

第5章 回溯法

□ 回溯法概述

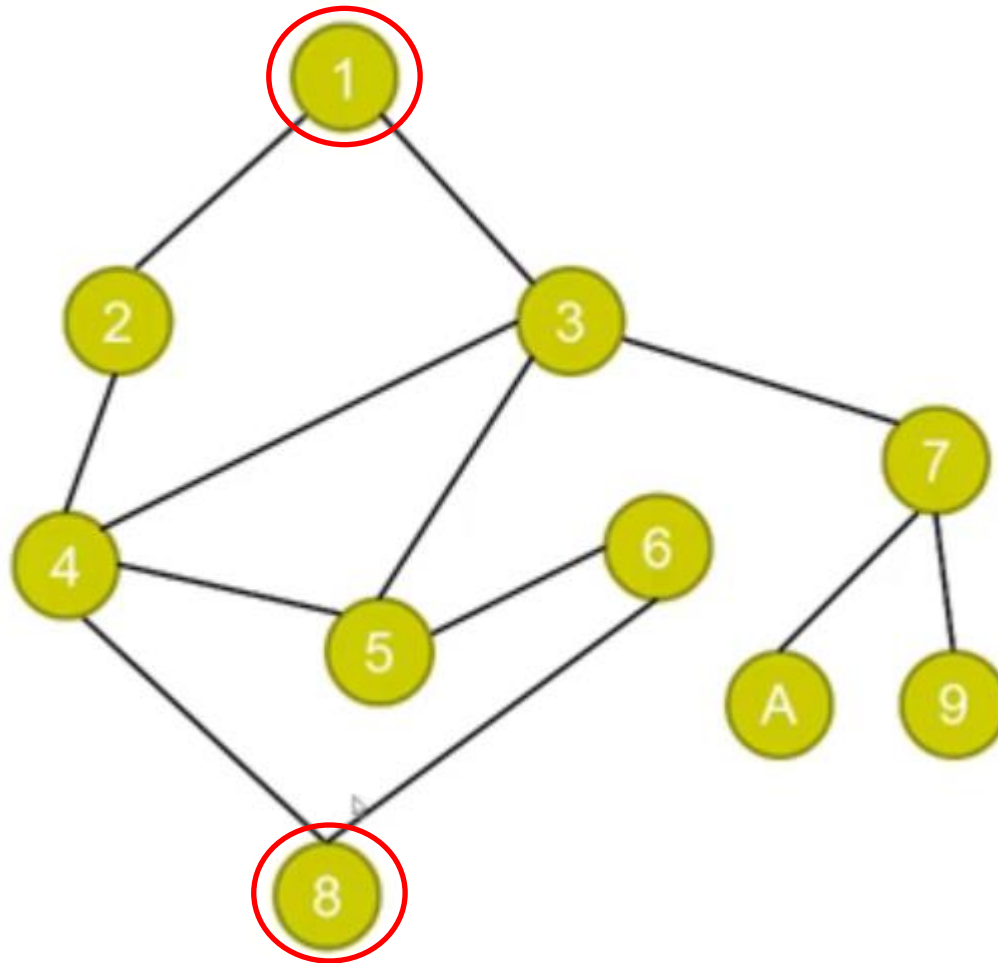
- 问题的解空间
- 回溯法的基本思想
- 回溯法的算法框架

□ 典型应用

- 0-1背包问题 (3-1)★
- 迷宫问题 (3-2)★
- n后问题 (3-3)★
- 图的m着色问题 (3-4)★
- 全排列问题 (3-5)★
- 素数环问题 (3-6)★

□ 总结

问题：如何寻找从1到8的路径。



回溯法的直观印象（1）

对于有些最优解问题，没有任何的理论也无法采用精确的数学公式来帮助我们找到最优解，我们只能用穷举算法。

回溯法实际上一个类似穷举的**搜索**尝试过程，主要是在搜索尝试过程中按照**深度优先**的方法寻找问题的解，当发现已不满足求解条件时，就“**回溯(Backtracking)**”（**即回退，走不通就掉头**），尝试别的路径。

回溯法的直观印象（2）

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。

回溯法的直观印象（3）

- 回溯法在用来求问题的所有解（或最优解）时，要回溯到根，且根结点的所有子树都已被搜索遍才结束。
- 而在求任一解（存在性问题）时，只要搜索到一个解就结束。
- 这种以深度优先（**DFS**）的方式系统地搜索问题的解的算法称为回溯法，它适合于解一些组合数较大的问题。

剪枝函数和回溯法

- 为了提高搜索效率，在搜索过程中使用**约束函数** (constraint function)，可以避免无谓地搜索那些不满足约束条件的子树。
- 如果是最优化问题，还可使用**限界函数** (bound function) 剪去那些不可能包含最优答案结点的子树。
- 约束函数**和**限界函数**的目的是相同的，都为了剪去不必要搜索的子树，减少问题求解所需实际生成的状态结点数，它们统称为**剪枝函数** (pruning function)。

具有限界函数的深度优先生成法称为回溯法。

5.1 回溯法概述

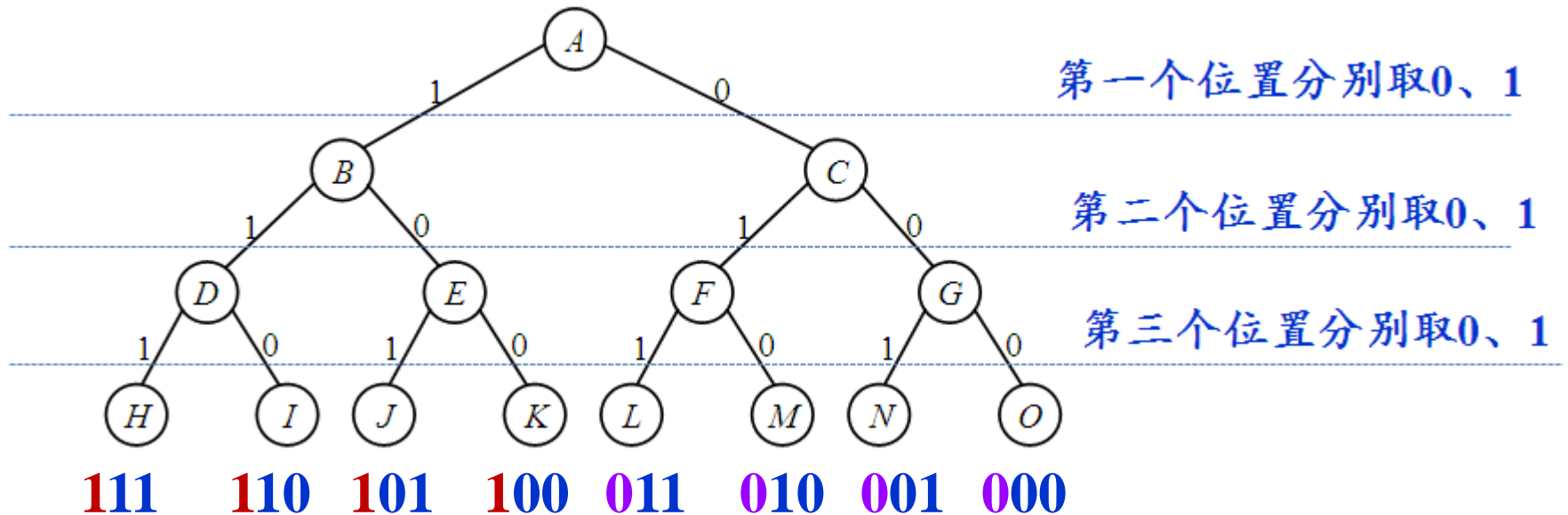
5.1.1 问题的解空间

一个复杂问题的解决方案是由若干个小的决策步骤组成的决策序列，解决一个问题的所有可能的决策序列构成该问题的解空间。一般用树形式来组织，也称为解空间树或状态空间。

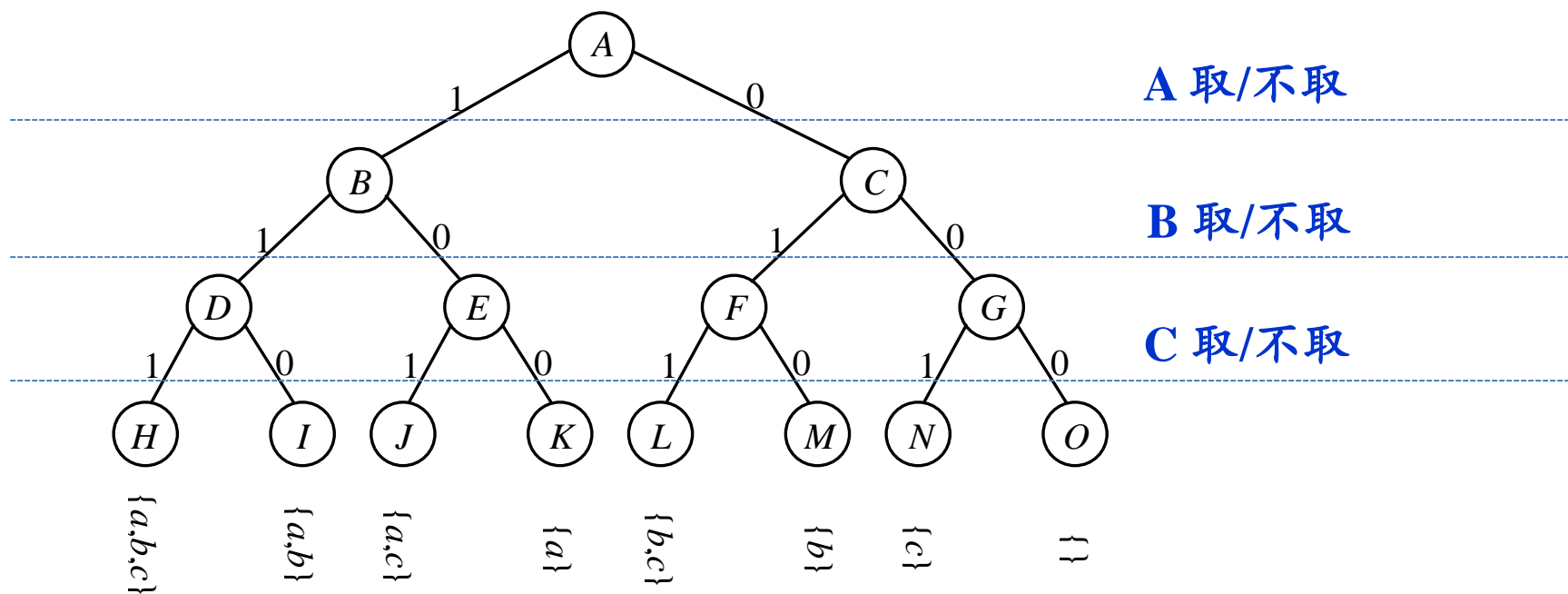
应用回溯法求解问题时，首先应该明确问题的解空间。解空间中满足约束条件的决策序列称为可行解（是解空间的子集）。

一般来说，解任何问题都有一个目标，在约束条件下使目标达到最优的可行解称为该问题的最优解。

例如，如下图所示是求集合3位2进制数全排列的解空间树，求解过程分为3步，分别对3个位置做决策，该解空间的每个叶子结点都构成一个解。



例如，如下图所示是求集合 $\{a,b,c\}$ 的幂集的解空间树，求解过程分为3步，分别对 a 、 b 、 c 元素做决策，该解空间的每个叶子结点都构成一个解（很多情况并非如此）。



相关名词:

- **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。
- **扩展结点**: 是指正在产生孩子结点的结点。
- **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)。

可以扩展的节点

无法扩展的节点

5.1.2 回溯法的基本思想

在包含问题的所有解的解空间树中，按照**深度优先搜索的策略**，从根结点（开始结点）出发搜索解空间树。

(1) 首先根结点成为**活结点**，同时也成为当前的扩展结点。

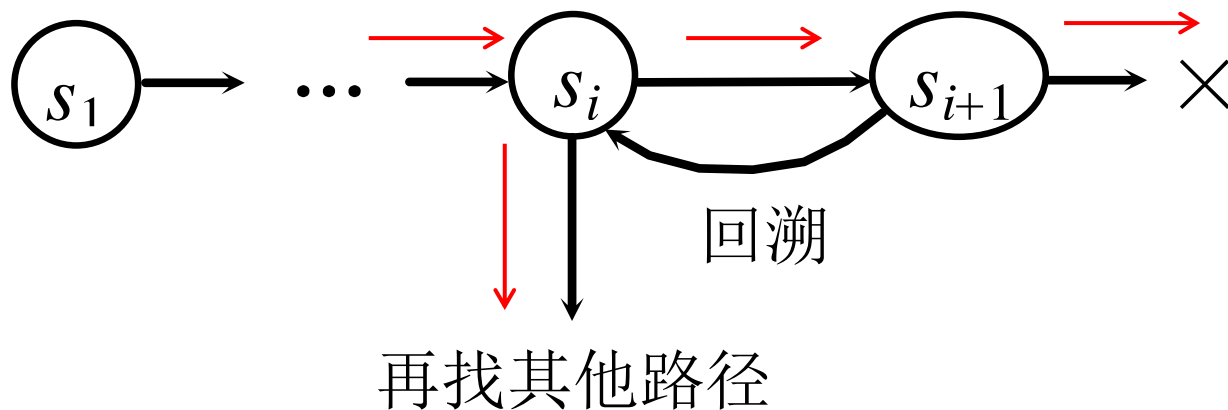
(2) 在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。

