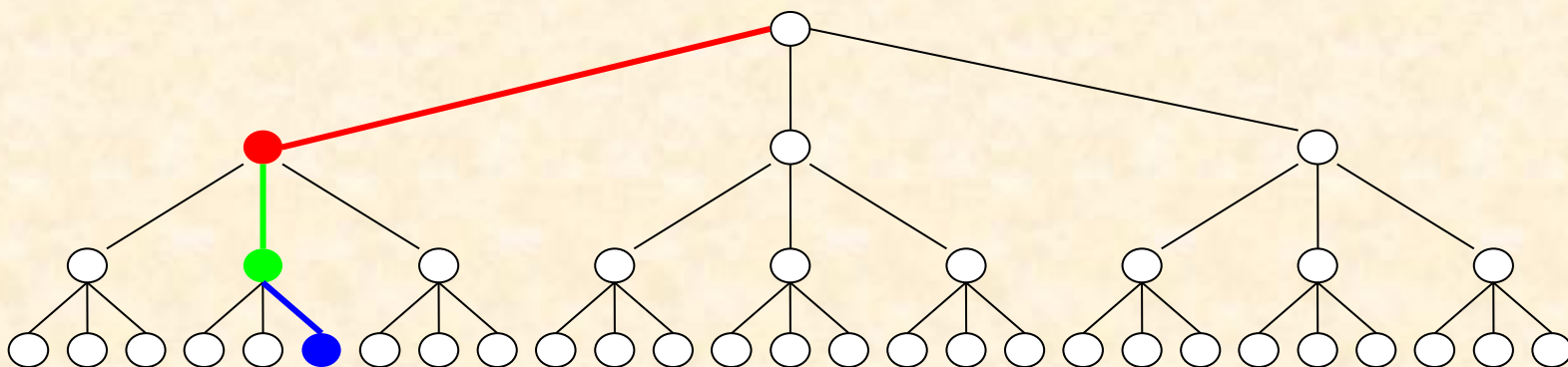


7 回溯法

Backtracking

引例

- 图的3着色问题
- 给定一个无向图 $G=(V,E)$ ，及3种颜色 $\{1,2,3\}$ ，现要为图的每个顶点着色。每顶点只能着一种颜色，并且要求相邻的顶点具有不同的颜色。
- 一个具有 n 个顶点的图，可用一个 n 维的向量 (c_1, c_2, \dots, c_n) 表示一种着色方案， $c_i \in \{1, 2, 3\}$, $i=1, 2, \dots, n$ ，共有 3^n 种可能的着色，可用一棵完全的3叉树表示。
- 下图为有3个顶点的所有可能的着色搜索树，从根到叶节点的每条路径表示一种着色方案。

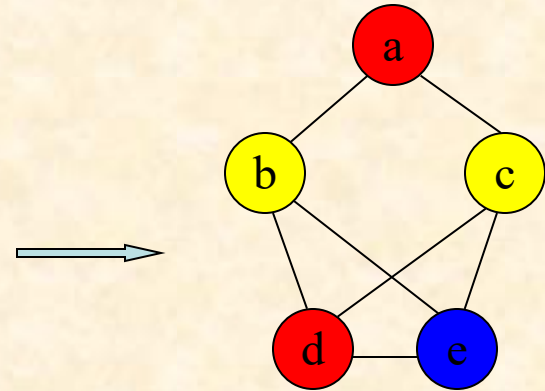
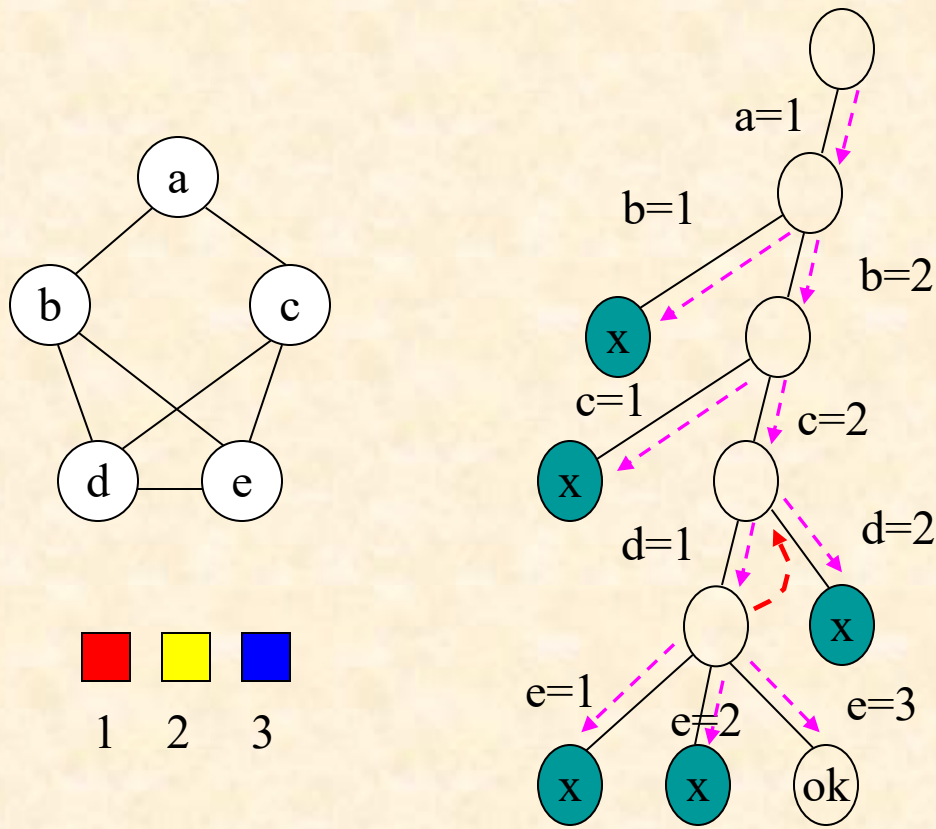


几个概念

- 问题的解向量：问题的解能够表示成一个 n 维向量 (x_1, x_2, \dots, x_n) 的形式。
- 显式约束：对分量 x_i 的取值限定。
- 隐式约束：为满足问题的解而对不同分量之间施加的限定。
- 问题的解空间：对于问题的一个实例，解向量满足**显式约束**条件的所有 n 维向量，构成了该问题实例的一个解空间。
- (0-1背包问题)

