批处理作业调度

给定n个作业的集合$\{J_1,J_2,\ldots,J_n\}$。每个作业必须先由机器1处理，然后由机器2处理。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$f=\sum_{i=1}^{n} F_{2i}$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。

批处理作业调度

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。

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

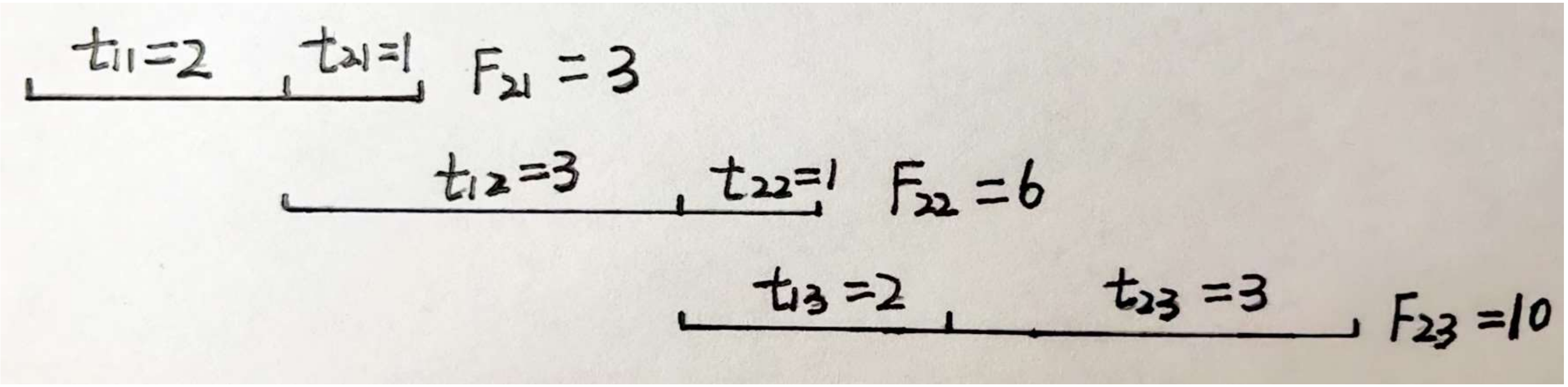
这3个作业的6种可能的调度方案是(1,2,3)；(1,3,2)；(2,1,3)；(2,3,1)；(3,1,2)；(3,2,1)；

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。

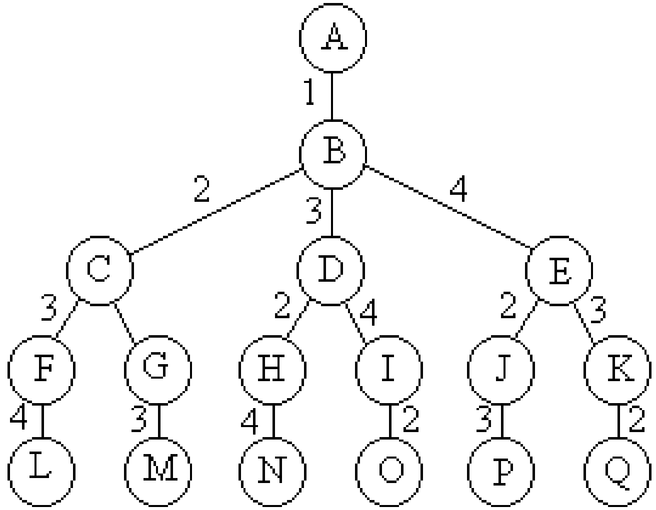
批处理作业调度

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

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，调度方案(1,2,3) 如下图所示，其完成时间为：3+6+10=19。



