课程编号：20006026

算法分析与设计

主讲教师：刘 瑶

电子科技大学信息与软件工程学院

第5章：回溯法

（Backtracking Algorithm）

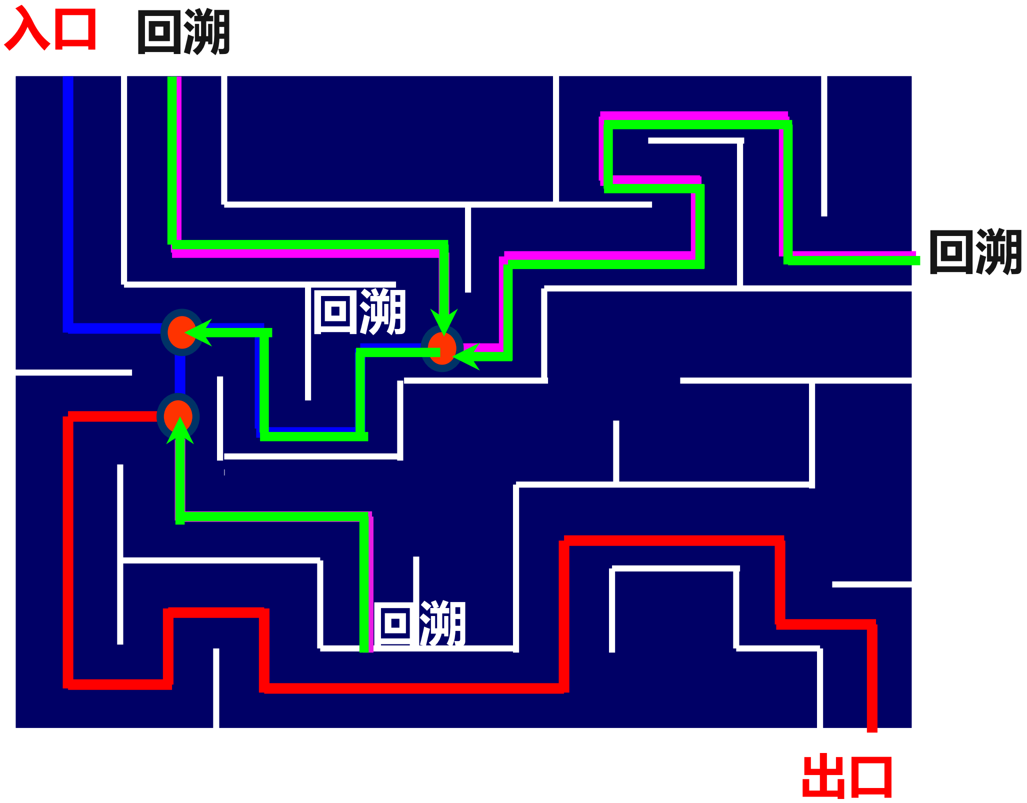
知识要点

- **掌握用回溯法解题的算法框架**
  - 子集树算法框架
  - 排列树算法框架
- **了解NP完全问题**
  - NP完全问题的定义和研究意义
- **通过应用范例学习回溯法的设计策略**
  - 0/1背包问题；旅行商问题（TSP）；最优装载问题
  - 批处理作业调度；连续邮资问题；圆排列问题
  - N-皇后问题；最大团问题；图的m着色问题

5.1 回溯法算法框架

（Backtracking Algorithm Paradigm）

- 迷宫游戏



搜索算法

- 对某些问题建立数学模型时，即使有一定的数学模型，但采用数学方法解决有一定的困难。对于这一类试题，我们用模拟或搜索求解。

- 在缺乏解决问题的有效模型时，搜索却是一种行之有效的解决问题的基本方法。

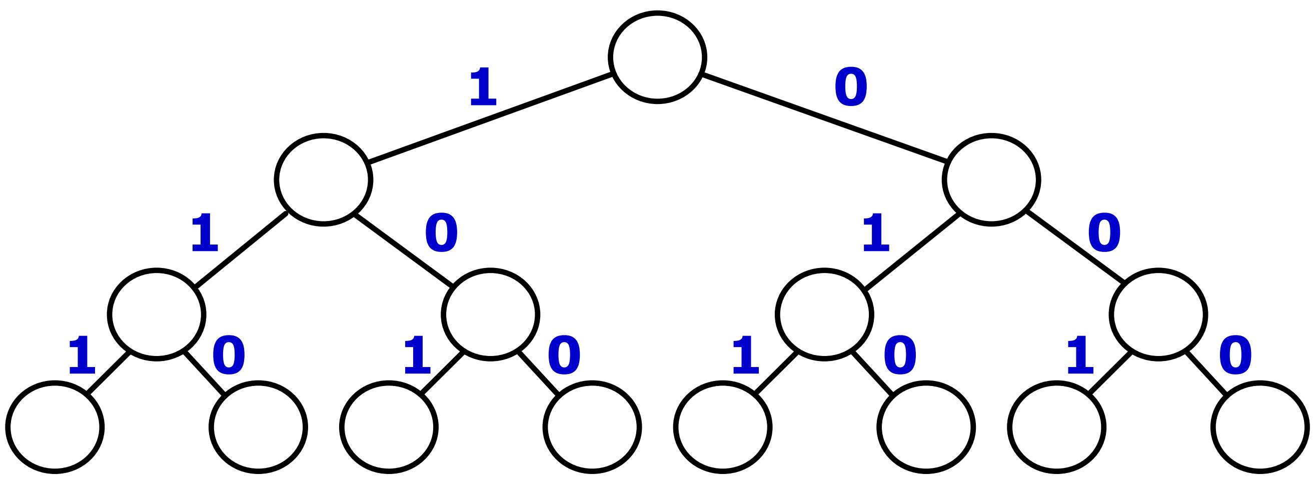
  枚举法（穷举法）
  深度优先搜索（回溯法）
  广度优先搜索

回溯法的基本概念

- 回溯法是一种选优搜索法（试探法），被称为通用的解题方法
- 基本思想：将n元问题P的状态空间E表示成一棵高为n的带权有序树T，把在E中求问题P的解转化为在T中搜索问题P的解
- 解题方法：按选优条件对T进行深度优先搜索，以达到目标
  - 从根结点出发深度优先搜索解空间树
  - 当探索到某一结点时，要先判断该结点是否包含问题的解
    - 如果包含，就从该结点出发继续按深度优先策略搜索
    - 否则逐层向其祖先结点回溯（退回一步重新选择）
    - 满足回溯条件的某个状态的点称为“回溯点”
  - 算法结束条件：
    - 求所有解：回溯到根，且根的所有子树均已搜索完成
    - 求任一解：只要搜索到问题的一个解就可以结束

问题的解空间

- 应用回溯法解题时，首先应明确问题的解空间
  - 问题的解空间应至少包含该问题的一个（最优）解
  - 例如：对于有n种备选物品的0/1背包问题而言
    - 解空间可以由长度为n的向量来表示
    - 显然：该解空间包含了对该问题所有可能的解法
- 定义了问题的解空间后，可以将其组织成树或图的形式
  - 例如：n = 3 的0/1背包问题，解空间可用一棵完全二叉树表示
    - 从根到任一叶结点的路径表示解空间的一个元素



生成问题状态的基本方法

- 基本概念
  - 扩展结点：一个正在产生子结点的结点称为扩展结点
  - 活结点：一个自身已生成但其子结点尚未全部生成的结点
  - 死结点：一个所有子结点已经产生的结点称做死结点
- 深度优先的问题状态生成法
  - 对一个扩展结点R，一旦产生了它的一个子结点C
    - 则将其作为新扩展结点，并对以C为根的子树进行穷尽搜索
    - 在完成对子树C的穷尽搜索后，将R重新变成扩展结点
    - 继续生成R的下一个子结点，若存在，则对其进行穷尽搜索
- 宽度优先的问题状态生成法
  - 在一个扩展结点变成死结点之前，它一直是扩展结点

回溯法的解题思路

- 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
- 从根结点开始深度优先搜索解空间（利用**剪枝**避免无效搜索）
  - 此时：根结点成为活结点，并成为当前的扩展结点
  - 进一步的搜索从当前扩展结点开始
    - 向纵深方向移至一个新结点
    - 该新结点成为新的活结点，并成为当前扩展结点
  - 若在当前扩展结点处不能再向纵深方向移动
    - 则当前扩展结点变为死结点
    - 此时应回溯至最近的活结点，将其作为当前扩展结点
- 回溯法以这种方式递归地在解空间中搜索
  - 直至找到所要求的解，或者解空间中已经没有活结点为止

递归回溯：通用算法框架

- 回溯法对解空间作深度优先搜索
  - 因此在一般情况下用递归方法实现回溯法

```
void backtrack (int t) {
    if (t > n){
        output(x);
    }
    else{
        for (int i = f(n, t); i <= g(n, t); i++) {


        }
    }
}
```

t表示递归深度
即当前扩展结点在解空间树中的深度
n用来控制递归深度
即解空间树的高度
表示已到达叶子结点
Output(x)
对可行解进行处理：记录或输出

递归回溯：通用算法框架

- 回溯法对解空间作深度优先搜索
  - 因此在一般情况下用递归方法实现回溯法

```
void backtrack (int t) {
    if (t > n){
        output(x);
    }
    else{
        for (int i = f(n, t); i <= g(n, t); i++) {
            x[t] = h(i);
            if (constraint(t) && bound(t)){
                backtrack(t+1);
            }
        }
    }
}
```

h(i)表示在当前扩展结点处x[t]的第i个可选值

f(n, t)表示在当前扩展结点处未搜索过的子树的起始编号

g(n, t)表示在当前扩展结点处未搜索过的子树的终止编号

constraint(t)为true表示在当前扩展结点处x[1:t]的取值满足问题的约束条件

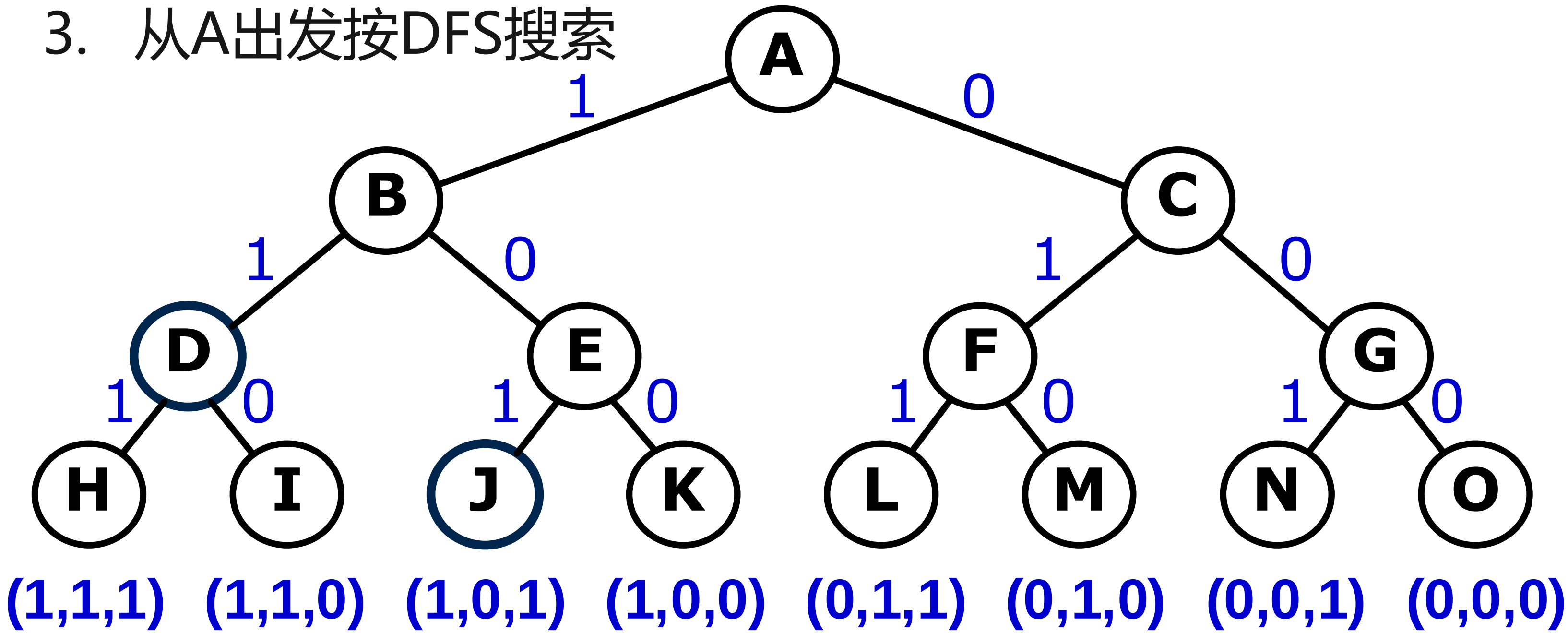
bound(t)为true表示在当前扩展结点处x[1:t]的取值尚未导致目标函数越界

两类常见的解空间树

- **用回溯法解题时常用到两种典型的解空间树：子集树与排列树**
- **第一类解空间树：子集树**
  - 当问题是：从n个元素的集合S中找出满足某种性质的子集时
  - 相应的解空间树称为子集树，例如n个物品的0/1背包问题
    - 这类子集树通常有2^n个叶结点
    - 解空间树的结点总数为$2^{n+1}-1$
    - 遍历子集树的算法需$\Omega(2^n)$计算时间
- **第二类解空间树：排列树**
  - 当问题是：确定n个元素满足某种性质的排列时
  - 相应的解空间树称为排列树，例如旅行商问题
    - 排列树通常有n！个叶结点
    - 因此遍历排列树需要$\Omega(n!)$计算时间

子集树示例： 0/1背包问题

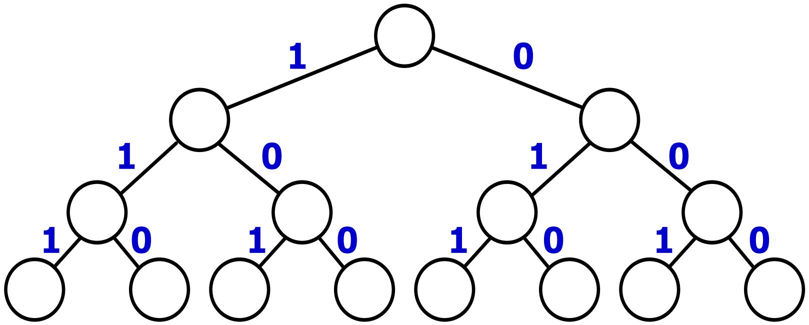
- 设：n=3, w=(16, 15, 15), v=(45, 25, 25), c=30
  1. 定义解空间：X={(0,0,0), (0,0,1), (0,1,0),..., (1,1,0), (1, 1, 1) }
  2. 构造解空间树如图
  3. 从A出发按DFS搜索



可行解的值： 45 50 25 25 0

最优解： x = (0, 1, 1) 最优值：m = 50

子集树回溯算法框架

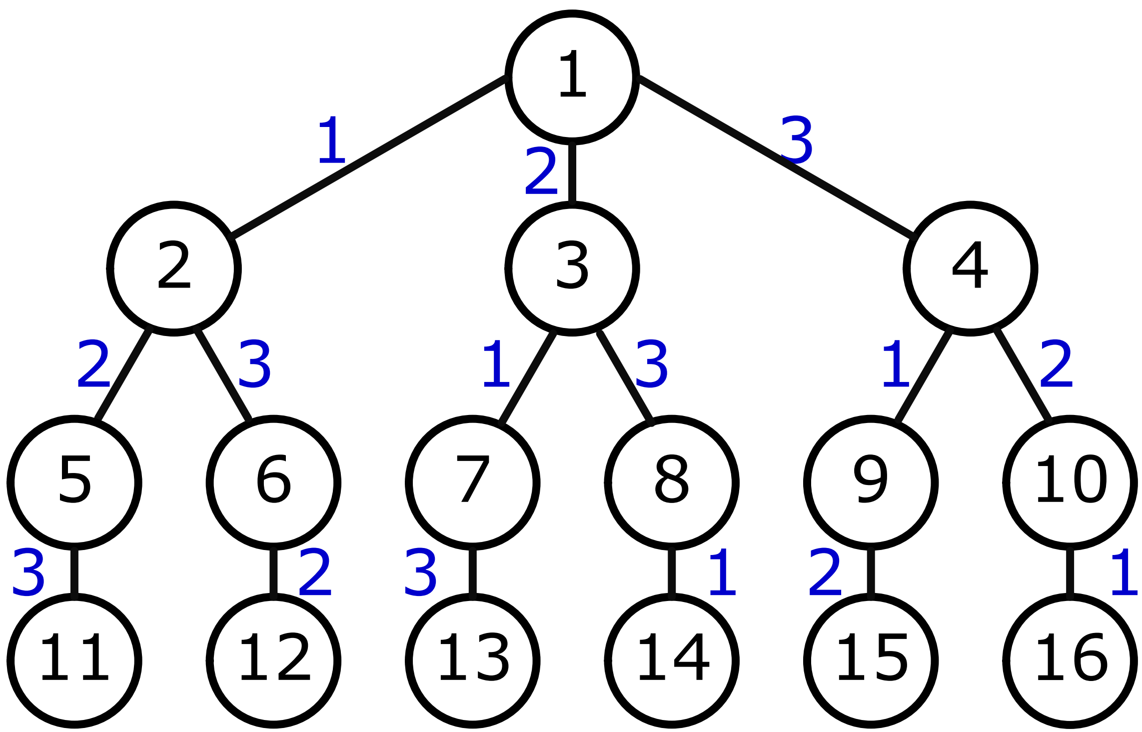



```
void backtrack (int t) {
    if (t > n){
        output(x);
    }
    else{
        // 对当前扩展结点的所有可能取值进行枚举
        for (int i = 0; i <= 1; i++) {
            x[t] = i;
            if (constraint(t) && bound(t)) backtrack(t+1);
        }
    }
}  // 执行时，从Backtrack(1)开始
```

遍历子集树：$O(2^n)$

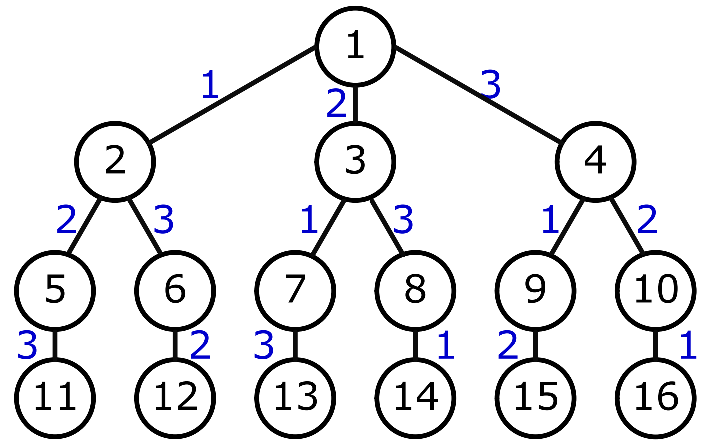
排列生成问题

- 通过排列生成问题帮助理解排列树回溯算法框架
- 问题定义：给定正整数n，要求生成1, 2, ..., n的所有排列
- 示例：n=3，解空间树如下图所示



排列树回溯算法框架

```
void backtrack (int t) {
    if (t > n){
        output(x);
    }
    else{
        for (int i = t; i <= n; i++) {
            swap(x[t], x[i]);
            if (constraint(t) && bound(t))  backtrack(t+1);
            swap(x[t], x[i]);
        }
    }
} // 调用Backtrack(1)前，首先将数组x初始化为单位排列[1,2, …, n]
```



遍历排列树：O(n!)

