贪心法和动态规划

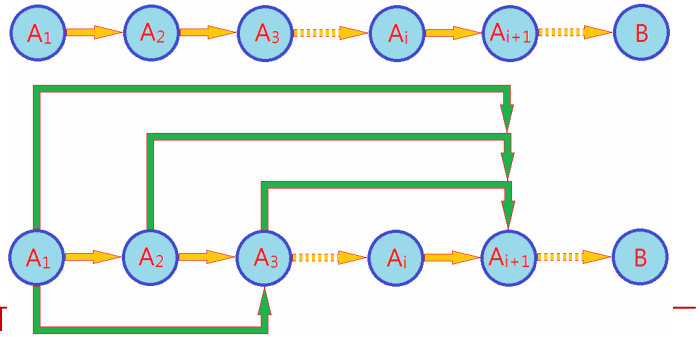
七月算法 邹博

2015年5月5日

认识论

- 认识事物的方法：概念、判断、推理
- 推理中，又分为归纳、演绎。
- 重点考察归纳推理的具体方法。
- 形式化表述：
  - 已知：问题规模为n的前提A
  - 求解/求证：未知解B/结论B
  - 记号：用A_n表示“问题规模为n的已知条件”

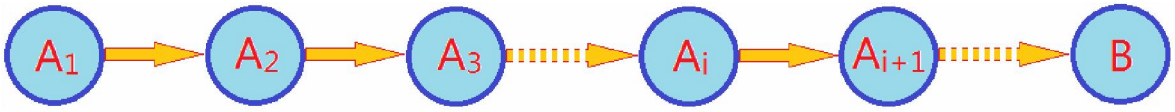
对归纳推理的理解



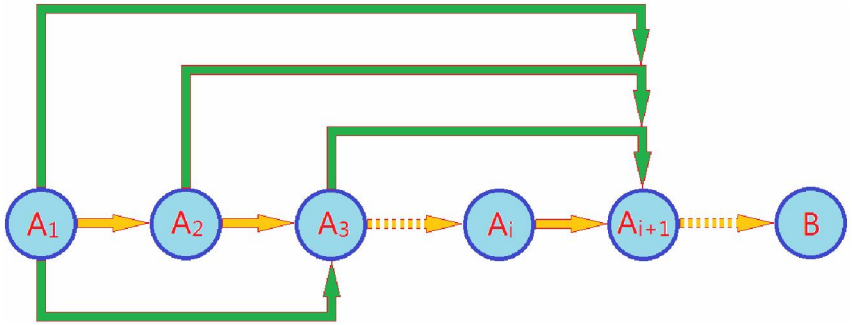
- 若将问题规模降低到0，即已知A_0，很容易计算或证明B，则有：$A_0 \rightarrow B$
- 同时，考察从A_0增加一个元素，得到A1的变化过程。即：$A_0 \rightarrow A_1$；
  - 进一步考察$A_1 \rightarrow A_2$，$A_2 \rightarrow A_3$ ……$A_i \rightarrow A_{i+1}$
  - 这种方法是(严格的)归纳推理，常常被称作**数学归纳法**。
  - 此时，由于上述推导往往不是等价推导(Ai和Ai+1不是互为充要条件)，导致随着i的增加，有价值的前提信息越来越少；为避免这一问题，采取如下方案：
  - $\{A_1\} \rightarrow A_2$，$\{A_1A_2\} \rightarrow A_3$ …… $\{A_1A_2 \ldots A_i\} \rightarrow A_{i+1}$
  - 相对应的，修正后的方法依然是(严格的)归纳推理，有时被称作**第二数学归纳法**。

对归纳推理的理解

- 基本归纳法：对于A_{i+1}，只需考察前一个状态A_i即可完成整个推理过程，它的特点是只要状态A_i确定，则计算A_{i+1}时不需要考察更前序的状态$A_1 \ldots A_{i-1}$，在图论中，常常称之为马尔科夫模型；

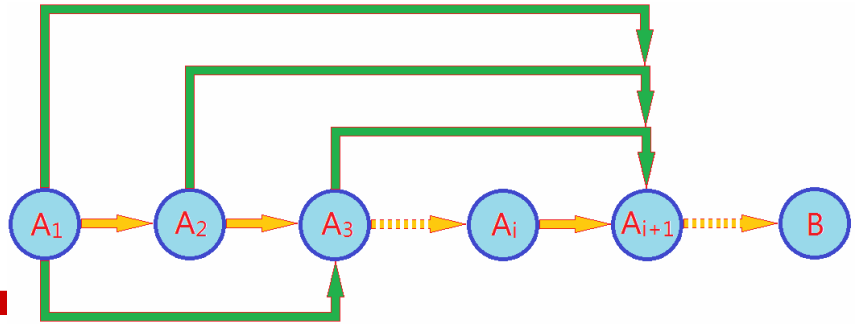


- 高阶归纳法：相应的，对于A_{i+1}，需考察前i个状态集$\{A_1 \ldots A_{i-1} A_i\}$才可完成整个推理过程，往往称之为高阶马尔科夫模型；



- 在计算机算法中，高阶马尔科夫模型的推理，叫做“动态规划”，马尔科夫模型的推理，对应“贪心法”。

说明



- 无论动态规划还是贪心法，都是根据A[0...i]计算A[i+1]的过程
  - ——计算A[i+1]不需要A[i+2]、A[i+3]……，
  - ——一旦计算完成A[i+1]，再后面计算A[i+2]、A[i+3]……时，不会更改A[i+1]的值。
  - 这即无后效性。
- 鉴于无后效性的原因，可以这么理解动态规划：计算A[i+1]只需要知道A[0...i]的值，无需知道A[0...i]是如何计算得到的——无需知道它们是通过何种途径得到的。如果将A[0...i]的全体作为一个整体，则可以认为动态规划法是马尔科夫过程，而非高阶马尔科夫过程。

贪心法

- 根据实际问题，选取一种度量标准。然后按照这种标准对n个输入排序，并按序一次输入一个量。
- 如果输入和当前已构成在这种量度意义下的部分最优解加在一起不能产生一个可行解，则不把此输入加到这部分解中。否则，将当前输入合并到部分解中从而得到包含当前输入的新的部分解。
- 这一处理过程一直持续到n个输入都被考虑完毕，则记入最优解集合中的输入子集构成这种量度意义下的问题的最优解。
- 这种能够得到某种量度意义下的最优解的分级处理方法称为贪心方法。

