

# 法律声明

---

□ 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：小象

■ 新浪微博：ChinaHadoop



# 图(下)

---

## ——实践问题



小象学院  
ChinaHadoop.cn

邹博

# 主要内容

---

- 搜索手段：
  - 动态规划
  - 广度优先搜索
  - 深度优先搜索
- 单词变换问题/周围区域问题
- 八皇后问题/数独问题/分割词汇问题
- 括号匹配的字符串
- 马踏棋盘
- 蚁群算法

# 广度优先搜索：Breadth First Search, BFS

---

## □ 最简单、直接的图搜索算法

### ■ 从起点开始层层扩展

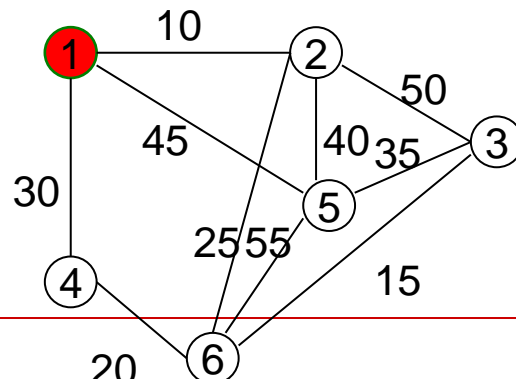
□ 第一层是离起点距离为1的

□ 第二层是离起点距离为2的

□ ....

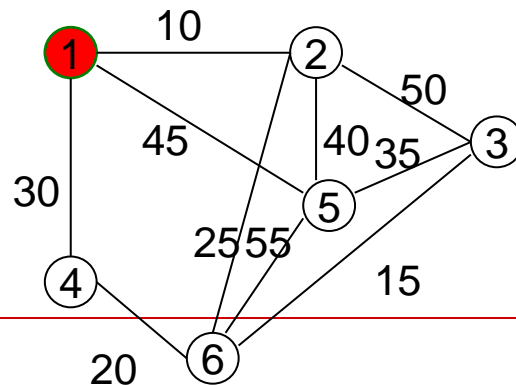
### ■ 本质就是按层(距离)扩展，无回退

# BFS分析



- 给定某起点a，将a放入缓冲区，开始搜索；
- 过程：假定某时刻缓冲区内结点为abc，则访问结点a的邻接点 $a_1a_2...a_x$ ，同时，缓冲区变成bc  $a_1a_2...a_x$ ，为下一次访问做准备；
- 辅助数据结构：队列
- 先进先出
- 从队尾入队，从队首出队
- 只有队首元素可见

# BFS分析的两个要点



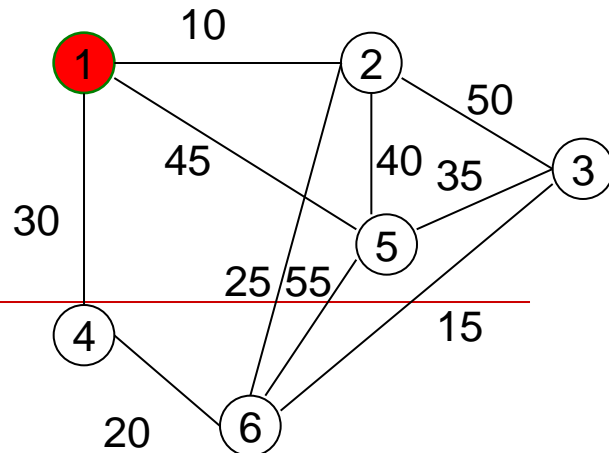
## □ 结点判重

- 如果在扩展中发现某结点在前期已经访问过，则本次不再访问该结点；显然，第一次访问到该结点时，是访问次数最少的：最少、最短；

## □ 路径记录

- 一个结点可能扩展出多个结点：多后继 $a_1 a_2 \dots a_x$ ，
- 但是任意一个结点最多只可能有1个前驱(起始结点没有前驱)：单前驱
- 用结点数目等长的数组 $pre[0 \dots N-1]$ ：
- $pre[i]=j$ ：第 $i$ 个结点的前一个结点是 $j$
- 注：再次用到“存索引，不存数据本身”的思路。

# BFS算法框架



## □ 辅助数据结构

- 队列q;
- 结点是第几次被访问到的 $d[0...N-1]$ : 简称**步数**;
- 结点的前驱 $pre[0...N-1]$ ;

## □ 算法描述:

## □ 起点start入队q

- 记录步数 $d[start]=0$ ;
- 记录start的前驱 $pre[start]=-1$ ;

## □ 如果队列q非空, 则**队首结点x**出队, 尝试扩展x

- 找到x的邻接点集合 $\{y|(x,y) \in E\}$ 
  - 对每个新扩展结点y**判重**, 如果y是新结点, 则入队q;
  - 同时, 记录步数 $d[y]=d[x]+1$ ; 前驱 $pre[y]=x$ ;

# BFS算法的思考

- 隐式图：实践中，往往乍看不是关于图的问题，但如果是给定一个起始状态和一些规则，求解决方案的问题：往往可以根据这些规则，将各个状态(动态的)建立连接边，然后使用BFS/DFS框架，一步一步的在解空间中搜索。
  - 树的层序遍历，是按照从根到结点的距离遍历，可以看做是图BFS过程。
  - 树的先序/后序/中序遍历，是从根搜索到树的叶子结点，然后回溯，可以看做是图DFS过程。
- 对BFS的改进——双向BFS
  - 从起点和终点分别走，直到相遇；
  - 将树形搜索结构变成纺锤形；
  - 经典BFS中，树形搜索结构若树的高度非常高时，叶子非常多(树的宽度大)，而把一棵高度为 $h$ 的树，用两个近似为 $h/2$ 的树代替，宽度相对较小。



# 单词变换问题 Word ladder

□ 给定字典和一个起点单词、一个终点单词，每次只能变换一个字母，问从起点单词是否可以到达终点单词？最短多少步？

□ 如：

■ start= "hit"

■ end = "cog"

■ dict = ["hot", "dot", "dog", "lot", "log"]

■ "hit" -> "hot" -> "dot" -> "dog" -> "cog"

# Word Ladder问题分析

```
start= "hit"  
end = "cog"  
dict = ["hot","dot","dog","lot","log"]  
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
```

## □ 使用临界表，建立单词间的联系

- 图的结点为单词，若两个单词只有1个字母不同，则两单词间存在无向边；

## □ 建图：

- 预处理：对字典中的所有单词建立map、hash或Trie结构，利于后续查找
- 对于某单词w，单词中的第i位记为 $\beta$ ，则将 $\beta$ 替换为 $[\beta+1, 'Z']$ ，查找新串nw是否在字典中。如果在，将(w-nw)添加到邻接表项w和nw中(无向边)
- 循环处理第二步

# 思考

---

- 若使用map，查找串的时间复杂度是 $O(\log N)$ ，总体的时间复杂度为 $O(N * \text{len} * 13 * \log N)$ ，即 $O(N * \log N)$ 。  
若使用hash或Trie，复杂度为 $O(N)$
- 从起始单词开始，广度优先搜索，计算能否到达终点单词。若可达，则这条路径上的变化是最快的。
  - 注意：虽然从起点单词开始到终点单词的路径内的单词，必须在词典内，但起点和终点本身是无要求的。
- 是否需要事先计算图本身？
- 体会路径记录问题。

# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("hot");
    dict.insert("dot");
    dict.insert("dog");
    dict.insert("lot");
    dict.insert("log");
    string start = "hit";
    string end = "cog";
    cout << CalcLadderLength(start, end, dict);
    return 0;
}
```

