第6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；

6.1 分支限界法的基本思想

◆类似于回溯法，分支限界法也是一种在问题的解空间树T上搜索问题解的算法。

分支限界法与回溯法

（1）求解目标：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，而分支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在某种意义下的最优解。

（2）搜索方式：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树。

- 分支限界法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。

- 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。

- 儿子结点中，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中。

- 从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

分支限界法解题步骤

- 在问题的边带权的解空间树中进行广度优先搜索
- 找一个叶结点使其对应路径的权最小(最大)
- 当搜索到达一个扩展结点时,一次性扩展它的所有儿子
- 将满足约束条件且最小耗费函数≤目标函数限界的儿子,插入活结点表中
- 从活结点表中取下一结点同样扩展
- 直到找到所需的解或活动结点表为空为止

分支限界法与回溯法的差别

- 分支限界法不仅通过约束条件，而且可通过目标函数的限界来减少无效搜索.
- 回溯法是深度优先搜索，而分支限界法是广度优先搜索.
- 采用广度优先搜索策略的目的是: **尽早发现剪枝点**.

常见的两种分支限界法

（1）队列式(FIFO)分支限界法

将活结点表组织成一个队列，按照先进先出（FIFO）原则选取下一个结点为扩展结点。

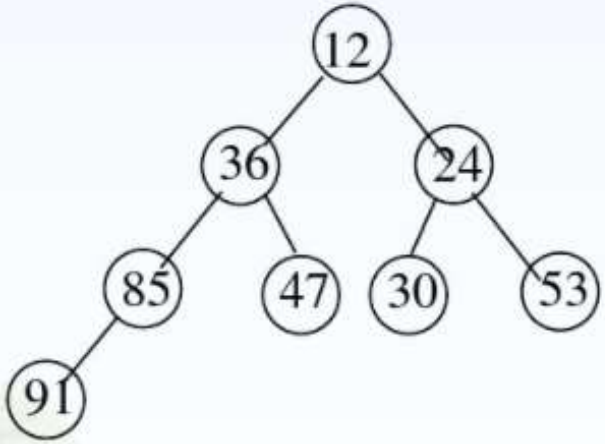
（2）优先队列式分支限界法

将活结点表组织成一个优先队列，按照规定的优先级选取优先级最高的结点成为当前扩展结点。

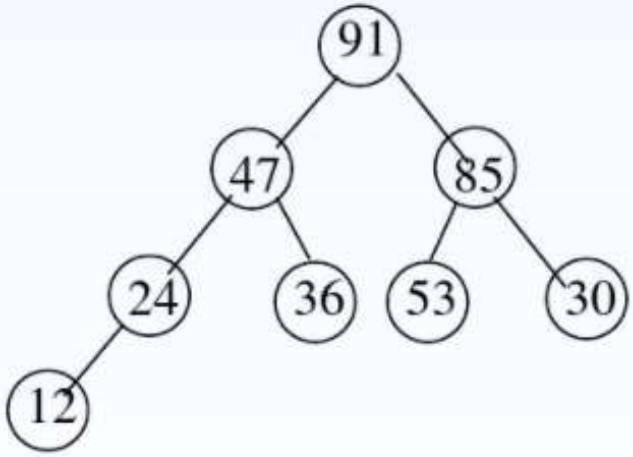
- 算法实现时，**通常用极大（小）堆来实现最大（小）优先队列**，提取堆中下一个结点为当前扩展结点，体现最大（小）费用优先的原则。
- 极大堆满足**一个节点必定不小于其子节点**，极小堆正好相反。
- 极大堆中**最大的元素必定是其根节点**，堆排序算法正是根据这个特性而产生的：**对一个序列，将其构造为极大堆，然后将根节点（数组首元素）和数组的尾元素交换，之后对除了尾元素之外的序列继续以上过程，直到排序完成**。
- （堆的一些简单介绍见后页：）

堆的定义

- n个元素的序列{$k_1,k_2,\ldots,k_n$}当且仅当满足如下关系时，称之为堆（heap）。
- n个元素的序列 a_1，a_2，…，a_n，当且仅当满足下述关系时，称之为堆。
- ①$a_i \le a_{2i}$，$a_i \le a_{2i+1}$：小根堆，堆顶为序列的最小值；
- 或：②$a_i \ge a_{2i}$，$a_i \ge a_{2i+1}$：大根堆，堆顶为序列的最大值。



小根堆

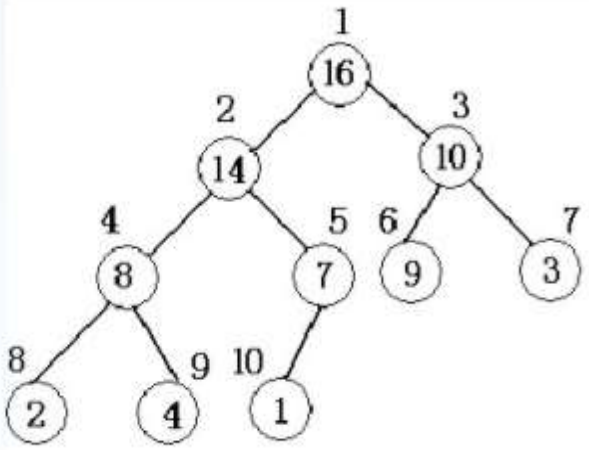


大根堆

堆结构及其应用

堆结构

堆结构是一种数组对象，它可以被视为一棵完全二叉树，树中每个结点与数组中存放该结点中值的那个元素相对应，如下图：



(a)

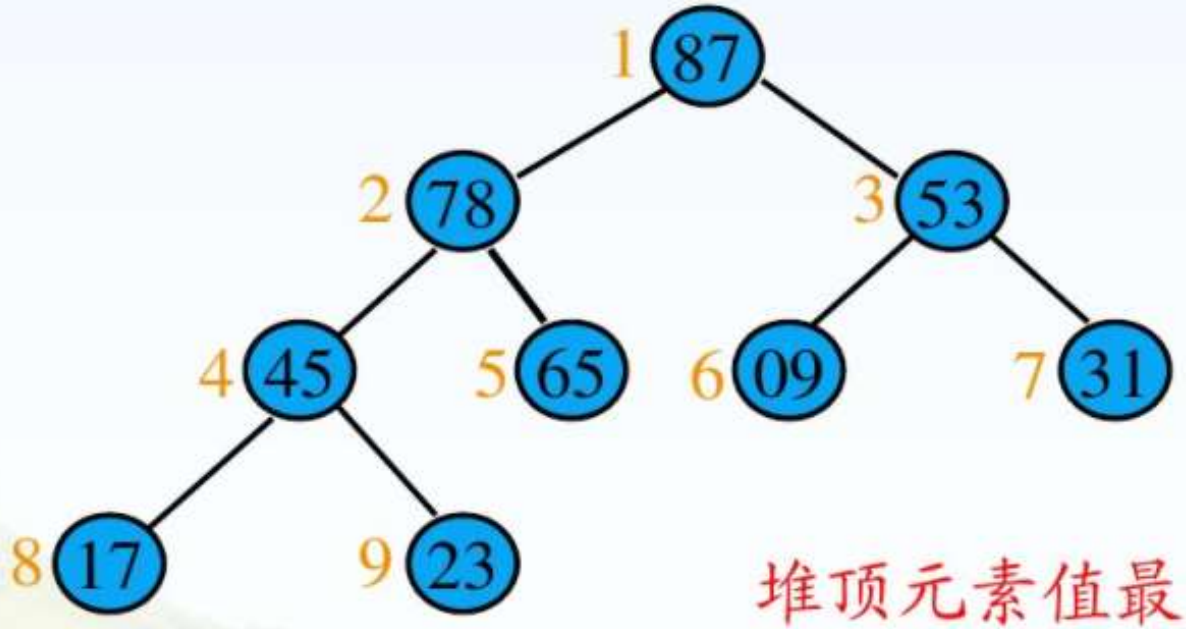
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

(b)

左边的图（a）是一棵典型的完全二叉树，结点上方为编号，结点的值在圆圈当中。右边的图(b)是我们非常熟悉的一维数组，当又不是一般意义上的数组，因为这个数组存储了左边的二叉树结构。

