算法分析与设计

主讲教师： 刘峤

第6章：回溯算法

（Backtracking Algorithm）

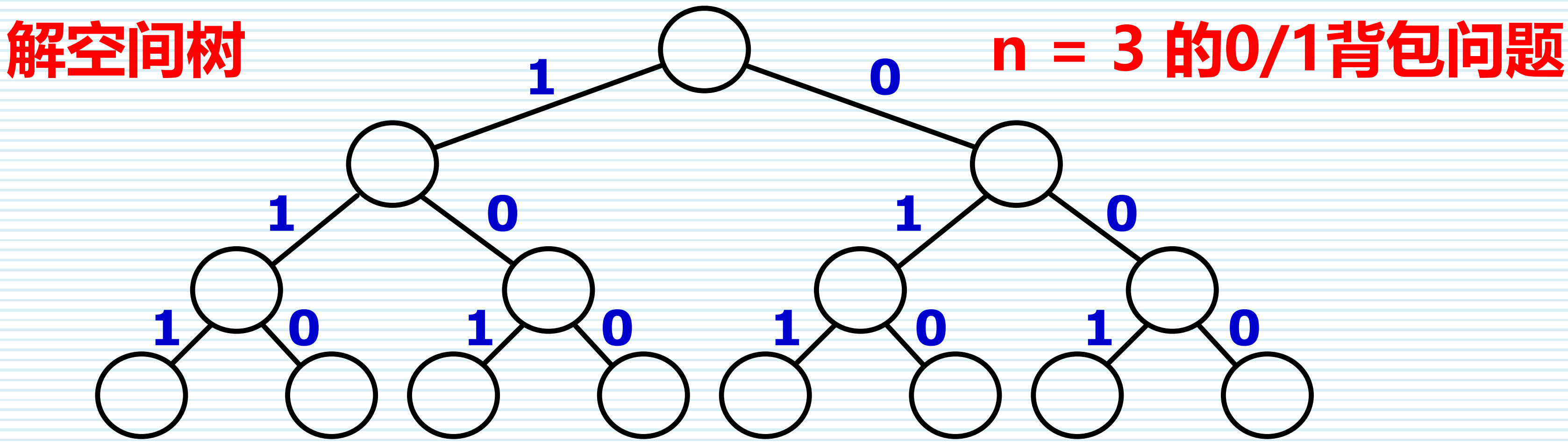
知识要点

- 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；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

回溯法算法框架

（ Backtracking Algorithm Paradigm ）

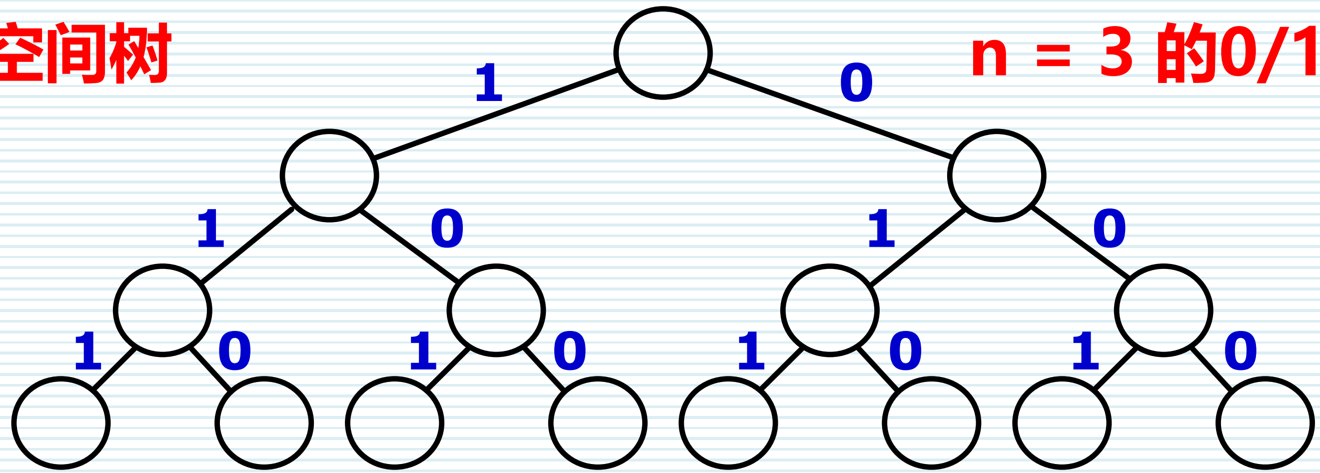
回溯法的基本概念



- 回溯法是一种有组织的穷举式搜索选优方法
  - 有组织是指：穷举过程中能够避免一些不必要的搜索
  - 通用的解题方法：适用于解一些组合数相当大的问题
- 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

问题的解空间

解空间树　　**n = 3 的0/1背包问题**



- 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

回溯法的基本概念

- 回溯法解题思路
  - 按选优条件对解空间树T进行深度优先搜索以找到目标
- 回溯法解题流程
  - 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，首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）
  - 算法结束条件
    - 求任一解：搜索到问题的一个解，算法结束
    - 求所有解：回溯到根，且根的所有子树均已搜索完成

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
    - 从当前扩展结点开始进一步的搜索
        - 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
- 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

```
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：为解空间树的高度

t > n：表示已搜索到一个叶结点

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

```
void backtrack (int t) {
    if ( t > n)   output(x);
    else{
        for (int i = s(n, t); i <= e(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

s(n, t)表示在当前扩展结点处未搜索过的子树的起始编号

n(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

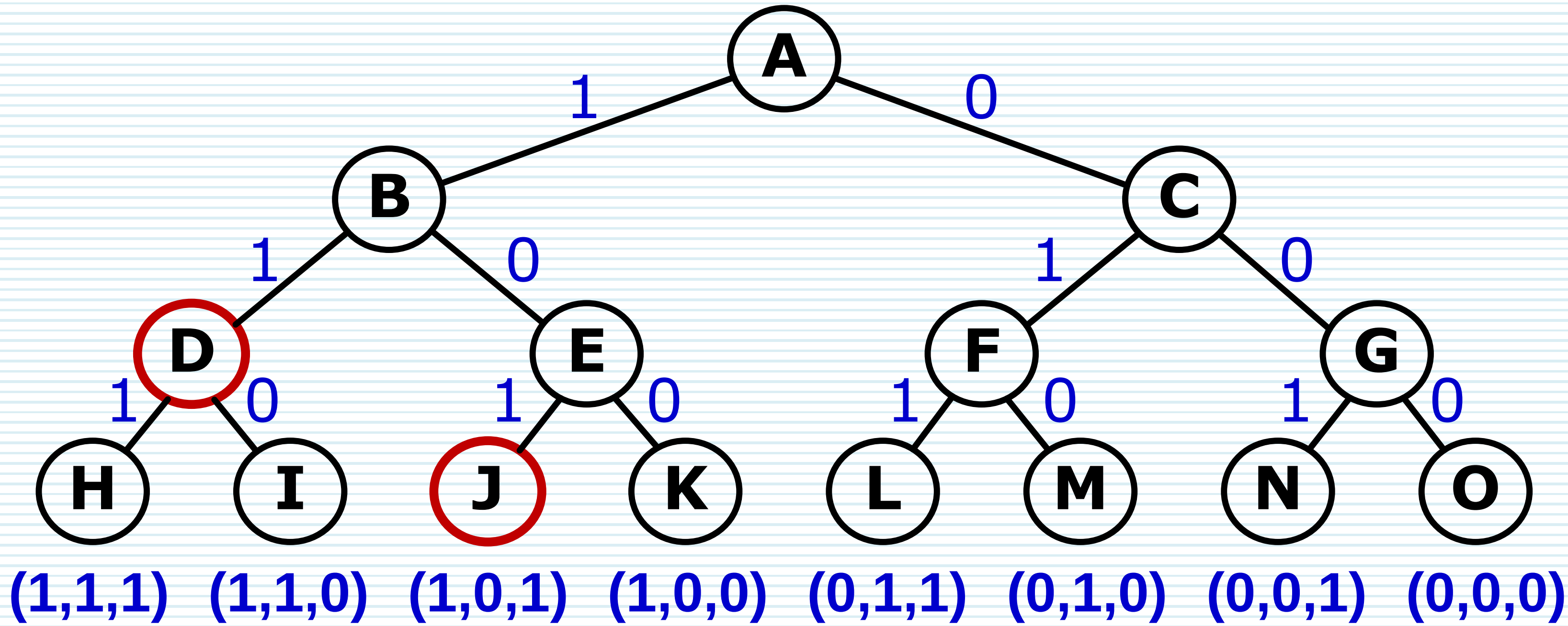
bound(t)为true表示在当前扩展结点处x[1:t]的取值尚未导致目标函数越界

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**
  - 问题：从n个元素的集合S中找出满足某种性质的子集
  - 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
    - 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**
  - 问题：确定n个元素满足某种性质的排列
  - 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，遍历排列树需要$\Omega(n!)$计算时间