```
//搜索cur的孩子结点，存入children中
void Extend(const string& cur, vector<string>& children, const set<string>& dict,
            const string& end, set<string>& visit)
{
    string child = cur;
    children.clear();
    int i;
    char c,t;
    for(i = 0; i < (int)cur.size(); i++)
    {
        t = child[i];
        for(c = 'a'; c != 'z'; c++)
        {
            if(c == t)
                continue;
            child[i] = c;
            if(((child == end) || (dict.find(child) != dict.end()))
                && (visit.find(child) == visit.end()))
            {
                children.push_back(child);
                visit.insert(child);
            }
        }
        child[i] = t;
    }
}

int CalcLadderLength(const string& start, const string& end, const set<string>& dict)
{
    queue<string> q;
    q.push(start);
    vector<string> children; //从当前结点可扩展得到的新结点集合
    set<string> visit;
    int step = 0;
    string cur;
    int curNumber = 1; //当前层剩余节点数目
    int nextNumber = 0; //下一层孩子节点数目
    while(!q.empty())
    {
        cur = q.front(); //从cur开始扩展
        q.pop();
        curNumber--;
        Extend(cur, children, dict, end, visit);
        nextNumber += (int)children.size();
        if(curNumber == 0) //当前层遍历完，则遍历下一层，所以step加1
        {
            step++;
            curNumber = nextNumber;
            nextNumber = 0;
        }
        for(vector<string>::const_iterator it = children.begin(); it != children.end(); it++)
        {
            if(*it == end)
                return step;
            q.push(*it);
        }
    }
    return 0;
}
```

# Code

```
int CalcLadderLength(const string& start, const string& end, const set<string>& dict)
{
    queue<string> q;
    q.push(start);
    vector<string> children;    //从当前结点可扩展得到的新结点集合
    set<string> visit;
    int step = 0;
    string cur;
    int curNumber = 1;        //当前层剩余节点数目
    int nextNumber = 0;       //下一层孩子结点数目
    while(!q.empty())
    {
        cur = q.front();    //从cur开始扩展
        q.pop();
        curNumber--;
        Extend(cur, children, dict, end, visit);
        nextNumber += (int)children.size();
        if(curNumber == 0)    //当前层遍历完，则遍历下一层，所以step加1
        {
            step++;
            curNumber = nextNumber;
            nextNumber = 0;
        }
        for(vector<string>::const_iterator it = children.begin(); it != children.end(); it++)
        {
            if(*it == end)
                return step;
            q.push(*it);
        }
    }
    return 0;
}
```

# Code aux

//搜索cur的孩子结点，存入children中

```
void Extend(const string& cur, vector<string>& children, const set<string>& dict,  
           const string& end, set<string>& visit)
```

```
{  
    string child = cur;  
    children.clear();  
    int i;  
    char c,t;  
    for(i = 0; i < (int)cur.size(); i++)  
    {  
        t = child[i];  
        for(c = 'a'; c != 'z'; c++)  
        {  
            if(c == t)  
                continue;  
            child[i] = c;  
            if(((child == end) || (dict.find(child) != dict.end()))  
                && (visit.find(child) == visit.end()))  
            {  
                children.push_back(child);  
                visit.insert(child);  
            }  
        }  
        child[i] = t;  
    }  
}
```

# 周围区域问题

□ 给定 $M \times N$ 的二维平面，格点处要么是‘X’，要么是‘O’。将完全由‘X’围成的区域中的‘O’替换成‘X’。

■ 假定数据是4连通

|   |   |   |   |
|---|---|---|---|
| X | X | X | X |
| X | O | O | X |
| X | X | O | X |
| X | O | X | X |



|   |   |   |   |
|---|---|---|---|
| X | X | X | X |
| X | X | X | X |
| X | X | X | X |
| X | O | X | X |

# Surrounded Regions分析

|   |   |   |   |
|---|---|---|---|
| X | X | X | X |
| X | 0 | 0 | X |
| X | X | 0 | X |
| X | 0 | X | X |



|   |   |   |   |
|---|---|---|---|
| X | X | X | X |
| X | X | X | X |
| X | X | X | X |
| X | 0 | X | X |

□ 反向思索最简单：哪些 ‘O’ 是应该保留的？

■ 从上下左右四个边界往里走，凡是能碰到的 ‘O’，都是跟边界接壤的，应该保留。

□ 思路：

■ 对于每一个边界上的 ‘O’ 作为起点，做若干次广度优先搜索，对于碰到的 ‘O’，标记为其他某字符 Y；

■ 最后遍历一遍整个地图，把所有的 Y 恢复成 ‘O’，把所有现有的 ‘O’ 都改成 ‘X’。



# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int M = 10;
    const int N = 10;
    //随机生成数据
    vector<vector<int> > land(M, vector<int>(N));
    int i, j;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)
        {
            land[i][j] = ((rand() % 3) == 0) ? WATER : SOIL;
        }
    }

    Print(land, M, N);
    FillLake(land, M, N); //围湖造田
    Print(land, M, N);
    return 0;
}
```

```
bool IsOcean(vector<vector<int> >& land, int M, int N, int i, int j)
{
    if((i < 0) || (i >= M) || (j < 0) || (j >= N))
        return false;
    return land[i][j] == WATER;
}

void Ocean(vector<vector<int> >& land, int M, int N, int i, int j)
{
    queue<pair<int, int> > q;
    q.push(make_pair(i, j));
    int iDirect[] = {-1, 1, 0, 0};
    int jDirect[] = {0, 0, -1, 1};
    int iCur, jCur;
    int k;
    while(!q.empty())
    {
        i = q.front().first;
        j = q.front().second;
        q.pop();
        for(k = 0; k < 4; k++)
        {
            iCur = i + iDirect[k];
            jCur = j + jDirect[k];
            if(IsOcean(land, M, N, iCur, jCur))
            {
                q.push(make_pair(iCur, jCur));
                land[iCur][jCur] = OCEAN;
            }
        }
    }
}

void FillLake(vector<vector<int> >& land, int M, int N)
{
    int i, j;
    //从边缘开始, 获得海洋区域
    for(i = 0; i < M; i++)
    {
        if(land[i][0] == WATER)
            Ocean(land, M, N, i, 0);
        if(land[i][N-1] == WATER)
            Ocean(land, M, N, i, N-1);
    }
    for(j = 1; j < N-1; j++)
    {
        if(land[0][j] == WATER)
            Ocean(land, M, N, 0, j);
        if(land[M-1][j] == WATER)
            Ocean(land, M, N, M-1, j);
    }

    //恢复海洋, 填湖
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)
        {
            if(land[i][j] == OCEAN) //是海洋, 恢复成水
                land[i][j] = WATER;
            else if(land[i][j] == WATER) //是湖泊
                land[i][j] = SOIL; //填湖
        }
    }
}
```

# Code split

```
bool IsOcean(vector<vector<int>> & land, int M, int N, int i, int j)
{
    if((i < 0) || (i >= M) || (j < 0) || (j >= N))
        return false;
    return land[i][j] == WATER;
}
```

```
void Ocean(vector<vector<int>> & land, int M, int N, int i, int j)
{
    queue<pair<int, int>> q;
    q.push(make_pair(i, j));
    int iDirect[] = {-1, 1, 0, 0};
    int jDirect[] = {0, 0, -1, 1};
    int iCur, jCur;
    int k;
    while(!q.empty())
    {
        i = q.front().first;
        j = q.front().second;
        q.pop();
        for(k = 0; k < 4; k++)
        {
            iCur = i + iDirect[k];
            jCur = j + jDirect[k];
            if(IsOcean(land, M, N, iCur, jCur))
            {
                q.push(make_pair(iCur, jCur));
                land[iCur][jCur] = OCEAN;
            }
        }
    }
}
```