大根堆示例

1	2	3	4	5	6	7	8	9
87	78	53	45	65	09	31	17	23

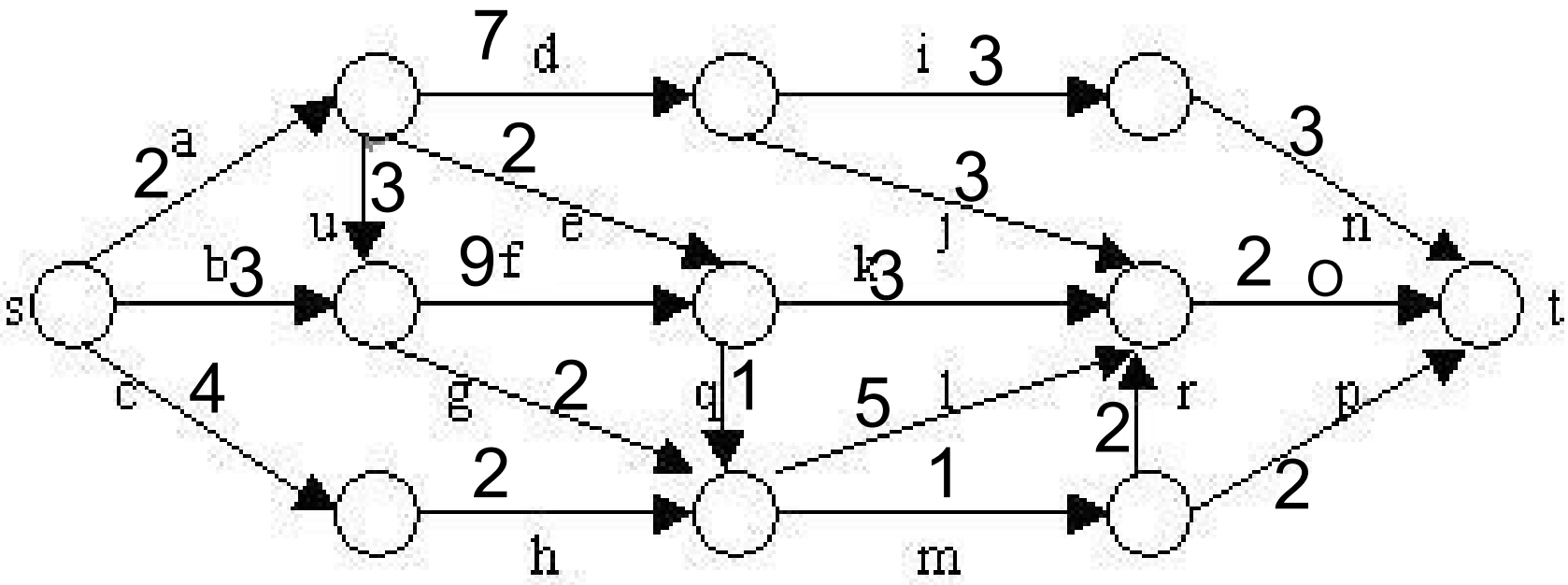


堆顶元素值最大

6.2 单源最短路径问题

1. 问题描述

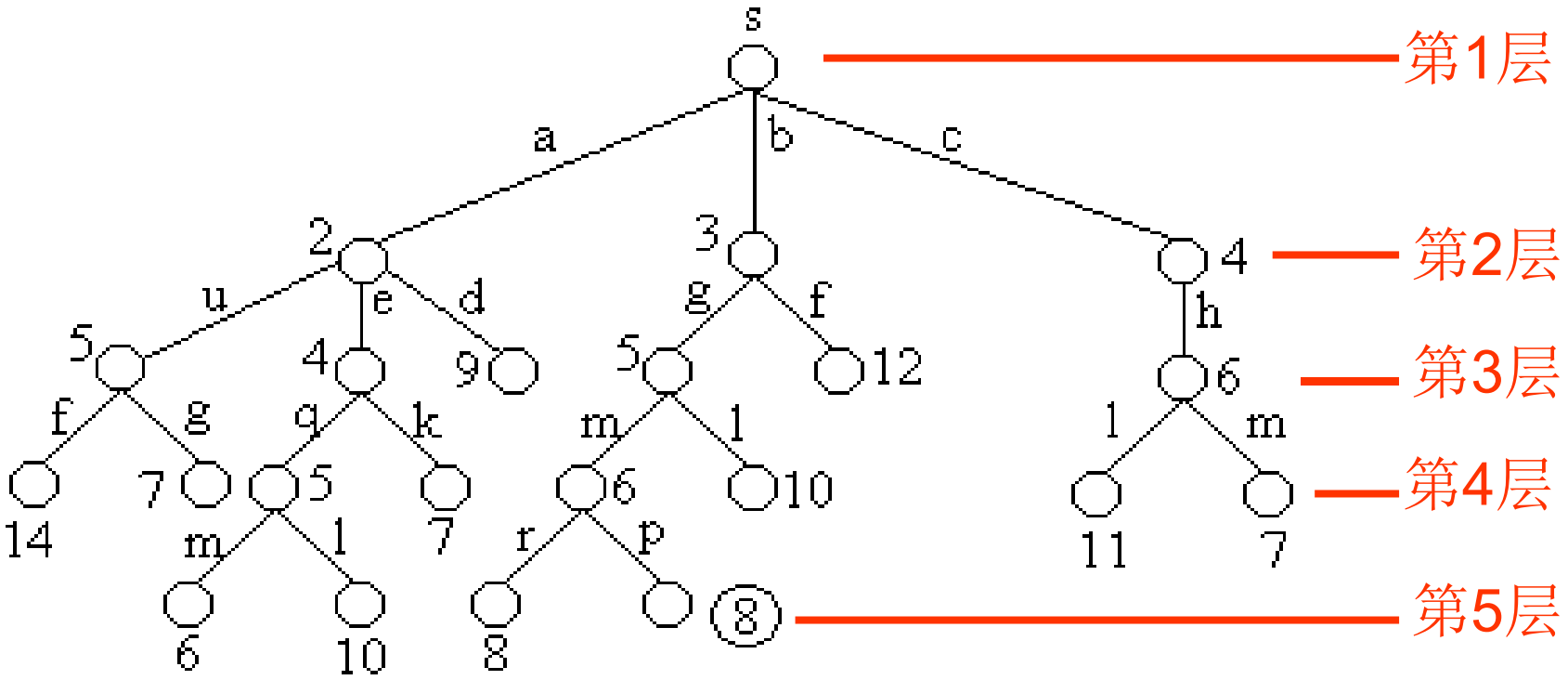
下面以一个例子来说明单源最短路径问题：在下图所给的有向图G中，每一边都有一个非负边权。要求图G的从源顶点s到目标顶点t之间的最短路径。