子集树示例： 0/1背包问题

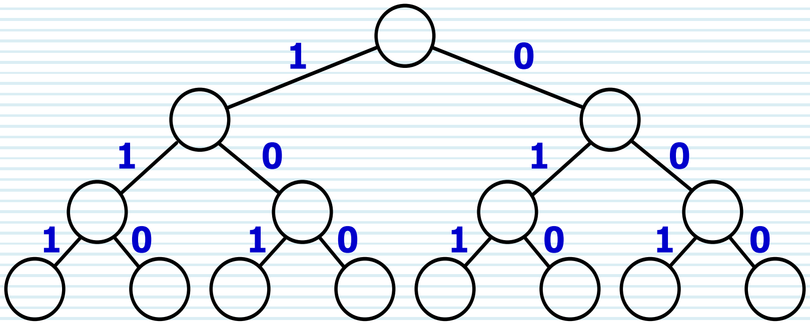
- 设：n=3, w=(16, 15, 15), v=(45, 25, 25), c=30
  - 定义解空间：X={(0,0,0), (0,0,1), (0,1,0),…, (1,1,0), (1, 1, 1) }
  - 构造解空间树如图：从A出发按DFS搜索



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

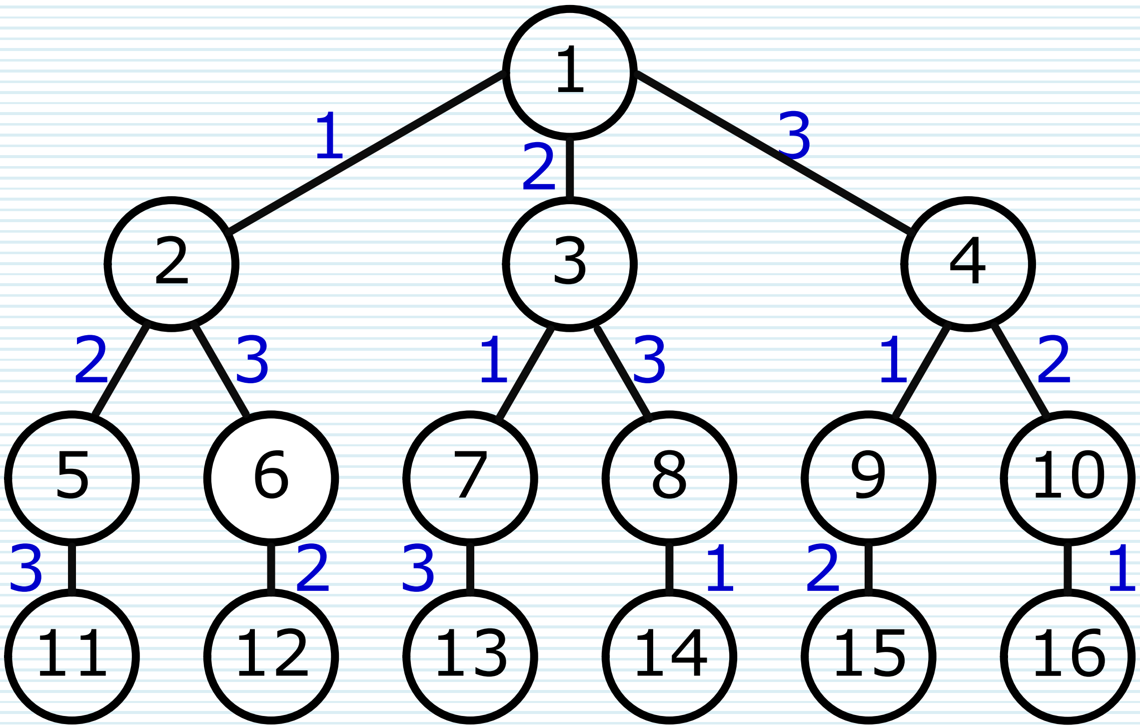



```
void backtrack (int t) {
    if (t > n)  output(x);
    else{
        // 对当前扩展结点的所有可能取值进行枚举
        for (int i = 0; i <= 1; i++) {
            x[t] = i;
            if (constraint(t) && bound(t))
                backtrack(t+1);
        }
    }
}   // 执行时，从Backtrack(1)开始
```

遍历子集树：$O(2^n)$

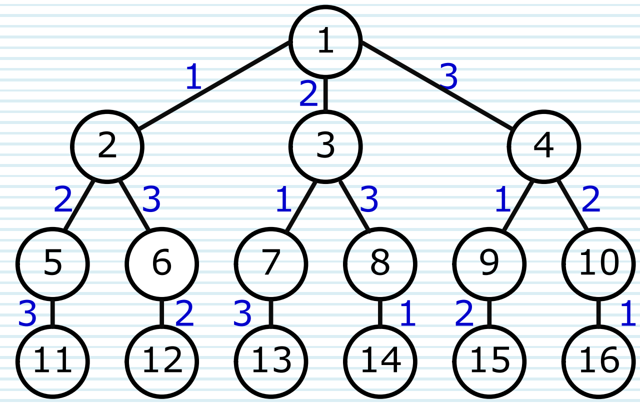
排列生成问题

- 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]);
      }
   }
}
```

算法输出：

123
132
213
231
321
312

