第六章 分支限界法

- 学习要点
  - 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
    1. 队列式(FIFO)分支限界法
    2. 优先队列式分支限界法
  - 通过应用范例学习分支限界法的设计策略。
    1. 单源最短路径问题
    2. 装载问题；
    3. 布线问题
    4. 0-1背包问题；
    5. 最大团问题；
    6. 旅行售货员问题
    7. 电路板排列问题
    8. 批处理作业调度问题

第六章 分支限界法

搜索法

- 在动态产生问题的解空间，并搜索问题的可行解或最优解。
- 在生成的结点中，抛弃那些不满足约束条件（或者说不可能导出最优可行解）的结点。

搜索方式

- 深度优先搜索
- 广度优先搜索

第六章 分支限界法

方法1：深度优先搜索

- 通常深度优先搜索法不全部保留结点，扩展完的结点从数据存储结构栈中弹出删去，这样，一般在数据栈中存储的结点数就是解空间树的深度，因此它占用空间较少。
- 所以，当搜索树的结点较多，用其它方法易产生内存溢出时，深度优先搜索不失为一种有效的求解方法。

第六章 分支限界法

方法2：广度优先搜索

- 广度优先搜索算法，一般需存储产生的所有结点，占用的存储空间要比深度优先搜索大得多，因此，程序设计中，必须考虑溢出和节省内存空间的问题。
- 但广度优先搜索法一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索要快些。

第六章 分支限界法----基本思想

什么是分支限界法

- 采用广度优先产生状态空间树的结点，并使用剪枝函数的方法称为分支限界法。
  - 所谓“分支”是采用广度优先的策略，依次生成扩展结点的所有分支（即：儿子结点）。
  - 所谓“限界”是在结点扩展过程中，计算结点的上界（或下界），边搜索边减掉搜索树的某些分支，从而提高搜索效率

第六章 分支限界法----基本思想

搜索策略

- 按照广度优先的原则，一个活结点一旦成为扩展结点后，算法将依次生成它的全部孩子结点，将那些导致不可行解或导致非最优解的儿子舍弃，其余儿子加入活结点表中。
- 然后，从活结点表中取出一个结点作为当前扩展结点。
- 重复上述结点扩展过程，直至找到问题的解或判定无解为止。

第六章 分支限界法----基本思想

分支限界法 Vs 回溯法

- 相同点：
  - 两者在进行问题求解前，都需要完成解空间的定义和组织；
  - 都是通过在解空间搜索来寻找问题的解；

第六章 分支限界法----基本思想

不同点	回溯法	分支限界法
求解目标	找出树中满足约束条件的所有解	找出满足约束条件的一个解或找出使目标函数达到极大(小)的最优解
搜索方式	深度优先	广度优先或最小耗费优先
扩展结点	多次机会成为扩展结点:扩展结点变为活结点后又可成为扩展结点	每个活结点只有一次机会成为扩展结点
树结点的生成顺序	生成最近一个有希望结点的单个子女	选择其中最有希望的结点,并生成它的所有子女
行进方向	随机性	方向性:活结点表,搜索朝着解空间树上有最优解的分支推进

第六章 分支限界法----分类

分支限界法的主要分类

- 分支限界法的主要分类
  - 根据从活结点表中选择下一个扩展结点的方式
    - 队列式**FIFO**分支限界法
    - 优先队列式分支限界法

6.2.1 队列式(FIFO)分支限界法

- 基本思想:
  - 按照队列的先进先出（FIFO）原则选取下一个活结点为扩展结点。
- 搜索策略
  - 一开始，根结点是唯一的活结点，根结点入队。
  - 从活结点队中取出根结点后，作为当前扩展结点。
  - 对当前扩展结点，先从左到右地产生它的所有儿子，用约束条件检查，把所有满足约束函数的儿子加入活结点队列中。
  - 再从活结点表中取出队首结点（队中最先进来的结点）为当前扩展结点，……，直到找到一个解或活结点队列为空为止。

第六章 分支限界法----分类

6.2.2优先队列式分支限界法

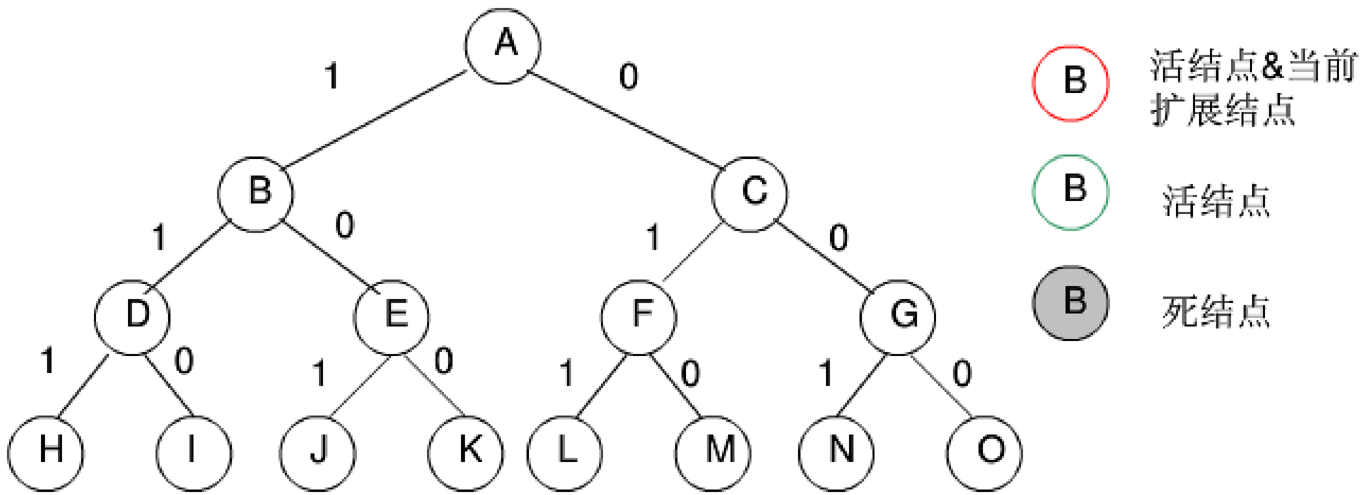
- 基本思想：
  - 为了加速搜索的进程，应采用有效地方式选择活结点进行扩展。
  - 按照优先队列中规定的优先级选取优先级最高的结点成为当前扩展结点。
- 基本策略
  - 对每一活结点计算一个优先级（某些信息的函数值），并根据这些优先级；
  - 从当前活结点表中优先选择一个优先级最高（最有利）的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。
  - 再从活结点表中下一个优先级别最高的结点为当前扩展结点，……，直到找到一个解或活结点队列为空为止。

实例说明——0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

解空间为：

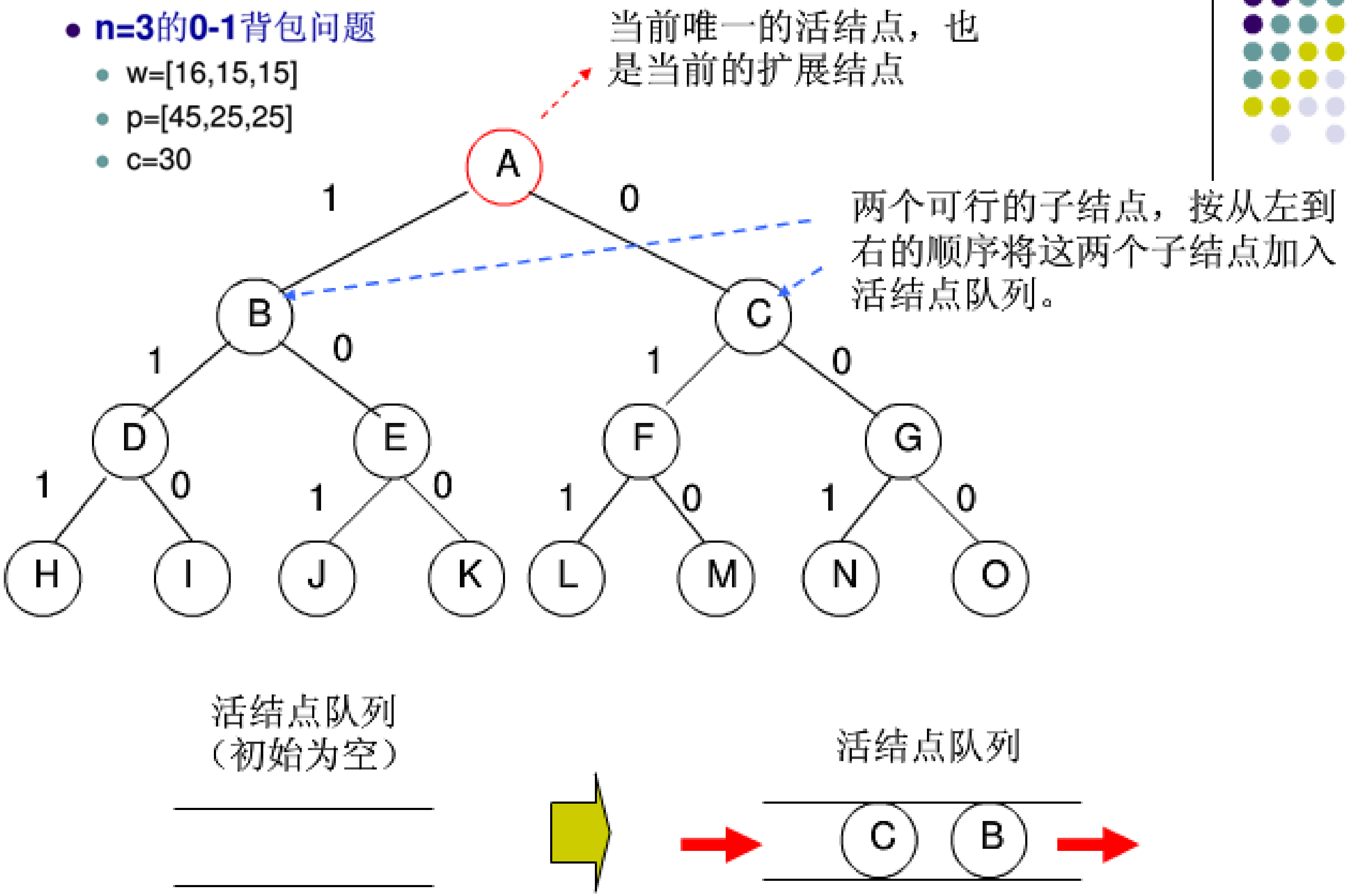
{(0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)}



n=3的0-1背包问题的解空间树

队列式（FIFO）分支限界法 ----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

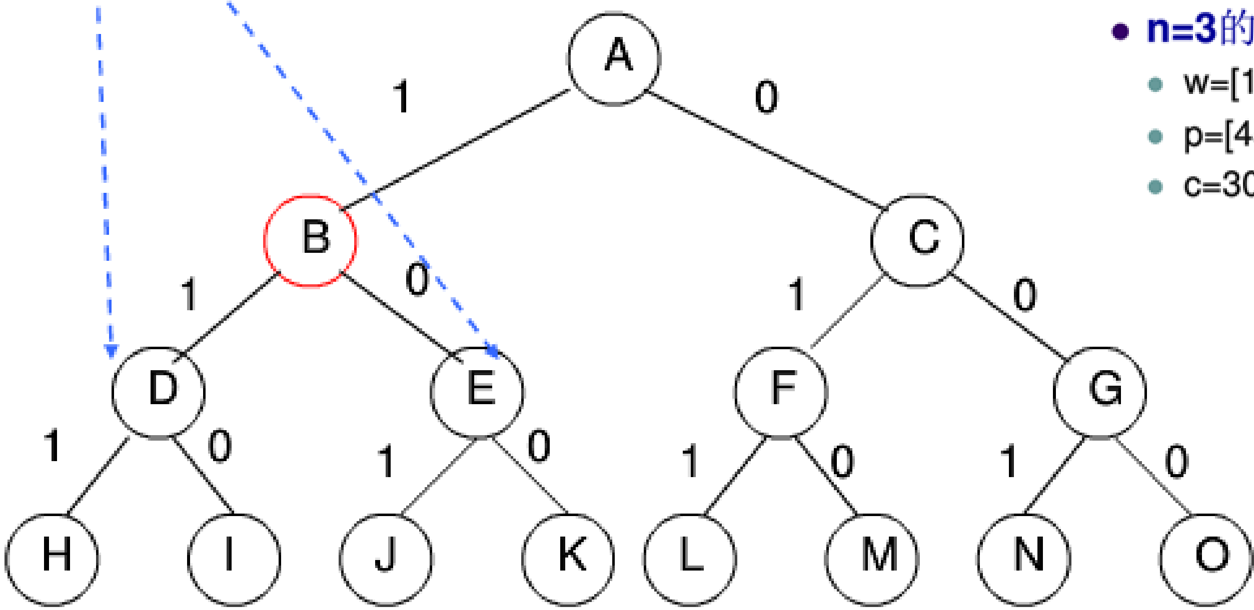


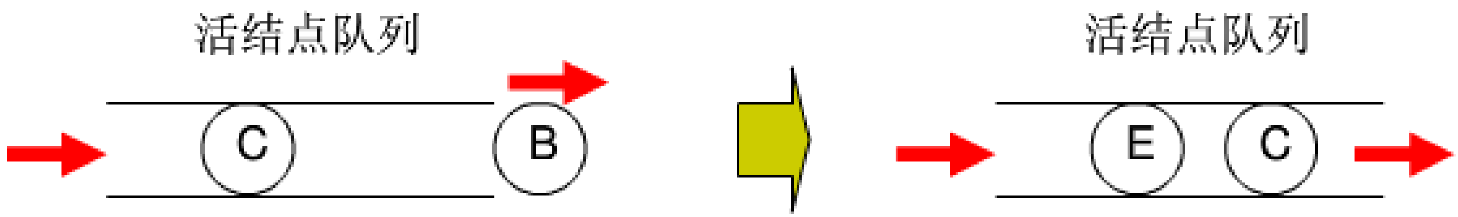
队列式（FIFO）分支限界法 ----0-1背包问题

两个子结点，但D是不可行结点（因为超过背包容量），所以将D舍去。E是可行结点。

将结点E加入到活结点队列中

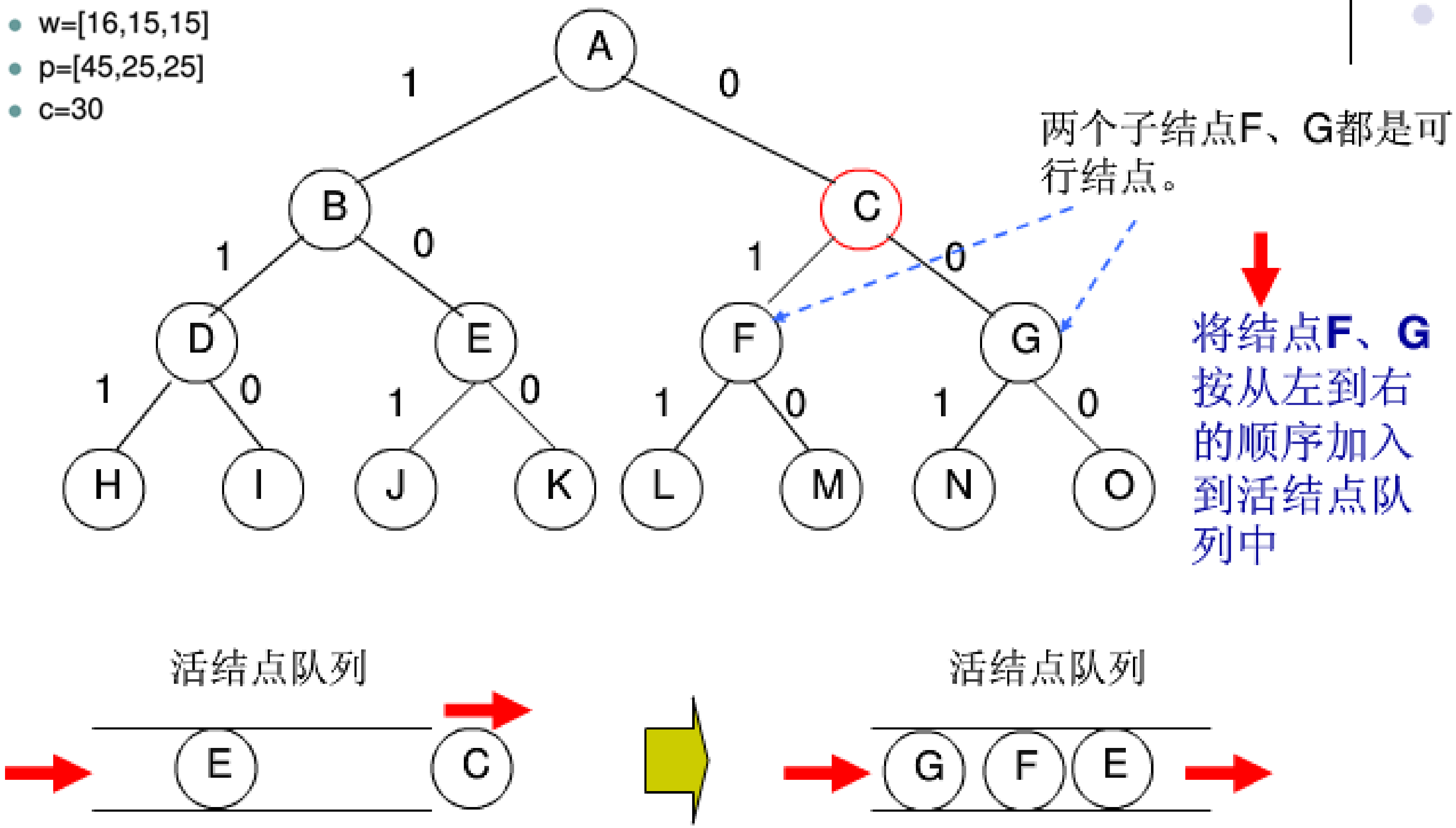
- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30





