回溯法（穷举法）

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

  （1）递归回溯

  （2）迭代回溯

  （3）子集树回溯

  （4）排列树回溯

学习要点

- 通过应用范例学习回溯法的设计策略
  （1）0-1背包问题；
  （2）旅行售货员问题；
  （3）装载问题
  （4）批处理作业调度；
  （5）n后问题；
  （6）图的m着色问题；

回溯法基础

回溯法

- 搜索问题解空间的方法-回溯法
  - 可以枚举问题的所有解
  - 通用解题法
  - 在解空间树中，按深度优先策略搜索
- 可以解决
  - 搜索问题的一个可行解
    - 搜索到第一个可行解则停止搜索
  - 搜索问题的最优解
    - 遍历解空间找到最优解

回溯法

- 定义问题的解空间
- 搜索解空间

确定解空间的组织结构后，从开始结点（根节点）出发，以深度优先方式搜索整个解空间。开始结点成为活结点，也是当前的扩展结点。在当前扩展结点处，搜索向纵深方向移动一个结点。新结点成为新的活节点，并成为当前的扩展结点。如果当前扩展结点不能再向纵深方向移动，那么当前结点成为死结点，此时往回移动（回溯）至最近的活结点处，并使这个结点成为当前扩展结点。

回溯法用这种递归的方式搜索整个解空间，直至找到所要求的解或者解空间中已无活结点为止。

0-1背包问题

0-1背包问题

- 形式化描述（重量w 价值v 容量C）
  - 输入：$\{<w_1, v_1>, <w_2, v_2>, \ldots, <w_n, v_n>\}$和$C$
  - 输出：$(x_1, x_2, \ldots, x_n)$，$x_i \in \{0, 1\}$满足 $\sum_{i=1}^{n} w_i x_i \le C$
  - 优化目标：$\max \sum_{i=1}^{n} v_i x_i$

0-1背包问题

实例

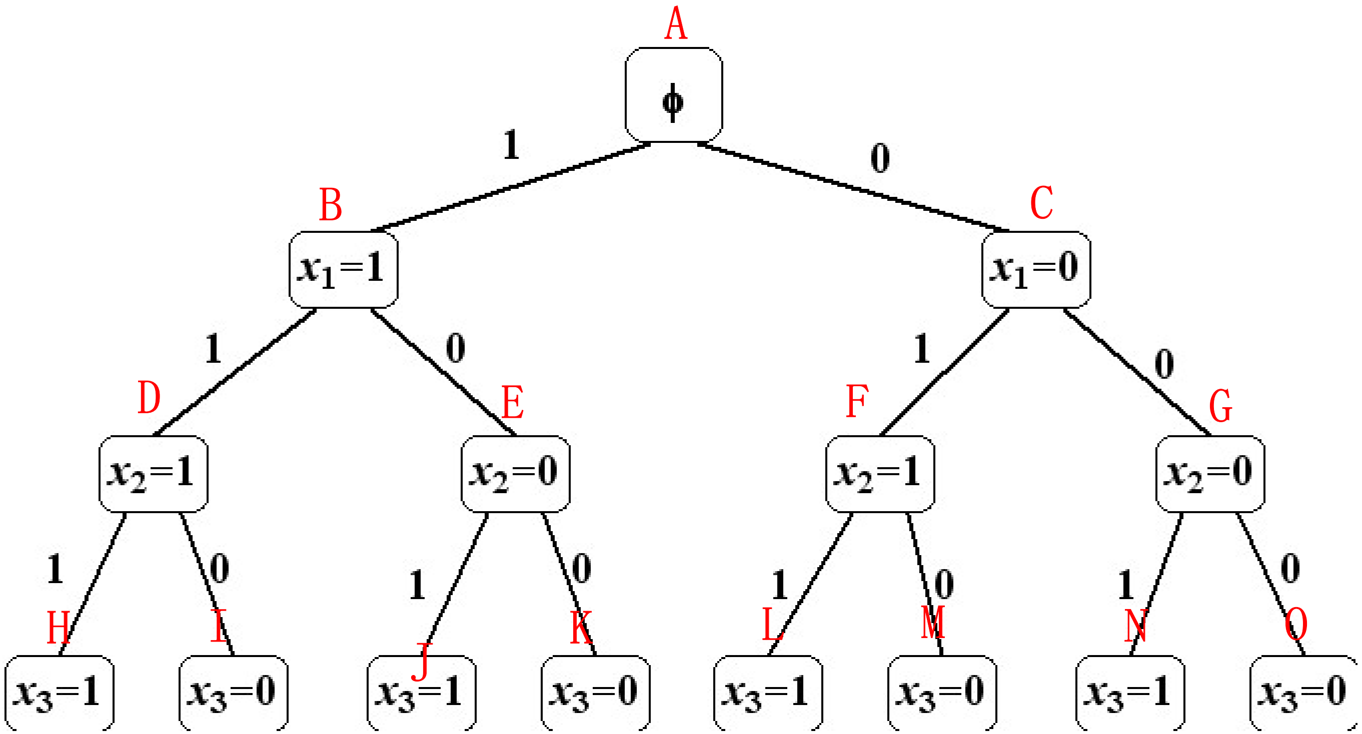
- 物品个数为 n=3
- 背包的容量为 C=30
- 物品的重量分别为 w={16，15，15}
- 物品的价值分别为 v={45，25，25}

解空间

- (x_1, x_2, x_3)的所有可能取值
- {(0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)}
- 可用一颗完全二叉树表示该问题解空间，解空间树

