

算法设计与分析

回溯法

学习要点

2

- 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) 子集树算法框架
 - (4) 排列树算法框架

5.1 回溯法的算法框架

3

- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法有“通用的解题法”之称。

- 回溯法的**基本做法**是搜索，或是一种**组织得井井有条的，能避免不必要搜索的穷举式搜索法**。这种方法适用于解一些组合数相当大的问题。
- 回溯法在问题的解空间树中，按**深度优先**策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

1. 问题的解空间

5

- 用回溯法解决问题时，应明确定义问题的解空间。
- 解空间往往用向量集表示。
- 问题的解向量：回溯法希望一个问题的解能够表示成一个 n 元式 (x_1, x_2, \dots, 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：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

注意：

同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。

例：

6

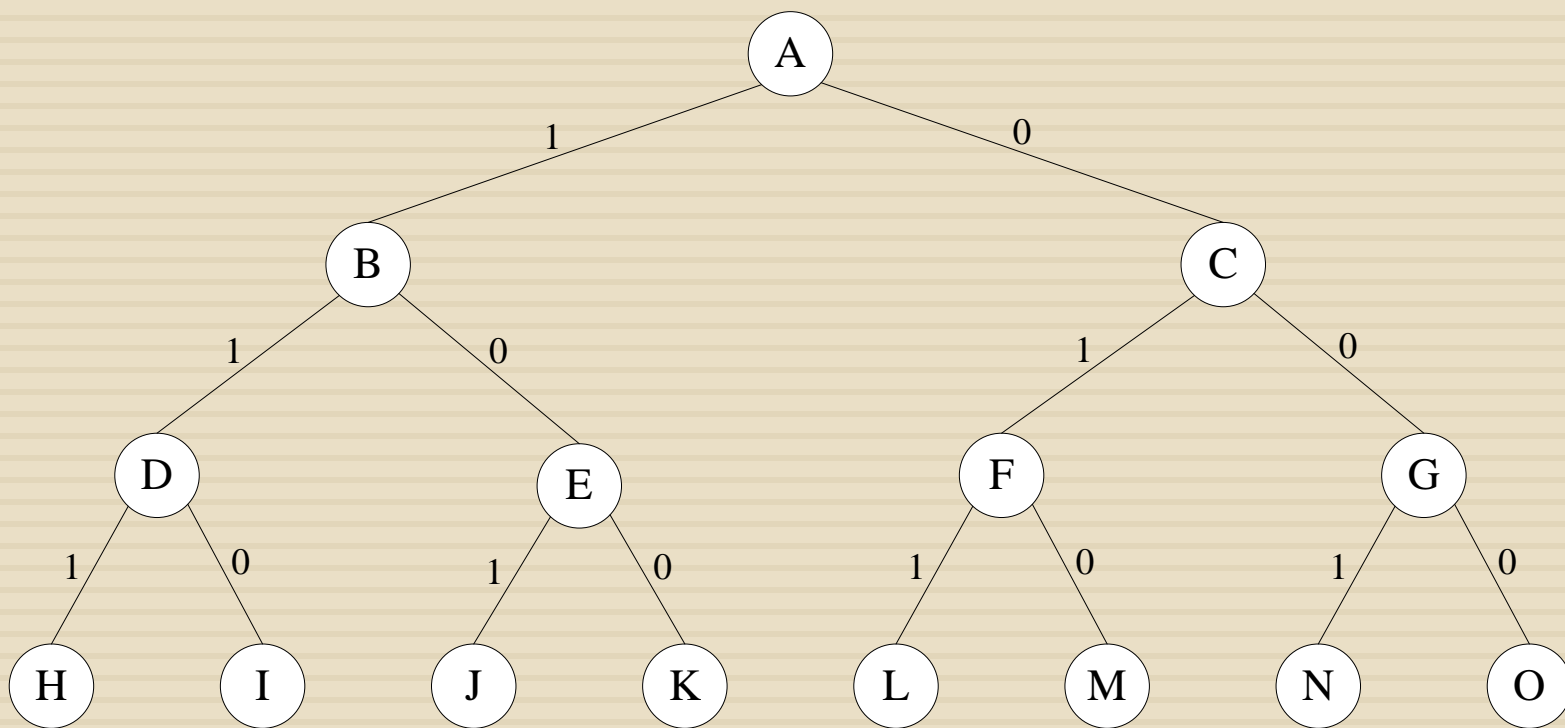
- 对于有 n 种可选物品的0-1背包问题，其解空间由长度为 n 的0-1向量组成。
- 该解空间包含对变量的所有可能的0-1赋值。
- 当 $n=3$ 时，其解空间是：
 $\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$

- 定义了问题的解空间后，还应将解空间很好地组织起来，以便能用回溯法方便地搜索整个解空间。
- 通常将解空间组织成树或图的形式。

例：

8

- 对于 $n=3$ 的0-1背包问题，可用一棵完全二叉树表示其解空间。



2. 回溯法的基本思想

9

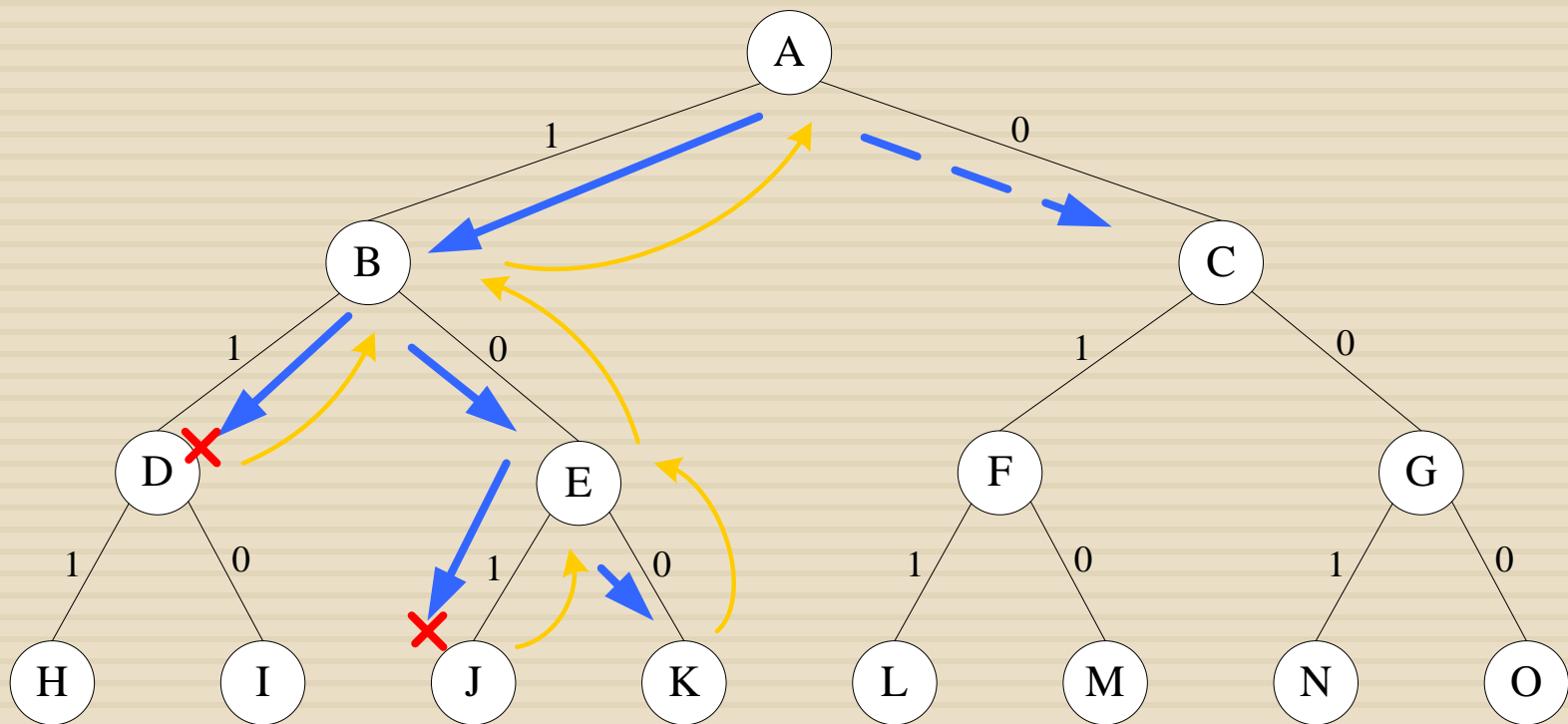
- **扩展结点**：一个正在产生儿子的结点称为扩展结点。
- **活结点**：一个自身已生成但其儿子还没有全部生成的节点称做活结点。
- **死结点**：一个所有儿子已经产生的结点称做死结点。

- **深度优先的问题状态生成法**：如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）。
- **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。
- 回溯法从开始结点（根结点）出发，以**深度优先方式**搜索整个解空间。

例：0-1背包问题

11

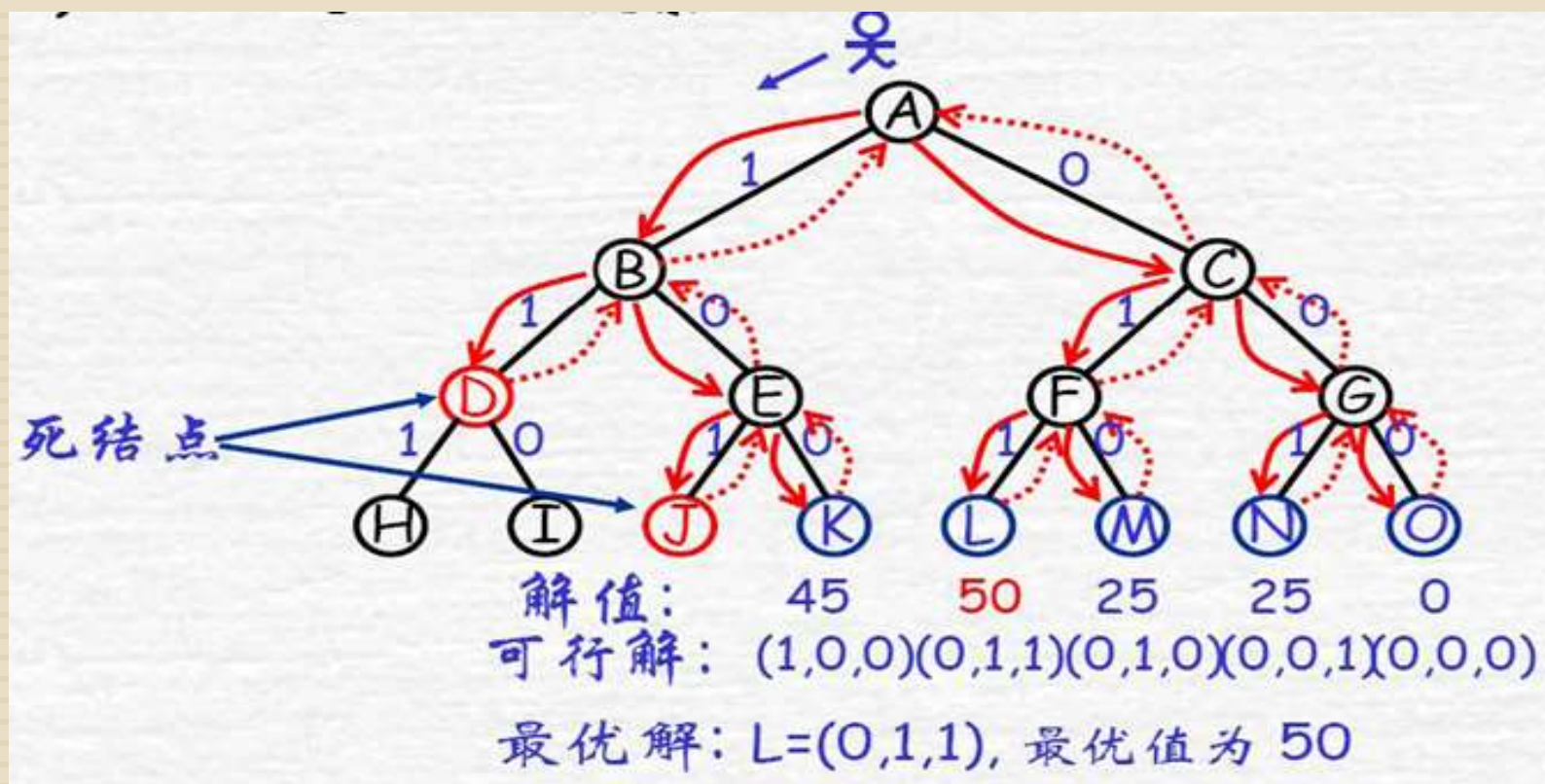
□ $n=3$, $w=[16,15,15]$, $p=[45,25,25]$, $c=30$



例：0-1背包问题

12

$n=3$, $w=(16, 15, 15)$, $v=(45, 25, 25)$, $c=30$



例：旅行售货员问题

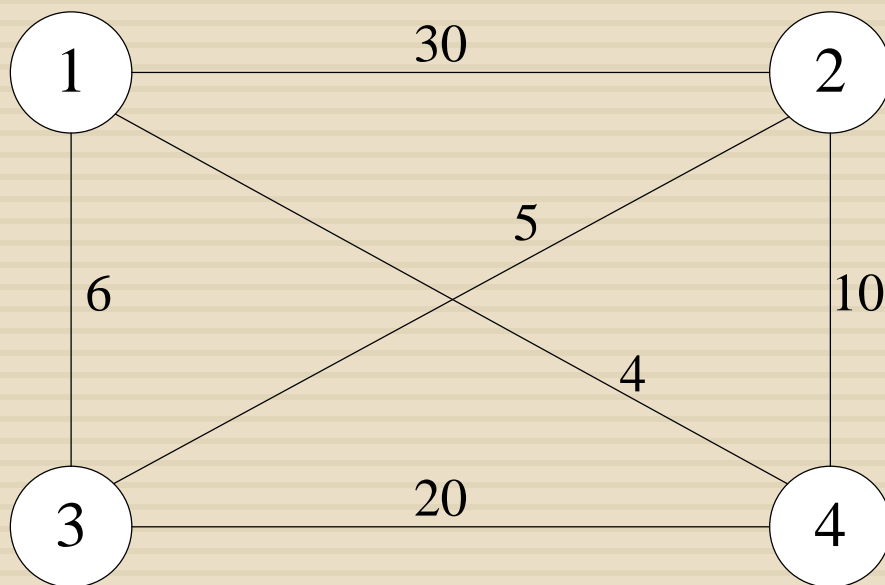
13

- Traveling Salesman Problem (TSP)
- 某售货员要到若干城市去推销商品，已知各城市之间的费用（路程或旅费）。他要选择一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的费用最少。

例：旅行售货员问题

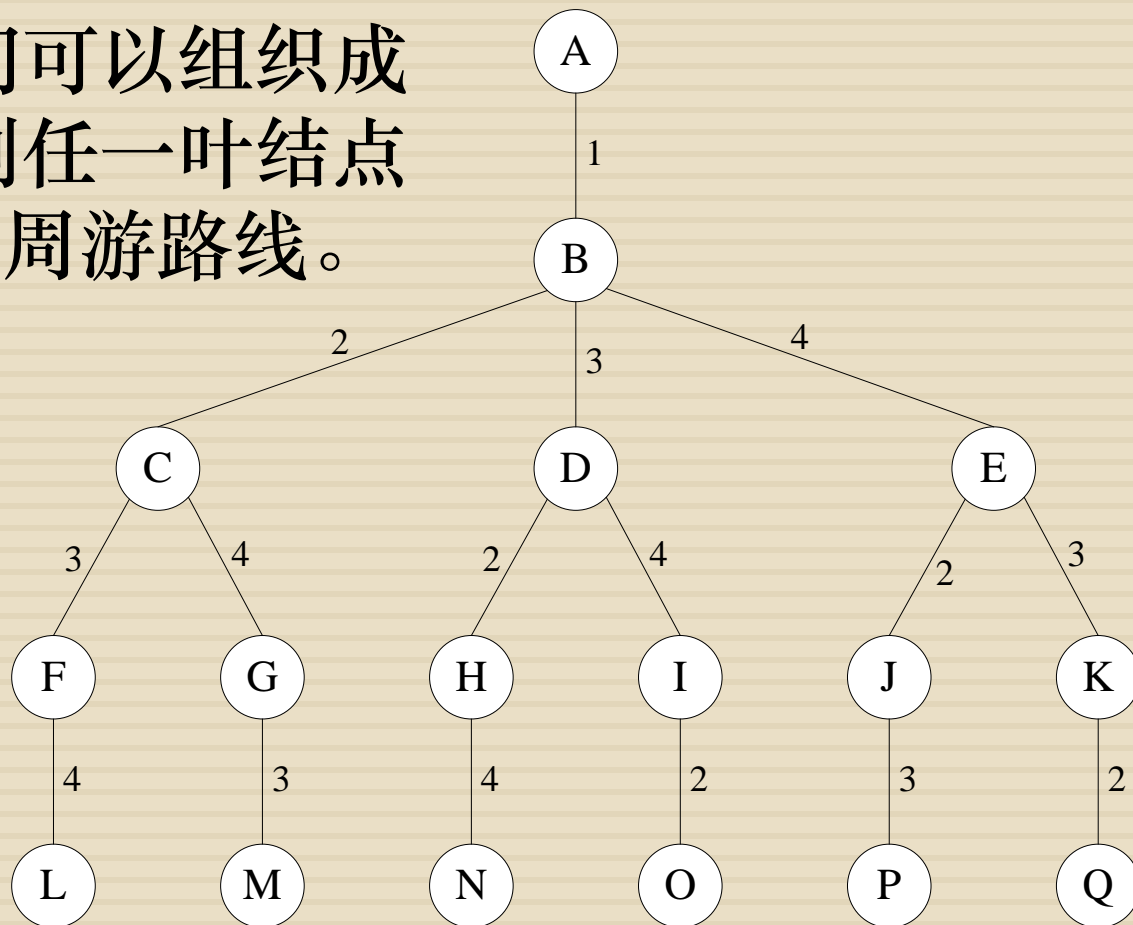
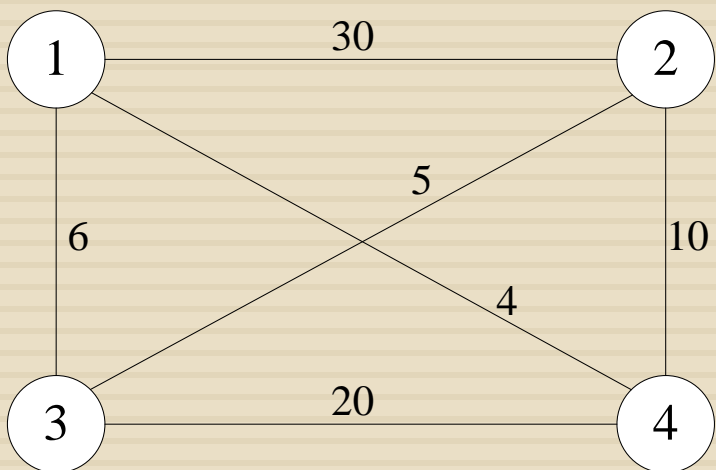
14