回溯法的工作原理

- 回溯法的基本做法是搜索，是一种组织得井井有条，能避免不必要搜索的穷举式搜索法。
- 回溯法按深度优先策略搜索问题的解空间树。算法搜索至解空间树的任意一节点时，先判断该节点是否可能包含问题的解：1) 如果肯定不包含，则跳过这个节点；2) 如果可能包含，进入该子树，继续按深度优先策略搜索；3) 若某节点 i 的所有子节点都不可能包含问题的解，则回溯到 i 的父节点，生成下一个节点，继续搜索。



理论上: $3^0 + 3^1 + \dots + 3^5 = 364$
 实际上: 10

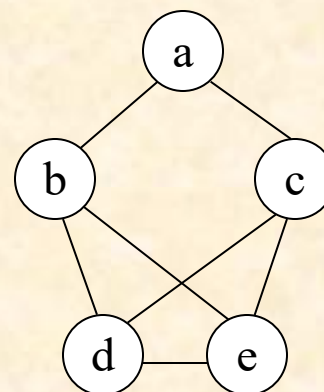
- “合法着色” — 全部顶点都已经被着色，且没有两个相邻的顶点是同样的颜色。
- “部分着色” — 一部分顶点还未着色；在已经着色的顶点中，没有两个相邻的顶点是同样的颜色。
- “非法着色” — 在已经着色的顶点中，有相邻的顶点是同样的颜色。

递归回溯

输入：无向图 $G=(V,E)$

输出：G的顶点的3着色 $c[1\dots n]$ ，其中每个 $c[j]$ 为1,2或3.

1. for $k \leftarrow 1$ to n
2. $c[k] \leftarrow 0$ //no color
3. end for
4. flag \leftarrow false
5. *graphcolor*(1)
6. if flag then output c
7. else output “no solution”



graphcolor(k)

1. for color=1 to 3
2. $c[k] \leftarrow$ color
3. if c 为合法着色 then flag \leftarrow true and exit
4. else if c 是部分着色 then *graphcolor*($k+1$)
5. end for

迭代回溯

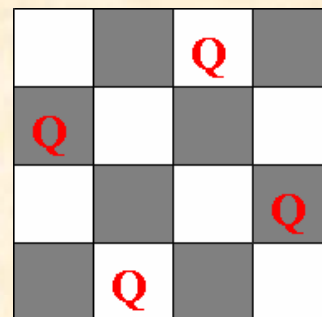
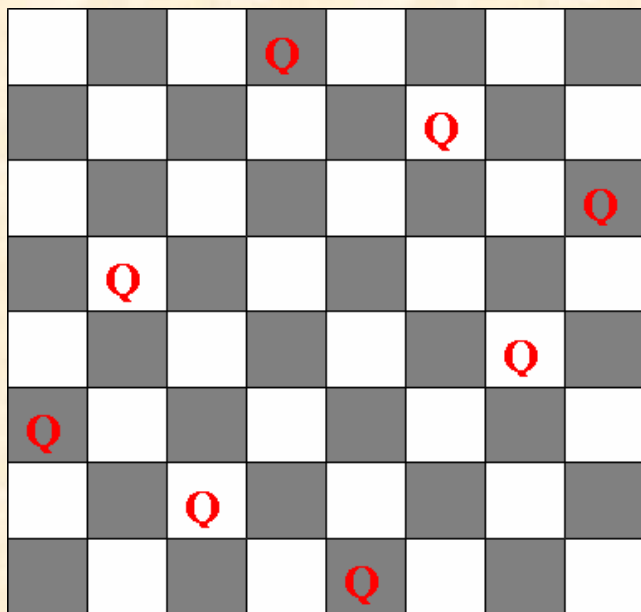
输入：无向图 $G=(V,E)$

输出：G的顶点的3着色 $c[1...n]$ ，其中每个 $c[j]$ 为1,2或3.

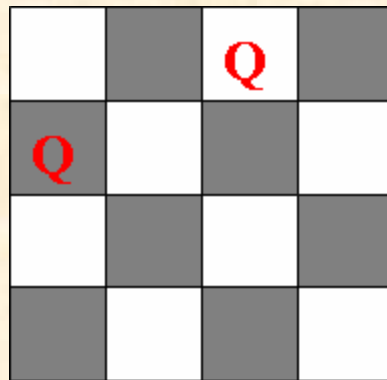
1. for $k \leftarrow 1$ to n
2. $c[k] \leftarrow 0$
3. end for
4. flag \leftarrow false
5. $k \leftarrow 1$ //start from v_1
6. while $k \geq 1$
7. while $c[k] \leq 2$ //为第 k 个顶点着色
8. $c[k] \leftarrow c[k] + 1$
9. if c 为合法着色 then flag \leftarrow true and exit
10. else if c 为部分着色 then $k \leftarrow k + 1$ //准备为下一顶点着色
11. end while //假设着色既不合法也非部分的，即死节点，试其他颜色
12. $c[k] \leftarrow 0$ // v_k 试验了所有颜色均失败，当前顶点颜色只好归0，回溯
13. $k \leftarrow k - 1$ //回溯到上一个顶点 (配合第6句的 $k \geq 1$ 来理解)
14. end while //注意：回溯到上一个顶点后， $c[k] + 1$ ，即尝试下一种颜色