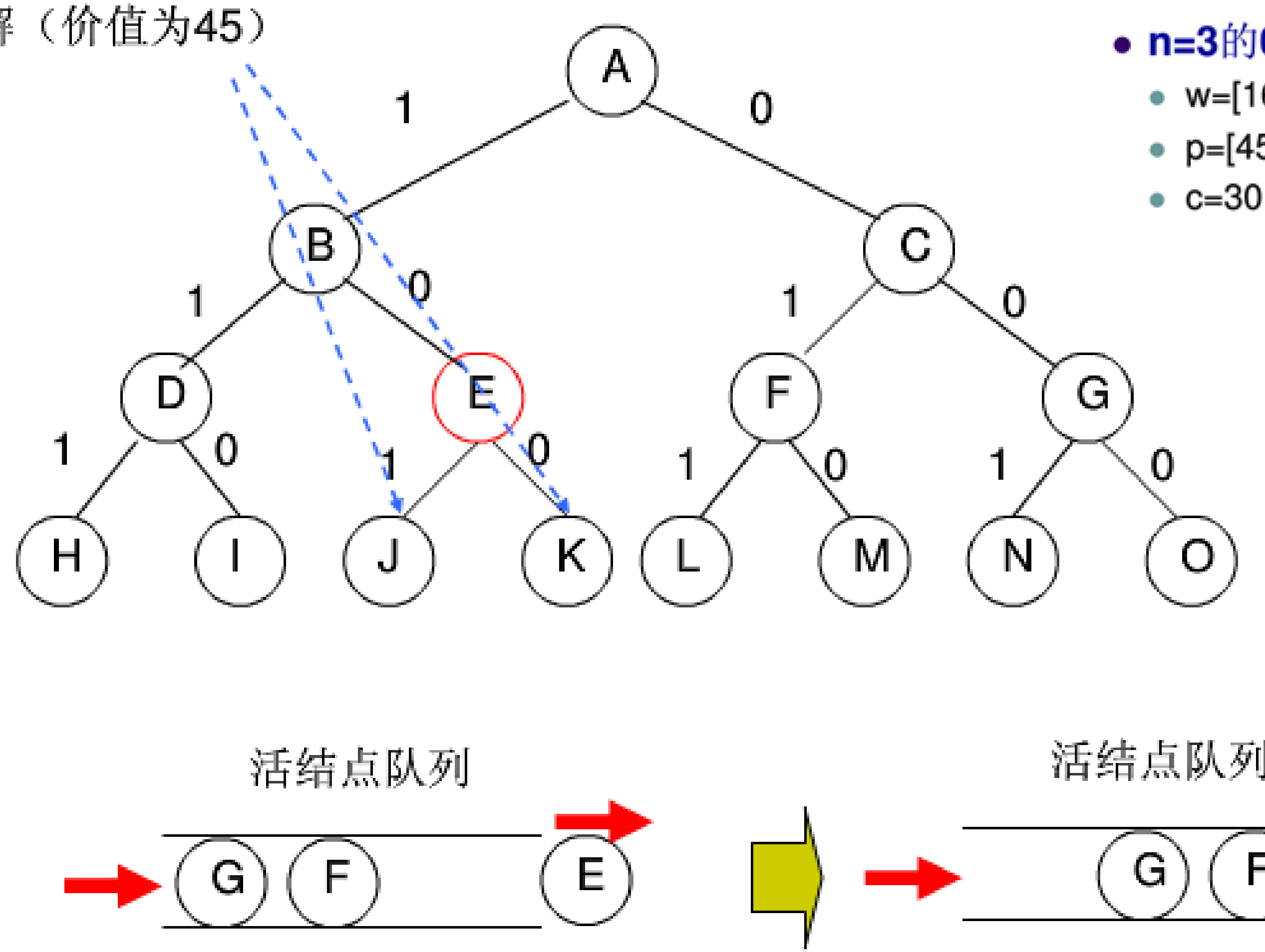
队列式（FIFO）分支限界法 ----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30



队列式（FIFO）分支限界法 ----0-1背包问题

两个子结点都是叶结点，其中J是不可行解，而K是可行解（价值为45）

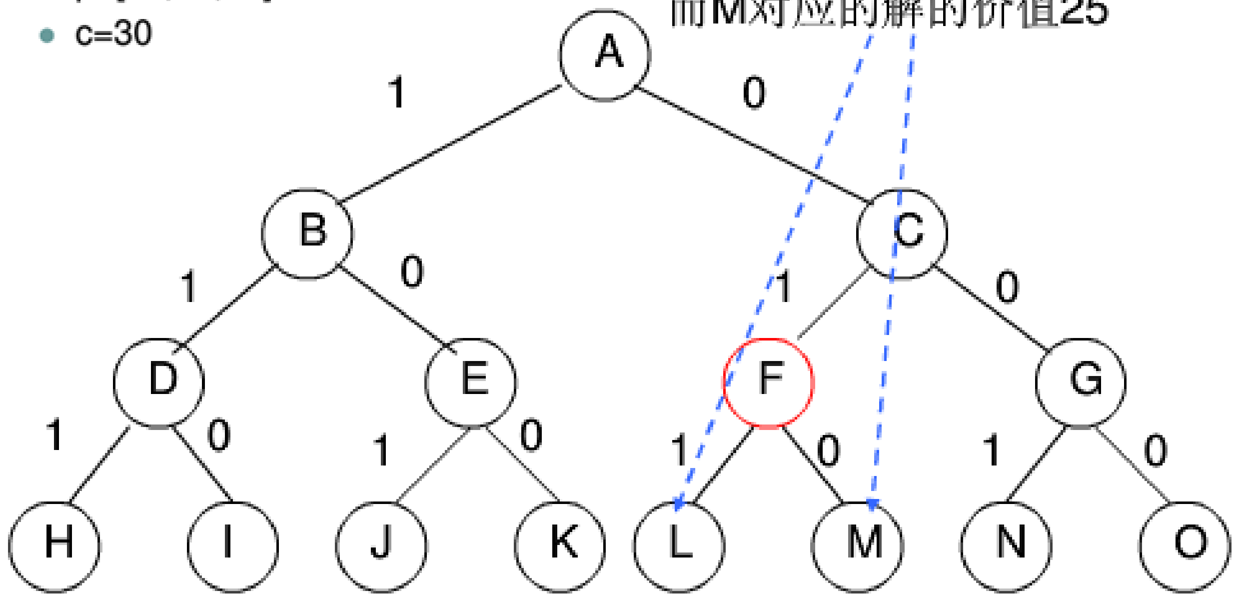


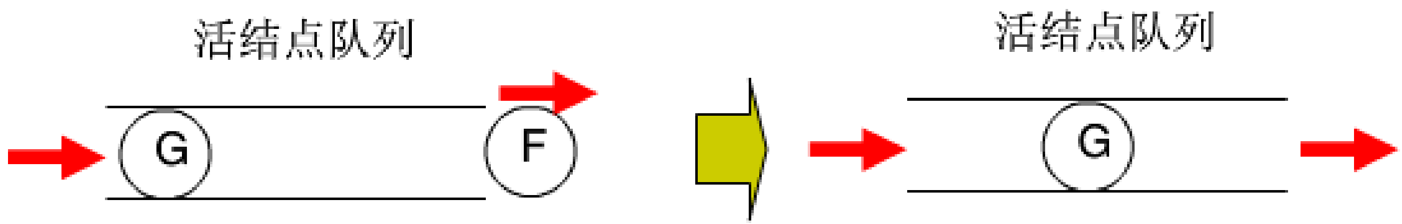
- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

队列式（FIFO）分支限界法 ----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

两个子结点都是叶结点，而且都分别对应一个可行解，其中L对应的解的价值50，而M对应的解的价值25



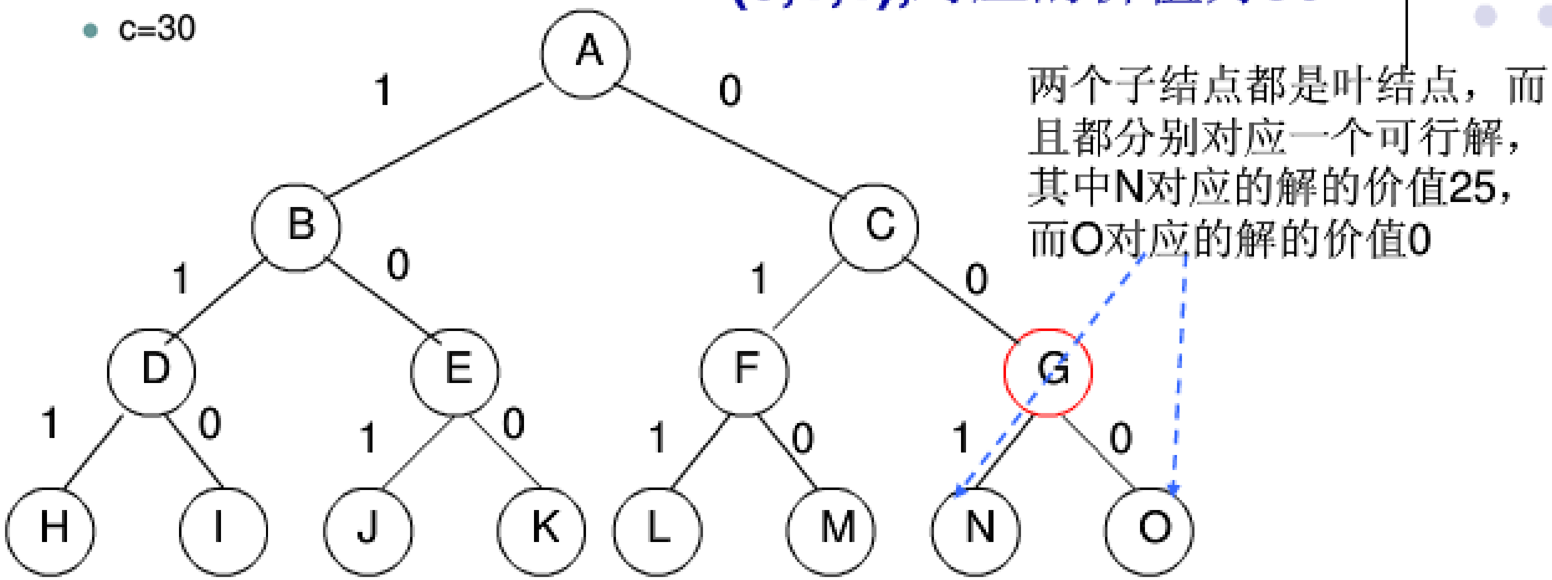


队列式（FIFO）分支限界法 ----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

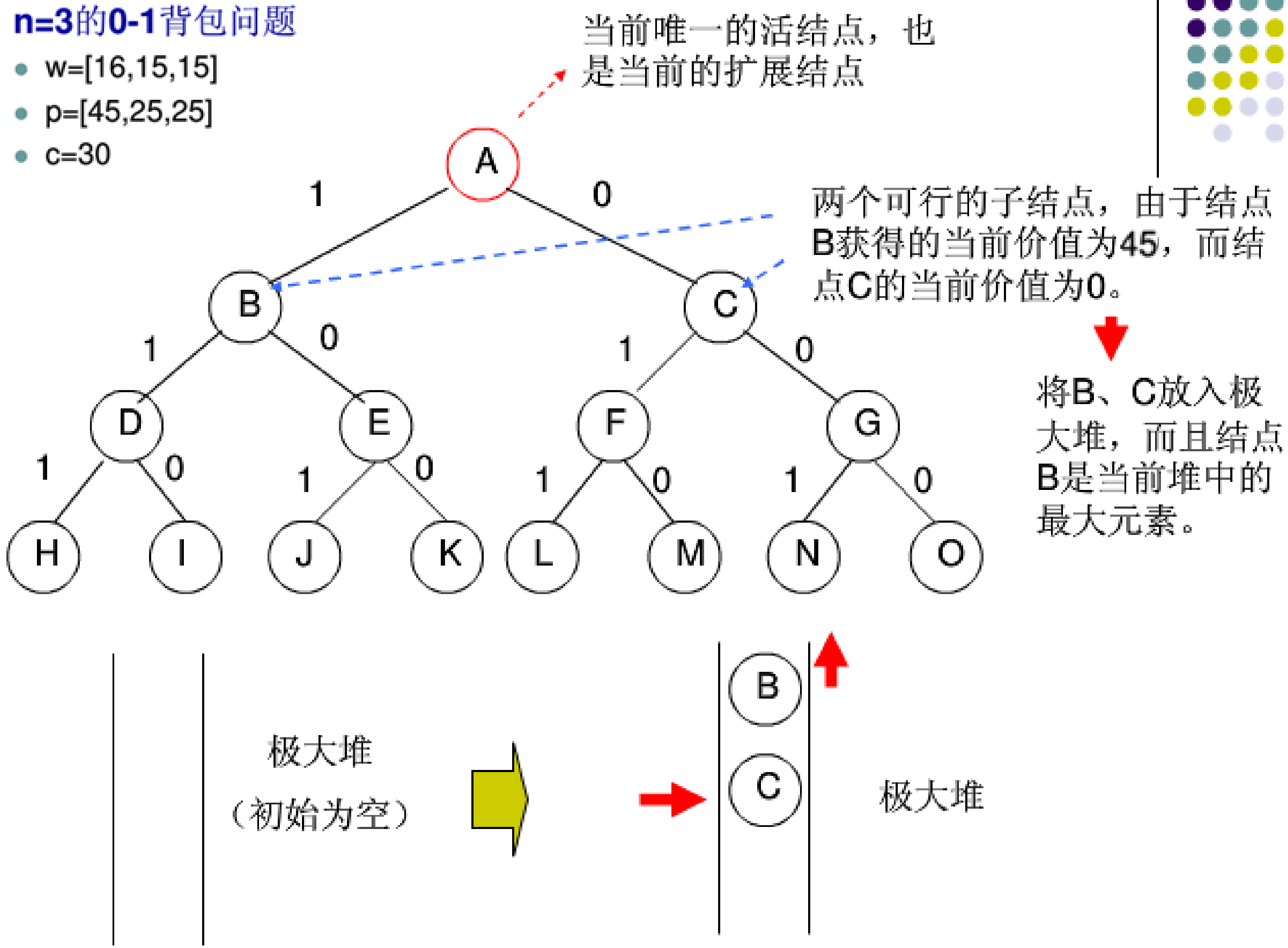
最优解为：

(0,1,1),对应的价值为50



优先队列式分支限界法
----0-1背包问题

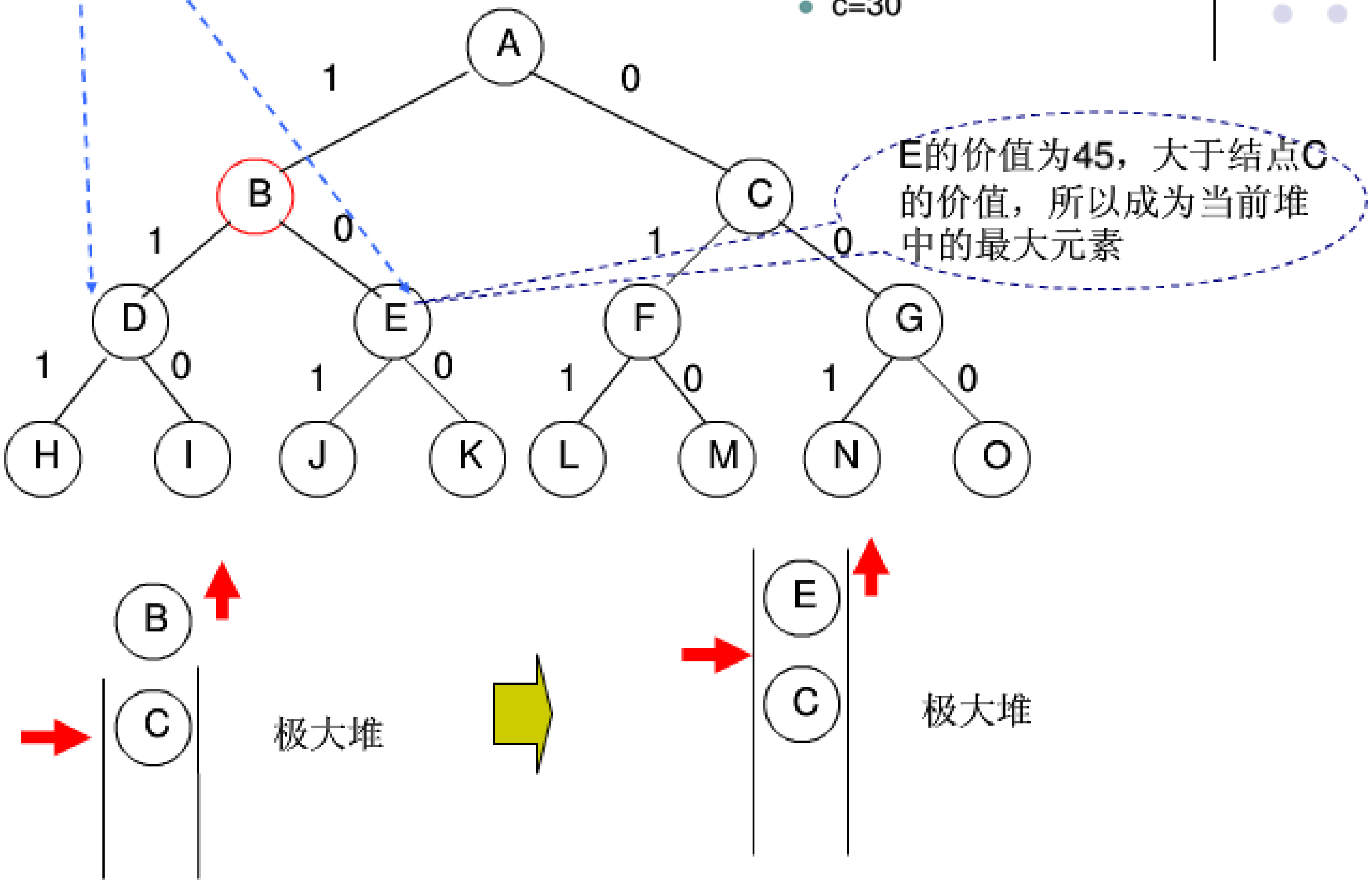
- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30