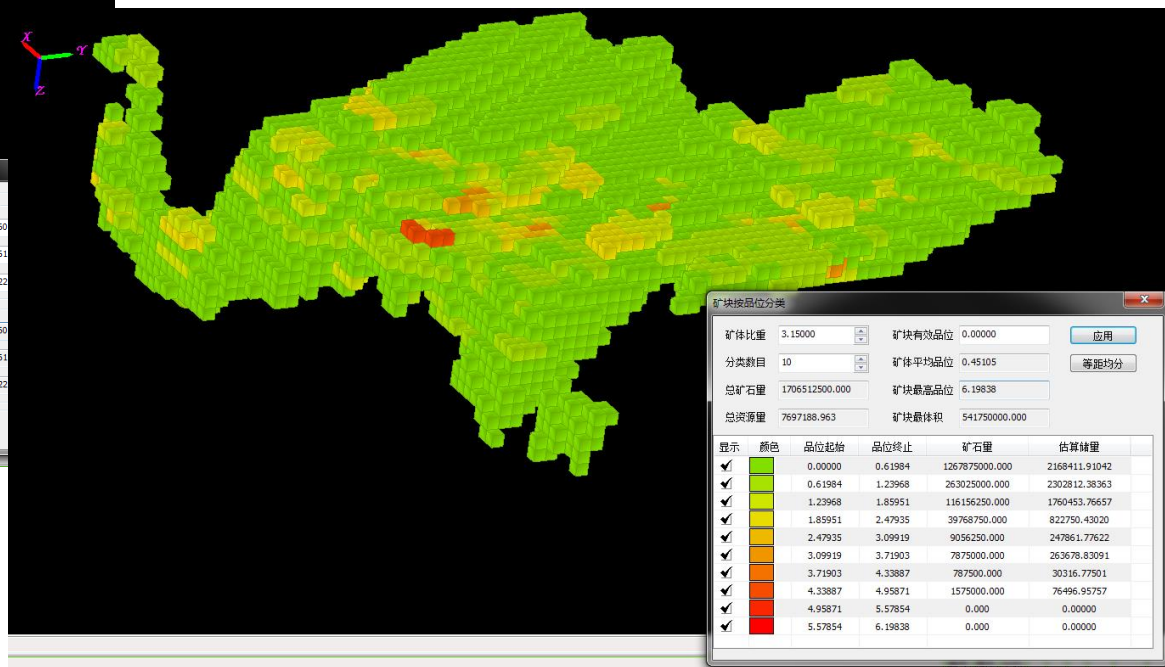
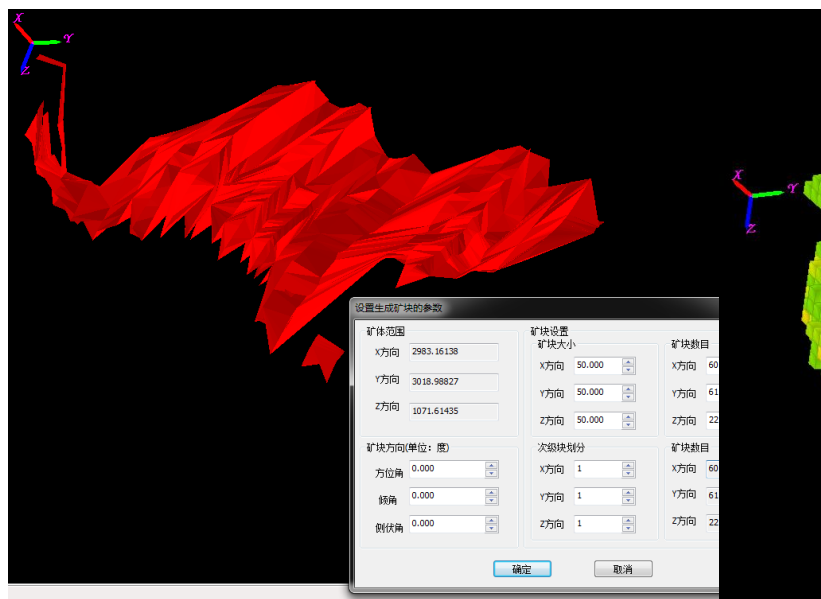
```
void FillLake(vector<vector<int>> & land, int M, int N)
{
    int i, j;

    //从边缘开始, 获得海洋区域
    for(i = 0; i < M; i++)
    {
        if(land[i][0] == WATER)
            Ocean(land, M, N, i, 0);
        if(land[i][N-1] == WATER)
            Ocean(land, M, N, i, N-1);
    }
    for(j = 1; j < N-1; j++)
    {
        if(land[0][j] == WATER)
            Ocean(land, M, N, 0, j);
        if(land[M-1][j] == WATER)
            Ocean(land, M, N, M-1, j);
    }

    //恢复海洋, 填湖
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)
        {
            if(land[i][j] == OCEAN) //是海洋, 恢复成水
                land[i][j] = WATER;
            else if(land[i][j] == WATER) //是湖泊
                land[i][j] = SOIL; //填湖
        }
    }
}
```

# 思考与拓展

□ 如果目标区域是三维的呢？



# 深度优先搜索DFS

---

## ☐ 理念:

- 不断深入，“走到头”回退。(回溯思想)

## ☐ 一般所谓“暴力枚举”搜索都是指DFS

- 回忆字符串章节中“全排列问题”的递归解法

## ☐ 实现

- 一般使用堆栈，或者递归

## ☐ 用途:

- DFS的过程中，能够获得的信息

- ☐ “时间戳”、“颜色”、父子关系、高度

# 所有括号匹配的字符串

---

- N对括号能够得到的有效括号序列有哪些？
- 如N=3时，有效括号串共5个，分别为：
  - 1: ()()()
  - 2: ()(())
  - 3: (())()
  - 4: (()())
  - 5: (((())))

# 问题分析

- 任何一个括号序列，都可以写成形式  $A(B)$ 
  - $A$ 、 $B$  都是若干括号对形成的合法串(可为空串)
  - 若  $N=0$ ，括号序列为空。
  - 若  $N=1$ ，括号序列只能是  $()$  这一种。
- 算法描述：  $i \in [0, N-1]$ 
  - 计算  $i$  对括号的可行序列  $A$ ；
  - 计算  $N-i-1$  对括号的可行序列  $B$ ；
  - 组合得到  $A(B)$ 。
  - 注：加上额外一对括号  $()$ ，总括号共  $N$  对

# Code

|              |                |
|--------------|----------------|
| 1: ()()()()  | 22: (((()))()) |
| 2: ()()()()  | 23: (((()))()) |
| 3: ()()()()  | 24: ((()())()) |
| 4: ()()()()  | 25: ((()())()) |
| 5: ()()()()  | 26: ((()())()) |
| 6: ()()()()  | 27: ((()())()) |
| 7: ()()()()  | 28: ((()())()) |
| 8: ()()()()  | 29: ((()())()) |
| 9: ()()()()  | 30: ((()())()) |
| 10: ()()()() | 31: ((()())()) |
| 11: ()()()() | 32: ((()())()) |
| 12: ()()()() | 33: ((()())()) |
| 13: ()()()() | 34: ((()())()) |
| 14: ()()()() | 35: ((()())()) |
| 15: ()()()() | 36: ((()())()) |
| 16: ()()()() | 37: ((()())()) |
| 17: ()()()() | 38: ((()())()) |
| 18: ()()()() | 39: ((()())()) |
| 19: ()()()() | 40: ((()())()) |
| 20: ()()()() | 41: ((()())()) |
| 21: ()()()() | 42: (((()))()) |

```

void Unit(vector<string>& result,
const vector<string>& prefix, const vector<string>& suffix)
{
    vector<string>::const_iterator ip, is;
    for(ip = prefix.begin(); ip != prefix.end(); ip++)
    {
        for(is = suffix.begin(); is != suffix.end(); is++)
        {
            result.push_back("");
            string& r = result.back();
            r += "(";
            r += *ip;
            r += ")";
            r += *is;
        }
    }
}

vector<string> AllParentheses(int n)
{
    if(n == 0)
        return vector<string>(1, "");
    if(n == 1)
        return vector<string>(1, "()");
    vector<string> prefix, suffix, result;
    for(int i = 0; i < n; i++)
    {
        prefix = AllParentheses(i);
        suffix = AllParentheses(n-i-1);
        Unit(result, prefix, suffix);
    }
    return result;
}

```

# 思考

- 可以通过增加缓存的方式，对已经计算得到的字符串直接获取，以空间换时间，降低时间复杂度。
- 如果只是计算可行括号串的数目，如何计算？
  - 事实上，数组 $A[i]$ 表示长度为 $i$ 的括号串的可行数目，即著名的Catalan数。
  - 该问题在动态规划中继续讨论。
- Catalan数(从0开始数):
  - 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452.....