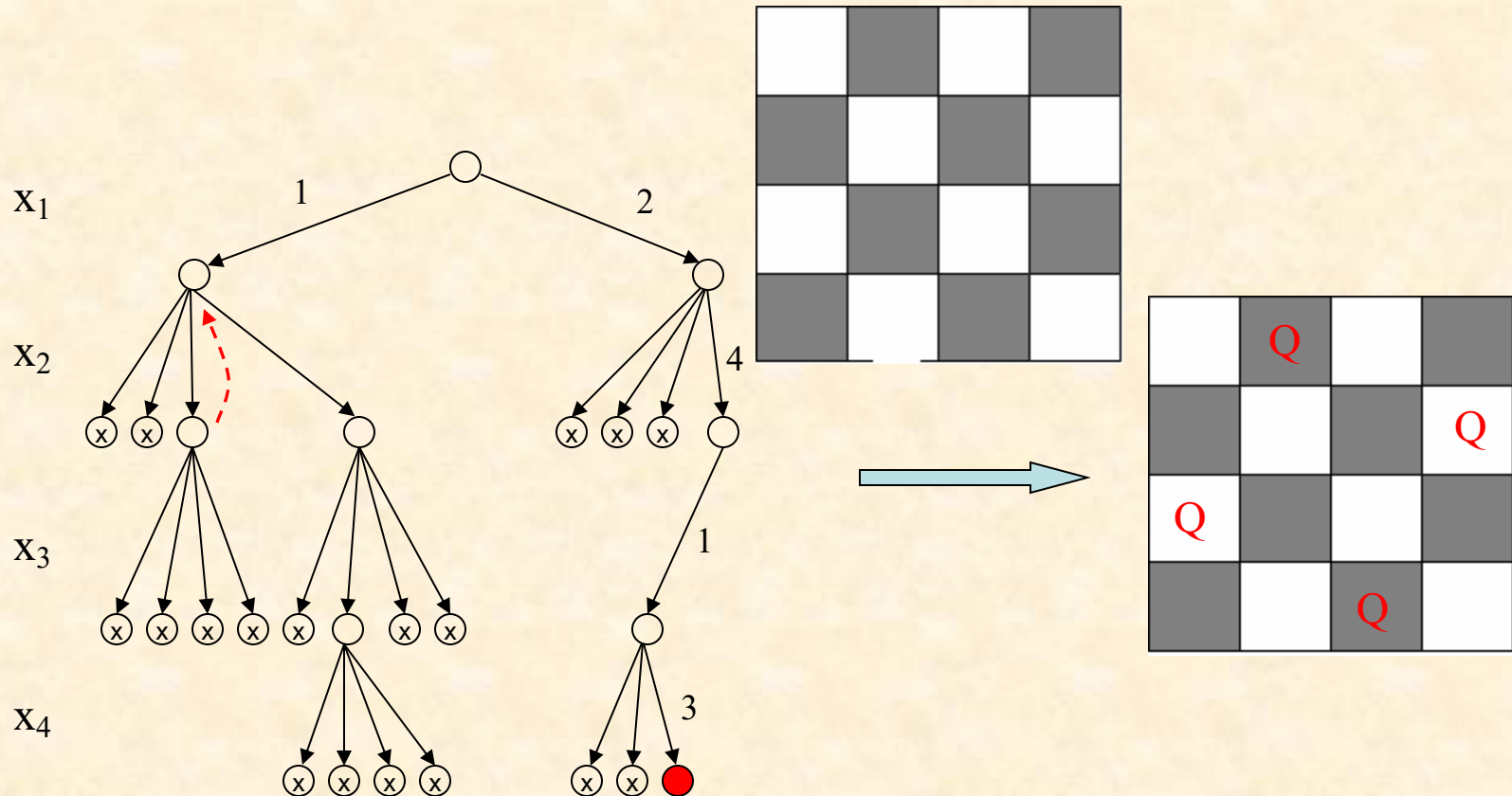
皇后问题

- 著名的数学家高斯在1850年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即：任意两个皇后都不能处于同一行、同一列或同一斜线上，问如何摆放？



- 考察n皇后问题n=4的情形：解空间有 4^4 (可减少至 $4!$)种布局,可用一棵高度为4的完全4叉树表示：树的根对应于没有放置皇后的布局，第一层节点对应于皇后在第一行(列)可能放置列(行)的情况，依此类推。
- 合法布局：一个不互相攻击的4 皇后布局
- 部分布局：一个不互相攻击的少于4个皇后的布局
- 下图表示一个部分布局，用向量(3,1,0,0)表示；

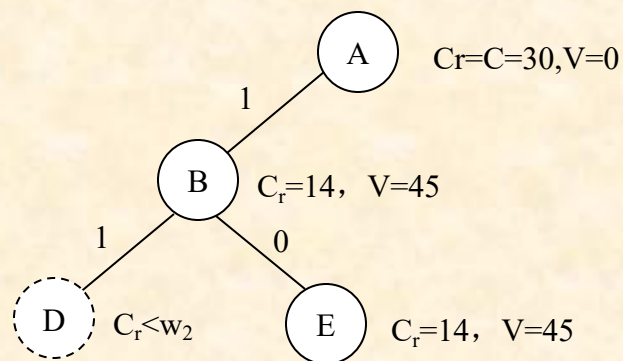




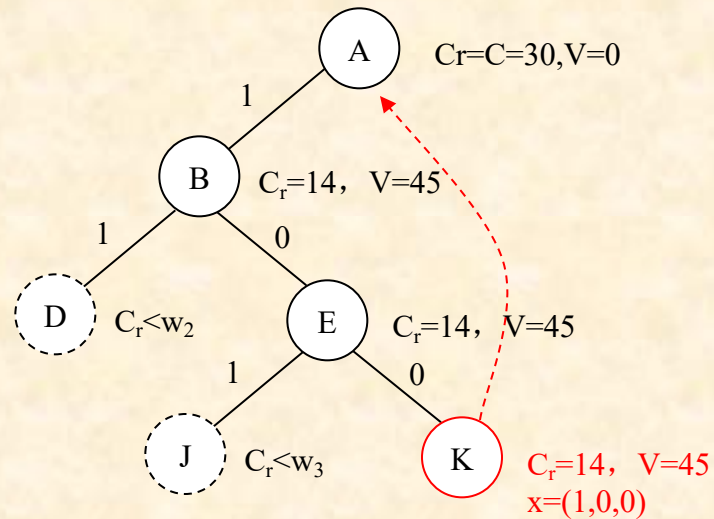
- 效率问题：蛮力方法候选解的个数是 $n!$ ；而使用回溯法，候选解的个数是 n^n ，回溯的效率是否太低了？
- 然而，仔细分析可以发现，回溯法可以极大减少测试次数：
 - 例如，假设前两个皇后均放在了第一列，蛮力方法仍旧要测试 $(n-2)!$ 个候选解；而回溯法只要进行一次测试就可以避免剩余无意义的测试。
 - 所以，尽管在最坏情况下要用 $O(n^n)$ 时间来求解，然而大量实际经验表明，它在有效性上远远超过蛮力方法 $O(n!)$ 。例如在4皇后问题中，只搜索了341个节点中的27个就找到了解。

0-1背包问题的回溯求解 (示例)