排列生成问题的回溯算法

```
void backtrack (int t) {
    if (t > n) output(x);
    else{
        for (int i = t; i <= n; i++) {
            swap(x[t], x[i]);
            backtrack(t+1);
            swap(x[t], x[i]);
        }
    }
}
main(int n){
    for (int i=1; i <= n; i++) x[i] = i;
    backtrack(1);
}
```

算法输出：

123

132

213

231

312

321

回溯法的特点

- 回溯法解题思路小结
  - 该方法的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，算法只保存从根结点到当前扩展结点的路径
  - 如果解空间树中从根结点到叶结点的最长路径的长度为h(n)
    - 则回溯法所需的内存空间通常为：$O(h(n))$
    - 而显式地存储整个解空间则需要：$O(2^{h(n)})$或$O(h(n)!)$

回溯法的特点

- 常用剪枝函数
  - 约束函数：在扩展结点处剪去不满足约束的子树
  - 限界函数：剪去得不到最优解的子树
- 约束函数
  - 回溯法要求问题的解能够表示成一个n元向量形式$(x_1,x_2,\ldots,x_n)$
  - 显式约束：对分量 x_i 的取值范围限制
  - 隐式约束：为满足问题的解而对不同分量之间施加的约束
- 限界函数（bounding function）
  - 为了避免生成那些不可能产生最佳解的问题状态
  - 要不断地利用限界函数来剔除那些不能产生所需解的活结点
  - 具有限界函数的深度优先生成法就称为回溯法

5.2 NP完全性问题简介

（Introduction to NP-Complete）

算法理论的研究对象：两类抽象问题

- **优化问题（也称为极值问题）**
  - 一个优化问题通常可以用以下四个部分来描述
    - 实例集合：若干实例I组成集合D，其中每一个实例I含有一个问题所有输入的数据信息
    - 可行解集：每一个实例I有一个解集合S(I)，其中的每一个解都满足问题的条件，称为可行解
    - 目标函数：映射$c(\sigma)$: $S(I)\rightarrow\Re$
    - 最优化：求最优解 $\sigma_{opt}(I) \in S(I)$，使得对任意一个可行解$\sigma\in S(I)$，都有$c(\sigma_{opt}(I)) \geq c(\sigma)$ 或者$c(\sigma_{opt}(I)) \leq c(\sigma)$
  - 一个优化问题也可以视为一个判定问题

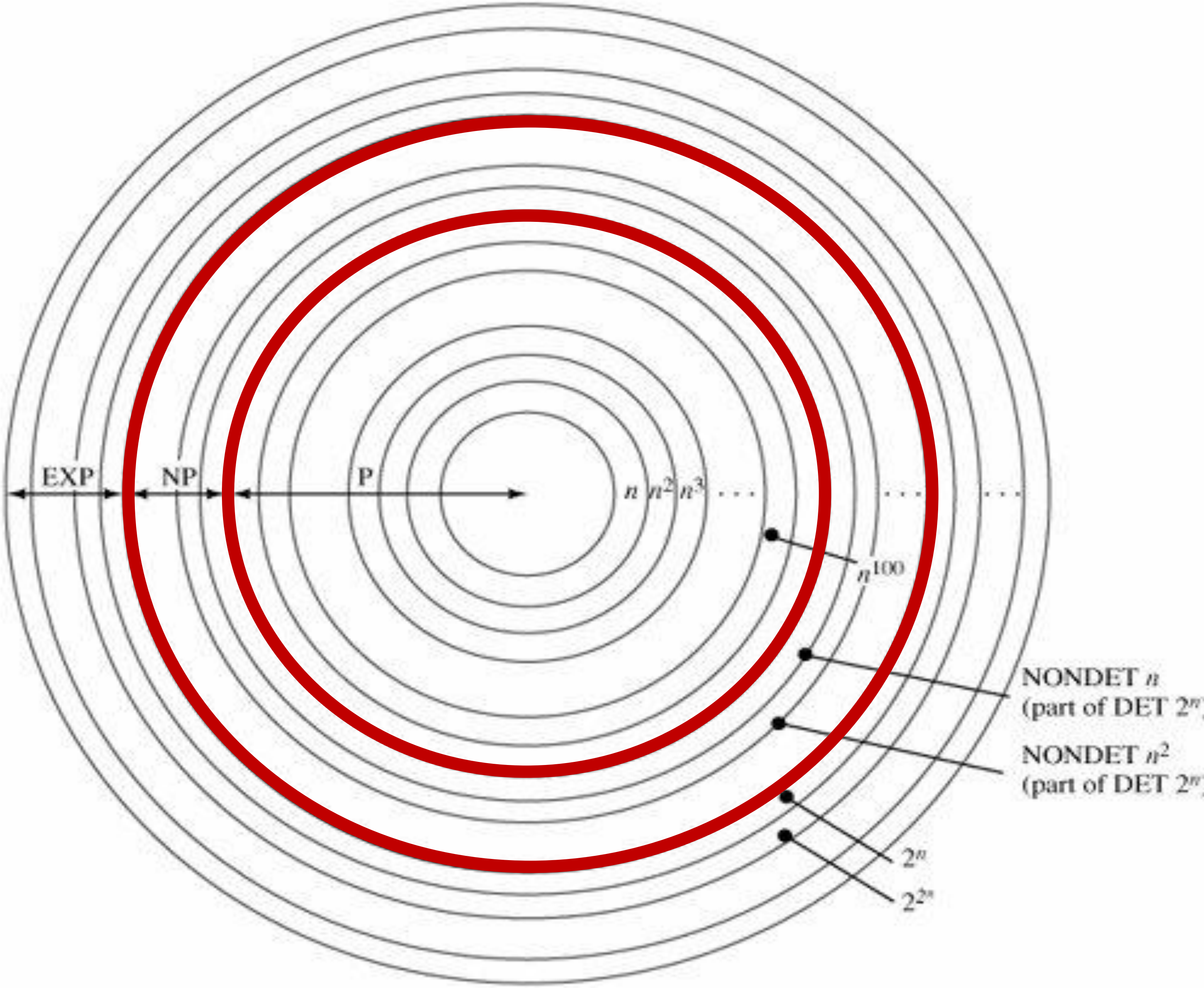
算法理论的研究对象：两类抽象问题

- **判定问题（也称为识别问题）**
  - 仅有两种可能的答案：“是”或者“否”
  - 可以将一个判定问题视为一个函数
    - 它将问题的输入集合I映射到问题解的集合{0 1}
  - 以路径判断问题为例：
    - 给定一个图G=(V, E) 和顶点集V中的两个顶点u, v
    - 判断 G 中是否存在一条路u和v之间的路
    - 如果用 i=<G, u, v>表示该问题的一个输入
      - 则：函数PATH(i)=1 （当u和v之间存在一条路时）
      - 则：函数PATH(i)=0 （当u和v之间不存在一条路时）

P和NP

- **P和NP都是问题的集合**
  - P是所有可在多项式时间内用确定算法**求解**的判定问题的集合
    - 对于一个问题X，若存在一个算法XSolver，能在$O(n^k)$时间内求解（k为某个常数），那么就称这个问题属于P
  - NP是所有可用多项式时间算法**验证**其猜测准确性的问题的集合
    - 对于一个问题X，若存在一个算法XChecker，能在多项式时间复杂度内给出验证结果，那么就称这个问题属于NP
  - 显然：　$P \subseteq NP$
  - 数学的世纪难题，计算机科学领域的顶级难题：　**P ＝ NP ?**
    - 目前的研究结果倾向于认为：P!=NP
    - 即：有些问题就是（不可快速计算的）难处理的问题
    - 通常将可以由多项式时间的算法解决的问题看作是易处理的

计算复杂性的层次结构


EXP
NP
P
n n^2 n^3 . . .
n^{100}
NONDET n
(part of DET 2^n)
NONDET n^2
(part of DET 2^n)
2^n
2^{2^n}

NPC：NP-Complete

- **NP-Complete的非形式化定义**
  - 如果一个问题属于NP
  - 且该问题与NP中的任何问题是一样难（hard）的
  - 则称该问题属于NPC，或称之为NP完全的（ NP-Complete ）
- **研究NP完全问题的意义**
  - NPC问题是20世纪的最伟大的发现之一
    - 1971年，Cook发现所有的NP问题都可以规约到SAT问题
    - 1972年，Karp证明了21种问题是NP完全的
  - 直接推论：**如果任何一个NPC问题可以在多项式时间内解决**
    - **则NP中的所有问题都有一个多项式时间的算法**
  - 迄今尚未发现任何一个NPC问题的多项式时间解决方案

NP完全性的证明

- **如何证明一个问题属于NPC类?**
  - 证明一个问题是NP完全问题时（目的是证其困难）
    - 不是要证明存在某个有效的算法
    - 而是要证明不太可能存在一个有效的算法
  - 证明的方法依赖于三个关键概念：
    - **判定问题**：NP完全性只适用于判定问题
    - **规约**： NP完全性的定义和证明方法
    - **第一个NP完全问题**：应用规约技术的前提
      - 已知一个NP完全的问题
      - 才能通过规约的方法证明另一个问题也是NP完全的
      - 第一个已知NPC问题是电路可满足性问题（SAT）

问题的规约

- **对于给定的判定问题A，希望在多项式时间内解决该问题**
  - 称某一特定问题的输入为该问题的一个实例（instance）
  - 假设有另一个不同的判定问题B可以在多项式时间内求解
  - 假设有如下过程，可以将A的任意实例α转化为B的实例β
    - 转化操作需要多项式时间
    - 两个实例的答案相同（ α的答案为真 **if** β的答案为真）
  - 称该过程为多项式时间的**规约算法**（reduction algorithm）
  - 并且它提供了一种在多项式时间内解决问题A的方法
    1. 首先利用规约算法将A的实例α转化为B的实例β
    2. 然后对实例β运行B的多项式时间判定算法
    3. 最后将β的答案作为α的答案
    - 由于每一步只需多项式时间，因此判定α只需多项式时间

问题的规约（续）

- **证明某一问题是NP完全的**
  - 通过将对问题A的求解“规约”为对问题B的求解
    - 就可以利用B的“易求解性”来证明A的“易求解性”
  - 证明问题是NPC的思路恰恰与之相反
    - 利用规约表明对特定问题而言不存在多项式时间的算法
  - 设：已知判定问题A不可能存在多项式时间的算法
    - 并设有一个多项式时间的规约将A的实例转化为B的实例
    - 则可以利用反证法证明B不可能存在多项式时间的算法
      - 相反假设：B有一个多项式时间的算法
      - 则根据规约算法：可以在多项式时间内解决问题A
      - 显然与已知矛盾（判定问题A没有多项式时间的算法）
  - 注意：无法假设问题A绝对没有多项式时间的算法

NPC和NPH

- **NP-Complete的形式化定义**
  1. 如果一个判定问题A属于NP
  2. 而且NP中的任何问题均可在多项式时间内规约到A
  - 则称问题A是NP完全的（NP-Complete）
  - 判断A是否属于NP类可以看其解是否可在多项式时间内被验证
- **NP-Hard的形式化定义**
  - 如果一个问题B满足上述条件2，则称之为NP-Hard问题
  - 也就是说
    - 无论问题B是否属于NP类（是否满足条件1）
    - 若某一NPC问题可在多项式时间内规约到B
    - 则称问题A是NPH问题（NP-Hard）

一些经典的NP问题

- **经典NPC问题**
  - **SAT问题**：对于输入的包含n个布尔变量的逻辑表达式，求解使表达式为真的变量值组合
  - **背包问题**：给定背包容量C和n件物品及其重量，求解物品选取方案，使得选出的物品重量之和恰好为C
  - **旅行商问题**（最优）：对于输入的包含n个点的带权完全图，要求输出一条遍历了所有顶点的总权值和最小的路径
  - **n皇后问题**：对于输入的n，要求输出一个在nxn的国际象棋棋盘上放置了n个互不攻击的皇后的方案
  - **精确覆盖问题**：对于输入0/1矩阵，要求输出矩阵的若干个行号，使得输入的0/1矩阵只保留输出的行后每列正好有一个1
- **NP-Hard问题示例**
  - **旅行商问题**：对于输入的包含n个点的带权完全图和一个正实数c，要求输出一条遍历了所有点的总权值和不超过c的路径

NP完全性小结

- **一个判定问题A是NP完全的**
  - 如果问题A属于NP类
  - 而且NP中的任何问题均可在多项式时间内规约到A
- **研究NP完全问题的意义**
  - 如果任何一个NPC问题可以在多项式时间内解决
    - 则NP中的所有问题都有一个多项式时间的算法
    - 因此：NPC问题是计算机科学领域最引入注目的问题
  - 要成为一名优秀的算法设计者，熟悉这类问题是非常重要的
    - 很多自然有趣的问题并不比图的搜索等问题更困难
    - 目前已经证明的NP完全问题高达上千种
    - 迄今尚未发现任何一个NPC问题的多项式时间解决方案
    - 如果问题是NPC的，更好的做法是采用**近似算法**求解

搜索算法简介

- **穷举搜索（ brute-force ）**
- **盲目搜索（blind search)**
  - 深度优先（DFS）： 回溯法(Backtracking)
  - 广度优先搜索（BFS）： 分支限界法(Branch & Bound)
  - 博弈树搜索（game-tree）： α-β剪枝算法
- **启发式搜索（heuristic search)**
  - A*算法：IDA*算法， B*，局部择优搜索法，最好优先搜索法
  - 仿生算法：蚁群算法，蜂群算法，禁忌算法，粒子群算法
  - 进化计算：遗传算法（ 1975 ）
  - 随机搜索：将随机过程引入搜索
    - 随机梯度下降算法，随机爬山算法
    - 模拟退火算法（ 1983 ） ，量子退火算法 ……

算法之美

- **大道至简：简单就是美**
  - 爱因斯坦质能方程：$E=mc^2$　（1905）
  - 冯·诺依曼体系结构（1946）
    - 将指令和数据同时存放在存储器中
    - 计算机组成：控制器、运算器、存储器、输入、输出设备
  - 递归：PageRank algorithm（ Larry Page ， 1998 ）

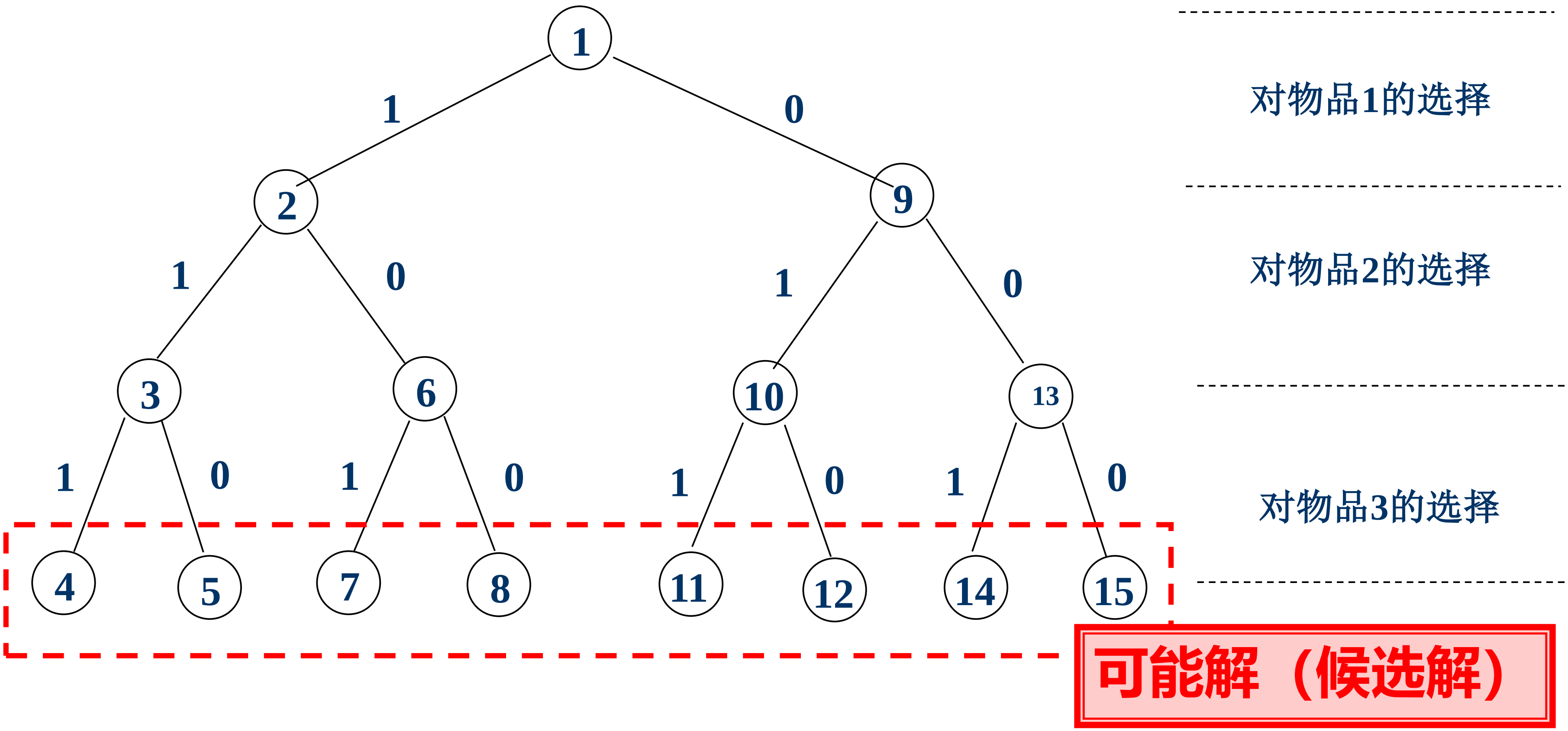
$$\mathrm{PR}(p_i)=\frac{1-\alpha}{n}+\alpha\sum_{p_j\in N(p_i)}\frac{\mathrm{PR}(p_j)}{D(p_j)}$$

  - 动态规划：Viterbi algorithm（Andrew Viterbi，1967）

小结： 回溯法的算法框架

- **处理一个复杂的问题，常常会有很多可能解，这些可能解构成了问题的解空间。解空间也就是进行穷举的搜索空间，所以，解空间中应该包括所有的可能解。**
- **举例：0-1背包问题。对于有n=3个物品的0/1背包问题，其可能解的表示方式可以为：**
  - 可能解由一个等长向量$\{x_1, x_2, \ldots, x_n\}$组成，其中$x_i=1(1\le i\le n)$表示物品$i$装入背包，$x_i=0$表示物品$i$没有装入背包，当$n=3$时，其解空间是：{(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1) }

