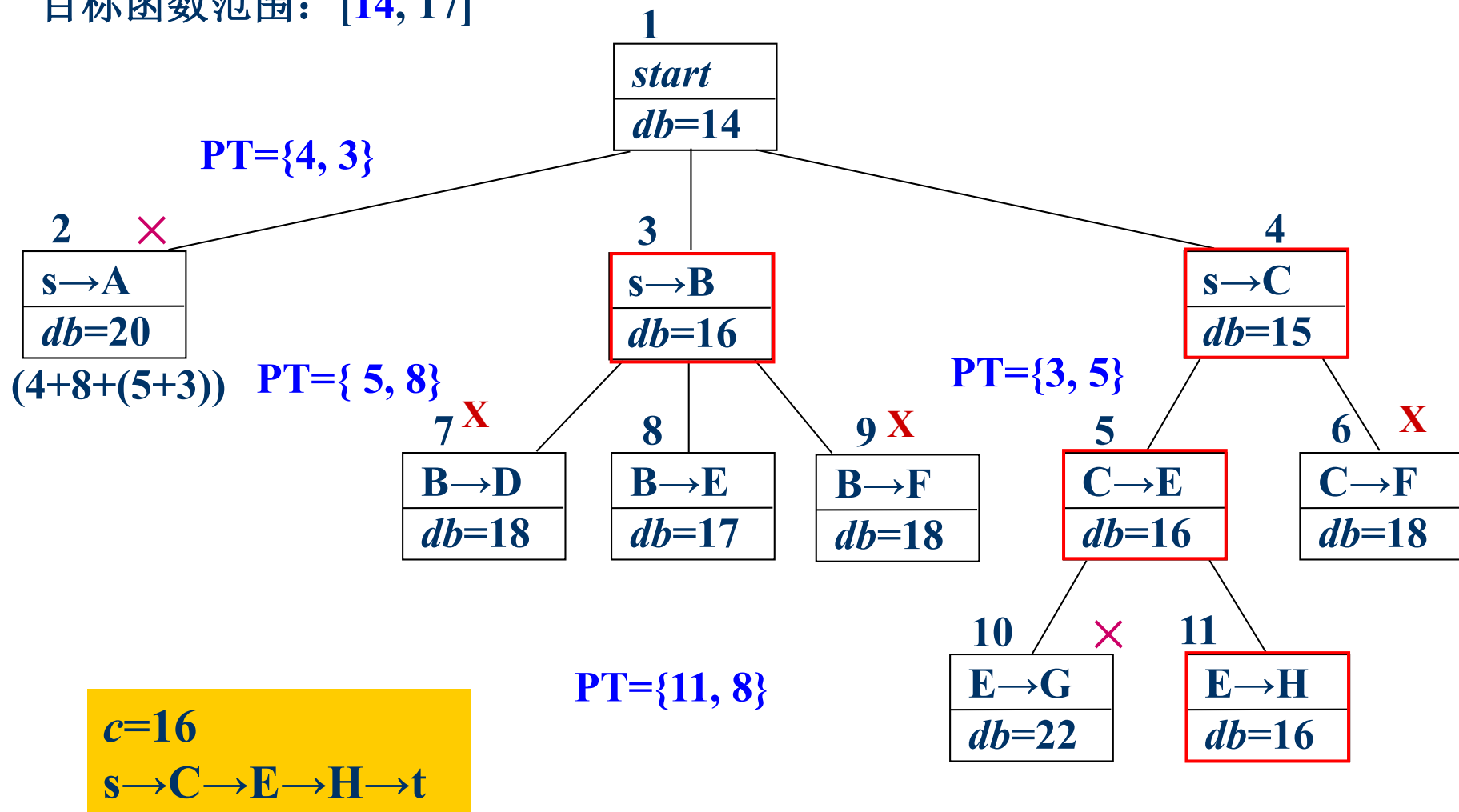
↓  
与顶点  $r_{i+1}$  相连的边中, 代价最小的边

(剩余顶点能够达到的最小代价)

# ☞ 优先队列式分支限界法求解：



目标函数范围：[14, 17]



## 搜索算法描述:

```
while (true) { //while循环体完成对解空间内部结点的扩展
```

```
    for (int j = 1; j <= n; j++)
```

```
        if ((c[E.i][j]<inf)&&(E.length+c[E.i][j]<dist[j])) {
```

```
            // 顶点i到顶点j可达, 且满足控制约束
```

```
            dist[j]=E.length+c[E.i][j]; //数组dist记录
```

```
            prev[j]=E.i; //数组prev记录从源到各顶点的
```

```
            // 加入活结点优先队列
```

```
            MinHeapNode<Type> N;
```

```
            N.i=j;
```

```
            N.length=dist[j];
```

```
            H.Insert(N);}
```

```
try {H.DeleteMin(E);} // 取下一扩展结点
```

```
catch (OutOfBounds) {break;} // 优先队列空
```

```
}
```

顶点*i*和*j*间有边, 且此  
路径长小于原先从源点  
到*j*的路径长

dist:最短距离数组  
prev: 前驱顶点数组  
E: 当前的扩展节点  
C: 邻接矩阵  
H: 活节点优先队列

## 6.4 装载问题

## 6.4 装载问题

### 1. 问题描述

有一批 $n$ 个集装箱要装上2艘载重量分别为 $C_1$ 和 $C_2$ 的轮船，其中集装箱 $i$ 的重量为 $W_i$ ，且

$$\sum_{i=1}^n w_i \leq c_1 + c_2$$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

容易证明：如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

- (1) 首先将第一艘轮船尽可能装满；
- (2) 将剩余的集装箱装上第二艘轮船。

## 分析：

⊕ 解空间：  $X=(x_1, x_2, \dots, x_n)$ ,  $x_i \in S_i = \{0, 1\}$ ,  
 $i=1, 2, \dots, n$

⊕ 约束函数：  $\sum_{i=1}^n w_i x_i \leq c_1$

⊕ 目标函数：  $\max \sum_{i=1}^n w_i x_i$

- 下界： ...