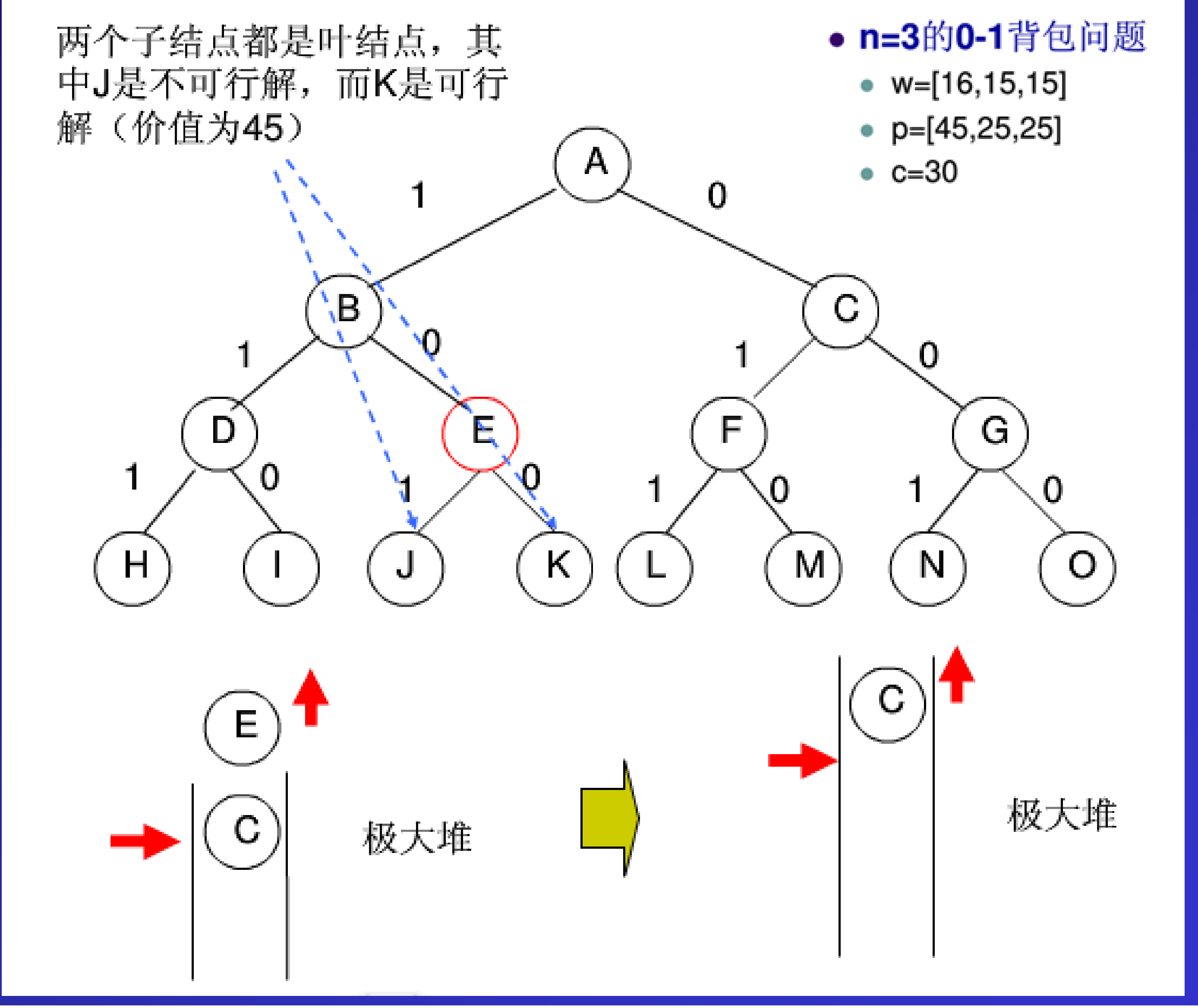
优先队列式分支限界法 ----0-1背包问题

两个子结点，但D是不可行结点（因为超过背包容量），所以将D舍去。E是可行结点。

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

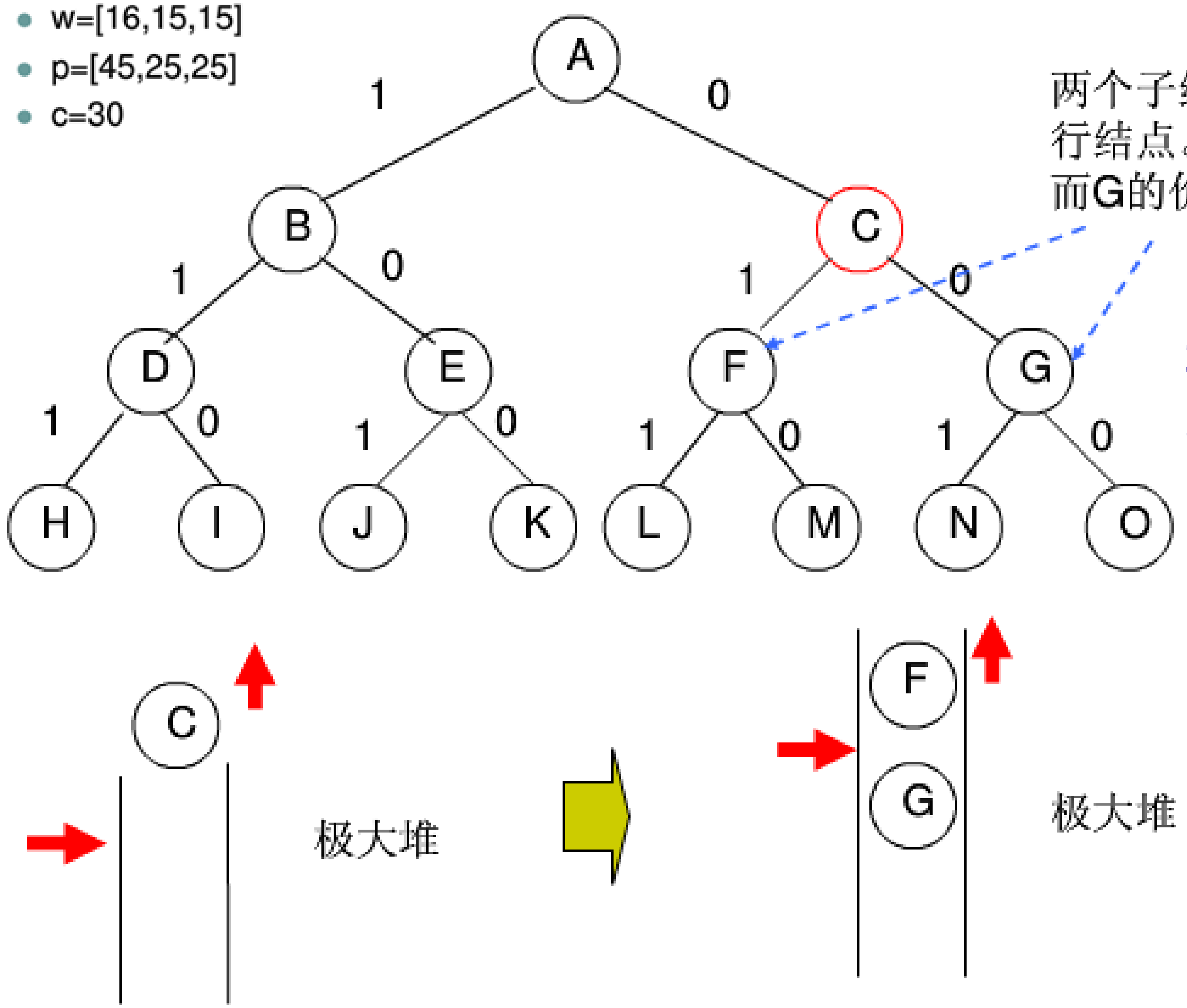


优先队列式分支限界法
----0-1背包问题



优先队列式分支限界法 ----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

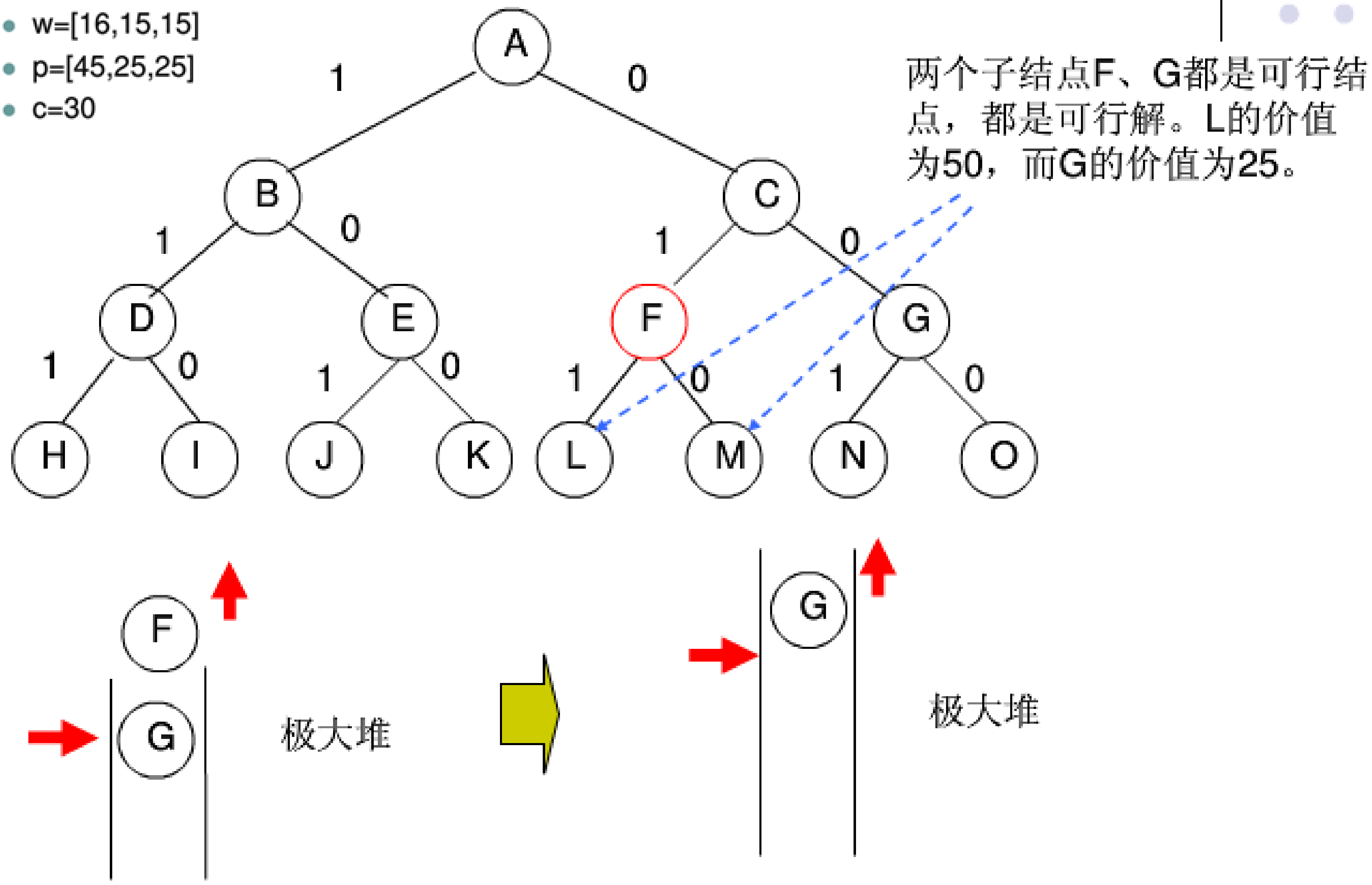


两个子结点F、G都是可行结点。F的价值为25，而G的价值为0。

将结点F、G加入到极大堆中，结点F成为当前堆中的最大元素。

优先队列式分支限界法
----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

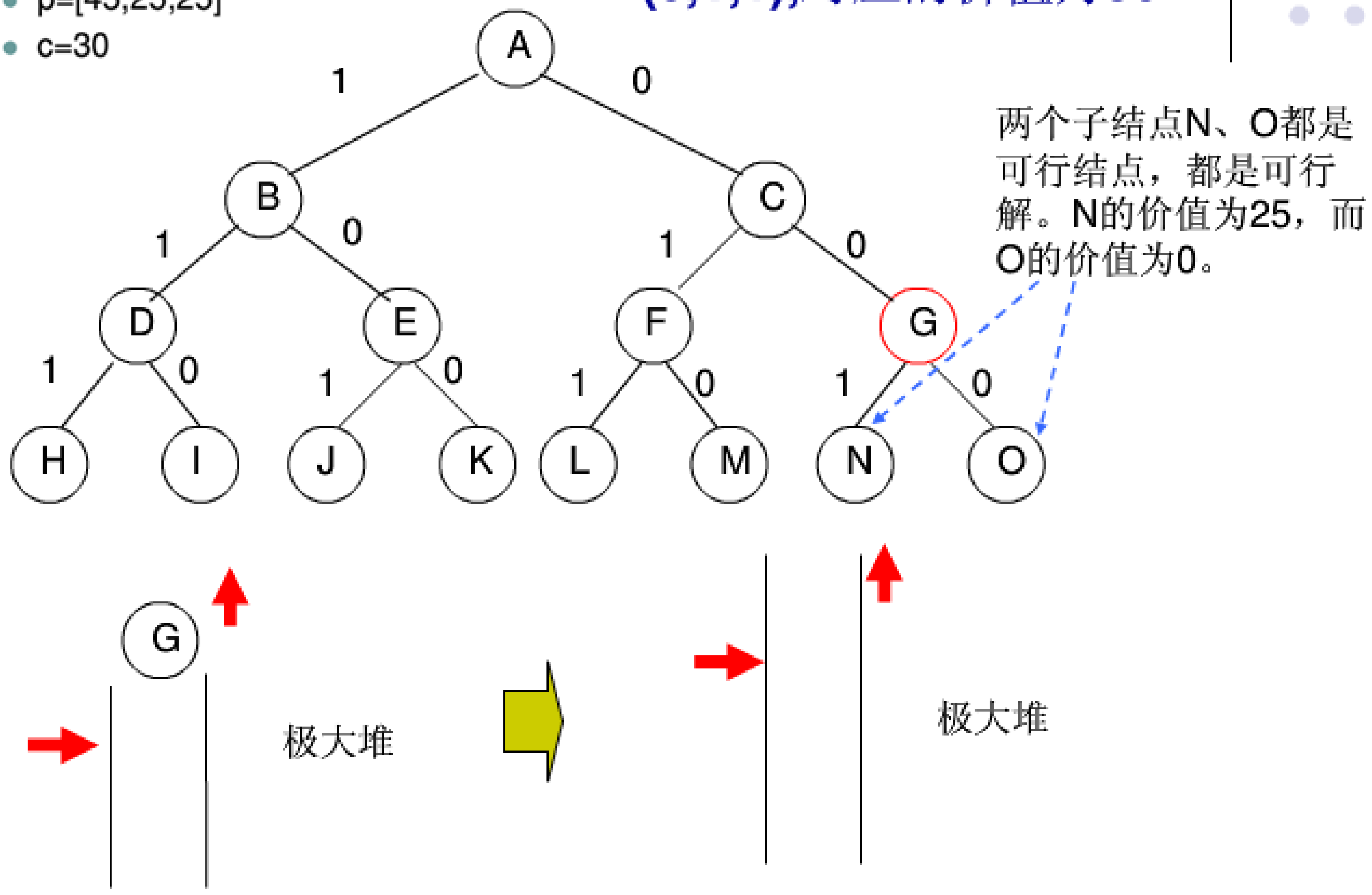


优先队列式分支限界法
----0-1背包问题

- **n=3的0-1背包问题**
  - w=[16,15,15]
  - p=[45,25,25]
  - c=30

最优解为：

(0,1,1),对应的价值为50



分支限界法----0-1背包问题

- 分支限界搜索过程

cw:当前装包重量
cp: 当前装包价值

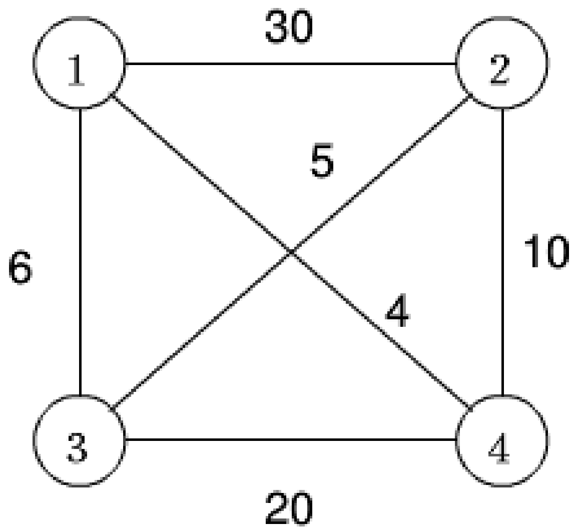
```
while (i != n+1) {                  // 非叶结点
   // 检查当前扩展结点的左儿子结点
   Typew wt = cw + w[i];
   if (wt <= c) {                   // 左儿子结点为可行结点
     if (cp+p[i] > bestp) bestp = cp+p[i];
     AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);}//加入活结点队列
     up = Bound(i+1);
   // 检查当前扩展结点的右儿子结点
   if (up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

   //  取下一个扩展节点（略）}
```