0-1背包问题

❐ 解空间树

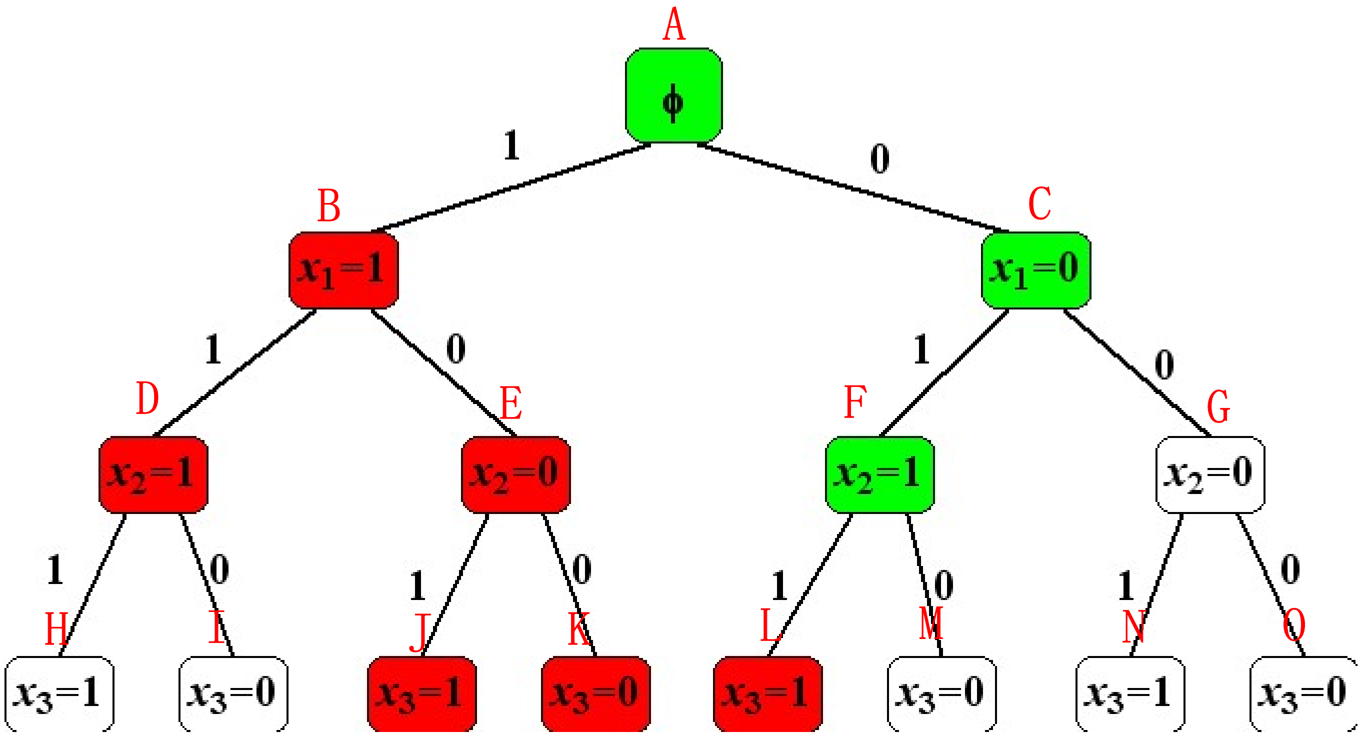


A
ϕ
1
0
B
C
$x_1=1$
$x_1=0$
1
0
1
0
D
E
F
G
$x_2=1$
$x_2=0$
$x_2=1$
$x_2=0$
1
0
1
0
1
0
1
0
H
I
J
K
L
M
N
O
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$

0-1背包问题

w={16，15，15}
v={45，25，25}
物品重量小于30
最大化物品价值

- 搜索解空间（深度优先）



$x_3=1$: W=30
$x_2=1$: W=15
$x_1=0$: W=0
ϕ: W=0

最优解: (1

最优解: (0,1,1)，V=50

0-1背包问题

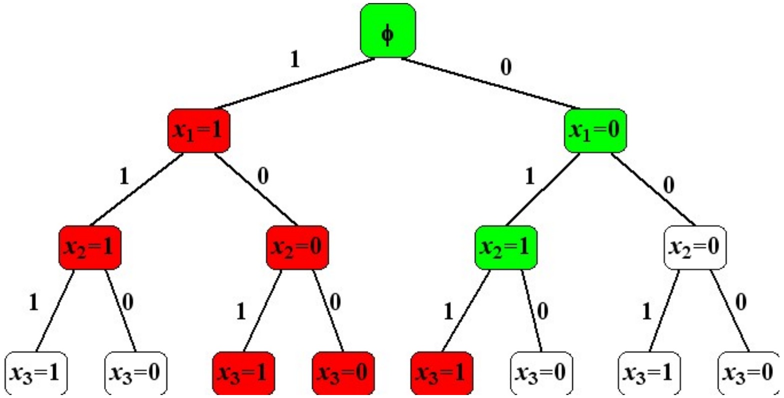
- 剪枝策略
  - 回溯法搜索解空间树通常采用两种方法避免无效搜索。
  - 约束函数，剪去不可行的子树（01背包）
  - 限界函数，剪去得不到最优解的子树（旅行商）

0-1背包问题

- 子集树
  - 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. (01背包问题)
- 子集树搜索代价
  - 叶节点数量为2^n，节点总数为$2^{n+1}-1$
  - 遍历解空间需要 $\Omega(2^n)$
- 回溯求解方法

递归、迭代

0-1背包问题



递归算法

```
void Backtrack(int t){
    if (t>n) 输出x；//已经搜索到了一个叶节点，输出解
    else
        for(i=0; i<=1; i++){
            x[t]=i;  // 0 or 1 ， 左右，两个取值
            if (所有已选物品的重量<C)
              Backtrack(t+1);
        }
}
```

0-1背包问题

迭代算法

```
void IterativeBacktrack( ){
    int t=1;
    for (i=1; i<=n; i++) x[i]=-1;
    while (t>0){
        if (t>n) {输出x;   t--; continue;}//找到解
        x[t]++;
        if (x[t]>1) t--;
        else{
            if (已选物品重量小于C) t++; //深一层
            else t--; //回溯
        }
    }
}
```