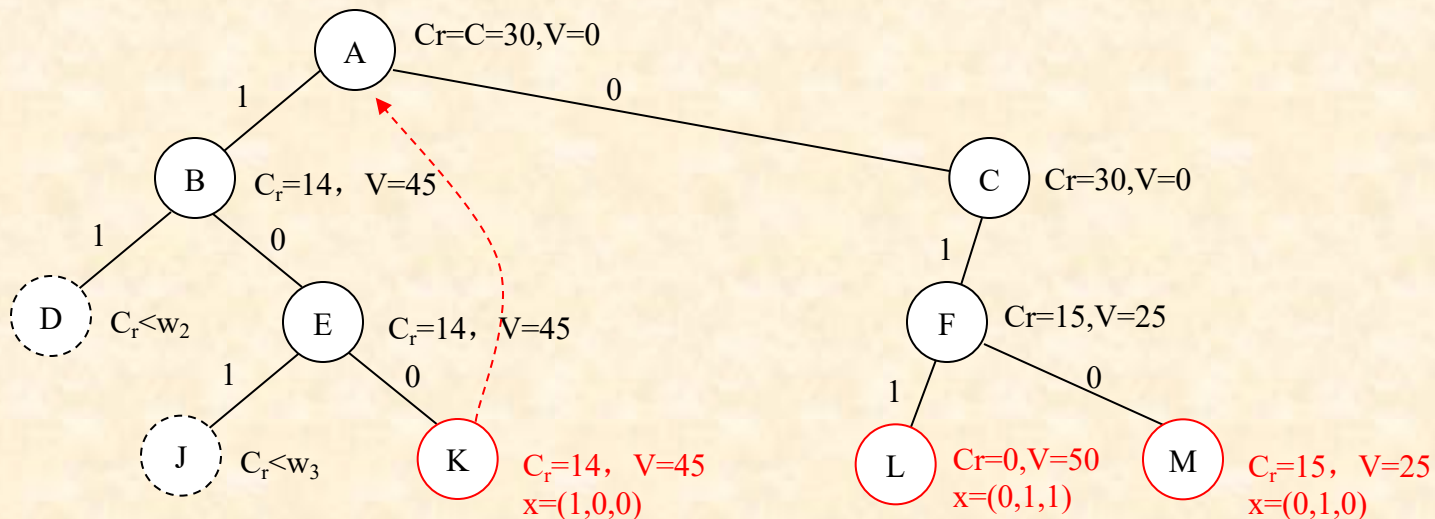
- 问题: $n=3$, $C=30$, $w=\{16, 15, 15\}$, $v=\{45, 25, 25\}$
- 开始时, $C_r=C=30$, $V=0$, A为唯一活节点, 也是当前的扩展节点
- 扩展A, 先到达B节点
 - $C_r=C_r-w_1=14$, $V=V+v_1=45$
 - 此时A、B为活节点, B成为当前扩展节点
- 扩展B, 先到达D
 - $C_r < w_2$, D导致一个不可行解, 尝试B的下一个元素
- 再扩展B到达E



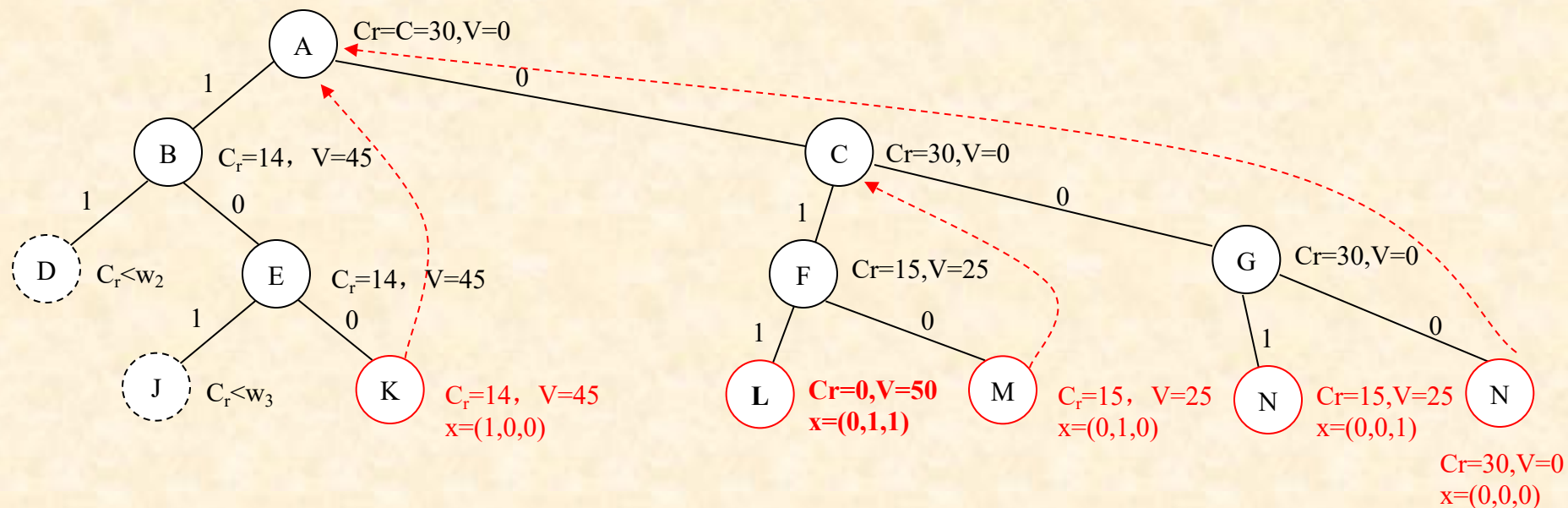
- 再扩展B到达E(续)
 - E可行, 此时A、B、E是活节点, E成为新的扩展节点
 - 扩展E, 先到达J
 - $C_r < w_3$, J导致一个不可行解, 尝试E的下一个元素
 - 再次扩展E到达K
 - 由于K是叶节点, 即得到一个可行解 $x=(1,0,0)$, $V=45$
 - K不可扩展, 成为死节点, 返回到E
 - E没有可扩展节点, 成为死节点, 返回到B
 - B没有可扩展节点, 成为死节点, 返回到A (回溯到A)



- A再次成为扩展节点，扩展A到达C (尝试A的下一元素)
 - $C_r=30$, $V=0$, 活节点为A、C, C为当前扩展节点
 - 扩展C, 先到达F
 - $C_r=C_r-w_2=15$, $V=V+v_2=25$, 此时活节点为A、C、F, F成为当前扩展节点
 - 扩展F, 先到达L
 - $C_r=C_r-w_3=0$, $V=V+v_3=50$
 - L是叶节点, 且 $50>45$, 皆得到一个可行解 $x=(0,1,1)$, $V=50$
 - L不可扩展, 成为死节点, 返回到F
 - 再扩展F到达M
 - M是叶节点, 且 $25<50$, 不是最优解
 - M不可扩展, 成为死节点, 返回到F
 - F没有可扩展节点, 成为死节点, 返回到C



