孤蓬万里征

自己的命运在自己手里

日志

动态规划

2009-10-15 11:36:29 | 分类： 算法

动态规划

在数学与计算机科学领域，**动态规划**用于解决那些可分解为重复子问题（overlapping subproblems，***想想递归求阶乘吧***）并具有最优子结构（optimal substructure，***想想最短路径算法***）（如下所述）的问题，动态规划比通常算法花费更少时间。

上世纪40年代，Richard Bellman最早使用动态规划这一概念表述通过遍历寻找最优决策解问题的求解过程。1953年，Richard Bellman将动态规划赋

关于我

english man2008

加博友 关注他

献，动态规划的核心方程被命名为贝尔曼方程，该方程以递归形式重申了一个优化问题。

在“动态规划”（dynamic programming）一词中，programming与“计算机编程”（computer programming）中的programming并无关联，而是来自“数学规划”（mathematical programming），也称优化。因此，规划是指对生成活动的优化策略。举个例子，编制一场展览的日程可称为规划。 在此意义上，规划意味着找到一个可行的活动计划。

- **概述**

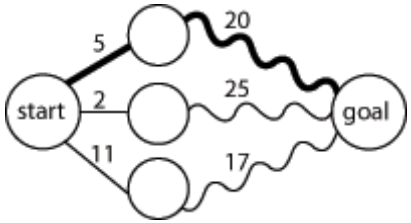

图1 使用最优子结构寻找最短路径：直线表示边，波状线表示两顶点间的最短路径（路径中其他节点未显示）；粗线表示从起点到终点的最短路径。

不难看出，start到goal的最短路径由start的相邻节点到goal的最短路径及start到其相邻节点的成本决定。

最优子结构即可用来寻找整个问题最优解的子问题的最优解。举例来说，寻找图上某顶点到终点的最短路径，可先计算该顶点所有相邻顶点至终点的最短路径，然后以此来选择最佳整体路径，如**图1**所示。

一般而言，最优子结构通过如下三个步骤解决问题：

文章分类

LOFTER精选

b) 通过递归使用这三个步骤求出子问题的最优解；

c) 使用这些最优解构造初始问题的最优解。

子问题的求解是通过不断划分为更小的子问题实现的，直至我们可以在常数时间内求解。

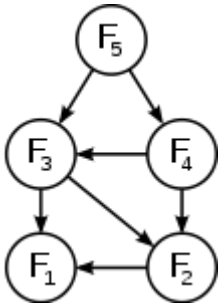

图2 Fibonacci序列的子问题示意图：使用有向无环图（DAG, directed acyclic graph）而非树表示重复子问题的分解。

为什么是DAG而不是树呢？答案就是，如果是树的话，会有很多重复计算，下面有相关的解释。

一个问题可划分为重复子问题是指通过相同的子问题可以解决不同的较大问题。例如，在Fibonacci序列中，F3 = F1 + F2和F4 = F2 + F3都包含计算F2。由于计算F5需要计算F3和F4，一个比较笨的计算F5的方法可能会重复计算F2两次甚至两次以上。这一点对所有重复子问题都适用：愚蠢的做法可能会为重复计算已经解决的最优子问题的解而浪费时间。

为避免重复计算，可将已经得到的子问题的解保存起来，当我们要解决相同的子问题时，重用即可。该方法即所谓的缓存（memoization，而不是存储

简直就是可意会不可言传，其意义是没计算过则计算，计算过则保存）。当我们确信将不会再需要某一解时，可以将其抛弃，以节省空间。在某些情况下，我们甚至可以提前计算出那些将来会用到的子问题的解。

总括而言，动态规划利用：

1) 重复子问题

2) 最优子结构

3) 缓存

动态规划通常采用以下两种方式中的一种两个办法：

自顶向下：将问题划分为若干子问题，求解这些子问题并保存结果以免重复计算。该方法将递归和缓存结合在一起。

自下而上：先行求解所有可能用到的子问题，然后用其构造更大问题的解。该方法在节省堆栈空间和减少函数调用数量上略有优势，但有时想找出给定问题的所有子问题并不那么直观。

为了提高按名传递（call-by-name，这一机制与按需传递call-by-need相关，***复习一下参数传递的各种规则吧，简单说一下，按名传递允许改变实参值***）的效率，一些编程语言将函数的返回值“自动”缓存在函数的特定参数集合中。一些语言将这一特性尽可能简化（如Scheme、Common Lisp和Perl），也有一些语言需要进行特殊扩展（如C++，***C++中使用的是按值传递和按引用传递，因此C++中本无自动缓存机制，需自行实现，具体实现的一个例子是Automated Memoization in C++***）。无论如何，只有指称透明

