

第五章 搜索法

■ 教学目标

- ∞ 掌握回溯法的算法框架
- ∞ 理解回溯法及分支限界法的基本思想
- ∞ 掌握回溯法及分支限界法的异同
- ∞ 掌握子集树、排列树及满 m 叉树的算法设计模式
- ∞ 通过实例学习，掌握回溯法及分支限界法解决问题的方法步骤



5.1 穷举搜索

■ 思想

∞ 针对问题的可能解是有限种的情况，逐一检查所有可能的情况，从中找到问题真正的解。

■ 示例

∞ 给定一个有向带权图 $G=(V, E)$ ，权非负，如图5-1所示。
找出顶点 $1 \rightarrow 5$ 的最短路径及其长度。

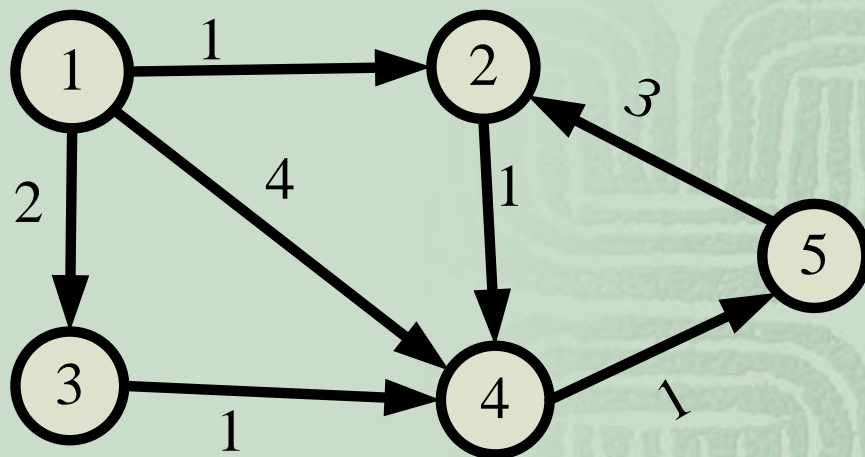


图5-1 有向带权图

■ 问题分析

☞ 该问题所有可能的路径只有三条，分别是：

■ $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$

■ $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$

■ $1 \rightarrow 4 \rightarrow 5$

☞ 采用穷举搜索的方法逐一检查这三条路径的长度。

☞ 最短路径为 $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ ，其长度为3。

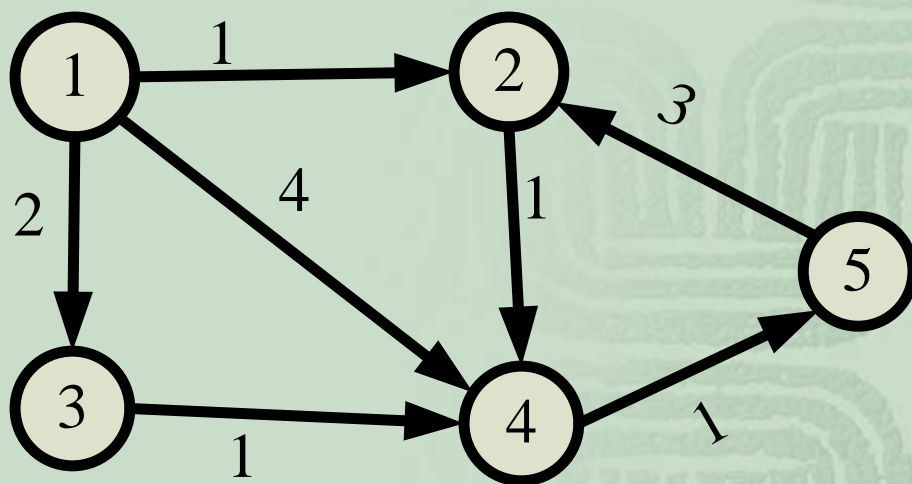


图5-1 有向带权图

5.2 深度优先搜索

■ 思想（给定图 $G=(V, E)$ ）

∞ 初始时，所有顶点均未被访问过，任选一个顶点 v 作为源点。该方法先访问源点 v ，并将其标记为已访问过；然后从 v 出发，选择 v 的一个邻接点（子结点） w ，如果 w 已访问过，则选择 v 的另外一个邻接点；如果 w 未被访问过，则标记 w 为已访问过，并以 w 为新的出发点，继续进行深度优先搜索；如果 w 及其子结点均已搜索完毕，则返回到 v ，再选择它的另外一个未曾访问过的邻接点继续搜索，直到图中所有和源点有路径相通的顶点均已访问过为止；若此时图 G 中仍然存在未被访问过的顶点，则另选一个尚未访问过的顶点作为新的源点重复上述过程，直到图中所有顶点均已访问过为止。

■ 示例:

∞ 给定一个有向图 $G=(V, E)$, 如图5-2所示。给出深度优先搜索该图的一个访问序列。

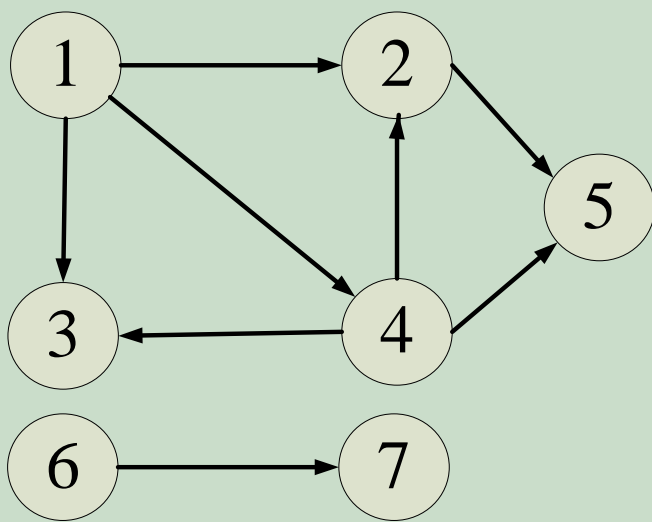
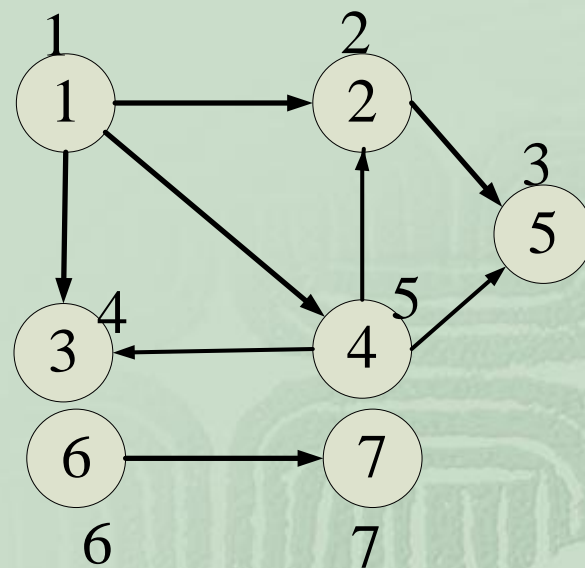


图5-2 有向图



5-3 搜索顺序

∞ 输出一个深度优先搜索序列: 1,2,5,3,4,6,7

■ 练习

- ∞ 给定一个无向图 $G=(V, E)$ ，如图5-4所示。
给出深度优先搜索该图的一个搜索序列。

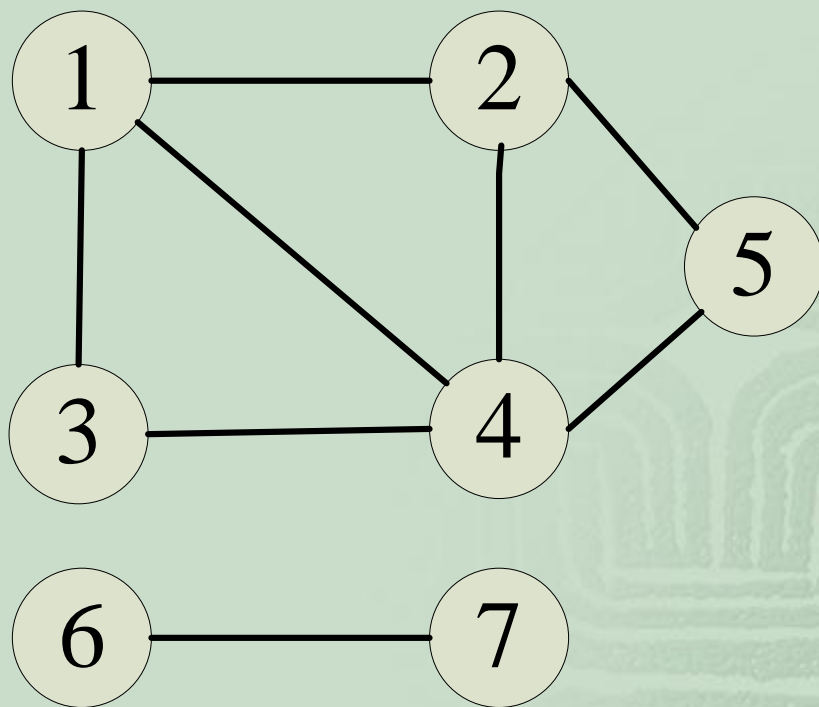


图5-4 无向图



算法描述

```
bool Visited[n+1];    //标记图中顶点是否被访问过
for(int i=1;i<=n;i++)
    Visited[i]=0;      //用0表示顶点未被访问过
```

//从顶点k出发进行深度优先搜索

```
Dfsk(int k)
{
    Visited[k]=1;    //标记顶点k已被访问过
    for(int j=1; j<=n; j++)
        if(c[k][j]==1 && Visited[j]==0)    //c[][]是邻接矩阵
            Dfsk(j);
}
```

//深度优先搜索整个图G

```
Dfs( )
{
    for(int i=1; i<=n; i++)
        if(Visited[i]==0)
            Dfsk(i);
}
```

如果产生所有从K点出发的路径，应该怎么改？

5.3回溯法

- 回溯法是在仅给出初始结点、目标结点及产生子结点的条件（一般由问题题意隐含给出）的情况下，构造一个图（隐式图），然后按照深度优先搜索的思想，在有关条件的约束下扩展到目标结点，从而找出问题的解。
- 通俗地讲：回溯法是一种“能进则进，进不了则换，换不了则退”的基本搜索方法



5.3.1 回溯法的算法框架及思想

■ 明确搜索范围（定义问题的解空间）

∞ 解的形式：一个 n 元式 (x_1, x_2, \dots, x_n)

∞ 确定 x_i 的取值范围（显约束），确定解空间的大小。

■ 确定解空间的组织结构

∞ 树或图

■ 搜索解空间（隐约束）

∞ 确定是否能够导致可行解——约束条件

∞ 确定是否能够导致最优解——限界条件



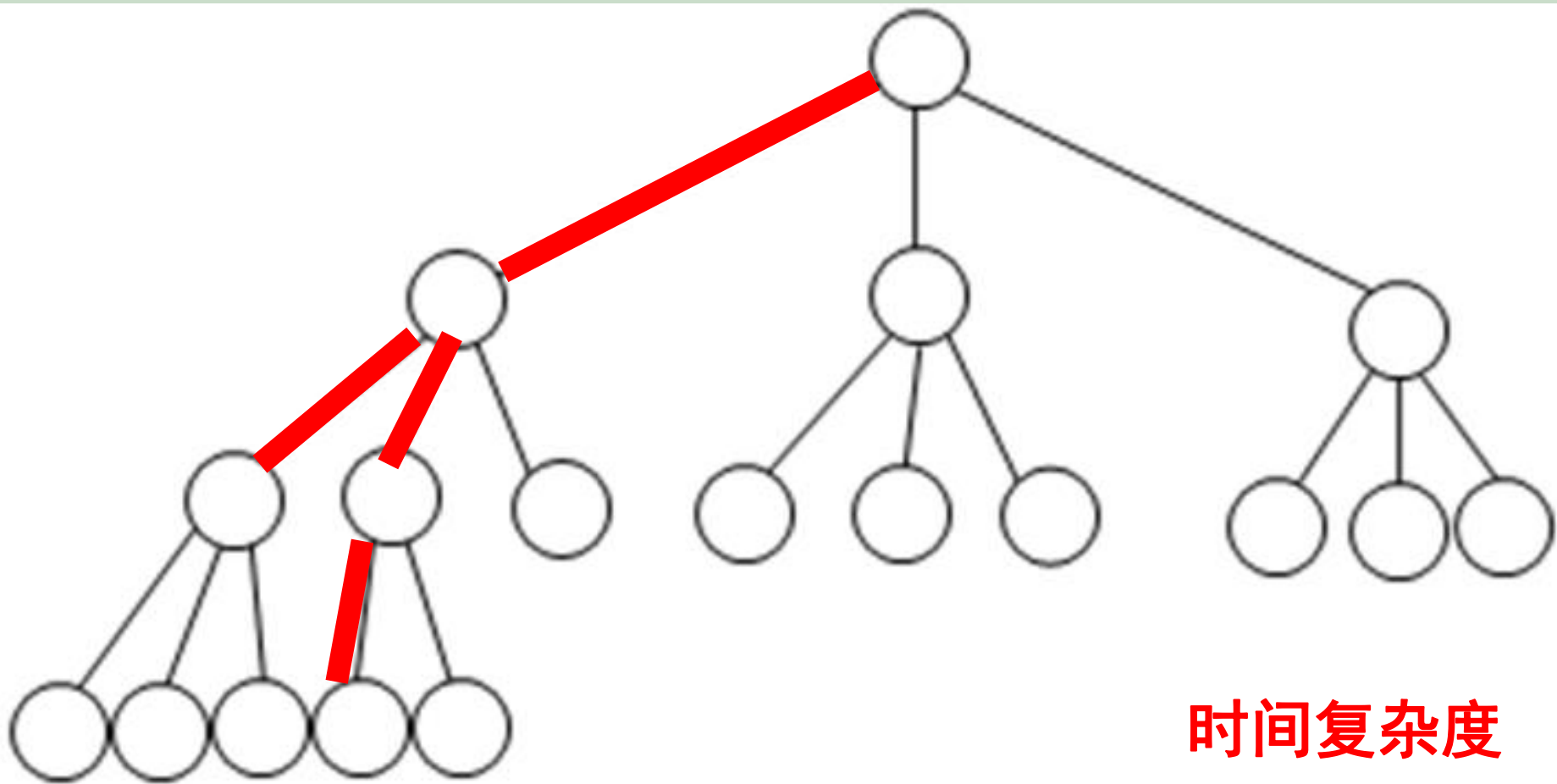
- 用约束函数剪去不满足约束的子树
- 用限界函数剪去得不到最优解的子树。
- 几个名词
 - ❧ 扩展结点:一个正在产生儿子的结点称为扩展结点
 - ❧ 活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点
 - ❧ 死结点:一个所有儿子已经产生的结点称做死结点
 - ❧ 搜索树: 搜索过程中动态形成的树



回溯法的基本思想

- 从根开始，以深度优先搜索的方式进行搜索。
- 根结点是活结点并且是当前的扩展结点。在搜索的过程中，当前的扩展结点向纵深方向移向一个新结点，判断该新结点是否满足隐约束。
- 如果满足，则新结点成为活结点，并且成为当前的扩展结点，继续深一层的搜索；
- 如果不满足，则换该新结点的兄弟结点（扩展结点的其它分支）继续搜索；
- 如果新结点没有兄弟结点，或其兄弟结点已全部搜索完毕，则扩展结点成为死结点，搜索回溯到其父结点处继续进行。
- 搜索过程直到找到问题的解或根结点变成死结点为止。

回溯法的基本思想



搜索树：扩展结点、活结点、死节点

n后问题

➤问题描述:

在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则，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 后问题等价于在 $n \times n$ 格的棋盘上放置 n 个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。