超市装东西

- 面前有大米、小米、糯米、黄米、紫米、红豆、黑豆、绿豆若干，各种粮食具有不同的价格。要求在总重量20kg以内，装最高价值的物品。

最小生成树MST

- 最小生成树要求从一个带权无向连通图中选择n－1条边并使这个图仍然连通(也即得到了一棵生成树)，同时还要考虑使树的权最小。为了得到最小生成树，人们设计了很多算法，最著名的有Prim算法和Kruskal算法，这两个算法都是贪心算法。
  - Prim算法：从某个(任意一个)结点出发，选择与该结点邻接的权值最小的边；随着结点的不断加入，每次都选择这些结点发出的边中权值最小的；重复n-1次。
  - Kruskal算法：将边按照权值递增排序，每次选择权值最小并且不构成环的边，重复n-1次。

最短路径

- 将Prim算法稍做调整，就得到Dijkstra最短路径算法：
  - 结点集V初始化为源点S一个元素：V={S}，到每个点的最短路径的距离初始化为dist[u]=graph[S][u]；
  - 选择最小的dist[u]：记dist[v]是最小的，则v是当前找到的不在V中且距离S最近的结点，更新V=V∪{v}，调整dist[u]=min{dist[u],dist[v]+graph[v][u]}；
  - 重复n-1次。

贪心法的思考

- 可以看到，在从A_i到A_{i+1}的扩展过程中，上述三个算法都没有使用A[0...i-1]的值。
- 往往看名字，认为它很简单；事实上，贪心法其实并不轻松，它需要严格证明一定与更先序的值无关。
  - 思考：从1元、2元、5元的纸币，问给定总价值N元，最少需要几张纸币？

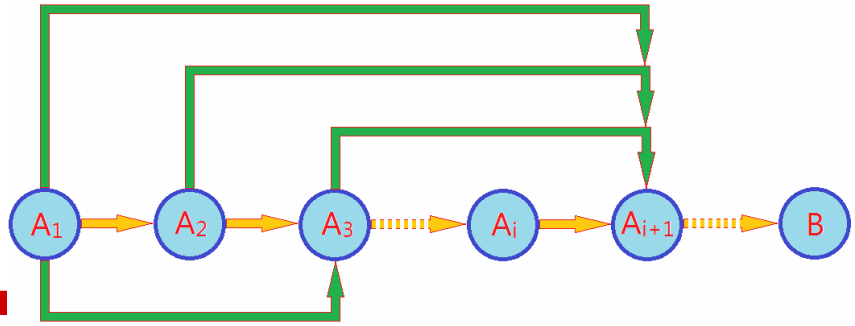
最长递增子序列LIS

- Longest Increasing Subsequence
- 给定一个长度为N的数组，找出一个最长的单调递增子序列(不一定连续，但是顺序不能乱)。例如：给定一个长度为6的数组A{5，6，7，1，2，8}，则其最长的单调递增子序列为{5，6，7，8}，长度为4。
  - 分析：其实此LIS问题可以转换成最长公子序列问题，为什么呢？

附：使用LCS解LIS问题

- 原数组为A {5，6，7，1，2，8}
- 排序后：A’{1，2，5，6，7，8}
- 因为，原数组A的子序列顺序保持不变，而且排序后A’本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若想求数组A的最长递增子序列，其实就是求数组A与它的排序数组A’的最长公共子序列。

LIS的算法分析



- 长度为N的数组记为$A=\{a_0a_1a_2...a_{n-1}\}$；
- 记A的前i个字符构成的前缀串为$A_i= a_0a_1a_2...a_{i-1}$，以a_i结尾的最长递增子序列记做L_i，其长度记为a[i]；
- 假定已经计算得到了a[0,1…,i-1]，如何计算a[i]呢？
- 根据定义，L_i必须以a_i结尾，如果将a_i缀到$L_0 L_1 \ldots\ldots L_{i-1}$的后面，是否允许呢？
  - 如果$a_j < a_i$，则可以将a_i缀到L_j的后面，并且使得Lj的长度变长。
- 从而：$a[i]=\{\max(a(j))+1, 0 \leqslant j < i \text{且} a[j] \leqslant a[i]\}$
  - 需要遍历在i之前的所有位置j，找出满足条件$a[j] \leqslant a[i]$的a[j]；
  - 计算得到a[0…n-1]后，遍历所有的a[i]，找出最大值即为最大递增子序列的长度。
  - 时间复杂度为$O(N^2)$。
- 思考：如何求最大递增子序列本身？
  - 记录前驱

LIS Code

```
#include <vector>
#include <algorithm>
using namespace std;

int LIS(const int* p, int length, int* pre, int& nIndex)
{
    int* longest = new int[length];
    int i,j;

    for(i = 0; i < length; i++)
    {
        longest[i] = 1;
        pre[i] = -1;
    }

    int nLis = 1;
    nIndex = 0;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                if(longest[i] < longest[j]+1)
                {
                    longest[i] = longest[j]+1;
                    pre[i] = j;
                }
            }
        }
        if(nLis < longest[i])
        {
            nLis = longest[i];
            nIndex = i;
        }
    }

    delete[] longest;

    return nLis;
}

void GetLIS(const int* array, const int* pre, int nIndex, vector<int>& lis)
{
    while(nIndex >= 0)
    {
        lis.push_back(array[nIndex]);
        nIndex = pre[nIndex];
    }
    reverse(lis.begin(), lis.end());
}

void Print(int* p, int size)
{
    for(int i = 0; i < size; i++)
        cout << p[i] << '\t';
    cout << '\n';
}

int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {1,4,5,6,2,3,8,9,10,11,12,12,1};
    int size = sizeof(array)/sizeof(int);
    int* pre = new int[size];
    int nIndex;
    int max = LIS(array, size, pre, nIndex);
    vector<int> lis;
    GetLIS(array, pre, nIndex, lis);
    delete[] pre;
    cout << max << endl;
    Print(&lis.front(), (int)lis.size());
    return 0;
}
```

LIS Code split

```
int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {1,4,5,6,2,3,8,9,10,11,12,12,1};
    int size = sizeof(array)/sizeof(int);
    int* pre = new int[size];
    int nIndex;
    int max = LIS(array, size, pre, nIndex);
    vector<int> lis;
    GetLIS(array, pre, nIndex, lis);
    delete[] pre;
    cout << max << endl;
    Print(&lis.front(), (int)lis.size());
    return 0;
}
```

```
void GetLIS(const int* array, const int* pre,
            int nIndex, vector<int>& lis)
{
    while(nIndex >= 0)
    {
        lis.push_back(array[nIndex]);
        nIndex = pre[nIndex];
    }
    reverse(lis.begin(), lis.end());
}
```

```
#include <vector>
#include <algorithm>
using namespace std;

int LIS(const int* p, int length, int* pre, int& nIndex)
{
    int* longest = new int[length];
    int i,j;

    for(i = 0; i < length; i++)
    {
        longest[i] = 1;
        pre[i] = -1;
    }

    int nLis = 1;
    nIndex = 0;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                if(longest[i] < longest[j]+1)
                {
                    longest[i] = longest[j]+1;
                    pre[i] = j;
                }
            }
        }
        if(nLis < longest[i])
        {
            nLis = longest[i];
            nIndex = i;
        }
    }

    delete[] longest;

    return nLis;
}
```

```
void Print(int* p, int size)
{
    for(int i = 0; i < size; i++)
        cout << p[i] << '\t';
    cout << '\n';
}
```