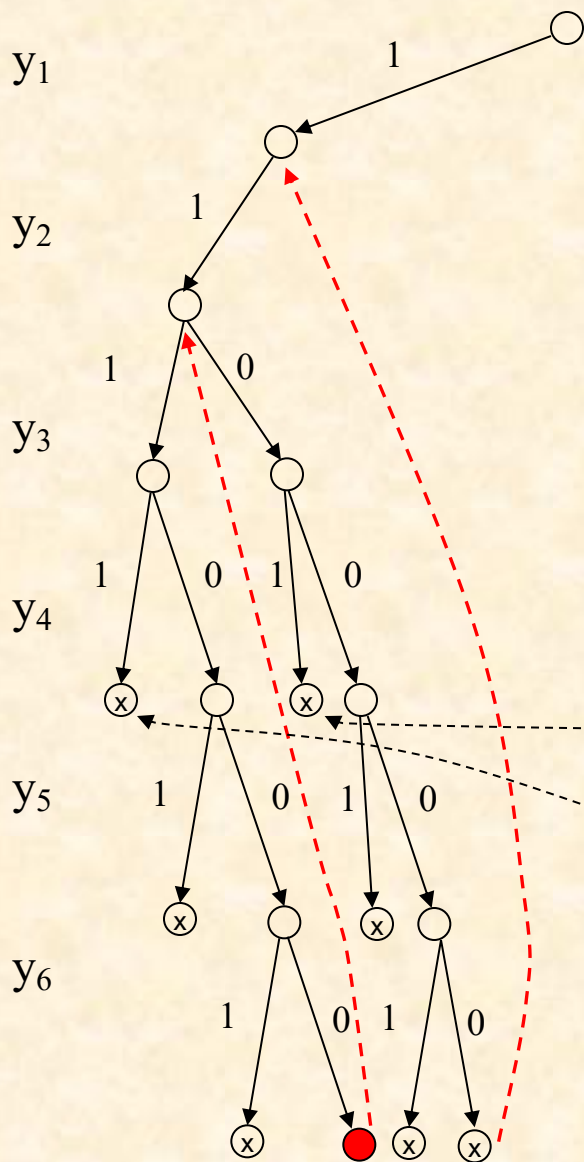
- 再扩展C到达G
 - $C_r=30$, $V=0$, 活节点为A、C、G, G为当前扩展节点
 - 扩展G, 先到达N, N是叶节点, 且 $25 < 50$, 不是最优解, 又N不可扩展, 返回到G
 - 再扩展G到达O, O是叶节点, 且 $0 < 50$, 不是最优解, 又O不可扩展, 返回到G
 - G没有可扩展节点, 成为死节点, 返回到C
- C没有可扩展节点, 成为死节点, 返回到A
- A没有可扩展节点, 成为死节点, 算法结束, 最优解 $X=(0,1,1)$, 最优值 $V=50$



- 给定一个问题的实例，如果该问题实例的解可以表示成定长的向量形式，那么就可以考虑使用回溯法来解决。
- 如果一个问题的实例，其解是一个变长的向量形式，是否可以使用回溯法？

一个例子

- 考虑如下的划分问题：给定 n 个整数的集合 $X = \{x_1, x_2, \dots, x_n\}$ ，整数 y ，试找到 X 的一个子集 Y ， Y 中所有元素的和等于 y 。
- 例如： $X = \{10, 20, 30, 40, 50, 60\}$ ， $y = 60$ 。那么该问题的合法解有 $Y = \{10, 20, 30\}$ 、 $\{20, 40\}$ 、 $\{60\}$ 。(非定长的向量形式)
- 可以设法将问题的解转化为定长的向量形式： $Y = \{y_1, y_2, y_3, y_4, y_5, y_6\}$
- $\{1, 1, 1, 0, 0, 0\}$ 、 $\{0, 1, 0, 1, 0, 0\}$ 、 $\{0, 0, 0, 0, 0, 1\}$



$X=\{10,20,30,40,50,60\}, \quad y=60$

$Y=\{y_1,y_2,y_3,y_4,y_5,y_6\}$

$\{1,1,1,0,0,0\}, \{0,1,0,1,0,0\}, \{0,0,0,0,1\}$

通用回溯方法框架

- 本小节描述通用回溯方法的一般框架，可以作为系统搜索的基本框架，在解决实际问题时，修改该基本框架中的相应部分使之适合实际问题即可。
- 对于以下这样一类问题，可以使用回溯法：这类问题的解满足事先定义好的某种约束向量 (x_1, \dots, x_i) ，这里 i 是依赖于问题规模 n 的常量。

在回溯法中，每个 x_i 均是属于某个有限集合的，不妨称之为 X_i ，那么回溯法实质上是按照词典顺序考虑笛卡儿积：

$$X_1 \times X_2 \times \dots \times X_n$$

中的元素。

- 算法最初从空向量开始，先选择 X_1 中的最小元素作为 x_1 ；
- 如果 (x_1) 是部分解，那么选择 X_2 中的最小元素作为 x_2 ；
- 如果 (x_1, x_2) 是部分解，则继续考虑 X_3 中的最小元素作为 x_3 ；
- 否则，考虑 X_2 中的第二个元素作为 x_2 。依此类推。

一般说来，当算法已经检测到部分解 (x_1, x_2, \dots, x_j) ，需要继续考虑 $v = (x_1, x_2, \dots, x_j, x_{j+1})$ 时，有以下几种情形：

1. 若 v 表示问题的最终解，算法记录下它作为一个解。如果只需要一个解，算法结束；否则，继续找其它解。
2. 如果 $v = (x_1, x_2, \dots, x_j, x_{j+1})$ 是一个部分解，那么选择集合 X_{j+2} 中未使用过的最小元素作为 x_{j+2} 。(向前搜索)
3. 如果 v 既非最终解，也非部分解，那么会有以下两种情况：
 - a. 如果集合 X_{j+1} 中还有其它未曾使用过的元素，则选择下一个未曾使用过的元素作为 x_{j+1} 。
 - b. 如果集合 X_{j+1} 中没有其它未曾使用过的元素，则回溯，将 X_j 中未曾使用的下一元素作为 x_j ，并将 x_{j+1} 进行重置。如果集合 X_j 中没有未曾使用的元素，那么继续回溯(同样注意要进行重置)。

