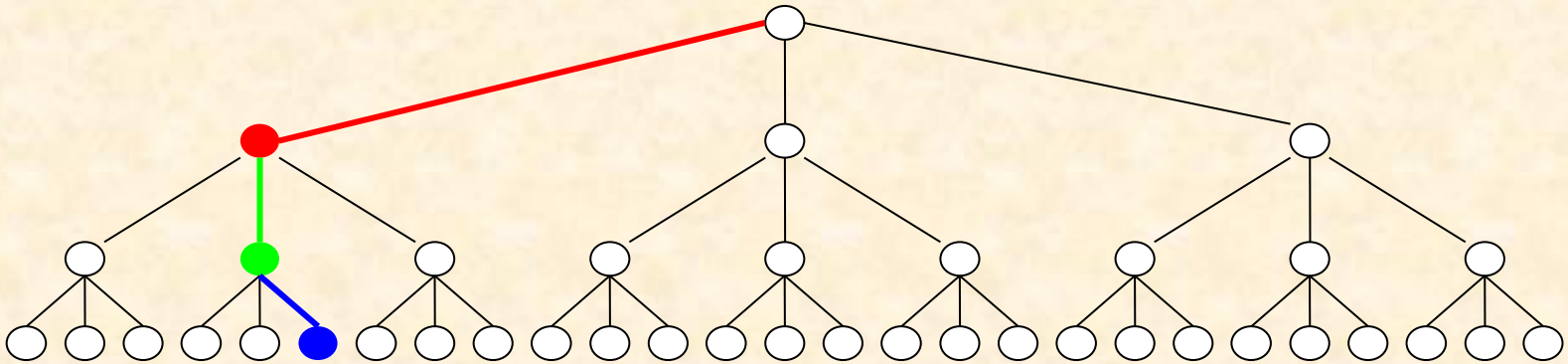
7　回溯法
Backtracking

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$(c_1,c_2,\ldots,c_n)$表示一种着色方案，$c_i \in \{1,2,3\}$, i=1,2,...,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。

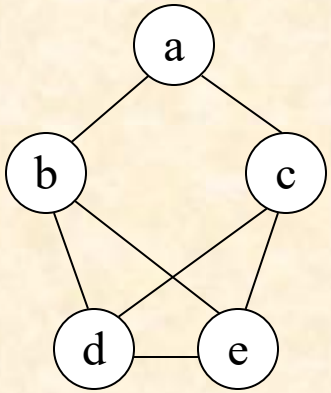


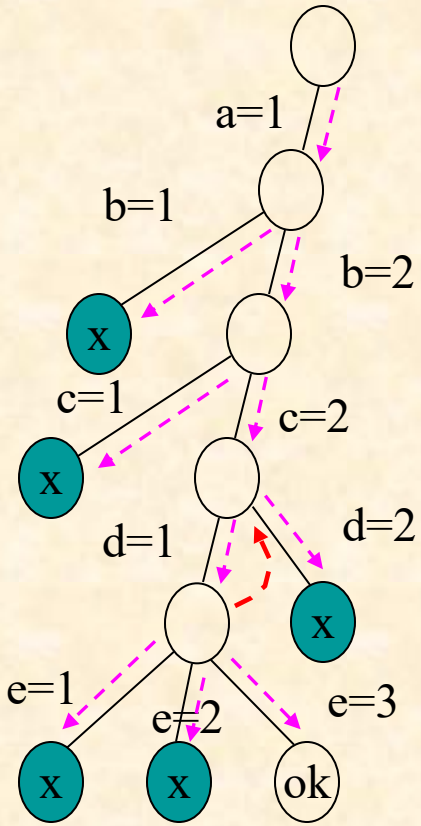
几个概念

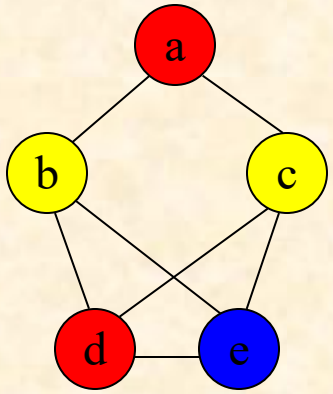
- 问题的解向量：问题的解能够表示成一个n维向量(x_1,x_2,,…,x_n)的形式。
- 显式约束：对分量x_i的取值限定。
- 隐式约束：为满足问题的解而对**不同分量**之间施加的限定。
- 问题的解空间：对于问题的一个实例，解向量满足显式约束条件的所有n维向量，构成了该问题实例的一个解空间。
- (0-1背包问题)