1. 找到一个解
2. 找到所有解（不考虑旋转对称）

1			Q					
2					Q			
3							Q	
4		Q						
5						Q		
6	Q							
7			Q					
8				Q				
	1	2	3	4	5	6	7	8

➤ 定义问题的解空间

- 解的形式: (x_1, x_2, \dots, x_n)
- x_i 的取值范围: $x_i = 1, 2, \dots, n$

➤ 组织解空间

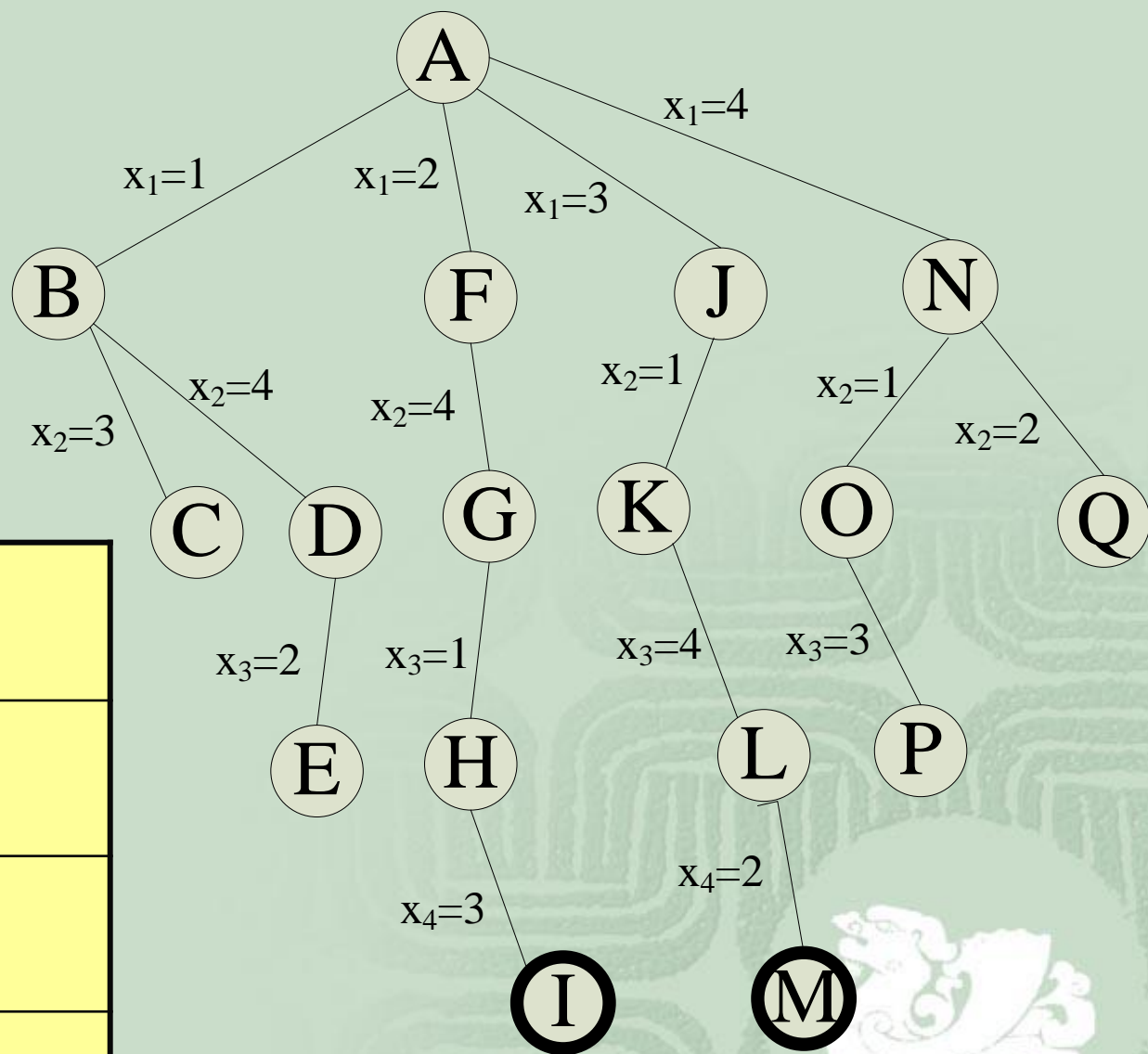
- 满 n 叉树, 树的深度为 n

➤ 搜索解空间

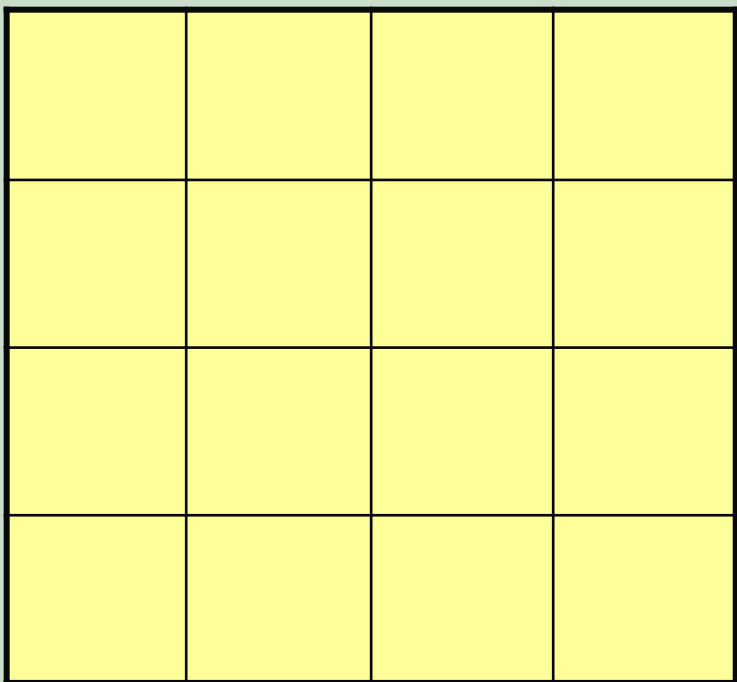
- 约束条件: (1) 不同行
(2) 不处于正反对角线上: $|i-j| \neq |x_i - x_j|$
- 限界条件: (×)

➤ 搜索过程 (以4皇后问题为例)





4皇后问题的搜索树



思考：进一步改进？

➤ 定义问题的解空间

➤ 解的形式： (x_1, x_2, \dots, x_n)

➤ x_i 的取值范围： $x_i \in S - \{x_1, x_2, \dots, x_{i-1}\}$

➤ 思考：该解空间比改进前少了多少可能的解？

➤ 组织解空间

➤ 排列树，树的深度为 n


➤ 搜索解空间

➤ 约束条件：(1) 不处于正反对角线上： $|i-j| \neq |x_i - x_j|$

➤ 限界条件：(×)

回溯法递归模板

```
void Bcktrack(int t)
{
    if(t>n)    //另一种写法是 if( solution(t) ), 即可能提前结束
        output(x);
    else
        for(int i=s(n,t);i<=e(n,t);i++) //只是比喻遍历
        {
            x[t]=d(i);    //只是比喻
            if( constraint(t) && bound(t) ) //只是比喻
                Bcktrack(t+1);
            ... //一般这里有还原操作, 因为x是全局的,
                不会在递归结束后还原
        }
}
```



回溯法非递归模板

```
void NBacktrack( )
```

```
{  int t=1;
```

```
  while(t>0)
```

```
  {  if( s(n,t)<=e(n,t) )    //只是表明当前是活节点
```

```
    for(int i=s(n,t);i<=e(n,t);i++)
```

```
    {  x[t]=d(i);
```

```
      if( constraint(t) && bound(t) )
```

```
        if(t>n) output(x); //另一种写法是 if(solution(t))
```

```
        else    t++; //此处隐含表明t++后会跳出循环
```

```
      } //或写成    else{ t++; break; }
```

```
    else t--; //此处省略了改变上层s(n,t)和e(n,t)的语句
```

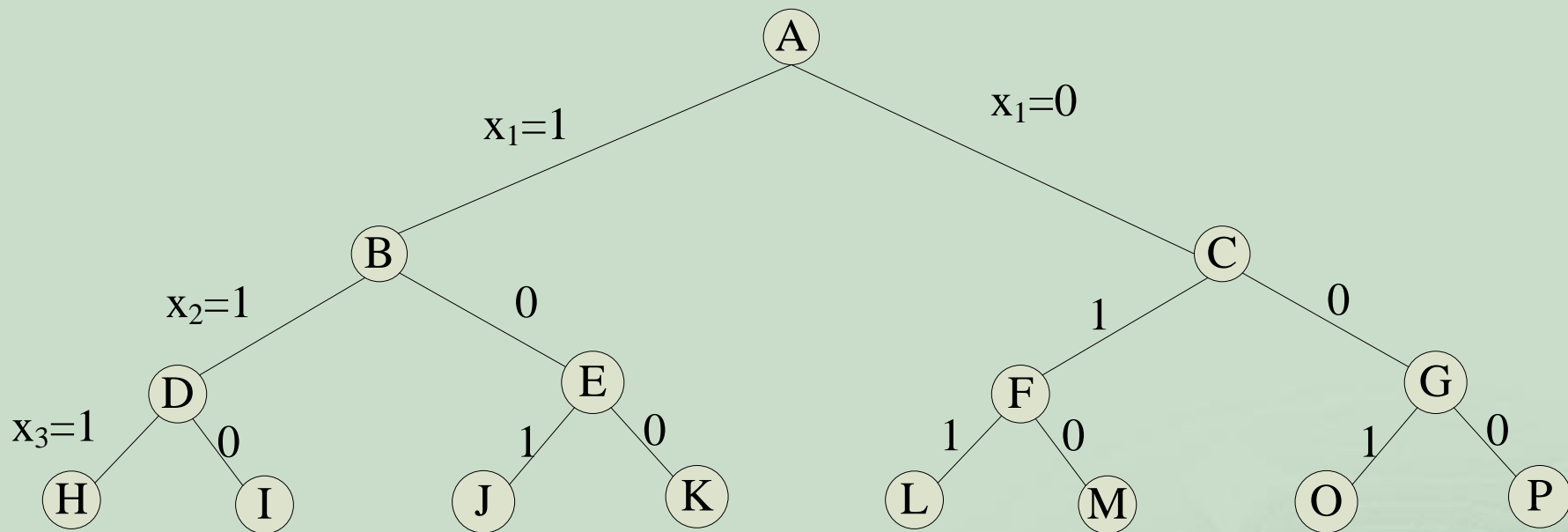
```
  }
```

```
}
```

5.3.2子集树

- 当所给的问题是从 n 个元素组成的集合 S 中找出满足某种性质的一个子集时，相应的解空间树称为子集树。
- 此类问题解的形式为 n 元组 (x_1, x_2, \dots, x_n) ，分量 $x_i (i=1, 2, \dots, n)$ 表示第 i 个元素是否在要找的子集中。
- x_i 的取值为0或1， $x_i=0$ 表示第 i 个元素不在要找的子集中； $x_i=1$ 表示第 i 个元素在要找的子集中。





n=3时的子集树

- 树的根结点：初始状态
- 中间结点：某种情况下的中间状态
- 叶子结点：结束状态
- 分支：从一个状态过渡到另一个状态的行为
- 从根结点到叶子结点的路径：一个可能的解（一个子集）
- 子集树的深度：等于问题的规模。

子集树递归模板(1)

```
void Bcktrack(int t)
```

```
{  if(t>n)
```

```
    output(x);
```

```
else
```

```
    for(int i=0;i<=1;i++)
```

```
    {  x[t]=i;
```

```
        if( constraint(t) && bound(t) )
```

```
            Bcktrack(t+1);
```

... **//一般这里有还原操作因为x是全局的，
不会在递归结束后还原**

```
    }
```

```
}
```



子集树递归模板(2)

```
void Backtrack (int t)
{
    if (t>n) output(x);
    if( constraint(t) && bound(t) ) //判断为1时能否进行扩展
    {
        做相关标识;
        Backtrack(t+1);
        做相关标识的反操作;
    }
    if( bound(t) ) //判断为0时能否进行扩展, 约束可以省略
    {
        做相关标识; //为0即不做选择, 故此语句常省略
        Backtrack(t+1);
        做相关标识的反操作; //不做选择, 故此语句常省略
    }
}
```

因为左限界可以反映在右分支或延迟到下一步, 所以为避免重复计算常常省略左限界, 或将**bound**约束提出来作为两种情况的先决条件。



示例1：0-1 背包问题

■ 问题描述

∞ 给定 n 种物品和一背包。物品 i 的重量是 w_i ，其价值为 v_i ，背包的容量为 W 。一个物品要么全部装入背包，要么全部不装入背包，不允许部分装入。装入背包的物品的总重量不超过背包的容量。问应如何选择装入背包的物品，使得装入背包中的物品总价值最大？

■ 定义问题的解空间

∞ 解的形式 (x_1, x_2, \dots, x_n) ，其中 $x_i=0$ 或 1

∞ $x_i=0$: 第 i 个物品不装入背包

∞ $x_i=1$: 第 i 个物品装入背包



► 确定接空间的组织结构

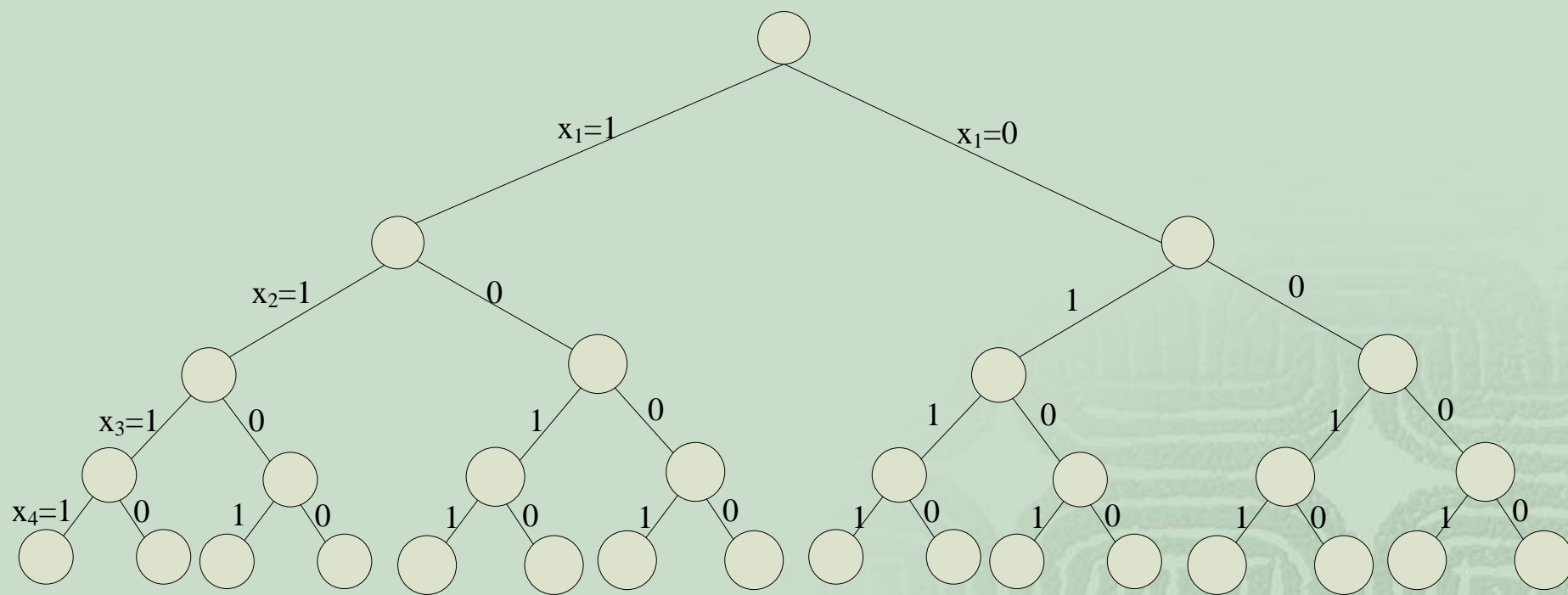


图5-11 $n=4$ 时的解空间树

► 搜索解空间



■ 约束条件

∞ $w_i \leq c'$ (c' 为包的剩余容量) 也就是: $\sum_{i=1}^n w_i x_i \leq W$

■ 限界条件: $cp+rp>bestp$ 其中

∞ cp : 当前已装入背包的物品的总价值

∞ rp : 剩余不知道是否装入的物品的总价值

∞ $bestp$: 当前已经找到的最优解的价值

■ 搜索过程

∞ 以 $n=4$, $W=7$, $w=(3,5,2,1)$, $v=(9,10,7,4)$ 为例
展示搜索过程

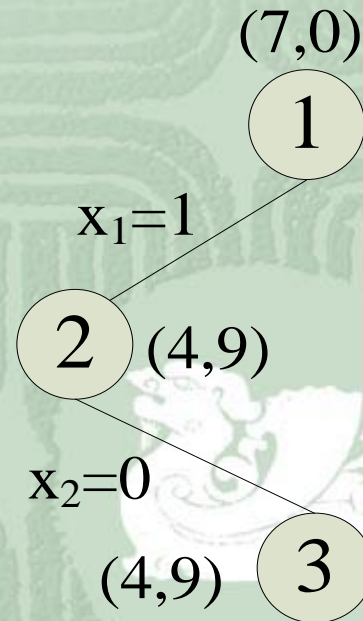
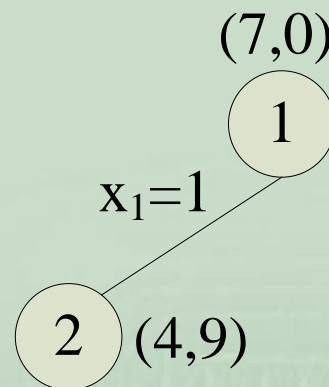
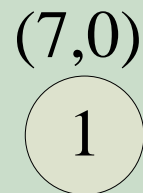
$W=7, \quad w=(3,5,2,1), \quad v=(9,10,7,4)$