可行性约束

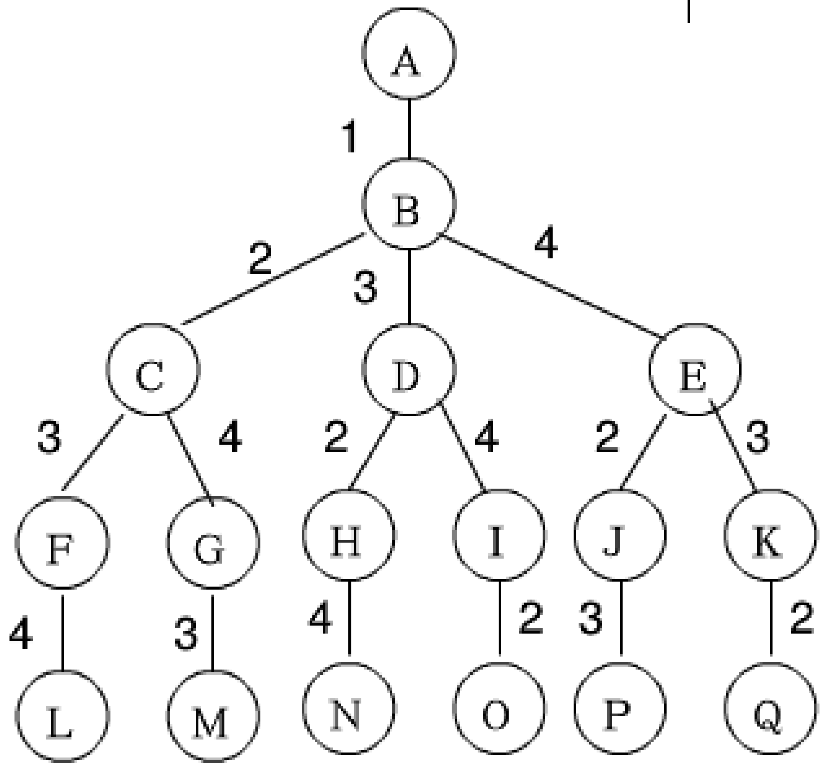
上界约束

将一个新的活结点插入到子集树和优先队列中
i+1: 层数

实例说明——TSP问题

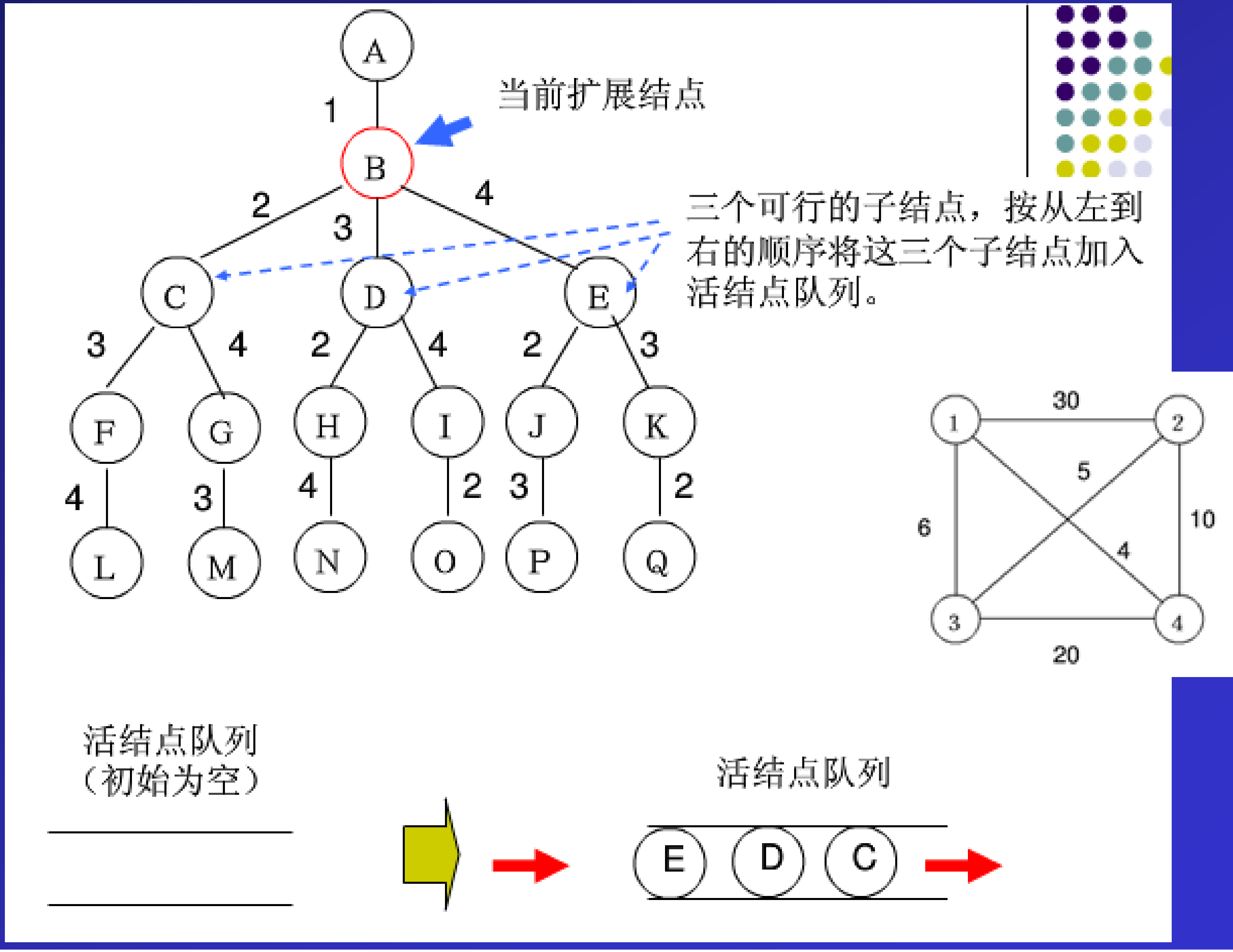


4顶点的无向带权图

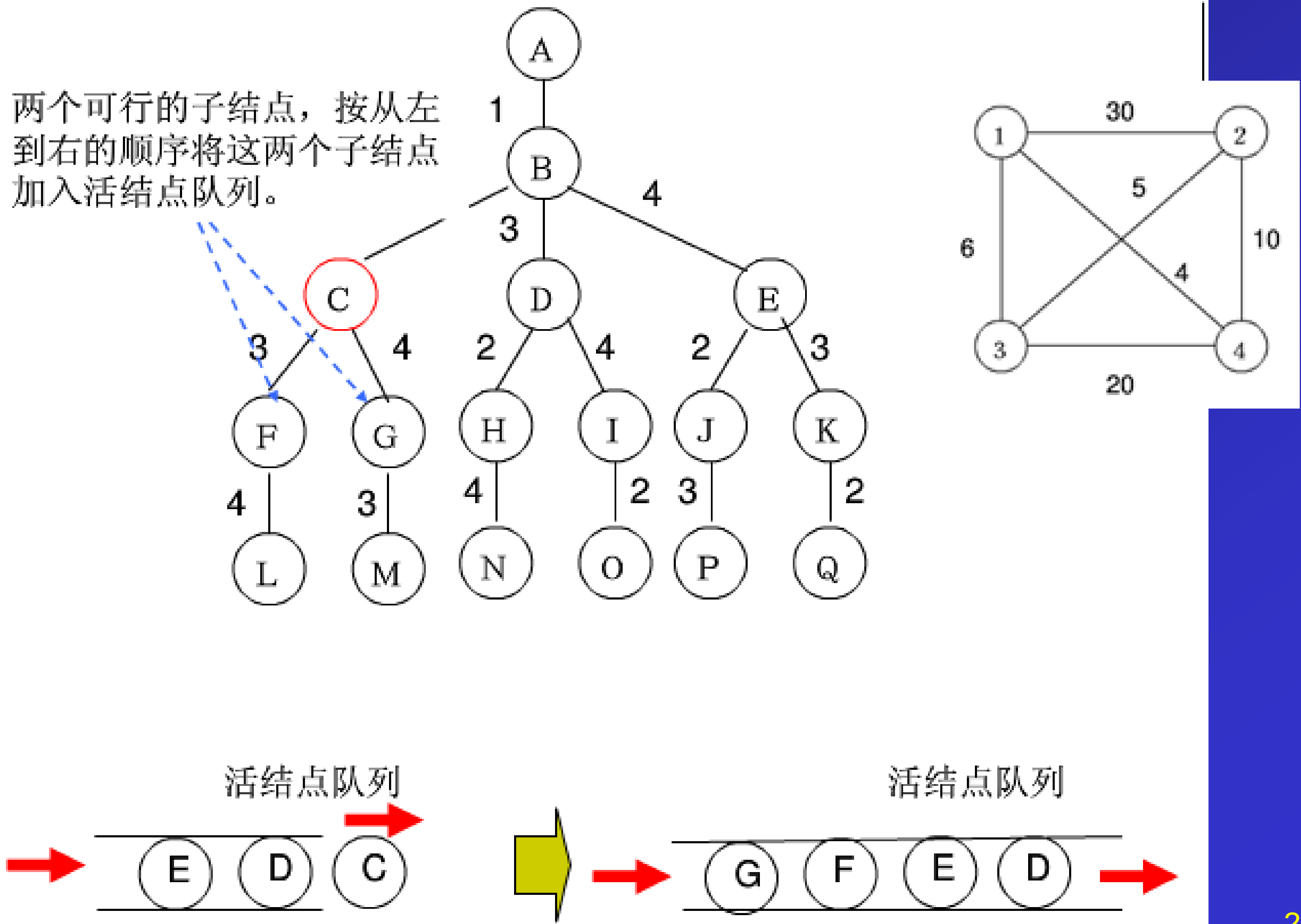


n=4的TSP问题的解空间树（排列树）

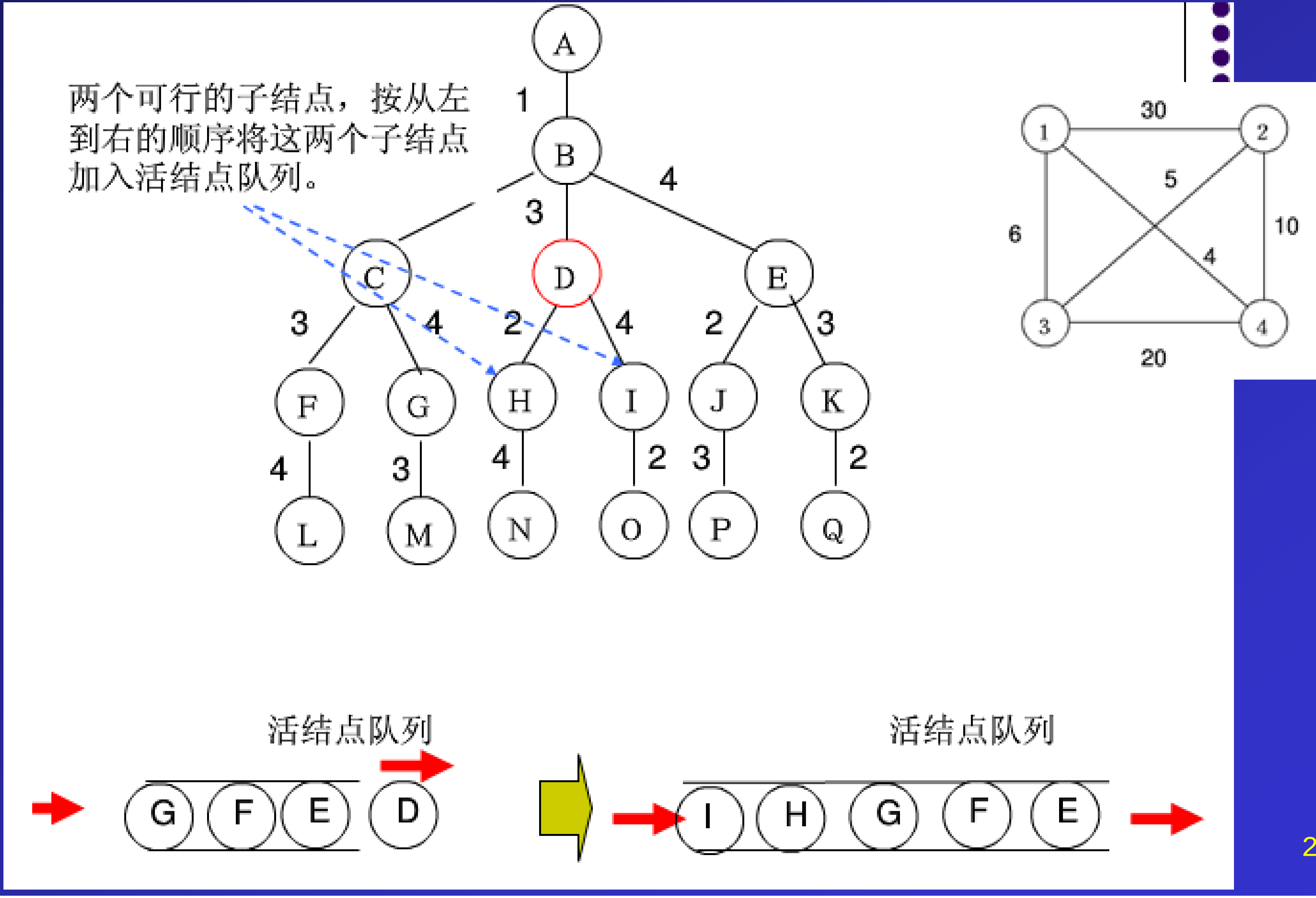
队列式（FIFO）分支限界法 ----TSP问题



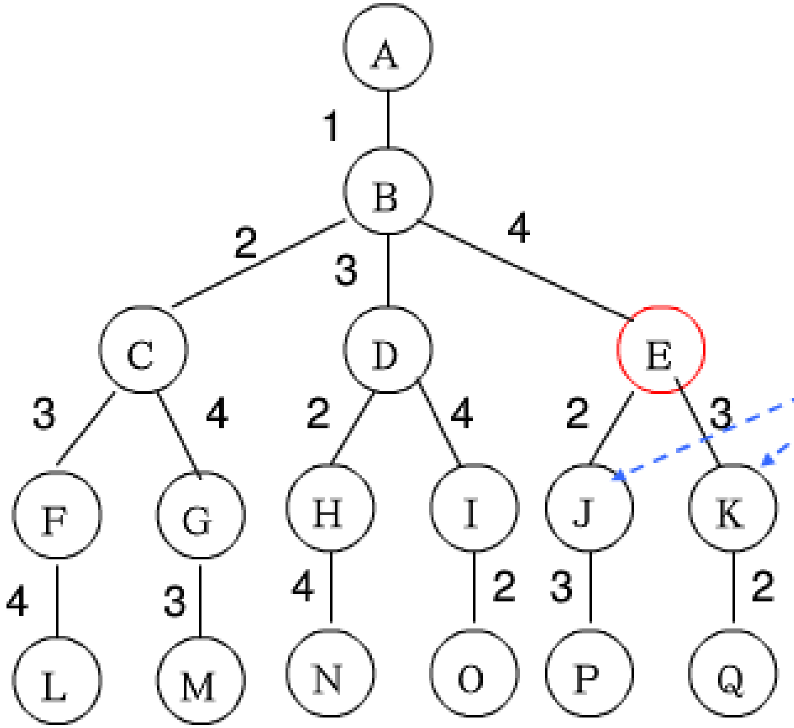
队列式（FIFO）分支限界法 ----TSP问题



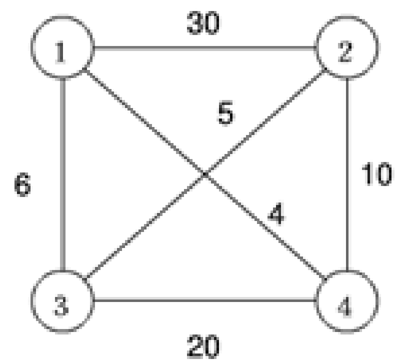
队列式（FIFO）分支限界法 ----TSP问题

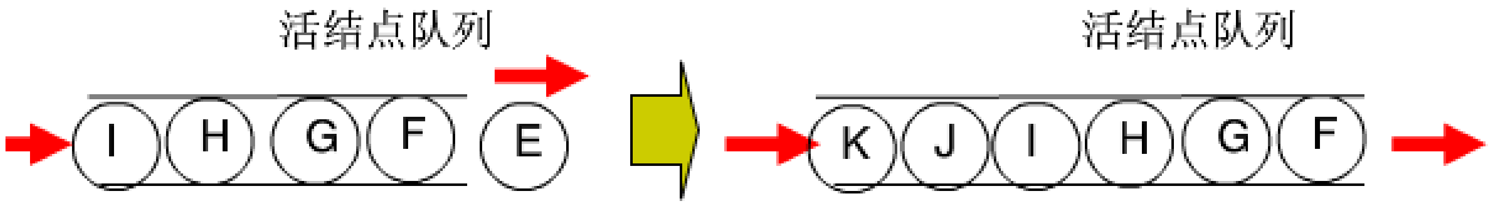


队列式（FIFO）分支限界法 ----TSP问题



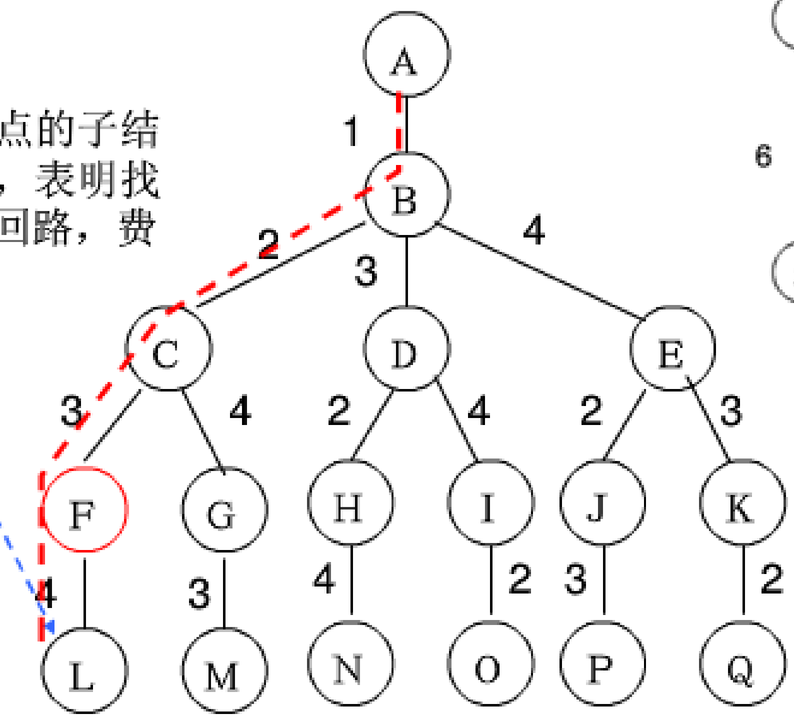
两个可行的子结点，按从左到右的顺序将这两个子结点加入活结点队列。

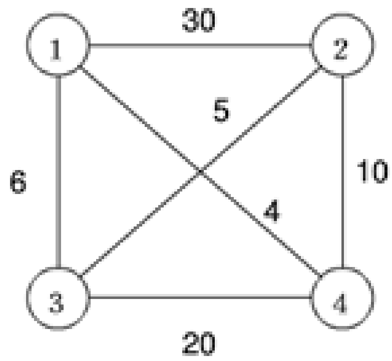




队列式（FIFO）分支限界法 ----TSP问题

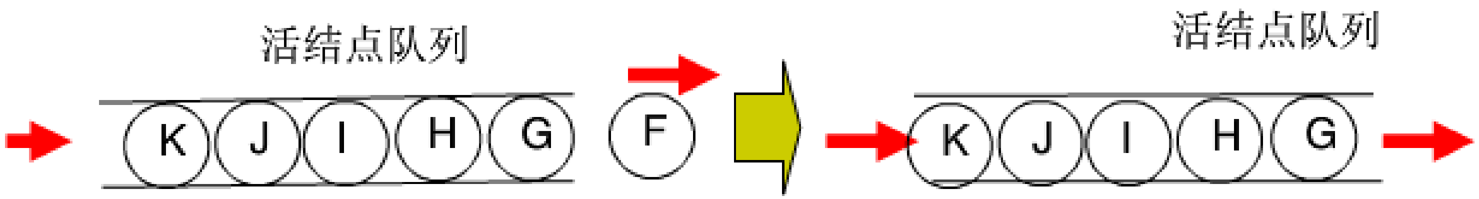
当前扩展结点的子结点是叶结点，表明找到一条TSP回路，费用为59。





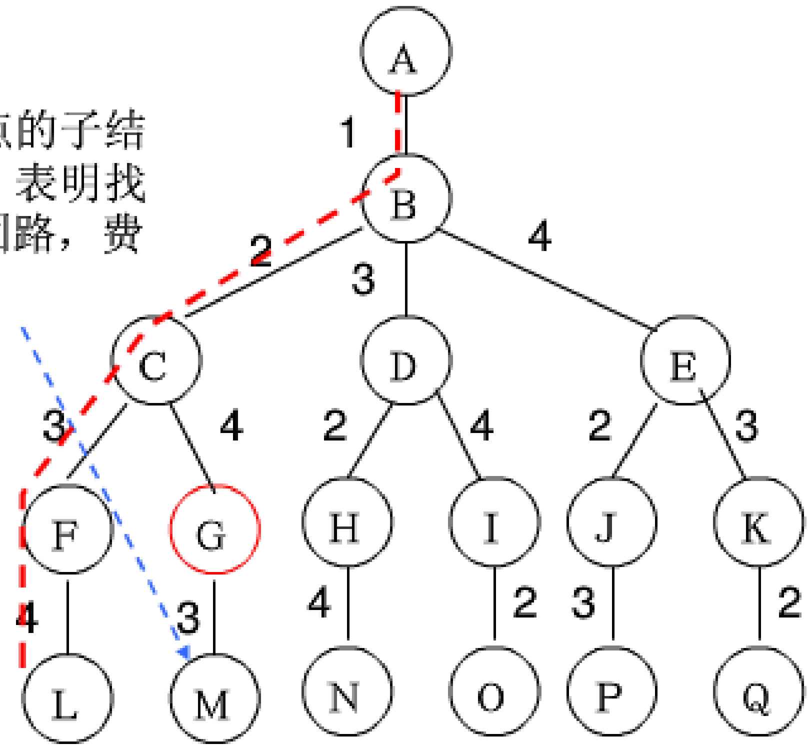
当前最优解为**12341**

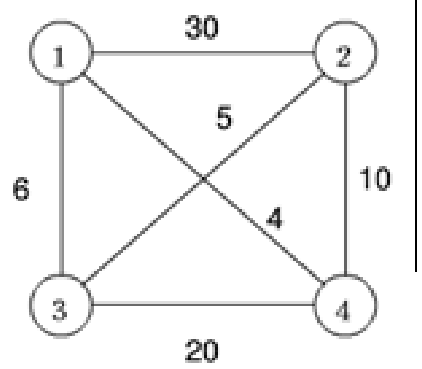
费用：**59**



队列式（FIFO）分支限界法 ----TSP问题

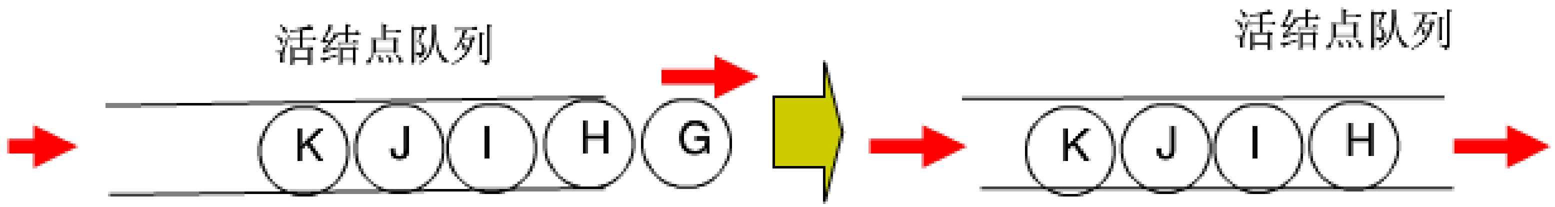
当前扩展结点的子结点是叶结点，表明找到一条TSP回路，费用为66。