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：$3^0+3^1+\ldots+3^5=364$

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。

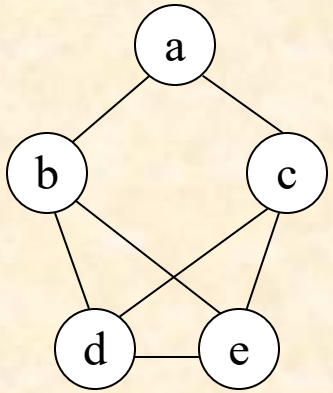
“部分着色”－部分顶点还未着色；在已经着色的顶点中，没有两个相邻的顶点是同样的颜色。

“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…n]，其中每个c[j]为1,2或3.

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



graphcolor(k)

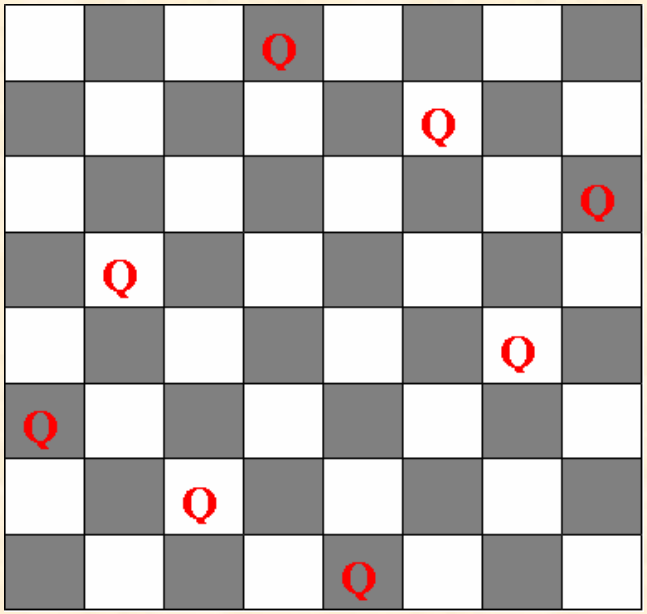
1. for color=1 to 3
2. c[k] ←color
3. if c为合法着色 then flag ←true and exit
4. else if c是部分着色 then *graphcolor*(k+1)
5. end for

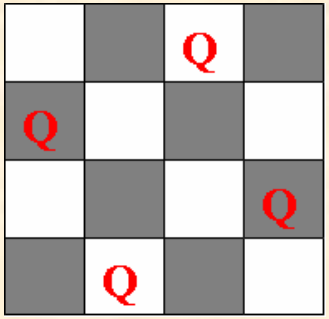
迭代回溯

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 ←1 to n
2. c[k] ←0
3. end for
4. flag ←false
5. k ←1 //start from v_1
6. while k≥1
7. while c[k]≤2 //为第k个顶点着色
8. c[k] ←c[k]+1
9. if c为合法着色then flag ←true and exit
10. else if c为部分着色then k ←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] ←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 ←k-1 //回溯到上一个顶点 (配合第6句的k≥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