6.2 单源最短路径问题

1. 问题描述

下图是用优先队列式分支限界法解有向图G的单源最短路径问题产生的解空间树。其中，每一个结点旁边的数字表示该结点所对应的当前路长。



2. 算法思想

- 解单源最短路径问题的优先队列式分支限界法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。

- 算法从图G的源顶点s和空优先队列开始。结点s被扩展后，它的儿子结点被依次插入堆中。

- 算法每次从堆中取出具有最小当前路长的结点作为当前扩展结点，并依次检查与当前扩展结点相邻的所有顶点。

- 如果从当前扩展结点i到j有边可达，且从源出发，途经i再到j的所相应路径长度，小于当前最优路径长度，则将该顶点作为活结点插入到活结点优先队列中。

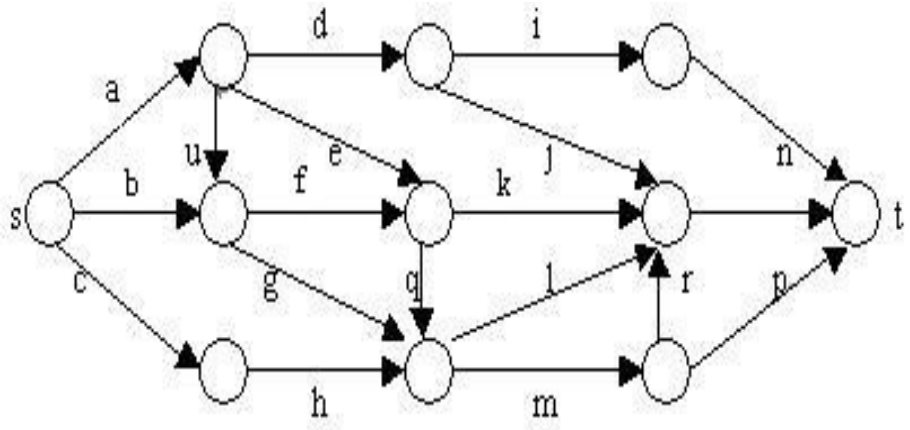
- 结点扩展过程一直继续到活结点优先队列为空时为止

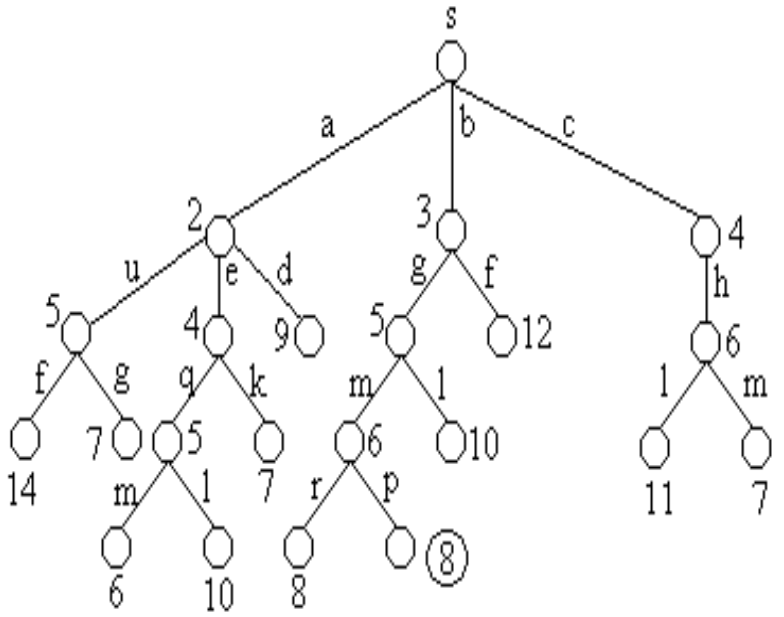
3. 剪枝策略

- 扩展结点过程中，一旦发现一个结点的下界不小于当前找到的最短路长，则算法剪去以该结点为根的子树。

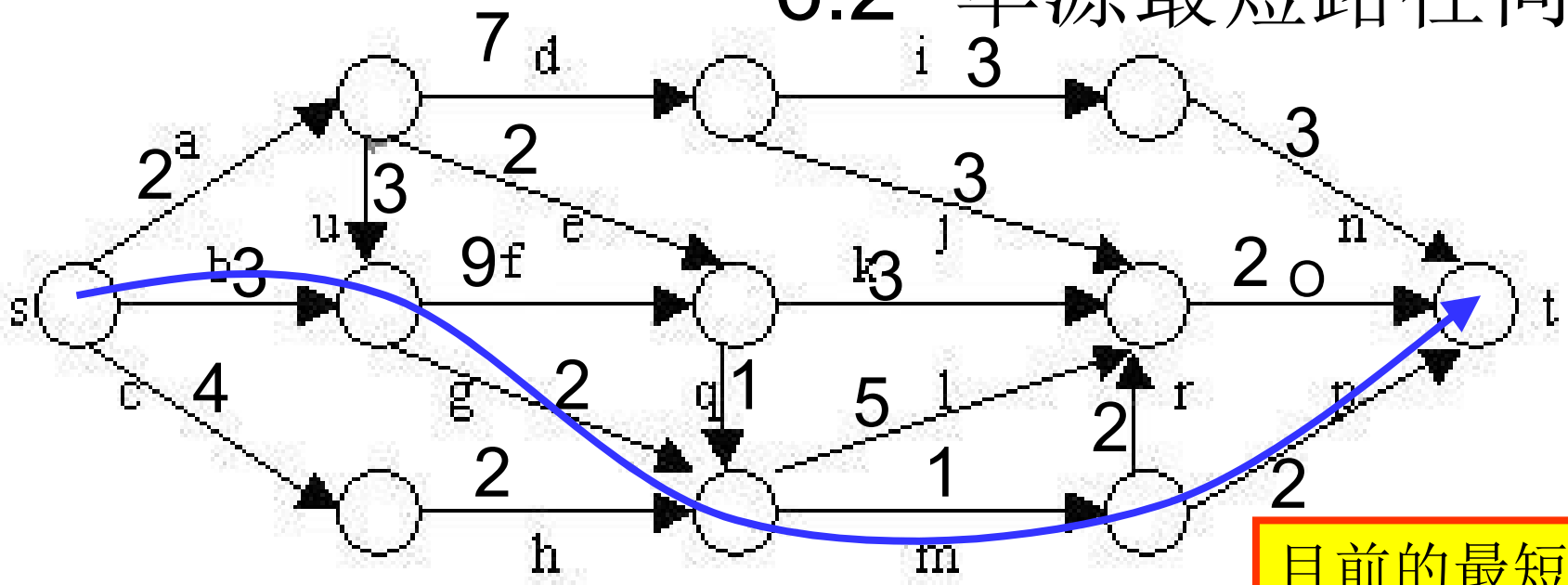
- 利用结点间的控制关系进行剪枝。若从源顶点s出发，有2条不同路径到达图G的同一顶点。可将路长长的路径所对应的树中的结点为根的子树剪去。

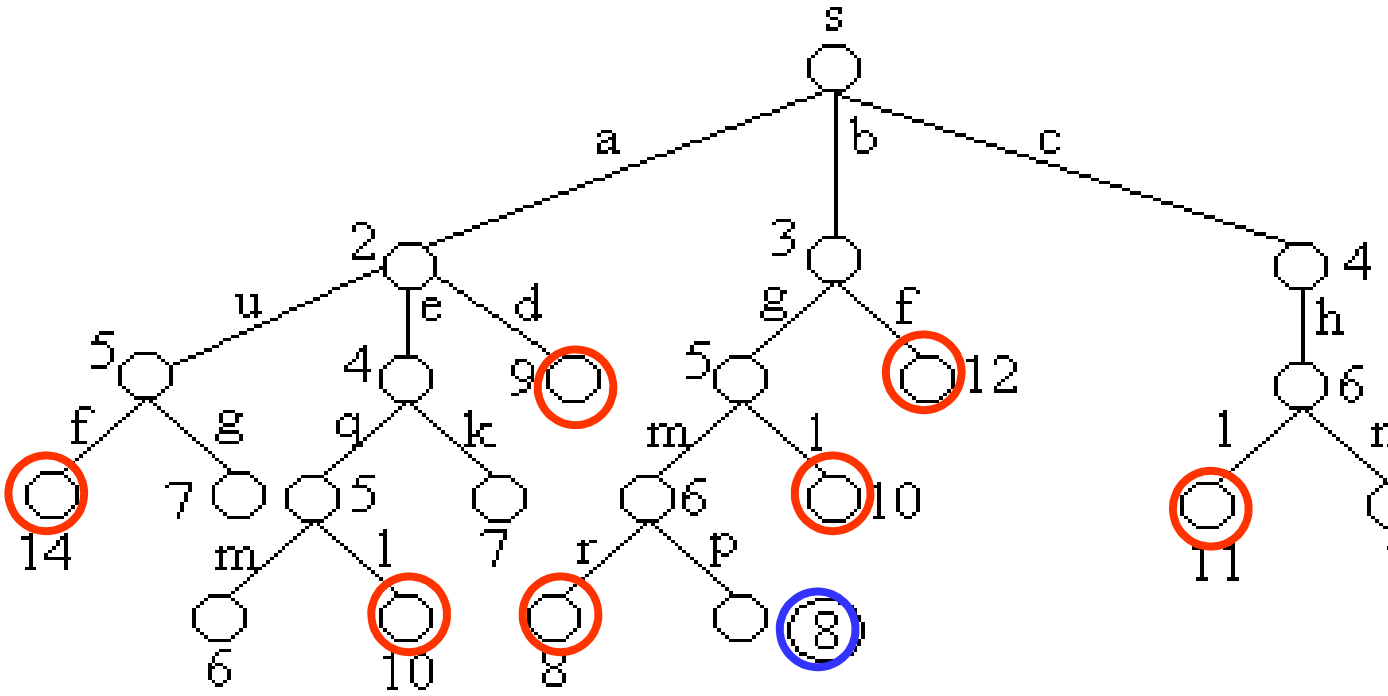
- 例如，例中从s出发经边a、e、q（路长5）和经c、h（路长6）的两条路径到达G的同一顶点。
- 但在解空间树中，这两条路径相应于解空间树的2个不同的结点A和B。
- 由于A的路长较小，故可将以结点B为根的子树剪去。此时称**结点A控制了结点B**。