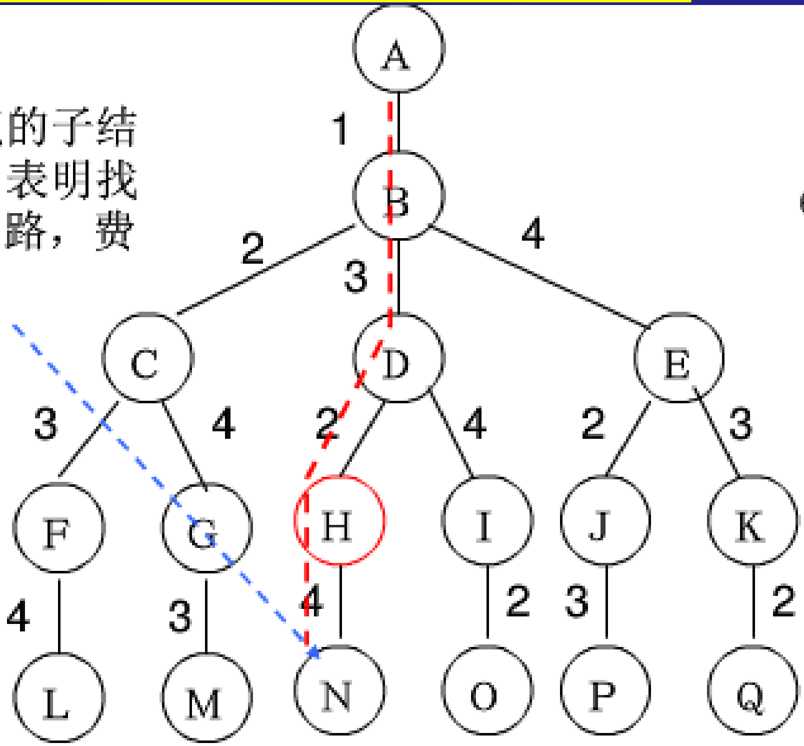
当前最优解为**12341**

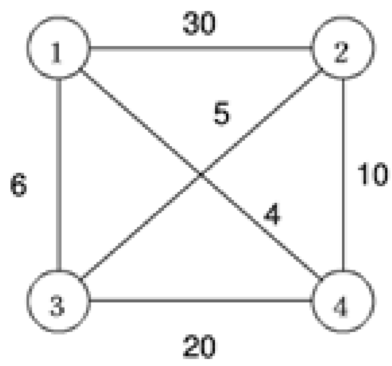
费用：**59**



队列式（FIFO）分支限界法 ----TSP问题

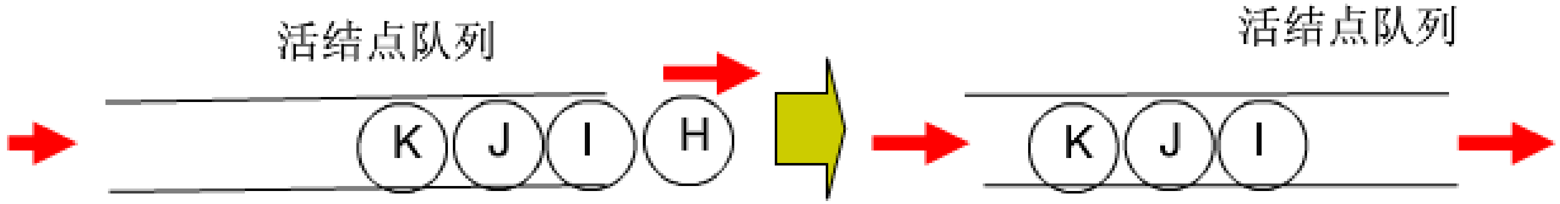
当前扩展结点的子结点是叶结点，表明找到一条TSP回路，费用为25。



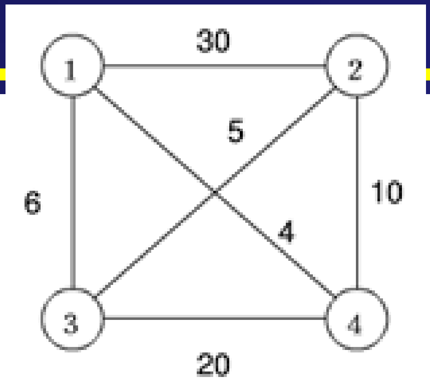


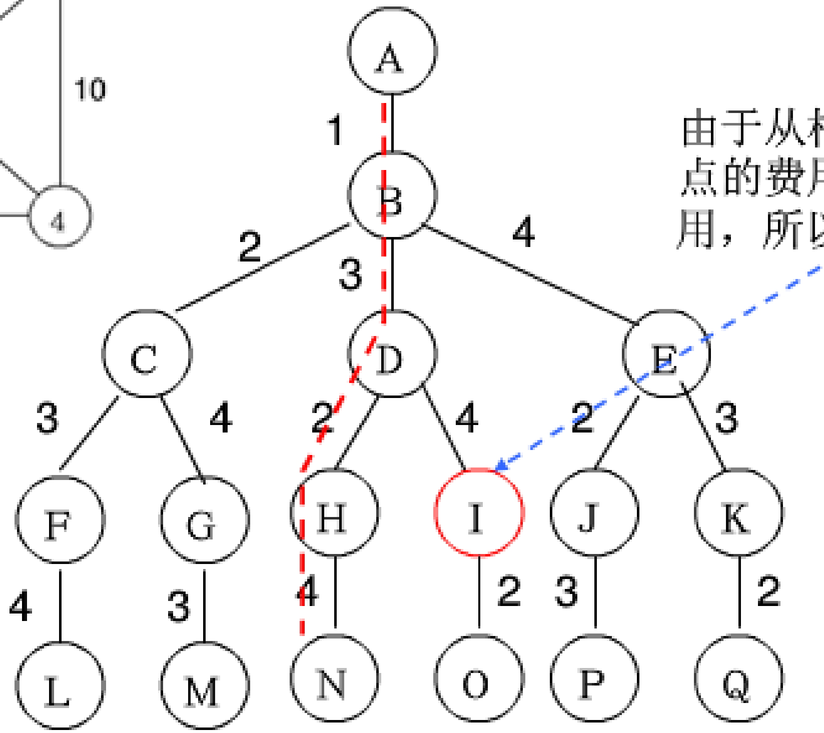
当前最优解为13241

费用：25



队列式（FIFO）分支限界法 ----TSP问题

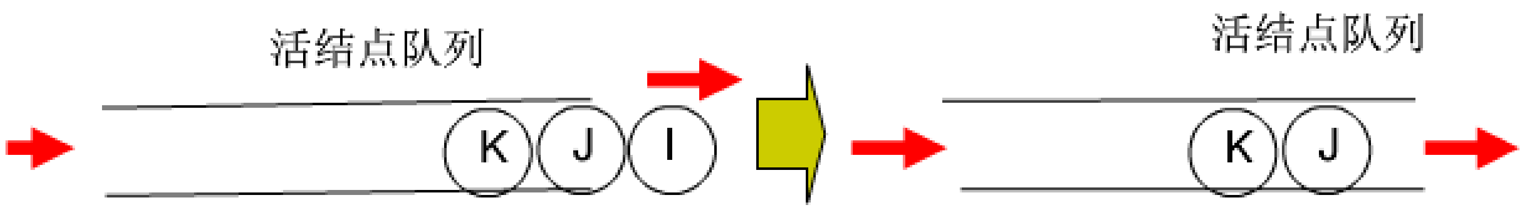




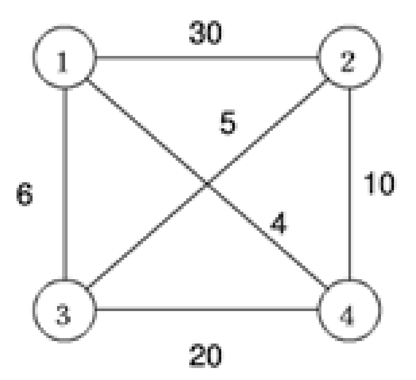
由于从根结点到当前结点的费用>当前最优费用，所以结点I被舍弃。

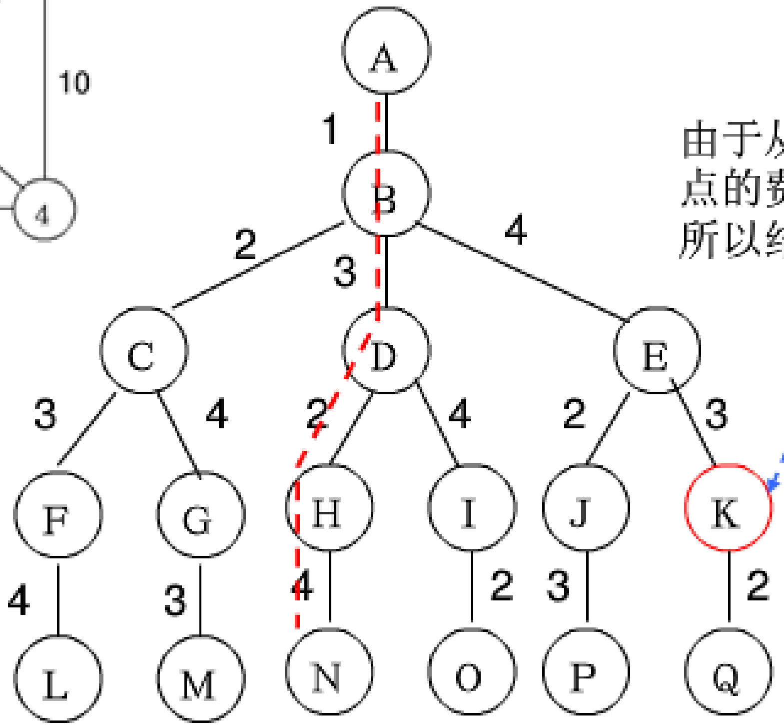
当前最优解为13241

费用：25



队列式（FIFO）分支限界法 ----TSP问题

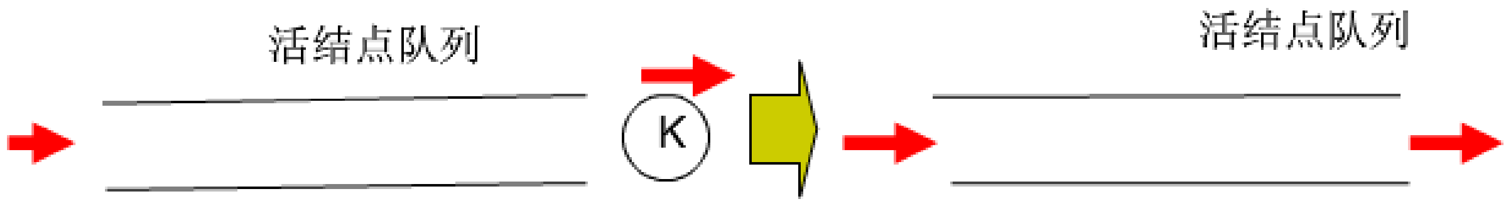




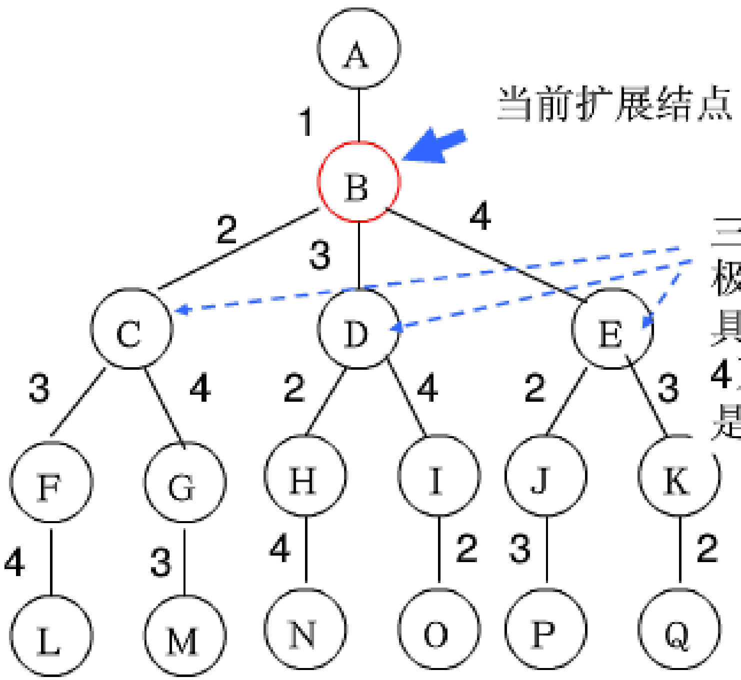
由于从根结点到当前结点的费用>当前最优费用，所以结点K被舍弃。

当前最优解为13241

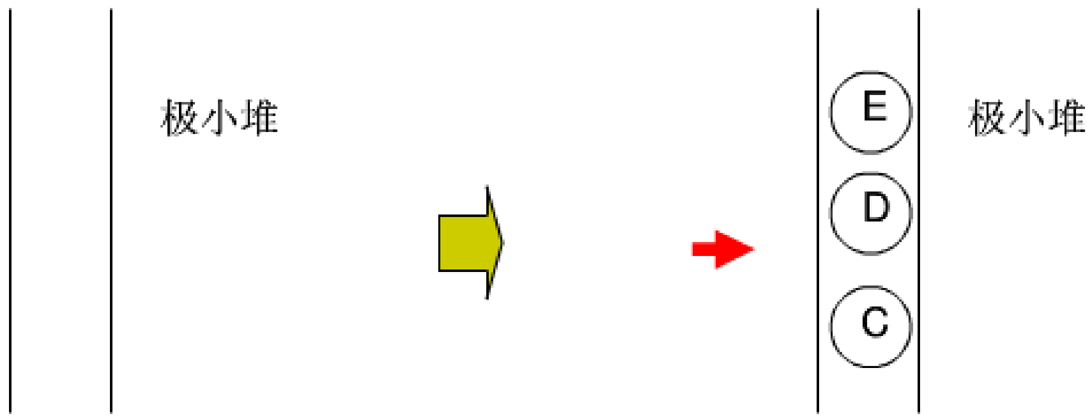
费用：25

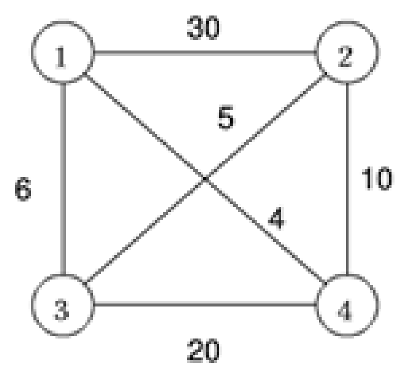


优先队列式分支限界法 ----TSP问题



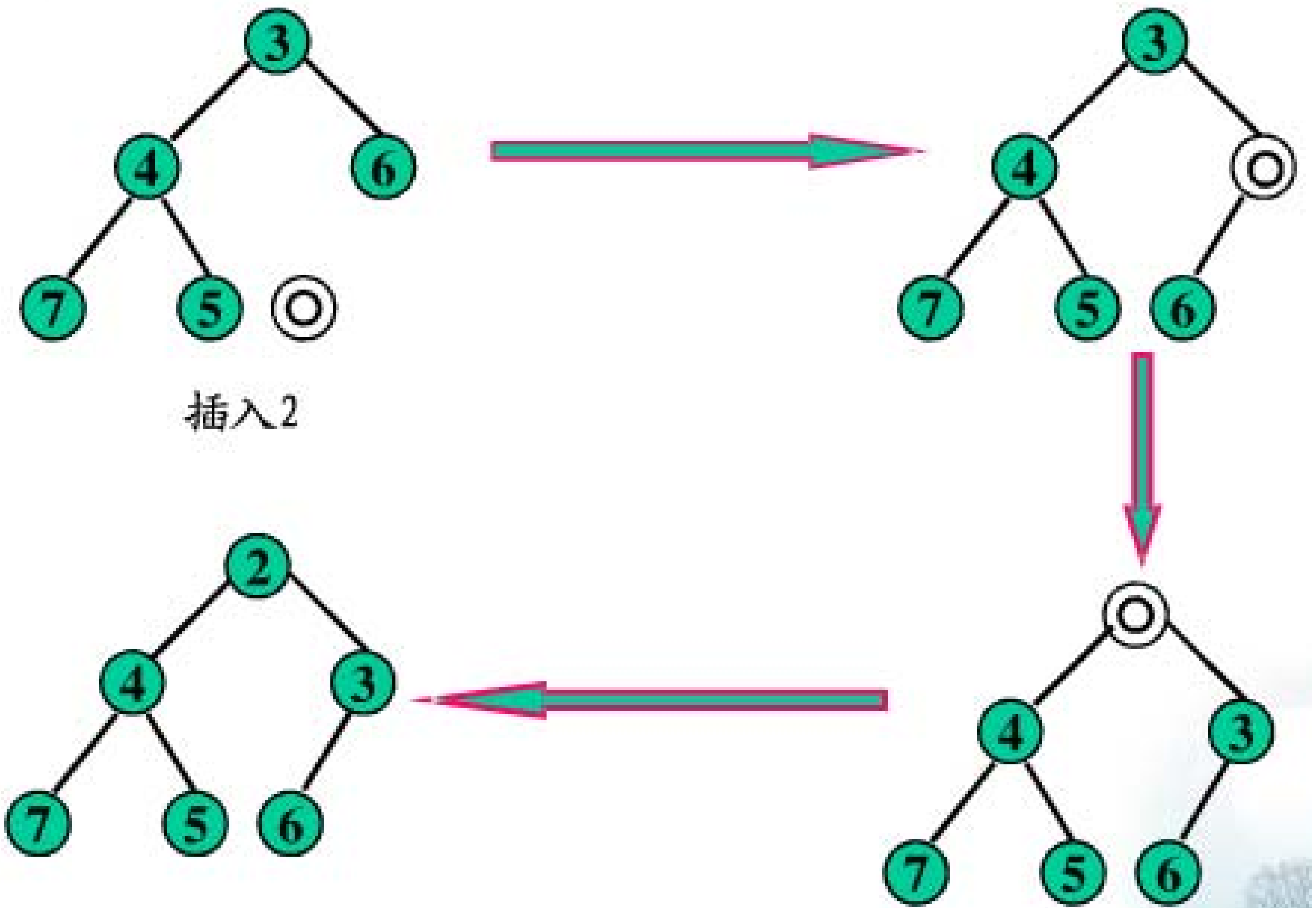
三个可行的子结点，被加入到极小堆中。由于E是当前堆中具有最小费用的结点（费用为4），所以处于堆顶，向下依次是D和C。