0-1背包问题

❑ 最优解上界

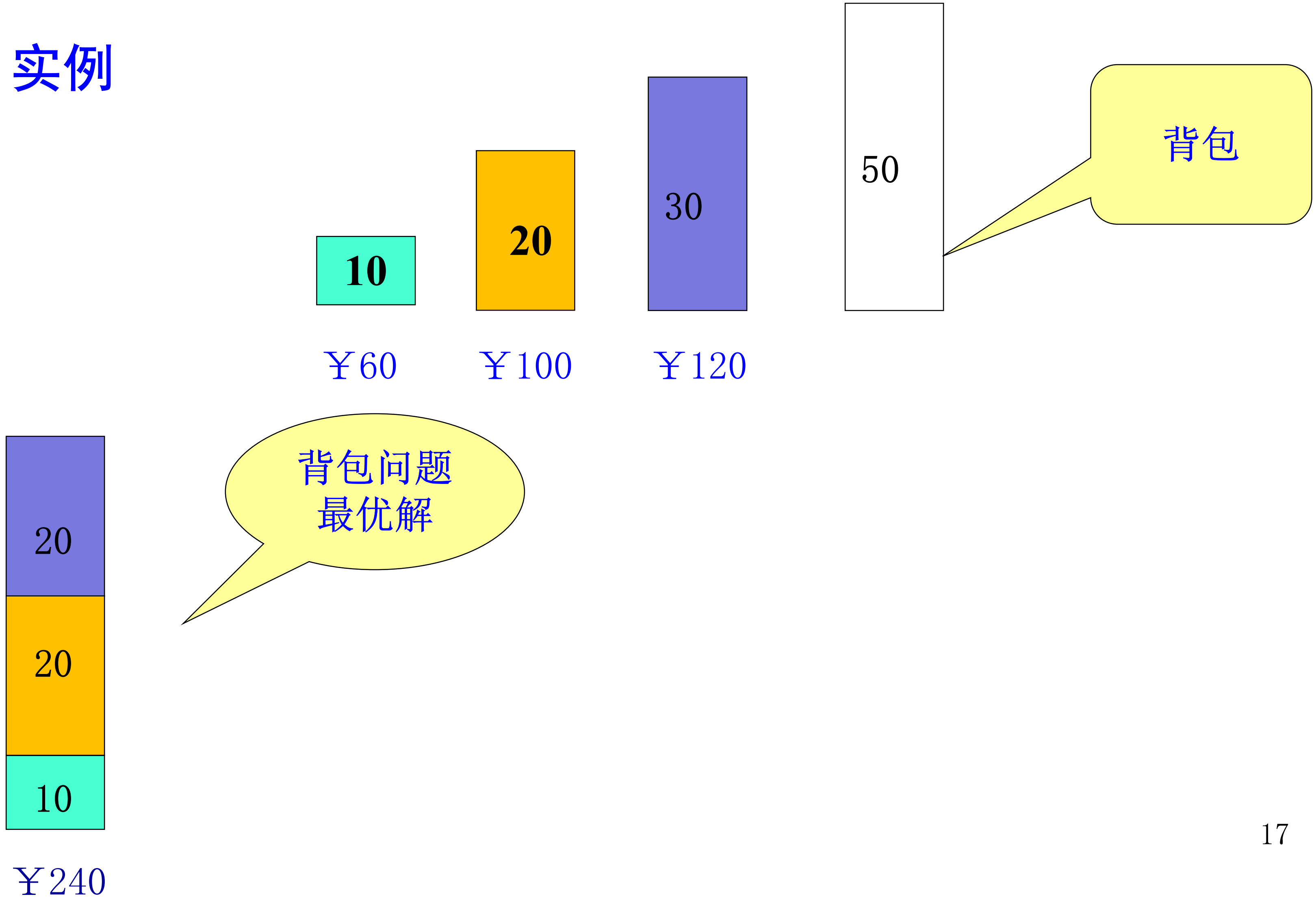
$$\max \sum_{i=1}^{n} v_i x_i$$

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

- 不超过一般背包问题的最优解
  - 一般：每种物品可以只取一部分
- 一般背包问题的最优解
  - 将物品按照单位重量的价值排序
  - 先装价重比最高的物品，直到背包装满为止
    - 最后一个物品可以只装一部分

背包问题最优解

➢ 实例



0-1背包问题

回溯法改进

- 将所有物品按照价重比排序
- 设当前背包中所有物品的价值为P，背包剩余容量为C’，剩余物品为{i, …, n}
- 那么装入背包的最大价值不会超过bound(i)
  - bound(i)＝P＋X
  - X是针对输入{i, …, n}和C’的背包问题的最优解

x_1	x_2	…	x_{i-1}	x_i	x_{i+1}	…	x_n

P　　　　X

0-1背包问题

- 回溯法改进
  - bestp保存当前的最优解的价值

```
void Backtrack(int t){
    if (t>n) {
        if (当前解x的代价>bestp){ 更新bestp;    输出x;}
    }
    else
        for(i=0; i<=1; i++){
            x[t]=i; //固定x[t]之后
            if (所有已选物品的重量<C && bound(t+1)>bestp)
                Backtrack(t+1);
        }
}
```

旅行商问题

旅行商问题

- 问题描述：

售货员要到若干个城市去推销商品，已知各城市之间的路程（或旅费），他要选定一条从驻地出发，经过每个城市一次，最后返回驻地的路线，使得总的行程（或花费）最少。

旅行商问题

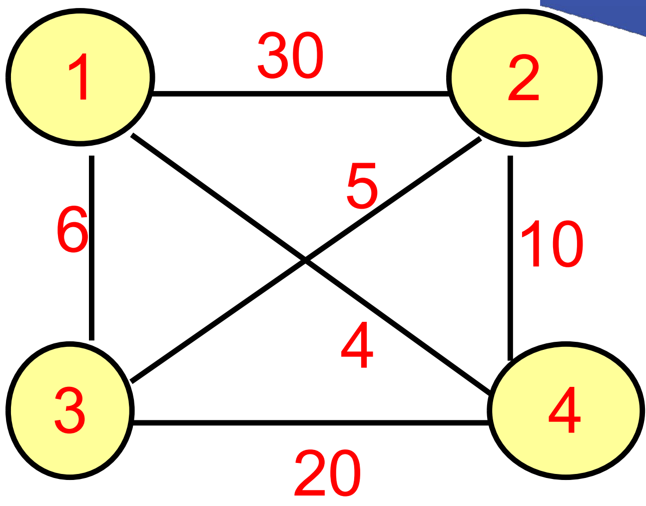
- 输入
  - 完全无向带权图G=(V, E)
    - |V|=n, |E|=m
    - 对于E中的某条边e，其长度为c(e)
- 输出
  - 最短的哈密尔顿回路
    - 经过每个节点一次且仅一次的回路

NP难问题

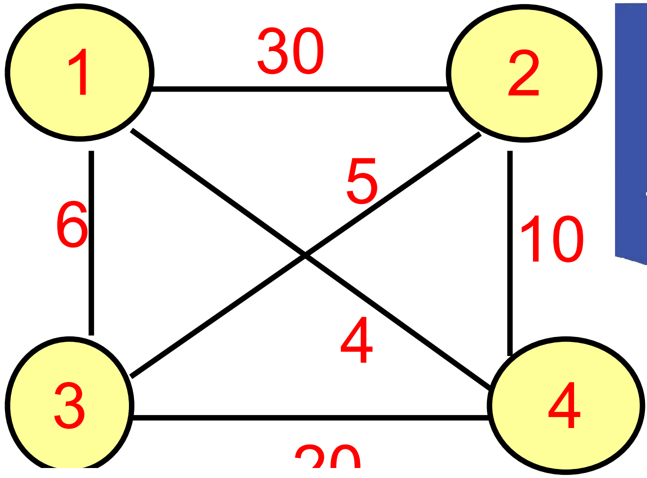
旅行商问题

- 实例
- 解空间
  - [1, 2, 4, 3, 1]
    [1, 3, 2, 4, 1]
    [1, 4, 3, 2, 1]
    [……]
  - 共(n-1)!个：