# Code

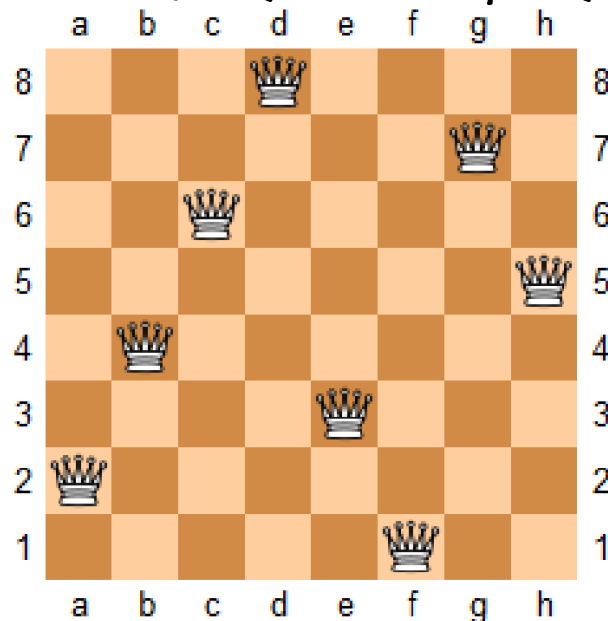
0: 1  
1: 1  
2: 2  
3: 5  
4: 14  
5: 42  
6: 132  
7: 429  
8: 1430  
9: 4862  
10: 16796  
11: 58786  
12: 208012  
13: 742900  
14: 2674440  
15: 9694845  
16: 35357670  
17: 129644790  
18: 477638700  
19: 1767263190

```
void GetCatalan(int* pCatalan, int N)
{
    pCatalan[0] = 1;
    pCatalan[1] = 1;
    int i, j;
    int c;
    for (i = 2; i <= N; i++)
    {
        pCatalan[i] = 1;
        c = 0;
        for (j = 0; j < i; j++)
        {
            c += pCatalan[j] * pCatalan[i-j-1];
        }
        pCatalan[i] = c;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 19;
    int catalan[n+1];
    GetCatalan(catalan, n);
    PrintNumber(catalan, n);
    return 0;
}
```

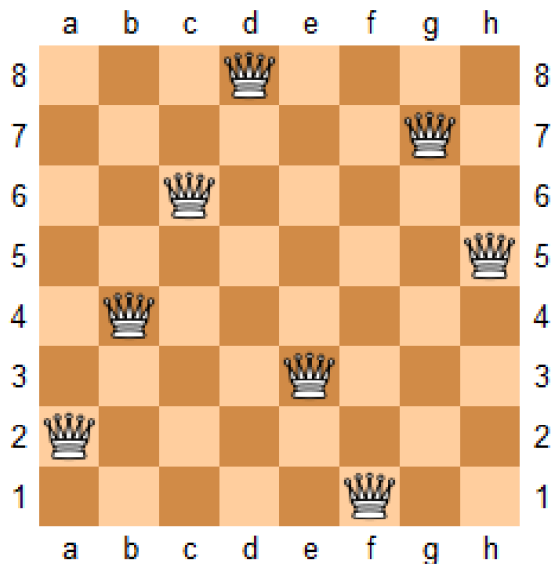
# 八皇后问题

- 在 $8 \times 8$ 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，问有多少种解法。

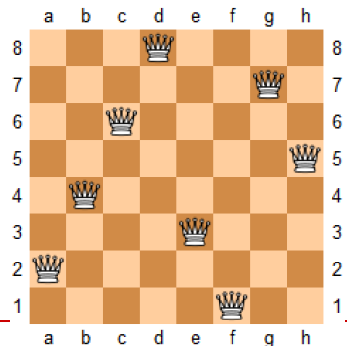


# 八皇后问题算法分析

- 分析：显然任意一行有且仅有1个皇后，使用数组 `queen[0...7]` 表示第*i*行的皇后位于哪一列。
- 对于“12345678”这个字符串，调用**全排列问题**的代码，并且加入分支限界的条件判断是否相互攻击即可；



# 八皇后问题算法分析



- 深度优先搜索：将第*i*个皇后放置在第*j*列上，如果当前位置与其他皇后相互攻击，则剪枝掉该结点。
- 分析对角线：
  - 主对角线上 $(i-j)$ 为定值，取值范围是 $-(N-1) \leq (i-j) \leq N-1$ ，从而： $0 \leq (i-j+N-1) \leq 2*N-2$ ；
  - 次对角线上 $(i+j)$ 为定值，取值范围是 $0 \leq (i+j) \leq 2*N-2$ ；
  - 使用 $m1[0...2N-2]$ 、 $m2[0...2N-2]$ 记录皇后占据的对角线
- 上述数据结构与剪枝过程适用于N皇后问题。

# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    CQueen queen(8);
    queen.Queen();
    queen.Print();
    return 0;
}
```

```
class CQueen
{
private:
    int m_nQueen;
    vector<bool> m_Column; //path已经占据的列
    vector<bool> m_MainDiagonal; //path已经占据的主对角线
    vector<bool> m_MinorDiagonal; //path已经占据的次对角线
    vector<vector<int>> m_Answer; //最终解

public:
    CQueen(int N) : m_nQueen(N)
    {
        m_Column.resize(N, false);
        m_MainDiagonal.resize(2*N-1, false);
        m_MinorDiagonal.resize(2*N-1, false);
    }

    void Queen()
    {
        int* path = new int[m_nQueen]; //一个可行解
        CalcNQueen(path, 0);
        delete[] path;
    }
}
```

```
class CQueen
{
private:
    int m_nQueen;
    vector<bool> m_Column; //path已经占据的列
    vector<bool> m_MainDiagonal; //path已经占据的主对角线
    vector<bool> m_MinorDiagonal; //path已经占据的次对角线
    vector<vector<int>> m_Answer; //最终解

public:
    CQueen(int N) : m_nQueen(N)
    {
        m_Column.resize(N, false);
        m_MainDiagonal.resize(2*N-1, false);
        m_MinorDiagonal.resize(2*N-1, false);
    }

    void Queen()
    {
        int* path = new int[m_nQueen]; //一个可行解
        CalcNQueen(path, 0);
        delete[] path;
    }

private:
    void CalcNQueen(int* path, int row)
    {
        if(row == m_nQueen)
        {
            m_Answer.push_back(vector<int>(path, path+m_nQueen));
            return;
        }
        for(int col = 0; col < m_nQueen; col++)
        {
            if(CanLay(row, col))
            {
                path[row] = col;
                m_Column[col] = true;
                m_MinorDiagonal[row+col] = true;
                m_MainDiagonal[m_nQueen-1+row-col] = true;
                CalcNQueen(path, row+1);
                //回溯
                m_Column[col] = false;
                m_MinorDiagonal[row+col] = false;
                m_MainDiagonal[m_nQueen-1+row-col] = false;
            }
        }
    }

    bool CanLay(int row, int col) const
    {
        return !m_Column[col] && !m_MinorDiagonal[row+col] && !m_MainDiagonal[m_nQueen-1+row-col];
    }

public:
    void Print() const
    {
        cout << "所有解的个数: " << (int)m_Answer.size() << "\n";
        for(vector<vector<int>>::const_iterator it = m_Answer.begin(); it != m_Answer.end(); it++)
        {
            PrintOne(*it);
        }
    }

    void PrintOne(const vector<int>& v) const
    {
        for(vector<int>::const_iterator it = v.begin(); it != v.end(); it++)
            cout << *it << '\t';
        cout << endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    CQueen queen(8);
    queen.Queen();
    queen.Print();
    return 0;
}
```

# Code