(3) 如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为**死结点**。此时应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。

回溯法以这种方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活结点为止。

如下图所示，当从状态 s_i 搜索到状态 s_{i+1} 后，如果 s_{i+1} 变为死结点，则从状态 s_{i+1} 回退到 s_i ，再从 s_i 找其他可能的路径，所以回溯法体现出走不通就退回再走的思路。

若用回溯法求问题的所有解时，需要回溯到根结点，且根结点的所有可行的子树都要已被搜索完才结束。而若使用回溯法求任一解时，只要搜索到问题的一个解就可以结束。



归纳起来，用回溯法解题的一般步骤如下：

- 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 (DFS) 搜索解空间树，并在搜索过程中用剪枝函数来避免无效搜索。

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。

DSF (Depth First Search) 算法简介

- Depth First Search (深度优先搜索)
- 从起点开始尽可能深入地扩展
 - 每次试图访问一个之前没有访问过的点
 - 有回退：如果此路不通，回到上一层节点
 - 。 。 。
- 注意点：
 - 节点判重
 - 路径记录
 - 回溯回来记得清理数据
- 堆栈：
 - 通常我们使用递归 – 系统内部堆栈
 - 递归层数很深的时候，系统堆栈不够用，需要自己实现堆栈

5.1.3 回溯法的算法框架

1. 递归回溯框架



```
void backtrack (int t)
{
    if (t>n) output(x);           //递归结束条件: 搜索到叶子结点, 输出一个可行解
    else                          //枚举t的所有可能的扩展节点
        for (int i=f(n,t);i<=g(n,t);i++) // f(n, t) : 可扩展的下界; g(n, t) : 上界;
        {
            x[t]=h(i);           //获取新的扩展点并标记该扩展节点
            if (constraint(t)&&bound(t)) //节点t未越界且满足扩展/减枝条件可扩展
            {
                backtrack(t+1);   //进一步深搜前的工作: 步长++、记录路径
                                //递归执行进行下一层 (深一层) 搜索
                                //注意: 进行回溯回来的清理工作!
            }
        }
}
```


例：穷举n位2进制数全排列

```
//深度优先的方法穷举n位二进制数
// g_n: 总位数（全局变量）
// i: 排列到第i位
void search(int i) //处理第i位数
{
    if (i == g_n) //找到了一个叶子结点，n位都排列完成了
    {
        print_it(g_a, g_n); //打印当前解（排列）
        return;
    }

    //-----1: 对第i个位，置1      <=  获取新的扩展点
    g_a[i]=1;
    search(i+1);

    //-----0: 对第i个位，置0      <=  获取新的扩展点
    g_a[i]=0;
    search(i+1);
}
```


2. 迭代（非递归）的算法框架



不做特别
要求！

```
void iterativeBacktrack ()
{
    int t=1;           // t: 记录深度（层数）
    while (t>0) {      // 尚未回溯到头，有路可走
        if (f(n,t)<=g(n,t)) //当前扩展结点处未搜索过的子树的起始编号<=终止编号
            for (int i=f(n,t);i<=g(n,t);i++) { //枚举t的所有所有可能的扩展节点
                x[t]=h(i);                      //获取新的扩展点并标记该扩展节点
                if (constraint(t)&&bound(t)) { //未越界且满足扩展/减枝条件可扩展
                    {
                        //记录：步长++、路径
                        if (solution(t)) output(x); //如果得到解，输出
                        else //否则，未找到解
                            t++; //进行下一层（深一层）搜索
                    }
                }
            }
        else t--; //回溯到上一层
    }
}
```

5.1.4 回溯法算法的时间分析

回溯法属于蛮力穷举法，最坏时间复杂性不可期望。

回溯法的有效性体现在当问题规模 n 很大时，对解空间的大量剪枝上，这将使回溯法具有很好的平均时间性能。

通常情况下，回溯法的效率会高于穷举法。

5.1.5 回溯法的解空间树构造

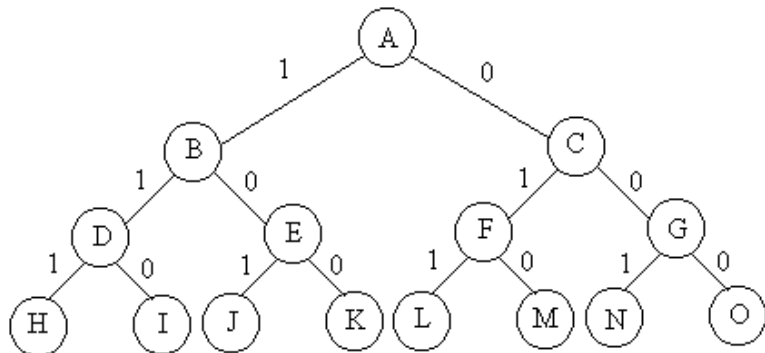
1.子集树:当所给问题是从 n 个元素的集合 S 中找出满足某种性质的子集时, 解空间为子集树。子集树的分支分别用1或者0代表,对应于该元素的取和舍。

例如: 0-1背包问题、装载问题。

2.排列树:当所给问题是从 n 个元素的集合 S 中找出满足某种性质的排列时, 解空间为排列树。

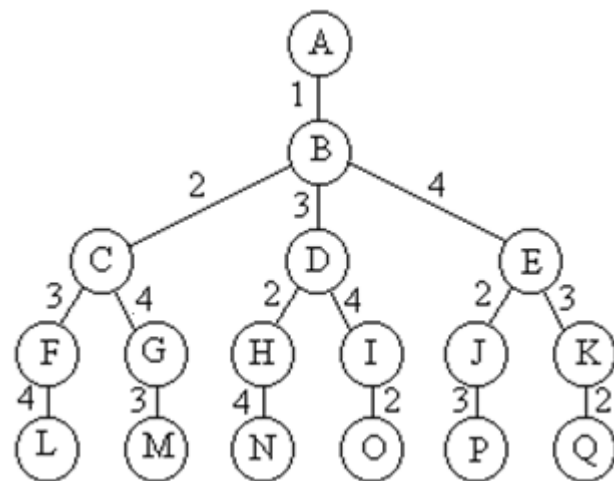
例如: 素数环问题、全排列问题

子集树与排列树



遍历子集树需 $O(2^n)$ 计算时间

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=0;i<=1;i++) {
            x[t]=i;
            if (legal(t)) backtrack(t+1); }
}
```



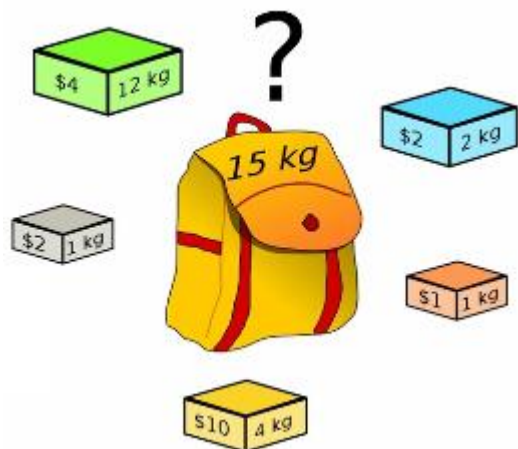
遍历排列树需要 $O(n!)$ 计算时间

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=t;i<=n;i++) {
            swap(x[t], x[i]);
            if (legal(t)) backtrack(t+1);
            swap(x[t], x[i]); }
}
```

5-1 求解0/1背包问题

问题描述：有 n 个重量分别为 $\{w_1, w_2, \dots, w_n\}$ 的物品，它们的价值分别为 $\{v_1, v_2, \dots, v_n\}$ ，给定一个容量为 W 的背包。设计从这些物品中选取一部分物品放入该背包的方案，每个物品要么选中要么不选中，要求选中的物品不仅能够放到背包中，而且具有最大的价值。

例：对下表所示的4个物品求出背包限重 $C=7$ 时的所有解和最佳解。



物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

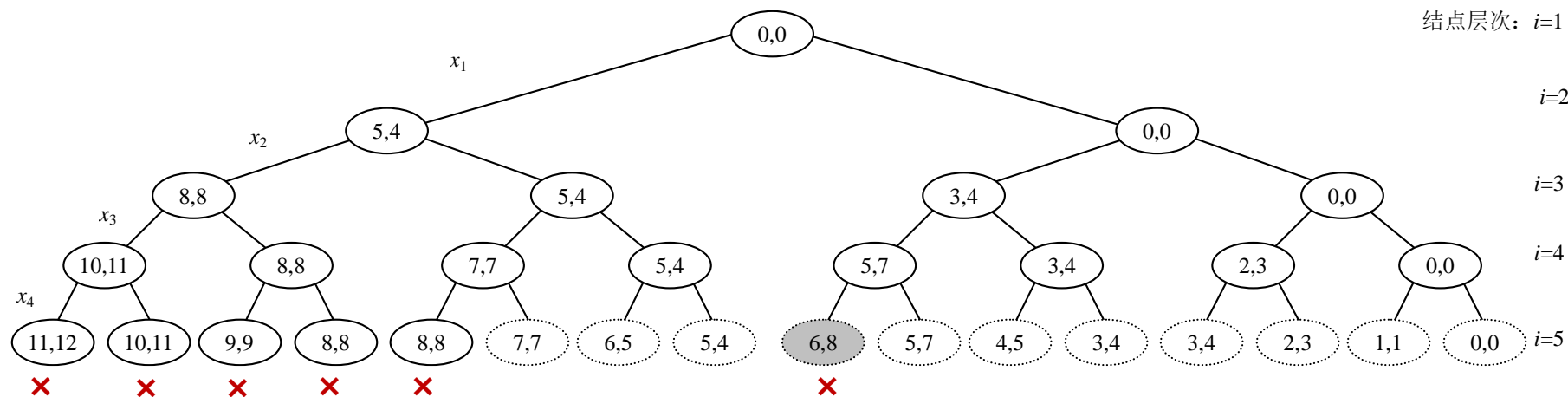
问题求解：对问题的求解过程可用一棵二叉树来描述，每个结点表示背包的一种状态，记录当前放入背包的物品总重量和总价值，每个分枝结点下面有两条边表示对某项物品是否放入背包的两种可能的选择。

用 $x[1..n]$ 数组存放最优解，其中每个元素取1或0， $x[i]=1$ 表示第*i*个物品放入背包中， $x[i]=0$ 表示第*i*个物品不放入背包中。这是一个求最优解问题。

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

对于以下0/1背包问题，在限制背包总重量 $C=7$ 时，描述问题求解过程的解空间树如图所示，每个结点中有两个数值，前者表示放入背包中物品的总重量，后者表示总价值。在所有树叶子结点中，虚线结点表示满足条件 $tw \leq C$ 的结点，其中带阴影的结点的总价值最大，该结点即为最优解结点。

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1



活节点 (栈) : A、B、E、J、U K

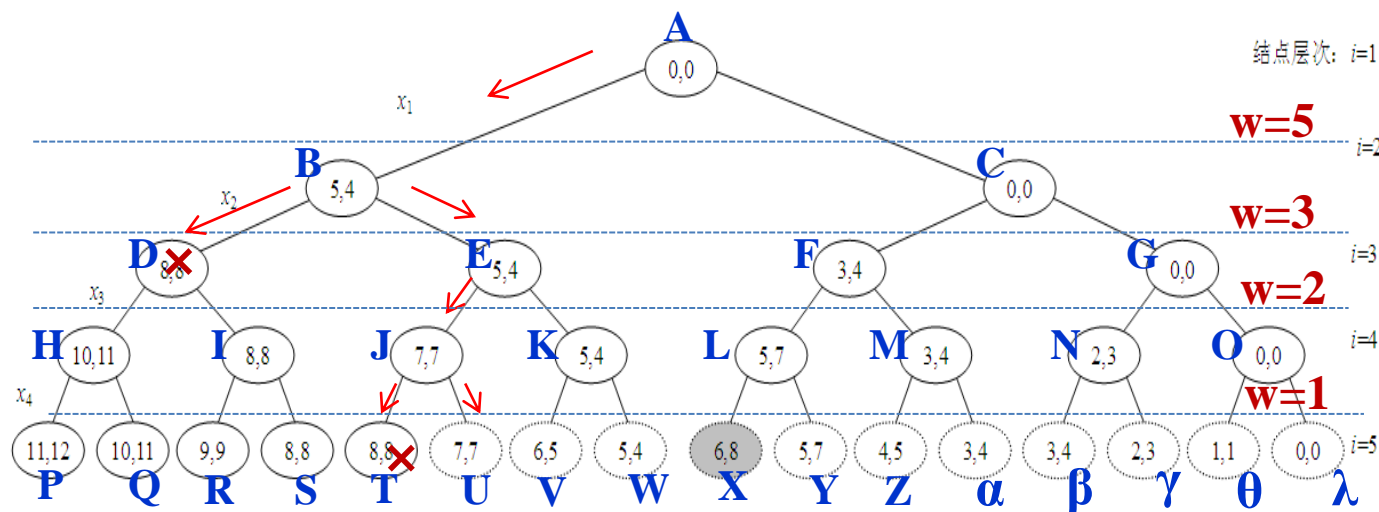
扩展节点: A → B → E → J → U → J → K

死节点: D、T

$w = \{5, 3, 2, 1\}$

$p = \{4, 4, 3, 1\}$

$C = 7$



记录当前背包动态重量 $c_w = 5 \ 7 \ 5$

记录当前背包动态价值 $c_v = 4 \ 7 \ 4$

记录当前最优价值 $best_va = 7$

//深度优先的方法搜索背包问题

// i: 第i件物品

void **search**(int i) //考虑第i个物品

{

if (i>num_all) //找到了一个叶子结点

{

check_answer(); //检查当前解是否可行解, 若是, 与max比较, 如更大, 则更新max

print_answer(); //打印当前解

return;