➤根节点是活节点并且是当前的扩展结点

➤第一个物品的重量为**3**， **$3 < 7$** ,满足约束条件

➤第二个物品的重量为**5**， **$5 > 4$** ，不满足约束条件

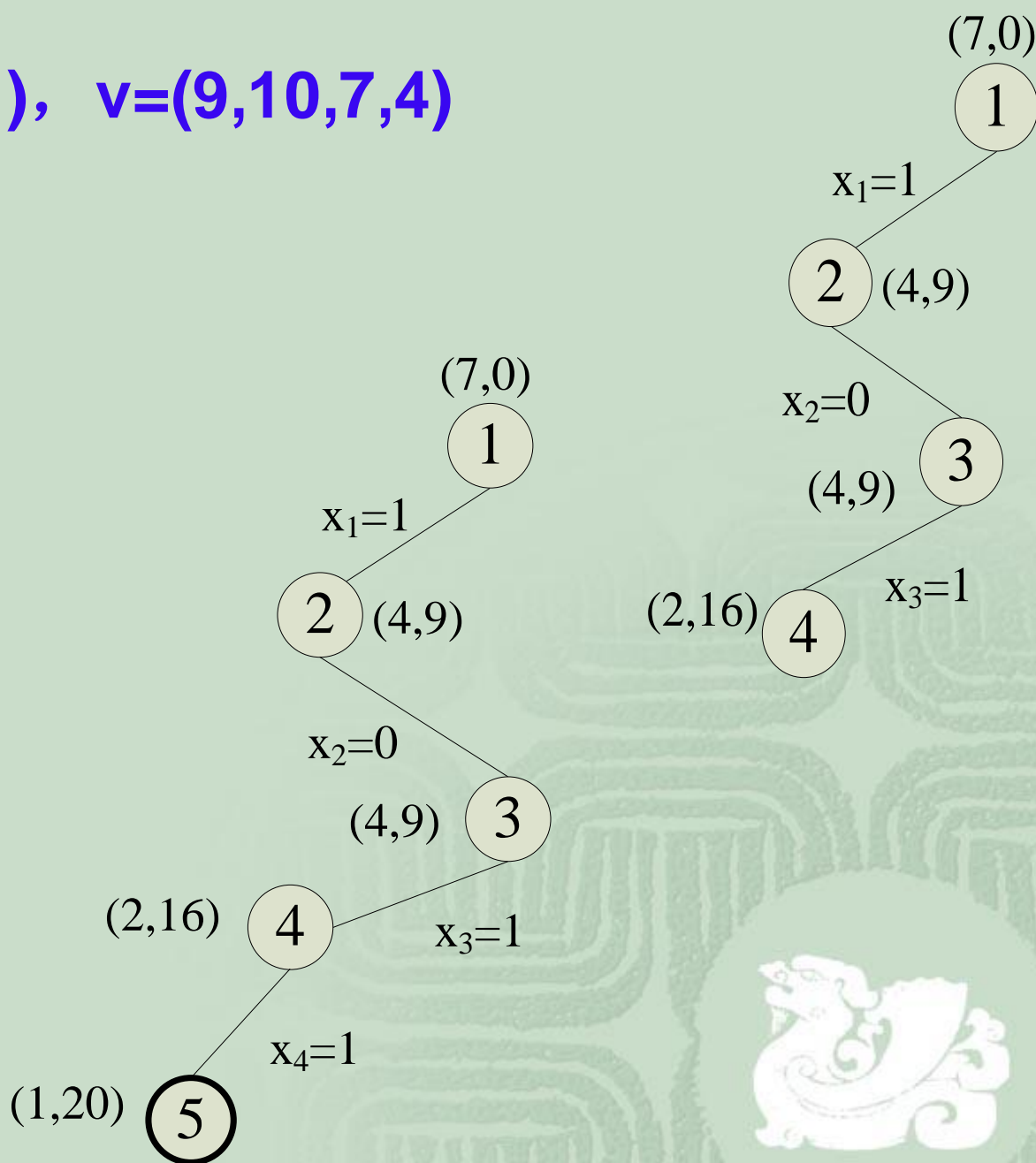
➤ **$cp=9$** ， **$rp=11$** ， **$bestp=0$** ， **$cp+rp > bestp$** ,满足限界条件。



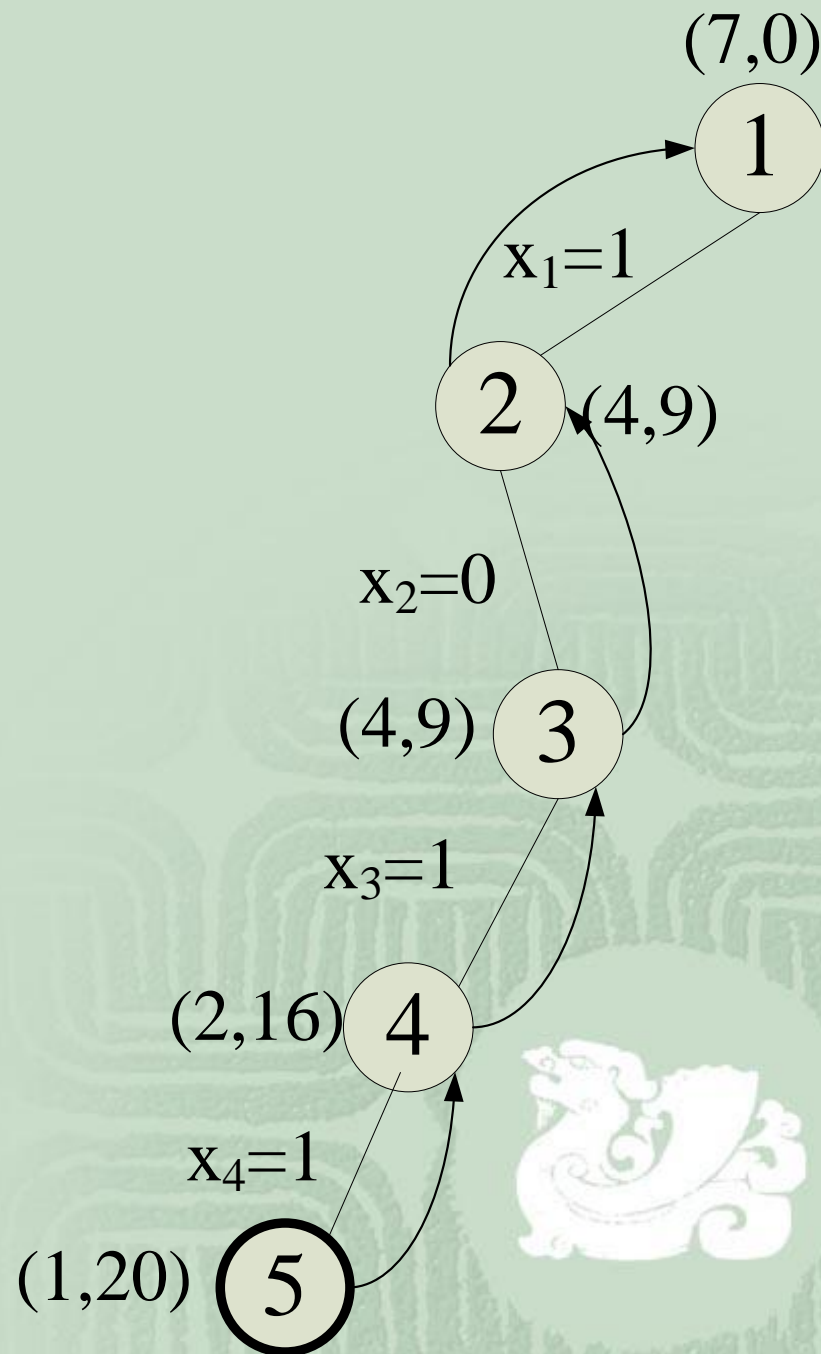
$W=7$, $w=(3,5,2,1)$, $v=(9,10,7,4)$

➤第3个物品的重量是2，背包的剩余容量为4，满足约束条件

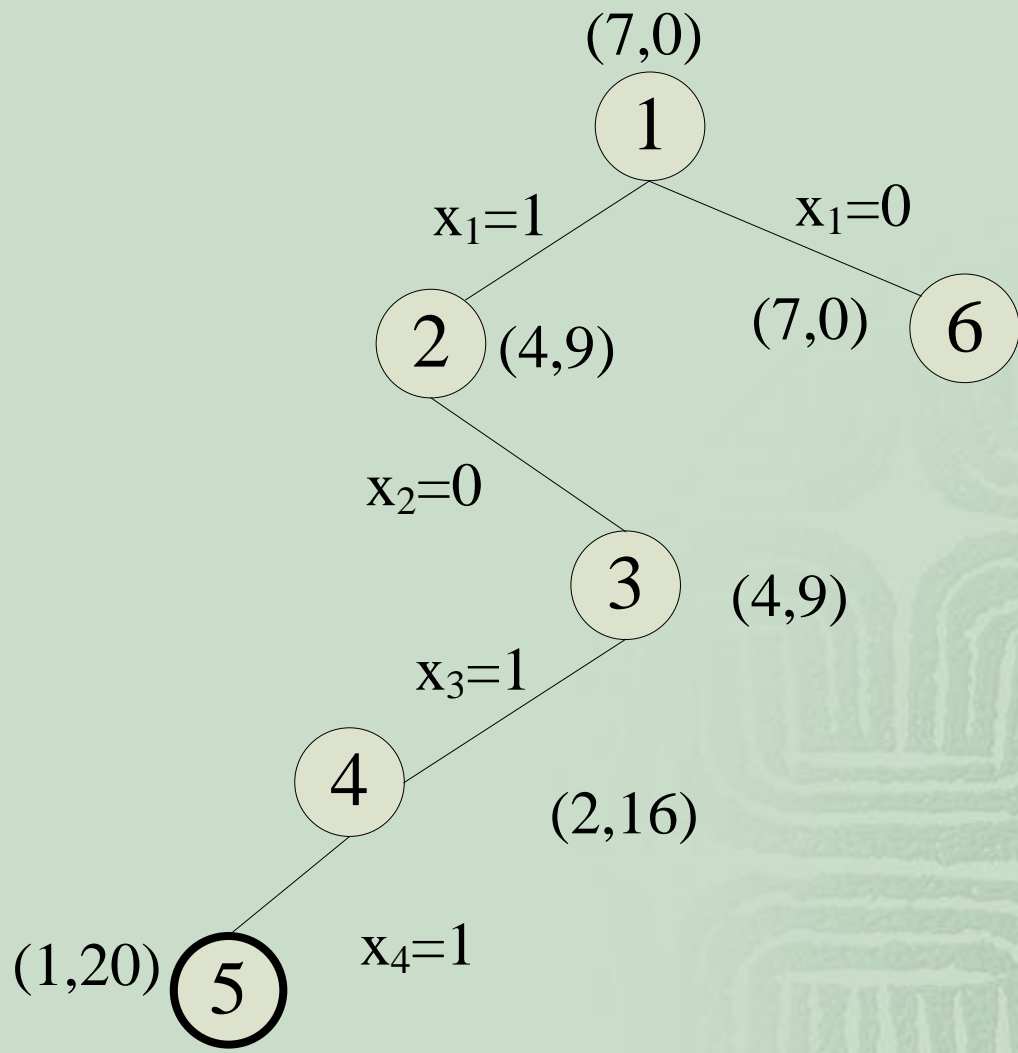
➤第4个物品的重量为1，背包的剩余容量为2，满足约束条件。现在已经到达叶子，找到当前最优解，**bestp=20**



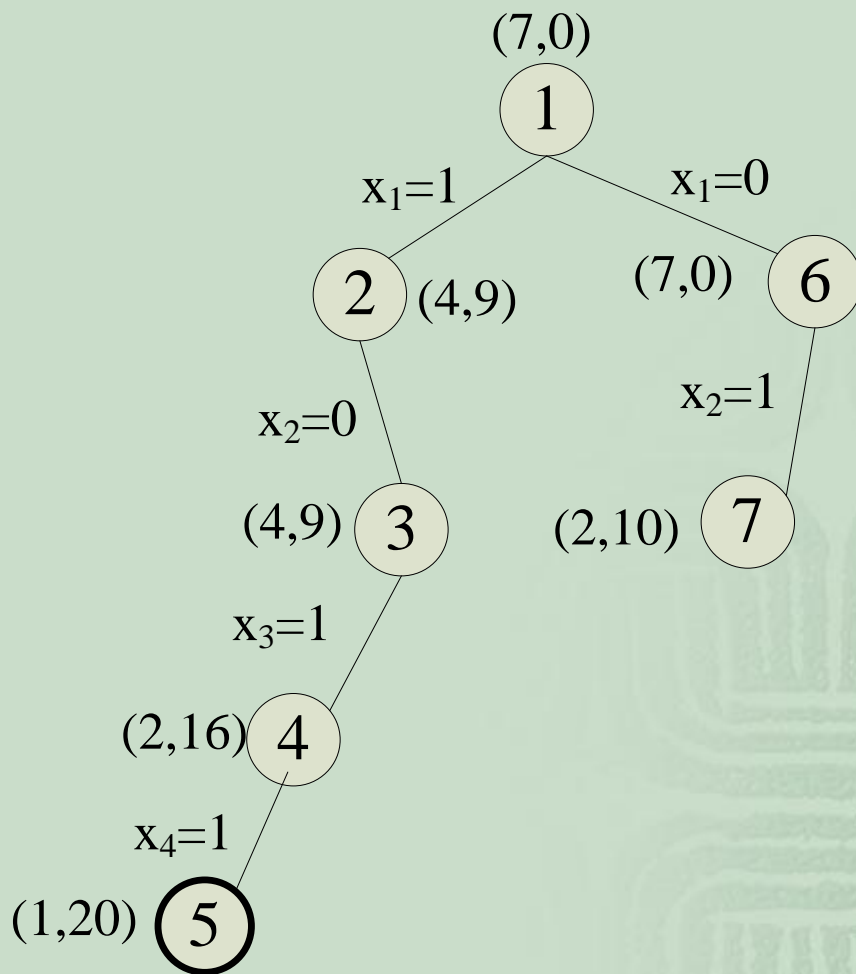
- 此时要回溯到离结点5最近的活结点4，结点4再次成为扩展结点
- 限界条件 $cp=16$ ， $rp=0$ ， $bestp=20$ $cp+rp < bestp$ ，限界条件不满足；
- 回溯到最近的活结点3，结点3再次成为扩展结点
- 限界条件是否满足， $cp=9$ ， $rp=4$ ， $bestp=20$ ， $cp+rp < bestp$ ，限界条件不满足
- 回溯到最近的活结点2，继续回溯到结点1



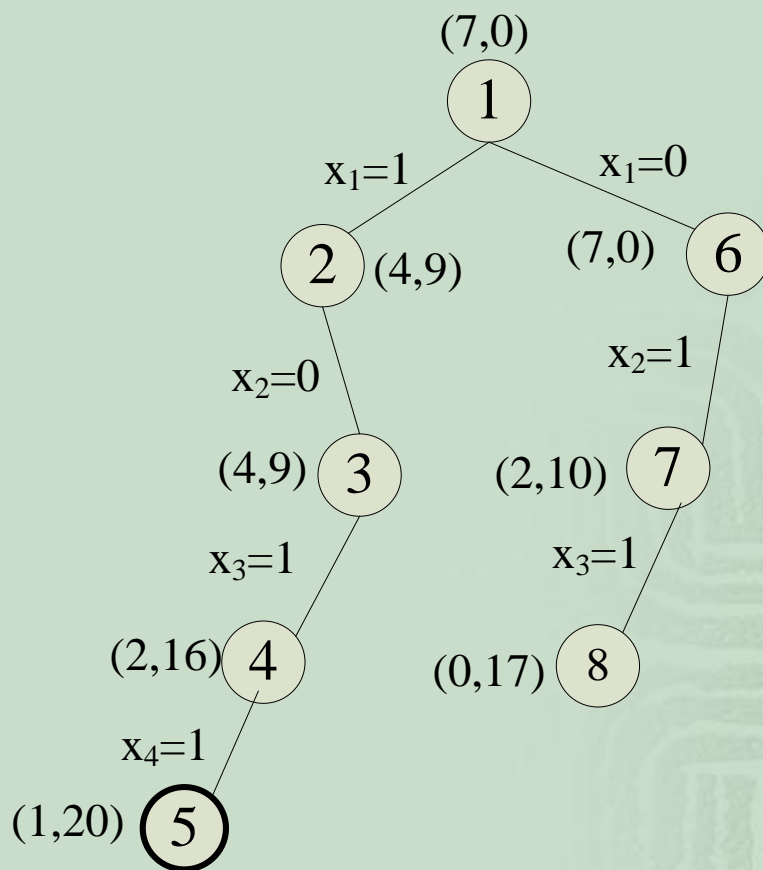
- 扩展结点1沿着右分支继续扩展， $cp=0$ ， $rp=21$ ， $bestp=20$ ， $cp+rp>bestp$ ，限界条件满足



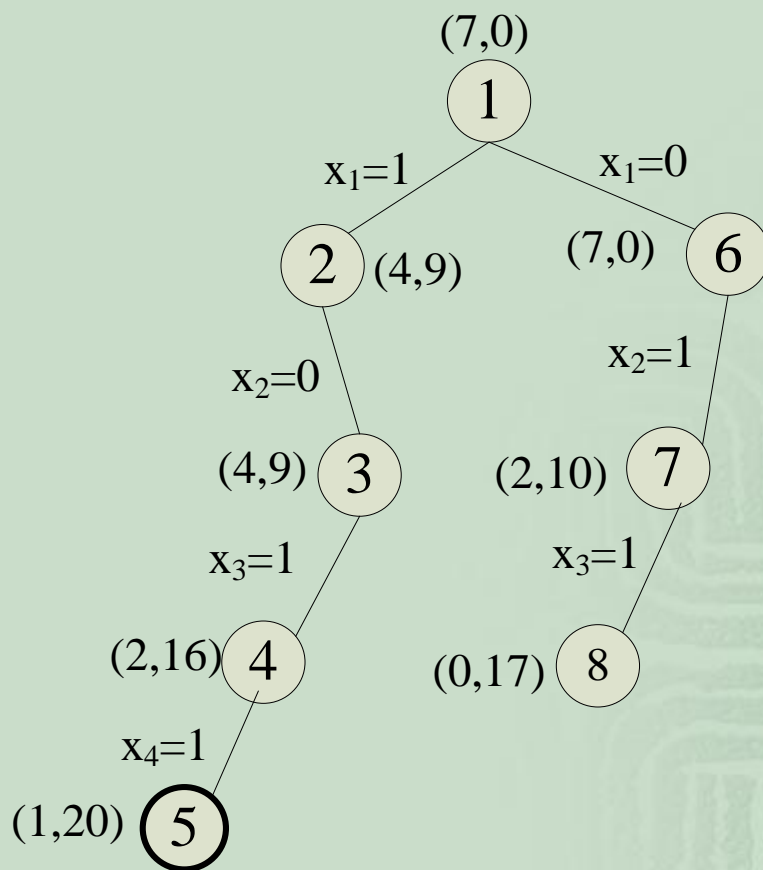
- 扩展结点**6**沿着左分支继续扩展，第二个物品的重量为5， $5 < 7$ ，满足约束条件