或以其值替换对程序结果没有任何影响）函数才具有这一特性。

- **例子**

1. Fibonacci序列

寻找Fibonacci序列中第n个数，基于其数学定义的直接实现：

```
function fib(n)
    if n = 0
        return 0
    else if n = 1
        return 1
    return fib(n-1) + fib(n-2)
```

如果我们调用fib(5)，将产生一棵对于同一值重复计算多次的调用树：

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

特别是，fib(2)计算了3次。在更大规模的例子中，还有更多fib的值被重复计算，将消耗指数级时间。

现在，假设我们有一个简单的映射（map）对象*m*，为每一个计算过的fib及其返回值建立映射，修改上面的函数fib，使用并不断更新*m*。新的函数将只需$O(n)$的时间，而非指数时间：

```
function fib(n)
    if map m does not contain key n
        m[n] := fib(n-1) + fib(n-2)
    return m[n]
```

这一保存已计算出的数值的技术即被称为缓存，这儿使用的是**自顶向下**的方法：先将问题划分为若干子问题，然后计算和存储值。

在**自下而上**的方法中，我们先计算较小的fib，然后基于其计算更大的fib。这种方法也只花费线性（*O*(*n*)）时间，因为它包含一个*n*-1次的循环。然而，这一方法只需要常数（*O*(1)）的空间，相反，**自顶向下**的方法则需要O(*n*)的空间来储存映射关系。

```
function fib(n)
    var previousFib := 0, currentFib := 1
    if n = 0
        return 0
    else if n = 1
        return 1
    repeat n-1 times
        var newFib := previousFib + currentFib
        previousFib := currentFib
        currentFib  := newFib
    return currentFib
```

在这两个例子，我们都只计算fib(2)一次，然后用它来计算fib(3)和fib(4)，而不是每次都重新计算。

考虑*n***n*矩阵的赋值问题：只能赋0和1，*n*为偶数，使每一行和列均含*n*/2个0及*n*/2个1。例如，当*n*=4时，两种可能的方案是：

```
+ - - - - +       + - - - - +
| 0 1 0 1 |       | 0 0 1 1 |
| 1 0 1 0 |       | 0 0 1 1 |
| 0 1 0 1 |       | 1 1 0 0 |
| 1 0 1 0 |       | 1 1 0 0 |
+ - - - - +       + - - - - +
```

问：对于给定*n*，共有多少种不同的赋值方案。

至少有三种可能的算法来解决这一问题：穷举法（brute force）、回溯法（backtracking）及动态规划（dynamic programming）。穷举法列举所有赋值方案，并逐一找出满足平衡条件的方案。由于共有C(*n*, *n*/2)^*n*种方案（***在一行中，含n/2个0及n/2个1的组合数为C(n,n/2)，相当于从n个位置中选取n/2个位置置0，剩下的自然是1***），当*n*=6时，穷举法就已经几乎不可行了。回溯法先将矩阵中部分元素置为0或1，然后检查每一行和列中未被赋值的元素并赋值，使其满足每一行和列中0和1的数量均为*n*/2。回溯法比穷举法更加巧妙一些，但仍需遍历所有解才能确定解的数目，可以看到，当*n*=8时，该题解的数目已经高达116963796250。动态规划则无需遍历所有解便可确定解的数目（***意思是划分子问题后，可有效避免若干子问题的重复计算***）。

通过动态规划求解该问题出乎意料的简单。考虑每一行恰含*n*/2个0和*n*/2个1的*k***n*(1<=*k*<=*n*)的子矩阵，函数f根据每一行的可能的赋值映射为一个向

数分别表示该列上该行以下已经放置的0和1的数量。该问题即转化为寻找f((*n*/2,*n*/2),(*n*/2,*n*/2),...,(*n*/2,*n*/2))（具有*n*个参数或者说是一个含*n*个元素的向量）的值。其子问题的构造过程如下：

1) 最上面一行（***第k行***）具有C(*n*, *n*/2)种赋值；

2) 根据最上面一行中每一列的赋值情况（为0或1），将其对应整数对中相应的元素值减1；

3) 如果任一整数对中的任一元素为负，则该赋值非法，不能成为正确解；

4) 否则，完成对*k***n*的子矩阵中最上面一行的赋值，取*k*=*k*-1，计算剩余的(*k*-1)**n*的子矩阵的赋值；

5) 基本情况是一个1**n*的细小的子问题，此时，该子问题的解的数量为0或1，取决于其向量是否是*n*/2个(0, 1)和*n*/2个(1, 0)的排列。

例如，在上面给出的两种方案中，向量序列为：

((2, 2) (2, 2) (2, 2) (2, 2))　　((2, 2) (2, 2) (2, 2) (2, 2))　　k = 4
 0　　1　　0　　1　　　　0　　0　　1　　1

