

# 算法设计与分析

## 分枝限界法

讲师：杨垠晖 博士

# 本讲内容

---

**6.1 分枝限界法概述**

**6.2 求解0/1背包问题**

**6.3 求解图的单源最短路径**

**6.4 求解任务分配问题**

**6.5 求解流水作业调度问题**

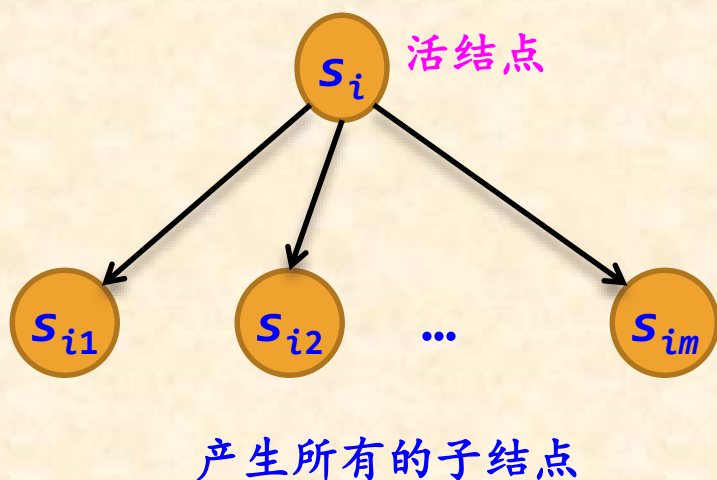
## 6.1 分枝限界法概述

### 6.1.1 什么是分枝限界法

分枝限界法类似于回溯法，也是一种在问题的解空间树上搜索问题解的算法。

但在一般情况下，分枝限界法与回溯法的求解目标不同。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，而分枝限界法的求解目标则是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使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

所谓“分枝”就是采用广度优先的策略，依次搜索活结点的  
所有分枝，也就是所有相邻结点。



求最优解时，选择哪一个子结点？

采用一个**限界函数**，计算限界函数  
数值，选择一个最有利的子结点作为  
扩展结点，使搜索朝着解空间树上有  
最优解的分枝推进，以便尽快地找出  
一个最优解。

## 分枝限界法与回溯法的主要区别

方法	解空间搜索方式	存储结点的 数据结构	结点存储特性	常用应用
回溯法	深度优先	栈	活结点的所有可行子结点被遍历后才从栈中出栈	找出满足条件的所有解
分枝限界法	广度优先	队列，优先队列	每个结点只有一次成为活结点的机会	找出满足条件一个解或者特定意义的最优解

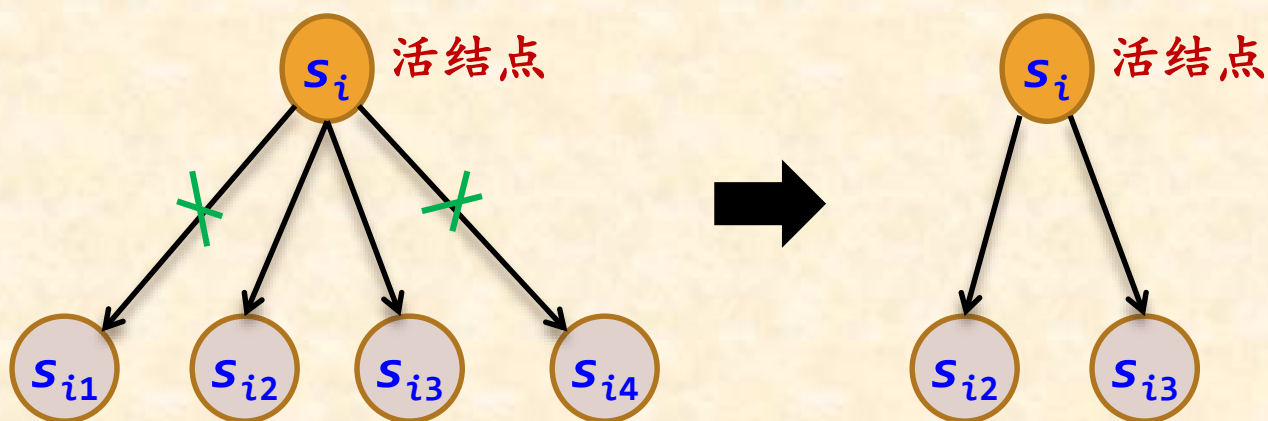
## 6.1.2 分枝限界法的设计思想

### 1. 设计合适的限界函数

在搜索解空间树时，每个活结点可能有很多孩子结点，其中有些孩子结点搜索下去是不可能产生问题解或最优解的。

可以设计好的限界函数在扩展时删除这些不必要的孩子结点，从而提高搜索效率。

假设活结点 $s_i$ 有4个孩子结点，而满足限界函数的孩子结点只有2个，可以删除这2个不满足限界函数的孩子结点，使得从 $s_i$ 出发的搜索效率提高一倍。



限界函数设计难以找出通用的方法，需根据具体问题来分析。一般地，先要确定问题解的特性：

- **目标函数是求最大值：**则设计上界限界函数ub（根结点的ub值通常大于或等于最优解的ub值），若 $s_i$ 是 $s_j$ 的双亲结点，应满足 $ub(s_i) \geq ub(s_j)$ ，当找到一个可行解 $ub(s_k)$ 后，将所有小于 $ub(s_k)$ 的结点剪枝。
- **目标函数是求最小值：**则设计下界限界函数lb（根结点的lb值一定要小于或等于最优解的lb值），若 $s_i$ 是 $s_j$ 的双亲结点，应满足 $lb(s_i) \leq lb(s_j)$ ，当找到一个可行解 $lb(s_k)$ 后，将所有大于 $lb(s_k)$ 的结点剪枝。





## 2. 组织活结点表

根据选择下一个扩展结点的方式来组织活结点表，不同的活结点表对应不同的分枝搜索方式。

- 队列式分枝限界法
- 优先队列式分枝限界法

## (1) 队列式分枝限界法

队列式分枝限界法将活结点表组织成一个队列，并按照队列先进先出（FIFO）原则选取下一个结点为扩展结点。步骤如下：

- ① 将根结点加入活结点队列。
- ② 从活结点队中取出队头结点，作为当前扩展结点。
- ③ 对当前扩展结点，先从左到右地产生它的所有孩子结点，用约束条件检查，把所有满足约束条件的孩子结点加入活结点队列。
- ④ 重复步骤②和③，直到找到一个解或活结点队列为空为止。

## (2) 优先队列式分枝限界法

优先队列式分枝限界法的主要特点是将活结点表组织成一个优先队列，并选取优先级最高的活结点成为当前扩展结点。步骤如下：

- ① 计算起始结点（根结点）的优先级并加入优先队列（与特定问题相关的信息的函数值决定优先级）。
- ② 从优先队列中取出优先级最高的结点作为当前扩展结点，使搜索朝着解空间树上可能有最优解的分枝推进，以便尽快地找出一个最优解。
- ③ 对当前扩展结点，先从左到右地产生它的所有孩子结点，然后用约束条件检查，对所有满足约束条件的孩子结点计算优先级并加入优先队列。
- ④ 重复步骤②和③，直到找到一个解或优先队列为空为止。

### 3. 确定最优解的解向量

分枝限界法在搜索解空间树时，结点的处理是跳跃式的，回溯也不是单纯地沿着双亲结点一层一层地向上回溯，因此当搜索到某个叶子结点且该结点对应一个可行解时，如何得到对应的解向量呢？