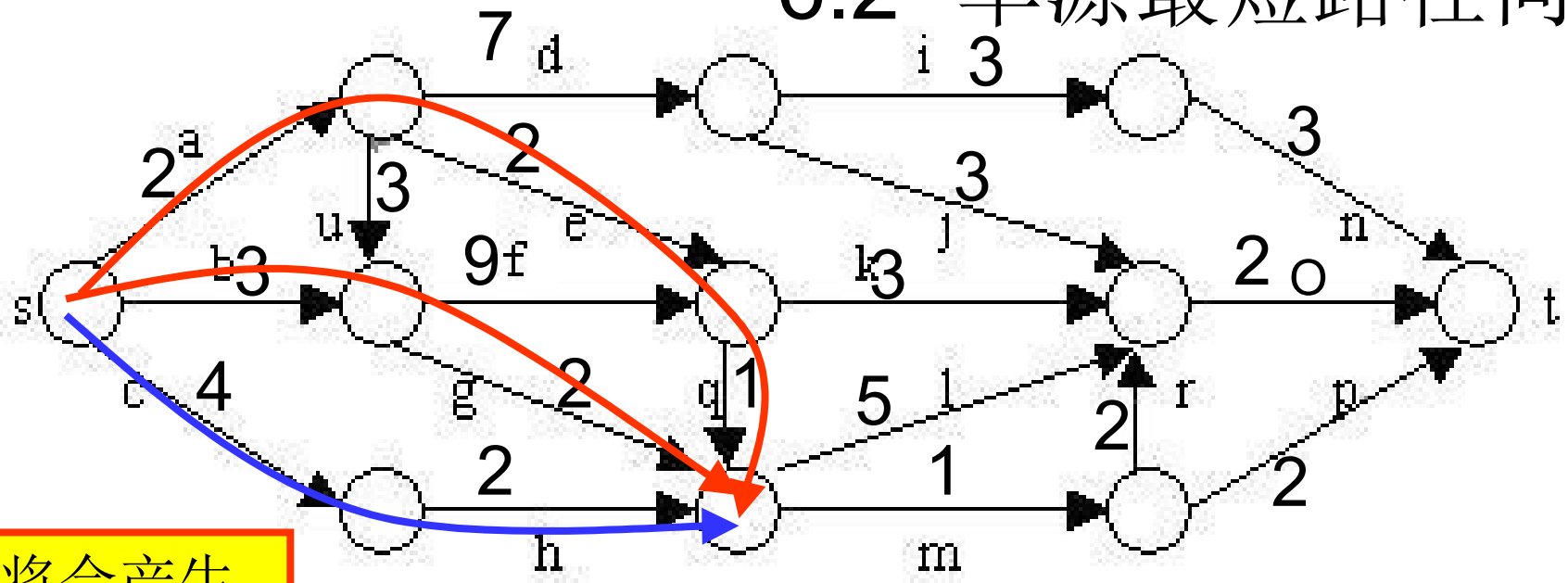
6.2 单源最短路径问题



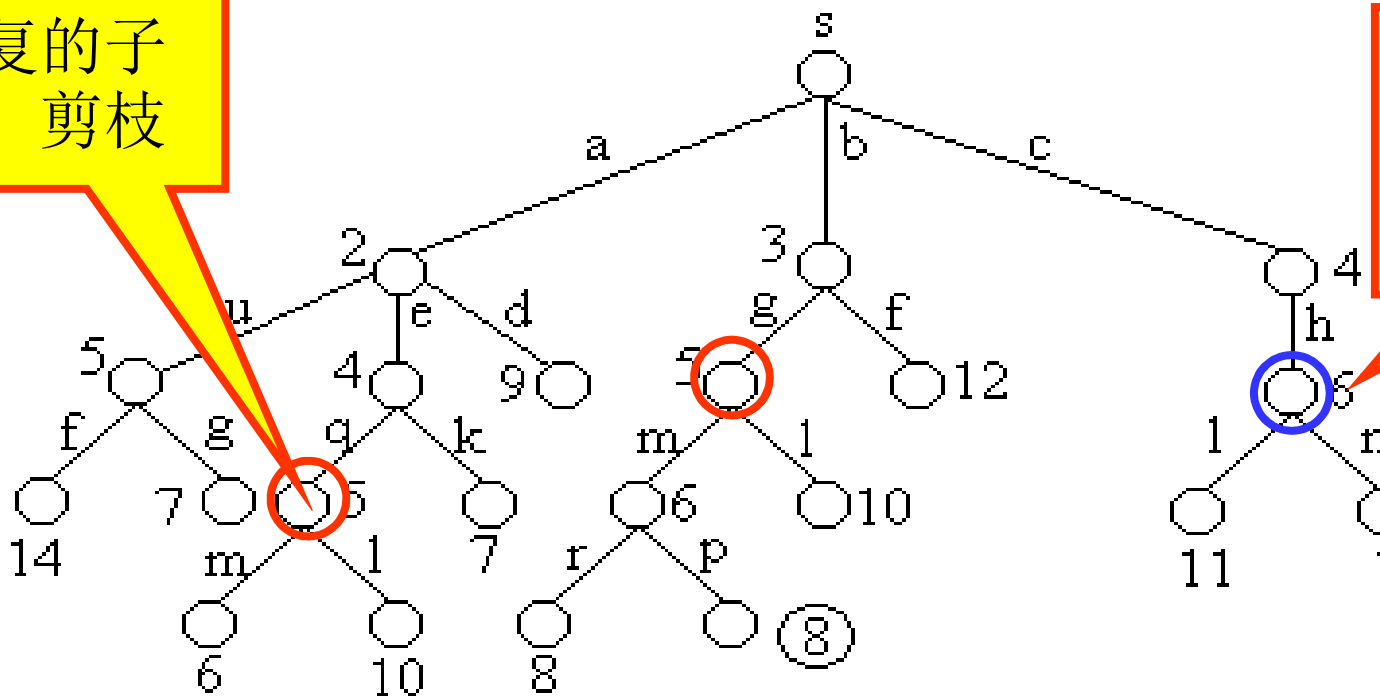


目前的最短路径是8，一旦发现某个节点的下界不小于这个最短路进，则剪枝

6.2 单源最短路径问题



将会产生重复的子树，剪枝



利用节点的控制关系剪枝

6.2 单源最短路径问题

2. 剪枝策略(回顾)

在算法扩展结点的过程中，一旦发现一个结点的下界不小于当前找到的最短路长，则算法剪去以该结点为根的子树。

在算法中，利用结点间的控制关系进行剪枝。从源顶点s出发，2条不同路径到达图G的同一顶点。由于两条路径的路长不同，因此可以将路长长的路径所对应的树中的结点为根的子树剪去。

6.2 单源最短路径问题

3. 算法思想（回顾）

解单源最短路径问题的优先队列式分支限界法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。

算法从图G的源顶点s和空优先队列开始。结点s被扩展后，它的儿子结点被依次插入堆中。此后，算法从堆中取出具有最小当前路长的结点作为当前扩展结点，并依次检查与当前扩展结点相邻的所有顶点。如果从当前扩展结点i到顶点j有边可达，且从源出发，途经顶点i再到顶点j的所相应的路径的长度小于当前最优路径长度，则将该顶点作为活结点插入到活结点优先队列中。这个结点的扩展过程一直继续到活结点优先队列为空时为止。

6.2 单源最短路径问题

```
while (true) {
    for (int j = 1; j <= n; j++)
        if ((a[enode.i][j]<Float.MAX_VALUE)&&(enode.length+a[enode.i][j]<dist[j])) {
            // 顶点i到顶点j可达，且满足控制约束
            dist[j]=enode.length+a[enode.i][
            p [j]=enode.i;
            HeapNode Node=new HeapNo
            Heap.put(node); // 加入活结点优先队
    }
}
```

记载最短路径

顶点i和j间有边，且此路径长小于原先从原点到j的路径长
这个判断，实现了剪枝

dist:最短距离数组
p: 前驱顶点数组
enode： 当前的扩展节
a: 邻接矩阵

6.3 装载问题

1. 问题描述

有一批共个集装箱要装上2艘载重量分别为C1和C2的轮船，其中集装箱i的重量为Wi，且

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

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。

6.3 装载问题

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。

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

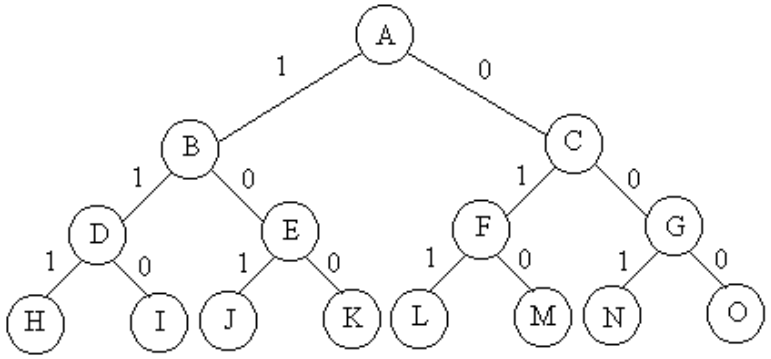
$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

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

例如：

$$W = <10,8,5>$$

$$C = 16$$



6.3 装载问题

2. 队列式分支限界法

在算法的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。

活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部标记-1，故在取队首元素时，活结点队列一定不空。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。

2. 队列式分支限界法

- 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。
- 函数MaxLoading具体实施对解空间的分支限界搜索p159
- 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。

2. 队列式分支限界法

```
while (true) {
    // 检查左儿子结点
    if (ew + w[i] <= c) // x[i] = 1
       enQueue(ew + w[i], i);
    // 右儿子结点总是可行的
    enQueue(ew, i); // x[i] = 0
    ew=((Integer) queue.remove()).intValue();     // 取下一扩展结点
    if (ew == -1) {      // 同层结点尾部
        if (queue.isEmpty()) return bestw;
        queue.put(new Integer(-1)); // 同层结点尾部标志
        ew=((Integer) queue.remove()).intValue();     // 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
queue:　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

◆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。**

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≤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。

3. 算法的改进

```
// 检查左儿子结点
  Type wt = Ew + w[i];   // 左儿子结点的重量
     if (wt <= c) {     // 可行结点
        if (wt > bestw) bestw = wt;
        // 加入活结点队列
        if (i < n) Q.Add(wt);
}
// 检查右儿子结点
     if (Ew + r > bestw && i < n)
         Q.Add(Ew);     // 可能含最优解
     Q.Delete(Ew);     // 取下一扩展结点
```

提前更新 bestw

右儿子剪枝

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
```

6.3 装载问题

4. 构造最优解

找到最优值后，可以根据parent回溯到根节点，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[i]取值为0/1，表示是否取该货物。

5. 优先队列式分支限界法

◆解装载问题的优先队列式分支限界法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再加上剩余集装箱的重量之和。