}

//尚未找完所有物品

//-----1: 选第i件物品, 放入的情况

good_an[i] = 1; //选取第i个物品


search (i+1);

//-----0: 不选第i件物品, 不放入的情况

good_an[i] = 0; //不选取第i个物品

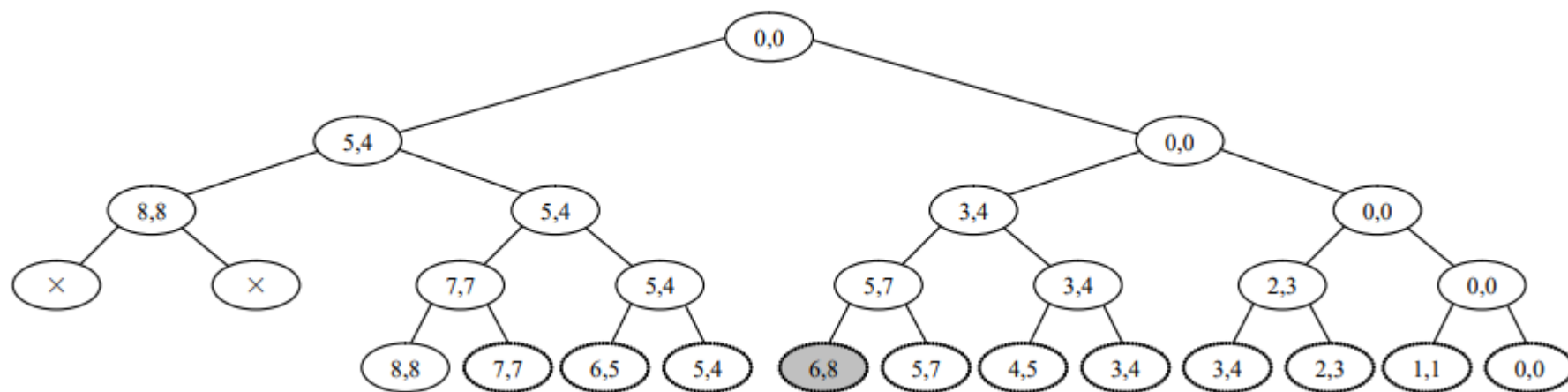
search (i+1);

}



解法1:
不减枝

从中看到，有些分枝结点的总重量已超过了 W ，仍在扩展其孩子结点，这是不必要的，可以增加一个限界条件进行剪枝，剪枝后的解空间树如图所示。



超重剪枝后的解空间树

//深度优先的方法搜索背包问题

// i: 第i件物品

void **search**(int i) //考虑第i个物品

{

if (i>num_all) //找到了一个叶子结点

{

check_max(); //检查当前解, 与max比较, 如果更大, 则覆盖max

print_answer(); //打印当前解

return;

}

//尚未找完所有物品

//-----1: 选第i件物品, 放入的情况, 检查: 第i个物品放入, 是否超重

if (c_w+weight[i] <= max_w) //左孩子结点剪枝: 满足条件时才放入

{

good_an[i] = 1; //选取第i个物品

c_w += weight[i]; //记录选取物品后当前包的重量

c_va += value[i]; //记录选取物品后当前包的价值

search (i+1);

c_w -= weight[i]; // 回溯回来恢复未放入时的状态

c_va -= value[i];

}

//-----0: 不选第i件物品, 不放入的情况

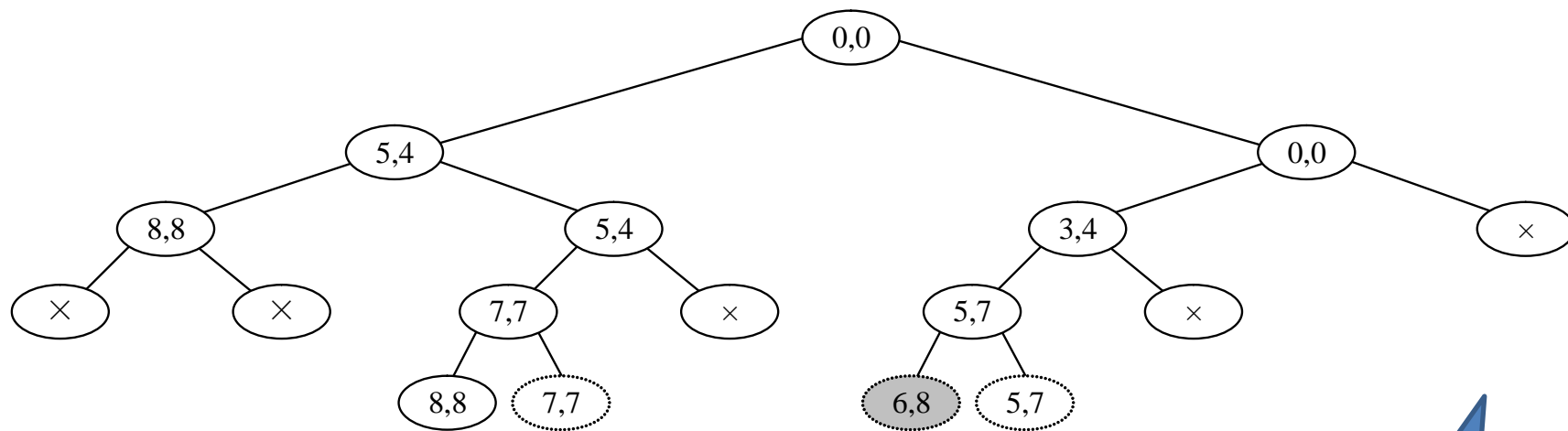
good_an[i] = 0; //不选取第i个物品

search (i+1);

}

解法2:
减枝

从上图看到，只对左子树进行限定，但没有对右子树进行限定，实际上对右子树进行限定很困难。假设**最优解至少取其中的3个物品**（也就是说不选取的物品数应小于等于1，这个条件不一定合理！），从而产生进一步剪枝后的解空间树如图下图所示（这里的算法仍能产生最优解，但比“超重剪枝”页面图的效率高很多）。



进一步剪枝后的解空间树

程序解法：
自己考虑

5-2 装墩问题

问题描述：有2艘船，载重量分别是 c_1 、 c_2 ， n 个集装箱，重量是 w_i ($i=1\dots n$)，且所有集装箱的总重量不超过 c_1+c_2 。确定是否有可能将所有集装箱全部装入这2艘船。

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

例如 当 $c_1=c_2=50$ ， $n=3$ ，

- $w=[10, 40, 40]$ ，可将货箱1和2装到第一艘船上；货箱3装到第二艘船上；
- $w=[20, 40, 40]$ ，虽然货物总重量满足2艘船的可装载总重量，但是无法将全部货箱装船。

问题求解:

若装载问题有解, 采用如下策略可得一个最优装载方案:

- (1)将第一艘轮船尽可能装满;
- (2)将剩余的货箱装到第二艘轮船上, 能装下则找到解!

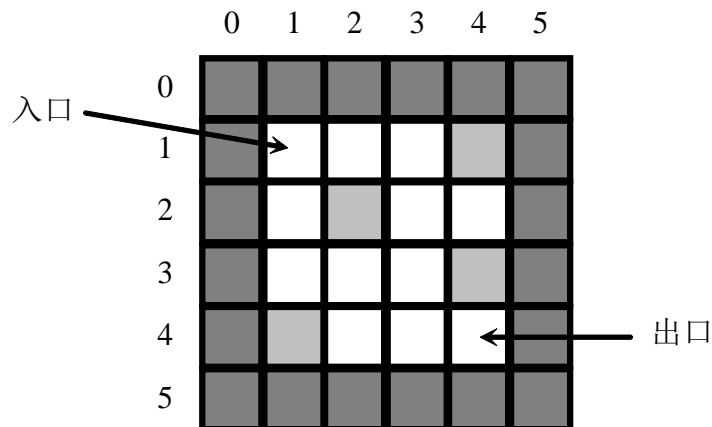
将第一艘船尽可能装满等价于如下0-1背包问题:

$$\max \sum_{i=1}^n w_i x_i \quad \sum_{i=2}^n w_i x_i \leq c_1 \quad x_i \in \{0, 1\}, 1 \leq i \leq n$$

5-3 求解迷宫问题

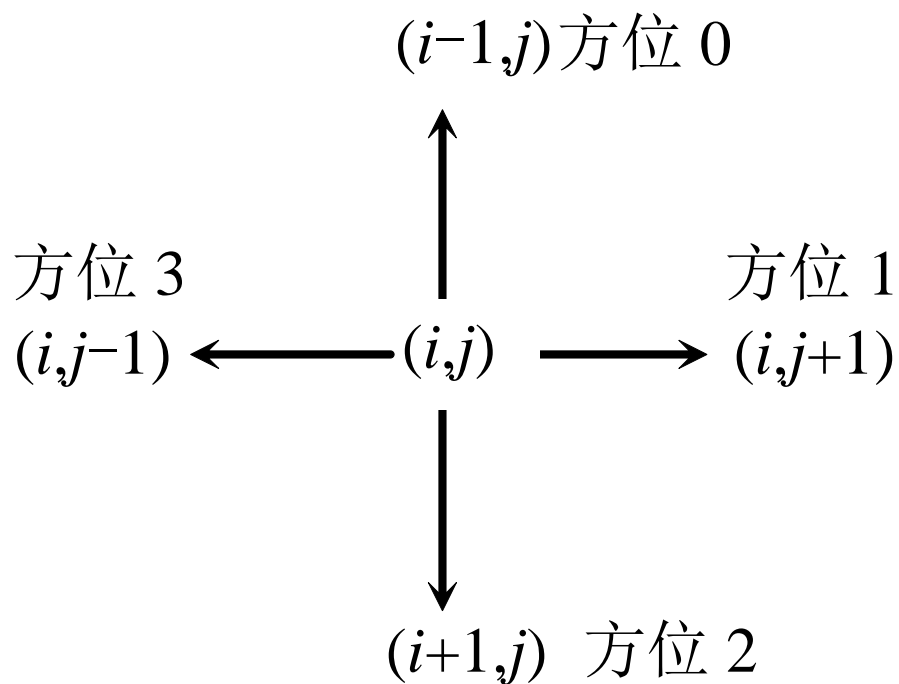
问题描述：给定一个 $M \times N$ 的迷宫图，求所有从指定入口到出口的路径。假设迷宫图如下图所示（其中 $M=6$ ， $N=6$ ，含外围加上一圈不可走的方块，这样做的目的是避免在查找时出界），迷宫由方块构成，空白方块表示可以走的通道，带阴影方块表示不可走的障碍物。

要求所求路径必须是简单路径，即在求得的路径上不能重复出现同一空白方块，而且从每个方块出发只能走向上下左右四个相邻的空白方块。



对于迷宫中的每个方块，有上下左右四个方块相邻，如下图所示。第 i 行第 j 列的方块的位置记为 (i,j) ，

规定上方方块为方位0，并按顺时针方向递增编号。在试探过程中，假设从方位0到方位3的方向查找下一个可走的方块。



采用回溯法递归框架求解迷宫问题

设 $\text{dfs_path}(xi, yi, xe, ye, \text{path})$ 是求迷宫中从 (xi, yi) 到 (xe, ye) 的迷宫路径， path 变量用于保存一条迷宫路径。