两种方法：

① 对每个扩展结点保存从根结点到该结点的路径。

每个结点带有一个可能的解向量。这种做法比较浪费空间，但实现起来简单，后面的示例均采用这种方式。

② 在搜索过程中构建搜索经过的树结构。

每个结点带有一个双亲结点指针，当找到最优解时，通过双亲指针找到对应的最优解向量。这种做法需保存搜索经过的树结构，每个结点增加一个指向双亲结点的指针。

采用分枝限界法求解的3个关键问题如下：

- (1) 如何确定合适的限界函数。
- (2) 如何组织待处理结点的活结点表。
- (3) 如何确定解向量的各个分量。

### 6.1.3 分枝限界法的时间性能

一般情况下，在问题的解向量 $X = (x_1, x_2, \dots, x_n)$ 中，分量 $x_i$  ( $1 \leq i \leq n$ ) 的取值范围为某个有限集合 $S_i = (s_{i1}, s_{i2}, \dots, s_{ir})$ 。

问题的解空间由笛卡尔积 $S_1 \times S_2 \times \dots \times S_n$ 构成：

- 第1层根结点有 $|S_1|$ 棵子树
- 第2层有 $|S_1|$ 个结点，第2层的每个结点有 $|S_2|$ 棵子树，第3层有 $|S_1| \times |S_2|$ 个结点
- ...
- 第 $n+1$ 层有 $|S_1| \times |S_2| \times \dots \times |S_n|$ 个结点，它们都是叶子结点，代表问题的所有可能解

在最坏情况下，时间复杂性是指数阶。

## 6.2 求解0/1背包问题

**【问题描述】** 有 $n$ 个重量分别为 $\{w_1, w_2, \dots, w_n\}$ 的物品，它们的价值分别为 $\{v_1, v_2, \dots, v_n\}$ ，给定一个容量为 $W$ 的背包。

设计从这些物品中选取一部分物品放入该背包的方案，每个物品要么选中要么不选中，要求选中的物品不仅能够放到背包中，而且重量和为 $W$ 具有最大的价值。



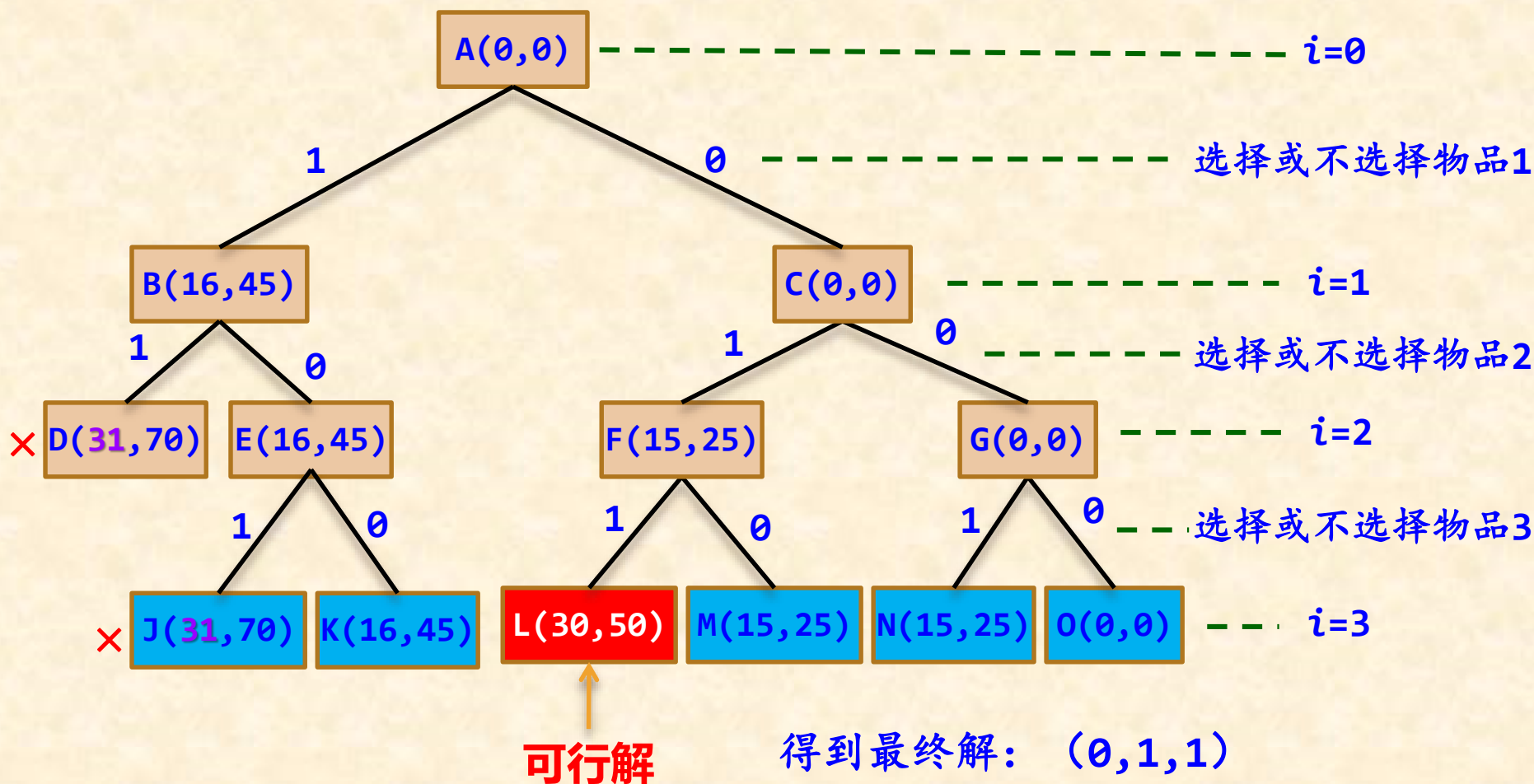
假设一个0/1背包问题是， $n=3$ ，重量为 $w=(16, 15, 15)$ ，价值为 $v=(45, 25, 25)$ ，背包限重为 $W=30$ ，解向量为 $x=(x_1, x_2, x_3)$ 。

编号	1	2	3
重量	16	15	15
价值	45	25	25

## 6.2.1 采用队列式分枝限界法求解

首先不考虑限界问题，用FIFO表示队列（实际上对应层次遍历）。初始时， $FIFO=[ ]$ 。

编号	1	2	3
重量	16	15	15
价值	45	25	25



采用STL的queue<NodeType>容器qu作为队列，队列中的结点类型声明如下：

```
struct NodeType           //队列中的结点类型
{   int no;               //结点编号，从1开始
    int i;                //当前结点在搜索空间中的层次
    int w;                //当前结点的总重量
    int v;                //当前结点的总价值
    int x[MAXN];          //当前结点包含的解向量
    double ub;            //上界
};
```

