**回溯**

1、概念

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

许多复杂的，规模较大的问题都可以使用回溯法，因此回溯法有“通用解题方法”的美称。

2、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，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。

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

3、用回溯法解题的一般步骤：

（1）针对所给问题，确定问题的解空间：

首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。

（2）确定结点的扩展搜索规则

（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。

4、算法框架

（1）问题框架

设问题的解是一个n维向量(a1,a2,………,an),约束条件是ai(i=1,2,3,…..,n)之间满足某种条件，记为f(ai)。

（2）非递归回溯框架

1: int a[n],i;

2: 初始化数组a[];

3: i = 1;

4: while (i>0(有路可走) and (未达到目标)) // 还未回溯到头

5: {

6: if(i > n) // 搜索到叶结点

7: {

8: 搜索到一个解，输出；

9: }

10: else // 处理第i个元素

11: {

12: a[i]第一个可能的值；

13: while(a[i]在不满足约束条件且在搜索空间内)

14: {

15: a[i]下一个可能的值；

16: }

17: if(a[i]在搜索空间内)

18: {

19: 标识占用的资源；

20: i = i+1; // 扩展下一个结点

21: }

22: else

23: {

24: 清理所占的状态空间； // 回溯

25: i = i –1;

26: }

27: }

（3）递归的算法框架

回溯法是对解空间的深度优先搜索，在一般情况下使用递归函数来实现回溯法比较简单，其中i为搜索的深度，框架如下：

1: int a[n];

2: try(int i)

3: {

4: if(i>n)

5: 输出结果;

6: else

7: {

8: for(j = 下界; j <= 上界; j=j+1) // 枚举i所有可能的路径

9: {

10: if(fun(j)) // 满足限界函数和约束条件

11: {

12: a[i] = j;

13: ... // 其他操作

5、深度优先搜索

深度优先搜索算法（Depth-First-Search），简称BFS，是搜索算法的一种。是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所有边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。

因发明“深度优先搜索算法”，霍普克洛夫特与陶尔扬共同获得计算机领域的最高奖：图灵奖。

**深度优先搜索**

1、概念

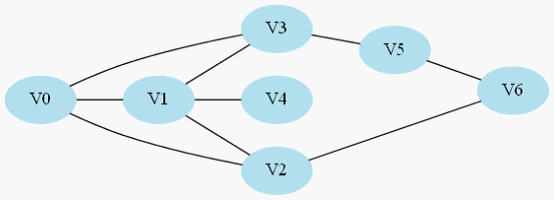
深度优先搜索算法（Depth-First-Search），是搜索算法的一种。是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所有边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。

因发明“深度优先搜索算法”，霍普克洛夫特与陶尔扬共同获得计算机领域的最高奖：图灵奖.

2、搜索过程图解

题目要求，只需要让你找出一条V0到V6的道路，无需最短路。



假设按照以下的顺序来搜索：

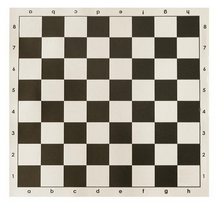
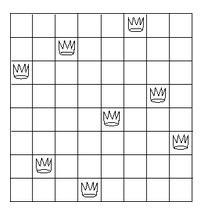
1.V0->V1->V4，此时到底尽头，仍然到不了V6，于是原路返回到V1去搜索其他路径；

2.返回到V1后既搜索V2，于是搜索路径是V0->V1->V2->V6,，找到目标节点，返回有解。这样搜索只是2步就到达了，但是如果用BFS的话就需要多几步。

例题1 八皇后问题

八皇后问题是一个古老而著名的问题，是回溯算法的典型例题。该问题是十九世纪著名的数学家高斯1850年提出：在8X8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、同一斜线及反斜线上，问有多少种摆法。

高斯认为有76种方案。1854年在柏林的象棋杂志上不同的作者发表了40种不同的解，后来有人用图论的方法解出92种结果。

**public** **class** QueenN {

**public** **static** **int** *num* = 0; // 累计方案总数

**public** **static** **final** **int** *MAXQUEEN* = 8;// 皇后个数，同时也是棋盘行列总数

**public** **static** **int**[] *cols* = **new** **int**[*MAXQUEEN*]; // 定义cols数组，表示8列棋子摆放情况

**public** QueenN() {

// 核心函数

getArrangement(0);

System.*out*.println();

System.*out*.println(*MAXQUEEN* + "皇后问题有" + *num* + "种摆放方法。");

}

**public** **void** getArrangement(**int** n) {

// 遍历该列所有不合法的行，并用rows数组记录，不合法即rows[i]=true

**boolean**[] rows = **new** **boolean**[*MAXQUEEN*];

**for** (**int** i = 0; i < n; i++) {

//上方不合法

rows[*cols*[i]] = **true**;

**int** d = n - i;

//斜上不合法

**if** (*cols*[i] - d >= 0)

rows[*cols*[i] - d] = **true**;

//反斜上不合法

**if** (*cols*[i] + d <= *MAXQUEEN* - 1)

rows[*cols*[i] + d] = **true**;

}

**for** (**int** i = 0; i < *MAXQUEEN*; i++) {

// 判断该行是否合法

**if** (rows[i])

**continue**;

// 设置当前列合法棋子所在行数

*cols*[n] = i;

// 当前列不为最后一列时

**if** (n < *MAXQUEEN* - 1) {

getArrangement(n + 1);

} **else** {

// 累计方案个数

*num*++;

// 打印棋盘信息

printChessBoard();

}

}

}

**public** **void** printChessBoard() {

System.*out*.println("第" + *num* + "种走法 ");

**for** (**int** i = 0; i < *MAXQUEEN*; i++) {

**for** (**int** j = 0; j < *MAXQUEEN*; j++) {

**if** (i == *cols*[j]) {

System.*out*.print("0 ");

} **else**

System.*out*.print("+ ");

}

System.*out*.println();

}

}

**public** **static** **void** main(String args[]) {

QueenN queen = **new** QueenN();

}

}

例题2 迷宫问题

给定一个迷宫，入口为左上角，出口为右下角，问是否有路径从入口到出口，若有则输出一条这样的路径。注意移动可以从上、下、左、右、上左、上右、下左、下右八个方向进行。迷宫输入0表示可走，输入1表示墙。易得可以用1将迷宫围起来避免边界问题。

本题采用DFS算法给出解。

0 1 1 1 0 1 1 1

1 0 1 0 1 0 1 0

0 1 0 0 1 1 1 1

0 1 1 1 0 0 1 1

1 0 0 1 1 0 0 0

0 1 1 0 0 1 1 0