矩阵乘积

- 根据矩阵相乘的定义来计算 C=A×B，需要m*n*s次乘法。
- 三个矩阵A、B、C的阶分别是$a_0 \times a_1$, $a_1 \times a_2$, $a_2 \times a_3$，从而$(A \times B) \times C$和$A \times (B \times C)$的乘法次数是$a_0a_1a_2+a_0a_2a_3$、 $a_1a_2a_3+a_0a_1a_3$，二者一般情况是不相等的。
  - 问：给定n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，如何添加括号来改变计算次序，使得乘法的计算量最小？
- 此外：若A、B都是n阶方阵，C的计算时间复杂度为$O(n^3)$
  - 问：可否设计更快的算法？
  - 答：分治法：Strassen分块——理论意义大于实践意义。

矩阵连乘的提法

- 给定n个矩阵$\{A_1,A_2,\ldots,A_n\}$，其中A_i与A_{i+1}是可乘的，i=1，2...，n-1。考察该n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的乘法次数最少。
  - 即：利用结合律，通过加括号的方式，改变计算过程，使得数乘的次数最少。

分析

- 将矩阵连乘积$A_iA_{i+1}...A_j$记为A[i:j]，这里i≤j
  - 显然，若i==j，则A[i:j]即A[i]本身。
- 考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，i≤k<j，则其相应的完全加括号方式为

$$(A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$$

- 计算量：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量

最优子结构

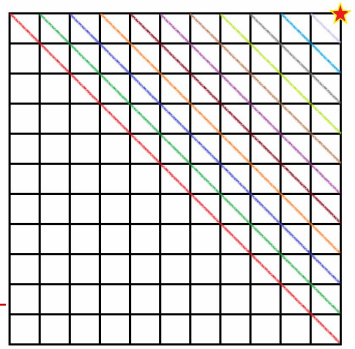
- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。
- 最优子结构性质是可以使用动态规划算法求解的显著特征。

状态转移方程 $(A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$

- 设计算A[i:j](1≤i≤j≤n)所需要的最少数乘次数为m[i,j]，则原问题的最优值为m[1,n]；
- 记A_i的维度为 $p_{i-1} \times p_i$
- 当i==j时，A[i:j]即A_i本身，因此，m[i,i]=0；(i=1,2,…,n)
- 当i<j时，$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
- 从而：

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & i < j \end{cases}$$

从算法到实现

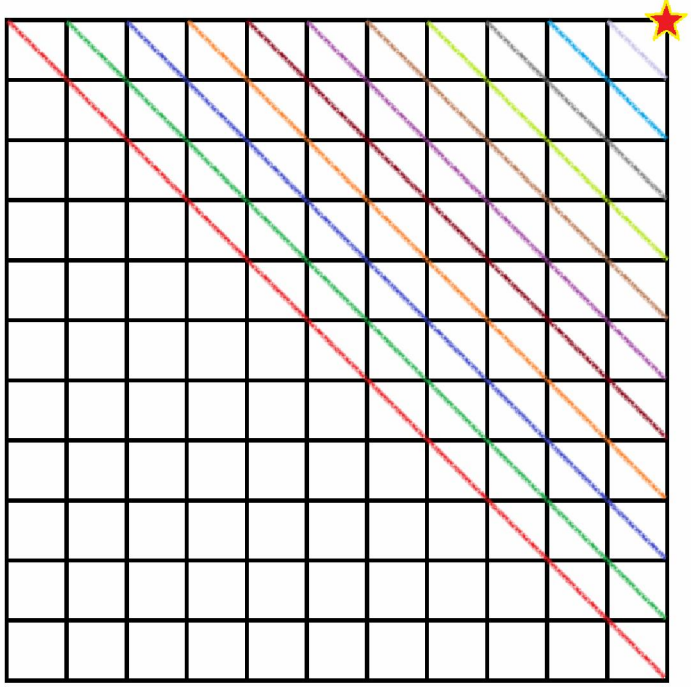


- 由m[i,j]的递推关系式可以看出，在计算m[i,j]时，需要用到m[i+1,j]，m[i+2,j]…m[j-1,j]；

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

- 因此，求m[i,j]的前提，不是m[0…i-1;0…j-1]，而是沿着主对角线开始，依次求取到右上角元素。
- 因为m[i,j]一个元素的计算，最多需要遍历n-1次，共$O(n^2)$个元素，故算法的时间复杂度是$O(n^3)$，空间复杂度是$O(n^2)$。

Code



```
//p[0...n]存储了n+1个数，其中，(p[i-1],p[i])是矩阵i的阶；
//s[i][j]记录A[i...j]从什么位置断开；m[i][j]记录数乘最小值
void MatrixMultiply(int* p, int n, int** m, int** s)
{
    int r,i,j,k,t;
    for(i = 1; i <= n; i++)
        m[i][i] = 0;

    //r个连续矩阵的连乘：上面的初始化，相当于r=1
    for(r = 2; r <= n; r++)
    {
        for(i = 1; i <= n-r+1; i++)
        {
            j=i+r-1;
            m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
            s[i][j] = i;
            for(k = i+1; k < j; k++)
            {
                t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if(t < m[i][j])
                {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

字符串的交替连接

- 输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交错而成，即不改变s1和s2中各个字符原有的相对顺序，例如当s1=“aabcc”，s2=“dbbca”，s3=“aadbbcbcac”时，则输出true，但如果s3=“accabdbbca”，则输出false。
- 换个表述：
  - s1和s2是s3的子序列，且s1∪s2=s3

问题说明

- 若s1和s2没有字符重复：遍历s3的同时，考察是否是s1和s2的字符即可；
- 若字符重复：可以用压栈的方式解决；
- 此外，还能用动态规划，代码更为简洁。

状态转移函数

- 令dp[i,j]表示s3[1...i+j]是否由s1[1...i]和s2[1...j]的字符组成：即dp[I,j]取值范围为true/false
- 如果s1[i]==s3[i+j]，且dp[i-1,j]为真，那么dp[i][j]为真；
- 如果s2[j]==s3[i+j]，且dp[i,j-1]为真，那么dp[i][j]为真；
- 其它情况，dp[i][j]为假。

Code