```
int main(void){  int n = 3;
   for (int i=1; i <= n; i++) x[i] = i;
   backtrack(1);
}
```

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

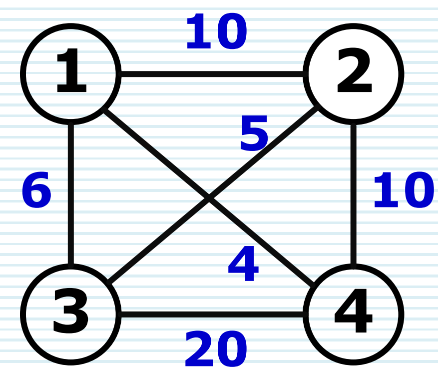
- 约束函数：在扩展结点处剪去**不满足约束**的子树
  - 回溯法要求问题的解能够表示成n元向量形式($x_1,x_2,\ldots,x_n$)
  - 显式约束：对分量 x_i 的取值范围限制
  - 隐式约束：为满足问题的解而对不同分量之间施加的约束
- 限界函数（bounding function）：剪去**得不到最优解**的子树
  - 为了避免生成那些不可能产生最佳解的问题状态
  - 要不断地利用限界函数来剔除那些不能产生所需解的活结点
  - 具有限界函数的深度优先搜索法就称为回溯法

旅行商问题

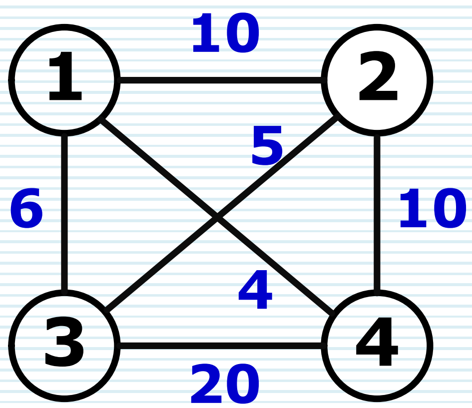
（ Travelling Salesman Problem ）

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）
  - 要求选择一条满足条件且总开销最小的路线
    - 限制条件：从驻地出发途经每个城市一次后回到驻地
  - 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



求解TSP问题



解空间：X =

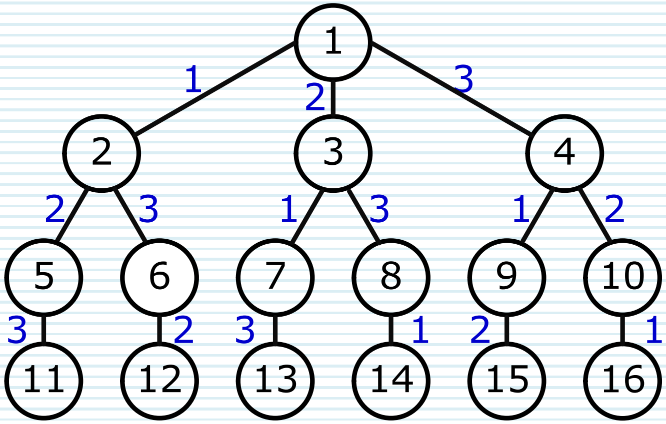
12341, 12431

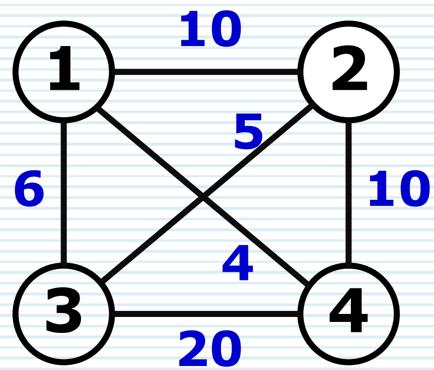
13241, 13421

14231, 14321

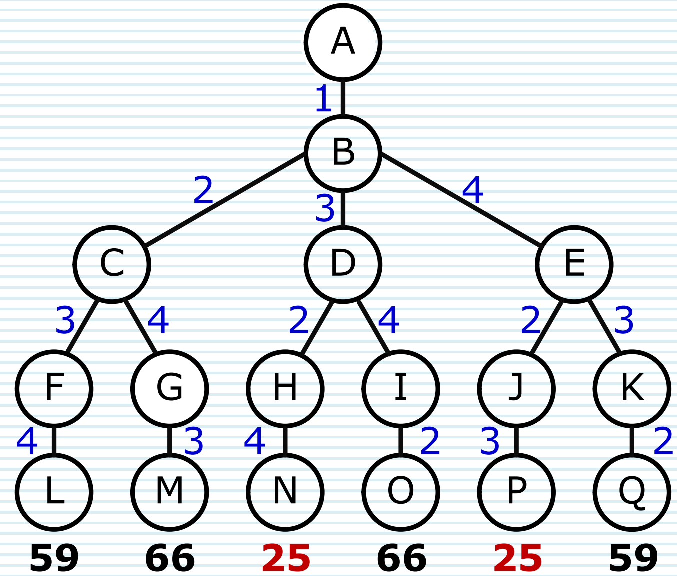
- 问题分析：与排列生成问题相比多了一个回路
- 解题思路：利用排列生成问题的回溯算法Backtrack()
  - 提示：Backtrack(2)表示？
    - 对x={1, 2, …, n}中的x[2..n]进行全排列
    - 则：(x[1], x[2])，…, (x[n], x[1])构成回路
  - 在全排列算法的基础上进行路径计算保存以及限界剪枝

排列树示例：旅行商问题





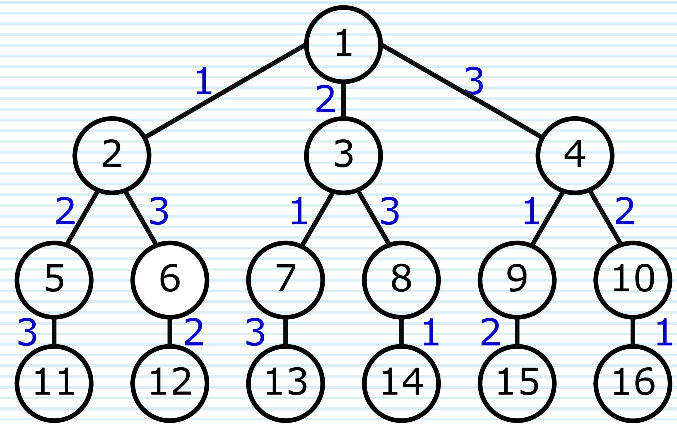
最优解1：13241
最优解2：14231
最优值：m = 25

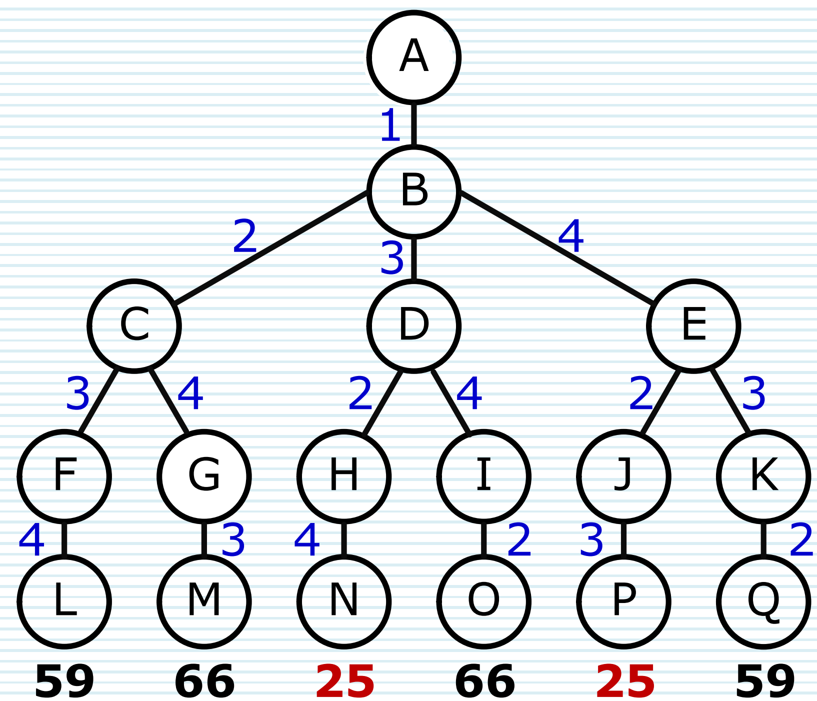


- 解空间：X={12341, 12431, 13241, 13421, 14231, 14321}
- 构造解空间树
  - 从根结点到任一叶结点的路径定义了图G的一条周游路线
    - 例如：A->L 对应周游路线（1, 2, 3, 4, 1）
  - 解空间树中的每个叶结点恰好对应于图G的每一条周游路线
    - 解空间树中的叶结点个数为：**(n-1)!**

排列树回溯算法框架

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





求解TSP问题

```
void backtrack(int i){   // d为静态变量；m保存当前最优值
    if (i > n)
        if ( d+ A[x[n]][1] < m ){
            m = d+ A[x[n]][1];   output(x); }
    else{
        for( k = i ; k<=n; k++ )
            swap( x[i], x[k]);   d+= A[x[i-1]][x[i]];
            if ( d+ A[x[i-1]][x[i]] < m )
                backtrack(i+1);
            d-= A[x[i-1]][x[i]];   swap(x[i], x[k]);
    }
}
// 初始调用：Backtrack(2)
```

算法复杂度：O(n!)

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：$\mathbf{x} = (x_1, x_2, \ldots, x_n)$
  - 解空间树：子集树
  - 剪枝
    - 约束函数： $\sum_{i=1}^{n} w_i x_i \le c$
    - 限界函数：剔除不能产生所需解的活结点
    - 思考：怎样设计限界函数?

0/1背包问题

- 限界函数的设计思路
  - 设：当前扩展结点为x_r
  - 考查：所有经过 x_r 的可行解
    - 其中某些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 x_r 的可行解的价值上界？
    - 思路：分为两部分
    - value（从根到x_r ）+ value（以 x_r 为根的子树）

计算可行解的价值上界

- 例：n = 4，c = 7，v = [9, 10, 7, 4]，w = [3, 5, 2, 1]
  - 物品的单位重量价值为：[3, 2, 3.5, 4]
    - 按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
int Bound(int i, int vc, int wc, int c){
    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, int vc, int wc, int c){  // m当前最优价值
    if( i > n) { m = ( m < vc )? vc : m;   output(x); }
    else {
        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, vc, wc, c);
            x[i]= 0; wc -= w[i]; vc -= v[i];
        }
        if(Bound(i+1, vc, wc, c) > m){  // 右子树（不放入i）
            backtrack(i+1, , vc, wc, c);
        }
    }
}
```

算法复杂度：$O(n2^n)$

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方案可将这n个集装箱装上这2艘轮船？如果问题有解，给出一种装载方案
- 示例：n=3，$c_1 = c_2 = 50$
  - 若：w=[10, 40, 40]
    - 则可将1和2装上第一艘船，将3装上第二艘船
  - 若：w=[20, 40, 40]
    - 则无法将这三个集装箱全部装船

装载问题

- 回顾：一艘船的装载问题（满足重量约束，尽可能多装集装箱）
  - 采用贪心选择策略：从轻到重依次装船，直至超重
  - 思考：现在有两艘船，如何求解？
- 若给定问题有解，则采用如下策略可得最优装载方案
  - 首先将第一艘轮船尽可能装满
  - 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，使其重量之和最接近 c_1
- 显然：装载问题等价于以下特殊的0-1背包问题

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

回溯法求解最优装载问题

- 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
  - 剪枝策略
    - 约束函数：$\sum_{i=1}^{n} w_i x_i \le c_1$
    - 在子集树第 k 层的结点 x_r 处
      - 以 c_k 表示当前的装载重量：$c_k = \sum_{i=1}^{k} w_i x_i$
    - 则当 $c_k > c_1$ 时，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

回溯法求解最优装载问题

- 算法设计：用回溯法求解最优装载问题
  - 剪枝策略：限界函数（提前修剪不含最优解的子树）
    - 设：x_r 是解空间树第k层上的当前扩展结点
    - 设：wc表示 x_r 对应的的装载重量 $wc = \sum_{i=1}^{k} w_i x_i$
    - 设：m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x_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≤m 时，可将 x_r 的右子树剪去

