

贪心算法

陈长建

计算机科学系

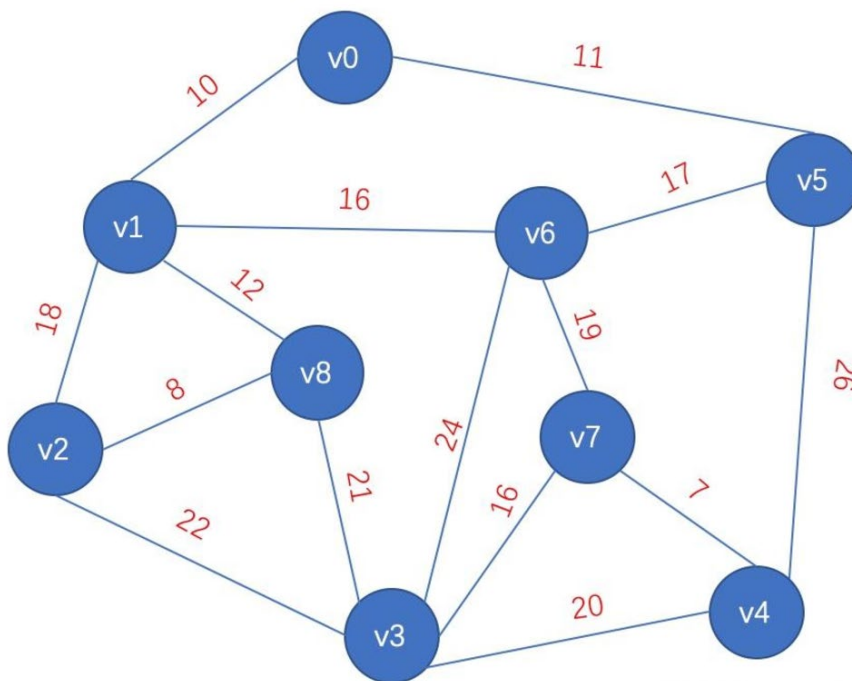
上课提醒

- 本周开始，周一、周三各一次课

星期一	星期二	星期三
		算法设计与分析*(课程授课) 陈长建 讲课学时:40 30102(13-16周) 复307 上课人数:92 计科2204-06 重修人数:
算法设计与分析*(课程授课) 陈长建 讲课学时:40 10506(1-16周) 复307 上课人数:92 计科2204-06 重修人数:		

课前问题

- 阿里面试题，最小生成树



课前问题

- 0-1背包问题的伪代码

课前问题

backtrack(t)

if $(t \geq n)$:

if $bestp < cp$:

$bestp = cp$

else:

if $cw + w[t] \leq c$:

$x[t] = 1$

$cw = cw + w[t]$

$cp = cp + v[t]$

backtrack(t+1)

$cw = cw - w[t]$

$cp = cp - v[t]$

$x[t] = 0$

backtrack(t+1)

例3 装载问题

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 $\sum_{i=1}^n w_i \leq c_1 + c_2$

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。

装载问题

- 容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。
- (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第一艘轮船的载重量。

装载问题

由此可知，装载问题等价于以下特殊的0-1背包问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n w_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n w_i x_i \leq c_1 \\ & x_i \in \{0,1\}, 1 \leq i \leq n \end{aligned}$$

什么情况下回溯法要优于动态规划？

例子

- 例如 $n=4$, $c1=12$, $w=[8, 6, 2, 3]$.

装载问题

- 解空间：子集树
- 可行性约束函数(选择当前元素): $\sum_{i=1}^n w_i x_i \leq c_1$
- 上界函数(不选择当前元素): $cw+r$ (当前载重量+剩余集装箱的重量)
- 在以当前扩展结点为根的子树中任一叶结点所相应的载重量均不超过 $cw+r$

装载问题回溯算法思路

用排序树表示解空间,则解为n元向量 $\{x_1, \dots, x_n\}$,

$x_i \in \{0, 1\}$

约束条件: $\sum_{i=1}^j w_i x_i + w_{j+1} \leq c_1$

由于是最优化问题: $\max \sum_{i=1}^n w_i x_i$, 可利用此条件进一步剪去含最优解的子树.

设bestw: 当前最优载重量,(某个叶节点)

$cw = \sum_{i=1}^j w_i x_i$: 当前扩展结点的载重量;

$\sum_{i=j+1}^n w_i$: 剩余集装箱的重量;