```
public boolean IsInterleave(String s1, String 2, String 3){
    int n = s1.length(), m = s2.length(), s = s3.length();
    //如果长度不一致，则s3不可能由s1和s2交错组成
    if (n + m != s)
        return false;

    boolean[][]dp = new boolean[n + 1][m + 1];

    //在初始化边界时，认为空串可以由空串组成，因此dp[0][0]赋值为true
    dp[0][0] = true;

    for (int i = 0; i < n + 1; i++){
        for (int j = 0; j < m + 1; j++){
            if ((i - 1 >= 0 && dp[i - 1][j] == true &&
                s1.charAT(i - 1) == s3.charAT(i + j - 1))//取s1字符
                ||
                (j - 1 >= 0 && dp[i][j - 1] == true &&
                s2.charAT(j - 1) == s3.charAT(i + j - 1)))//取s2字符
                dp[i][j] = true;
            else
                dp[i][j] = false;
        }
    }
    return dp[n][m];
}
```

走棋盘/格子取数

- 给定m*n的矩阵，每个位置是一个非负整数，从左上角开始，每次只能朝右和下走，走到右下角，求总和最小的路径。

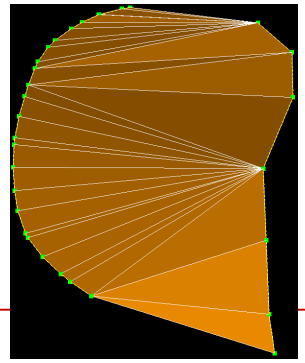
	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	13	0	0	6	0	0
3	0	0	0	0	7	0	0	0
4	0	0	0	14	0	0	0	0
5	0	21	0	0	0	4	0	0
6	0	0	15	0	0	0	0	0
7	0	14	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

棋盘走法的状态转移函数

- 走的方向决定了同一个格子一定不会经过两次。