ᨀ对于n=3的0/1背包问题，其解空间树如图所示，树中的8个**叶子结点**分别代表该问题的8个**可能解**。



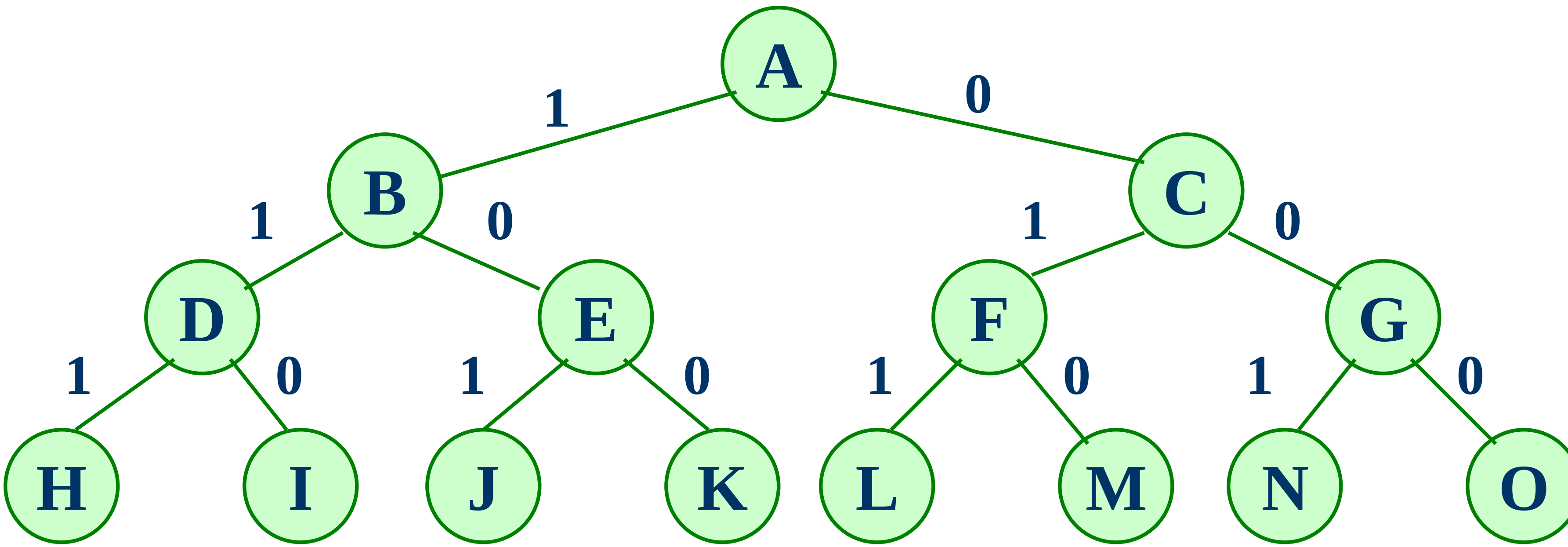
结论1：

- 对于一个解结构形式为n元组的问题，其可能解构成了问题的**解空间**，可以用一颗**解空间树**表示，树的**所有叶子结点为可能解**。

回溯法的基本思想

- 回溯法又被称为“通用解题法”。
- 回溯法在包含问题所有解的解空间树中，按深度优先策略，从根结点出发搜索解空间树。
- 算法搜索至解空间树的任意一结点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

0-1背包问题的解空间



–**回溯法在求问题**<u>**所有解**</u>时，要回溯到根，且根结点的所有子树都要被遍历。

–**回溯法在求问题**<u>**任一解**</u>时，只要搜索到问题的一个解就可以结束。

结论2：

- **回溯法是一种按深度优先策略，从根结点出发搜索解空间树的穷举式搜索法。**
- **为了提高搜索效率，则需要按照某种策略，能避免不必要搜索过程。**
  - 放弃当前候选解（放弃搜索该候选解为根的子树），寻找下一个候选解（搜索该候选解兄弟为根的子树）的过程称为回溯。
  - 扩大当前候选解的规模（继续向下搜索该候选解的子孙），以继续试探的过程称为向前试探。

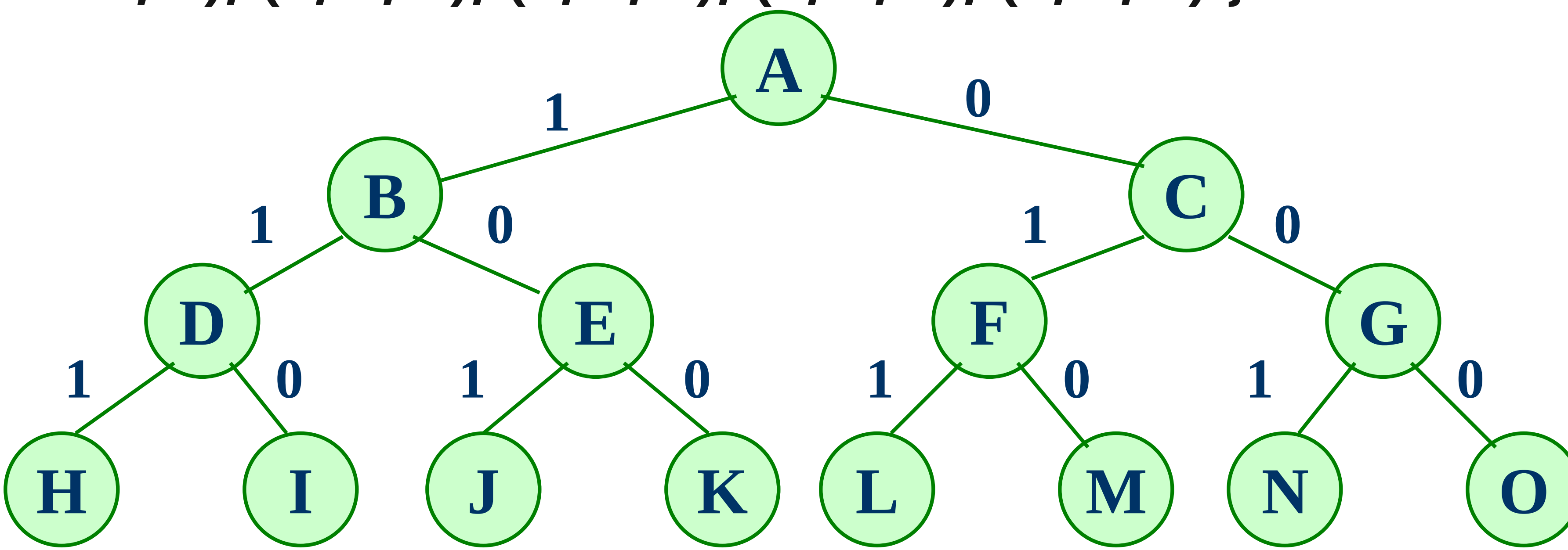
- 问题描述中给出用于判定一个候选解是否是可行解的约束条件，满足约束条件的候选解称为可行解。同时还给定一个数值函数称为目标函数，用于衡量每个可行解的优劣。
- 约束条件可分成两类：显式约束和隐式约束。

举例：0-1背包问题

- 可能解由一个等长向量$\{x_1, x_2, …, x_n\}$组成，其中$x_i=1(1\leq i\leq n)$表示物品i装入背包，$x_i=0$表示物品i没有装入背包。

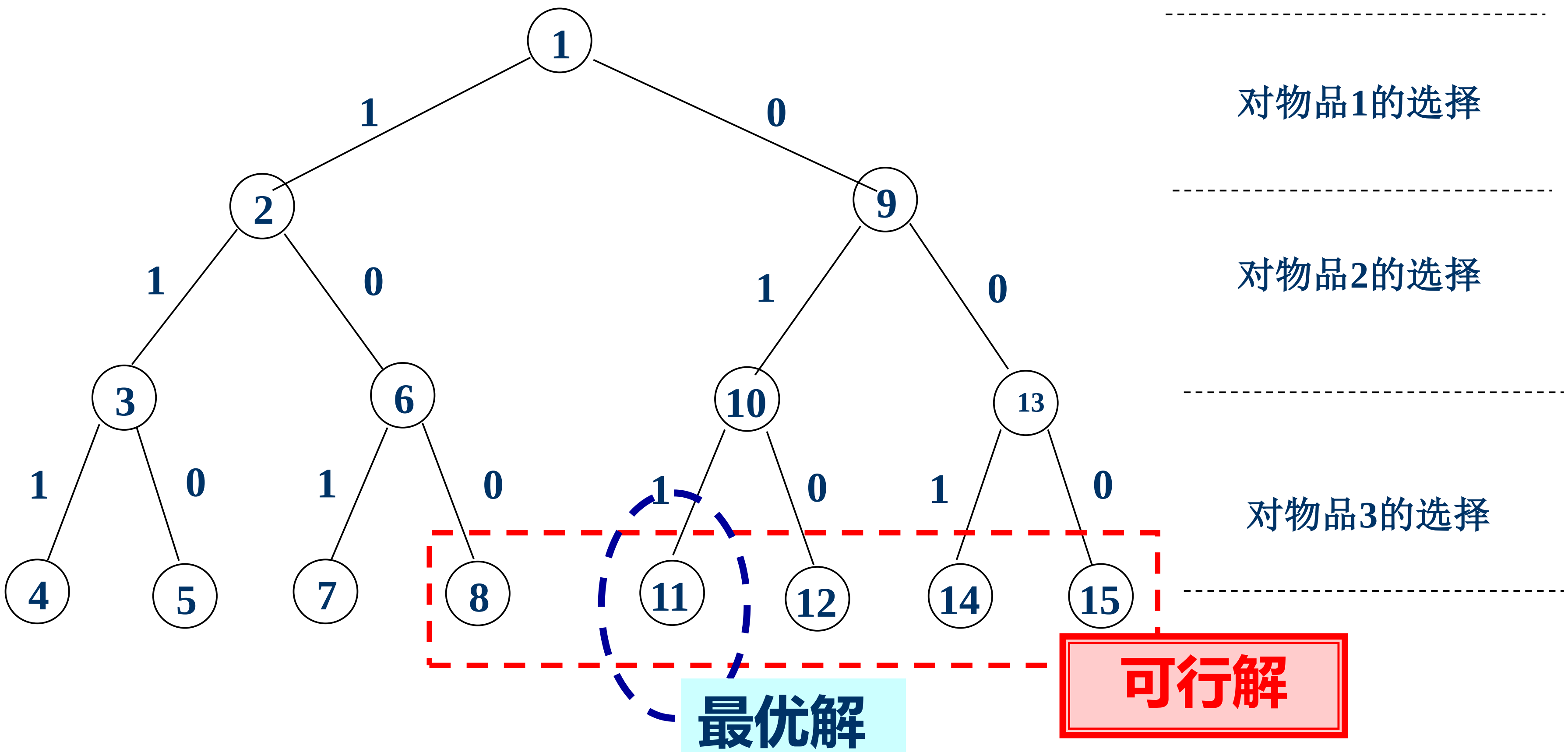
显式约束

- 当$n=3$时，其解空间是：{(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1) }



- **目标函数**，也称**代价函数**（cost function）用来衡量每个可行解的优劣，使目标函数取最大（或最小）值的可行解为问题的**最优解**。

$$V = \sum v_i \times x_i$$



- 如果所求解的是最优化问题，还必须用目标函数衡量每个答案结点，从中找出使目标函数取最优值的最优答案结点。
- 扩展结点：一个正在产生儿子的结点称为扩展结点。
- 活结点：一个自身已生成但其儿子还没有全部生成的结点称做活结点。
- 死结点：一个所有儿子已经产生的结点称做死结点。
- 回溯法：为了避免生成那些不可能产生最佳解的问题状态，要不断地利用限界函数（隐式约束）来处死那些实际上不可能产生所需解的活结点，以减少问题的计算量。

结论3：

- 结点的状态分为：活结点、扩展结点和死结点。
- 叶子结点为可能解，最终的问题解来自于从根到某个叶子结点（解状态）的路径。

- 约束函数和限界函数的目的相同，都是为了剪去不必要搜索的子树，减少问题求解所需实际生成的状态结点数，它们统称为剪枝函数（pruning function）。
- 使用剪枝函数的深度优先生成状态空间树中结点的求解方法称为回溯法（backtracking）；
- 广度优先生成结点，并使用剪枝函数的方法称为分枝限界法（branch-and-bound）。

回溯法的递归形式的一般框架

```
void backtrack (int t)//当前扩展结点在解空间树中的深度
{
      if (t>n) output(x);//已搜索至叶节点，得到可行解x
      else//当前扩展结点处未搜索过的子树的起始编号和终止编号
        for (int i=f(n,t);i<=g(n,t);i++) {
          x[t]=h(i); //当前扩展结点处x[t]的第i个可选值
          if (constraint(t)&&bound(t)) backtrack(t+1);
          }
}
```

回溯法迭代形式的一般框架

```
void iterativeBacktrack ()
{
  int t=1;
  while (t>0) {
    if (f(n,t)<=g(n,t))
      for (int i=f(n,t);i<=g(n,t);i++) {
        x[t]=h(i);
        if (constraint(t)&&bound(t)) {
          if (solution(t)) output(x);
          else t++;} }
    else t--; }//回溯
}
```

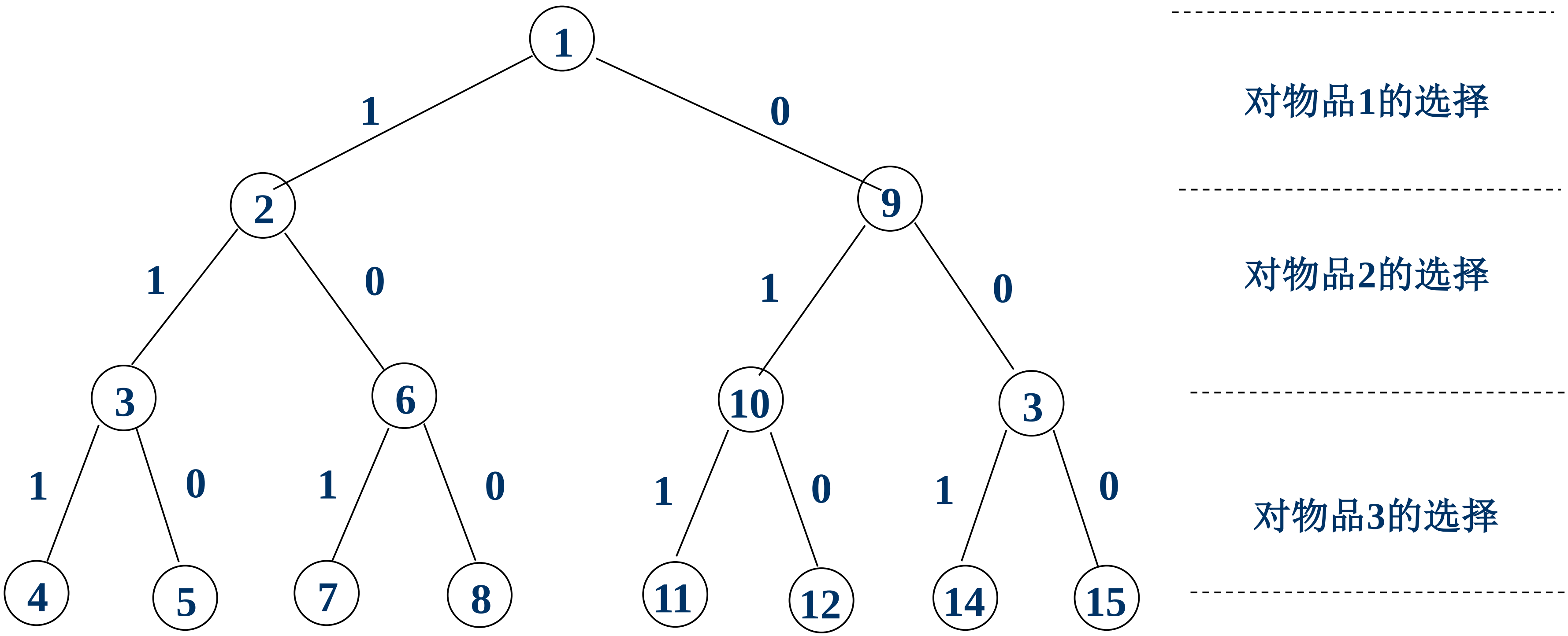
回溯法的时间性能分析

- 一般情况下，在问题的解向量$X=(x_0, x_1, \ldots, x_{n-1})$中，分量$x_i$ $(0\le i<n)$的取值范围为某个有限集合$S_i=\{a_{i_1}, a_{i_2}, \ldots, a_{i_{ri}}\}$；
- 问题的解空间由笛卡儿积$A=S_0\times S_1\times\ldots\times S_{n-1}$构成，并且第1层的根结点有$|S_0|$棵子树，则第2层共有$|S_0|$个结点；第2层的每个结点有$|S_1|$棵子树，则第3层共有$|S_0|\times|S_1|$个结点。
- 依此类推，第n层的每个结点有$|S_{n-1}|$棵子树，则第n+1层共有$|S_0|\times|S_1|\times\ldots\times|S_{n-1}|$个结点，它们都是叶子结点，代表问题的所有可能解。

两种典型的解空间树

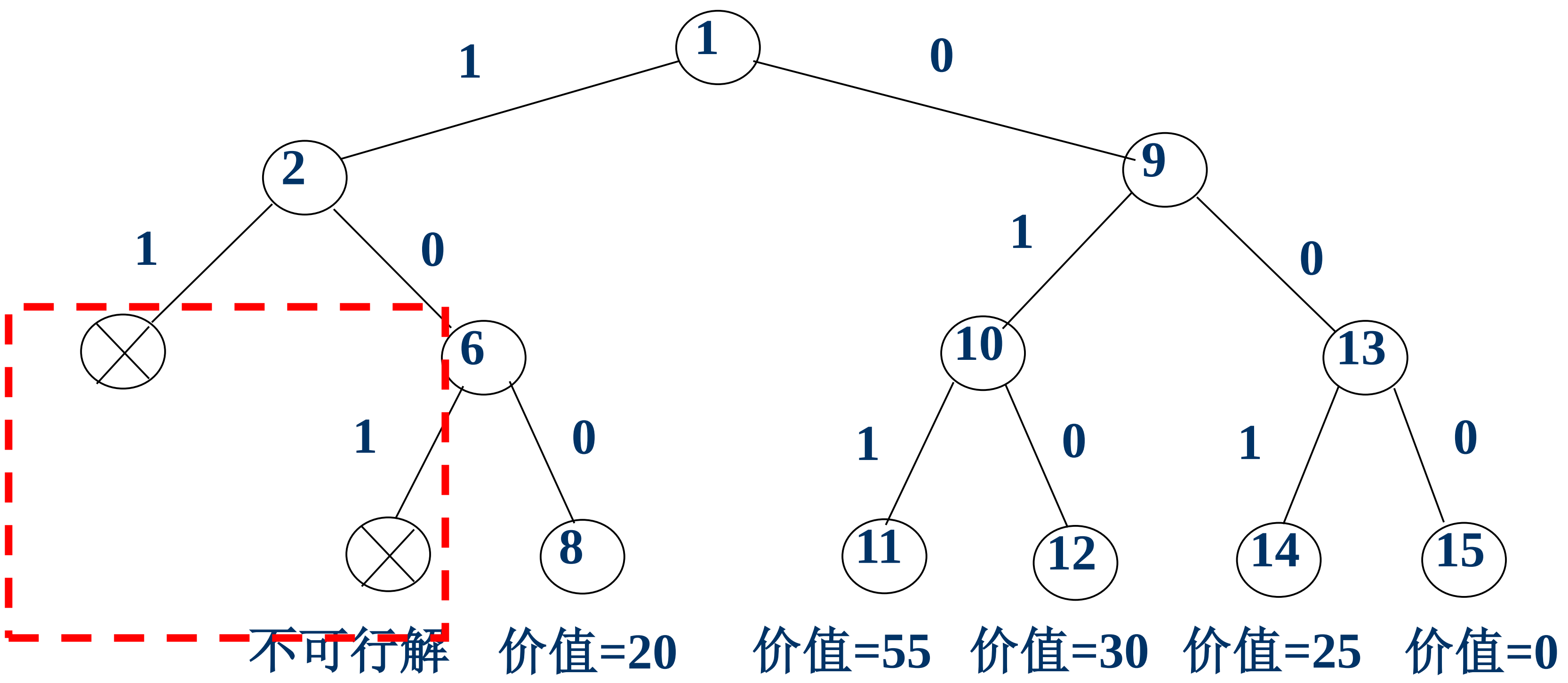
- **子集树（Subset Trees）**：当所给问题是从n个元素的集合中找出满足某种性质的**子集**时，相应的解空间树称为子集树。在子集树中，$|S_0|=|S_1|=\ldots=|S_{n-1}|=c$，即每个结点有相同数目的子树，通常情况下$c=2$，所以，子集树中共有$2^n$个叶子结点，因此，遍历子集树需要$O(2^n)$时间。
- **排列树（Permutation Trees）**：当所给问题是确定n个元素满足某种性质的**排列**时，相应的解空间树称为排列树。在排列树中，通常情况下，$|S_0|=n$，$|S_1|=n-1$，…，$|S_{n-1}|=1$，所以，排列树中共有$n!$个叶子结点，因此，遍历排列树需要$O(n!)$时间。