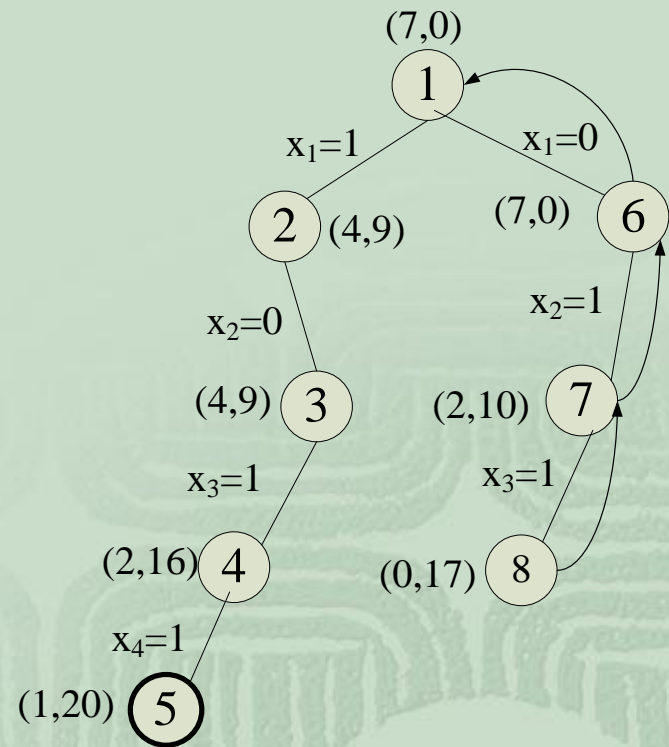
- 扩展结点7沿着左分支继续扩展，判断约束条件，当前背包剩余容量为2，第3个物品的重量为2，满足约束条件

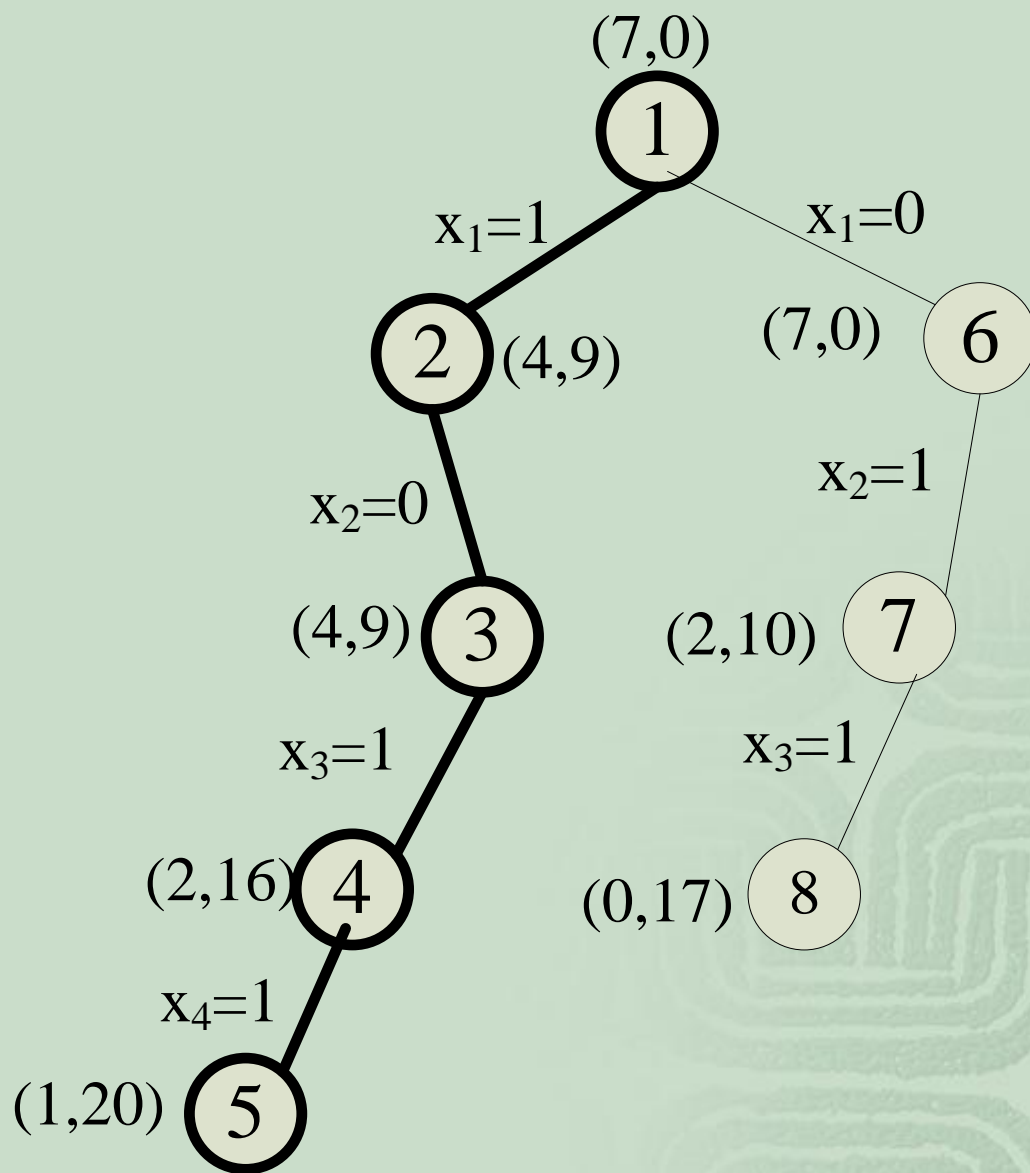


- 扩展结点7沿着左分支继续扩展，判断约束条件，当前背包剩余容量为2，第3个物品的重量为2，满足约束条件



- 扩展结点8沿着左分支继续扩展，当前背包剩余容量为0，第4个物品的重量为1， $0 < 1$ ，不满足约束条件
- 沿着扩展结点8的右分支进行扩展，判断限界条件， $cp=17$ ， $rp=0$ ， $bestp=20$ ， $cp+rp < bestp$ ，不满足限界条件
- 回溯到最近的活结点7，扩展结点7沿着右分支继续扩展，判断限界条件，当前 $cp=10$ ， $rp=4$ ， $bestp=20$ ， $cp+rp < bestp$ ，限界条件不满足
- 回溯到活结点6，扩展结点6沿着右分支继续扩展，当前 $cp=0$ ， $rp=11$ ， $bestp=20$ ， $cp+rp < bestp$ ，限界条件不满足
- 回溯到活结点1，结点1又成为扩展结点。结点1的分支全部搜索完毕，结点1成为死结点，搜索结束。





示例0-1背包问题的搜索树



■ 思考：能否通过更改限界来提高搜索效率？

∞ 在限界条件 $cp+rp>bestp$ 中， rp 表示第 $t+1$ 个物品到第 n 个物品的总价值。事实上，背包的剩余容量不一定能够容纳从第 $t+1$ 个物品到第 n 个物品的全部物品，那么剩余容量所能容纳的从第 $t+1$ 个物品到第 n 个物品的最大价值（用 brp 表示）肯定小于或等于 rp ，用 brp 取代 rp

∞ 限界条件可改写为： $cp+brp>bestp$

∞ 对 $bestp$ 做调整能否提高搜索效率？

∞ 练习：参考教材，写一份对比分析报告。要求画出搜索过程，并说明与改进前的搜索过程对比有哪些改进。



■ 关键代码分析

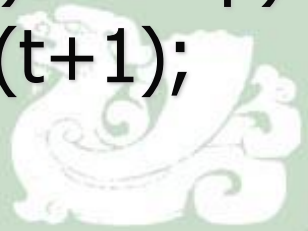
```
int Bound(int i)           //计算上界
{   int cleft=c-cw;        //剩余容量
    int b=cp;
    //以物品单位重量价值递减序装入物品
    while(i<=n && w[i]<=cleft)
    {   cleft = cleft - w[i];
        b = b+p[i]; i++;
    }
    if(i<=n) //装满背包
        b = b+p[i]/w[i]*cleft;
    return b;
}
```




```
void Backtrack(int t)
{  if(t>n) //到达叶子结点
    {  for(int j=1;j<=n;j++)
        bestx[j]=x[j];
        bestp=cp;
        return;
    }
```

```
//搜索左子树
if(cw+w[t]<=c)
{  x[t]=1;
    cw+=w[t];
    cp+=p[t];
```

```
        Backtrack(t+1);
        cw-=w[t];
        cp-=p[t];
        x[t]=0;
    }
    //搜索右子树
    if(Bound(t+1)>bestp)
    {  Backtrack(t+1);
    }
}
```



■ 复杂性分析

- ∞ 判断约束函数需 $O(1)$ ，在最坏情况下有 2^n-1 个孩子，约束函数耗时最坏为 $O(2^n)$ 。
- ∞ 计算上界限函数需要 $O(n)$ 时间，在最坏情况下有 2^n-1 个孩子需要计算上界，限界函数耗时最坏为 $O(n2^n)$ 。
- ∞ 0-1背包问题的回溯算法所需的计算时间为 $O(2^n)+O(n2^n)=O(n2^n)$ 。
- ∞ 但实际上远远小于这个数量级，通过实验测试。



思考

- 0-1背包问题用非递归算法如何实现？



示例2：最大团问题

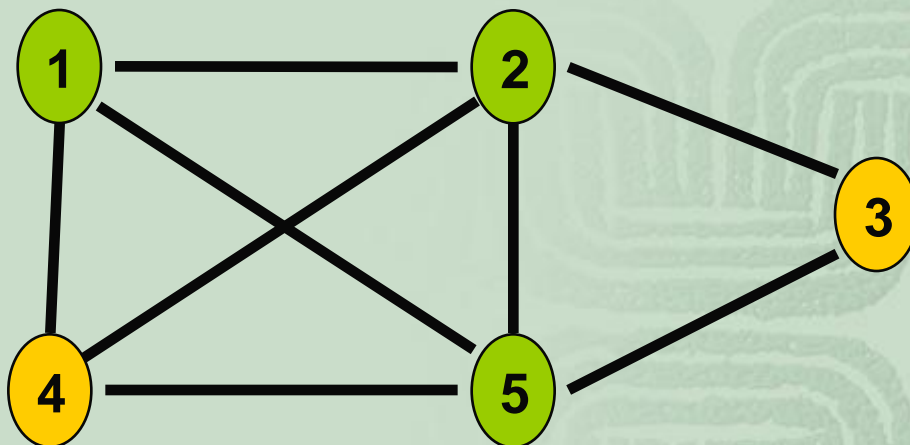
➤ 基本概念

➤ **完全子图**：给定无向图 $G=(V, E)$ 。如果 $U \subseteq V$ ，且对任意 $u, v \in U$ 有 $(u, v) \in E$ ，则称 U 是 G 的完全子图。

➤ **团**： G 的完全子图 U 是 G 的团当且仅当 U 不包含在 G 的更大的完全子图中。

➤ **最大团**：是指 G 中所含顶点数最多的团。

➤ **问题描述**：最大团问题要求找出给定无向图 G 的最大团



■ 问题分析

∞ 最大团问题就是要求找出无向图 $G=(V, E)$ 的 n 个顶点集合 $\{1,2,3,...,n\}$ 的一个子集，这个子集中的任意两个顶点在无向图 G 中都有边相连，且包含顶点个数是最多的。

■ 解空间：

∞ 子集树

■ 约束条件：

∞ 顶点 i 与已选入的顶点集合中每一个顶点都有边相连。

```
Bool Place(int t)
{
    Bool OK=true;
    for (int j=1; j<t; j++)
        // 顶点t与顶点j不相连
        if (x[j]==1 && a[t][j]==0)
        {
            OK=false;
            break;
        }
    return OK;
}
```



➤ 限界条件

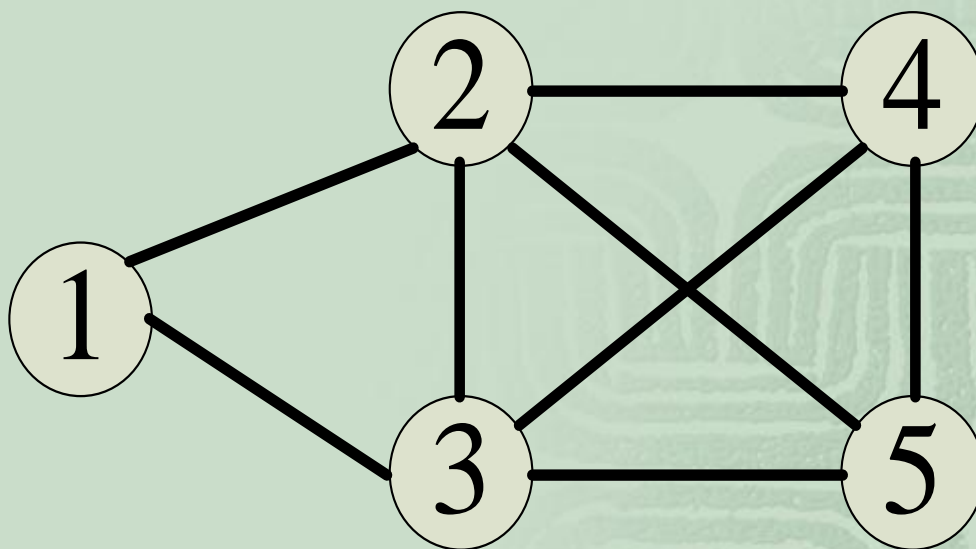
➤ $cn + rn > bestn$

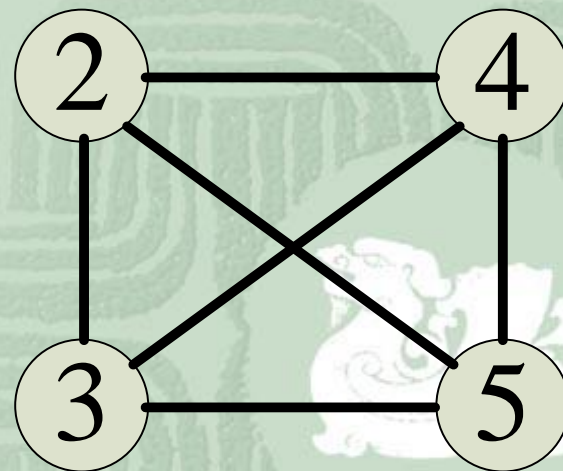
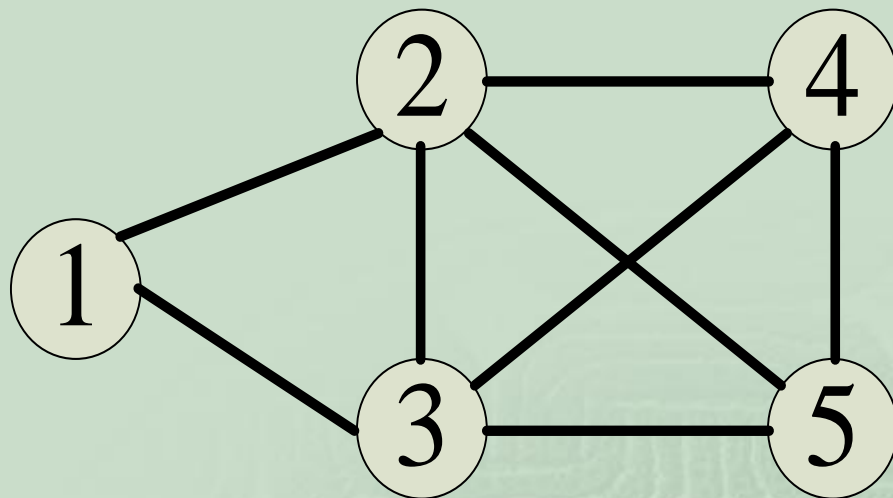
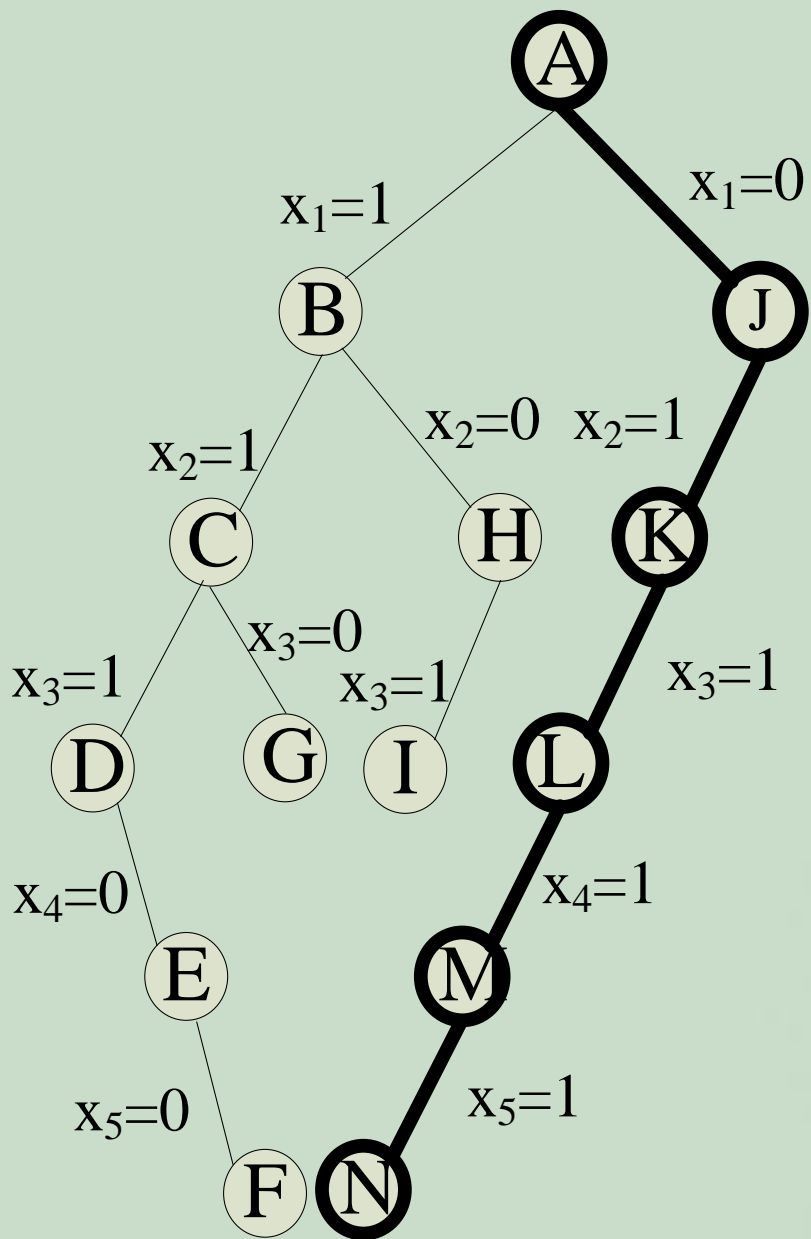
➤ **cn**: 当前已包含在顶点集中的顶点个数