回溯法求解最优装载问题

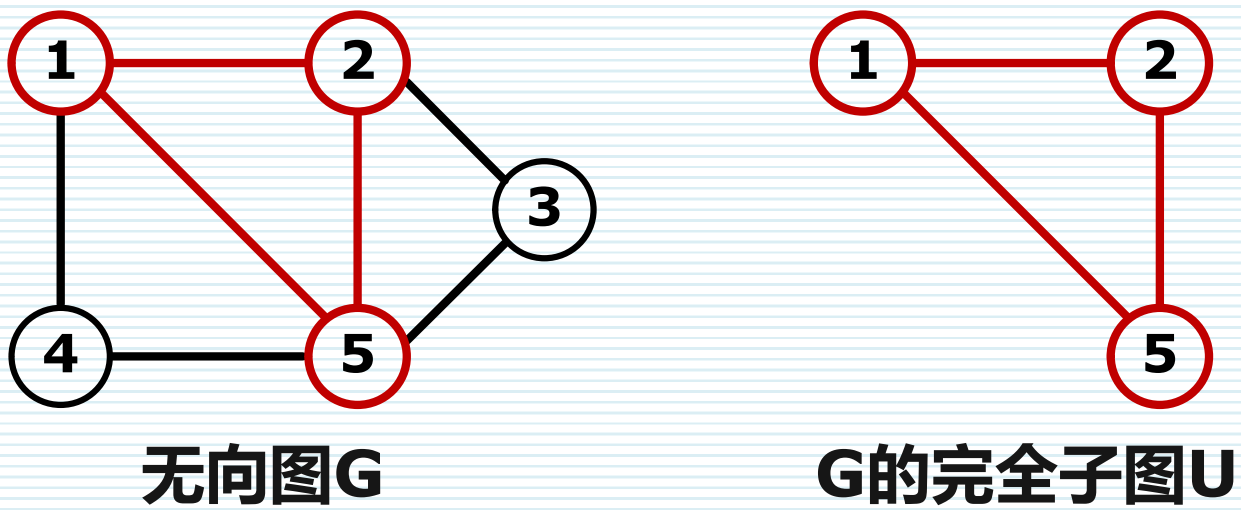
```
void backtrack (int i) {
    if (i > n){
        if(wc > m)  m = wc;  return;
    }
    wr -= w[i];
    if (wc + w[i] <= c){   // x[i] = 1; 搜索左子树
        x[i]= 1;  wc += w[i];
        backtrack(i+1);
        x[i]= 0;  wc -= w[i];
    }
    if (wc + wr > m){   // x[i] = 0; 搜索右子树
        backtrack(i+1);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

算法复杂度：$O(2^n)$

最大团问题

（ Maximum Clique Problem ）

最大团问题

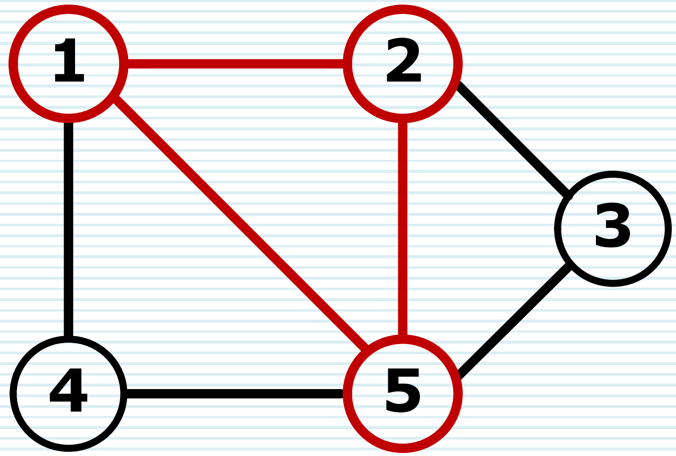


- 问题描述

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
- 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

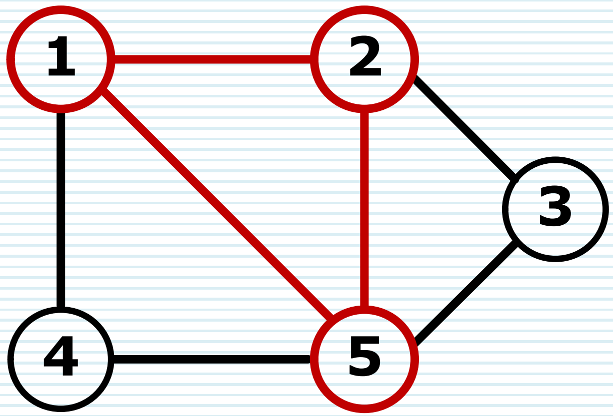
最大团问题



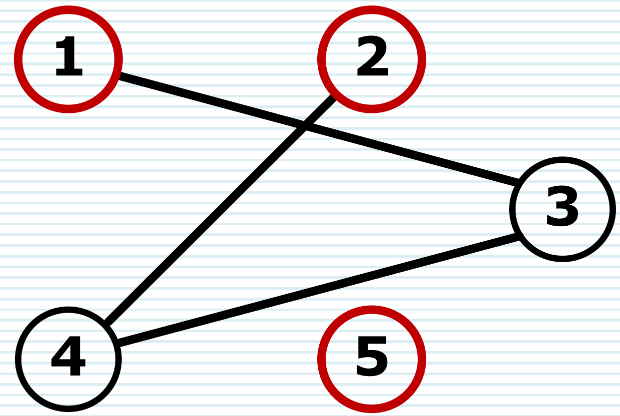
无向图G

- 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



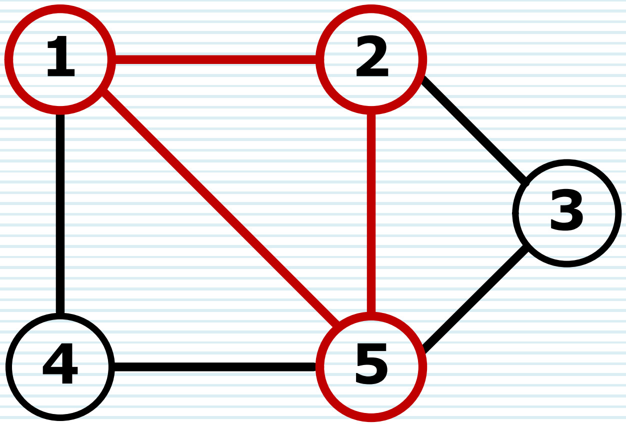
无向图G



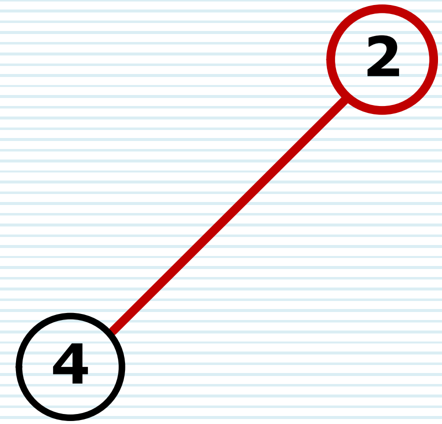
图G的补图G′

- 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

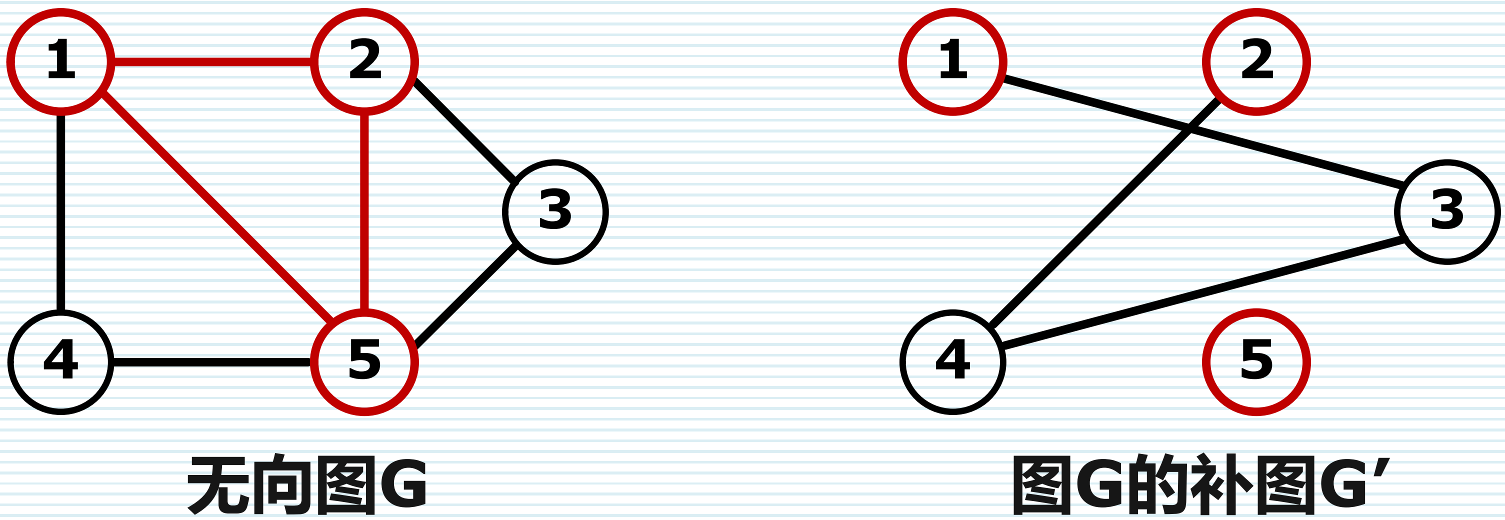


图G的空子图

最大独立集

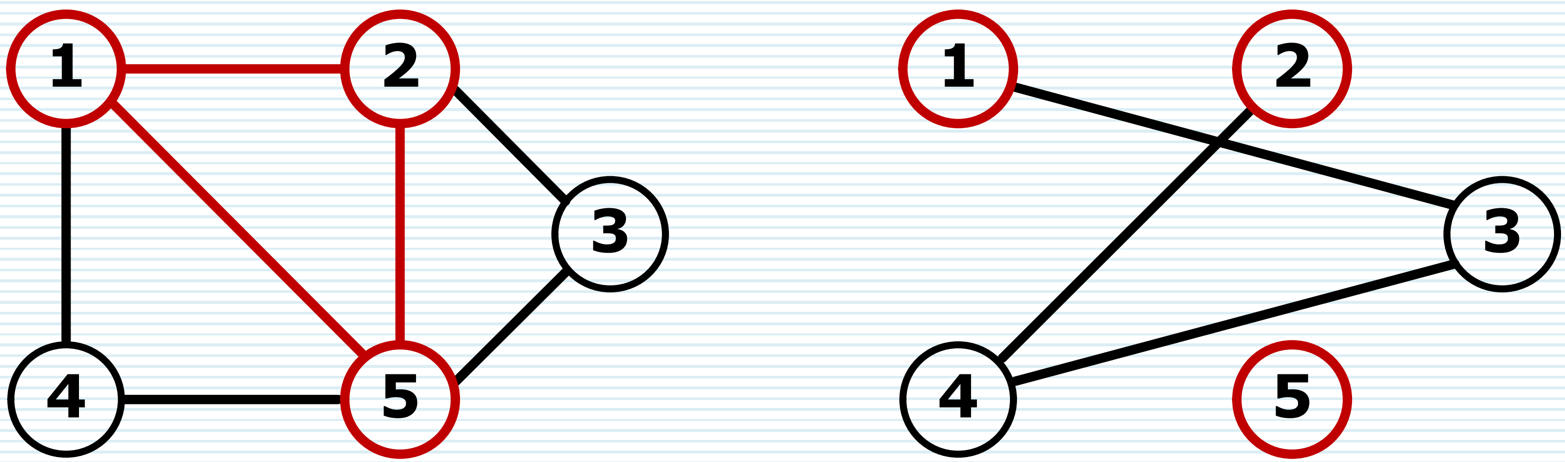
- 如果$U \subseteq V$且对任意u，$v \in U$有$(u，v) \notin E$，则称U是G的空子图
- 空子图U是G的独立集当且仅当U不包含在G的更大的空子图中
- G的最大独立集：是G中所含顶点数最多的独立集

最大团问题



- 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**

最大团问题分析

- 剪枝策略
  - 约束函数：新加入的顶点是否构成团
    - 顶点 v_i 到顶点集U中每一个顶点都有边相连
    - 否则可对以 v_i 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v_i 为根的右子树进行剪枝

最大团问题