- 对于n=3的0/1背包问题，其解空间树如图所示，树中的8个叶子结点分别代表该问题的8个可能解。

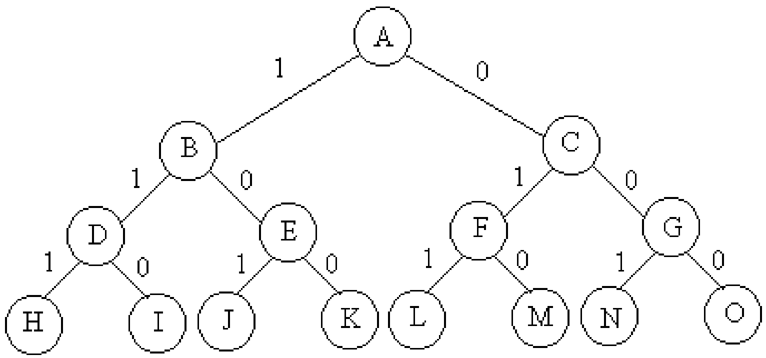


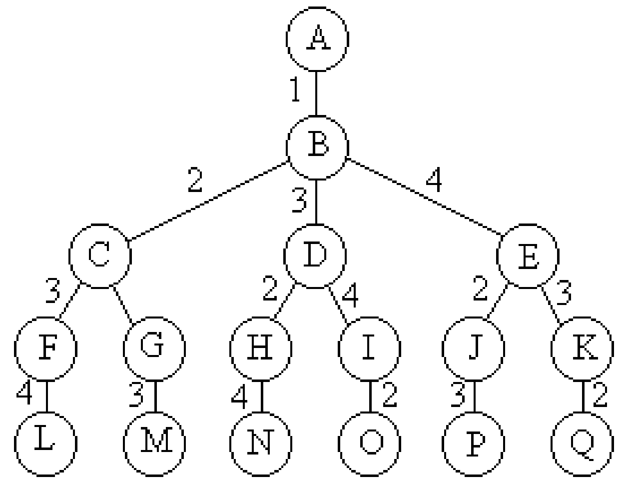
子集树！

ᨒ **例如**，对于n=3的0/1背包问题，三个物品的重量为{20, 15, 10}，价值为{20, 30, 25}，背包容量为25，从前图所示的解空间树的根结点开始搜索，搜索过程如下：



子集树与排列树





遍历子集树需O(2^n)计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (legal(t)) backtrack(t+1); }
}
```

遍历排列树需要O(n!)计算时间

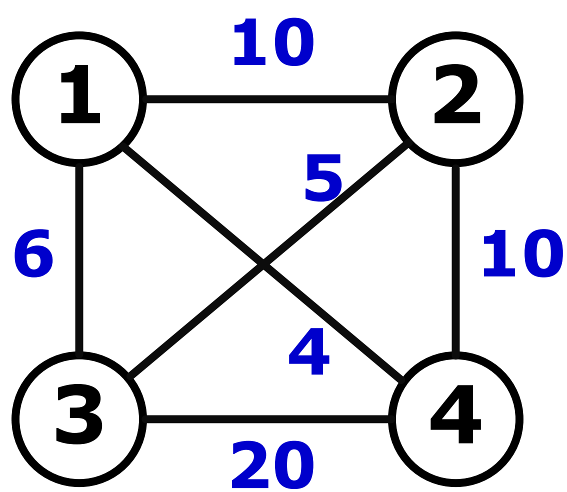
```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++) {
        swap(x[t], x[i]);
        if (legal(t)) backtrack(t+1);
        swap(x[t], x[i]); }
}
```

5.3 旅行商问题

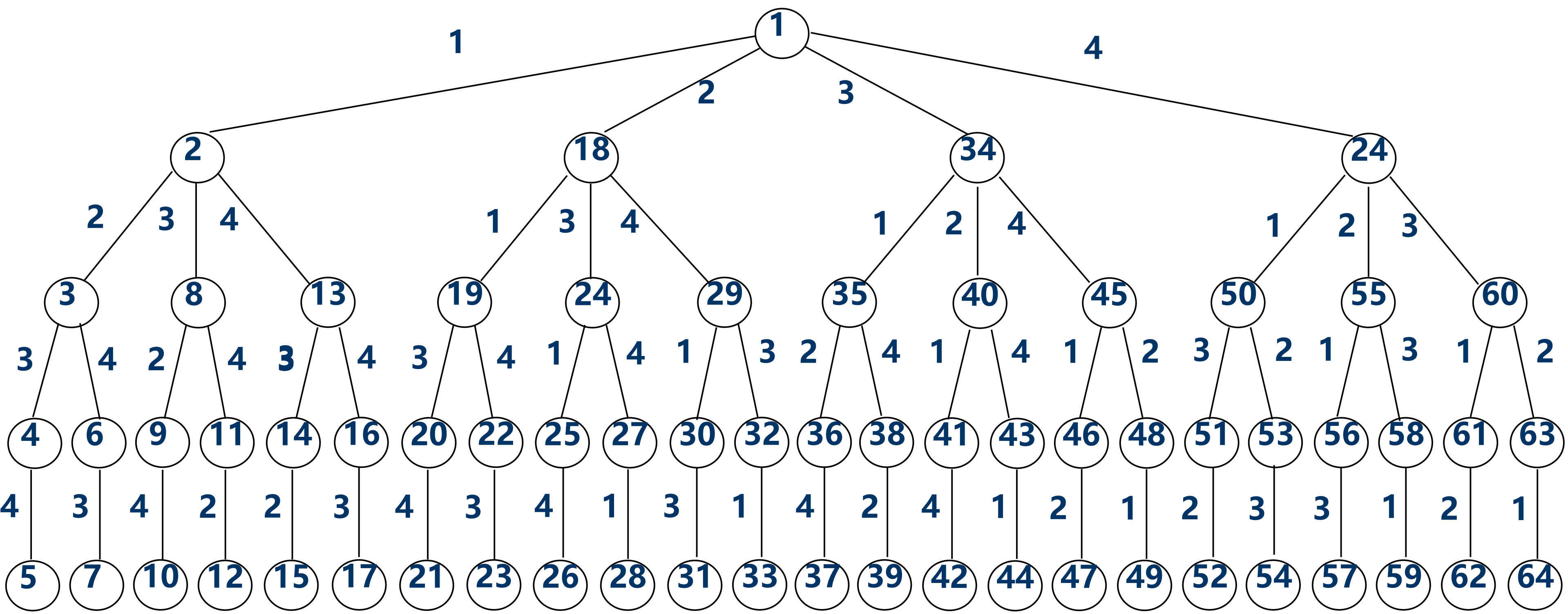
（Travelling Salesman Problem）

排列树示例：旅行商问题

- 旅行商问题：某推销员要去若干城市推销商品
  - 已知各城市间的开销（路程或旅费），要求选择一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，**使总开销最小**
  - 这是一个NP完全问题，形式化描述如下
    - 给定带权图G=(V,E)，已知边的权重为正数
    - 图中的一条周游路线是包括V中每个顶点的一条回路
    - 一条周游路线的开销是这条路线上所有边的权重之和
    - 要求在图G中找出一条具有最小开销的周游路线



对于n=4的TSP问题，其解空间树如图所示，树中的4!=24个叶子结点分别代表该问题的24个可能解，例如结点5代表一个可能解，路径为1→2→3→4→1，长度为各边代价之和。



n=4的TSP问题的解空间树

求解TSP问题

- 问题分析：与排列生成问题相比，多了一个回路
- 基本思想：利用排列生成问题的回溯算法Backtrack()
  - Backtrack(2)表示：对x={1, 2, ..., n}的x[2..n]进行全排列
  - 则：(**x[1]**, x[2])，(x[2], x[3])，..., (x[n], **x[1]**)构成回路
  - 在全排列算法的基础上，进行路径计算保存以及限界剪枝

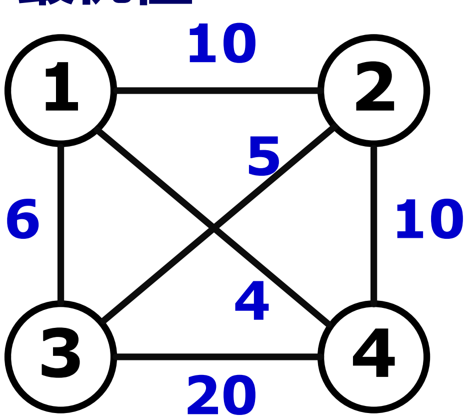
```
main(int n){
    // 输入邻接矩阵 A[n][n];
    x[n] = {1,2,...,n};
    sum=0.0;       // 记录(x[1],x[2]),..., (x[i-2],x[i-1])的距离和
    S[n] = {0};    // S[n]保存当前最佳路径
    m = ∞;         // m保存当前最优值
    backtrack(2, S, m, &sum);
    output( m, S);
}
```

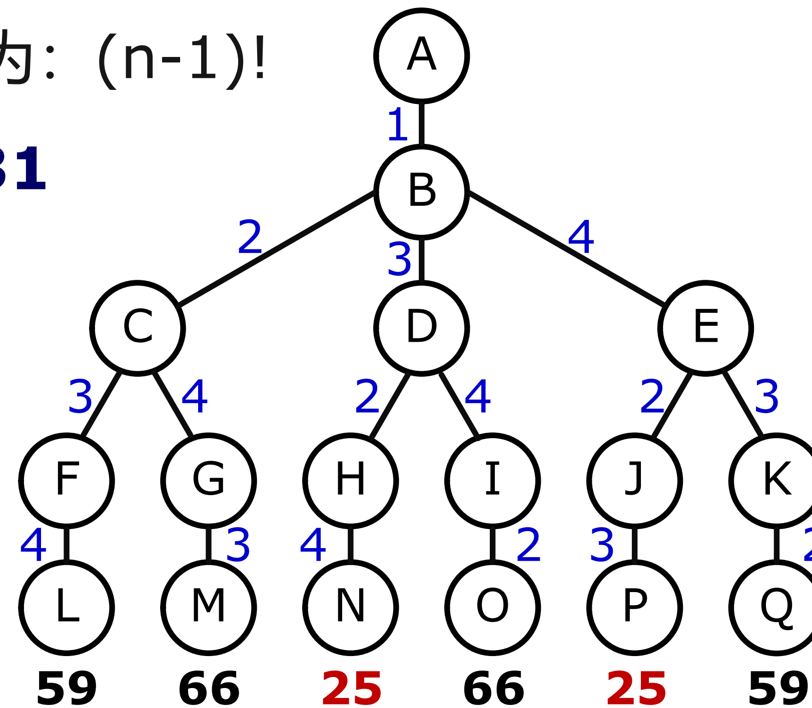
排列树示例：旅行商问题

- 解空间：X={12341, 12431, 13241, 13421, 14231, 14321}
- 构造解空间树
  - 从根结点到任一叶结点的路径定义了图G的一条周游路线
    - 例如：A->L 对应周游路线（1, 2, 3, 4, 1）
  - 解空间树中的每个叶结点恰好对应于图G的每一条周游路线
    - 解空间树中的叶结点个数为：(n-1)!

最优解： 13241，14231

最优值： m = 25





解空间树

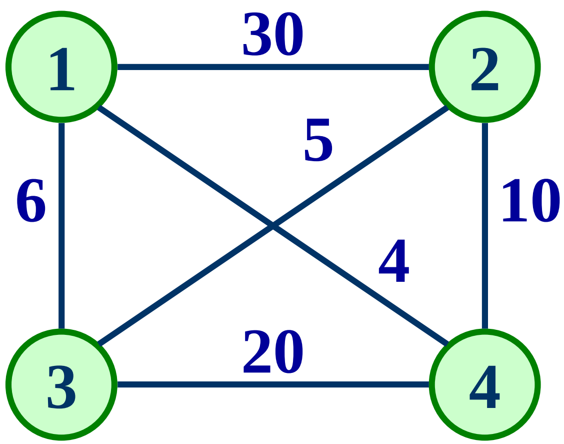


图5-2 4顶点带权图

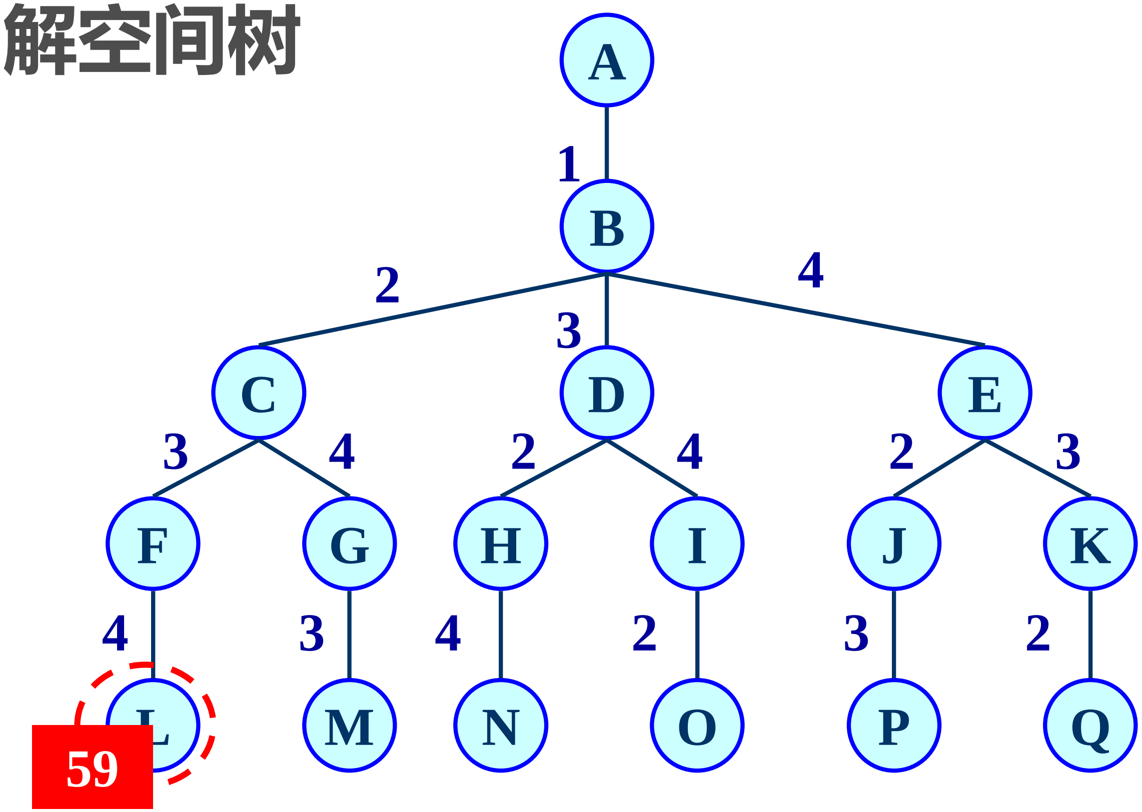


图5-3 旅行售货员问题的解空间树

- **用回溯法找最小费用周游路线时，从解空间树的根结点A出发，搜索至B，C，F，L。在叶子结点L处记录找到的周游路线1,2,3,4,1，该周游路线的费用为59。**

解空间树

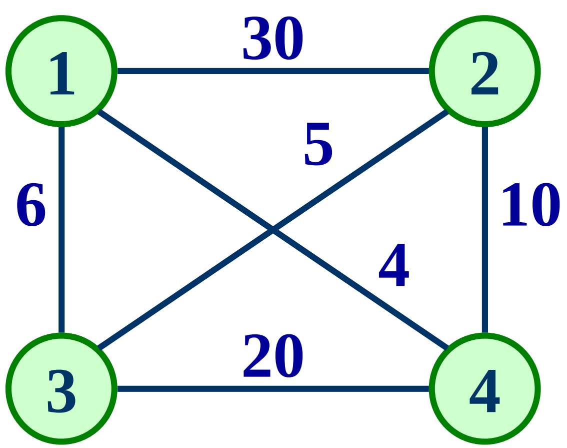


图5-2 4顶点带权图

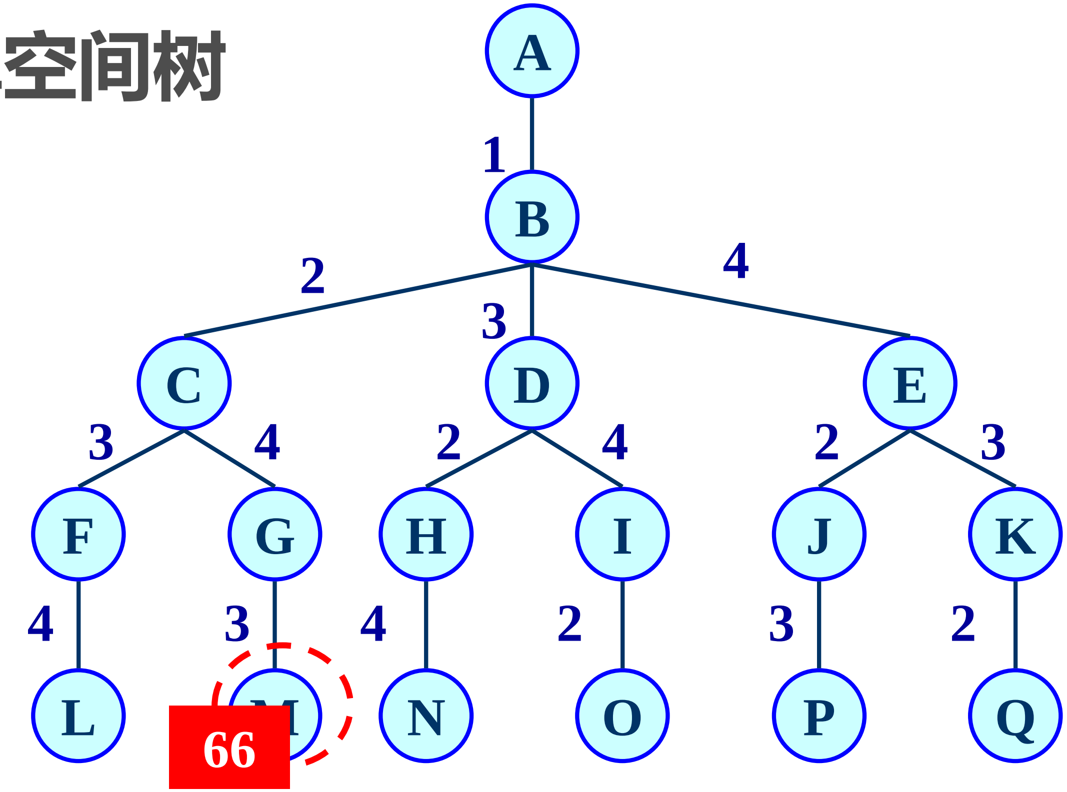


图5-3 旅行售货员问题的解空间树

- 从叶子结点L返回最近活结点F处。由于F已没有可扩展结点，算法又返回到结点C处。结点C成为新扩展结点，由新扩展结点，算法再移动到结点G又移至结点M，得到新的周游路线1,2,4,3,1，其费用为66。费用比前一个大，舍弃！

解空间树

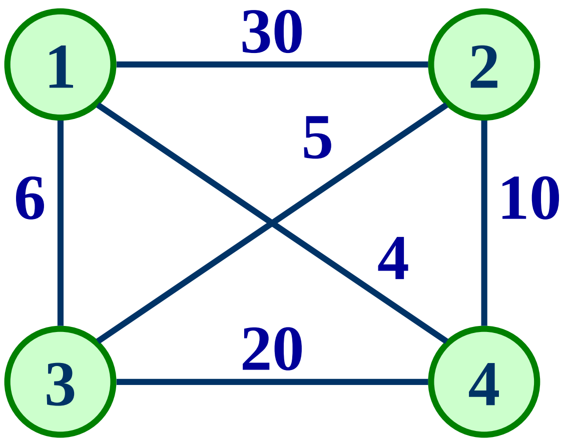


图5-2 4顶点带权图

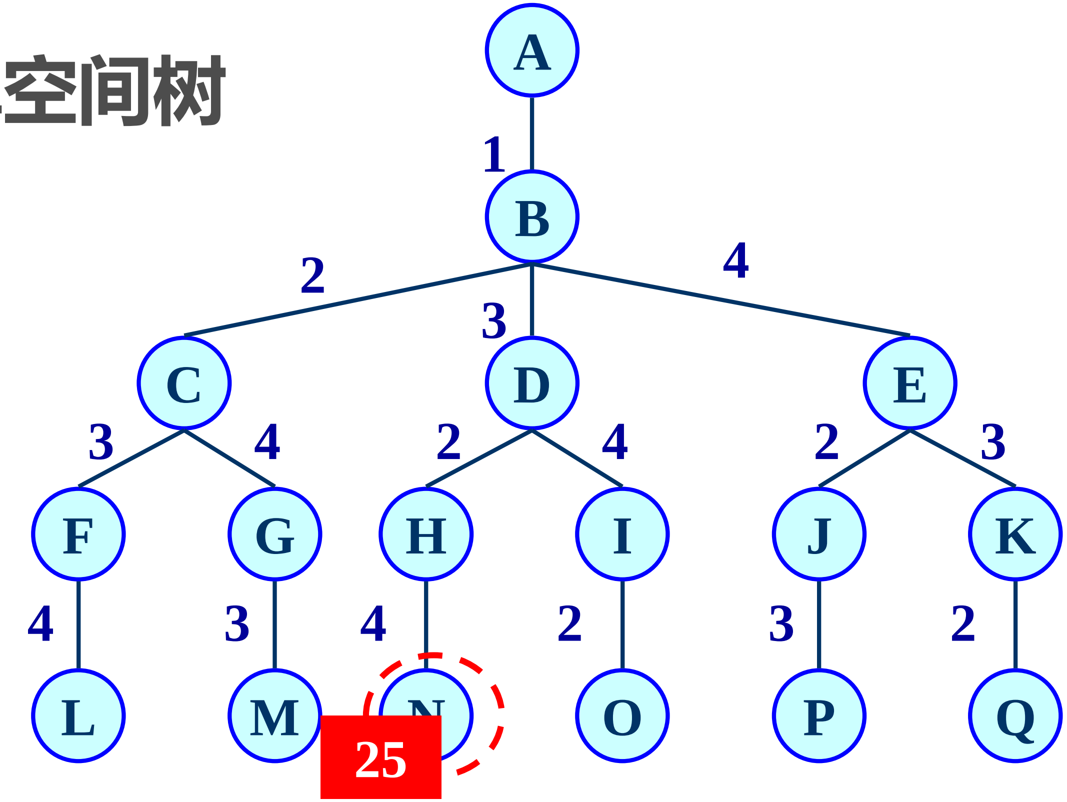


图5-3 旅行售货员问题的解空间树

- 算法又依次返回到结点G，C，B。从结点B，算法继续搜索到结点D，H，N。在叶子结点N处，相应的周游路线1,3,2,4,1，其费用为25。它是当前找到的**最好的一条周游路线**。

解空间树

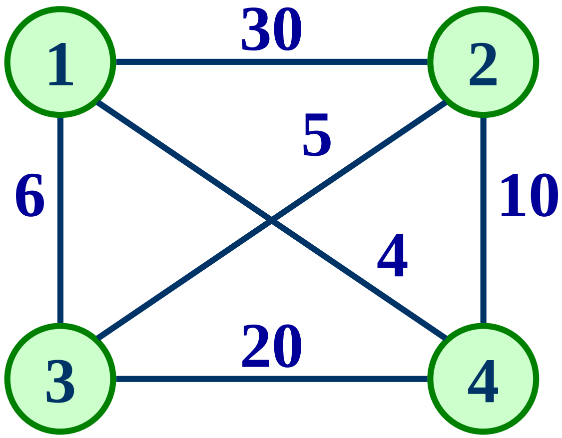


图5-2 4顶点带权图

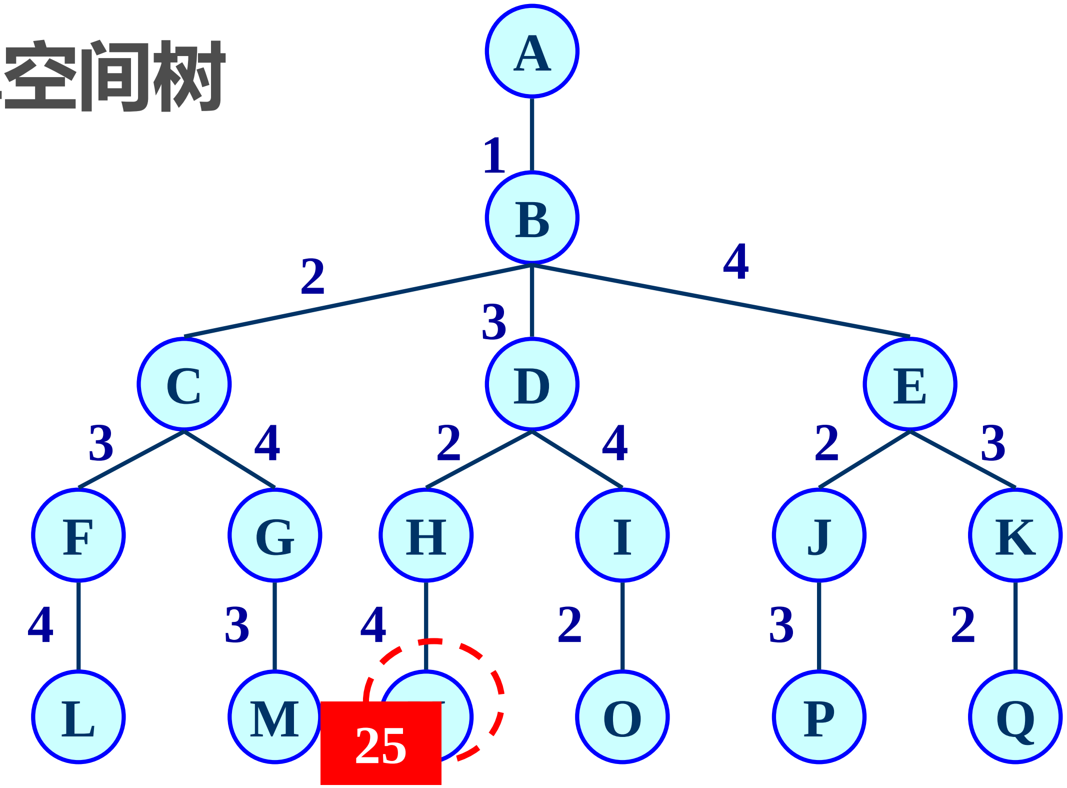
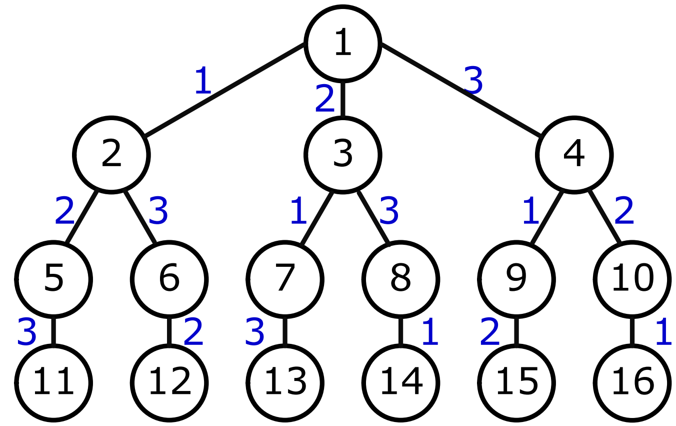


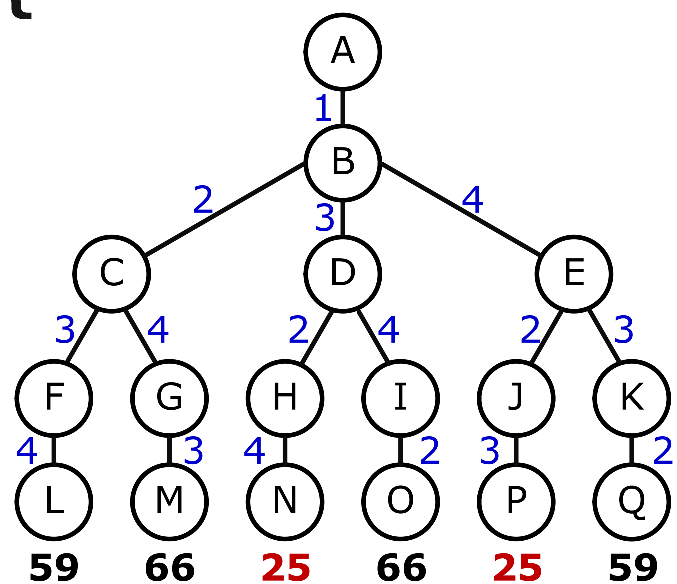
图5-3 旅行售货员问题的解空间树

- 从结点N算法返回至结点H，D，然后再从结点D开始继续向纵深搜索至结点O。依此方式继续搜索整个解空间树，最终得到最小费用周游路线1,3,2,4,1。

排列树回溯算法框架