◆优先队列中优先级最大的活结点成为下一个扩展结点。

◆以结点x为根的子树中，所有结点相应路径的载重量不超过它的优先级。

◆子集树中叶结点所相应的载重量与其优先级相同。

- 在优先队列式分支限界法中，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。

- 有两种方式求得最优解：

（1）在优先队列的每一个活结点中，保存从解空间树的根结点到该活结点的路径，在算法确定了达到最优值的叶结点时，就在该叶结点处同时得到相应的最优解。

（2）在算法的搜索进程中，保存当前已构造出的部分解空间树，这样在算法确定了达到最优值的叶结点时，可以在解空间树中从该叶结点开始向根结点回溯，构造出相应的最优解。

6.4 布线问题

问题描述

- 印刷电路板将布线区域分成n×m个方格。
- 精确的电路布线问题: 要求确定连接方格a中点到方格b中点的最短布线方案。

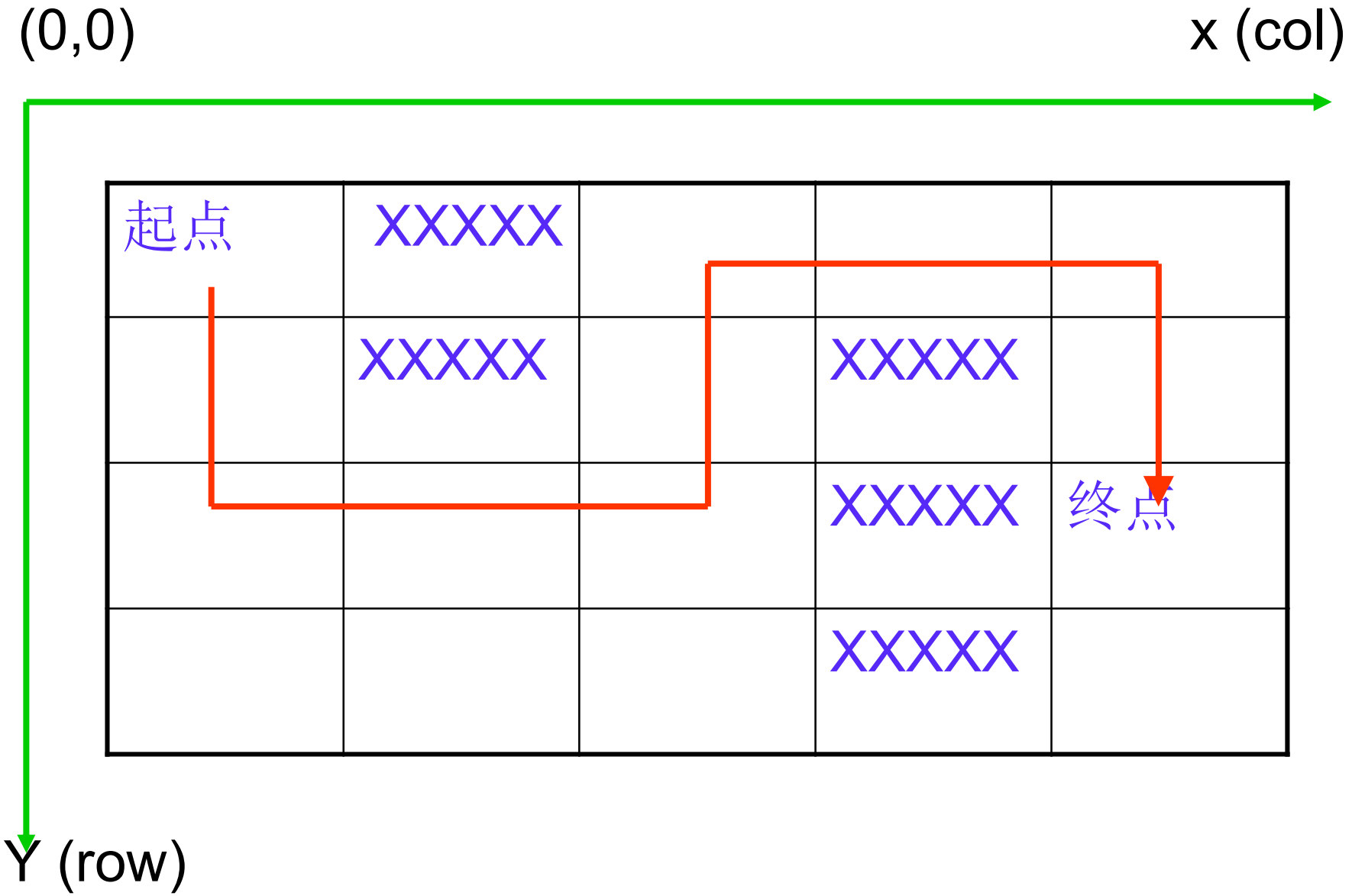
- 布线时，电路只能沿着直线或直角布线，见p167图6-3。

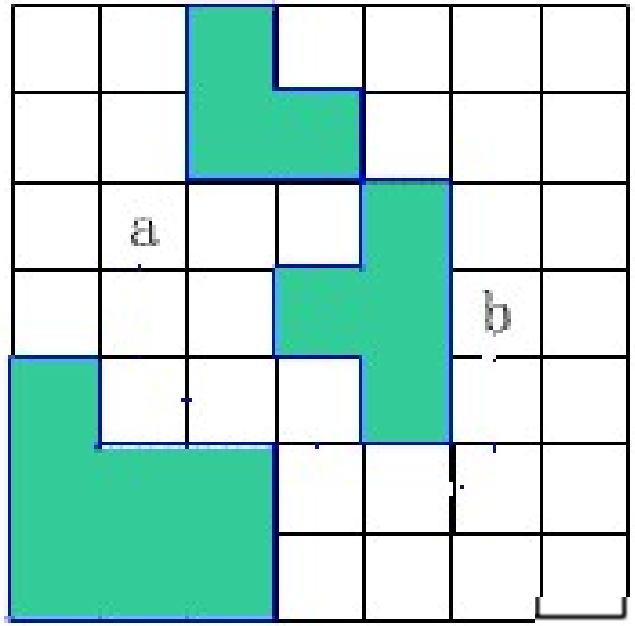
- 为了避免线路相交，已布线的方格做了封锁标记，其他线路不允许穿过被封锁的方格。

- 用队列式分支限界法解布线问题

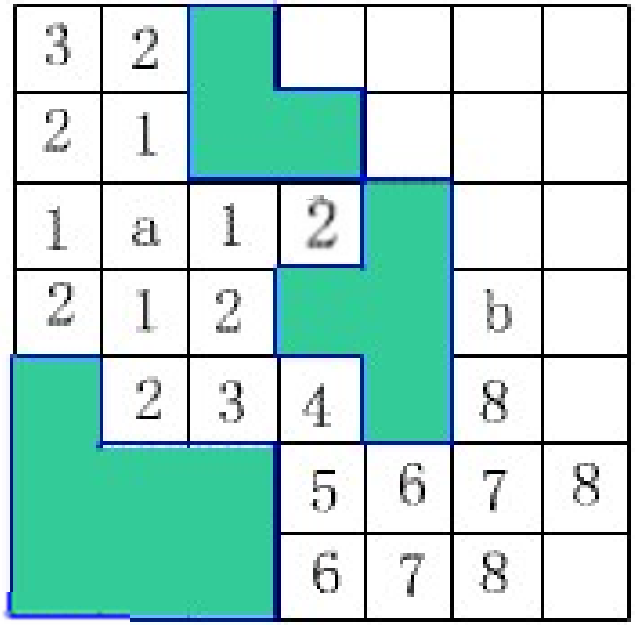
- 布线问题的解空间是一个图

6.4 布线问题

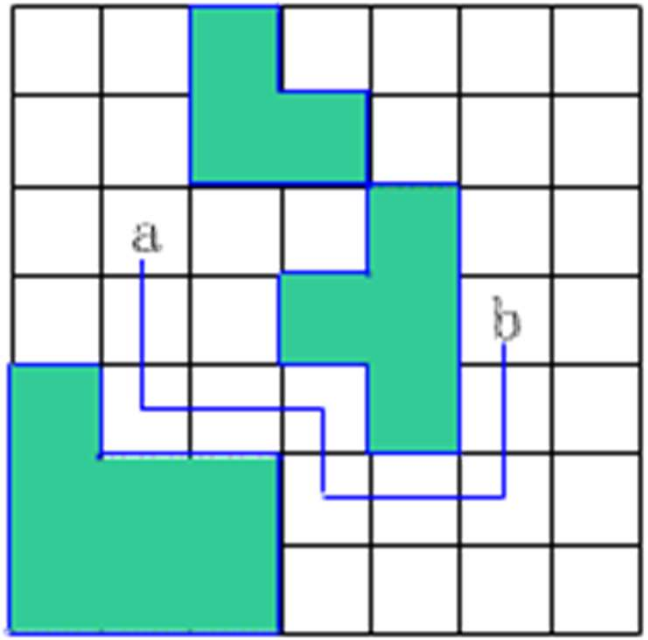




原图



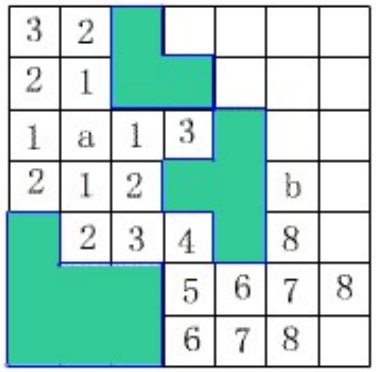
(a)标记距离



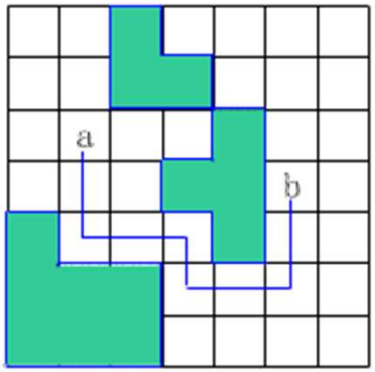
(b)最短布线路径

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。



(a)标记距离



(b)最短布线路径

- 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。

6.4 布线问题

```
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
```

定义移动方向的相对位移

设置边界的围墙

```
for (int i = 0; i <= m+1; i++)
     grid[0][i] = grid[n+1][i] = 1; // 顶部和底部
for (int i = 0; i <= n+1; i++)
     grid[i][0] = grid[i][m+1] = 1; // 左翼和右翼
```