```
void CalcNQueen(int* path, int row)
{
    if(row == m_nQueen)
    {
        m_Answer.push_back(vector<int>(path, path+m_nQueen));
        return;
    }
    for(int col = 0; col < m_nQueen; col++)
    {
        if(CanLay(row, col))
        {
            path[row] = col;
            m_Colomn[col] = true;
            m_MinorDiagonal[row+col] = true;
            m_MainDiagonal[m_nQueen-1+row-col] = true;
            CalcNQueen(path, row+1);

            //回溯
            m_Colomn[col] = false;
            m_MinorDiagonal[row+col] = false;
            m_MainDiagonal[m_nQueen-1+row-col] = false;
        }
    }
}
```


```
bool CanLay(int row, int col) const
{
    return !m_Colomn[col] && !m_MinorDiagonal[row+col] && !m_MainDiagonal[m_nQueen-1+row-col];
}
```

# 数独Sudoku

□ 解数独问题，初始化时的空位用 ‘.’ 表示。

■ 每行、每列、每个九宫内，都是1-9这9个数字。

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 |   |   | 7 |   |   |   |   |
| 6 |   |   | 1 | 9 | 5 |   |   |   |
|   | 9 | 8 |   |   |   |   | 6 |   |
| 8 |   |   |   | 6 |   |   |   | 3 |
| 4 |   |   | 8 |   | 3 |   |   | 1 |
| 7 |   |   |   | 2 |   |   |   | 6 |
|   | 6 |   |   |   |   | 2 | 8 |   |
|   |   |   | 4 | 1 | 9 |   |   | 5 |
|   |   |   |   | 8 |   |   | 7 | 9 |



|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

# 数独Sudoku分析

- 若当前位置是空格，则尝试从1到9的所有数；  
如果对于1到9的某些数字，当前是合法的，  
则继续尝试下一个位置——调用自身即可。

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 2 | 0 | 6 | 3 | 0 | 0 | 9 |
| 6 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 5 |
| 3 | 0 | 0 | 0 | 2 | 0 | 4 | 8 | 0 |
| 1 | 0 | 0 | 5 | 0 | 2 | 6 | 0 | 8 |
| 4 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 1 |
| 9 | 0 | 5 | 6 | 0 | 0 | 0 | 0 | 7 |
| 0 | 3 | 6 | 0 | 5 | 0 | 0 | 0 | 2 |
| 2 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 4 |
| 7 | 0 | 0 | 2 | 9 | 0 | 8 | 5 | 0 |



|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 2 | 8 | 6 | 3 | 7 | 1 | 9 |
| 6 | 8 | 7 | 4 | 1 | 9 | 2 | 3 | 5 |
| 3 | 9 | 1 | 7 | 2 | 5 | 4 | 8 | 6 |
| 1 | 7 | 3 | 5 | 4 | 2 | 6 | 9 | 8 |
| 4 | 6 | 8 | 9 | 3 | 7 | 5 | 2 | 1 |
| 9 | 2 | 5 | 6 | 8 | 1 | 3 | 4 | 7 |
| 8 | 3 | 6 | 1 | 5 | 4 | 9 | 7 | 2 |
| 2 | 5 | 9 | 3 | 7 | 8 | 1 | 6 | 4 |
| 7 | 1 | 4 | 2 | 9 | 6 | 8 | 5 | 3 |



# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int chess[9][9] = {
        {0, 4, 2, 0, 6, 3, 0, 0, 9},
        {6, 0, 0, 0, 1, 0, 0, 0, 5},
        {3, 0, 0, 0, 2, 0, 4, 8, 0},
        {1, 0, 0, 5, 0, 2, 6, 0, 8},
        {4, 0, 0, 0, 0, 7, 0, 0, 1},
        {9, 0, 5, 6, 0, 0, 0, 0, 7},
        {0, 3, 6, 0, 5, 0, 0, 0, 2},
        {2, 0, 0, 0, 7, 0, 0, 0, 4},
        {7, 0, 0, 2, 9, 0, 8, 5, 0},
    };
    CSudoku sudoku(chess);
    sudoku.Print(true);
    sudoku.Sudoku();
    sudoku.Print(false);
    return 0;
}
```

```
class CSudoku
{
private:
    int m_chess[9][9];
    int m_result[9][9];
    bool m_bSolve;

public:
    CSudoku(int chess[9][9])
    {
        memcpy(m_chess, chess, sizeof(m_chess));
        m_bSolve = false;
    }

    bool IsValid(int i, int j)
    {
        int t = m_chess[i][j];
        int k;
        for(k = 0; k < 9; k++)
        {
            if((j != k) && (t == m_chess[i][k])) //列
                return false;
            if((i != k) && (t == m_chess[k][j])) //行
                return false;
        }
        int iGrid = (i / 3) * 3;
        int jGrid = (j / 3) * 3;
        int k1, k2;
        for(k1 = iGrid; k1 < iGrid+3; k1++)
        {
            for(k2 = jGrid; k2 < jGrid+3; k2++)
            {
                if((k2 == j) && (k1 == i))
                    continue;
                if(t == m_chess[k1][k2])
                    return false;
            }
        }
        return true;
    }

    bool Sudoku()
    {
        int i, j, k;
        for(i = 0; i < 9; i++)
        {
            for(j = 0; j < 9; j++)
            {
                if(m_chess[i][j] == 0)
                {
                    for(k = 1; k < 10; k++)
                    {
                        m_chess[i][j] = k;
                        if(IsValid(i, j) && Sudoku())
                        {
                            if(!m_bSolve)
                                memcpy(m_result, m_chess, sizeof(m_chess));
                            m_bSolve = true;
                            return true;
                        }
                        m_chess[i][j] = 0;
                    }
                    return false;
                }
            }
        }
        return true; //说明所有位置都有值了
    }
};
```

# Code

```
bool Sudoku()
{
    int i, j, k;
    for (i = 0; i < 9; i++)
    {
        for (j = 0; j < 9; j++)
        {
            if (m_chess[i][j] != 0)
                continue;
            for (k = 1; k < 10; k++)
            {
                m_chess[i][j] = k;
                if (IsValid(i, j) && Sudoku())
                {
                    if (!m_bSolve)
                        memcpy(m_result, m_chess, sizeof(m_chess));
                    m_bSolve = true;
                    return true;
                }
                m_chess[i][j] = 0;
            }
            return false;
        }
    }
    return true;    //说明所有位置都有值了
}
```

```
class CSudoku
{
private:
    int m_chess[9][9];
    int m_result[9][9];
    bool m_bSolve;

public:
    CSudoku(int chess[9][9])
    {
        memcpy(m_chess, chess, sizeof(m_chess));
        m_bSolve = false;
    }
}
```

```
bool IsValid(int i, int j)
{
    int t = m_chess[i][j];
    int k;
    for (k = 0; k < 9; k++)
    {
        if ((j != k) && (t == m_chess[i][k])) //列
            return false;
        if ((i != k) && (t == m_chess[k][j])) //行
            return false;
    }
    int iGrid = (i / 3) * 3;
    int jGrid = (j / 3) * 3;
    int k1, k2;
    for (k1 = iGrid; k1 < iGrid+3; k1++)
    {
        for (k2 = jGrid; k2 < jGrid+3; k2++)
        {
            if ((k2 == j) && (k1 == i))
                continue;
            if (t == m_chess[k1][k2])
                return false;
        }
    }
    return true;
}
```

# 非递归数独Sudoku

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 |   |   | 7 |   |   |   |   |
| 6 |   |   | 1 | 9 | 5 |   |   |   |
|   | 9 | 8 |   |   |   | 6 |   |   |
| 8 |   |   | 6 |   |   |   | 3 |   |
| 4 |   |   | 8 |   | 3 |   |   | 1 |
| 7 |   |   | 2 |   |   |   | 6 |   |
|   | 6 |   |   |   | 2 | 8 |   |   |
|   |   |   | 4 | 1 | 9 |   |   | 5 |
|   |   |   |   | 8 |   |   | 7 | 9 |



|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

Figure showing the implementation of a non-recursive Sudoku solver in C++.