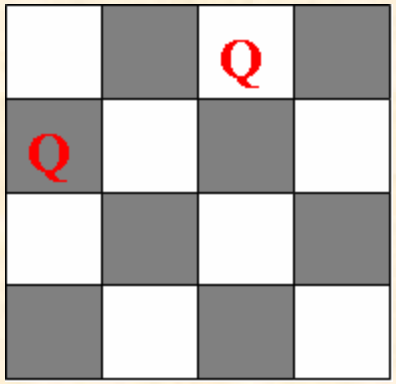
皇后问题

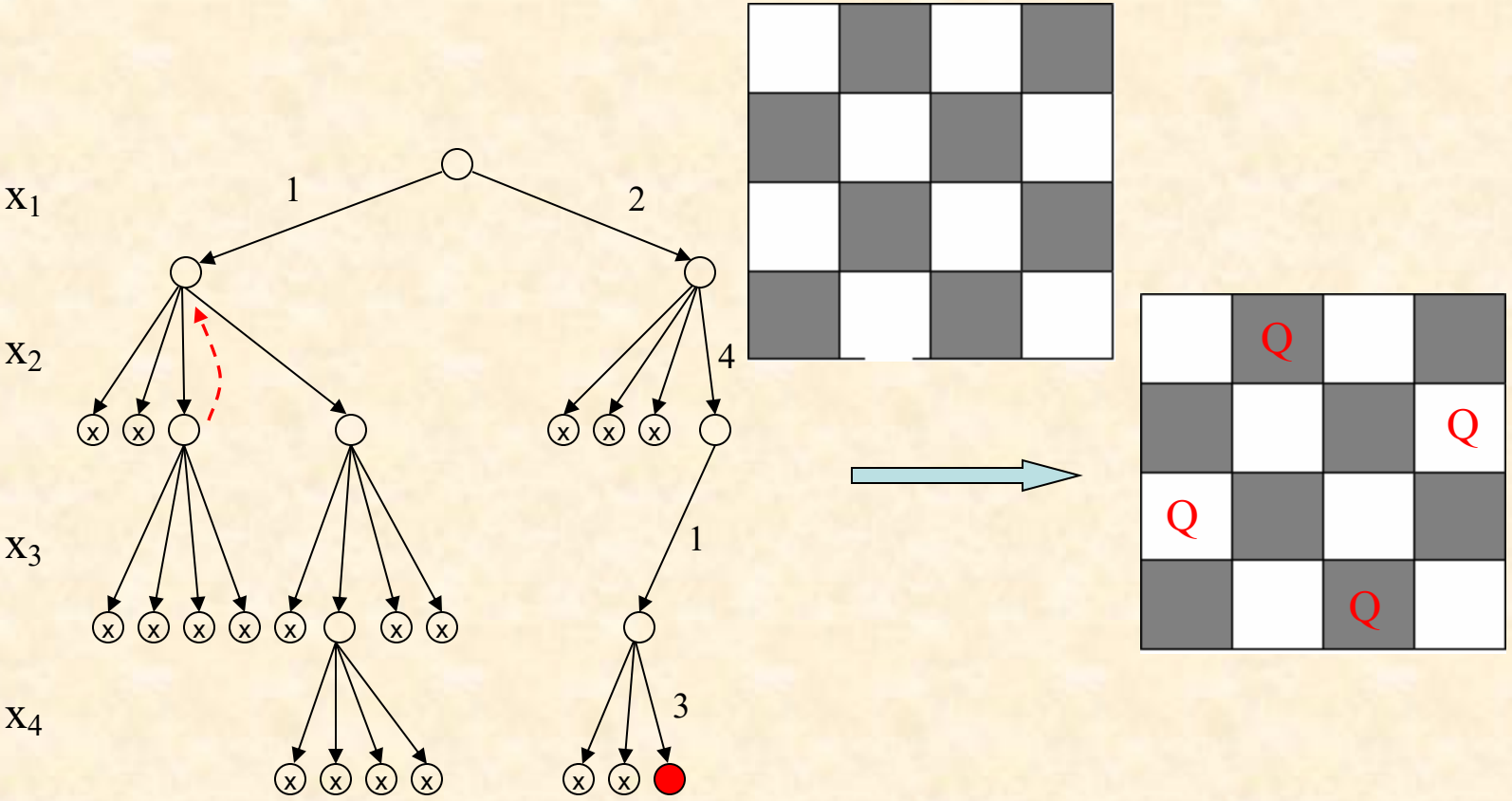
- 著名的数学家高斯在1850年提出：在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即：任意两个皇后都不能处于同一行、同一列或同一斜线上，问如何摆放？