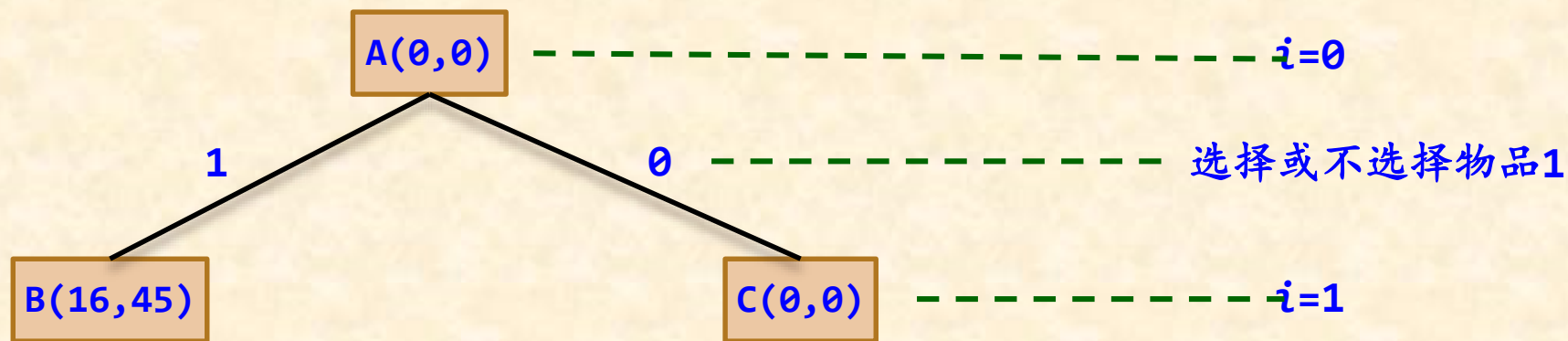
现在设计限界函数，为了简便，设根结点为第0层，然后各层依次递增，显然 $i=n$ 时表示是叶子结点层。

由于该问题是求装入背包的最大价值，属求最大值问题，采用上界设计方式。

对于第 $i$ 层的某个结点 $e$ ，用 $e.w$ 表示结点 $e$ 时已装入的总重量，用 $e.v$ 表示已装入的总价值：

- 如果所有剩余的物品都能装入背包，那么价值的上界 $e.ub = e.v + (v[i+1] + \dots + v[n])$
- 如果所有剩余的物品不能全部装入背包，那么价值的上界 $e.ub = e.v + (v[i+1] + \dots + v[k]) + (\text{物品 } k+1 \text{ 装入的部分重量}) \times \text{物品 } k+1 \text{ 的单位价值}$

编号	1	2	3
重量	16	15	15
价值	45	25	25



根结点A的层次 $i=0$ ,  $w=0$ ,  $v=0$ :

$$ub=0 + 45 + (30-16) \times 25/15 = 68 \text{ (采用取整运算)}$$



$w=0$



$w[1]=16 < 30$   
可选物品1  
 $v[1]=45$



可选物品2的一  
部分, 即 $30-16$ ,  
对应的价值

求结点 $e$ 的上界 $e.ub$ 的算法如下：

```
void bound(NodeType &e)
{  int i=e.i+1;
   int sumw=e.w;
   double sumv=e.v;
   while ((sumw+w[i]<=W) && i<=n)
   {  sumw+=w[i];
      sumv+=v[i];
      i++;
   }

   if (i<=n)
       e.ub=sumv+(W-sumw)*v[i]/w[i];
   else
       e.ub=sumv;
}
```

//计算分枝结点 $e$ 的上界  
//考虑结点 $e$ 的余下物品  
//求已装入的总重量  
//求已装入的总价值

//计算背包已装入载重  
//计算背包已装入价值

//余下物品只能部分装入

//余下物品全部可以装入

//问题表示

int n=3,W=30;

int w[]={0,16,15,15};

int v[]={0,45,25,25};

//求解结果表示

int maxv=-9999;

int bestx[MAXN];

int total=1;

struct NodeType

{ int no;

int i;

int w;

int v;

int x[MAXN];

double ub;

};

//重量,下标0不用

//价值,下标0不用

//存放最大价值,初始为最小值

//存放最优解,全局变量

//解空间中结点数累计,全局变量

//队列中的结点类型

//结点编号

//当前结点在搜索空间中的层次

//当前结点的总重量

//当前结点的总价值

//当前结点包含的解向量

//上界



```
void EnQueue(NodeType e, queue<NodeType> &qu)
```

```
// 结点e进队qu
```

```
{ if (e.i==n)
```

//到达叶子结点

```
{ if (e.v>maxv)
```

//找到更大价值的解

```
{ maxv=e.v;
```

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

```
bestx[j]=e.x[j];
```

```
}
```

```
}
```

```
else qu.push(e);
```

//非叶子结点进队

```
}
```

→ 在结点进队时判断是否为叶子结点:

- 叶子结点对应一个解
- 叶子结点不再扩展

```
void bfs()  
{  int j;  
    NodeType e,e1,e2;  
    queue<NodeType> qu;  
  
    e.i=0;  
    e.w=0; e.v=0;  
    e.no=total++;  
    for (j=1;j<=n;j++)  
        e.x[j]=0;  
    bound(e);  
    qu.push(e);
```

//求0/1背包的最优解

//定义3个结点  
//定义一个队列

//根结点置初值，其层次计为0

//求根结点的上界  
//根结点进队

```

while (!qu.empty())
{
    e=qu.front(); qu.pop();

    if (e.w+w[e.i+1]<=W)
    {
        e1.no=total++; e1.i=e.i+1;
        e1.w=e.w+w[e1.i];
        e1.v=e.v+v[e1.i];
        for (j=1;j<=n;j++)
            e1.x[j]=e.x[j];
        e1.x[e1.i]=1;
        bound(e1);
        EnQueue(e1,qu);
    }

    e2.no=total++;
    e2.i=e.i+1;
    e2.w=e.w; e2.v=e.v;
    for (j=1;j<=n;j++)
        e2.x[j]=e.x[j];
    e2.x[e2.i]=0;
    bound(e2);

    if (e2.ub>maxv)
        EnQueue(e2,qu);
}
}

```

```

//队不空循环
//出队结点e
//剪枝：检查左孩子结点
//建立左孩子结点

//复制解向量

//求左孩子结点的上界
//左孩子结点进队操作

//建立右孩子结点

//复制解向量

//求右孩子结点的上界
//若右孩子结点可行,则进队,否则被剪枝

```

结点 $e \rightarrow e1, e2$ , 剪枝

- 左孩子:  $e.w + w[e.i+1] \leq W$
- 右孩子:  $e2.ub > \max v$

## 6.2.2 采用优先队列式分枝限界法求解

采用优先队列式分枝限界法求解就是将一般的队列改为优先队列，但必须设计限界函数，因为优先级是以限界函数值为基础的。

限界函数的设计与前面的相同。这里用大根堆表示活结点表，取优先级为活结点所获得的价值。

```

struct NodeType                                //队列中的结点类型
{
    int no;                                    //结点编号
    int i;                                    //当前结点在搜索空间中的层次
    int w;                                    //当前结点的总重量
    int v;                                    //当前结点的总价值
    int x[MAXN];                              //当前结点包含的解向量
    double ub;                                //上界

    bool operator<(const NodeType &s) const //重载<关系函数
    {
        return ub<s.ub;                    //ub越大越优先出队
    }
};

```

```
void bfs()
```

```
{  int j;
```

```
    NodeType e,e1,e2;
```

```
    priority_queue<NodeType> qu;
```

```
    e.i=0;
```

```
    e.w=0; e.v=0;
```

```
    e.no=total++;
```