```
void backtrack(int i, int c){
    int valid = 1;  // c当前顶点数， m当前最大顶点数
    if (i > n){  output(x);  m =  c;  return; }
    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){ valid = 0;   break; }
    }
    if (valid){   // 进入左子树
        c++; x[i] = 1;   backtrack(i+1, c);   x[i] = 0; c--;
    }
    if (c + n - i >= m){ backtrack(i+1, c);  } //进入右子树
}
```

算法复杂度？ $O(n2^n)$

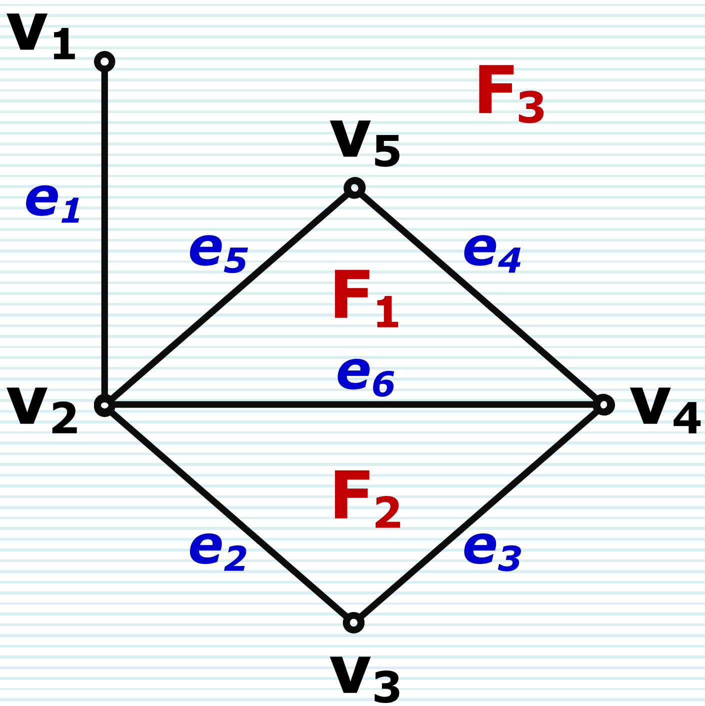
图的m着色问题

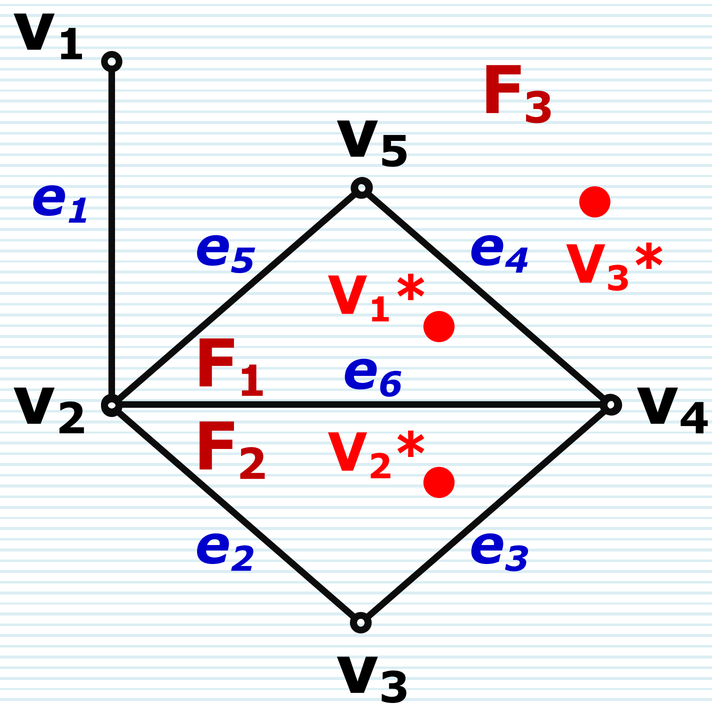
（ The M-Coloring Problem ）

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
  - 问题最早起源于地图的着色
    - 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，美国数学家用计算机证明了四色猜想成立
    - 从1976年以后就把四色猜想这个名词改成四色定理了

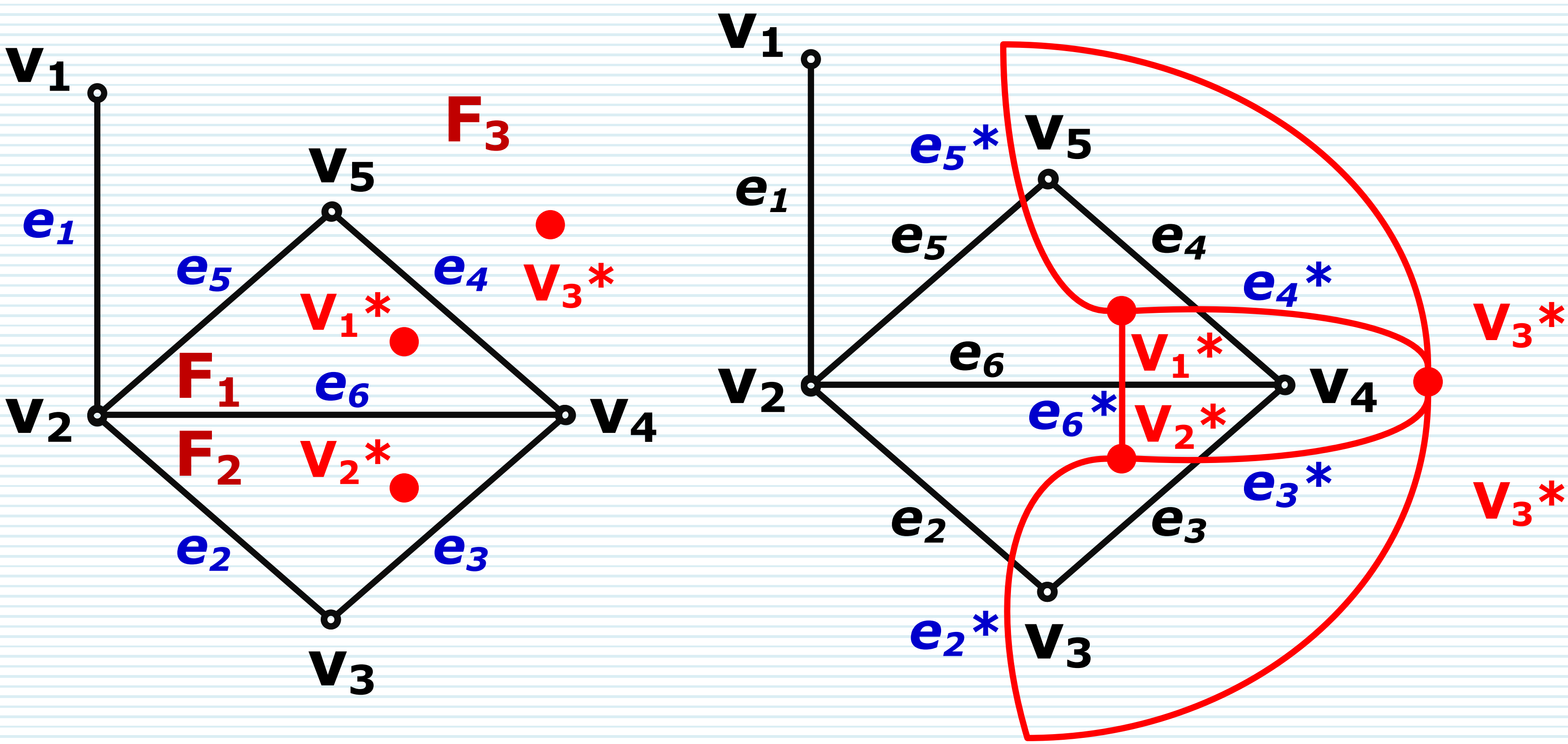
对偶图（dual of graph）





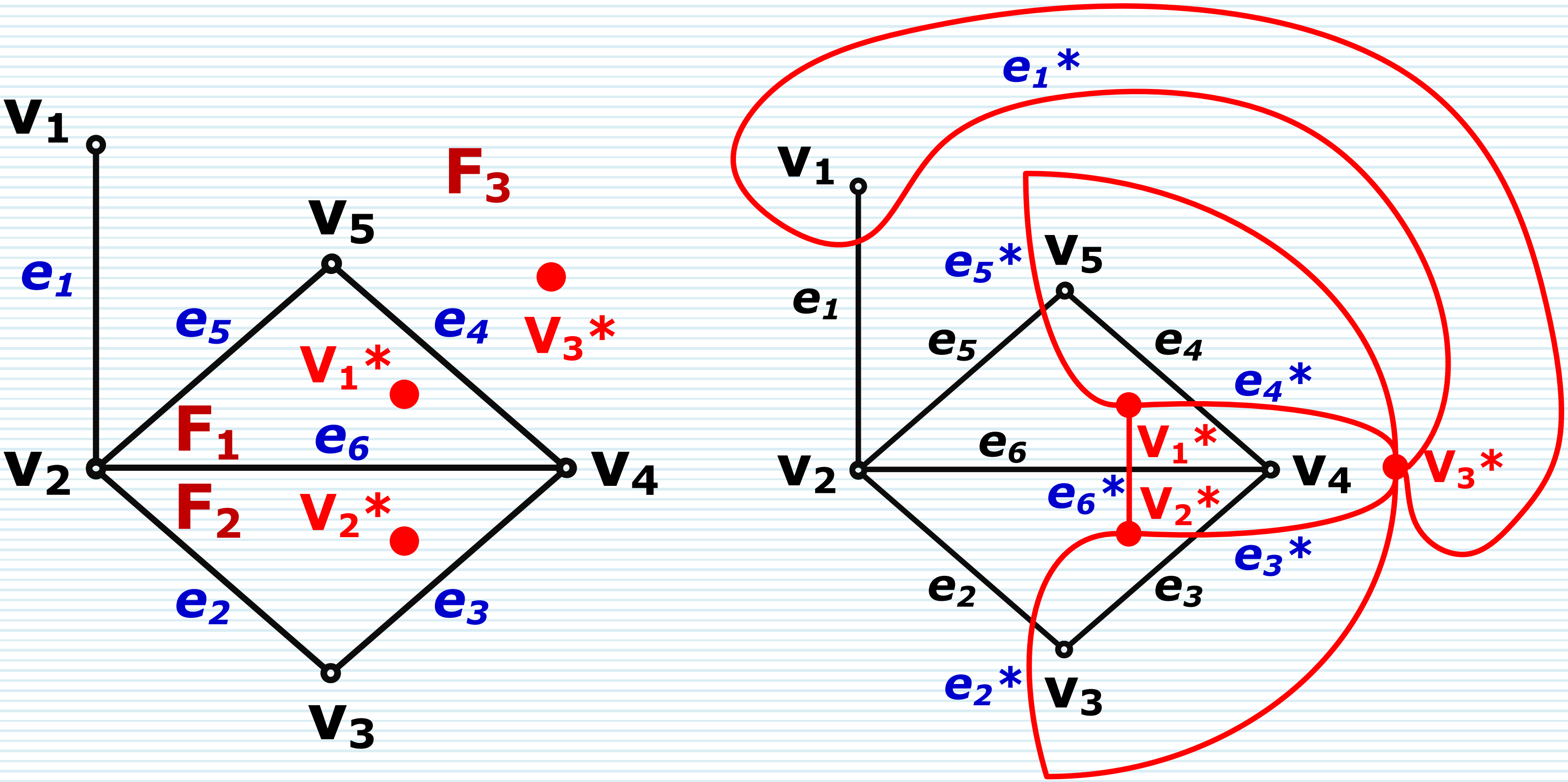
- 设：连通平面图G=<V,E>具有n个面：F_1 ,F_2, … , F_n
- 若：存在一个图G*=<V*，E*>满足下述条件
  - (1) 在G的每一个面 F_i 的内部作一个G*的顶点 v_i*
    - 即对图G的任一个面 F_i 内部有且仅有一个结点 $v_i^* \in V^*$

对偶图（dual of graph）



- (2) 若G的面 F_i 和 F_j 有公共边界 e_k
  - 则过边界e_k作关联v_i*与v_j*的一条边：$e_k^*=(v_i^*, v_j^*)$
    - 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）



- (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
  - 着色问题就是使得：有公共边的两个面有不同的颜色
  - 由此地图着色问题可以转化为对平面图的结点着色问题
- 图的色数：对图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
  - 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m着色问题

- 问题分析
  - 问题的解向量：$(x_1, x_2, \dots, x_n)$
    - 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
  - 剪枝策略
    - 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

- 剪枝策略
  - 约束条件：顶点 i 着色时不能与已着色的相邻顶点颜色重复