- 上界： ...

⊕ 限界函数：  $ub = Ew + \sum_{j=i+1}^n w_j$

**改进**  $\Rightarrow$   $\left\{ \begin{array}{l} \text{左孩子: } Ew + w[i] \leq c_1 \\ \text{右孩子: } Ew + \sum_{j=i+1}^n w_j > bestw \end{array} \right.$



## ⊕ 优先队列式分支限界：

采用最大优先队列存储活结点表。活结点 $x$ 在优先队列中的优先级定义为：从根结点到结点 $x$ 的路径所相应的载重量  $EW +$  剩余集装箱的重量  $r$ 。

子集树中叶结点所相应的载重量与其优先级相同，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法。

- 实现方法：(PT表结点的结构)
  - 在活结点中保存从解空间树的根结点到该活结点的路径；
  - 搜索进程中保存当前已构造出的部分解空间树；

# 装载问题

## 2. 实现

- ◆解装载问题的队列式分支限界法**仅求出所要求的最优值**，稍后将进一步构造最优解。
- ◆在while循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是，则将其加入到活结点队列Q中。
- ◆然后，将其右儿子结点加入到活结点队列中(**右儿子结点一定是可行结点**)。2个儿子结点都产生后，当前扩展结点被舍弃。
- ◆活结点队列中，队首元素被取出作为当前扩展结点。
- ◆队列中每一层结点之后，都有一个**尾部标记-1**。
- ◆在取队首元素时，活结点队列一定不空。

# 装载问题

## 2. 实现

- ◆当取出的元素是-1时，再判断当前队列是否为空。
- ◆如果队列非空，则将尾部标记-1加入活结点队列，**算法开始处理下一层的活结点。**

//子函数，将当前活节点加入队列

```
void EnQueue(Queue<Type> &Q, Type wt, Type &bestw, int i, int n)
{
    if(i == n)    //可行叶结点
    {
        if(wt > bestw) bestw = wt ;
    }
    else Q.Add(wt) ; //非叶结点，加入活结点队列
}
```

# 装载问题

## 2. 实现

```
Type MaxLoading(Type w[],Type c,int n){
    ... //初始化数据
    while (true) { //搜索解空间树
        // 检查左儿子结点
        if (Ew + w[i] <= c) // x[i] = 1 判断是否可以装上船
            EnQueue(Q, Ew + w[i], bestw, i, n); //将活结点加入到活结点队列Q中
        // 右儿子结点总是可行的, 将其加入到Q中
        EnQueue(Q, Ew, bestw, i, n); // x[i] = 0 左孩子是选择,右孩子是不选,总有其它方案.
        Q.Delete(Ew); // 取下一扩展结点
        if (Ew == -1) { // 同层结点尾部
            if (Q.IsEmpty())
                return bestw;
            Q.Add(-1); // 同层结点尾部标志
            Q.Delete(Ew); // 取下一扩展结点
            i++; // 进入下一层
        }
    }
}
```

Ew: 扩展节点的载重量  
W: 重量数组  
Q: 活节点队列  
bestw: 当前最优载重量  
i: 当前处理到的层数  
n: 总货物数

# 装载问题

## 3. 算法的改进

- 节点的左子树表示将此集装箱装船，右子树表示不将此集装箱装船。
- 设 $bestw$ 是当前最优解； $ew$ 是当前扩展结点所相应的重量； $r$ 是剩余集装箱的重量。
- 当 $ew + r \leq bestw$ 时，可将其右子树剪去。此时若要船装最多集装箱，就应该把此箱装上船。
- 算法MaxLoading初始时 $bestw = 0$ ，直到搜索到第一个叶结点才更新 $bestw$ 。在搜索到第一个叶结点前，总有 $ew + r > bestw$ ，此时右子树测试不起作用。
- 为确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新 $bestw$ 的值。

# 装载问题

## 3. 算法的改进

```
while(true) {  
    //检查左儿子  
    Type wt=Ew+w[i]; //wt为左儿子节点的重量  
    if(wt<=c) //若装载之后不超过船体可承受范围  
        if(wt>bestw) { //更新最优装载重量  
            bestw=wt;  
            if(i<n) Q.Add(wt); //将左儿子添加到队列  
        }  
    //将右儿子添加到队列  
    if(Ew+r>bestw && i<n)  
        Q.Add(Ew); //可能含有最优解  
    Q.Delete(Ew); //取下一个节点为扩展节点并将重量保存在Ew  
    if(Ew==-1) { //检查是否到了同层结束  
        if(Q.IsEmpty()) return bestw; //遍历完毕, 返回最优值  
        Q.Add(-1); //添加分层标志  
        Q.Delete(Ew); //取下一扩展结点  
        i++;  
        r-=w[i]; //剩余集装箱重量  
    }  
}
```

提前更新bestw

右儿子剪枝

# 装载问题

## 4. 构造最优解

□为了在算法结束后能方便地构造出与最优值相应的最优解，**算法必须存储相应子集树中从活结点到根结点的路径。**

□在每个结点处设置**指向其父结点的指针**，并设置左、右儿子标志。

```
class QNode
{QNode *parent;    // 指向父结点的指针
  bool LChild;      // 左儿子标志
  Type weight;      // 结点所相应的载重量
```

# 装载问题

## 4. 构造最优解

找到最优值后，可以根据parent回溯到根结点，找到最优解。

// 构造当前最优解

```
for (int j = n - 1; j > 0; j--) {  
    bestx[j] = bestE->LChild; //bestx存储最优解路径  
    bestE = bestE->parent; //回溯构造最优解  
}
```

