

# 寻寻觅觅·道阻且长

## 动态规划与寻路问题

### v0.1 preview

崔添翼 (Tianyi Cui, a.k.a. dd\_engi)

September 16, 2011

本文题为《寻寻觅觅·道阻且长》，副题为《动态规划与寻路问题》，从属于《动态规划的思考艺术》系列。

本文的修订历史及最新版本请访问 <https://github.com/tianyicui/DP-Book> 查阅。

本文版权归原作者所有，采用 [CC BY-NC-SA](#) 协议发布。

本文的这个版本并未公开发布，只由作者直接发布给若干试读者，试读者没有传播给别人的权利。若你并非作者指定的试读者，而通过其它途径读到本文，则说明存在至少一位试读者未尽保密义务。

## Contents

1 寻路问题导引	2
2 特殊图中的寻路	2
2.1 矩阵中的寻路	2
2.1.1 基础问题	2
2.1.2 变形问题	3
2.2 有向无环图中的寻路	4
3 一般图中的寻路	4
3.1 单起点寻路的三种算法	4
3.1.1 Dijkstra算法	4
3.1.2 Bellman-Ford算法	4
3.1.3 Yen算法	4
3.2 全图寻路的Floyd算法	4
3.3 Hamiltonian回路	4
3.3.1 简单的动态规划算法	4
3.3.2 优化的动态规划算法	4
4 扩展的寻路问题	4
4.1 扩展的Dijkstra算法	4

## 1 寻路问题导引

本文所探讨的寻路问题是指：给定某个可抽象为图论中的模型的图，要求找到图中满足某种性质的一条路径。除判别存在性之外，一般均要求此条路径具备某种最优性，亦即在某种计算方法下最短或最长。对于路径的起点和终点也常常有着限制。

寻路问题一般都是最优化问题，这一点上与动态规划是一致的。所以说，很多寻路类型的题目都可用动态规划解答，图论中的Dijkstra算法（见3.1.1）、Floyd算法（见3.2）都是动态规划在图论中的经典应用。

另一方面，很多原本不存在图论模型的动态规划题目，也可转化成或建立起图论模型，并利用图论中的寻路算法进行解答。例如完全背包问题（TODO：引用《背包问题九讲》的相应章节），就可转化为图论模型：容量为 $V$ 的背包转化为编号从0到 $V$ 的点，每个费用为 $v_i$ 、价值为 $w_i$ 的物品转化为 $V - v_i + 1$ 条从 $k + v_i$ 到 $k$ 的长度为 $w_i$ 的边（ $0 \leq k \leq V - v_i$ ）。而找到一组最优解则对应为找到这个有向无环图中点 $V$ 到点0的一条最长路（具体算法见2.2）。

综上，寻路问题与动态规划有着千丝万缕的联系。本文后面的部分便着重探讨这种联系衍生出的千变万化的动态规划题目与解法。

## 2 特殊图中的寻路

### 2.1 矩阵中的寻路

#### 2.1.1 基础问题

问题是这样的：设有一 $M \times N$ 的矩阵 $P$ ，矩阵单元格中的元素皆是数字。从矩阵的左上角走到右下角，只能向右或向下走，怎样能使覆盖到的数字之和最大？（TODO：配图？）

$M \times N$ 的矩阵从左上走到右下，需要走 $M + N - 2$ 步，其中向下走 $M - 1$ 步，向右走 $N - 1$ 步，故可能的方案数共有 $\binom{M + N - 2}{M - 1}$ 种。用搜索穷举所有方案的时间复杂度太高，考虑使用动态规划算法。

动态规划算法的一个典型思考方式是“只想一步”。以这道题目为例来说明，尽管要求做出的是 $M + N - 2$ 个向右或者向下的指令，我们不妨想想：知道了什么信息，就可以得出第一步应该怎么走。

第一步有两种走法，向右走从 $(0, 0)$ 走到 $(0, 1)$ ，或者向下走从 $(0, 0)$ 走到 $(1, 0)$ ，二者的分野不仅存在于这一步覆盖到的数字不同，也体现在今后可选择的策略不同。例如若第一步向右走则左边第一列的数字再也无法拿到。

如何才能判别从 $(0, 0)$ 到 $(M - 1, N - 1)$ 的路途中第一步怎么走呢？怎样比较第一步的两种走法的优劣呢？很简单，这一方面取决于 $(0, 1)$ 和 $(1, 0)$ 上面的数字，另一方面也取决于，从这两点出发，走到终点的过程中还会覆盖哪些数字，可能覆盖的最大的数字之和是多少。

于是这样抽象出动态规划的子问题：

设 $F[i, j]$ 表示从 $(i, j)$ 出发，走到 $(M - 1, N - 1)$ ，可以覆盖到的数字之和的最大值，需求解的原问题既是 $F[0, 0]$ ，边界条件是 $F[M - 1, N - 1] = P_{M - 1, N - 1}$ ，状态转移方程是：

$$F[i, j] = P_{i, j} + \max \begin{cases} F[i, j + 1] & \text{if } j + 1 < N \\ F[i + 1, j] & \text{if } i + 1 < M \end{cases}$$