- 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；

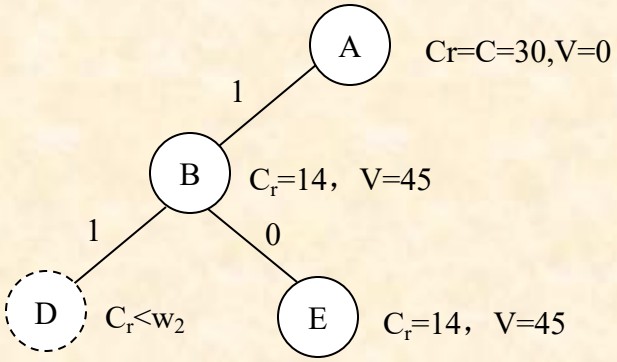




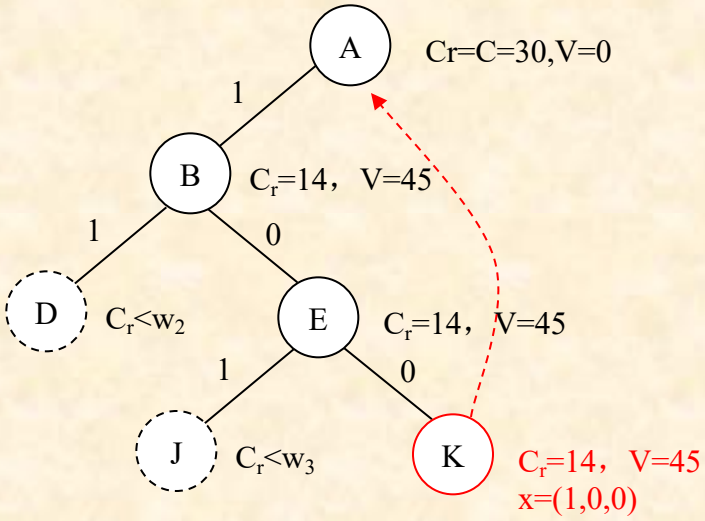
- 效率问题：蛮力方法候选解的个数是n!；而使用回溯法，候选解的个数是n^n，回溯的效率是否太低了？
- 然而，仔细分析可以发现，回溯法可以极大减少测试次数：
  - 例如，假设前两个皇后均放在了第一列，蛮力方法仍旧要测试(n −2)!个候选解；而回溯法只要进行一次测试就可以避免剩余无意义的测试。
  - 所以，尽管在最坏情况下要用$O(n^n)$时间来求解，然而大量实际经验表明，它在有效性上远远超过蛮力方法O(n!)。例如在4 皇后问题中，只搜索了341个节点中的27个就找到了解。

0-1背包问题的回溯求解 (示例)

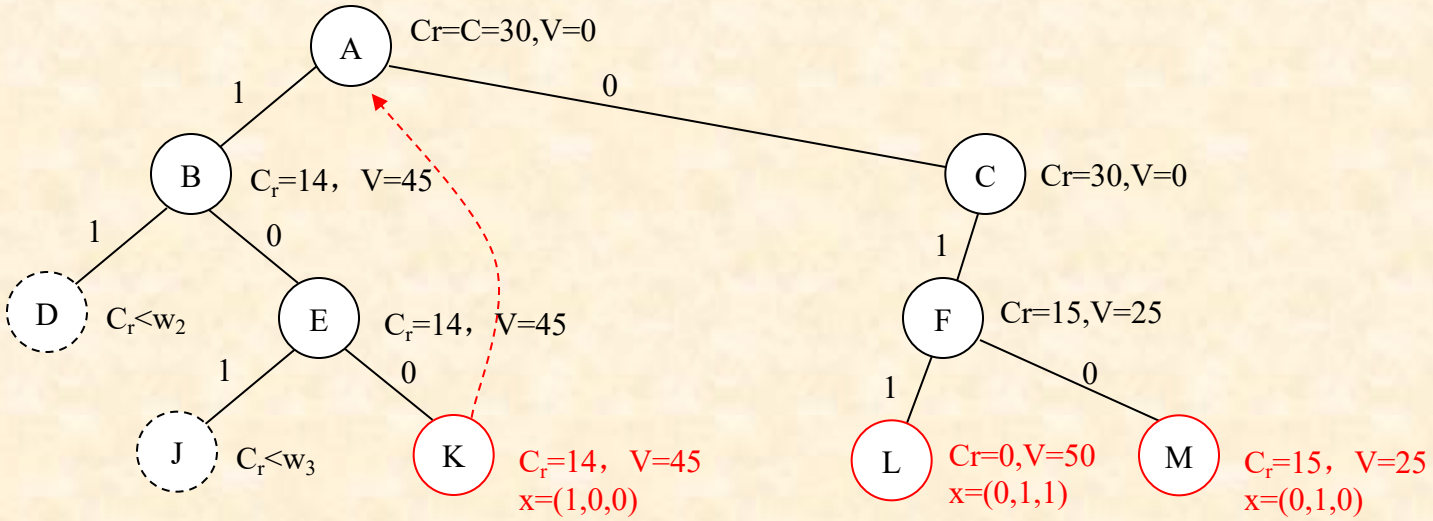
- 问题：n=3, C=30, w={16, 15, 15}, v={45, 25, 25}
- 开始时，C_r=C=30，V=0，A为唯一活节点，也是当前的扩展节点
- 扩展A，先到达B节点
  - $C_r=C_r-w_1$=14，$V=V+v_1$=45
  - 此时A、B为活节点，B成为当前扩展节点
- 扩展B，先到达D
  - $C_r<w_2$，D导致一个不可行解，尝试B的下一个元素
- 再扩展B到达E