设 $G=(V,E)$ 是一个带权图。图中各边的费用（权）为正数。图中的一条周游路线是包括 V 中的每个顶点在内的一条回路，周游费用是这条边上所有边的费用之和。旅行售货员问题要在图 G 中找出费用最少的周游路线。



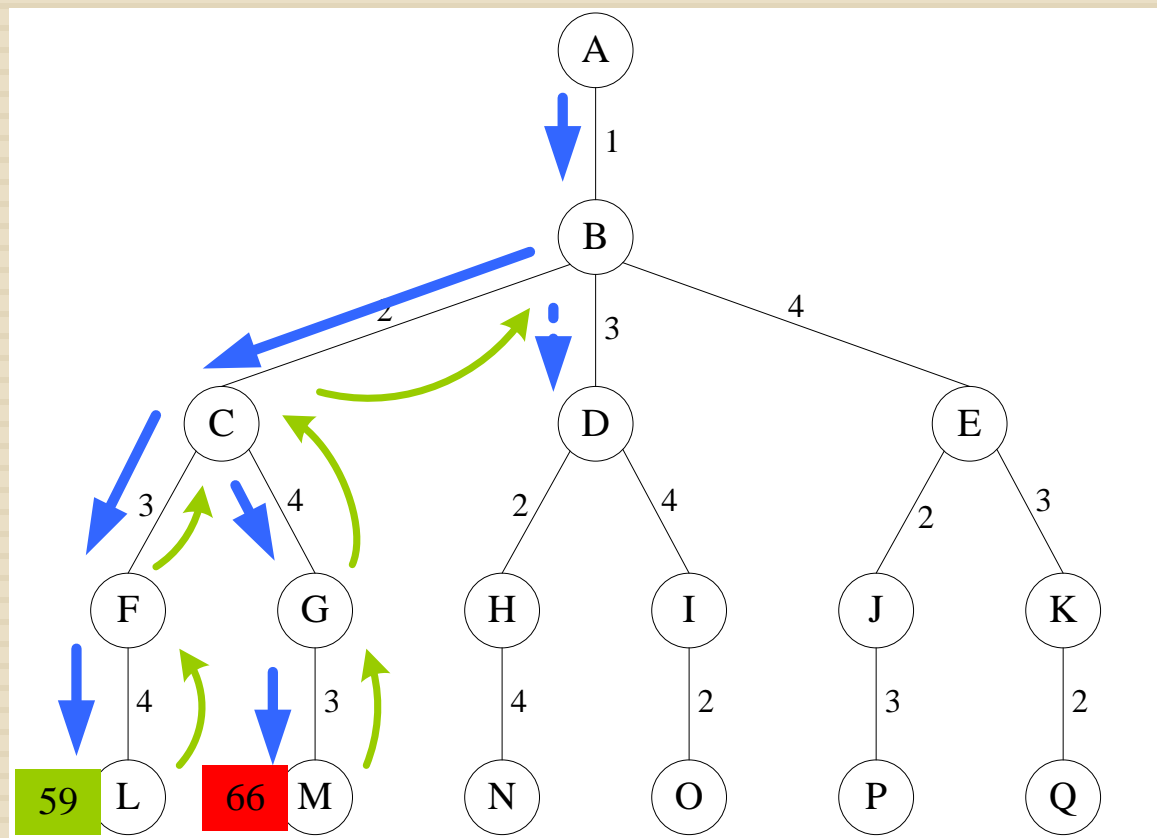
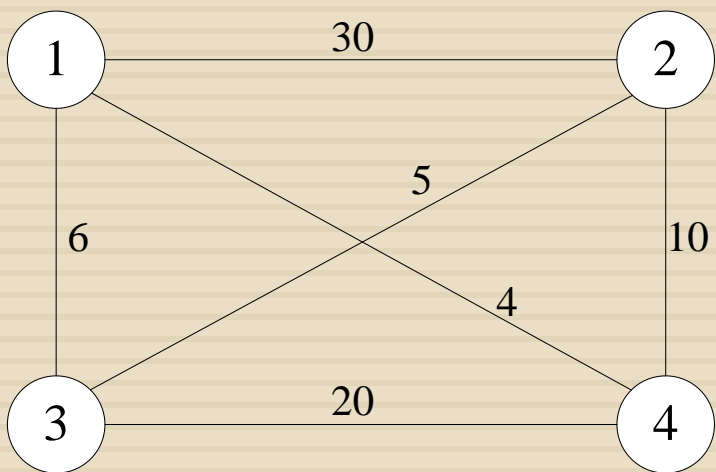
例：旅行售货员问题

旅行售货员问题的解空间可以组织成一棵树，从树的根结点到任一叶结点的路径定义了图G的一条周游路线。



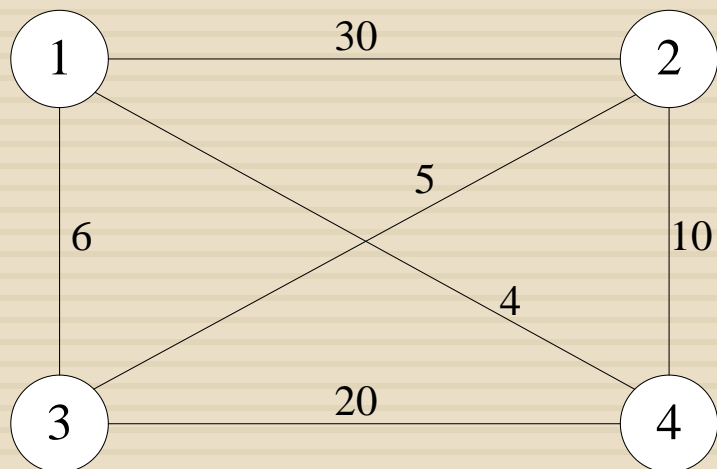
例：旅行售货员问题

16



例：旅行售货员问题

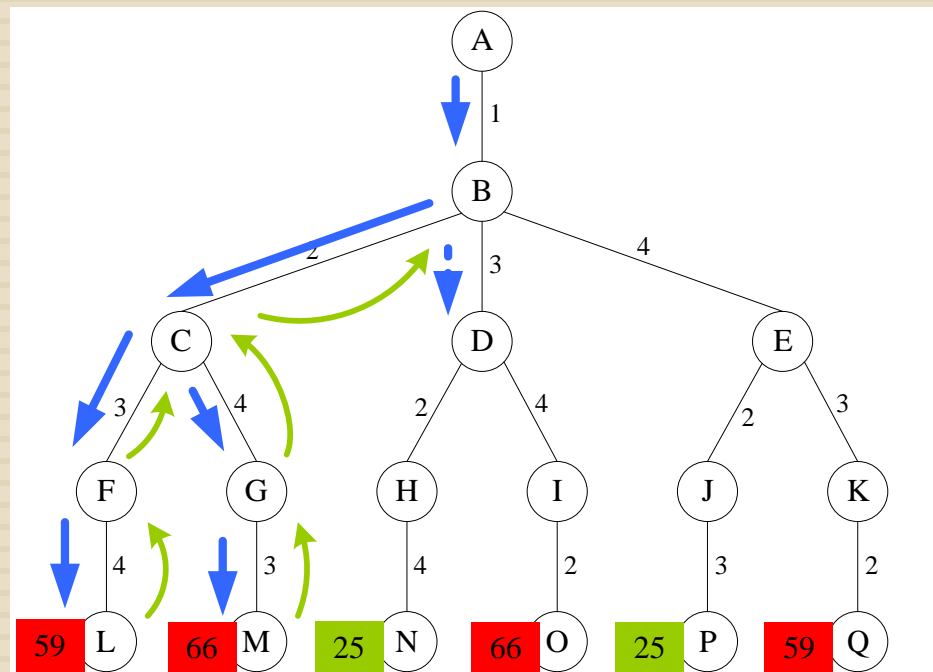
17



(1) 定义解空间: $X=\{12341,$
12431, 13241,
13421, 14231, 14321\}

(2) 构造解空间树

(3) 从A出发按DFS搜索整棵树



回溯法的基本思想

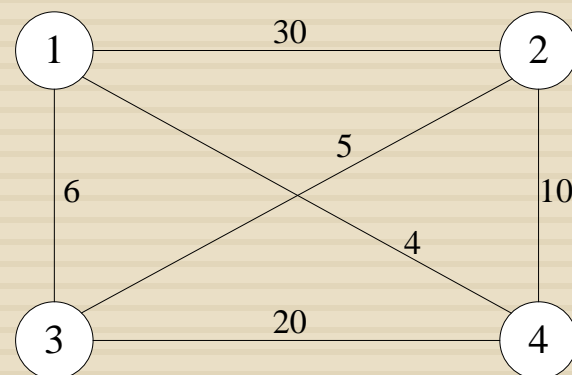
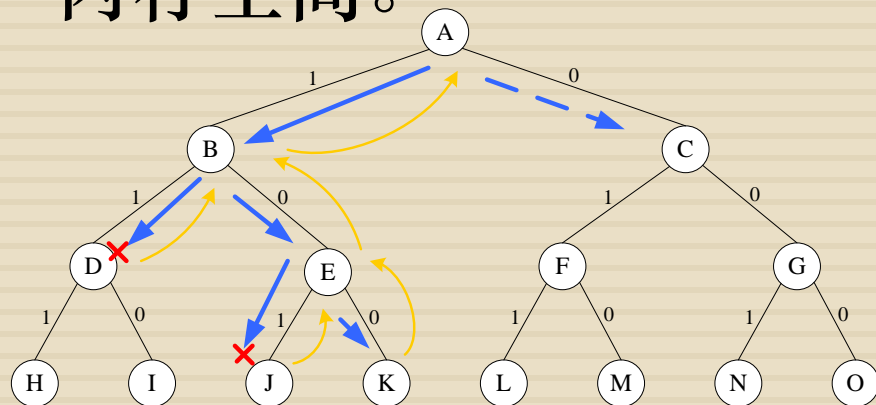
18

- (1) 针对所给问题，定义问题的**解空间**；
- (2) 确定易于搜索的**解空间结构**；
- (3) 以深度优先方式搜索解空间，并在搜索过程中用**剪枝函数**避免无效搜索。

常用剪枝函数：

用**约束函数**在扩展结点处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。

- 用回溯法解题的一个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 $h(n)$ ，则回溯法所需的计算空间通常为 $O(h(n))$ 。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 内存空间。



3. 递归回溯

20

```
void Backtrack (int t) //t为递归深度
{
    if (t>n)
        Output(x); //记录或输出可靠解x, x为数组
    else
        for (int i=f(n,t); i<=g(n,t); i++)
        {
            //f(n,t)表示在当前扩展结点处未搜索过的子树的起始编号
            //g(n,t)为终止编号
            x[t]=h(i); //h(i)表示当前扩展结点处x[t]的第i个可选值
            if (Constraint(t)&&Bound(t)) //剪枝
                Backtrack(t+1);
        }
}
```

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

4. 迭代回溯

21

```
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(t)&&Bound(t)){
                    if (solution(t)) //判断是否已得到可行解
                        Output(x);
                    else
                        t++;
                }
            }
        else
            t--;
    }
}
```

$f(n,t)$ 表示在当前扩展结点处未搜索过的子树的起始编号
 $g(n,t)$ 为终止编号
 $h(i)$ 表示当前扩展结点处 $x[t]$ 的第 i 个可选值

5. 子集树与排列树

22

- 前面两例中的两棵解空间树是用回溯法解题时常遇到的两类**典型**的解空间树。
- 当所给的问题是从 n 个元素的集合 S 中找出满足某种性质的子集时，相应的解空间树称为**子集树**。时间复杂度 $\Omega(2^n)$ 。
- 当所给的问题是确定 n 个元素满足某种性质的排列时，相应的解空间树称为**排列树**。时间复杂度 $\Omega(n!)$ 。

回溯法搜索子集树算法描述

23

```
void Backtrack (int t)
{
    if (t>n)
        Output(x);
    else
        for (int i=0; i<=1; i++)
        {
            x[t]=i;
            if (Constraint(t)&&Bound(t))
                Backtrack(t+1);
        }
}
```

回溯法搜索排列树算法描述

24

```
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]);
        }
}
```


5.2 装载问题

25

- 本装载问题是上一章最优装载问题的一个变形。

1. 问题描述

26

- 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些集装箱装上这2艘轮船。如果有，找出一种装载方案。

例：

27

- $n=3$, $c_1=c_2=50$, 且 $w=[10,40,40]$ 。
- 装载方案：
- 第一艘轮船装集装箱1和2；
- 第二艘轮船装集装箱3。

- 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 c_1 。由此可知，装载问题等价于以下特殊的0-1背包问题。

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

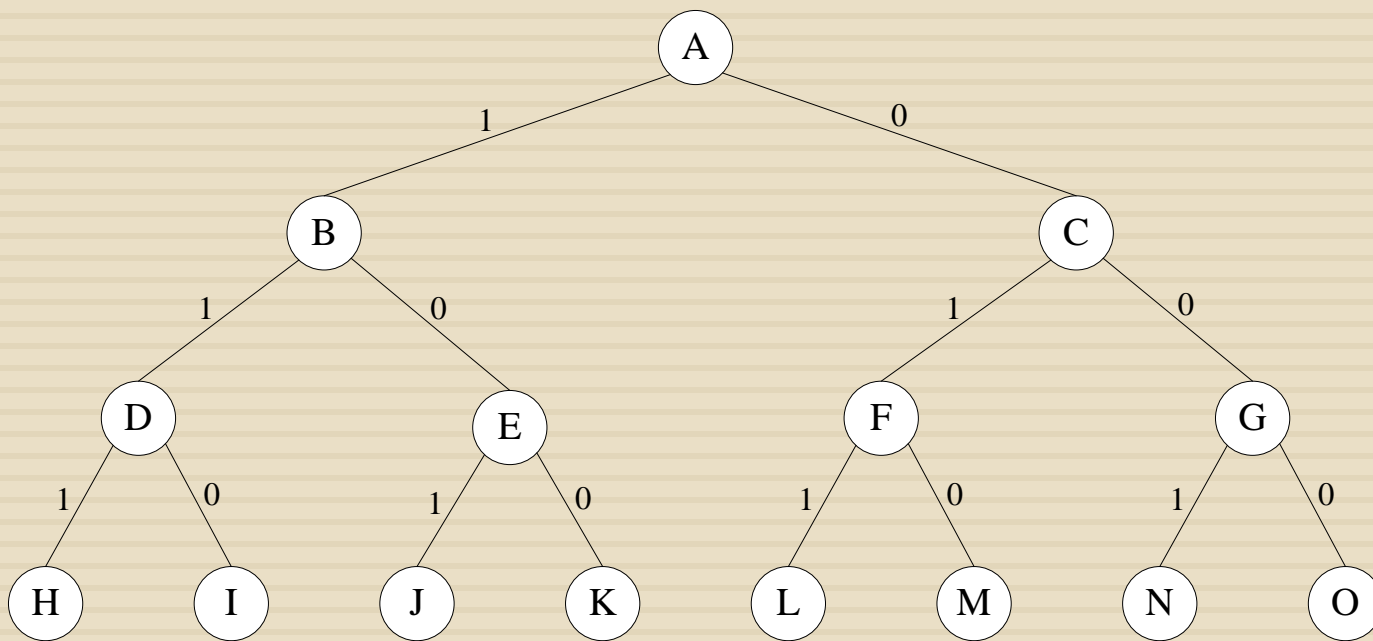
2. 算法设计

30

□ 解空间：子集树

□ 可行性约束函数(选择当前元素):