LChild是左子树标志，1表示左子树，0表示右子树；  
bestx[j]取值为0/1，表示是否取该货物。



## 5. 例子

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。由此可知，装载问题等价于以下特殊的0-1背包问题。

例如：

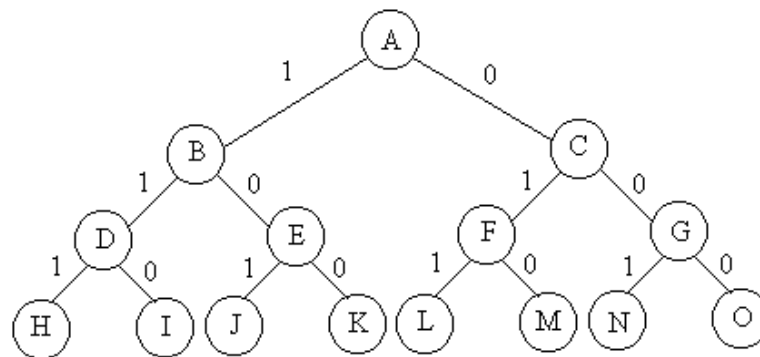
$$W = \langle 10, 30, 50 \rangle$$

$$C1 = 60$$

$$\max \sum_{i=1}^n w_i x_i$$

$$\text{s.t.} \sum_{i=1}^n w_i x_i \leq C_1$$

$$x_i \in \{0,1\}, 1 \leq i \leq n$$



# LC-搜索的过程

- 1) 初始队列中只有结点A;
- 2) 结点A变为扩展结点扩充B入堆,  $bestw=10$ ;  
结点C的装载上界 $30+50=80 > bestw$ , 入堆; 堆中B上界为90在优先队列首。
- 3) 结点B变为扩展结点扩充D入堆,  $bestw=40$ ;  
结点E的装载上界 $60 > bestw$ , 入堆; 堆中D上界为90为优先队列首。
- 4) 结点D变为扩展结点, 叶结点H超过重量, 叶结点I的装载为40,  $bestw$ 仍为40; 此时堆中C上界为80为优先队列首。
- 5) 结点C变为扩展结点扩充F入堆,  $bestw$ 仍为40;  
结点G的装载上界 $50 > bestw$ , 入堆; 此时堆中上界为60为优先队列首
- 6) 结点E变为扩展结点, 叶结点J装载量为60入堆,  $bestw$ 变为60;  
叶结点K上界 $10 < bestw$ 被剪掉; 此时堆中J上界为60为优先队列首。
- 7) 结点J变为扩展结点, 扩展的层次为4算法结束。

虽然此时堆并不空, 但可以确定已找到了最优解。

优先队列限界搜索解空间的过程是: A-B-D-C-F-E-J

# FIFO限界搜索过程

---

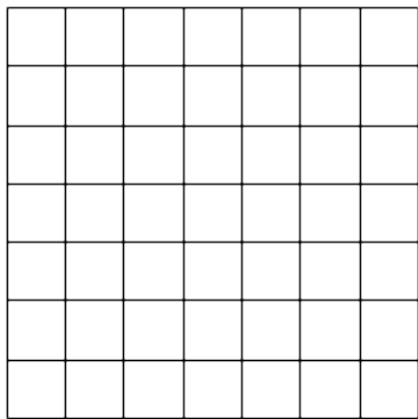
- 1) 初始队列中只有结点A;
- 2) 结点A变为扩展结点扩充B入队,  $bestw=10$ ;  
结点C的装载上界为 $30+50=80 > bestw$ , 也入队;
- 3) 结点B变为扩展结点扩充D入队,  $bestw=40$ ;  
结点E的装载上界为 $60 > bestw$ , 也入队;
- 4) 结点C变为扩展结点扩充F入队,  $bestw$ 仍为40;  
结点G的装载上界为 $50 > bestw$ , 也入队;
- 5) 结点D变为扩展结点结点, 叶结点H超过容量,  
叶结点I的装载为40,  $bestw$ 仍为40;
- 6) 结点E变为扩展结点结点, 叶结点J装载量为60,  $bestw$ 为60;  
叶结点K被剪掉;
- 7) 结点F变为扩展结点结点, 叶结点L超过容量,  $bestw$ 为60;  
叶结点M被剪掉;
- 8) 结点G变为扩展结点结点, 叶结点N、O都被剪掉;  
此时队列空算法结束。

## 6.5 布线问题

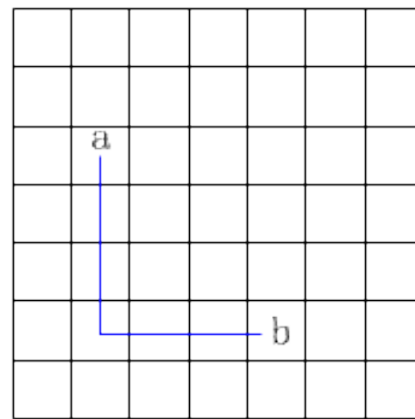
# 布线问题

## 1. 算法思想

- ◆ 印刷电路板将布线区域划分成 $n \times m$ 个方格如图 $a$ 所示。
- ◆ 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。