```
void backtrack (int i) {
    if (i > n){
        output(x);
    }
    else{
        for (int k = i; k <= n; k++) {
            swap(x[k], x[i]);
            backtrack(i+1);
            swap(x[k], x[i]);
        }
    }
}
```





求解TSP问题

```
backtrack(int i){
    if (i>n){ // 输出可行解，与当前最优解比较
        if (sum + A[x[n]][x[1]] < m || m = ∞ ) {
            m = sum + A[x[n]][x[1]];
            for( k=1 ; k <= n; k++) S[k] = x[k];
        }
    } sum记录(x[1],x[2]),…, (x[i-2],x[i-1])的距离和
    else{
        for( k = i ; k<=n; k++ )// 依次处理当前扩展结点的分支
            if ( sum + A[x[i-1]][x[k]] < m || m = ∞  ){
                swap( x[i], x[k]);
                sum += A[x[i-1]][x[k]];
                backtrack(i+1);
                sum -= A[x[i-1]][x[k]];
                swap(x[i],x[k]);
            }
    }
} // 初始调用：Backtrack(2)
```

算法复杂度：O(n!)

5.4 0/1背包问题

（0/1 Backpack Problem）

0/1背包问题

- **问题分析**
  - 问题的解向量：(x_1, x_2, … , x_n)
  - 解空间树：子集树
  - 剪枝函数
    - 约束函数：$\sum_{i=1}^{n} w_i x_i \le c_1$
    - 限界函数：怎样设计 **?**
- **限界函数的设计思路**
  - 考查经过当前扩展结点R的可行解
    - 该可行解的最终价值有可能小于已知的最优值m
    - 问题：怎样计算经过R的可行解的价值上界?
    - 思路：分为两部分：v（从根到R）+v（以R为根的子树）

0/1背包问题

- **怎样计算价值上界?**
  - 例：n = 4，c = 7，p = [9, 10, 7, 4]，w = [3, 5, 2, 1]
    - 易求得这四个物品的单位重量价值分别为：[3, 2, 3.5, 4]
    - 按物品单位重量价值递减的顺序装入物品
    - 依次装入物品4、3、1之后，剩余背包容量为1
      - 所以只能容纳物品2的20%
      - 得到解向量x = [1, 0, 0.2, 1]，相应价值为22
    - 虽然x并不是0/1背包问题的可行解
      - 但它提供了一个最优的价值上界（最优值不超过22)
  - 为便于计算上界函数，可先对物品按单位价值从大到小排序
    - 对每个扩展结点，只需按顺序考查排在其后的物品即可

限界函数的实现

```
// 根据当前背包内物品情况求出：当前可行解的价值上界
// w[i]和v[i]均已按物品单位价值递减顺序排好序
int Bound(int i){
    int wr = c - wc;  // 背包剩余容量
    int vb = vc;         // vc为当前背包价值
    // 按单位价值递减顺序装入物品
    while(i <= n && w[i] <= wr){
        wr -= w[i]; vb += v[i]; ++i;
    }
    if(i <= n) vb += (v[i]/w[i])*wr; // 继续装满背包
    return vb; // 返回背包价值上界
}
```

0/1背包问题的回溯算法

```
backtrack(int i){
    if( i > n) {   // vc当前背包价值，m当前最优价值
        m = ( m < vc )? vc : m;   output(x);
    } else {  // wc当前背包重量
        if ( wc + w[i] <= C ) {  // 左子树（将 i 放入背包）
            x[i]= 1; wc += w[i]; vc += v[i];
            backtrack(i+1);
            x[i]= 0; wc -= w[i]; vc -= v[i];
        }
        if(Bound(i+1) > m){  // 右子树（拿出物品i）
            backtrack(i+1);
        }
    }
}
```

算法复杂度：$O(n2^n)$

5.5 装载问题

(The Container Loading Problem)

装载问题

- **问题描述**
  - 有n个集装箱要装上2艘载重量分别为 c_1 和 c_2 的轮船
  - 其中集装箱 i 的重量为 w_i，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n个集装箱装上这2艘轮船？如果有，找出一种装载方案。
- **示例：n=3，c_1=c_2=50**
  - 若：w=[10, 40, 40]
    - 则可以将集装箱1和2装到第一艘船上
    - 将3号集装箱装到第二艘船上
  - 若：w=[10,40,40]
    - 则无法将这三个集装箱全部装船

装载问题

问题分析

- 回顾一艘船的情况
  - 采用贪心选择策略：从轻到重依次装船，直至超重
  - 思考：目前有两艘船，需要什么样的策略?
- 若给定问题有解，则采用如下策略可得最优装载方案
  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
- 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 c_1
- 由此可知，装载问题等价于以下特殊的0-1背包问题

装载问题

- **与上述最优装载问题等价的0/1背包问题**

$$\max\left(\sum_{i=1}^{n} w_i x_i\right) \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \le c_1 \text{，} \ x_i \in \{0,1\}, 1 \le i \le n$$

- **算法设计：用回溯法求解最优装载问题**
  - 解空间的表达：子集树
  - 剪枝函数（1）：约束函数：$\sum_{i=1}^{n} w_i x_i \le c_1$
    - 在子集树的第k层的结点R处，以c_k表示当前的装载重量
      - 即：$c_k = \sum_{i=1}^{k} w_i x_i$
      - 则当 $c_k > c_1$ 时，以结点R为根的子树中所有结点均不满足约束条件
      - 因而该子树中的解均为不可行解，故可将该子树剪去

装载问题

算法设计：用回溯法求解最优装载问题

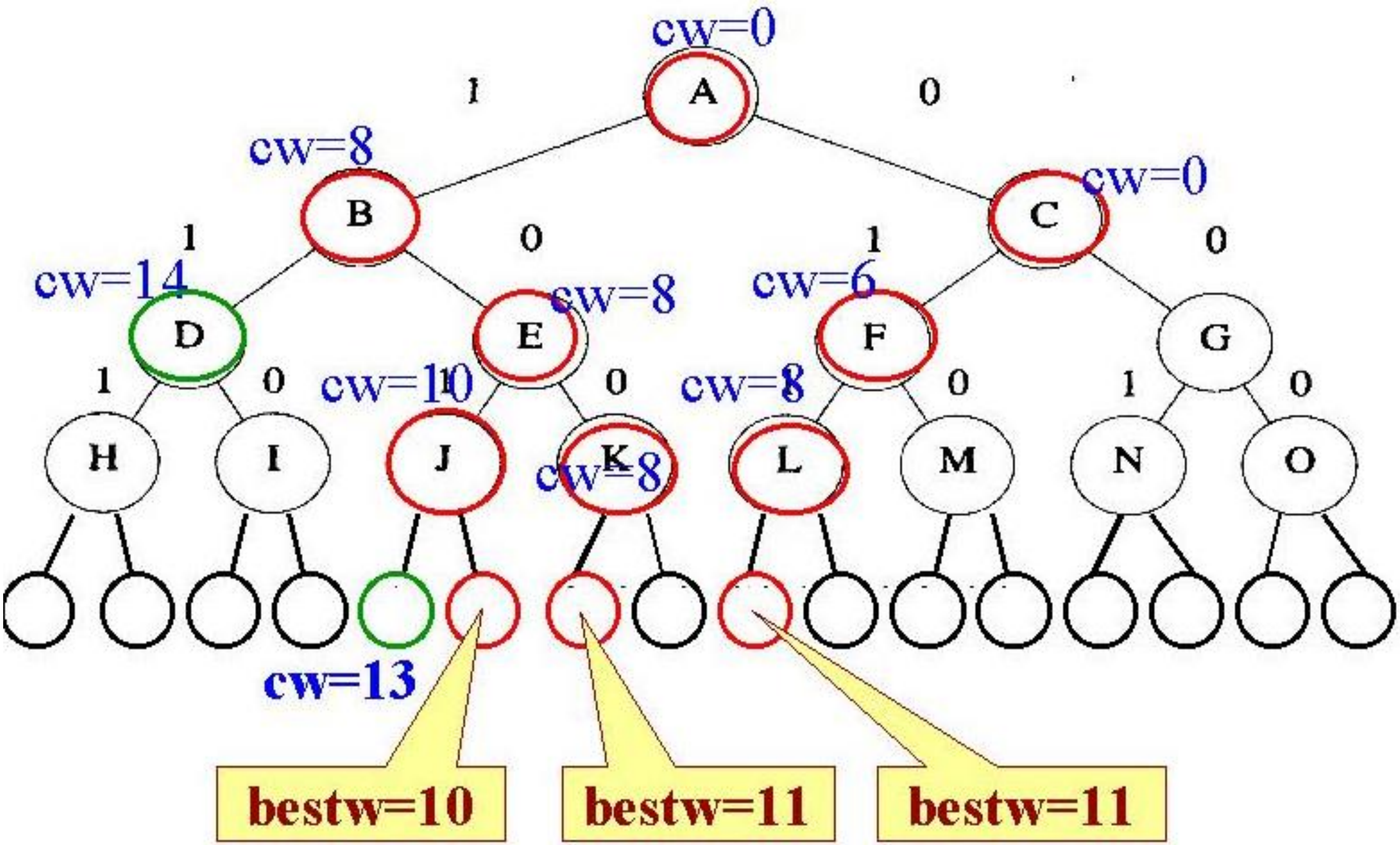
- 剪枝函数（2）：限界函数（用于剪去不含最优解的子树）
  - 设：R是解空间树第k层上的当前扩展结点
  - 设：wc表示当前结点对应的的装载重量 $wc = \sum_{i=1}^{k} w_i x_i$
  - 设：wm表示当前的最优载重量
  - 设：wr表示剩余集装箱的重量 $wr = \sum_{i=k+1}^{n} w_i$
  - 定义限界函数为： $w = wc + wr$
  - 以R为根的子树中任一叶结点对应的载重量均不会超过w
  - 因此当 w≤wm 时，可将以R为根的子树剪去

装载问题