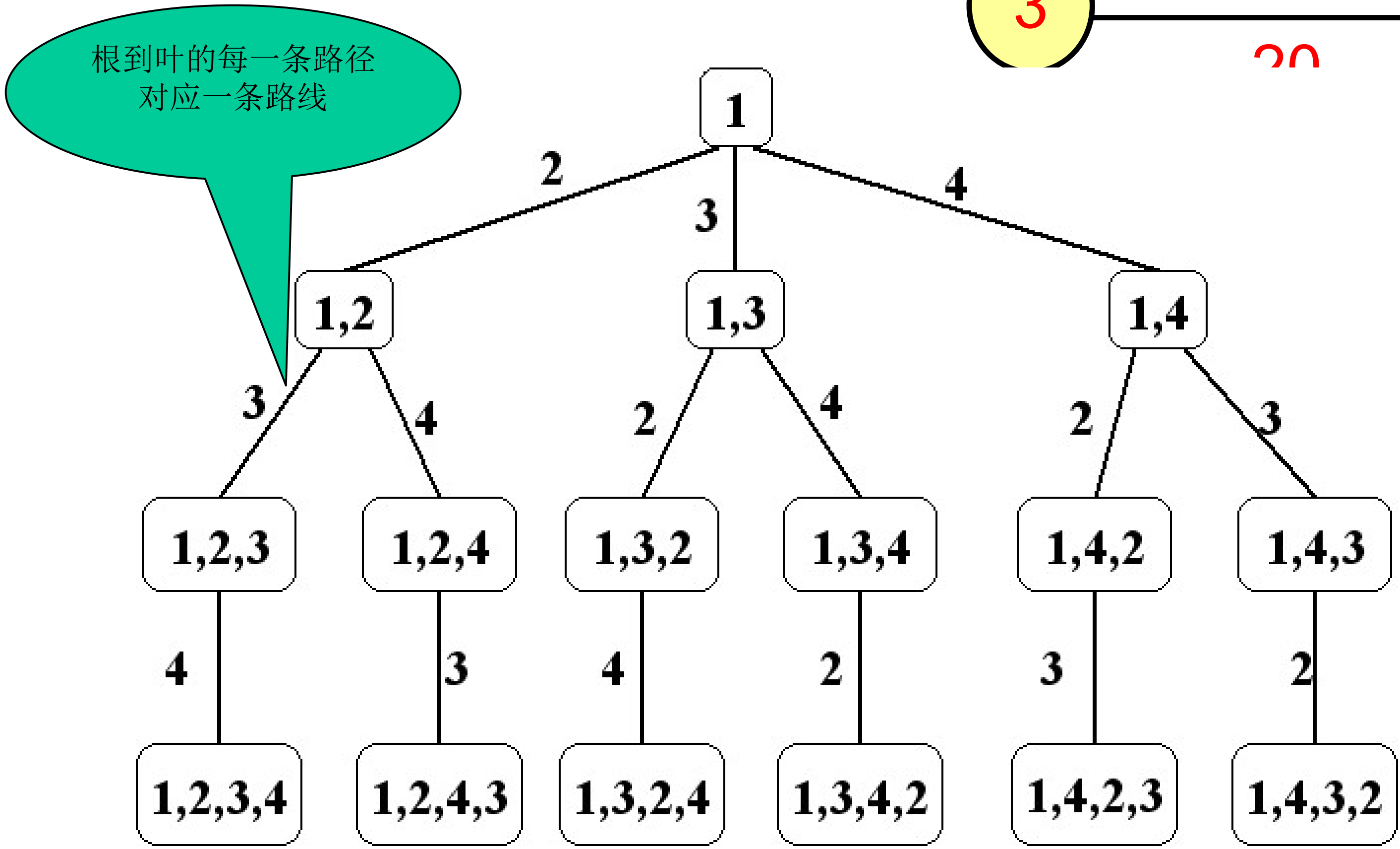
解= 起始点，除去起始点的其它点的全排列，起始点



旅行商问题

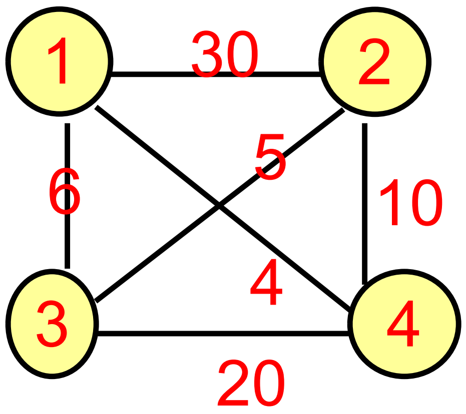

1
2
3
4
30
5
6
10
4

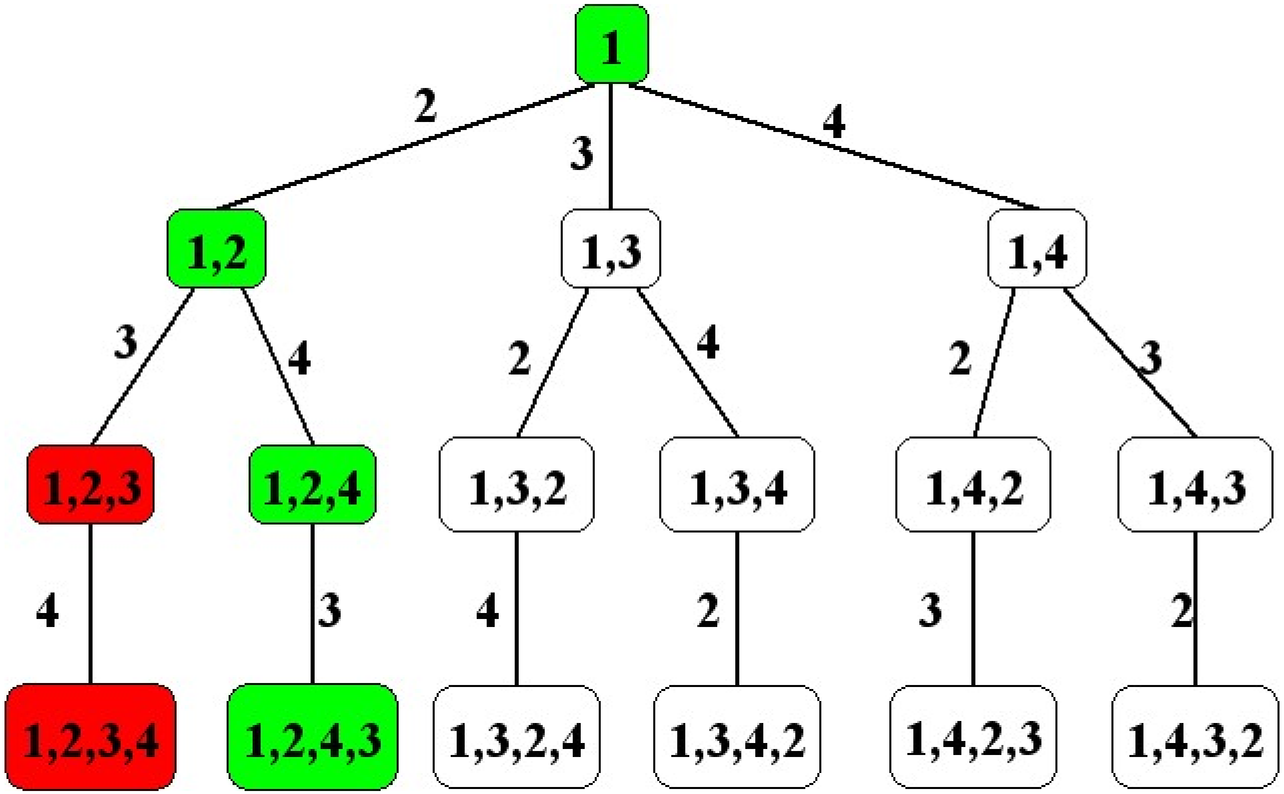
- 解空间树


根到叶的每一条路径
对应一条路线
1
2
3
4
1,2
1,3
1,4
3
4
2
4
2
3
1,2,3
1,2,4
1,3,2
1,3,4
1,4,2
1,4,3
4
3
4
2
3
2
1,2,3,4
1,2,4,3
1,3,2,4
1,3,4,2
1,4,2,3
1,4,3,2