```
// 检查颜色可用性
int bound(int k) {
    for (int i = 1; i <= n; i++)
        if ((G[k][i]==1)&&(x[i]==x[k]))
            return 0;  // 颜色发生冲突
    return 1;
}
```

图的m着色问题

```
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：$\sum_{i=0}^{n-1} m^i$

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)

算法复杂度？

$O(nm^n)$

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

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$\{\mathcal{J}_1, \mathcal{J}_2, \ldots, \mathcal{J}_n\}$
  - 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 $\mathcal{J}_i$ 需要机器 **k** 的处理时间为 t_{ki}（k=1,2）
- 对于一个确定的作业调度
  - 设：作业 $\mathcal{J}_i$ 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

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, … , 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
  - 剪枝策略
    - 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) {  output(x);   m = f; }  // m记录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完成处理的时间
            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？

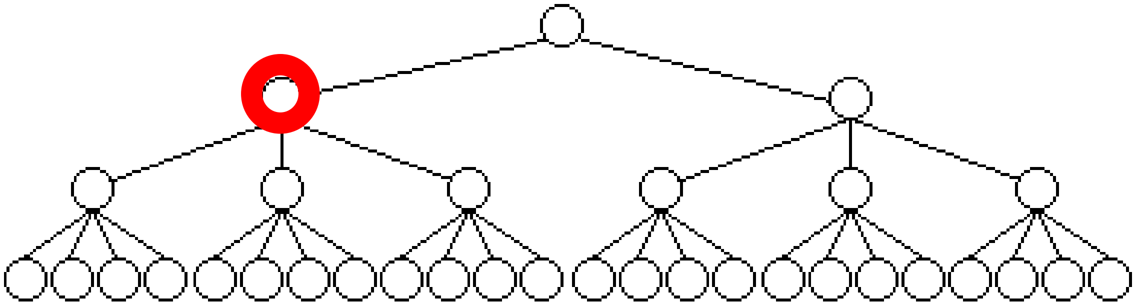
O(n!)