  - 若当前位于(x,y)处，它的上一步来自于哪些格子呢？
  - dp[0,0]=a[0,0]
  - dp[x,y] = min(dp[x-1,y]+a[x,y],dp[x,y-1]+a[x,y])
  - 即：dp[x,y] = min(dp[x-1,y],dp[x,y-1]) +a[x,y]

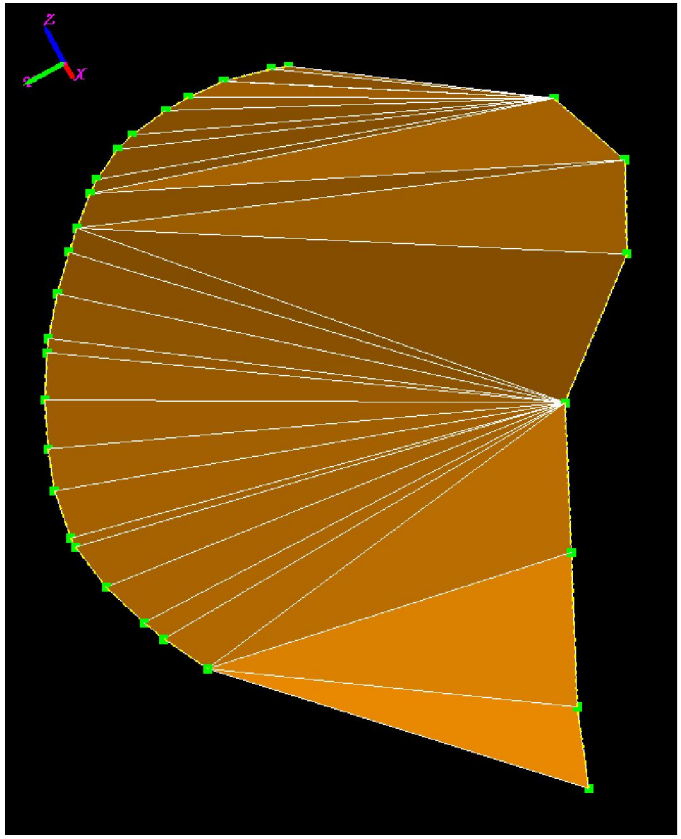
- 思考：若将上述问题改成“求从左上到右下的最大路径”呢？

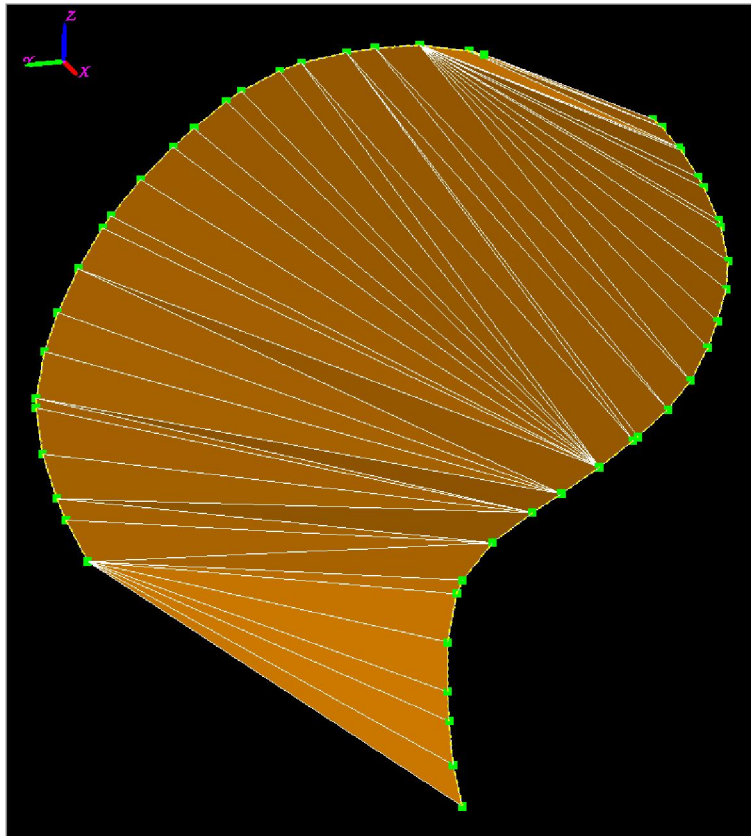
实践应用



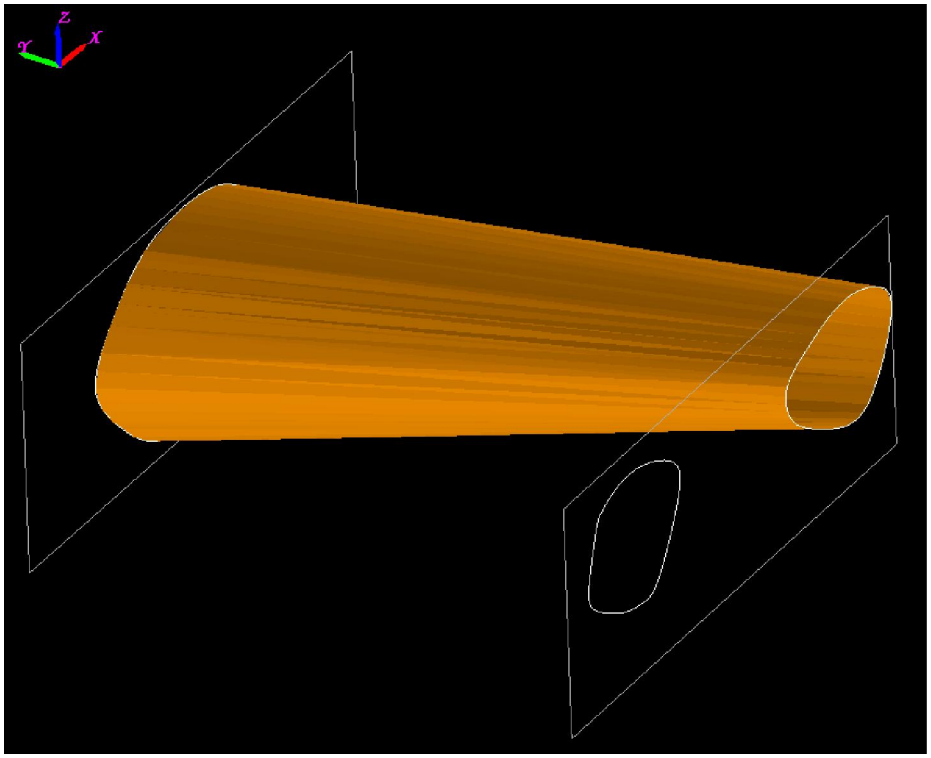
- 给定空间两条曲线，设计经过这两条曲线的曲面，使得该曲面是“最合理”的。
  - 探讨如下方案：两条曲线上的点序列，记做P1[0…N-1]和P2[0…M-1]，计算P1[i]和P2[j]的距离，从而得到二维表格T[N][M]。从T[0][0]开始出发，只能向右和向下走(曲面不能自相交)，从(0,0)到(N-1,M-1)的最短路径，就是曲面的一种“合理连接”。

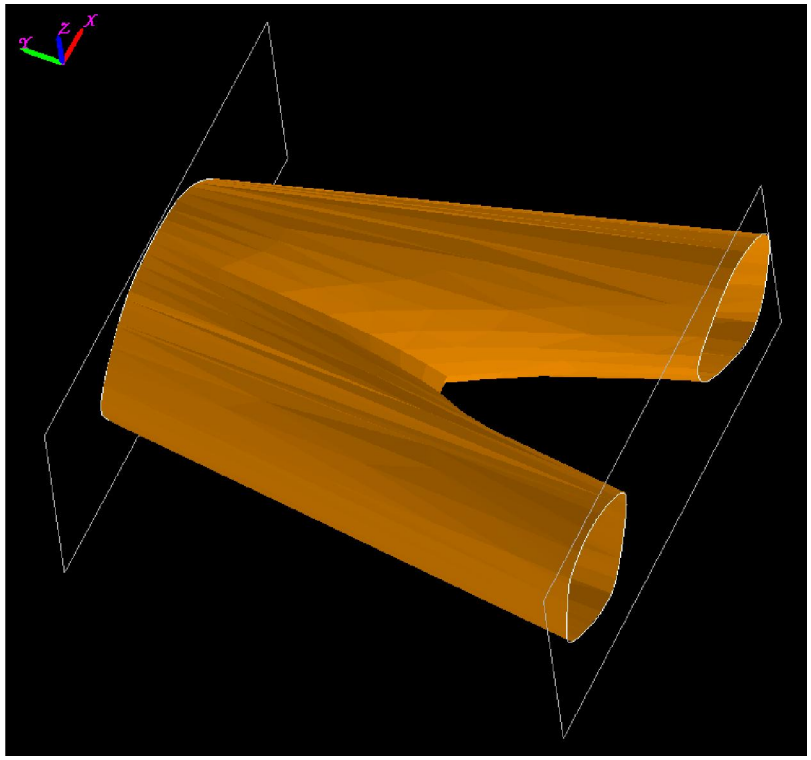
GIS中的应用





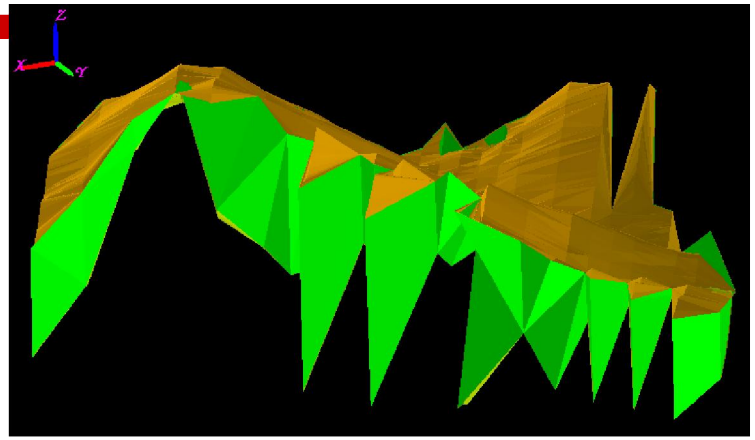
如果三维曲线是封闭线...

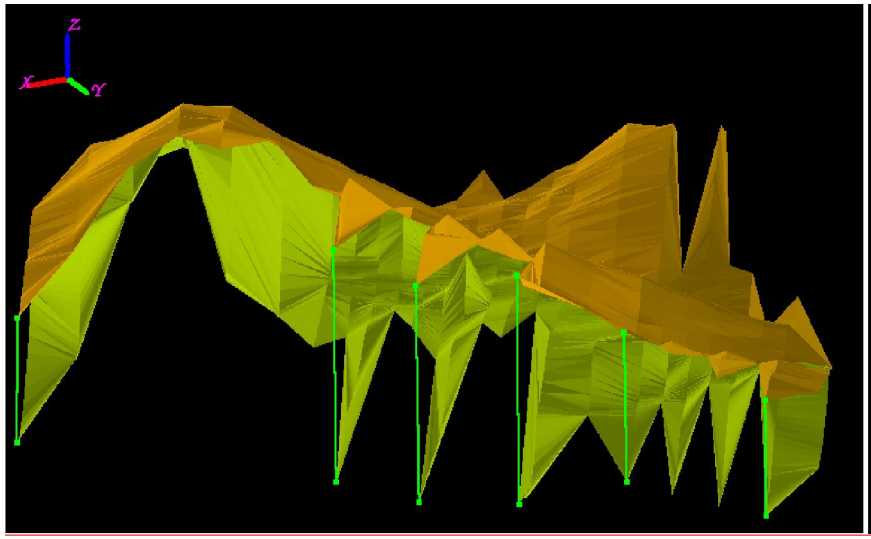


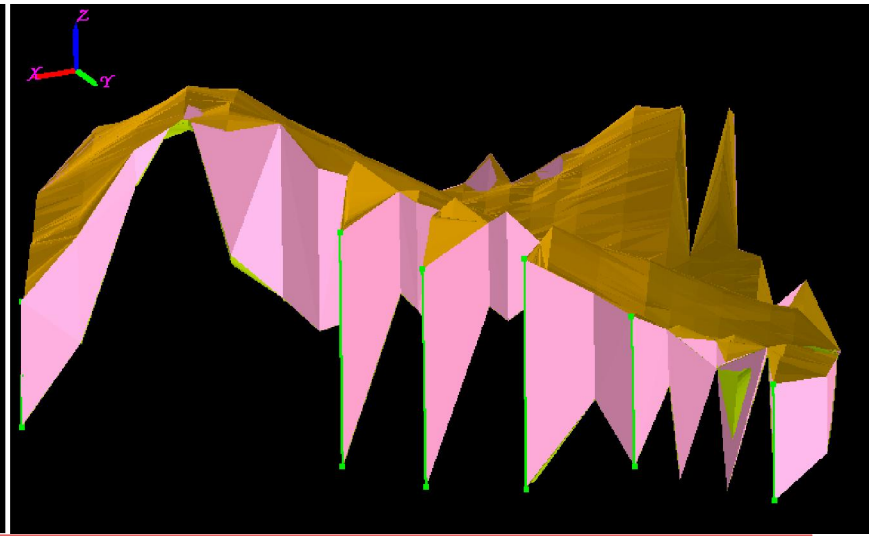


GIS中三维建模的实际应用

- 右上：未使用引导线
- 左下：输入的引导线
- 右下：过引导线的曲面







带陷阱的走棋盘

- 有一个n*m的棋盘网格，机器人最开始在左上角，机器人每一步只能往右或者往下移动。棋盘中有些格子是禁止机器人踏入的，该信息存放在二维数组blocked中，如果blocked[i][j]为true，那么机器人不能踏入格子(i,j)。请计算有多少条路径能让机器人从左上角移动到右下角。

状态转移方程

- dp[i][j]表示从起点到(i,j)的路径条数。
- 只能从左边或者上边进入一个格子
- 如果(i,j)被占用
  - dp[i][j]=0
- 如果(i,j)不被占用
  - dp[i][j]=dp[i-1][j]+dp[i][j–1]

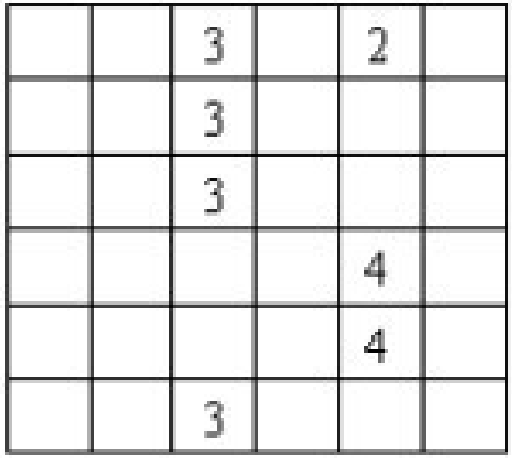
- 思考：如果没有占用的格子呢?
- 一共要走m+n–2步，其中(m–1)步向右，(n-1)步向下。组合数C(m+n–2,m-1)=C(m+n-2,n-1)