旅行商问题

- 回溯法





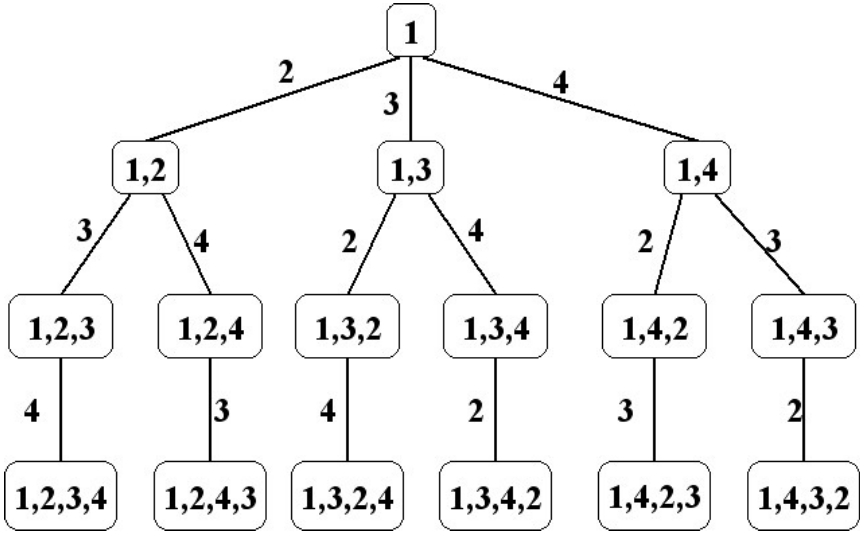
最优解: (1,3,2,4,1)，代价=25

旅行商问题

- 剪枝策略
  - 如果当前搜索节点处的代价超过已找到的最优解代价（限界），剪去其子树

旅行商问题

- 排列树
  - 问题的解是n个元素满足某种性质的排列时，解空间树称为排列树
- 排列数搜索代价
  - 叶节点n!
  - 遍历解空间需要 $\Omega(n!)$
- 回溯求解方法
  - 递归、迭代



旅行商问题

递归算法

初始时：x = [1, 2, 3, …, n], 即x[i]=i

```
void Backtrack(int t){
    if (t>n) 输出x；//找到叶子节点
    else
        for(i=t; i<=n; i++){//对于深度为t的节点，取值有多少个？
            Swap(x[t], x[i]);//x[i]为其取值
            if (现有路径长度小于已得到的最优值)
                Backtrack(t+1)； //有潜力，固定t，取下一个
            Swap(x[t], x[i]); //交换回来，准备重新选
        }
}
```

旅行商问题

迭代算法

```
//y[t]记录x[t]选择了t到n中的哪个元素,初始时y[t]=t
void IterativeBacktrack( ){
    int t=1;
    while(t>0){
        if(t>n) {输出x;  t--; continue;}//找到叶子
        y[t]++;   //选择下一个,x[t]=t（不管有没有潜力，都选择下一个）
        if(y[t]>n)  {t--; continue; } //x[t]=n，所有取值都选完了
        swap(x[t], x[y[t]]);
        if(现有路径长度小于已得到的最优值) {
            t++;
            y[t]=t;//有潜力，固定当前t
        }
        else{
            swap(x[t], x[y[t]]);    t--; //没潜力，反交换，回溯
        }
    }
```

回溯法算法框架

回溯法搜索子集树

```
void Backtrack(int t){
    if (t>n) 输出x;
    else
        for(i=0; i<=1; i++){
            x[t]=i;
            if (Constraint(t) && Bound(t))
            //如果当前的部分解可行 且 可能产生最优解
                Backtrack(t+1);
        }
}
```

回溯法搜索排列树

初始时：x[n]=（1, 2, 3, ···, n）

```
void Backtrack(int t){
    if (t>n) 输出x;
    else
        for(i=t; i<=n; i++){
            Swap(x[t], x[i]);
            if (Constraint(t) && Bound(t))
            //如果当前的部分解可行 且 可能产生最优解
                Backtrack(t+1) ;
            Swap(x[t], x[i]);
        }
}
```

回溯法总结

- 剪枝策略
  - 用约束函数 **Constraint(t)** 剪去不可行子树
  - 用限界函数 **Bound(t)** 剪去得不到最优解的子树
- 时间复杂性
  - 搜索子集树$\Omega(2^n)$
  - 搜索排列树$\Omega(n!)$
- 空间复杂性
  - **O(h(n))**
    - h(n)为解空间树的高度

装载问题

装载问题

输入

- n个集装箱，其中集装箱i的重量为w_i
- 载重量分别为C_1和C_2的轮船

$$\sum_{i=1}^{n} w_i \le C_1 + C_2$$

输出

- （是否有）合理的装载方案将所有集装箱装上船

NP难问题

当 $\sum_{i=1}^{n} w_i = C_1 + C_2$ 时，等价于子集合问题，即判断是否存在一个子集和等于一个常数。

装载问题

- 如果有解，可以用以下方法获得
  - 将第一艘轮船尽可能装满
  - 然后将剩余的集装箱装上第二艘轮船

- 问题等价于

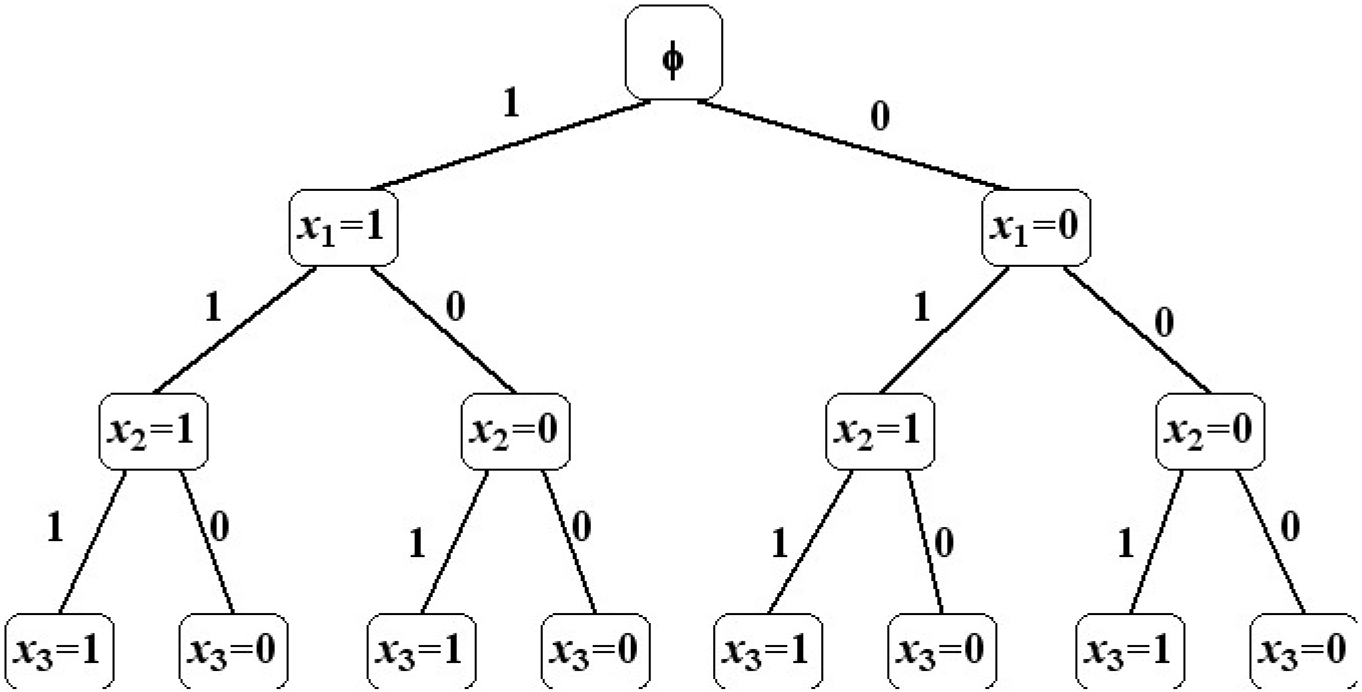
$$\max \sum_{i=1}^{n} w_i x_i$$

s.t.
$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C_1 \\ x_i \in \{0,1\},\ 1 \le i \le n \end{cases}$$