(a) 布线区域

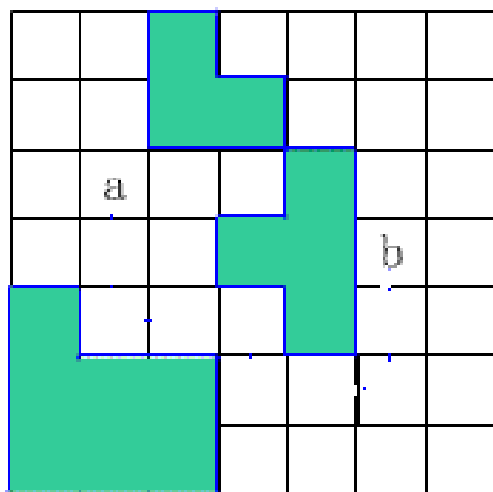


(a) 沿直线或直角布线

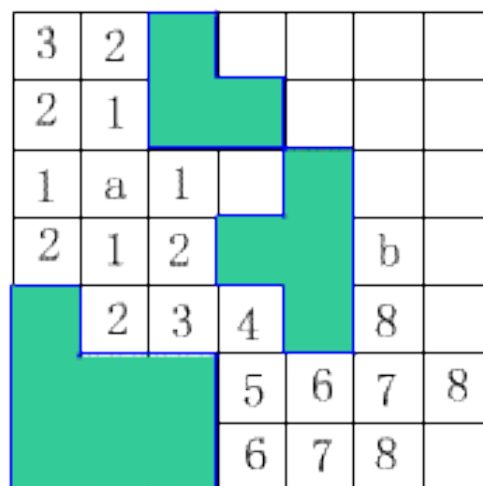
# 布线问题

## 1. 算法思想

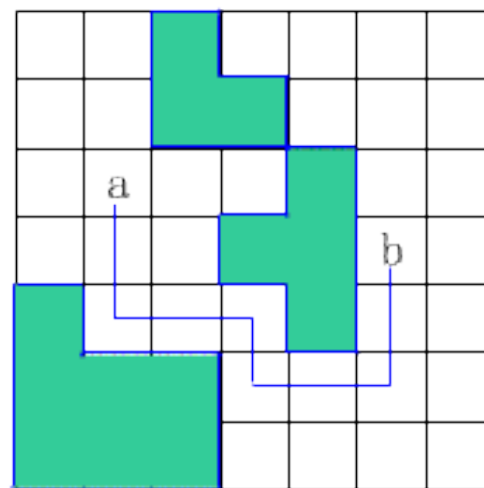
一个布线的例子：图中包含障碍。起始点为 $a$ ，目标点为 $b$ 。



原图



(a) 标记距离



(b) 最短布线路径

# 布线问题

## 1. 算法思想

- 解此问题的队列式分支限界法,从起始位置a开始,作为第一个扩展结点。
- 与该扩展结点相邻并可达的方格,成为可行结点被加入到活结点队列中,且将这些方格标记为1,即从起始方格a到这些方格的距离为1。
- 算法从活结点队列中,取出队首结点作为下一个扩展结点,将与当前扩展结点相邻且未标记过的方格标记为2,并存入活结点队列。
- 上述过程一直继续到算法搜索到目标方格b,或活结点队列为空时为止。

- 一旦方格b成为活节点, 表示找到了最优方案。
- 为什么这条路径一定就是最短的呢? 由搜索过程的特点决定。
- 假设存在一条由a至b的更短的路径, b一定会更早地被加入到活结点队列中, 并得到处理。
- 最后一个图表示了a和b之间的最短布线路径。



- 搜索中，可以知道结点距起点的路径长度，但无法直接获得具体的路径描述（最优解）。
- 为构造具体路径，需要从目标方格开始向起始方格回溯，逐步构造出最优解。
- 每次向标记距离比当前方格距离少1的相邻方格移动，直至到达起始方格时为止。

# 布线问题

Position offset[4]; ];//offset是位移矩阵

定义移动方向的  
相对位移

```
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
```

设置边界的围墙

//grid表示方格阵列，n行，m列

```
for (int i = 0; i <= m+1; i++)
```


```
    grid[0][i] = grid[n+1][i] = 1; // 顶部和底部被封锁
```

```
for (int i = 0; i <= n+1; i++)
```

```
    grid[i][0] = grid[i][m+1] = 1; // 左翼和右翼被封锁
```

# 布线问题

```
for (int i = 0; i < NumOfNbrs; i++) {// NumOfNbrs相邻方格数
    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);
    }
}
```



标记可达相邻方格

找到目标位置后，可以通过回溯方法找到这条最短路径。

## ■ 这个问题用回溯法来处理如何？

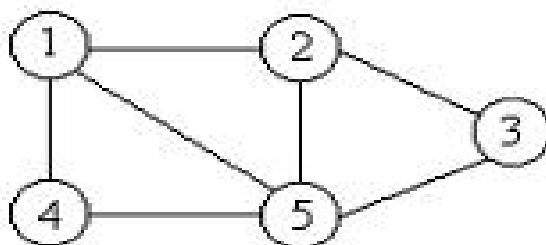
- 回溯法的搜索是依据深度优先的原则进行的。
- 如果把上下左右四个方向规定一个固定的优先顺序去进行搜索，搜索会沿着某个路径一直进行下去，直到碰壁才换到另一个子路径。
- 开始时，根本无法判断正确的路径方向，这就造成了搜索的盲目和浪费。
- 即使搜索到了一条由a至b的路径，根本无法保证它就是所有路径中最短的。
- 必须把整个区域的所有路径逐一搜索后，才能得到最优解。
- **因此，布线问题不适合用回溯法解决。**

## 6.6 最大团问题

### 1. 问题描述

- 给定无向图 $G=(V, E)$ 。如果 $U \subseteq V$ ，且对任意 $u, v \in U$ 有 $(u, v) \in E$ ，则称 $U$ 是 $G$ 的**完全子图**。
- $G$ 的完全子图 $U$ 是 $G$ 的**团**当且仅当 $U$ 不包含在 $G$ 的更大的完全子图中。 $G$ 的**最大团**是指 $G$ 中所含顶点数最多的团。
- 最大团问题的解空间树是一棵子集树
- 解最大团问题的优先队列式分支限界法, 与解装载问题的优先队列式分支限界法相似。

➤ 下图G中，子集 $\{1, 2\}$ 是G的大小为2的完全子图。这个完全子图不是团，因为它被G的更大的完全子图 $\{1, 2, 5\}$ 包含。



➤  $\{1, 2, 5\}$ 是G的最大团。  $\{1, 4, 5\}$ 和 $\{2, 3, 5\}$ 也是G的最大团。

## 2. 上界函数

- 活结点优先队列中元素类型为CliqueNode

- 对每个活结点，用 $cn$ 表示与该结点相应团的顶点数； $level$ 表示结点在子集空间树中所处的层次；用 $cn+n-level+1$ 作为顶点数上界 $un$ 的值，其中 $un$ 是该结点为根的子树中最大顶点数的上界。