递归回溯

输入：集合 X_1, X_2, \dots, X_n 清楚或是隐含的描述

输出：解向量 $v = (x_1, x_2, \dots, x_n)$, $0 \leq i \leq n$

1. $v \leftarrow ()$
2. $\text{flag} \leftarrow \text{false}$
3. $\text{advance}(1)$
4. if flag then output v
5. else output “no solution”

过程 $\text{advance}(k)$

1. for 每个 $x \in X_k$
2. $x_k \leftarrow x$; 将 x_k 加入 v
3. if c 为最终解 then set $\text{flag} \leftarrow \text{true}$ and exit
4. else if v 是部分解 then $\text{advance}(k+1)$
5. end for

迭代回溯

输入：集合 X_1, X_2, \dots, X_n 清楚或是隐含的描述

输出：解向量 $v = (x_1, x_2, \dots, x_n)$, $0 \leq i \leq n$

1. $v \leftarrow ()$
2. $\text{flag} \leftarrow \text{false}$
3. $k \leftarrow 1$
4. while $k \geq 1$
5. while X_k 没有被穷举
6. $x_k \leftarrow X_k$ 里的下一个元素；将 x_k 加入 v
7. if v 为最终解 then set $\text{flag} \leftarrow \text{true}$ and exit
8. else if v 是部分解 then $k \leftarrow k+1$ //前进
9. end while
10. 重置 X_k //一般将 $x_k \leftarrow 0$ ，表示将其重置
11. $k \leftarrow k-1$ //回溯
12. end while
13. if flag then output v
14. else output “no solution”

小结

- 回溯法的基本步骤：
 - 针对所给问题，定义问题的解空间；
 - 确定易于搜索的解空间结构；
 - 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。
- 常用剪枝函数：
 - 用约束函数在扩展节点处剪去不满足约束的子树；
 - 用限界函数剪去得不到最优解的子树。
- 关于复杂性：
 - 用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根节点到当前扩展节点的路径。如果解空间树中从根节点到叶节点的最长路径的长度为 $h(n)$ ，则回溯法所需的计算空间通常为 $O(h(n))$ 。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 的空间。

作业

- 设计一个回溯算法来生成数字 $1, 2, \dots, n$ 的所有排列。

分支限界法(Branch and Bounds)

- 分支限界法类似于回溯法，也是一种在问题的解空间树T中搜索问题解的算法。
- 分支限界法与回溯法的求解目标不同：
 - 回溯法寻找的是满足给定性质的一个解或解的集合(如3着色问题、8皇后问题)。
 - 分枝限界法关心的是目标函数的最大化或最小化。
例如背包问题(如前所示)，若用回溯法求解，则在解空间搜索树中找到许多可行解，然后从可行解中找到使得背包内物品价值最大化的解，即最优解。然而，利用分枝限界法，可直接找到问题的最优解。
- 分支限界法与回溯法的搜索方式不同
 - 回溯法：深度优先
 - 分支限界法：广度优先或最小耗费（最大收益）优先

基本思想

- 分支限界法常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。
- 在分支限界法中，每一个活节点只有一次机会成为扩展节点。活节点一旦成为扩展节点，就~~一次性~~产生其所有儿子节点。在这些儿子节点中，导致不可行解儿子节点被舍弃，其余儿子节点被加入活节点表(队列或是优先队列，详见后面的例子)中。
- 此后，从活节点表中取下一节点成为当前扩展节点，并重复上述节点扩展过程。这个过程一直持续到找到所需的解或活节点表为空时为止。

常见的两种分支限界法

- 从活节点表中选择下一扩展节点的不同方式导致不同的分支限界法：
 - 队列式(FIFO)分支限界法(**宽度优先**)：按照队列先进先出(FIFO)原则选取下一个节点为扩展节点。
 - 优先队列式分支限界法(**最小耗费-[最小堆]或是最大收益优先-[最大堆]**)：按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。
 - 最大优先队列：使用最大堆，体现最大收益优先
 - 最小优先队列：使用最小堆，体现最小耗费优先

0-1背包问题, 应用优先队列式分支限界法

考虑 $n=4$ 的背包问题（背包承重量为 $C=10$ ）：

物品	重量	价值	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

其中物品已经按照“价值-重量比”降序排列，这样会带来处理上的方便。

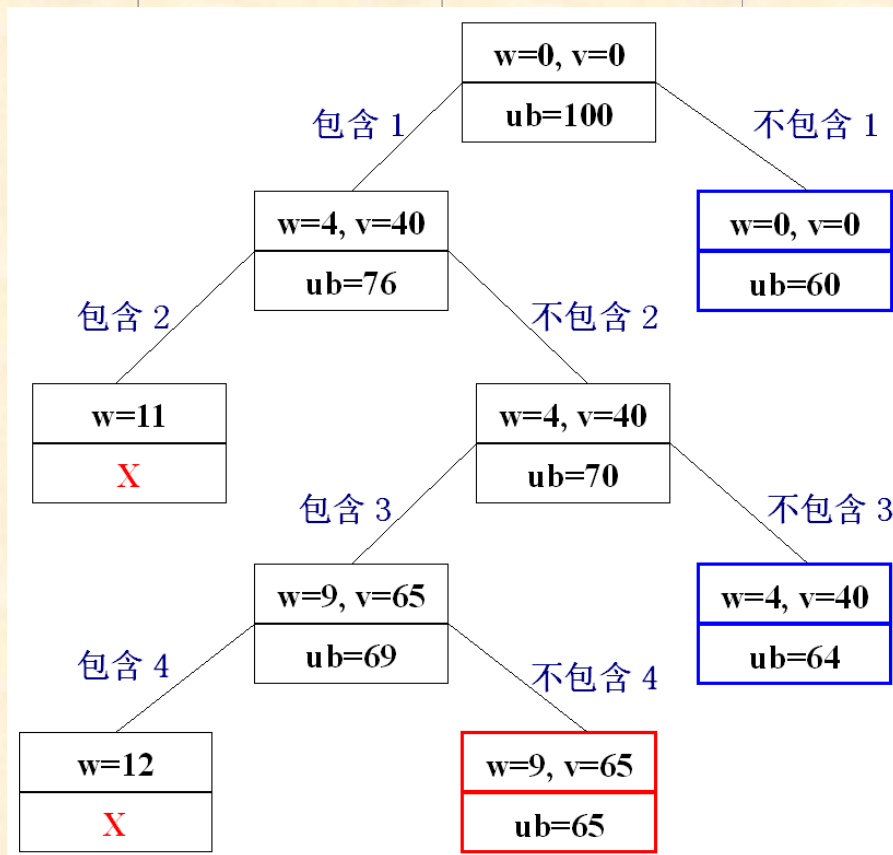
- 为每个搜索节点设置一个上界ub (upper bound)，一个简单方法是：把已经选择的物品总价值v，加上背包剩余承重量C-w与剩下可选择的物品的最高“价值-重量比”的乘积，即 $ub = v + (C-w) \times (v_i/w_i)_{\max}$
- 比如，若已装物品1，则价值的上界为 $40 + (10 - 4) \times 6 = 76$