特殊的0-1背包问题：每种物品的价值等于重量

装载问题

- 解空间树
  - 子集树



装载问题

- 回溯法（搜索子集树）

```
void Backtrack(int t){
    if (t>n) 输出x;
    else
        for(i=0; i<=1; i++){
            x[t]=i;
            if (Constraint(t) && Bound(t))
            //如果当前的部分解可行 且 可能产生最优解
                Backtrack(t+1);
        }
}
```

装载问题

剪枝

- 约束函数 **Constraint**(t)： $\sum_{i=1}^{t} w_i x_i \le C_1$
- 限界函数 **Bound**(t)： $\sum_{i=1}^{t} w_i x_i + \sum_{i=t+1}^{n} w_i > BestC$

回溯法

时间复杂性：O(2^n)

空间复杂性：O(n)

批处理作业调度

批处理作业调度

输入

- n个作业$\{1, \ldots, n\}$
- 两台机器（M1和M2）
  - 作业 i 在M1和M2上的处理时间分别为 $a[i]$ 和 $b[i]$
  - 每个作业必须先由M1处理，再由M2处理

输出

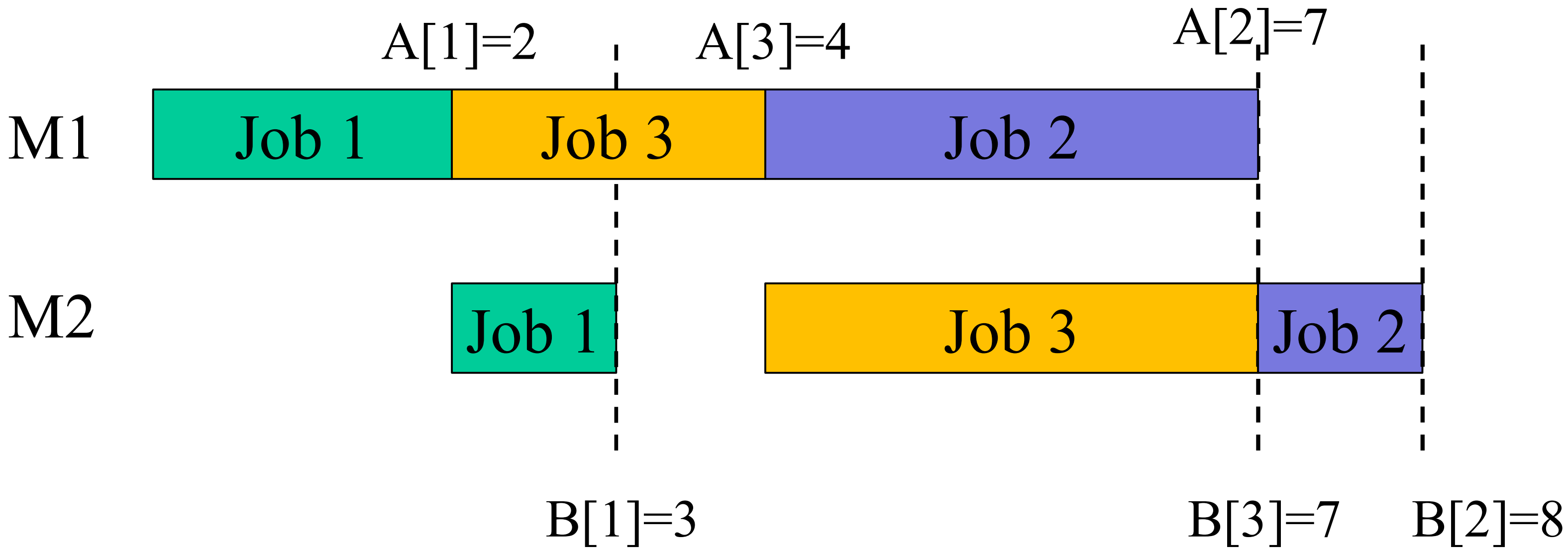
- 作业调度方案使得*总等待时间*最小
  - 作业 i在M1和M2上的完成时间分别为 $A[i]$ 和 $B[i]$ （从计时开始）
  - 总等待时间为 $\sum_{i=1}^{n} B[i]$

可以证明，存在一种最佳作业调度，使得两个机器上的作业以相同次序完成。

批处理作业调度

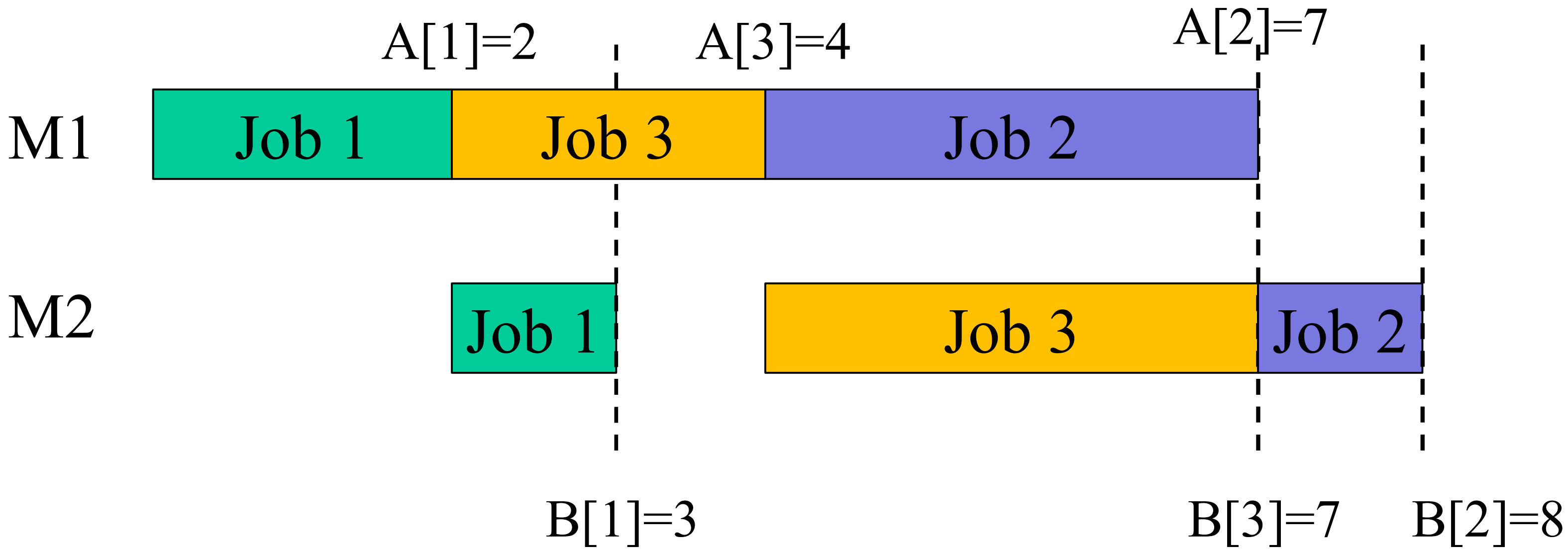
作业	*a*[*i*]	*b*[*i*]
Job 1	2	1
Job 2	3	1
Job 3	2	3