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

```
        e.x[j]=0;
```

```
    bound(e);
```

```
    qu.push(e);
```

```
//求0/1背包的最优解
```

```
//定义3个结点
```

```
//定义一个优先队列（大根堆）
```

```
//根结点置初值，其层次计为0
```

```
//求根结点的上界
```

```
//根结点进队
```

```

while (!qu.empty())
{
    e=qu.top(); qu.pop();
    if (e.w+w[e.i+1]<=W)
    {
        e1.no=total++;
        e1.i=e.i+1;
        e1.w=e.w+w[e1.i];
        e1.v=e.v+v[e1.i];
        for (j=1;j<=n;j++) e1.x[j]=e.x[j]; //复制解向量
        e1.x[e1.i]=1;
        bound(e1);
        EnQueue(e1,qu);
    }
    e2.no=total++;
    e2.i=e.i+1;
    e2.w=e.w; e2.v=e.v;
    for (j=1;j<=n;j++) e2.x[j]=e.x[j]; //复制解向量
    e2.x[e2.i]=0;
    bound(e2);
    if (e2.ub>maxv)
        EnQueue(e2,qu);
}
}

```

//队不空循环

//出队结点e

//剪枝：检查左孩子结点

//建立左孩子结点

//求左孩子结点的上界

//左孩子结点进队操作

//建立右孩子结点

//复制解向量

//求右孩子结点的上界

//若右孩子结点剪枝



---

**【算法分析】** 无论采用队列式分枝限界法还是优先队列式分枝限界法求解0/1背包问题，最坏情况下要搜索整个解空间树，所以最坏时间和空间复杂度均为 $O(2^n)$ ，其中 $n$ 为物品个数。

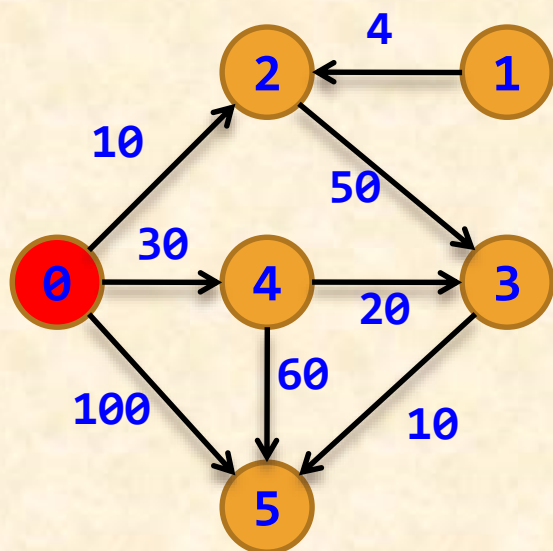
## 6.3 求解图的单源最短路径

**【问题描述】** 给定一个带权有向图 $G=(V, E)$ ，其中每条边的权是一个正整数。

另外，还给定 $V$ 中的一个顶点 $v$ ，称为源点。计算从源点到其他所有顶点的最短路径长度。这里的长度是指路上各边权之和。

### 6.3.1 采用队列式分枝限界法求解

实例图



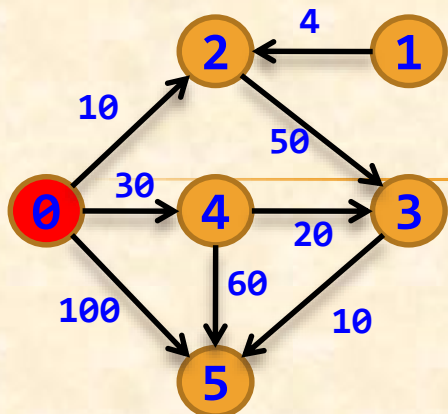
0	$\infty$	10	$\infty$	30	100
$\infty$	0	4	$\infty$	$\infty$	$\infty$
$\infty$	$\infty$	0	50	$\infty$	$\infty$
$\infty$	$\infty$	$\infty$	0	$\infty$	10
$\infty$	$\infty$	$\infty$	20	0	60
$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0

队列结点类型声明如下：

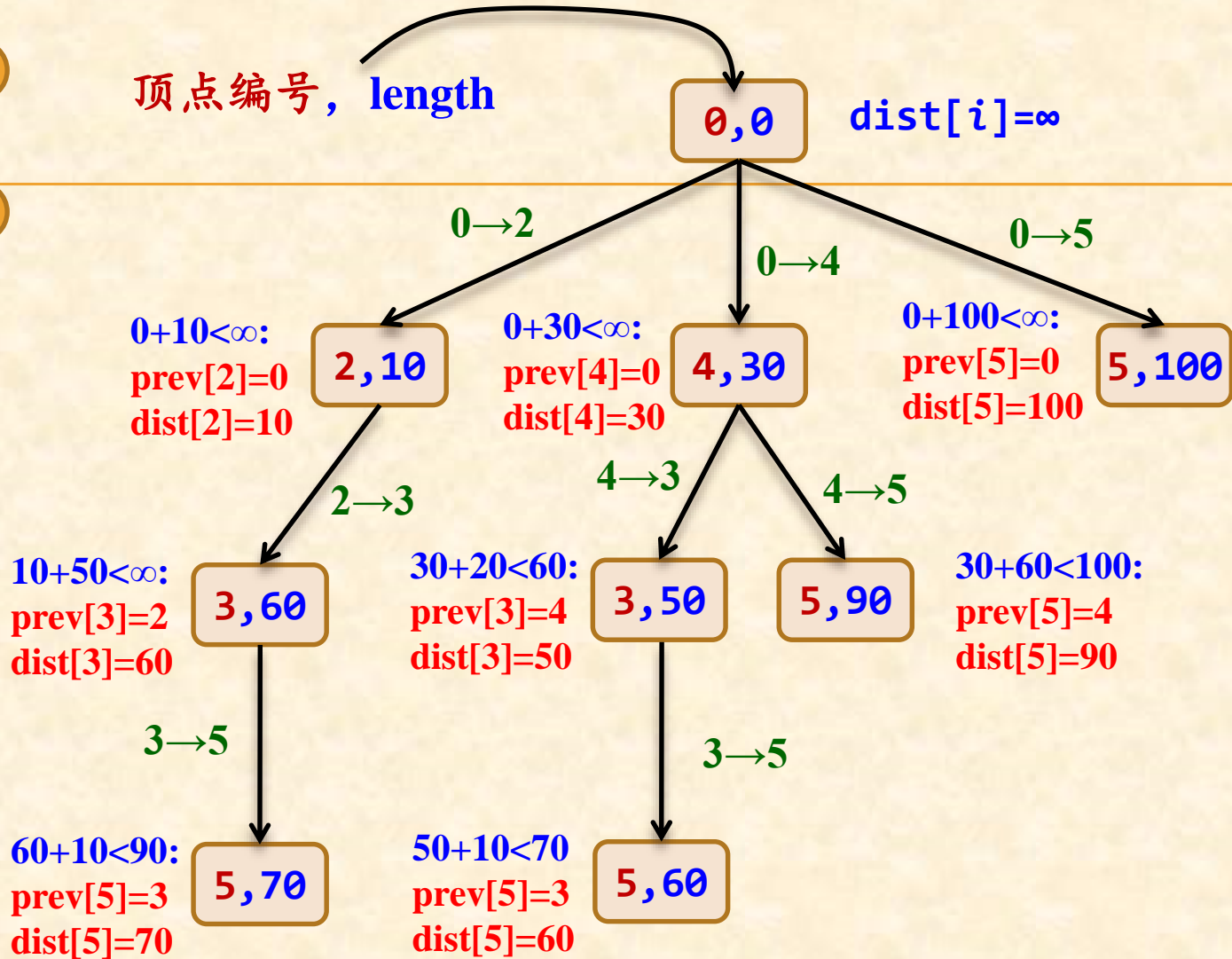
```
struct NodeType           //队列结点类型
{   int vno;              //顶点编号
    int length;           //路径长度
};
```

