算法分析与设计

Analysis and Design of Algorithm

第12次课

课程回顾

- 贪心算法的基本概念
  - 贪心选择性质
  - 局部最优和全局最优
- 贪心算法的应用
  - 哈夫曼编码、最小生成树、单源最短路径
- NP完全问题
  - 多机调度问题、旅行商问题

第五章 回溯法

学习要点

- 理解回溯法的深度优先搜索策略
- 掌握用回溯法解题的算法框架
  - 递归回溯
  - 迭代回溯
  - 子集树算法框架
  - 排列树算法框架
- 应用范例
  - 装载问题；批处理作业调度；符号三角形问题；n后问题；0-1背包问题；最大团问题；图的m着色问题；旅行售货员问题

例子：0-1背包问题

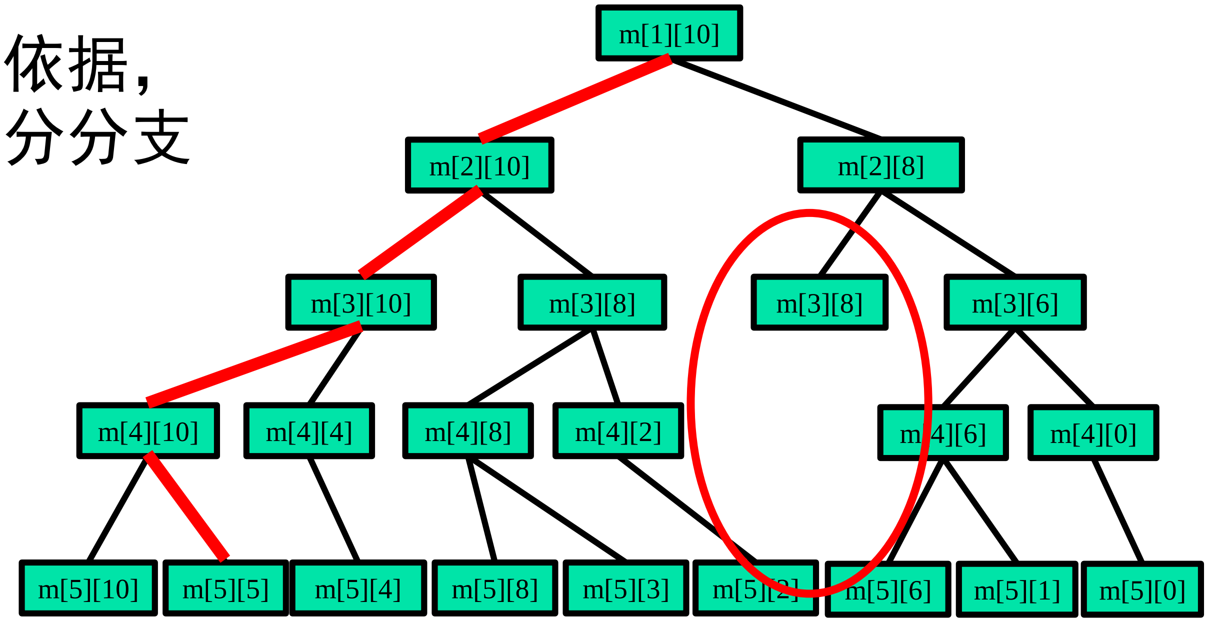
问题：有n种物品，每种物品的重量和价值分别为w_i, v_i。如果背包的最大承重限制是B，每种物品至多放1个。怎么样选择放入背包的物品使得背包所装物品价值最大？

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

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

解空间树与剪枝

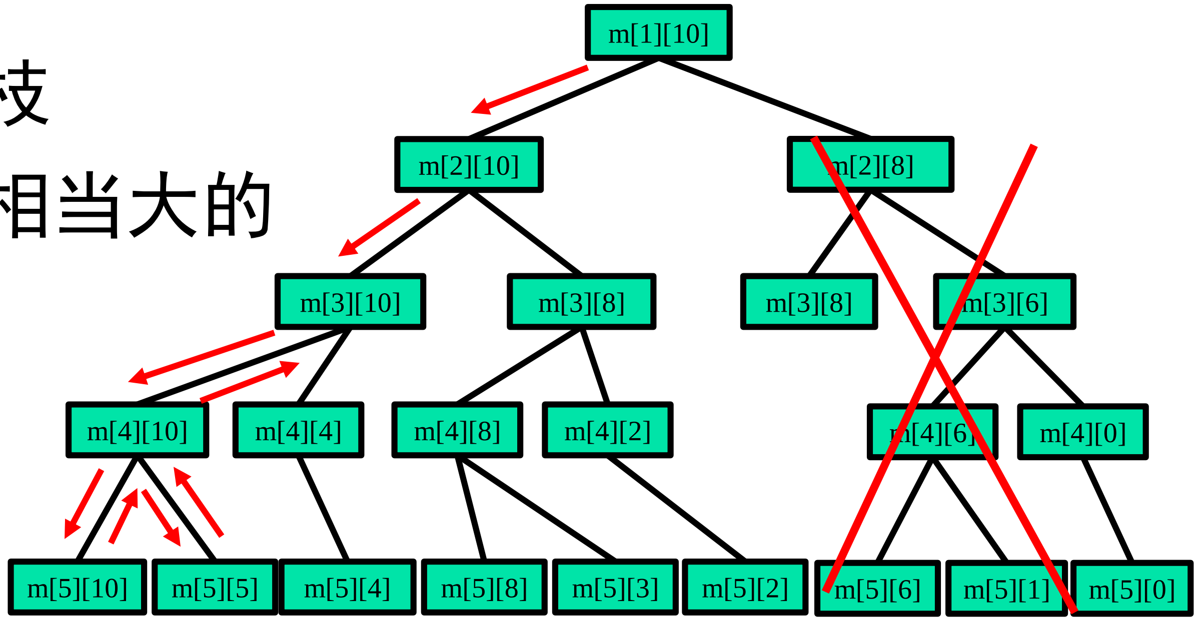
- 动态规划法的剪枝策略
  - 考虑的搜索空间中子问题的重叠性
- 贪心算法的剪枝策略
  - 以贪心选择策略为依据，遍历搜索空间的部分分支



0-1背包问题的搜索空间

什么是回溯法

- 一种“通用的解法”
  - 将问题建模为解空间树
  - 深度优先搜索
  - 搜索过程中剪枝
  - 适合解组合数相当大的问题



回溯法的两个核心问题

1 如何构建解空间树?

2 如何设计剪枝函数?

回溯问题的解空间

对n个物品的0-1背包问题

- 可能解由一个不等长向量组成
  - 解向量的长度等于装入背包的物品个数
  - 如n=3，解空间{( ),(1),(2),(3),(1,2),(1,3),(2,3),(1,2,3)}

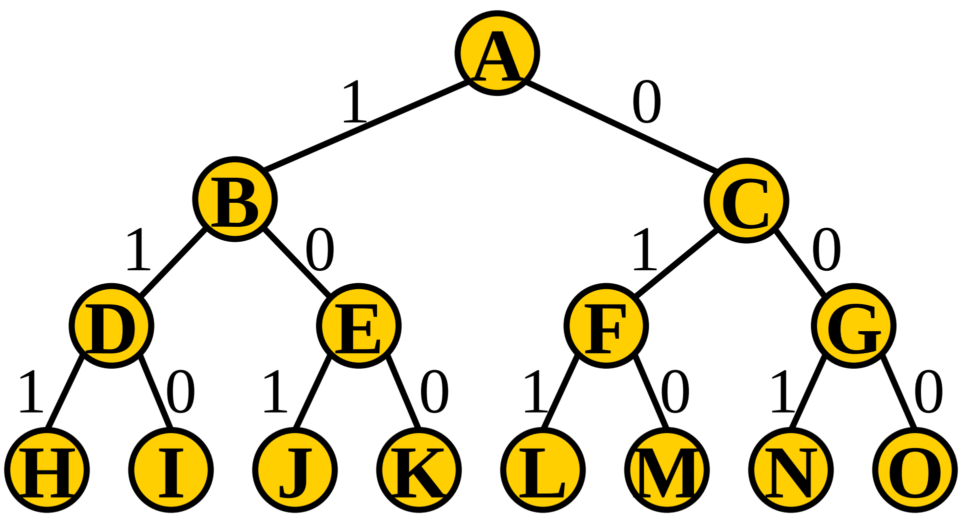
- 可能解由一个等长向量$\{x_1,\ldots, x_n\}$组成
  - x_i表示是否放入物品i
  - 如n=3，解空间为{(0,0,0), (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), (0,1,1), (1,1,1)}

回溯问题的解空间

- 问题的解向量：
  - n元式$(x_1,x_2,\ldots,x_n)$的形式。
  - 显约束：对分量x_i的取值限定。
  - 隐约束：为满足问题的解而对不同分量之间施加的约束。
  - 解空间：满足显式约束条件的所有多元组

回溯问题的解空间树

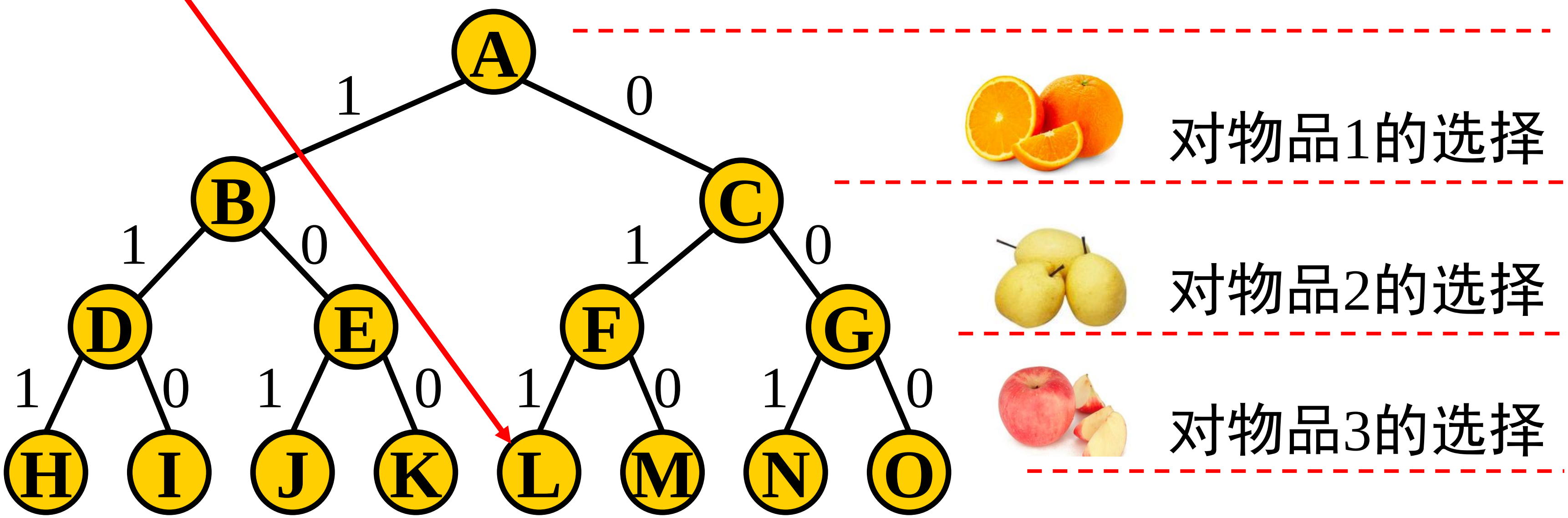
- 解空间树
  - 问题的解空间的表示方式
  - 第0层为初始状态
  - 第k层为第k个分量做出选择后到达的状态
  - 从树的根节点到叶子节点的路径



n=3时的0-1背包问题的解空间树

回溯问题的解空间树(0-1背包问题)