当从 (xi, yi) 方块找到一个可走相邻方块 (i, j) 后，递归调用 $\text{dfs_path}(i, j, xe, ye, \text{path})$ 继续求从方块 (i, j) 到出口 (xe, ye) 的一条迷宫路径。伪代码如下：

```
dfs_path(xi, yi, xe, ye, path) ≡  
    将(xi, yi)添加到path中;  
    输出path中的一条迷宫路径;                若(xi, yi)=(xe, ye)  
  
dfs_path(xi, yi, xe, ye, path) ≡  
    将(xi, yi)添加到path中;                若(xi, yi)不为出口且可走  
    找出(xi, yi)四周的一个相邻可走方块(i, j);  
    maze[xi][yi]=-1;  
    dfs_path(i, j, xe, ye, path);  
  
    path回退一步并置  
    maze[xi][yi]=0;
```

```

void dfs_path(int xi,int yi,int xe,int ye,PathType path)
//求(xi,yi)→(xe,ye)的所有迷宫路径
{
    int di,i,j;
    if (xi==xe && yi==ye)           //找到了出口,输出路径
    {
        path.data[path.length].i=xi; //将出口放入路径中
        path.data[path.length].j=yi; path.length++;
        dispapath(path);             //输出一条路径
    }
    else
    {
        if (maze[xi][yi]==0)         //(xi,yi)不是出口
            //若(xi,yi)是一个可走方块
        {
            di=0;
            while (di<4)             //找(xi,yi)的四周的相邻方块(i,j)
            {
                path.data[path.length].i = xi;
                path.data[path.length].j = yi;
                path.length++;        //路径长度增1
                switch(di)           //找一个相邻方块(i,j)
                {
                    case 0: i=xi-1; j=yi; break;
                    case 1: i=xi;   j=yi+1; break;
                    case 2: i=xi+1; j=yi; break;
                    case 3: i=xi;   j=yi-1; break;
                }
                maze[xi][yi]=-1;      //避免重复找路径
                dfs_path(i,j,xe,ye,path); //以新扩展点为开始,继续搜索

                maze[xi][yi]=0;       //恢复(xi,yi)为可走的
                path.length--;        //回退一个方块,路径长度减1
                di++;                 //找(xi,yi)的下一个方位的相邻方块
            }
        }
    }
}

```


教材上的
代码写法。

```
// 深度优先搜索路径(start_r, start_c)→(end_r, end_c)
// start_r, start_c: 本次搜索的开始位置
// end_r, end_c: 搜索的最终目标
// 返回值: 找到路径 true; 没有找到 false;
```

```
bool dfs_path(int start_r, int start_c, int end_r, int end_c)
{
    int new_r, new_c;

    // 搜索到终点, 则打印路径, 结束本条路径的搜索。
    if (start_r==end_r && start_c==end_c)
    {
        print_path(end_r, end_c);
        return true;
    }
}
```

// 下面处理还未搜索到终点的情况, 分别在4个方向上寻找新的扩展点, 下页继续



提高代码阅读性, 将子功能放入子函数。

// 分别在4个方向上寻找新的扩展点

int new_p; // 记录新的扩展位置

此处进行四个方向的搜索!

for (int i=0; i<4; i++)

{ new_p = make_new_place(start_r, start_c, i);

// 如果能够成功创建, 则表示在这个位置上可以扩展

if (new_p != -1)

{

//----- 1:choice 扩展 -----

//record the path:

path[pathLen++] = new_p //记录路径, 步长++

//把这个点作为开始起点, 继续进行深度搜索

new_r = new_p/COLUMN; new_c = new_p%COLUMN;

maze[new_r][new_c] = -1; //this is coved, 避免重复找路径, important!

if (dfs_path(new_r, new_c, end_r, end_c)) // dfs

{

return true; //如果成功搜索到终点返回结束, 如果搜索所有路径的情况, 这里怎么处理?

}

//---- 0:recover data 恢复这个位置为未走过!!! 准备遍历下个方向-----

maze[new_r][new_c] = 0;

path[pathLen] = 0;

pathLen --;

}

} // for (int i=0; i<4; i++)

return false; // 没有成功返回, 则返回失败

} //end

解法1: 搜索
时用path数组
记录路径。

// 分别在4个方向上寻找新的扩展点

int new_p; // 记录新的扩展位置

for (int i=0; i<4; i++)

{ // 分别在右、下、左、上方向上创建新的位置

new_p = make_new_place(start_r, start_c, i);

// 如果能够成功创建，则表示在这个位置上可以扩展

if (new_p != -1)

{

//----- choice 扩展 -----

//记录new_p的父亲节点 (start_r, start_c)，同时表示该节点已经被扫描过了。

record_its_father(new_p, start_r, start_c);

steps++; //记录搜索的步数

// 把这个点作为开始起点，继续进行深度搜索

place_r = new_p/COLUMN; place_c = new_p%COLUMN;

result= dfs_path(place_r, place_c, end_r, end_c); // dfs

if (result==true) //如果成功搜索到终点，返回结束

{

return true; //考虑一下搜索所有路径的情况，这里怎么处理？

}

//----- recover data 恢复这个位置为未走过!!! -----

maze[place_r][place_c] = 0;

steps--;

}

} // for (int i=0; i<4; i++)

return false; // 没有成功返回，则返回失败

} //end

此处进行四个方向的搜索！

解法2：走过的路径在迷宫上直接做标记！

```

// 根据director创建新的扩展位置 (place_r, place_c), 如果不能创建新扩展位, 返回-1
// place_r, place_c: 开始位置
// director: 扩展方向
// 0: right; (0, +1)  1: down;  (+1, 0)  2: left;  (0, -1)  3: up;    (-1, 0)
// return: 根据扩展的位置计算所得的数值(row*COLUMN+column)
// return: -1, 如果该director方向不能扩展
int make_new_place(int place_r, int place_c, int director)
{
    int ret_place = -1;
    // 获取新的可能位置
    int new_r=place_r, new_c=place_c;
    switch (director)
    {
        case 0:    new_c++; //right
                   break;
        case 1:    new_r++; // down;
                   break;
        case 2:    new_c--; //left
                   break;
        case 3:    new_r--; //up
                   break;
    }
    // 检查新位置是否越界
    if ((new_r <=0) || (new_r >= ROW) || (new_c <=0) || (new_c >= COLUMN))
    {
        ret_place = -1; // 越界的情况, 返回-1
    }else
    {
        // 如果该位置是空白, 可以扩展, 则返回其位置号
        if (maze[new_r][new_c] == 0)
            ret_place = new_r*COLUMN + new_c;
    }
    return ret_place; // 统一返回
}

```

int direction = {{1,0},{0,1}}{-1,0},{0,-1}}
 new_c = new_c + direction[i][0];
 new_r = new_c + direction[i][1];

一个迷宫数据的例子

```
// -2: start
// -1: wall, can not get
// >99: record its path
// 0: can move to

int maze[ROW][COLUMN] = {
    // 0      3      5      7      9
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, 0, 0, 0, -1, 0, 0, -1, 0, -1}, //1
    {-1, 0, -1, 0, -1, -1, 0, 0, 0, -1},
    {-1, 0, 0, 0, -1, -1, 0, -1, -1, -1}, //3
    {-1, 0, -1, 0, 0, 0, 0, 0, 0, -1},
    {-1, 0, -1, -1, -1, -1, -1, -1, -1, -1}, //5
    {-1, 0, 0, 0, -1, 0, -1, 0, 0, -1},
    {-1, 0, -1, -1, -1, 0, 0, 0, -1, -1}, //7
    {-1, 0, 0, 0, 0, 0, -1, 0, 0, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
    // 0      3      5      7      9
};
```

简化版迷宫问题描述：给定一个 $M \times N$ 的迷宫图，求是否从指定入口到出口存在路径：

给一个 20×20 的迷宫、起点坐标和终点坐标，问从起点是否能到达终点。

输入数据：'.'表示空格；'X'表示墙。



//以 (row、col) 为起点, 搜索是否可以到达终点

void search(int row, int col)

```
{  
    int r,c;  
    a[row][col]=1;    // 走过的地方都刷1  
  
    r=row;            //左  
    c=col-1;  
    if(canplace(r,c)) //判断(r,c)位置是否已经走过  
        search(r,c); //递归搜索(r,c)  
  
    r=row+1;          //下  
    c=col;  
    if(canplace(r,c)) //判断(r,c)位置是否已经走过  
        search(r,c); //递归搜索(r,c)  
  
    r=row;            //右  
    c=col+1;  
    if(canplace(r,c)) //判断(r,c)位置是否已经走过  
        search(r,c); //递归搜索(r,c)  
  
    r=row-1;          //上  
    c=col;  
    if(canplace(r,c)) //判断(r,c)位置是否已经走过  
        search(r,c); //递归搜索(r,c)  
}
```



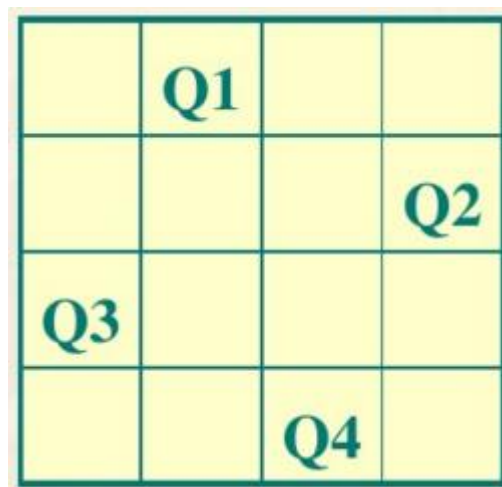
该递归函数什么情况终止?

```
int canplace(int row, int col)
{
    //如果在20*20范围内，且该位置是可走的
    if(row>=0&&row<20&&col>=0&&col<20&&a[row][col]==0)
        return 1;
    else
        return 0;
}
```

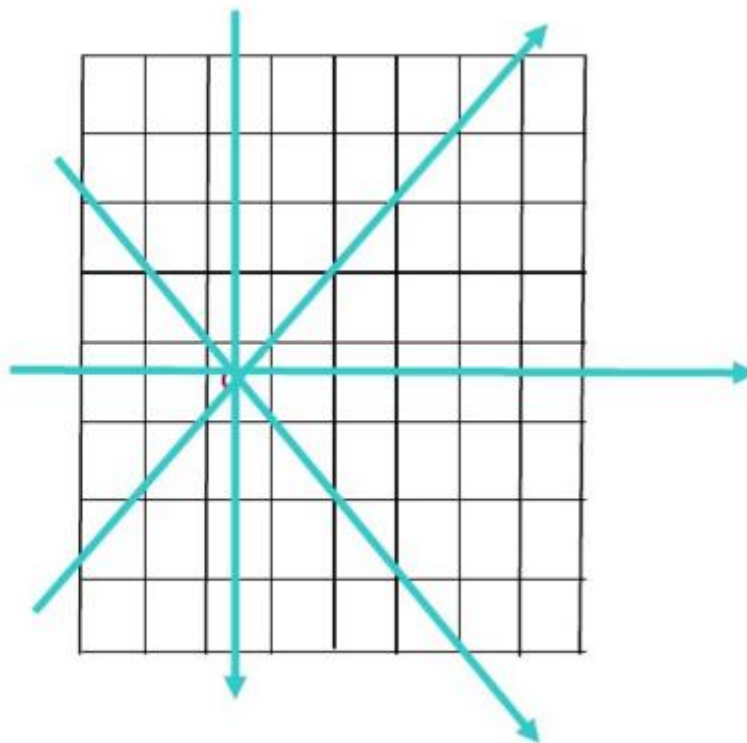
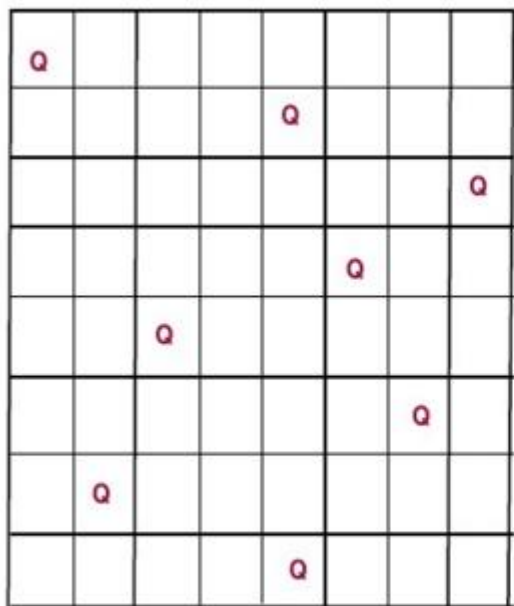
5-4 求解 n 皇后问题

问题描述：在 8×8 （ $n \times n$ ）格的国际象棋上摆放 n 个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。

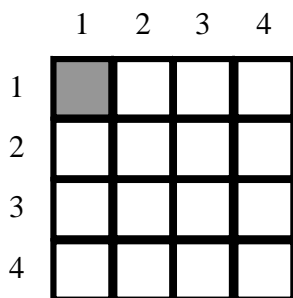
八皇后问题，是一个古老而著名的的问题，是回溯算法的典型案例。该问题是国际西洋棋棋手马克斯·贝瑟尔于1848年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。高斯认为有76种方案。