用dist数组存放源点v出发的最短路径长度，dist[i]表示源点v到顶点i的最短路径长度，初始时所有dist[i]值为 $\infty$ 。

用prev数组存放最短路径，prev[i]表示源点v到顶点i的最短路径中顶点i的前驱顶点。



顶点编号, length



dist[1]= $\infty$ , prev[1]=\*    dist[2]=10, prev[2]=0  
dist[3]=50, prev[3]=4    dist[4]=30, prev[4]=0  
dist[5]=60, prev[5]=3

求顶点0出发的最短路径

```
#define INF 0x3f3f3f3f //表示∞
#define MAXN 51
//问题表示
int n; //图顶点个数
int a[MAXN][MAXN]; //图的邻接矩阵
int v; //源点

//求解结果表示
int dist[MAXN]; //dist[i]源点到顶点i的最短路径长度
int prev[MAXN]; //prev[i]表示源点到j的最短路径中顶点j的前驱顶点

struct NodeType //队列结点类型
{
    int vno; //顶点编号
    int length; //路径长度
};
```

<b>void bfs(int v)</b>	<b>//求解算法</b>
<b>{ Node e,e1;</b>	
<b>queue&lt;Node&gt; pqu;</b>	
<b>e.vno=v;</b>	<b>//建立源点结点e（根结点）</b>
<b>e.length=0;</b>	
<b>pqu.push(e);</b>	<b>//源点结点e进队</b>
<b>dist[v]=0;</b>	
<b>while(!pqu.empty())</b>	<b>//队列不空循环</b>
<b>{ e=pqu.front(); pqu.pop();</b>	<b>//出队列结点e</b>
<b>for (int j=0; j&lt;n; j++)</b>	
<b>{ if(a[e.vno][j]&lt;INF &amp;&amp; e.length+a[e.vno][j]&lt;dist[j])</b>	
<b>{</b>	<b>//剪枝: e.vno到顶点j有边并且路径长度更短</b>
<b>dist[j]=e.length+a[e.vno][j];</b>	
<b>prev[j]=e.vno;</b>	
<b>e1.vno=j;</b>	<b>//建立相邻顶点j的结点e1</b>
<b>e1.length=dist[j];</b>	
<b>pqu.push(e1);</b>	<b>//结点e1进队</b>
<b>}</b>	
<b>}</b>	
<b>}</b>	
<b>}</b>	

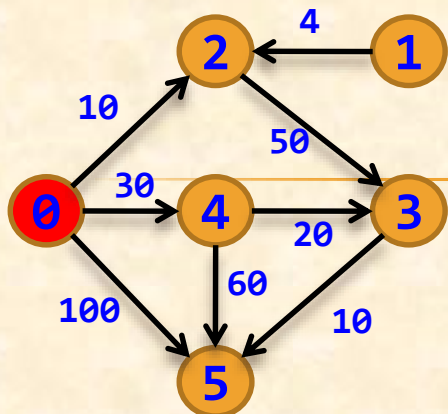
## 6.3.2 采用优先队列式分枝限界法求解

采用STL的`priority_queue<NodeType>`容器作为优先队列（小根堆），优先队列结点类型与前面的相同，添加比较重载函数，即按结点的`length`成员值越小越优先出队，为此设计`NodeType`结构体的比较重载函数如下：

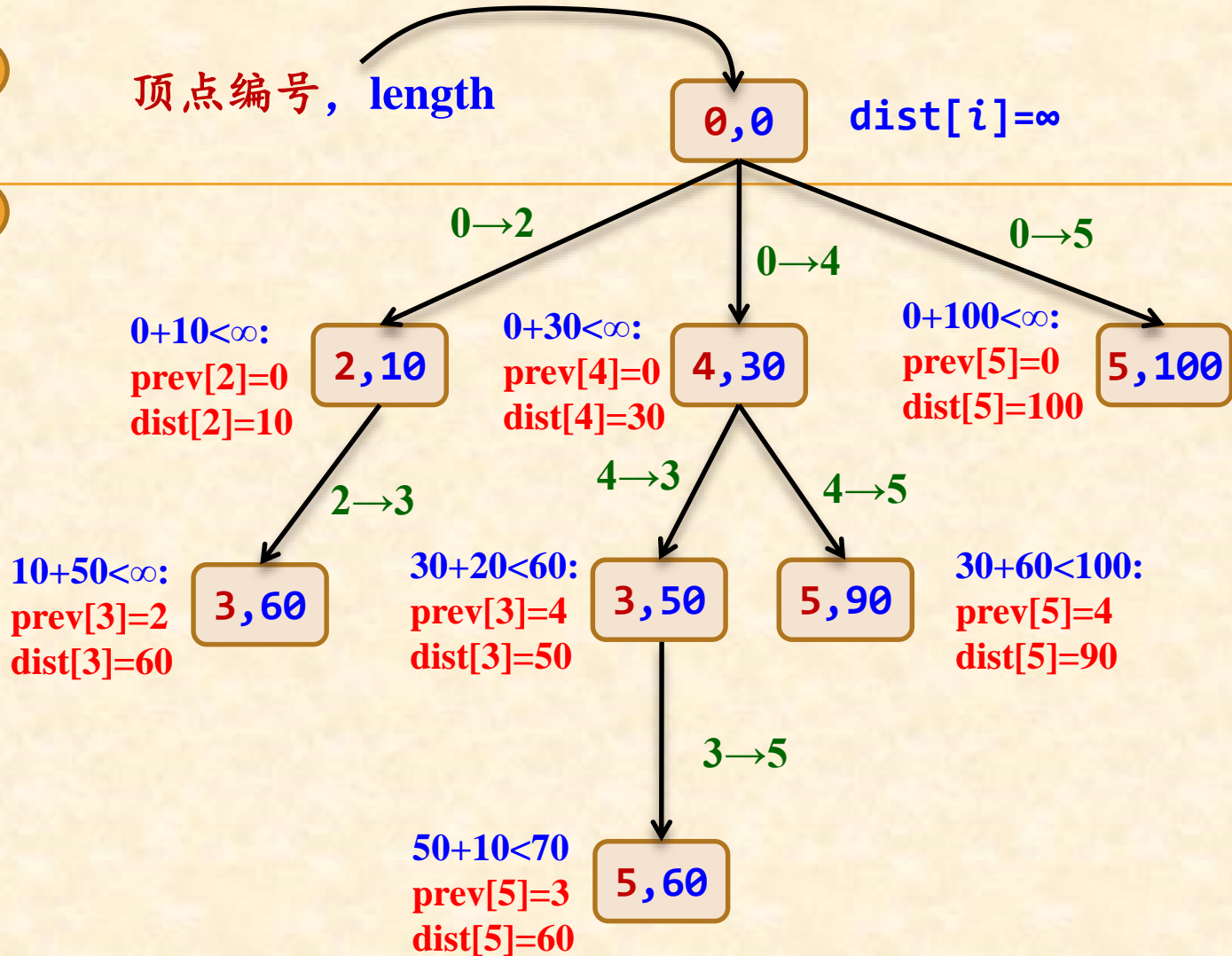
```
bool operator<(const NodeType & node) const
{
    return length>node.length;           //length越小越优先出队
}
```