- $un$ 作为优先队列中元素的优先级

- 算法总是从活结点优先队列中，抽取具有最大 $un$ 值的元素作为下一个扩展结点。

### 3. 算法思想

- ◆子集树的根结点是初始扩展结点，对于这个特殊的扩展结点，其 $cn$ 的值为0。  $cn$ 表示当前团的顶点数。
- ◆算法在扩展内部结点时，首先考察其左儿子结点。
- ◆在左儿子结点处，将顶点 $i$ 加入到当前团中，并检查该顶点与当前团中其它顶点之间是否有边相连。
- ◆当顶点 $i$ 与当前团中所有顶点之间都有边相连，则相应的左儿子结点是可行结点，将它加入到子集树中并插入活结点优先队列，否则就不是可行结点。



- ◆继续考察当前扩展结点的右儿子结点。
- ◆当 $un > bestn$ 时，右子树中可能含有最优解，此时将右儿子结点加入到子集树中，并插入到活结点优先队列中。
- ◆while循环的终止条件是：遇到子集树中的一个叶结点（即 $n+1$ 层结点）成为当前扩展结点。
- ◆对于子集树中的叶结点，有 $un = cn$ 。
- ◆此时活结点优先队列中，剩余结点的 $un$ 值均不超过当前扩展结点的 $un$ 值。
- ◆进一步搜索不可能得到更大的团，此时算法已找到一个最优解。

$un$ 是当前团最大顶点数的上界， $cn$ 表示当前团的顶点数。

## 6.7 旅行售货员问题

### 1. 问题描述

- 某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，**经过每个城市一次**，最后回到驻地的路线，使总的路程(或总旅费)最小。
- 路线是一个**带权图**。图中各边的费用（权）为正数。图的一条周游路线是**包括V中的每个顶点在内的一条回路**。周游路线的费用是这条路线上所有边的费用之和。
- 旅行售货员问题的解空间可以组织成一棵排列树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。
- 旅行售货员问题要在图G中找出费用最小的周游路线。

□与子集树的讨论相似，实现对排列树搜索的优先队列式分支限界法也可以有两种不同的实现方式：

(1) 用优先队列来存储活结点。优先队列中每个活结点都存储从根到该活结点的相应路径。

(2) 用优先队列来存储活结点，并同时存储当前已构造出的部分排列树。优先队列中的活结点不必存储从根到该活结点的相应路径，该路径必要时从存储的部分排列树中获得。

□旅行售货员问题采用第一种实现方式。

## 2. 算法描述

- ◆ 要找最小费用旅行售货员回路，选用最小堆表示活结点优先队列。
- ◆ 算法开始时创建一个最小堆，用于表示活结点优先队列。
- ◆ 堆中每个结点的子树费用的下界 $lcost$ 值，是优先队列的优先级。
- ◆ 计算每个顶点的最小费用出边并用 $minout$ 记录
- ◆ 如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法结束。
- ◆ 如果每个顶点都有出边，则根据计算出的 $minout$ 作算法初始化。

◆ **while**循环完成对排列树内部结点的扩展。对于当前扩展结点，算法分两种情况进行处理：

**第一种情况：**

➤ 考虑排列树层次 $s=n-2$ 的情形，此时当前扩展结点是排列树中某个**叶结点的父结点**。

➤ 如果该叶结点**相应一条可行回路，且费用小于当前最小费用**，则将该叶结点插入到优先队列中，否则舍去该叶结点。

## 第二种情况：

- 当排列树层次 $s < n-2$ 时，算法依次产生当前扩展结点的所有儿子结点。
- 当前扩展结点所相应的路径是 $x[0:s]$ ，其可行儿子结点是从剩余顶点 $x[s+1:n-1]$ 中选取的顶点 $x[i]$ ，且 $(x[s], x[i])$ 是所给有向图 $G$ 中的一条边。
- 对于当前扩展结点的每一个可行儿子结点，计算出其前缀 $(x[0:s], x[i])$ 的费用 $cc$ 和相应的下界 $lcost$ 。
- 当 $lcost < bestc$ 时，将这个可行儿子结点插入到活结点优先队列中。

# 算法描述

- **while循环的终止条件：排列树的一个叶结点成为当前扩展结点**
- **当 $s=n-1$ 时，已找到的回路前缀是 $x[0: n-1]$ ，它已包含图G的所有 $n$ 个顶点。**
- **因此，当 $s=n-1$ 时，相应的扩展结点表示一个叶结点。该叶结点所相应的回路的费用等于当前费用 $cc$ 和子树费用下界 $lcost$ 的值。**

# 算法描述

- 剩余活结点的lcost值，**不小于已找到的回路的费用**。它们都**不可能导致费用更小的回路**。
- 已找到叶结点所相应的回路，是一个最小费用旅行售货员回路，算法结束。
- 算法结束时，返回找到的最小费用，相应的最优解由数组v给出。