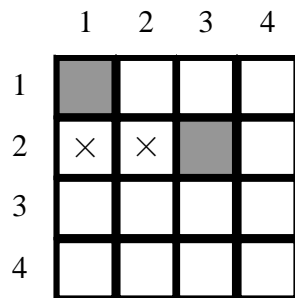
任意两个皇后都不能处于同一行、
同一列或同一斜线上：



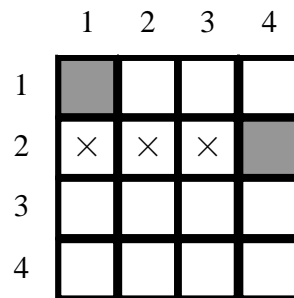
以4皇后问题为例，找第一个解的过程如图所示。



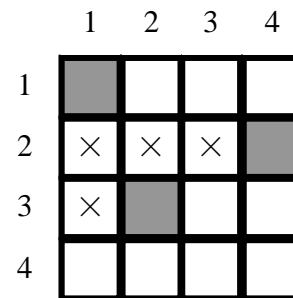
(a) 放第 1 个皇后



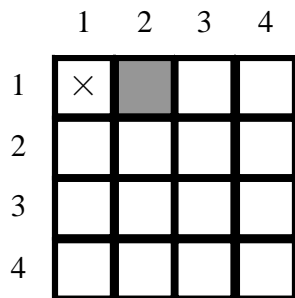
(b) 放第 2 个皇后



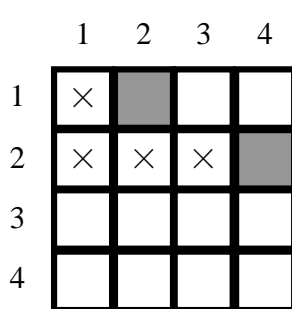
(c) 第 3 个皇后放不下，回溯到第 2 个皇后，找其下一个位置



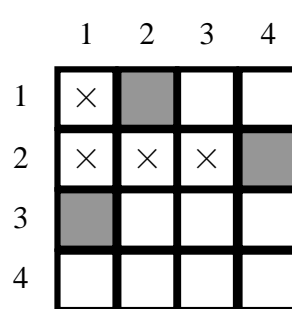
(d) 放第 3 个皇后



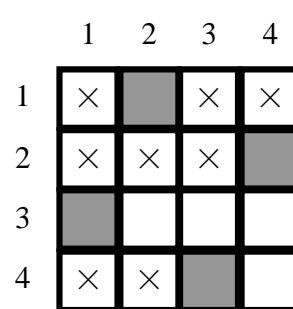
(e) 第 4 个皇后放不下，回溯到第 3 个皇后，没有合适的位置，回溯到第 2 个皇后，也没有合适的位置，回溯到第 1 个皇后，找其下一个位置



(f) 放第 2 个皇后



(g) 放第 3 个皇后



(h) 放第 4 个皇后，找到一个解

动画演示，见下一页

K=0

搜索过程：进入新一行，该行上按顺序逐个格子尝试，直到能放为止（不冲突、不越界）

K=1

K=2

K=3

K=4

算法描述：

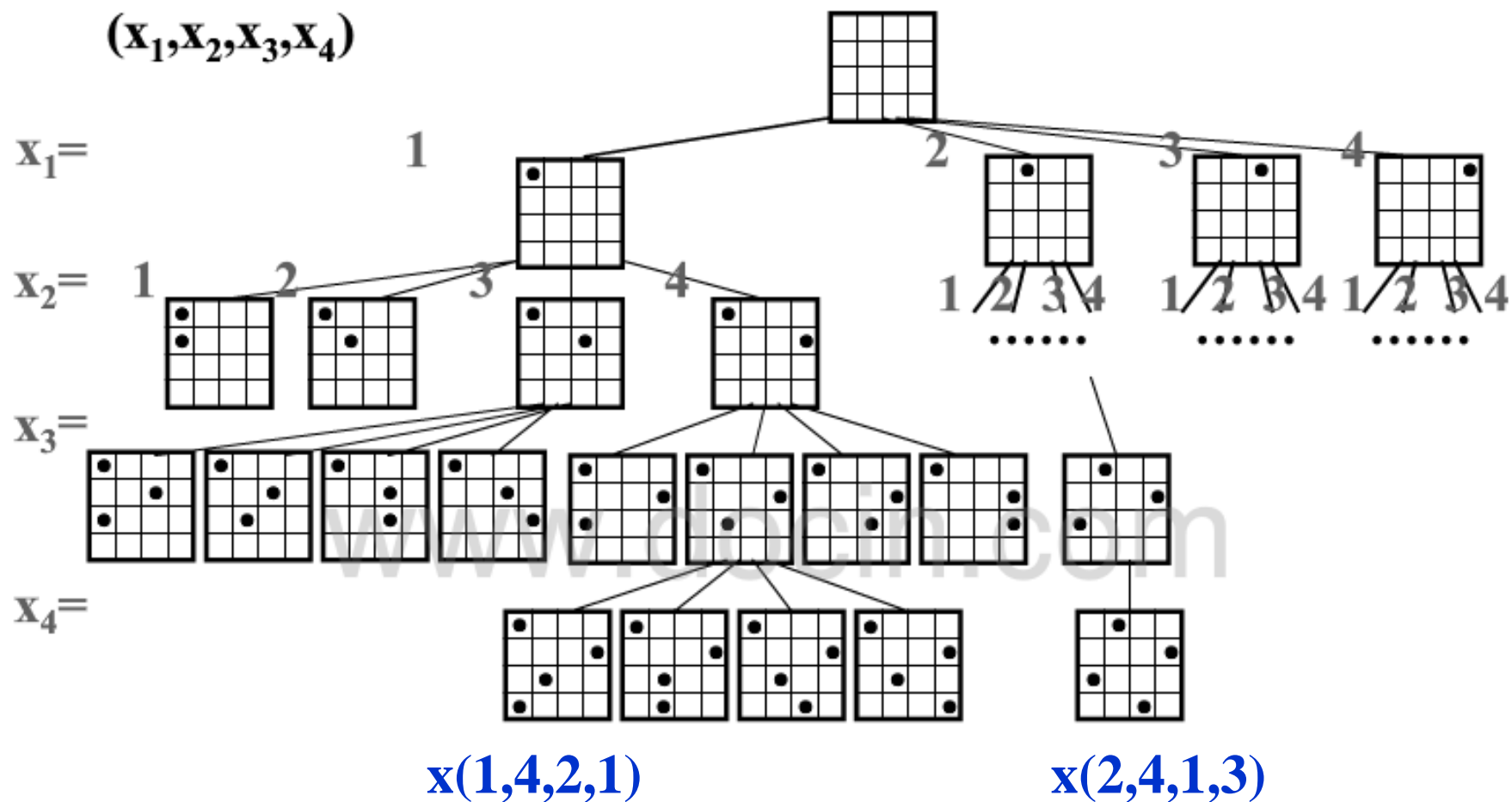
1. 产生一种新放法
2. 冲突，继续找，直到找到不冲突----不超范围
3. if 不冲突 then $k < n \rightarrow k+1$
 $k = n \rightarrow$ 一组解
4. if 冲突 then 回溯

回溯









回溯

得出解后可以继续刚才的做法

以4皇后问题为例的解空间树。



如何存储皇后?

q [1]							
q [2]							
q [3]							
q [4]							
q [5]							
q [5]							
q [7]							
q [8]							

8皇后问题的一个解

$x(2,4,6,8,3,1,7,5)$

q[i]代表第i个皇后的列位置。第i个皇后自然处于第i行。

从中总结出 n 皇后求解的规则：

- ① 用数组 $q[]$ 存放皇后的位置， $q[k]$ 表示第 k 个皇后放置的位置， n 皇后问题的一个解是： $q[1], q[2], \dots, q[n]$ ，数组 q 的下标0元素不用。
- ② 先放置第1个皇后，然后依2、3、...、 n 的次序放置其他皇后，当第 n 个皇后放置好后产生一个解。为了找所有解，此时算法还不能结束，继续试探第 n 个皇后的下一个位置。
- ③ 第 k ($k < n$) 个皇后放置后，接着放置第 $k+1$ 个皇后，在试探第 $k+1$ 个皇后的位置时，都是从第1列开始的。

- ④ 当第 k 个皇后试探了所有列都不能放置时，则回溯到第 $k-1$ 个皇后，此时与第 $k-1$ 个皇后的位置 $q[k-1]$ 有关，如果第 $k-1$ 个皇后的列号小于 n 即 $q[k-1] < n$ ，则将其移到下一列，继续试探；否则再回溯到第 $k-2$ 个皇后，依此类推。
- ⑤ 若第1个皇后的所有位置回溯完毕，则算法结束。
- ⑥ 放置第 k 个皇后应与前面已经放置的 $k-1$ 个皇后不发生冲突。

// 对第t个皇后（皇后t）设置她的位置

```
void queens(int t)  // t代表第几个皇后，数值从1~8
{
    if(t>num)  //当t大于num（例如8）时，表示走通了一条路径，所有皇后都在位置上了
    {
        sum++;  // 累计已经找到的可行方案数目
        print_result();
        return;
    }

    search_times++;  // 记录总共的递归搜索次数。

    for(int col= 1; col<num+1; i++)  // 横向遍历棋盘，看皇后t放到哪个位置合适
    {
        if(check_place(t, col))  // 检查t皇后放到的位置 col 是否正确。
        {
            // if right, continue check next queen.
            q[t] = col;  // 将皇后t放到col这个位置
            queens(t+1);
            q[t] = 0;  // 递归回来恢复递归之前的状态
        }
        // go on to check next place(col)
    }
}
```

// 检查将第k个皇后，放置到（当然是第k行的） col位置是否合适
 // k皇后的位置，不能与其他已经放置的皇后（q[1]...q[k-1]），位于同一列上。
 // 也不能在斜线上（行上的差与列上的差构成等腰直角三角形）

```
bool check_place(int k, col)
```

```
{
```

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

```
    {
```

```
        if(abs(col - q[j]) == abs(k-j) || col == q[j])
```

```
            return false;
```

```
    }
```

```
    return true;
```



```
}
```



在同一斜线上

在同一列上

q[j]

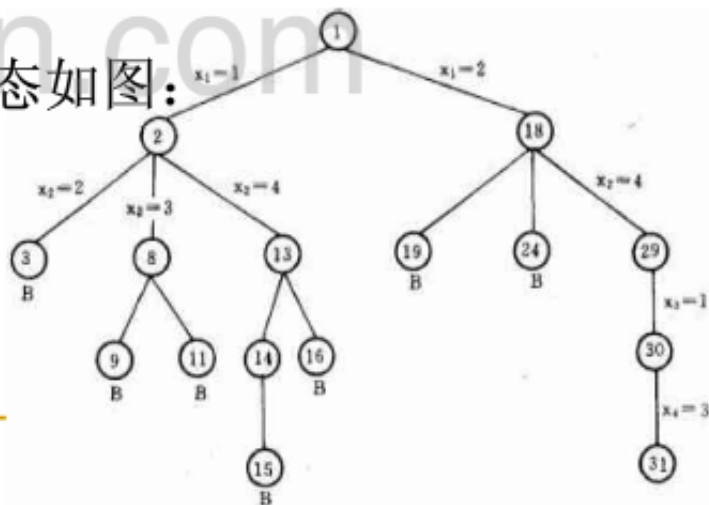
q[k]

		1	2	3	4	5	6	7	8
1	1	0	1	2		-4	5	6	7
2	-1	0	1	2	3	4	5	6	7
3	-2	-1	0	1	2	3	4	5	6
4	-3	-2	-1	0	1	2	3	4	5
5	-4	-3	-2	-1	0	1	2		3
6	-5	-4	-3	-2	-1	0	1	2	3
7	-6	-5	-4	-3	-2	-1	0	1	2
8	-7	-6	-5	-4	-3	-2	-1	0	1

		1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8	9
2	2	3	4	5	6	7	8	9	10
3	3	4	5	6	7	8	9	10	11
4	4	5	6	7	8	9	10	11	12
5	5	6	7	8	9	10	11	12	13
6	6	7	8	9	10	11	12	13	
7	7	8	9	10	11	12	13	14	15
8	8	9	10	11	12	13		15	16

搜索代价

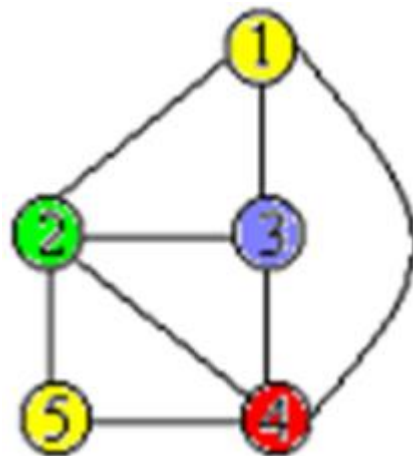
- 对于**4Queen**，若每个分支都是“一分为四”，则：
 - **0**个皇后的状态：**1**个，即 **4^0** ；
 - **1**个皇后的状态：**4**个，即 **4^1** ；
 - **2**个皇后的状态：**16**个，即 **4^2** ；
 - **3**个皇后的状态：**64**个，即 **4^3** ；
 - **4**个皇后的状态：**256**个，即 **4^4** ；
 - 共**341**个状态。
- 使用回溯策略，实际扫描过的状态如图：
 - 共**16**个，仅占**5%**。