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



```
template<typename Type>
class Loading
{
    template<typename T>
    friend T MaxLoading(T [],T,int);

private:
    void Backtrack(int i);
    int n;    //集装箱数
    Type *w; //集装箱重量数组
    Type c;  //第1艘轮船的载重量
    Type cw; //当前载重量
    Type bestw; //当前最优载重量
};
```

```
template<typename Type>
void Loading<Type>::Backtrack(int i) //搜索第i层结点
{
    if(i>n) //到达叶结点
    {
        if(cw>bestw)
            bestw=cw;
        return;
    }
    if(cw+w[i]<=c) //进入左子树, x[i]=1
    {
        cw+=w[i];
        Backtrack(i+1); //继续搜索下一层
        cw-=w[i]; //退出左子树
    }
    Backtrack(i+1); //进入右子树, x[i]=0
}
```



```
template<typename Type>
Type MaxLoading(Type w[],Type c,int n) //返回最优载重量
{
    Loading<Type> X;
    X.w=w; //初始化X
    X.c=c;
    X.n=n;
    X.bestw=0;
    X.cw=0;
    X.Backtrack(1); //从第1层开始搜索
    return X.bestw;
}
```

```
int main()
{
    const int n=6;
    int c=80;
    int w[]={ 0,20,40,40,10,30,20}; //下标从1开始
    int s=MaxLoading(w,c,n);
    cout<<s<<endl;
    return 0;
}
```

算法在每个结点处花费 $O(1)$ 时间，子集树中结点个数为 $O(2^n)$ ，故算法的计算时间为 $O(2^n)$ 。

3. 上界函数

35

- 对于上一算法可引入一个上界函数，用于剪去不含最优解的子树。
- 上界函数(不选择当前元素):
当前载重量 cw +剩余集装箱的重量 $r \leq$ 当前最优载重量 $bestw$ 。

- `template<typename Type>`
- `class Loading`
- `{`
- `template<typename T>`
- `friend T MaxLoading(T [],T,int);`
- `private:`
- `void Backtrack(int i);`
- `int n; //集装箱数`
- `Type *w; //集装箱重量数组`
- `Type c; //第1艘轮船的载重量`
- `Type cw; //当前载重量`
- `Type bestw; //当前最优载重量`
- `Type r; //剩余集装箱重量`
- `};`

```
template<typename Type>
void Loading<Type>::Backtrack(int i) //搜索第i层结点
{
    if(i>n) //到达叶结点
    {
        if(cw>bestw)
            bestw=cw;
        return;
    }
    r-=w[i]; //剩余集装箱重量
```

```
if(cw+w[i]<=c) //进入左子树, x[i]=1
{
    cw+=w[i];
    Backtrack(i+1); //继续搜索下一层
    cw-=w[i]; //退出左子树
}
if(cw+r>bestw) //进入右子树, x[i]=0
    Backtrack(i+1);
r+=w[i];
}
```

```
template<typename Type>
Type MaxLoading(Type w[],Type c,int n) //返回最优载重量
{
    Loading<Type> X;
    X.w=w; //初始化X
    X.c=c;
    X.n=n;
    X.bestw=0;
    X.cw=0;
    X.r=0; //初始化r
    for(int i=1;i<=n;i++)
        X.r+=w[i];
    X.Backtrack(1); //从第1层开始搜索
    return X.bestw;
}
```

```
□ int main()  
□ {  
□     const int n=6;  
□     int c=80;  
□     int w[]={ 0,20,40,40,10,30,20}; //下标从1开始  
□     int s=MaxLoading(w,c,n);  
□     cout<<s<<endl;  
□     return 0;  
□ }
```


4. 构造最优解

41

- 为构造最优解，需在算法中记录与当前最优值相应的当前最优解。
- 在类Loading中增加两个私有数据成员：
 - ① `int* x`：用于记录从根至当前结点的路径；
 - ② `int* bestx`：记录当前最优解。
- 算法搜索到叶结点处，就修正bestx的值。

- `template<typename Type>`
- `class Loading`
- `{`
- `template<typename T>`
- `friend T MaxLoading(T [],T,int,int []);`
- `private:`
- `void Backtrack(int i);`
- `int n; //集装箱数`
- `int *x; //当前解`
- `int *bestx; //当前最优解`
- `Type *w; //集装箱重量数组`
- `Type c; //第1艘轮船的载重量`
- `Type cw; //当前载重量`
- `Type bestw; //当前最优载重量`
- `Type r; //剩余集装箱重量`
- `};`

```
template<typename Type>
void Loading<Type>::Backtrack(int i) //搜索第i层结点
```

```
{
```

```
    if(i>n) //到达叶结点
```

```
    {
```

```
        if(cw>bestw)
```

```
        {
```

```
            for(int j=1;j<=n;j++)
```

```
                bestx[j]=x[j];
```

```
            bestw=cw;
```

```
        }
```

```
        return;
```

```
    }
```

```
    r-=w[i]; //剩余集装箱重量
```

```
□ if(cw+w[i]<=c) //进入左子树, x[i]=1
□ {
□   x[i]=1; //装第i个集装箱
□   cw+=w[i];
□   Backtrack(i+1); //继续搜索下一层
□   cw-=w[i]; //退出左子树
□ }
□ if(cw+r>bestw) //进入右子树, x[i]=0
□ {
□   x[i]=0; //不装第i个集装箱
□   Backtrack(i+1);
□ }
□ r+=w[i];
□ }
```

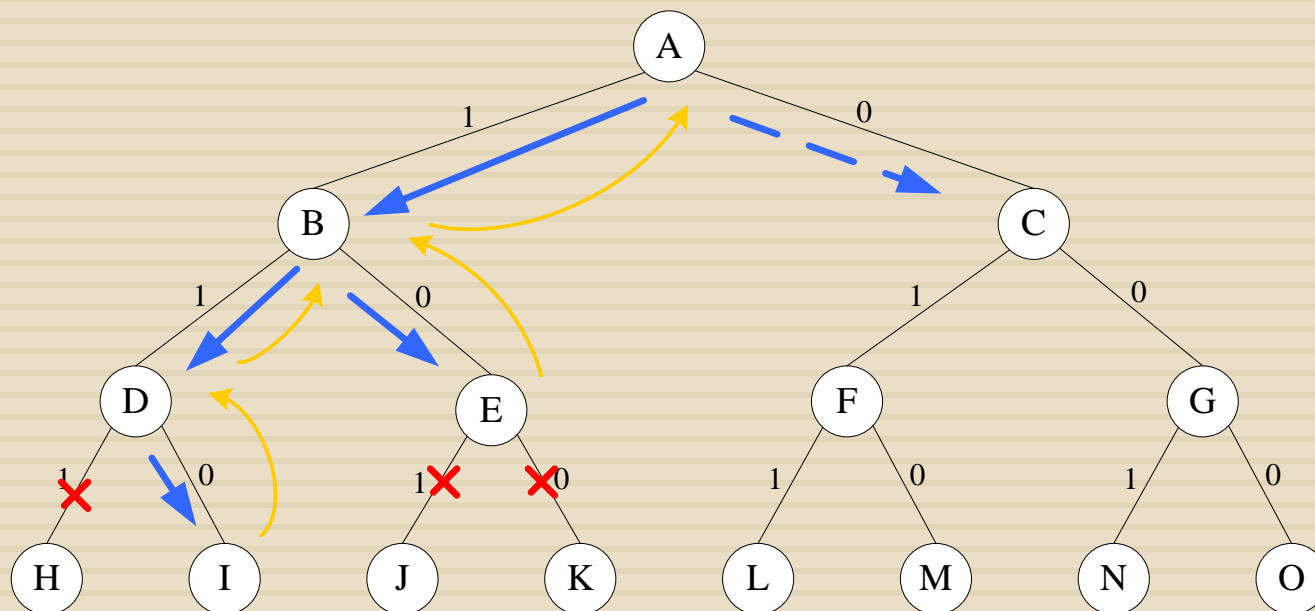
```
template<typename Type>
Type MaxLoading(Type w[],Type c,int n,int bestx[]) //返回最优载重量
{
    Loading<Type> X;
    X.w=w; //初始化X
    X.x=new int[n+1];
    X.c=c;
    X.n=n;
    X.bestx=bestx;
    X.bestw=0;
    X.cw=0;
    X.r=0; //初始化r
    for(int i=1;i<=n;i++)
        X.r+=w[i];
    X.Backtrack(1); //从第1层开始搜索
    delete[] X.x;
    return X.bestw;
}
```

```
□ int main()
□ {
□     const int n=6;
□     int c=80;
□     int w[]={ 0,20,40,40,10,30,20}; //下标从1开始
□     int bestx[n+1];
□     int s=MaxLoading(w,c,n,bestx);
□     cout<<s<<endl;
□     for(int i=1;i<=n;i++)
□         cout<<bestx[i]<<' ';
□     cout<<endl;
□     return 0;
□ }
```

由于bestx可能被更新 $O(2^n)$ 次，故算法的时间复杂性为 $O(n2^n)$ 。

5. 迭代回溯

- 由于数组x记录了解空间树中从根到当前扩展结点的路径，利用这些信息，可将上述回溯法表示成非递归的形式。
- $n=3, c1=c2=50$ ，且 $w=[10,40,40]$



//迭代回溯法，返回最优载重量

```
template<typename Type>
```

```
Type MaxLoading(Type w[],Type c,int n,int bestx[])
```

```
{
```

```
    //初始化根结点
```

```
    int i=1;
```

```
    int *x=new int[n+1];
```

```
    Type bestw=0;
```

```
    Type cw=0;
```

```
    Type r=0;
```

```
    for(int j=1;j<=n;j++)
```

```
        r+=w[j];
```



```
while(true) //搜索子树
```

```
{
```

```
    while(i<=n&&cw+w[i]<=c) //进入左子树，条件为真，则一直往左搜索
```

```
    {
```

```
        r-=w[i];
```

```
        cw+=w[i];
```

```
        x[i]=1;
```

```
        i++;
```

```
    }
```

```
    if(i>n) //到达叶结点
```

```
    {
```

```
        for(int j=1;j<=n;j++)
```

```
            bestx[j]=x[j];
```

```
        bestw=cw;
```

```
    }
```

```
    else //进入右子树
```

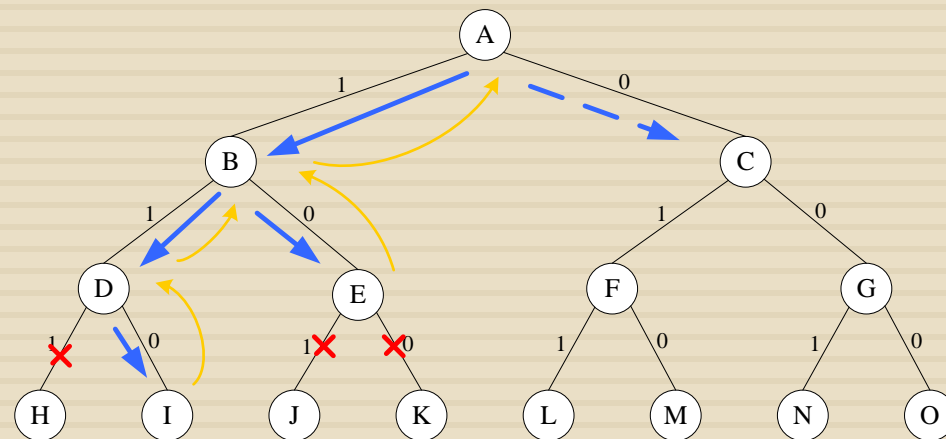
```
    {
```

```
        r-=w[i];
```

```
        x[i]=0;
```

```
        i++;
```

```
    }
```



```
while(cw+r<=bestw) //剪枝回溯
```

```
{
```

```
  i--;
```

```
  while(i>0&&!x[i]) //从右子树返回
```

```
  {
```

```
    r+=w[i];
```

```
    i--;
```

```
  }
```

```
  if(i==0) //如返回到根，则结束
```

```
  {
```

```
    delete[] x;
```

```
    return bestw;
```

```
  }
```

```
  //进入右子树
```

```
  x[i]=0;
```

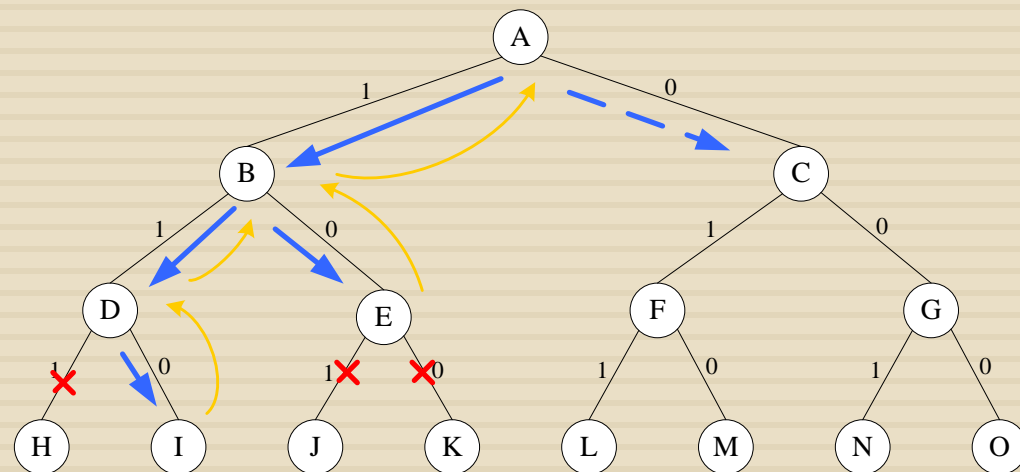
```
  cw-=w[i];
```

```
  i++;
```

```
}
```

```
}
```

```
}
```



算法的计算时间为 $O(2^n)$ 。

5.3 批处理作业调度

1. 问题描述

51

- 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。
- 注：作业 i 在机器 j 上完成处理的时间 F_{ji} 指的是计时从0开始，第 F_{ji} 时刻完成。

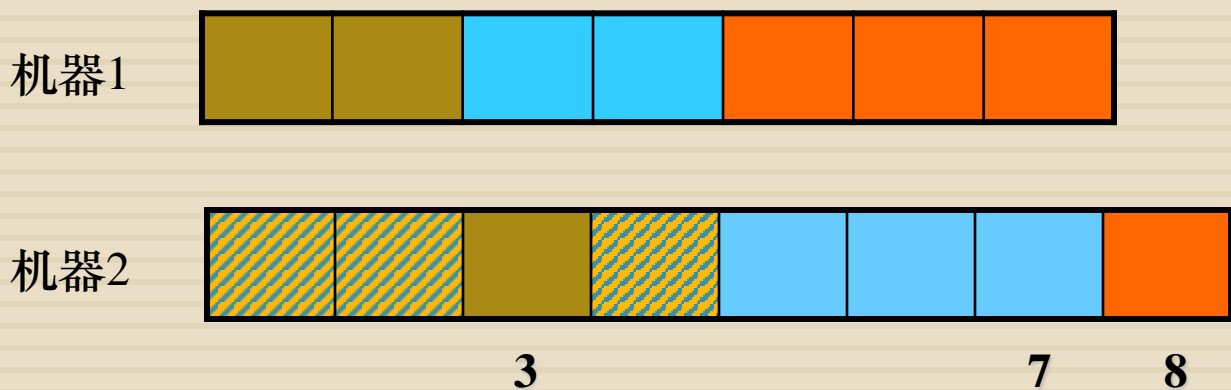
- 批处理作业调度问题要求对于给定的 n 个作业，制定最佳作业调度方案，使其完成时间和达到最小。

例:

53

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。易知, 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要从 n 个作业的所有排列中找出有最小完成时间和的作业调度。
- 解空间：排列树。
- 设开始时 $x=[1,2,\dots,n]$ 是所给的 n 个作业，则相应的排列树由 $x[1:n]$ 的所有排列构成。

```
int number;//作业数量
int x1[10];//作业在机器1运行的时间
int x2[10];//作业在机器2运行的时间
int sum = 0;//作业完成的总时间
int bestsum = MAX;//作业完成的最优时间
int order[10];//作业完成的次序,要用于交换
int bestorder[10];//作业完成的最优的顺序
int f1 = 0;//机器1累计的时间
int f2[10];//作业在机器2处理完累计的时间, 即每一个作业的调度时间
int vis[10];//记录作业以否已被选
```

```
void backtrack(int cur)
{
    //到达边界
    if(cur >= number)
    {
        for(int i=0; i<number; i++)
        {
            bestorder[i] = order[i];
        }
        bestsum = sum;
    }
}
```



```
□ else{  
□     //cur表示正在赋值的位置, cur+1去到下一层子节点, i递增, 在当前层遍历  
    兄弟节点  
□     for(int i=cur; i<number; i++)  
□     {  
□         f1 += x1[order[i]];  
□         if (cur > 0)  
□         {  
□             f2[cur] = (f2[cur - 1] > f1 ? f2[cur - 1] : f1) + x2[order[i]];  
□         }  
□         else  
□         {  
□             f2[cur] = f1 + x2[order[i]];  
□         }  
□     }
```

```
sum += f2[cur];  
swap(order[cur], order[i]);  
if(sum < bestsum){//剪枝，如果当前sum都大于bestsum了，则不再遍历此节点  
    backtrack(cur+1);  
}  
swap(order[cur], order[i]);  
sum -= f2[cur];  
f1 -= x1[order[i]];  
}
```

```
int main() {  
    number = 3;  
    x1[0] = 2;x1[1] = 3;x1[2] = 2;x2[0] = 1;x2[1] = 1;x2[2] = 3;  
    //初始化第一个序列, 从1开始到number  
    for(int i=0; i<number; i++)  
        order[i] = i;  
    backtrack(0);  
    cout<<"最节省的时间为: "<<bestsum<< endl;  
    cout<<"对应的方案为: ";  
    for(int i=0;i<number;i++)  
        cout<<bestorder[i]<<" ";  
    cout<< endl;  
    system("pause");  
}
```