<code>void bfs(int v)</code>	<code>//求解算法</code>
<code>{   NodeType e,e1;</code>	
<code>    priority_queue&lt;NodeType&gt; pqu;</code>	<code>//定义优先队列</code>
<code>    e.vno=v;</code>	<code>//建立源点结点e</code>
<code>    e.length=0;</code>	
<code>    pqu.push(e);</code>	<code>//源点结点e进队</code>
<code>    dist[v]=0;</code>	
 <code>while(!pqu.empty())</code>	<code>//队列不空循环</code>
<code>{   e=pqu.top(); pqu.pop();</code>	<code>//出队列结点e</code>
<code>    for (int j=0; j&lt;n; j++)</code>	
<code>    {   if(a[e.vno][j]&lt;INF &amp;&amp; e.length+a[e.vno][j]&lt;dist[j])</code>	
<code>        {   //剪枝: e.vno到顶点j有边并且路径长度更短</code>	
<code>            dist[j]=e.length+a[e.vno][j];</code>	
<code>            prev[j]=e.vno;</code>	
<code>            e1.vno=j;</code>	<code>//建立相邻顶点j的结点e1</code>
<code>            e1.length=dist[j];</code>	
<code>            pqu.push(e1);</code>	<code>//结点e1进队</code>
<code>        }</code>	
<code>    }</code>	
<code>}</code>	



顶点编号, length



$\text{dist}[1]=\infty, \text{prev}[1]=*$      $\text{dist}[2]=10, \text{prev}[2]=0$   
 $\text{dist}[3]=50, \text{prev}[3]=4$      $\text{dist}[4]=30, \text{prev}[4]=0$   
 $\text{dist}[5]=60, \text{prev}[5]=3$

求顶点0出发的最短路径

## 6.4 求解任务分配问题

**【问题描述】** 有 $n$  ( $n \geq 1$ ) 个任务需要分配给 $n$ 个人执行，每个任务只能分配给一个人，每个人只能执行一个任务。

第 $i$ 个人执行第 $j$ 个任务的成本是 $c[i][j]$  ( $1 \leq i, j \leq n$ )。求出总成本最小的分配方案。

4个人员、4个任务的信息

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

【问题求解】 这里采用优先队列式分枝限界法求解。

## 符号表示

- 任务和人员的编号均为 $1 \sim n$ ，解空间每一层对应一个人员的分配。
- 根结点对应人员 $0$ （虚结点），依次为人员 $1$ 、 $2$ 、...、 $n$ 分配任务。
- 叶子结点对应人员 $n$ 。
- 解向量为 $x$ ： $x[i]$ 表示人员 $i$ 分配任务编号。初始时所有元素值为 $0$ ，表示没有分配。
- 临时标识数组 $worker$ ： $worker[i]=true$ 表示任务 $i$ 已经分配。初始时所有元素值为 $false$ ，表示没有分配。
- 用 $bestx[MAXN]$ 存放最优分配方案，  $mincost$ （初始值为 $\infty$ ）存放最优成本。

## 队列结点的类型

```
struct NodeType                                //队列结点类型
{
    int no;                                    //结点编号
    int i;                                    //人员编号
    int x[MAXN];                              //x[i]为人员i分配的任务编号
    bool worker[MAXN];                        //worker[i]=true表示任务i已经分配
    int cost;                                //已经分配任务所需要的成本
    int lb;                                  //下界

    bool operator<(const NodeType &s) const //重载<关系函数
    {
        return lb>s.lb;
    }
};
```

## 下界限界函数设计

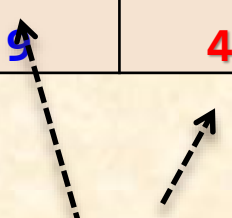
**lb**为当前结点对应分配方案的成本下界。

例如对于结点 $e$ :  $x[]=[2, 1, 0, 0]$ , 表示第1个人员分配任务2, 第2个人员分配任务1, 第3、4个人员没有分配任务;

相对应有 $worker[]=[true, true, false, false]$ , 表示任务1和2已经分配, 而任务3、4还没有分配。此时计算结果是:  $e.cost=c[1][2]+c[2][1]=2+6=8$ 。

下一步最好的情况是在数组 $c$ 中第3行和第4行中找到非第1、2列（因为任务1、2已经分配）中最小元素和, 显然为 $1+4=5$ , 即其 $e.lb=e.cost+5=13$ 。

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	5	4


$$1+4=5$$

$$\Rightarrow e.lb=e.cost+5=13$$

## 求lb的算法

$e.i=2$   
 $x=[2, 1, 0, 0]$

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

$1+4=5 \Rightarrow e.lb=e.cost+5=13$

```
void bound(NodeType &e) //求结点e的限界值
{
    int minsum=0;
    for (int i1=e.i+1;i1<=n;i1++) //求c[e.i+1..n]行中最小元素和
    {
        int minc=INF;
        for (int j1=1;j1<=n;j1++) //各列中仅仅考虑没有分配的任务
            if (e.worker[j1]==false && c[i1][j1]<minc)
                minc=c[i1][j1];
        minsum+=minc;
    }
    e.lb=e.cost+minsum;
}
```

## 剪枝操作

用  $\text{bestx}[\text{MAXN}]$  存放最优分配方案，  $\text{mincost}$ （初始值为  $\infty$ ）存放最优成本。

显然一个结点的  $\text{lb} > \text{mincost}$ ，则不可能从其子结点中找到最优解，进行剪枝。仅仅扩展  $\text{lb} \leq \text{mincost}$  的结点。



//问题表示

```
int n=4;
```

```
int c[MAXN][MAXN]={ {0}, {0,9,2,7,8}, {0,6,4,3,7},  
                    {0,5,8,1,8}, {0,7,6,9,4} };
```