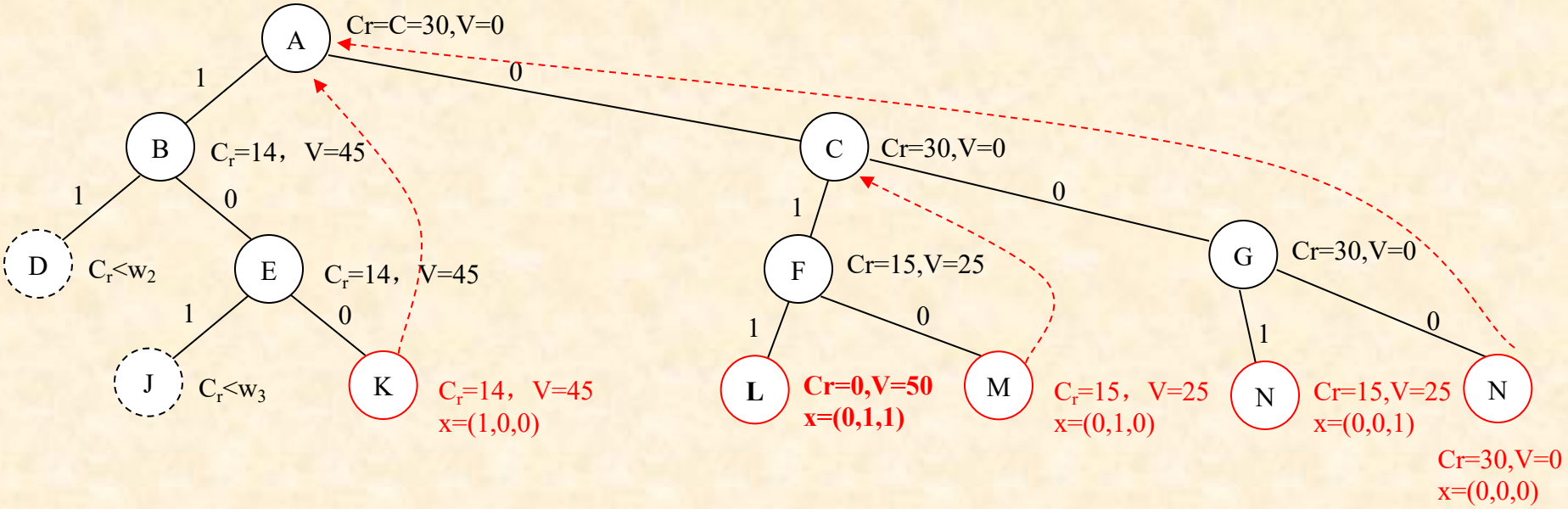
- 再扩展B到达E(续)
  - E可行，此时A、B、E是活节点，E成为新的扩展节点
  - 扩展E，先到达J
    - $C_r<w_3$，J导致一个不可行解，尝试E的下一个元素
  - 再次扩展E到达K
    - 由于K是叶节点，即得到一个可行解x=(1,0,0)，V=45
    - K不可扩展，成为死节点，返回到E
  - E没有可扩展节点，成为死节点，返回到B
- B没有可扩展节点，成为死节点，返回到A　(回溯到A)



- A再次成为扩展节点，扩展A到达C (尝试A的下一元素)
  - Cr=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