算法Backtrack在每个结点处耗费 $O(1)$ 计算时间, 故最坏情况下, 整个算法的时间复杂性为 $O(n!)$ 。

5.4 符号三角形问题

1. 问题描述

60

- 下面由“+”、“-”组成的符号三角形中，2个同号下面都是“+”号，2个异号下面都是“-”号。

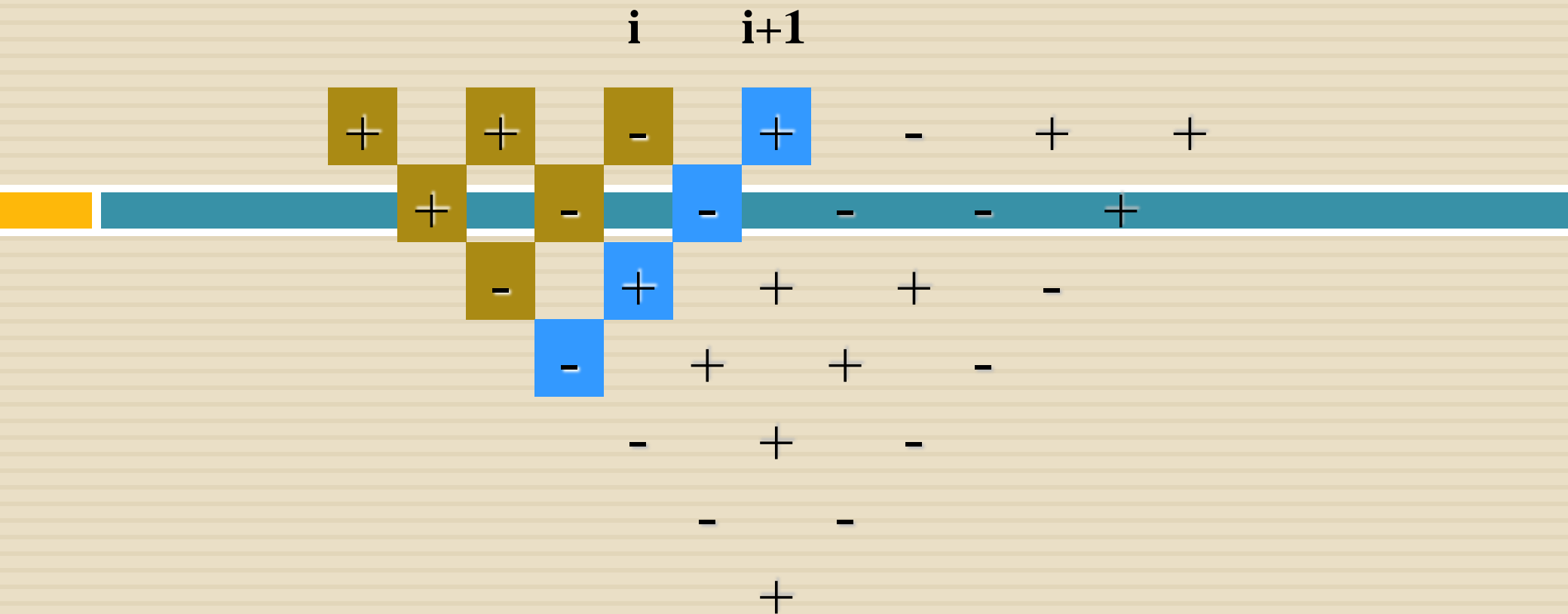
+	+	-	+	-	+	+
	+	-	-	-	-	+
		-	+	+	+	-
			-	+	+	-
				+	-	
				-	-	
					+	

- 问题：对于第一行有 n 个符号的符号三角形，计算有多少个不同的符号三角形，其所含的“+”和“-”的个数相同。

2. 算法设计

62

- 解向量：用 n 元组 $x[1:n]$ 表示符号三角形的第一行。
 - ▣ $x[i]=1$ ，表示第1行第 i 个符号为“+”；
 - ▣ $x[i]=0$ ，表示第1行第 i 个符号为“-”。
- 可以用一棵完全二叉树表示其解空间。



- 在 $x[1:i]$ 确定后，就确定了一个由 $i(i+1)/2$ 个符号组成的符号三角形。
- $x[i+1]$ 的值确定后，只要在已确定的三角形右边加一条边，即可得到 $x[1:i+1]$ 所对应的符号三角形。

- 对于给定的 n ，符号总个数为 $n(n+1)/2$ 。
- **约束条件**：当前三角形中“+”与“-”的个数均不超过 $n(n+1)/4$ 。
- 当 $n(n+1)/2$ 为奇数时，显然不是满足条件的符号三角形。


```
□ class Triangle
□ {
□     friend int Compute(int);
□ private:
□     void Backtrack(int t);
□     int n;    //第一行的符号个数
□     int half; //n*(n+1)/4
□     int count; //当前 “+”号个数
□     int **p;  //符号三角形矩阵
□     long sum; //已找到的符号三角形数
□ };
```

```
void Triangle::Backtrack(int t) //第1行有t个符号
{
    if((count>half)||((t*(t-1)/2-count)>half) //"+"或"-"的个数超过n(n+1)/4
        return;
    if(t>n)
        sum++;
    else
        for(int i=0;i<2;i++) //每一位上取值只有两种情况"-" (0) 或"+" (1)
        {
            p[1][t]=i; //第一行的第t位的符号为i
            count+=i; //累记" +"的个数
```

```
for(int j=2;j<=t;j++) //计算新增边的其余各点
```

```
{
```

```
    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++) //恢复计数，为t位为下一种情况作准备
```

```
    count-=p[j][t-j+1];
```

```
count-=i;
```

```
}
```

```
}
```

	1	1	0	1	0	1	1
	1	0	0	0	0	1	
	0	1	1	1	0		
	0	1	1	0			
	0	1	0				
	0	0					
	1						

```
□ int Compute(int n)
□ {
□   Triangle X;
□   X.n=n;
□   X.count=0;
□   X.sum=0;
□   X.half=n*(n+1)/2;
□   if(X.half%2==1) //n(n+1)/2为奇数时，无解
□       return 0;
□   X.half=X.half/2;
```

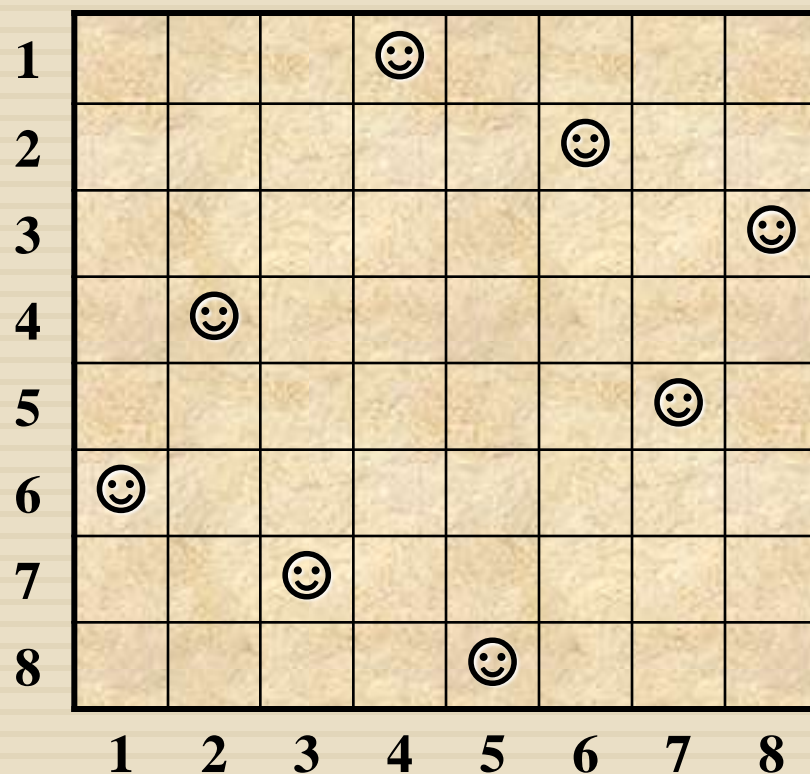
- `int **p=new int*[n+1];`
- `for(int i=0;i<=n;i++)`
- `p[i]=new int[n+1];`
- `for(int i=0;i<=n;i++)`
- `for(int j=0;j<=n;j++)`
- `p[i][j]=0;`
- `X.p=p;`
- `X.Backtrack(1);`
- `return X.sum;`
- `}`

5.5 n后问题

1. 问题描述

70

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。



2. 算法设计

71

□ 解向量： (x_1, x_2, \dots, x_n)

用 n 元组 $x[1:n]$ 表示 n 后问题的解，其中 $x[i]$ 表示皇后 i 放在棋盘的第 i 行的第 $x[i]$ 列。

□ 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： $|i-j| \neq |x_i - x_j|$

- 回溯法解n后问题时，用完全n叉树表示解空间，用可行性约束Place()剪去不满足行、列和斜线约束的子树。
- Backtrack(i)搜索解空间中第i层子树。
- sum记录当前已找到的可行方案数。

四皇后问题

73

□ 问题描述

- ▣ 在4 x 4棋盘上放上4个皇后，使皇后彼此不受攻击。不受攻击的条件是彼此不在同行（列）、斜线上。求出全部的放法。

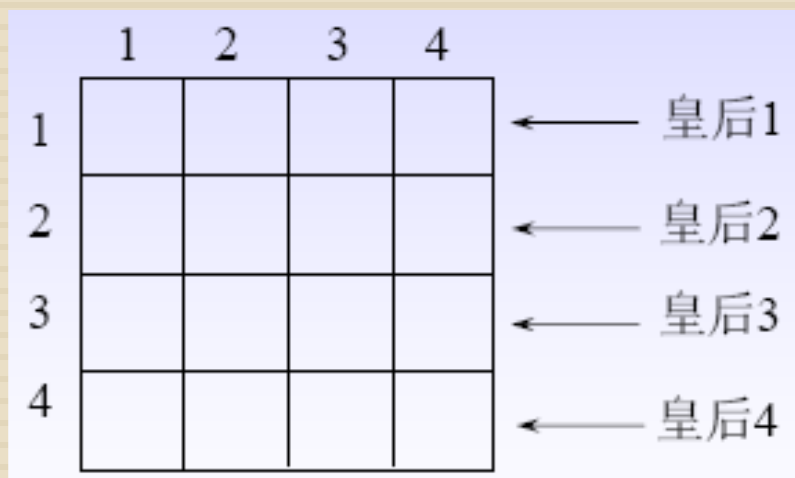
□ 解表示

- ▣ **解向量**：4元向量 $X=(x_1, x_2, x_3, x_4)$ ， x_i 表示皇后i放在i行上的列号，如(3,1,2,4)
- ▣ **解空间**： $\{(x_1, x_2, x_3, x_4) \mid x_i \in S, i=1\sim 4\}$ $S=\{1,2,3,4\}$
- ▣ **可行性约束函数**
 - 显约束： $x_i \in S, i=1\sim 4$
 - 隐约束($i \neq j$): $x_i \neq x_j$ (不在同一列)
 $|i - x_i| \neq |j - x_j|$ (不在同一斜线)

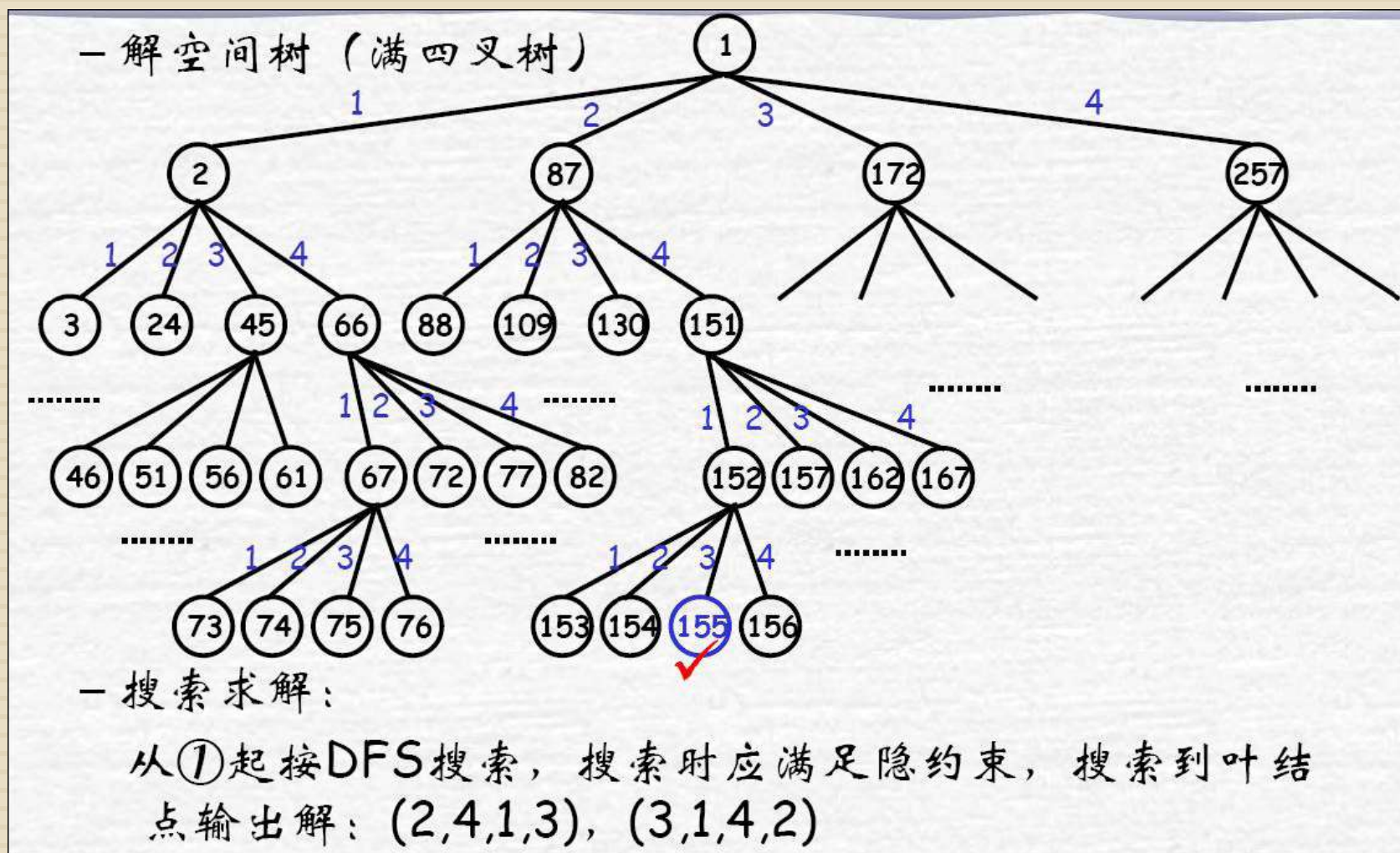
四皇后问题

74

- 四皇后问题的解空间树是一棵**完全4叉树**，树的根结点表示搜索的初始状态，从根结点到第2层结点对应皇后1在棋盘中第1行的可能摆放位置，从第2层结点到第3层结点对应皇后2在棋盘中第2行的可能摆放位置，依此类推。



四皇后问题



四皇后问题

76

□ 回溯法求解4皇后问题的搜索过程

Q			

(a)

Q			
×	×	Q	

(b)

Q			
×	×	Q	
×	×	×	×

(c)

Q			
			Q

(d)

Q			
			Q
×	Q		

(e)

Q			
			Q
×	Q		
×	×	×	×

(f)

	Q		

(g)

	Q		
×	×	×	Q

(h)

	Q		
			Q
Q			

(i)