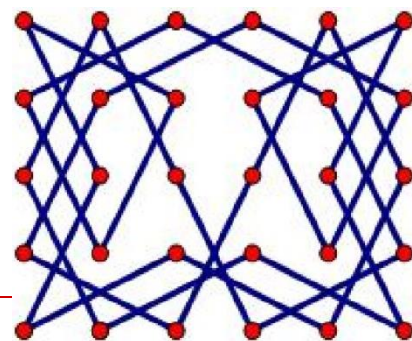
The main window displays the C++ code for `Sudoku.cpp`, which implements a backtracking algorithm using a state array and a chess array to track the current state and the current row's values.

```
i = 0;
int n;
while((i >= 0) && (i < 81))
{
    if(state[i] == 1)
    {
        if(i == 80) //找到一个解
        {
            result++;
            Print(chess, result);
            memcpy(solution, chess, 81*sizeof(char));
        }
        Roll(chess, state, i); //回溯
    }
    else
    {
        i++;
        if(state[i] != 1)
            chess[i] = 0;
    }
    else
    {
        n = GetN(chess, i);
        if(n == 0) //没有了, 回溯
        {
            Roll(chess, state, i);
        }
        else
        {
            chess[i] = n;
            if(i == 80) //找到一个解
            {
                result++;
                Print(chess, result);
                memcpy(solution, chess, 81*sizeof(char));
            }
            Roll(chess, state, i); //回溯
        }
    }
}
```

Two windows titled "数独Sudoku" show the initial and solved grids, respectively. The initial grid is a 9x9 grid with some numbers filled in, and the solved grid shows the complete solution.

The right side of the image shows the Visual Studio interface with the solution resource manager and the project files.

# 马踏棋盘




□ 给定 $m \times n$ 的棋盘，将棋子“马”放在任意位置上，按照走棋规则将“马”移动，要求每个方格只能进入一次，最终使得“马”走遍棋盘的所有位置。

■ 如给定 $8 \times 8$ 的国际象棋棋盘(右)

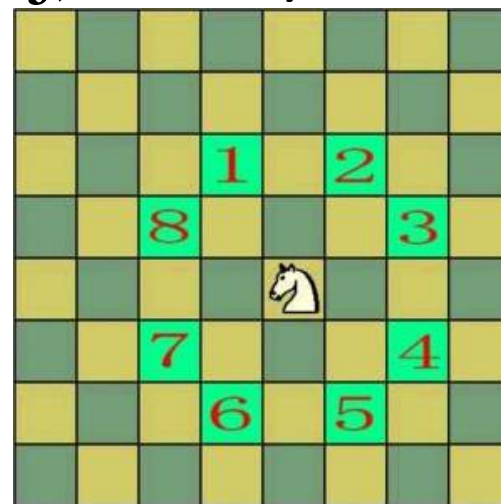
■ 如给定 $8 \times 9$ 的中国象棋棋盘(左)

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 01 | 70 | 17 | 36 | 03 | 52 | 07 | 38 | 05 |
| 18 | 35 | 02 | 71 | 60 | 37 | 04 | 45 | 08 |
| 69 | 16 | 67 | 58 | 51 | 56 | 53 | 06 | 39 |
| 34 | 19 | 72 | 61 | 54 | 59 | 46 | 09 | 44 |
| 15 | 68 | 33 | 66 | 57 | 50 | 55 | 40 | 29 |
| 20 | 65 | 22 | 49 | 62 | 47 | 28 | 43 | 10 |
| 23 | 14 | 63 | 32 | 25 | 12 | 41 | 30 | 27 |
| 64 | 21 | 24 | 13 | 48 | 31 | 26 | 11 | 42 |

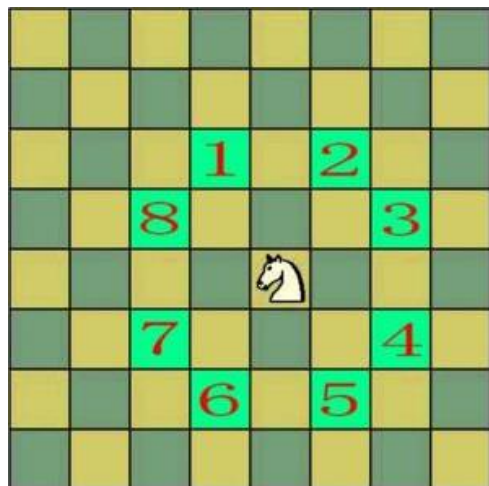
|   |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|
|  | 23 | 48 | 35 | 10 | 21 | 56 | 33 |
| 49  | 36 | 11 | 22 | 55 | 34 | 9  | 20 |
| 12  | 1  | 24 | 47 | 40 | 57 | 32 | 63 |
| 25  | 50 | 37 | 54 | 43 | 46 | 19 | 8  |
| 2   | 13 | 44 | 41 | 58 | 39 | 62 | 31 |
| 51  | 26 | 53 | 38 | 45 | 42 | 7  | 18 |
| 14  | 3  | 28 | 59 | 16 | 5  | 30 | 61 |
| 27  | 52 | 15 | 4  | 29 | 60 | 17 | 6  |

# 问题分析

- 显然，如果从A点能够跳到B点，则从B点也能够跳到A点。所以，马的起始位置可以从任意一点开始，不妨从左上角开始。
- 若当前位置为 $(i,j)$ ，则遍历 $(i,j)$ 的八邻域，如果邻域尚未经过，则跳转。
  - 深度优先搜索



# Code



```
int iHorse[] = {-2, -2, -1, +1, +2, +2, +1, -1};
int jHorse[] = {-1, +1, +2, +2, +1, -1, -2, -2};
int m = 8;
int n = 9;
```

```
bool CanJump(const vector<vector<int> >& chess, int i, int j)
{
    if((i < 0) || (i >= m) || (j < 0) || (j >= n))
        return false;
    return (chess[i][j] == 0);
}

bool Jump(vector<vector<int> >& chess, int i, int j, int step)
{
    if(step == m*n) //遍历结束
        return true;
    int iCur, jCur;
    for(int k = 0; k < 8; k++)
    {
        iCur = i + iHorse[k];
        jCur = j + jHorse[k];
        if(CanJump(chess, iCur, jCur))
        {
            chess[iCur][jCur] = step+1;
            if(Jump(chess, iCur, jCur, step+1))
                return true;
            chess[iCur][jCur] = 0;
        }
    }
    return false;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    vector<vector<int> > chess(m, vector<int>(n));
    chess[0][0] = 1;
    Jump(chess, 0, 0, 1);
    Print(chess);
    return 0;
}
```

# 启发式搜索

- 若棋盘规模较大，则在较深的棋位才能发现“无路可走”而不得不回溯。
- 贪心的启发式策略：
  - 最多情况下，每个棋位有8个后继。由于棋盘边界和已经遍历的原因，往往是少于8个的。
- 当前棋位可以跳转的后继棋位记为 $x$ 个，这 $x$ 个棋位的后继棋位数记做 $h_1h_2...h_x$ ，优先选择最小的 $h_i$ 。
  - 策略：优先选择孙结点数目最少的那个子结点
  - 原因：孙结点最少的子结点，如果当前不跳转则最容易在后期无法跳转。

# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    vector<vector<int>> chess(m, vector<int>(n));
    chess[0][0] = 1;
    Jump2(chess, 0, 0, 1);
    Print(chess);
    return 0;
}
```

```
typedef struct tagSHorse
{
    int nDirect;    //跳转方向
    int nValidStep; //有效步数

    bool operator < (const tagSHorse& horse)
    {
        return nValidStep < horse.nValidStep;
    }
} SHorse;
```

```
int GatherHorseDirect(SHorse* pHorse, int i, int j, const vector<vector<int>>& chess, bool bLast)
{
    int nHorse = 0;
    int iCur, jCur;
    for(int k = 0; k < 8; k++)
    {
        iCur = i + iHorse[k];
        jCur = j + jHorse[k];
        if(bLast) //最后一步
        {
            if(CanJump(chess, iCur, jCur))
            {
                pHorse[nHorse].nValidStep = 1;
                pHorse[nHorse].nDirect = k;
                nHorse++;
                break;
            }
        }
        else //正常情况
        {
            pHorse[nHorse].nValidStep = GetNextStep(chess, iCur, jCur);
            if(pHorse[nHorse].nValidStep != 0)
            {
                pHorse[nHorse].nDirect = k;
                nHorse++;
            }
        }
    }
    if(nHorse == 0)
        return 0;
    sort(pHorse, pHorse+nHorse);
    return nHorse;
}

bool Jump2(vector<vector<int>>& chess, int i, int j, int step)
{
    if(step == m*n) //遍历结束
        return true;
    SHorse pHorse[8];
    int nHorse = GatherHorseDirect(pHorse, i, j, chess, step == m*n-1);
    int iCur, jCur;
    int nDirect;
    step++;
    for(int k = 0; k < nHorse; k++)
    {
        nDirect = pHorse[k].nDirect;
        iCur = i + iHorse[nDirect];
        jCur = j + jHorse[nDirect];
        chess[iCur][jCur] = step;
        if(Jump2(chess, iCur, jCur, step))
            return true;
        chess[iCur][jCur] = 0;
    }
    return false;
}
```



# Code

```
bool Jump2(vector<vector<int> >& chess, int i, int j, int step)
{
    if(step == m*n) //遍历结束
    {
        AddSolution(chess);
        return true;
    }
    SHorse pHorse[8];
    int nHorse = GatherHorseDirect(pHorse, i, j, chess, step == m*n-1);
    int iCur, jCur;
    int nDirect;
    step++;
    for(int k = 0; k < nHorse; k++)
    {
        nDirect = pHorse[k].nDirect;
        iCur = i + iHorse[nDirect];
        jCur = j + jHorse[nDirect];
        chess[iCur][jCur] = step;
        if(Jump2(chess, iCur, jCur, step)) //找到一个解
        {
            //return true; //删去本行, 则算法计算所有解
        }
        chess[iCur][jCur] = 0;
    }
    return false;
}
```

# Code

```
int GatherHorseDirect(SHorse* pHorse, int i, int j, const vector<vector<int>>& chess, bool bLast)
{
    int nHorse = 0;
    int iCur, jCur;
    for(int k = 0; k < 8; k++)
    {
        iCur = i + iHorse[k];
        jCur = j + jHorse[k];
        if(bLast) //最后一步
        {
            if(CanJump(chess, iCur, jCur))
            {
                pHorse[nHorse].nValidStep = 1;
                pHorse[nHorse].nDirect = k;
                nHorse++;
                break;
            }
        }
        else //正常情况
        {
            pHorse[nHorse].nValidStep = GetNextStep(chess, iCur, jCur);
            if(pHorse[nHorse].nValidStep != 0)
            {
                pHorse[nHorse].nDirect = k;
                nHorse++;
            }
        }
    }
    if(nHorse == 0)
        return 0;
    sort(pHorse, pHorse+nHorse);
    return nHorse;
}
```

|       |    |    |    |    |
|-------|----|----|----|----|
| 32    |    |    |    |    |
| ===== |    |    |    |    |
| 1     | 20 | 5  | 12 | 9  |
| 6     | 13 | 10 | 17 | 4  |
| 19    | 2  | 15 | 8  | 11 |
| 14    | 7  | 18 | 3  | 16 |
| ===== |    |    |    |    |
| 1     | 18 | 5  | 12 | 9  |
| 6     | 13 | 10 | 17 | 4  |
| 19    | 2  | 15 | 8  | 11 |
| 14    | 7  | 20 | 3  | 16 |
| ===== |    |    |    |    |
| 1     | 16 | 5  | 12 | 9  |
| 6     | 13 | 10 | 19 | 4  |
| 17    | 2  | 15 | 8  | 11 |
| 14    | 7  | 18 | 3  | 20 |
| ===== |    |    |    |    |
| 1     | 16 | 5  | 12 | 9  |
| 6     | 13 | 10 | 17 | 4  |
| 15    | 2  | 19 | 8  | 11 |
| 20    | 7  | 14 | 3  | 18 |
| ===== |    |    |    |    |
| 1     | 16 | 5  | 12 | 9  |
| 6     | 19 | 10 | 15 | 4  |
| 17    | 2  | 13 | 8  | 11 |
| 20    | 7  | 18 | 3  | 14 |
| ===== |    |    |    |    |

# 蚁群算法

---

- 蚁群优化算法1991年由Dorigo提出并应用于TSP，已经发展了20多年，具有鲁棒性强、全局搜索、并行分布式计算、易与其他问题结合等优点。
  - 使用传统算法难以求解或无法求解的问题，可以尝试蚁群算法及其改进版本。
- 基本思想：蚂蚁在爬行中会在路径中释放外激素，也能感知路径中已有的外激素：蚂蚁倾向于朝外激素强度高的方向移动。
  - 信息正反馈现象：某路径经过的蚂蚁越多，则后来者选择该路径的概率就越大。

# 蚁群算法的步骤

---

- 核心：路径中的信息素以一定的比例挥发减少，而某蚂蚁经过的路径，信息素以一定的比例释放增加。
- 算法过程： $m$ 只蚂蚁，最大迭代次数为 $K$ 
  - 信息素的初始化
  - 路径构建
  - 信息素更新
  - 2、3步迭代 $K$ 次或最短路径不再变化

# 信息素的初始化

□ 如果初值太小，算法容易早熟，蚂蚁会很快全部集中到一条局部最优路径中。反之，如果初值太大，信息素对搜索方向的指导作用太低，影响算法性能。

□ 一般可以如下初始化：

$$\tau_{ij} = \frac{m}{dist}, \forall i, j$$

■ 其中，m是蚂蚁的数目，dist是路径的估计值

# 路径构建

□ 每只蚂蚁随机选择一城市作为出发点，维护该蚂蚁经过城市的列表(即路径)。在构建路径的每一步中，依概率选择下一个城市。

□ 从第*i*个城市到第*j*个城市的转移概率：

$$P_{ij} \propto \frac{\tau_{ij}^{\alpha}}{d_{ij}^{\beta}}, j \in \{neighbors\ of\ i\}$$

□ 其中， $d_{ij}$  为城市*i*和城市*j*之间的距离，

□  $\alpha, \beta$  为权值调节因子。

# 信息素更新

- 为了模拟蚂蚁在较短的路径留下较多的信息素，当所有蚂蚁到达终点时，更新各路径的信息素浓度。
- 更新公式：
$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{k=1}^m v_{ij}^{(k)}$$
- 其中， $\rho \in (0, 1]$  为信息素的挥发率。
- $v_{ij}^{(k)}$  为第k只蚂蚁在城市i到城市j的边释放的信息素，该值往往取该蚂蚁经过的整个路径的长度倒数。

# Code