//下标0的元素不用

```
int bestx[MAXN];
```

//最优分配方案

```
int mincost=INF;
```

//最小成本

```
int total=1;
```

//结点个数累计

```
void bfs() //求解任务分配
{
    int j;
    NodeType e,e1;
    priority_queue<NodeType> qu;
    memset(e.x,0,sizeof(e.x)); //初始化根结点的x
    memset(e.worker,0,sizeof(e.worker));
    //初始化根结点的worker
    e.i=0; //根结点，指定人员为0
    e.cost=0;
    bound(e); //求根结点的lb
    e.no=total++;
    qu.push(e); //根结点进队列
}
```

```
while (!qu.empty())
{
    e=qu.top(); qu.pop();           //出队结点e, 当前考虑人员e.i
    if (e.i==n)                     //达到叶子结点
    {
        if (e.cost<mincost)        //比较求最优解
        {
            mincost=e.cost;
            for (j=1;j<=n;j++)
                bestx[j]=e.x[j];
        }
    }
}
```

```

e1.i=e.i+1;           //扩展分配下一个人员的任务, 对应结点e1
for (j=1;j<=n;j++)    //考虑n个任务
{  if (e.worker[j])    //任务j是否已分配人员,若已分配, 跳过
    continue;

    for (int i1=1;i1<=n;i1++) //复制e.x得到e1.x
        e1.x[i1]=e.x[i1];
    e1.x[e1.i]=j;           //为人员e1.i分配任务j
    for (int i2=1;i2<=n;i2++) //复制e.worker得到e1.worker
        e1.worker[i2]=e.worker[i2];
    e1.worker[j]=true;      //表示任务j已经分配
    e1.cost=e.cost+c[e1.i][j];
    bound(e1);              //求e1的lb
    e1.no=total++;
    if (e1.lb<=mincost)     //剪枝
        qu.push(e1);
    }
}
}

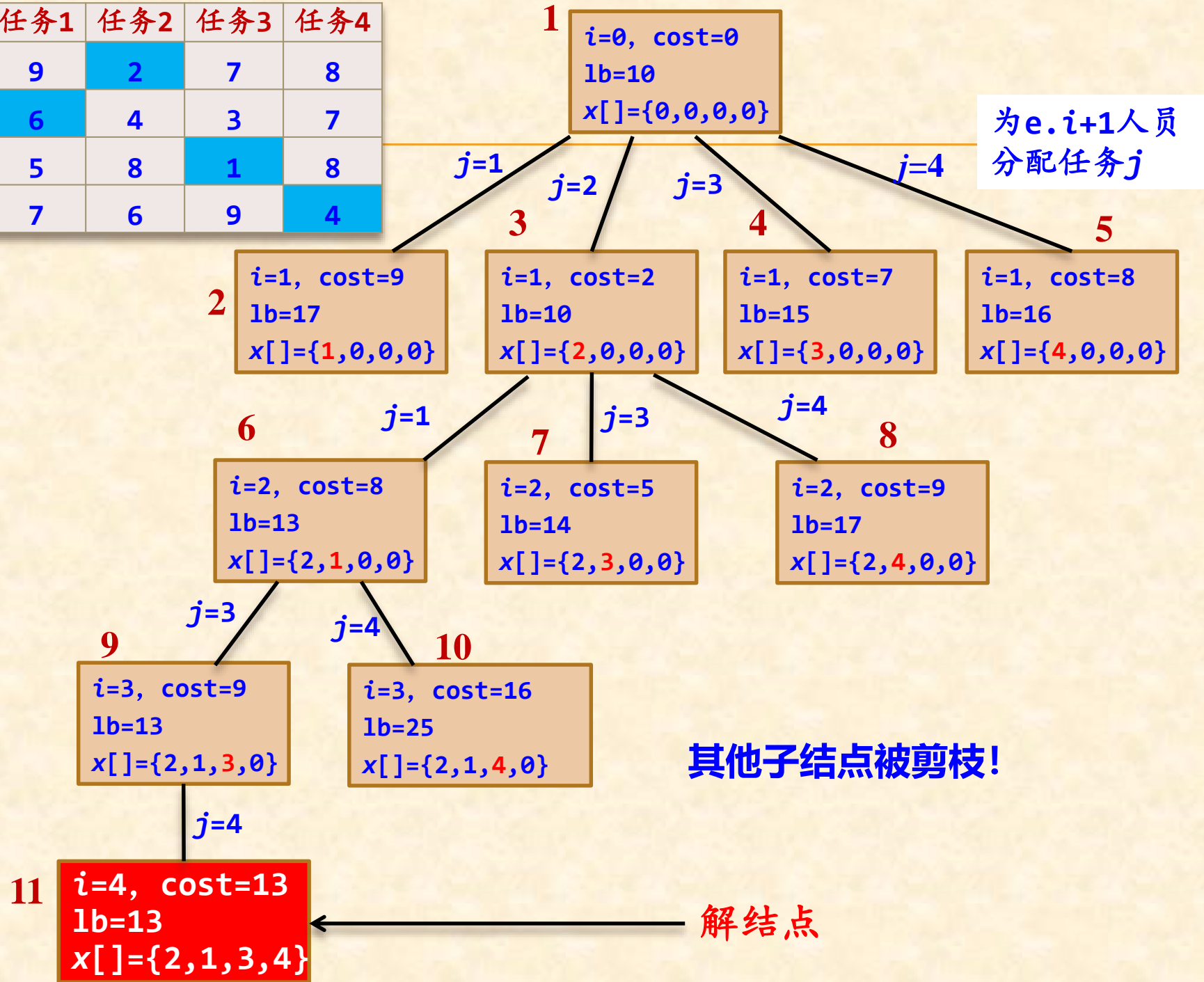
```

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

1

$i=0$ ,  $cost=0$   
 $lb=?$   
 $x[]=\{0,0,0,0\}$

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4



程序的执行结果:

人员	任务1	任务2	任务3	任务4
1	9	2	7	8
2	6	4	3	7
3	5	8	1	8
4	7	6	9	4

最优方案:

第1个人员分配第2个任务  
第2个人员分配第1个任务  
第3个人员分配第3个任务  
第4个人员分配第4个任务  
总成本=13

$x[] = \{2, 1, 3, 4\}$

## 6.5 求解流水作业调度问题

**【问题描述】** 有 $n$ 个作业（编号为 $1\sim n$ ）要在由两台机器M1和M2组成的流水线上完成加工。每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1和M2加工作业 $i$ 所需的时间分别为 $a_i$ 和 $b_i$ （ $1\leq i\leq n$ ）。

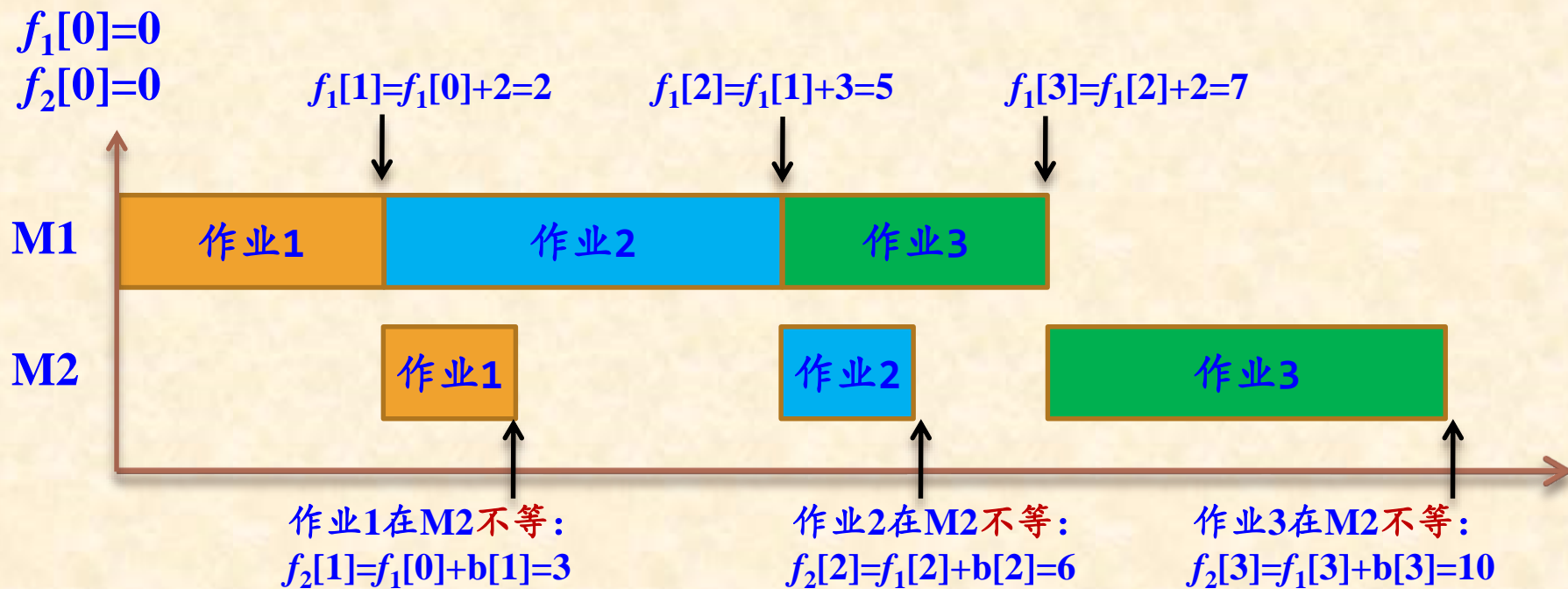
流水作业调度问题要求确定这 $n$ 个作业的最优加工顺序，使得从第一个作业在机器M1上开始加工，到最后一个作业在机器M2上加工完成所需的时间最少。可以假定任何作业一旦开始加工，就不允许被中断，直到该作业被完成，即非优先调度。