➤ **rn**: 剩余顶点个数;

➤ **bestn**: 当前已找到的最优解包含的顶点个数

➤ 搜索过程（以图5-14为例）






```
void Backtrack(int t)
{ if (t > n)
    { for(int i=1;i<=n;i++)
        bestx[i]=x[i];
      bestn=cn;
      return;
    }
```

// 进入左子树

```
if (Place(t))
{ x[t]=1;
  cn++;
  Backtrack(t+1);
  cn--;
}
```

关键代码分析

// 进入右子树

```
if (cn+n-t>bestn)
{ x[t]=0;
  Backtrack(t+1);
}
```



■ 算法分析

- ❧ 判断约束函数需 $O(n)$ ，在最坏情况下有 2^n-1 个左孩子，耗时最坏为 $O(n2^n)$ 。
- ❧ 判断限界函数需要 $O(1)$ 时间，在最坏情况下有 2^n-1 个右孩子结点需要判断限界函数，耗时最坏为 $O(2^n)$ 。
- ❧ 最大团问题的回溯算法所需的计算时间为 $O(2^n)+O(n2^n)= O(n2^n)$ 。



5.3.3排列树

- 当所给的问题是从 n 个元素的排列中找出满足某种性质的一个排列时，相应的解空间树称为排列树。
- 此类问题解的形式为 n 元组 (x_1, x_2, \dots, x_n) ，分量 $x_i (i=1, 2, \dots, n)$ 表示第 i 个位置的元素是 x_i 。 n 个元素组成的集合为 $S=\{1, 2, \dots, n\}$ ， $x_i \in S - \{x_1, x_2, \dots, x_{i-1}\}$ ， $i=1, 2, \dots, n$ 。
- $n=3$ 时的排列树如图5-17所示：



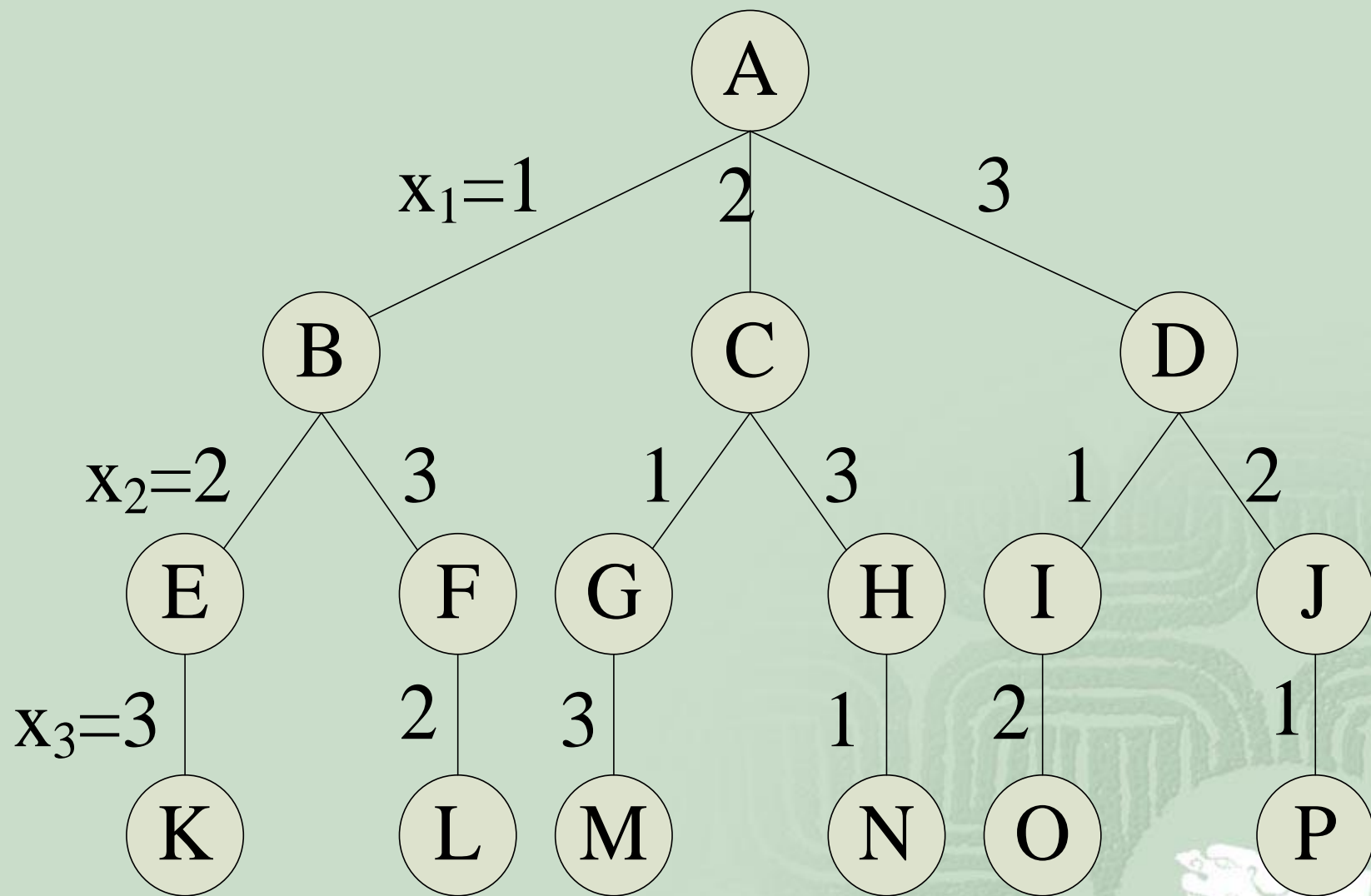


图5-17 $n=3$ 的排列树

算法设计

```
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]);
    }
}
```



■ 全排列算法的三种实现方法

∞字典法

∞递归法

∞经典算法

<http://blog.chinaunix.net/uid-22663647-id-1771845.html>

<http://blog.chinaunix.net/uid-22663647-id-1771842.html>

<http://blog.chinaunix.net/uid-22663647-id-1771843.html>

<http://blog.chinaunix.net/uid-22663647-id-1771844.html>



示例1：批处理作业调度

➤ 问题描述：

给定 n 个作业的集合 $\{J_1, J_2, \dots, J_n\}$ 。每个作业必须先由机器1处理，然后由机器2处理。作业 J_i 需要机器 j 的处理时间为 t_{ji} 。对于一个确定的作业调度，设 F_{ji} 是作业 i 在机器 j 上完成处理的时间。所有作业在机器2上完成处理的时间和称为该作业调度的完成时间和。

批处理作业调度问题要求对于给定的 n 个作业，制定最佳作业调度方案，使其完成时间和达到最小。

■ 符号意义

⌘ t_{ji} : 作业 i 在 j 机器上的处理时间

⌘ F_{ji} : 作业 i 在 j 机器上处理完成的时间

⌘ $f = \sum F_{2i}$: 作业调度的完成时间和

■ 目标：要求找出使得 f 最小的调度方案



n=3的实例如下:

t_{ji}	机器1	机器2
J_1	2	1
J_2	3	1
J_3	2	3

调度1, 2, 3

$$F_{11}=2 \quad F_{21}=3$$

$$F_{12}=5 \quad F_{22}=5+1=6$$

$$F_{13}=7 \quad F_{23}=7+3=10$$

$$f=F_{21}+F_{22}+F_{23}=19$$

调度1, 3, 2

$$F_{11}=2 \quad F_{21}=3$$

$$F_{12}=4 \quad F_{22}=4+3=7$$

$$F_{13}=7 \quad F_{23}=7+1=8$$

$$f=F_{21}+F_{22}+F_{23}=18$$



■ 定义问题的解空间

∞ 解的形式

■ (x_1, x_2, \dots, x_n)

∞ x_i 取值范围

■ 令 $S = \{1, 2, \dots, n\}$, 则 $x_i \in S - \{x_1, x_2, \dots, x_{i-1}\}$

■ 解空间的组织结构

∞ 排列树, 深度为问题的规模

■ 搜索解空间

∞ 约束条件 (×)

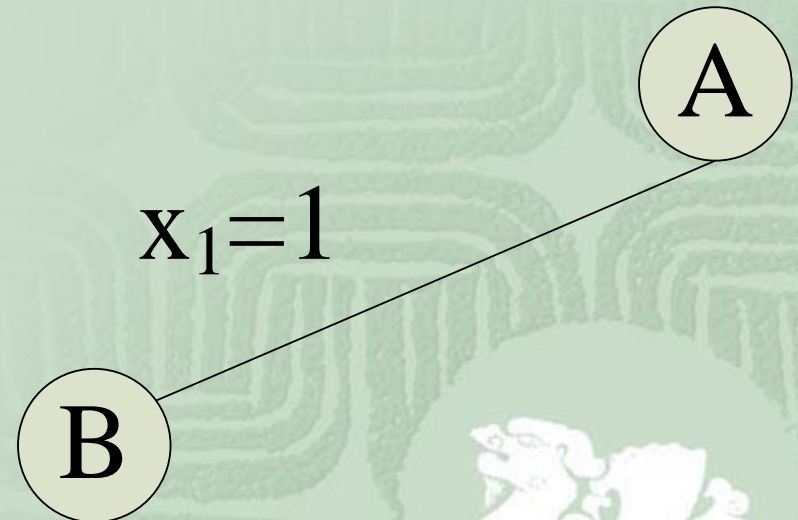
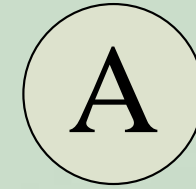
∞ 限界条件:

■ 当前部分排列的作业在机器2上的完成时间和 $f <$ 当前找到的最排列所需要的完成时间和 $bestf$

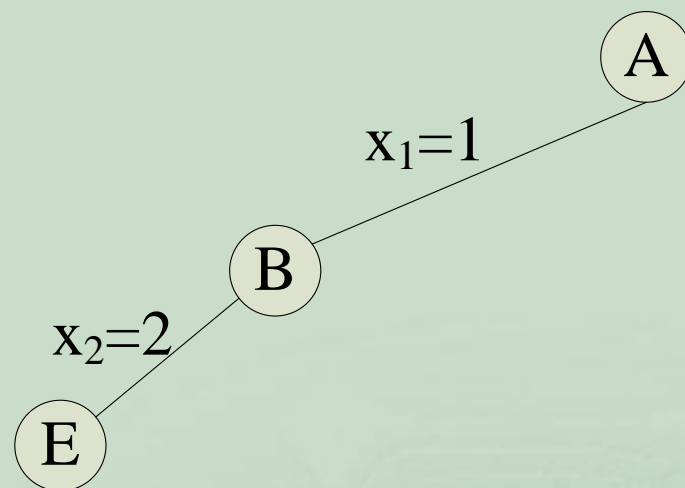


搜索过程（以上述实例为例）

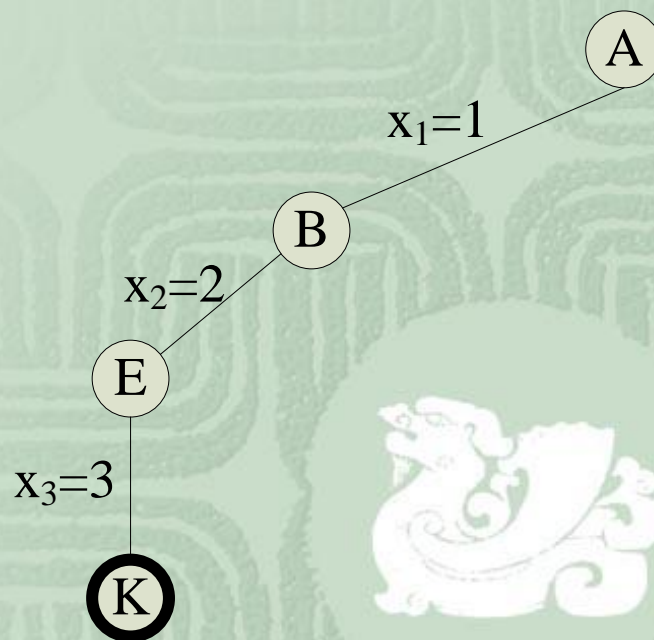
- 从根结点A开始，结点A成为活结点，并且是当前的扩展结点。
- 扩展结点A沿着 $x_1=1$ 的分支扩展， $F_{11}=2$ ， $F_{21}=3$ ，故 $cf=3$ ， $bestf=+\infty$ ， $cf < bestf$ ，限界条件满足。



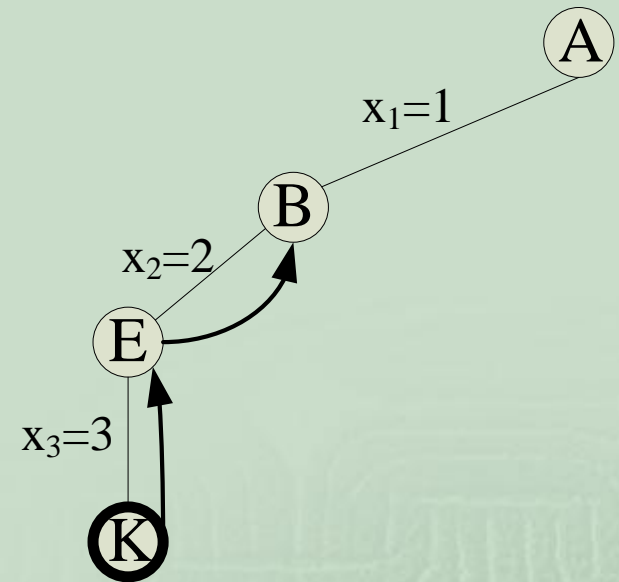
- 扩展结点B沿着 $x_2=2$ 的分支扩展, $F_{12}=5$, $F_{22}=6$, 故 $cf=F_{21}+F_{22}=9$, $bestf=+\infty$, $cf<bestf$, 限界条件满足。



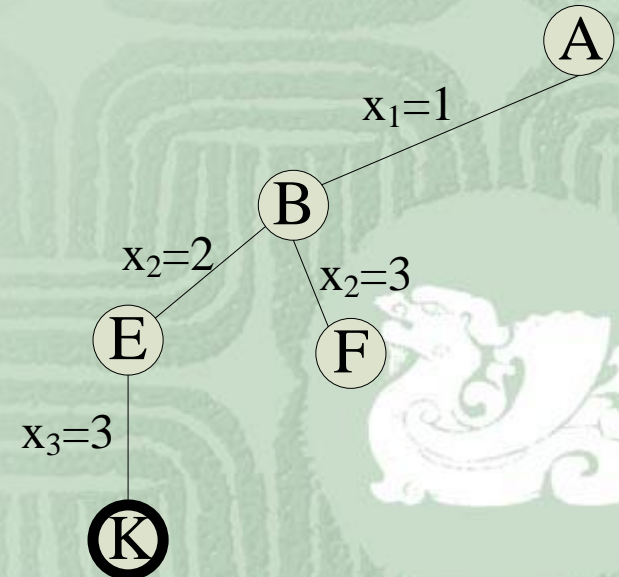
- 扩展结点E沿着 $x_3=3$ 的分支扩展, $F_{13}=7$, $F_{23}=10$, 故 $cf=F_{21}+F_{22}+F_{23}=19$, $bestf=+\infty$, $cf<bestf$, 限界条件满足, 扩展生成的结点K是叶子结点。此时, 找到当前最优的一种调度方案 (1,2,3), 同时修改 $bestf=19$



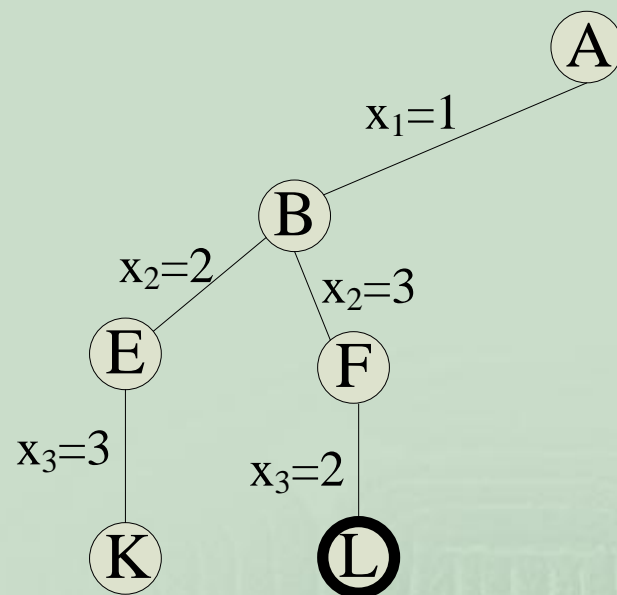
- 叶子结点K不具备扩展能力，开始回溯到活结点E。结点E只有一个分支，且已搜索完毕，因此结点E成为死结点，继续回溯到活结点B。