回溯法的效率分析

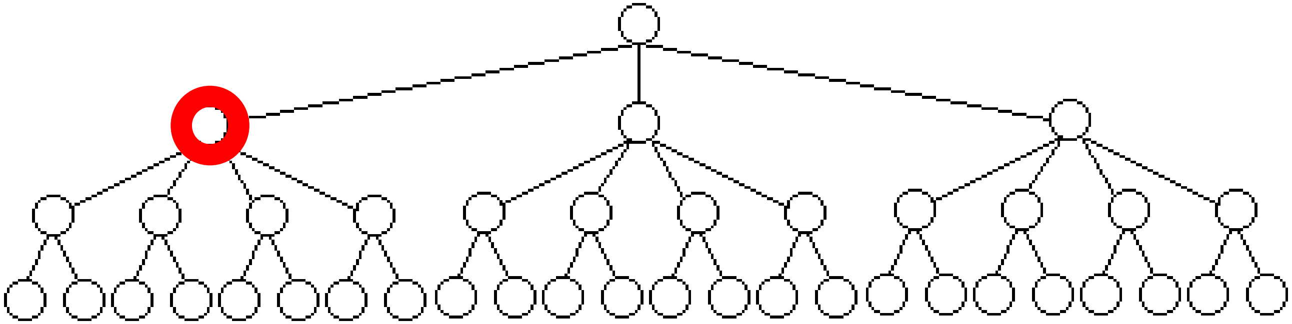
回溯法的效率分析

- 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. 满足显式约束的x[k]值的个数
     - 对分量 x_i 的取值范围限制
  3. 满足约束函数和上界函数约束的所有x[k]的个数
  4. constraint()函数和bound()函数的运行（计算）时间
- 好的约束函数设计能显著地减少所生成的结点数
  - 然而由此带来的计算开销也不容忽视
  - 因此算法设计通常是在二者之间取得折衷

回溯法的效率分析：解空间的结构

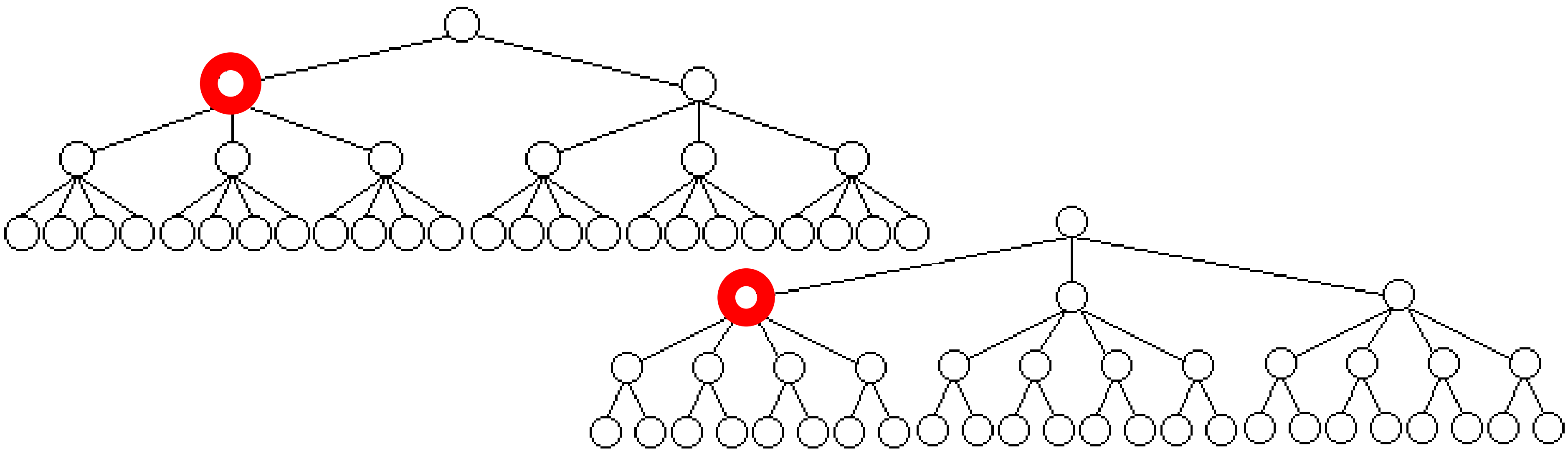


- 从第1层剪去1棵子树
  - 则从所有应当考虑的3元组中一次消去12个3元组



- 从第1层剪去1棵子树
  - 只从应当考虑的3元组中消去8个3元组

回溯法的效率分析：解空间的结构



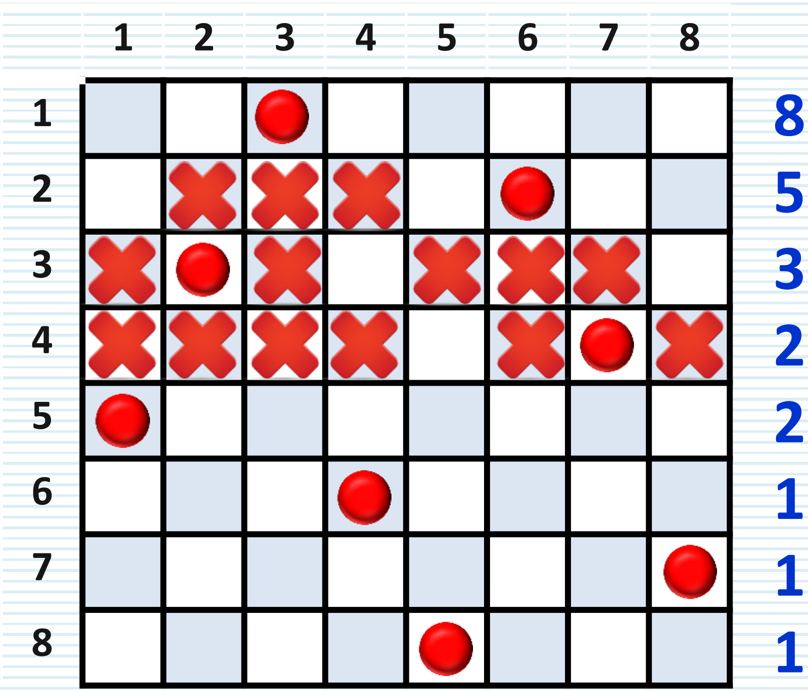
- 解空间的结构
  - 对许多问题而言，解向量**x**中元素的顺序是任意的
  - 通过对**x**进行重排，有时可以提高算法的执行效率
    - 优先试探搜索可取值最少的x[i]

回溯法的效率分析

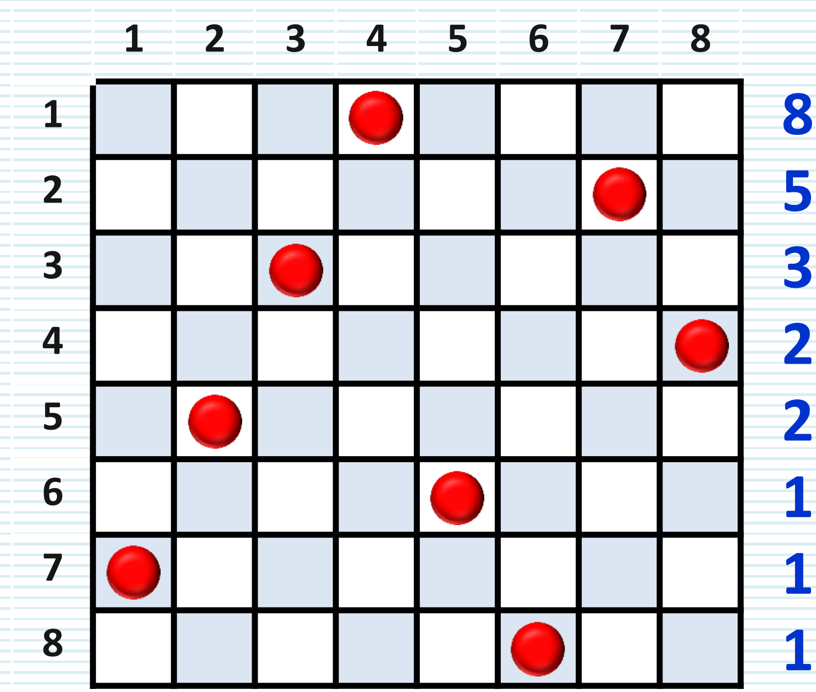
- 效率估算：蒙特卡洛方法
  - 以8-皇后问题为例

解空间树结点总数=109601

回溯法搜索效率≈2.12%



结点总数=2329



结点总数=2329

结点总数： $m = 1 + m_0 + m_0 m_1 + \ldots\ldots = 1 + \sum_{i=0}^{n-1} \left(\prod_{j=0}^{i} m_j \right)$

分支限界法

（Branch and Bound Method）

分支限界法

- 分支限界法的基本思想
  - 以广度优先或以最小耗费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子结点
    - 其中不可能导出最优解的子结点被舍弃（限界策略）
    - 其余子结点被加入活结点表中（分支策略）
  - 然后从活结点表中取下一结点成为当前扩展结点
  - 重复上述结点扩展过程
    - 直至到找到所需的解或活结点表为空时为止