当 $cw + r$ (限界函数) $\leq \text{bestw}$ 时, 将cw对应的子树剪去。

装载问题回溯算法思路

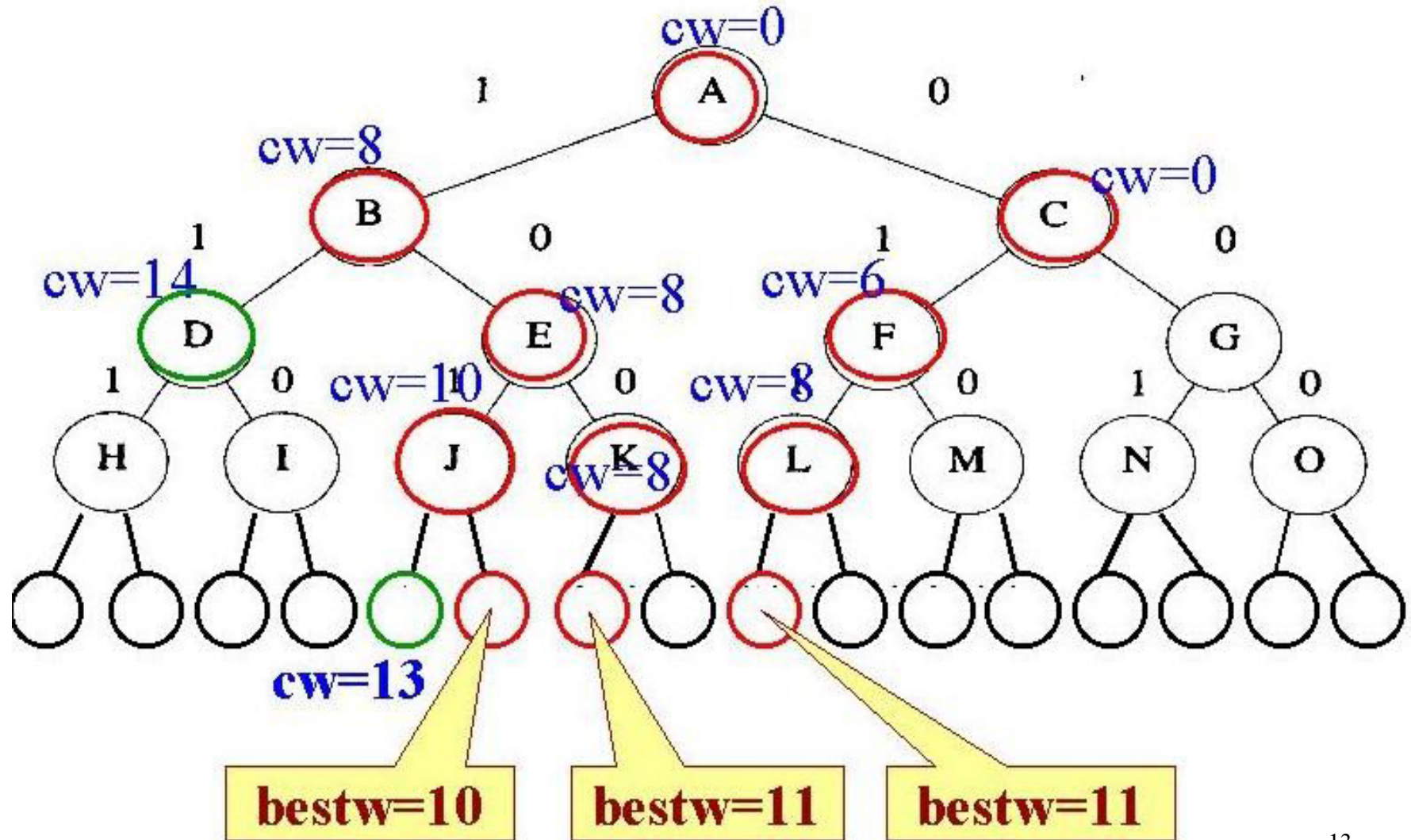
- 限界方法1：设cw为已装重量，如 $cw + w_{j+1} > c_1$ 则杀死该子结点。
- 限界方法2：设bestw为当前最优装箱重量，r为未装的货箱的总重量，如 $cw + r \leq bestw$ ，则停止展开该结点。
- 两种限界同时使用。

剪枝条件:

$$\sum_{i=1}^j w_i x_i + w_{j+1} > c_1$$

$$cw + r \leq bestw$$

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

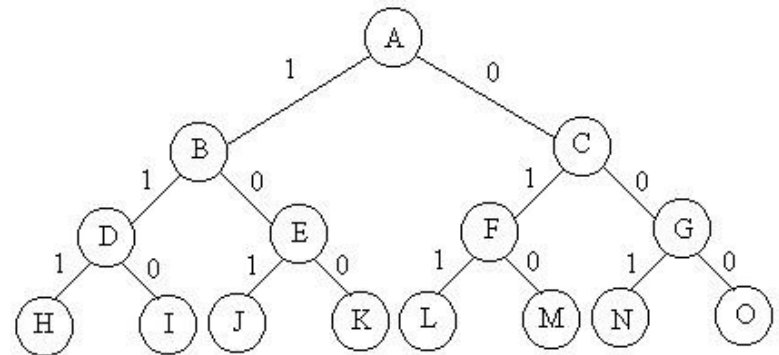


装载问题

```

template < class Type >
Type Maxloading(type w[], type c, int n,) //返回最优装载量
{ loading <Type> X;    //初始化X
  X. w=w; //集装箱重量数组
  X. c=c; //第一艘船载重量
  X. n=n; //集装箱数
  X. bestw=0; //当前最优载重
  X. cw=0; //当前载重量
  X. r=0; //剩余集装箱重量
  for (int i=1; i<=n; i++)
    X. r +=w[i]
  //计算最优载重量
  X.backtrack(1);
  return X.bestw; }

```



装载问题

```
template <class Type>
void Loading<Type>:: void backtrack (int i)
{ // 搜索第i层结点
    if (i > n) { // 到达叶结点
        bestw=cw;return;}
    //搜索子树
    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) {
        x[i] = 0; // 搜索右子树
        backtrack(i + 1);    }
    r += w[i]; }
```

时间复杂度

装载问题等价于以下特殊的0-1背包问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n w_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n w_i x_i \leq c_1 \\ & x_i \in \{0,1\}, 1 \leq i \leq n \end{aligned}$$

用回溯法设计解装载问题的 $O(2^n)$ 计算时间算法。在某些情况下该算法优于动态规划算法。

例4 - 批处理作业调度

- 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} 。
- 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