表示解（0，1，1）即选物品2和物品3，不选物品1

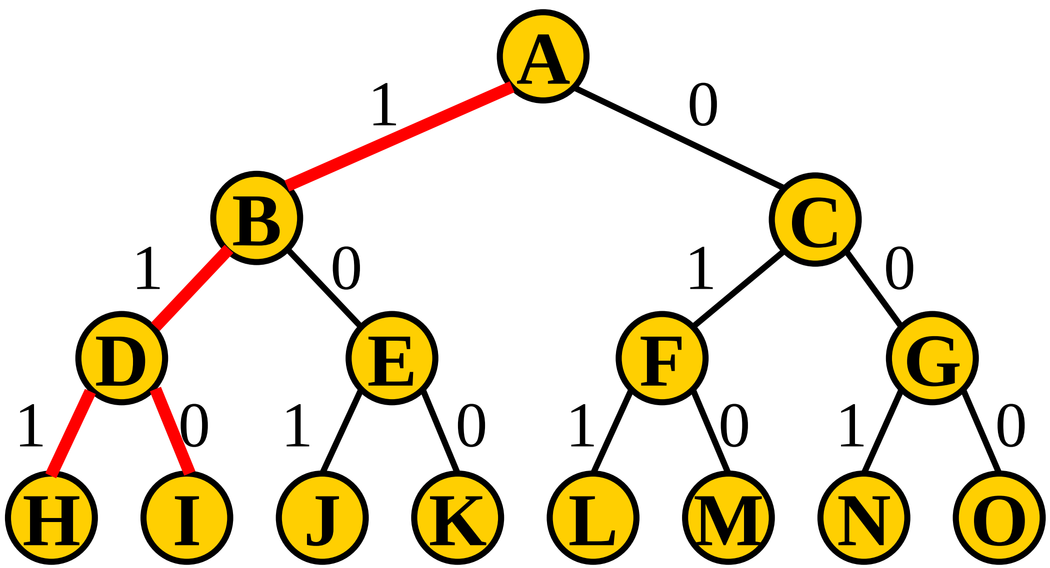


n=3时的0-1背包问题的解空间树

树中第i层与第i+1层节点之间的边上给出了对物品i的选择结果，8个叶子代表8个可能解

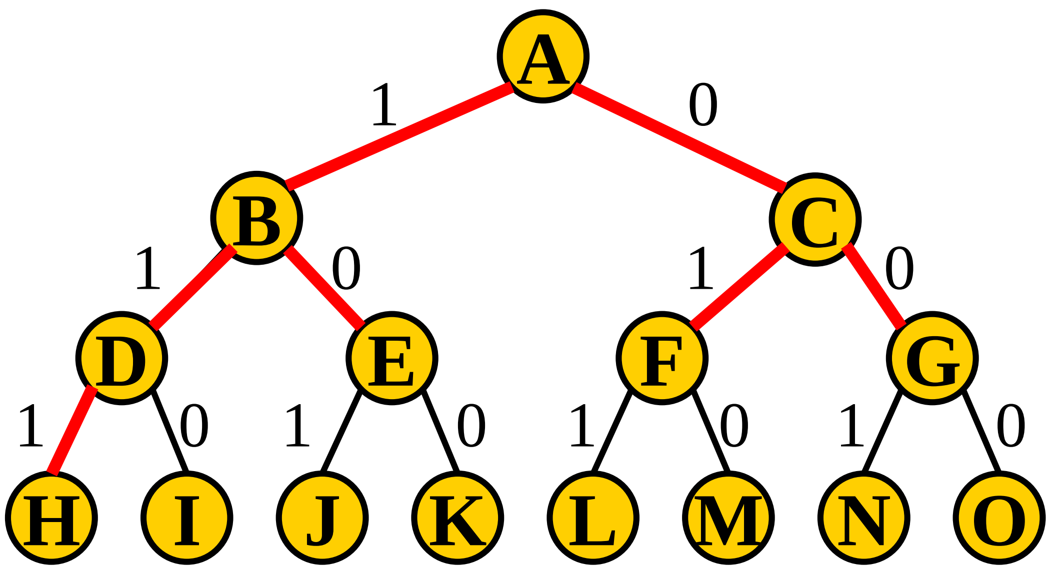
解空间树的生成方法—深度优先

- 基于深度优先搜索
  - 英文缩写为DFS即Depth First Search
  - 对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次



解空间树的生成方法—广度优先

- 基于广度优先搜索（第六章分支限界算法）
  - 英文缩写为BFS即Breadth First Search
  - 从根开始，辐射状地优先遍历其周围较广的区域，而且每个节点只能访问一次



0-1背包问题的实例

问题：有n种物品，每种物品的重量和价值分别为w_i, v_i。如果背包的最大承重限制是B，每种物品至多放1个。怎么样选择放入背包的物品使得背包所装物品价值最大？

实例：V={12,11,9,8}, W={8,6,4,3}, B=13

最优解：$\langle 0,1,1,1 \rangle$，价值：28，重量：13

算法设计

- 解：n维0-1向量$\langle x_1, x_2, \ldots, x_n\rangle$

  $x_i = 1 \Leftrightarrow$物品i选入背包

- 搜索空间：一棵0-1取值的二叉树，有2^n片树叶

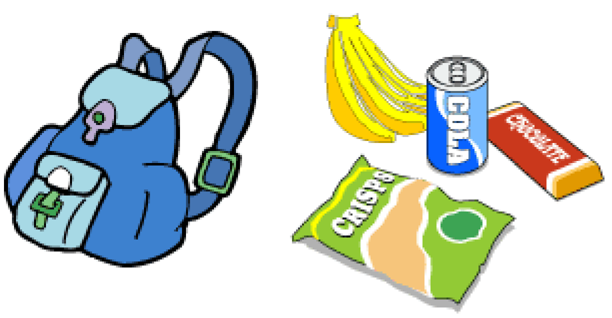
- 结点：$\langle x_1, x_2, \ldots, x_k\rangle$（部分向量）

- 可行解：满足约束条件(不超重)的解

- 最优解：可行解中价值达到最大的解

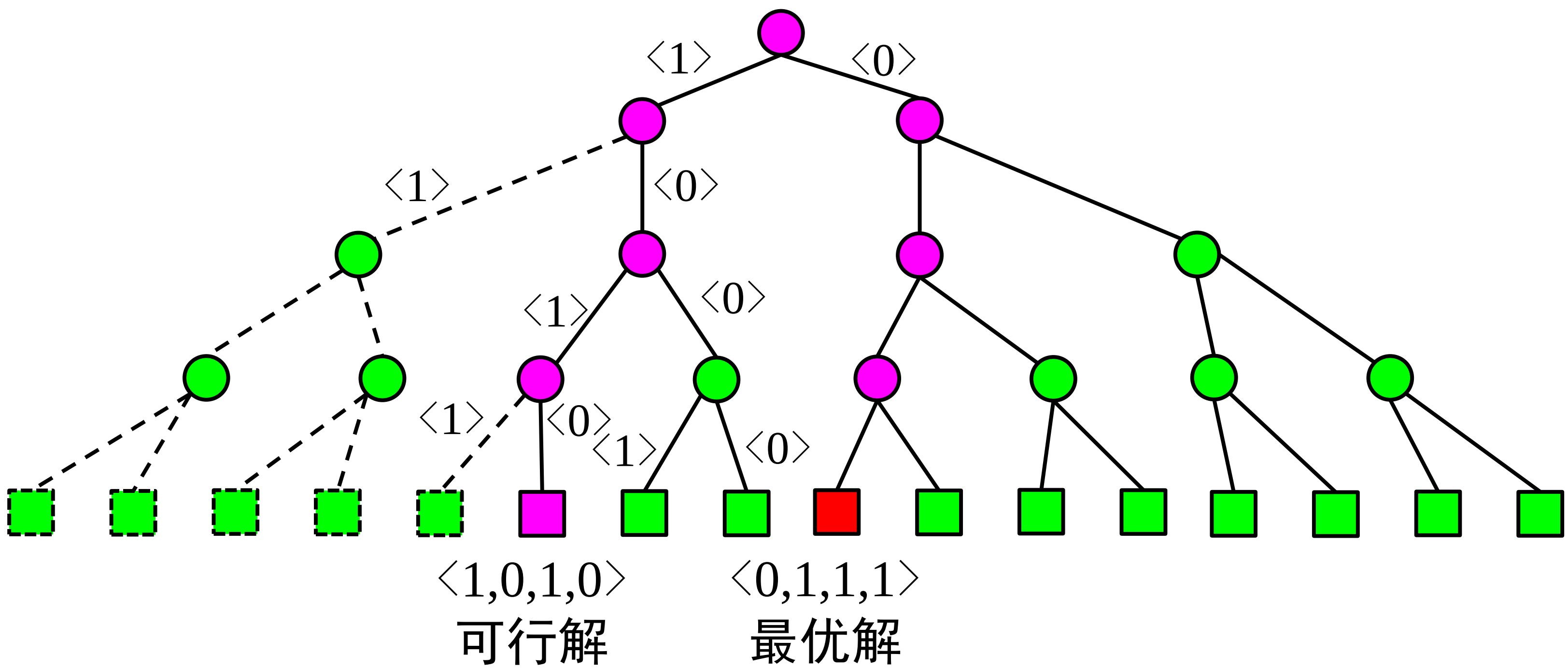
实例

- 输入：$V=\{12,11,9,8\}$, $W=\{8,6,4,3\}$, $B=13$
- 2个可行解：
  - $\langle 0,1,1,1\rangle$，价值：28，重量：13
  - $\langle 1,0,1,0\rangle$，价值：21，重量：12
- 最优解：$\langle 0,1,1,1\rangle$



搜索空间

- 实例：$V=\{12,11,9,8\}$, $W=\{8,6,4,3\}$, $B=13$
- 搜索空间：2^n片树叶

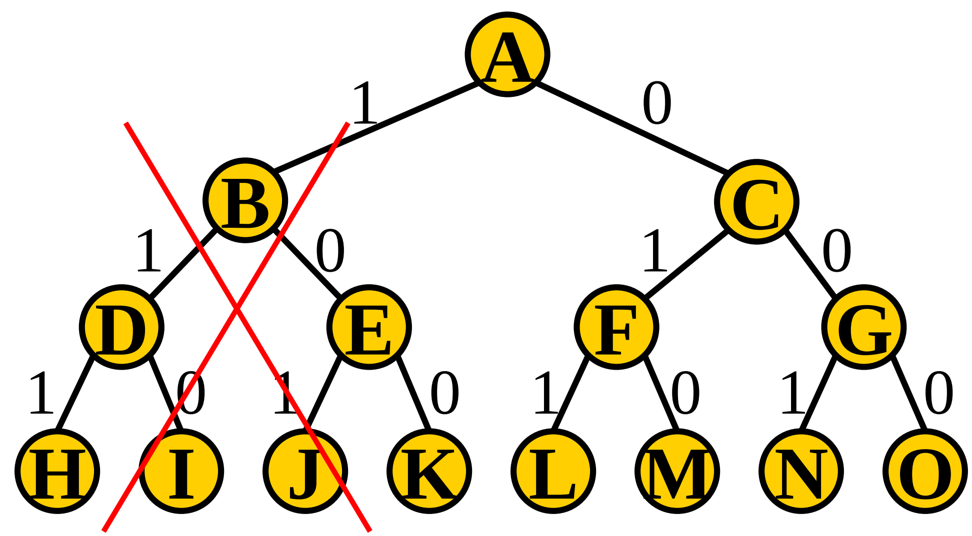


回溯法的基本思想

- 问题的求解方式
  - 定义整个解空间完成
  - 确定易于搜索的解空间结构
  - 深度优先方式遍历解空间并剪枝

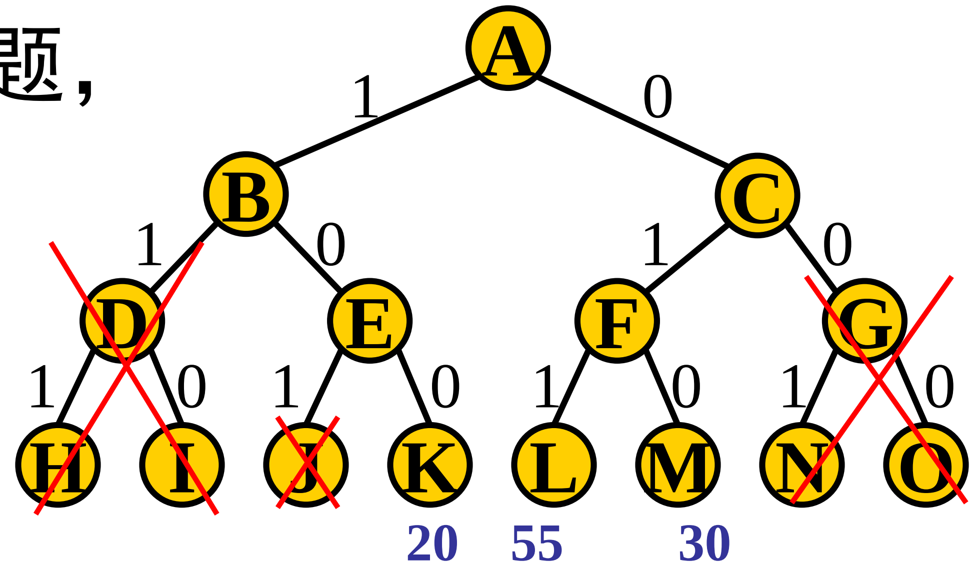
应包括所有的可能解

回溯法是具有剪枝函数的深度优先生成法



回溯法的例子

- 例：n=3的0-1背包问题，
  - 重量{20, 15, 10}
  - 价值{20, 30, 25}
  - 背包容量为25



- 深度优先遍历
  - A→B选物品1，则容量为5，价值为20；
  - B→D，因为选物品2放不下，对以D为根的子树剪枝；
  - 从D回溯到B，选右子树E，不选物2，价仍然为20；
  - E→J，选物3放不下，所以以J为根的**子树剪枝**；
  - 从J回溯到E，再由E→K，K不需容量，构成一个可行解(1,0,0)，价为20。

剪枝的基本思想

- 在搜索至树上任意一点时判断
  - 是否满足约束条件
  - 是否包含问题的（最优）解。

不包含

跳过对以该节点为根的子树的搜索，剪枝(pruning)

包含

进入以该节点为根的子树，继续按深度优先搜索。

剪枝的基本思想

- 在搜索至树上任意一点时判断
  - 是否满足约束条件
  - 是否包含问题的（最优）解。
- 两种用于剪枝的函数
  - 约束函数：用约束条件剪去得不到可行解的子树
  - 限界函数：用目标函数剪去得不到最优解的子树

利用剪枝函数可避免无效搜索，
使算法无需搜索整个搜索树。

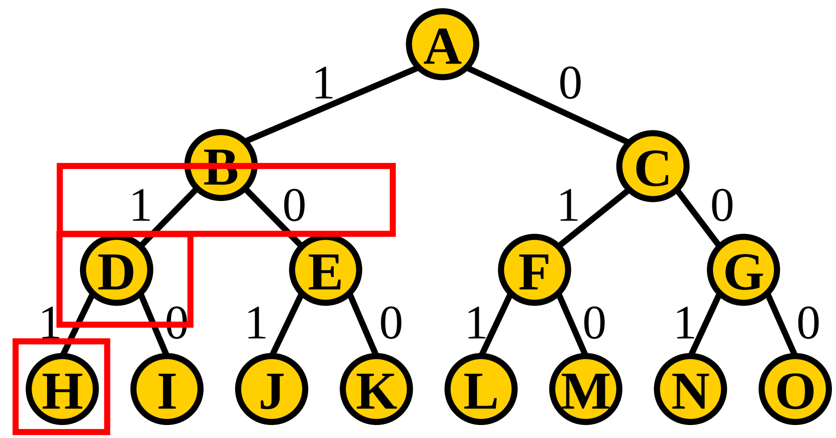
回溯法算法框架——递归回溯

- 递归形式

```
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(x,t)&&bound(x,t)) 
            backtrack(t+1);
      }
}
```