6.4 布线问题

```
for (int i = 0; i < 4; i++) {   //一共有4种走的方向
        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;  // 向前走了一步
            if ((nbr.row == finish.row) &&
               (nbr.col == finish.col)) break; // 完成布线
            q.put(new Position(nbr.row,nbr.col));
        }
}
```

找到目标位置后，可以通过回溯方法找到这条最短路径。**怎么找？---找前驱位置，详见P169**

- 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。

6.5 0-1背包问题

问题描述

给定n种物品和一背包。物品i的重量是w_i，其价值为p_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1背包问题是一个特殊的整数规划问题。

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

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

$$W =< 10,8,5 >$$

例如：$P =< 5,4,1 >$

$$C = 16$$

最优解为：(1,0,1)

此时的价值为：6

6.5　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，**节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。**当扩展到叶节点时为问题的最优值。**

6.5　0-1背包问题

上界函数

```
b=cp;                                          // 初始化为目前背包的重量
// n表示物品总数，cleft为剩余空间
while (i <= n && w[i] <= cleft) {
        cleft -= w[i];                         //w[i]表示i所占空间
        b += p[i];                             //p[i]表示i的价值
        i++;
      }
if (i <= n) b += p[i] / w[i] * cleft;          // 装填剩余容量装满背包
return b;                                      // b为上界函数
```

6.5　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);

    // 取下一个扩展节点（略）

}
```

分支限界搜索过程

小结

- 分支限界法类似于回溯法，在问题的解空间树上搜索问题解的算法；
- 分支限界法的求解目标通常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。

第7章　概率算法

概率算法

- 数值概率算法
- 蒙特卡罗算法
- 拉斯维加斯算法
- 舍伍德算法

1.数值概率算法

- 数值概率算法常用于数值问题的求解。
- 此类算法所得到的解往往为近似解，且近似解的精度随计算时间的增加而不断提高。
- 再问题的精确解是不可能或无必要时，使用数值概率算法较好。

2.蒙特卡罗算法

- 蒙特卡罗算法用于求问题的准确解。
- 蒙特卡罗算法能球的问题的一个解，但不一定是正确解；
- 求得正确解的概率依赖于算法所用时间—算法所用时间越多，得到正确解得概率就越高。
- 一般情况下，无法有效得判定所得解是否正确。

3.拉斯维加斯算法

- 拉斯维加斯如果能获得解，一定是正确的解。（有可能无法获得解）
- 拉斯维加斯算法找到正确解的概率随它所用的计算时间的增加而提高。

4.舍伍德算法

- 舍伍德算法总能求得问题的一个解，且所求解总是正确的。
- 为确定性算法引入随机性，改造成一个舍伍德算法，消除或减少好坏实例间的差别。
- 舍伍德算法设法消除最坏情形行为与特定实例间的关联性。

7.1 随机数

随机数

在离散系统仿真中，随机数是一个必不可少的基本元素

（0,1）均匀分布随机数是产生其他许多分布的随机数的基础

一个随机数序列必须满足两个重要的统计性质：均匀性和独立性

随机数

随机数序列必须满足的统计性质

均匀性
如果将区间[0,1]分为n个等长的子区间，那么在每个区间的期望观察次数为N/n，其中N为观察的总次数

独立性
观察值落在某个特定区间的概率与以前的观察值无关

随机数

随机数的产生方法

（1）物理方法

利用某些物理过程来产生均匀分布的随机数。真正的随机数只能用随机物理过程来产生

（2）随机数表

利用物理过程得到的大量随机数，制成随机数表

（3）随机数产生程序

要实现严格数学意义上的随机数，在理论上是可行的，但在实际中却不可行，也没有必要。

按照一定的算法计算出具有类似于均匀分布随机变量的独立取样值性质的数——伪随机数

随机数

计算机产生随机数的要求

（1）产生的随机数要尽可能的逼近理想的均匀性和独立性统计性质

实际上在伪随机数的产生过程中，一定会出现一些与理想随机数背离的地方：可能不是均匀分布；可能是离散而不是连续的；平均值可能太大或太小；方差可能太大或太小；数字之间可能不是相互独立的

（2）产生的随机数要有足够长的周期

（3）产生随机数的速度要快，占用的内存空间要小

（4）随机数必须是可重复的：对于给定的起始点或初始条件，应当能够产生相同的随机数序列

在某些应用比如游戏中，不同的用户需要随机生成各自的场景，但同一用户再次登录时又需要场景与上次登录时相同（这样才可以继续游戏）。这种情形下就需要随机场景可重复——重复随机数。这种情况可以通过为每个用户设置不同的初始值而同一用户的初始值不变解决。

随机数

计算机产生随机数的方法一——平方取中法
第一个随机数生成器，由20世纪40年代冯。诺依曼提出
设有一个4位正整数X_0，平方后得到8位正整数（如果不够8位，在左侧补0凑够8位），之后取中间4位获得新的4位正整数X_1，...
举例：种子X_0=1234
1234,5227,3215,3362,3030,1809,2724,4201,6484...

缺点：种子选择很重要，否则很难保证有足够长的周期（例如X_0=100）而且容易出现退化现象（以后的随机数为同一常数或为零）
现已很少使用

随机数

计算机产生随机数的方法二——斐波那契法

$$X(n+2)=(X(n+1)+X(n)) \bmod m$$

给定初始种子X_0和X_1以及与周期有关的m即可

优点：计算简单，速度快，周期较长（可达3m/2）
缺点：序列中数重复出现，独立性较差，且有不居中现象（第三个数要么大于前两个数，要么小于前两个数，永远不会在两者之间）

随机数

计算机产生随机数的方法三——线性同余法

$$X(n+1)=a*X(n)+c \bmod m$$

a，c，m为参数，初始种子X0

（1）m与周期有关，越大越好

（2）要达到满周期（周期为m）需要满足三个条件，但满周期序列随机性不一定好

（3）为加快处理速度，最好当然是$m=2^k$，但此时低4位会是周期为16的循环，随机性不好。因此可取$m=2^k-1$

（4）为使序列周期尽量大，可取a为质数

随机数

常用的线性同余随机数发生器

（1）$X_n=（5^{15}X_{n-1}+1）\ \mathrm{mod}\ 2^{35}$

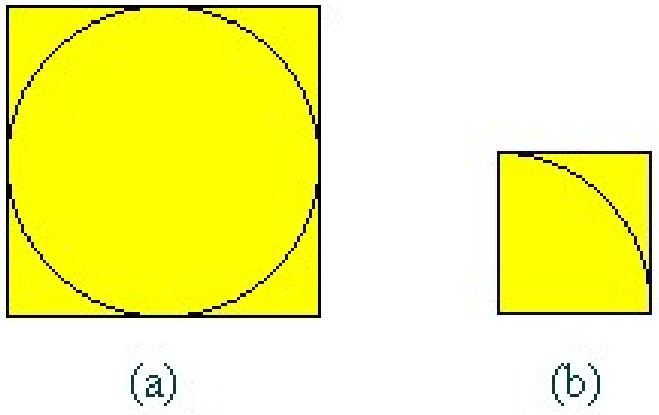
（2）$X_n=（314159269X_{n-1}+453806245）\ \mathrm{mod}\ 2^{31}$

7.2 数值概率算法

用随机投点法计算π值

设有一半径为r的圆及其外切四边形。向该正方形随机地投掷n个点。设落入圆内的点数为k。由于所投入的点在正方形上均匀分布，因而所投入的点落入圆内的概率为$\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$。所以当n足够大时，k与n之比就逼近这一概率。从而 $\pi \approx \frac{4k}{n}$ 。