```
void backtrack (int i) {
    if (i > n){
        if(wc > wm) wm = wc; return;
    }
    wr -= w[i];
    if (wc + w[i] <= c){  // x[i] = 1; 搜索左子树
        wc += w[i];
        backtrack(i+1);
        wc -= w[i];
    }
    if (wc + wr > wm){  // x[i] = 0; 搜索右子树
        backtrack(i+1);
    }
    wr += w[i];
}
```

算法复杂度：$O(2^n)$

例如 n=4, c1=12, w=[8, 6, 2, 3]. bestw初值=0;

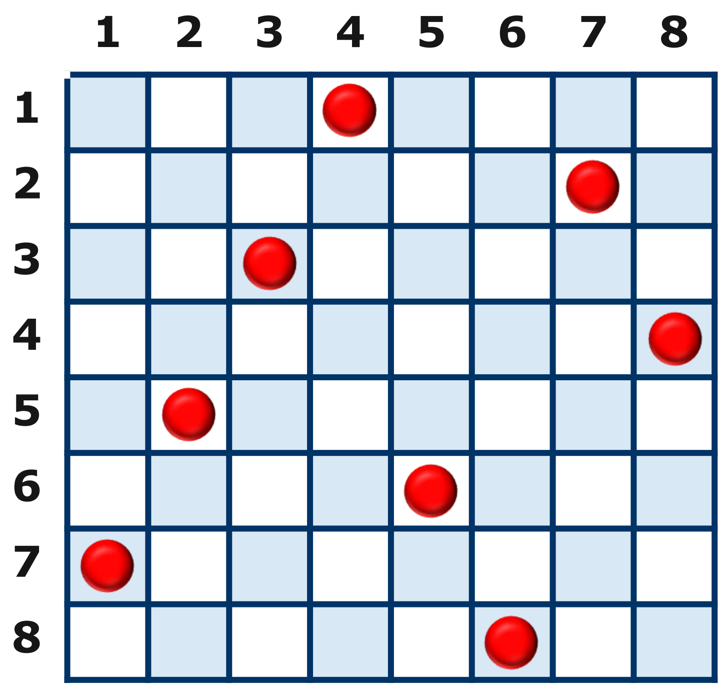


5.6 n-皇后问题

(The n-queens puzzle)

n-皇后问题

- 根据国际象棋的规则
  - 皇后可以攻击与之处在**同一行**或**同一列**或**同一斜线**上的棋子
- **n-皇后问题**
  - 在n×n的棋盘上放置彼此不受攻击的n个皇后
  - 即：任何2个皇后不放在同一行或同一列或同一斜线上



n-皇后问题

问题分析

- 问题的解向量：$(x_1, x_2, \ldots, x_n)$
  - 采用数组下标 **i** 表示皇后所在的行号
  - 采用数组元素**x[i]**表示皇后 **i** 的列号
- 思考：采用哪种解空间树？ **子集树**
  - 提示：这是排列问题还是子集问题?
- 剪枝函数
  - 显约束（对解向量的直接约束）：$x_i = 1, 2, \ldots, n$
  - 隐约束1：任意两个皇后不同列 ：$x_i \neq x_j$
  - 隐约束2：任意两个皇后不处于同一对角线?
    - $|i-j| \neq |x_i - x_j|$

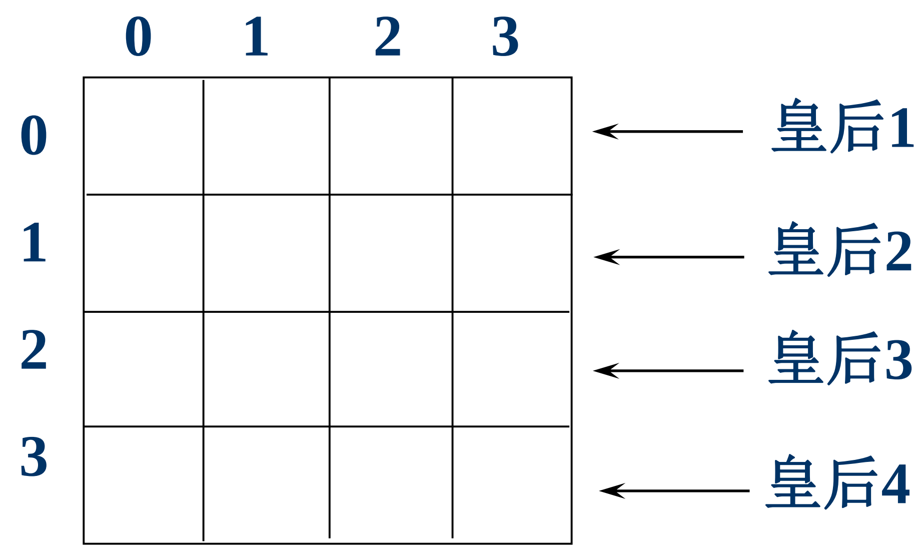
n-皇后问题

```
bool Bound(int k){
    for (int i = 1; i < k; i++){
        if ((abs(k-i)==abs(x[i]-x[k]))||(x[i]==x[k]))
            return false;
    }
    return true;
}