两次走棋盘

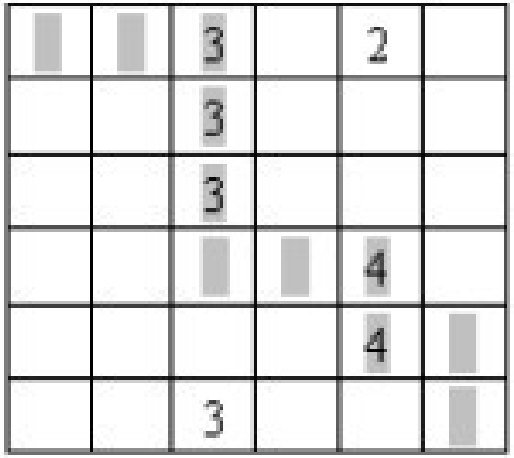
- 给定m*n的矩阵，每个位置是一个非负的权值，从左上角开始，每次只能朝右和下走，走到右下角；然后，从右下角开始，每次只能朝左和朝上走，走到左上角。求权值总和最小的路径。若相同格子走过两次，则该位置的权值只算一次。

分析

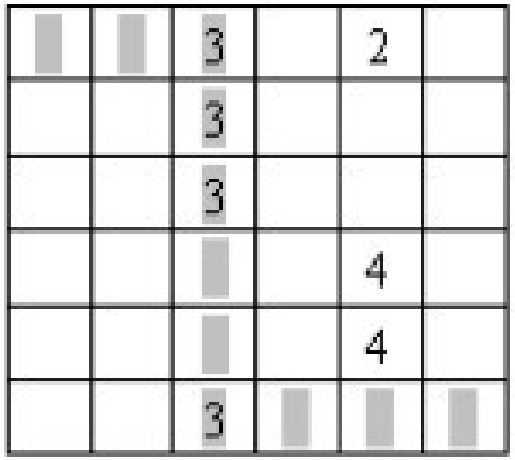
- 贪心不能解决问题



图一



图二



图三

分析格网棋盘的特点

- 考察5*7的矩阵棋盘C，其中，C[i,j]表示的值v表示第v步能够到底的位置。

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

举例说明状态的定义

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

- 在经过8步后，肯定处于C中编号为8的位置。而C中共有3个编号为8的，它们分别是C的第2、3、4行。故假设第1次经过8步走到了C中的第2行，第2次经过8步走到了C中的第3行，用dp[8,2,3]表示。
- 用dp[s,i,j]记录两次所走的路径获得的最大值，其中s表示走的步数，i和j表示在s步后第1次走的位置和第2次走的位置。由于s=m+n-2，0≤i＜n，0≤j＜m，所以共有$O(n^3)$个状态。

状态转移函数的定义

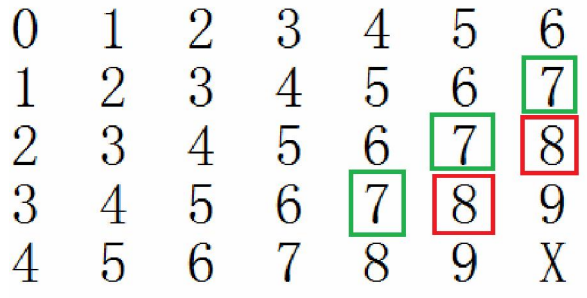
0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

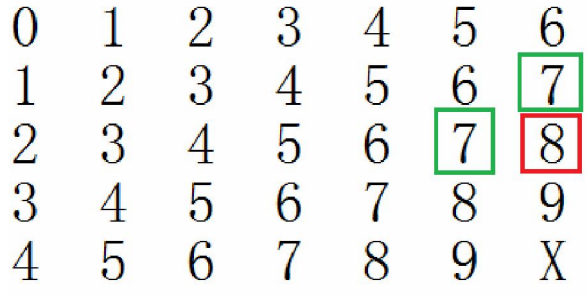
状态转移函数的定义

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

完整的状态转移函数

- if(i!=j)
  - DP[s,i,j]=Max(
    - DP[s-1,i-1,j-1],
    - DP[s-1,i-1,j],
    - DP[s-1,i,j-1],
    - DP[s-1,i,j])
    - + W[i,s-i] + W[j,s-j]
- Else
  - DP[s,i,j]=Max(
    - DP[s-1,i-1,j-1],
    - DP[s-1,i-1,j],
    - DP[s-1,i,j])
    - +W[i,s-i]
- 其中W[x,y]表示棋盘位置(x,y)的权值。





Code