- 可能的调度方案
  - 123，132，213，231，312，321
- 最佳方案是132（总等待时间：18）



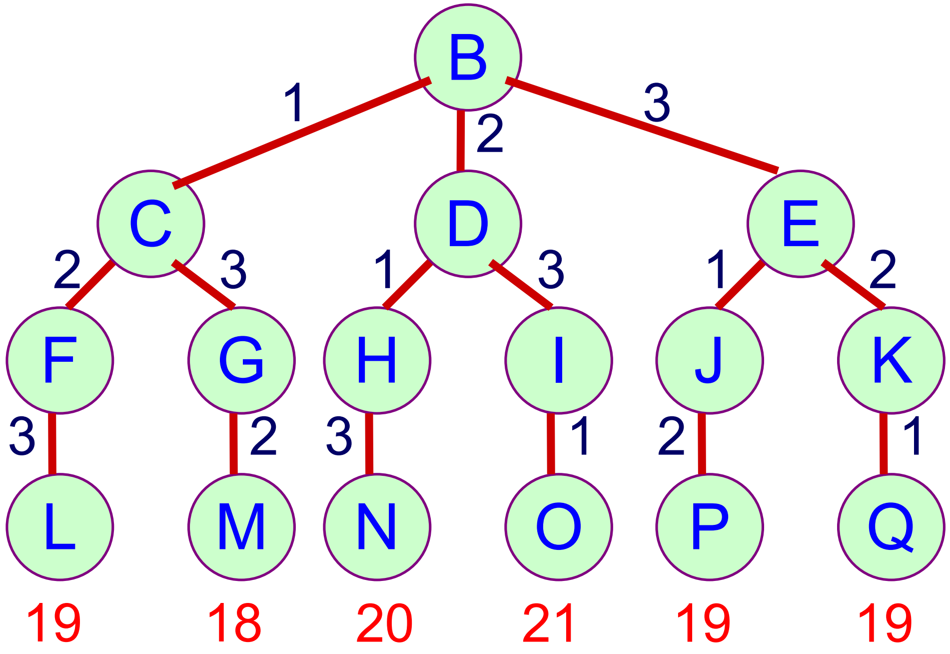
批处理作业调度

- 计算调度$\{J_1, J_2, \ldots, J_n\}$的等待时间
  - 计算B$[J_i]$
    - 计算A$[J_i]$=A$[J_{i-1}]$ + a$[J_i]$
    - 比较A$[J_i]$和B$[J_{i-1}]$
    - B$[J_i]$＝较大者 + b$[J_i]$



批处理作业调度

- 解空间树
  - 排列树



批处理作业调度

❐ 回溯法（搜索排列树）

初始时：x[n]=（1,2,3,…,n）

```
void Backtrack(int t){
    if (t>n) 输出x;
    else
        for(i=t; i<=n; i++){
            Swap(x[t], x[i]);
            if (Bound(t))
            //如果当前的部分解可行 且 可能产生最优解
                Backtrack(t+1) ;
            Swap(x[t], x[i]);
        }
}
```

时间复杂性：O(n!)
空间复杂性：O(n)

批处理作业调度

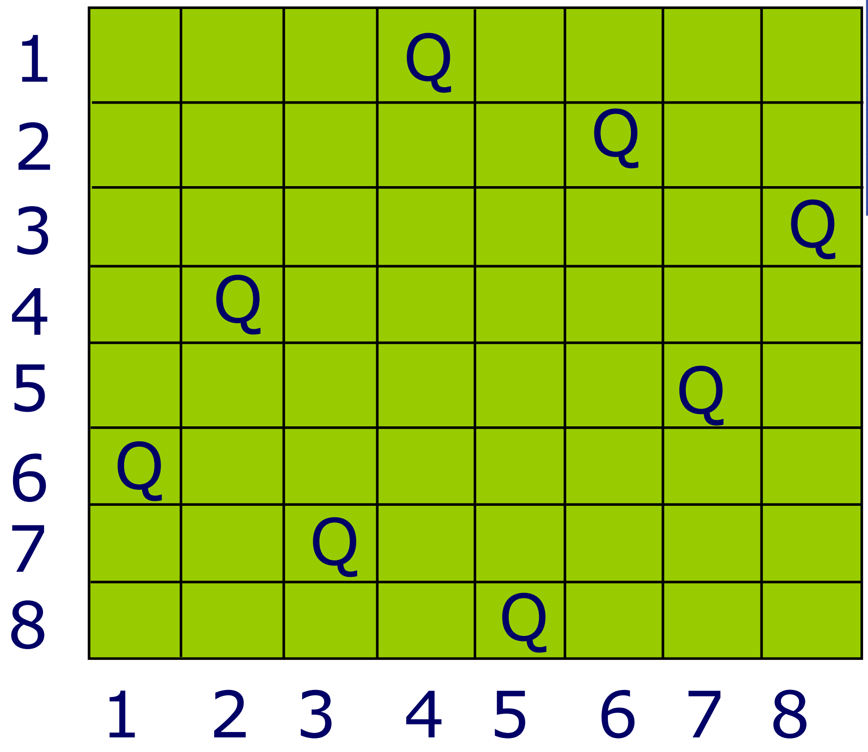
- 剪枝
  - 限界函数 **Bound**(t)： $\sum_{i=1}^{t} B[x[i]] < bestT$

    当前等待时间和小于当前最优等待时间。

n后问题

n后问题

- 输入
  - n×n的棋盘
  - n个皇后
- 输出
  - n个皇后的放置方案
    - 任意两个皇后都不在同一行、同一列或同一斜线上（正方形的对角线）



n后问题

❑ 解空间

- 每行有且仅有一个皇后
- 用x[i]表示第i行皇后位于第几列
  - 此皇后的坐标为(i, x[i])
- 问题的解是x[1,…, n]，满足
  - 任意两个皇后不在同一列上：**$x[i] \neq x[j]$**
  - 任意两个皇后不在同一斜线上

$$|i - j| \neq |x[i] - x[j]|$$

- x[1, …, n]是{1, …, n}的一个排列

解空间树：排列树

n后问题

❑ 回溯法

初始时：x[n]=（1, 2, 3, ···, n）

```
void Backtrack(int t){
    if (t>n) 输出x;
    else
        for(i=t; i<=n; i++){
            Swap(x[t], x[i]);
            if ( Constraint(t) )
                Backtrack(t+1) ;
            Swap(x[t], x[i]);
        }
}
```

n后问题

❑ 剪枝

约束函数

```
Constraint(t){
    for (i=1; i<t; i++){
        if(|i – t| == |x[i] – x[t]| ) return false;
    }
    return true;
}
```

回溯法

时间复杂性：O(n*n!)