5-5 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。

问题背景：用 m 种颜色为地图着色，使得地图上相邻区域颜色不同。可以证明至多可用4种颜色来着色，并使任何两个有公共边界的相邻区域没有相同的颜色。

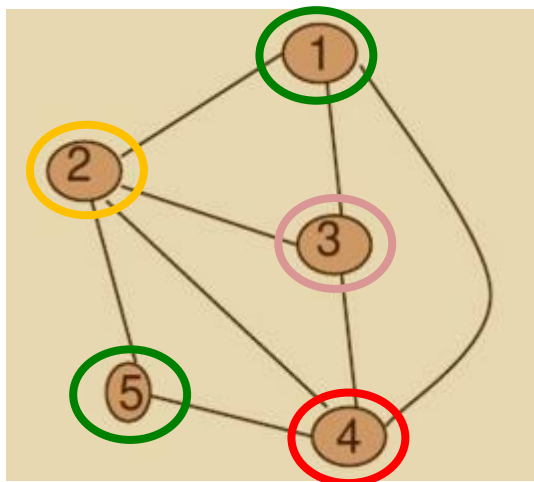


问题求解：

输入： m （颜色种类）、连通图。

输出：是否这个图是 m 可着色的。

思考：如何存储图？



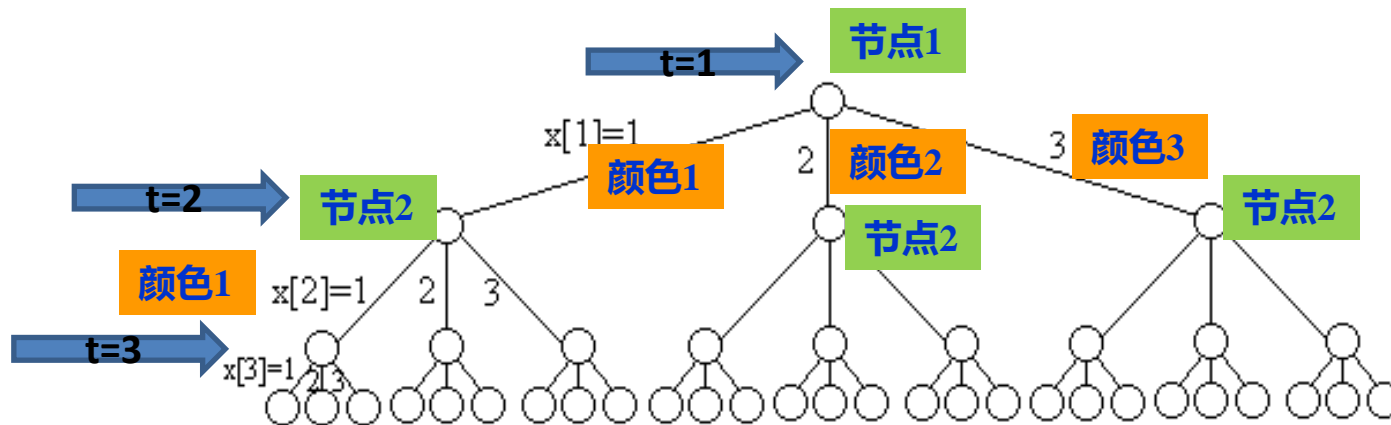
不可达

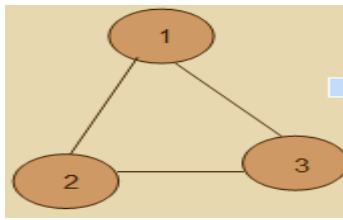
可达

	N1	N2	N3	N4	N5	color
N1	1	1	1	1	0	color1
N2	1	1	1	1	1	color2
N3	1	1	1	1	0	color3
N4	1	1	1	1	1	color4
N5	0	1	0	1	1	color5

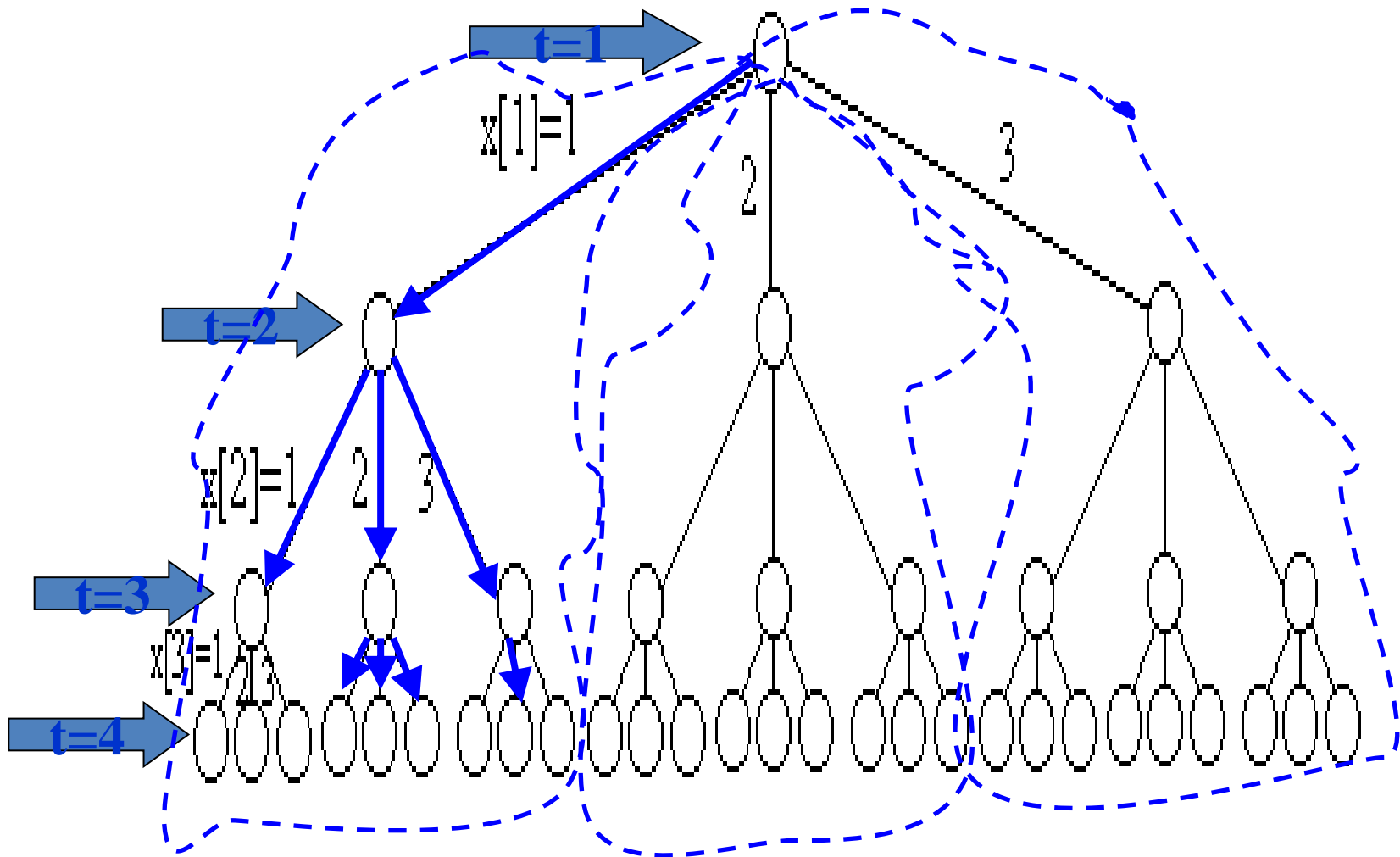
例如有一组解：43214（5个点的着色编号）

- **解向量**: $\text{color}[t] = i$ (i 为1~ m 的整数) 表示第 t 个节点着色 i 号颜色。
- $(\text{color}_1, \text{color}_2, \dots, \text{color}_n)$ 表示**顶点** $t(1 \dots N)$ 所着颜色 $x[i]$
- **可行性约束函数**: **顶点** t 与已着色的相邻顶点颜色不重复。





模拟演示



- **//对顶点t, 进行着色**

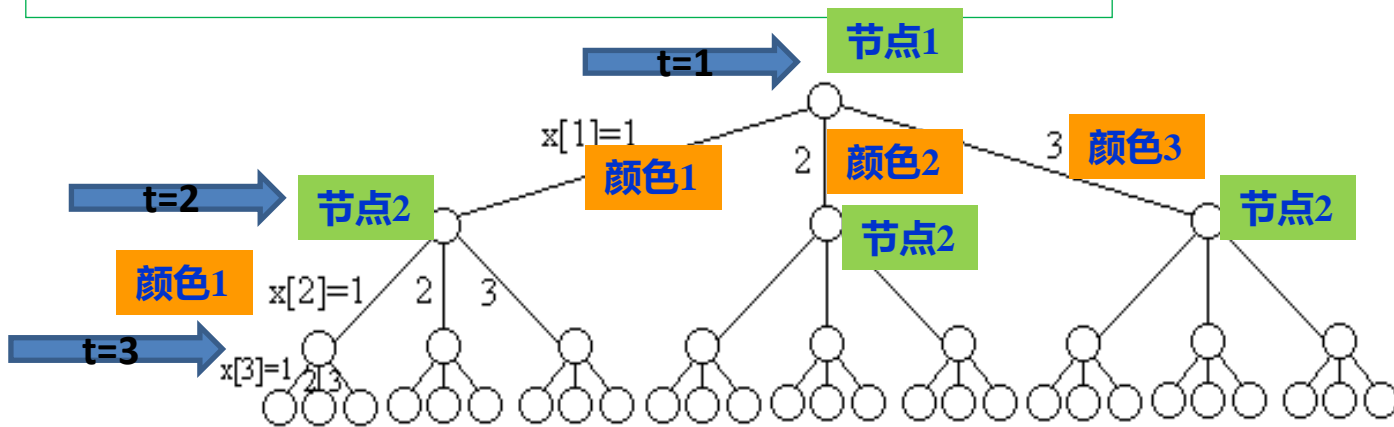
```
void put_color(int t)
{
    if (t>N) solution++; 打印这一组color解!
    else
        for (int i=1;i<=m;i++) {
            if (ok(t, i)) {
                color[t]=i;
                put_color(t+1);
                color[t]=0; //Backtrace
            }
        }
}
```

**N个节点都已经着上了色，
有解！**

solution记录这是第几组解
没有解的情况，走不到最后节点的。

遍历所有颜色*i*，对当前节点*t*，如果可着色*i*，则着色，并看下一个节点。

回溯回来，恢复初始值！



- // 检查颜色可用性（根据上位顶点已经着过的颜色，查找可以着的颜色）
- 对节点 k ，给其着色 i ，是否可行。
 - 可行，返回 $true$ ，否则，返回 $false$ 。

节点 k

颜色 i

```
bool ok(int  $k$ , int  $i$ )
{
    for (int  $j=1$ ;  $j \leq k$ ;  $j++$ )    //对节点 $k$ 之前的所有已着色节点进行检查
    {
        if (( $a[k][j]==1$ ) && ( $color[j]==i$ )) return false;
    }
    return true;
}
```

遍历已经着色过的所有节点 j （ $j \leq k$ ），检查节点 k ，所有与 k 联通的已经着色的边（ $a[k][j] == 1$ ），是否：已经着色为 i ？

注意节点数和图矩阵的对应关系。

```
// graph_color.cpp : Defines the entry point for the console application.
//回溯法：地图着色问题

#include "stdafx.h"
#include<stdio.h>
#include<string.h>

#define N1 5 // node number 节点数

// 图1的情况
int a1[N1+1][N1+1]={
    0,0,0,0,0,0, //为了编号对应，把第一行和第一列都废弃不用
    0,1,1,1,1,0,
    0,1,1,1,1,1,
    0,1,1,1,1,0,
    0,1,1,1,1,1,
    0,0,1,0,1,1,
}; //adjacency matrix
```