到达叶子节点，输出结果

剪枝函数

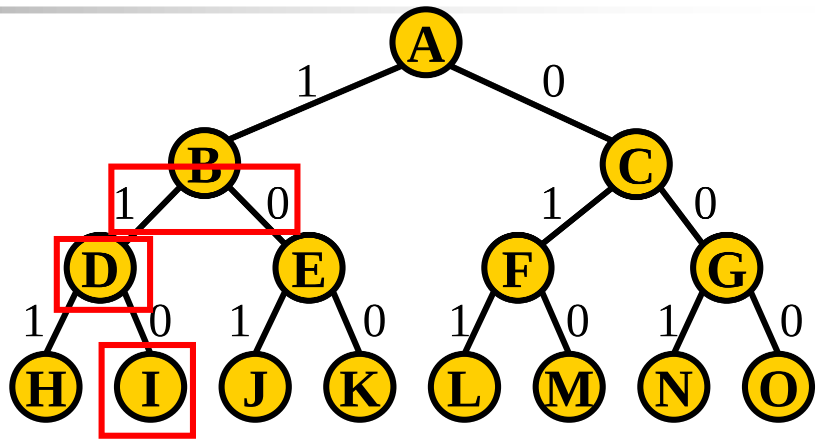


$f(n, t)$：第t层未搜索过子树的起始编号

$g(n, t)$：第t层未搜索过子树的终止编号

迭代回溯

- 非递归的迭代形式。




```
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(x,t)&&bound(x,t)) {
                    if (solution(t)) output(x);
                    else {t++;break;}
                }}
        else t--;
    }
}
```

剪枝函数

到达叶子节点，输出结果

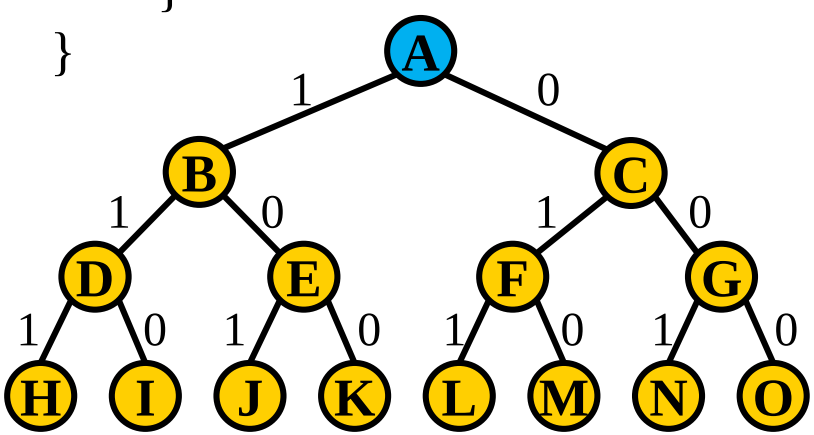
$f(n, t)$：第t层未搜索过子树的起始编号

$g(n, t)$：第t层未搜索过子树的终止编号

回溯法算法框架-Step0

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

t	操作/x的值



正被访问结点

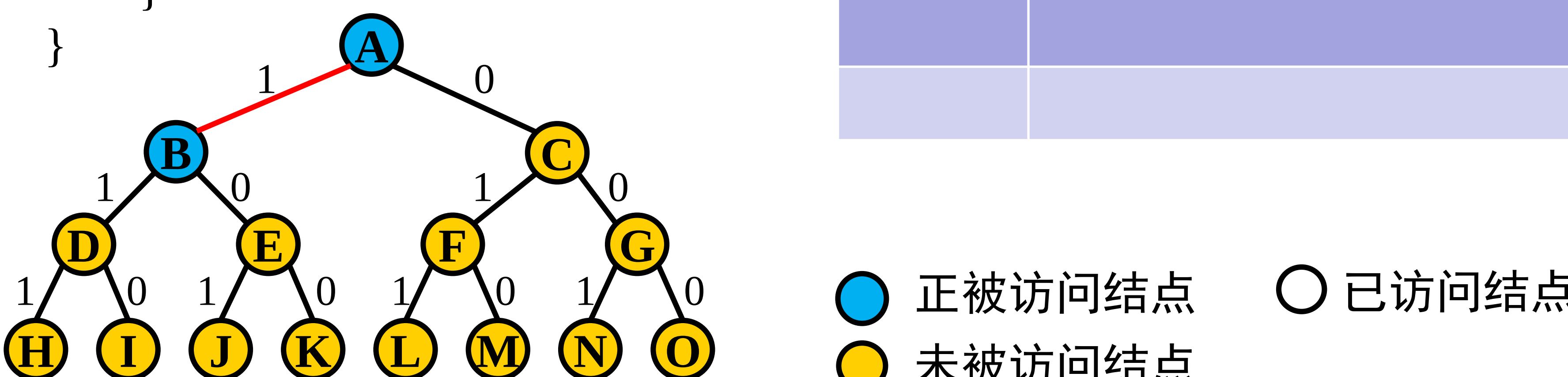
已访问结点

未被访问结点

回溯法算法框架-Step1

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

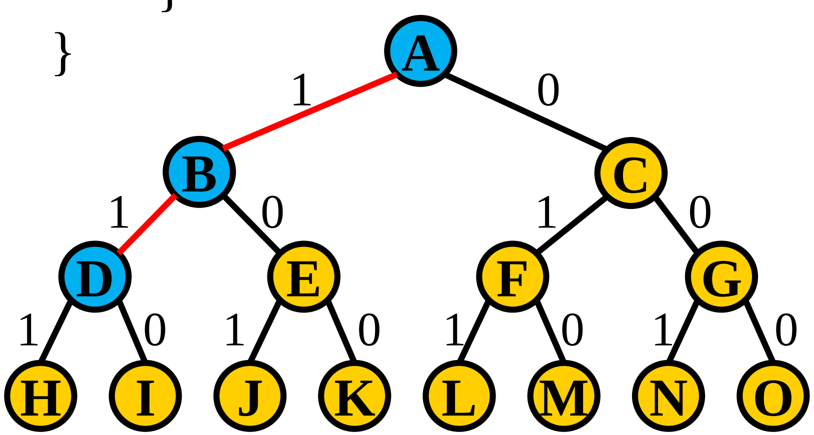
t	操作/x的值
0	<1>



回溯法算法框架-Step2

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

t	操作/x的值
0	<1>
1	<1, 1>



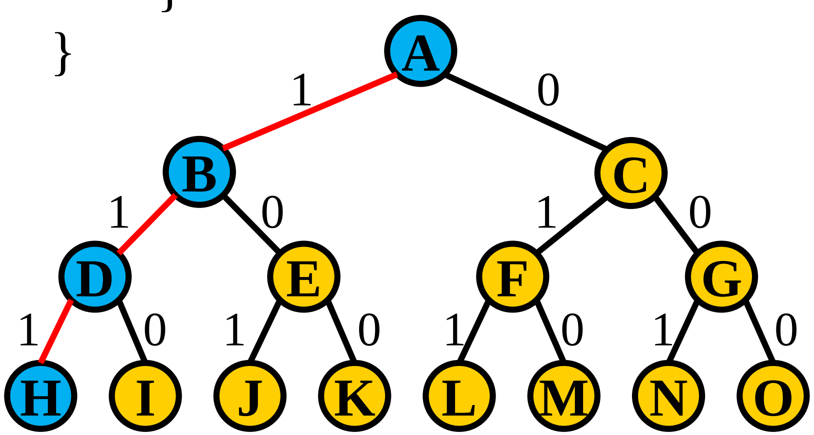
正被访问结点

已访问结点

未被访问结点

回溯法算法框架-Step3

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



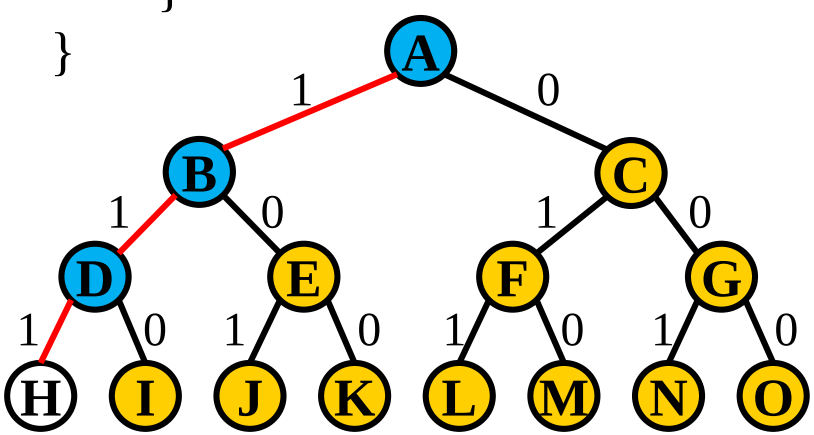
t	操作/x的值
0	<1>
1	<1, 1>
2	<1, 1, 1>

正被访问结点　　已访问结点

未被访问结点

回溯法算法框架-Step4

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



t	操作/x的值
0	<1>
1	<1, 1>
2	<1, 1, 1>
3	输出<1, 1, 1>，更新界

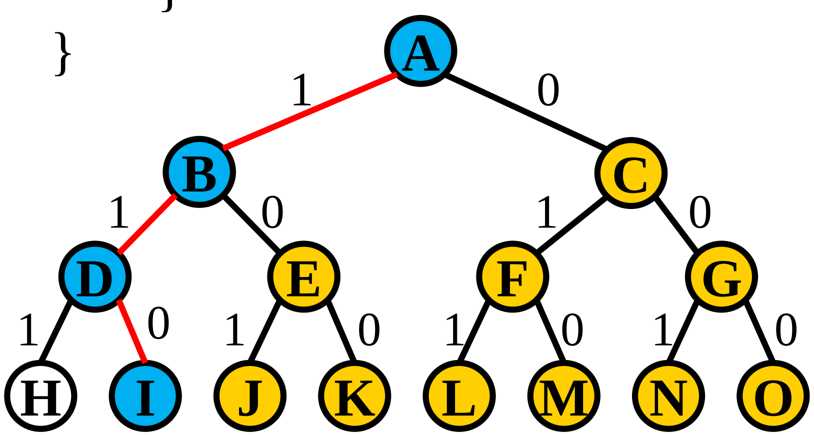
正被访问结点　　已访问结点

未被访问结点

回溯法算法框架-Step5

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

t	操作/x的值
0	<1>
1	<1, 1>
2	<1, 1, 1>
3	输出<1, 1, 1>，更新界
2	<1, 1, 0>



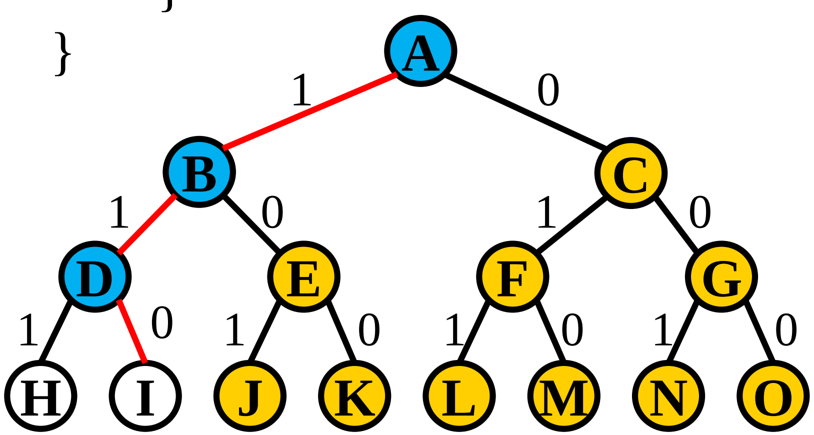
正被访问结点　　已访问结点

未被访问结点

回溯法算法框架-Step6

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

t	操作/x的值
0	<1>
1	<1, 1>
2	<1, 1, 1>
3	输出<1, 1, 1>，更新界
2	<1, 1, 0>
3	输出 <1, 1, 0>，更新界

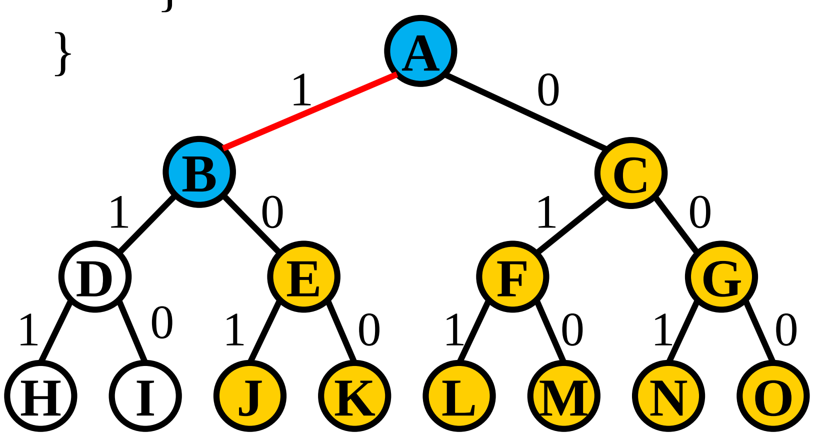


正被访问结点
已访问结点
未被访问结点

回溯法算法框架-Step7

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

t	操作/x的值
0	<1>
1	<1, 1>
2	<1, 1, 1>
3	输出<1, 1, 1>，更新界
2	<1, 1, 0>
3	输出 <1, 1, 0>，更新界
2	返回上一层