批处理作业调度




```
private static void backtrack(int i){
    if (i > n)
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestf = f;
    else
        for (int j = i; j <= n; j++) //现在正在要做第i个的加工, 选择作业j作为第i个要进行的作业
            f1 += m[x[j]][1];
            f2[i]=(f2[i-1]>f1?f2[i-1]:f1)+m[x[j]][2];//红色计算作业x[j]在机器2上的开始时间
            f+=f2[i];
            if (f < bestf) {
                swap(x,i,j);
                backtrack(i+1);
                swap(x,i,j);
            f1 -= m[x[j]][1];
            f -= f2[i];
}
```

backtrack(1)

```
public class FlowShop
        int  n,       // 作业数
             f1,      // 机器1完成处理时间
             f,       // 完成时间和
             bestf;   // 当前最优值
        int [][] m;   // 各作业所需的处理时间
        int [] x;     // 当前作业调度
        int [] bestx; // 当前最优作业调度
        int [] f2;    // 机器2完成处理时间
```

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”-”。

```
+ + - + - + +
 + - - - - +
  - + + + -
   - + + -
    - + -
     - -
      +
```

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。

符号三角形问题

- 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"-"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

count: “+”的个数; half: n*(n+1)/4

```
private static void backtrack (int t){//处理第t个符号
    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; //第1行第t个符号
            count+=i;
            for (int j=2;j<=t;j++) //第2至t行最后一列
                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;
}
```

backtrack(1)

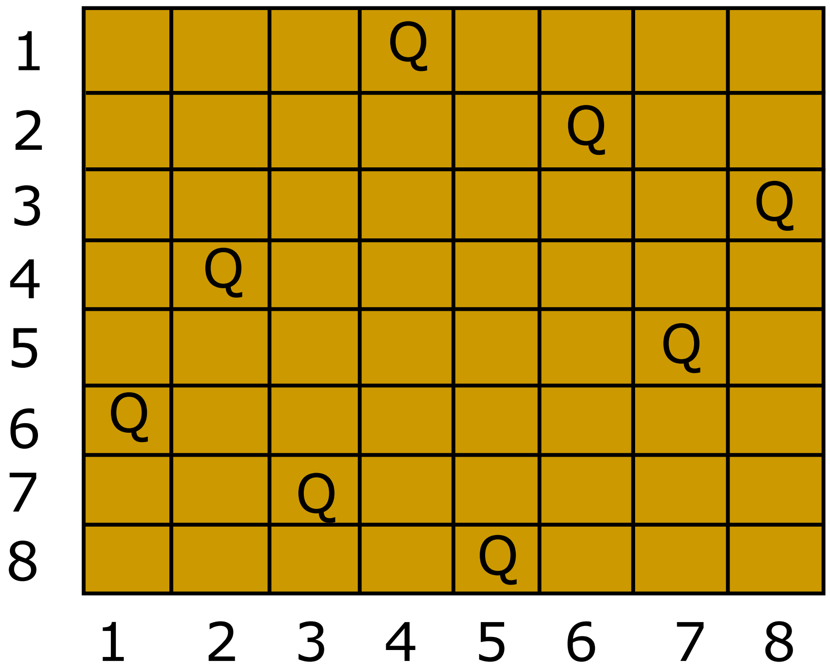
```
      +   +   -   +   -   +   +
                              t
j=2     +   -   -   -   -   +
                              t-j+1
          -   +   +   +   -
            -   +   +   -
              -   +   -
                -   -
                  +
```

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×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×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。