5-6 全排列问题

问题描述：穷举 n 个元素的所有排列。

为方便起见，用123来示例。123的全排列有123、132、213、231、312、321这六种。

问题分析：如何确定问题的解空间？

一组 n 元一维向量 $(x_1, x_2, x_3, \dots, x_n), 1 \leq x_i \leq n, 1 \leq i \leq n$

可用一维数组存储正在搜索的向量。

约束条件： x_i 互不相同。

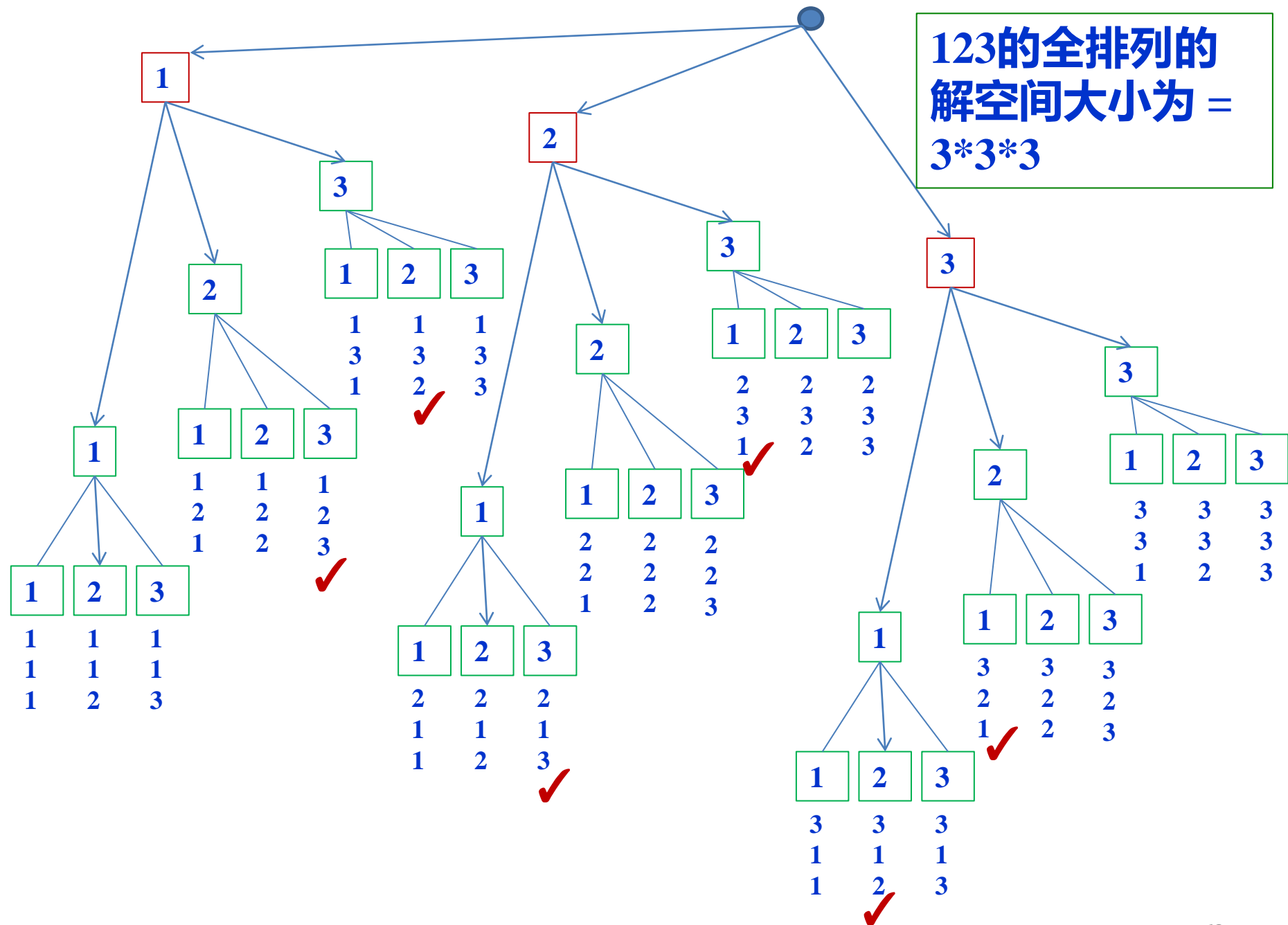
可设置 n 个元素的数组 $d[]$ ，数组元素用来记录数据1—— n 的使用情况，已使用置1，未使用置0。

// 对第K个位置置值

```
void set(int place)
{
    // 如果搜索到终点, 则打印路径, 结束本条路径的搜索。
    if (place == g_n)
    {
        print_it();
        g_solution++;
        return;
    }
    // 寻找下一个扩展点, 分别在这n个数里面挑选
    for (int i=1; i<=g_n; i++)
    {
        if (ok(i)) // 如果数字i是可以扩展的数字
        {
            g_a[place] = i; // 扩展它!
            mark(i);        // 标记它已使用!
            set(place+1);    // 深度搜索下一个位置!

            g_a[place] = 0; // 回溯回来的恢复!
            unmark(i);
        }
    }
}
```

123的全排列的
解空间大小为 =
 $3*3*3$



算法分析：以上全排列问题的复杂度为 $O(n^n)$ ，不是一个好的算法。

问题思考：首先考虑213和321这二个数是如何得出的。显然这二个都是123中的1与后面两数交换得到的。然后将123的第二个数和每三个数交换得到132。同理可以根据213和321来得231和312。

因此可以知道——全排列就是从第一个数字起每个数分别与它后面的数字交换。

123

位置1和1交换, 2和2交换, 3和3 交换

132

位置2和位置3 交换

213

位置1和2 交换, 2和2交换, 3和3交换

231

位置2和位置3 交换

321

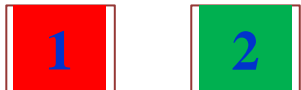
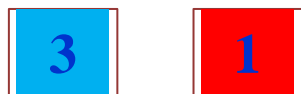
位置1和3 交换, 2和2交换, 3和3交换

312

位置2和位置3 交换



place1 place2 place3

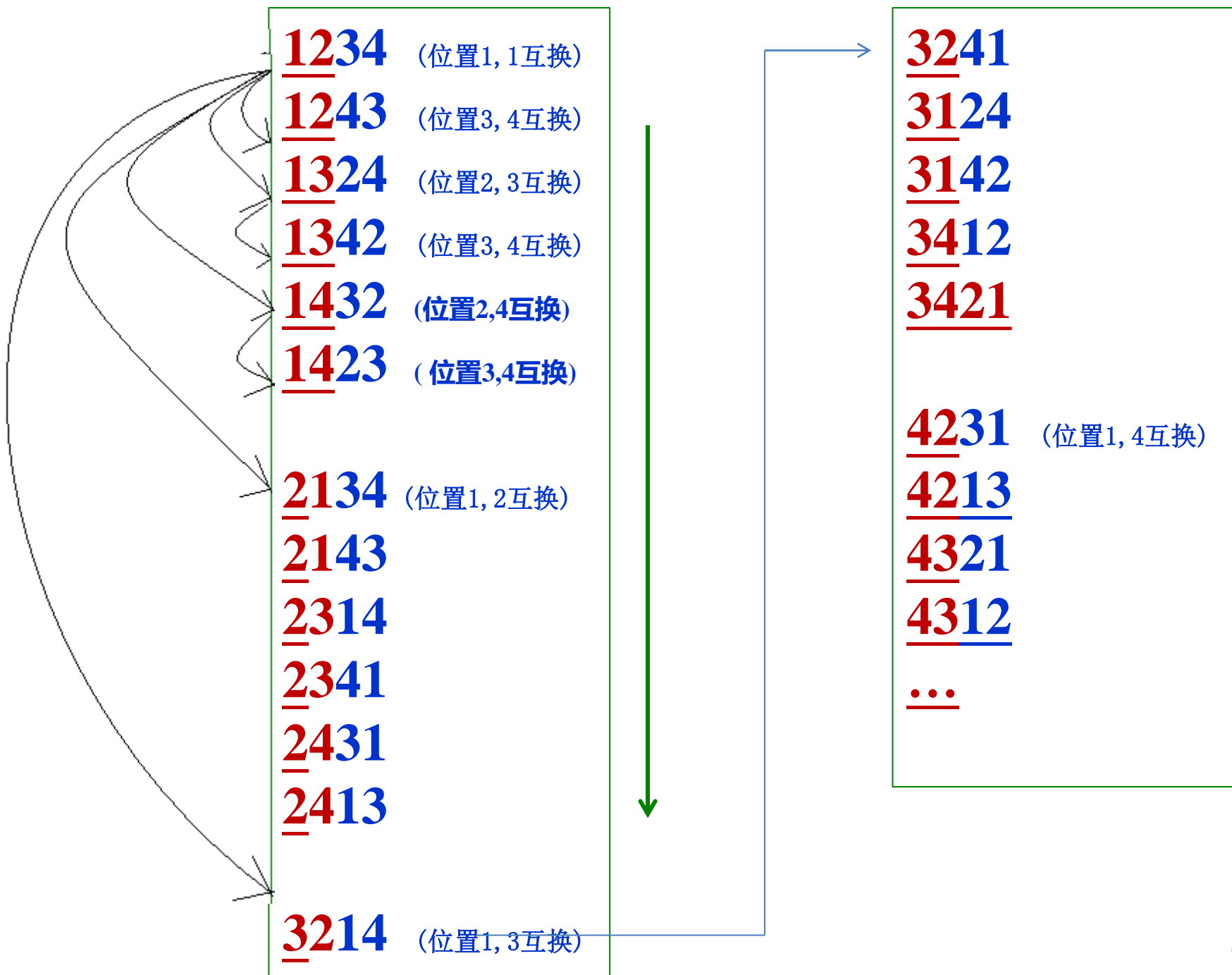


```
for (int i=place; i<=g_n; i++) // 从当前位置起，与它后面的每个数挨个交换
{
    swap_it(place, i); //当前位置数与后面位置数交换！由交换获得新扩展！
    set(place+1);      //深度递归
    swap_it(place,i);  // 恢复原排列
}
```

第一个位置的第1次交换

第一个位置的第2次交换

第一个位置的第3次交换



// 对第K个位置置值(使用交换的方法获取扩展)

```
void set(int place) //place位置范围: 0 ~ g_n-1
{
    // 如果搜索到终点, 则打印路径, 结束本条路径的搜索。
    if (place > g_n)
    { print_it(); g_solution++; return; }

    for (int i=place; i<=g_n; i++) // 从当前位置起, 与它后面的每个数挨个交换
    {
        swap_it(place, i); //当前位置数与后面位置数交换! 由交换获得新扩展!
        set(place+1);      //深度递归
        swap_it(place, i); // 恢复原排列
    }
}
```

算法分析: 全排列算法的复杂度为 $O(n!)$ 。

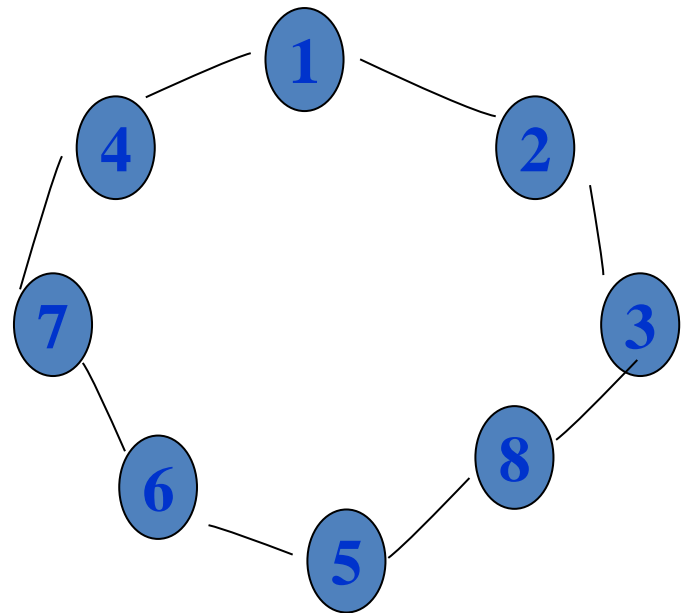
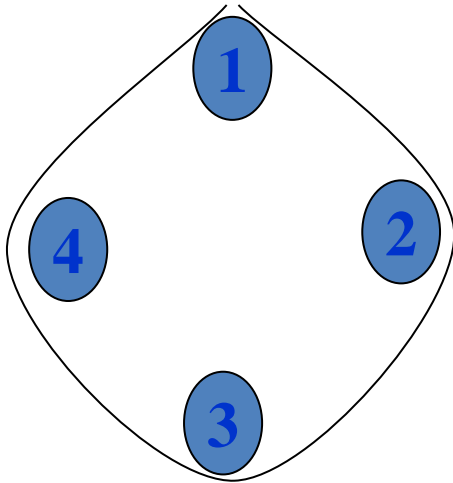


```
void backtrack (int t)
{
    if (t>n) output(x); //递归结束条件：搜索到叶子结点,输出一个可行解
    else
        for (int i=t; i<=n; i++) //枚举t的所有可能的扩展节点
        {
            swap(x[t], x[i]); //通过交换获取新的扩展点
            if (legal(t)) //节点t满足可扩展
            {
                backtrack(t+1); //递归执行进行下一层（深一层）搜索
            }
            swap(x[t], x[i]); //注意：进行回溯回来的清理工作!
        }
}
```

遍历排列树需要 $O(n!)$ 计算时间

5-7 素数环问题

问题描述： 把从1到20这20个数摆成一个环，要求相邻的两个数的和是一个素数。



【想法】这个素数环有20个位置，每个位置可以填写的整数有1~20共20种可能，可以对每个位置从1开始进行试探，约束条件是正在试探的数满足如下条件：

- (1) 与已经填写到素数环中的整数不重复；**
- (2) 与前面相邻的整数之和是一个素数；**
- (3) 最后一个填写到素数环中的整数与第一个填写的整数之和是一个素数。**

在填写第 k 个位置时，如果满足上述约束条件，则继续填写第 $k+1$ 个位置；如果1~20个数都无法填写到第 k 个位置，则取消对第 k 个位置的填写，回溯到第 $k-1$ 个位置。