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。

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

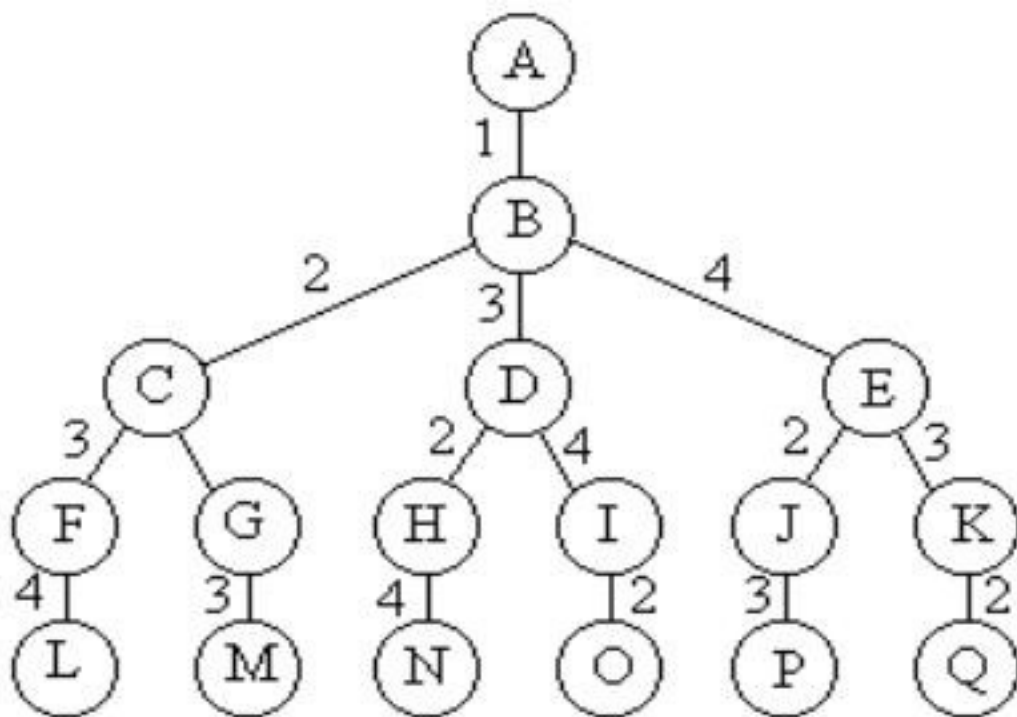
这3个作业的6种可能的调度方案是1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; 3,2,1; 它们所相应的完成时间和分别是19, 18, 20, 21, 19, 19。易见，最佳调度方案是1,3,2，其完成时间和为18。

使用Johnson算法?

- (1) 令 $N_1 = \{i \mid a_i < b_i\}, N_2 = \{i \mid a_i \geq b_i\}$;
- (2) 将 N_1 中作业依 a_i 的升序排序; 将 N_2 中作业依 b_i 的降序排序;
- (3) N_1 中作业接 N_2 中作业构成满足Johnson法则的最优调度。

批处理作业调度

- 解空间：排列树



批处理作业调度

- 类Flowshop的定义

```
class Flowshop {  
    friend Flow(int**, int, int []);  
    private:  
        void Backtrack(int i);  
        int  **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};
```

批处理作业调度

```
void Flowshop::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];//难点1
            f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];//难点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]; }
}
```

批处理作业调度的运行实例

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



符号三角形问题

- 下图是由14个 “+”和14个 “-”组成的符号三角形。2个同号下面都是 “+”，2个异号下面都是 “-”。

```

+  +  -  +  -  +  +
  +  -  -  -  -  +
    -  +  +  +  -
      -  +  +  -
        -  +  -
          -  -
            +

```


符号三角形问题

- 在一般情况下，符号三角形的第一行有 n 个符号。符号三角形问题要求对于给定的 n ，计算有多少个不同的符号三角形，使其所含的 “+” 和 “-” 的个数相同。
- 解向量：用 n 元组 $x[1:n]$ 表示符号三角形的第一行。
- 可行性约束函数：当前符号三角形所包含的 “+” 个数与 “-” 的个数均不超过 $n*(n+1)/4$
- 无解的判断： $n*(n+1)/2$ 为奇数

符号三角形问题

```
void Triangle::Backtrack(int t)
{
    if ((count>half)||((t*(t-1)/2-count>half)) return; if
    (t>n) sum++;
    else
        for (int i=0;i<2;i++) {
            p[1][t]=i;
            count+=i;
            for (int j=2;j<=t;j++) {
                p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
                count+=p[j][t-j+1];
            }
            Backtrack(t+1);
            for (int j=2;j<=t;j++)
                count-=p[j][t-j+1];
            count-=i;
        }
}
```

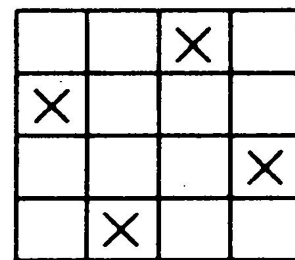
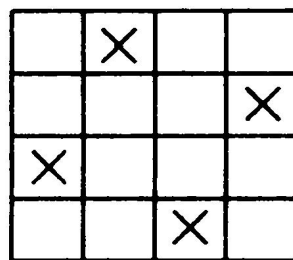
符号三角形问题

复杂度分析

计算可行性约束需要 $O(n)$ 时间，在最坏情况下有 $O(2^n)$ 个结点需要计算可行性约束，故解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$ 。

N后问题

- 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。
- 如4后问题的2个解



N后问题

8后问题的1个解 (共92个解)

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

N后问题

算法思路:将棋盘从左至右,从上到下编号为 $1, \dots, n$,皇后编号为 $1, \dots, n$. 设解为 (x_1, \dots, x_n) , x_i 为皇后 i 的列号,且 x_i 的值表示位于第 x_i 行.解空间: $E = \{ (x_1, \dots, x_n) \mid x_i \in S_i, i=1, \dots, n \}$, $S_i = \{1, \dots, n\}$, $1 \leq i \leq n$. 解空间为排列树.