((1, 2) (2, 1) (1, 2) (2, 1))　　((1, 2) (1, 2) (2, 1) (2, 1))　　k = 3
 1　　0　　1　　0　　　　0　　0　　1　　1

((1, 1) (1, 1) (1, 1) (1, 1))　　((0, 2) (0, 2) (2, 0) (2, 0))　　k = 2
 0　　1　　0　　1　　　　1　　1　　0　　0

((0, 1) (1, 0) (0, 1) (1, 0))　　((0, 1) (0, 1) (1, 0) (1, 0))　　k = 1
 1　　0　　1　　0　　　　1　　1　　0　　0

动态规划在此的意义在于避免了相同f的重复计算，更进一步的，上面着色的两个f，虽然对应向量不同，但f的值是相同的，想想为什么吧:D。

该问题解的数量（序列a058527在OEIS）是1, 2, 90, 297200, 116963796250, 67362182874304607 52, ...

下面的外部链接中包含回溯法的Perl源代码实现，以及动态规划法的MAPLE和C语言的实现。

3. 棋盘

考虑*n***n*的棋盘及成本函数C(*i*,*j*)，该函数返回方格(*i*,*j*)相关的成本。以5*5的棋盘为例：BGX这是一个反向逆推的关系。

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

可以看到：C(1,3)=5

从棋盘的任一方格的第一阶（即行）开始，寻找到达最后一阶的最短路径（使所有经过的方格的成本之和最小），假定只允许向左对角、右对角或垂直移动一格。

4 |
3 |
2 | x x x
1 | o
- + - - - - -
| 1 2 3 4 5

该问题展示了最优子结构。即整个问题的全局解依赖于子问题的解。定义函数q(*i*,*j*)，令：q(*i*,*j*)表示到达方格(*i*,*j*)的最低成本。

如果我们可以求出第*n*阶所有方格的q(*i*,*j*)值，取其最小值并逆向该路径即可得到最短路径。

记q(*i*,*j*)为方格(*i*,*j*)至其下三个方格（***(i-1,j-1)***、***(i-1,j)***、***(i-1,j+1)***）最低成本与c(*i*,*j*)之和，例如：

5 |
4 | *A*
3 | *B C D*
2 |
1 |
- + - - - - -
| 1 2 3 4 5

q(*A*) = min(q(*B*),q(*C*),q(*D*)) + c(*A*)

定义q(*i*,*j*)的一般形式：

$$q(i,j) = \begin{cases} \min_{i=1} c(i,j) & \\ \min(q(i-1,j-1),q(i-1,j),q(i-1,j+1))+c(i,j) & \text{otherwise.} \end{cases}$$

方程的第一行是为了保证递归可以退出（处理边界时只需调用一次递归函数）。第二行是第一阶的取值，作为计算的起点。第三行的递归是算法的重要组成部分，与例子*A*、*B*、*C*、*D*类似。从该定义我们可以直接给出计算q(*i*,*j*)的简单的递归代码。在下面的伪代码中，*n*表示棋盘的维数，C(*i*,*j*)是成本函数，min()返回一组数的最小值：

```
function minCost(i, j)
    if j < 1 or j > n
        return infinity
    else if i = 1
        return c(i,j)
    else
        return min(minCost(i-1,j-1),minCost(i-1,j),minCost(i-1,j+1))+c(i,j)
```

需要指出的是，minCost只计算路径成本，并不是最终的实际路径，二者相去不远。与Fibonacci数相似，由于花费大量时间重复计算相同的最短路径，这一方式慢的恐怖。不过，如果采用自下而上法，使用二维数组q[i,j]代替函数minCost，将使计算过程快得多。我们为什么要这样做呢？选择保存值显然比使用函数重复计算相同路径要简单的多。

我们还需要知道实际路径。路径问题，我们可以通过另一个前任数组p[i,j]解决。这个数组用于描述路径，代码如下：

```
for x from 1 to n
    q[1, x] := c(1, x)
for y from 1 to n
    q[y, 0]     := infinity
    q[y, n + 1] := infinity
for y from 2 to n
    for x from 1 to n
        m := min(q[y-1, x-1], q[y-1, x], q[y-1, x+1])
        q[y, x] := m + c(y, x)
        if m = q[y-1, x-1]
            p[y, x] := -1
        else if m = q[y-1, x]
            p[y, x] :=  0
        else
            p[y, x] :=  1
```

剩下的求最小值和输出就比较简单了：

```
function computeShortestPath()
    computeShortestPathArrays()
    minIndex := 1
    min := q[n, 1]
    for i from 2 to n
        if q[n, i] < min
            minIndex := i
```

```
    printPath(n, minIndex)

function printPath(y, x)
    print(x)
    print("<-")
    if y = 2
        print(x + p[y, x])
    else
        printPath(y-1, x + p[y, x])
```