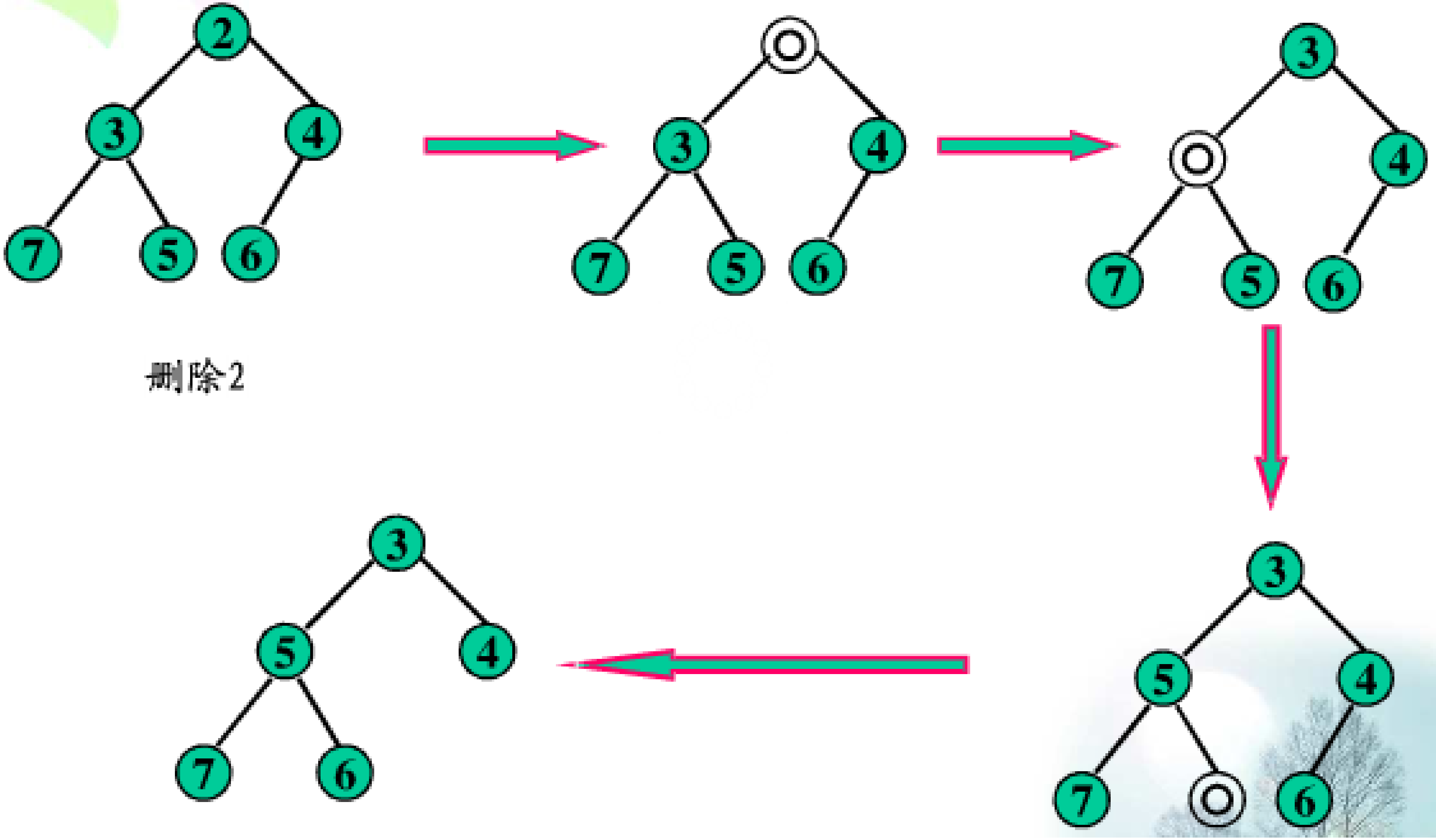


最小堆的插入和删除

1)插入

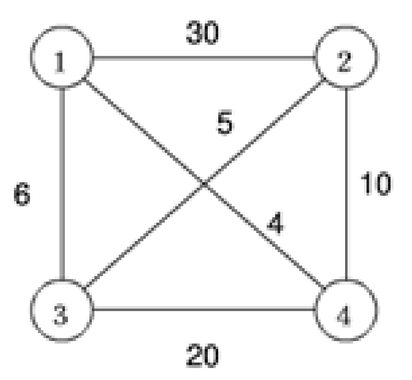


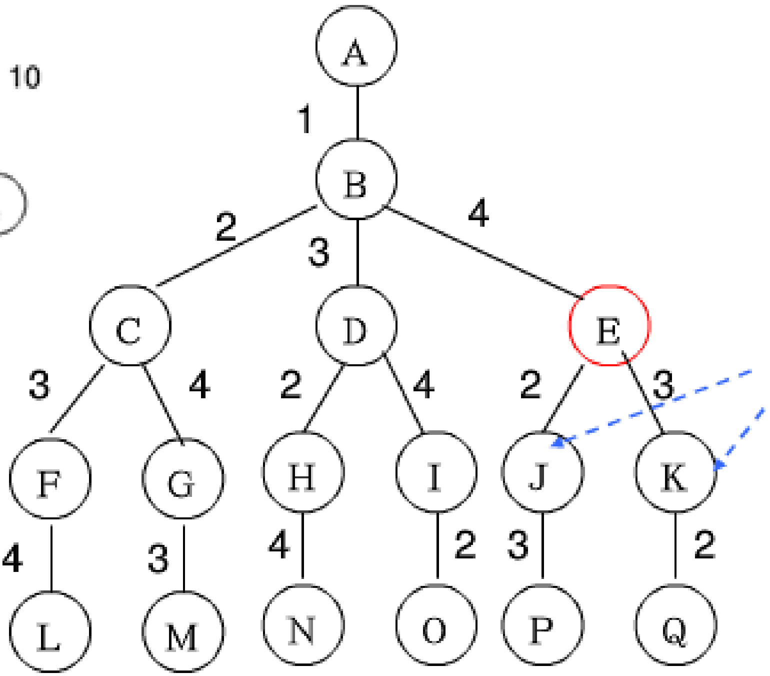
2)删除



删除2

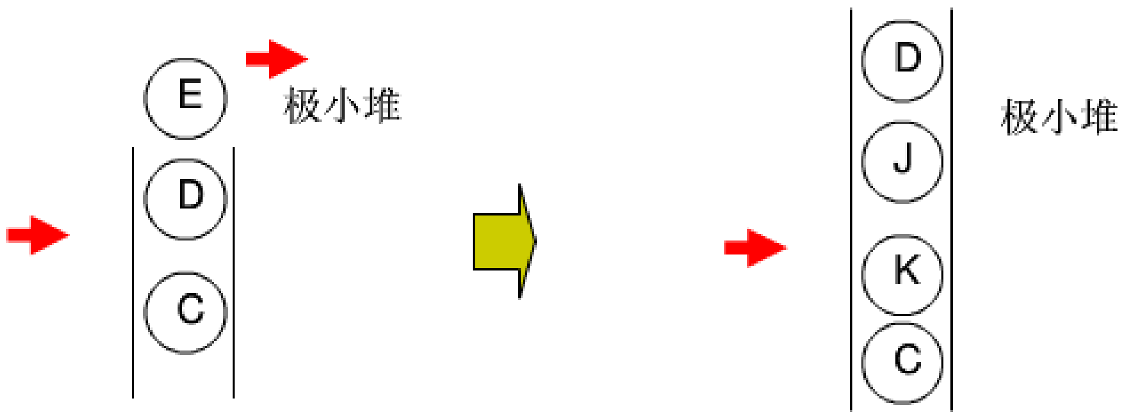
优先队列式分支限界法 ----TSP问题



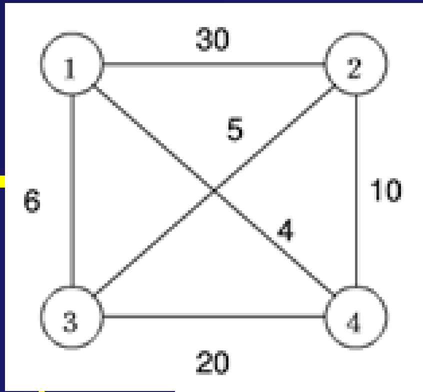


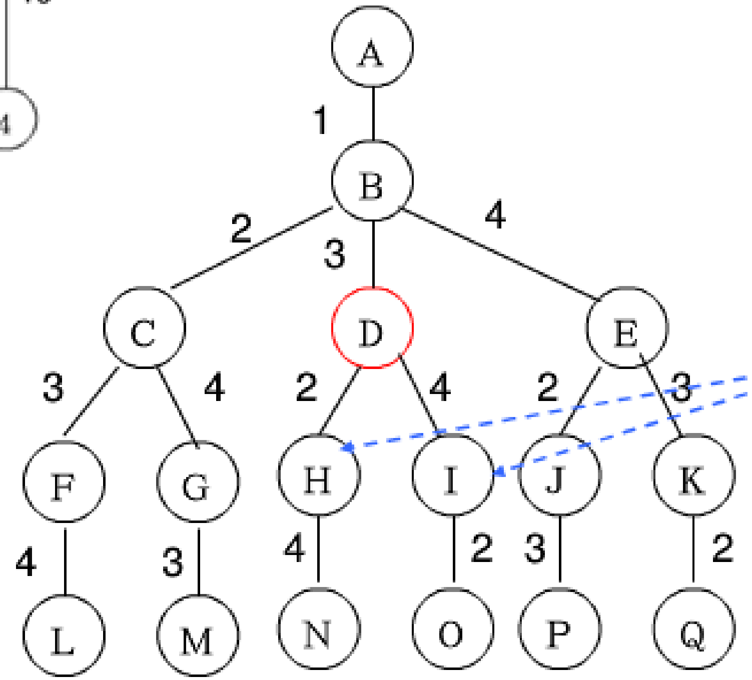
两个可行的子结点，被加入到极小堆中。其中J的费用为14，K的费用为24。

——D在当前堆中费用最小，所以处于堆顶



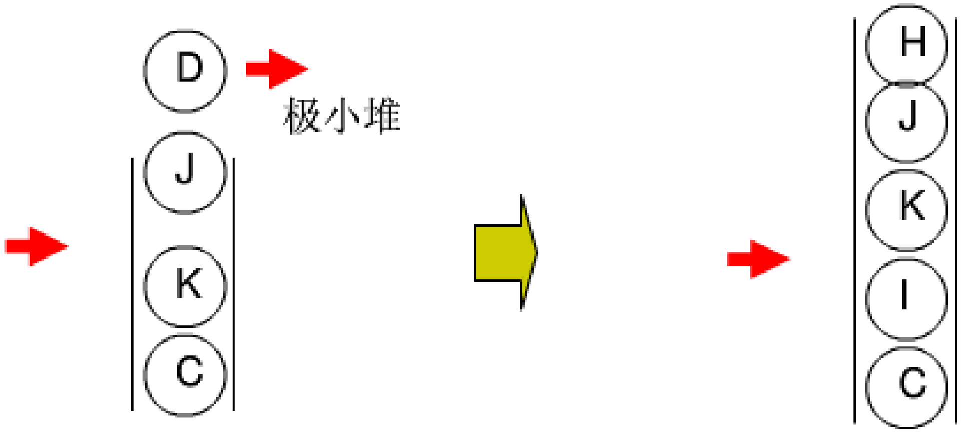
优先队列式分支限界法 ----TSP问题





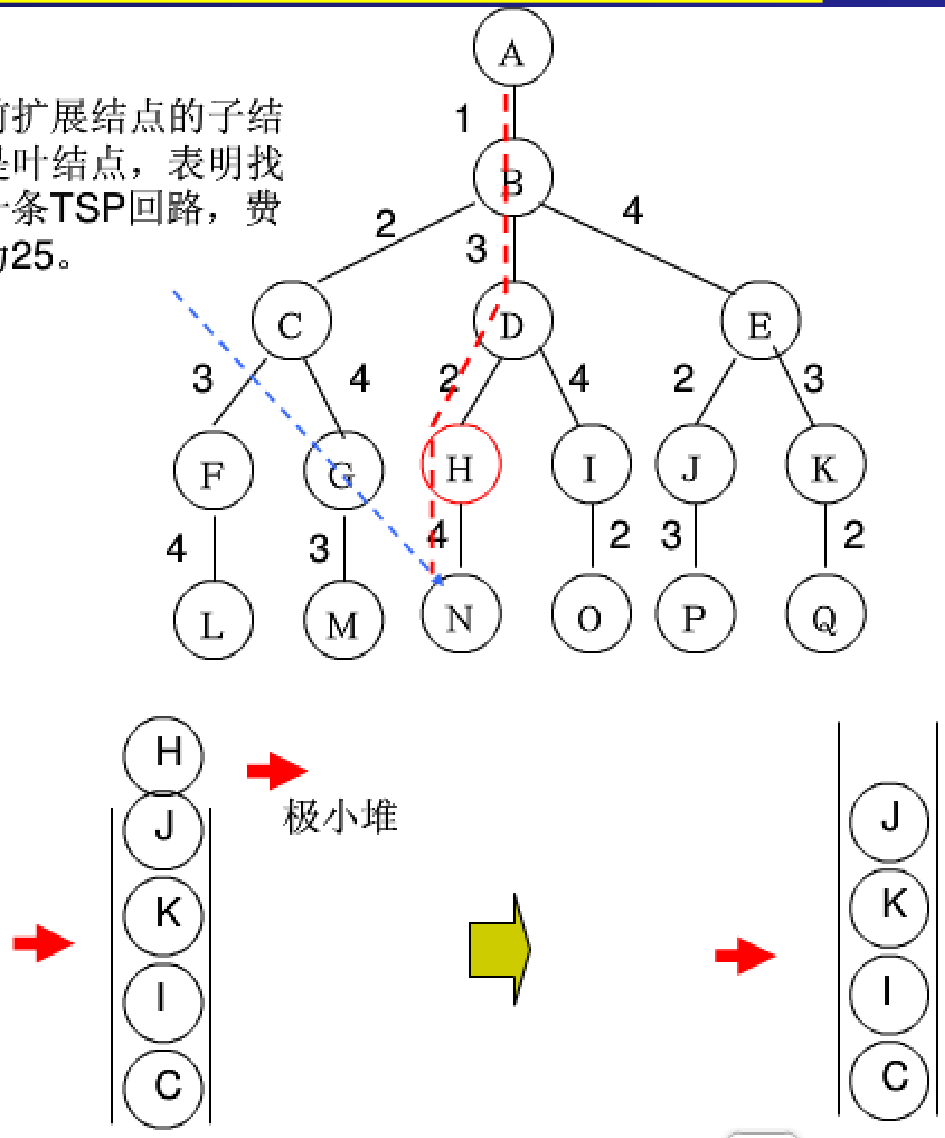
两个可行的子结点，被加入到极小堆中。其中H的费用为11，K的费用为26。

——H在当前堆中费用最小，所以处于堆顶



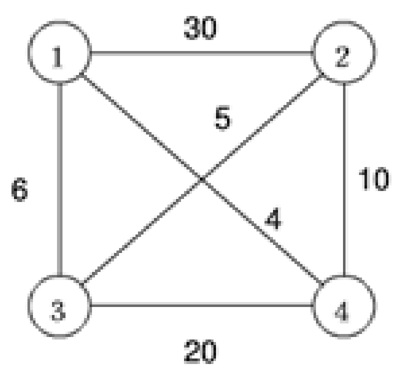
优先队列式分支限界法 ----TSP问题

当前扩展结点的子结点是叶结点，表明找到一条TSP回路，费用为25。

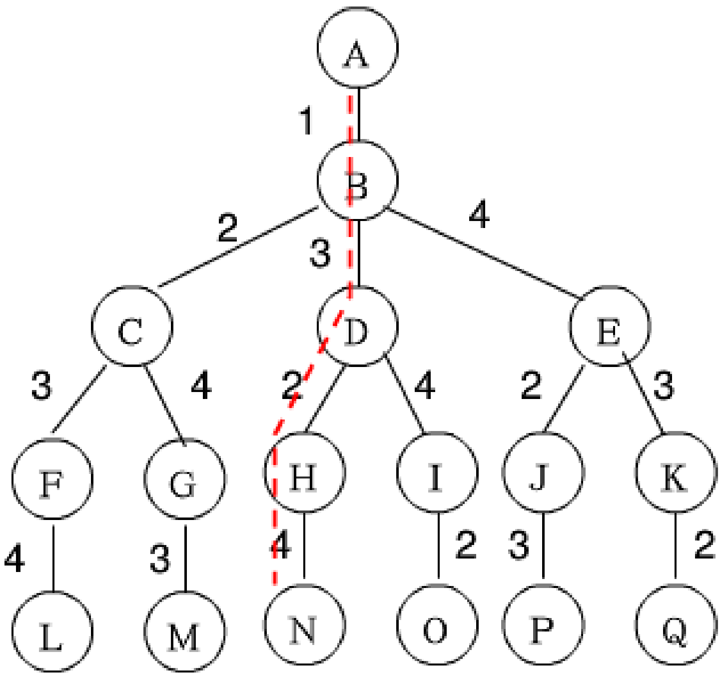


当前最优解为**13241**

费用：**25**



优先队列式分支限界法 ----TSP问题

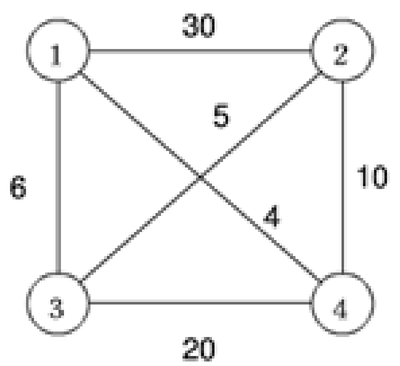


最优解为**13241**

费用：**25**

以此类推

极小堆



1.队列式搜索

- 注意：