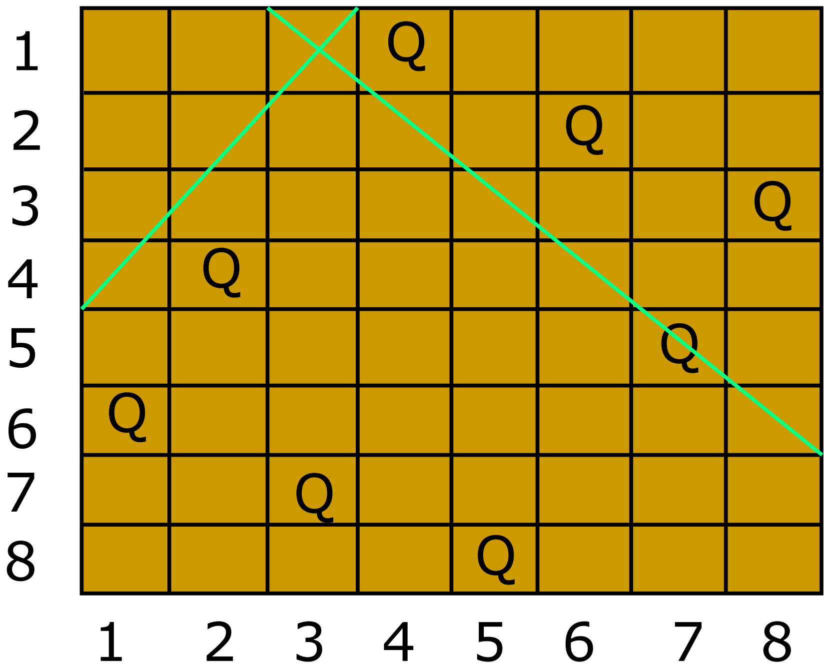


n后问题

n叉树、排列树

- 解向量：$(x_1, x_2, \ldots, x_n)$　　问题：解空间的结构?
- 显约束：$x_i=1,2, \ldots ,n$，表示第i行的皇后放在第x[i]列（不同行）
- 隐约束：1)不同列：$x_i \neq x_j$

  2)不处于同一斜线同一正、反对角线：$|i-j| \neq |x_i-x_j|$



n后问题（n叉树）

```
private static void backtrack (int t){
    if (t>n)
        sum++;
    else
        for (int i=1;i<=n;i++)
            x[t]=i;
            if (place(t))
                backtrack(t+1);
}
```

问题：排列树?

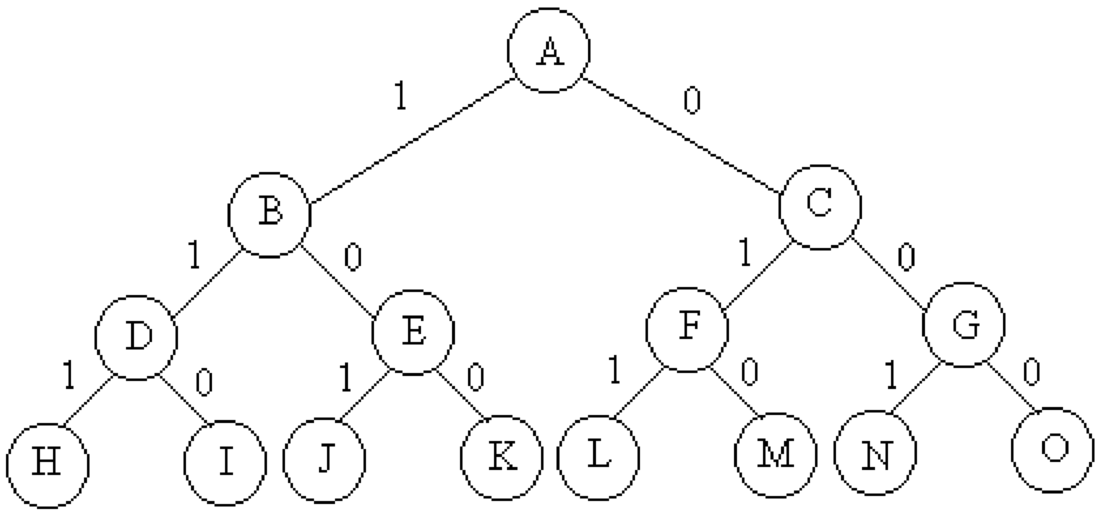
- 显约束：不同行 & 不同列
- 隐约束：不处于同一正、反对角线

```
private static boolean place (int k){
    for (int j=1;j<k;j++)
        if ((Math.abs(k-j)==Math.abs(x[j]-
                                    x[k]))||(x[j]==x[k]))
            return false;
    return true;
}
```

backtrack(1)