void Backtrack(int t){
    if (t>n) output(x);
    else {
        for (int i = 1; i <= n; i++) {
            x[t] = i;
            if (Bound(t)) Backtrack(t+1);
        }
    }
 }
```

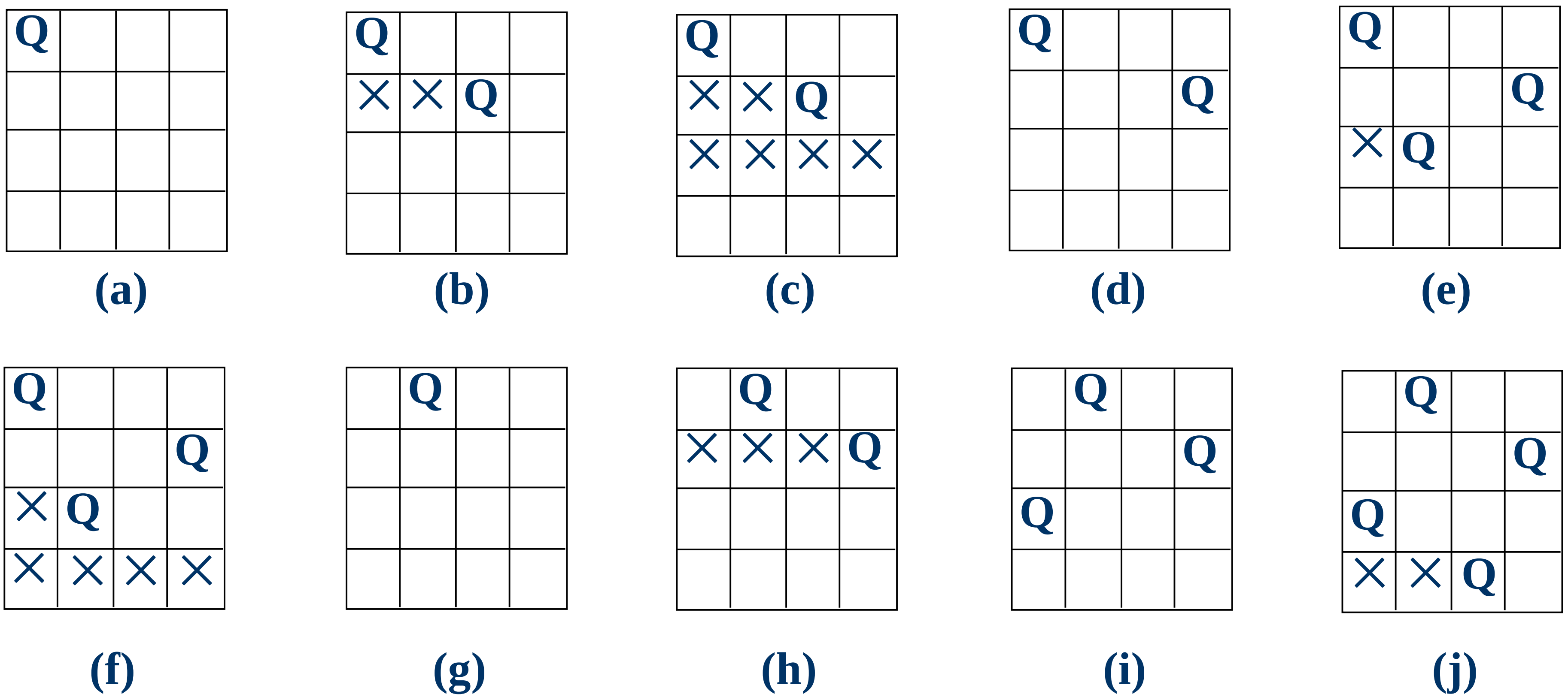
4皇后问题

- 四皇后问题的解空间树是一个完全4叉树，树的根结点表示搜索的初始状态，从根结点到第1层结点对应皇后1在棋盘中第0行的可能摆放位置，从第1层结点到第2层结点对应皇后2在棋盘中第1行的可能摆放位置，依此类推。



四皇后问题

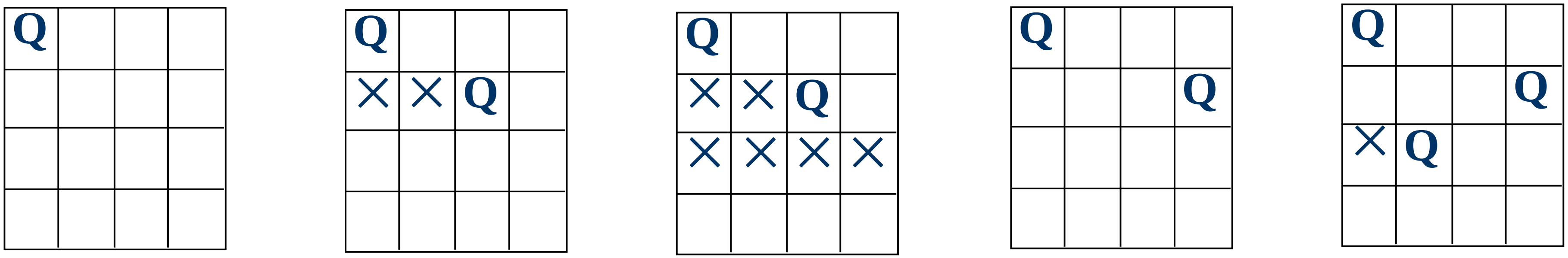
回溯法求解4皇后问题的搜索过程（一个可行解）

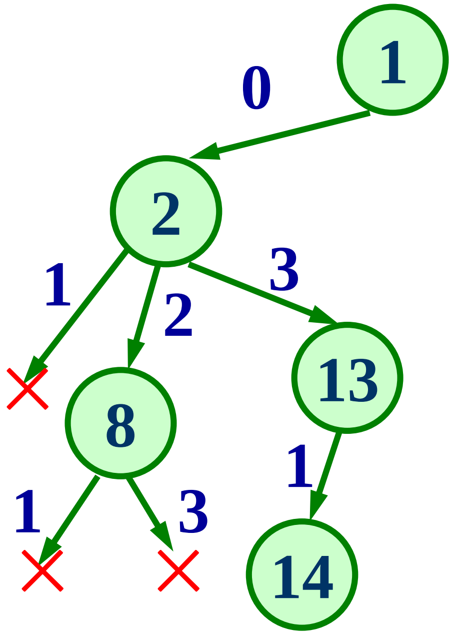


(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

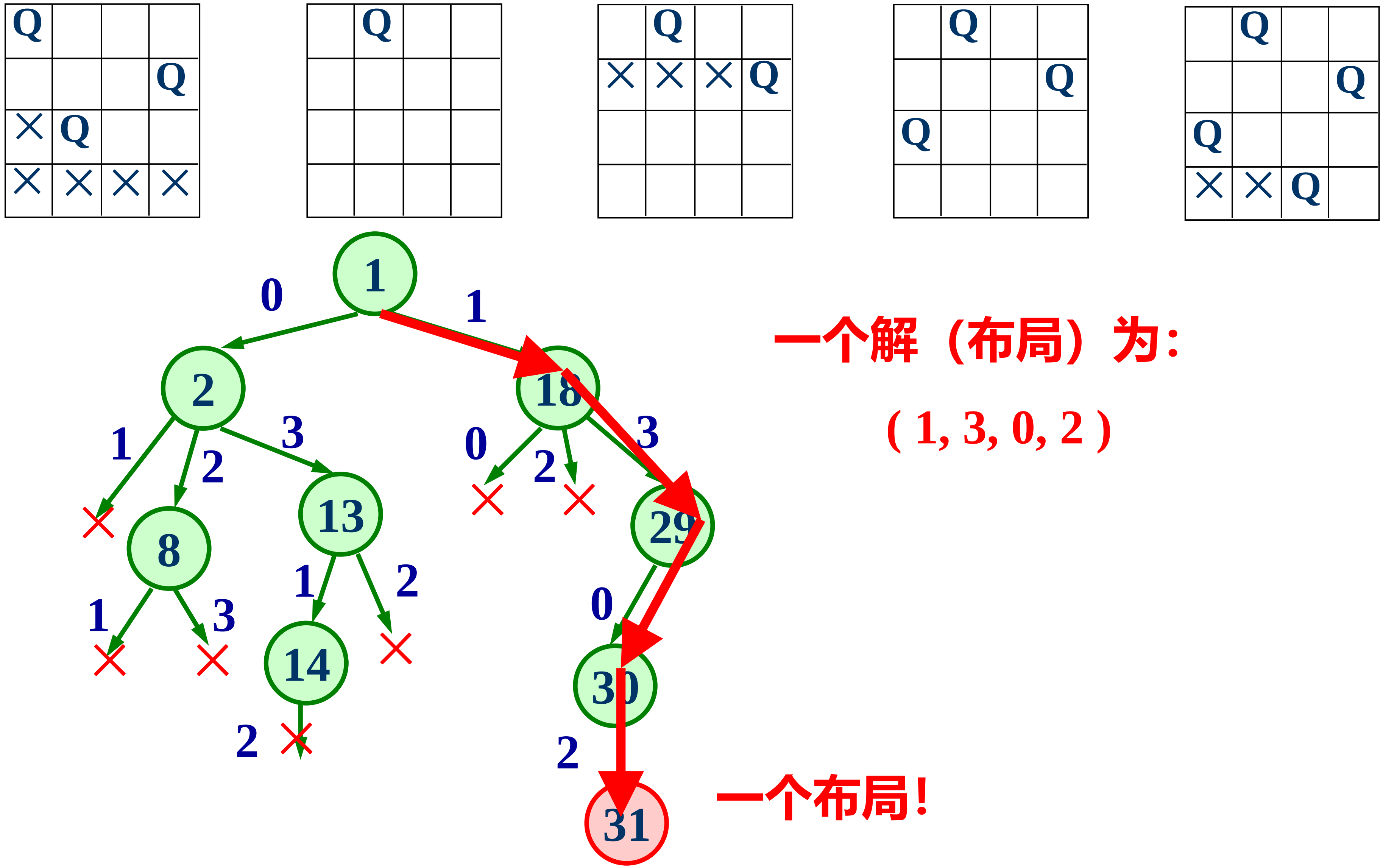
4皇后问题的解空间树的生成





4皇后问题形式化描述：

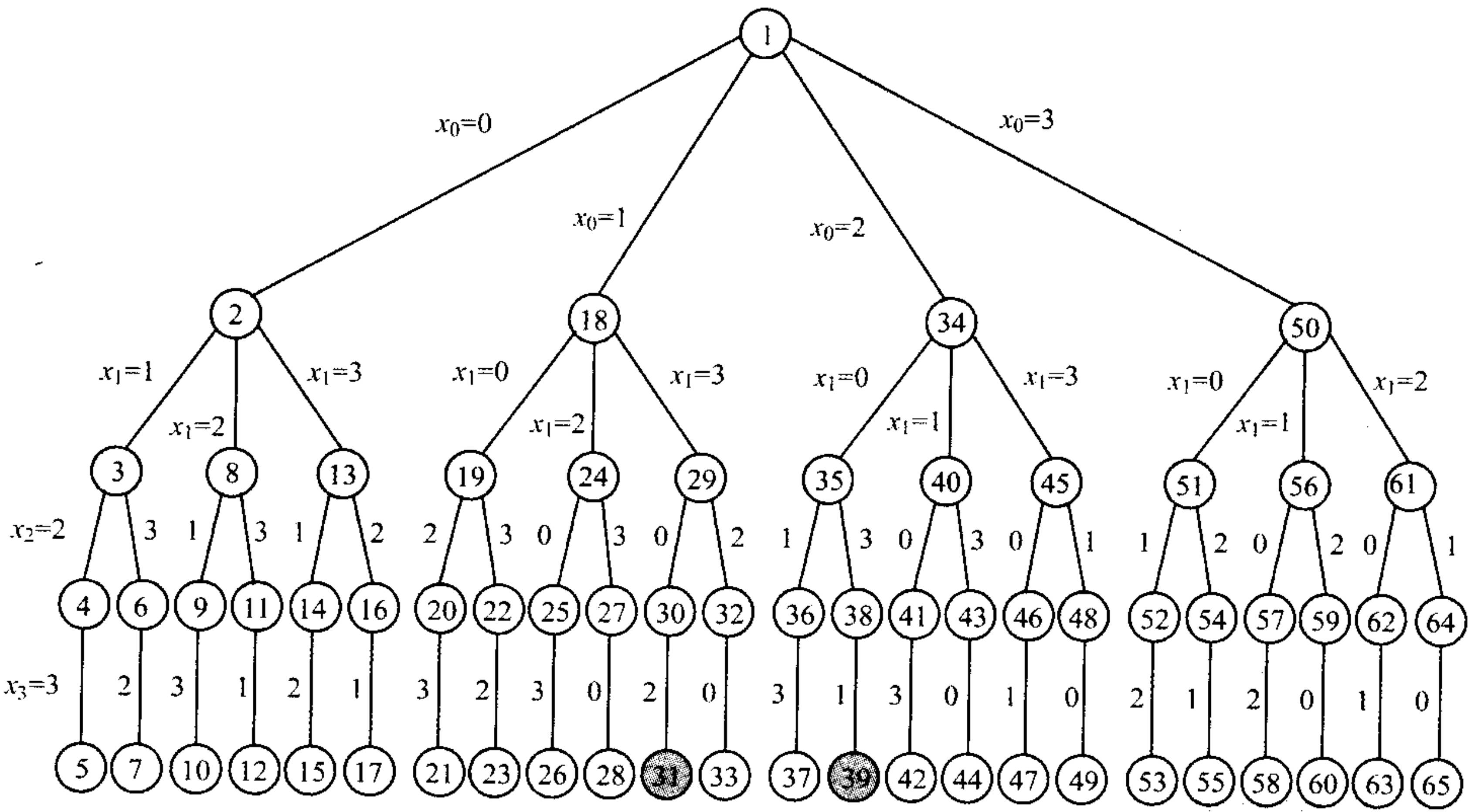
- $S_i = \{ 0, 1, 2, 3 \}$，$0 \le i < n$，且 $x_i \ne x_j (0 \le i,j < n, i \ne j)$。
- **相应的隐式约束为：对任意 $0 \le i,j < n$，当 $i \ne j$ 时，$|i-j| \ne |x_i - x_j|$。**
- **对应的解空间大小为 $n!$。**



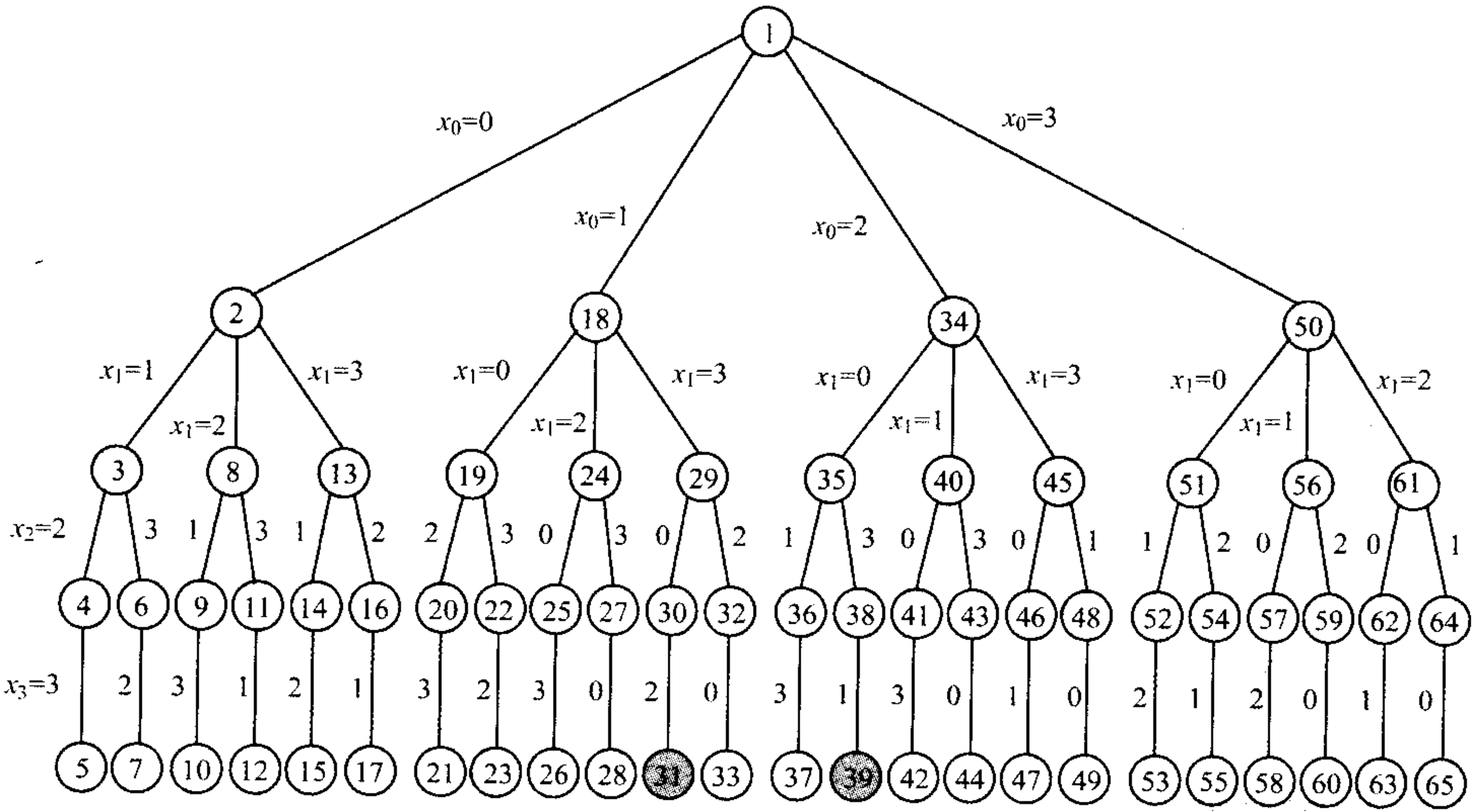
一个解（布局）为：

(1, 3, 0, 2)

一个布局!



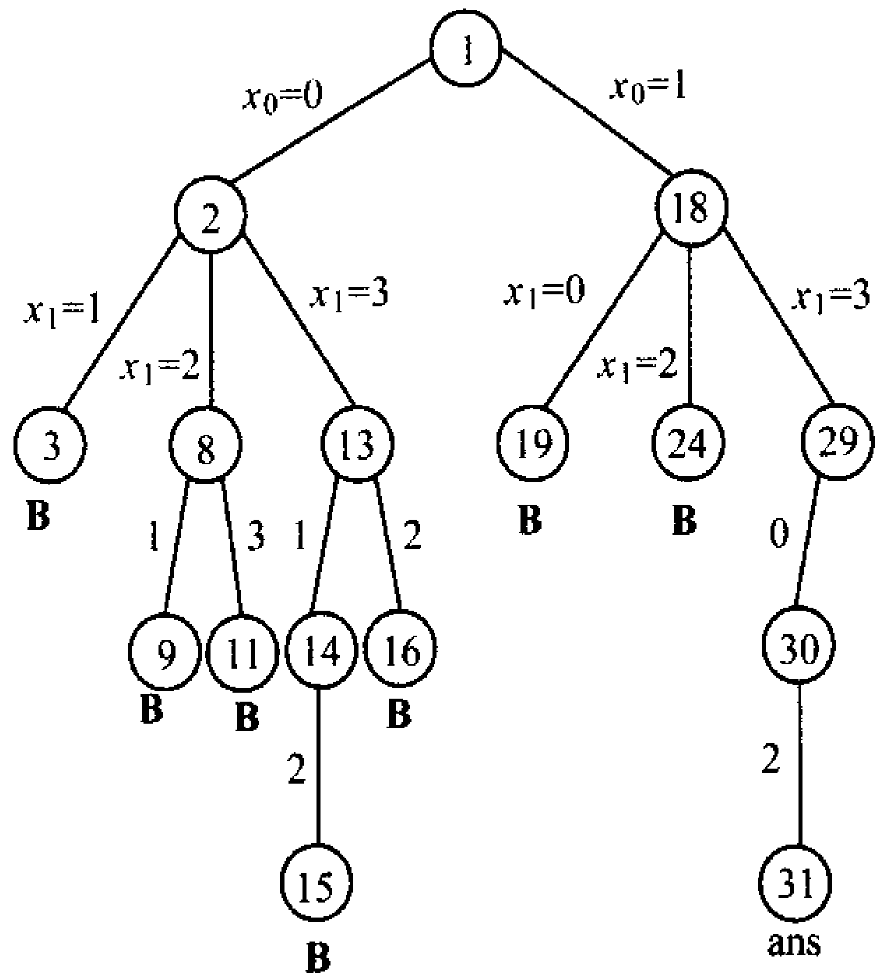
- **图是4-皇后问题的解空间树。**
- **解状态中包含答案状态31和39。**



- **一般称这种用于确定n个元素的排列满足某些性质的解空间树为排列树（permutationtree）。排列树有n!个叶子结点，遍历排列树的时间为O(n!)。**

分析

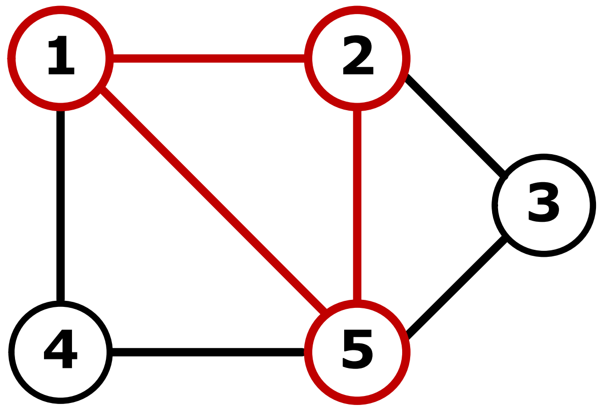
- 如图显示了4-皇后问题在得到第一个答案状态时，实际生成的那部分解空间树。图中，B代表被限制的结点，ans是第一个答案结点。



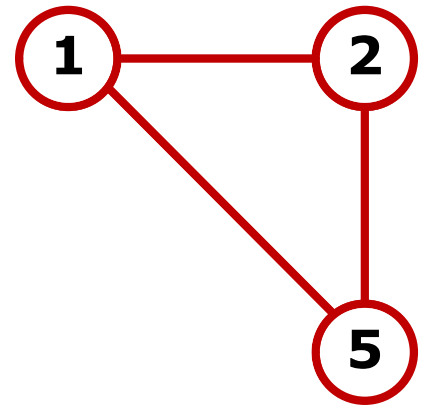
5.7 最大团问题

（ Maximum Clique Problem）

最大团问题



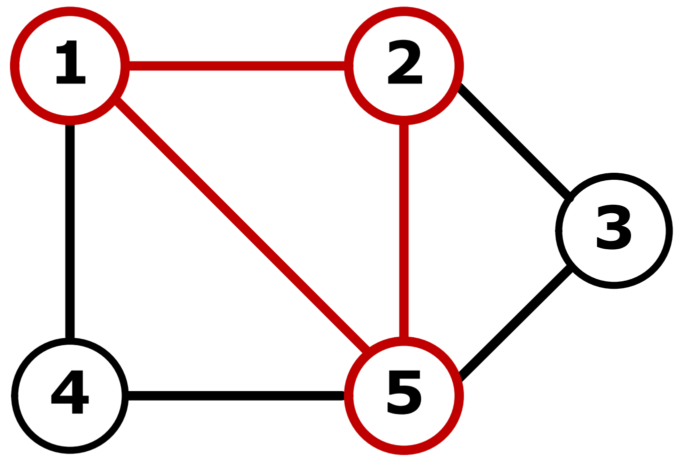
无向图G



G的完全子图U

- **问题描述** **思考：团和最大团的区别?**
  - 给定无向图G=(V，E)和G的完全子图U
    - 完全子图： U⊆V且对任意u∈U和v∈U，有(u，v) ∈ E
  - 定义：U是G的**团**，当且仅当U不包含在G的更大的完全子图中
  - G的**最大团**是指：G中所含顶点数最多的团

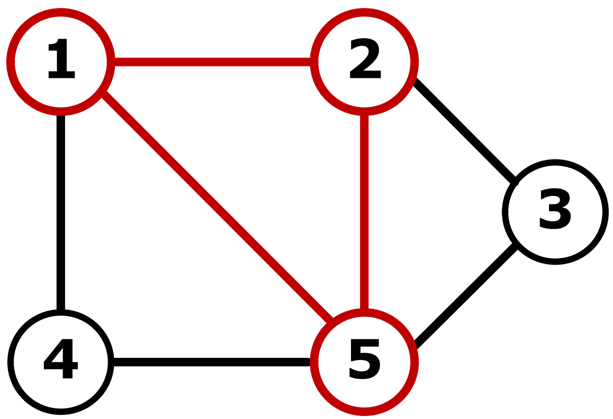
最大团问题



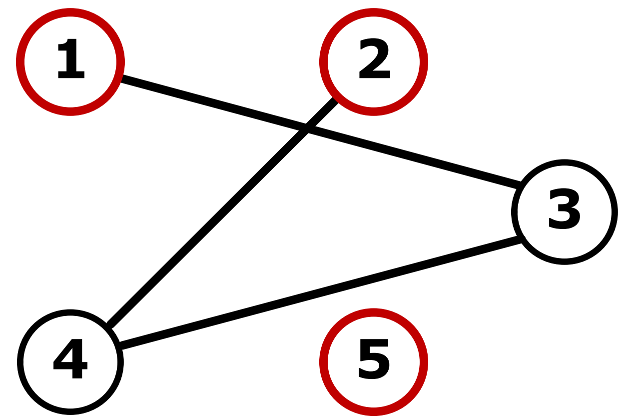
无向图G

- **基本概念**
  - 如图：子集{1,2}是图G的大小为2的完全子图，但不是一个团
    - 因为它包含于G的更大的完全子图{1,2,5}之中
    - 子集{1,2,5}是G的一个最大团 …… **最大团是唯一的么?**
    - 子集{1,4,5}和{2,3,5}也是G的最大团

最大团问题



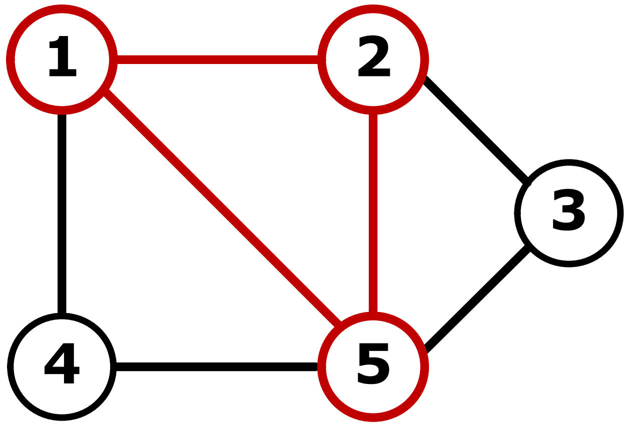
无向图G



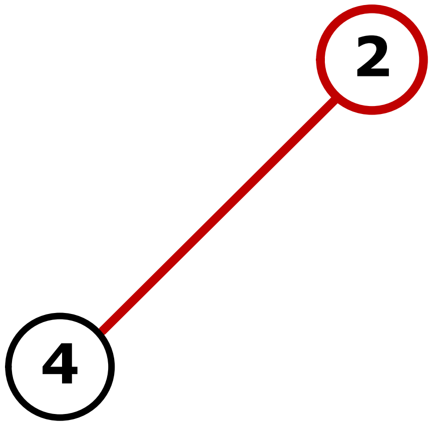
图G的补图G′

- **无向图的补图**
  - 无向图$G=(V，E)$的补图$G'=(V'，E')$ 定义为
    - $V'=V$，且$(u，v)\in E'$ 当且仅当 $(u，v)\notin E$
  - 显然：补图的概念是相对于完全图定义的

最大团问题



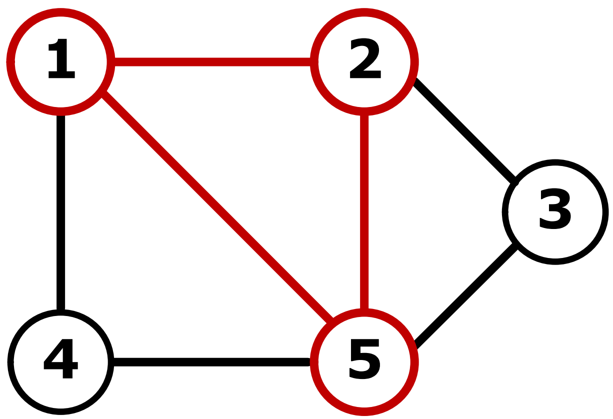
无向图G



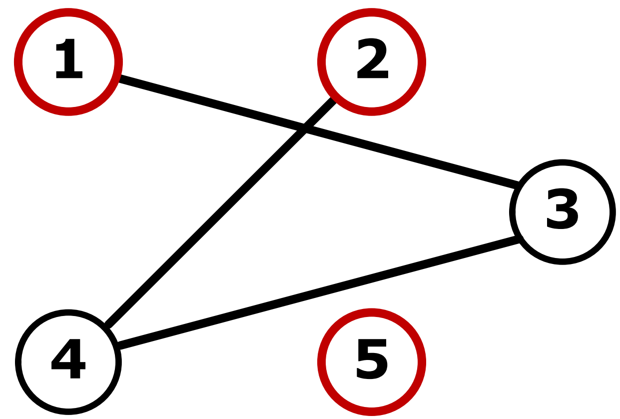
图G的空子图

- **最大独立集**
  - 如果U$\subseteq$V且对**任意**u，v$\in$U有(u，v)$\notin$E，则称U是G的**空子图**
  - 空子图U是G的独立集**当且仅当**U不包含在G的更大的空子图中
  - G的最大独立集：是G中所含顶点数最多的独立集

最大团问题



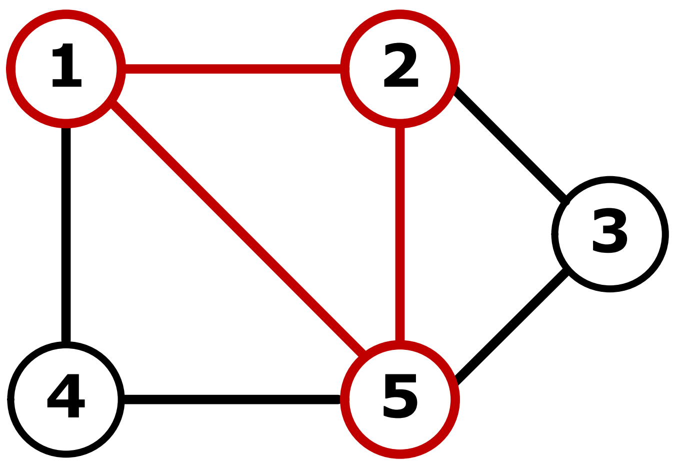
无向图G



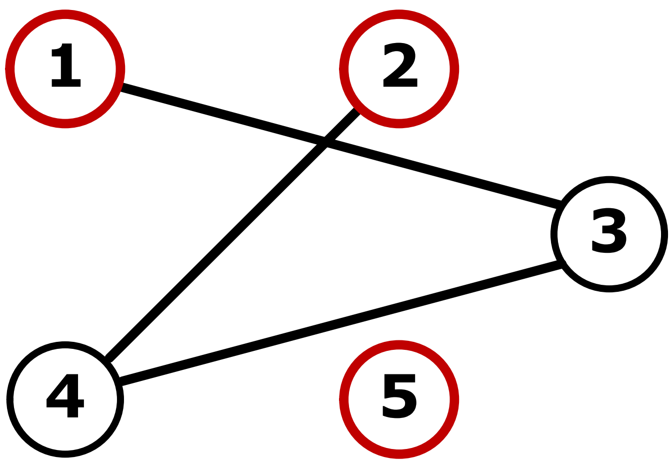
图G的补图G′

- **最大独立集**
  - 如图：{2,4}是G的一个空子图，同时也是G的一个最大独立集
  - 子集{1,2}是G'的空子图，但它不是G'的独立集
    - 因为它包含在G'的空子图{1,2,5}中
    - 子集{1,2,5}是G'的最大独立集
    - 子集{1,4,5}和{2,3,5}也是G'的最大独立集

最大团问题



无向图G



图G的补图G′

- 无向图G的最大团和最大独立集问题是等价的
  - U是G的完全子图，则它也是G'的空子图，反之亦然
  - 推论：U是G的最大团**当且仅当**U是G′的最大独立集
  - 二者都可以看做是图G的顶点集V的子集选取问题
  - 二者都可以用回溯法在$O(n2^n)$的时间内解决

最大团问题

问题分析

- 问题的解向量：(x_1, x_2, … , x_n) 为0/1向量
  - **x_i** 表示该顶点是否入选最大团
- 思考：采用哪种解空间树？ **子集树**
- 解题思路（mark）
  - 首先设最大团U为空集，向其中加入一个顶点v_0
  - 然后依次（递归地）考查其他顶点v_i
    - 若 v_i 加入后，U不再是团，则舍弃顶点v_i（考查右子树）
    - 若 v_i 加入后，U仍然是团?
      - 考虑将该顶点加入团或者舍弃两种情况
      - 怎样判断？ v_i 与U中其余顶点均直接相连

最大团问题

问题分析

- 剪枝函数
  - 约束函数：新加入的顶点是否构成团
    - 顶点vi到顶点集U中每一个顶点都有边相连
    - 否则可对以vi为根的左子树进行剪枝
  - 限界函数：当前扩展结点代表的团是否小于当前最优解
    - 若剩余顶点数加上当前团中顶点数不大于当前最优解
    - 则可以对以vi为根的右子树进行剪枝

最大团问题