```
public static double darts(int n)
   { // 用随机投点法计算π值
     int k=0;
     for (int i=1;i <=n;i++) {
       double x=dart.fRandom();
       double y=dart.fRandom();
       if ((x*x+y*y)<=1) k++;
       }
     return 4*k/(double)n;
   }
```



7.3 舍伍德算法

舍伍德(Sherwood)算法

设A是一个确定性算法，当它的输入实例为x时所需的计算时间记为$t_A(x)$。设X_n是算法A的输入规模为n的实例的全体，则当问题的输入规模为n时，算法A所需的平均时间为

$$\bar{t}_A(n) = \sum_{x \in X_n} t_A(x) / |X_n|$$

这显然不能排除存在x∈Xn使得$t_A(x) >> \bar{t}_A(n)$的可能性。

为了消除实例与计算时间之间的关联，在算法中添加随机性，成为随机算法B，使得对问题的输入规模为n的所有实例x，其运行时间都服从期望为 $t_B(x) = \bar{t}_A(n) + s(n)$ 的概率分布。其中s(n)是使算法随机化增加的时间，只跟问题规模有关。这样实例x的计算时间其实就与x本身无关。当s(n)与$t_A(n)$相比可忽略时，舍伍德算法可获得很好的平均性能。

如何将快速排序算法改造为舍伍德算法？

- 快速排序算法以第1个元素为基准将数组分为三部分，中间部分为基准元素，第一部分由所有小于等于基准元素的元素组成，第三部分由所有大于等于基准元素的元素组成。再对第一部分和第三部分本别递归调用快速排序算法。

- 从数组中随机选取一个元素作为基准元素，算法其余部分不变，就成为舍伍德算法。

舍伍德(Sherwood)算法

学过的Sherwood算法：**快速排序算法**

有时也会遇到这样的情况，即所给的确定性算法无法直接改造成舍伍德型算法。此时可借助于随机预处理技术，不改变原有的确定性算法，仅对其输入进行随机洗牌，同样可收到舍伍德算法的效果。例如，对于确定性选择算法，可以用下面的洗牌算法**shuffle**将数组a中元素随机排列，然后用确定性选择算法求解。这样做所收到的效果与舍伍德型算法的效果是一样的。