0-1背包问题

- 解空间：子集树
- 可行性约束：$\sum_{i=1}^{n} w_i x_i \le c_1$,进入左子树时判定；
- 上界函数：cp+r>bestp，cp是当前价值，r是当前剩余物品价值，bestp是当前最优价值，进入右子树时判定。



0-1背包问题

bestp: 当前最优值;
cw: 背包中已装入物品的重量;
cp: 背包中已装入物品的价值

```
void backtrack( int i) {
        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);
}
```

backtrack(1)

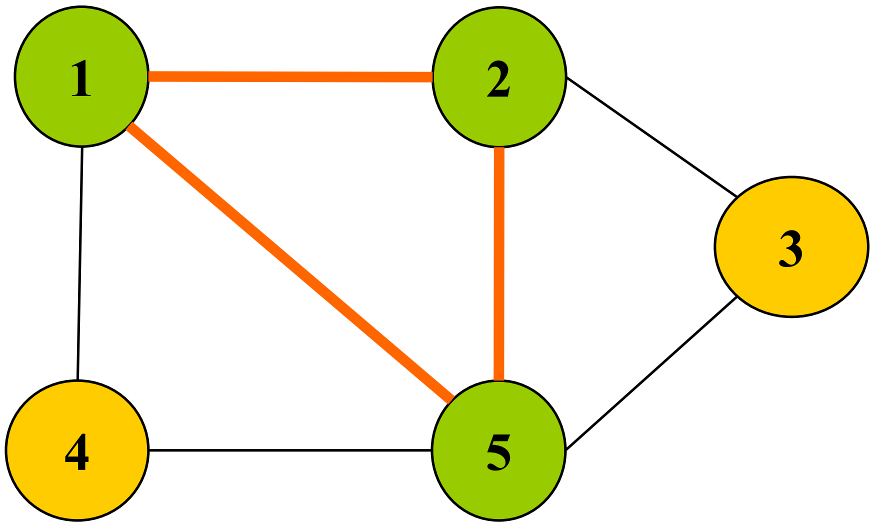
0-1背包问题

剩余物品按单位重量的价值从大到小排序。物品i的单位重量价值为p[i]/w[i]，p[i]为物品i的价值，w[i]为物品i的重量。

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

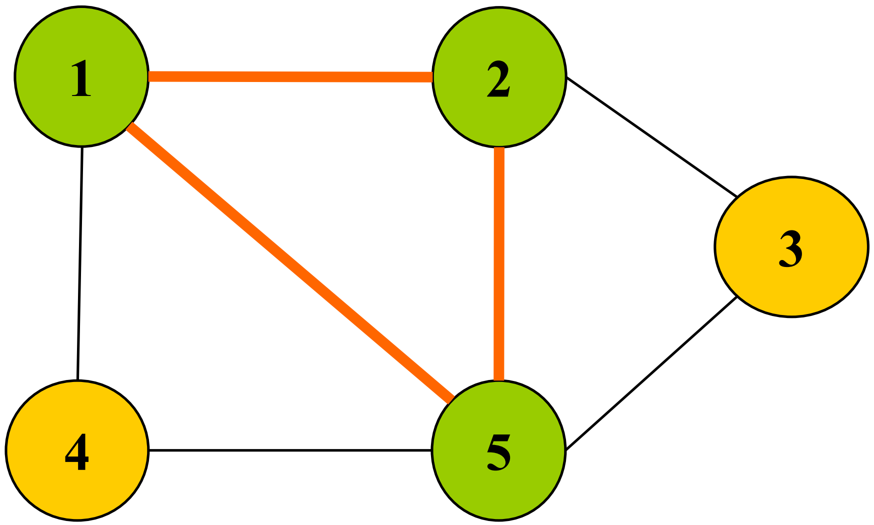
最大团问题

- 给定无向图G=(V，E)。如果U⊆V，且对任意u，v∈U有(u，v)∈E，则称U是G的完全子图。例如{1,2}。
- 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，例如{1,2,5}。
- G的最大团是指G中所含顶点数最多的团。 {1,2,5} {1,4,5}



最大团问题

- 如果U⊆V且对任意u，v∈U有(u，v)∉E，则称U是G的空子图，例如{2,4}。
- 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。