约束条件D:

- | | |
|---------------------------|---------------------|
| 1) $x_i \neq x_j$ | 皇后 i, j 不在同一列上 |
| 2) $x_i - i \neq x_j - j$ | } 皇后 i, j 不在同一斜线上 |
| 3) $x_i + i \neq x_j + j$ | |

N后问题

- 解向量: (x_1, x_2, \dots, x_n)
- 显约束: $x_i=1, 2, \dots, n$
- 隐约束:
 - 不同列: $x_i \neq x_j$
 - 不处于同一正、反对角线: $|i-j| \neq |x_i-x_j|$

N后问题

```
bool Queen::Place(int k)
{ //判定两个皇后是否在同一斜线或同一列上
    for (int j=1;j<k;j++)
        if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
    return true;
}
```

用函数Place(k) 测试待确定的第k皇后是否和前面已确定的k - 1个皇后是否在同一条斜角线或同一列上。

第k个皇后只要与任一皇后在同一条斜角线，就返回false，当第k个皇后能放置于X(k)的当前值处时，这个返回值为true。

N后问题

```
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);
        }
}
```

N后问题

```
int nQueen(int n)
{
    QueenX;
    //初始化X
    X.n=n; //皇后个数
    X.sum=0;
    int*p=new int [n+1];
    for(int i=0; i<=n; i++) p[i]=0;
    X.x=p;
    X.Backtrack(1);
    delete [] p;
    return X.sum;
}
```

N后问题

- 当皇后数从1到12时各自所对应的可行解的总数如下：

```
n=1      sum=1
n=2      sum=0
n=3      sum=0
n=4      sum=2
n=5      sum=10
n=6      sum=4
n=7      sum=40
n=8      sum=92
n=9      sum=352
n=10     sum=724
n=11     sum=2680
n=12     sum=14200
Press any key to continue
```

8后问题的运行实例 (92个可行解)

```
1-----Display all
2-----Display one by one
0-----exit
```

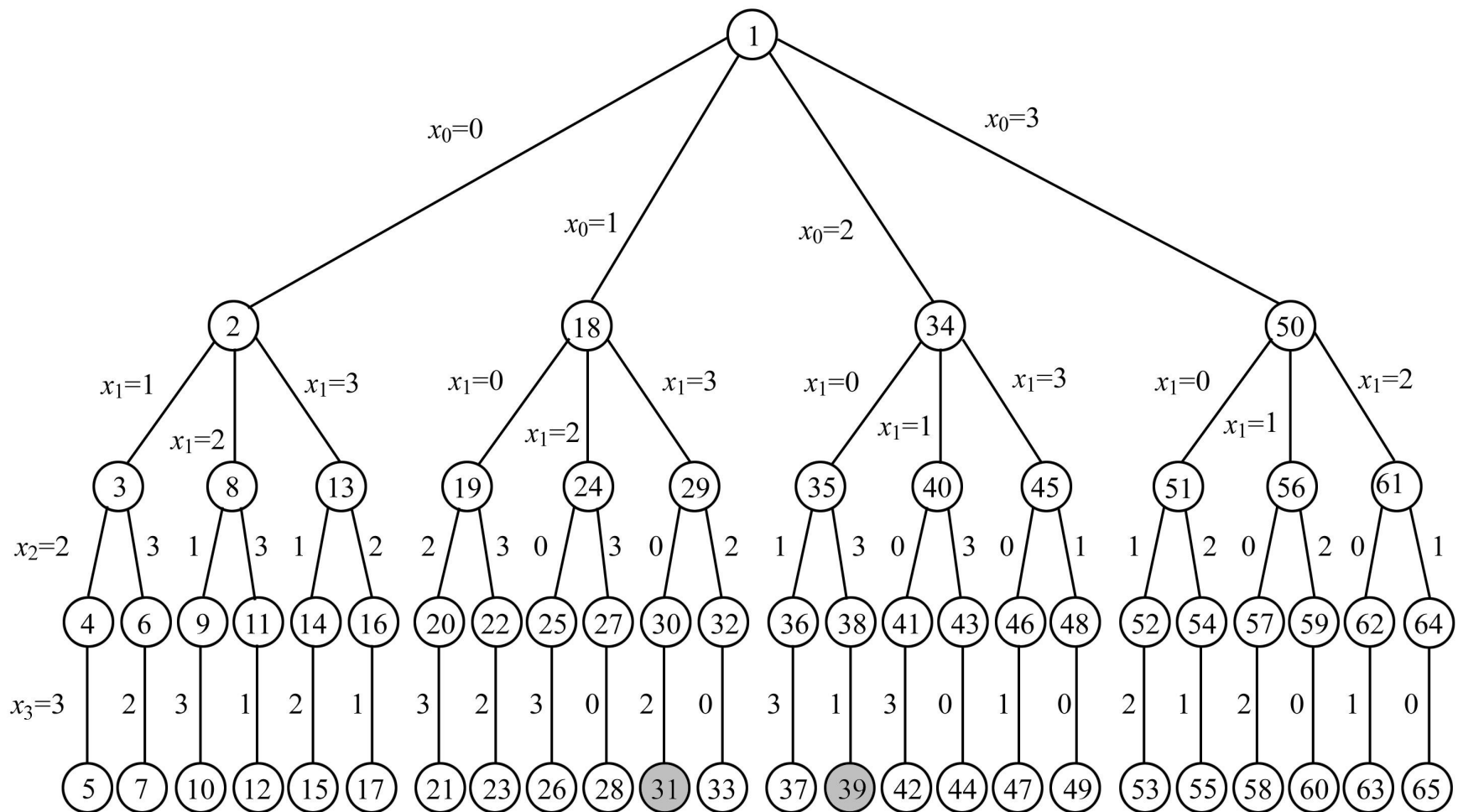
1

Please input the number of queens < Q < <MAX=17>:8

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

Out the 8 queen's sum is:92

4后问题的排列树 (共65个结点)