```
const int N = 202;
const int inf = 1000000000;  //无穷大
int dp[N * 2][N][N];
bool IsValid(int step, int x1, int x2, int n)
{
    int y1 = step - x1, y2 = step - x2;
    return ((x1 >= 0) && (x1 < n) && (x2 >= 0) && (x2 < n) && (y1 >= 0) && (y1 < n) && (y2 >= 0) && (y2 < n));
}

int GetValue(int step, int x1, int x2, int n)
{
    return IsValid(step, x1, x2, n) ? dp[step][x1][x2] : (-inf);
}

//dp[step][i][j]表示在第step步两次分别在第i行和第j行的最大得分
int MinPathSum(int a[N][N], int n)
{
    int P = n * 2 - 2; //最终的步数
    int i, j, step;

    //不能到达的位置 设置为负无穷大
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[0][i][j] = -inf;
        }
    }
    dp[0][0][0] = a[0][0];

    for(step = 1; step <= P; ++step)
    {
        for(i = 0; i < n; ++i)
        {
            for(j = i; j < n; ++j)
            {
                dp[step][i][j] = -inf;
                if(!IsValid(step, i, j, n))   //非法位置
                    continue;
                //对于合法的位置进行dp
                if(i != j)
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                    dp[step][i][j] += a[i][step - i] + a[j][step - j];  //不在同一个格子，加两个数
                }
                else
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j,   n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j,   n));
                    dp[step][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次
                }
            }
        }
    }
    return dp[P][n - 1][n - 1];
}
```

Code split

```
//dp[step][i][j]表示在第step步两次分别在第i行和第j行的最大得分
int MinPathSum(int a[N][N], int n)
{
    int P = n * 2 - 2; //最终的步数
    int i, j, step;

    //不能到达的位置 设置为负无穷大
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[0][i][j] = -inf;
        }
    }
    dp[0][0][0] = a[0][0];