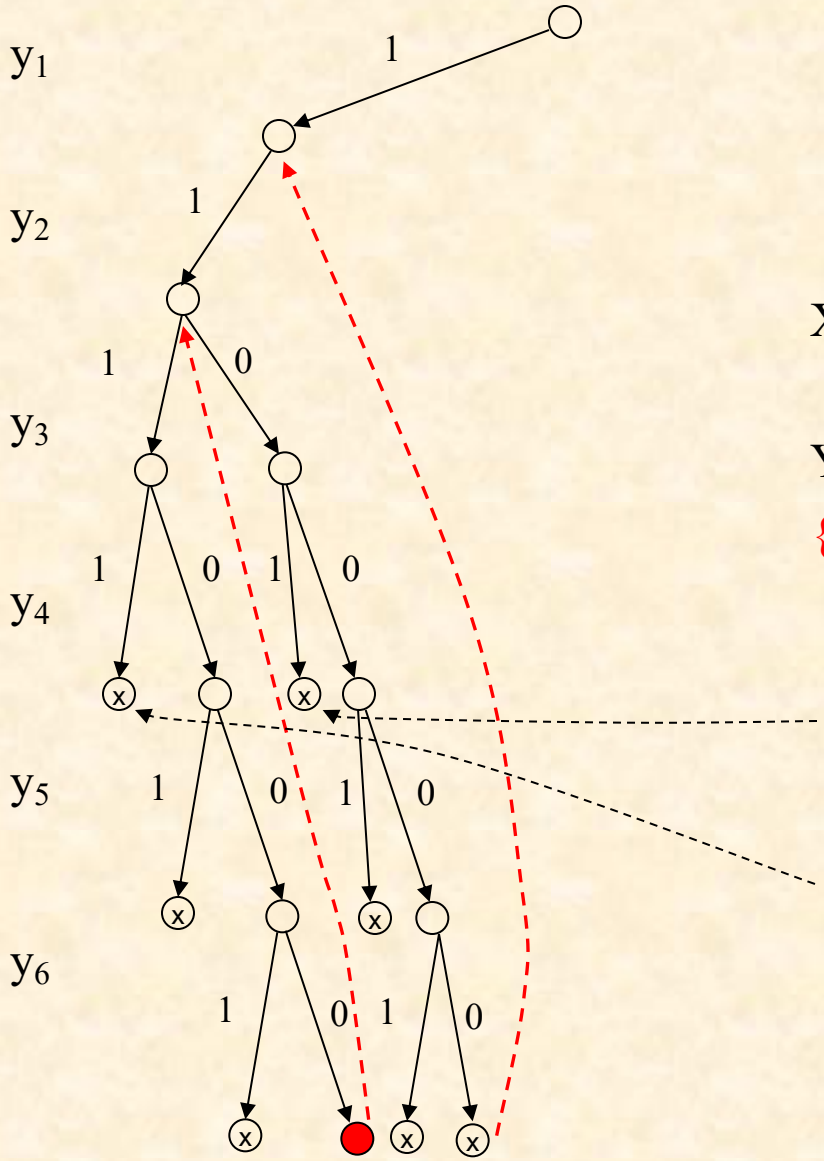
- 再扩展C到达G
  - C_r=30，V=0，活节点为A、C、G，G为当前扩展节点
  - 扩展G，先到达N，N是叶节点，且25<50，不是最优解，又N不可扩展，返回到G
  - 再扩展G到达O，O是叶节点，且0<50，不是最优解，又O不可扩展，返回到G
  - G没有可扩展节点，成为死节点，返回到C
- C没有可扩展节点，成为死节点，返回到A
- A没有可扩展节点，成为死节点，算法结束，最优解X=(0,1,1)，最优值V=50



- 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＝$\{x_1,x_2,\ldots x_n\}$，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}， 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}

10+20+40>60

10+20+30+40>60

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$(x_1,...,x_i)$，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:

$$X_1 \times X_2 \times \ldots \times X_n$$

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$(x_1, x_2,\ldots, x_j)$， 需要继续考虑 $v = (x_1, x_2,\ldots, x_j, x_{j+1})$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$v = (x_1, x_2,\ldots, x_j, x_{j+1})$ 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$X_1, X_2, \ldots, X_n$清楚或是隐含的描述
输出：解向量 $v = (x_1, x_2, \ldots, x_n)$， $0 \le i \le n$
1. v ← ()
2. flag ← false
3. advance(1)
4. if flag then output v
5. else output “no solution”

过程 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 flag ← true and exit
4. else if v是部分解 then advance(k +1)
5. end for

迭代回溯

输入：集合$X_1, X_2, \ldots, X_n$清楚或是隐含的描述
输出：解向量 $v = (x_1, x_2, \ldots, x_n)$， $0 \le i \le n$

```
1. v ← ()
2. flag ← false
3. k←1
4. while k≥1
5.     while Xk 没有被穷举
6.         xk←Xk 里的下一个元素；将xk加入v
7.         if v为最终解 then set flag ← true and exit
8.         else if v是部分解 then k←k+1 //前进
9.      end while
10.    重置Xk //一般将xk←0，表示将其重置
11.     k←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,…,n的所有排列。