8后问题的效率分析

- 如果用蛮力处理,即在 8×8 的棋盘上安排出8个位置,这有 C_{64}^8 种安排法,逐一测试,即要检查将近 4.4×10^9 个8元组。
- 用回溯法最多只要作 $8!$ 次检查,即最多只要查40320个8元组。
- 实际上还少得多,对于8皇后问题,不受限结点是8皇后解空间树的结点总数的2.34%
- 因此,用回溯法处理比用枚举法好得多!

图的 m 着色问题

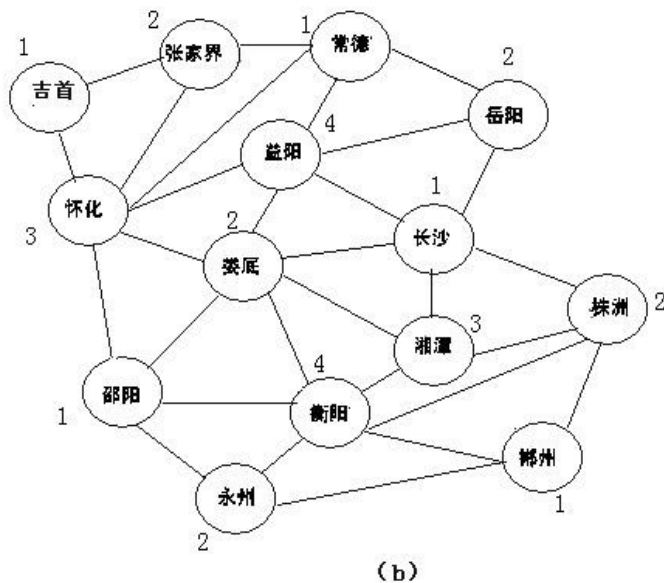
- 图的着色问题是由地图的着色问题引申而来的：用 m 种颜色为地图着色，使得地图上的每一个区域着一种颜色，且相邻区域颜色不同。



图的m着色问题

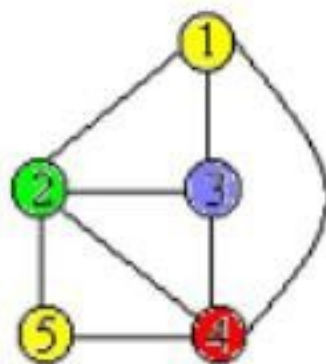
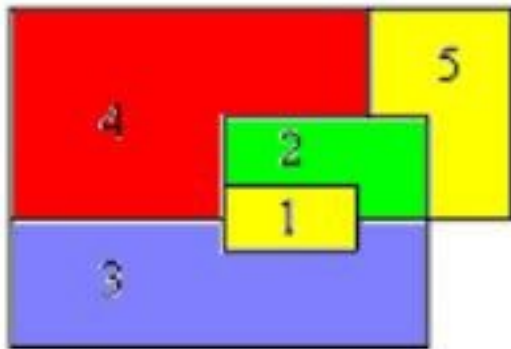
- 问题处理**：如果把每一个区域收缩为一个顶点，把相邻两个区域用一条边相连接，就可以把一个区域图抽象为一个平面图。

例如，图 (a) 所示的区域图可抽象为 (b) 所表示的平面图。19世纪50年代，英国学者提出了任何地图都可以4种颜色来着色的4色猜想问题。过了100多年，这个问题才由美国学者在计算机上予以证明，这就是著名的四色定理。例如，在图中，区域用城市名表示，颜色用数字表示，则图中表示了不同区域的不同着色问题。



图的 m 着色问题

- 图着色问题描述为：给定无向连通图 $G=(V, E)$ 和正整数 m ，求最小的整数 m ，使得用 m 种颜色对 G 中的顶点着色，使得任意两个相邻顶点着色不同。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。



图的 m 着色问题

- 由于用 m 种颜色为无向图 $G=(V, E)$ 着色, 其中, V 的顶点个数为 n , 可以用一个 n 元组 $C=(c_1, c_2, \dots, c_n)$ 来描述图的一种可能着色, 其中, $c_i \in \{1, 2, \dots, m\} (1 \leq i \leq n)$ 表示赋予顶点 i 的颜色。
- 例如, 5元组 $(1, 2, 2, 3, 1)$ 表示对具有5个顶点的无向图的一种着色, 顶点1着颜色1, 顶点2着颜色2, 顶点3着颜色2, 如此等等。
- 如果在 n 元组 C 中, 所有相邻顶点都不会着相同颜色, 就称此 n 元组为可行解, 否则为无效解。

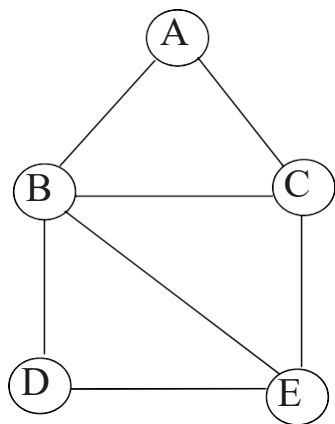
图的m着色问题

回溯法求解图着色问题:

- 首先把所有顶点的颜色初始化为0，然后依次为每个顶点着色。如果其中 i 个顶点已经着色，并且相邻两个顶点的颜色都不一样，就称当前的着色是有效的局部着色；否则，就称为无效的着色。
- 如果由根节点到当前节点路径上的着色，对应于一个有效着色，并且路径的长度小于 n ，那么相应的着色是有效的局部着色。这时，就从当前节点出发，继续探索它的儿子节点，并把儿子结点标记为当前结点。在另一方面，如果在相应路径上搜索不到有效的着色，就把当前结点标记为 $d_$ 结点，并把控制转移去搜索对应于另一种颜色的兄弟结点。
- 如果对所有 m 个兄弟结点，都搜索不到一种有效的着色，就回溯到它的父亲结点，并把父亲结点标记为 $d_$ 结点，转移去搜索父亲结点的兄弟结点。这种搜索过程一直进行，直到根结点变为 $d_$ 结点，或者搜索路径长度等于 n ，并找到了一个有效的着色为止。

图的 m 着色问题

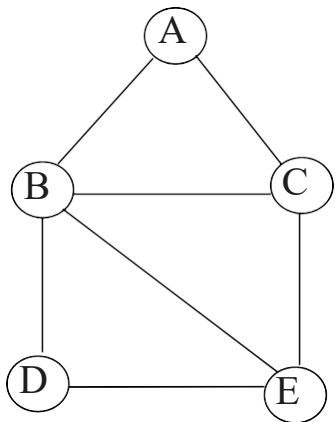
回溯法求解图着色问题示例
($m=3$)



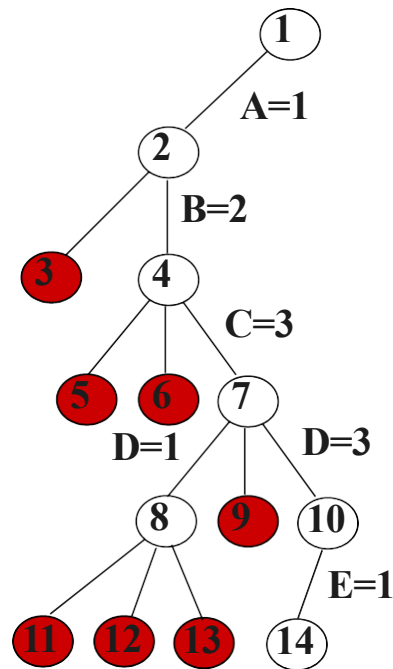
(a) 一个无向图

图的 m 着色问题

回溯法求解图着色问题示例
($m=3$)



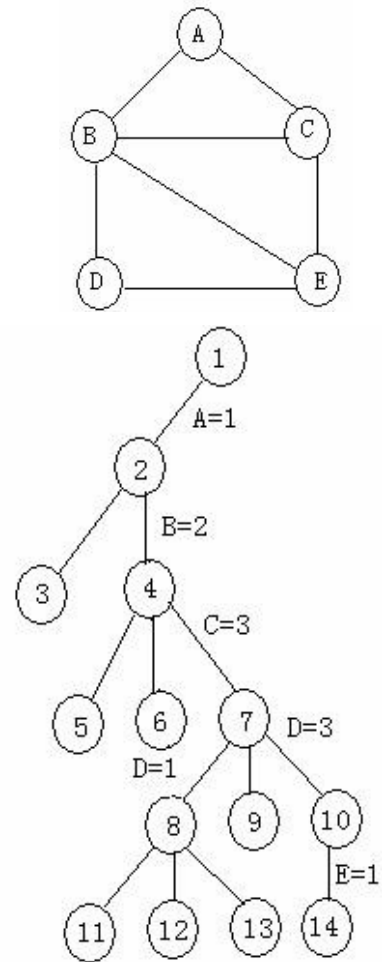
(a) 一个无向图



(b) 回溯法搜索空间

图的m着色问题

- ① 把5元组初始化为 $(0,0,0,0,0)$ ，从根结点开始向下搜索，以颜色1为顶点A着色，生成结点2时，产生 $(1,0,0,0,0)$ ，是个有效着色。
- ② 以颜色1为顶点B着色生成结点3时，产生 $(1,1,0,0,0)$ ，是个无效着色，结点3为d_结点。
- ③ 以颜色2为顶点B着色生成结点4，产生 $(1,2,0,0,0)$ ，是个有效着色。
- ④ 分别以颜色1和2为顶点C着色生成结点5和6，产生 $(1,2,1,0,0)$ 和 $(1,2,2,0,0)$ ，都是无效着色，因此结点5和6都是d_结点。
- ⑤ 以颜色3为顶点C着色，产生 $(1,2,3,0,0)$ ，是个有效着色。重复上述步骤，最后得到有效着色 $(1,2,3,3,1)$ 。



图的m着色问题

- 设数组color[n]表示顶点的着色情况，回溯法求解m着色问题的算法如下：
 - 1. 将数组color[n]初始化为0;
 - 2. $k=1$;
 - 3. while ($k \geq 1$)
 - 3.1 依次考察每一种颜色，若顶点k的着色与其他顶点的着色不发生冲突，则转步骤3.2; 否则，搜索下一个颜色;
 - 3.2 若顶点已全部着色，则输出数组color[n]，返回;
 - 3.3 否则
 - 3.3.1 若顶点k是一个合法着色，则 $k=k+1$ ，转步骤3处理下一个顶点;
 - 3.3.2 否则，重置顶点k的着色情况， $k=k-1$ ，转步骤3。