    for(step = 1; step <= P; ++step)
    {
        for(i = 0; i < n; ++i)
        {
            for(j = i; j < n; ++j)
            {
                dp[step][i][j] = -inf;
                if(!IsValid(step, i, j, n))    //非法位置
                    continue;
                //对于合法的位置进行dp
                if(i != j)
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                    dp[step][i][j] += a[i][step - i] + a[j][step - j];
                }
                else
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j,   n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j,   n));
                    dp[step][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次
                }
            }
        }
    }
    return dp[P][n - 1][n - 1];
}
```

矩阵连乘问题的进一步思考

- 给定n个矩阵$\{A_1,A_2,\ldots,A_n\}$，其中A_i与A_{i+1}是可乘的，i=1，2...，n-1。考察该n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，有多少种不同的计算次序？换句话说，有多少种加括号的方式？

分析

- n个矩阵连乘，可以分解成i个矩阵连乘和(n-i)个矩阵连乘，最后，再将这两个矩阵相乘。故：

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases}\Rightarrow P(n)=\Omega(\frac{4^n}{\sqrt{\pi}*n^{3/2}})$$

$$P(n)=\frac{1}{n}C_{2n-2}^{n-1}$$

- 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……

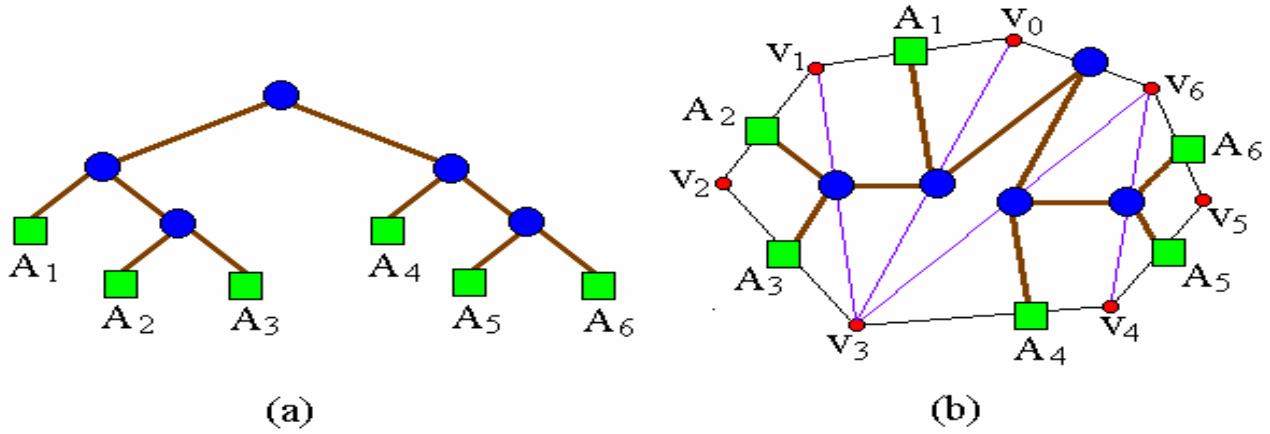
卡塔兰数 Catalan $H(n)=\frac{1}{n+1}C_{2n}^{n}$

- 有N个节点的二叉树共有多少种情形？
- 一个栈(无穷大)的进栈序列为1,2,3,..n,有多少个不同的出栈序列？
- 凸多边形三角化：将一个凸多边形划分成三角形区域的方法有多少种？
- 由左而右扫描由n个1和n个0组成的2n位二进制数，要求在任何时刻，1的累计数不小于0的累计数。求满足这样条件的二进制数的个数。

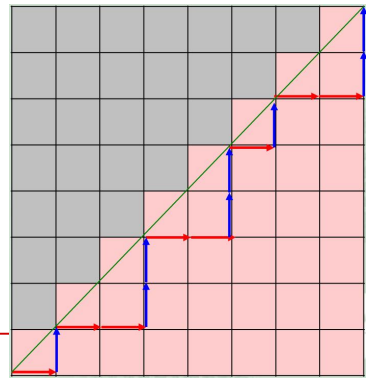
- 注：由h(n)=C(n,2n)/(n+1)很容易求得：h(n)=h(n-1)*(4*n-2)/(n+1)=c(2n,n)-c(2n,n+1)

上述问题的相互联系

- 一个矩阵(一个表达式)的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积((A1(A2A3))(A4(A5A6)))所相应的语法树如图(a)所示。
- 凸多边形{v0,v1,...vn-1}的三角剖分也可以用语法树表示。例如，图(b)中凸多边形的三角剖分可用图(a)所示的语法树表示。
  - 矩阵连乘积中的每个矩阵Ai对应于凸(n+1)边形中的一条边vi-1vi。三角剖分中的一条弦vivj，i<j，对应于矩阵连乘积A[i+1:j]。



附：证明Catalan数公式1/2相等

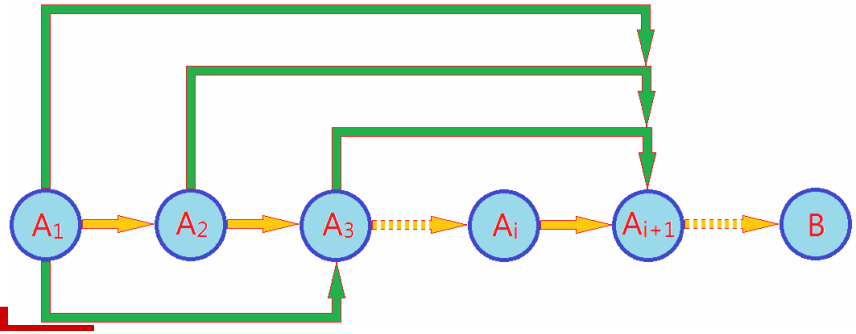


- 考察问题：n*n棋盘从左下角走到右上角而不穿过主对角线的走法。
- 考虑n*n棋盘，记主对角线为L。从左下角走到右上角不穿过对角线L的所有路径，不算起点，一定有第一次接触到L的位置(可能是终点)，设此位置为M，坐标为(x,x)——设第一个数为横轴坐标。该路径一定从下方的(x,x-1)而来，而起点处第一步也一定是走向(1,0)，两者理由相同——否则就穿过了主对角线。考虑从(1,0)到(x,x-1)的(x-1)*(x-1)的小棋盘中，因为在此中路径一直没有接触过主对角线(M的选取)，所以在此小棋盘中路径也一定没有穿过从(1,0)到(x,x-1)的小棋盘的对角线L1。这样在这个区域中的满足条件的路径数量就是一个同构的子问题，解应该是F(x-1)，而从M到右上角终点的路径数量也是一个同构的子问题，解应该是F(n-x)，而第一次接触到主对角线的点可以从(1,1)取到(n,n)，这样就有F(n)=∑(k=1…n){F(k-1)*F(n-k)}=∑(k=0…n-1){F(k-1)*F(n-k)}。
- 注：抽象成2n个操作组成的操作链，其中A操作和B操作各n个，且要求截断到操作链的任何位置都有：A操作(向右走一步)的个数不少于B操作(向上走一步)的个数。

附：Catalan数公式1和公式2相等的证明

- 现在证明上述问题的解为C(2n,n)/(n+1)
- 思路是先求所有从(0,0)到(n,n)的路径数X，再求所有穿过主对角线L的从(0,0)到(n,n)的路径数Y，用前者减去后者得到所求。
- 从(0,0)到(n,n)的路径数显然是C(2n,n)，一共要走2n步到达右上角，其中向右和向上各n步，总走法是C(2n,n)。
- 考虑一个新增的位置(n-1,n+1)，它位于终点的左上角一个格处，设所有从(0,0)到(n-1,n+1)的路径数为Z，下面要证明Y和Z相等，从而通过求Z来求Y。
  - 考虑从(0,1)到(n-1,n)的对角线L2，对于所有穿过L而到达终点的路径，一定会接触到L2，找出某路径第一次接触到L2的位置M1，将从M1到终点的路径沿L2做对折一定会得到一条从M1到(n-1,n+1)的路径，故每条穿过L到达终点的路径都对应一条到达(n-1,n+1)的路径，即有Y<=Z。
  - 所有从起点到达(n-1,n+1)的路径都一定会穿过L2，找出某路径第一次穿过L2的位置M2，将M2到(n-1,n+1)的路径沿L2对折，就得到一条M2到(n,n)的路径，且该条路径一定穿过L，故每条到达(n-1,n+1)的路径都对应一条穿过L到达终点的路径，即有Z<=Y。
- 故Z==Y。
- Z是显然的从(0,0)到(n-1,n+1)共需走2n步，其中向右n-1步、向上n+1步，故Z=C(2n,n-1)。
- 由以上可知F(n)=X-Y=X-Z=C(2n,n)-C(2n,n-1)=C(2n,n)/(n+1)。

总结



- 相对于前面使用存储结构来划分章节：“数组”、“字符串”、“树”、“图”(它们是世界观)，动态规划是方法论，是解决一大类问题的通用思路。事实上，前面章节论述的很多内容都可以归结为动态规划的思想。
  - 如：KMP中求next数组的过程：已知next[0…i-1]，求next[i]；
- 何时可以考虑使用动态规划：
  - 初始规模下能够方便的得出结论
    - 空串、长度为0的数组、自身等
  - 能够得到问题规模增大导致的变化
    - 递推式——状态转移方程
- 事实上，动态规划还有个“无后效性”的要求
  - 一旦计算得到了A[0…i-1]，那么，计算A[i]时只可能读取A[0…i-1]，而不会更改它们的值——过去发生的，只能承认，不能改变；
  - 一旦计算得到了A[0…i-1]，那么，计算A[i]时只需要读取A[0…i-1]的值即可，不需要事先知道A[i+1…n-1]的值——未来的事情，完全未知。
- 在实践中往往忽略无后效性这一要求：
  - 要么问题本身决定了它是成立的：格子取数问题；
  - 要么通过更改计算次序，可以达到该要求：矩阵连乘I问题。

我们在这里

- 更多算法面试题在 7 七月算法
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @研究者July
  - @七月问答
  - @邹博_机器学习

感谢大家！

恳请大家批评指正！