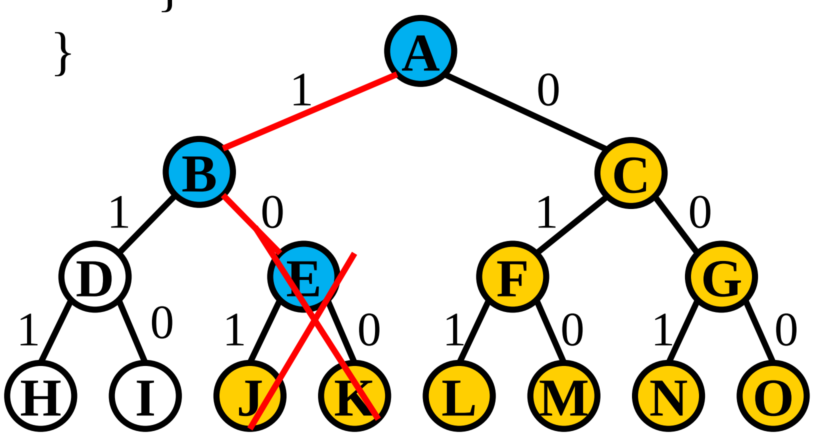
正被访问结点　已访问结点

未被访问结点

回溯法算法框架-Step8

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

t	操作/x的值
0	<1>
1	<1, 1>
2	<1, 1, 1>
3	输出<1, 1, 1>，更新界
2	<1, 1, 0>
3	输出 <1, 1, 0>，更新界
2	返回上一层
1	<1, 0>（剪枝）



正被访问结点 已访问结点

未被访问结点

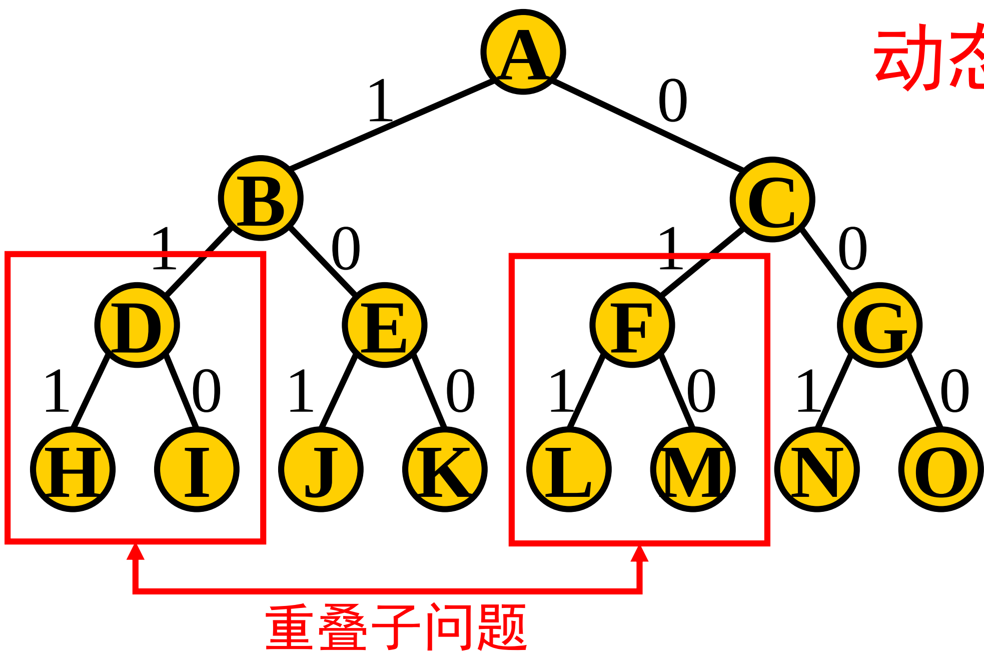
回溯法的空间复杂度

- 回溯法的存储特点
  - 动态产生问题的解空间
  - 只保存从根结点到当前扩展结点的路径。

- 空间复杂度
  - 根到叶子的最长路径的长度为$h(\mathrm{n})$
  - 空间复杂性通常为$O(h(\mathrm{n}))$。
  - 显式地存储整个解空间则需要$O(2^{h(n)})$或$O(h(n)!)$

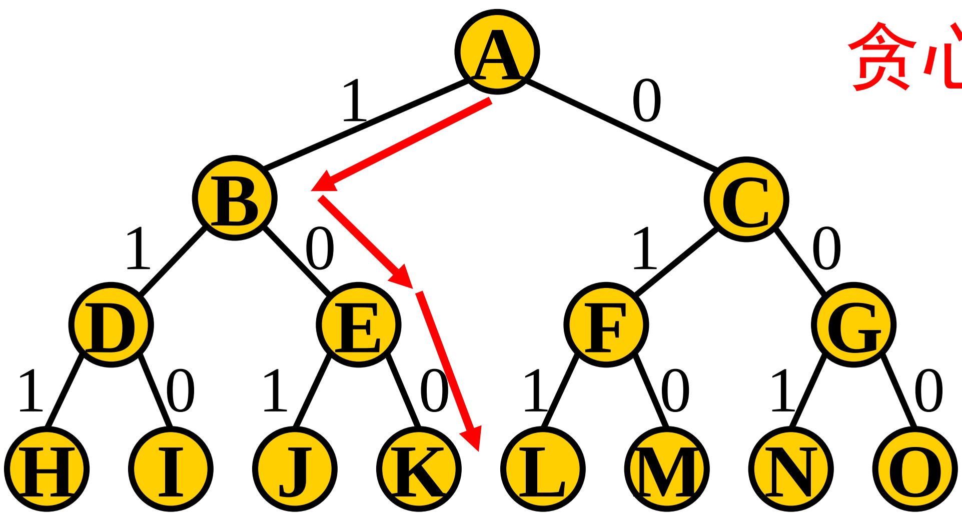
回溯法与其他算法比较

- 保证算法高效性的机制
  - 动态规划：避免计算重叠子问题
  - 贪心算法：只考虑局部最优解
  - 回溯法：利用剪枝函数



回溯法与其他算法比较

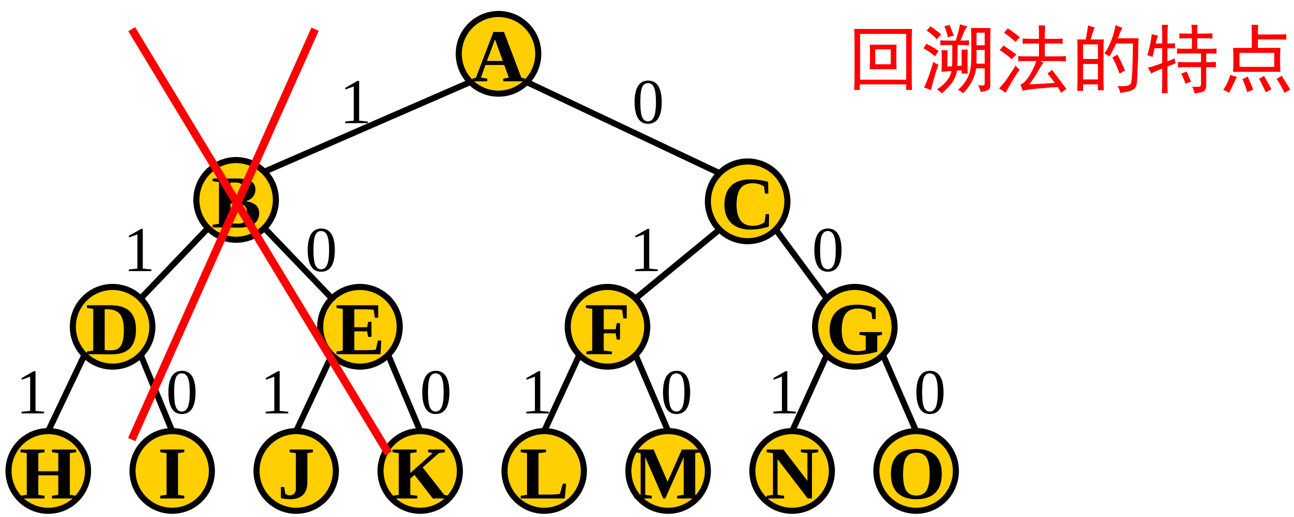
- 保证算法高效性的机制
  - 动态规划：避免计算重叠子问题
  - 贪心算法：只考虑局部最优解
  - 回溯法：利用剪枝函数



贪心算法的特点

回溯法与其他算法比较

- 保证算法高效性的机制
  - 动态规划：避免计算重叠子问题
  - 贪心算法：只考虑局部最优解
  - 回溯法：利用剪枝函数



回溯法的两个核心问题

1 如何构建解空间树?

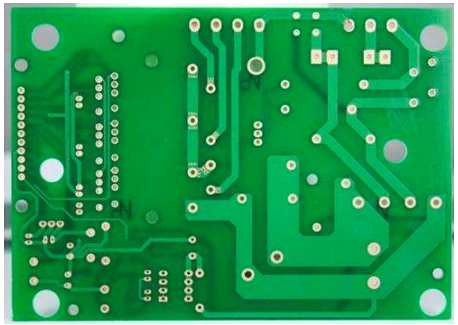
2 如何设计剪枝函数?

排列树树：旅行商问题

回顾：旅行商问题

- 旅行商问题（Travelling Salesman Problem,TSP）：旅行家旅行*n*个城市，要各城市经历且经历一次，然后回到源点，求出最短路程。

规划快递线路



电路板钻洞

DNA测序

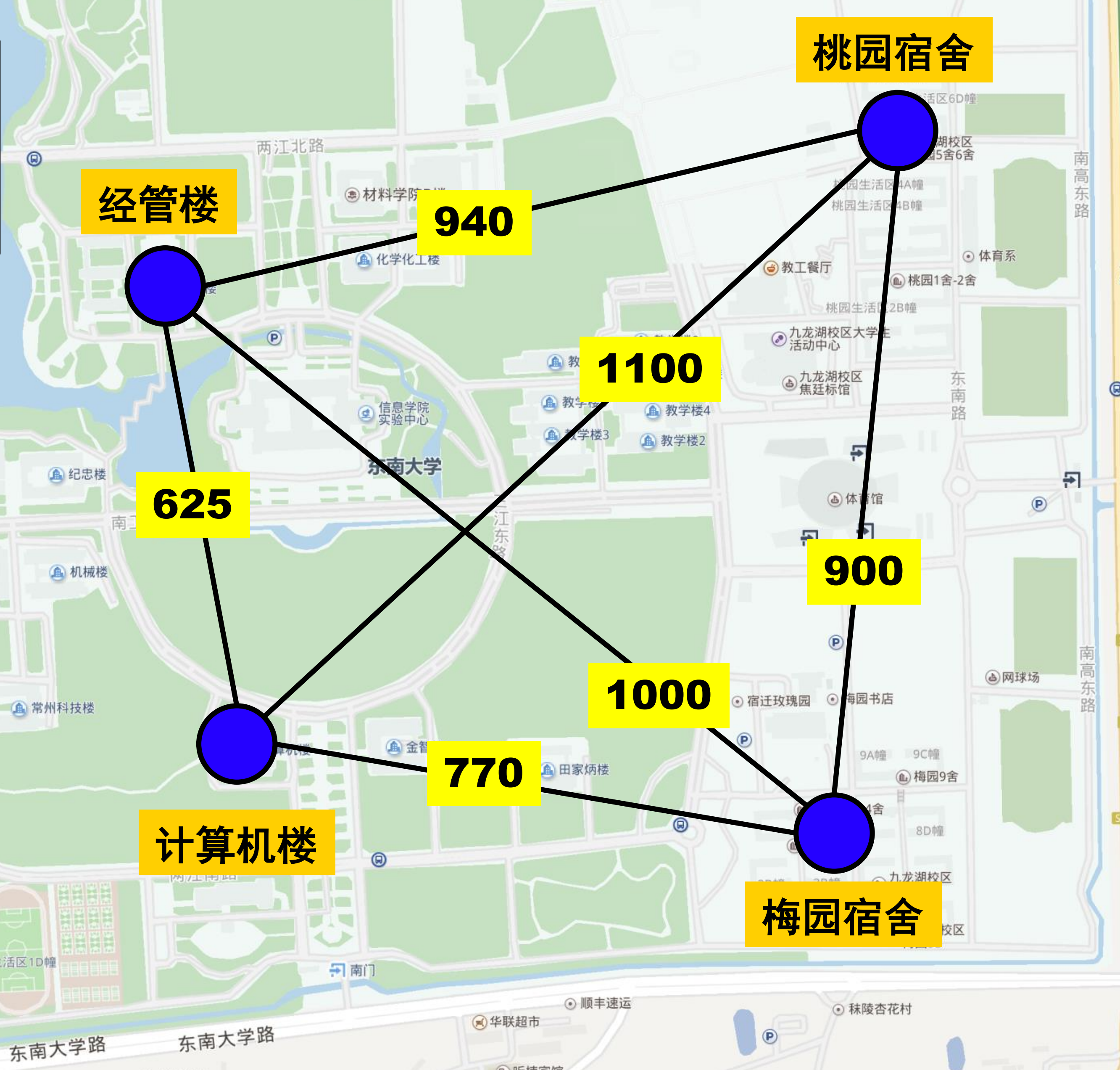
旅行商问题的描述

- 问题：一个旅行商需要在n个城市销售商品，已知任两个城市之间的距离，求一条每个城市恰好经过一次的回路，使得总长度最小。
- 建模：城市集$C=\{c_1,c_2,\ldots,c_n\}$,距离$d(c_i,c_j)=d(c_j,c_i)$
- 求解：1,2,...,n的排列$k_1,k_2,\ldots,k_n$使得

$$\min\left\{\sum_{i=1}^{n-1} d(c_{k_i},c_{k_{i+1}})+d(c_{k_n},c_{k_1})\right\}$$