```
public static void shuffle(Comparable []a, int n)
  {// 随机洗牌算法
     rnd = new Random();
     for (int i=1;i<n;i++) {
       int j=rnd.random(n-i+1)+i;
       MyMath.swap(a, i, j);
       }
}
```

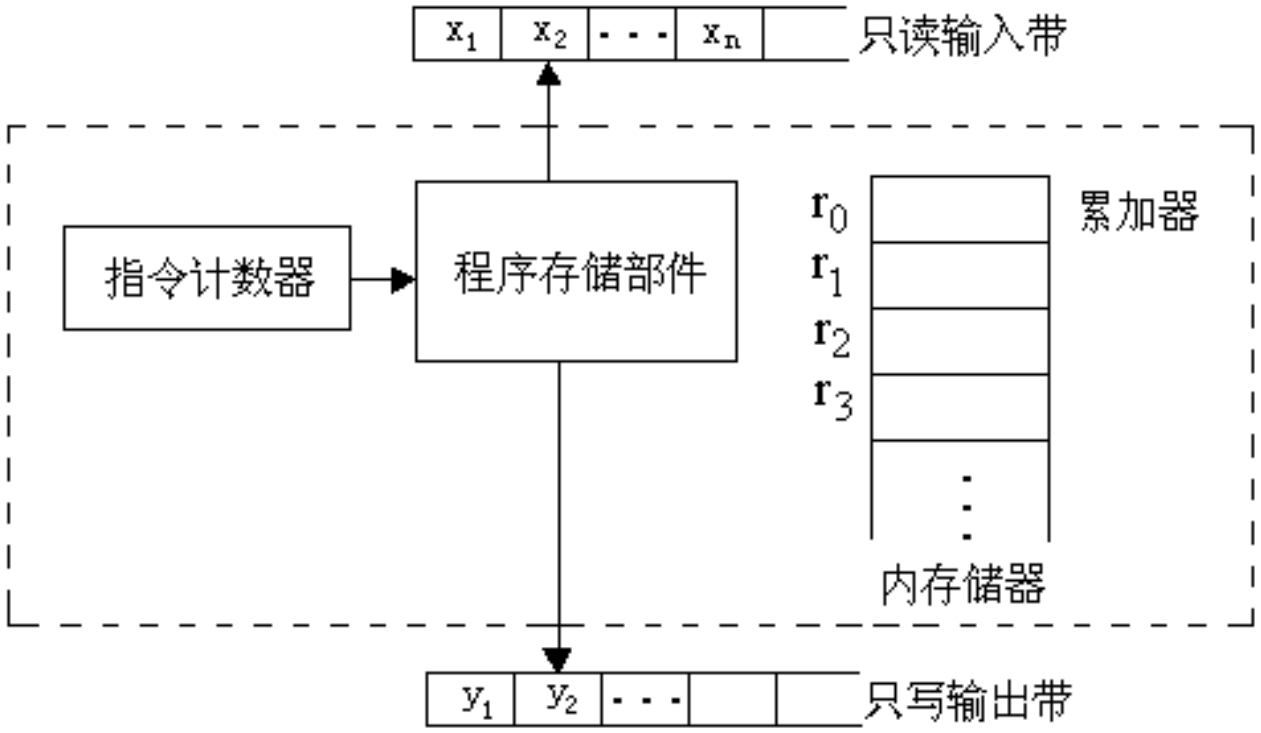
第8章　NP完全性理论

8.1 计算模型

- 8.1.1 随机存取机RAM
- 8.1.2 随机存取存储程序机RASP
- 8.1.3 RAM模型的变形与简化
- 8.1.4 图灵机
- 8.1.5 图灵机模型与RAM模型的关系
- 8.1.6 问题变换与计算复杂性归约

8.1.1 随机存取机RAM

1. RAM的结构



8.1.1 随机存取机RAM

2. RAM程序

一个RAM程序定义了从输入带到输出带的一个映射。可以对这种映射关系作2种不同的解释。

解释一：把RAM程序看成是计算一个函数

若一个RAM程序P总是从输入带前n个方格中读入n个整数x_1，x_2，…，x_n，并且在输出带的第一个方格上输出一个整数y后停机，那么就说程序P计算了函数**$f(x_1, x_2, \ldots, x_n)=y$**

解释二：把RAM程序当作一个语言接受器。

将字符串$S=a_1a_2\ldots a_n$放在输入带上。在输入带的第一个方格中放入符号a_1，第二个方格中放入符号a_2，…，第n个方格中放入符号a_n。然后在第n+1个方格中放入0，作为输入串的结束标志符。如果一个RAM程序P读了字符串S及结束标志符0后，在输出带的第一格输出一个1并停机，就说程序P接受字符串S。

8.1.1 随机存取机RAM

3. RAM程序的耗费标准

标准一：均匀耗费标准

在均匀耗费标准下，每条RAM指令需要一个单位时间；每个寄存器占用一个单位空间。以后除特别注明，RAM程序的复杂性将按照均匀耗费标准衡量。

标准二：对数耗费标准

对数耗费标准是基于这样的假定，即执行一条指令的耗费与以二进制表示的指令的操作数长度成比例。在RAM计算模型下，假定一个寄存器可存放一个任意大小的整数。因此若设l(i)是整数i所占的二进制位数，则

$$l(i)=\begin{cases}\lfloor \log |i| \rfloor & i \neq 0 \\ 1 & i = 0\end{cases}$$

8.1.2 随机存取存储程序机RASP

1. RASP的结构

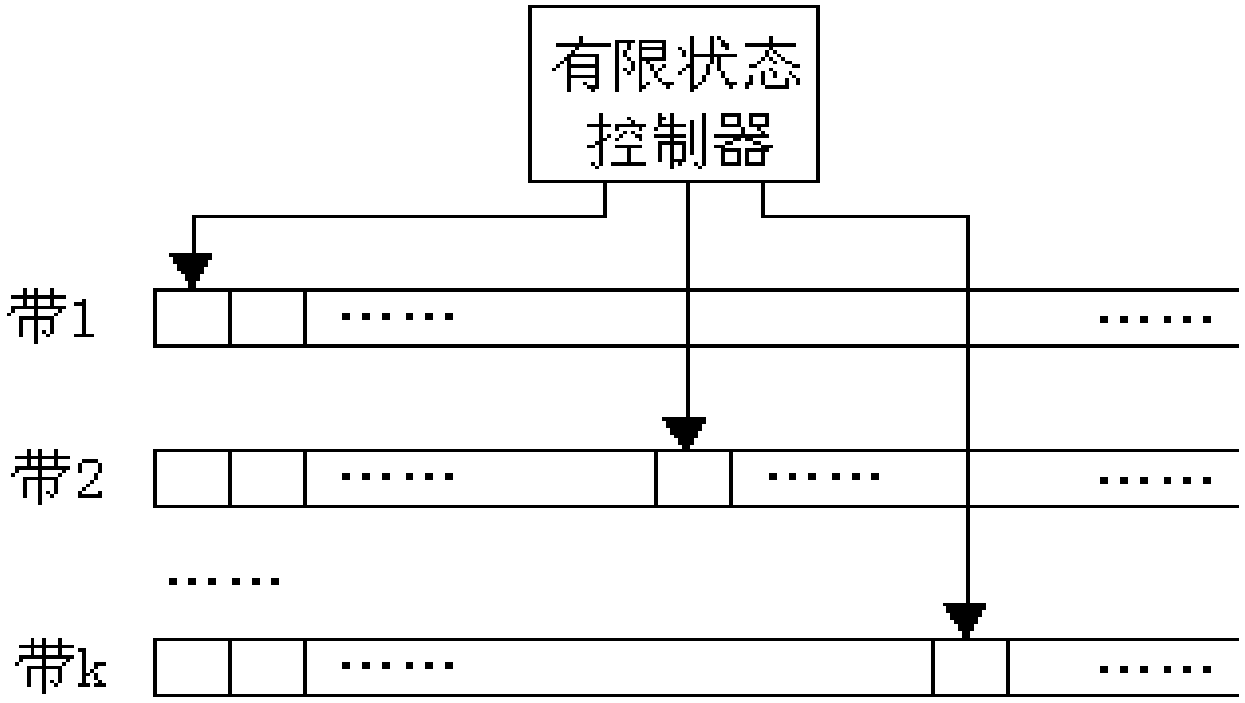
RASP的整体结构类似于RAM，所不同的是RASP的程序是存储在寄存器中的。每条RASP指令占据2个连续的寄存器。第一个寄存器存放操作码的编码，第二个寄存器存放地址。RASP指令用整数进行编码。

2. RASP程序的复杂性

不管是在均匀耗费标准下，还是在对数耗费标准下，RAM程序和RASP程序的复杂性只差一个常数因子。在一个计算模型下T(n)时间内完成的输入-输出映射可在另一个计算模型下模拟，并在kT(n)时间内完成。其中k是一个常数因子。空间复杂性的情况也是类似的。

8.1.4 图灵机

1. 多带图灵机



8.1.4 图灵机

1. 多带图灵机

根据有限状态控制器的当前状态及每个读写头读到的带符号，图灵机的一个计算步可实现下面3个操作之一或全部。

(1)改变有限状态控制器中的状态。

(2)清除当前读写头下的方格中原有带符号并写上新的带符号。

(3)独立地将任何一个或所有读写头，向左移动一个方格(L)或向右移动一个方格(R)或停在当前单元不动(S)。

k带图灵机可形式化地描述为一个7元组(Q，T，I，δ，b，q_0，q_f)，其中:

(1)Q是有限个状态的集合。 (2)T是有限个带符号的集合。

(3)I是输入符号的集合，$I \subseteq T$。(4)b是惟一的空白符，$b \in T-I$。

(5)q_0是初始状态。 (6)q_f是终止(或接受)状态。

(7)δ是移动函数。它是从$Q \times T^k$的某一子集映射到$Q \times (T \times \{L, R, S\})^k$的函数。

8.1.4 图灵机

1. 多带图灵机

与RAM模型类似，图灵机既可作为语言接受器，也可作为计算函数的装置。

图灵机M的时间复杂性T(n)是它处理所有长度为n的输入所需的最大计算步数。如果对某个长度为n的输入，图灵机不停机，T(n)对这个n值无定义。

图灵机的空间复杂性S(n)是它处理所有长度为n的输入时，在k条带上所使用过的方格数的总和。如果某个读写头无限地向右移动而不停机，S(n)也无定义。

8.1.5 图灵机模型与RAM模型的关系

图灵机模型与RAM模型的关系是指同一问题在这2种不同计算模型下的复杂性之间的关系。

定理8-3 对于问题P的任何长度为n的输入，设求解问题P的算法A在k带图灵机模型TM下的时间复杂性为 $T(n)$ ，那么，算法A在RAM模型下的时间复杂性为 $O(T^2(n))$ 。

定理8-4 对于问题P的任何长度为n的输入，设求解问题P的算法A在RAM模型下，不含有乘法和除法指令，且按对数耗费标准其时间复杂性为 $T(n)$ ，那么，算法A在k带图灵机模型TM下的时间复杂性为 $O(T^2(n))$ 。

8.1.6 问题变换与计算复杂性归约

通过问题变换的技巧，可以将2个不同问题的计算复杂性联系在一起。这样就可以将一个问题的计算复杂性归结为另一个问题的计算复杂性，从而实现问题的计算复杂性归约。

具体地说，假设有2个问题A和B，将**问题A变换为问题B**是指：
(1)将问题A的输入变换为问题B的适当输入。
(2)解出问题B。
(3)把问题B的输出变换为问题A的正确解。
若用O(τ(n))时间能完成上述变换的第(1)步和第(3)步，则称问题A是τ(n)时间可变换到问题B，且简记为$\mathbf{A} \propto_{\tau(n)} \mathbf{B}$。其中的n通常为问题A的规模(大小)。
当τ(n)为n的多项式时，称问题A可在多项式时间内变换为问题B。特别地，当τ(n)为n的线性函数时，称问题A可线性地变换为问题B。

8.2 P类与NP类问题

- 8.2.1 非确定性图灵机
- 8.2.2 P类与NP类语言
- 8.2.3 多项式时间验证

8.2.1 非确定性图灵机

在图灵机计算模型中，移动函数**δ是单值的**，即对于$Q\times T^k$中的每一个值，当它属于δ的定义域时，$Q\times(T\times\{L，R，S\})^k$中只有惟一的值与之对应，称这种图灵机为**确定性图灵机**，简记为**DTM**(Deterministic Turing Machine)。

非确定性图灵机（NDTM）：一个k带的非确定性图灵机M是一个7元组：(Q，T，I，δ，b，q_0，q_f)。与确定性图灵机不同的是非确定性图灵机允许移动函数**δ具有不确定性**，即对于$Q\times T^k$中的每一个值$(q;x_1,x_2,\ldots,x_k)$，当它属于δ的定义域时，$Q\times(T\times\{L，R，S\})^k$中有惟一的一个**子集**$\delta(q;x_1,x_2,\ldots,x_k)$与之对应。可以在$\delta(q;x_1,x_2,\ldots,x_k)$中随意选定一个值作为它的函数值。

8.2.2 P类与NP类语言

P类和NP类语言的定义：

P={L|L是一个能在**多项式时间内**被一台**DTM**所接受的语言}

NP={L|L是一个能在**多项式时间**内被一台**NDTM**所接受的语言}

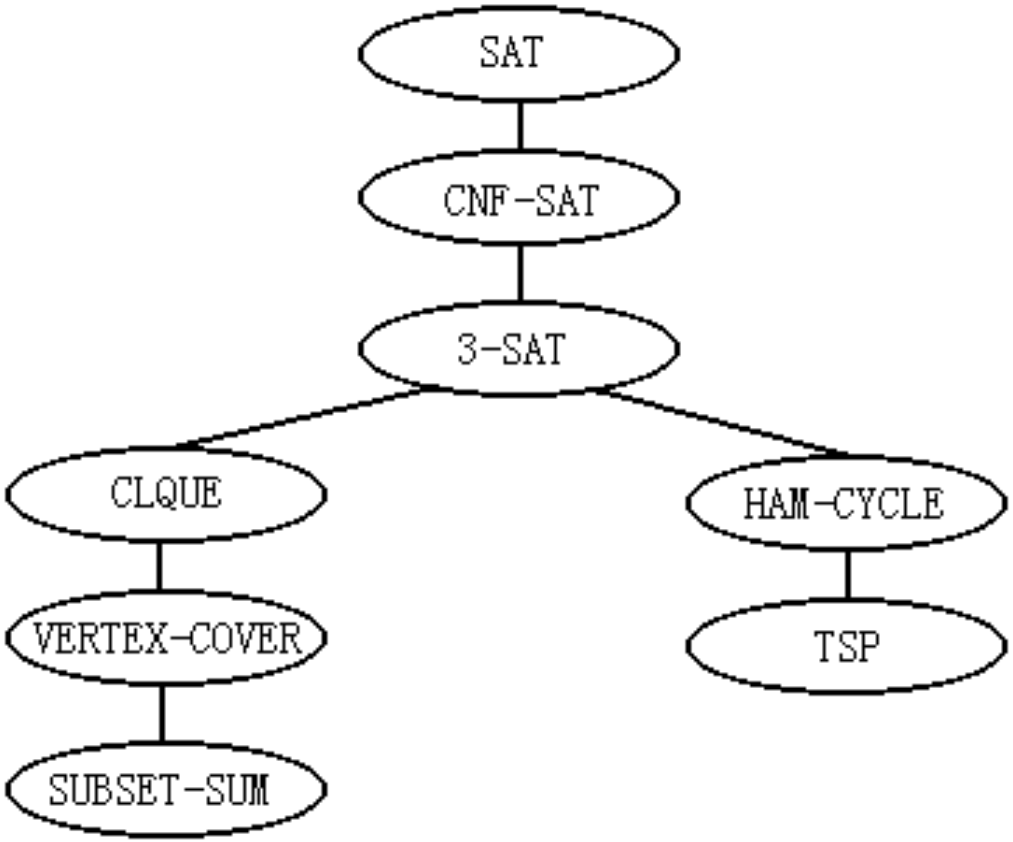
由于一台确定性图灵机可看作是非确定性图灵机的特例，所以可在多项式时间内被确定性图灵机接受的语言也可在多项式时间内被非确定性图灵机接受。故**P ⊆ NP**。

8.3 NP完全问题

一些说明:

- 用确定性图灵机在多项式时间内可解的判定问题称为P类问题;
- 用非确定性图灵机在多项式时间内可解的问题称为NP类问题;
- 对于一个NP问题X，如果其他所有的NP问题都可以在多项式时间内归约为X，则X称为NP完全问题.

8.4 一些典型的NP完全问题



部分NP完全问题树