问题分析：

- ✓ 搜索从1开始，每个空位有2~20共19种可能；
- ✓ 填进去的数合法：与前面的数不相同；与前边相邻的数的和是一个素数；
- ✓ 第20个数还要判断和第1个数的和是否素数。

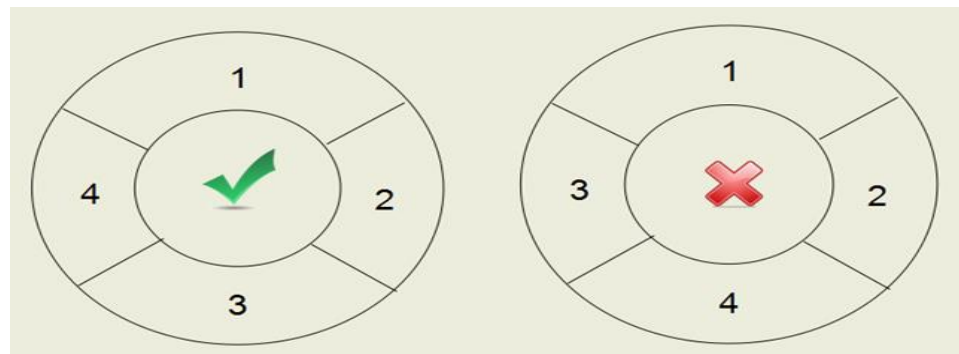
算法流程：

- 1、数据初始化；
- 2、递归地填数：

与前面的数不相同；
与前边相邻的数的和是一个素数；

判断第*i*种可能是否合法？

- A、如果合法：填数；判断是否到达目标（20个已填满）：
是，打印结果；不是，递归填下一个；
- B、如果不合法：选择下一种可能；



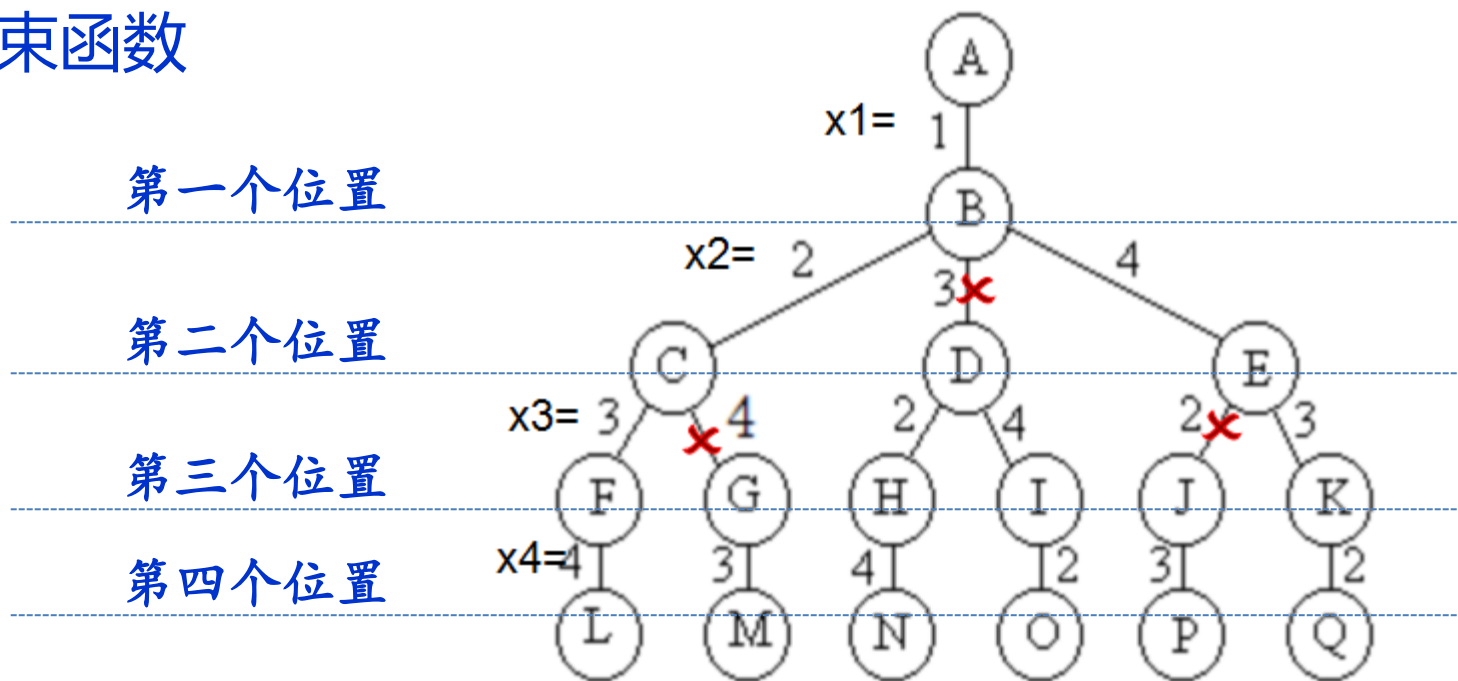
问题分析

1> 解向量: $\langle x_1, x_2, \dots, x_n \rangle$

2> 解空间树: 排列树, $(n-1)!$ 个叶子结点

3> 剪枝函数: $\text{isPrime}(x[t-1] + x[t])$,

$t=2, 3, \dots, n$ 约束函数



// 对第K个位置置值(使用传统挑选方法)

```
void set(int place) //place位置范围: 1~g_n
{
    // 如果搜索到终点, 则打印路径, 结束本条路径的搜索。
    if (place > g_n)
    {
        if (isprime(g_a[1]+g_a[g_n])) //如果首尾可以成环, g_a[]用于存放对应的数
        {
            print_it();
            g_solution++;
        }
        return;
    }

    for (int i=1; i<g_n+1; i++) //遍历所有的可能扩展
    {
        if (ok(place, i)) //如果向位置place设置i是可以的
        {
            g_a[place] = i; //向位置place设置i
            mark(i); // 标记它已使用: g_mark[i] = 1;
            set(place+1); // 搜索下一个位置!
            g_a[place] = 0; // 回溯回来的复位: recover
            unmark(i); // 复原之前的标记: g_mark[i] = 0;
        }
    }
}
```

// 检查是否位置`place`，放置数字`i`是可行的。

// 返回值: `true` 可行; `false` 不可行

```
bool ok(int place, int i)
```

```
{
```

```
    if (g_mark[i] == 1)    //数字i已经被占用
        return false;
```

```
    if (isprime(g_a[place-1] + i)) //数字i与前位构成素数环
    {
        return true;
```

```
    }
```

```
else
```

```
{
```

```
    return false;
```

```
}
```

```
}
```

// 对第K个位置置值(使用交换的方法获取扩展)

```
void set(int place) //place位置范围: 1~g_n
{
    // 如果搜索到终点, 则打印路径, 结束本条路径的搜索。
    if (place > g_n)
    {
        if (isprime(g_a[1]+g_a[g_n])) //如果首尾可以成环
        {
            print_it();
            g_solution++;
        }
        return;
    }

    for (int i=place; i<g_n+1; i++) //遍历所有的可能扩展
    {
        swap_it(place, i); //由交换获得新扩展!
        if (isprime(g_a[place]+g_a[place-1])) //如获得的值: 与上位数的和是素数
        {
            set(place+1);
        }
        swap_it(place, i); // 恢复原排列
    }
}
```

总结

□ 回溯法的基本思想

□ 回溯法的算法总结

□ Depth First Search (深度优先搜索)

□ 从起点开始尽可能深入地扩展

- 每次试图访问一个之前没有访问过的点

- 有回退：如果此路不通，回到上一层节点

- 。 。 。

□ 注意点：

- **节点判重**

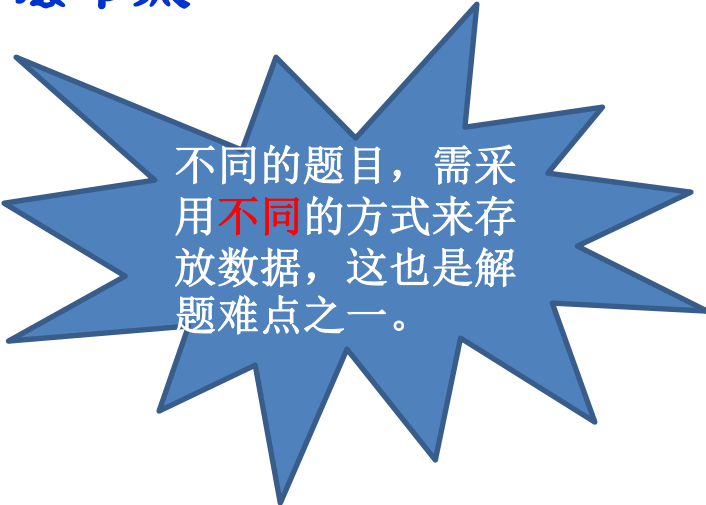
- **路径记录**

- **回溯回来记得清理数据**

□ 堆栈：

- 通常我们使用递归 – 系统内部堆栈

- 递归层数很深的时候，系统堆栈不够用，需要自己实现堆栈



不同的题目，需采用**不同**的方式来存放数据，这也是解题难点之一。

第二回 作业

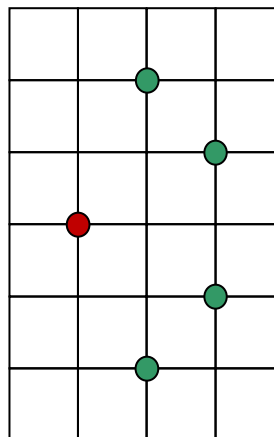
- (1) 调试迷宫问题
 - 设置不同的迷宫
 - 设置不同的起始位置
 - 如何找到从起点到终点的所有不同的路径？
 - 如何改善代码使得能更快地找到最优路径？
- (2) 完成并调试比较素数环的两种实现方法。
- (3) 完成骑士遍历问题
 - 在 $n*n$ 的方格的棋盘上，从任意指定的方格出发，为象棋中的马寻找一条走遍棋盘每一格并只经过一次的路径方法。
 - 如果求任意两点的最短路径？程序应如何完成？
- 总结回溯算法的算法基本框架（包括子集树的一般模式）
- 2~3人一组，完成实验报告，格式不限，但要说明问题
- 提交方式，电子文档，注意文档size不要超过1M

提示:

马每一步可以有8种走法!

图示为其中的4种。

可以越子, 没有"中国象棋"的"蹩马腿"限制。



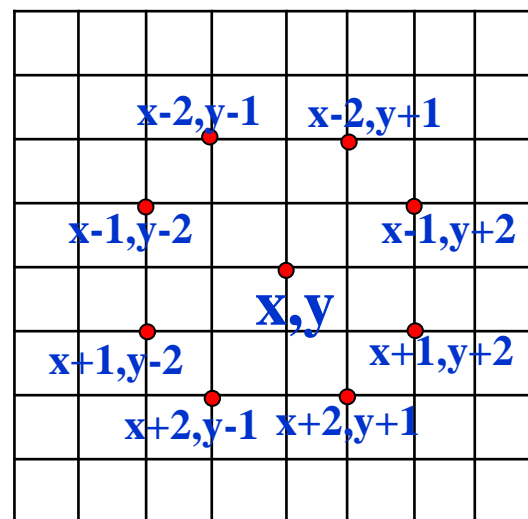
- 对于棋盘上任意一点 $A(x,y)$, 有八个扩展方向:

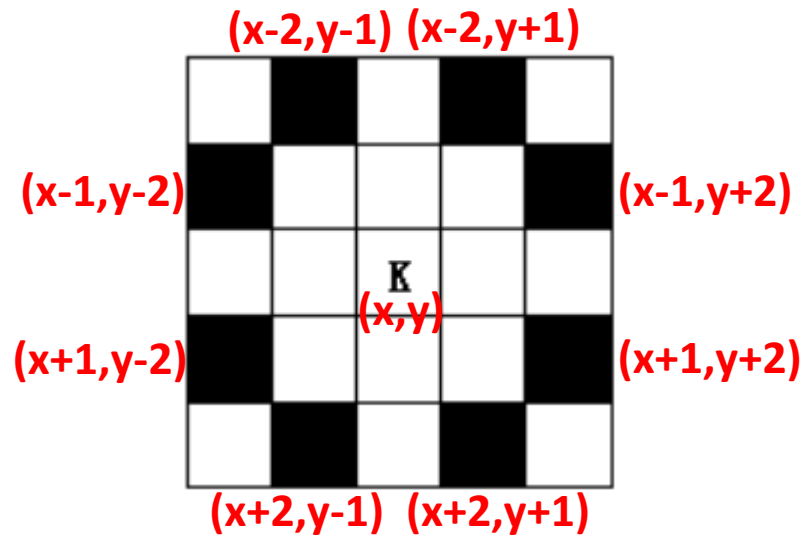
$A(x+1,y+2), A(x+2,y+1)$

$A(x+2,y-1), A(x+1,y-2)$

$A(x-1,y-2), A(x-2,y-1)$

$A(x-2,y+1), A(x-1,y+2)$





END!