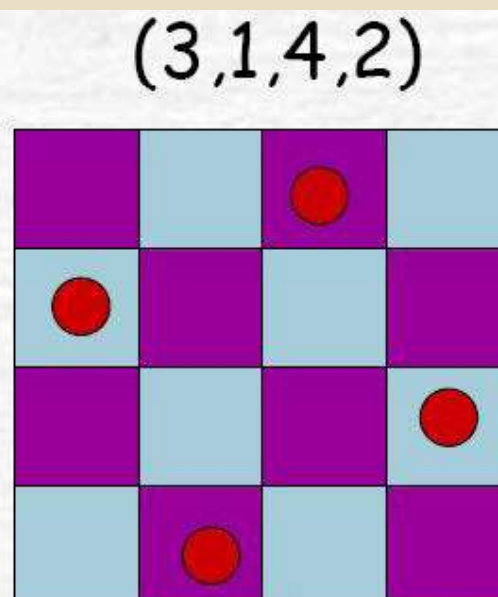
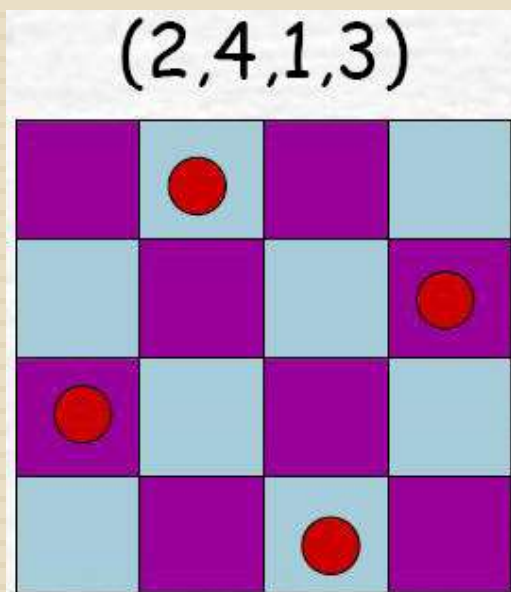
	Q		
			Q
Q			
×	×	Q	

(j)

四皇后问题

77

□ 输出解



- #include <iostream>
- #include <cstdlib> //声明了abs(int)
- using namespace std;

- class Queen
- {
- friend int nQueen(int);
- private:
- bool Place(int k);
- void Backtrack(int t);
- int n; //皇后个数
- int *x; //当前解
- long sum; //当前已找到的可行方案数
- };

```
bool Queen::Place(int k) //第k行
{
    for(int j=1;j<k;j++)
        if((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k]))
            //在同一斜线或同一列上
            return false;
    return true;
}
```

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


- `int nQueen(int n)`

- `{`

- `Queen X;`

- `//初始化X`

- `X.n=n;`

- `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;`

- `}`

3. 迭代回溯

82

- 数组x记录了解空间树中从根到当前扩展结点的路径，利用这些信息可将递归回溯法表示成非递归形式。

```
void Queen::Backtrack()
{
    x[1]=0;
    int k=1;
    while(k>0)
    {
        x[k]+=1; //第k行的放到下一列
        //x[k]不能放置，则放到下一列，直到可放置
        while((x[k]<=n)&&!Place(k))
            x[k]+=1;
```

if(x[k]<=n) //放在n列范围内

if(k==n) //已放n行

sum++;

else //不足n行

{

k++; //放下一行

x[k]=0; //下一行又从第0列的下列开始试放

}

else //第k行无法放置，则重新放置上一行（放到下一列）

k--;

}

}

5.6 0-1背包问题

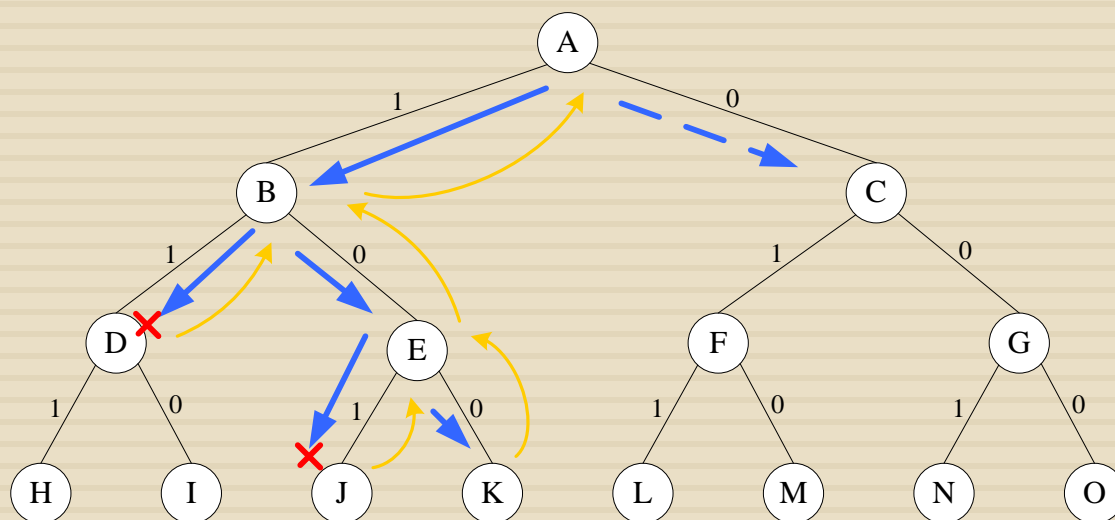
1. 算法描述

85

解空间：子集树

0-1背包问题是子集选取问题，其解空间可用子集树表示。

可行性约束函数：
$$\sum_{i=1}^n w_i x_i \leq c_1$$



□ 上界约束:

当右子树中有可能包含最优解时才进入右子树搜索，否则剪去右子树。

- 设 r 是当前剩余物品价值总和， cp 是当前价值， $bestp$ 是当前最优价值，当 $cp+r \leq bestp$ 时，剪去右子树。

- 计算右子树中解的上界更好的方法是将剩余的物品依其单位重量价值排序，然后依次装入物品，直到装不下时，再装入该物品一部分而装满背包，由此得到的价值是右子树中解的上界。

- template<typename Typew,typename Typep>
- class Knap
- {
- friend Typep Knapsack<>(Typep *,Typew *,Typew,int);
- //<>指明友员函数为模板函数
- private:
- Typep Bound(int i); //计算上界
- void Backtrack(int i);
- Typew c; //背包容量
- int n; //物品数
- Typew *w; //物品重量数组
- Typep *p; //物品价值数组
- Typew cw; //当前重量
- Typep cp; //当前价值
- Typep bestp; //当前最优价值
- };


```
template<typename Typew,typename Typep>
void Knap<Typew,Typep>::Backtrack(int i) //回溯
{
    if(i>n)
    {
        bestp=cp;
        return;
    }
```

```
□ if(cw+w[i]<=c) //进入左子树
□ {
□     cw+=w[i];
□     cp+=p[i];
□     Backtrack(i+1);
□     cw-=w[i];
□     cp-=p[i];
□ }
□ if(Bound(i+1)>bestp) //进入右子树
□     Backtrack(i+1);
□ }
```

```
template<typename Typew,typename Typep>
```

```
Typep Knap<Typew,Typep>::Bound(int i) //计算上界
```

```
{
```

```
    Typew cleft=c-cw; //剩余的背包容量
```

```
    Typep b=cp; //b为当前价值
```

```
    while(i<=n&&w[i]<=cleft) //依次装入单位重量价值高的整个物品
```

```
    {
```

```
        cleft-=w[i];
```

```
        b+=p[i];
```

```
        i++;
```

```
    }
```

```
    if(i<=n) //装入物品的一部分
```

```
        b+=p[i]*cleft/w[i];
```

```
    return b; //返回上界
```

```
}
```

```
❑ class Object //物品类
❑ {
❑     friend int Knapsack(int *,int *,int,int);
❑ public:
❑     int operator <(Object a) const
❑     {
❑         return (d>a.d);
❑     }
❑     int ID; //物品编号
❑     float d; //单位重量价值
❑ };
```

```
template<typename Typew,typename Typep>
Typep Knapsack(Typep p[],Typew w[],Typew c,int n)
{
    Typew W=0; //总重量
    Typep P=0; //总价值
    Object* Q=new Object[n]; //创建物品数组，下标从0开始
    for(int i=1;i<=n;i++) //初始物品数组数据
    {
        Q[i-1].ID=i;
        Q[i-1].d=1.0*p[i]/w[i];
        P+=p[i];
        W+=w[i];
    }
}
```

- `if(W<=c) //能装入所有物品`
- `return P;`

- `QuickSort(Q,0,n-1); //依物品单位重量价值非增排序`

- `Knap<Typew,Typep> K;`
- `K.p=new Typep[n+1];`
- `K.w=new Typew[n+1];`
- `for(int i=1;i<=n;i++)`
- `{`
- `K.p[i]=p[Q[i-1].ID];`
- `K.w[i]=w[Q[i-1].ID];`
- `}`

```
K.cp=0;  
K.cw=0;  
K.c=c;  
K.n=n;  
K.bestp=0;  
K.Backtrack(1);  
delete[] Q;  
delete[] K.w;  
delete[] K.p;  
return K.bestp;  
}
```

2. 算法效率

96

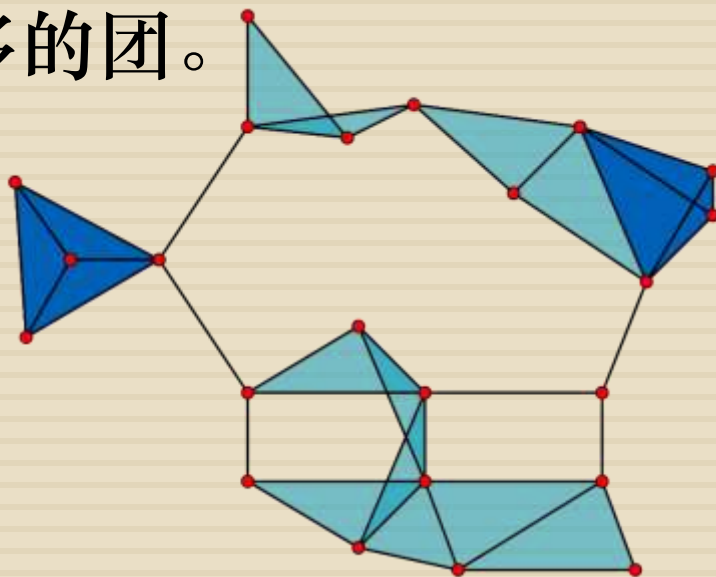
- 计算上界需要 $O(n)$ 时间，最坏情况下有 $O(2^n)$ 个右儿子结点需要计算上界，故算法所需要的时间为 $O(n2^n)$ 。

5.7 最大团问题

1. 问题描述

97

- 给定无向图 $G=(V,E)$ 。如果 $U\subseteq V$ ，且对任意 $u,v\in U$ 有 $(u,v)\in E$ ，即 U 中任意两点之间有边相连，则称 U 是 G 的完全子图。
- 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。 G 的最大团是指 G 中所含顶点数最多的团。

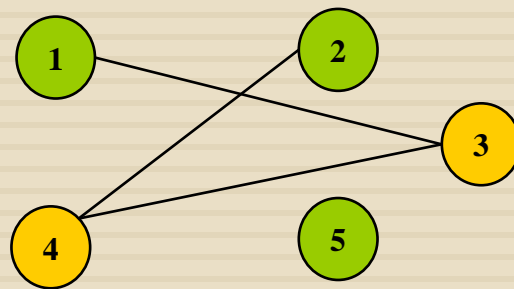
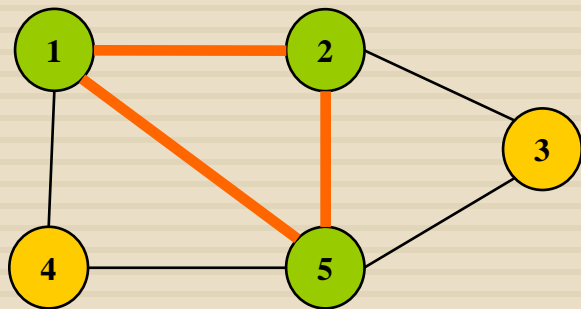


- A graph with 23×1 -vertex cliques (the vertices),
- 42×2 -vertex cliques (the edges),
- 19×3 -vertex cliques (light and dark blue triangles), and
- 2×4 -vertex cliques (dark blue areas).

The 11 light blue triangles form cliques. The two dark blue 4-cliques are both maximum and maximal, and the clique number of the graph is 4.

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**。
 G 的空子图 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。
 对于任一无向图 $G=(V, E)$ 其补图 $\overline{G}=(V_1, E_1)$ 定义为： $V_1=V$ ，
 且 $(u, v) \in E_1$ 当且仅当 $(u, v) \notin E$ 。

U 是 G 的最大团当且仅当 U 是 \overline{G} 的最大独立集。



2. 算法设计

99

- 解空间：子集树
- 图 G 的最大团问题都可以看作是图 G 的顶点集 V 的子集选取问题，因此可用子集树表示问题的解空间。

- 设当前扩展结点 Z 位于解空间树的第 i 层。
- **可行性约束函数**：顶点 i 到已选入的顶点集中每一个顶点都有边相连。
- **上界函数**：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。

```
class Clique
{
    friend int MaxClique(int **,int [],int);
public:
    void Print(); //输出最优解
private:
    void Backtrack(int i);
    int **a; //图G的邻接矩阵，下标从1开始
    int n; //图G的顶点数
    int *x; //当前解
    int *bestx; //当前最优解
    int cn; //当前顶点数
    int bestn; //当前最大顶点数
};
```

```
void Clique::Backtrack(int i)
```

```
{
```

```
    if(i>n)
```

```
    {
```

```
        for(int j=1;j<=n;j++)
```

```
            bestx[j]=x[j];
```

```
        bestn=cn;
```

```
        return;
```

```
    }
```

```
    int OK=1; //用来判断第i个顶点是否与已选顶点都有边相连
```

```
    for(int j=1;j<i;j++) //x[1:i-1], 已入选的顶点集
```

```
        if(x[j]&& a[i][j]==0) //i与当前团中的顶点无边相连
```

```
        {
```

```
            OK=0;
```

```
            break; //只要与当前团中一个顶点无边相连，则中止
```

```
        }
```

```
if(OK) //进入左子树
```

```
{
```

```
    x[i]=1; //i放入当前团中
```

```
    cn++;
```

```
    Backtrack(i+1);
```

```
    x[i]=0;
```

```
    cn--;
```

```
}
```

```
if(cn+n-i>bestn) //如有可能在右子树中找到更大的团，则进入右子树
```

```
{
```

```
    x[i]=0;
```

```
    Backtrack(i+1);
```

```
}
```

```
}
```

```
int MaxClique(int **a,int v[],int n)
{
    Clique Y;
    Y.x=new int[n+1];
    Y.a=a;
    Y.n=n;
    Y.cn=0;
    Y.bestn=0;
    Y.bestx=v;
    Y.Backtrack(1);
    delete[] Y.x;
    Y.Print();
    return Y.bestn;
}
```

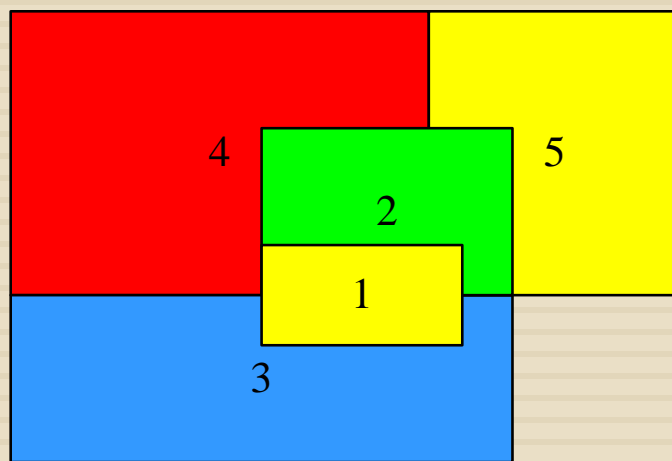
计算时间：
 $O(n2^n)$

5.8 图的m着色问题

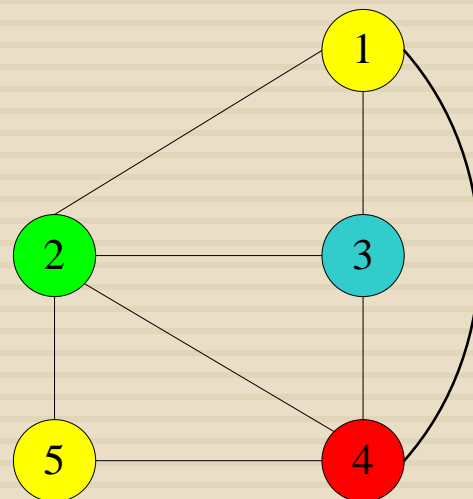
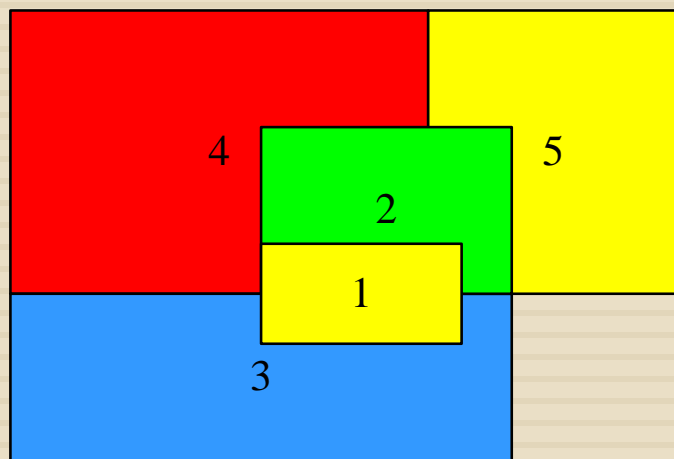
1. 问题描述