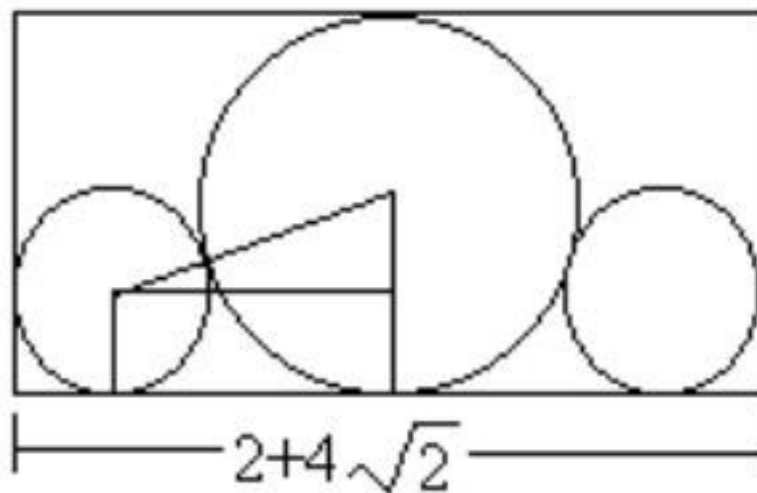
图的m着色问题

```

GraphColor(int n,int m,int color[],bool c[][5])
{
    int i,k;
    for (i=0; i<n; i++ )           //将解向量color[n]初始化为0
        color[i]=0;
    k=0;
    while (k>=0)
    {
        color[k]=color[k]+1;         //使当前颜色数加1
        while ((color[k]<=m) &&(!ok(color,k,c,n))) //当前颜色是否有效
            color[k]=color[k]+1;      //无效，搜索下一个颜色
        if (color[k]<=m)              //求解完毕，输出解
        {
            if (k==n-1)break;         //是最后的顶点，完成搜索
            else k=k+1;                //否，处理下一个顶点
        }
        else                          //搜索失败，回溯到前一个顶点
        {
            color[k]=0;
            k=k-1;
        }
    }
}
    
```


圆排列问题

- 给定 n 个大小不等的圆 c_1, c_2, \dots, c_n ，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如，当 $n=3$ ，且所给的3个圆的半径分别为1, 1, 2时，这3个圆的最小长度的圆排列如图所示。其最小长度为 $2+4\sqrt{2}$



0-1背包问题 – 再思考

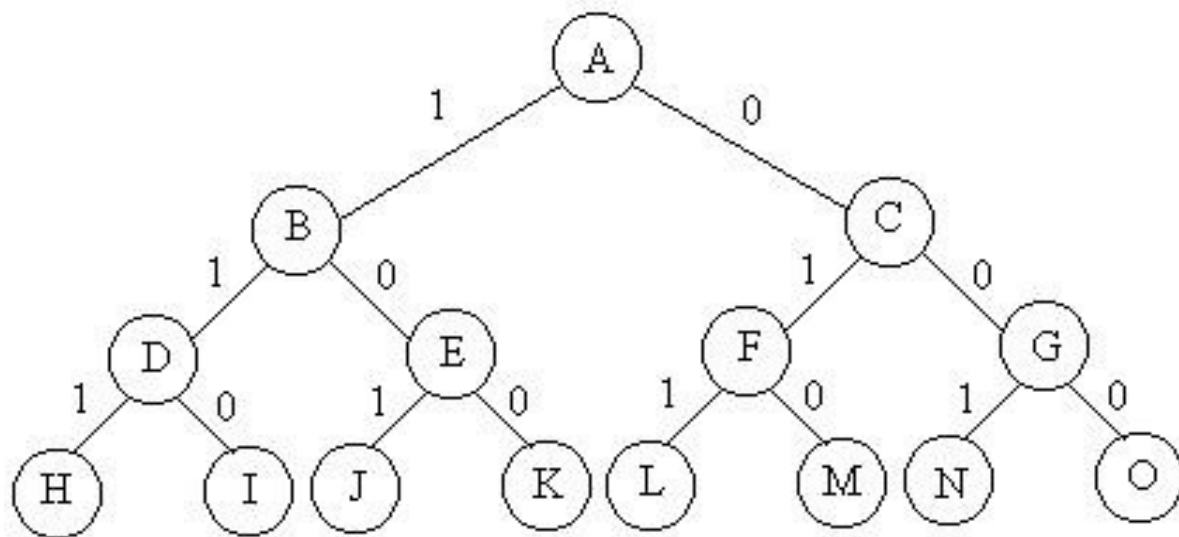
- 0-1背包问题还能否再优化?

0-1背包问题 – 再思考

- 在第四章中,我们用动态规划的方法讨论了背包问题。这里,我们将用回溯的方法求解背包问题。
- 问题的解空间: 由各分量 x_i (取值0或1) 的 2^n 个不同的 n 元向量组成。与子集和数问题的解空间相同。也用树结构表示 (满二叉树)。
- **限界函数**: 取能产生某些值的上界函数。如果扩展给定活结点和它的任一子孙所导致最好可行解的上界不大于迄今所确定的最好解的值, 就可杀死此活结点。
- 0-1背包问题的限界函数: **用贪心方法求取上界值。**