旅行商问题的应用：快递小哥送货

从计算机楼出发，
如何规划线路，
使得总路程最少？



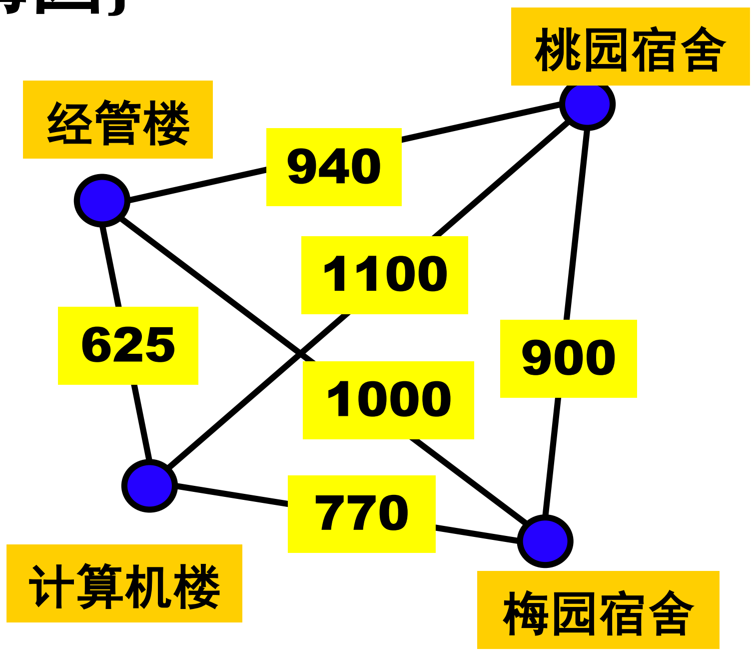
实例

输入：

- C={计算机楼, 经管楼, 桃园, 梅园}（编号依次为 1, 2, 3, 4）

$$
\begin{array}{c} \\ 计 \\ 经 \\ 桃 \\ 梅 \end{array}
\begin{array}{c}
\begin{array}{cccc} 计 & 经 & 桃 & 梅 \end{array} \\
\begin{bmatrix}
\infty & 625 & 1100 & 770 \\
625 & \infty & 940 & 1000 \\
1100 & 940 & \infty & 900 \\
770 & 1000 & 900 & \infty
\end{bmatrix}
\end{array}
$$

距离矩阵

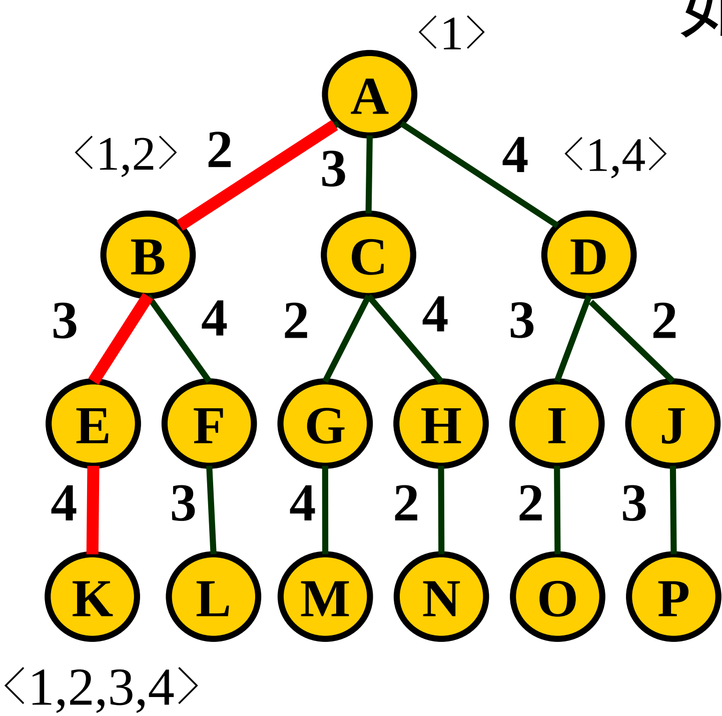


最优解：⟨1, 2, 3, 4⟩,

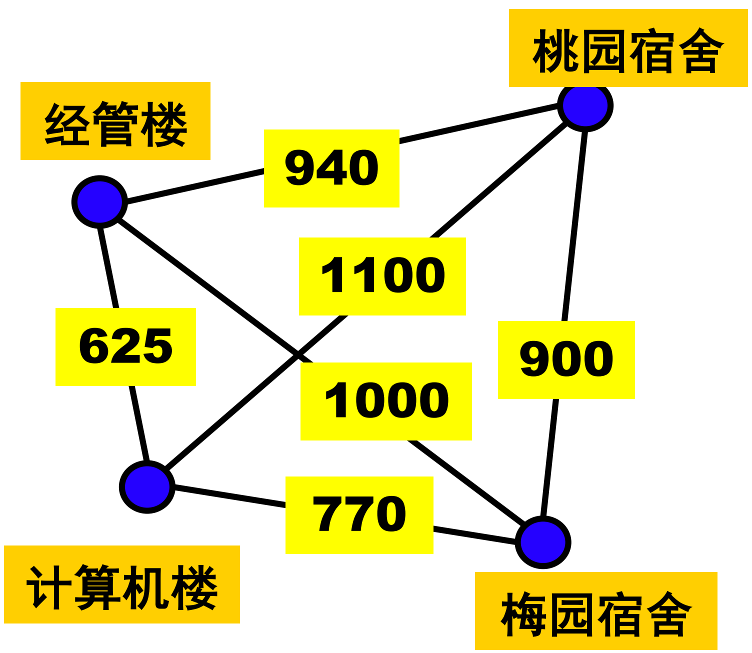
长度=625+940+900+770=3235

搜索空间

解空间树

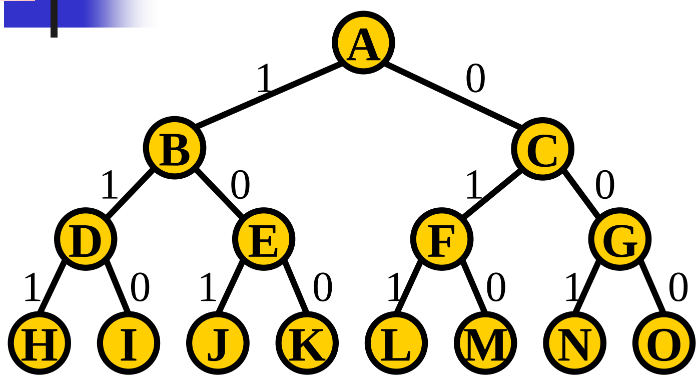


思考：
如果有5个地点，解空间有多大？



旅行商问题的解空间树有$(n-1)!$片树叶

子集树与排列树



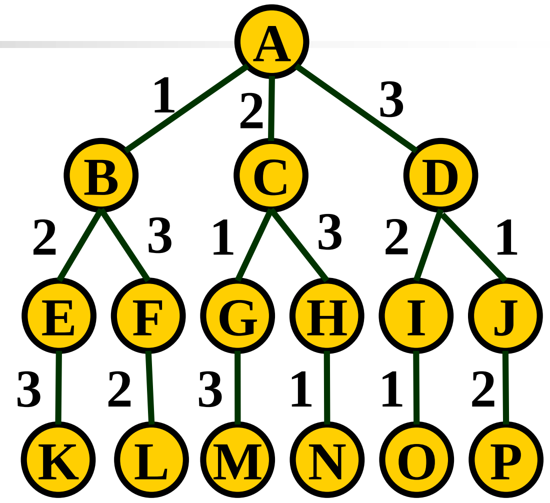
- 子树集(subset trees)：
  - 从n个元素中找出满足某种性质子集，相应解空间为子集树。
  - 如0-1背包问题

```
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);
      }
}
```

- 时间复杂度
  - 通常各节点有相同数目子树，记为C
  - C=2时，子集树中共有2^n个叶子，因此需要$O(2^n)$时间。

子集树与排列树

- 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
  - 如旅行商问题




```
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]);
      }
}
```

- 时间复杂度
  - 第1层每个节点有n个子节点
  - 第2层每个节点有n-1个子节点
  - 第n层每个节点有1个子点
  - 有n!个叶子节点，需时间$O(n!)$

回溯法的时间复杂度

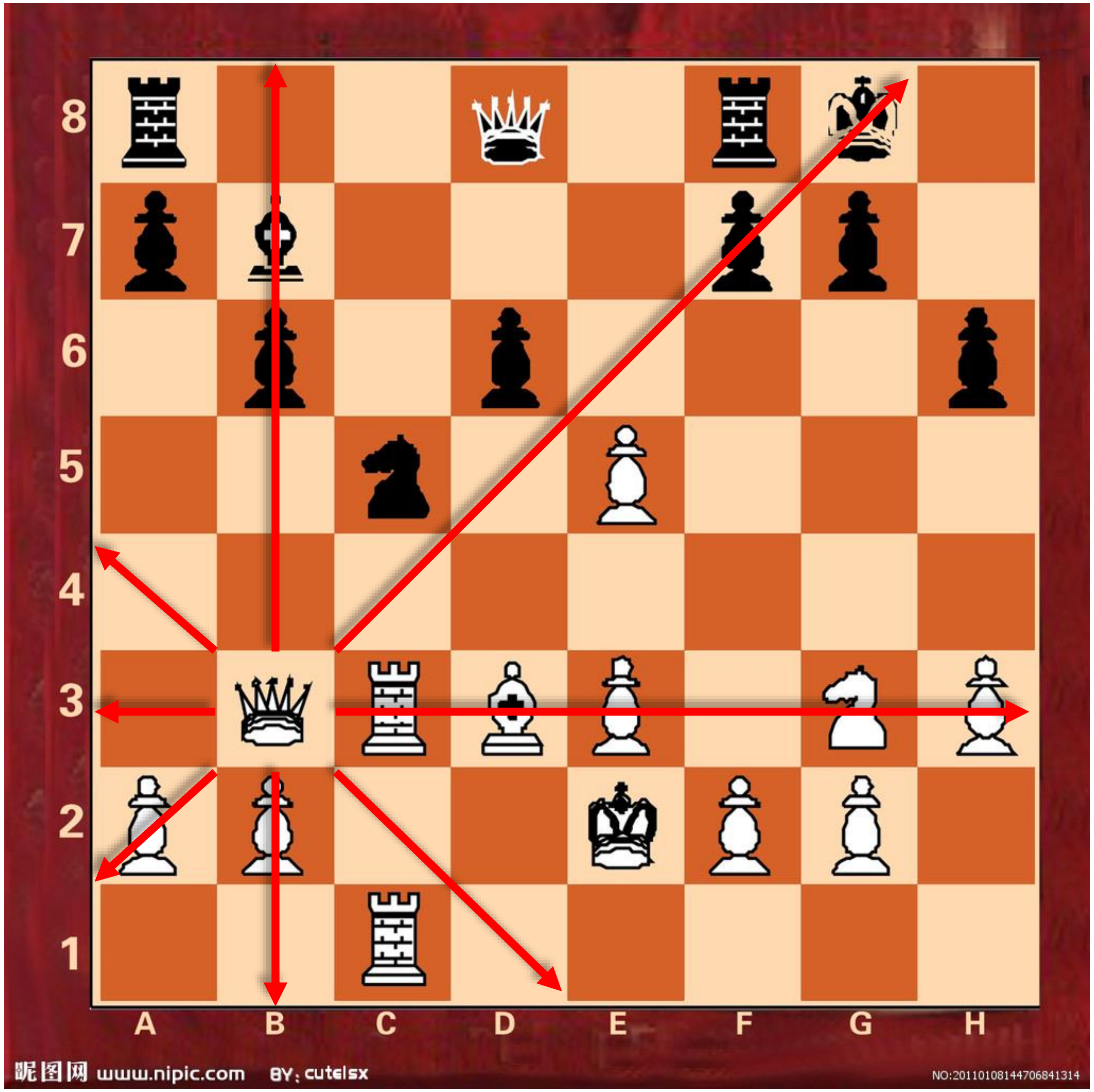
- 时间复杂度
  - 子集树： $O(2^n)$
  - 排列树： $O(n!)$

- 蛮力穷举法，最坏时间复杂性不可指望。
  - 通用性强
  - 平均时间性能较好
  - 需设计较好的剪枝函数

n叉树：n皇后问题

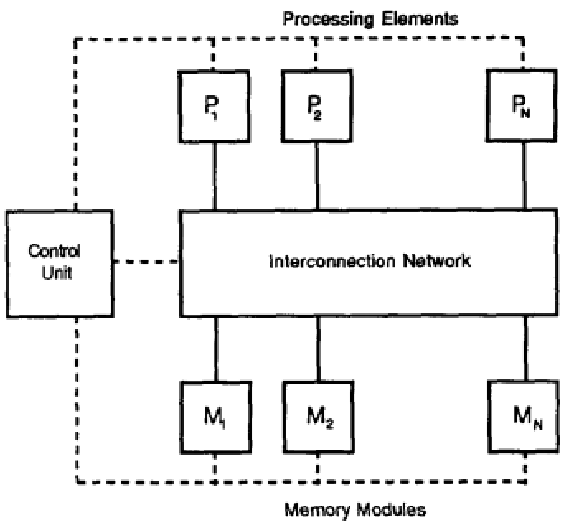
国际象棋

- **皇后的走法**
- 又称“皇后”。走法是横、直、斜走均可，格数不限，但不可越过其他棋子。吃子和走法相同。



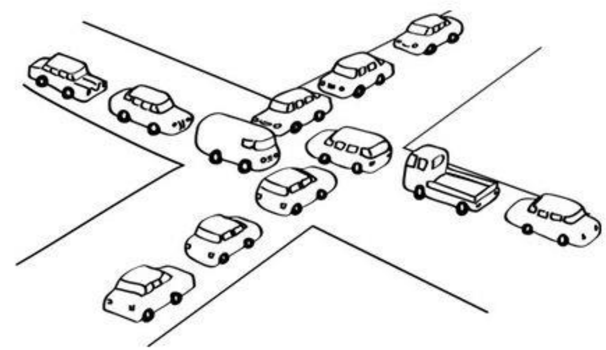
n皇后问题及其应用

- 在一个n×n的方格内放置n个皇后，使得没有两个皇后在同一行、同一列、也不在同一条45度的斜线上。问有多少种可能的布局?