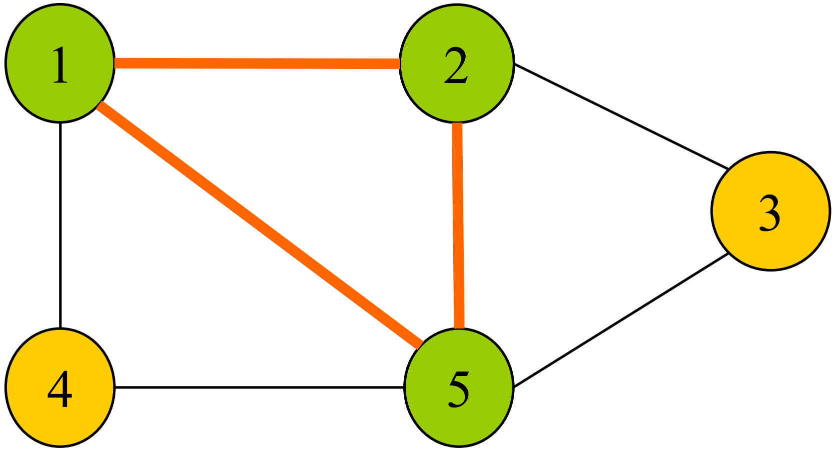


最大团问题

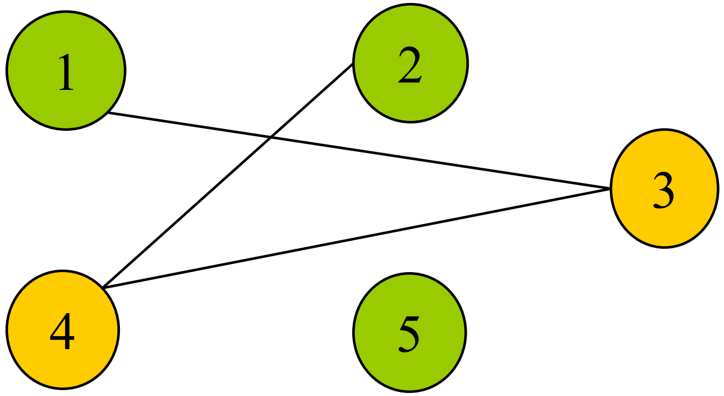
- 对于任一无向图G=(V，E)其补图G'=(V1，E1)定义为：

  V1=V，且(u，v)∈E1当且仅当(u，v)∉E。

U是G的最大团当且仅当U是G'的最大独立集。

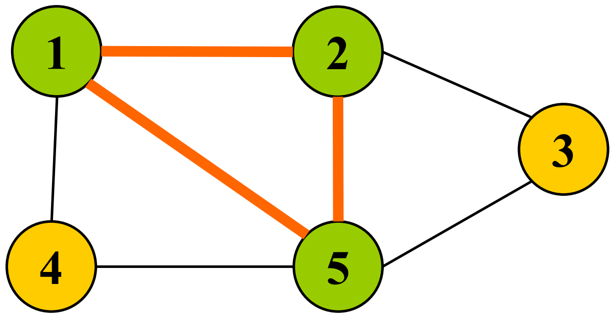


图G



图G'

最大团问题



最大团问题找出给定图G=(V, E)的最大团，可看做图G的顶点集V的子集选取问题。

- 解空间：子集树
- 可行性约束函数：顶点i到已选入顶点集中的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。

最大团问题

```
private static void backtrack(int i) {
    if (i > n)   // 到达叶结点
        for (int j = 1; j <= n; j++)   bestx[j] = x[j];
        bestn = cn;
        return;
    boolean ok = true;
    for (int j = 1; j < i; j++)  // 检查顶点 i 与当前团的连接
        if (x[j] == 1 && !a[i][j])  // i与j不相连
            ok = false;
            break;
    if (ok)   // 进入左子树
        x[i] = 1;  cn++;
        backtrack(i + 1);
        cn--;
    if (cn + n - i > bestn) // 进入右子树—还有足
        x[i] = 0;
        backtrack(i + 1);
}
```