分支限界法

- 分支限界法与回溯法的类似之处
  - 基本思路：在问题的解空间树上搜索问题的解
- 分支限界法与回溯法的区别
  - 求解目标不同
    - 回溯法的目标是找出解空间树中满足约束条件的所有解
    - 分支限界法的目标则是尽快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
    - 通常用于解决离散值的最优化问题
  - 搜索方式不同
    - 回溯法以深度优先的方式搜索解空间树（遍历）
    - 分支限界法以广度优先或最小耗费优先的方式搜索解空间树

分支限界法

- 分支限界法与回溯法的区别
  - 对扩展结点的扩展方式不同
    - 分支限界法中，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
  - 存储空间的要求不同
    - 分支限界法的存储空间比回溯法大得多
    - 因此当内存容量有限时，回溯法成功的可能性更大
- 二者区别小结
    - 回溯法空间效率高；分支限界法往往更“快”
    - 限界函数常基于问题的目标函数，适用于求解最优化问题

两种常见的分支限界法

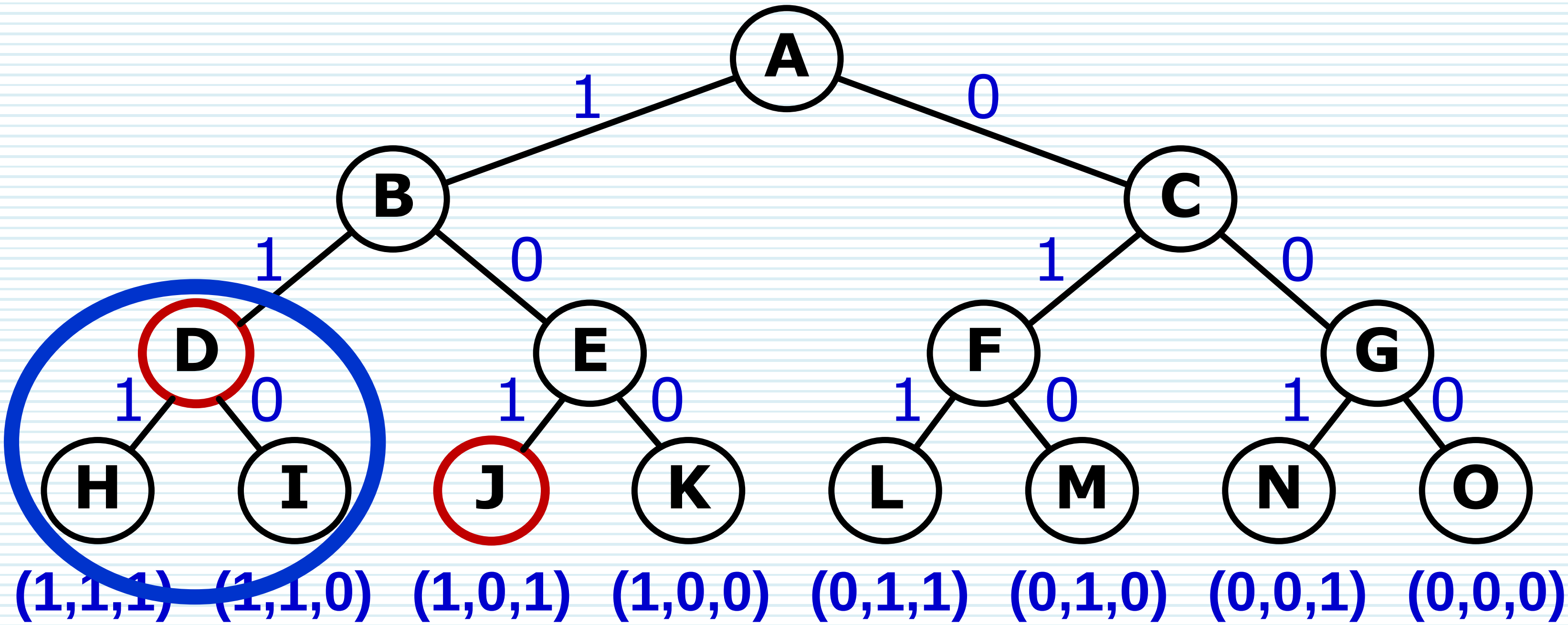
- 队列式分支限界法
  - 按照先进先出（FIFO）原则选取下一个结点为扩展结点
  - 从活结点表中取出结点的顺序与加入结点的顺序相同
  - 因此活结点表的性质与队列相同

两种常见的分支限界法

- 优先队列分支限界法（代价最小或效益最大）
  - 每个结点都有一个对应的耗费或收益
    - 以此决定结点的优先级
  - 从优先队列中选取优先级最高的结点成为当前扩展结点
    - 如果希望搜索一个具有最小耗费的解
      - 则可用小顶堆来构造活结点表
      - 下一个扩展结点就是具有最小耗费的活结点
    - 如果希望搜索一个具有最大收益的解
      - 则可用大顶堆来构造活结点表
      - 下一个扩展结点就是具有最大收益的活结点

示例：0/1背包问题

- 设：n=3, w=(16, 15, 15), v=(45, 25, 25), c=30
- 解空间：{(0,0,0), (0,0,1), (0,1,0),…, (1,1,0), (1,1,1) }
- 构造解空间树如图；从A出发按BFS搜索



可行解的值：　45　50　25　25　0

最优解： x = (0, 1, 1)　最优值：m = 50

示例：0/1背包问题（优先队列）

- 设：n=3, w=(16, 15, 15), v=(45, 25, 25), c=30
- 与回溯法相比，分支限界法可根据限界函数不断调整搜索方向，选择最可能得最优解的子树优先进行搜索，进而找到问题的解

优先队列：A

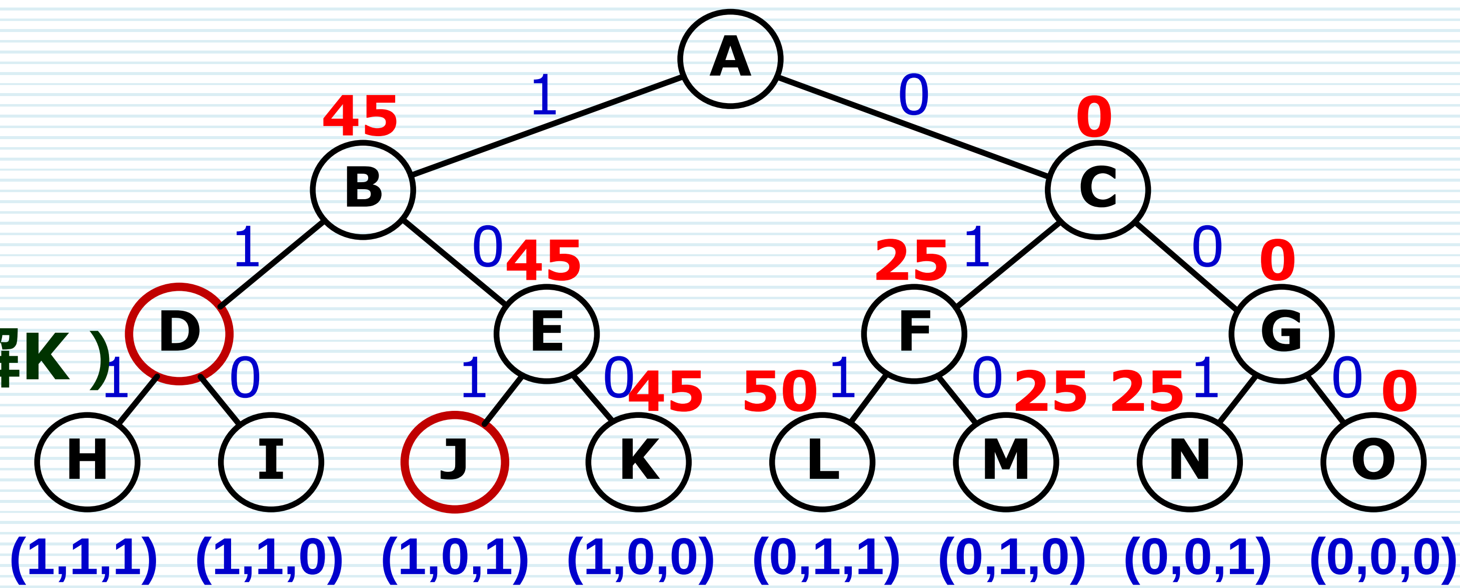
B, C

E, C

K, C（可行解K）

F, G

L, M, G



可行解：L, M　最优解： x = (0, 1, 1)　最优值：m = 50

分支限界法小结

- 适用于求解最优解或任意一个解的问题
- 分支限界条件
  - 优化问题：约束条件 + 代价函数
  - 采用启发式算法估计上界
    - 如果一个解的下限超出了这个上界
    - 则这个解不可能是最优解，此时这个分支就可以被剪去
- 算法复杂性
  - 遍历搜索树的时间：最坏情况为指数复杂度
  - 平均时间复杂性较小（相对于回溯法而言）

电子科技大学

UESTC

1956