并行内存系统的存储模式

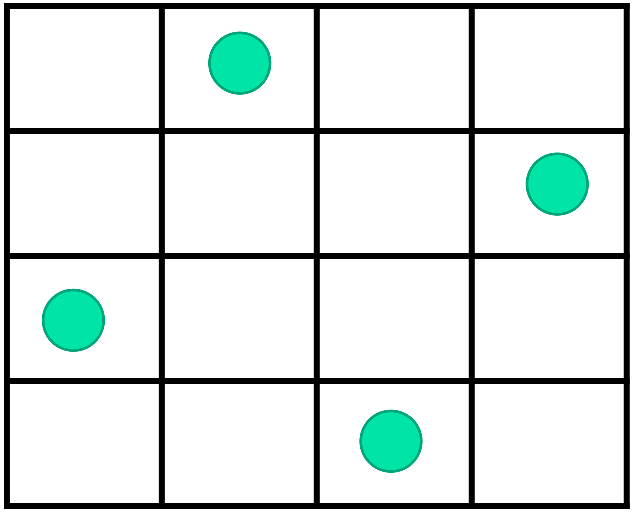
超大规模集成电路设计



检测程序中的死锁问题

n皇后问题的解空间

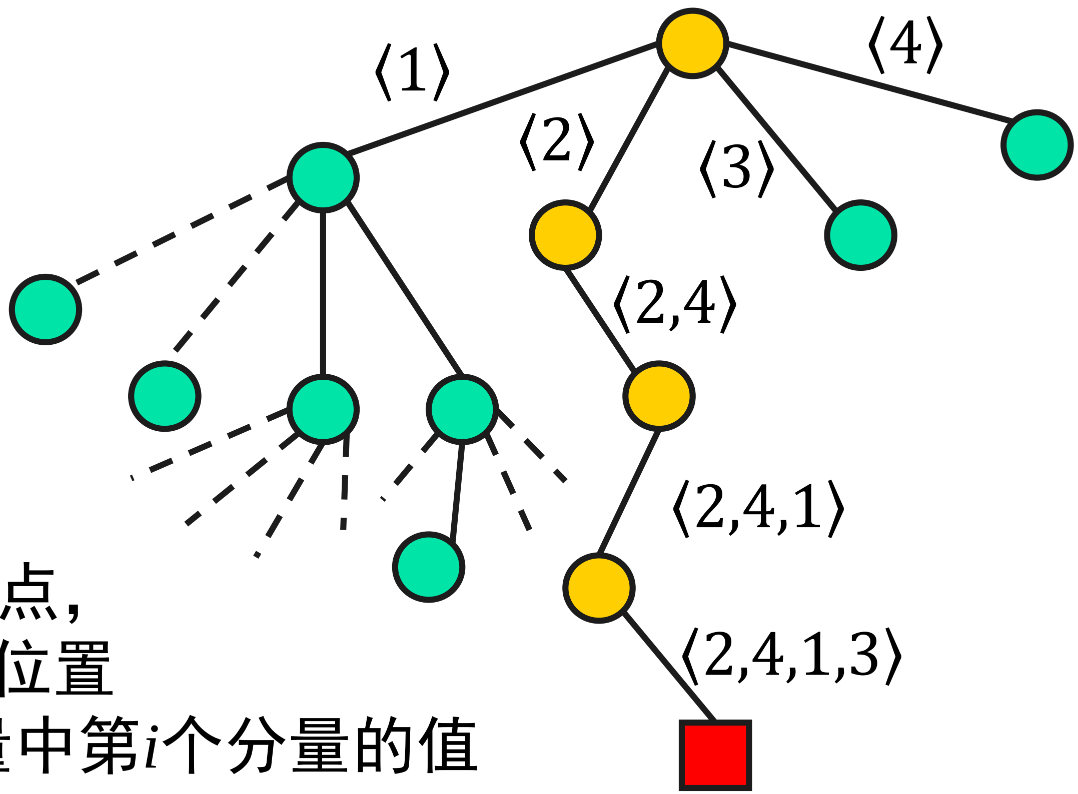
- 当n=4时
  - 解是4维向量$\langle x_1, x_2, x_3, x_4 \rangle$
  - 解：$\langle 2,4,1,3 \rangle$，$\langle 3,1,4,2 \rangle$



- 当n=8时
  - 解是8维向量，有92个解
  - 例如：$\langle 1,5,8,6,3,7,2,4 \rangle$是解

n皇后问题的解空间树

- 一棵n叉树（假设n=4）



树的特点：
- 每个节点有四个子节点，
  表示选择1,2,3,4四个位置
- 第i层选择表示解向量中第i个分量的值
- 最深层的叶子是解
- 按深度优先次序遍历树，找到所有解

n皇后问题的算法实现

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

void Queen::Backtrack(int t)
{
    if (t>n) sum++;
    else
        for (int i=1;i<=n;i++) {
            x[t]=i;
            if (Place(t)) Backtrack(t+1);
        }
}
```

小结

问题	解性质	解描述向量	搜索空间	搜索方式	约束条件
n皇后	可行解	$\langle x_1, x_2, \cdots, x_n \rangle$ x_i : 第 i 行列号	n叉树	深度优先搜索	彼此不攻击
0-1背包	最优解	$\langle x_1, x_2, \cdots, x_n \rangle$ $x_i \in \{0,1\}$	子集树	深度优先搜索	不超过 总重量
旅行商	最优解	$\langle k_1, k_2, \cdots, k_n \rangle$ $1,2,\cdots,n$的排列	排列树	深度优先搜索	选没有经过 的城市

特点	搜索解	向量，不断扩张部分向量	树	跳跃式遍历	约束条件，回溯判定

回溯法的几个应用

装载问题

贪心法的最优装载问题（回忆）

- 有一批集装箱$\{a_1,a_2,\ldots,a_n\}$要装上一艘载重量为C的轮船，其中集装箱a_i的重量为W_i。

- 最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

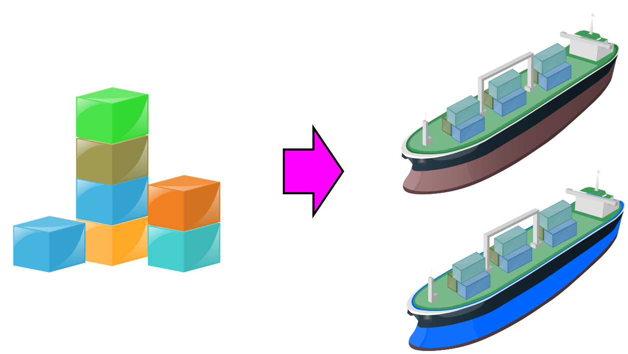
贪心策略：轻者优先

装载问题及其应用

- 有一批共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

$$w_1 + w_2 + \cdots + w_n \le c_1 + c_2$$

- 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。



例子：出国旅行的装载问题

装载问题的解空间

- 实例：
  - 物品的重量$W = \langle \underline{90},65,\underline{40},30,20,\underline{12},\underline{10} \rangle$
  - 旅行箱允许载重 $c_1 = 152,\ c_2 = 130$

- 问题的解
  - 将物品1,3,6,7装第一个箱子
  - $c_1 + c_3 + c_6 + c_7 = 152$
  - 解的表示$\langle 1,0,1,0,0,1,0 \rangle$

装载问题的求解思路

- 输入：物品重量W，旅行箱载重c_1, c_2
  - 首先将第一个旅行箱尽可能装满；
  - 将剩余的物品装上第二个旅行箱。
  - 将第一个旅行箱尽可能装满等价于选取全体物品的一个子集，使该子集中物品重量之和最接近。

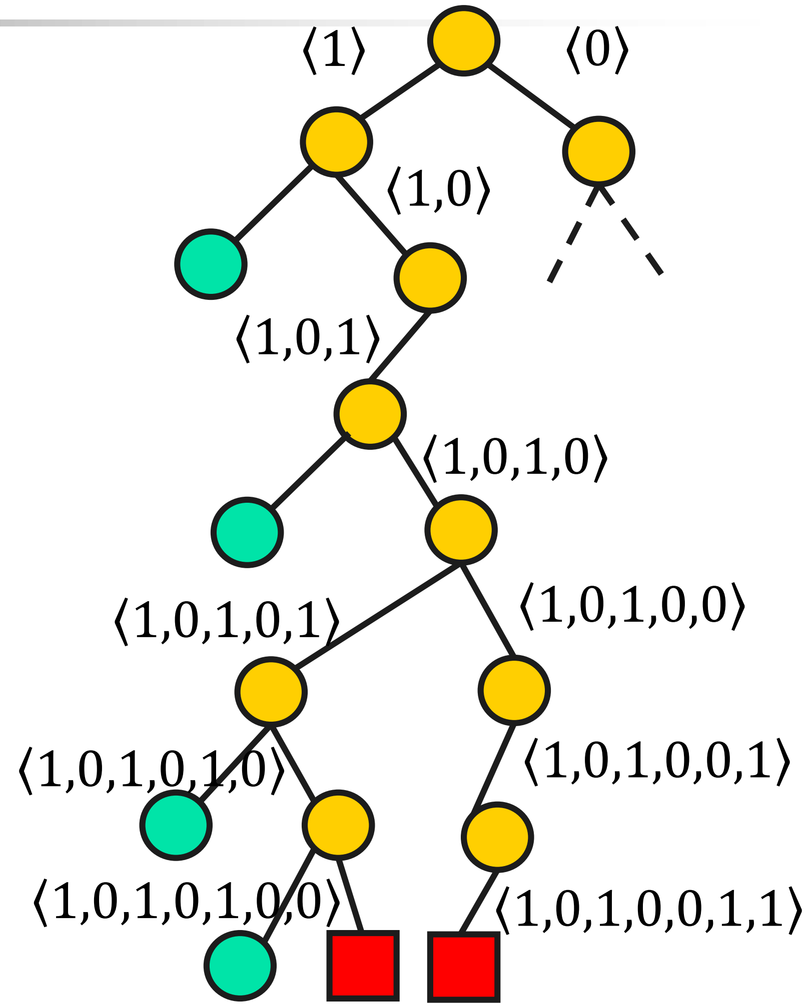
$$\max \sum_{i=1}^{n} w_i x_i$$

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

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

装载问题的解空间树

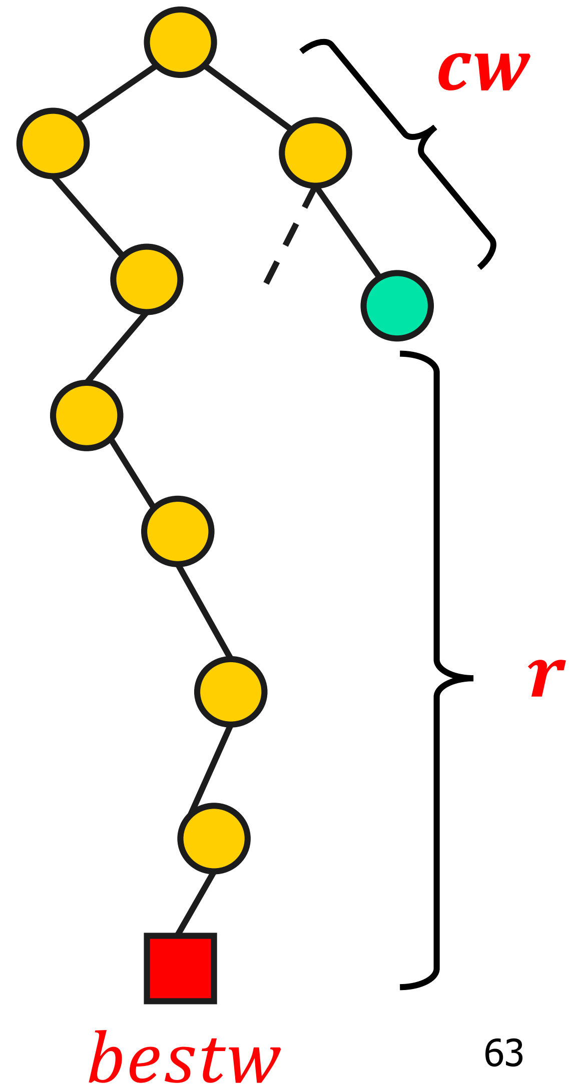
- 一棵二叉树（子集树）
- 实例
  - 物品重量$W = \langle 90,65,40,30,20,12,10 \rangle$
  - $c_1 = 152,\ c_2 = 130$
  - 最优解：$\langle 1,0,1,0,0,1,1 \rangle$



装载问题的剪枝函数

- 可行性约束函数
- 限界函数
  - 有用的变量
    - 当前旅行箱内重量：cw
    - 当前最优解：$bestw$
  - 上界函数：剩余物品的重量

    $$r = w_{i+1} + w_{i+2} + \cdots + w_n$$
  - 剪枝条件：
  - 若$cw + r \leq bestw$，则剪枝



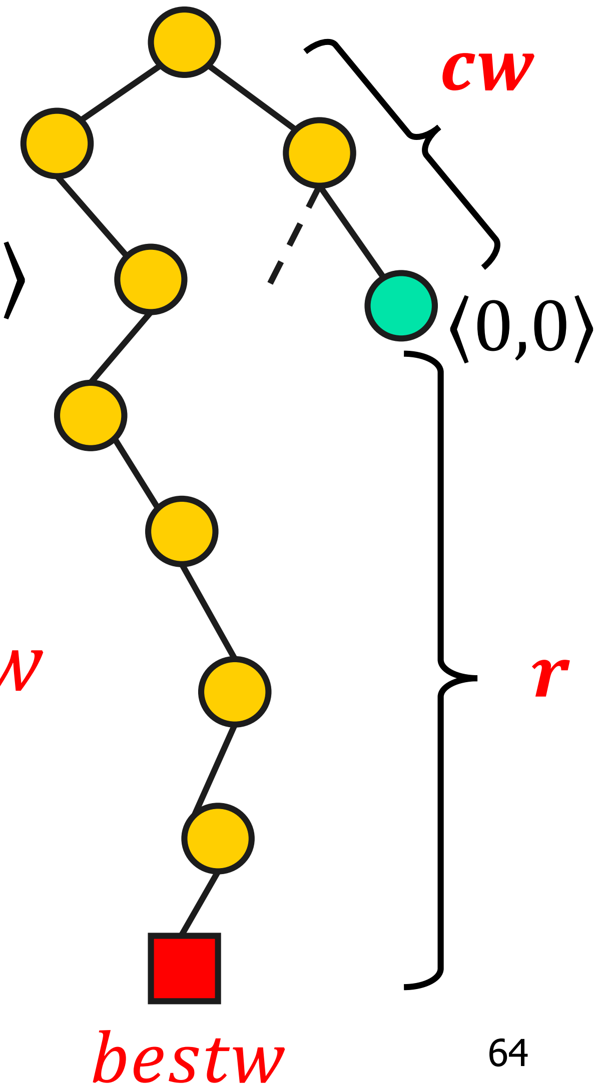
装载问题的剪枝函数

- 实例
  - 物品重量

    $$W = \langle 90,65,40,30,20,12,10 \rangle$$
  - $c_1 = 152,\ c_2 = 130$
  - 最优解：$\langle 1,0,1,0,0,1,1 \rangle$
  - *bestw*=152
  - *cw*=0
  - *r*=40+30+20+12+10=112

$$cw + r \leq bestw$$

剪枝！



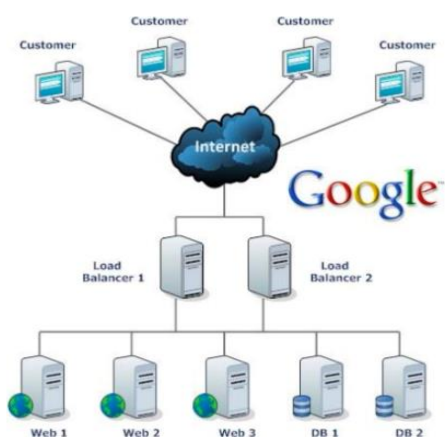
装载问题的算法实现