➤ 确定目标函数的界[down, up]。

（提示：如何确定上、下界？）

➤ 确定目标函数值的计算方法（限界函数）。

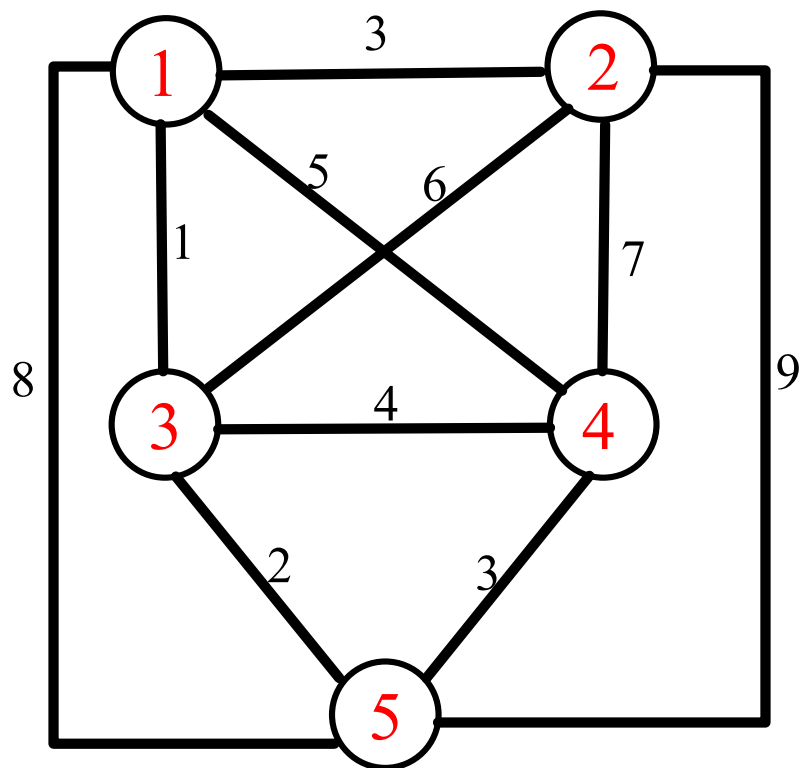
➤ 基于上、下界，按分支限界法设计思想，搜索解空间树，得到最优解。



# 旅行售货员问题

## 3. 算法举例

参见一个五结点的TSP实例。



(a) 一个无向图

$$C = \begin{bmatrix} \infty & 3 & 1 & 5 & 8 \\ 3 & \infty & 6 & 7 & 9 \\ 1 & 6 & \infty & 4 & 2 \\ 5 & 7 & 4 & \infty & 3 \\ 8 & 9 & 2 & 3 & \infty \end{bmatrix}$$

(b) 无向图的代价矩阵

## 1. 确定问题的上界16，下界10。



如何设计求上、下界策略？

## 2. 确定限界函数计算方法

一般情况下，假设当前已确定的路径为 $U=(r_1, r_2, \dots, r_k)$ ，即路径上已确定了 $k$ 个顶点，此时，该部分解的目标函数值的计算方法如下：

$$lb = \sum_{i=1}^{k-1} c[r_i][r_{i+1}] + \sum \text{剩余行最小的元素}$$

## 3. 基于分支限界法的思想，从根结点开始依次计算目标函数值加入待处理结点表中直至叶子结点。

## 算法描述:

1. 根据限界函数计算目标函数的下界**down**;  
采用贪心法得到上界**up**;
2. 计算根节点的目标函数值并加入待处理结点表**PT**;
3. 循环直到某个叶子节点的目标函数值在表**PT**中取得极小值
  - 3.1  $i$  = 表**PT**中具有最小值的结点;
  - 3.2 对结点*i*的每个孩子结点**x**执行下列操作:
    - 3.2.1 估算结点**x**的目标函数值**lb**;
    - 3.2.2 若( $lb \leq up$ ), 则将结点**x**加入表**PT**中;  
否则丢弃该结点;
4. 将叶子结点对应的最优值输出, 回溯求得最优解的各个分量;

## 6.8 批处理作业调度问题

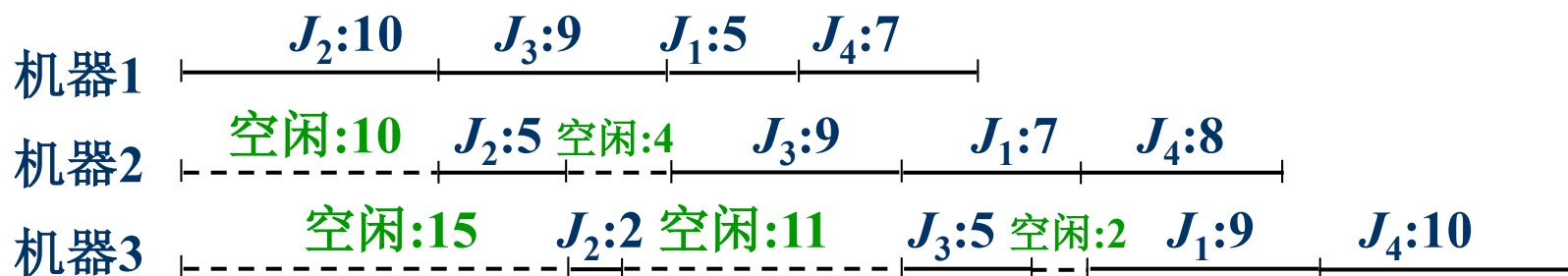
∞ 问题描述：给定  $n$  个作业的集合  $J = \{J_1, J_2, \dots, J_n\}$ ，每个作业都有3项任务分别在3台机器上完成，作业  $J_i$  需要机器  $j$  的处理时间为  $t_{ij}$  ( $1 \leq i \leq n, 1 \leq j \leq 3$ )，每个作业必须先由机器1处理，再由机器2处理，最后由机器3处理。

⊕ 批处理作业调度问题要求确定这  $n$  个作业的最优处理顺序，使得从第1个作业在机器1上处理开始，到最后一个作业在机器3上处理结束所需的时间最少。

**实例：设 $J=\{J_1, J_2, J_3, J_4\}$ 是4个待处理的作业，需要的处理时间如下所示。**

$$T = \begin{matrix} & \begin{matrix} \text{机器1} & \text{机器2} & \text{机器3} \end{matrix} \\ \begin{matrix} J_1 \\ J_2 \\ J_3 \\ J_4 \end{matrix} & \begin{pmatrix} 5 & 7 & 9 \\ 10 & 5 & 2 \\ 9 & 9 & 5 \\ 7 & 8 & 10 \end{pmatrix} \end{matrix}$$