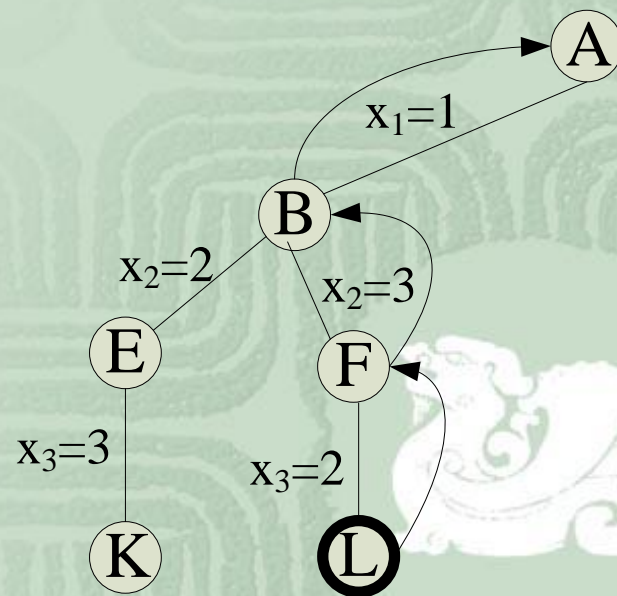
- 扩展结点B沿着 $x_2=3$ 的分支扩展， $cf=10$ ， $bestf=19$ ， $cf < bestf$ ，限界条件满足。



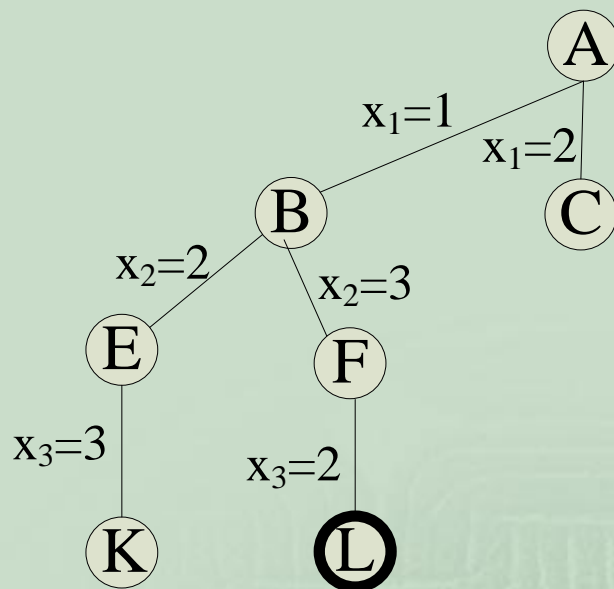
- 扩展结点F沿着 $x_3=2$ 的分支扩展， $cf=18$ ， $bestf=19$ ， $cf < bestf$ ，限界条件满足，扩展生成的结点L是叶子结点。此时，找到比先前更优的一种调度方案（1,3,2），修改 $bestf=18$ 。



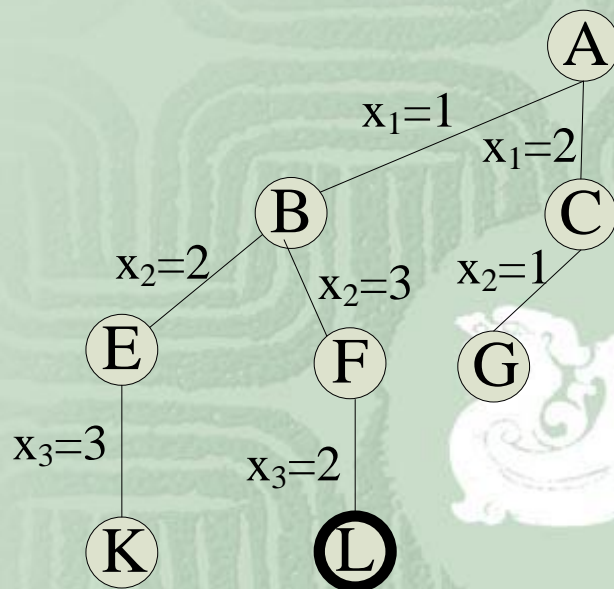
- 从叶子结点L开始回溯到活结点F。结点F的一个分支已搜索完毕，结点F成为死结点，回溯到活结点B。结点B的两个分支已搜索完毕，回溯到活结点A。



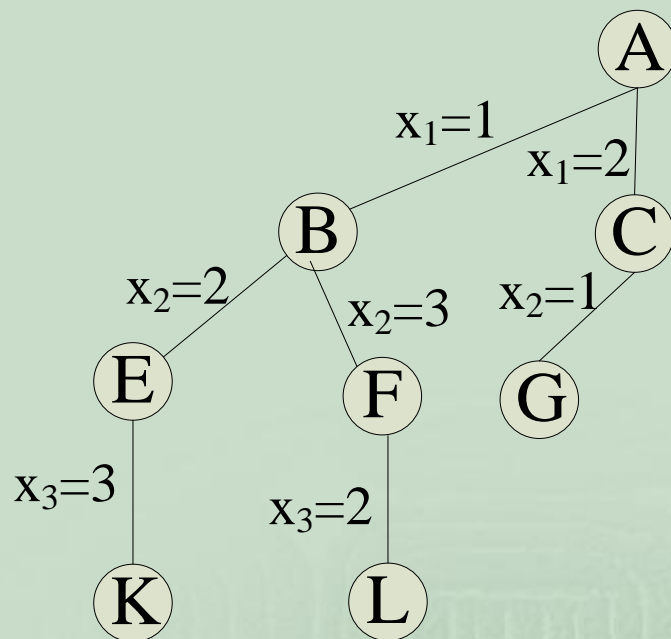
- 扩展结点A沿着 $x_1=2$ 的分支扩展， $cf=4$ ， $bestf=18$ ， $cf < bestf$ ，限界条件满足。



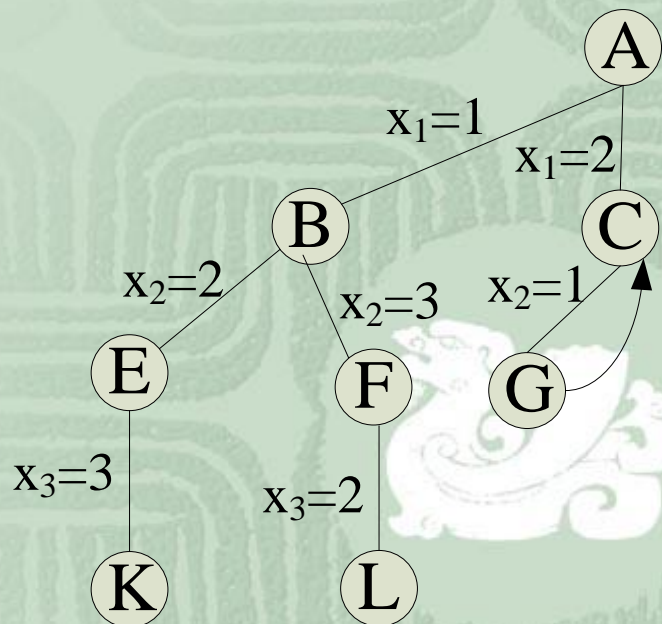
- 扩展结点C沿着 $x_2=1$ 的分支扩展， $cf=10$ ， $bestf=18$ ， $cf < bestf$ ，限界条件满足，扩展生成的结点G成为活结点，并且成为当前的扩展结点。



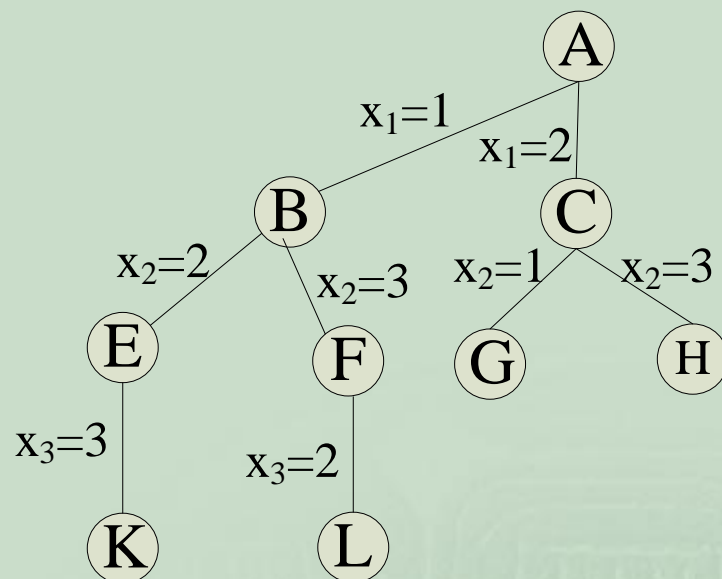
- 扩展结点G沿着 $x_3=3$ 的分支扩展， $cf=20$ ， $bestf=18$ ， $cf > bestf$ ，限界条件不满足，扩展生成的结点被剪掉



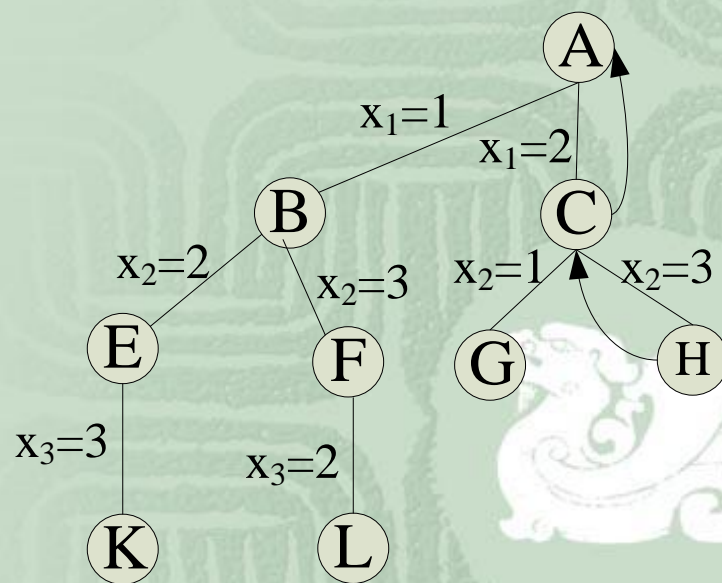
- 结点G的一个分支搜索完毕，结点G成为死结点，回溯到活结点C。



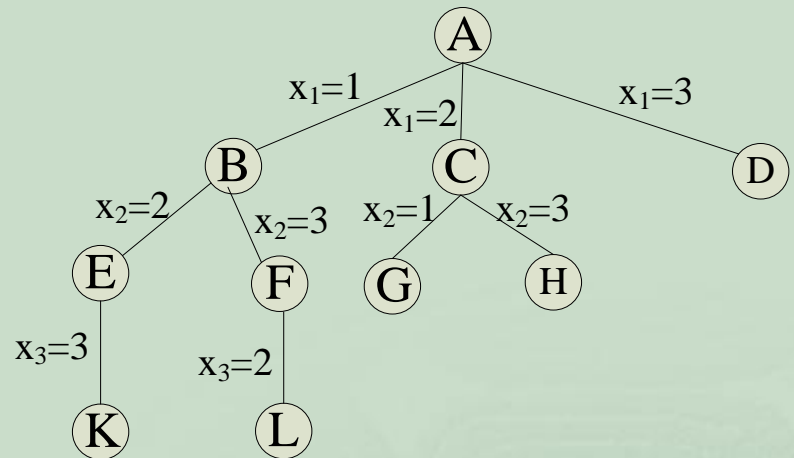
- 扩展结点C沿着 $x_2=3$ 的分支扩展， $cf=12$ ， $bestf=18$ ， $cf < bestf$ ，限界条件满足。



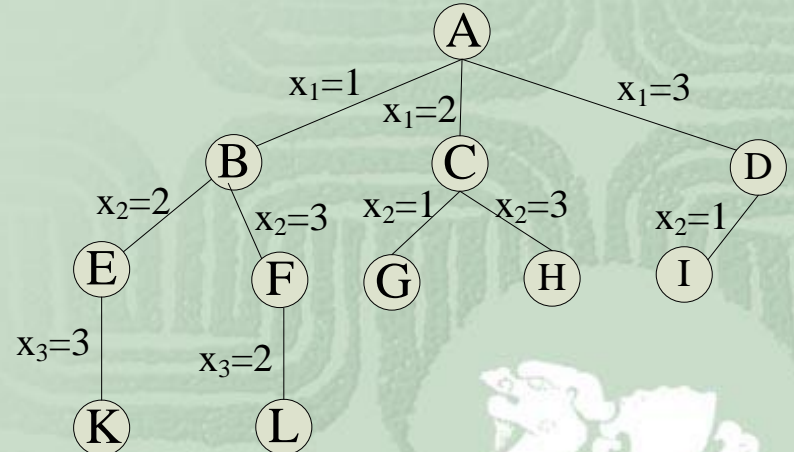
- 扩展结点H沿着 $x_3=1$ 的分支扩展， $cf=21$ ， $bestf=18$ ， $cf > bestf$ ，限界条件不满足，扩展生成的结点被剪掉。结点H的一个分支搜索完毕，开始回溯到活结点C。此时，结点C的两个分支已搜索完毕，继续回溯到活结点A。



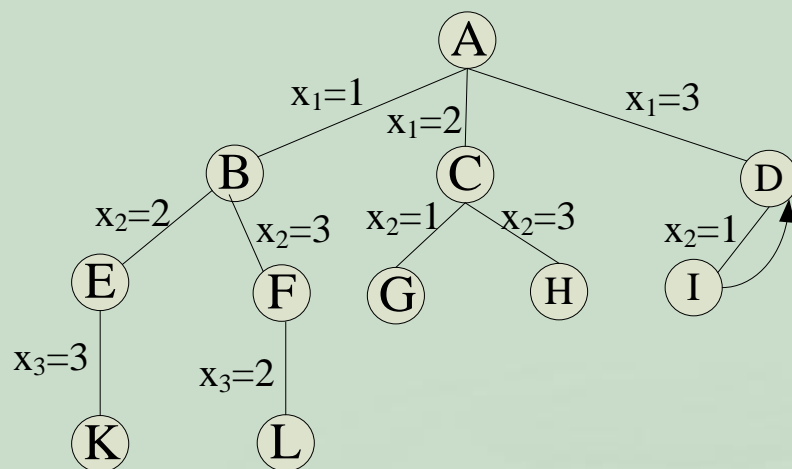
- 扩展结点A沿着 $x_1=3$ 的分支扩展， $cf=5$ ， $bestf=18$ ， $cf < bestf$ ，限界条件满足



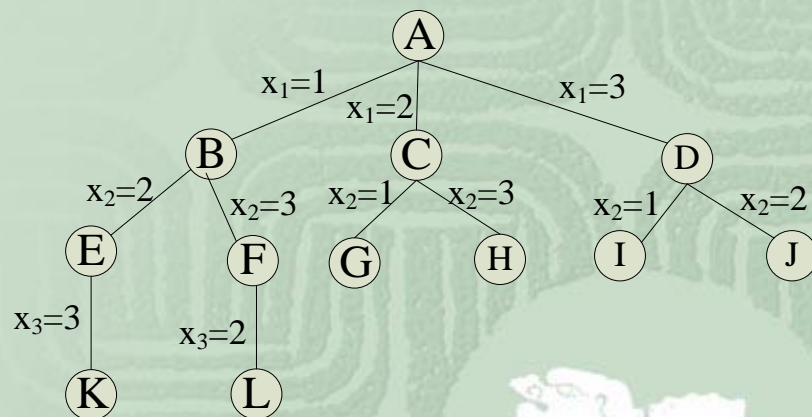
- 扩展结点D沿着 $x_2=1$ 的分支扩展， $cf=11$ ， $bestf=18$ ， $cf < bestf$ ，限界条件满足，扩展生成的结点I成为活结点



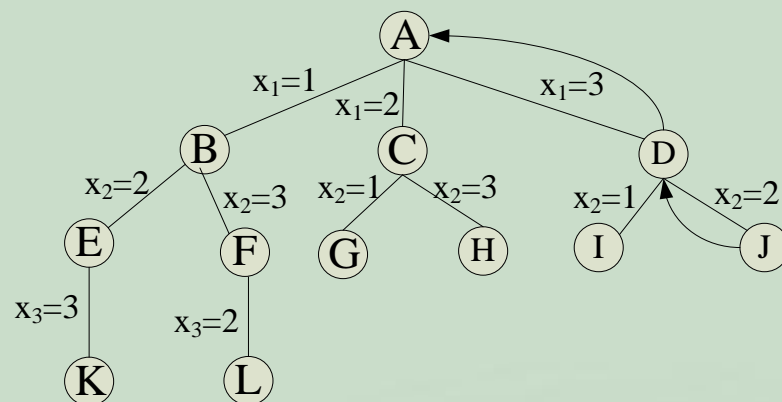
- 扩展结点I沿着 $x_3=2$ 的分支扩展， $cf=19$ ， $bestf=18$ ， $cf > bestf$ ，限界条件不满足，扩展生成的结点被剪掉，开始回溯到活结点D。



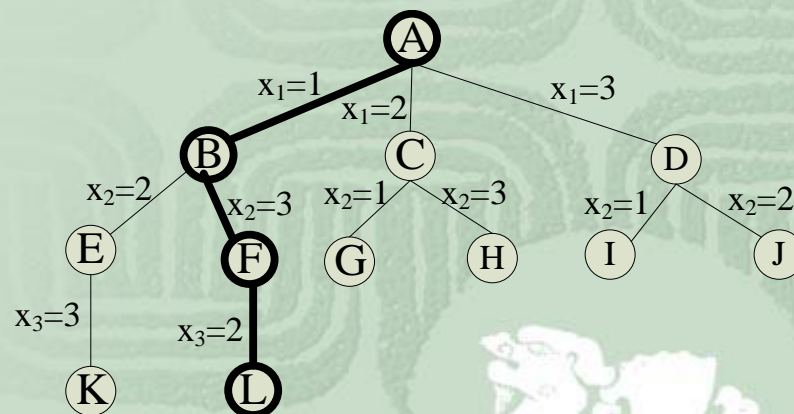
- 扩展结点D沿着 $x_2=2$ 的分支扩展， $cf=11$ ， $bestf=18$ ， $cf < bestf$ ，限界条件满足，扩展生成的结点J成为活结点。