```
void backtrack (int i)
{// 搜索第i层结点
    if (i > n)
        更新最优解bestx,bestw;return;          到达叶结点
    r -= w[i];
    if (cw + w[i] <= c) {                  搜索左子树
        x[i] = 1;
        cw += w[i];
        backtrack(i + 1);
        cw -= w[i];        }
    if (cw + r > bestw)   {                搜索右子树
        x[i] = 0;
        backtrack(i + 1);        }
    r += w[i];
}
```

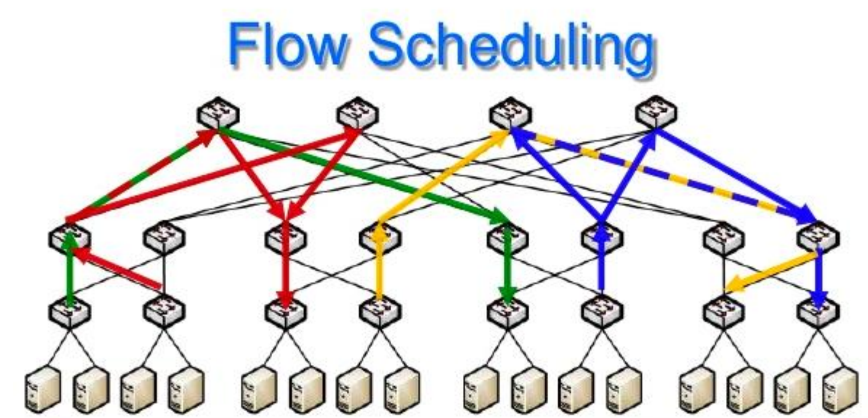
批处理作业调度问题

批处理作业调度及其应用

- 给定n个作业的集合$\{J_1,J_2,\ldots,J_n\}$。每个作业必须先由机器1处理，然后由机器2处理。
  - t_{ji}是作业J_i需要机器j的处理时间。
  - F_i是作业i的完成时间。
- 目标：求最佳作业调度方案使作业完成时间和最小



Web服务器调度



网络交换机的流调度

批处理作业调度问题的解空间

- 实例：

梅园打印店每天要处理很多打印任务，分两步：

- **将文件拷贝至电脑**
- **在打印机打印**

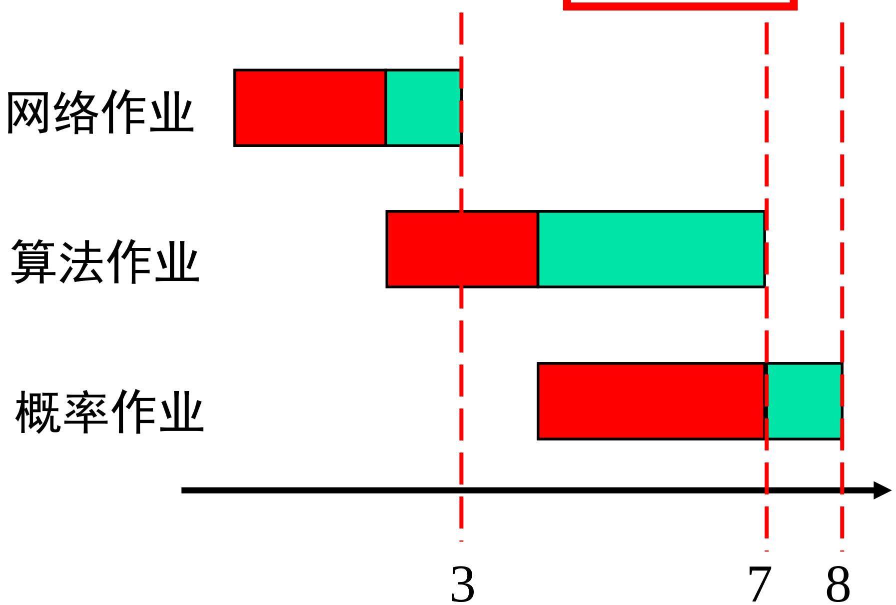
问题：给定一系列任务，且任务的拷贝和打印时间已知，请找出这些任务先后顺序，使所有任务的总完成时间最短。

t_{ji}	拷贝	打印
网络作业	2	1
概率作业	3	1
算法作业	2	3

批处理作业调度问题的解空间

- 问题的解

可行解	⟨1,2,3⟩	⟨1,3,2⟩	⟨2,1,3⟩	⟨2,3,1⟩	⟨3,1,2⟩	⟨3,2,1⟩
完成时间	19	18	20	21	19	19

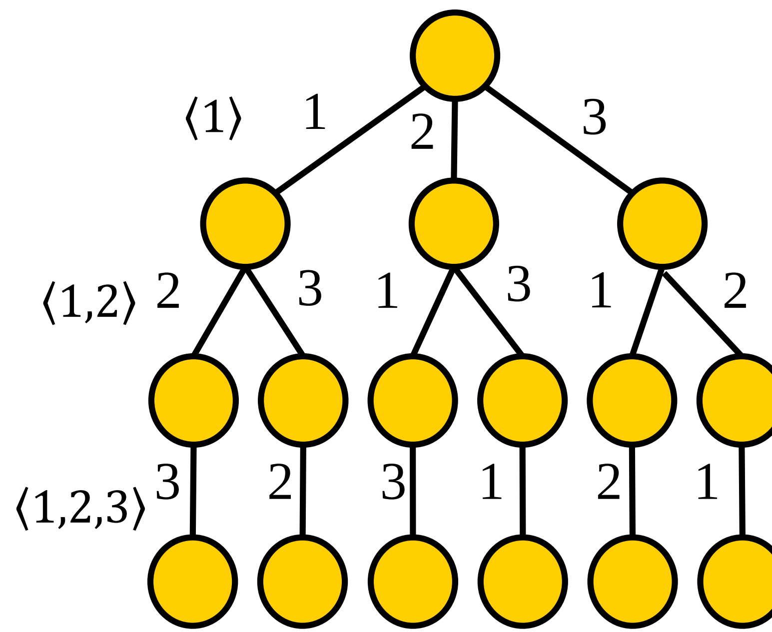


	任务	拷贝	打印
1	网络作业	2	1
2	概率作业	3	1
3	算法作业	2	3

批处理作业调度问题的解空间树

- 实例

	任务	拷贝	打印
1	网络作业	2	1
2	概率作业	3	1
3	算法作业	2	3



- 解空间
  - 排列树（n=3）
- 剪枝函数
  - 当前方案的执行时间>最优解

批处理作业调度问题的算法实现