物品	重量	价值	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

考虑n=4的背包问题（背包承重量为C=10）：

C=10

物品	重量	价值	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4



$$ub = v + (C-w) \times (v_i/w_i)_{\max}$$

使用优先队列，即最大堆。

注意，辅助的堆并没有显式地画出来。

回溯法与分支限界法的对比

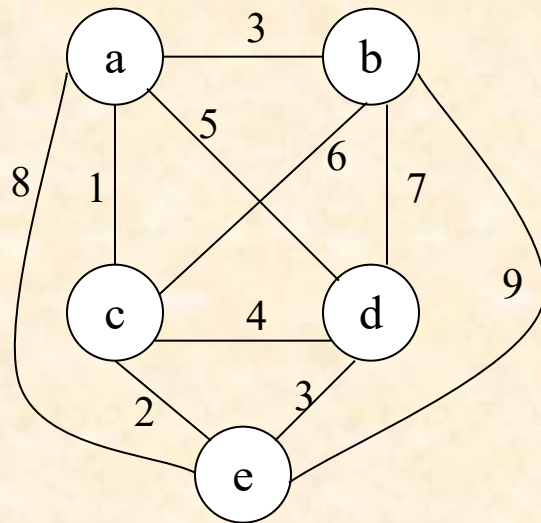
方法	对解空间树的搜索方式	存储节点的常用数据结构	节点存储特性	常用应用
回溯法	深度优先搜索	堆栈	活节点的所有可行子节点被遍历后才被从栈中弹出	找出满足约束条件的所有解
分支限界法	广度优先 或 最小消耗(最大效益)优先搜索	队列 或 优先队列	每个节点只有一次成为活节点的机会	找出满足约束条件的一个解或特定意义下的最优解

旅行商问题(TSP)

- 问题描述：一个商人要到 n 个城市去推销货物，从某城市出发，要求找到一条闭合路径， n 个城市都经历一次，最后回到出发点，且使得旅行的花费最小。
- 与背包问题(最大效益)相反，本问题(最小耗费)需要估计每个搜索节点的下界。

一个实例

- 考虑如右图所示的TSP问题 (假设每条边上的权值为非负整数)。
- 作为演示, 附加一个特殊的约束条件: b 在c之前被访问。



从该节点继续搜索, 所获得的收益不会超过ub

0-1背包问题 → 最大收益 → 为搜索节点设置一个上界ub (upper bound) → ub大的节点优先搜索

TSP问题 → 最小耗费 → 为搜索节点设置一个下界lb (lower bound) → lb小的节点优先搜索

从该节点继续搜索, 所需要的耗费不会低于lb

为搜索节点设置有效的下界

(1) 简单下界：距离矩阵D的最小元素乘以n即可。

(2) 一个有效的下界：

(2.1) 对每个顶点i, 用 s_i 表示顶点i到最近两个顶点的距离(分别用 s_i^1 和 s_i^2)之和, 即 $s_i = s_i^1 + s_i^2$; 对 s_i 求和得到s, 然后除以2并向上取整作为搜索节点的下界, 即 $lb = \lceil s/2 \rceil$

如右图, 搜索节点的下界为:

$$lb = \lceil [(1+3) + (3+6) + (1+2) + (3+4) + (2+3)]/2 \rceil = 14$$

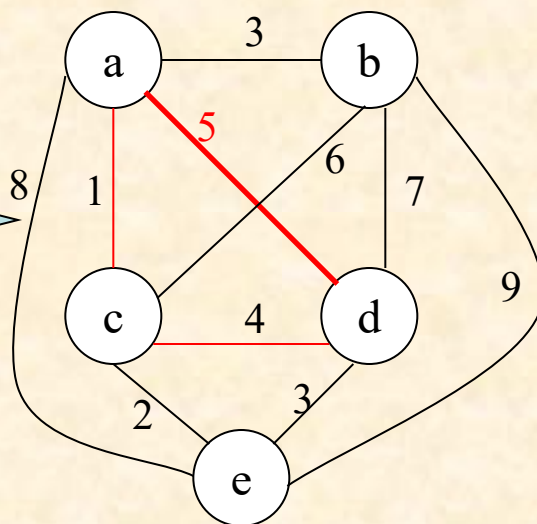
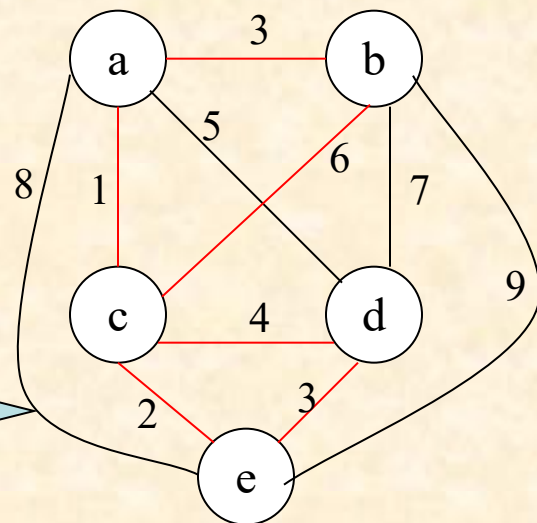
$s_a \quad s_b \quad s_c \quad s_d \quad s_e$

问题：为什么可以如此设置呢？下页分析。

(2.2) 若必须包含某条边, 如(a,d): 对于顶点a, 其 $s_a = |(a,d)| + a$ 到其它顶点最短的边; 对于顶点d, 其 $s_d = |(a,d)| + d$ 到其它顶点最短的边, 即:

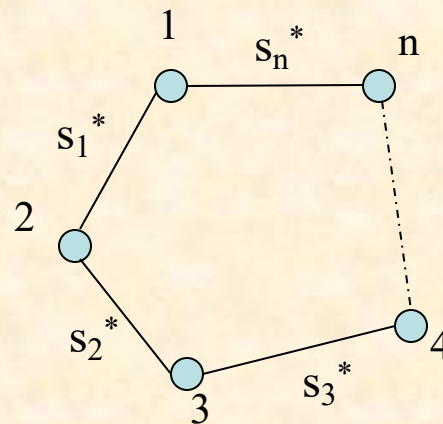
$$lb = \lceil [(1+5) + (3+6) + (1+2) + (3+5) + (2+3)]/2 \rceil = 16$$

$s_a \quad s_b \quad s_c \quad s_d \quad s_e$



为什么可以如此设置？

- 给定 n 个顶点的无向图，边的权值已经给定。将符合题意的TSP回路记为 T 。
 - 显然，每个顶点在 T 中只出现一次。
 - 在回路 T 中的顶点依次编号， $1, 2, 3, \dots, n$
 - 回路 T 中的边依次记为： $s_1^*, s_2^*, s_3^*, \dots, s_n^*$



- 显然，我们有：
 - 对于顶点1： $s_1^* + s_n^* \geq s_1^1 + s_1^2 = s_1$
 - 对于顶点2： $s_1^* + s_2^* \geq s_2^1 + s_2^2 = s_2$
 - ...
 - 对于顶点 n ： $s_{n-1}^* + s_n^* \geq s_n^1 + s_n^2 = s_n$

+

$$2(s_1^* + s_2^* + \dots + s_n^*) \geq s_1 + \dots + s_n = s$$

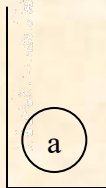
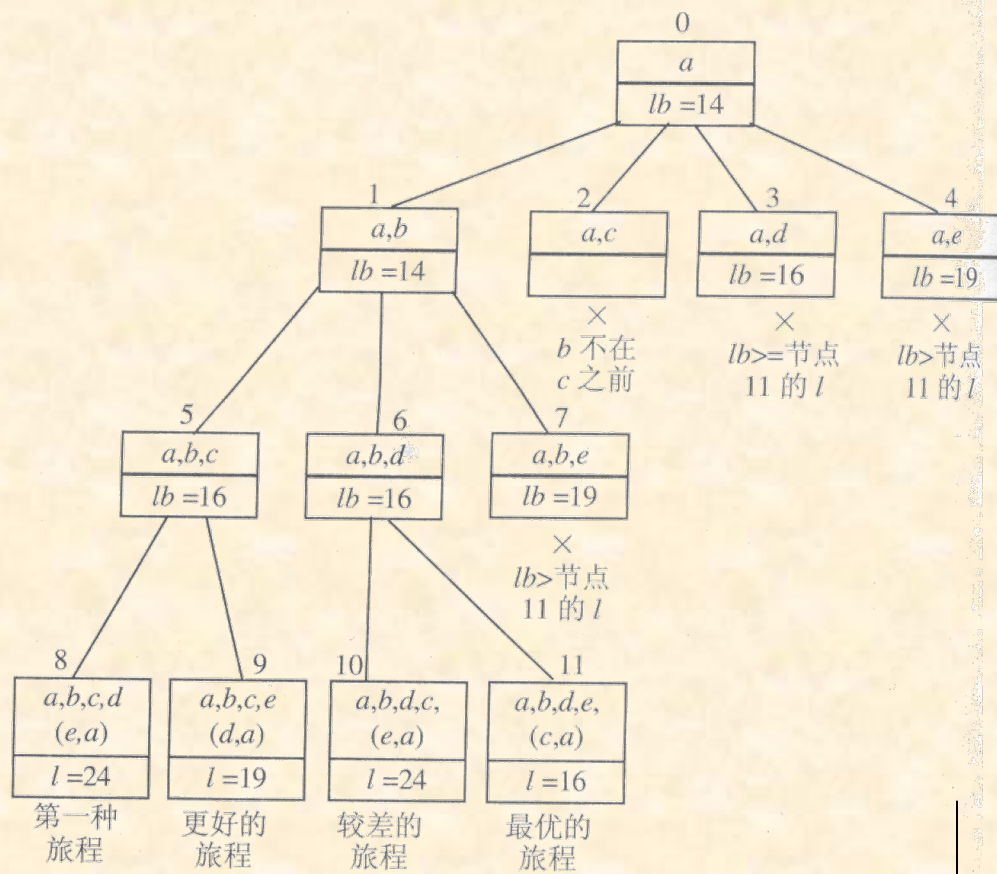
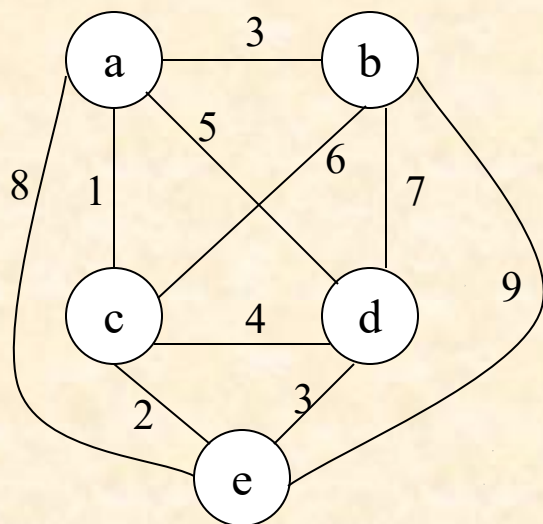
$$(s_1^* + s_2^* + \dots + s_n^*) \geq s/2$$

又因为 $(s_1^* + s_2^* + \dots + s_n^*)$ 是整数，必定有

$$(s_1^* + s_2^* + \dots + s_n^*) \geq \lceil s/2 \rceil$$

所以，可以取 $lb = \lceil s/2 \rceil$

搜索树



作业：分派问题

- 假设有A,B,C,D四位教师，每人都能教课程I,II,III,IV。但是各人备各门课所需要的时间不同，如下表所示：
- 现要求每门课有一个人教，并且每个人只能教一门课。问如何分配课程，才能使得总备课时间最少？
- 显然穷举法需要4！种方案。试尝试使用分支限界法。

	I	II	III	IV
A	2	10	9	7
B	15	4	14	8
C	13	14	16	11
D	4	15	13	9