```
double AS(const vector<CCity>& pCity, const vector<vector<double>>& ppDistance, vector<int>& bestRoute)
{
    int nCity = (int)pCity.size();
    int m = nCity/2;    //蚂蚁数目
    double t = m/(nCity*GetAvgDistance(ppDistance));    //假定C是某一条路径长度，则t初始化为m/C
    int times = 50;    //进行50轮迭代
    double p = 0.5;    //挥发率
    int i, j;

    //初始化任意两点间的信息素
    vector<vector<double>> pheromone(nCity, vector<double>(nCity));
    for(i = 0; i < nCity; i++)
    {
        for(j = 0; j < nCity; j++)
            pheromone[i][j] = t;
        pheromone[i][i] = 0;
    }

    //AS
    int k;
    vector<vector<int>> r(m, vector<int>(nCity+1));    //蚂蚁按照信息素随机得到的一条路径
    vector<double> rLen(m);    //rLen[i]: r[i]的长度
    double bestRouteLen = -1;    //最优路径的长度
    int best;    //最优路径是哪只蚂蚁得到的
    for(t = 0; t < times; t++)    //迭代若干次
    {
        best = -1;
        for(k = 0; k < m; k++)    //每一只蚂蚁
        {
            RandomRoute(ppDistance, pheromone, r[k]);    //计算第k只蚂蚁的路径
            rLen[k] = CalcLength(r[k], ppDistance);
            if((bestRouteLen < 0) || (bestRouteLen > rLen[k]))
            {
                bestRouteLen = rLen[k];
                best = k;
            }
        }
        if(best != -1)
            bestRoute = r[best];    //当前的最好路径

        //挥发
        Volatilize(pheromone, p);

        //遗留
        for(k = 0; k < m; k++)    //每一只蚂蚁
            AddPheromone(pheromone, r[k], 1/rLen[k]);
    }
    return bestRouteLen;
}
```



# Code

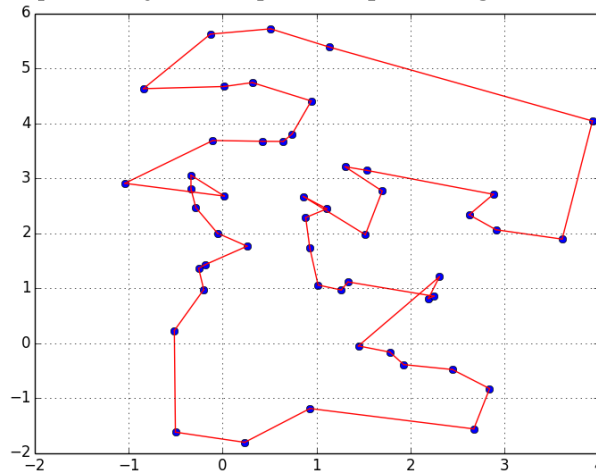
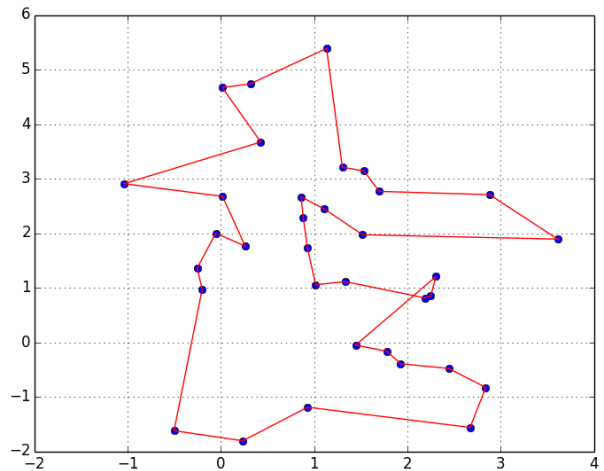
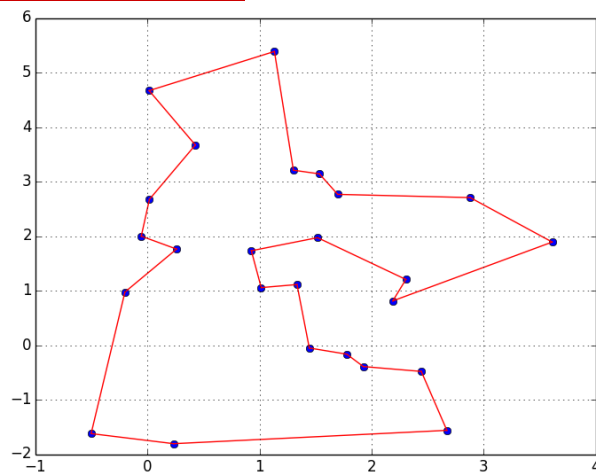
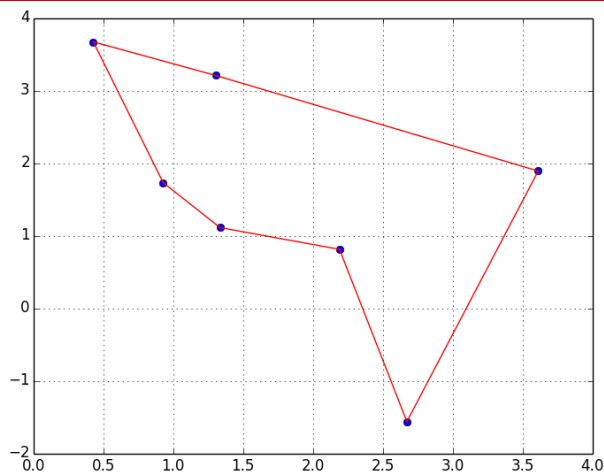
```
void RandomRoute(const vector<vector<double> >& ppDistance, const vector<vector<double> >& pheromone, vector<int>& r)
{
    int nCity = (int)ppDistance.size(); //城市数目
    r[0] = Rand(nCity); //随机挑选初始城市, [0,nCity)
    for(int i = 1; i < nCity; i++)
    {
        r[i] = Select(r, i, ppDistance[r[i-1]], pheromone[r[i-1]]);
    }
    r[nCity] = r[0];
}
```

```
double CalcLength(const vector<int>& r, const vector<vector<double> >& ppDistance)
{
    double s = 0;
    for(int k = 1; k < (int)r.size(); k++)
        s += ppDistance[r[k-1]][r[k]];
    return s;
}
```

```
void Volatilize(vector<vector<double> >& pheromone, double p)
{
    p = 1-p; //输入的p为挥发因子, 所以, 1-p即剩余因子
    int size = (int)pheromone.size();
    int i, j;
    for(i = 0; i < size; i++)
        for(j = 0; j < size; j++)
            pheromone[i][j] *= p;
}
```

```
void AddPheromone(vector<vector<double> >& pheromone, vector<int>& r, double a)
{
    for(int k = 1; k < (int)r.size(); k++)
        pheromone[r[k-1]][r[k]] += a;
}
```

# 蚁群算法效果



# 总结与思考

- 通过递推关系，很容易写出递归代码或动态规划代码。一般的说，动态规划即利用空间存放小规模问题的解，以期便于总问题的求解。递归的过程中，可以借鉴这种方案，保存中间解的结果，避免重复计算。
- 因为递归计算的中间结果必然是最终结果所需要的，有些情况下，可以避免动态规划中计算所有小规模解造成的浪费。
  - 思考：走迷宫问题，往往是从出口回溯。
- 如果需要通过计算具体解，则需要回溯；如果需要计算所有解，则需要深度/广度优先搜索。

# 我们在这里

□ <http://wenda.ChinaHadoop.cn>

■ 视频/课程/社区

□ 微博

■ @ChinaHadoop

■ @邹博\_机器学习

□ 微信公众号

■ 小象

■ 大数据分析挖掘



---

感谢大家！

恳请大家批评指正！