**若处理顺序为 $(J_2, J_3, J_1, J_4)$ ，则从作业2在机器1处理开始到作业4在机器3处理完成的调度方案如下：**



( --- 表示空闲，最后完成处理时间为54)

↓  
**等待时间+处理时间**

## ∞分析:

⊕ 解向量:  $X=(x_1, x_2, \dots, x_n)$ ——排列树

⊕ 约束条件:

• 显式:  $x_i=j_1, j_2, \dots, j_n$

⊕ 目标函数:  $sum3 = \max\{sum2[n], sum3[n-1]\} + t_{n3}$     极小化

其中,  $sum2[n] = \max\{sum1[n], sum2[n-1]\} + t_{n2}$  ;

$sum1[n] = sum1[n-1] + t_{n1}$

⊕ 限界函数:    设M是已安排好的作业集合,  $M \subset \{1, 2, \dots, n\}$ ,  $|M|=k$ 。  
现在要处理作业 $k+1$ , 有:

$$db = \max\{sum1[k+1], sum2[k]\} + \sum_{i \notin M} t_{i2} + \min\left\{ t_{j3} \right\}_{j \neq k+1, j \notin M}$$

$sum1[k+1]$  =  $sum1[k] + t_{k+1,1}$

$sum2[k+1]$  =  $\max\{sum1[k+1], \underline{sum2[k]}\} + t_{k+1,2}$

- 目标函数的下界:  $sum3_{db} = \min\{t_{i1}\} + \sum_{j=1}^n t_{j2} + \min_{k \neq i}\{t_{k3}\}$

**说明：**最理想的情况下，机器1和机器2均无空闲，最后处理的作业恰好是在机器3上处理时间最短的作业。

如上实例，  $sum3_{db} = t_{11} + \sum_{j=1}^4 t_{j2} + t_{23} = 36$

⊕ 遍历并估算解空间树上各结点的目标函数的下界值：

根结点：  $sum1=0, sum2=0, M=\{\}$

$k=0$

	机器1	机器2	机器3
$J_1$	5	7	9
$J_2$	10	5	2
$J_3$	9	9	5
$J_4$	7	8	10

$T =$

# ☞ 优先队列式分支限界法求解过程：

机器1 机器2 机器3

$J_1$	5	7	9
$J_2$	10	5	2
$J_3$	9	9	5
$J_4$	7	8	10

$T=$

下界 $sum3_{db}=36$

$$db = \max \{ \text{sum1}[k], \text{sum2}[k-1] \} + \sum_{i \notin M} t_{i2} + \min \{ t_{j3} \}_{j=k, j \notin M}$$

$$\text{sum1}[k] = \text{sum1}[k-1] + t_{k,1}$$

$$\text{sum2}[k] = \max \{ \text{sum1}[k], \text{sum2}[k-1] \} + t_{k,2}$$

1  $M=\{\}$   $k=0$

start

sum1=0, sum2=0

2  $M=\{\}$   $k=1$

$J_1$ , sum1=0+5

db=36

sum2=5+7=12

PT={2, 5, 4, 3}

3  $M=\{\}$   $k=1$

$J_2$ , sum1=0+10

db=44

sum2=15

4  $M=\{\}$   $k=1$

$J_3$ , sum1=0+9

db=40

sum2=18

5  $M=\{\}$   $k=1$

$J_4$ , sum1=0+7

db=38

sum2=15

6  $M=\{1\}$   $k=2$

$J_1J_2$ , sum1=15

db=42

sum2=20

7  $M=\{1\}$   $k=2$

$J_1J_3$ , sum1=14

db=38

sum2=22

8  $M=\{1\}$   $k=2$

$J_1J_4$ , sum1=12

db=36

sum2=20

PT={8, 7, 5, 4, 6, 3}

9  $M=\{1, 4\}$   $k=3$

$J_1J_4J_2$ , sum1=22

db=41

sum2=27

10  $M=\{1, 4\}$   $k=3$

$J_1J_4J_3$ , sum1=21

db=37

sum2=30

PT={6, 7, 9, 10, 3, 4, 5}

11  $M=\{1, 4, 3\}$   $k=4$

$J_1J_4J_3J_2$ , sum1=31

db=36

sum2[4]=36

PT={6, 7, 9, 11, 3, 4, 5}

$X=(J_1, J_4, J_3, J_2)$

sum3=sum2[4]+ $t_{23}$ =38



∞ **从上例可知：优先队列式分支限界法中，**

- ⊕ 扩展结点表PT取得极值的叶子结点就对应的是问题的最优解；
- ⊕ 扩展结点的过程，一开始实际类似“深度优先”。

∞ **思考：**

- ⊕ 改成FIFO式分支限界法，搜索过程有何不同？
- ⊕ 在算法的实现上，FIFO式分支限界法和优先队列式分支限界法的PT表数据结构一样吗？

## **6.9 电路板排列问题**

## 6.9 电路板排列问题

### 问题描述

- ❧ 将 $n$ 块电路板以最佳排列方案插入带有 $n$ 个插槽的机箱中.
- ❧  $n$ 块电路板的不同的排列方式对应于不同的电路板插入方案.
- ❧ 电路板集合:  $B=\{1, 2, \dots, n\}$
- ❧ 连接块集合:  $L=\{N_1, N_2, \dots, N_m\}$  其中:  
 $N_j \subseteq B$ ,  $N_j$ 中所有电路板用一根导线连接
- ❧ 排列:  $X=\langle x_1, x_2, \dots, x_n \rangle$ , 即在机箱的第 $i$ 个插槽中插入电路板 $x_i$ .
- ❧ 排列密度 $\text{density}(X)$ :  
跨越相邻电路板插槽的最大连线数
- ❧ 求解: 具有最小排列密度 $\text{density}(X)$ 的排列.