若将此方程以记忆化递归的方法实现，伪代码如下：

```

def MatrixWalkRec( $F, i, j$ )
    if  $i \geq M$  or  $j \geq N$ 
        return 0
    if  $F[i, j] = \text{undefined}$ 
         $F[i, j] \leftarrow$ 
 $P_{i,j} + \max(\text{MatrixWalkRec}(F, i, j + 1), \text{MatrixWalkRec}(F, i + 1, j))$ 
    return  $F[i, j]$ 

```

刚才应用了“只想一步”的方法，成功地解决了这道题目如何划分子问题、如何写出状态转移方程的问题。美中不足的是，上面给出的自然的实现是递归的实现。怎样将递归的状态转移方程实现成非递归的程序呢？

这就需要对我们的状态转移方程所决定的状态之间的相互关系进行分析。我们看到，每个状态 $(i, j)$ 依赖于它右方和下方的 $(i + 1, j)$ 和 $(i, j + 1)$ ，非递归的实现的实质是要确定一种合适的状态的求解顺序，保证任何状态的求解都在其依赖的状态之求解完成之后才开始。一种合适的顺序是：从下到上，从右至左。伪代码如下：

```

def MatrixWalk
     $F[0 \dots M, N] \leftarrow 0$ 
     $F[M, 0 \dots N] \leftarrow 0$ 
    for  $i \leftarrow M - 1$  to 0
        for  $j \leftarrow N - 1$  to 0
             $F[i, j] \leftarrow P_{i,j} + \max(F[i, j + 1], F[i + 1, j])$ 

```

至此，本问题的动态规划解法分析完毕。留一个问题让读者思考：前面利用“只想一步”的思考方式时，想的是“第一步应该怎么走”，如果考虑的角度变成“最后一步应该怎么走”，会导出怎样的子问题定义、状态转移方程以及伪代码实现呢？

## 2.1.2 变形问题

第一种变形问题是：改变计算权值的方法。

例如，将求最大值改成最小值，将求和的加法运算改成乘法，将矩阵单元格中的值由实数改成向量，等等……

一般而言，这种变形问题只需对状态转移方程进行一些微调就可以。

但要注意的是，子问题的局部最优性是否还代表着全局最优性。例如，将问题中的“数字之和最大”改成“数字之积最大”之后，还按照上文中“如何踏出第一步”的方法思考时，假如说 $(0, 0)$ 中的数字是个负数，那么接下来向右走及向下走之后要解决的子问题就不是求“数字之积最大”而是希望求得一个“积是负数且绝对值最大”的子路径。

也就是说，将“和”改成“积”后，对于每个子问题 $(i, j)$ ，需要运算及记录的是两个值：一个是路径上数字之积的可能的最大值 $F[i, j]$ ，一个是可能的最小值 $G[i, j]$ （或称绝对值最大的负值）。

这个变形问题的伪代码及其实现强烈建议读者尝试一下。

(TODO: 是否在这里给出伪代码呢?)

第二种变形问题是：改变路径的可行性。

在原问题中，所有 $\binom{M-1}{M+N-2}$ 种路径都是可行的，但我们可以人为设置一些限制条件。例如，设置一些单元格为禁止通行的单元格。或设置一些单元格之间的边界为禁止通过的边界。或设置通过单元格或单元格之间的边界时必须满足某种条件。这样的限制条件并不会更改状态转移方程的形式，只不过

可以转移到的状态要在程序中根据题义进行判断，这种判断可以设计得相当复杂。

给一个这种变形的例题：单元格之间有可能存在锁着的门，打开一扇门必须耗费一把钥匙，不同的门需要的钥匙的种类不同，单元格中可能会捡到钥匙；问题仍然是求解最优的路径。

这道例题的求解，需要在状态转移方程中增加一维来表示手中现有的钥匙种类和数量。然后在计算过程中根据捡到和消耗的钥匙进行状态之间的转移。

(TODO: 加个示意图?)

第三种变形问题是：增加路径的条数。

(TODO: 写具体内容。)

(TODO: 还有什么变形问题的种类吗?)

## 2.2 有向无环图中的寻路

# 3 一般图中的寻路

## 3.1 单起点寻路的三种算法

### 3.1.1 Dijkstra算法

### 3.1.2 Bellman-Ford算法

### 3.1.3 Yen算法

## 3.2 全图寻路的Floyd算法

## 3.3 Hamiltonian回路

### 3.3.1 简单的动态规划算法

### 3.3.2 优化的动态规划算法

# 4 扩展的寻路问题

## 4.1 扩展的Dijkstra算法