  - 队列式分支法的搜索过程是对子集树进行盲目搜索，我们虽然不能将搜索算法改进为多项式复杂度，但若在在算法中加入了“限界”技巧，还是能降低算法的复杂度。
  - 一个简单的现象：
    - 若当前分支的“装载的价值上界”，比现有的最大装载的价值小，则该分支就无需继续搜索。

2.优先队列式搜索

- 优先队列式分支法：
  - 优先队列式扩展方式，若不加入限界策略其实是无意义的。
    - 要说明解的最优性，不搜索完所有问题空间是不能下结论的，而要对问题空间进行完全搜索，考虑优先级也就没有意义了。
  - 事实上：
    - 优先队列式搜索通过结点的优先级，可以使搜索尽快朝着解空间树上有最优解的分支推进，这样当前最优解一定较接近真正的最优解。

- 优先队列式分支限界法
  - 将优先队列搜索得到当前最优解作为一个“界”，对上界（或下界）不可能达到（大于）这个界的分支则不去进行搜索，这样就缩小搜索范围，提高了搜索效率。
  - 这种搜索策略称为优先队列式分支限界法。

实例——8-拼块游戏问题

输入：具有8个编号小方块的魔方

输出：移动系列，经过这些移动，魔方达到目标状态

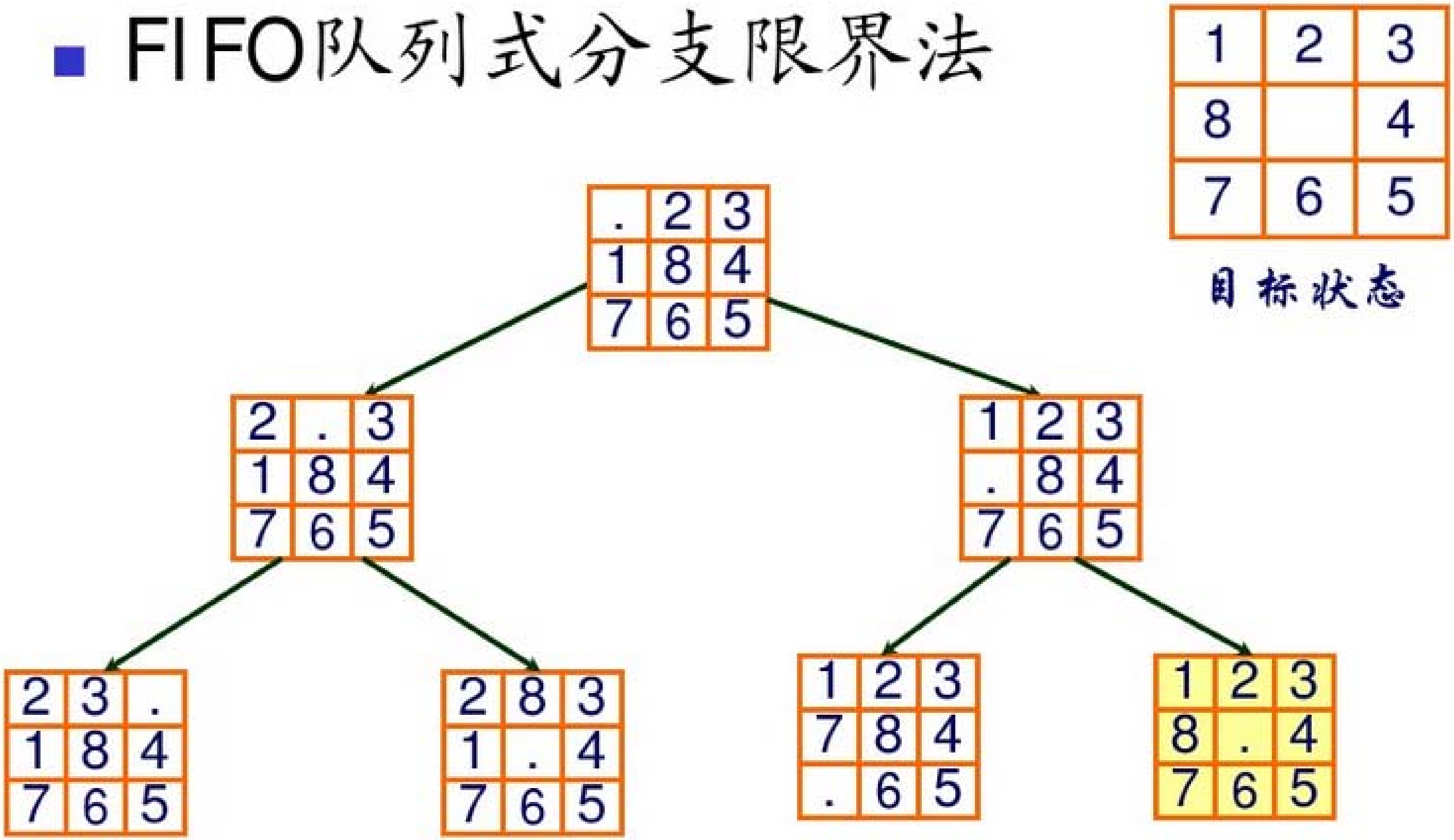
2	8	3
1	6	4
7		5

初始状态

1	2	3
8		4
7	6	5

目标状态

- FIFO队列式分支限界法



- 优先队列式分支限界法
  - “有智能”的排序函数C( )—最小代价估计函数:
  - C’(X) = 从初始状态到X所移动的次数 + 还未到位的数字方块数。
  - 从初始状态到X所移动的次数是实际耗费的代价，还未到位的数字方块数表示至少还要移动的次数。

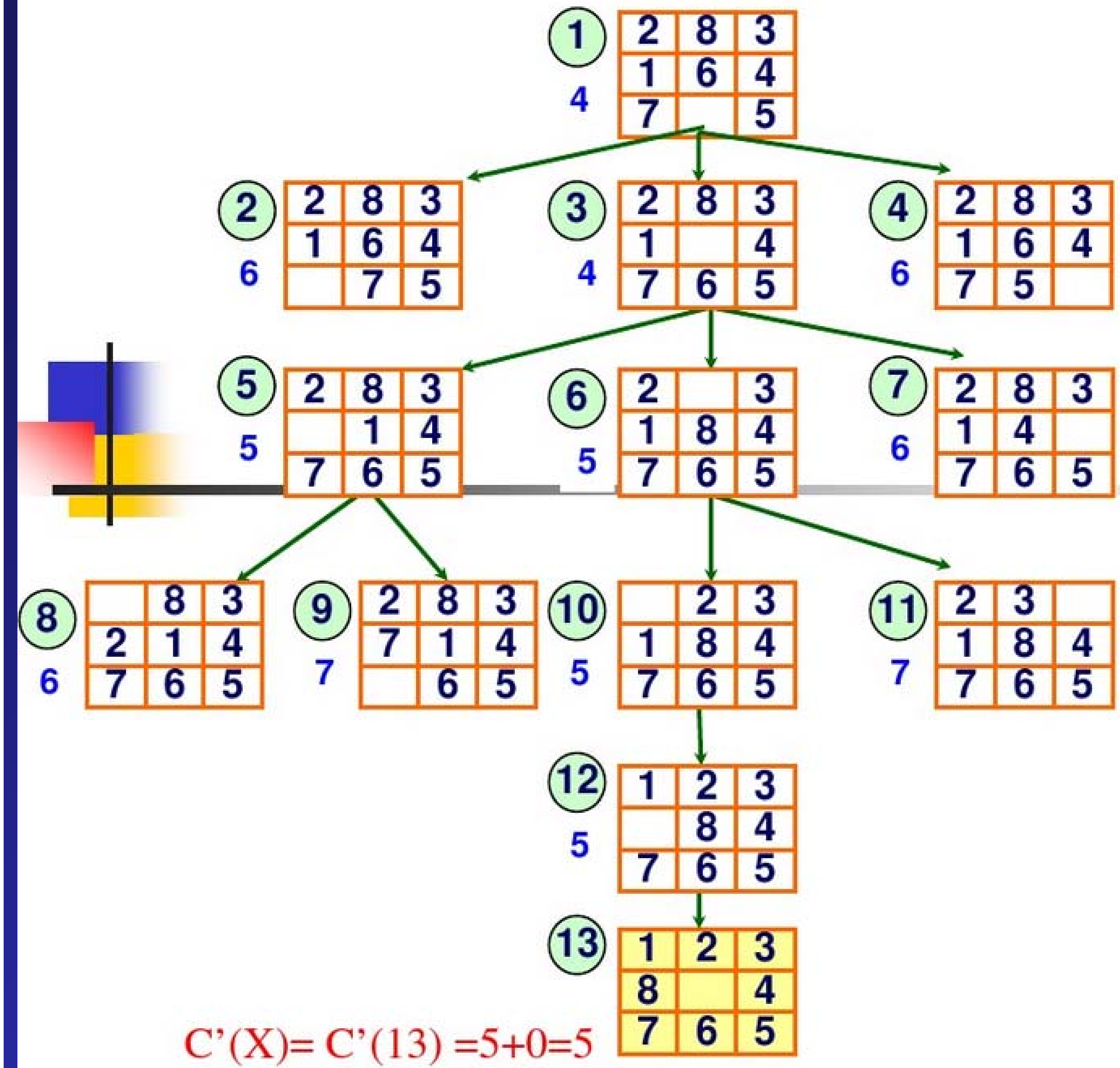
2	8	3
1	6	4
7		5

初始状态

→

1	2	3
8		4
7	6	5

目标状态



C'(X)= C'(13) =5+0=5

优先队列

L=(1)