105

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。



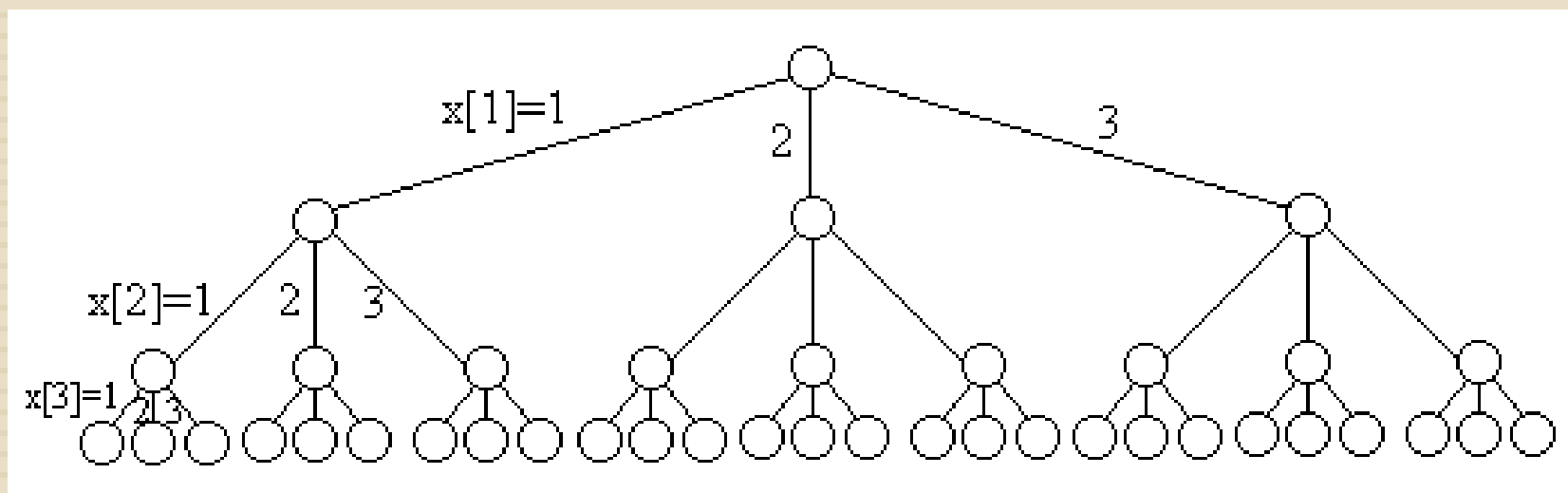
这个问题是图的 m 可着色判定问题。若一个图最少需要 m 种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数 m 为该图的色数。求一个图的色数 m 的问题称为图的 m 可着色优化问题。



2. 算法设计

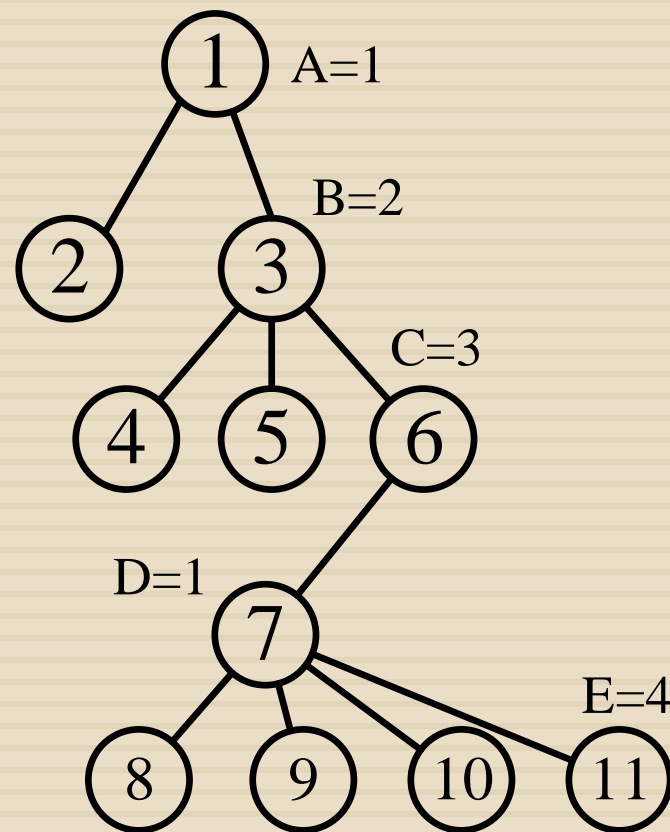
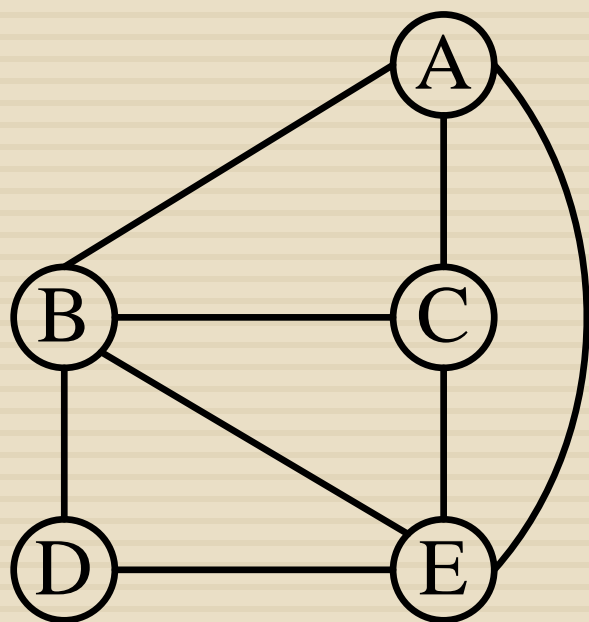
107

- 用图的邻接矩阵 a 表示无向连通图 G 。
- 解向量： (x_1, x_2, \dots, x_n) 表示顶点 i 所着颜色 $x[i]$
- 可行性约束函数：顶点 i 与已着色的相邻顶点颜色不重复。



2. 算法设计

108



```
□ class Color
□ {
□     friend int mColoring(int,int,int **);
□ private:
□     bool OK(int k); //检查颜色是否可用
□     void Backtrack(int t);
□     int n; //图的顶点数
□     int m; //可用颜色数
□     int **a; //图的邻接矩阵
□     int *x; //当前解
□     long sum; //当前已找到的可m着色方案数
□ };
```

```
bool Color::OK(int k) //检查顶点k颜色是否可用
{
    for(int j=1;j<=n;j++)
        if((a[k][j]==1)&&(x[j]==x[k])) //有边相连且两顶点颜色相同
            return false;
    return true;
}
```

```
void Color::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++) //m种颜色

{

 x[t]=i; //顶点t使用颜色i

 if(OK(t))

 Backtrack(t+1);

 x[t]=0; //恢复x[t]的初值

}

}


```
int mColoring(int n,int m,int **a)
{
```

```
    Color X;
```

```
    //初始化X
```

```
    X.n=n;
```

```
    X.m=m;
```

```
    X.a=a;
```

```
    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
```

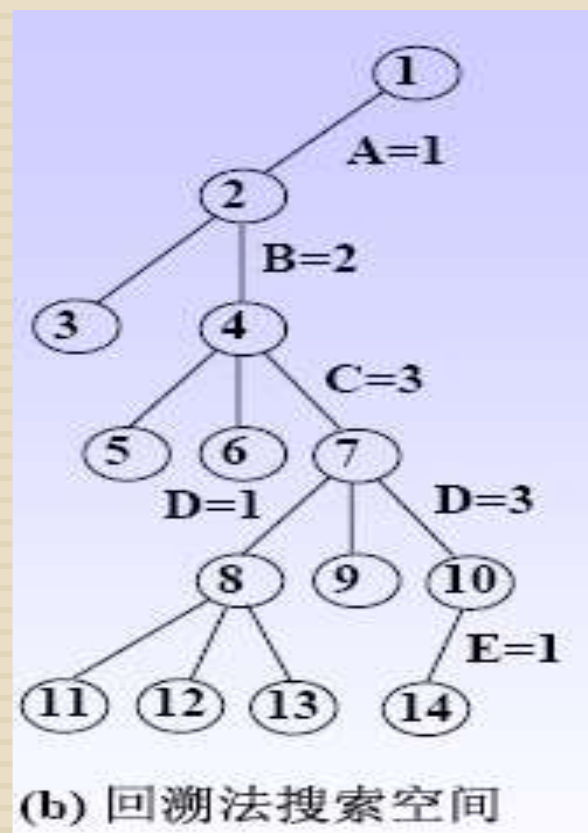
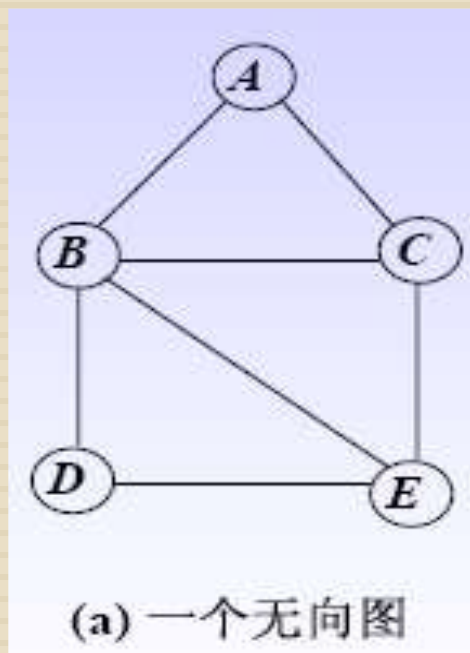
3. 算法效率

114

- 图 m 可着色问题的解空间树中内结点个数是 $\sum_{i=0}^{n-1} m^i$ 。
- 对于每一个内结点，在最坏情况下，用ok检查当前扩展结点的每一个儿子所相应的颜色可用性需耗时 $O(mn)$ 。因此，回溯法总的时间耗费是

$$\sum_{i=0}^{n-1} m^i (mn) = nm(m^n - 1) / (m - 1) = O(nm^n)$$

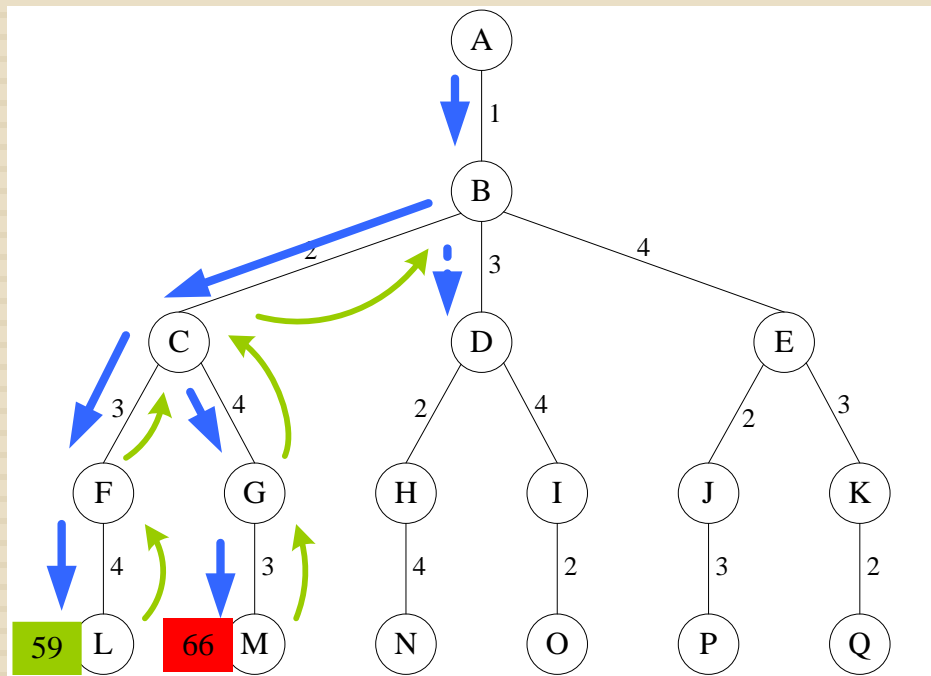
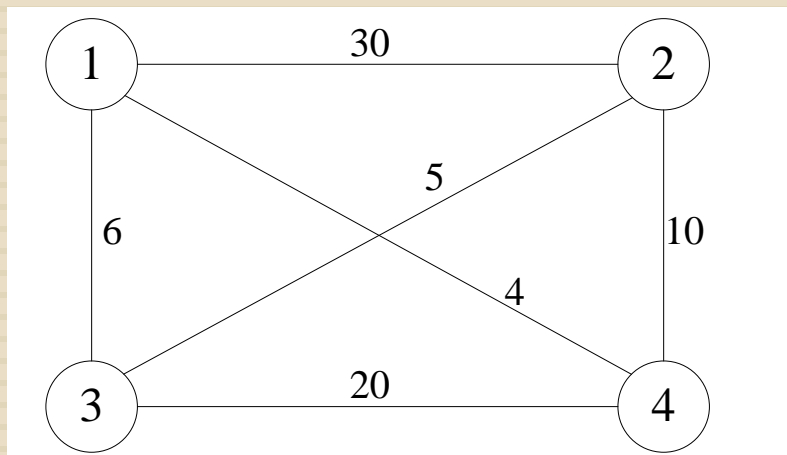
例：判断下图是否是3可着色。



5.9 旅行售货员问题

1. 算法描述

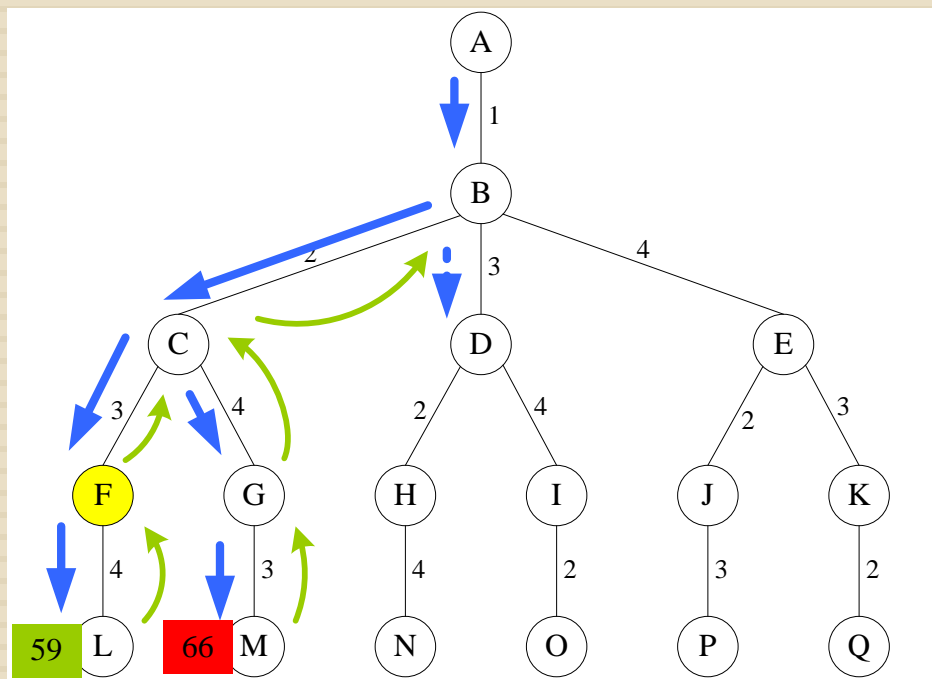
- 解空间：排列树
- 开始时 $x=[1,2,\dots,n]$ ，则相应的排列树由 $x[1:n]$ 的所有排列构成。



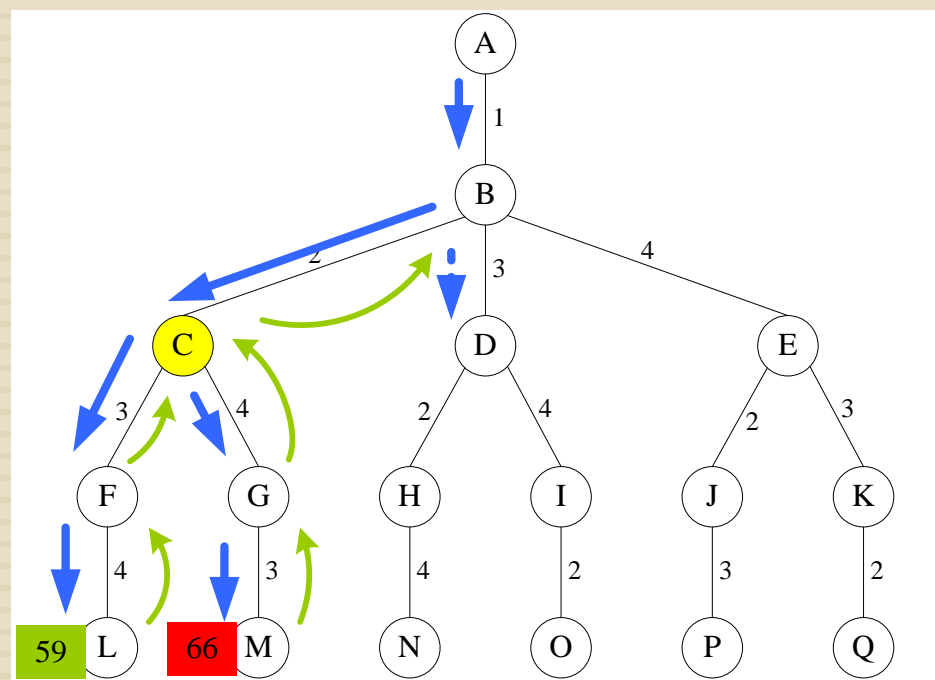
递归算法

117

- 当 $i=n$ 时，当前扩展结点是排列树的叶结点的父结点，此时检查图G是否存在一条从顶点 $x[n-1]$ 到顶点 $x[n]$ 的边和一条从顶点 $x[n]$ 到顶点1的边，如果两条边都存在，则找到一条回路。再判断此回路的费用是否优于当前最优回路的费用，是则更新当前最优值和最优解。




- 当 $i < n$ 时，当前扩展结点位于排列树的第 $i-1$ 层。图G中存在从顶点 $x[i-1]$ 到 $x[i]$ 的边时，检查 $x[1:i]$ 的费用是否小于当前最优值，是则进入排列树的第 i 层，否则剪去相应子树。