0-1背包问题

- 解空间：子集树
- 可行性约束函数： $\sum_{i=1}^n w_i x_i \leq c_1$
- 上界函数：

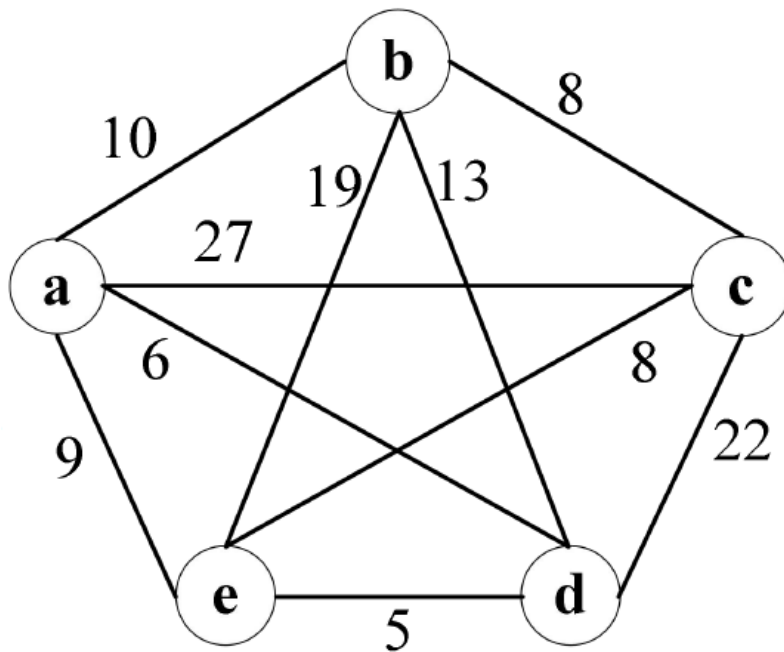


0-1背包问题

```
template<class Typew, class Typep> Typep
Knap<Typew, Typep>::Bound(int i)
{ // (用贪心方法求取上界值)
    Typew cleft = c - cw; // 剩余容量
    Typep 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 -
        = w[i];
        b += p[i];
        i++; }
    // 装满背包
    if (i <= n) b += p[i]/w[i] * cleft;
    return b; // 右子树中解的上界
}
```

TSP问题 – 再思考

- 能否再进一步优化?



旅行售货员问题

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
    if (i == n) { //当前扩展结点是排列树的叶结点的父结点
        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 { //当前扩展结点位于排列树的第i-1层
        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]);} }
    }
```

回溯法的效率分析

- 本节要点：
- 一、影响回溯算法效率的因素
- 二、重排原理
- 三、回溯法的效率

影响回溯算法效率的因素

- 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;
 - (2) 满足显约束的 $x[k]$ 值的个数;
 - (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。

影响回溯算法效率的因素

- 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。

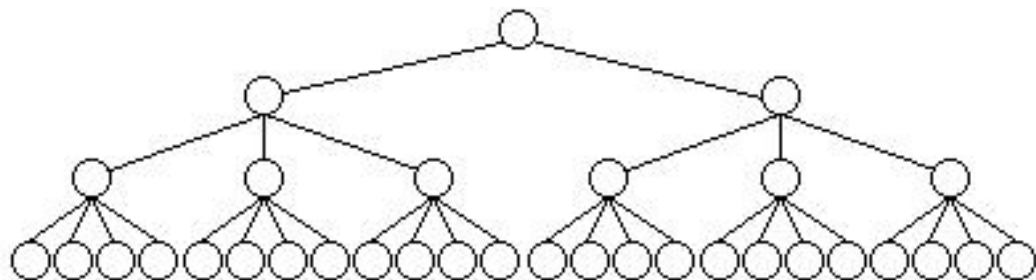
影响回溯算法效率的因素

- 回溯算法的**最坏情况时间复杂度**可达 $O(p(n)n!)$ （或 $O(p(n)2^n)$ 或 $O(p(n)n^n)$ ），这里 $p(n)$ 是 n 的多项式，是生成一个结点所需的时间。

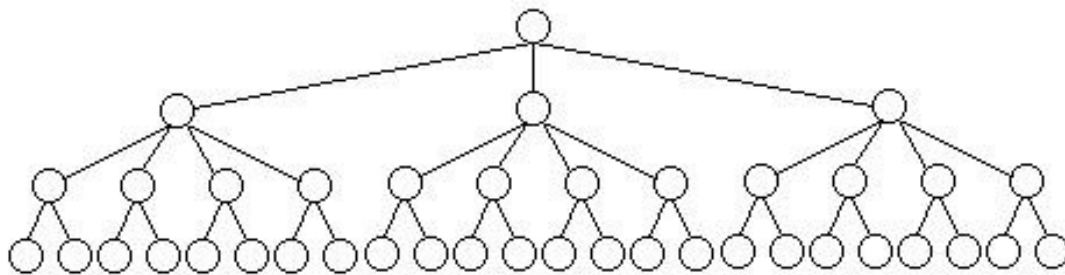
重排原理

- 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。
- 从下图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。

重排原理



(a)



(b)

图(a)中，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。

对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

回溯法小结

- **回溯法要求**问题P的状态能表达为n元组(x_1, x_2, \dots, x_n), **要求**
 $x_i \in S_i$, $i = 1, 2, \dots, n$, **S_i 为有限集**, **对于给定关于n元组中的分量的**
一个约束集D, 满足D的全部约束条件的所有n元组为问题P的
解。
- 从 $k=1$ 开始构造k元组, 如果k元组满足约束, $k=k+1$, 扩展
搜索; 如果试探了 $x[k]$ 的所有值, 仍不能满足约束, 则 $k=k-1$,
回溯到上一层重新选择 $x[k-1]$ 的值。这种扩展回溯的搜索方法
可以表示为状态空间树上的带约束条件的深度优先的搜索。