- 扩展结点J沿着 $x_3=1$ 的分支扩展， $cf=19$ ， $bestf=18$ ， $cf > bestf$ ，限界条件不满足，扩展生成的结点被剪掉，开始回溯到活结点D，结点D的两个分支搜索完毕，继续回溯到活结点A。



- 活结点A的三个分支也已搜索完毕，结点A变成死结点，搜索结束。至此，找到的最优的调度方案为从根结点A到叶子结点L的路径（1, 3, 2）



关键代码分析

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]; }
}
```



■ 算法分析

- ✧ 计算限界函数需要 $O(1)$ 时间，需要判断限界函数的结点在最坏情况下有 $1+n+n(n-1)+n(n-1)(n-2)+\dots+n(n-1)+\dots+2 \leq nn!$ 个，故耗时 $O(nn!)$;
- ✧ 在叶子结点处记录当前最优解需要耗时 $O(n)$ ，在最坏情况下会搜索到每一个叶子结点，叶子结点有 $n!$ 个，故耗时为 $O(nn!)$ 。
- ✧ 批处理作业调度问题的回溯算法所需的计算时间为 $O(nn!)+O(nn!)=O(nn!)=O((n+1)!)$ 。



示例2：旅行售货员问题

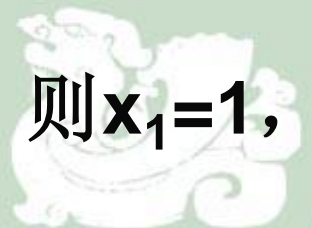
■ 问题描述

∞ 设有 n 个城市组成的交通图，一个售货员从住地城市出发，到其它城市各一次去推销货物，最后回到住地城市。假定任意两个城市 i, j 之间的距离 $d_{ij}(d_{ij}=d_{ji})$ 是已知的，问应该怎样选择一条最短的路线？

■ 定义问题的解空间

∞ 解的形式 (x_1, x_2, \dots, x_n)

∞ 令 n 个城市组成的集合为 $S=\{1, 2, \dots, n\}$ ，则 $x_1=1$ ， $x_i \in S - \{x_1, x_2, \dots, x_{i-1}\}$ ， $i=2, \dots, n$ 。



确定解空间的组织结构

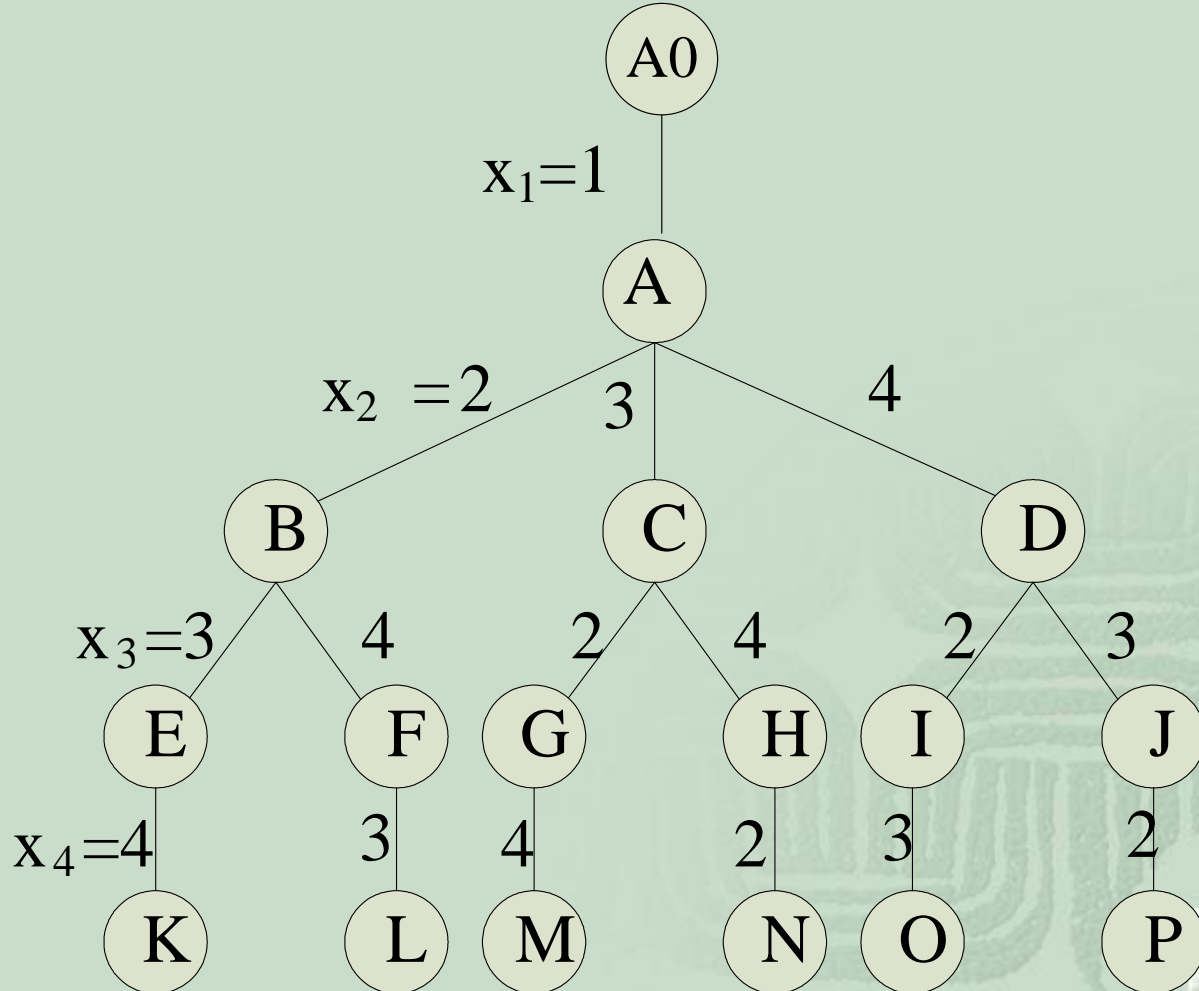


图5-19 $n=4$ 的解空间树

$n=4$ 的旅行售货员问题的解空间树

搜索解空间

- 设置约束条件;

- ∞ 用二维数组 $c[i][j]$ 存储无向带权图的邻接矩阵, 如果 $c[i][j] \neq \infty$ 表示城市 i 和城市 j 有边相连, 能走通。

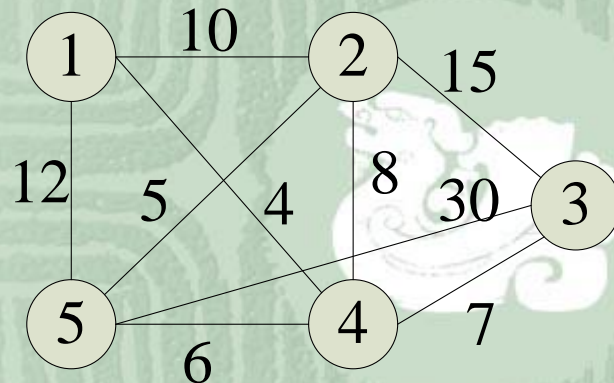
- 设置限界条件

- ∞ $cl < bestl$, cl 的初始值为 0, $bestl$ 的初始值为 $+\infty$ 。

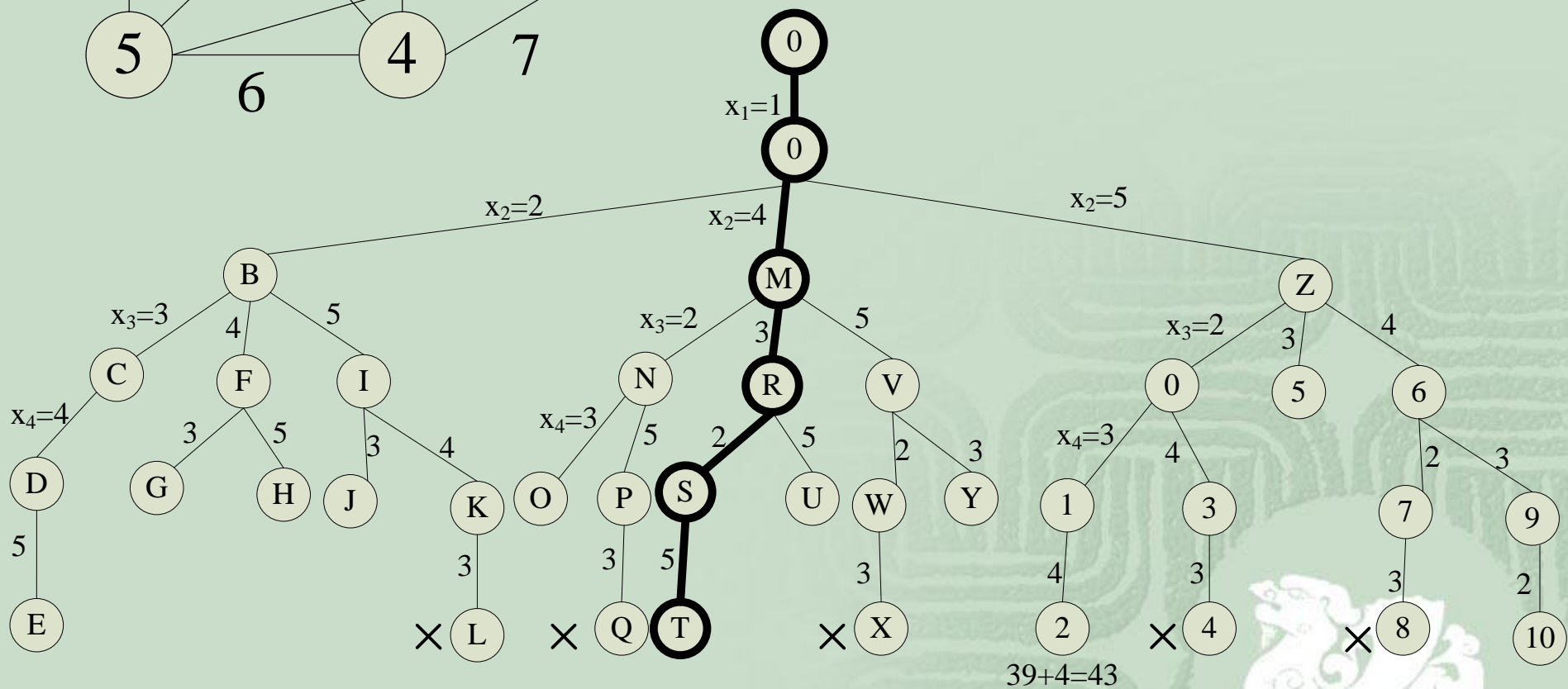
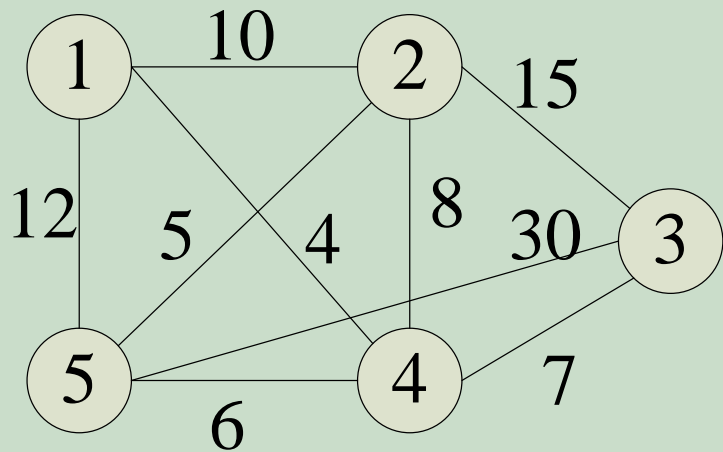
- ∞ cl : 当前已走过的城市所用的路径长度

- ∞ $bestl$: 表示当前找到的最短路径的路径长度

- 搜索过程: 以右图为例说明



搜索树



Backtrack(int i)

算法描述