```
template<typename Type>
class Traveling
{
    friend Type TSP<>(int **,int [],int,Type);
private:
    void Backtrack(int i);
    int n; //图G的顶点数
    int *x; //当前解
    int *bestx; //当前最优解
    Type **a; //图G的邻接矩阵
    Type cc; //当前费用
    Type bestc; //当前最优值（费用）
    Type NoEdge; //无边标记，可用INT_MAX
};
```

```
template<typename Type>
void Traveling<Type>::Backtrack(int i)
{
    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++)
        if(a[x[i-1]][x[j]]!=NoEdge&&(cc+a[x[i-1]][x[j]]<bestc||bestc==NoEdge))
        {
            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]);
        }
}
```

```
template<typename Type>
Type TSP(Type **a,int v[],int n,Type NoEdge)
{
    Traveling<Type> Y;
    //初始化Y
    Y.x=new int[n+1];
    for(int i=1; i<=n; i++)
        Y.x[i]=i;
    Y.a=a;
    Y.n=n;
    Y.bestc=NoEdge;
    Y.bestx=v;
    Y.cc=0;
    Y.NoEdge=NoEdge;

    //搜索x[2:n]的全排列
    Y.Backtrack(2);
    delete[] Y.x;
    return Y.bestc;
}
```

2. 算法效率

123

- 算法backtrack在最坏情况下可能需要更新当前最优解 $O((n-1)!)$ 次，每次更新bestx需计算时间 $O(n)$ ，从而整个算法的计算时间复杂性为 $O(n!)$ 。

5.10 圆排列问题

1. 问题描述

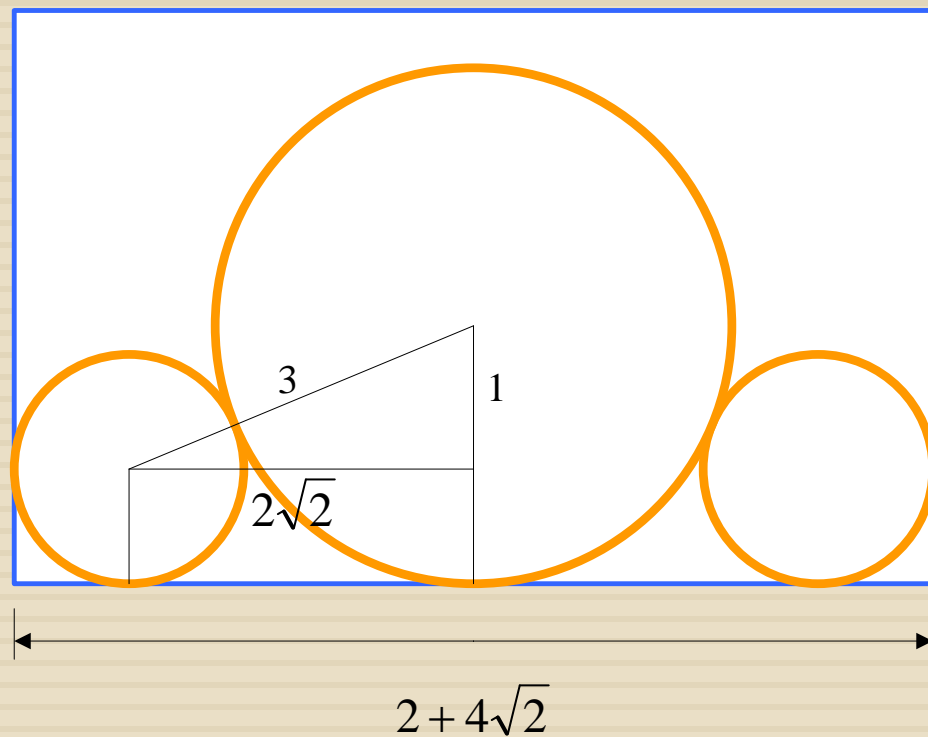
124

- 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：

125

- 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 $2+4\sqrt{2}$ 。



2. 算法设计

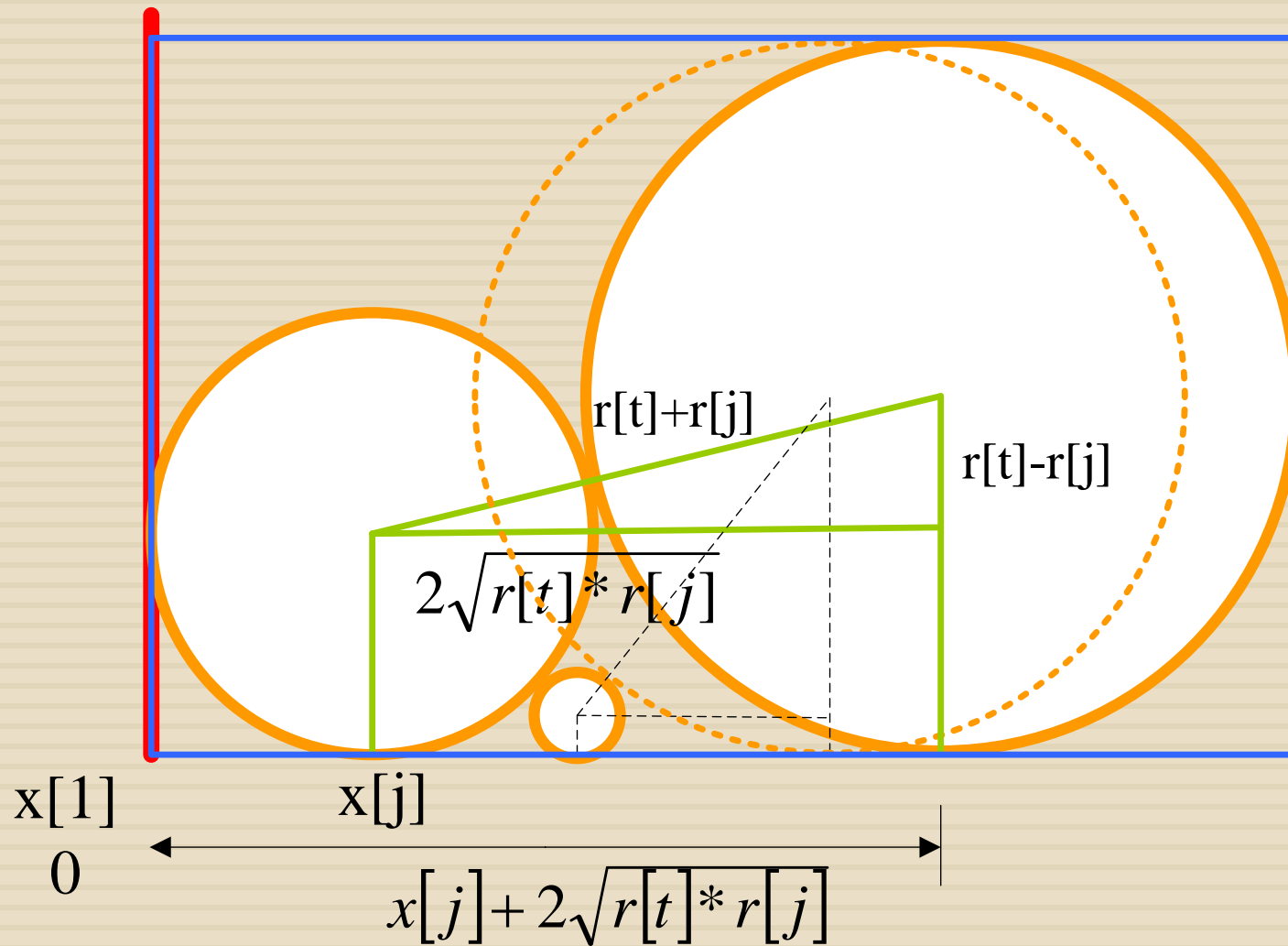
126

- 解空间：排列树
- 本问题的难点：
 - ▣ 计算当前所选择的圆在当前圆排列中圆心的横坐标 (Center) 。
 - ▣ 计算当前圆排列的长度 (Compute) 。
- 数组 r 表示当前圆排列， $r[i]$ 为第 i 个圆的半径。
- 数组 x 记录当前圆排列中各圆的圆心横坐标。
- 约定：在当前圆排列中排在第一个圆的圆心的横坐标为0.

```
□ class Circle
□ {
□     friend float CirclePerm(int,float*);
□ private:
□     float Center(int t); //计算第t个圆的圆心横坐标
□     void Compute(); //计算当前圆排列的长度
□     void Backtrack(int t);
□     float min; //当前最优值
□     float *x; //当前圆排列圆心横坐标
□     float *r; //当前圆排列
□     int n; //待排列圆的个数
□ };
```

Center

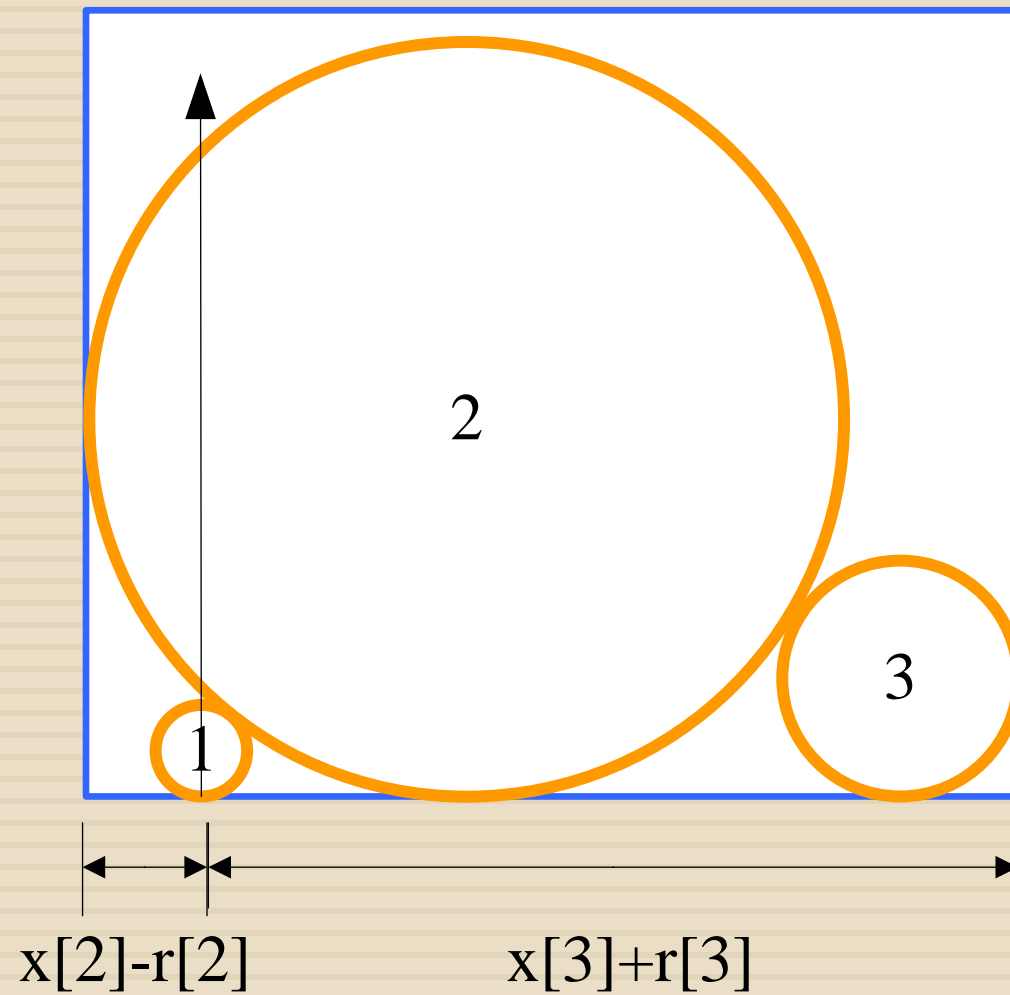
128




```
float Circle::Center(int t) //计算第t个圆的圆心横坐标
{
    float temp=0;
    for(int j=1;j<t;j++)
    {
        float valuel=x[j]+2.0*sqrt(r[t]*r[j]); //t与j相切
        if(valuel>temp)
            temp=valuel;
    }
    return temp;
}
```

Compute

130



```
void Circle::Compute() //计算当前圆排列的长度
{
    float low=0;
    float high=0;
    for(int i=1;i<=n;i++)
    {
        if(x[i]-r[i]<low)
            low=x[i]-r[i]; //最左端坐标(0或负数)
        if(x[i]+r[i]>high)
            high=x[i]+r[i]; //最右端坐标
    }
    if(high-low<min)
        min=high-low;
}
```

```
void Circle::Backtrack(int t)
{
    if(t>n)
        Compute();
    else
        for(int j=t;j<=n;j++)
        {
            Swap(r[t],r[j]);
            float centerx=Center(t);
            if(centerx+r[t]+r[1]<min) //下界约束
            {
                x[t]=centerx;
                Backtrack(t+1);
            }
            Swap(r[t],r[j]);
        }
}
```