## 回顾

作业编号	1	2	3
M1时间a	2	3	2
M2时间b	1	1	3

现在的调用方案为 (1, 2, 3)，即按作业1、2、3的顺序执行。首先将 $f_1$ 和 $f_2$ 数组所有元素初始化为0。该调度方案的总时间计算：



该调度方案的  
总时间: 10

---

**【问题求解】** 作业编号为1到 $n$ ，调度方案的执行步骤为1到 $n$ ，解空间每一层对应一个步骤的作业分配。

根结点对应步骤0（虚结点），依次为步骤1、2、...、 $n$ 分配任务，叶子结点对应步骤 $n$ 。

对于按 $1 \sim n$ 顺序执行的某种调度方案,  $f_1$ 表示在 $M_1$ 上执行完当前第 $i$ 步的作业对应的总时间,  $f_2$ 数组表示在 $M_2$ 上执行完当前第 $i$ 步的作业的总时间。

若第 $i$ 步执行作业 $j$ , 计算公式如下:

$$f1=f1+a[j];$$

$$f2[i]=\max(f1, f2[i-1])+ b[j]$$

这里由于每个结点中都保存了 $f_1$ 和 $f_2$ ，因此可以将 $f_2$ 数组改为单个变量。将每个队列结点的类型声明如下：

```
struct NodeType           //队列结点类型
{
    int no;               //结点编号
    int x[MAX];           //x[i]表示第i步分配作业编号
    int y[MAX];           //y[i]=1表示编号为i的作业已经分配
    int i;                //步骤编号
    int f1;               //已经分配作业M1的执行时间
    int f2;               //已经分配作业M2的执行时间
    int lb;               //下界

    bool operator<(const NodeType &s) const //重载<关系函数
    {
        return lb>s.lb;           //lb越小越优先出队
    }
};
```

作业编号	1	2	3	4
M1时间a	5	12	4	8
M2时间b	6	2	14	7

lb为当前结点对应调度方案的时间下界。

例如，对于出队结点e，如果在第2步选择作业1 (j=1)，对应结点为e1，则：

$$e1.i = e.i + 1 = 2, \quad e1.f1 = e.f1 + a[1] = 9$$

$$e1.f2 = \max(e.f2, e1.f1) + b[1] = 18 + 6 = 24$$

$$e1.x = [3, 1, 0, 0]$$

$$e1.y = [1, 0, 1, 0]$$

结点e

i=1, f1=4  
f2=18, lb=19  
x[]={3, 0, 0, 0}  
y[]={0, 0, 1, 0}

j=1

结点e1

i=2, f1=9  
f2=24, lb=18  
x[]={3, 1, 0, 0}  
y[]={1, 0, 1, 0}

作业编号	1	2	3	4
M1时间a	5	12	4	8
M2时间b	6	2	14	7

结点e

$i=1, f1=4$   
 $f2=18, lb=19$   
 $x[]=\{3,0,0,0\}$   
 $y[]=\{0,0,1,0\}$

$j=1$

结点e1

$i=2, f1=9$   
 $f2=24, lb=18$   
 $x[]=\{3,1,0,0\}$   
 $y[]=\{1,0,1,0\}$

那么如何计算lb呢?

对于结点e1, 后面还有两步, 只能选择作业2和4, 其最少的执行时间为:

$e1.f1 + \text{作业2和4在M2上的时间和}$

这是考虑作业没有等待的情况, 所以lb定义为:

$lb = e1.f1 + \text{没有分配的作业在M2上的时间和}$

上面有  $e1.lb = e1.f1 + \text{作业2和4在M2上的时间和} = 9 + 9 = 18$ 。

对应的求结点e的lb的算法如下：

```
void bound(NodeType &e)           //求结点e的限界值
{
    int sum=0;
    for (int i=1;i<=n;i++)         //扫描所有作业
        if (e.y[i]==0) sum+=b[i];
        //仅累计e.x中还没有分配的作业的b时间
    e.lb=e.f1+sum;
}
```

## 算法设计：

- 用**bestf**（初始值为 $\infty$ ）存放最优调度时间。
- **bestx**数组存放当前作业最优调度。
- 采用的剪枝原则是，仅仅扩展 **$e.f2 < \text{bestf}$** 的结点。



//问题表示

int n=4;

int a[MAX]={0,5,12,4,8};

int b[MAX]={0,6,2,14,7};

//求解结果表示

int bestf=INF;

int bestx[MAX];

int total=1;

//作业数

//M1上的执行时间,不用下标0的元素

//M2上的执行时间,不用下标0的元素

//存放最优调度时间

//存放当前作业最佳调度

//结点个数累计

```
void bfs()
```

```
{  NodeType e,e1;
```

```
    priority_queue<NodeType> qu;
```

```
    memset(e.x,0,sizeof(e.x));
```

```
    memset(e.y,0,sizeof(e.y));
```

```
    e.i=0;
```

```
    e.f1=0;
```

```
    e.f2=0;
```

```
    bound(e);
```

```
    e.no=total++;
```

```
    qu.push(e);
```

```
//求解流水作业调度问题
```

```
//定义优先队列
```

```
//初始化根结点的x
```

```
//初始化根结点的y
```

```
//根结点
```

```
//根结点进队列
```

```
while (!qu.empty())
{
    e=qu.top(); qu.pop();           //出队结点e
    if (e.i==n)                     //达到叶子结点
    {
        if (e.f2<bestf)             //比较求最优解
        {
            bestf=e.f2;
            for (int j1=1;j1<=n;j1++)
                bestx[j1]=e.x[j1];
        }
    }
}
```

`e1.i=e.i+1;`      `//扩展分配下一个步骤的作业, 对应结点e1`

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

`//考虑所有的n个作业`

`{ if (e.y[j]==1) continue;`

`//作业j是否已分配,若已分配, 跳过`

`for (int i1=1;i1<=n;i1++)`

`//复制e.x得到e1.x`

`e1.x[i1]=e.x[i1];`

`for (int i2=1;i2<=n;i2++)`

`//复制e.y得到e1.y`

`e1.y[i2]=e.y[i2];`

`e1.x[e1.i]=j;`

`//为第i步分配作业j`

`e1.y[j]=1;`

`//表示作业j已经分配`

`e1.f1=e.f1+a[j];`

`//求f1=f1+a[j]`

`e1.f2=max(e.f2,e1.f1)+b[j];` `//求f[i+1]=max(f2[i],f1)+b[j]`

`bound(e1);`

`if (e1.f2<=bestf)`

`//剪枝,剪去不可能得到更优解的结点`

`{ e1.no=total++;`

`//结点编号增加1`

`qu.push(e1);`

`}`

`}`

`}`

`}`

```
void main()
```

```
{  bfs();
```

```
    printf("最优方案:\n");
```

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

```
        printf("    第%d步执行作业%d\n",k,bestx[k]);
```

```
    printf("    总时间=%d\n",bestf);
```

```
}
```

作业编号	1	2	3	4
M1时间a	5	12	4	8
M2时间b	6	2	14	7



最优方案：

第1步执行作业3

第2步执行作业1

第3步执行作业4

第4步执行作业2

总时间=33