空间复杂性：O(n)

图的m着色问题

图的m着色问题

- 输入
  - 无向连通图G
  - m种颜色
- 输出
  - 顶点着色方案
    - 任意两个相邻顶点都有不同颜色

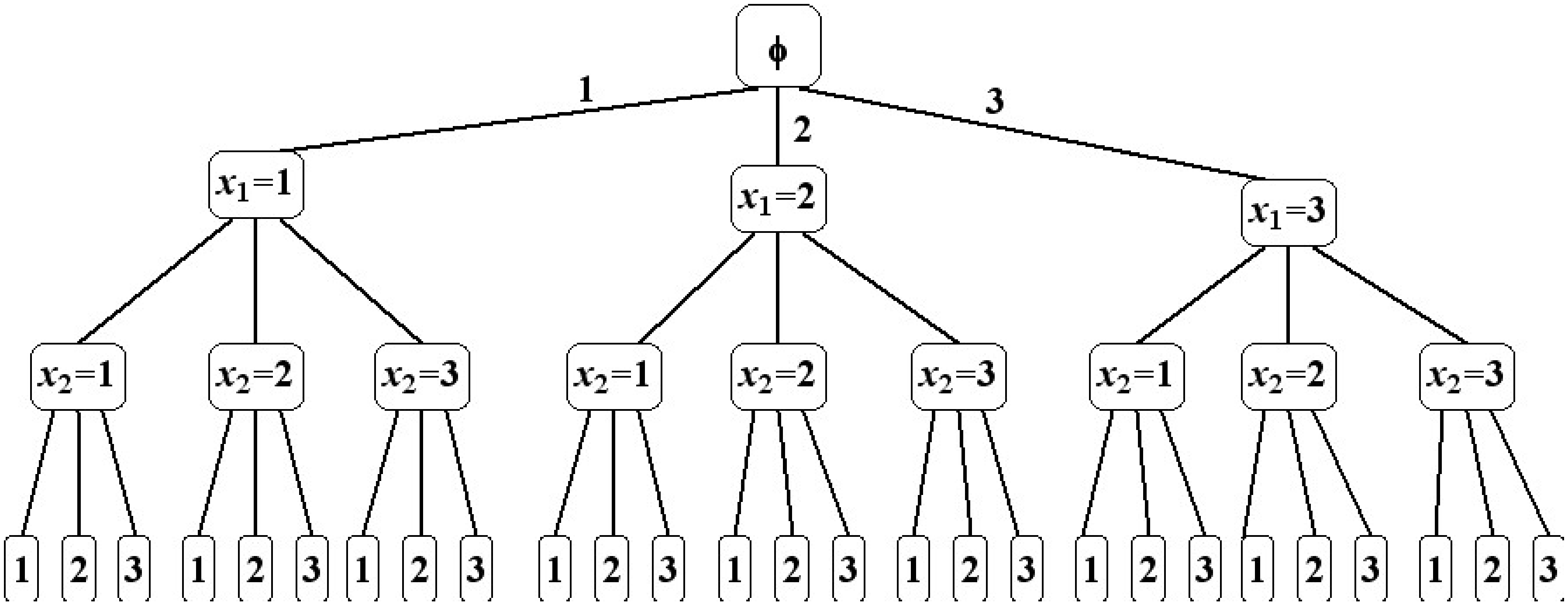
对于一个给定图和m中颜色，判断是否能m着色，如果能，找出所有的方案。

图的m着色问题

- 解空间
  - x[i]表示顶点i的颜色
    - $x[i] \in \{1, \ldots, m\}$
  - 每个x[i]有m种不同取值
  - x[1,…,n]有m^n种不同取值

图的m着色问题

- 解空间树(m=3)
  - 类似于子集树，每个x[i]有m个取值，完全m叉树。



图的m着色问题

❒ 回溯法

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

图的m着色问题

剪枝

约束函数

```
Constraint(t){
    for (i=1; i<t; i++){
        if(存在边(i, t)且x[i]==x[t]) return false;
    }
    return true;
}
```

回溯法

时间复杂性：$O(n*m^n)$

空间复杂性：$O(n)$

回溯法效率分析

回溯法效率分析

- 回溯法的效率取决于
  - 解空间中解的数量
    - 即满足显约束的x[1,…, n]的值的个数
  - 计算约束函数**Constraint**(t)所需时间
  - 计算限界函数**Bound**(t)所需时间
  - 满足约束函数和限界函数的解的数量
  - x[1,…,n]的选取顺序

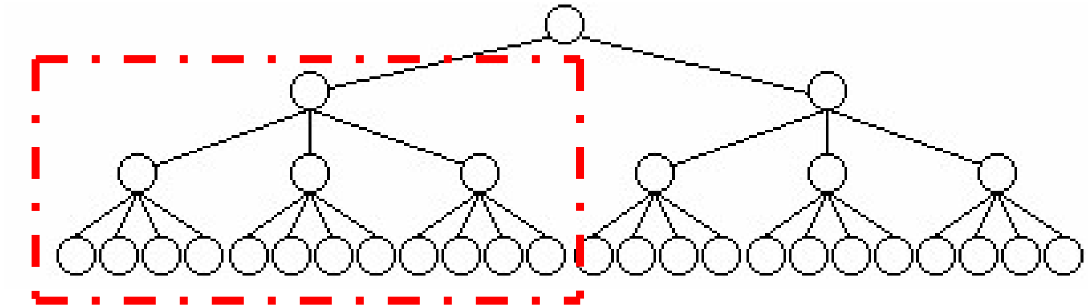
回溯法效率分析

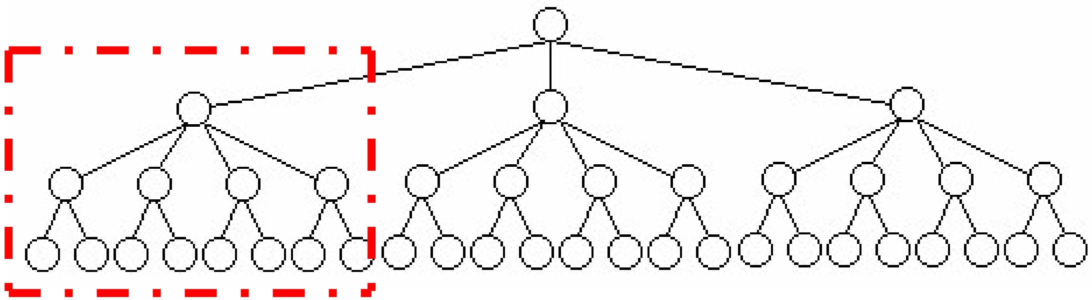
- 好的约束（限界）函数能显著地减少所生成的结点数。但这样的约束（限界）函数往往计算量较大。因此，在选择约束（限界）函数时通常存在搜索结点数与约束函数计算量之间的折衷。
- 对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的。在其它条件相当的前提下，让可取值最少的x[i]优先。

回溯法效率分析

- 实例

  图中关于同一问题的2棵不同解空间树


(a)


(b)

- 前者的效果明显比后者好

总结

✓ 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

（1）递归回溯最优子结构性质

（2）迭代回溯贪心选择性质

（3）子集树算法框架

（4）排列树算法框架