```
float CirclePermutation(int n,float *a)
{
    Circle X;
    X.n=n;
    X.r=a;
    X.min=FLT_MAX; //cfloat中定义了FLT_MAX (最大float值)
    float *x=new float[n+1];
    X.x=x;
    X.Backtrack(1);
    delete[] x;
    return X.min;
}
```

3. 算法效率

134

在最坏情况下可能需要计算 $O(n!)$ 次当前圆排列长度，每次计算需 $O(n)$ 时间，从而整个算法的时间复杂度为 $O((n+1)!)$ 。

5.11 电路板排列问题

1. 问题描述

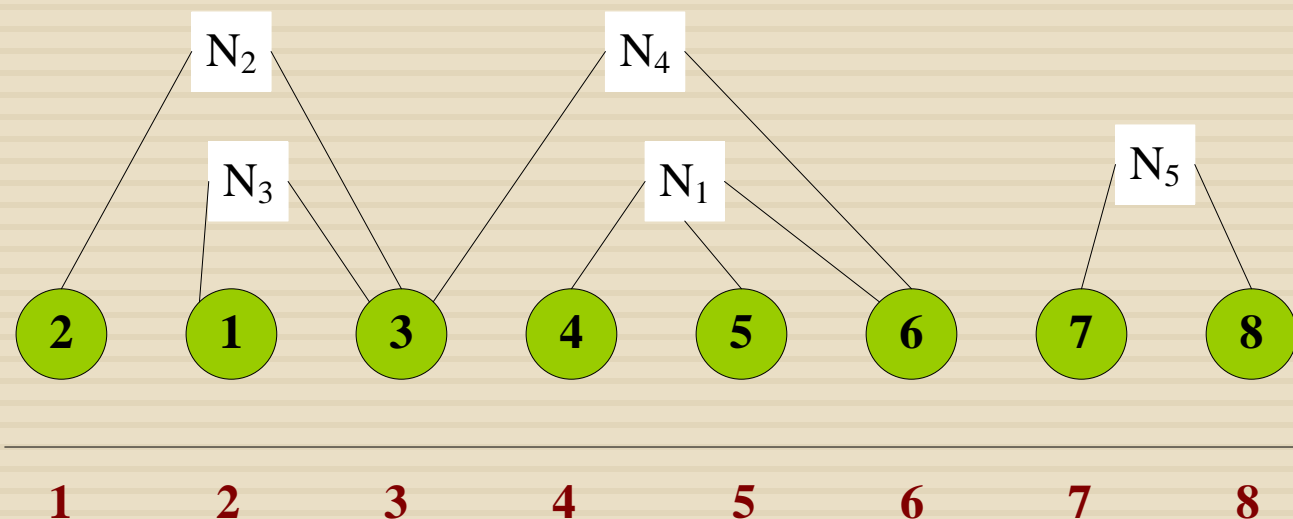
135

- 将 n 块电路板以最佳排列方式插入带有 n 个插槽的机箱中， n 块电路板的不同排列方式对应于不同的电路板插入方案。
- 设 $B=\{1,2,\dots,n\}$ 是 n 块电路板， $L=\{N_1,N_2,\dots,N_m\}$ 是 n 块电路板的 m 个连接块，其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。

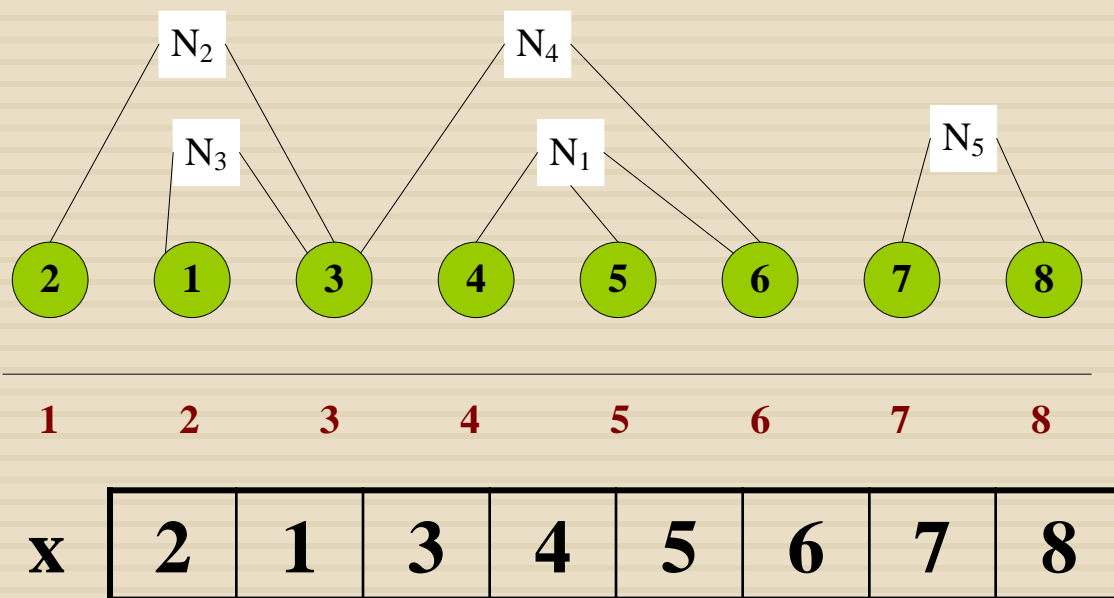
例：

136

- 设 $n=8$, $m=5$, 给定的 n 块电路板及 m 个连接块如下：
- $B=\{1,2,\dots,8\}$; $L=\{N_1,N_2,\dots,N_5\}$;
- $N_1=\{4,5,6\}$; $N_2=\{2,3\}$; $N_3=\{1,3\}$; $N_4=\{3,6\}$; $N_5=\{7,8\}$ 。



- 设 x 表示 n 块电路板的排列，即在第 i 个插槽中插入电路板。
- x 所确定的**电路板排列密度** $\text{density}(x)$ 定义为跨越相邻电路板插槽的最大连线数。



$$\text{density}(x)=2$$

问题：

138

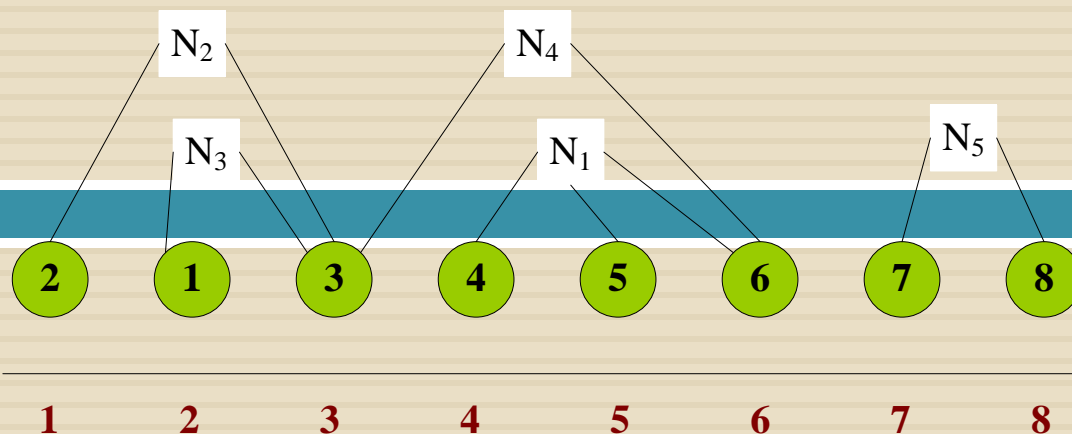
- 在设计机箱时，插槽一侧的布线间隙由电路板排列的密度所确定。因此电路板排列问题要求对于给定的电路板连接条件，确定电路板的最佳排列，使其具有最小密度。

2. 算法设计

139

- 用整型二维数组B表示输入。 $B[i][j]$ 的值为1表示电路板i在连接块 N_j 中。
- $total[j]$ 是连接块 N_j 中的电路板数。
- $now[j]$ 是电路板的部分排列 $x[1:i]$ 中所含 N_j 中的电路板数。

	0	0	1	0	0
	0	1	0	0	0
B	0	1	1	1	0
	1	0	0	0	0
	1	0	0	0	0
	1	0	0	1	0
	0	0	0	0	1
	0	0	0	0	1



x

2	1	3	4	?	?	?	?
---	---	---	---	---	---	---	---

total

3	2	2	2	2
---	---	---	---	---

now

1	2	2	1	0
---	---	---	---	---

- 连接块 N_j 的连线跨越插槽 i 和 $i+1$ 当且仅当 $\text{now}[j] > 0$ 且 $\text{now}[j] \neq \text{total}[j]$ 。

```
□ class Board
□ {
□     friend int Arrangement(int **,int,int,int []);
□ public:
□     void Print();
□ private:
□     void Backtrack(int i,int cd); //cd为当前密度
□     int n; //电路板数
□     int m; //连接块数
□     int *x; //当前解
□     int *bestx; //当前最优解
□     int bestd; //当前最优密度
□     int *total; //total[j], 连接块j的电路板数
□     int *now; //now[j], 当前解中所含连接块j的电路板数
□     int **B; //连接块数组
□ };
```

```
❑ void Board::Backtrack(int i,int cd)
❑ {
❑     if(i==n)
❑     {
❑         for(int j=1;j<=n;j++)
❑             bestx[j]=x[j];
❑         bestd=cd;
❑     }
❑     else
❑         for(int j=i;j<=n;j++)
❑         {
❑             int ld=0;
❑             for(int k=1;k<=m;k++) //m个连接块
❑             {
❑                 now[k]+=B[x[j]][k];
❑                 if(now[k]>0&&total[k]!=now[k])
❑                     ld++; //跨越i,i+1的连线
❑             }
❑         }
```

```
□      if(cd>ld)
□      ld=cd;
□      if(ld<bestd) //搜索子树
□      {
□          Swap(x[i],x[j]);
□          Backtrack(i+1,ld);

□          //恢复状态
□          Swap(x[i],x[j]);
□          for(int k=1;k<=m;k++)
□              now[k]-=B[x[j]][k];
□      }
□  }
□ }
```

- `int Arrangement(int **B,int n,int m,int bestx[])`

- `{`

- `Board X;`

- `X.x=new int[n+1];`

- `X.total=new int[m+1];`

- `X.now=new int[m+1];`

- `X.B=B;`

- `X.n=n;`

- `X.m=m;`

- `X.bestx=bestx;`

- `X.bestd=m+1; //密度最大为m`

- `for(int i=1;i<=m;i++)`

- `{`

- `X.total[i]=0;`

- `X.now[i]=0;`

- `}`

- for(int i=1;i<=n;i++)
- {
- X.x[i]=i; //初始化x为单位排列
- for(int j=1;j<=m;j++)
- X.total[j]+=B[i][j]; //计算total
- }

- X.Backtrack(1,0); //回溯搜索
- X.Print();
- delete[] X.x;
- delete[] X.total;
- delete[] X.now;
- return X.bestd;
- }

按照教务处要求，线上课程需要有
签到信息，请同学们尽快签到
签到格式：班级+姓名

5.12 连续邮资问题

1. 问题描述

147

- 假设国家发行了 n 种不同面值的邮票，并且规定每张信封上最多只允许贴 m 张邮票。连续邮资问题要求对于给定的 n 和 m 的值，给出**邮票面值**的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

例：

148

- 当 $n=5$ 和 $m=4$ 时，面值为 $(1,3,11,15,32)$ 的5种邮票可以贴出邮资的最大连续邮资区间是1到70。
- 1: 1
- 2: 1,1
- 3: 3
- 4: 1,3
-
- 69: 11,11,15,32
- 70: 3,3,32,32

2. 算法设计

149

- 解向量：用 n 元组 $x[1:n]$ 表示 n 种不同的邮票面值，并约定它们从小到大排列。
- $x[1]$ 的值一定为1，此时的最大连续邮资区间是 $[1:m]$ 。
- $x[2]$ 的可取范围是 $[2:m+1]$ 。
- 可行性约束函数：已选定 $x[1:i-1]$ ，最大连续邮资区间是 $[1:r]$ ，接下来 $x[i]$ 的可取值范围是 $[x[i-1]+1:r+1]$ 。
- 解空间树中各结点的度随 x 的不同取值而变化。

如何确定r的值？

150

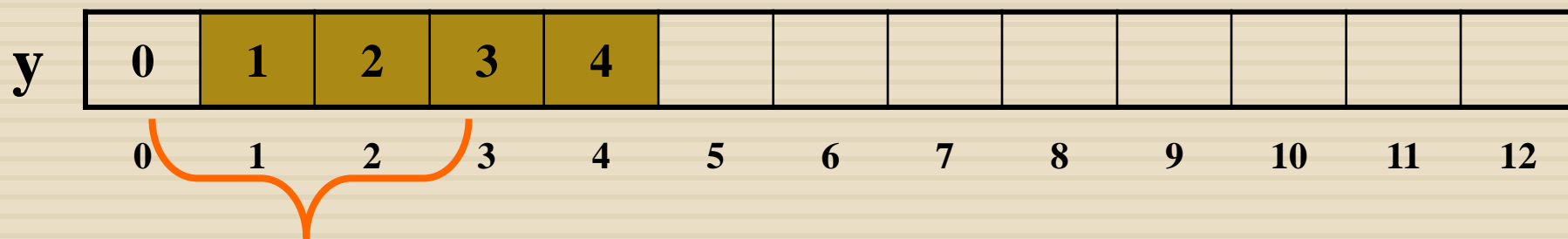
计算 $x[1:i]$ 的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法。考虑到直接递归的求解复杂度太高，我们不妨尝试计算用不超过 m 张面值为 $x[1:i]$ 的邮票贴出邮资 k 所需的最少邮票数 $y[k]$ 。通过 $y[k]$ 可以很快推出 r 的值。事实上， $y[k]$ 可以通过递推在 $O(n)$ 时间内解决。

如： $m=4, x=\{1,3\}$ 时

y	0	1	2	1	2	3	2	3	4	3	4		4
	0	1	2	3	4	5	6	7	8	9	10	11	12

还可加1~m-y[j]张新面值的，加上后会得到新的邮资。

$m=4, x=\{1\}$



$m=4, x=\{1, 3\}$



有了新面值时，计算 $x[1:i-1]$ 时的 y ：

```
for(int j=0; j<=x[i-1]*(m); j++)//面值为j的信封上
```

```
if(y[j]<m)
```

```
for(int k=1; k<=m-y[j]; k++)//再贴k张新面值
```

```
if(y[j]+k<y[j+x[i]*k])//如果张数少
```

```
y[j+x[i]*k]=y[j]+k;//更新为少的这种方案
```



```
class Stamp
{
    friend int MaxStamp(int,int,int[]);
private:
    void Backtrack(int i,int t);
    int n; //邮票面值数
    int m; //每张信封最多允许贴的邮票数
    int maxvalue; //当前最优值
    int maxint; //大整数
    int maxl; //邮资上界
    int *x; //当前解
    int *y; //贴出各种邮资所需最少邮票数
    int *bestx; //当前最优解
};
```

void Stamp::Backtrack(int i,int r) //r, i-2连续邮资区间的
最大值

```
{  
    for(int j=0; j<=x[i-1]*(m); j++) //计算i-1时的y  
        if(y[j]<m)  
            for(int k=1; k<=m-y[j]; k++)  
                if(y[j]+k<y[j+x[i]*k])  
                    y[j+x[i]*k]=y[j]+k;  
    while(y[r]<maxint) //找r的最大值,不是初始值的最大r  
        r++; //向后推进,从而再次判断,直到不满足
```

```
if(i>n)
{
    if(r-1>maxvalue)
    {
        maxvalue=r-1;
        for(int j=1; j<=n; j++)
            bestx[j]=x[j];
    }
    return;
}
```

```
int *z=new int[maxl];
for(int k=0; k<maxl; k++) //保存y的状态
    z[k]=y[k];
for(int j=x[i]+1; j<=r; j++)
{
    x[i+1]=j;
    Backtrack(i+1,r-1);
    for(int k=0; k<maxl; k++) //恢复y的状态
        y[k]=z[k];
}
delete[] z;
}
```

```
int MaxStamp(int n,int m,int bestx[])
```

```
{
```

```
    Stamp X;
```

```
    int maxint=32767;
```

```
    int maxl=1500;
```

```
    X.n=n;
```

```
    X.m=m;
```

```
    X.maxvalue=0;
```

```
    X.maxint=maxint;
```

```
    X.maxl=maxl;
```

```
    X.bestx=bestx;
```

```
    X.x=new int[n+1];
```

```
    X.y=new int[maxl+1];
```

```
for(int i=0; i<=n; i++)  
    X.x[i]=0;  
for(int i=0; i<maxl; i++)  
    X.y[i]=maxint;
```

```
X.x[1]=1;
```

```
X.y[0]=0;
```

```
X.Backtrack(1,0);
```

```
for(int i=1; i<=n; i++)  
    cout<<bestx[i]<<' ';
```

```
cout<<endl;
```

```
delete[] X.x;
```

```
delete[] X.y;
```

```
return X.maxvalue;
```

```
}
```

5.9 回溯法效率分析

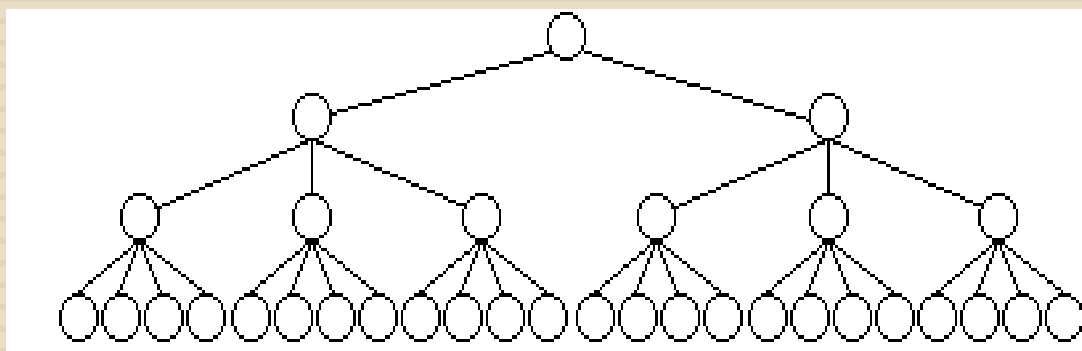
159

- 通过前面具体实例的讨论容易看出，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。
- 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。

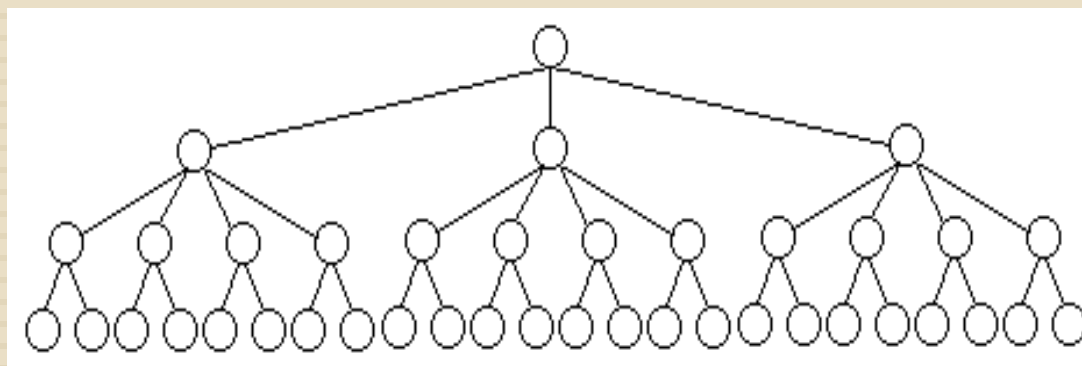
重排原理

160

- 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。



(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。