```
void backtrack(int t){ int valid = 1;
    if (i > n){ // 数组m[]保存当前最优解
        for (int k = 1; k <= n; k++) m[k] = x[j];
        mn =  cn;  return;  // cn当前顶点数， mn 当前最大顶点数
    }
    for (int k = 1; k < i; k++){    // Vi是否与当前子图构成团
        if (x[k] && G[i][k] == 0){ // 不是完全图
            valid = 0; break;
        }
    }
    if (valid){ // 满足约束条件，进入左子树
        cn++; x[i] = 1;   backtrack(i+1);   x[i] = 0; cn--;
    }
    if (cn + n - i >= mn){ // 满足限界条件，进入右子树
        x[i] = 0;  backtrack(i+1);
    }
}
```

算法复杂度？ $O(n2^n)$

5.8 图的m着色问题

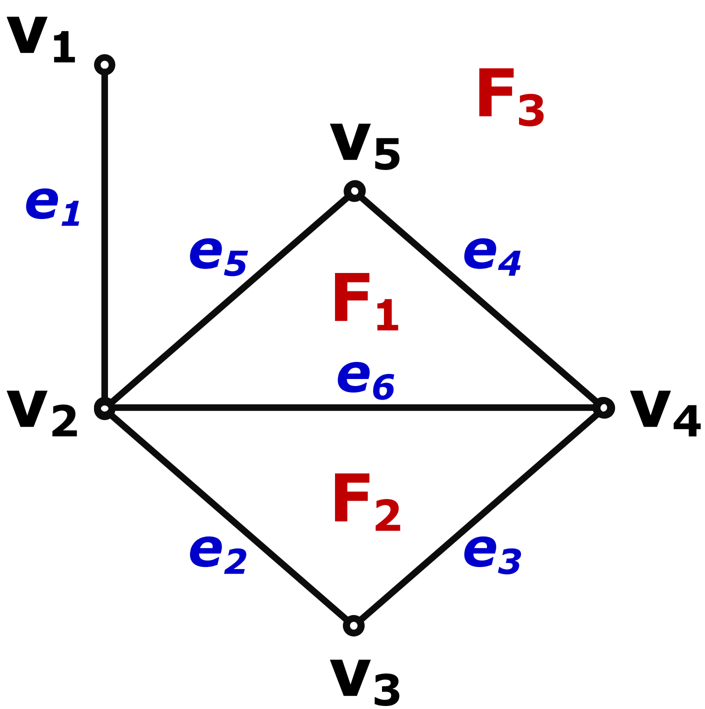
（ The M-Coloring Problem）

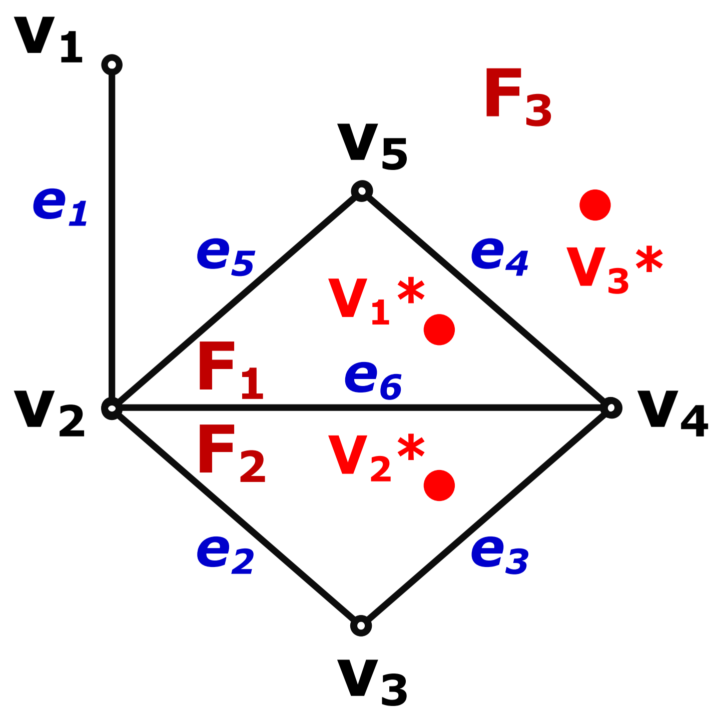
图的m着色问题

- 与平面图有密切关系的一个图论的应用是图形的着色问题
  - 这个问题最早起源于地图的着色
    - 一个地图中相邻国家着以不同颜色，最少需用多少种颜色？
  - 四色猜想：英国Guthrie提出了用四色即可对地图着色的猜想
  - 1879年，Kempe给出了这个猜想的第一个证明
  - 1890年，Hewood发现Kempe的证明是错误的
  - 然而Kempe的方法可证明用五种颜色就够了
  - 此后四色猜想一直成为数学家感兴趣而未能解决的难题
  - 1976年，美国数学家用电子计算机证明了四色猜想是成立的
  - 从1976年以后就把四色猜想这个名词改成四色定理了

对偶图（dual of graph）

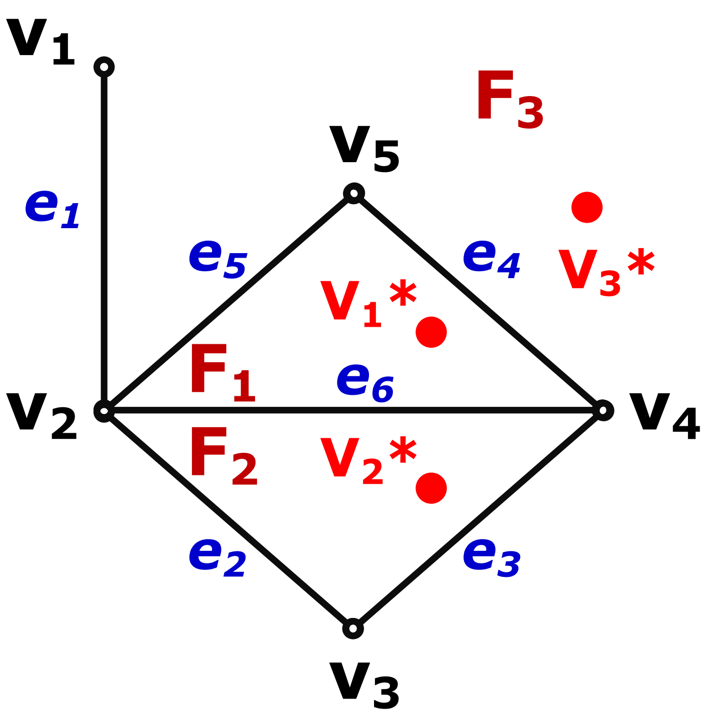
- 设：连通平面图G=<V,E>具有n 个面：F_1 ,F_2,…, F_n
- 如果存在一个图G*=<V*，E*>满足下述条件：
  - (1) 在G的每一个面 F_i 的内部作一个G*的顶点 v_i*
    - 即对图G的任一个面 F_i 内部有且仅有一个结点 v_i*∈V*

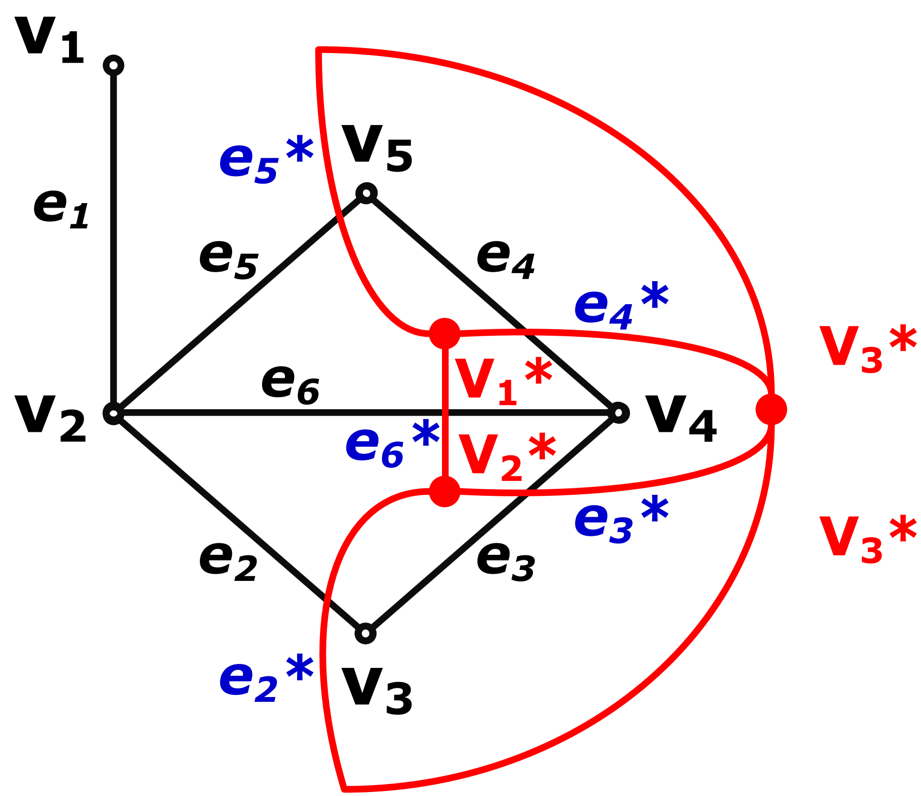




对偶图（dual of graph）

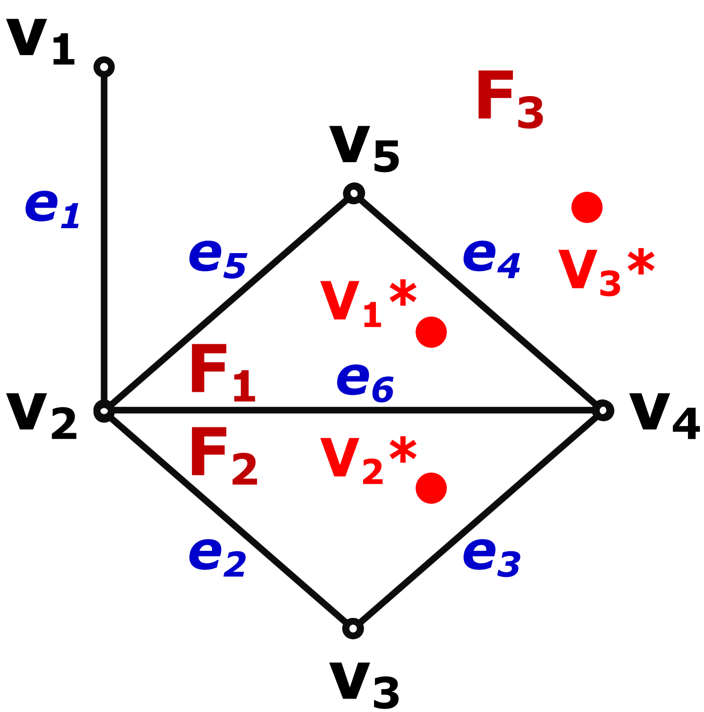
- 如果存在一个图G*=<V*，E*>满足下述条件：
  - (2) 若G的面 F_i 和 F_j 有公共边 e_k
    - 则：作 e_k*=(v_i*，v_j*)
    - 且：e_k* 与 e_k 相交；e_k* 与 G* 的其它边不相交

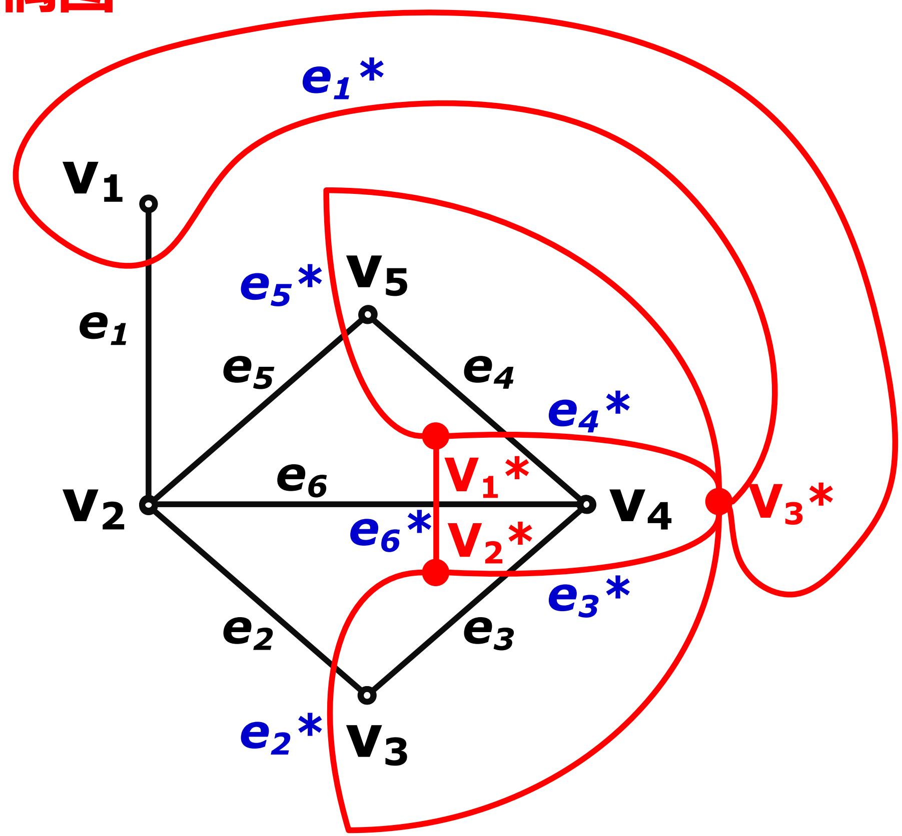




对偶图（dual of graph）

- 如果存在一个图G*=<V*，E*>满足下述条件：
  - (3) 当且仅当： e_k 只是一个面 F_i 的边界时（割边）
    - v_i* 存在一个环： e_k* 与 e_k 相交
- 由此得到的图G*称为图G的**对偶图**





图的m着色问题

- 从对偶图的概念可知
  - 给面着色就是给每个面指定一种颜色
    - 使得有公共边的两个面有不同的颜色
  - 因此地图着色问题可以转化为对平面图的结点着色问题
- **图的m着色问题**
  - 给定：无向连通图G和m种不同的颜色
  - 要求：用m种颜色为图G的每个顶点着一种颜色
    - 使得G中每条边的2个顶点着不同颜色
  - 该问题也称为图的m可着色判定问题
- **图的色数**
  - 对于图G着色时，需要的最少颜色数称为G的色数

图的m着色问题

问题分析

- 问题的解向量：(x_1, x_2, … , x_n)
  - 数组元素 **x[i]** 表示顶点所着的颜色编号
- 思考：采用哪种解空间树？　**子集树**
  - 问题的解空间可以表示为高度为n+1的完全m叉树
  - 每一层的结点都有m个子节点，表示m种可能的着色
- 剪枝函数
  - 为顶点 i 着色时，不能与已着色的相邻顶点颜色重复

图的m着色问题

```
bool Bound(int k) { // 检查颜色可用性
    for (int i = 1; i <= n; i++)
        if ((G[k][i]==1)&&(x[i]==x[k])) return false;
    return true;
}

void Backtrack(int t){
    if (t > n) {  // sum记录当前已找到的m着色方案数
        Output(x);   sum++;
    }
    else {
        for(int i = 1; i <= m; i++) {
            x[t] = i;
            if (Bound(t)) Backtrack(t+1);
        }
    }
}
```

$$\sum_{i=0}^{n-1} m^i (mn) = nm(m^n - 1)/(m-1) = O(nm^n)$$

解空间树中内结点个数： $\sum_{i=0}^{n-1} m^i$

算法复杂度?

$O(nm^n)$

在最坏情况下，对于每一个内结点，
检查其每种颜色的可用性需耗时O(mn)

图的m着色问题的应用

- **示例：考试安排问题**
  - 如何安排一次7门课程的考试日程？
  - 即：没有学生在同一时段需参加两门以上考试
- **问题分析**
  - 用无向图的结点表示课程
  - 若两门课程的学生有交集
    - 则在这两个结点之间增加一条边
  - 用不同颜色来表示考试的各个时间段
  - 则考试安排问题就转化为图的着色问题
    - 对结点进行正确着色，就可以避免学生的考试时间冲突
    - 对色数m的优化，即是对考试时间的优化

5.9 批处理作业调度问题

（ Batch Job Scheduling Problem）

批处理作业调度问题

- **给定n个作业的集合$\{J_1, J_2, \ldots, J_n\}$**
  - 每一个作业都有两项任务，需要分别在两台机器上完成
  - 每个作业必须先由机器1处理，然后再由机器2处理
  - 设：作业 J_i 需要机器 k 的处理时间为 t_{ki} (k=1,2)
- **定义：作业调度的完成时间和**
  - 对于一个确定的作业调度
    - 设：作业 J_i 在机器 k 上**完成处理**的时间为 F_{ki}
    - 所有作业在机器2上完成处理的时间之和 $f=\sum_{i=1}^{n} F_{2i}$
    - 称为该作业调度的完成时间和
- **批处理作业调度问题**
  - 对给定的n个作业，制定作业调度方案，使其完成时间和最小

批处理作业调度问题

- 示例：考虑如下 n=3 的批处理作业调度问题

t_{ki}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

- 这3个作业共有6种可能的调度方案
  - (1,2,3)；(1,3,2)；(2,1,3)；(2,3,1)；(3,1,2)；(3,2,1)
  - 相应的完成时间和分别是为：19；18；20；21；19；19
  - 显然：最佳调度方案是(1,3,2)；其完成时间和为18

批处理作业调度问题

- **问题分析**
  - 问题的解向量：$(x_1, x_2, \ldots, x_n)$
    - 数组元素 **x[i]** 表示该任务的调度顺序为 **i**
  - 思考：采用哪种解空间树？ **排列树**
    - 当$i<n$时，当前扩展结点位于排列树的第$i-1$层
    - 此时算法选择下一个要安排的作业
  - 剪枝函数
    - 若当前完成时间和大于已知的最优值，则剪去该子树

批处理作业调度问题

```
void Backtrack(int i){
    if (i > n) {
        for (int k = 1; k <= n; k++) mx[k] = x[k];
        m = f;  // 当前最小完成时间和
    }
    else {
        for (int k = i; k <= n; k++) {
            f1 += T[x[k]][1];  // 机器1完成处理时间
            f2[i] = ((f2[i-1] > f1) ? f2[i-1] : f1) + T[x[k]][2];
            f += f2[i];  // 当前的完成时间和
            if (f < m) {
                Swap(x[i], x[k]);
                Backtrack(i+1);
                Swap(x[i], x[k]);
            }
            f1 -= T[x[k]][1];  f -= f2[i];
        }
    }
}
```

算法复杂度?

$O(n!)$

5.10 回溯法的效率分析

回溯法的效率分析

- **回溯算法的效率在很大程度上依赖于以下因素**

1. 解空间的结构设计和产生x[k]的时间
2. 满足显约束的x[k]值的个数
3. 满足约束函数和上界函数约束的所有x[k]的个数
4. 计算约束函数constraint()和上界函数bound()的时间

- 好的约束函数设计能显著地减少所生成的结点数
  - 但这样的约束函数往往计算量较大
  - 因此，通常存在生成结点数与约束函数计算量之间的折衷
    - 我们希望总的计算时间较少
    - 而不是只考虑生成的结点数较少或约束函数容易计算

回溯法的效率分析

解空间的结构

- 对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的
- 重排：在其它条件相当的前提下，让可取值最少的x[i]优先
- 下图是关于同一问题的2棵不同的解空间树
  - 从中可以体会到这种策略的潜力