```
{ if (i == n)
    { if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
        (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge))
        { for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
            bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];
        }
    }
else
{ for (int j = i; j <= n; j++) //是否可进入x[j]子树?
    if (a[x[i-1]][x[j]] != NoEdge &&
        (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) // 搜索子树
    { Swap(x[i], x[j]);
      cc += a[x[i-1]][x[i]];
      Backtrack(i+1);
      cc -= a[x[i-1]][x[i]];
      Swap(x[i], x[j]);
    }
}
```



■ 算法分析

- ✧ 判断限界函数需要 $O(1)$ 时间，在最坏情况下有 $1+(n-1)+(n-1)(n-2)+\dots+(n-1)+\dots+2 \leq n(n-1)!$ 个结点需要判断限界函数，故耗时 $O(n!)$;
- ✧ 在叶子结点处记录当前最优解需要耗时 $O(n)$ ，在最坏情况下会搜索到每一个叶子结点，叶子结点有 $(n-1)!$ 个，故耗时为 $O(n!)$ 。
- ✧ 旅行售货员问题的回溯算法所需的计算时间为 $O(n!)+O(n!)=O(n!)$ 。



思考

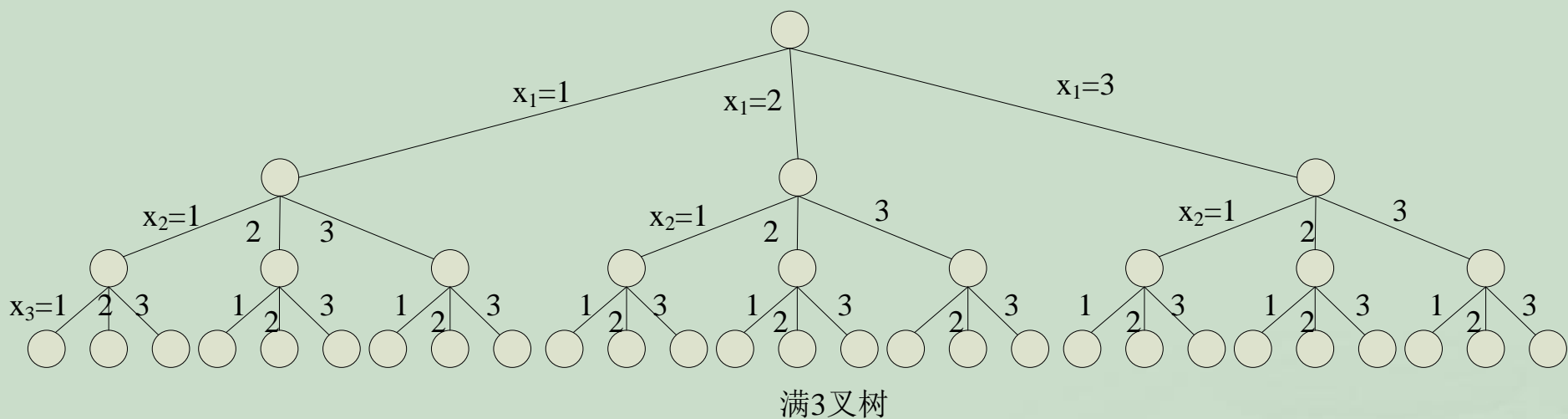
- 该算法效率不高，为什么？
- 怎么找更好的下界函数？规约矩阵



5.3.4 满 m 叉树

- 当所给问题的 n 个元素中每一个元素均有 m 种选择，要求确定其中的一种选择，使得对这 n 个元素的选择结果组成的向量满足某种性质，即寻找满足某种特性的 n 个元素取值的一种组合。这类问题的解空间树称为满 m 叉树。
- 解的形式为 n 元组 (x_1, x_2, \dots, x_n) ，分量 $x_i (i=1, 2, \dots, n)$ 表示第 i 个元素的选择为 x_i





树的根结点：初始状态

中间结点：某种情况下的中间状态

叶子结点：结束状态

分支：从一个状态过渡到另一个状态的行为

从根结点到叶子结点的路径：一个可能的解（**n**个元素值的一个组合）

子集树的深度：等于问题的规模。



满m叉树问题算法模板

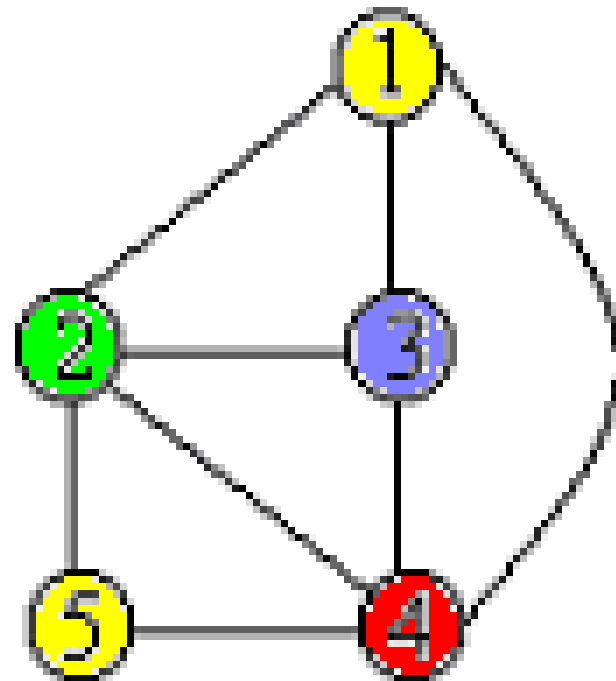
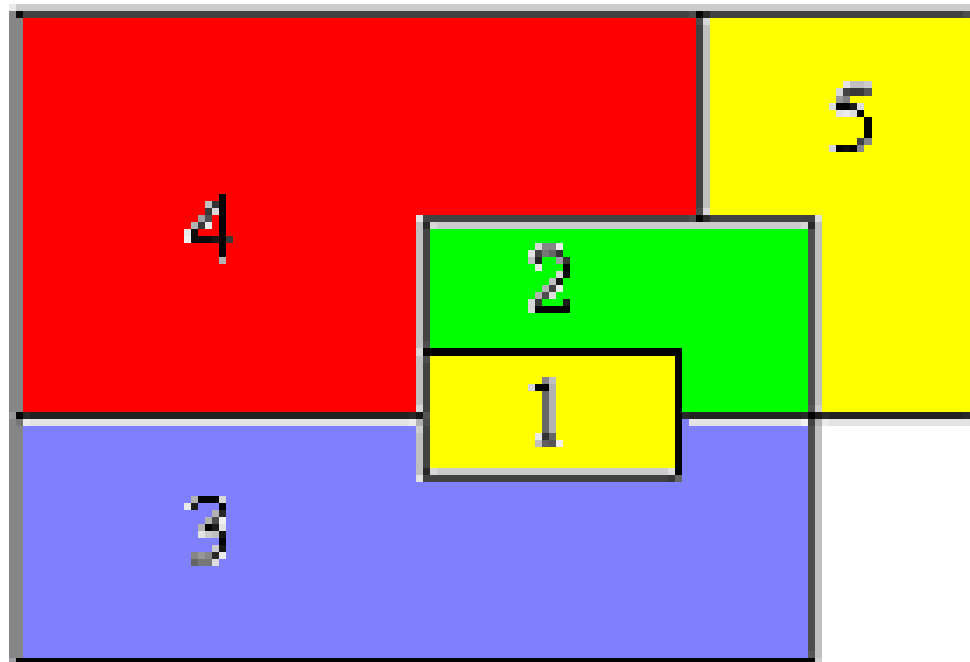
```
void Backtrack (int t)
{
    if (t>n)
        output(x);
    else
        for (int i=1;i<=m;i++)
            if ( constraint(t) && bound(t) )
            {
                x[t]=i;
                做其他相关标识;
                Backtrack(t+1);
                做其他相关标识的反操作; //退回相关标识
            }
}
```

示例1：图的 m 着色问题

➤ 问题描述

给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。是否有一种着色法使 G 中每条边的2个顶点着不同颜色。这个问题是图的 m 可着色判定问题。若一个图最少需要 m 种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数 m 为该图的色数。求一个图的色数 m 的问题称为图的 m 可着色优化问题。

示例1：图的m着色问题



地图：四色原理



➤ 定义问题的解空间

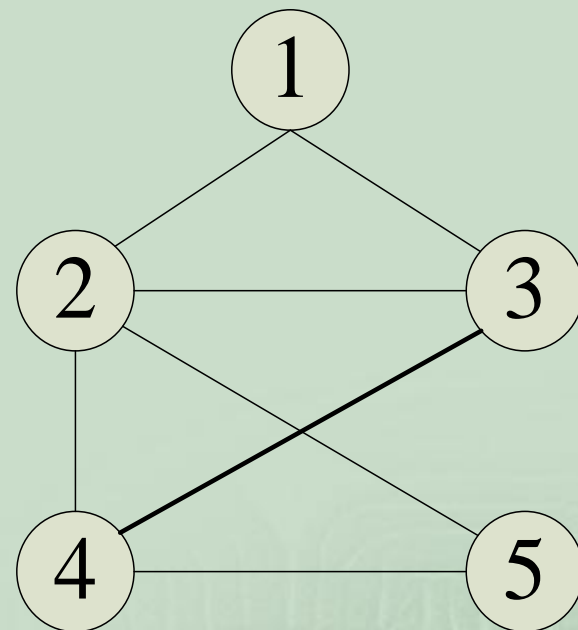
- 解的形式: (x_1, x_2, \dots, x_n)
- x_i 的取值范围: $x_i = 1, 2, \dots, m$

➤ 组织解空间

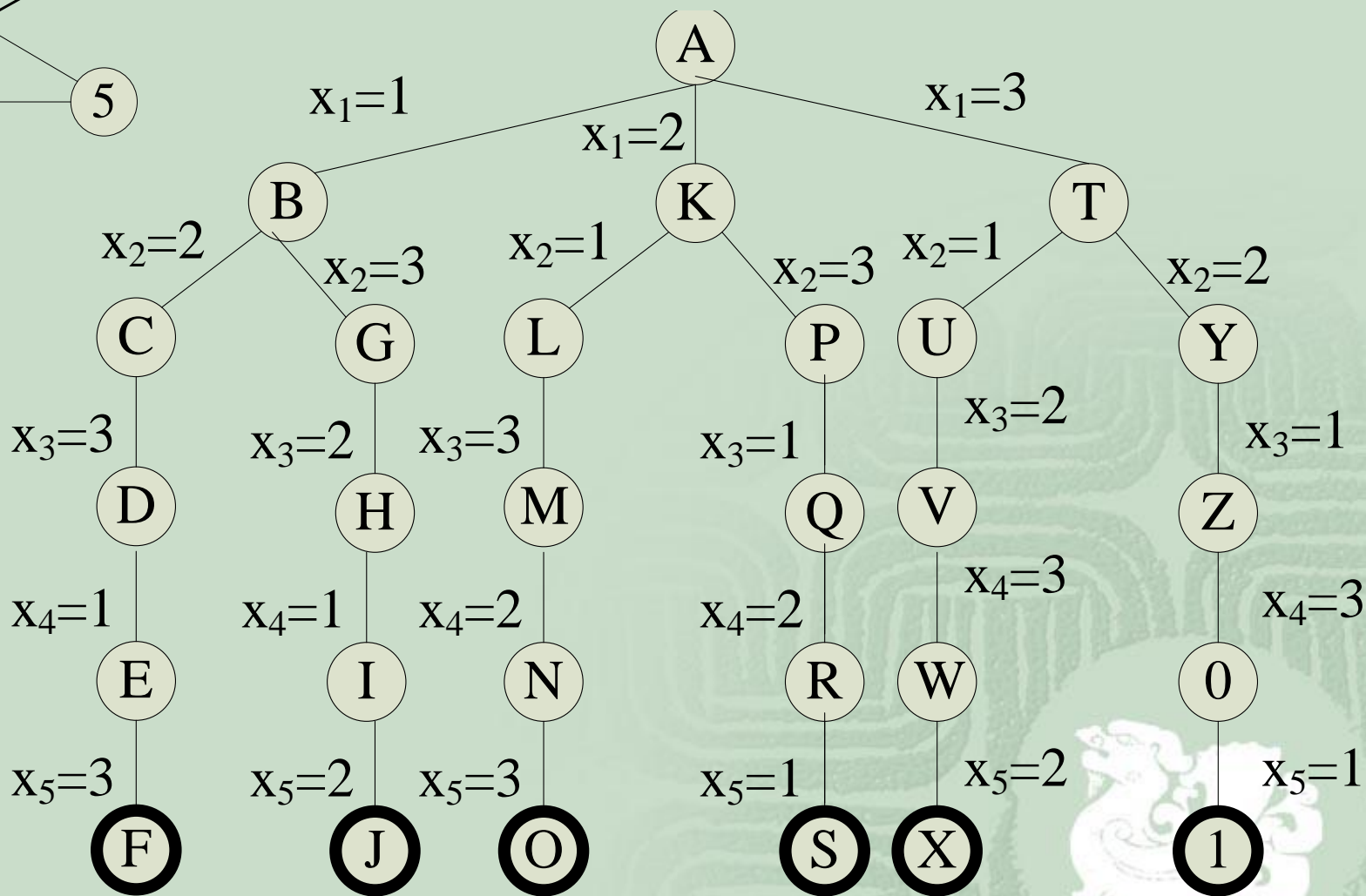
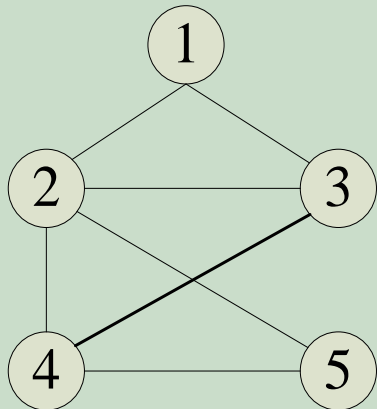
- 满 m 叉树, 树的深度为 n

➤ 搜索解空间

- 约束条件: 和已经确定颜色且有边相连的顶点颜色不相同。
- 限界条件: (\times)
- 搜索过程
- (以右图为例, 给定 $m=3$)



该问题只需求出某一个可行解, 如果想得到多个解, 怎么处理?



搜索树

关键代码分析

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

```
Backtrack(int t)
{  if (t>n)
    {  sum++;
        for (int i=1; i<=n; i++)
            cout << x[i] << ' ';
        cout << endl;
    }
    else
        for (int i=1;i<=m;i++)
        {  x[t]=i;
            if (Ok(t))
                Backtrack(t+1);
        }
}
```



■ 算法分析

∞ 计算约束函数需要 $O(n)$ 时间，需要判断限界函数的结点在最坏情况下有：

$$1+m+m^2+m^3+\dots+m^{n-1}=(m^n-1)/(m-1) \text{ 个}$$

故耗时 $O(nm^n)$ ；

∞ 在叶子结点处输出着色方案需要耗时 $O(n)$ ，在最坏情况下会搜索到每一个叶子结点，叶子结点有 m^n 个，故耗时为 $O(nm^n)$ 。

∞ 图的 m 着色问题的回溯算法所需的计算时间为 $O(nm^n)+O(nm^n)=O(nm^n)$ 。

示例2：最小重量机器设计问题

■ 问题描述

☞ 设某一机器由 n 个部件组成，每一个部件可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量， c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

■ 问题分析

☞ 该问题实质上是为机器部件选供应商。机器由 n 个部件组成，每个部件有 m 个供应商可以选择，要求找出 n 个部件供应商的一个组合，使其满足 n 个部件总价格不超过 c 且总重量是最小的。

■ 定义问题的解空间。

∞ 该问题的解空间形式为 (x_1, x_2, \dots, x_n) ，分量 $x_i (i=1, 2, \dots, n)$ 表示第 i 个部件从第 x_i 个供应商处购买。 m 个供应商的集合为 $S=\{1, 2, \dots, m\}$ ， $x_i \in S$ ， $i=1, 2, \dots, n$ 。

■ 确定解空间的组织结构。

∞ 一棵满 m 叉树，树的深度为 n 。

■ 搜索解空间。

∞ 约束条件：

$$\sum_{i=1}^n c_{ix_i} \leq c$$

$$cw = \sum_{i=1}^t w_{ix_i}$$

∞ 限界条件： $cw < bestw$



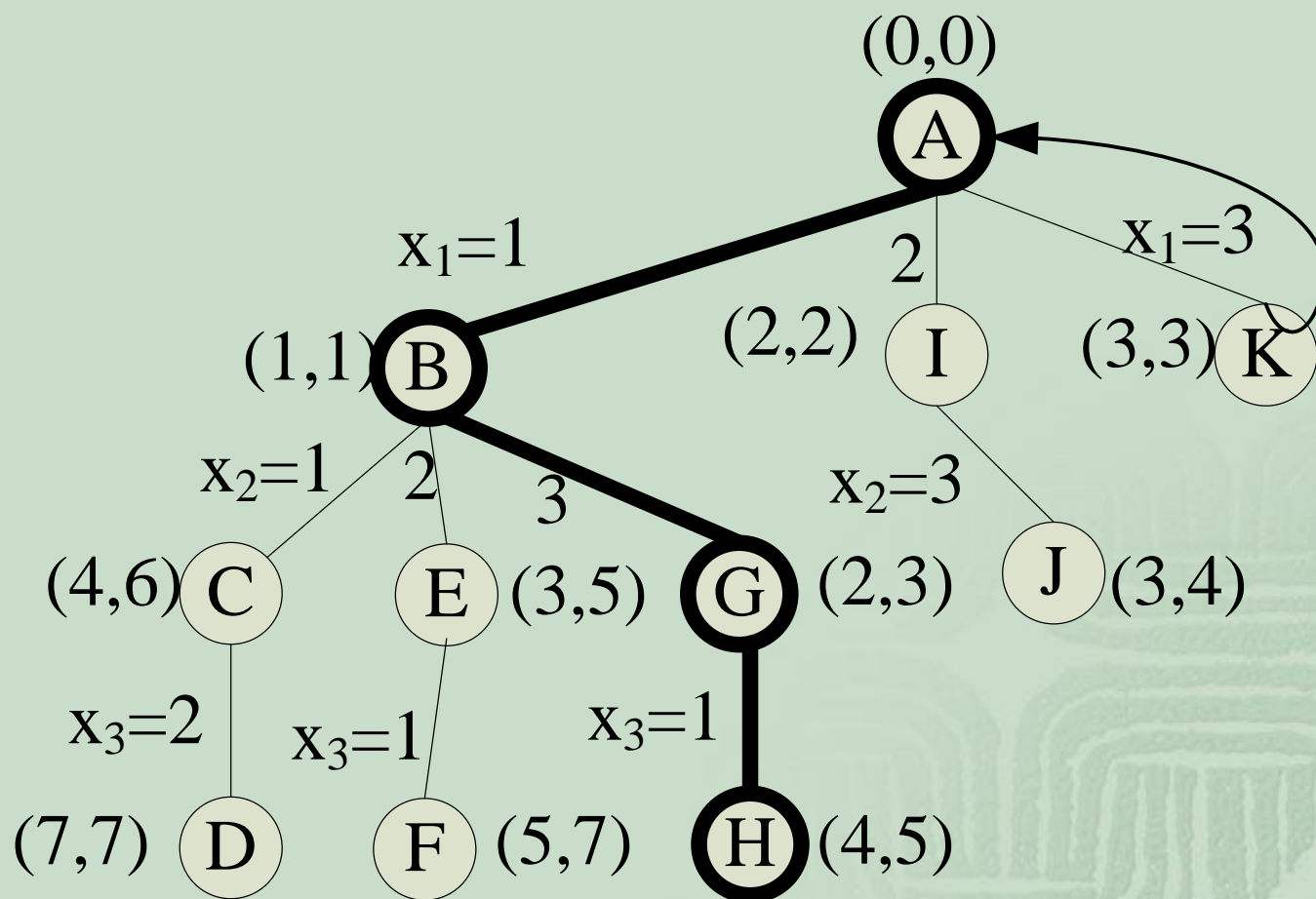
- 考虑 $n=3$, $m=3$, $c=7$ 的实例。部件的重量如表5-2所示、价格如表5-3所示。

表5-2 部件的重量表

W	1	2	3
1	1	2	3
2	3	2	1
3	2	3	2

表5-3 部件的价格表

C	1	2	3
1	1	2	3
2	5	4	2
3	2	1	2



搜索树



关键代码分析

```
void Backtrack(int t)
{
    if(t>n) //到达叶子结点
    {
        bestw=cw;
        for(int j=1;j<=n;j++) //保存当前最优解
            bestx[j]= x[j];
        return;
    }
    for(int j=1; j<=m; j++) //依次搜索每一个供应商
    {
        x[t]=j;
        if( cc+c[t][j]<=COST && cw+w[t][j]<bestw)
        {
            cc+=c[t][j]; cw+=w[t][j];
            Backtrack(t+1); cc-=c[t][j]; cw-=w[t][j];
        }
    }
}
```

是否有更好的限界？重量的规约、价格的规约

■ 算法分析

- ✧ 计算约束函数和限界函数需要 $O(1)$ 时间，需要判断约束函数和限界函数的结点在最坏情况下有 $1+m+m^2+m^3+\dots+m^{n-1}=(m^n-1)/(m-1)$ 个，故耗时 $O(m^n-1)$;
- ✧ 在叶子结点处记录当前最优方案需要耗时 $O(n)$ ，在最坏情况下会搜索到每一个叶子结点，叶子结点有 m^n 个，故耗时为 $O(nm^n)$ 。
- ✧ 最小重量机器设计问题的回溯算法所需的计算时间为 $O(m^{n-1})+O(nm^n)=O(nm^n)$ 。



回溯法效率分析

回溯算法的效率在很大程度上依赖于以下因素：

- (1)产生 $x[k]$ 的时间，与深度有关；
- (2)满足显约束的 $x[k]$ 值的个数，解空间的大小；
- (3)计算约束函数**constraint**的时间；
- (4)计算上界函数**bound**的时间；
- (5)满足约束函数和上界函数约束的所有 $x[k]$ 的个数。

好的约束函数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，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。

三类问题的时间复杂性

- 子集树问题的时间复杂性
 $\propto O(n2^n)$
- 排列树问题的时间复杂性
 $\propto O(n!)$
- 满 m 叉树问题的时间复杂性
 $\propto O(nm^n)$



思考题（递归与迭代）

- 1.组合问题用回溯法。（?? 树）见备注，关注限界的写法，统计限界前后的计算节点个数。
- 2.有 n 位顾客顺序等待某项服务， T_i 表示顾客 i 需要的服务时间，共有 S 处提供此服务，如何安排顾客分别到 S 处接受服务而总体等待时间最小。（?? 树）这题用贪心法更简单
- 3.双 n 皇后问题/多 n 皇后问题。（?? 树）
- 4.地图上每块方格中都包含不等的矿产资源，你只能建立一个面积为 S （ S 个方格）的基地，并且基地是个连通图，找出基地能够包含的最多资源是多少。（?? 树）

思考题（递归与迭代）

- 5. 桥本分数：把1,2,...,9这9个数字不重复地填入下式的9个方格中使等式成立。试求出所有解答，等式左边两个分数交换次序只算一个解答。蓝桥杯怎么解的（？？树）

$$\frac{\square}{\square\square} + \frac{\square}{\square\square} = \frac{\square}{\square\square}$$

蓝桥用M叉树，但用排列树更好



思考题（递归与迭代）

- 6. 农夫带着一只狼、一只羊和一筐菜划船过河，农夫一次只能带一种过河，应该如何安排。（？？树）

