寻寻觅觅·道阻且长
动态规划与寻路问题
v0.2 preview

崔添翼 (Tianyi Cui)*

2011-09-17†

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。

Contents

*a.k.a. dd_engi

†Build 20110917093700

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（$0 \le k \le V-v_i$）。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$\begin{pmatrix} M-1 \\ M+N-2 \end{pmatrix}$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：

$$F[i,j]=P_{i,j}+\max\left\{\begin{array}{l} F[i,j+1] \text{ if } j+1<N \\ F[i+1,j] \text{ if } i+1<M \end{array}\right\}$$

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 ≥ M or j ≥ N
    return 0
  if F[i, j] = undefined
    F[i, j] ←
P_{i,j} + max(MatrixWalkRec(F,i,j + 1),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 . . . M, N] ← 0
  F[M, 0 . . . N] ← 0
  for i ← M − 1 to 0
    for j ← N − 1 to 0
      F[i, j] ← 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$\begin{pmatrix} M-1 \\ M+N-2 \end{pmatrix}$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。

典型问题如，要求两条从矩阵左上到右下的路径，这两条路径不可以相交（或相交后同一单元格的数值只算一次），最大化路径经过的权值和。

这种题目可以以两条路径所经过的单元格距左上单元格的距离划分阶段，以两条路径所经过的单元格坐标做为状态，列出状态转移方程。

（TODO：方程其实写出来很难看，不如写程序代码附录进去。）

以上说明了矩阵中寻路问题的三种变形问题的方向，在实际中，将这道基础的动态规划题目经由多个方向同时变形，就可以构造出看似很复杂的题目。

（TODO：还有什么变形问题的种类吗？）

2.2 有向无环图中的寻路

2.2.1 从矩阵寻路到有向无环图

在上一节2.1中，我们给出的基础问题及其所有变形问题都有一个共同点：要求找到从左上角到右下角的一条路径，且只能向相邻的右方、下方的单元格走。

这个限制条件给所有的单元格排了个顺序，亦即，单元格$(i+p, j+q)$和(i,j)若均在路径中出现，$(i+p, j+q)$一定出现在(i,j)之后（$p \geq 0, q \geq 0, p+q>0$）。这个顺序决定了我们把动态规划的状态的表示确定为单元格的坐标时，状态之间的依赖关系是很明晰的，状态的求解顺序也不是个难题。

我们来看一道没有这个只能向右、下走的限制条件的例题[1]：

仍然是$M \times N$的数字矩阵P，要求寻找的路径可以从任意一点出发，可以向上、下、左、右任意方向延伸，但路径经过的数字必须越来越小、单调递减。求矩阵中存在的最长的满足要求的路径。

去掉了仅能向右、下走及固定起点、终点的限制条件，加上了一个路径中数字单调递减的限制条件。这里还是利用“只想一步”的方法来思考。路径的起点在哪里？不知道，设为(i,j)。路径的第一步往哪儿走？往哪里走能走得更远就往哪里走。于是状态转移方程是：

$$F[i,j] = 1 + \max \left\{ \begin{array}{c} 0 \\ F[i,j-1] \text{ if } P_{i,j-1} < P_{i,j} \\ F[i,j+1] \text{ if } P_{i,j+1} < P_{i,j} \\ F[i-1,j] \text{ if } P_{i-1,j} < P_{i,j} \\ F[i+1,j] \text{ if } P_{i+1,j} < P_{i,j} \end{array} \right\}$$

考察这个状态转移方程所隐含的状态之间的依赖关系：一个状态的计算有可能依赖于它上下左右的任意一个状态，这看上去很可能存在状态间的循环依赖以至于状态转移方程无法被求解。但实际上，一个状态只依赖于其上下左右的单元格中元素数值比自身小的那些。换句话说，若将所有的单元格按照其中元素从小到大的顺序排个序，并按这个顺序求解状态的话，则只会有后面的状

[1] PKU 1088

态依赖于前面的状态，不会出现求解某一状态时它所依赖的状态还未解出的可能。下面给出伪代码：

def MatrixDecPath
  $F[0 \dots M+1, 0 \dots N+1] \leftarrow 0$
  $Q \leftarrow [(P_{i,j}, i, j) \mid \text{for } i, j \text{ in } (1 \dots M, 1 \dots N)]$
  sort Q
  for p, i, j in Q

$$F[i,j] \leftarrow 1 + \max \left\{ \begin{array}{c} 0 \\ F[i,j-1] \text{ if } P_{i,j-1} < P_{i,j} \\ F[i,j+1] \text{ if } P_{i,j+1} < P_{i,j} \\ F[i-1,j] \text{ if } P_{i-1,j} < P_{i,j} \\ F[i+1,j] \text{ if } P_{i+1,j} < P_{i,j} \end{array} \right\}$$

（TODO: 伪代码直接抄状态转移方程写成这样是不是真的有点丑？）

最后的答案就是$F[1 \dots M, 1 \dots N]$中的最大值。

值得指出的是，在看清了状态的求解不存在循环依赖的时候，就可以不用排序，而直接用记忆化递归的方法（TODO：引用MatrixWalkRec）实现。

2.2.2 有向无环图的拓扑排序

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