L=(3,2,4)

L=(5,6,2,4,7)

L=(6,2,4,7,8,9)

L=(10,2,4,7,8,9,11)

L=(12,2,4,7,8,9,11)

L=(2,4,7,8,9,11)

第六章 分支限界法----基本思想

不同点	回溯法	分支限界法
求解目标	找出树中满足约束条件的所有解	找出满足约束条件的一个解或找出使目标函数达到极大(小)的最优解
搜索方式	深度优先	广度优先或最小耗费优先
扩展结点	多次机会成为扩展结点:扩展结点变为活结点后又可成为扩展结点	每个活结点只有一次机会成为扩展结点
树结点的生成顺序	生成最近一个有希望结点的单个子女	选择其中最有希望的结点,并生成它的所有子女
行进方向	随机性	方向性:活结点表,搜索朝着解空间树上有最优解的分支推进

6.3 装载问题

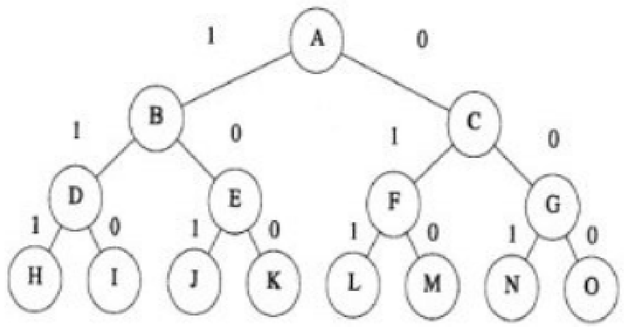
问题描述

- 有两艘船，n 个货箱。
- 第一艘船的载重量是c_1，第二艘船的载重量是c_2，w_i是货箱i 的重量，且
- $w_1+w_2+......+w_n \le c_1+c_2$。
- 我们希望确定是否有一种可将所有n 个货箱全部装船的方法。若有的话，找出该方法。

分析

- 解空间：子集树空间

- $W=\{10, 30, 50\}$, $C_1=60$,

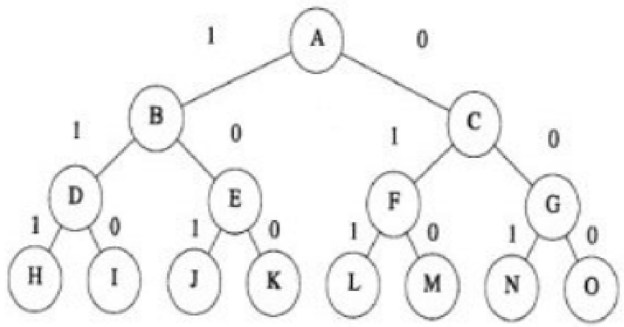


6.3 装载问题

- 队列式分支限界法的搜索过程为：

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，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，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，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，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，叶结点N、O都被剪掉；此时队列空算法结束。

- $W=\{10,30,50\}$，$C_1=60$，



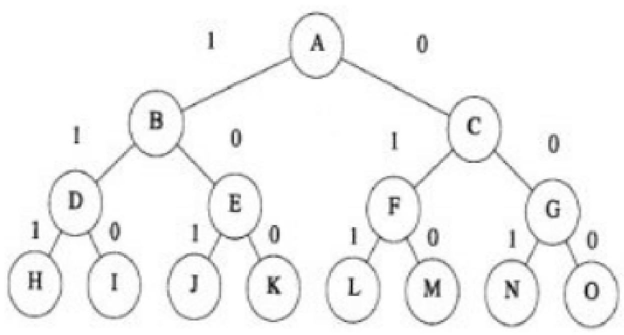
6.3 装载问题

- 优先级队列式分支限界法的搜索的过程：

1. 初始队列中只有结点A；
2. 结点A变为E-结点扩充B入堆，bestw=10；
3. 结点C的装载上界为30+50=80> bestw，也入堆；堆中B上界为90在优先队列首。
4. 结点B变为E-结点扩充D入堆，bestw=40；
5. 结点E的装载上界为60> bestw，也入堆；此时堆中D上界为90为优先队列首。
6. 结点D变为E-结点，叶结点H超过容量，叶结点I的装载为40入堆， bestw仍为40；此时堆中C上界为80为优先队列首。

6.3 装载问题

- $W=\{10, 30, 50\}$，$C_1=60$，



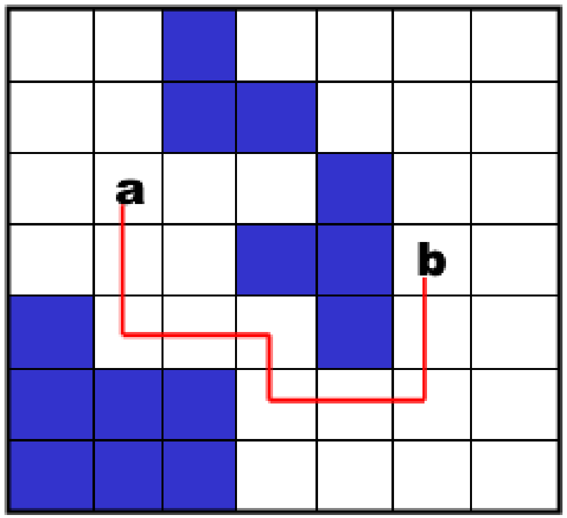
5. 结点C变为E-结点扩充F入堆，bestw仍为40；
6. 结点G的装载上界为50> bestw，也入堆；此时堆中E上界为60为优先队列首。
7. 结点E变为E-结点，叶结点J装载量为60入队，bestw变为60；
8. 叶结点K上界为10<bestw被剪掉；此时堆中J上界为60为优先队列首。
9. 结点J变为E-结点，扩展的层次为4算法结束。

   虽然此时堆并不空，但可以确定已找到了最优解。

6.4 布线问题

一、问题描述

- 印刷电路板将布线区域划分为n×m个方格阵列，如图所示。
- 精确的电路板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。
- 布线时电路只能沿直线或直角布线。
- 为避免线路相交，已布线方格做上封闭标记，其他线路布线不允许穿过封闭区域。

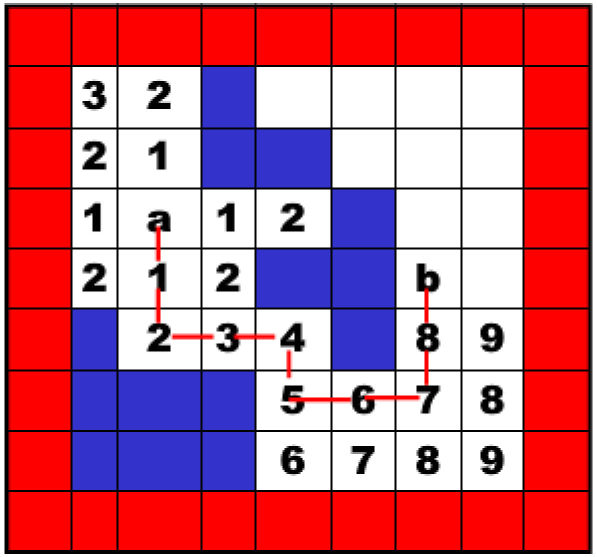


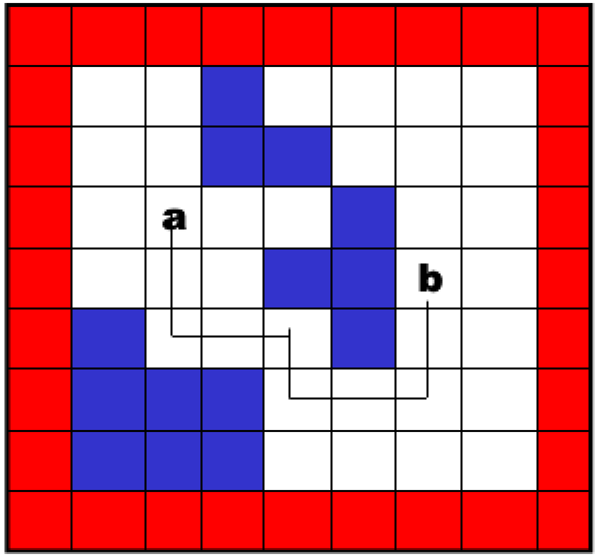
6.4 布线问题

二、算法思想

- 解此问题的队列式分支限界法从起始位置a开始将它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中，并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。
- 接着，算法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。这个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。即加入剪枝的广度优先搜索。

布线问题





现在来看上面两张图。演示了分支限界法的过程。最开始，队列中的活结点为标1的格子，随后经过一个轮次，活结点变为标2的格子，以此类推，一旦b方格成为活节点便表示找到了最优方案。为什么这条路径一定就是最短的呢？这是由于我们这个搜索过程的特点所决定的，假设存在一条由a至b的更短的路径，b结点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。值得一提的是，在搜索的过程中我们虽然可以知道结点距起点的路径长度，却无法直接获得具体的路径描述。为了构造出具体的路径，我们需要从目标方格开始向起始方格回溯，逐步构造出最优解，也就是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。

6.4 布线问题

- 布线问题不适合使用回溯法

回溯法的搜索是依据深度优先的原则进行的，如果我们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，但是我们最开始根本无法判断正确的路径方向是什么，这就造成了搜索的盲目和浪费。更为致命的是，即使我们搜索到了一条由a至b的路径，我们根本无法保证它就是所有路径中最短的，这要求我们必须把整个区域的所有路径逐一搜索后才能得到最优解。正因为如此，布线问题不适合用回溯法解决。

6.4 布线问题

在实现上述算法时，首先定义一个表示电路板上方格位置的类**Position**，它的**2**个私有成员**row**和**col**分别表示方格所在的行和列。在电路板的任何一个方格处，布线可沿右、下、左、上四个方向进行。沿这**4**个方向的移动分别记为移动**0,1,2,3**。在下面的表格中，**offset[i].row**和**offset[i].col(i=0,1,2,3)**分别给出沿这**4**个方向前进**1**步相对于当前方格的相对位置。

6.4 布线问题

移动方向的相对位移

移动i	方向	offset[i].row	offset[i].col
0	右	0	1
1	下	1	0
2	左	0	-1
3	上	-1	0

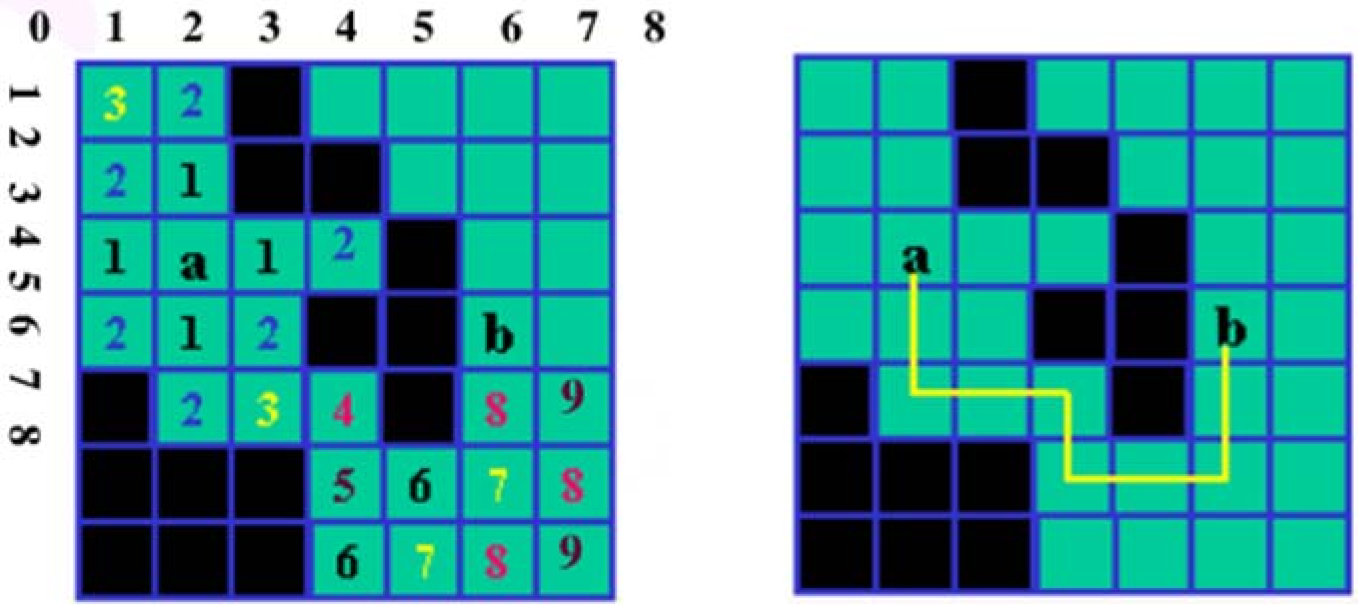
在实现上述算法时，用一个二维数组grid表示所给的方格阵列。

$$\text{grid[i][j]}=\begin{cases}0 & \text{方格[i][j]允许布线}\\ 1 & \text{方格[i][j]不允许布线}\end{cases}$$

算法将起始位置的距离标记为2，因为0，1用于表示方格的开放或封锁状态，实际距离应为标记距离减2。算法从起始位置start开始，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，直到达到目标方格finish或活结点队列为空时为止。

6.4 布线问题

对于前面的例子，计算过程过程和结果如下：



由于每个方格成为活结点进入活结点队列最多1次，因此活结点队列中最多只处理O(mn)。每个扩展结点需O(1)时间，因此算法共耗时O(mn)。构造相应的最短距离需要O(L)时间，其中L是最短布线路径的长度。

6.4 布线问题

定义移动方向的相对位移

设置边界的围墙

```
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
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
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);}
  }
```

第六章 分支限界法

本章小结

- 应用分支限界法的关键问题
  - （1）如何确定合适的限界函数
  - （2）如何组织待处理活结点表
  - （3）如何确定最优解中的各个分量

本章小结

分支限界法

- 对问题的解空间树中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向上回溯，因此，当搜索到某个叶子结点且该叶子结点的目标函数值在活结点表中最大时（假设求解最大化问题），求得了问题的最优值。
- 但是，却无法求得该叶子结点对应的最优解中的各个分量。这个问题可以用如下方法解决：
  - ① 对每个扩展结点保存该结点到根结点的路径；
  - ② 在搜索过程中构建搜索经过的树结构，在求得最优解时，从叶子结点不断回溯到根结点，以确定最优解中的各个分量。

最优化问题的求解策略比较

- 动态规划与搜索算法
  - (1)撇开时空效率的因素不谈，在解决最优化问题的算法中，搜索可以说是“万能”的，所以动态规划可以解决的问题，搜索也一定可以解决。
  - (2)动态规划要求阶段决策具有无后向性，而搜索算法没有此限制。
  - (3)动态规划通常是自底向上的递推求解；而无论深度优先搜索或广度优先搜索都是自顶向下求解。

动态规划与搜索算法

- (4)问题解空间的构造
  - 利用动态规划法进行算法设计时，设计者在进行算法设计前已经用大脑自己构造好了问题的解空间，因此可以自底向上的递推求解；
  - 而搜索算法是在搜索过程中根据一定规则自动构造，并搜索解空间树。
  - 由于在很多情况下，问题的解空间太复杂用大脑构造有一定困难，仍然需要采用搜索算法。
- (5)动态规划在递推求解过程中，需要用数组存储有关信息，而数组的下标只能是整数，所以要求问题中相关的数据必须为整数。

第六章 分支限界法

最优化问题的求解策略比较

- 动态规划与搜索算法
  - 动态规划算法在时间效率上的优势是搜索法无法比拟的，但动态规划总要遍历所有的状态，
  - 而搜索可以排除一些无效状态，更重要的是搜索还可以剪枝，可能剪去大量不必要的状态，因此在空间开销上往往比动态规划要低很多。
- 思考
  - 如何协调好高效率与高消费之间的矛盾呢？
- 一种折衷的办法
  - 记忆化搜索算法