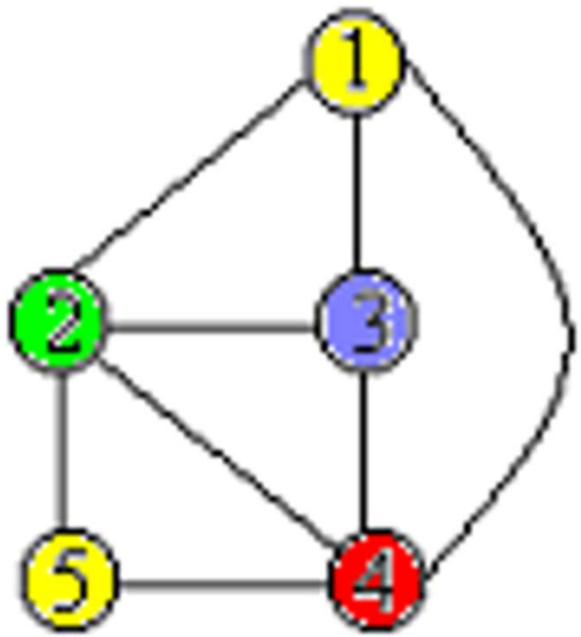
复杂度分析

最大团问题的回溯算法**backtrack**所需的计算时间为$O(n2^n)$。

backtrack(1)

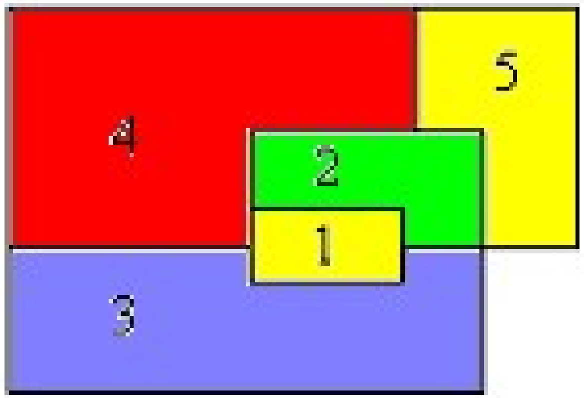
图的m着色问题

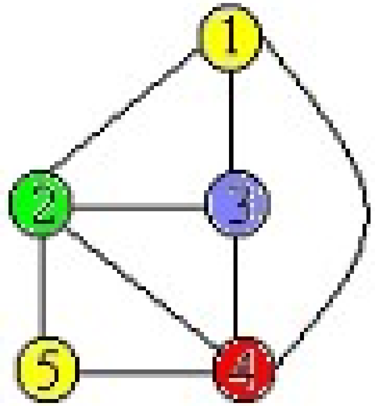
- 给定无向连通图G和m种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色。
- 是否有一种着色法使G中每条边的2个顶点着不同颜色。这个问题是图的m可着色判定问题。
- 若一个图最少需要m种颜色才能使图中每条边连接的2个顶点着不同颜色，称这个数m为该图的色数。
- 求一个图的色数m的问题称为图的m可着色优化问题。