```
void Backtrack(int i)
{
    if (i > n) {
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestf = f;}
    else
        for (int j = i; j <= n; j++) {
            f1+=M[x[j]][1];
            f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];
            f+=f2[i];
            if (f < bestf) {
                Swap(x[i], x[j]);
                Backtrack(i+1);
                Swap(x[i], x[j]); }
            f1- =M[x[j]][1];
            f- =f2[i];
        }
}
```

到达叶子结点

当前方案的执行时间

若不被剪枝

M,	// 各作业所需的处理时间
x,	// 当前作业调度
bestx,	// 当前最优作业调度
f2,	// 机器2完成处理时间
f1,	// 机器1完成处理时间
f,	// 完成时间和
bestf,	// 当前最优值
N	// 作业数

两个核心问题小结

- 定义解空间
  - 解向量为$\langle x_1, x_2, \ldots, x_n \rangle$
  - 确定x_i的取值集合为X_i
  - 子集树、排列树、n叉树
- 定义剪枝函数
  - 可行性约束函数
  - 限界函数

回溯法的剪枝技巧

两种剪枝函数

1. 可行性约束函数

2. 限界函数

两种剪枝函数

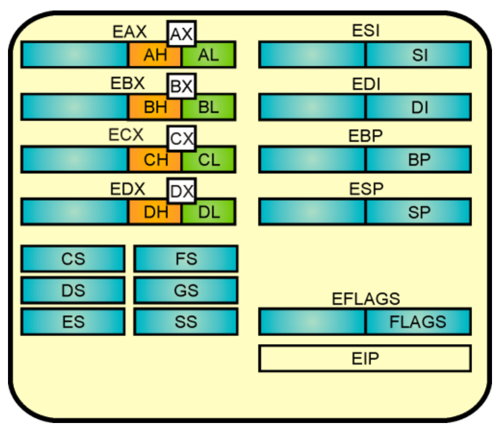
1 可行性约束函数

2 限界函数

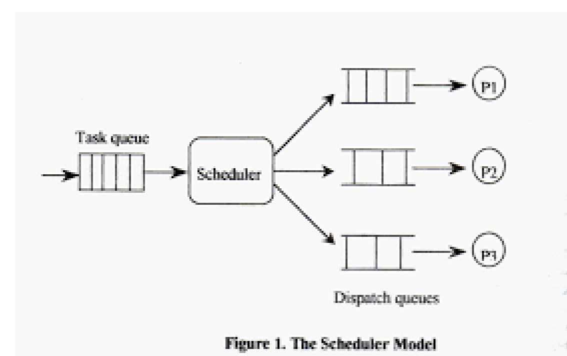
图着色问题及其应用（回顾）

- 给定无向连通图$G=(V,E)$，是否可k种颜色对G中顶点着色，可使任意两个顶点着色不同。
  - 是与否的判定问题
  - 解向量：$\langle x_1, x_2, \cdots, x_n \rangle$, x_i表示顶点i所着颜色

地图着色

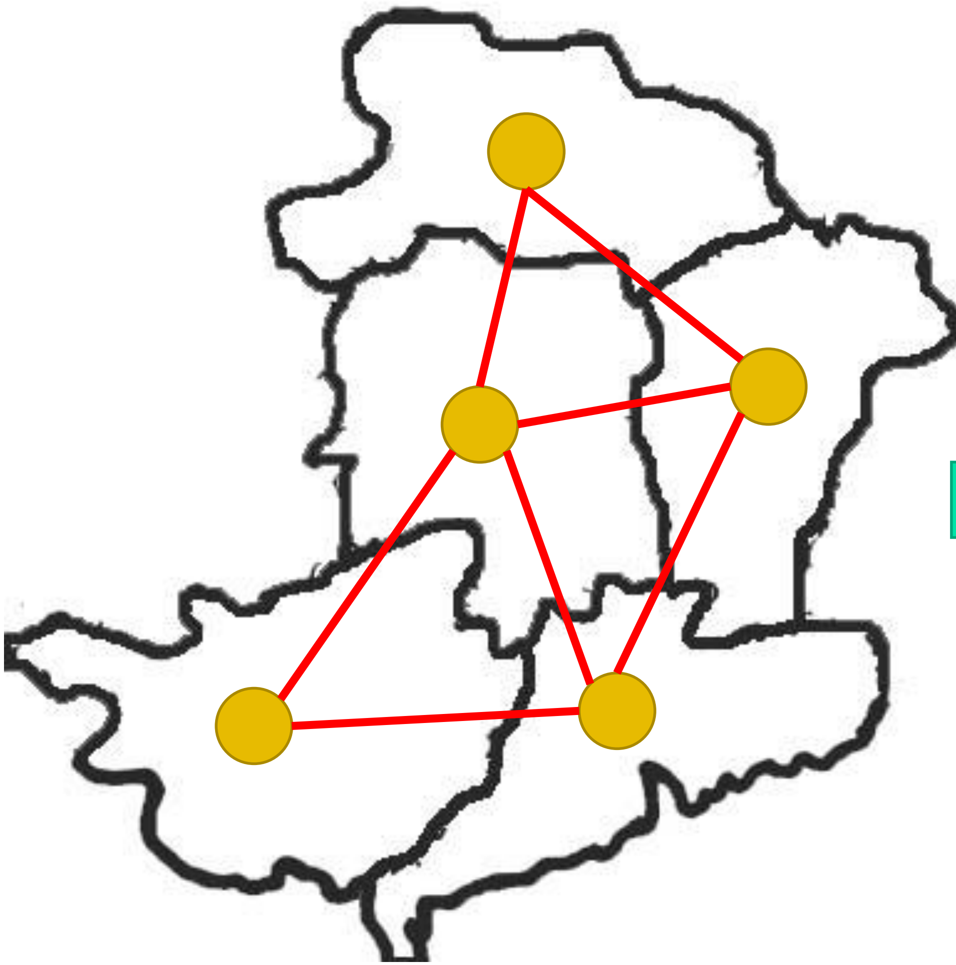


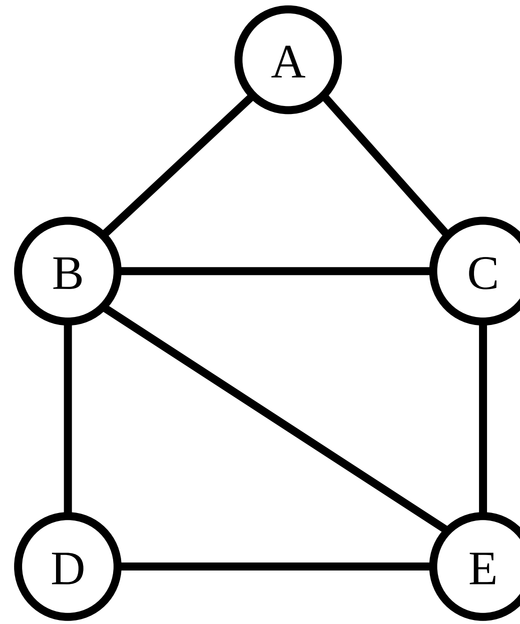
程序编译器的
寄存器分配算法



任务调度

实例：给中国地图着色



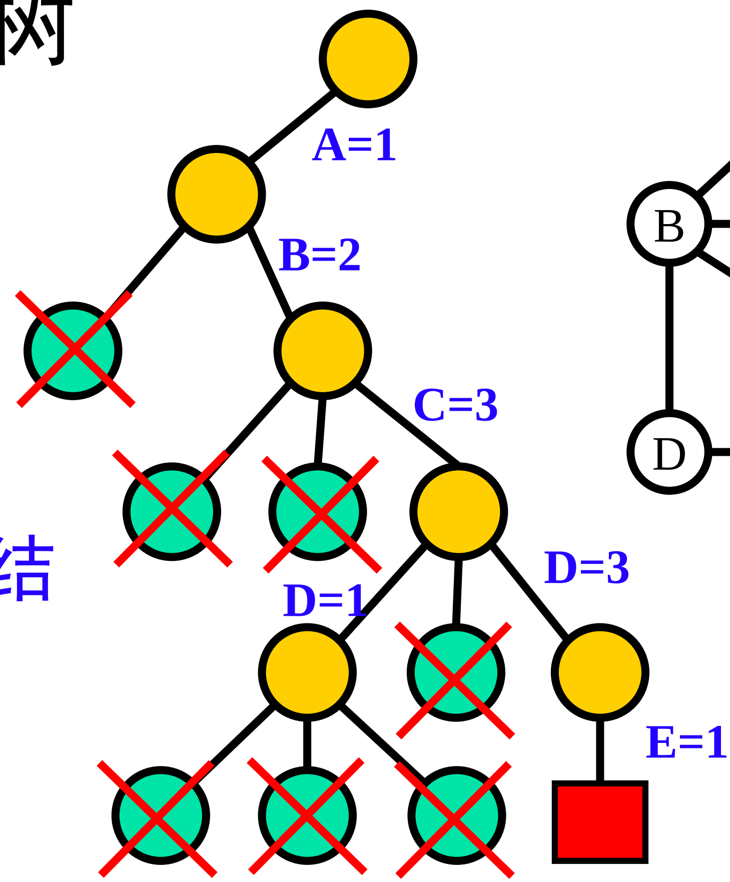


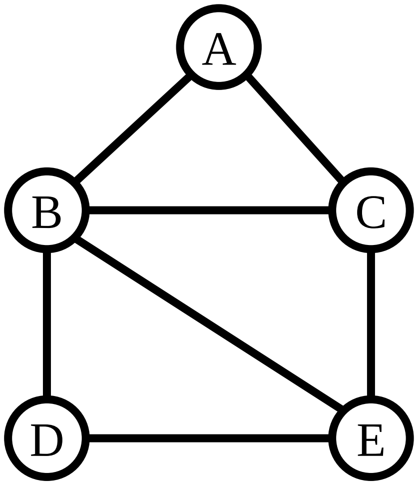
是否可以用三种不同颜色给这个地图着色？

图的*m*着色问题的解空间树

- 3着色问题——三叉树
- 剪枝函数
  - 可行性约束函数

解空间有3^5=243个结点，而回溯只搜了其中14个结点就找到了解。





两种剪枝函数

1 可行性约束函数

2 限界函数

回溯法与组合优化问题（回顾）

例如：0-1背包问题

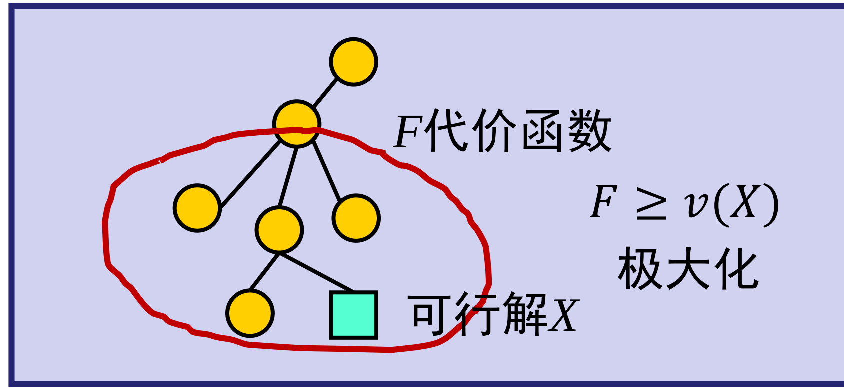
最大化 $x_1 + 3x_2 + 5x_3 + 10x_4$ 最大化价值

满足约束条件 $\begin{cases} 2x_1 + 3x_2 + 6x_3 + 7x_4 \leq 10 \\ x_i \in \{0,1\}, \quad i = 1, 2, 3, 4 \end{cases}$ 重量约束 定义域约束

- 组合优化问题
  - 目标函数（极大化或极小化）
  - 约束条件（解满足的条件）
  - 可行解：搜索空间满足约束条件的解
  - 最优解：使目标函数达极大（或极小）的可行解

回溯法的代价函数

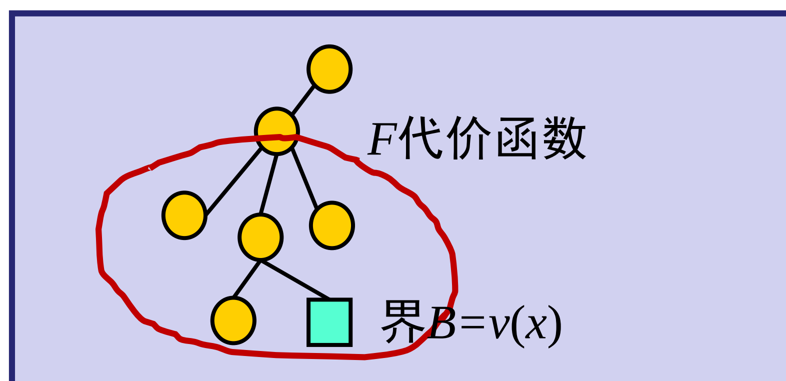
- 代价函数的计算位置
  - 搜索树的结点
- 代价函数的值
  - 对于极大化问题，以该点为根的子树所有可行解的值的上界（极小化问题为下界）



- 代价函数的性质
  - 对于极大化问题，父结点代价不小于子结点的代价（极小化问题相反）

回溯法的界

- 界的含义
  - 当前得到可行解的目标函数的最大值（极小化问题相反）
- 界的初值
  - 极大化问题初值为0（极小化问题为最大值）
- 界的更新
  - 得到更好的可行解时



回溯法的剪枝函数

- 剪枝函数
  - 不满足约束条件（可行性约束函数）
  - 代价函数值不优于当前的界（限界函数）
- 界的更新
  - 对于极大化问题，如果一个新的可行解的优化函数值大于（极小化问题为小于）当前的界，则把界更新为该可行解的值

回溯法剪枝的实例

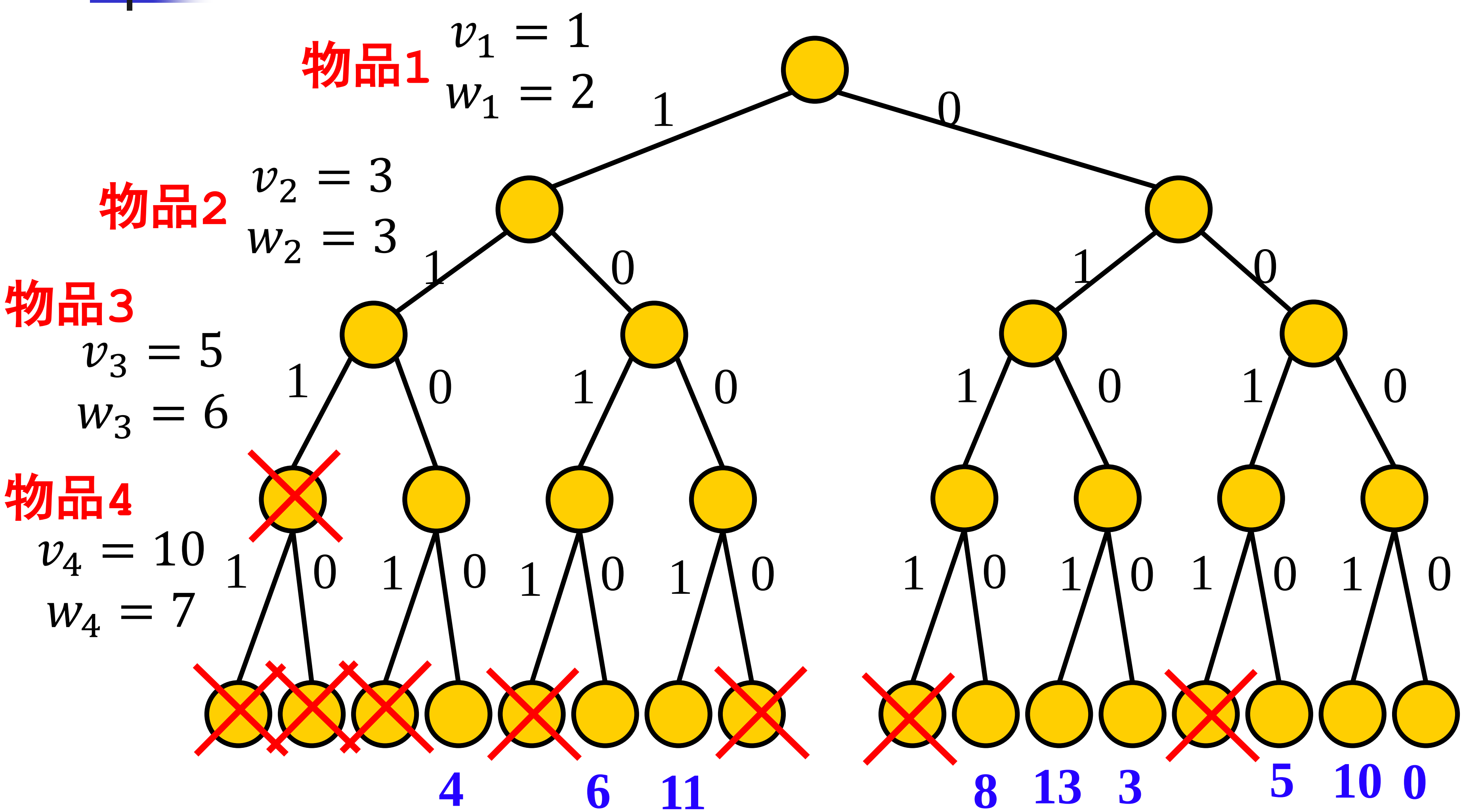
- 0-1背包问题
  - 4种物品，重量w_i和价值v_i分别为
  - $v_1 = 1, v_2 = 3, v_3 = 5,\ v_4 = 10$
  - $w_1 = 2, w_2 = 3, w_3 = 6,\ w_4 = 7$
  - 背包重量限制为10

例如：0-1背包问题

最大化 $x_1 + 3x_2 + 5x_3 + 10x_4$

满足约束条件
$$\begin{cases} 2x_1 + 3x_2 + 6x_3 + 7x_4 \le 10 \\ x_i \in \{0,1\},\quad i = 1, 2, 3, 4 \end{cases}$$

通常的回溯法做法



物品1 $v_1 = 1$ $w_1 = 2$
物品2 $v_2 = 3$ $w_2 = 3$
物品3 $v_3 = 5$ $w_3 = 6$
物品4 $v_4 = 10$ $w_4 = 7$
1 0
1 0 1 0
1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
4 6 11 8 13 3 5 10 0