4. 序列比对

序列比对是动态规划的一个重要应用。序列比对问题通常是使用编辑操作（替换、插入、删除一个要素等）进行序列转换。每次操作对应不同成本，目标是找到编辑序列的最低成本。

可以很自然地想到使用递归解决这个问题，序列*A*到*B*的最优编辑通过以下措施之一实现：

插入*B*的第一个字符，对*A*和*B*的剩余序列进行最优比对；

删去*A*的第一个字符，对*A*和*B*进行最优比对；

用*B*的第一个字符替换*A*的第一个字符，对*A*的剩余序列和*B*进行最优比对。

局部比对可在矩阵中列表表示，单元(*i*,*j*)表示A[1..*i*]到b[1..*j*]最优比对的成本。单元(*i*,*j*)的成本计算可通过累加相邻单元的操作成本并选择最优解实现。至于序列比对的不同实现算法，参见Smith-Waterman和Needleman-Wunsch。

介绍一下。

- **应用动态规划的算法**

1) 许多字符串操作算法如最长公共子列、最长递增子列、最长公共字串；

2) 将动态规划用于图的树分解，可以有效解决有界树宽图的生成树等许多与图相关的算法问题；

3) 决定是否及如何可以通过某一特定上下文无关文法产生给定字符串的Cocke-Younger-Kasami (CYK)算法；

4) 计算机国际象棋中转换表和驳斥表的使用；

5) Viterbi算法（用于隐式马尔可夫模型）；

6) Earley算法（一类图表分析器）；

7) Needleman-Wunsch及其他生物信息学中使用的算法，包括序列比对、结构比对、RNA结构预测；

8) Levenshtein距离（编辑距离）；

9) 弗洛伊德最短路径算法；

10) 连锁矩阵乘法次序优化；

11) 子集求和、背包问题和分治问题的伪多项式时间算法；

12) 计算两个时间序列全局距离的动态时间规整算法；

13) 关系型数据库的查询优化的Selinger（又名System R）算法；

14) 评价B样条曲线的De Boor算法；

16) 价值迭代法求解马尔可夫决策过程；

17) 一些图形图像边缘以下的选择方法，如“磁铁”选择工具在Photoshop；

18) 间隔调度；

19) 自动换行；

20) 巡回旅行商问题（***又称邮差问题或货担郎问题***）；

21) 分段最小二乘法；

22) 音乐信息检索跟踪。

对于这些算法应用，大多未曾接触，甚至术语翻译的都有问题，鉴于本文主要在于介绍动态规划，所以仓促之中，未及查证。

- **相关**

1) 贝尔曼方程

2) 马尔可夫决策过程

3) 贪心算法

- **参考**

- Adda, Jerome, and Cooper, Russell, 2003. *Dynamic Economics.* MIT Press. An accessible introduction to dynamic programming in economics. The link contains sample programs.
- Richard Bellman, 1957, *Dynamic Programming*, Princeton University Press. Dover paperback edition (2003), ISBN 0486428095.

Vols. 1 & 2, 2nd ed. Athena Scientific. ISBN 1-886529-09-4.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, 2001. *Introduction to Algorithms*, 2nd ed. MIT Press & McGraw-Hill. ISBN 0-262-03293-7. Especially pp. 323–69.
- Giegerich, R., Meyer, C., and Steffen, P., 2004, "A Discipline of Dynamic Programming over Sequence Data," *Science of Computer Programming 51*: 215-263.
- Nancy Stokey, and Robert E. Lucas, with Edward Prescott, 1989. *Recursive Methods in Economic Dynamics*. Harvard Univ. Press.
- S. P. Meyn, 2007. Control Techniques for Complex Networks, Cambridge University Press, 2007.

- **外部链接**

- Dyna, a declarative programming language for dynamic programming algorithms
- Wagner, David B., 1995, "Dynamic Programming." An introductory article on dynamic programming in Mathematica.
- Ohio State University: CIS 680: class notes on dynamic programming, by Eitan M. Gurari
- A Tutorial on Dynamic programming
- MIT course on algorithms - Includes a video lecture on DP along with lecture notes -- See lecture 15.
- More DP Notes
- King, Ian, 2002 (1987), "A Simple Introduction to Dynamic Programming in Macroeconomic Models." An introduction to dynamic

- Dynamic Programming: from novice to advanced A TopCoder.com article by Dumitru on Dynamic Programming
- Algebraic Dynamic Programming - a formalized framework for dynamic programming, including an entry-level course to DP, University of Bielefeld
- Dreyfus, Stuart, "Richard Bellman on the birth of Dynamic Programming."
- Dynamic programming tutorial
- An Introduction to Dynamic Programming

—

关于动态规划，这只是一篇译文，后面将根据实际问题具体写点动态规划的应用。