四色猜想

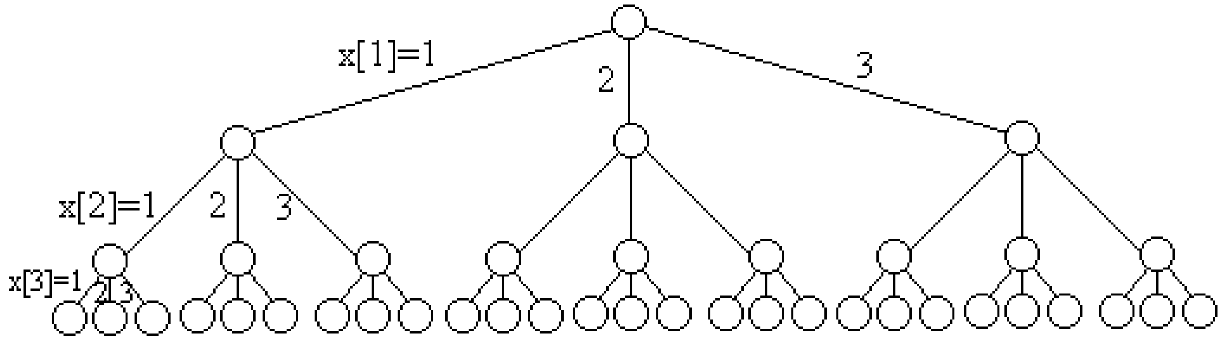
- 四色猜想：在一个平面或球面上的任何地图能够只用4种颜色来着色，使相邻的国家在地图上着不同的颜色。
- 假设每个国家在地图上是单连通域，两个国家相邻是指这两个国家有一段长度不为0的公共边界，而不仅有一个公共点。
- 这样的地图用平面图表示，地图上的每个区域相应平面图的一个顶点。相邻区域相应的两个顶点之间有一条边。





图的m着色问题

- 解向量：$(x_1, x_2, \ldots, x_n)$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。

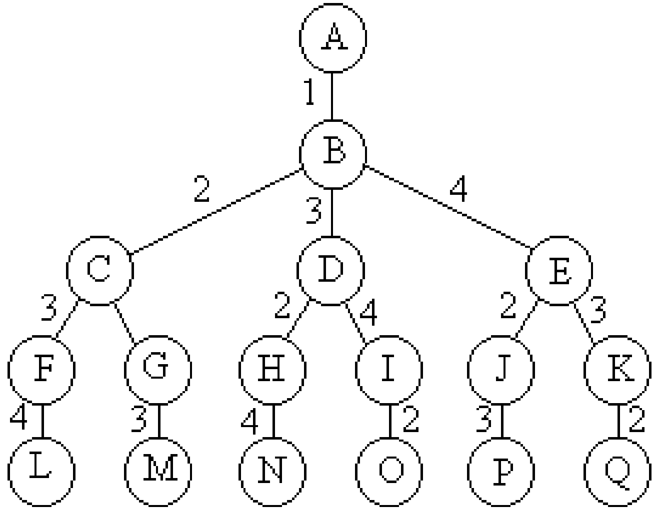



```
private static void backtrack(int t){
    if (t>n) sum++;
    else
        for (int i=1;i<=m;i++) {
            x[t]=i;
            if (ok(t))
                backtrack(t+1);
}
```

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

backtrack(1)

旅行售货员问题



- 从一个给定的城市出发，访问所有城市
- 解空间：排列树，由x[1:n]的所有排列
  Perm。
- 当前扩展结点位于n-1层时：
  - 其是叶结点的父节点，检测是否存在一条从顶点x[n-1]到x[n]的边和一条从x[n]到1的边。如果两条边都存在则找到一条回路，若这条回路的费用小于已找到的当前最优值bestc，则更新bestc。
- 当前扩展结点位于第i-1层，i<n 时：
  - 若存在一条从x[i-1]到x[i]的边，则比较x[1:i]的费用和当前最优值bestc。若x[1:i]的费用小于bestc，则进入第i层，否则剪去相应的子树。

旅行售货员问题

```
private static void backtrack(int i){
    if (i == n) {
        if (a[x[n - 1]][x[n]] < MAX && a[x[n]][1] < MAX &&
        (bestc =                                              )
            fo

            be
    else
        for (in
            if                                     a[x[i -1]][x[j]]<bestc))

                cc += a[x[i - 1]][x[i]];
                backtrack(i + 1);
                cc -= a[x[i - 1]][x[i]];
                swap(x, i, j);
}
```

是否有更高效的界限函数计算？

--最小出边：从顶点i发出的边最小费用记为$\min(i)$。

--前缀$x[1:i]$的回路费用至少为

$$\sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min(x_j)$$

backtrack(2)

复杂度分析

算法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时间为O(n)，从而整个算法的时间复杂性为O(n!)。