## 6.6 电路板排列问题

➤  $n=8, m=5$ , 给定 $n$ 块电路板及其 $m$ 个连接块

➤ 电路板:

$B=\{1, 2, 3, 4, 5, 6, 7, 8\}$

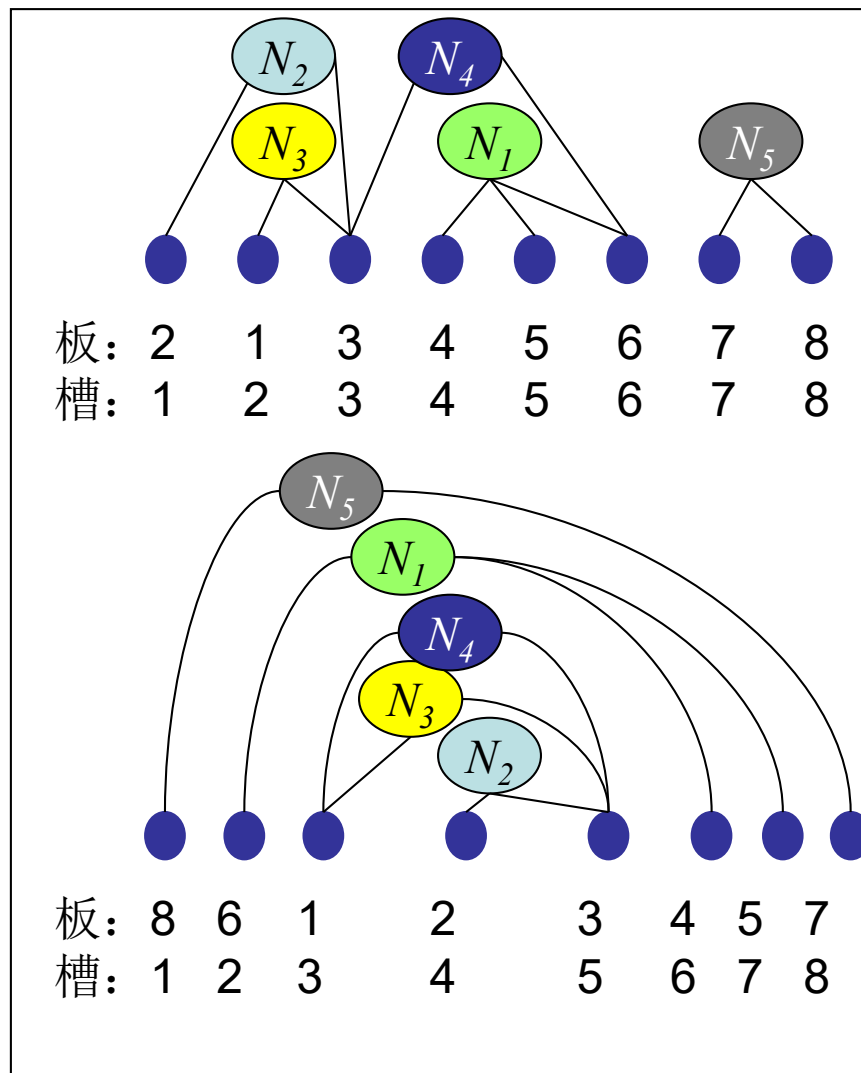
➤ 连接块:

$N_1=\{4, 5, 6\};$

$N_2=\{2, 3\}; N_3=\{1, 3\};$

$N_4=\{3, 6\}; N_5=\{7, 8\};$

其中两个可能的排列如图所示, 则该电路板排列的密度分别是2, 3。



## 算法描述

□解空间为一棵排列树，采用**优先队列式分支限界法**，找出所给电路板的最小密度布局。

□算法用一个**最小堆**表示活结点优先队列

□开始时，将排列树的根结点置为当前扩展结点。

□在do-while循环体内，算法依次从活结点优先队列中，取出具有最小**当前密度cd**值的结点，作为当前扩展结点，并加以扩展。

算法将当前扩展结点分为两种情形处理：

(1)  $s=n-1$ 的情形：此时已排定 $n-1$ 块电路板，故当前扩展结点是排列树中的一个叶结点的父结点。

➤  $x$ 表示相应于该叶结点的电路板排列。计算出与 $x$ 相应的密度，并在必要时更新当前最优值和相应的当前最优解。

(2) 当 $s < n-1$ 时：算法依次产生当前扩展结点的所有儿子结点。

➤ 对于当前扩展结点的每一个儿子结点 $N$ ，计算出其相应的密度 $N.cd$ 。

➤ 当 $N.cd < \text{当前最小密度} \text{bestd}$ 时，将该儿子结点 $N$ 插入到活结点优先队列中。反之，可将结点 $N$ 舍去。

# 算法描述

**S=n-1的情况，计算出此时的密度和bestd进行比较。**

```
do { // 结点扩展
```

```
    if (E.s == n - 1) { // 仅一个儿子结点(还有一块未排定)
```

```
        int ld = 0; // 最后一块电路板的密度
```

```
        for (int j = 1; j <= m; j++) // m为连接块数
```

```
            ld += B[E.x[n]][j]; // B[i],[j]的值为1 iff 电路板i在连接块Nj
```

中

```
        if (ld < bestd) { // 密度更小的电路板排列
```

```
            delete [ ] bestx;
```

```
            bestx = E.x;
```

```
            bestd = max(ld, E.cd);
```

```
        }
```

**S < n-1 的情况**

**else**

**{// 产生当前扩展结点的所有儿子结点**

**for (int i = E.s + 1; i <= n; i++) {**

**BoardNode N;**

**N.now = new int [m+1];**

**for (int j = 1; j <= m; j++)**

**// 新插入的电路板**

**N.now[j] = E.now[j] + B[E.x[i]][j];**



```
int ld = 0; // 新插入电路板的密度
```

```
for (int 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) {// 可能产生更好的叶结点
```

```
    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;}
```

计算出每一个儿子结点的  
密度与bestd进行比较大  
于bestd时加入队列